DESIGN AND ANALYSIS OF ALGORITHM

NOTE:

MAKAUT course structure and syllabus of 4th semester has been changed from 2020. Previously **DESIGN AND ANALYSIS OF ALGORITHM** was in 5th [CS & IT] semester. This subject has been completely redesigned and shifted in 4th semester in present curriculum. Few new topics have been introduced also. Taking special care of this matter we are providing the relevant MAKAUT university solutions and some model questions & answers for newly introduced topics, so that students can get an idea about university questions patterns.

INTRODUCTION

☞ Chapter at a Glance

- **Algorithms** are classified into two major groups. One is an "analytic" approach of recognizing individual characters, while the other is a "holistic" approach. In some non-heuristic studies, analysis consists of two phases:
  1. A priori Analysis: By theoretical method we may calculates the computing time and space required by an algorithm by formulating a function and it is independent of programming language and the machine.
  2. A posteriori Analysis: In some cases we cannot calculate the time and space requirement of an algorithm using theoretical method. Then we observe its execution and measure the time and space actually used by the algorithm. This is called a posteriori analysis.
- **Space complexity**: It is defined as how much space is required to execute an algorithm. The (space) complexity of an algorithm (for a given input) is the number of elementary objects that the algorithm needs to store during its implementation. This number is computed with respect to the size n of the input data. This space is generally computer memory space. Naturally, an algorithm to achieve a specific growth with less space requirement is considered a better algorithm in terms of space complexity.
  The space complexity of an algorithm X can be defined as, **S(X) = C (X) + I(X)** Where C(X) is the required constant space whereas, I(X) is the instantaneous space requirement for algorithm X.
- **Time complexity** of an algorithm is the measure of total time required to execute that algorithm. Time complexity is independent of computer, programming language, programmer, and other implementation details. Usually, it is depending on the size of the input. The (time) complexity of a algorithm (for a given input) is the number of elementary instructions that this algorithm implements.
- **Asymptotic Notations:** The algorithm that takes minimum time to execute given an input of specific size is asymptotically more efficient algorithm.
  - **Worst-case complexity:** The worst-case complexity is the complexity of an algorithm when the input is the worst possible with respect to complexity.
  - **Average case complexity:** The average complexity is the complexity of an algorithm that is averaged over all the possible inputs.
- **Master's Theorem:**

  The Master Theorem applies to recurrences of the following forms: $T(n) = aT(n/b) + f(n)$

  where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.
  There are 3 cases:

  1. If $f(n) = O(n^{\log_b a})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
  2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
  3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ with $\varepsilon \geq 0$ and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$.

  Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

- **Recursion tree Method:**
  In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically an arithmetic or geometric series.
  So, the main problem of substitution method is to find out the good guess otherwise, it will not find out the correct solution. But in recurrence tree method, we start from the initial source to expand the tree.
- **Substitution Method:** We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect. So, this method consists of two steps:
  - Guess the form of the solution.
  - Use mathematical induction to find constants in the form and show that the solution works.

Multiple Choice Type Questions

1. Lower bound for any comparison sort is **[WBUT 2006, 2007, 2008, 2009, 2017]**
a) O (log n) **b) O (n^2)** **c) O (n log n)** **d) O (n^2 log n)**
Answer: (c)

2. O(g(n)) is [Read as small oh of g(n) is] **[WBUT 2006, 2008, 2009, 2012]**
a) Asymptotically loose
b) Asymptotically tight
c) Same as big oh
d) none of these
Answer: (a)

3. Time complexity for recursive relation $T(n)=2T\left(\sqrt{n}\right)+1$ is
[WBUT 2006, 2008, 2013]
a) $\theta(n^2)$ **b)** $\theta(\log n)$ **c)** $\theta(n\log n)$ **d)** $\theta(n)$
Answer: (b)

4. Tight bound for building a max heap algorithm will be **[WBUT 2006, 2015]**
a) O (log n) **b) O (n^2)** **c) O (n log n)** **d) O (n)**
Answer: (c)

5. Ω-Notation provides an asymptotic **[WBUT 2006, 2015, 2017]**
a) Upper bound
b) Lower bound
c) One that is sandwiched between the two bounds
d) None of these
Answer: (b)

6. Time complexity for recurrent relation $T(n)=2T(n/2)+n$ is
[WBUT 2007, 2018, 2019]
a) O (log n) **b) O (n log n)** **c) O (n)** **d) O (n^2)**
Answer: (b)

7. The Big O notation of the expression $f(n) = n\log n + n^2 + e^{\log n}$ is [WBUT 2008, 2012]

a) O (n) b) O (n^2) c) O (n log n) d) O ($e^{\log n}$)

Answer: (b)

8. 'Small o' of g (n) is [WBUT 2010, 2012]

a) asymptotically loose
b) asymptotically tight
c) same as 'Big O'
d) none of these

Answer: (d)

9. The time factor when determining the efficiency of an algorithm is measured by [WBUT 2010, 2016]

a) counting microseconds
b) counting the number of key operations
c) counting the number of statements
d) counting the kilobyte algorithm

Answer: (b)

10. Which of the following is used to depict the working of algorithm? [WBUT 2010]

a) Flowchart b) Pseudo code c) Source code d) All of these

Answer: (d)

11. The running time of an algorithm $T(n)$, where 'n' is the input size is given by $T(n)=8T(n/2)+\text{qn}$, if $n>1$ and $T(n)=p$, if $n=1$, where p and q are constants. The order of this algorithms is [WBUT 2011]

a) $\Theta(n^2)$ b) $\Theta(n^n)$ c) $\Theta(n^3)$ d) $\Theta(n^{\log n})$

Answer: (c)

12. Complexity the recurrence relation $T(n)=8T\left(\frac{n}{2}\right)+n^2$ is [WBUT 2012, 2016]

a) $O(n)$ b) $O(n^2)$ c) $O(\log_2 n)$ d) $O(n^3)$

Answer: (b)

13. Complexity of Tower of Hanoi problem is [WBUT 2012]

a) $O(n)$ b) $O(n^2)$ c) $O(2^n)$ d) none of these

Answer: (c)

14. Consider the following three claims: [WBUT 2013]

I) $(n+k)^m = O(n^m)$ where k and m are constants.

II) $2^{n+1} = O(2^n)$

III) $2^{2n+1} = O(2^n)$

Which of the following claims are correct?

a) I and II b) I and III c) II and III d) I, II and III

Answer: (d)

15. Which of the following functions is asymptotically smallest? [WBUT 2013]

a) 2^n **b)** $n^{\log n}$ **c)** $n^{\sqrt{n}}$ **d)** $\sqrt[3]{\log n}$

Answer: (d)

16. An algorithm is made up of two independent time complexities $f(n)$ and $g(n)$. Then the complexities of the algorithm is in the order of [WBUT 2014]

a) $f(n)\times g(n)$ **b)** $\max\left(f(n),g(n)\right)$

c) $\min\left(f(n),g(n)\right)$ **d)** $f(n)+g(n)$

Answer: (b)

17. Which of the following is used to depict the working of algorithm? [WBUT 2014]

a) flowchart **b) pseudo code** **c) source code** **d) all of these**

Answer: (d)

18. The space requirement for the quick sort method depends on the [WBUT 2015]

a) number of nested recursive calls **b) size of the stack**

c) both (a) and (b) **d) none of these**

Answer: (c)

19. Which of the following property/properties is/are necessary for an algorithm?

a) Definiteness **b) Effectiveness [WBUT 2017]**

c) Both (a) and (b) **d) None of these**

Answer: (c)

20. Time complexity for recurrence relation $T(n)=2T(n-1)+c$ is [WBUT 2017]

a) $O(n^2)$ **b)** $O(\log n)$ **c)** $O(n\log n)$ **d)** $O(2^n)$

Answer: (b)

21. The Big O Notation of the expression $f(n)=n\log_2 n+n^2+e^{\log_2 n}$ is [WBUT 2018, 2019]

a) $O(n^2)$ **b)** $O(\log_2 n)$ **c)** $O(n\log_2 n)$ **d)** $O\left(e^{\log_2 n}\right)$

Answer: (a)

22. Time complexity of quick sort in worst case is [WBUT 2019]

a) O (n) **b) O (log n)** **c) O (n2)** **d) O (n log n)**

Answer: (d)

Short Answer Type Questions

1. Prove that $n! = O(n^n)$. **[WBUT 2008, 2017]**

Answer:

The $n!$ is defined for integers $n \geq 0$ as

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n.(n-1)! & \text{if } n > 0 \end{cases}$$

Thus, n! = 1.2.3… n.

A weak upper bound on the factorial function is $n! \leq n^n$ …(1)

since each of the n terms in the factorial product is at most n. ***Stirling's approximation***,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$$

Where e is the base of the natural logarithm, gives us a tighter upper bound, and a lower bound as well.

So, from the above equation (1), we can say that f(n) = n! and g(n) = n^n and according to the definition of O-notation, f(n) =n! ≤n^n = O (n^n) where n≥ 0

2. Why Recursion Tree method is better than the Substitution method for solving a recurrence relation? Find asymptotic upper bound of the following recurrence relation with help of recursion tree method. **[WBUT 2014]**

$$T(n) = T(n/4) + T(n/2) + \Theta(n^2)$$

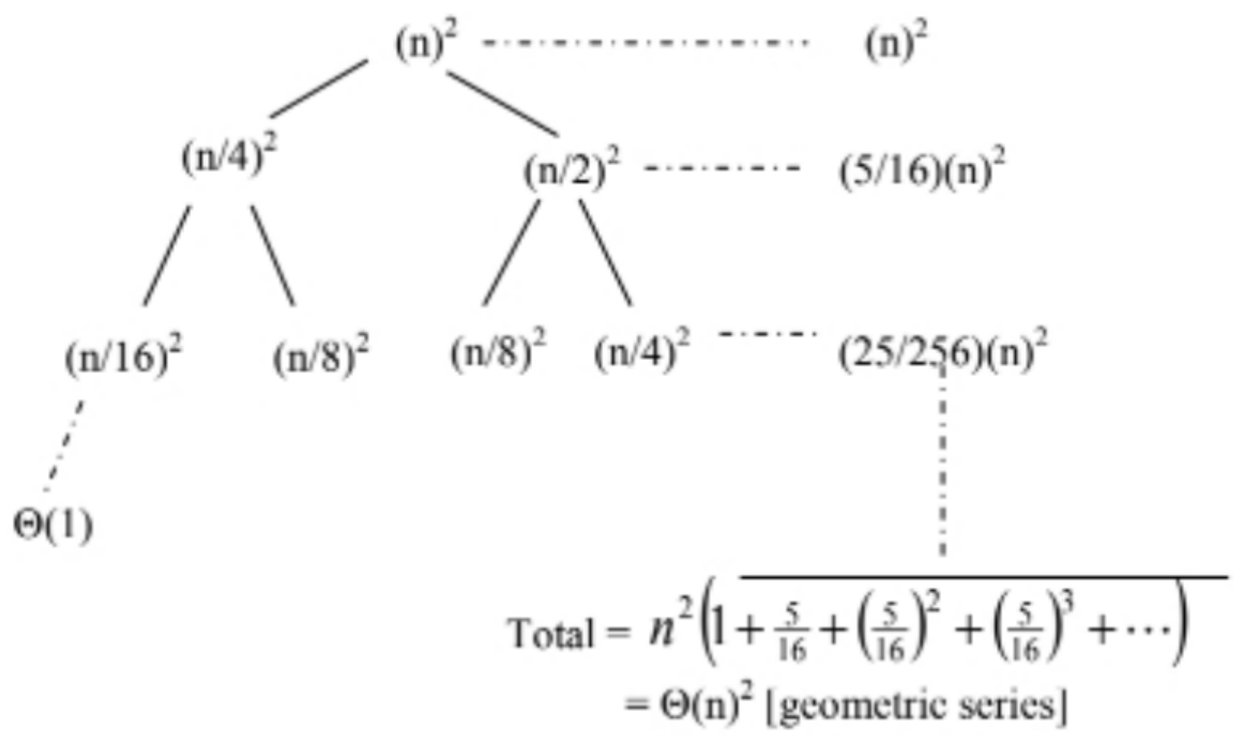
Answer:

Substitution Method: We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect. So, this method consists of two steps:

- Guess the form of the solution.
- Use mathematical induction to find constants in the form and show that the solution works

Recurrence Tree Method: In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically an arithmetic or geometric series.

So, the main problem of substitution method is to find out the good guess otherwise, it will not find out the correct solution. But in recurrence tree method, we start from the initial source to expand the tree. So, recurrence tree method is better than the substitution method,



3. State Master theorem and find the time complexity for the recurrence relation $T(n)=2T(n/4)+\sqrt{n}$. **[WBUT 2015]**

Answer:

1st Part:

The master theorem often yields asymptotically tight bounds to some recurrences from divide and conquer algorithms that partition an input into smaller sub-problems of equal sizes, solve the sub-problems recursively, and then combine the sub-problem solutions to give a solution to the original problem. The time for such an algorithm can be expressed by adding the work that they perform at the top level of their recursion (to divide the problems into sub-problems and then combine the sub-problem solutions) together with the time made in the recursive calls of the algorithm.

2nd Part:

$$T(n)=\sqrt{n}+2T\left(\frac{n}{4}\right)=\sqrt{n}+2\left(\sqrt{\frac{n}{4}}+2T\left(\frac{n}{4^2}\right)\right)=\sqrt{n}+2\sqrt{\frac{n}{4}}+2^2T\left(\frac{n}{4^2}\right)$$

$$=\sqrt{n}+2\sqrt{\frac{n}{4}}+2^2\left(\sqrt{\frac{n}{4^2}}+2T\left(\frac{n}{4^3}\right)\right)=\sqrt{n}+2\sqrt{\frac{n}{4}}+2^2\sqrt{\frac{n}{4^2}}+2^3T\left(\frac{n}{4^3}\right)$$

$$\cdots$$

$$=\sqrt{n}+2\sqrt{\frac{n}{4}}+2^2\sqrt{\frac{n}{4^2}}+2^3\sqrt{\frac{n}{4^3}}+2^4\sqrt{\frac{n}{4^4}}+\ldots+2^{\log_4 n}\sqrt{\frac{n}{4^{\log_4 n}}}$$

$$=\underbrace{\sqrt{n}+\sqrt{n}+\sqrt{n}+\sqrt{n}+\ldots+\sqrt{n}}_{\log_4 n+1}=\sqrt{n(\log_4 n+1)}=\Theta\left(\sqrt{n}\lg n\right)$$

4. Solve the following recurrence relation using generating function $a_n=6a_{n-1}-11a_{n-2}+6a_{n-3}$ **for** $n\geq 3$ **with initial condition** $a_0=1, a_1=-1$ **and** $a_2=1.$ **[WBUT 2016]**

Answer:

We have

$$a_n - 6a_{n-1} + 11a_{n-2} - 6a_{n-3} = 0 \qquad \ldots. (1)$$

Let, $g(x) = \sum_{n=0}^{\infty} a_n x^n$

Multiplying (1) by x^n and summing from $n = 3$ to ∞,

We get, $\sum_{n=3}^{\infty} a_n x^n - 6\sum_{n=3}^{\infty} a_{n-1} x^n + 11\sum_{n=3}^{\infty} a_{n-2} x^n - 6\sum_{n=3}^{\infty} a_{n-3} x^n = 0$

or, $(\sum_{n=0}^{\infty} a_n x^n - a_0 - a_1 x - a_2 x^2) - 6x\sum_{n=3}^{\infty} a_{n-1} x^{n-1} + 11x^2 \sum_{n=3}^{\infty} a_{n-2} x^{n-2} - 6x^3 \sum_{n=3}^{\infty} a_{n-3} x^{n-3} = 0$

or, $\{g(x) - a_0 - a_1 x - a_2 x^2) - 6x(\sum_{n=1}^{\infty} a_{n-1} x^{n-1} - a_0 - a_1 x)$

$$+ 11x^2 (\sum_{n=2}^{\infty} a_{n-2} x^{n-2} - a_0) - 6x^3 \sum_{n=0}^{\infty} a_{n-3} x^{n-3} = 0$$

or, $(g(x) - 1 + x - x^2) - 6x(\sum_{n-1}^{\infty} a_{n-1} x^{n-1} - 1 + x) + 11x^2 (\sum_{n-2}^{\infty} a_{n-2} x^{n-2} - 1) - 6x^3 g(x) = 0$

or, $\{g(x) - 1 + x - x^2\} - 6x\{g(x) - 1 + x\} + 11x^2 \{g(x) - 1\} - 6x^3 g(x) = 0$

or, $-g(x)\{6x^3 - 11x^2 + 6x - 1\} = 18x^2 - 7x + 1$

or, $g(x) = \dfrac{-(18x^2 - 7x + 1)}{6x^3 - 11x^2 + 6x - 1} \qquad \ldots. (2)$

Now, we can simplify,

$$6x^3 - 11x^2 + 6x - 1 = 6x^2(x-1) - 5x(x-1) + 1(x-1)$$

$$= (x-1)\{6x^2 - 5x + 1\} = (x-1)(2x-1)(3x-1)$$

So, from eq^n. (2), we get,

$$g(x) = \frac{-(18x^2 - 7x + 1)}{(x-1)(2x-1)(3x-1)} = \frac{(18x^2 - 7x + 1)}{(1-x)(1-2x)(1-3x)} \qquad \ldots. (3)$$

Now from eq^n. (3), we can write,

$$\frac{A}{1-x} + \frac{B}{1-2x} + \frac{C}{1-3x}$$

$$\Rightarrow A(1-2x)(1-3x) + B(1-x)(1-3x) + C(1-x)(1-2x) = 18x^2 - 7x + 1$$

Putting, $x = 1$, we get $A = 6$

Putting, $x = 1/2$, we get $B = -8$

Putting, $x = 1/3$, we get $C = 3$

Apply these values in eq^n. (3)

$$g(x)=6.\frac{1}{1-x}-8.\frac{1}{1-2x}+3.\frac{1}{1-3x}=6.(1-x)^{-1}-8(1-2x)^{-1}+3(1-3x)^{-1}$$

$$=6\sum_{n=0}^{\infty}x^n-8\sum_{n=0}^{\infty}2^n x^n+3\sum_{n=0}^{\infty}3^n x^n=\sum_{n=0}^{\infty}x^n(6-8.2^n+3.3^n)x^n$$

Hence, $a_n=6-8.2^n+3.3^n$ for $\forall n\geq 0$

5. Prove that if $f(n)=a_m n^m+a_{m-1}n^{m-1}+.....+a_1 n+a_0$ then $f(n)=O\left(n^m\right)$.

[WBUT 2019]

Answer:

$$f(n)\leq\sum_{i=0}^{m}|a_i|n^i$$

$$\leq n^m\sum_{i=0}^{m}|a_i|n^{i-m}$$

$$\leq n^m\sum_{i=0}^{m}|a_i| \qquad \text{for } n\geq 1$$

So, $f(n)=O\left(n^m\right)$ (assuming that m is fixed).

6. What is algorithm? Define the characteristics and categories of Algorithm.

[MODEL QUESTION]

Answer:

1st Part:

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

2nd Part:

The important categories of algorithms are :

(i) Search − Algorithm to search an item in a data structure.
(ii) Sort − Algorithm to sort items in a certain order.
(iii) Insert − Algorithm to insert item in a data structure.
(iv) Update − Algorithm to update an existing item in a data structure.
(v) Delete − Algorithm to delete an existing item from a data structure.

3rd part:

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

1. Unambiguous − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
2. Input − An algorithm should have 0 or more well-defined inputs.

3. Output − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
4. Finiteness − Algorithms must terminate after a finite number of steps.
5. Feasibility − Should be feasible with the available resources.
6. Independent − An algorithm should have step-by-step directions, which should be independent of any programming code.

7. Write an algorithm to add two numbers and discuss the time and space complexity. **[MODEL QUESTION]**

Answer:

Calculate the Space complexity:

```
                    Add (a,b)
                    {
Step 1                      a= 10;
Step 2              b= 20;
Step 3                      x= a + b;
Step 4              return x;
                    }
```

Here we consider the first algorithm **Add ()**.The size of the each variable (a, b and x) is fixed. If the space required for a single constant variable is n bytes then the total space required for all the constant variables used above is

C(**Add()**)= space requirement of (a, b, x)

= (n+ n+ n) bytes = 3n bytes

As, there is no instantaneous space requirement, so, the space complexity of the algorithm is S (**Add()**) = (3n+0) bytes = 3n bytes.

Calculate the Time complexity:

```
[Initialize c = 0]
                    Add (a,b)
            {
Step 1              a= 10;
                     [ c = c+1]      //   c is now 1.
Step 2                    b= 20;
                     [ c = c+1 ]     //   c is now 2.
Step 3              x= a + b;
                       [c = c+1]     //   c is now 3.
Step 4                    return x;
                       [c = c+1]     //   c is now 4.
            }
```

In the above example, an auxiliary variable c is inserted into the algorithm, which is initialized to 0 and incremented after each instruction. This variable c is not a part of the algorithm Add ().

After executing of **Add ()** , the value of c will be 4 as shown in the commented lines in the algorithm. This time is constant time and there is no instantaneous time in this algorithm.
So, the time complexity of **Add ()** is T(**Add()**)= 4.

8. Calculate the space for recursive factorial function of an element.
[MODEL QUESTION]

Answer:

```
Fact (n)
                {
Step 1          if n ≤ 0 then
Step 1.1                return 1;
Step 2          else
Step 2.1                return ( n* Fact(n-1));
                }
```

The return address of a recursive algorithm requires only one word of memory. Each call of **Fact ()** function requires two words, one for the value of n and another for return address. Now, this recursive function is executed (n+1) times, n times for variable n and last one for checking the value if n≤ 0. Now if the size of each word is k bytes then the space complexity of the above algorithm is
S(**Fact()**) ≥ 2(n+1)k bytes.

9. Calculate the space requirement for **[MODEL QUESTION]**
(i) Sequential search algorithm.
(ii) Find an element from a set of array elements.

Answer:

(i)

```
SeqSearch (a, n, k)
{
for index = 0 to n do
{
 if k ← a[index]
ans = index ;
            break;
 else
ans = -1;
}
return ans;
}
```

This is a simple sequential search algorithm, where we search whether an element k is in the list a[ ] or not. Variable **ans** returns the index value if the search is successful . Otherwise it returns -1.
There are four constant variables, k, index, ans and n. If the space required for a single constant variable is m bytes then the total space required for all the constant variables used above is C(**SeqSearch ()**) = 4 * m bytes.

The space required for the array depends on n and n is either equal to or less than the array size but not greater than a[ ]. So, the space required for the array a[ ] is at least (n . m) bytes, if m bytes of space are required for each array content.
The space complexity of the above algorithm is
S(**SeqSearch ()**) ≥ (4 + n).m bytes.

(ii)

```
FindMax(a, n)
{
    max = a[0];
 for index = 1 to n do
    {
 if max < a[index] then
max = a[index];
        }
return max;
}
```

There are three variables, max, index and n. So, it requires C(**FindMax ()**) = 3. k bytes, if the space required for a single constant variable is k bytes. The size of the array is also (n . k) bytes, which we have already discussed in the previous example.
The space complexity of the above algorithm is
S(**FindMax ()**) ≥ (3+n)k bytes.

10. Find the time complexity of an algorithm that computes the multiplication of n numbers stored in an array. **[MODEL QUESTION]**

Answer:

Multiply (a ,n)		**complexity**
	{	
Step 1	Initialize m ←1;	**O(1)**
Step 2	**for** i ←1 to n do	**O(n)**
Step 2.1	m= m * a[i];	**O(1)**
Step 3	return m;	**O(1)**
	}	

So the total time complexity of algorithm **Multiply ()** is **O(1)+O(n).O(1)+O(1) = O(n)**.

Long Answer Type Questions

1. Discuss different types of asymptotic notation. **[WBUT 2005, 2011, 2013]**

OR,

Define the following notations with example
Big-oh (O), Little-oh (o), Ω, θ **[WBUT 2015]**

OR,

Write the significance of different asymptotic notations (Big-O, Big-omega, Big-theta) with graphical analysis. **[WBUT 2017]**

Answer:

Big-O notation: Big-O notation is a theoretical measure of resource requirement of an algorithm. Usually time and memory needed, given the problem size n (which is usually the number of inputs) do count as resource. Informally speaking, some equation
f(n) = O(g(n)) means, f(n) is less than some constant multiple of g(n).
So, f(n) = O(g(n)) means there are positive constants c and k, such that
$0 \leq f(n) \leq cg(n)$ for all $n \geq k$. The value of c and k must be fixed for the function f and must not depend on n.

Ω *notation:* Ω notation is also a theoretical measure of the resource requirement of an algorithm. Usually time and memory needed, given the problem size n (which is usually the number of inputs) are treated as resource as before. Informally, saying some equation f(n) = Ω(g(n)) means it is more than some constant multiple of g(n). So, f(n) = Ω(g(n)) means there are positive constants c and k, such that $0 \leq cg(n) \leq f(n)$ for all $n \geq k$. The values of c and k must be fixed for the function f and must not depend on n.

***ω -notation*:** ω notation is also a theoretical measure of the resource requirement of an algorithm. Usually the time and memory needed, matter as resources, given the problem size n, (which is usually the number of inputs). Informally, saying some equation $f(n) = \omega(g(n))$ means $g(n)$ relative to $f(n)$ becomes insignificant as n goes to infinity. The notation is read as, "*f of n is little omega of g of n*".

Given any $\varepsilon > 0$ however small, there exists a $k > 0$ such that $|\frac{g(n)}{f(n)}| < \varepsilon$, whenever $n \geq k$. The value of k must not depend on n, but may depend on c. In other words, $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

Θ *notation*: This is also a theoretical measure of the execution of an algorithm, in terms of the time or memory needed, given the problem size n. Informally, saying some equation f(n) = Θ (g(n)) means it is within a spectrum induced by a pair of constant multiples of g(n). So, f(n) = Θ (g(n)) means there are positive constants c_1, c_2 and k, such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq k$. The values of c_1, c_2, and k must be fixed for the function f and must not depend on n.

***Little-o notation*:** This is yet another theoretical measure of the execution of an algorithm with respect to time and / or memory resources needed, given the problem size n, which is usually the number of input items. Informally, f(n) = o(g(n)) means f(n) becomes insignificant relative to g(n) as n approaches infinity. The notation is read as, "*f of n is little oh of g of n*".

Given any $\varepsilon > 0$ however small, there exists a $k > 0$ such that $\left|\frac{f(n)}{g(n)}\right| < \varepsilon$, whenever $n \geq k$. The value of k must not depend on n, but may depend on c. This is equivalent to $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

2. a) Solve the following recurrence relation using generating function:
$a_n = 6a_{n-1} - 11a_{n-2} + 4a_{n-3}$ **with initial condition a_0 =1, a_1= 2, a_2 = 1.**

[WBUT 2007, 2011]

Answer:

$A(s) = \sum_{n=0}^{\infty} a_n s^n$ is the generating function of the sequence $\{a_n, n \geq 0\}$.

Relation, $a_n = 6a_{n-1} - 11a_{n-2} + 4a_{n-3}$ $n \geq 3$ and $a_0 = 1$, $a_1 = 2$, $a_2 = 1$.
Multiplying both sides by s^n and sum over the range $n = 3$ to ∞.
Then

$$\sum_{n=3}^{\infty} a_n s^n = \sum_{n=3}^{\infty} 6a_{n-1}s^n - \sum_{n=3}^{\infty} 11a_{n-2}s^n + \sum_{n=3}^{\infty} 4a_{n-3}s^n$$

$$\text{or, } A(s) - a_2 s^2 - a_4 s - a_0 = 6s\sum_{n=3}^{\infty} a_{n-1}s^{n-1} - 11s^2\sum_{n=3}^{\infty} a_{n-2}s^{n-2} + 4s^3\sum_{n=3}^{\infty} a_{n-3}s^{n-3}$$

$$\text{or, } A(s) - s^2 - 2s - 1 = 6s(A(s) - a_1 s - a_0) - 11s^2(A(s) - a_0) + 4s^3 A(s)$$

$$\text{or, } A(s) - s^2 - 2s - 1 = 6s(A(s) - 2s - 1) - 11s^2(A(s) - 1) + 4s^3 A(s)$$

$$\text{or, } A(s) - s^2 - 2s - 1 = 6sA(s) - 12s^2 - 6s - 11s^2 A(s) - 11s^2 + 4s^3 A(s)$$

$$\text{or, } A(s)[1 - 6s + 11s^2 - 4s^3] = -12s^2 + s^2 + 11s^2 - 6s + 2s + 1 = -4s + 1$$

$$\text{or, } A(s) = (4s - 1)(4s^3 - 11s^2 + 6s - 1)$$

$$= (4s - 1)(T - 1)^{-1} \text{ where, } T = 4s^3 - 11s^2 + 6s$$

$$= (1 - 4s)(1 - T)^{-1} = (1 - 4s)[1 + T + T^2 + T^3 + \ldots]$$

$$= (1 - 4s)[1 + (6s - 11s^2 + 4s^3) + (\ldots.)^2 + (\ldots.)^3 + \ldots]$$

Collect coefficient of s^r terms in general, for $r \geq 0$ and that will be a_r, $r \geq 0$

b) What is the Tail Recursion? Give an example. **[WBUT 2007]**

Answer:

A function call is said to be tail recursive if there is nothing to do after the function returns except return its value. This is a special form of recursion where the last operation of a function is a recursive call. The recursion may be optimized away by executing the call in the current stack frame and returning its result rather than creating a new stack frame.
Although it may not save a lot of run time, it can save a lot of memory.
The following finds the maximum value in a list.

There is an example which is written in two different methods. First one is written in general recursive method and second one is written in tail recursive method. Later, we shall compare these two methods.

Max_list() is an algorithm to find the maximum element from a list A.

```
Max_list (A, max)
{
if (A[ ] = null)   then
return max;
if (max < head(A)) then
return max_list (tail(A), head(A));
else
return max_list (tail(A), max);
}
```

The return value of the current algorithm is just the return value of the recursive call. A compiler could optimize it in the following way of implementation. So it doesn't allocate new space for l and max on each invocation as is happening now.

```
Max_list ( A, max)
{
while( true)
{
if  (A[ ] =null) then
return max;
if (max < head(A)) then
{
max = head(A);
A = tail(A);
}
else
A = tail(A);
}
}
```

Now, there is no need to allocate new memory for the parameters. So this is more efficient.

3. State master's theorem and find the time complexity for the following recurrence: $T(n) = 2T(n^{1/2}) + \log n$. [WBUT 2009, 2016]

Answer:

Master Theorem

The Master Theorem applies to recurrences of the following forms:

$T(n) = aT(n/b) + f(n)$

Where $a \geq 1$ and $b > 1$ are constants and f(n) is an asymptotically positive function.

There are 3 cases:

1. If $f(n) = O(n^{\log_b a})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

3. If $f(n) = \Omega(n^{\log_b a+\varepsilon})$ with $\varepsilon \geq 0$ and f(n) satisfies the regularity condition, then T (n) = Θ (f(n)).

Regularity condition: $af(n/b) \leq cf(n)$ for some constant c < 1 and all sufficiently large n.
Time complexity for the following recurrence: $T(n) = 2T(n^{1/2}) + \log n$
Let us consider that, p = log n
So, $T(2^p) = 2T(2^{p/2}) + p$
Now, we further consider that $K(p) = T(2^p)$
So, $K(p) = 2K(p/2) + p$ = 2. 2 K(p/2.2) +p+p = 2^2 K(p / 2^2) +2.p
Subject to some terminating condition (i.e. not given here), K(p) = O (p log p)
So, T(n) = T(2^p) = K(p) = O(p log p) = O(log n log(log n))

4. Suppose we have a recurrence relation $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. Show that the followings: **[WBUT 2010, 2012]**

a) If $af\left(\frac{n}{b}\right) = kf(n)$ for some constant k < 1, then $T(n) = O(f(n))$

b) If $af\left(\frac{n}{b}\right) = kf(n)$ for some constant k > 1, then $T(n) = O(n^{\log_b a})$

c) If $af\left(\frac{n}{b}\right) = kf(n)$ for some constant k = 1, then $T(n) = O(f(n)\log_b n)$

Answer:

To prove this, we basically draw a general version of the recursion tree. Master Theorem is just a special case of the use of recursion trees. This statement of the Master Theorem gives only upper bounds - although you can see that our analysis below allows us to calculate T(n) explicitly (where n is a multiple of b) - and assumes f has a very particular form. It is actually possible to give accompanying lower bounds even with weaker restrictions on f.

Consider the equation $T(n) = aT\left(\frac{n}{b}\right) + f(n)$.

We start by drawing a recursion tree.

- The root contains the value f(n)
- It has a children, each of which contains the value f(n/b).
- Each of these nodes has a children, containing the value f(n/b^2)
- In general, level i contains a_i nodes with values f(n/b^i)
- Hence the sum of the nodes at the i-th level is a^if(n/b^i)

The tree stops when we get to the base case for the recurrence. We will assume that T(1) = f(1) = Θ (1) is the base case. Thus the depth of the tree is $\log_b n$ and there are $\log_b$ n+1levels.
Let T(n) be the sum of all values stored in all levels of the tree. From the recursion tree, once again assuming that N is a power of b, we see

$T(n) = f(n)+a\ f(n/b)+a^2\ f(n/b^2)+\cdots+a^i\ f(n/b^i)+\cdots+a^{L}\ f(n/b^{L})$ .....(1)
Where $L = \log_b n$ is the depth of the tree
Since $f(1) = \Theta(1)$, the last term of this summation is
$\Theta(a^L) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$
Now,

a) If $af\left(\frac{n}{b}\right) = kf(n)$ for some constant $k < 1$, then f(n) is a constant factor larger than a f(n/b), then the sum is a descending geometric series. The sum of any geometric series is a constant times its largest term. In this case, the largest term is the first term f(n). So, from equation 1,

$T(n) = f(n)+a\ f(n/b)+a^2\ f(n/b^2)+\cdots+a^i\ f(n/b^i)+\cdots+a^{L}\ f(n/b^{L})$
$= f(n)\ [1 + k + k^2 + \ldots + k^{\log_b N}]$
if $k < 1$, then the series is O (1) - as may be seen by the closed form of the sum – so,
$T(n) = O(f(n))$.

b) If f(n) is a constant factor smaller than a f(n/b), then the sum is an ascending geometric series. The sum of any geometric series is a constant times its largest term. In this case, this is the last term, which by our earlier argument is $T(n) = O\left(n^{\log_b a}\right)$.

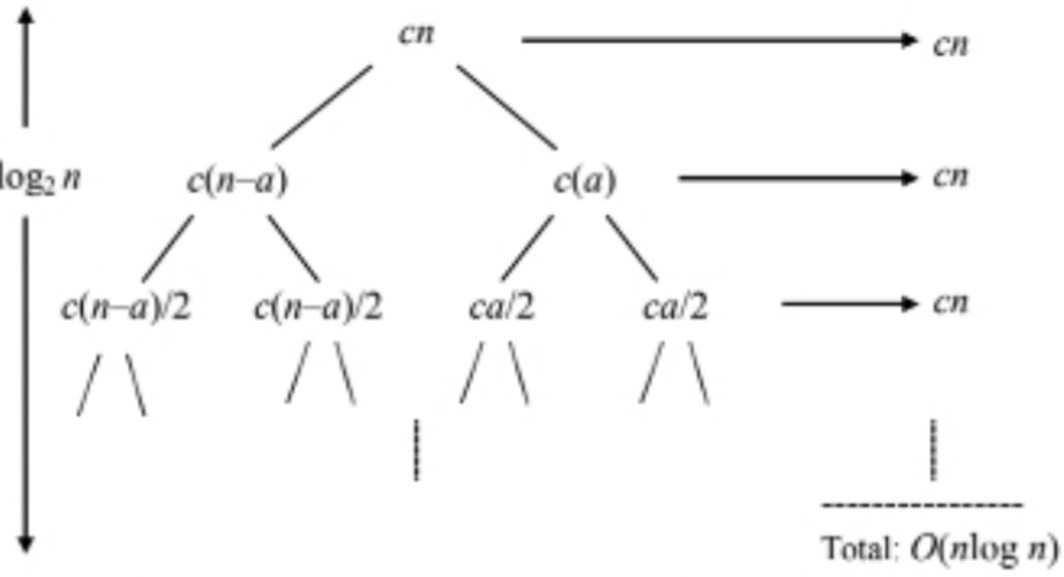
c) if $k = 1$, then all terms in the series are equal, and there are $\log_b n$ of them, so $T(n) = O(f(n)\log_b n)$.

5. State Master's theorem and find out the time complexity for the recurrence $T(n)=T(2n/3)+1$. **[WBUT 2012]**

Answer:
$T(n) = T(2n/3) + 1$, in which $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$.
[Let $a \geq 1$ and $b > 1$ be constants, let f (n) be a function, and let T (n) be defined on the nonnegative integers by the recurrence
$T(n) = aT(n/b) + f(n)$,
where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then T (n) can be bounded asymptotically as follows.
If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.]
Since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and thus the solution to the recurrence is
$T(n) = \Theta(\lg n)$.

6. Use the recursion tree to give an asymptotically tight solution to the recursion $T(n)=T(n-a)+T(a)+cn$ where $a>=1$ and $c>0$ are constant. **[WBUT 2012]**

Answer:
The recursion tree is



The recursion tree is full for $\log_2$n levels and each contributing cn, so we guess $\Omega(n\log n)$. It has $\log_2$ n levels, each contributing ≤ cn, so we guess O(nlog n).

7. Solve the recurrence relation using generating function $a_n - 7a_{n-1} + 10a_{n-2} = 0$, where $n \geq 2, a_0 = 10, a_1 = 41$. **[WBUT 2013]**

Answer:

Method I

Let $a_n = r^n$

So, the characteristic equation of the recurrence relation is $r^2 - 7r + 10 = 0$

Its roots are $r = 2$ and $r = 5$.

Hence the sequence $\{a_n\}$ is a solution to the recurrence relation if and only if

$$a_n = \alpha_1 2^n + \alpha_2 5^n$$

for some constant α_1 and α_2.

From the initial condition, it follows that

$$a_0 = 10 = \alpha_1 + \alpha_2 \quad \ldots. (1)$$
$$a_1 = 41 = 2\alpha_1 + 5\alpha_2 \quad \ldots. (2)$$

Solving the equations (1) and (2), we get $\alpha_1 = 3$, $\alpha_2 = 7$

Hence the solution is the sequence $\{a_n\}$ with $a_n = 3.2^n + 7.5^n$

Method II (Generating functions)

First, find the close form of the generating functions.

$$G(x) = \sum_{n=0}^{\infty} a_n x^n = (\sum_{n=2}^{\infty} a_n x^n) + a_1 x + a_0 = [\sum_{n=2}^{\infty} (7a_{n-1} - 10a_{n-2})x^n] + 41x + 10$$

$$= 7x(\sum_{n=2}^{\infty} a_{n-1} x^{n-1}) - 10x^2(\sum_{n=2}^{\infty} a_{n-2} x^{n-2}) + 41x + 10$$

$$= 7x(G(x) - a_0) - 10x^2 G(x) + 41x + 10 = 7xG(x) - 10x^2 G(x) - 29x + 10$$

So, we have, $G(x)\left(10x^2-7x+1\right)=-29x+10$

$$G(x)=\frac{-29x+10}{10x^2-7x+1}=\frac{-29x+10}{(1-2x)(1-5x)}=\frac{a}{(1-2x)}+\frac{b}{(1-5x)}$$

Then we can solve, $a=3$ and $b=7$

So, $G(x)=\frac{3}{(1-2x)}+\frac{7}{(1-5x)}=3\sum_{n=0}^{\infty}(2x)^n+7\sum_{n=0}^{\infty}(5x)^n=\sum(3.2^n+7.5^n)2x^n$

Thus, $a_n=3.2^n+7.5^n$

8. If $T(n)=\begin{cases}1, & n=1\\ 2T(n/2)+5n, & n>1\end{cases}$ **[WBUT 2015]**

Then show that $T(n)=O(n\log_2 \eta)$.

Show all steps of derivation.

Answer:

Let $a\geq 1$ and $b>1$ be constants and $f(n)$ be a function and $T(n)$ be defined on the non-negative integers by the recurrence

$T(n)=aT(n/b)+f(n)$, then

We can apply the second rule of Master Theorem,

If $f(n)=\Theta\left(n^{\log_b a}\right)$, then $T(n)=\Theta\left(n^{\log_b a}\lg n\right)$

Here, $a=2$, $b=2$ and $f(n)=5n$

Now, $n^{\log_b a}=n^{\log_2 2}=n$

i.e. $T(n)=\Theta(n^1\log_2 n)=O(n\log_2 n)$

9. Write short notes on the following:

a) Asymptotic notation **[WBUT 2008, 2012, 2014, 2016, 2019]**

b) Turing machines **[WBUT 2009]**

c) Recursion tree **[WBUT 2009, 2012, 2015, 2016]**

Answer:

a) Asymptotic notation:

Suppose we are considering two algorithms, *A* and *B*, for solving a given problem. Furthermore, let us say that we have done a careful analysis of the running times of each of the algorithms and determined them to be T_A (n) and T_B (n), respectively, where *n* is a measure of the problem size. Then it should be a fairly simple matter to compare the two functions T_A (n) and T_B (n) to determine which algorithm is *the best*.

One possibility arises if we know the problem size *a priori*. For example, suppose the problem size is n_0 and T_A (n) < T_B (n). Then clearly algorithm *A* is better than algorithm *B* for problem size n_0.

In the general case, we have not *a priori* knowledge of the problem size. However, if it can be shown, say, that T_A (n) ≤ T_B (n) for all n ≥ 0, then algorithm *A* is better than algorithm *B* regardless of the problem size.

Unfortunately, we usually don't know the problem size beforehand, nor is it true that one of the functions is less than or equal the other over the entire range of problem sizes. In this case, we consider the *asymptotic* behavior of the two functions for very large problem sizes.

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers **N** = {0, 1, 2, ...}. Such notations are convenient for describing the worst-case running-time function $T(n)$, which is usually defined only on integer input sizes.

Different types of asymptotic notations are,

Big-O notation

Ω-*notation*

ω -*notation*

Θ -*notation*

Little-o notation

b) Turing machines:

A Turning machine (TM) denoted by M, is defined as 7-tuple,

$M = (Q, \Sigma, \Gamma, \delta, q_0, \#, H)$.

Where,

$Q =$ a finite non-empty set of states,

$\Sigma =$ a non-empty set of input symbols (alphabets) which is a subset of Γ and $\# \notin \Sigma$,

$\Gamma =$ a finite non-empty set of tape symbols, i.e., $(\Sigma \cup \#)$

$\delta =$ the transition function $Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$,

It is mapping from the present state of automaton and tape symbol to next state, tape symbol and movement of head in left or right direction along the tape. This tells us that a Turing machine is in some present state $q \in Q$, after scanning an input symbol from $(\Sigma \cup \#)$ it goes to next state $q' \in Q$ by writing a symbol $x \in \Gamma$ in the current cell of input tape and finally takes a left or right move.

q_0 = the initial state, and $q_0 \in Q$

\# = the blank and $\# \in Q$

Make sure that # is not a symbol. A xell with # indicates that it does not contain any alphabet at all. Also δ may not be defined for some elements of $Q \times \Gamma$

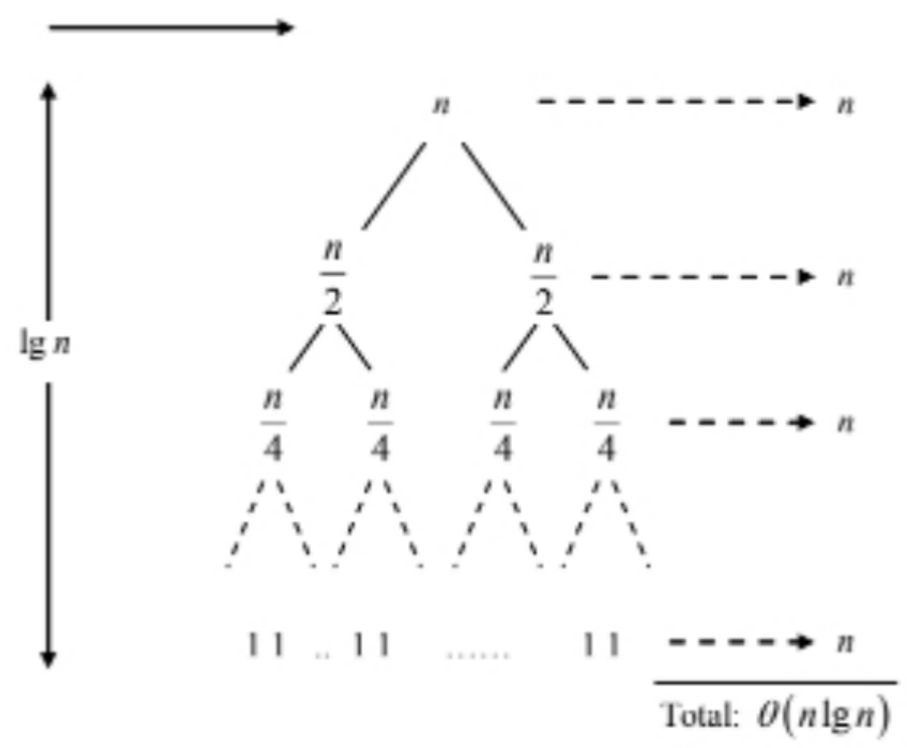
c) Recursion tree:

In ***recursive tree method***, we converts the recurrence into a tree whose nodes represent the costs incurred at each level in the tree, and sum the costs at each level to determine the total cost of all levels of the recursions. Recursion trees are particularly useful when the algorithm is based on divide-and-conquerer algorithm. A recursion tree can be used to visualize the iteration procedure.

Steps to construct recursion tree:

1. Draw the tree based on the recurrence
2. From the tree determine:
    a. # of levels in the tree
    b. cost per level
    c. # of nodes in the last level
    d. cost of the last level (which is based on the number found in 2c)
3. Write down the summation using $\sum$ notation – this summation sums up the cost of all the levels in the recursion tree
4. Recognize the sum or look for a closed form solution for the summation created in 3). Use Appendix A.
5. Apply that closed form solution to your summation coming up with your "guess" in terms of Big-O, or Θ, or Ω (depending on which type of asymptotic bound is being sought).
6. Then use Substitution Method or Master Method to prove that the bound is correct.

Example:



10. How to analysis the Algorithm? Describe the complexity of algorithms.
[MODEL QUESTION]

Answer:

Analysis of Algorithm: Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following −

- ***A Priori* Analysis** − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- ***A Posterior* Analysis** − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

Algorithm Complexity

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

1. Space Complexity: Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −

Algorithm: SUM(A, B)
Step 1 - START
Step 2 - C ← A + B + 10
Step 3 - Stop

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly

2. Time Complexity: Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes **n** steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

11. Calculate the time complexity of sequential search algorithm.

[MODEL QUESTION]

Answer:

[Initialize c = 0]

SeqSearch (a, n, k)

```
                {
Step 1          for index = 0 to n do
                {
                    [ c = c+1]                  // c is incremented 1 for
                          //'for' statement in Step 1.
                     [c = c+1]                        // c is incremented 1 for 'if'
                           //or 'else' statement in step 1.1
                                                //or Step 1.2.
Step 1.1            if k = = a[index]    then
Step 1.1.1      ans = index;
                          [c = c+1]             // c is incremented 1 for
                             //either step 1.1.1 or Step
                                                 //1.2.1 statement.
Step 1.1.2                 break;
                             [c = c+1]          // c is incremented 1 for
                           //'break' statement in Step 1.1.2.
Step 1.2        else
Step 1.2.1                 ans = -1;

                          }
                          [c = c+1]         //c is now incremented 1 for
                                  //last time execution of step 1.i.e
                                       //(n+1) th step execution of step 1.
Step 2                    return ans;     // c is incremented 1 for
                                  //'return' statement in Step 2.
                }
```

In the above example the time complexity of **SeqSearch ()** can be divided into two parts:

1. For successful search,

   T (**SeqSearch ()**) $\leq 3n + 1+1$

   If it is equal to the $(3n+2)$ then the search is successful at the last element of the array. Otherwise the search time is less than $(3n + 2)$.

2. For unsuccessful search,

   T (**SeqSearch ()**) $= 3n + 2$ i.e no element is matched with the search element.

FUNDAMENTAL ALGORITHMIC STRATEGIES

☞ Chapter at a Glance

- **Dynamic programming**: It is typically applied to optimization problems. There can be many possible solutions in such problems. There is a value of each solution and we have to find a solution with the optimal (minimum or maximum) value. We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value.
  There is a similarity between divide and conquer method and dynamic programming method. In divide and conquer method, we can divide the problem into many sub problems and solve them. Each time we have to solve each sub problem in divide and conquer method. In dynamic programming also we divide a problem into many sub problems. But unlike divide and conquer, here we do not solve the entire sub problem each time. In this method, we apply the result of a sub problem to another sub problem. So, it is more flexible than divide and conquer method.
  The development of a dynamic-programming algorithm can be broken into a sequence of four steps.
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution in a bottom-up fashion.
  4. Construct an optimal solution from computed information.
- **Traveling Salesman Problem: (TSP)**
  In the Traveling salesperson problem, a salesperson has to travel through all the cities in his domain exactly once and return to the starting city. The traveling salesperson problem is to find a tour of minimum cost. This traveling salesperson problem is similar to the Hamiltonian path problem. In Hamiltonian Path problem there may be more than one path from source vertex to visit all the vertices of the graph. The traveling salesperson problem is to find the minimum cost Hamiltonian path among all the Hamiltonian paths of that graph.
- **Backtracking:**
  The problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formulation. In many applications of the backtrack method, the desired solution is expressible as an n-tuple $(x_1, x_2,.. x_n)$, where the x_i are chosen from some finite set S_i. Often the problem to be solved calls for finding one vector that maximizes (or minimizes or satisfies) a criterion function $P(x_1, x_2,.. x_n)$.
- **8-Queen's Problem:**
  8-queens problem is one of the common problems. This happens to be an appropriate example, where backtracking may be used very effectively. The problem is to set 8 queens in an 8×8 chessboard that no two queens will attack each other. So, no two queens are in the same row or same column or same diagonal. In general, it can be described as to place n queens in an n×n chessboard in such a way that no two queens attack one another. The solution space required is $[8^2(8^2 - 1)\ (8^2 - 2)\ldots\ (8^2 - 7)]$ to solve this problem. So, in general we can say that for 8-queens problem, the solution space is $\prod_{i=0}^{8-1}(8^2 - i)$.

- **Greedy Method:**
  The characteristic of greedy method is basically same as that of solving a typical optimization problem preferably of time infeasible nature. The components of a typical greedy method are:
  1. A set of elements like nodes, edges in a graph.
  2. A set of elements which have already been used.
  3. To test whether a given set of elements provide a solution or not. However, the solutions may not be optimal.
  4. A selection function that picks up some elements which have not yet been used.
  5. An objective function which associates a value to a solution.
- **Knapsack Problem:**
  The knapsack problem states that we have to fill a knapsack having constant weight, with some objects having different weights and different profit values. We have to fill up the knapsack with objects in such a way that the total weight of selected objects does not cross the limit of the knapsack and we get the maximum profit.
  Suppose, The capacity of the knapsack is c. There are n many objects. An individual object is represented by i where i=1,2,...n. Weight of i-th object is z_i , $1 \le i \le n$. Profit for i-th object is a_i , $1 \le i \le n$.
  We can put maximum one unit of each object in the knapsack or some fractional part of that object. So, we consider that some fraction x_i, $0 \le x_i \le 1$ of the object i is kept into the knapsack.
  Now, the weight of the knapsack is $\sum_{i=1}^{n} z_i x_i$ and profit is $A = \sum_{i=1}^{n} a_i x_i$. so we can define this problem as:
  Maximize $A = \sum_{i=1}^{n} a_i x_i$ ...(1)
  Subject to $\sum_{i=1}^{n} z_i x_i \le C$ ...(2) and $0 \le x_i \le 1$, $0 \le i \le n$.

Multiple Choice Type Questions

1. Time complexity for the Floyd's algorithm to find all pairs of shortest path of a graph G with V vertices and E edges using dynamic programming method is **[WBUT 2007, 2015]**

a) O (V^2) b) O (E^2) c) O (V^3) d) O (E^3)

Answer: (c)

2. The edge, removal of which makes a graph disconnected is called [WBUT 2007]

a) pendant vertex b) bridge
c) articulation point d) colored vertex

Answer: (b)

3. The minimum number of colors needed to color a graph having n > 3 vertices and 2 edges is **[WBUT 2007, 2014, 2016]**

a) 2 b) 3 c) 4 d) 1

Answer: (b)

4. Which of the following algorithm design techniques is used in the quick sort algorithm? [WBUT 2008, 2010, 2016, 2018]

a) Dynamic programming b) Backtracking
c) Divide and conquer d) Greedy method

Answer: (c)

5. Kruskal algorithm is a [WBUT 2009, 2012]

a) Divide & conquer algorithm b) Branch and bound algorithm
c) Greedy algorithm d) Dynamic programming

Answer: (c)

6. Optimal substructure property is exploited by [WBUT 2009, 2013]

a) Dynamic programming b) Greedy method
c) Both (a) & (b) d) None of these

Answer: (a)

7. Which of the following approaches is adopted in Divide & Conquer algorithms?

a) Top- down b) Bottom-up [WBUT 2009, 2018]
c) Both (a) & (b) d) None of these

Answer: (a)

8. The fractional Knapsack problem can be solved by using [WBUT 2010, 2016]

a) Greedy method b) Divide and Conquer method
c) Dynamic programming d) none of these

Answer: (a)

9. Time complexity of Binary Search algorithm on n items is [WBUT 2010, 2016]

a) $O(n)$ b) $O(n\log n)$ c) $O(n^2)$ d) $O(n\log n)$

Answer: correct answer is $O(\log_2 n)$

11. Which of the following cannot be performed recursively? [WBUT 2011, 2013]

a) binary search b) quick sort c) DFS d) none of these

Answer: (d)

12. In which sorting technique, is an element placed in its proper position at each step? [WBUT 2011]

a) Bubble sort b) Quick sort c) Merge sort d) Heap sort

Answer: (a)

13. The average number of comparisons performed by merge sort algorithm in merging '2' sorted lists of length '2' is [WBUT 2011, 2018, 2019]

a) 8/5 b) 11/7 c) 11/6 d) 8/3

Answer: (d)

14. Which of the following design techniques is used in the quick-sort algorithm? **[WBUT 2011, 2013, 2018, 2019]**

a) Dynamic programming b) Back tracking
c) Greedy method d) Divide and conquer

Answer: (d)

15. The time-complexity of TSP is **[WBUT 2011]**

a) $O(n^2 2^n)$ b) $\Theta(n^2 2^n)$ c) $\Omega(n^2 2^n)$ d) none of these

Answer: (a)

16. Which of the following algorithms solves the All-Pair Shortest Path problem? **[WBUT 2011, 2016]**

a) Dijkstra's b) Floyd's Warshall's c) Prim's d) Kruskal's

Answer: (b)

17. Best case time complexity for Binary search in unsuccessful case is **[WBUT 2012, 2018]**

a) $O(l)$ b) $O(\log n)$ c) $O(n)$ d) None of these

Answer: (b)

18. The tight bound for building a max heap is **[WBUT 2012, 2016]**

a) $O(n)$ b) $O(\log_2 n)$ c) $O(n.\log_2 n)$ d) none of these

Answer: (c)

19. The worst case running time of a quick sort algorithm is **[WBUT 2012]**

a) $O(n^2)$ b) $O(n.\log_2 n)$ c) $O(n)$ d) $O(\log_2 n)$

Answer: (a)

20. Binary Search algorithm can't be applied to **[WBUT 2012]**

a) sorted linked lists b) sorted binary trees
c) sorted linear array d) sorted integer array

Answer: (a)

21. Kruskal's algorithm uses ..... and Prim's algorithm uses ..... in determining the MST. **[WBUT 2013]**

a) edges, vertex b) vertex, edges c) edges, edges d) vertex, vertex

Answer: (a)

22. Match the following **[WBUT 2014]**

a) Fractional Knapsack i) Greedy Algorithm
b) 0-1 Knapsack ii) Dynamic Programming Algorithm

a) a-i and b-i b) a-i and b-ii c) a-ii and b-i d) a-ii and b-ii

Answer: (b)

23. The average number of comparisons performed by merge sort algorithm in merging two sorted lists of 2 elements is **[WBUT 2014]**

a) 8/5 b) 11/7 c) 11/6 d) 8/3

Answer: (d)

24. Which of the following sorting methods would be most suitable for sorting a list which is almost sorted **[WBUT 2014]**

a) bubble sort b) insertion sort c) selection sort d) quick sort

Answer: (b)

25. Prime's algorithm is an example of **[WBUT 2014]**

a) backtracking b) dynamic programming
c) greedy method d) none of these

Answer: (c)

26. Kruskal's Algorithm for finding minimum spanning tree is an example of

a) Dynamic programming b) Greedy method **[WBUT 2015]**
c) Both (a) and (b) d) None of these

Answer: (b)

27. Given two sorted lists of size "m" and "n" respectively. The number of comparisons needed in the worst case by merge sort will be **[WBUT 2015]**

a) $m*n$ b) $\text{Max}(m,n)$ c) $\text{Min}(m,n)$ d) $m+n-1$

Answer: (a)

28. The running time $T(n)$ where "n" is the input size of a recursive algorithm is given by **[WBUT 2015]**

$$T(n) = c + T(n-1), \quad \text{if } n > 1$$
$$= d, \quad \text{if } n < 1$$

The order of the algorithm is

a) n^2 b) n c) n^3 d) n^n

Answer: (b)

29. Travelling salesman problem is **[WBUT 2015, 2016]**

a) P b) NP c) NP-complete d) NP-Hard

Answer: (d)

30. Locally best computation is done in **[WBUT 2017]**

a) Dynamic programming b) Greedy method
c) both (a) and (b) d) none of these

Answer: (b)

31. Which of the following algorithm design techniques is used for solving graph coloring problem? **[WBUT 2017]**

a) Divide and conquer
b) Backtracking
c) Dynamic programming
d) Greedy method

Answer: (b)

32. The total running time of matrix chain multiplication of n matrices is **[WBUT 2017]**

a) $o(n^4)$ b) $o(n^3)$ c) $o(n^2)$ d) $o(n)$

Answer: (b)

33. The sub-problems in Divide and Conquer are considered to be **[WBUT 2017]**

a) distinct b) overlapping c) large size d) small size

Answer: (d)

34. Which of the following algorithm design techniques is used in merge sort?

a) Dynamic programming
b) Backtracking **[WBUT 2017]**
c) Divided and conquer
d) Greedy method

Answer: (c)

35. Which of the following standard algorithm is not based on Dynamic Programming? **[WBUT 2018, 2019]**

a) Bellman-Ford Algorithm for single source shortest path
b) Floyd Warshall Algorithm for all pairs shortest paths
c) 0-1 Knapsack problem
d) Prim's Minimum Spanning Tree

Answer: (d)

36. Time complexity of Quick sort in worst case is **[WBUT 2018]**

a) $O(n)$ b) $O(\log n)$ c) $O(n^2)$ d) $O(n \log n)$

Answer: (c)

37. A machine needs a minimum of 100 ms to sort 1000 names by quick sort. The minimum time needed to sort 100 names will be approximately **[WBUT 2018, 2019]**

a) 50.2 ms b) 6.7 ms c) 72.7 ms d) 11.2 ms

Answer: (d)

38. What is the time complexity to insert an element into a heap? **[WBUT 2018, 2019]**

a) $O(n \log n)$ b) $O(\log n)$ c) $O(n)$ d) None of these

Answer: (b)

39. Best case time complexity for Binary search in unsuccessful case is **[WBUT 2019]**

a) *O (n)* b) *O (log n)* c) *O (1)* d) *O (n log n)*

Answer: (b)

40. Which of the following approaches is Divide & Conquer strategy? [WBUT 2019]

a) Top-down b) Bottom-up c) Both (a) & (b) d) None of these

Answer: (a)

41. A machine needs 200 ms to sort 200 names, using bubble sort. In 800 ms, it can approximately sort **[WBUT 2019]**

a) 400 names b) 800 names c) 750 names d) 1800 names

Answer: (a)

42. Which of the following is not a backtracking algorithm? [WBUT 2019]

a) N queen problem b) Tower of Hanoi

c) M coloring problem d) None of these

Answer: (d)

Short Answer Type Questions

1. What is union-find algorithm? [WBUT 2004, 2005, 2007, 2011, 2012]

OR,

What is union-find algorithm? Explain with an example. [WBUT 2015]

OR,

Write short note on Union Find Algorithm. [WBUT 2018, 2019]

Answer:

Union-Find Algorithm of disjoint sets

A disjoint set structure maintains a partition of a set $A = (A_1, A_2, \ldots A_n)$ such that no element belongs to more than one set simultaneously. For each set A_i in the partition, a representative, which is some member in the set, is kept track of. The initial partition has each element in a set by itself. Suppose A_i and A_j are two different sets where $i \neq j$. If they are disjoint sets, then there is no element that is both in A_i and A_j.

Union of disjoint-set

Union (A_i, A_j), where $i \neq j$, replaces A_i and A_j, and specifies a representative for $A_i \cup A_j$, i.e. the union of two disjoint sets A_i and A_j create a new set in which all the elements of A_i and A_j are kept and whose representative is x, where $x \in A_i$ or $x \in A_j$ is a representative of either A_i or A_j.

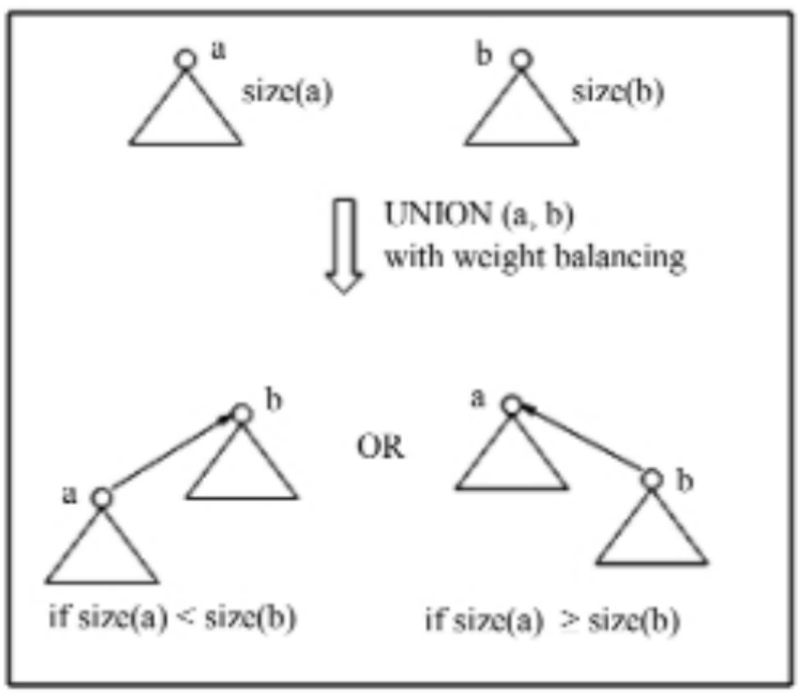
Algorithm

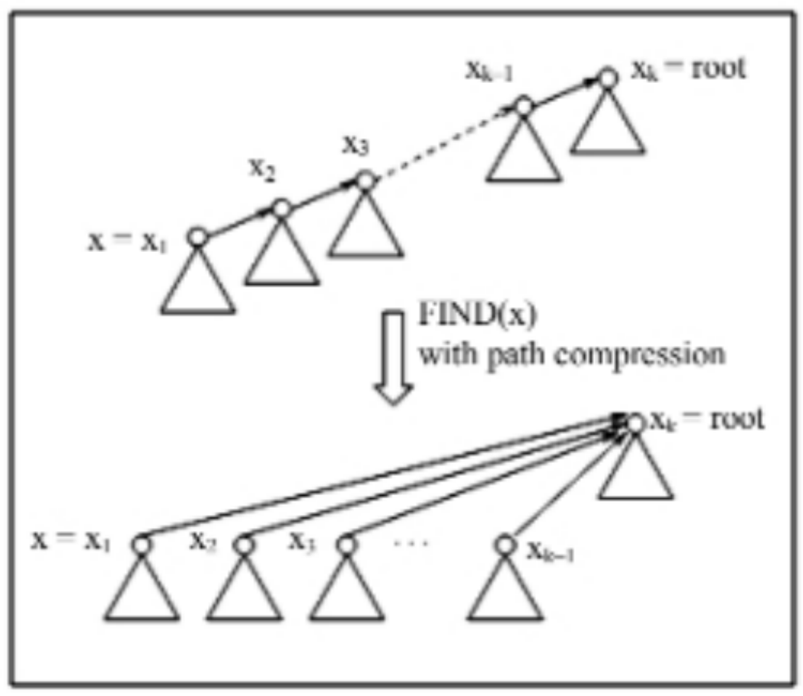
```
Union (x, y)
{
   parent[x] = y;
}
```

Find operation

Find(x) is used to find out the set which the element x belongs to.

```
Find(x)
{
   if (x≠parent[x]) then
   {
      parent[x] ←Find( parent [x]);
      return parent[x];
   }
   else
      return "not found";
}
```





2. What is the basic characteristic of a Greedy algorithm?

[WBUT 2004, 2006, 2008, 2012, 2015]

OR,

What do you mean by greedy method? **[WBUT 2017]**

Answer:

The characteristic of greedy method is basically same as that of solving a typical optimization problem preferably of time infeasible nature. The components of a typical greedy method are:

1. A set of elements like nodes, edges in a graph.
2. A set of elements which have already been used.
3. To test whether a given set of elements provide a solution or not. However, the solutions may not be optimal.
4. A selection function that picks up some elements which have not yet been used.
5. An objective function which associates a value to a solution.

3. State the 0/1 Knapsack problem. **[WBUT 2004, 2007, 2012, 2013]**

OR,

What is 0/1 knapsack problem? Explain it with an example. **[WBUT 2010]**

OR,

Critically comment on "Greedy strategy does not work for the 0-1 knapsack problem for all time". **[WBUT 2016]**

Answer:

A thief robbing a store finds n items. the i-th item is worth v_i rupees and weighs w_i Kgs, where v_i and w_i are integers. He wants to take as valuable a load as possible, but he can carry at most W Kgs in his knapsack for some integer W. Greedy strategy does not work for the 0-1 knapsack problem.

In the 0-1 problem, when we consider an item for inclusion in the knapsack, we must compare the solution to the sub-problem in which the item is included with the solution to the sub-problem in which the item is excluded before we can make the choice. The problem formulated in this way gives rise to many overlapping sub-problems— dynamic programming can be used to solve the 0-1 problem.

Let i be the highest-numbered item in an optimal solution S for W Kgs and items 1, . . . , n. Then $\acute{S} = S - \{i\}$ must be an optimal solution for

$W - w_i$ rupees and items 1, . . . , i − 1, and the value of the solution S is v_i plus the value of the sub-problem solution $\acute{S}$.

We can express this relationship in the following formula:

Define c[i,w] to be the value of the solution for items 1, . . . , i and maximum weight w.

Then

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,w] & \text{if } w_i > w \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & \text{if } i > 0 \text{ and } w \geq w_i \end{cases}$$

The last case says that the value of a solution for i items either includes item i , in which case it is v_i plus a sub-problem solution for i − 1 items and the weight excluding w_i, or doesn't include item i, in which case it is a sub-problem solution for i − 1 items and the same weight. That is, if the thief picks item i, he takes v_i value, and he can choose from items 1, . . ., i − 1 up to the weight limit $w - w_i$, and get $c[i-1, w-w_i]$ additional value. On the other hand, if he decides not to take item i, he can choose from items 1, . . ., i − 1 up to the weight limit w, and get

c[i − 1,w] value. The better of these two choices should be made.

The algorithm takes as inputs the maximum weight W, the number of items n, and the two sequences $v = (v_1, v_2, \ldots, v_n)$ and $w = (w_1, w_2, \ldots, w_n)$. It stores the c[i, j] values in a table c[0 . . n, 0 . .W] whose entries are computed in row-major order. (That is, the first row of c is filled in from left to right, then the second row, and so on.) At the end of the computation, c[n, W] contains the maximum value the thief can take.

4. Find out the best and worst case time complexity of merge sort.
[WBUT 2007, 2011, 2013]

OR,

Derive the complexity of merge sort. **[WBUT 2009, 2013]**

OR,

Analyze the time complexity of Merge Sort algorithm. **[WBUT 2018, 2019]**

Answer:

Best case

In each iteration, we split the array into two sub-lists, and recursively invoke the algorithm. At best case we split it exactly to half, and thus we reduce the problem (of each recursive call) to half of the original problem. We need $\log_2(n)$ iterations, and each iteration takes exactly O(n) (each iteration is on all sub lists, total size is still n), so at total time complexity $O(\log_2 n)$.

Worst case

In Merge sort all the comparisons are made in the procedure *Merge* which can merge two sorted sub arrays. If the size of one sub array is **s** and that of the other is **t** then merging these lists would require at most s+t–1 comparisons in the worst case.

So, W (n)=W(s)+W(t)+s + t –1

If $s = \lfloor n/2 \rfloor$ and $t = \lceil n/2 \rceil$ then s + t = n and

$W(n) = W(n/2) + W(n/2) + \Theta(n)$, where $\Theta(n) = n-1$

So, $W(n) = W(n/2)+W(n/2)+ \Theta(n)$

$= 2W(n/2)+ \Theta(n)$

$= 2^1 W(n/2)+ \Theta(n)$

$= W(n/4)+W(n/4)+W(n/4)+W(n/4)+ \Theta(n)$

$= 4W(n/4)+ \Theta(n)$

$= 2^2 W(n/4)+ \Theta(n)$

....

$= 2^k W(1)+\Theta(n)$, where $n > 1$ and $n = 2^k$

Also, W (1) = 1, for n=1

So, we can represent $k = \log_2 n$

Thus the total running time of Merge sort algorithm in worst case is $O(n \log_2 n)$.

5. Find the best and worst case time complexity of quick sort.
[WBUT 2008, 2012, 2015, 2016, 2019]

Answer:

Best case

- In the best case, the pivot is in the middle position of the array.
- To simplify the equations, we assume that the two sub arrays are each exactly half the length of the original one. So, we get $T(n) = 2T(n/2) + cn$, c>0 constant (independent of n) and $n \geq 2$ with T(1) = 1.
- This is very similar to the formula for Merge sort, and a similar analysis leads to $T(n) = cn \log_2 n + n$ which is $O(n \log_2 n)$.

Worst case

- If the pivot is always the smallest element, then $i = 0$ always
- We ignore the term $T(0)=1$, so the recurrence relation is $T(n)= T(n–1)+cn$
- So, $T(n–1)=T(n–2)+c(n–1)$ and so on until we get $T(2) = T(1) + c(2)$
- Substituting backwards, we get $T(n) = T(1) + c(n + \ldots + 2) = O(n^2)$
- It may be noted that this case happens if we always take the pivot to be the first element in the array and the array is already sorted.

6. What are the basic characteristics of dynamic programming? [WBUT 2009]

OR,

What do you mean by dynamic programming? [WBUT 2014]

Answer:

Dynamic programming is a method for efficiently solving a broad range of search and optimization problems which exhibit the characteristics of overlapping sub problems and optimal substructure. Optimal substructure means that optimal solutions of sub problems can be used to find the optimal solutions of the overall problem.

Greedy method is an algorithm that always takes the best immediate, or local, solution while finding an answer. Greedy algorithms find the overall, or globally, optimal solution for some optimization problems, but may find less-than-optimal solutions for some instances of other problems.

7. Write an algorithm of matrix chain multiplication. [WBUT 2010, 2013, 2018, 2019]

OR,

Write the algorithm of chain matrix multiplication. [WBUT 2014]

Answer:

This is the matrix chain multiplication algorithm which can generate the actual position of the parenthesis within the matrix chain $(A_1A_2..A_n)$.

```
Matrix-Chain-Order (p)
{
  n ←.length [p] - 1
  for i ← 1 to n do
      m[i, i] ← 0
  for l←2 to n do
  {
    for i←1 to n – l +1 do
    {
       j ← i + l -1;
       m[i, j] ← ∞;
       for  k ← i  to j-1 do
       {
          q ← m [i, k] + m [k+1, j] + pi-1.pk .pj
          if  q < m [i, j]  then
          {
 m [i, j] = q;
 s [i, j] =k;
}
```

```
            }
          }
        }
        return m and s;
    }
```

Optimal parenthesis position of the product of matrix chain ($A_1A_2..A_n$)

The algorithm below reports the optimal positions of the parentheses in the matrix chain and this can be shown by the help of table s.

Optimal parenthesis position of the product of matrix chain ($A_1A_2..A_n$)

The algorithm below reports the optimal positions of the parentheses in the matrix chain and this can be shown by the help of table s.

The following algorithm is:

```
Optimal parenthesis (s, i, j)
{
  if ( i = j) then
          print "A"
          else
  print "( "
  Optimal_parenthesis (s, i, s[i,j])
  Optimal_parenthesis (s, s[i, j] +1, j)
  print " )"
}
```

Here *l* is the length of the chain.

Now, in step 1, we first measure n which is equal to the length of the chain minus one.

In step 2.1, we make all m[i, i] = 0 for i= 1,2,..,n.

In step 3, for all value of *l* from 2 to n, we first evaluate the value of m[i, i+1], where $1 \leq i < n$. Next we compute m[i, i+2] and so on.

In step 3.1.3.1, we can compute the value of y according to the equation,

$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$, where, $i \leq k < j$).

Through steps 3.1.3.2 to 3.1.3.2.2, we can evaluate the minimum value of *y* according to the equation,

$$m[i, j] = 0 \quad \text{if } i = j$$

$$= \min_{i \leq k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} \quad \text{if } i < j$$

and put this value into the array **m** which is actually the ***m*** table discussed above and store the position of the parenthesis in the table **s**.

Time complexity of Matrix-Chain-Order (*x*):

One simple solution is called memorization: each time we compute the minimum cost needed to multiply out a specific subsequence, we save it. If we are ever asked to compute it again, we simply report the saved answer, and do not recomputed it. Since there are about n2/2 different subsequences, where n is the number of matrices, the space required to do this is reasonable. It can be shown that this simple trick brings the runtime down from O (2^n) to O (n^3), which is efficient enough for real applications. However, this is done at the cost of additional memory resource.

8. Write an algorithm of eight queens problem and find time complexity of the algorithm. Explain the algorithm using an example. [WBUT 2010, 2011]

OR,

Solve 8 Queen problem using Backtracking approach. [WBUT 2018, 2019]

Answer:

Algorithm: ***Refer to Question No. 3(i) of Long Answer Type Questions.***

Example

Here we take an example of 4-queen problem. In figure 1 there are some nodes denoted by label 'B' means backtracking. That means the path from root to the leaf node cannot reach the goal state. i.e. this path cannot generate a solution vector of appropriate position of each queen. So we have to backtrack from that node itself. For those x_i values two queens are in attacking position and then we use backtracking. For each path of the tree from root to a node marked B in the figure we represent the position of the queen in the chessboard. In figure, 'R' represents the result or solution space of 4-queens problems. i.e. through the path from root to the leaf node we can reach a solution, where each queen is not in mutual attacking position.

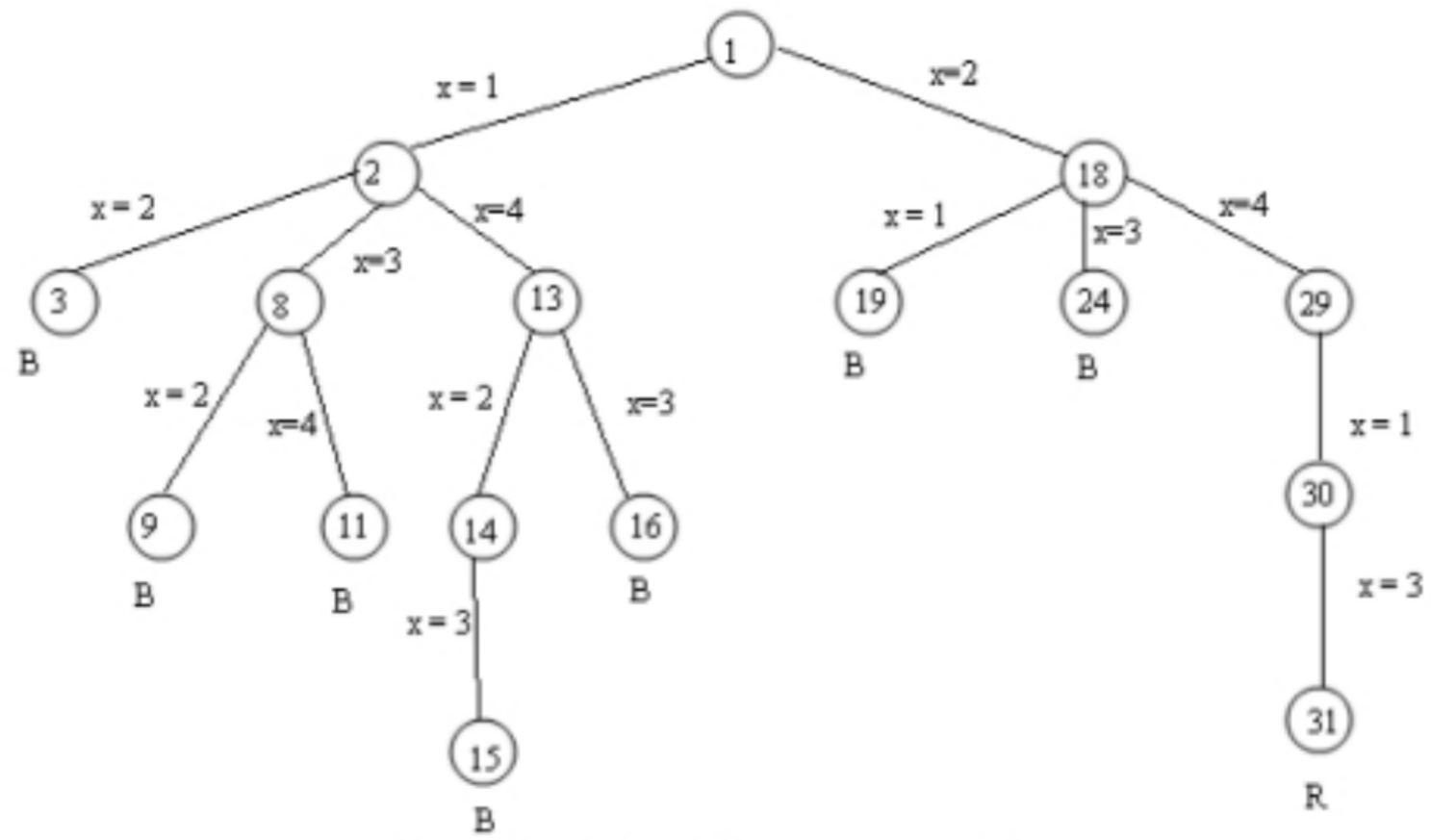


Fig: 1: Solution tree for 4-queens problem

9. a) Find the optimal parenthesization of a matrix-chain product whose sequence of dimensions is < 5, 10, 3, 12, 5, 50, and 6 >.

b) Give an algorithm for the above procedure.

c) Analyze its complexity. [WBUT 2011, 2013]

Answer:

a) We can represent $A_1, A_2, A_3, A_4, A_5, A_6$ in such a way, that,

$A_1 \rightarrow p_0 \times p_1$

$A_2 \rightarrow p_1 \times p_2$

$A_3 \rightarrow p_2 \times p_3$

$A_4 \rightarrow p_3 \times p_4$
$A_5 \rightarrow p_4 \times p_5$
$A_6 \rightarrow p_5 \times p_6$
where, $p_0 = 5$, $p_1 = 10$, $p_2 = 3$, $p_3 = 12$, $p_4 = 5$, $p_5 = 50$, $p_6 = 6$
Now,
$m[1, 1] = 0$
$m[2, 2] = 0$
$m[3, 3] = 0$
$m[4, 4] = 0$
$m[5, 5] = 0$
$m[5, 5] = 0$
Next, we can find out, that
$m[1, 2] = m[1, 1] + m[2, 2] + p_0 \cdot p_1 \cdot p_2 = 150$
$m[2, 3] = m[2, 2] + m[3, 3] + p_1 \cdot p_2 \cdot p_3 = 360$
$m[3, 4] = m[3, 3] + m[4, 4] + p_2 \cdot p_3 \cdot p_4 = 180$
$m[4, 5] = m[4, 4] + m[5, 5] + p_3 \cdot p_4 \cdot p_5 = 3000$
$m[5, 6] = m[5, 5] + m[6, 6] + p_4 \cdot p_5 \cdot p_6 = 1500$
Now, we can calculate the following

$$m[1, 3] = \min \begin{cases} m[1,1] + m[2, 3] + p_0 \cdot p_1 \cdot p_3 \\ m[1,2] + m[3, 3] + p_0 \cdot p_2 \cdot p_3 \end{cases} = 330$$

$$m[2, 4] = \min \begin{cases} m[2, 2] + m[3, 4] + p_1 \cdot p_2 \cdot p_4 \\ m[2, 3] + m[4, 4] + p_1 \cdot p_3 \cdot p_4 \end{cases} = 330$$

$$m[3,5] = \min \begin{cases} m[3,3] + m[4,5] + p_2 \cdot p_3 \cdot p_5 \\ m[3,4] + m[5,5] + p_2 \cdot p_4 \cdot p_5 \end{cases} = 930$$

$$m[4,6] = \min \begin{cases} m[4,4] + m[5,6] + p_3 \cdot p_4 \cdot p_6 \\ m[4,5] + m[6,6] + p_3 \cdot p_5 \cdot p_6 \end{cases} = 1860$$

$$m[1, 4] = \min \begin{cases} m[1, 1] + m[2, 4] + p_0 \cdot p_1 \cdot p_4 \\ m[1, 2] + m[3, 4] + p_0 \cdot p_2 \cdot p_4 \\ m[1, 3] + m[4, 4] + p_0 \cdot p_3 \cdot p_4 \end{cases} = 405$$

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 \cdot p_2 \cdot p_5 \\ m[2,3] + m[4,5] + p_1 \cdot p_3 \cdot p_5 \\ m[2,4] + m[5,5] + p_1 \cdot p_4 \cdot p_5 \end{cases} = 2430$$

$$m[3,6] = \min \begin{cases} m[3,3] + m[4,6] + p_2 \cdot p_3 \cdot p_6 \\ m[3,4] + m[5,6] + p_2 \cdot p_4 \cdot p_6 \\ m[3,5] + m[6,6] + p_2 \cdot p_5 \cdot p_6 \end{cases} = 1770$$

$$m[1,5] = \min \begin{Bmatrix} m[1,1] + m[2,5] + p_0.p_1.p_5 \\ m[1,2] + m[3,5] + p_0.p_2.p_5 \\ m[1,3] + m[4,5] + p_0.p_3.p_5 \\ m[1,4] + m[5,5] + p_0.p_4.p_5 \end{Bmatrix} = 1655$$

$$m[2,6] = \min \begin{Bmatrix} m[2,2] + m[3,6] + p_1.p_2.p_6 \\ m[2,3] + m[4,6] + p_1.p_3.p_6 \\ m[2,4] + m[5,6] + p_1.p_4.p_6 \\ m[2,5] + m[6,6] + p_1.p_5.p_6 \end{Bmatrix} = 1950$$

$$m[1,6] = \min \begin{Bmatrix} m[1,1] + m[2,6] + p_0.p_1.p_6 \\ m[1,2] + m[3,6] + p_0.p_2.p_6 \\ m[1,3] + m[4,6] + p_0.p_3.p_6 \\ m[1,4] + m[5,6] + p_0.p_4.p_6 \\ m[1,5] + m[6,6] + p_0.p_5.p_6 \end{Bmatrix} = 2010$$

j \ i	1	2	3	4	5	6
6	2010	1950	1770	1860	1500	0
5	1655	2430	930	3000	0	
4	405	330	180	0		
3	330	360	0			
2	150	0				
1	0					

Table m

j \ i	1	2	3	4	5
6	2	2	4	4	5
5	4	2	4	4	
4	2	2	3		
3	2	2			
2	1				

Table s

b) & c) ***Refer to Question No. 7 of Short Answer Type Questions.***

10. Differentiate between divide-and-conquer and dynamic programming.
[WBUT 2011, 2013, 2015]

Answer:

Divide-and-conquer	Dynamic programming
1. Solves problem by dividing problems into sub-problems. 2. In divide-and-conquer the sub problems are independent of each other. 3. Divide-and-conquer does more work on the sub problems and hence has more time consumption. **Examples:** Merge sort, Quick sort and Binary search.	1. Solves problem by dividing problems into sub-problems. 2. In dynamic programming the sub problems are not independent. 3. Dynamic programming solves the sub problem only once and then stores it in table. **Examples:** Matrix-chain-multiplications, All-pair shortest path problem.

11. a) Discuss the Bellman-Ford's algorithm for single-source shortest path problem.
b) Prove that the time-complexity of the algorithm is $\Theta(VE)$. [WBUT 2012, 2017]

Answer:

a) The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph G = (V, E) with source s and weight function w : E → R

```
BELLMAN-FORD(G,w, s)
{
INITIALIZE-SINGLE-SOURCE(G, s)
  for i ← 1 to |V[G]| − 1 do
  {
     for each edge (u, v) ∈ E[G] do
      {
        RELAX(u, v,w)
      }
  }
  for each edge (u, v) ∈ E[G] do
  {
        if d[v] > d[u] + w(u, v) then
         {
           return FALSE
         }
  }
return TRUE
}
```

- RELAX(*u, v,w)*

```
{
 if d[v] > d[u] + w(u, v) then
    {
```

```
            d[v] ← d[u] + w(u, v)
            π[v]← u
    }
}
```

- INITIALIZE-SINGLE-SOURCE*(G, s)*

```
{
  for each vertex v ∈ V[G] do
  {
        d[v]← ∞
        π[v]← NIL
  }
 d[s] ← 0
}
```

b) The initialization in INITIALIZE-SINGLE-SOURCE*(G, s)* takes O*(V)* time and each of the $|V|-1$ passes over the edges in lines 2–4 of BELLMAN-FORD *(G, w, s)* takes O*(E)* time and the **for** loop of lines 5–7 takes *O(E)* time. So, the Bellman-Ford algorithm runs in time *O(V E)*

12. What is meant by union by rank? Explain with an example. [WBUT 2013]

Answer:

We know that in **Union (x, y)**, one can either create a pointer from x to y, or a pointer from y to x. The **Union by Rank** rule specifies a way of making the choice by which we tend to keep the tree shallow. Associate with each node x a rank **rank(x)** which is initially 0. However, how to determine rank of a node x will be discuss later. When a Union requires choosing either a pointer from x to y or y to x, the pointer should run from the node of lower rank to the one of higher. If the ranks are equal, then the choice is arbitrary. However, the node to which the new pointer is directed gets its rank incremented by 1. The basic properties to assign rank to a node are given below.

(1) A node of rank k has at least one child of each having rank < k,

(2) The rank of any node is less than the rank of its parent,

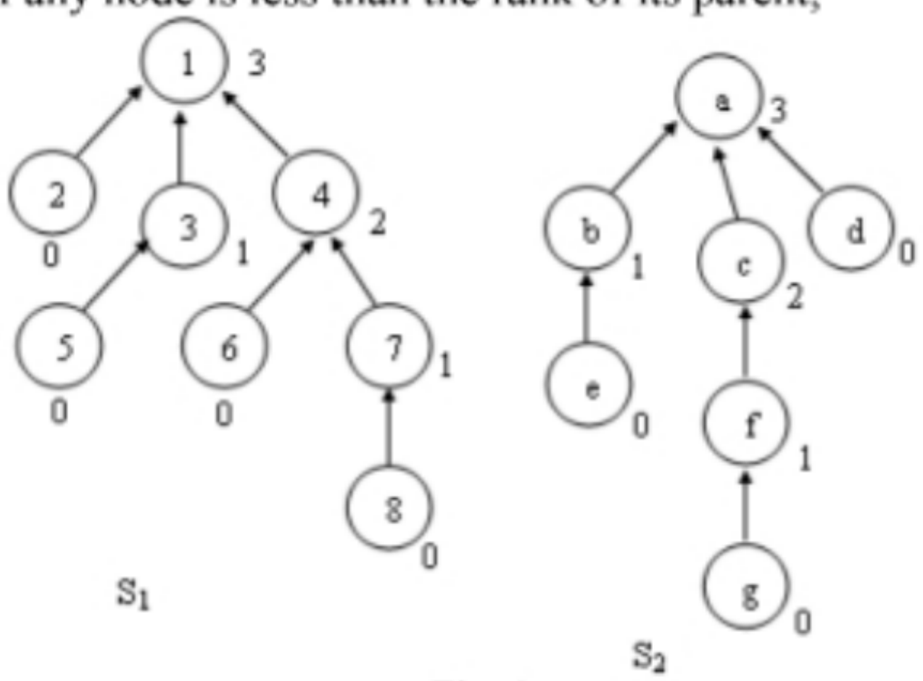


Fig: 1

We have shown the rank of above two sets S_1 and S_2 in the above figure 1. The number beside the circle denotes the rank at each level. Here we are starting the rank from lower level to higher level of the tree.
Now, the union of S_1 and S_2 create a new set X whose rank is given below in figure 2.

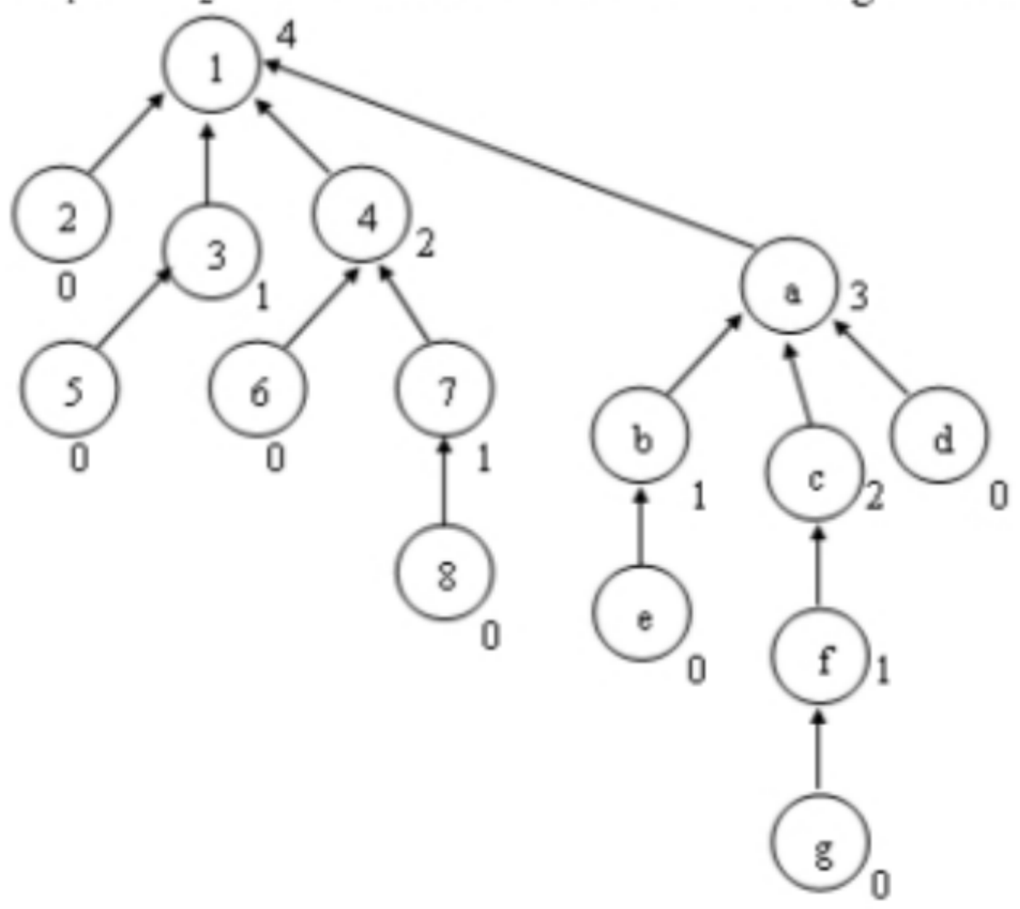


Fig: 2

There is another example of union by rank where two different set A and B have different ranks. The rank of set A is higher than the rank of set B. So after union by rank the root or the representative will be also the root of the set A∪B as shown in figures 3 and 4. In figure 4, the number appearing beside each node is the rank of that node.

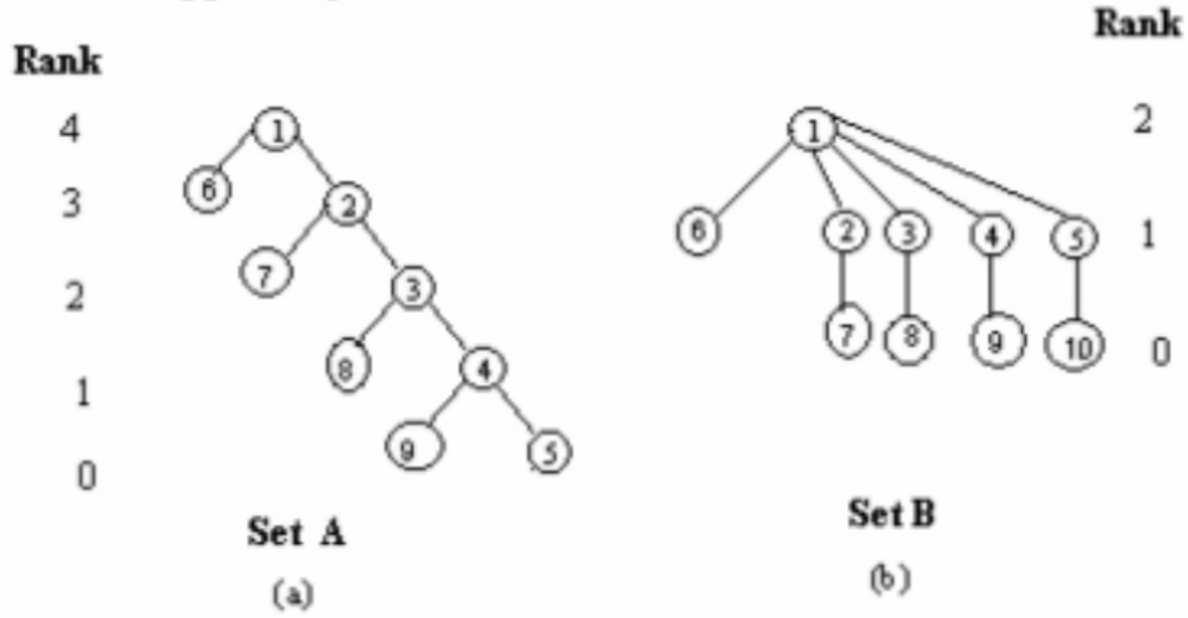


Fig: 3

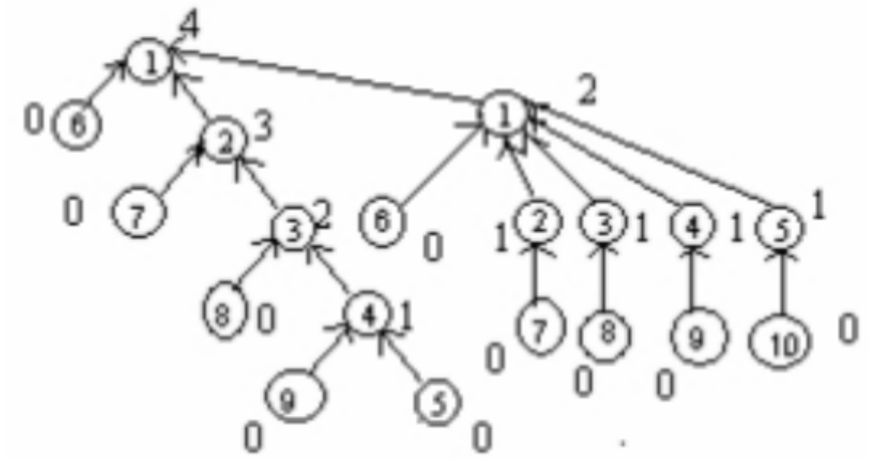


Fig: 4

13. Which type of knapsack problem cannot be solved using the greedy method? Explain your answer. **[WBUT 2013]**

Answer:

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property: At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem. If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming. But Greedy algorithms cannot always be applied. For example, Fractional Knapsack problem (See this) can be solved using Greedy, but 0-1 Knapsack cannot be solved using Greedy.

14. Find the recurrence relation of binary search and derive the time complexity of binary search. **[WBUT 2014]**

OR,

Find the best and worst case time complexity of binary search. **[WBUT 2017]**

Answer:

There is a recursive procedure here.

```
int search (vector v, int from, int to, int val)
{
if (from>to) return -1; //val not found
int mid = (from+to)/2;
if (v[mid] == val)
return mid;
else if (val > v[mid])
return search(v,mid+1,to,val);
else return search(v,from,mid-1,val);
}
```

Let's analyze the run time.

- It takes O(1) time to do the comparisons, then it cuts the search range in half.
  T(N) = T(N/2) + 1
- Repeat the recurrence...
  T(N)

= T(N/4) + 2
= T(N/8) + 3 ...
= T(N/2k) + k

- Round up N to nearest power of 2: $N \leq 2^m$.
  $T(N) \leq T(2^m/2^k)+k$
- Let k = m.
  $T(N) \leq T(2^m/2^m)+m = T(1)+m = 1+m = O(m)$
- If $N=2^m$, then m=log N.
  So T(N) = O(log N)

15. Write an algorithm for n-queen's problem. Find its time complexity.
[WBUT 2015]

Answer:

Algorithm for n-queens problem

```
        Nqueen (k, n)
                {
Step 1              for i← to n do
                  {
Step 1.1            if Place (k, i) then
                   {
Step 1.1.1          x[k] ←i
                   }
Step 1.2            if k ←n then
Step 1.2.1          write (x [1: n]);
Step 1.3            else
Step 1.3.1           Nqueen (k+1, n)
                  }
                }
```

The algorithm Place() is used to place a queen in k-th row and i-th column.

```
                Placc (k, i)
                {
Step 1                 for j← to k–1 do
                  {
Step 1.1             if ((x [j]= = i )or ( |(x[j]–i) |= = |j–k|)) then
                     {
Step 1.1.1            return false;
                     }
                  }
Step 2                 return true;
                }
```

In the above algorithm we can find the appropriate positions of each queen in an n × n chessboard using backtracking by satisfying the condition stated in equation (1). In Place(k, i), we find the proper position of each queen. The variable k and i denote the position of a queen at k-th row and i-th column. In Step 1.1, the "if" condition has two parts:

1. if (x[j] = = i) then this means that the two queens are in same column.
2. if (|(x[j]–i) = = |j–k|) means the absolute values of (x[j]–i) and (j–k) are same. This corresponds to the placement of two queens on the same diagonal. If one of the above two conditions is satisfied, then we cannot place k-th queen at row k of column i.

This function is used in Nqueens(k, n) algorithm. Here, the value of n denotes the index of the queen. From this algorithm, in Step 1.1, if the return value of Place (k, i) is true then we can place a queen at i-th column. From Step 1.2, it is evident that if there is no other queen left, then print the array x[1:n] to show the position of each queen finally. Otherwise we call the recursive function Nqueen (k+1, n), in Step 1.3.1 for next queen k+1.

The efficiency of both the backtracking algorithms we've just seen depends very much on four factors: (1) the time to generate the next x_k, (2) the number of x_k satisfying the explicit constraints, (3) the time for the bounding functions B_k, and (4) the number of x_k satisfying the B_k.

However, by allowing only placements of queens on distinct rows and columns, the time require is O(n!).

16. Which one is better and why? **[WBUT 2017]**

i) Max Min calculation using divide & conquer approach.

ii) Max Min calculation using normal approach.

Answer:

Let us consider the Max-Min problem which can be solved by both the straight forward and the divide-and-conquer technique. The problem is to find the maximum and minimum items in a set of n elements.

In analyzing the time complexity of this algorithm, we once again concentrate on the number of element comparisons. More importantly, when the elements in $a[1:n]$ are polynomials, vectors, very large numbers or strings of characters, the cost of an element comparison is much higher than the cost of the other operations. Hence the time is determined mainly by the total cost of the element comparisons.

Algorithm **StraightMaxMin(*a, n, max, min*)**

//Set max to the maximum and min to the minimum of *a[1:n]*.

```
{
        max := min := a[1]
        for i=2 to n do
        {
                if (a[i] > max) then max= a[i]
                if (a[i]<min) then min=a[i]
        }
}
```

StraightMaxMin() requires *2(n-1)* element comparisons in the best, average and worst cases. An immediate improvement is possible by realizing that the comparison $a[i] < min$ is necessary only when $a[i] > max$ is false.

Now the best case occurs when the elements are in increasing order. The number of element comparisons is *(n-1)*. The worst case occurs when the elements are in decreasing order. In this case the number of element comparisons is 2(n-1). The average number of element comparison is less than 2(n-1).

The algorithm MaxMin() is a recursive algorithm that finds the maximum and minimum of the set of elements {a(i), a(i+1),…,a(j)}. The situation of set sizes one (i=j) and two (i=j-1) are handled separately. For set containing more than two elements, the midpoint is determined, the two maxima and minima of these subproblems are generated. The two maxima are compared and the two minima are compared to achieve the solution for the entire set.

Algorithm MaxMin$(i,\ j,\ \max,\ \min)$

// $a[1:n]$ is a global array. Parameters i and j are integers,

// $1 \le i \le j \le n$. The effect is to set max and min to the

// largest and smallest values in $a[i:j]$, respectively.

```
{
    if (i = j) then max := min := a[i]; // Small(P)
    else if (i = j − 1) then // Another case of Small(P)
        {
            if (a[i] < a[j]) then
            {
                max := a[j]; min := a[i];
            }
            else
            {
                max := a[j]; min := a[i];
            }
    }
    else
    {   // If P is not small, divide P into subproblems.
        // Find where to split the set.
            mid := ⌊(i + j) / 2⌋;
        // Solve the subproblems.
            MaxMin(i, mid, max, min);
          MaxMin(mid+1,j,max1,min1);
        // Combine the solutions.
            if (max < max1) then max := max1;
            if (min > min1) then min := min1;
    }
```

}

Now, if T(n) represents the require time to execute Algorithm MaxMin(), then

$$T(n)=\begin{cases} T(\lfloor n/2 \rfloor)+T(\lceil n/2 \rceil)+2, & n>2 \\ 1, & n=2 \\ 0, & n=1 \end{cases}$$

When n is a power of two, $n=2^k$ for some positive integer k, then

$T(n) = 2T(n!2) + 2$

$= 2(2T(nl4) + 2) + 2$

$= 4\ T(nl4) + 4 + 2$

.

.

.

$= 2^{k-1}\ T(2) + \sum_{1<i<k-1} 2^i$

$= 2^{k-1} + 2^k - 2 = 3\ n/2 - 2$

Note that $(3n/2 - 2)$ is the best, average and worst case number of comparisons when n is a power of 2

17. Derive the worst case time complexity of quick sort. **[WBUT 2018]**

Answer:

Refer to Question No. 5 of Short Answer Type Questions.

18. Write an algorithm to insert an element into a heap. What is the complexity of the algorithm? Justify. **[WBUT 2018]**

Answer:

Refer to Question No. 12 of Long Answer Type Questions.

19. Write an algorithm for Graph Coloring Problem. What is the time complexity of the algorithm. **[WBUT 2018]**

Answer:

1st Part: ***Refer to Question No. 3(ii) of Long Answer Type Questions.***

2nd Part:

O(V^2 + E) in worst case. The above algorithm doesn't always use minimum number of colors. Also, the number of colors used sometime depend on the order in which vertices are processed.

20. Explain: Brute Force search. **[MODEL QUESTION]**

Answer:

Brute Force Algorithms refers to a programming style that does not include any shortcuts to improve performance, but instead relies on sheer computing power to try all possibilities until the solution to a problem is found. A classic example is the traveling salesman problem (TSP).

Suppose a salesman needs to visit 10 cities across the country. How does one determine the order in which cities should be visited such that the total distance travels is minimized. The brute force solution is simply to calculate the total distance for every possible route and then select the shortest one. This is not particularly efficient because it is possible to eliminate many possible routes through clever algorithms. Another example: 5 digit password, in the worst case scenario would take 10^5 tries to crack.
The time complexity of brute force is **O(n*m)** . So, if we were to search for a string of 'n' characters in a string of 'm' characters using brute force, it would take us n * m tries.

Long Answer Type Questions

1. a) Explain the basic concept of a divide-and-conquer algorithm. **[WBUT 2004, 2007]**

OR,

What do you mean by Divide and Conquer Strategy? **[WBUT 2018, 2019]**

Answer:

Divide and conquer (**D&C**) is an important algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.
In divide –and-conquer technique, the set of numbers are stored in an array. The numbers may or may not arrange in sorted order. To make these numbers in sorted order, we first split the array into two parts. In merge sort and binary search the divided arrays are equal but in quick sort it may or may not be equal. Then we further split the arrays into another two parts each. In this way, we split the initial array into small parts until all the elements of that small array are sorted (in sorting technique) or until we get the particular element in that small array (in searching technique). In sorting technique, we further merge the small array and get the initial array with sorted elements.

Example:

Merge sort Algorithm

b) Prove that the average case time-complexity of quick sort is O(n log n). You should state clearly the reasons behind the design of the recurrence relation you use for establishing this complexity. **[WBUT 2004, 2007]**

Answer:

We assume that each of the sizes of the left partitions are equally likely, and hence each has probability 1/n.

- With this assumption, the average value of T(i), and hence also of T(n–i–1), is $(T(0) + T(1) + \ldots + T(n–1))/n$
- Naturally, our recurrence relation becomes
  $T(n) = 2(T(0) + T(1) + \ldots + T(n–1))/n + cn$
- Multiplying both sides by n we find $nT(n) = 2(T(0) + T(1) + \ldots + T(n–1)) + cn^2$
- Substitution of *n* by *n* – 1 gives

$(n-1)T(n-1) = 2(T(0) + T(1) + \ldots + T(n-2)) + c(n-1)^2$

- Subtracting the last equation from the previous one, we get $nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c$
- Rearranging and ignoring constant c, we arrive at $nT(n) = (n+1)T(n-1) + 2cn$
- Division throughout by $n(n+1)$ gives, $T(n)/(n+1) = T(n-1)/n + 2c/(n+1)$
- Hence, $T(n-1)/n = T(n-2)/(n-1) + 2c/n$
  Similarly $T(2)/3 = T(1)/2 + 2c/3$
- Thus, $T(n)/(n+1) = T(1)/2 + 2c(1/3 + 1/4 + \ldots + 1/(n+1))$
- The sum in brackets is about $\log_e(n+1) + \gamma - 3/2$, where γ is Euler's constant, which is approximately 0.577.
- So, $T(n)/(n+1) = O(\log_2 n)$ and thus $T(n) = O(n \log_2 n)$

2. a) What is dynamic programming? [WBUT 2004, 2005, 2007, 2010]

Answer:

Dynamic programming is a method for reducing the runtime of algorithms exhibiting the properties of overlapping sub problems and optimal substructure. Dynamic Programming is an approach developed to solve sequential, or multi-stage, decision problems; hence, the name "dynamic" programming. But, as we shall see, this approach is equally applicable for decision problems where sequential property is induced solely for computational convenience.

Dynamic programming is both a mathematical optimization method and a computer programming method. In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner.

Algorithms which can be solved by Dynamic programming are,

Matrix-chain multiplication,
All pair shortest paths,
Single source shortest path,
Travelling Salesman problem.

b) Consider the evaluation of the product of n matrices: [WBUT 2004, 2013, 2016]
$M = M_1 * M_2 * \ldots\ldots * M_n$
Assuming that the multiplication of a p*q matrix by a q*r matrix require pqr scalar multiplication; write a dynamic programming algorithm for ordering this multiplication with minimum cost. Explain the algorithm in brief.

Answer:

Refer to Question No. 7 of Short Answer Type Questions.

3. Write an algorithm for *any one* of the following problems:
i) Eight queens problem. [WBUT 2004, 2009, 2013]
ii) Graph coloring problem [WBUT 2004, 2008, 2013, 2014, 2016]

Answer:

i) Eight queens problem:

Now, we discuss the general algorithm for 8-queens problem

```
Eightqueen (k, 8)
{
```

```
Step 1      for i← 1 to 8 do
{
Step 1.1 if Place (k, i) then
{
Step 1.1.1 x[k] ←i
}
Step 1.2 if k ←8 then
Step 1.2.1 write (x [1: 8]);
Step 1.3 else
Step 1.3.1 Eightqueen (k+1, 8)
}
}
```

The algorithm **Place()** is used to place a queen in k-th row and i-th column.

Place (k, i)

```
{
Step 1 for j← to k−1 do
{
Step 1.1if ((x [j]= = i ) or ( |(x[j]−i) |= = |j−k|)) then
{
Step 1.1.1 return false;
}
}
Step 2 return true;
}
```

In the above algorithm we can find the appropriate positions of each queen in an 8 × 8 chessboard using backtracking. In **Place(k, i),** we find the proper position of each queen. The variable k and i denote the position of a queen at k-th row and i-th column. In Step 1.1, the "if" condition has two parts:

3. if (x[j] = = i) then this means that the two queens are in same column.
4. if (|(x[j]–i) = = |j–k|) means the absolute values of (x[j]–i) and (j–k) are same. This corresponds to the placement of two queens on the same diagonal. If one of the above two conditions is satisfied, then we cannot place k-th queen at row k of column i.

This function is used in **Eightqueens(k, 8)** algorithm. Here, the value of 8 denotes the index of the queen. From this algorithm, in Step 1.1, if the return value of **Place (k, i)** is true then we can place a queen at i-th column. From Step 1.2, it is evident that if there is no other queen left, then print the array x[1:8] to show the position of each queen finally. Otherwise we call the recursive function **Eightqueens (k+1, 8)**, in Step 1.3.1 for next queen k+1.

ii) Graph coloring problem:

Applying backtracking method we are trying to color all the nodes of a graph in such a way that two adjacent nodes do not have same color.

Suppose, an undirected graph G = (V, E) is having n number of nodes. Now, we can represent an adjacency matrix C[n:n] such that,

C[i, j] = 1, if there is an edge in between node i and node j.
= 0, otherwise

If at most k colors are required to color all the vertices and they are labeled as c_1, c_2, $c_3 \ldots c_k$. So, a solution of graph coloring problem can be typically represented by n-tuples $(x_1, x_2, x_3 \ldots, x_n)$, where x_i is the color of node i and $1 \leq x_i \leq k$ for all i, $1 \leq i \leq n$.
The graph G = (V, E) has n number of vertices, i.e. |V| = n and there are k number of colors in an array A[1:k]

```
Graph_coloring(c)
{
 for i ←1 to n do
 {
      Nextcolor(c);      // insert a new color
      if (c> k) then     // there are no new color
      {
        print (no color left);
        return false;
      }
      else if (c==n) then // all colors are used
        print (A[1,n]); // print the color with respective vertex
      else
        Graph_coloring (c+1); // otherwise search next color
  }
return c;
}

Nextcolor(c)
{
  while (!False)
  {
    A[c] ←(A[c]+1) mod (k+1);  // find next Color
    if A[c] ←0 then            // all colors are used.
      return 0;
      for i ←1 to n do    // check for all vertices
      {
       if (G[c, i] ≠ 0) and (A[c] = A[i]) then
// if there is an edge between c and i but they are
// of same color.
            break;
       else
            return c;
      }
      if (i=n+1) then          //there is no new color
      return 0;
   }
}
```

The least number of color needed to color a graph is called its **chromatic number**. The **chromatic number**, say X of a complete graph G is X(G) = d+1, where d is the degree of the vertex. So if we consider a graph G is a complete graph, i.e., each node is connected

to all other nodes of the graph then total number of colors required to color all the nodes of the graph is d+1.e.g.,

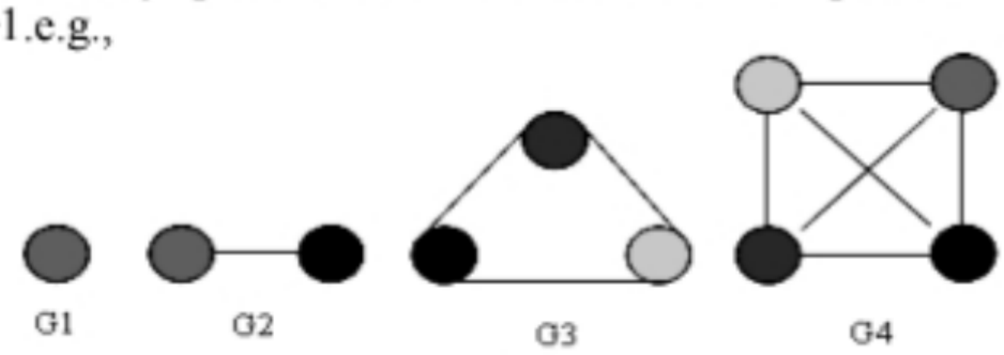


Fig: 1

In the above figure 1 each graph like G1, G2, G3 or G4 requires (d+1) colors to color all the nodes, where d is the degree of each node.

4. Discuss activity selection problem for job sequencing. [WBUT 2004, 2006, 2008]

OR,

Discuss the activity selection problem for job sequencing with an example. Prove that the time complexity of the algorithm is $O(n \log n)$. [WBUT 2012]

OR,

Discuss Job Sequencing with Deadlines with an example. [WBUT 2016, 2018]

Answer:

1st part:

Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed activities that wish to use a resource, which can be used by only one activity at a time. Each activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap (i.e., a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$). The activity-selection problem is to select a maximum-size subset of mutually compatible activities. For example, consider the following set S of activities, which we have sorted in monotonically increasing order of finish time.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

For this example, the subset $\{a3, a9, a11\}$ consists of mutually compatible activities. It is not a maximal subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger. In fact, $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities; another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

We only consider one choice, the greedy method and that when we make the greedy method, one of the subproblems is guaranteed to be empty, so that only one nonempty subproblem remains. Based on these observations, we shall develop a recursive greedy algorithm to solve the activity-scheduling problem. We shall complete the process of developing a greedy solution by converting the recursive algorithm to an iterative one.

2nd Part:

Probably wrong question.

Because, in job sequencing with deadline procedure, we are applying greedy approach to find the optimal solution of the problem. Now, to find the optimal solution, our objective is to choose the next job in such a way that it would be optimized. Let us consider the objective function $\sum_{i \in J} p_i$ and we have to optimize this function. So, to enter next job into the queue subject to the constraint that the resulting solution J must be a feasible solution, we must assure that it will increase the objective function $\sum_{i \in J} p_i$ maximum. Now, if we arrange the jobs in decreasing order according to their profits p_i, then we can assign the jobs sequentially to the job queue without violating the deadlines, i.e. the solution must be a feasible solution. This is the activity selection problem of jobs sequencing.

Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed activities that wish to use a resource, which can be used by only one activity at a time. Each activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i]$. Activities a_i and a_j are compatible if the intervals $[s_i, f_i]$ and $[s_j, f_j]$ do not overlap (i.e., a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$). The activity-selection problem is to select a maximum-size subset of mutually compatible activities. For example, consider the following set S of activities, which we have sorted in monotonically increasing order of finish time.

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
$\mathbf{s_i}$	1	3	0	5	3	5	6	8	8	2	12
$\mathbf{f_i}$	4	5	6	7	8	9	10	11	12	13	14

For this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. It is not a maximal subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger. In fact, $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities; another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

We only consider one choice, the greedy method and that when we make the greedy method, one of the sub-problems is guaranteed to be empty, so that only one nonempty sub-problem remains. Based on these observations, we shall develop a recursive greedy algorithm to solve the activity-scheduling problem. We shall complete the process of developing a greedy solution by converting the recursive algorithm to an iterative one. We still have to discuss the running time of the algorithm. The initial sorting can be done in time O(n log n), and the first loop takes time O(n). It is not hard to implement each body of the second loop in time O(n), so the total loop takes time $O(n^2)$. So the total algorithm runs in time $O(n^2)$.

5. a) Find the optimal solution using greedy criteria for a knapsack having capacity 100 kg for the following list of items having values and weights as shown in the table. **[WBUT 2007, 2012, 2013, 2019]**

Item	Value	Weight
I_1	10	15
I_2	20	25
I_3	30	35
I_4	40	45
I_5	50	55

OR,

Given weight vector (15, 25, 35, 45, 55) and the profit vector (10, 20, 30, 40, 50) and a Knapsack of capacity 100, find at least three feasible solutions including optimal one for the Knapsack problem of 5 times. **[WBUT 2014]**

Answer:

To find the optimal solutions we keep those elements whose a_i/z_i is maximum into the knapsack, where a_i represents the profit for i-th element and z_i represents the weight of the i-th element.

Now,

$$\frac{a_1}{z_1}=\frac{10}{15}=.66,\quad \frac{a_2}{z_2}=\frac{20}{25}=.8,\quad \frac{a_3}{z_3}=\frac{30}{35}=.85,\frac{a_4}{z_4}=\frac{40}{45}=.88,\frac{a_5}{z_5}=\frac{50}{55}=.90$$

So, the highest a_i/ z_i ratio is $\frac{a_5}{z_5}$ i.e. .9

So, we keep one unit of this element in the knapsack.

So, $x_5 = 1$ and $z_5x_5 = 55 \times 1 = 55$, $a_5x_5 = 50\times1=50$ and $M_1 = M - z_5x_5 = 100–55 = 45$

Now, next highest value of a_i/z_i is a_4/z_4

If $x_4 = 1$ then $z_4x_4 = 45\times1 = 45$ and $a_4x_4=40\times1=40$

So, $M_2 = M_1 - z_4x_4 = 45 – 45 = 0$

So, the knapsack is full and then maximum profit is

$= a_5x_5 + a_4x_4$

$= 50+40 =90$

Similarly, we may consider that the highest profitable item gives the highest profit. Then we insert the highest profitable item to the knapsack first and then next highest item and so on.

So, $x_5 > x_4 > x_3 > x_2 > x_1$.

First we insert the 5-th element into the knapsack because it is the highest profitable element.

So, $x_5 = 1$ and $z_5x_5 = 55 \times 1 = 55$, $a_5x_5 = 50\times1=50$ and $M_1 = M - z_5x_5 = 100–55 = 45$

Next, we insert the 4-th element into the knapsack because it is the next highest profitable element.

So, $x_4 = 1$ and $z_4x_4 = 45\times1 = 45$ and $a_4x_4=40\times1=40$ and $M_2 = M_1 - z_4x_4 = 45 – 45 = 0$

So, the knapsack is full and then maximum profit is

$= a_5x_5 + a_4x_4$

$=50+40 =90$

Similarly, we may consider that the lowest weight item gives the highest profit because, we can insert more weight to the knapsack. Then we insert the lowest weight item to the knapsack first and then next highest item and so on.
So, $x_1 > x_2 > x_3 > x_4 > x_5$.
First we insert the x_1 element into the knapsack because it is the lowest weight element.
So, $x_1 = 1$ and $z_1x_1 = 15 \times 1 = 15$, $a_1x_1 = 10 \times 1 = 10$ and $M_1 = M - z_1x_1 = 100 - 15 = 85$
Next, we insert the x_2 element into the knapsack because it is the next lowest weight element.
So, $x_2 = 1$ and $z_2x_2 = 25 \times 1 = 25$ and $a_2x_2 = 20 \times 1 = 20$ and $M_2 = M_1 - z_2x_2 = 85 - 25 = 60$
Next, we insert the x_3 element into the knapsack because it is the next lowest weight element.
So, $x_3 = 1$ and $z_3x_3 = 35 \times 1 = 35$ and $a_3x_3 = 30 \times 1 = 30$ and $M_3 = M_2 - z_3x_3 = 60 - 35 = 25$
Next, we insert the x_4 element into the knapsack because it is the next lowest weight element.
So, $x_4 = 25 / 45 = 0.55$ and $z_4x_4 = 45 \times 25 / 45 = 25$ and $a_3x_3 = 40 \times 0.55 = 22$ and $M_4 = M_3 - z_3x_3 = 25 - 25 = 0$
So, the knapsack is full and then maximum profit is
$= a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4$
$= 10+20+30+22 = 82$

b) What is the difference between dynamic programming and greedy method?
[WBUT 2007, 2008, 2012, 2017, 2019]

Answer:
Greedy method is an algorithm that always takes the best immediate, or local, solution while finding an answer. Greedy algorithms find the overall, or globally, optimal solution for some optimization problems, but may find less-than-optimal solutions for some instances of other problems.
So the difference between Greedy method and Dynamic Programming are,

- Both techniques are optimization techniques, and both build solutions from a collection of choices of individual elements.
- The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.
- Dynamic programming computes its solution bottom up by synthesizing them from smaller subsolutions, and by trying many possibilities and choices before it arrives at the optimal set of choices.
- There is no a priori test by which one can tell if the Greedy method will lead to an optimal solution.
- By contrast, there is a prior test for Dynamic Programming, called The Principle of Optimality(A problem is said to satisfy the Principle of Optimality if the subsolutions of an optimal solution of the problem are themselves optimal solutions for their subproblems.), which can say if Dynamic Programming will lead to an optimal solution.

6. a) Find the minimum number of operations required for the following matrix chain multiplication using dynamic programming:

$A(10\times20)*B(20\times50)*C(50\times1)*D(1\times100)$ **[WBUT 2008, 2013, 2015]**

Answer:

We can represent A_1 , A_2 , A_3 , A_4 in such a way, that,

$A_1 \rightarrow p_0 \times p_1$

$A_2 \rightarrow p_1 \times p_2$

$A_3 \rightarrow p_2 \times p_3$

$A_4 \rightarrow p_3 \times p_4$

Where, $p_0 = 10$, $p_1 = 20$, $p_2 = 50$, $p_3 = 1$, $p_4 = 100$

Now,

$m[1, 1] = 0$

$m[2, 2] = 0$

$m[3, 3] = 0$

$m[4, 4] = 0$

This will happen when we parenthesize in the following order

$(A_1), (A_2), (A_3), (A_4)$.

Next, we can find out, that

$m[1, 2] = m[1,1] + m[2, 2] + p_0 \cdot p_1 \cdot p_2$

$= 0 + 0 + 10.20.50$

$= 10000$

i.e. the parenthesis is (A_1A_2)

$m[2, 3] = m[2,2] + m[3, 3] + p_1 \cdot p_2 \cdot p_3$

$= 0 + 0 + 20.50.1$

$= 1000$

i.e. the parenthesis is $(A_2 A_3)$.

$m[3, 4] = m[3,3] + m[4, 4] + p_2 \cdot p_3 \cdot p_4$

$= 0 + 0 + 50.1.100$

$= 5000$

i.e. the parenthesis is (A_3A_4).

Now, we can calculate the following

$$m[1, 3] = \min \begin{cases} m[1,1] + m[2, 3] + p_0 \cdot p_1 \cdot p_3 \\ m[1,2] + m[3, 3] + p_0 \cdot p_2 \cdot p_3 \end{cases}$$

$$= \min \begin{cases} 0 + 1000 + 10.20.1 \\ 10000 + 0 + 10.50.1 \end{cases}$$

$$= \min \begin{cases} 1200 \\ 10500 \end{cases}$$

$= 1200$

Now, we can parenthesize $A_1A_2A_3$ either $((A_1A_2)(A_3))$ or $((A_1)(A_2A_3))$. But the minimum scalar multiplication will occur for $((A_1)(A_2A_3))$.

$$m[2,4] = \min \begin{cases} m[2,2] + m[3,4] + p_1 \cdot p_2 \cdot p_4 \\ m[2,3] + m[4,4] + p_1 \cdot p_3 \cdot p_4 \end{cases}$$

$$= \min \begin{cases} 0 + 5000 + 20.50.100 \\ 1000 + 0 + 20.1.100 \end{cases}$$

$$= \min \begin{cases} 105000 \\ 3000 \end{cases}$$

$$= 3000$$

Now, we can parenthesis $A_2A_3A_4$ either $((A_2A_3)(A_4))$ or $((A_2)(A_3A_4))$. But the minimum scalar multiplication will occur for $((A_2A_3)(A_4))$.

Now, we get the final parenthesis.

$$m[1,4] = \min \begin{cases} m[1,1] + m[2,4] + p_0 \cdot p_1 \cdot p_4 \\ m[1,2] + m[3,4] + p_0 \cdot p_2 \cdot p_4 \\ m[1,3] + m[4,4] + p_0 \cdot p_3 \cdot p_4 \end{cases}$$

$$= \min \begin{cases} 0 + 3000 + 10.20.100 \\ 10000 + 5000 + 10.50.100 \\ 1200 + 0 + 10.1.100 \end{cases}$$

$$= \min \begin{cases} 23000 \\ 65000 \\ 2200 \end{cases}$$

$$= 2200$$

So, The minimum scalar multiplication will occur for $((A_1A_2\ A_3)(A_4))$ and the total number of scalar multiplication are 2200.

b) Give an algorithm for matrix-chain multiplication. [WBUT 2008, 2009, 2010]

Answer:

Refer to Question No. 7 of Short Answer Type Questions.

7. Design a backtracking algorithm to find all the Hamiltonian cycles in a Hamiltonian graph. What is the worst-case time complexity of the algorithm? [WBUT 2008, 2013]

Answer:

The following algorithms find Hamiltonian cycles from a graph by applying backtracking method.

Search_node(k)

```
{
Step 1 while ((x[k] + 1) mod (n+1)!= 0);
{
Step 1.1 if (G[ x[k-1], x[k]] ≠ 0) then
{
Step 1.1.1 for j ← 1 to k-1 do
```

```
{
Step 1.1.1.1 if  (x[j] ← x[k]) then
Step 1.1.1.2 return 0 ;
}
Step 1.1.2  if ( j← k) then
{
Step 1.1.2.1  if ((k<n) or ((k=n) and G[ x[n], x[1] ≠ 0) then
Step 1.1.2.2          return k;
}
Step 2 else
Step 2.1 return ( " no path");
}
}
}
```

Hamiltonian(k)

```
{
Step 1 while( Search_node( k) != 0)
{
Step 1.1 if ( x[k]← n )then
Step 1.1.1 return ( x[1: n]);
Step 1.2 else
Step 1.2.1 Hamiltonian(k+1);
}
}
```

Complexity of Hamiltonian cycle algorithm

The time taken to scan n vertices is O(n) and O(n) time to extend all forced degree two paths. Since the iterations terminate unless a new vertex of degree two is created, at most n iterations can occur. At most O(m) edges can be deleted.

(O(m)) and the next branch is taken. Thus, an easy upper bound on the pruning time for a node searching from a vertex of degree d is $O(d(n^2 + m))$, but this is overly pessimistic.

Note that along any branch from the root of the search tree to a leaf, at most n vertices can be converted to degree 2. Also note that along each branch each edge can be deleted at most once. If the degree is high we seldom take more than a few branches before success.

The implementation is such that when several vertices have two neighbors of degree two at the beginning of an iteration, all redundant edges are removed in a single pass taking time proportional to n plus the number of edges removed and checked. In practice, on a graph G, it typically takes O(n + m) time per search node on simple Hamiltonian instances.

8. Find the optimal solution for the fractional Knapsack problem given below:

$I = \{I_1, I_2, I_3, I_4, I_5\}$

$w = \{5, 10, 20, 30, 40\}$

$v = \{30, 20, 100, 90, 160\}$

The knapsack capacity, W = 60 **[WBUT 2008]**

Answer:

Here v_i is the profit and w_i is the weight of the each knapsack elements.

So, according to the Greedy Knapsack problem,

$v_1 / w_1 = 30/5 = 6$

$v_2 / w_2 = 20/10 = 2$

$v_3 / w_3 = 100/20 = 5$

$v_4 / w_4 = 90/30 = 3$

$v_5 / w_5 = 160/40 = 4$

We consider that some fraction x_i, $0 \le x_i \le 1$ of the object i is kept into the knapsack.

The highest ratio is $v_1 / w_1 = 6$ and insert one unit of $w_1 = 5$

So, $x_1 = 1$, $x_1 . w_1 = 5$, $x_1 . v_1 = 30$ and $W = W - x_1 . w_1 = 60 - 5 = 55$

Next highest ratio is $v_3 / w_3 = 5$ and insert one unit of $w_3 = 20$

So, $x_3 = 1$, $x_3 . w_3 = 20$, $x_3 . v_3 = 100$ and $W = W - x_3 . w_3 = 55 - 20 = 35$

Next highest ratio is $v_5 / w_5 = 4$ and insert fractional part of w_5.

So, $x_5 = 35 / 40 = 7 / 8$. So, $x_5 = 7/8$, $x_5 . w_5 = \frac{7}{8} * 40 = 35$, $x_5 . v_5 = \frac{7}{8} * 160 = 140$ and

$W = W - x_5 . w_5 = 35 - 35 = 0$

So, the knapsack is full and then maximum profit is

$= x_1 . v_1 + x_3 . v_3 + x_5 . v_5 = 30 + 100 + 140 = 270$

9. a) State the general knapsack problem. Write a greedy algorithm for this problem and derive its time complexity. **[WBUT 2009, 2016]**

Answer:

The Knapsack problem states that we have to fill a knapsack having constant weight, with some objects having different weights and different profit values. We have to fill up the knapsack with objects in such a way that the total weight of selected objects does not cross the limit of the knapsack and we get the maximum profit.

Suppose,

The capacity of the knapsack is C.

There are n many objects. An individual object is represented by I where i=1,2,…n.

Weight of i-th object is z_i , $1 \le I \le n$.

Profit for i-th object is a_i , $1 \le I \le n$

We can put maximum one unit of each object in the knapsack or some fractional part of that object. So, we consider that some fraction x_i, $0 \le x_i \le 1$ of the object I is kept into the knapsack.

Now, the weight of the knapsack is $\sum_{i-1}^{n} z_i x_i$ and profit is $A = \sum_{i=1}^{n} a_i x_i$. So we can define this problem as:

Maximize $A = \sum_{i=1}^{n} a_i . x_i$ … (1)

Subject to $\sum_{j-1}^{n} z_i x_i \leq C$ … (2)

And $0 \leq x_i \leq 1, 0 \leq I \leq n$

Algorithm of Knapsack problem

To generate an algorithm to solve knapsack problem, first we consider that all the values of a_i / z_i of the i-th element are arranged in decreasing order. Two arrays a[1:n] and z[1:n] are used for maintaining profit and weight of n objects in such a way that a[i] / z[i] ≥ a[i+1] / z[i+1] respectively. M is the knapsack size and x[1:n] is the solution vector.

Knapsack (M, n)

```
{
Step 1for I ← 1 to n do
{
Step 1.1x[i] ← 0.0;
}
Step 2 u ← M;
Step 3 for  I ← 1 to  n do
{
Step 3.1 if z[i] > u then
Step 3.1.1 break;
Step 3.2 else
Step 3.2.1 x [i] ← 1.0;
Step 3.2.2.  u ← u - z[i],
}
Step 4 if    I ≤ n  then
Step 4.1 x[i] ← u/z[i];
Step 5 else
{
Step 5:1 for  I ← 1 to n do
Step 5.1.1 print ← x[i];
}
}
```

Time complexity of knapsack problem is O(nW) [W is the maximum weight of the knapsack].

b) Given the weight vector (2, 3, 5, 7, 1, 4, 1) and the profit vector (10, 5, 15, 7, 6, 18, 3) and a knapsack of capacity 15, find at least three feasible solutions including optimal one for the knapsack problem of seven objects. [WBUT 2009, 2016]

Answer:

To find the optimal solutions we apply third method among the above, i.e., keep those elements whose a_i/z_i is maximum into the knapsack.

[Here a=value and z = weight]

Now, $\frac{a_1}{z_1}=\frac{10}{2}=5$, $\frac{a_2}{z_2}=\frac{5}{3}=1.67$, $\frac{a_3}{z_3}=\frac{15}{5}=3$

$\frac{a_4}{z_4}=\frac{7}{7}=1$, $\frac{a_5}{z_5}=\frac{6}{1}=6$, $\frac{a_6}{z_6}=\frac{18}{4}=4.5$, $\frac{a_7}{z_7}=\frac{3}{1}=3$

So, the highest a_i / z_i ratio is $\frac{a_5}{z_5}$ i.e. 6

So, we keep one unit of this element in the knapsack.

So, $x_5 = 1$ and $z_5x_5 = 1 \times 1 = 1$, $a_5x_5 = 6\times1=6$ and $M_1 = M - z_5x_5 = 15–1 = 14$

Now, next highest value of a_i/z_i is a_1/z_1

If $x_1 = 1$ then $z_1x_1 = 2\times1 = 2$ and $a_1x_1=10\times1=10$

So, $M_2 = M_1 - z_1x_1 = 14 – 2 = 12$

Next highest value of a_i/z_i is a_6/z_6

If $x_6 = 1$ then $z_6x_6 = 4\times1=4$ and $a_6x_6 = 18\times1 = 18$ and

$M_3 = M_2 - z_6x_6 = 12 – 4 = 8$

The next highest value of a_i / z_i is a_3 / z_3 or a_7 / z_7 both are same. So we can choose any one. Suppose we choose a_7/z_7 for knapsack.

If $x_7 = 1$ then $z_7 .x_7 = 1\times1 = 1$ and $a_7x_7 = 3\times1 = 3$ and

$M_4 = M_3 - z_7x_7 = 8 – 1 = 7$

The next highest value is a_3 / z_3

If $x_3 = 1$ then $z_3x_3 = 5\times1 = 5$ and $a_3x_3 = 15\times1 = 15$ and $M_5 = M_4 - z_3x_3 = 7–5 = 2$.

The next highest value is a_2/z_2

If $x_2 = 1$ then $z_2x_2 = 3\times1 = 3$ which exceeds the value of knapsack. So, we have to put the fractional part of x_2 to the knapsack, i.e., remaining part of knapsack is 2.

So, $x_2 = \frac{2}{3}$ Now, $z_2x_2 = 3\times\frac{2}{3}=2$, $a_2x_2 = 5\times\frac{2}{3}=3.33$ and $M_6 = M_5 - z_2x_2 = 2 – 2 = 0$

So, the knapsack is full and then maximum profit is

$= a_5x_5 + a_1x_1 + a_6x_6 + a_7x_7 + a_3x_3 + a_2x_2 = 6+10+18+3+15+3.33 = 55.33$

10. Apply backtracking technique to solve the 3-colouring problem for the following graph. **[WBUT 2009, 2011, 2016]**

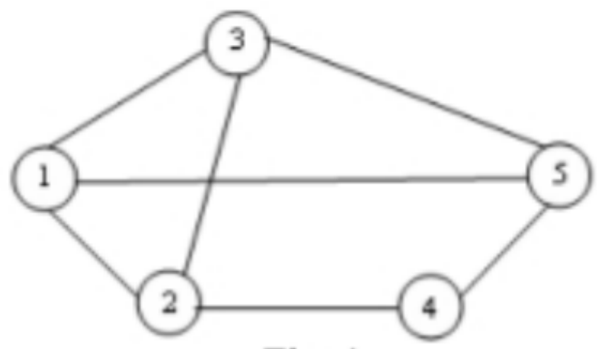


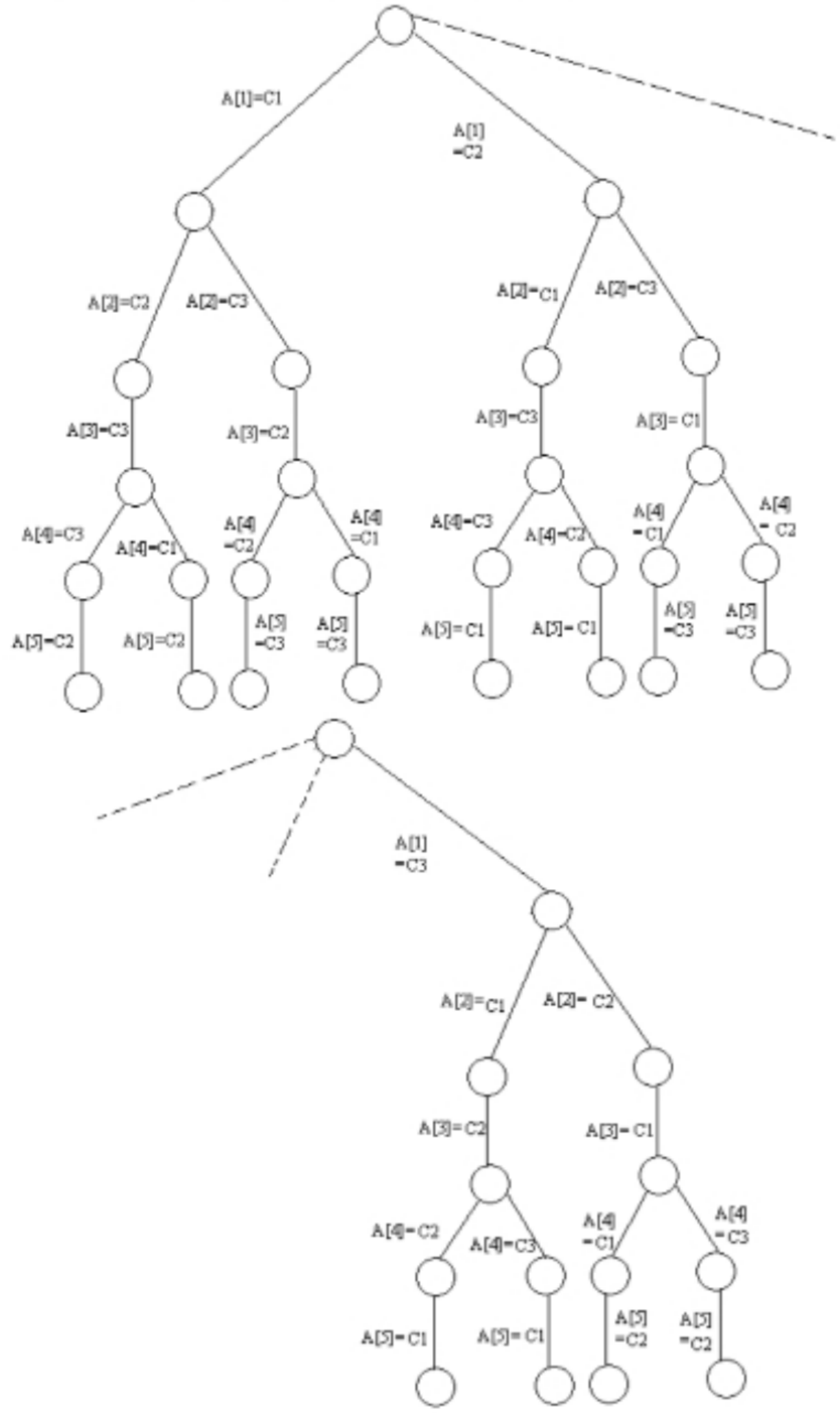
Fig: 1

Answer:

In the graph of figure 1, the number of vertices is n=5 and suppose we have to color it using three colors i.e., k=3

Let us denote three colors by c1, c2, c3.

Suppose all colors of five nodes 1, 2, 3, 4, 5 store in an array A[]. The index of each array element shows the vertex number and content of each array element denotes the respective color of the node. So the index of A[i] i.e., I represents the vertex of graph G. We start the assigning color to the first node of the graph. Applying backtracking method we can use different color for different nodes. So, we get



11. Write the algorithm of Quick sort. Find the best case, worst case and average case time complexities of this algorithm. **[WBUT 2009, 2013, 2014]**

Answer:

The following procedures together form the quick sort algorithm.

```
QuickSort(A, p, r)
{
  if p < r then
  {
    q ← Partition (A, p, r);
    Quick Sort (A, p, q);
    Quick Sort (A, q + 1, r);
  }
}
```

Algorithm for partition

Partition (A, p, r)

```
{
  initialize x ← A[p];
  i ← p-1, j ← r+1;
  while (i < j)
  {
    while (A[--j] > x)
    {
      while (A[++i] < x)
      {
        if (i < j) then
          Exchange (A[i], A[j]);
      }
    }
  }
 return j;
}
```

Exchange(A, I, j)

```
{
 p ← A[i];
 A[i] ← A[j];
 A[j] ← p;
}
```

Partition selects the first key x ← $A[p]$ as a pivot key about which the array is partitioned. If x ≤ A[p], $A[p]$ will move towards the left of the pivot, otherwise if x ≥ $A[p]$, $A[p]$ will move towards the right of the pivot key x.

Complexity Analysis

- ***Best case***
  - In the best case, the pivot is in the middle position of the array.
  - To simplify the equations, we assume that the two sub arrays are each exactly half the length of the original one. So, we get $T(n) = 2T(n/2) + cn$, c>0 constant (independent of n) and n ≥ 2 with T(1) = 1.

- This is very similar to the formula for Merge sort, and a similar analysis leads to $T(n) = cn \log_2 n + n$ which is $O(n \log_2 n)$.

- ***Average case:*** We assume that each of the sizes of the left partitions are equally likely, and hence each has probability 1/n.
  - With this assumption, the average value of T(i), and hence also of T(n–i–1), is (T(0) + T(1) + … + T(n–1))/n
  - Naturally, our recurrence relation becomes T(n) = 2(T(0) + T(1) + … + T(n–1))/n + cn
  - Multiplying both sides by n we find nT(n) = 2(T(0) + T(1) + … + T(n–1)) + cn^2
  - Substitution of n by n–1 gives (n–1)T(n–1) = 2(T(0) + T(1) + … + T(n–2)) + $c(n–1)^2$
  - Subtracting the last equation from the previous one, we get nT(n) – (n–1)T(n–1) = 2T(n–1) + 2cn – c
  - Rearranging and ignoring constant c, we arrive at nT(n) = (n+1)T(n–1) + 2cn
  - Division through out by n(n+1) gives T(n)/(n+1) = T(n–1)/n + 2c/(n+1)
  - Hence, T(n–1)/n = T(n–2)/(n–1) + 2c/n
    Similarly T(2)/3 = T(1)/2 + 2c/3
  - Thus T(n)/(n+1) = T(1)/2 + 2c(1/3 + ¼ + … + 1/(n+1))
  - The sum in brackets is about $\log_c(n+1) + \gamma - 3/2$, where γ is Euler's constant, which is approximately 0.577.
  - So, T(n)/(n+1) = $O(\log_2 n)$ and thus T(n) = $O(n \log_2 n)$

- ***Worst case:*** In quick sort technique, the worst case condition arises when the elements of the array are already sorted.
  - If the pivot is always the smallest element, then always $I = 0$
  - We ignore the term $T(0)=1$, so the recurrence relation is T(n) = T(n–1) + cn
  - So, T(n–1)=T(n–2)+c(n–1) and so on until we get T(2)=T(1)+c(2)
  - Substituting backwards, we get T(n) = T(1) + c(n + … + 2) = $O(n^2)$

It may be noted that this case happens if we always take the pivot to be the first element in the array and the array is already sorted.

12. What is Heap property? Write an algorithm to make a Heap containing elements. Then, show that how can you insert an element into a Heap. Then, write the algorithm of Heap sort and find the running time of this algorithm. Write an algorithm to find the existence of an element into a Heap. [WBUT 2010]

OR,

a) What is Heap property?
b) Write an algorithm of Heap Sort.
c) Find the running time of this algorithm. [WBUT 2018, 2019]

Answer:

Heap: A heap is a complete binary tree with the following properties:
1. If it is a max heap then the value of each node is greater than the value of its children.
2. If it is a min heap then the value of each node is less than the value of its children.

Algorithm to make a Heap

BUILD_HEAP () algorithm constructs a complete heap tree where each node of the tree is present at its proper position with the help of Heapify () algorithm. BUILD_HEAP () breaks the total length of the array into two equal parts. So, the elements in the sub arrays A[1, …$\lfloor n/2 \rfloor$] and A[$\lfloor n/2 \rfloor$+1 . . n] are all leaf nodes. Now it compares each node from length [A]/2 down to 1 with the help of Heapify (). The bottom-up order of processing guarantees that the sub tree rooted at each of the children are heap, before **Heapify()** is run at their parent node.

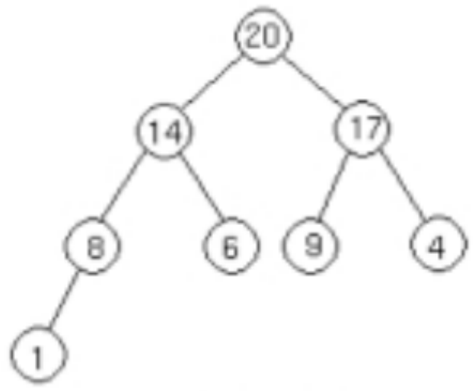
```
BUILD_HEAP (A, length)
{
  heap-size (A) ← length [A]
  for i ← ⌊length[A]/2⌋ down to 1 do
  Heapify (A, i) ;
}

Heapify (A, i)
{
   l ← left [i]
   r ← right [i]
  if l ≤ heap-size [A] and A[l] > A[i] then
     largest ← l
  else
    largest ← i
  if r ≤ heap-size [A] and A[r] > A[largest] then
    largest ← r
  if largest ≠ i then
        exchange A[i] ↔ A[largest]
  Heapify (A, largest)
}
```

This **Heapify ()** algorithm is totally based on the procedure to arrange a heap in a 1-D array which has already been discussed earlier. If the above rule is violated, then **Heapify()** algorithm set each node to its appropriate position.

Insert an element into the Heap

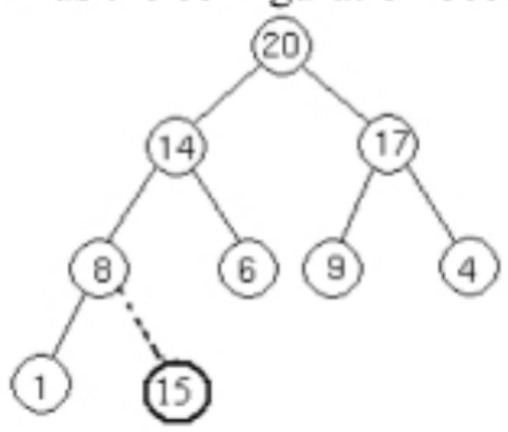
Suppose we have a heap as follows



Max heap

Fig: 1

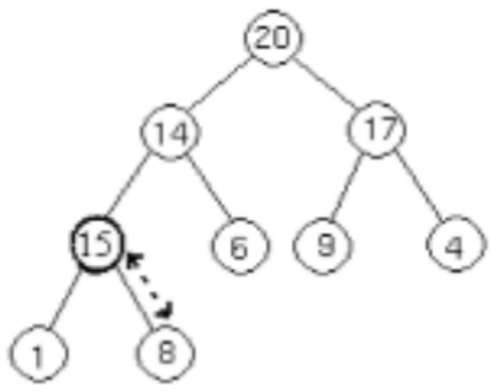
Suppose, we want to add a node with key 15 in to the heap in figure 1. First, we add the node to the tree at the next spot available at the lowest level of the tree. This is to ensure that the tree remains complete. Thus the configuration becomes,



Insert element 15 to the heap

Fig: 2

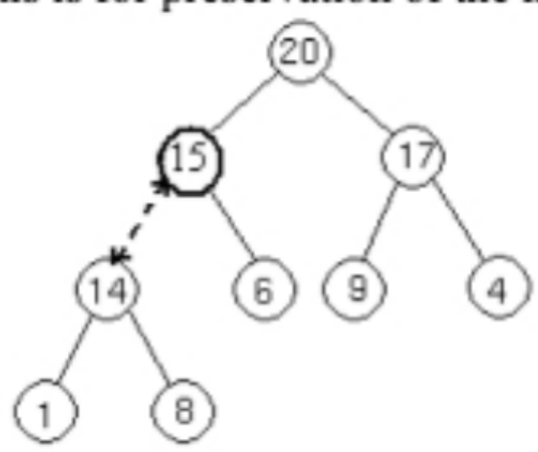
However, after inserting element 15, as shown in figure 2, we find that the heap property is not maintained any more. To ensure the heap property preserved, we have perform a few steps to convert it into a max heap. We interchange elements 15 and 8 as evident in figure 3.



Interchange between 15 and 8

Fig: 3

Next, another swap is required with respect to elements 14 and 15 in figure 3 to obtain a situation shown in figure 4. This is for preservation of the heap property.



Interchange between 15 and 14

Fig: 4

Now we have completed a heap tree and placed 15 in its appropriate position. No further move is required because $15 \leq 20$. But note that the left child having value 1, is less than right child having value 8, of the sub-tree rooted at the node containing value 14. In our subsequent discussion, we shall see how we can sort out this problem.

Heap Sort algorithm

The heap sort algorithm starts by using procedure BUILD-HEAP() to build a max heap on the input array A[1 . . n]. Since the maximum element of the array is stored at the root A[1], it can be put into its correct final position by exchanging it with A[n] (the last element in A). If we now discard node n from the heap then the remaining elements can be converted into heap. The new element at the root may violate the heap property. So it is necessary to restore the heap property further.

```
HEAPSORT (A)
{
  BUILD_HEAP (A);

  for i ← length (A) down to 2 do
  {
        exchange A[1] ↔ A[i]
        heap-size [A] ← heap-size [A] – 1
        Heapify (A, 1);
  }
}
```

Complexity

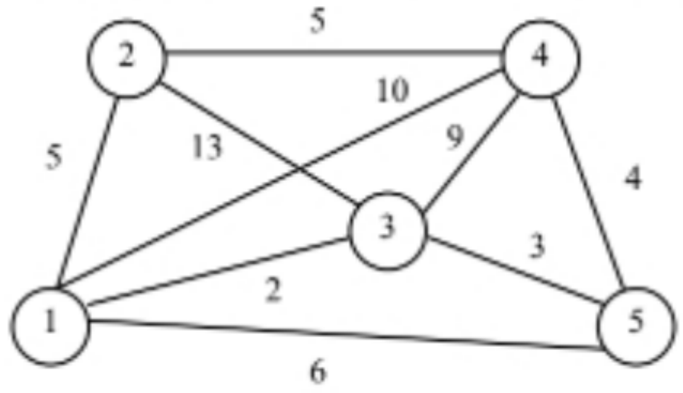
The HEAPSORT () procedure takes time at most O (n log $_2$ n). Since the call to BUILD_HEAP () takes time O (n) and each of the n -1 calls to Heapify () takes time O (log $_2$n) which we have already discussed earlier, so the worst case time complexity of heap sort is O (n log $_2$ n).

13. a) Explain the graph coloring problem and write the algorithm. [WBUT 2011]

Answer:

Refer to Question No. 3(ii) of Long Answer Type Questions.

b) Solve the single source shortest path problem for the following graph considering '1' as the source vertex using Dijkstra's algorithm. [WBUT 2011]



Answer:

Initially: Vertex 1 is the source vertex and we have to find out the shortest distance from vertex 1 to all the remaining vertices.

So, the distances from vertex 1 to all other vertices are given below.

$S = \{1\}, D[2] = 5, D[3] = 2, D[4] = 10 \; D[5] = 6$

Iteration 1: Now, vertex 1 to vertex 3 is the shortest distance and the distance is 3. Now, we have to find out the shortest distance of the other vertices.
Select w = 3, so that S = {1, 3}
$D[2] = \min(5, D[3] + C[3, 2]) = \min(1, 2+13) = 5$
$D[4] = \min(10, D[3] + C[3, 4]) = \min(1, 2+9) = 10$
$D[5] = \min(6, D[3] + C[3, 5]) = \min(6, 2+3) = 5$

Iteration 2: Select w = 5, so that S = {1, 3, 5}
$D[2] = \min(5, D[5] + C[5, 2]) = \min(5, 5+\infty) = 5$
$D[4] = \min(10, D[5] + C[5, 4]) = \min(10, 5+4) = 9$

Iteration 3: Select w = 4, so that S = {1, 3, 5, 4}
$D[2] = \min(5, D[4] + C[4, 2]) = \min(5, 9+5) = 5$

Iteration 4: Select w = 2, so that S = {1, 2}
$D[2] = 5$
So, the shortest distances from vertex 1 to other vertices are given below.
i) $D[2] = 5$. i.e. the shortest distance from vertex 1 to vertex 2 is 1 and the path is $1 \rightarrow 2$
ii) $D[3] = 2$ i.e. the shortest distance from vertex 1 to vertex 3 is 2 and the path is $1 \rightarrow 3$
iii) $D[4] = 9$ i.e. the shortest distance from vertex 1 to vertex 4 is 1 and the path is $1 \rightarrow 3 \rightarrow 5 \rightarrow 4$
iv) $D[5] = 5$ i.e. the shortest distance from vertex 1 to vertex 5 is 5 and the path is $1 \rightarrow 3 \rightarrow 5$

14. Solve the following Knapsack problem with the given conditions: $n=3$ weight of the Knapsack $M=20$, Profits $(p_1, p_2, p_3)=(25,24,15)$ and weight $(w_1, w_2, w_3)=(18,15,10)$. **[WBUT 2011, 2015]**

Answer:
To find the optimal solutions we apply third method among the above, i.e., keep those elements whose p_i/w_i is maximum into the knapsack.
[Here p_i = profit value and z_i = weight]

Now, $\frac{p_1}{w_1} = \frac{25}{18} = 1.38$, $\frac{p_2}{w_2} = \frac{24}{15} = 1.6$, $\frac{p_3}{w_3} = \frac{15}{10} = 1.5$

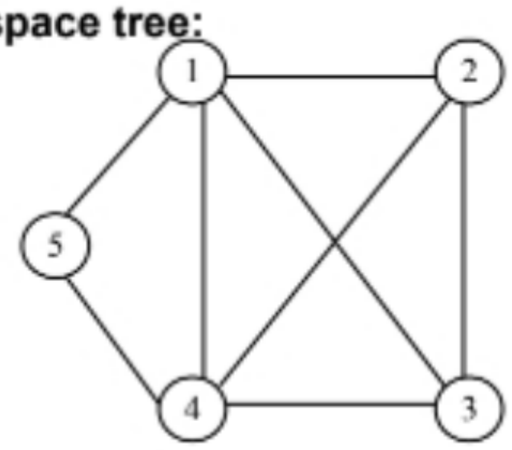
So, the highest p_i / w_i ratio is 1.6
So, we keep one unit of this element in the knapsack.
So, $x_2 = 1$ and $w_2 \cdot x_2 = 15 \times 1 = 1$, $p_2 x_2 = 24 \times 1 = 24$ and $M_1 = M - w_2 \cdot x_2 = 20\text{-}15 = 5$
Now, next highest value is 1.5
If $x_3 = 1$ then $w_3 x_3 = 10 \times 1 = 10$ which exceeds the weight of the knapsack. So, we have to put the fractional part of x_3 to the knapsack, i.e., remaining part of knapsack is 5.

So, $x_3 = \frac{5}{10} = 0.5$

Now, $w_3 x_3 = 10 \times \frac{5}{10} = 5$, $p_3 x_3 = 15 \times \frac{5}{10} = 7.5$ and $M_2 = M_1 - w_3 x_3 = 5 - 5 = 0$

So, the knapsack is full and then maximum profit is = $p_2 x_2 + p_3 x_3 = 24 + 7.5 = 33.5$

15. a) Apply backtracking technique to solve the following graph-coloring problem and also generate the state space tree: **[WBUT 2013]**



Answer:

Here the chromatic number is 4. i.e. 4 different colors are required to color the graph such that no two adjacency vertices are same color. In the figure 1, we have considered four different colors are c1, c2, c3, c4 and five vertices of the graph is represented by A[1], A[2], A[3], A[4] and A[5].

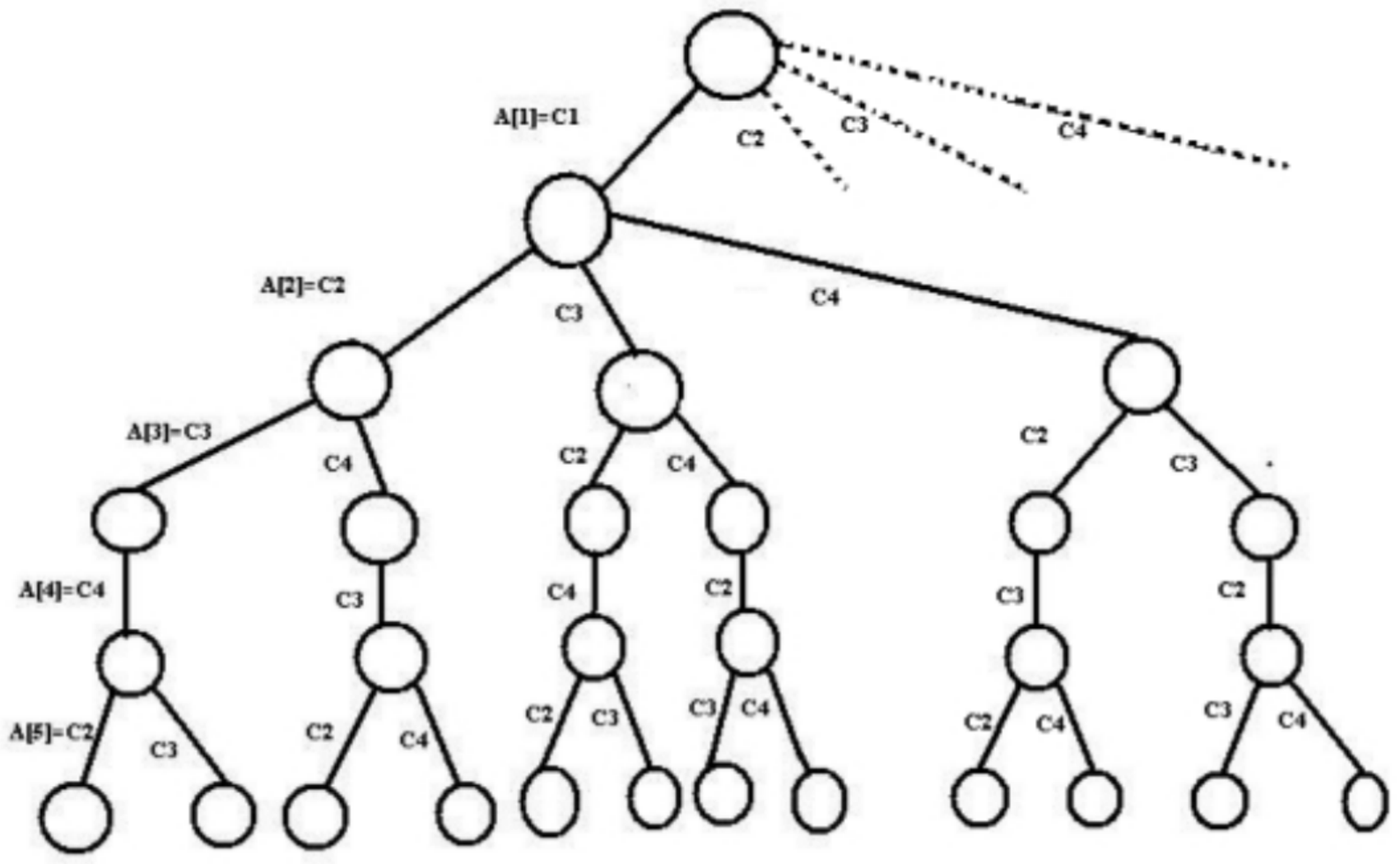


Fig 1: State space tree of the graph

[Try to draw the remaining three parts of the tree]

b) Write an algorithm for n-queen's problem. Find its time-complexity and explain the algorithm using an example. **[WBUT 2013]**

Answer:

1st Part: ***Refer to Question No. 15 of Short Answer Type Questions.***

2nd Part:

Our simple example is the n-queen problem, which happens to be NP-complete (which means computationally hard). On an n × n chessboard, a queen can move any number of squares up, down, to the right, to the left, and digonally. Fig: 1(a) illustrates a possible move for a queen for a case of n = 5. The problem is to determine n "safe" positions for n queens. A safe position means that none of the queens can move to a square occupied by another queen in only one move. Obvously there must be one and only one queen in each row and column. (If two or more queens are in a row or column, then one would be attacked by another, if no queens are in a row or coloumn, there must be a row or column that has more than one queen.) Similarly, there must be at most one queen in each diagonal direction (since there can be no queen in a diagonal direction). Fig: (b) is a solution for n = 5; generally, a solution is not unique for a specific value of n.

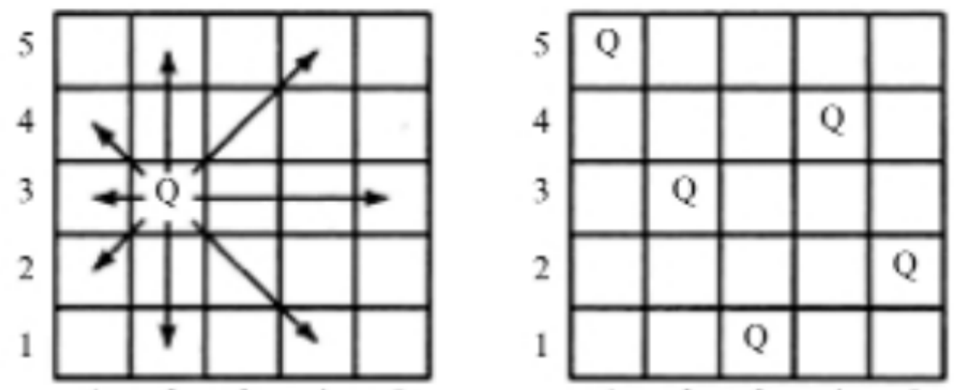
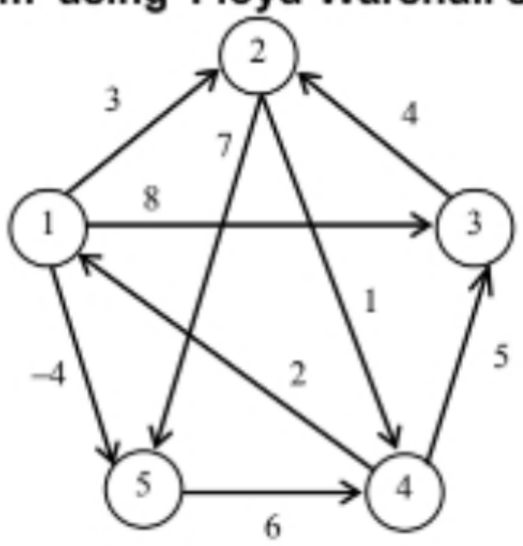


Fig: 1 The *n*-queen problem illustration for *n*=5.
(a) A possible move of a queen. (b) A solution

16. Solve the APSP problem using Floyd-Warshall's algorithm for the following graph: **[WBUT 2013]**



Answer:

The sequence of matrices $D^{(k)}$ and $\Pi^{(k)}$ computed by the Floyd-Warshall's algorithm for the graph is given below.

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & NIL & 4 & NIL & NIL \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

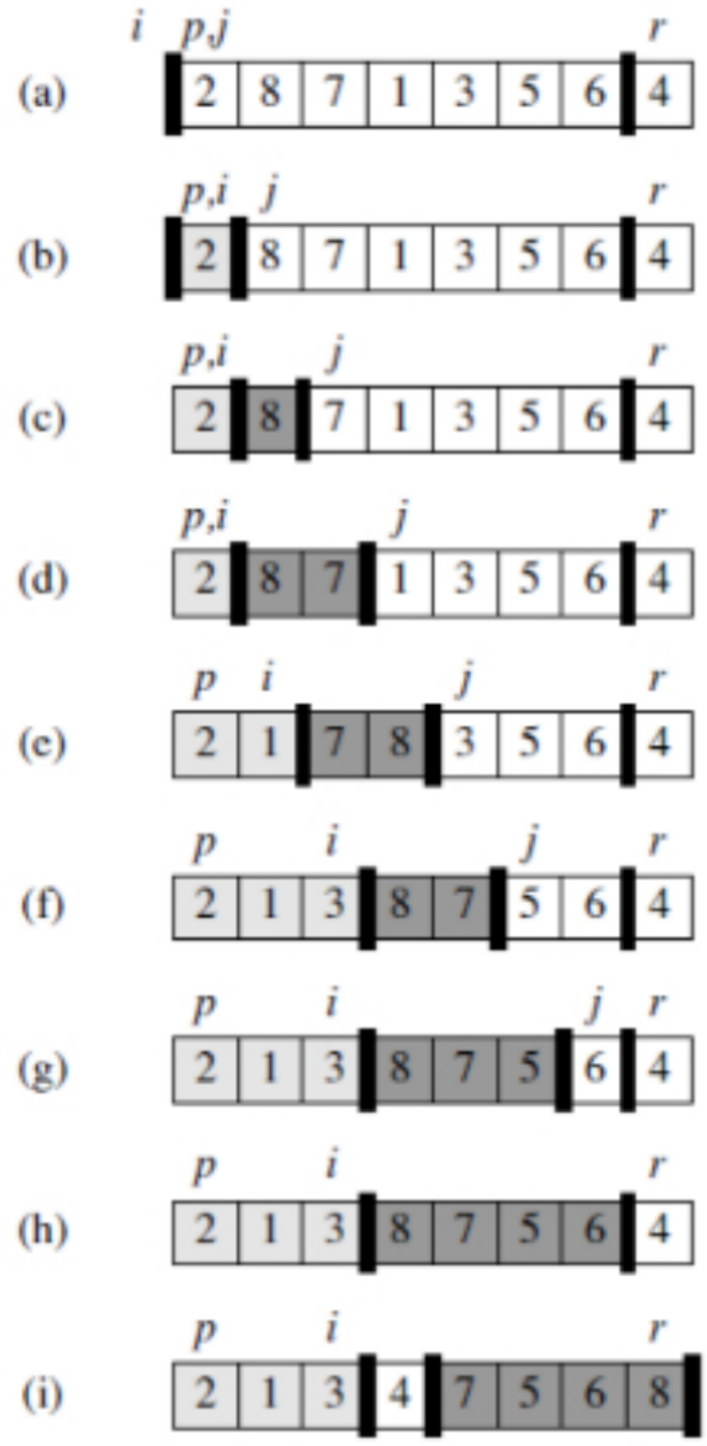
$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 3 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} NIL & 1 & 4 & 2 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

17. a) Perform the PARTITION operation once (one time) on the following array as per the requirement of the quicksort algorithm, assuming the last element of the array to be the pivot element. Clearly mention the steps.
arr[ ] = {2, 8, 7, 1, 3, 5, 6, 4} **[WBUT 2014]**

Answer:



The operation of PARTITION on a sample array. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The un-shaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot. (a) The initial array and variable settings. None of the elements have been placed in either of the first two partitions. (b) The value 2 is "swapped with itself" and put in the partition of smaller values. (c)–(d) The values 8 and 7 are added to the partition of larger values. (e) The values 1 and 8 are swapped, and the smaller partition grows. (f) The values 3 and 7 are swapped, and the smaller partition grows. (g)–(h) The larger partition grows to include 5 and 6 and the loop terminates. (i) In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

b) Using greedy strategy, schedule the following jobs within deadline so as to maximize the profit. Deadlines and profits are mentioned as follow: [WBUT 2014]

Job i	1	2	3	4
Deadline d_i	3	2	3	1
Profit g_i	9	7	7	2

Answer:

All the jobs are already arranged in the above matrix according to their profit.
Suppose, initially the solution vector is S={Φ}
Highest profitable job is J1. So, we add job J1 to the solution vector.
i.e. S={J1}.
Next, we add J2 to the solution vector. i.e. S={J1, J2}.
Next, we add J3 to the solution vector. i.e. S={J1, J2, J3}.
So, highest deadline is 3 and we can arrange maximum three jobs. The sequence of the jobs are S={J1, J2, J3} or S={J2, J1, J3} or S={J3, J2, J1} or S={J2, J3, J1} and the maximum profit is 23.

18. What is negative weight-cycle? Write Bellman-Ford algorithm to find single source shortest distance of a directed graph. [WBUT 2015]

Answer:

1st Part:

Some instances of the single-source shortest-paths problem may include edges whose weights are negative. If the graph $G=(V,E)$ contains no negative weight cycles reachable from the source s, then for all $v \in V$, the shortest-path weight $\delta(s,v)$ remains well defined, even if it has a negative value. If the graph contains a negative-weight cycle reachable from s, however, shortest-path weights are not well defined. No path from s to a vertex on the cycle can be a shortest path. If there is a negative weight cycle on some path from s to v we define $\delta(s,v) = -\infty$.

2nd Part: ***Refer to Question No. 11(a) of Short Answer Type Questions.***

19. a) Given the four matrices using $P_{10\times20}, Q_{20\times50}, R_{50\times1}, S_{1\times100}$. Find the optional sequence for the computation of multiplication operation. Write the algorithm also.
b) Consider the following table that consists of some items with weight and cost values:

Items	I_1	I_2	I_3	I_4	I_5
Weight	5	10	15	22	25
Cost	30	40	45	77	90

If the knapsack capacity W =60 kg, find optional solution using greedy criteria and write an algorithm for doing so. [WBUT 2017]

Answer:

a) ***Refer to Question No. 8 of Long Answer Type Questions.***

b) Here vi is the cost and wi is the weight of the each knapsack elements.
So, according to the Greedy Knapsack problem,
v1/ w1 = 30/5 = 6
v2/ w2 = 40/10 = 4
v3/ w3 = 45/15 = 3
v4/ w4 = 77/22 = 3.5
v5/ w5 = 90/25 = 3.6
The knapsack capacity W = 60 kg
We consider that some fraction xi, 0≤xi≤1 of the object i is kept into the knapsack.
The highest ratio is v1/ w1 =6 and insert one unit of w1=5
So, x1 = 1, x1 .w1 = 5, x1 . v1 =30 and W = W- x1 .w1 = 60-5=55
Next highest ratio is v2/ w2 = 4 and insert one unit of w2 = 10
So, x2 = 1, x2 .w2 = 10, x2 . v2 = 40 and W = W- x2 .w2 = 55-10=45
Next highest ratio is v5/ w5 = 3.6 and insert one unit of w5 = 25
So, x5 = 1, x5 .w5 = 25, x5 . v5 = 90 and W = W- x5 .w5 = 45-25=20
Next highest ratio is v4/ w4 = 3.5 and insert fractional part of w4.
So, x4 = 20 / 22 =10 / 11. So, x4 = 10/11, x4 .w4 =(10/11)*22 = 20,
x4 . v4 ={10/11)*77=70 and W = W- x4 .w4 = 20 – 20 =0
So, the knapsack is full and then maximum profit is
= x1 . v1 + x3 . v3 + x5 . v5 = 30+40+90+70= 230

20. a) Write an algorithm for Merge Sort. **[WBUT 2018, 2019]**

Answer:

The procedure MERGE-SORT (A, p, r) sorts the elements in the sub-array A[p . . r]. If p ≥ r, the sub-array has at most one element and it is already sorted. Otherwise, the divide step simply computes an index q that partitions A[p . . r] into two sub-arrays: A[p . . q], containing [n/2] elements, and A[q + 1 . . r], containing [n/2] elements.

```
MERGE-SORT(A, p, r)
{
   if (p < r) then
   {      q ← [(p + r)/2]
          MERGE-SORT(A, p, q)
          MERGE-SORT(A, q + 1, r)
          MERGE(A, p, q, r)
   }
}

MERGE(A, p, q, r)
{
   n1 ← q − p + 1
   n2 ← r − q
   create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
   for i ← 1 to n1 do
   {
```

```
    L[i] ← A[p + i − 1]
}
for j ← 1 to n2 do
{
    R[ j] ← A[q + j]
}
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r do
{
     if L[i] ≤ R[ j] then
     {
       A[k] ← L[i]
        i ← i + 1
      }
    else
       {
        A[k] ← R[ j]
        j ← j + 1
       }
    }
}
```

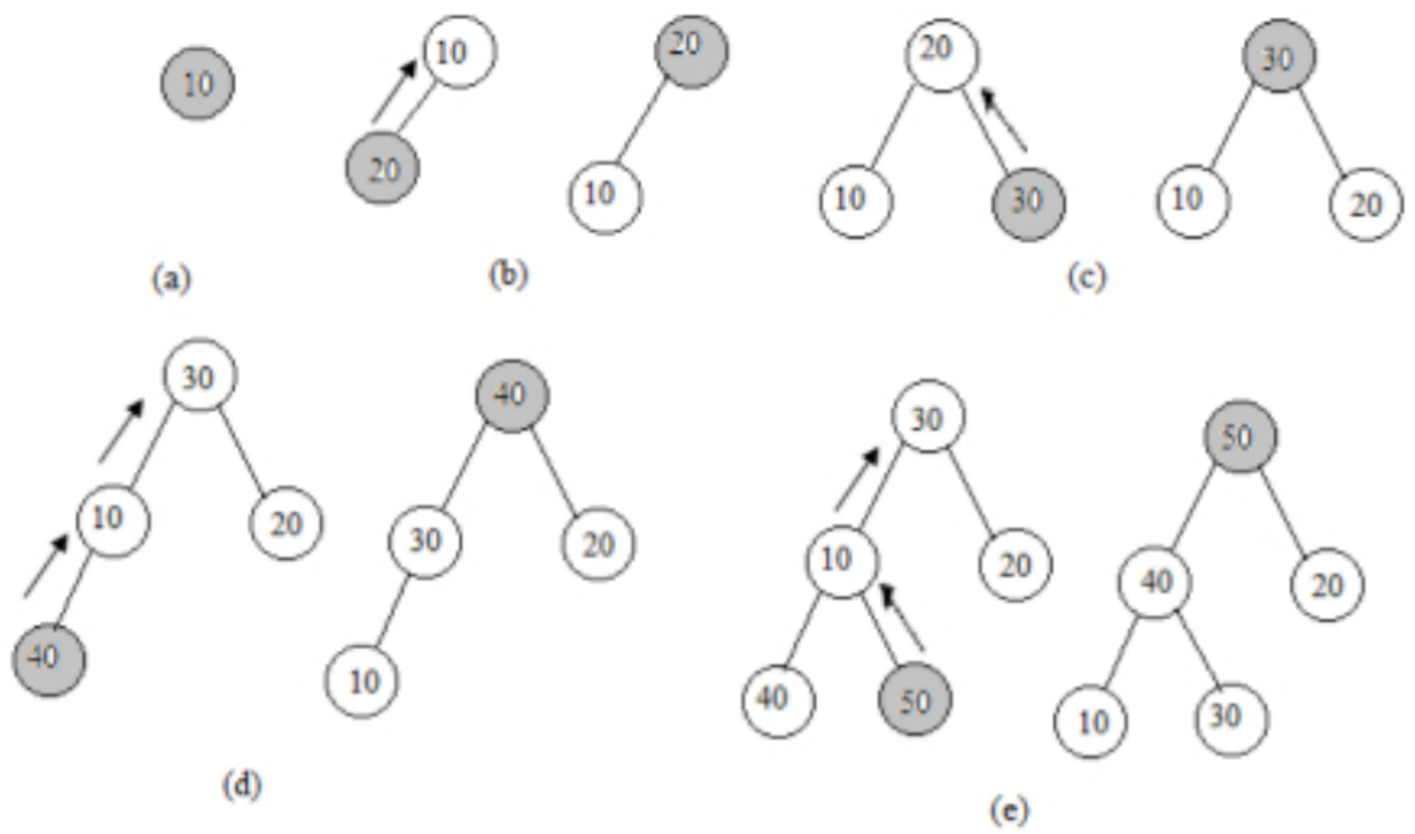
b) Create a Max-Heap containing the following elements: [WBUT 2018, 2019]

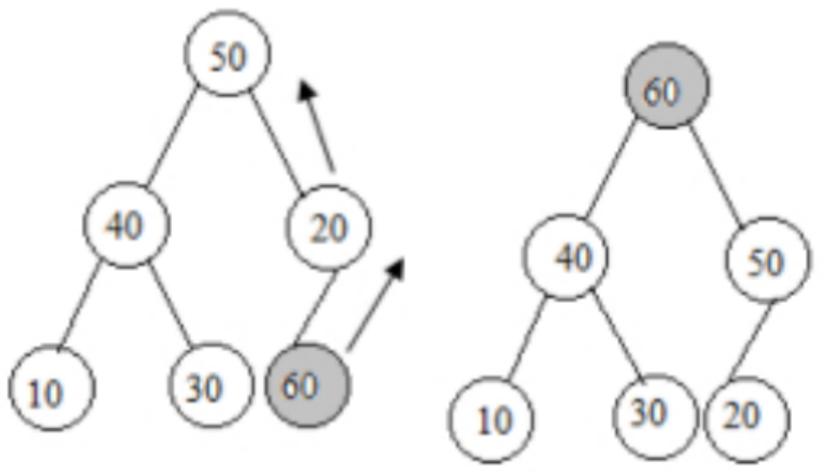
10, 20, 30, 40, 50, 60, 70, 80, 90, 100

Answer:

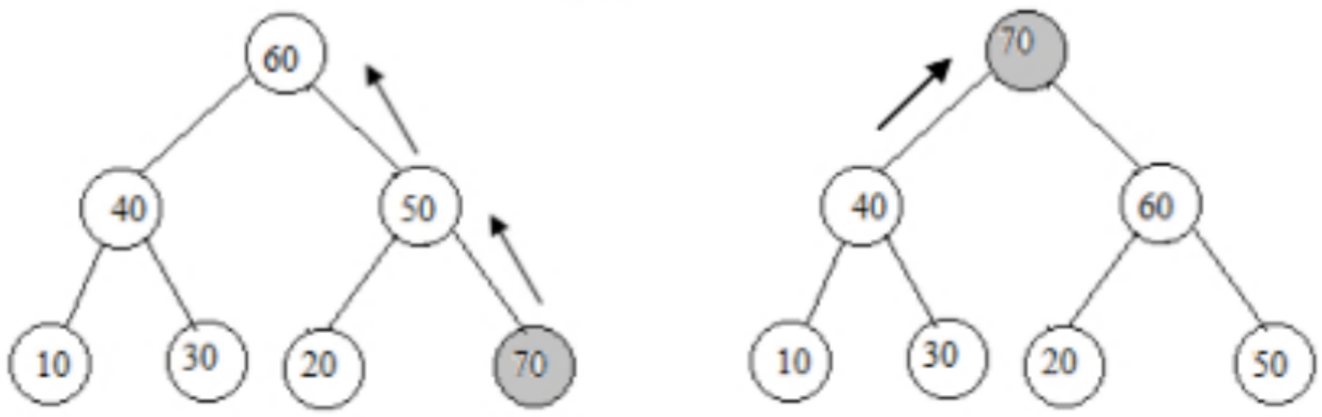
Max-Heap containing the following elements:

10, 20, 30, 40, 50, 60, 70, 80, 90, 100

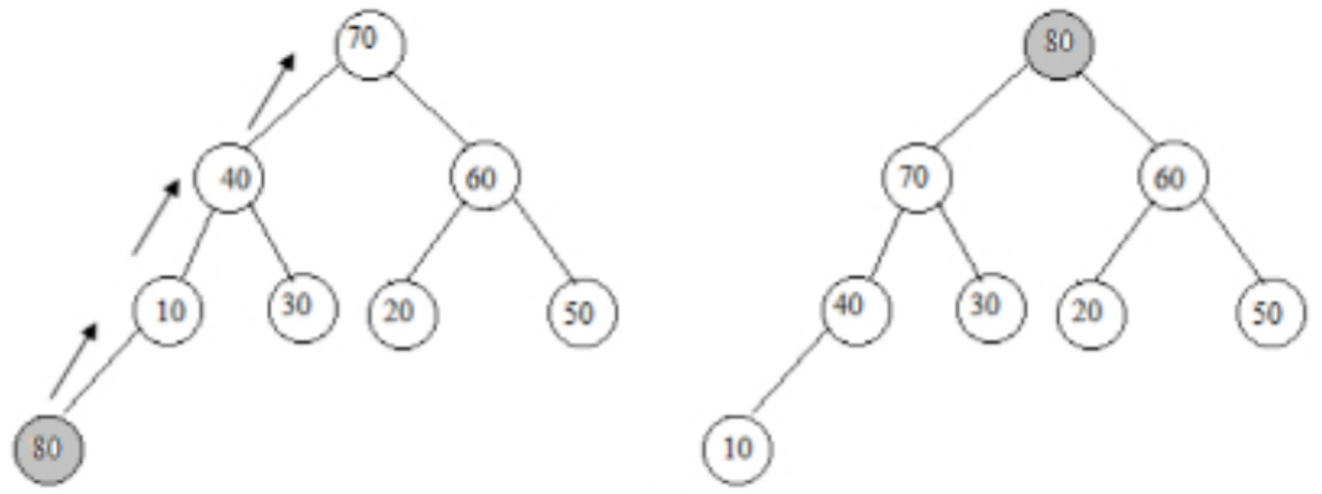



50
40
20
10
30
60
60
40
50
10
30
20

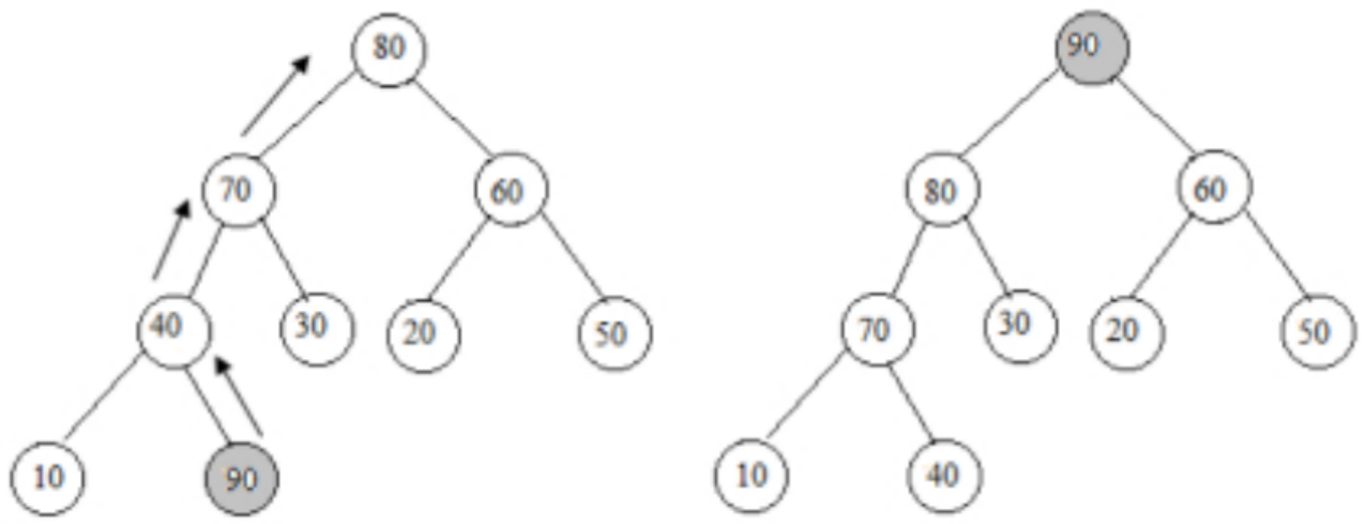
(f)


60
40
50
10
30
20
70
70
40
60
10
30
20
50

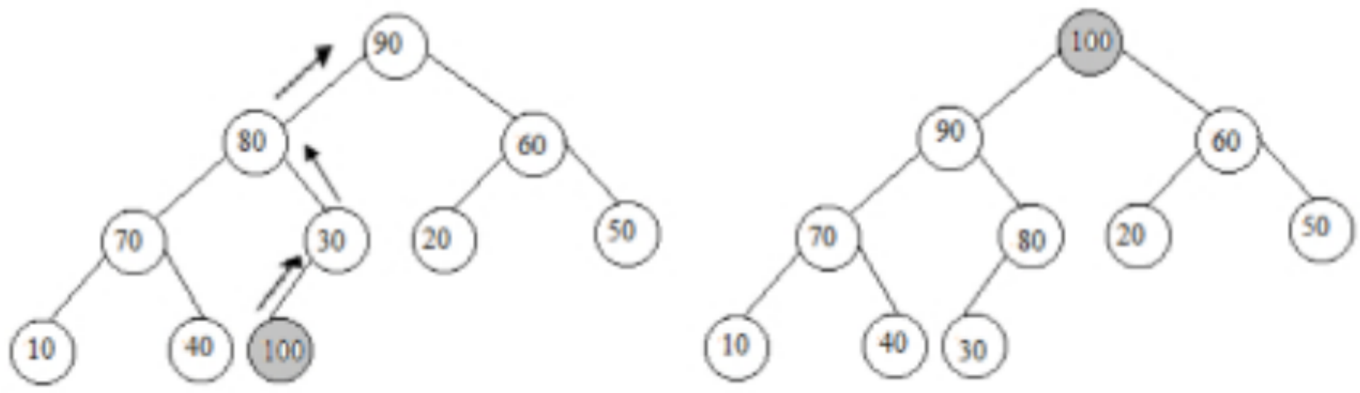
(g)


70
40
60
10
30
20
50
80
80
70
60
40
30
20
50
10

(h)


80
70
60
40
30
20
50
10
90
90
80
60
70
30
20
50
10
40

(i)



Max Heap

100	90	60	70	80	20	50	10	40	30

(j)

21. Write short notes on the following:

a) External sorting **[WBUT 2015]**
b) Heap Creation Technique **[WBUT 2016]**
c) 8 queens problem **[WBUT 2003, 2006, 2013]**
d) Hamiltonian cycle **[WBUT 2017]**
e) Heuristic Algorithm **[WBUT 2018]**
f) Bellman-Ford Algorithm **[WBUT 2018]**

Answer:

a) External sorting:

External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive).

Characteristics:

- Processing large files, unable to fit into the main memory
- Restrictions on the access, depending on the external storage medium Primary costs – for input-output
- Main concern: minimize the number of times each piece of data is moved between the external storage and the main memory.

General strategy - Sort-Merge

- Break the file into blocks about the size of the internal memory
- Sort these blocks
- Merge sorted blocks
- Usually several passes are needed, creating larger sorted blocks until the whole file is sorted

Basic Algorithm

Assumptions:
Four tapes:
Two for input - Ta1, Ta2,

Two for output - Tb1, Tb2.
Initially the file is on Ta1.
N records on Ta1
M records can fit in the memory

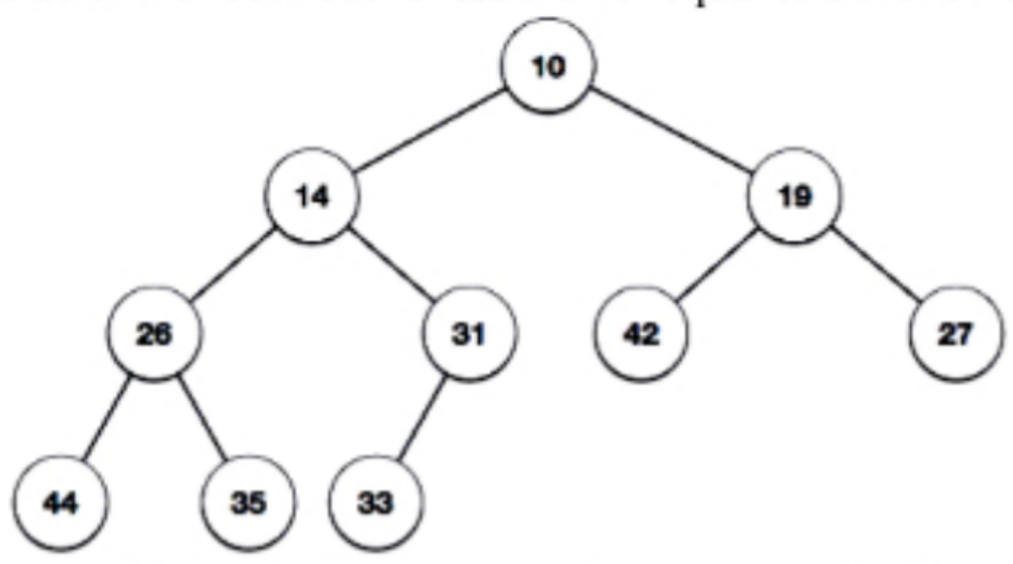
Step 1: Break the file into blocks of size M, [N/M]+1 blocks
Step 2: Sorting the blocks:
read a block, sort, store on Tb1
read a block, sort, store on Tb2,
read a block, sort, store on Tb1,
etc, alternatively writing on Tb1 and Tb2
Each sorted block is called a run.
Each output tape will contain half of the runs
Step 3: Merge:
a. From Tb1, Tb2 to Ta1, Ta2.
Merge the first run on Tb1 and the first run on Tb2, and store the result on Ta1:
Read two records in main memory, compare, and store the smaller on Ta1
Read the next record (from Tb1 or Tb2 - the tape that contained the record stored on Ta1) compare, store on Ta1, etc.
Merge the second run on Tb1 and the second run on Tb2, store the result on Ta2.
Merge the third run on Tb1 and the third run on Tb2, store the result on Ta1.
Etc, storing the result alternatively on Ta1 and Ta2.
Now Ta1 and Ta2 will contain sorted runs twice the size of the previous runs on Tb1 and Tb2
b. From Ta1, Ta2 to Tb1, Tb2.
Merge the first run on Ta1 and the first run on Ta2, and store the result on Tb1.
Merge the second run on Ta1 and the second run on Ta2, store the result on Tb2
Etc, merge and store alternatively on Ta1 and Ta2.
c. Repeat the process until only one run is obtained. This would be the sorted file

Analysis of two-way merge
The algorithm requires [log(N/M)] passes plus the initial run-constructing pass.
Each pass merges runs of length r to obtain runs of length 2*r.
The first runs are of length M. The last run would be of length N.
Let's assume that N is a multiple of M.
Initial situation:
1st tape contains N records = M records * N/M runs
After storing the runs on two tapes, each contains half of the runs:
2 tapes * M records_per_run * (1/2)(N/M) runs = N records
After merge 1st pass - double the length of the runs, halve the number of the runs:
2 tapes * 2M records_per_run * (1/2)(1/2)(N/M) runs = N records
After merge 2nd pass :
2 tapes * 4M records_per_run * (1/4)(1/2) (N/M) runs = N records
.....

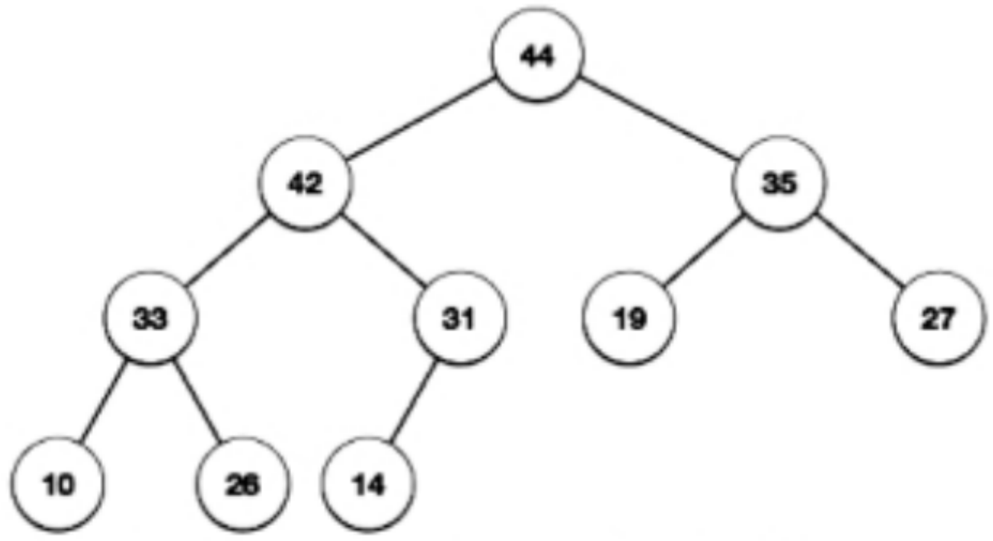
After merge s-th pass:
2 tapes * 2s M records_per_run * (1/2s)(1/2)(N/M) runs = N records
.....
Thus the length of the runs after the s-th merge is 2^sM.
After the last merge there is only one run equal to the whole file:
$2^sM = N$
$2^s = N/M$
$s = \log(N/M)$
if s is the last merge, $s = \log(N/M)$
At each pass we process N records, so the complexity is O(Nlog(N/M))

b) Heap Creation Technique:
A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array. Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * i + 1 and right child by 2 * i + 2 (assuming the indexing starts at 0).

Example:
Min-Heap – the value of the root node is less than or equal to either of its children.



Max-Heap – the value of the root node is greater than or equal to either of its children.



Both trees are constructed using the same input and order of arrival.

Max Heap Construction Algorithm

The procedure to create Min Heap is similar but we go for min values instead of max values. The technique to derive an algorithm for max heap is to insert one element at a time. At any point of time, heap must maintain its property. While insertion, it should be considered that the inserting a node in an already heapified tree.

Step 1 – Create a new node at the end of heap.
Step 2 – Assign new value to the node.
Step 3 – Compare the value of this child node with its parent.
Step 4 – If value of parent is less than child, then swap them.
Step 5 – Repeat step 3 & 4 until Heap property holds.

c) 8 queens problem:

8-queens problem is one of the common problems. This happens to be an appropriate example, where backtracking may be used very effectively. The problem is to set 8 queens in an 8×8 chessboard that no two queens will attack each other. So, no two queens are in the same row or same column or same diagonal. In general, it can be described as to place n queens in an n×n chessboard in such a way that no two queens attack one another.

The solution space required is $[8^2(8^2 - 1)(8^2 - 2)\ldots(8^2 - 7)]$ to solve this problem. So, in general we can say that for 8-queens problem, the solution space is $\prod_{i=0}^{8-1}(8^2 - i)$ and similarly, we can say for n-queens problem, that the required solution space is $\prod_{i=0}^{n-1}(n^2 - i)$. The condition for placing 8-queens in an 8 x 8 checkerboard, each queen can be placed in any square of each row or column, and there are 8 squares in each row and column, so we can place 8 queens in 8^8 locations. But if we consider initially that 8 different queens are in 8 different rows then the solution space reduces to 8!. Because, if we have placed queen 1 in row 1 then it takes any one of eight columns. But, there is an option for second queen being placed in just remaining 7 columns and third queen has remaining 6 columns to be placed in and so on. So, without loss of generality, we can place queen q_i at row i for $1 \leq i \leq 8$. Now, we can define solutions of this 8 queens problem as 8 tupples $(x_1, x_2 \ldots x_8)$,

where each x_i denotes the column index where queen q_i is placed, so that no two queens are attacking each other.
Here, we have shown some instances of 8-queens problem in figure 1.

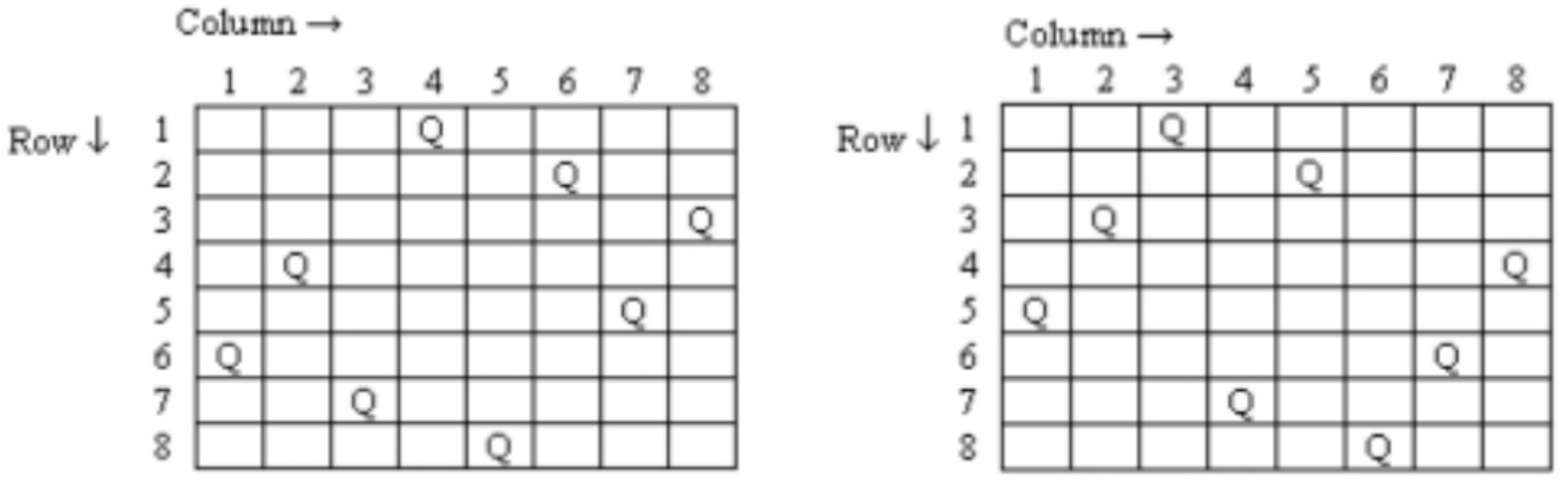


Fig: 1 Placing 8 queens in an 8 × 8 checkerboard in two different way

Now, before discussing 8-queens problem, first we shall study 4-queens problem. Because it is easy to solve and is possible to draw path of backtracking for better comprehension.

d) Hamiltonian cycle:
Refer to Question No. 7 of Long Answer Type Questions.

e) Heuristic Algorithm:
A heuristic algorithm is one that is designed to solve a problem in a faster and more efficient fashion than traditional methods by sacrificing optimality, accuracy, precision, or completeness for speed. Heuristic algorithms often times used to solve NP-complete problems, a class of decision problems. In these problems, there is no known efficient way to find a solution quickly and accurately although solutions can be verified when given. Heuristics can produce a solution individually or be used to provide a good baseline and are supplemented with optimization algorithms. Heuristic algorithms are most often employed when approximate solutions are sufficient and exact solutions are necessarily computationally expensive.
The following are well-known examples of "intelligent" algorithms that use clever simplifications and methods to solve computationally complex problems.
Swarm Intelligence: Swarm Intelligence systems employ large numbers of agents interacting locally with one another and the environment. Swarm intelligence refers to the collective behavior of decentralized systems and can be used to describe both natural and artificial systems. Specific algorithms for this class of system include the particle swarm optimization algorithm, the ant colony optimization algorithm, and artificial bee colony algorithm. Each of the previous algorithms was inspired by the natural, self-organized behavior of animals.
Genetic Algorithms: Genetic algorithms are a subset of a larger class of evolutionary algorithms that describe a set of techniques inspired by natural selection such as inheritance, mutation, and crossover. Genetic algorithms require both a genetic representation of the solution domain and a fitness function to evaluate the solution

domain. The technique generates a population of candidate solutions and uses the fitness function to select the optimal solution by iterating with each generation. The algorithm terminates when the satisfactory fitness level has been reached for the population or the maximum generations have been reached.

Artificial Neural Networks: Artificial Neural Networks (ANNs) are models capable of pattern recognition and machine learning, in which a system analyzes a set of training data and is then able to categorize new examples and data. ANNs are influenced by animals' central nervous systems and brains, and are used to solve a wide variety of problems including speech recognition and computer vision.

Support Vector Machines: Support Vector Machines (SVMs) are models with training data used by artificial intelligence to recognize patterns and analyze data. These algorithms are used for regression analysis and classification purposes. Using example data, the algorithm will sort new examples into groupings. These SVMs are involved with machine learning, a subset of artificial intelligence where systems learn from data, and require training data before being capable of analyzing new examples.

f) Bellman-Ford Algorithm:

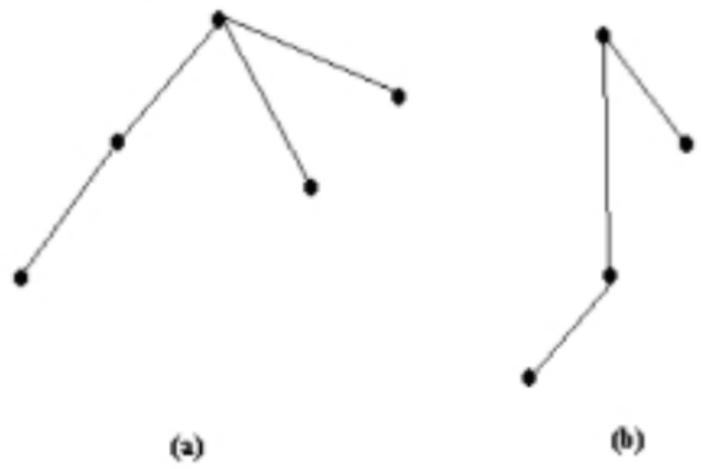
Refer to Question No. 11 of Short Answer Type Questions.

GRAPH AND TREE TRAVERSAL ALGORITHMS

☞ Chapter at a Glance

- **Tree:**

A connected graph with no cycle is called a tree. Generally, a graph that does not contain any cycles is called an acyclic graph. So, we can say that a connected acyclic graph is a tree.



(a) (b)

- **Minimum cost spanning tree:** A minimum cost spanning tree T of an edge weighted graph G contains all the vertices of G with total cost of the edges are minimum. It may not be unique.

  Two methods for finding the minimum cost spanning tree of a weighted graph G:

  1. Prim's Algorithm
  2. Kruskal's Algorithm

- **DFS (Depth First Search):**

  In a graph G = (V, E) Depth First Search (DFS) algorithm starts from a specific vertex u ϵ V, which is labeled as the current vertex. Then we traverse graph by any edge (u, v) incident to the current vertex u. If the edge (u, v) leads to an already visited vertex v, then we backtrack to current vertex *u*. Otherwise if edge (u, v) leads to an unvisited vertex v, then we move to v and v becomes our new current vertex. This process is continued until we get no vertex in this path. At this point, we start backtracking. The process terminates when backtracking leads back to the starting vertex with no vertex remaining unvisited.

- **BFS (Breadth First Search):**

  Breadth First search (BFS) is a general technique for traversing a graph. It may be a directed graph or an undirected graph. Breath First Search (BFS) starts from a given vertex of a graph, which is at level 0. In the first stage, we visit all vertices at level 1. In the second stage, we visit all vertices at second level. i.e., visit the unvisited vertices, which are adjacent to the vertices at level 1, and so on. The BFS traversal terminates when every vertex of that graph has been visited. BFS method can label each vertex by the length of a shortest path (in terms of number of edges) from the starting vertex up to that vertex.

- **Network Flow Diagram:**

  A flow network G = (V, E) is a directed graph in which each edge (u, v) $\in$ E has a nonnegative capacity c(u, v) $\geq$ 0. If (u, v) $\notin$ E, we assume that c(u, v) = 0. We distinguish

two vertices in a flow network: a source s and a sink t. For convenience, we assume that every vertex lies on some path from the source to the sink.
That is, for every vertex $v \in V$, there is a path $s \rightarrow v \rightarrow t$. The graph is therefore connected, and $|E| \geq |V| - 1$. We are now ready to define flows more formally. Let G = (V, E) be a flow network with a capacity function c. Let s be the source of the network, and let t be the sink. A flow in G is a real-valued function $f : V \times V \rightarrow R$ that satisfies the following three properties:
Capacity constraint: For all $u, v \in V$, we require $f(u, v) \leq c(u, v)$.
Skew symmetry: For all $u, v \in V$, we require $f(u, v) = -f(v, u)$.
Flow conservation: For all $u \in V - \{s, t\}$

Multiple Choice Type Questions

1. Complexity of BFS algorithm is ..... if graph is represented as adjacency list **[WBUT 2006, 2015]**

a) $\Theta(n+e)$ b) $\Theta(n^2)$ c) $\Theta(\log n)$ d) $\Theta(n + e \log n)$

Answer: (a)

2. BFS on a graph G = (V, E) has running time **[WBUT 2010, 2016]**

a) $O(|V|+|E|)$ b) $O(|V|)$ c) $O(|E|)$ d) none of these

Answer: (a)

3. Which of the following is useful in traversing a given graph using BFS? **[WBUT 2011]**

a) stack b) linked list c) array d) queue

Answer: (d)

4. The diagonal of the adjacency matrix of a graph with a self-loop contains only **[WBUT 2011]**

a) 1 b) 0 c) –1 d) ∞

Answer: (b)

5. Level order traversal of a rooted tree can be done by starting from the root and performing **[WBUT 2013]**

a) depth first search
b) breadth first search
c) pre-order traversal
d) in-order traversal

Answer: (b)

6. An undirected graph G with *n* vertices and *e* edges is represented by adjacency list. What is the time required to generate all the connected components? **[WBUT 2014]**

a) O(*n*) b) O(*e*) c) O(*e* + *n*) d) O(e^2)

Answer: (c)

7. An adjacency matrix representation of a graph cannot contain information of **[WBUT 2014]**

a) nodes
b) edges
c) direction of edges
d) parallel edges

Answer: (d)

8. The data structure required for Breadth First Traversal on a graph is **[WBUT 2014]**

a) queue **b) stack** **c) array** **d) tree**

Answer: (a)

9. BFS on a graph has running time **[WBUT 2014]**

a) O(|V| + |E|) **b) O(|V|)** **c) O(|E|)** **d) O($|V^2|$)**

Answer: (a)

Short Answer Type Questions

1. Compare and contrast: Depth-first-search and Breadth-first search. **[WBUT 2004, 2007, 2012, 2017]**

Answer:

Compare:

- Both BFS and DFS traverse or visit all the vertices of a connected graph.
- Both BFS and DFS are applicable on spanning tree, paths and cycles of connected graphs.

Contrast:

DFS	BFS
1. DFS cannot find a shortest path between two vertices.	1. BFS finds the shortest path between two vertices of a graph.
2. DFS visits all nodes on path first and Backtrack when path ends.	2. But BFS is terminated if it reaches end of a path in a graph.
3. In case of DFS we use stack operation.	3. To implement BFS we use queue structure
4. DFS can represent back edge i.e. an edge from descendent to ancestor. Let back_edge(v, w) denote that w is an ancestor of v in the tree of discovery edges where v and w are two vertices of the graph.	4. BFS can represent the cross edge, i.e. cross_edge (v, w) denotes that w is in the same level as v or in the next level in the tree of discovery edges where v and w are two vertices of the graph.

2. Describe Floyd's algorithm for all pair shortest path problem. Find its time complexity. **[WBUT 2005, 2011]**

OR,

Write down Floyd's algorithm to find all paired shortest paths of a graph. **[WBUT 2017]**

OR,

Write an algorithm for all pair shortest path also compute its complexity. **[WBUT 2018, 2019]**

Answer:

Given a directed graph G = (V, E), where each edge (v, w) has a nonnegative cost C[v, w], for all pairs of vertices (v, w) find the cost of the lowest cost path from v to w.

Floyd's algorithm takes as input the cost matrix C[v, w]

- C[v, w] = ∞ if (v, w) is not in E

It returns as output

- a distance matrix D[v, w] containing the cost of the lowest cost path from v to w and initially D[v, w] = C[v, w]
- a path matrix P, where P[v, w] holds the intermediate vertex k on the least cost path between v and w that led to the cost stored in D[v, w].

We iterate N times over the matrix D, using k as an index. On the k-th iteration, the D matrix contains the solution to the All pair shortest path problem, where the paths only use vertices numbered 1 to k.

On the next iteration, we compare the cost of going from i to j using only vertices numbered 1..k (stored in D[i,j] on the k-th iteration) with the cost of using the k+1th vertex as an intermediate step, which is D[i,k+1] (to get from i to k+1) plus D[k+1,j] (to get from k+1 to j).

If this results in a lower cost path, we maintain it.

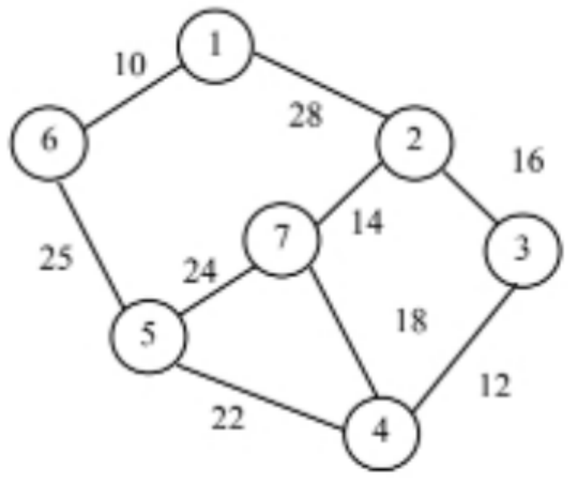
After N iterations, all possible paths have been examined, so D[v, w] contains the cost of the lowest cost path from v to w using all vertices if necessary.

Algorithm

```
FloydAPSP (N, C, D, P)
{
  for i ← 0  to N do
  {
     for j ← 0 to N do
     {
        D[i][j] ← C[i][j]
        P[i][j] ← -1
     }
     D[i][i] ← 0.0
  }
  for k ← 0 to N do
  {
      for i ← 0 to N do
      {
        for j ← 0 to N do
        {
            if ((D[i][k] + D[k][j] )< D[i][j] ) then
            {
                  D[i][j] ← D[i][k] + D[k][j]
                    P[i][j] ← k
            }
        }
     }
  }
}
```

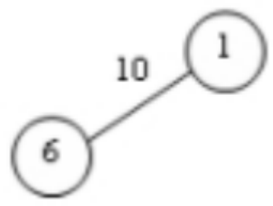
The complexity of the above algorithm depends upon two nested **for** loops. First one is executed at most n^2 time. Another nested **for** loop has three inner **for** loop and the total execution time is n^3. So, the complexity of the above algorithm is O (n^3).

3. Find out the minimum cost spanning tree using any algorithm: [WBUT 2007]

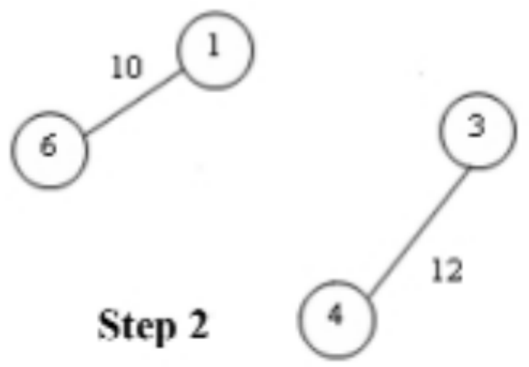


Answer:

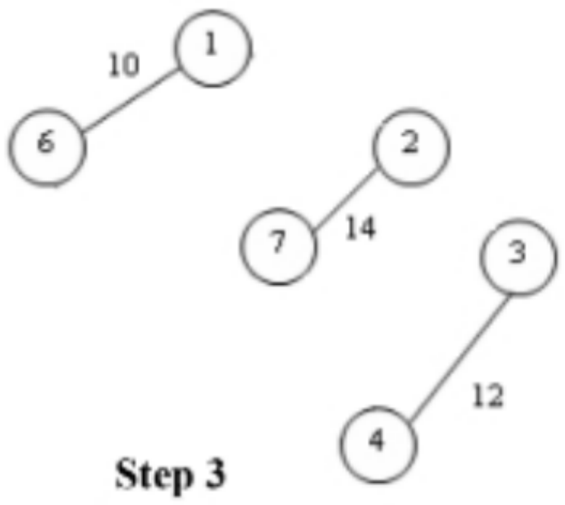
Applying Kruskal's Algorithm,



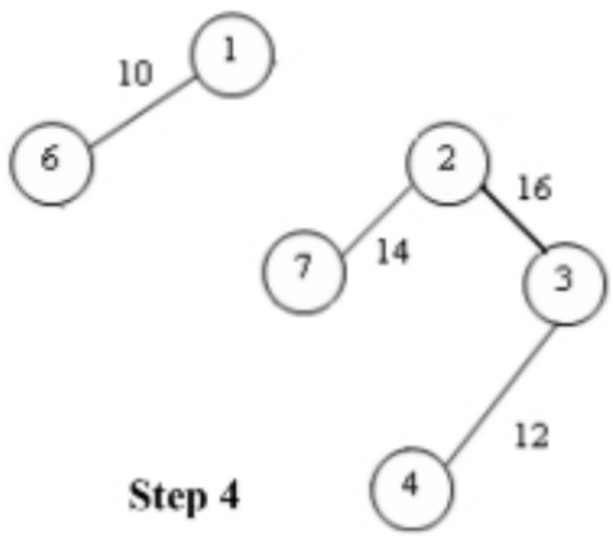
Step 1



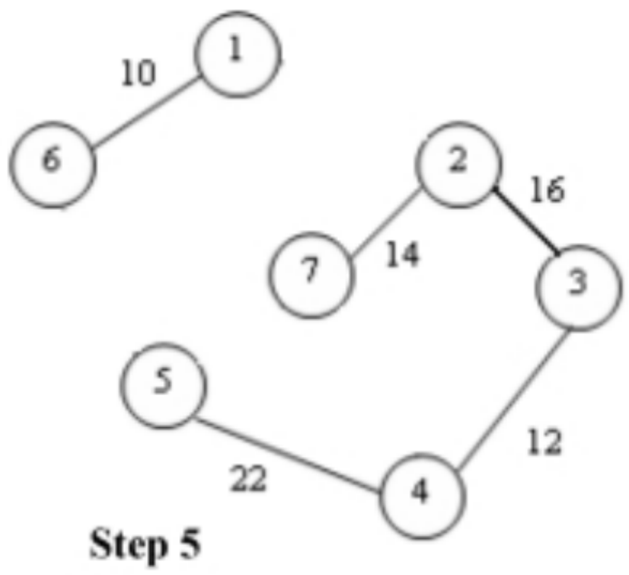
Step 2



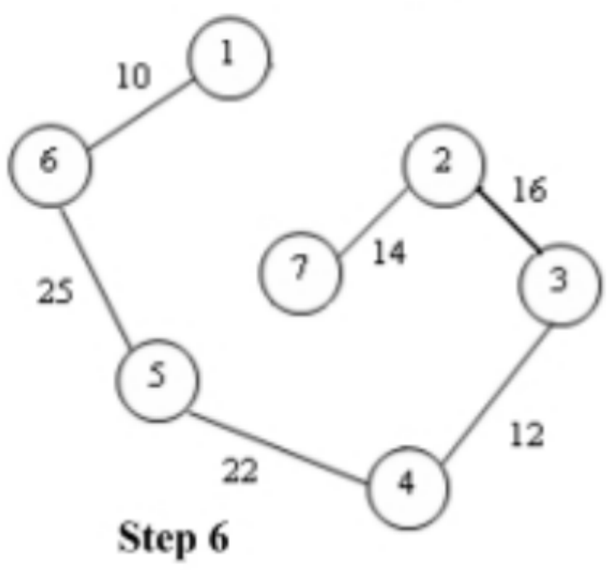
Step 3



Step 4



Step 5



Step 6

In step 6 we get the minimum cost spanning tree by kruskal's method and the minimum cost is 99.

4. Write down the difference between Prim's algorithm and Kruskal's algorithm. [WBUT 2007, 2017]

Answer:

Prim's	Kruskal's
Compare 1. Find the minimum cost spanning. 2. Generate the minimum cost spanning tree from a graph. **Contrast** 1. Generate the minimum cost spanning tree starting from a root node. 2. Selection of edges must be adjacent to the tree which is already created.	**Compare** 1. Find the minimum cost spanning. 2. Generate the minimum cost spanning tree from a graph. **Contrast** 1. Generate the minimum cost spanning tree starting from a least weighted untraverse non cycle forming edge. 2. The selection of edges may be discrete.

5. Explain the max-flow min-cut theorem with an example. [WBUT 2012, 2013, 2015]

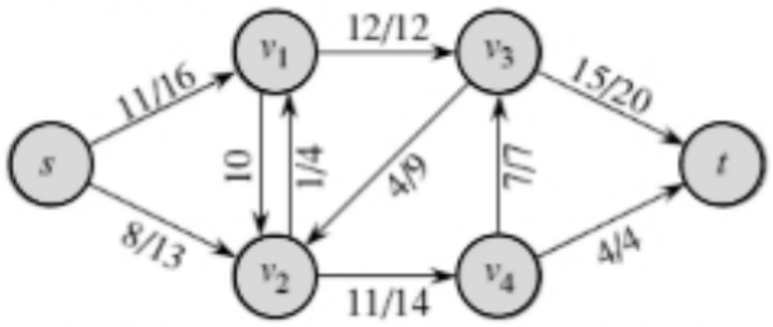
OR,

State the Max-flow min-cut theorem for network flow analysis. [WBUT 2014]

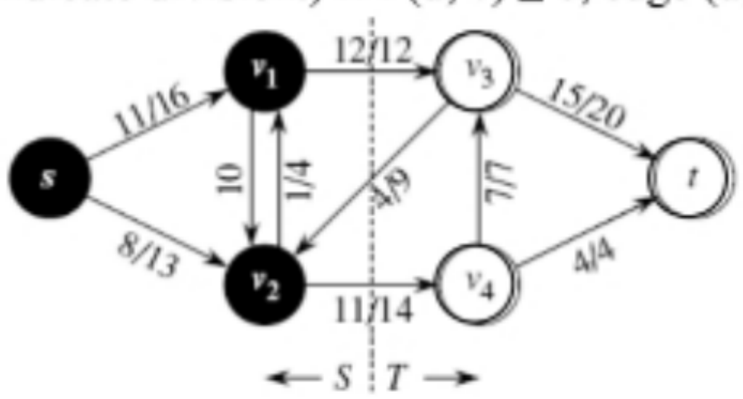
Answer:

If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of G.



A flow f in G with value $|f| = 19$. Only positive flows are shown. If f (u, v) > 0, edge (u, v) is labeled by f (u, v) /c(u, v). (The slash notation is used merely to separate the flow and capacity; it does not indicate division.) If f (u, v) ≤ 0, edge (u, v) is labeled only by its capacity.



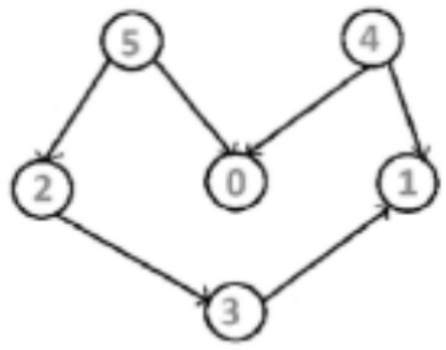
A cut (S, T) in the flow network of Figure, where S = {s, v1, v2} and T = {v3, v4, t }. The vertices in S are black, and the vertices in T are white. The net flow across (S, T) is f (S, T) = 19, and the capacity is c(S, T) = 26.

6. Define Topological sorting. Write down the difference between Topological sorting and DFS. **[MODEL QUESTION]**

Answer:

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



Topological Sorting vs Depth First Traversal (DFS):

In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex '5' should be printed before vertex '0', but unlike DFS, the vertex '4' should also be printed before vertex '0'. So Topological sorting is different from DFS. For example, a DFS of the shown graph is "5 2 3 1 0 4", but it is not a topological sorting

Long Answer Type Questions

1. Write any algorithm for performing BFS over a directed graph. Comment on the time-complexity of your algorithm. **[WBUT 2004, 2011]**

Answer:

BFS traversal algorithm of a graph

In the BFS algorithm, we maintain a queue. When one or more than one vertices are discovered, we can store them into the queue. i.e., adjacent vertices of an explored vertex are stored into the queue. Then we remove one vertex (say i) from front position of the queue and store the adjacent vertices of i at the end of the queue.

BFS(G, s)

```
{
Step 1 initialize vertices;
Step 2 Q = {s};
// Q is a queue; initialize to s
Step 3 while (Q not empty) then
{
Step 3.1 u = RemoveTop(Q);
Step 3.2 for each v ∈ u→adj do
{
```

```
Step 3.2.1 if (v→color = = WHITE) then
Step 3.2.1.1 v→color = GREY;
Step 3.2.2 v→d = u→d + 1;
Step 3.2.3 v→p = u;
Step 3.2.4 Enqueue(Q, v);
}
Step 3.3u->color = BLACK;
}
```

Enqueue(Q, v)

```
{
Step 1 if Q is full then
Step 1.1 return -1;
Step 2 else
Step 2.1 Add adjacency vertices of v to Q;
}
```

Complexity of BFS() algorithm

- BFS calculates the *shortest-path distance* from the source node
  - Shortest-path distance δ(s,v) = minimum number of edges from s to v, or ∞ if v is not reachable from s
- BFS builds *breadth-first tree*, in which paths to root represent shortest paths in G
  - Thus we can use BFS to calculate shortest path from one vertex to another in O(|V| + |E|) time
- The space complexity for BFS is equal to O(n) i.e. the total space for keeping n vertices in the memory

2. Write an algorithm for finding a minimum spanning tree of an undirected acyclic graph. Estimate the time-complexity of your algorithm. [WBUT 2004, 2006, 2008]

OR,

Write an algorithm for finding the minimum spanning tree of a graph. Discuss its time complexity. [WBUT 2009, 2016]

OR,

Write an algorithm to find a minimum spanning tree (MST) for an undirected graph. Estimate the time complexity of your algorithm. [WBUT 2014, 2019]

Answer:

We can write either Prim's or Kruskal's algorithm for minimum cost spanning tree. Here we define Prim's algorithm.

A minimum cost spanning tree T of an edge weighted graph G contains all the vertices of G with total cost of the edges are minimum. It may not be unique.

Prim's algorithm for minimum cost spanning tree

Input: A weighted, undirected graph G = (V, E, w).

Output: A minimum cost spanning tree T.

```
Prim (V, E, w)
{
Step 1Initialize treeT={};
```

```
Step 2 Let r be a root vertex from V.
Step 3 r ϵ V;
Step 4 U = {r};
Step 5 while  | U| < n do
{
Step 5.1 Find(u) ϵ U and Find(v) ϵ V-U
such that the edge (u, v) ϵ E
Step 5.2 if ((u, v) is the least cost edge between
U and (U-V)) then.
{
Step 5.2.1 T = T ∪{(u, v)}
Step 5.2.2 U= U ∪{v}
}
}
}
```

Complexity

The spanning tree of the G = (V,E) has n number of nodes and (n-1) number of edges. Now, in the above algorithm in Step 5.2.1 we have to put (n-1) number of edges in the set T one by one. So it requires O(n) times. Similarly in Step 5.2.2 requires O(n) time to keep n nodes in the set U.

Now, in Step 5.1 , we are trying to find out two nodes u and v from sets U and V-U and each Find() requires O(1) time. In Step 5, the while loop is executed at most (n-1) times and within the while loop there are instructions which are also executed n times. So the execution time of while loop is $O(n^2)$.

So the time complexity of Prim's Algorithm is $O(n^2)$.

3. a) Write down the Dijkstra's algorithm for finding out the shortest path.

[WBUT 2008, 2011]

Answer:

Dijkstra's algorithm finds the shortest paths from a single source to all other nodes of a weighted digraph with positive weights. It solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. So, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$. Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

DIJKSTRA (G, w, s)
{
INITIALIZE-SINGLE-SOURCE (G, s)
S ← ∅
Q ← V [G]
while Q ≠ ∅ do

```
{
        u ← EXTRACT-MIN(Q)
        S ← S ∪ {u}
        for each vertex v ∈ Adj[u] do
        {
                RELAX (u, v, w)
         }
  }
}
RELAX(u, v, w)
{
  if d[v] > d[u] + w(u, v) then
  {
         d[v] ← d[u] + w(u, v)
          π[v] ← u
  }
}
INITIALIZE-SINGLE-SOURCE(G, s)
{
  for each vertex v ∈ V [G] do
   {
      d[v] ← ∞
      π[v] ← NIL
    }
   d[s] ← 0
}
```

b) Prove that the time complexity of Dijkstra's algorithm is

$3(n-2)(n-1)/2 = O(n^2)$. **[WBUT 2008, 2011]**

Answer:

With adjacency matrix representation, the running time is $O(n^2)$ By using an adjacency list representation and a partially ordered tree data structure for organizing the set V - S, the complexity can be shown to be O(e log n) where e is the number of edges and n is the number of vertices in the digraph.

4. Apply the KMP algorithm for the pattern p = "ababaca" and string s = "bacbababababacaab". Show every step. **[WBUT 2013]**

Answer:

Let us execute the KMP algorithm to find whether 'p' occurs in 'S'.

Initially : n = size of S = 15;

m = size of p=7

Step 1: i = 1 , q = 0
comparing p [1] with S[1]
String **b** a c b a b a b a b a c a a b
Pattern **a** b a b a c a
P[1] does not match with S[1]. 'p' will be shifted one position to the right.

Step 2: i = 2 , q = 0
comparing p [1] with S[2]
String b **a** c b a b a b a b a c a a b
Pattern **a** b a b a c a

Step 3: i = 3 , q = 1
comparing p [2] with S[3] p [2] does not match with S[3]
String b a **c** b a b a b a b a c a a b
Pattern a **b** a b a c a
Backtracking on p , comparing p [1] and S[3]

Step 4: i = 4 , q = 0
comparing p [1] with S[4] p [1] does not match with S[4]
String b a c **b** a b a b a b a c a a b
Pattern **a** b a b a c a

Step 5: i = 5 , q = 0
comparing p [1] with S[5]
String b a c b **a** b a b a b a c a a b
Pattern **a** b a b a c a

Step 6: i = 6 , q = 1
comparing p [2] with S[6] p [2] matches with S[6]
String b a c b **a b** a b a b a c a a b
Pattern **a b** a b a c a

Step 7: i = 7 , q = 2
comparing p [3] with S[7] p [3] matches with S[7]
String b a c b **a b** a b a b a c a a b
Pattern **a b** a b a c a

Step 8: i = 8 , q = 3
comparing p [4] with S[8] p [4] matches with S[8]
String b a c b **a b a** b a b a c a a b
Pattern **a b a** b a c a

Step 9: i = 9 , q = 4
comparing p [5] with S[9] p [5] matches with S[9]

String b a c b **a b a b** a b a c a a b
Pattern **a b a b** a c a

Step 10: i = 10 , q = 5
comparing p [6] with S[1 0] p [6] does n ' t matches with S[1 0]
String b a c b **a b a b a b** a c a a b
Pattern **a b a b a c** a
Backtracking on p, comparing p [4] with S[1 0] because after mismatch q = π[5] = 3

Step 11: i = 11 , q = 4
comparing p [5] with S[1 1]
String b a c b a b **a b a b a** c a a b
Pattern **a b a b a** c a

Step 12: i = 12 , q = 5
comparing p [6] with S[1 2] p [6] matches with S[1 2]
String b a c b a b **a b a b a c a** a b
Pattern **a b a b a** c a

Step 13: i = 13 , q = 6
comparing p [7] with S[1 3] p [7] matches with S[1 3]
String b a c b a b **a b a b a c a** a b
Pattern **a b a b a c a**
pattern 'p' has been found to completely occur in string 'S'. The total number of shifts that took place for the match to be found are: i − m = 13-7 = 6 shifts.

5. a) Write the string matching algorithm due to Knuth, Morris and Pratt. Analyze its time complexity. **[WBUT 2014]**

OR,

Write Knuth-Morirs-Pratt algorithm for string matching problem. **[WBUT 2015]**

Answer:

KMP-MATCHER*(T, P)*

```
1    n = length[T ]
2    m = length[P]
3    π = COMPUTE-PREFIX-FUNCTION(P)
4    q = 0   // Number of characters matched.
5    for i ← 1 to n      // Scan the text from left to right.
6        do while q > 0 and P[q + 1] ≠ T [i ]
7            do q = π[q]      // Next character does not match.
8            if P[q + 1] = T [i ]
9                then q = q + 1          //Next character matches.
10           if q = m                    // Is all of P matched?
11               then print "Pattern occurs with shift" i − m
12               q = π[q]                //Look for the next match.
```

COMPUTE-PREFIX-FUNCTION*(P)*

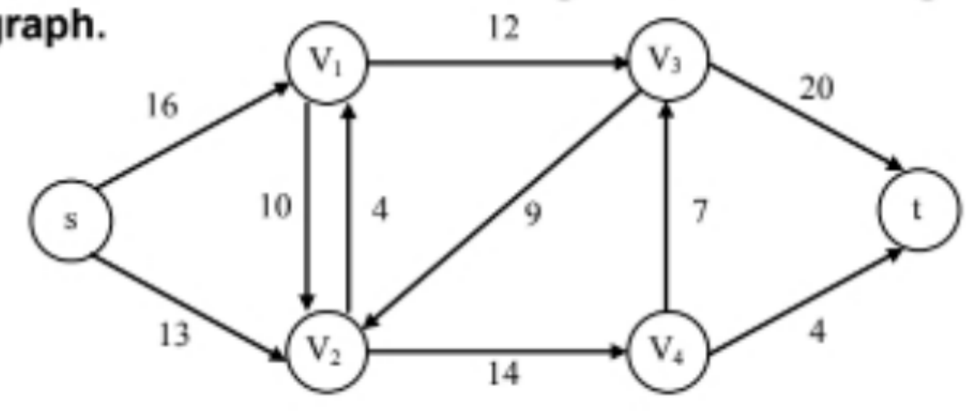
```
1     m ← length[P]
2     π[1] ← 0
3     k ← 0
4        for q ← 2 to m
5              do while k > 0 and P[k + 1] ≠ P[q]
6                    do k ← π[k]
7              if P[k + 1] = P[q]
8                    then k ← k + 1
9              π[q] ← k
10 return π
```

Running-time analysis

We begin with an analysis of the running times of these procedures. Proving these procedures correct will be more complicated.

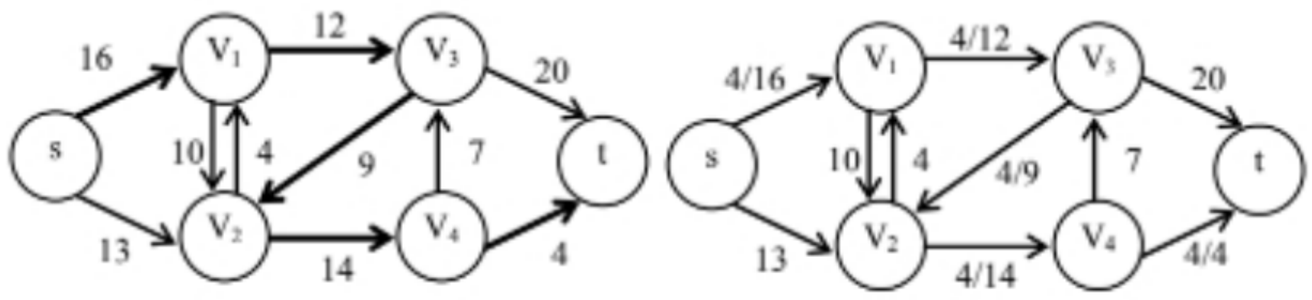
The running time of COMPUTE-PREFIX-FUNCTION is Θ(m), using the potential method of amortized analysis. We associate a potential of k with the current state k of the algorithm. This potential has an initial value of 0, by line 3. Line 6 decreases k whenever it is executed, since $\pi[k] < k$. Since $\pi[k] \geq 0$ for all k, however, k can never become negative. The only other line that affects k is line 8, which increases k by at most one during each execution of the for loop body. Since $k < q$ upon entering the **for loop**, and since q is incremented in each iteration of the for loop body, $k < q$ always holds. We can pay for each execution of the while loop body on line 6 with the corresponding decrease in the potential function, since $\pi[k] < k$. Line 8 increases the potential function by at most one, so that the amortized cost of the loop body on lines 5–9 is O(1). Since the number of outer-loop iterations is Θ(m), and since the final potential function is at least as great as the initial potential function, the total actual worst-case running time of COMPUTE-PREFIX-FUNCTION is Θ(m). A similar amortized analysis, using the value of q as the potential function, shows that the matching time of KMP-MATCHER is Θ(n). Compared to FINITE-AUTOMATON-MATCHER, by using π rather than δ, we have reduced the time for preprocessing the pattern from $O(m|\Sigma|)$ to Θ(m), while keeping the actual matching time bounded by Θ(n).

b) Trace the execution of Ford-Fulkerson algorithm for finding the maximum flow in the following graph. **[WBUT 2014]**

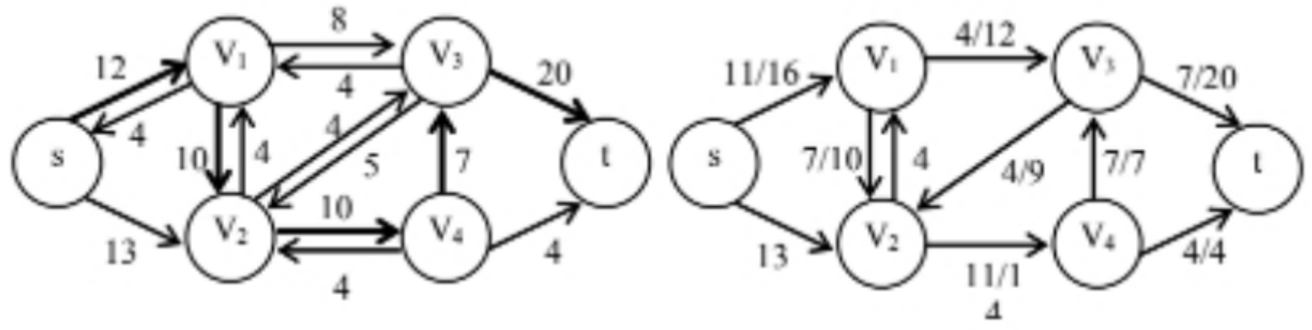


Answer:

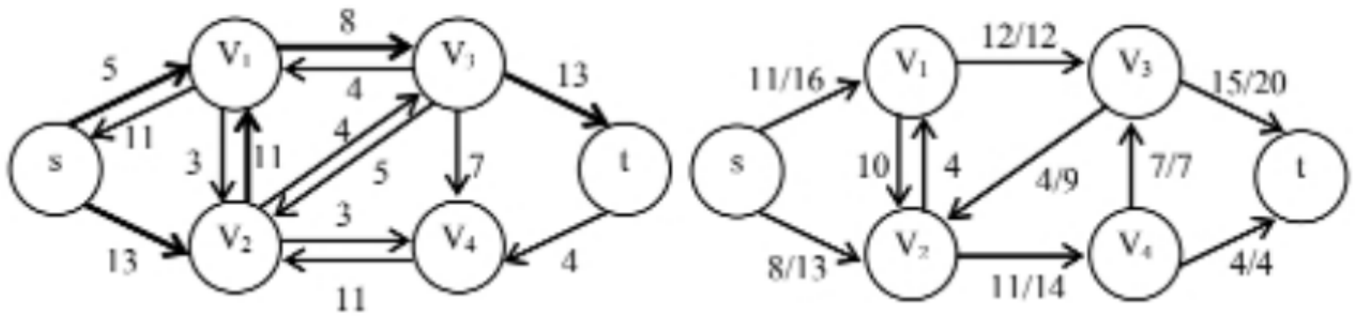
(a)



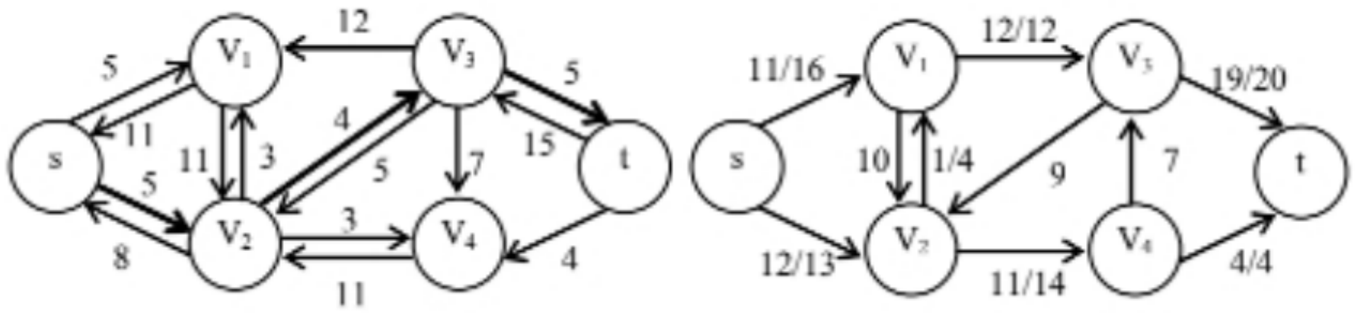
(b)



(c)

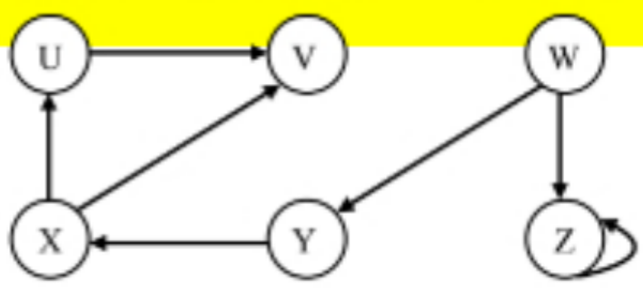


(d)

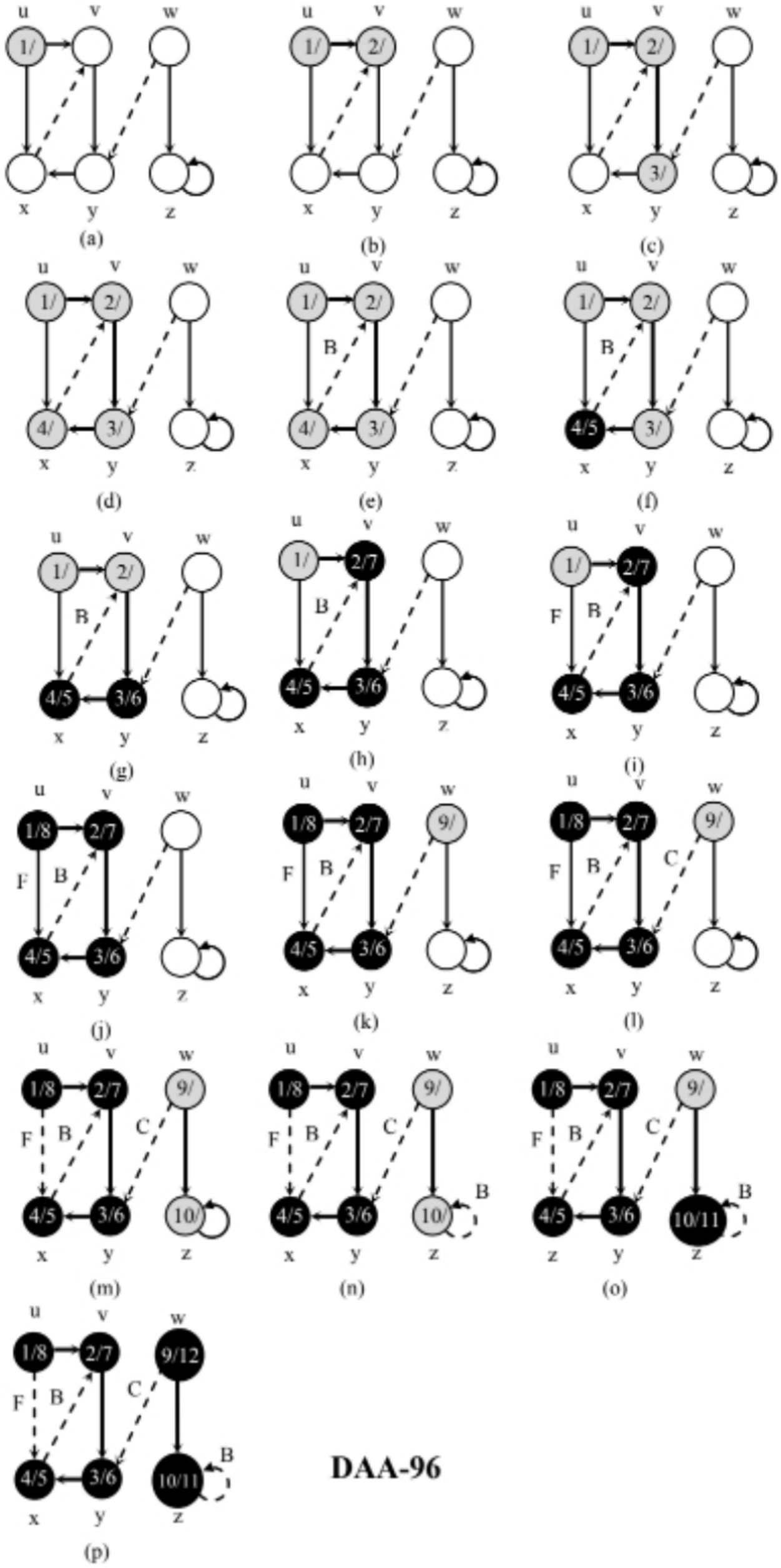


The execution of the basic Ford-Fulkerson algorithm: (a)–(d) Successive iterations of the while loop. The left side of each part shows the residual network G_f from line 4 with a shaded augmenting path p. The right side of each part shows the new flow f that result from adding f_p to f. The residual network in (a) is the input network G. (e) The residual network at the last while loop test. It has no augmenting paths, and the flow f shown in (d) is therefore a maximum flow.

6. Describe the Depth first search algorithm for a given graph and explain its time complexity. **[WBUT 2016]**



Answer:



In the above figure, the progress of the depth-first-search algorithm DFS is shown step by step on the given graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Non-tree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Timestamps within vertices indicate discovery time/finishing times.

The following pseudocode is the basic depth-first-search algorithm. The input graph G may be undirected or directed. The variable *time* is a global variable that we use for timestamping.

```
DFS(G)
1 for each vertex u ∈ G.V
2        u.color = WHITE
3        u.π =NIL
4 time= 0
5 for each vertex u∈ G.V
6        if u.color = = WHITE
7                DFS-VISIT.G; u/
DFS-VISIT(G, u)
1 time = time + 1
2 u.d=time
3 u.color= GRAY
4 for each v∈ G.Adj[u]
5        if  v.color = = WHITE
6                v.π = u
7                DFS-VISIT (G, v)
8 u.color = BLACK
9 time =time + 1
10 u.f = time
```

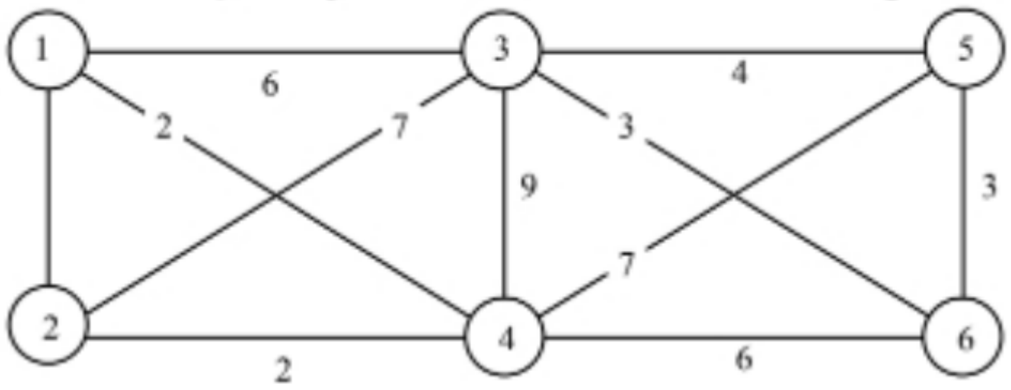
The loops on lines 1–3 and lines 5–7 of DFS take time $\Theta(V)$, exclusive of the time to execute the calls to DFS-VISIT. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$, since the vertex u on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint vertex u gray.

During an execution of DFS-VISIT (G,v), the loop on lines 4–7 executes $|Adj[v]|$ times. Since,

$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$

The total cost of executing lines 4–7 of DFS-VISIT is $\Theta(E)$. The running time of DFS is therefore $\Theta(V+E)$.

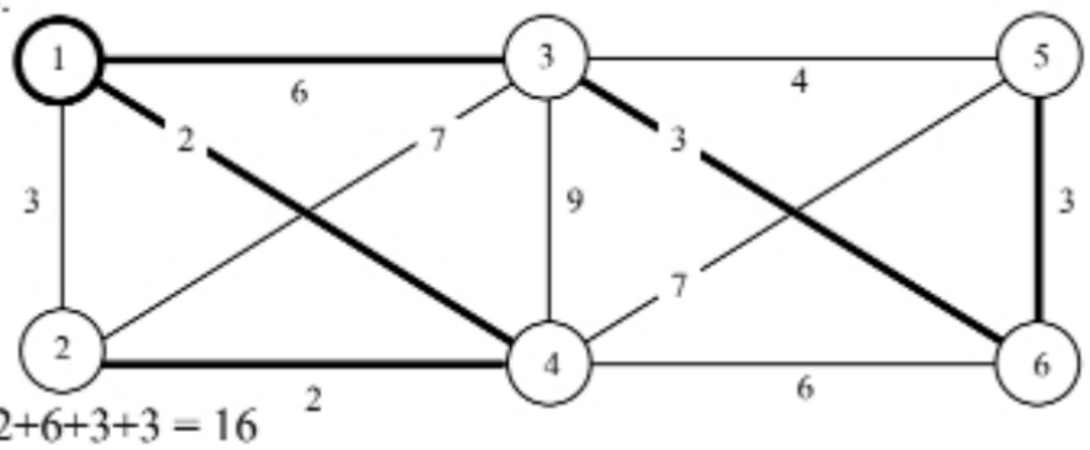
7. Find the minimum cost spanning tree using Prim's algorithm for the graph given below. Write down the complexity of Prim's and Kruskal's algorithm. **[WBUT 2017]**



Answer:

Minimum cost spanning tree (MST) of the graph

There is no weight between node 1 and node 2. So, we assume that it is 3 and node 1 is the starting node.



The cost is = 2+2+6+3+3 = 16

Complexity of Prim's and Kruskal's algorithm:

Time Complexity of the Prim's algorithm is $O(V^2)$. If the input graph is represented using adjacency list, then the time complexity of Prim's algorithm can be reduced to *O(E log V)* with the help of binary heap. The performance of Prim's algorithm depends on how we implement the min-priority queue *Q*.

Time Complexity of the Kruskal's algorithm is *O(ElogE)* or *O(ElogV)*. Sorting of edges takes *O(ELogE)* time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take at most *O(LogV)* time. So overall complexity is *O(ELogE + ELogV)* time. The value of E can be at most $O(V^2)$, so *O(LogV)* are *O(LogE)* same. Therefore, overall time complexity is *O(ElogE)* or *O(ElogV)*.

8. Write short notes on the following:

a) Dijkstra's Algorithm **[WBUT 2013, 2018, 2019]**

b) Kruskal's algorithm for finding MST **[WBUT 2014]**

c) Minimum spanning tree **[WBUT 2015]**

d) BFS and DFS **[WBUT 2015, 2017, 2019]**

e) Ford – Fulkerson algorithm **[WBUT 2017, 2018]**

Answer:

a) Dijkstra's Algorithm:

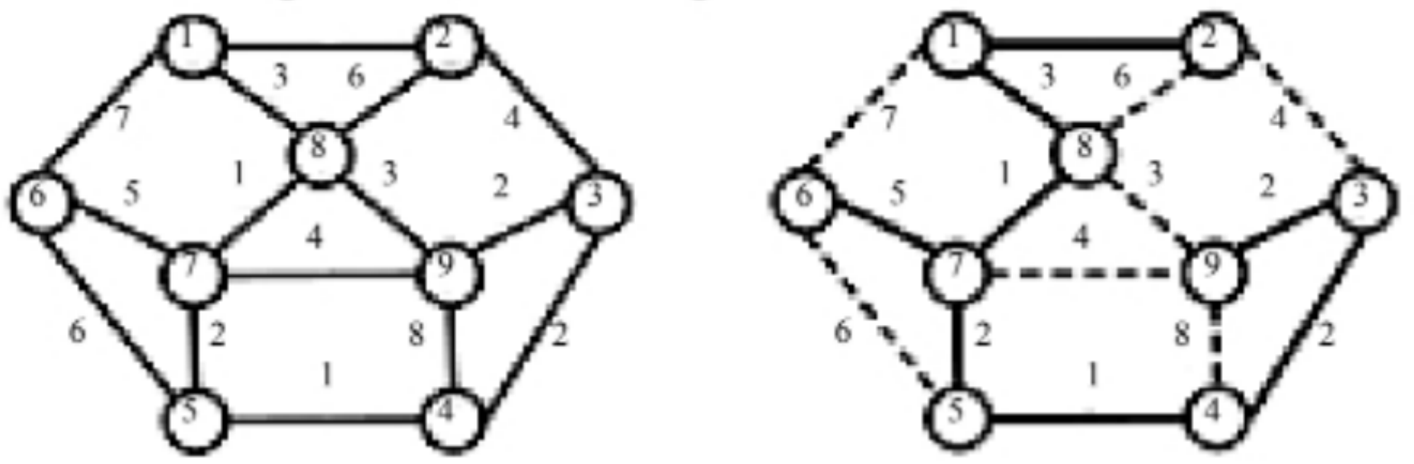
Refer to Question No. 3(a) of Long Answer Type Questions.

b) Kruskal's algorithm for finding MST:
Refer to Question No. 3 of Short Answer Type Questions.

c) Minimum spanning tree:
A spanning tree of a graph $G=(V,E)$ is a connected subgraph with $|V|-1$ edges spanning all of V. The cost (or weight) of a tree is the sum of the length of the edges in the tree. A minimum spanning tree (MST) is a spanning tree with minimum cost. It is well known and easy to show that a spanning tree can be found in polynomial time. If W* denotes the weight (cost) of the minimum spanning tree, then we must have $W^* \le L^*$ since deleting any edge from the optimal tour results in a spanning tree.
The minimum spanning tree can be used to find a feasible traveling salesman tour in polynomial time.
Two methods are there for finding the minimum cost spanning tree of a weighted graph G:

1. Prim's Algorithm
2. Kruskal's Algorithm

Kruskal's Algorithm: Examines edges in mondecreasing order of their lengths and include them in MST if the added edge does not form cycle with the edges already chosen. The proof of the algorithm uses the path potimality conditions. Attractive algorithm if the edges are already sorted in increasing order of their lengths. The precedure of Kruskal's algorithm is shown in figure below.



(a) a grap $G=(V,E)$ and weight $w_{ij}, \forall (i,j) \in V$

Fig: A small-scale example of MST

Procedure: Kruskal's Algorithm
Input: graph $G=(V,E)$, weight $w_g, \forall (i,j) \in V$
Output: spanning tree T
Begin
$T \leftarrow \phi$;
$A \leftarrow E$; //A: eligible edges
While $|T| < |V| - 1$ do
Choose an edge (u, v) $\leftarrow$ argmin $\{w_g | (i,j) \in A\}$:

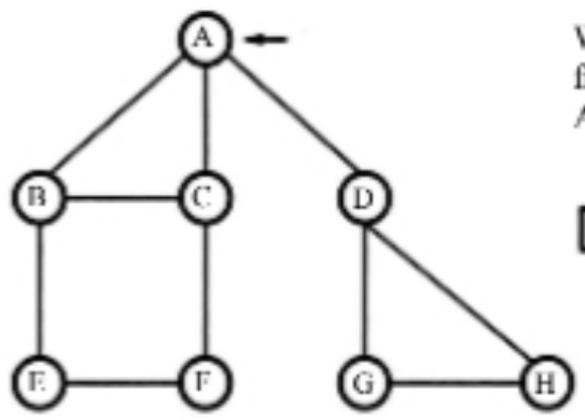
$A \leftarrow A \setminus \{(u,v)\}$;
output spanning tree T;
end

d) BFS and DFS:

Breadth first search algorithm:

1. Select an unvisited node v, visit it, have it as a root in a BFS. Its level is called the current level.
2. Form each node *x* in the current level, in the order in which the level node were visited. Visit all the unvisited neighbours of *x*. The newly visited nodes from this level form a new level. This new level becomes the next current level.
3. Repeat step 2 for all the unvisited vertices.
4. Repeat form step 1 until no more vertices are remaining.

Step 1:

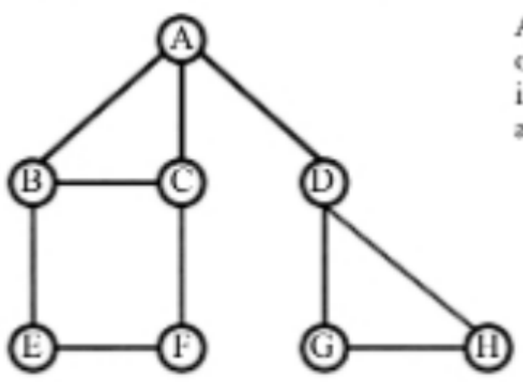


We will start search from node A. Insert A in the queue

Q

A	

Step 2:

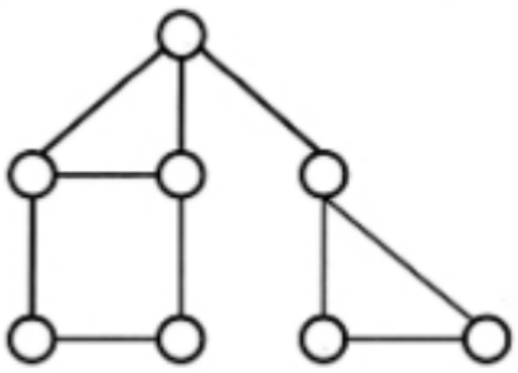


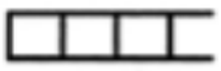
Adjacent nodes to A will be obtained and will be inserted in the Q. Delete A from Q and print it.

Q

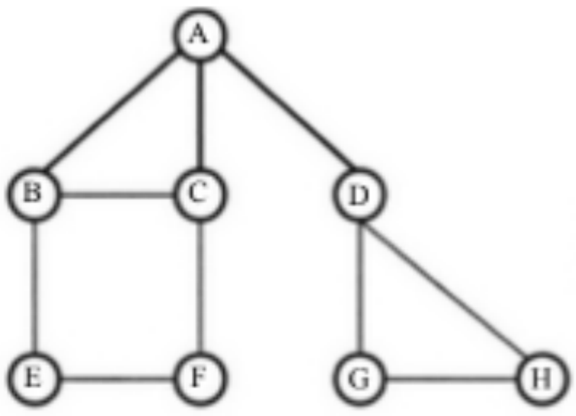
B	C	D

Step 3:





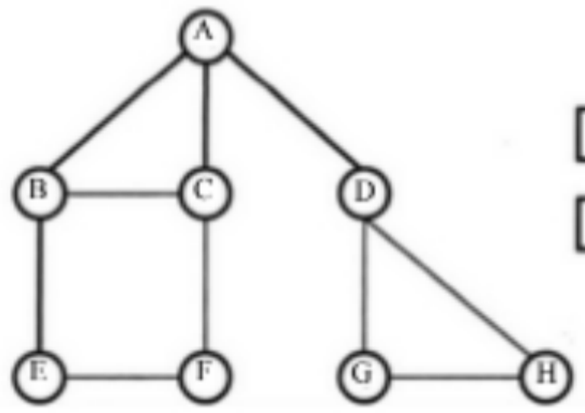
Step 4:

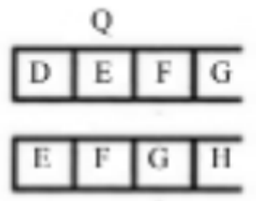


Q

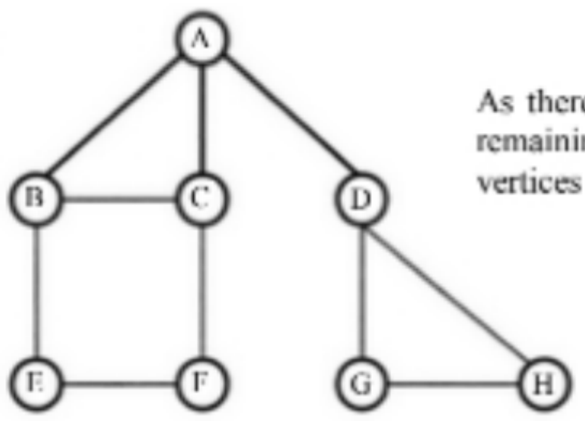
D	E	F

Step 5:





Step 6:



As there is no unvisited node remaining we will delete the vertices from queue and print

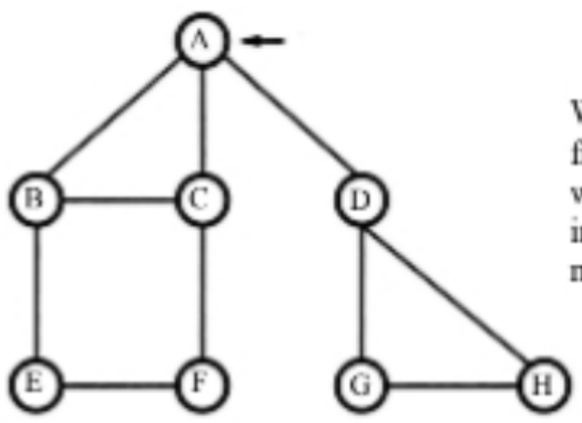
Hence we get a sequence as A, B, C, D, E, F, G, H.

Depth first search traversal

1. Select an unvisited node v, visit it and treat it as current node.
2. Find an unvisited neighbor of the current node, visit it and make it as new current node.
3. If current node has no unvisited neighbours back track to its parent and make it a new current node.
4. Repeat the step 2 and 3 until no more nodes can be visited.
5. Repeat from step 1 for the remaining nodes.

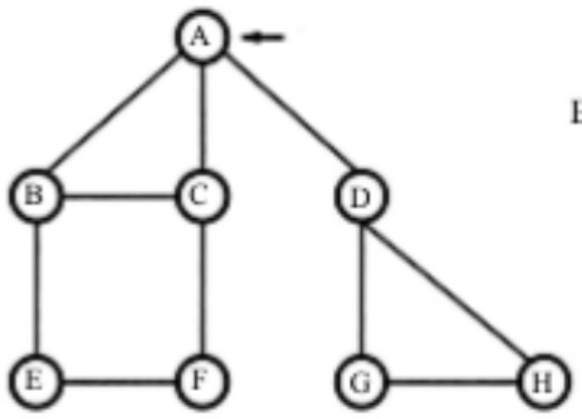
For example:

Step 1:



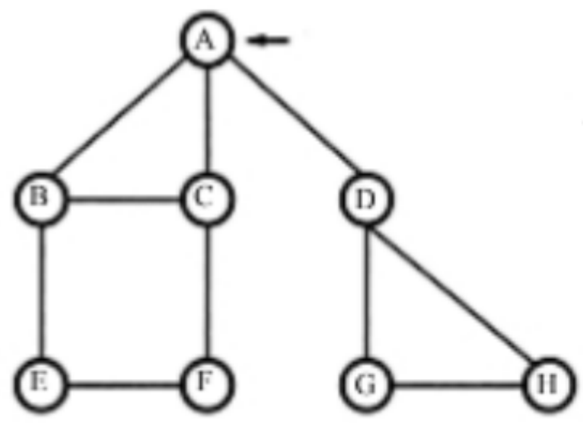
We will start search from node A. We will start searching in depth first manner.

Step 2:



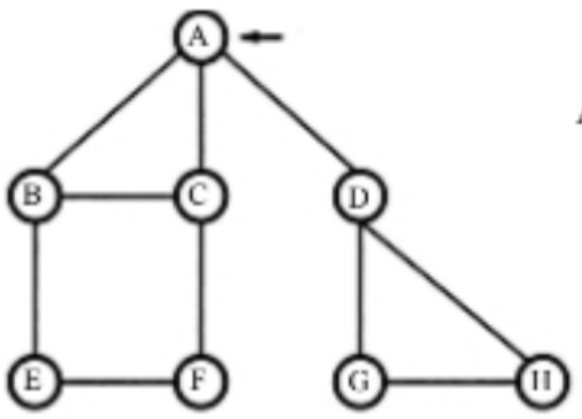
Edge A-B

Step 3:



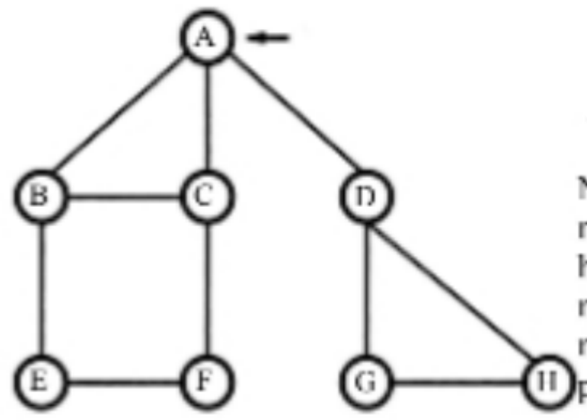
A-B-C

Step 4:



A-B-C-F

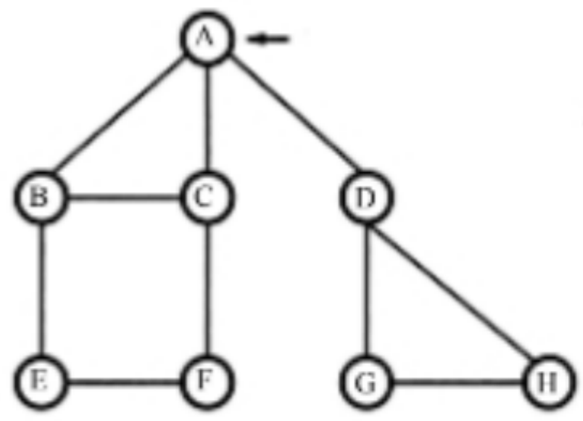
Step 5:



A-B-C-F-E

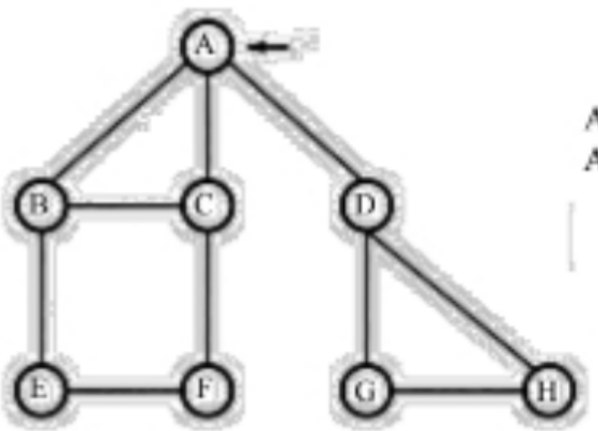
Now, there is no reachable node, further hence we will pop the nodes and search for a node from which we can proceed.

Step 6:



A-B-C-F-E-A-D

Step 7:



A-B-C-F-E
A-D-G-H

Hence we obtain DFS sequence as A, B, C, F, E, D, G, H.

e) Ford –Fulkerson algorithm:

Each iteration of the Ford-Fulkerson method, we find some augmenting path p and increase the flow f on each edge of p by the residual capacity $c_f(p)$. The following implementation of the method computes the maximum flow in a graph G=(V,E) by updating the flow f[u, v] between each pair (u,v) of vertices that are connected by an edge. If u and v are not connected by an edge in either direction, we assume implicitly that $f[u,v]=0$. The capacities $c(u,v)$ are assumed to be given along with the graph, and $c(u,v)=0$ if $(u,v) \notin E$ .The residual capacity $c_f(u,v)$ is computed in accordance with the formula

$c_f(u, v)= c(u, v) – f(u,v)$.

The expression $c_f(p)$ in the code is actually just a temporary variable that stores the residual capacity of the path p.

FORD-FULKERSON(G,s, t)

1 **for** each edge $(u, v) \in E[G]$
2 **do** $f[u, v] \leftarrow 0$
3 $f[v, u] \leftarrow 0$
4 **while** there exists a path p from s to t in the residual network G_f
5 **do** $c_f(p) \leftarrow \min \{c_f(u, v) : (u, v)$ is in $p\}$
6 **for** each edge (u, v) in p
7 **do** $f[u, v] \leftarrow f[u, v] + c_f(p)$
8 $f[v, u] \leftarrow -f[u, v]$

The FORD-FULKERSON algorithm simply expands on the FORD-FULKERSON METHOD pseudocode given earlier. Lines 1–3 initialize the flow f to 0. The while loop of lines 4–8 repeatedly finds an augmenting path p in G_f and augments flow f along p by the residual capacity $c_f(p)$. When no augmenting paths exist, the flow f is a maximum flow.

9. Given a flow network G = (V, E), let f_1 and f_2 be functions from V × V to R. The flow sum $f_1 + f_2$ is the function from V × V to R defined by $(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v)$ for all $u, v \in V$. If f_1 and f_2 are flows in G, which of the three flow properties must the flow sum $f_1 + f_2$ satisfy, and which might it violate? [MODEL QUESTION]

Answer:

The flow sum $(f_1 + f_2)$satisfies skew symmetry and flow conservation, but might violate the capacity constraint.

We give proofs for skew symmetry and flow conservation and an example that shows a violation of the capacity constraint.

Let $f(u, v) = (f_1 + f_2)(u, v)$.

For skew symmetry:

$$\begin{aligned} f(u, v) &= f_1(u, v) + f_2(u, v) \\ &= -f_1(v, u) - f_2(v, u) \qquad \text{(skew symmetry)} \\ &= -(f_1(v, u) + f_2(v, u)) \\ &= -f(v, u) . \end{aligned}$$

For flow conservation, let $u \in V - \{s, t\}$:

$$\sum_{v \in V} f(u,v) = \sum_{v \in V} \left(f_1(u,v) + f_2(u,v) \right)$$
$$= \sum_{v \in V} f_1(u,v) + \sum_{v \in V} f_2(u,v)$$
$$= 0 + 0 \qquad \text{(flow conservation)}$$
$$= 0$$

For the capacity constraint, let V = {s, t}, E = {(s, t)}, and c(s, t) = 1.
Let $f_1(s, t)=f_2(s, t)=1$. Then f_1 and f_2 obey the capacity constraint, but $(f_1 + f_2)(u, v)=2$, which violates the capacity constraint.

10. Let f be a flow in a network, and let α be a real number. The scalar flow product, denoted αf, is a function from V × V to R defined by (α f)(u, v) = α · f (u, v).
Prove that the flows in a network form a convex set. That is, show that if f_1 and f_2 are flows, then so is α f_1 + (1 − α) f_2 for all α in the range 0 ≤ α ≤ 1.

[MODEL QUESTION]

Answer:
To see that the flows form a convex set, we show that if f_1 and f_2 are flows, then so is α f_1 + (1 − α) f_2 for all α such that $0 \leq \alpha \leq 1$.
For the capacity constraint, first observe that $\alpha \leq 1$ implies that $1 - \alpha \geq 0$. Thus, for any u, $v \in V$,
We have, $\alpha f_1(u, v) + (1 - \alpha) f_2(u, v) \geq 0 \cdot f_1(u, v) + 0 \cdot (1 - \alpha) f_2(u, v) = 0$.
Since $f_1(u, v) \leq c(u, v)$ and $f2(u, v) \leq c(u, v)$, we also have
$\alpha f_1(u, v) + (1 - \alpha) f2(u, v) \leq \alpha c(u, v) + (1 - \alpha)c(u, v)$
$= (\alpha + (1 - \alpha))c(u, v) = c(u, v)$.
For skew symmetry, we have $f_1(u, v) = -f_1(v, u)$
and $f_2(u, v) = -f_2(v, u)$ for any $u, v \in V$.
Thus, we have
$\alpha f_1(u, v) + (1 - \alpha) f_2(u, v) = -\alpha f_1(v, u) - (1 - \alpha) f_2(v, u) = -(\alpha f_1(v, u) + (1 - \alpha) f_2(v, u))$
.

13. Write the Rabin-Karp algorithm and analysis the complexity.

[MODEL QUESTION]

Answer:
Algorithm:
The following procedure makes these ideas precise. The inputs to the procedure are the text T, the pattern P, the radix d to use (which is typically taken to be $|\Sigma|$), and the prime q to use.

RABIN-KARP-MATCHER (T, P, d, q)
1 $n \leftarrow length[T]$
2 $m \leftarrow length[P]$
3 $h \leftarrow d^{m-1} \bmod q$

```
4       p ← 0
5       t0 ← 0
6       for i ← 1 to m                          // Preprocessing.
7             do p ← (dp + P[i ]) mod q
8                t0 ← (dt0 + T [i ]) mod q
9       for s ← 0 to n − m                      //Matching.
10            do if p = ts
11                  then if P[1 . .m] = T [s + 1 . . s + m]
12                        then print "Pattern occurs with shift" s
13                if s < n − m
14                        then ts+1 ← (d(ts − T [s + 1]h) + T [s + m + 1]) mod q
```

Rabin and Karp have proposed a string-matching algorithm that performs well in practice and that also generalizes to other algorithms for related problems, such as two-dimensional pattern matching. The Rabin-Karp algorithm uses $\Theta(m)$ preprocessing time, and its worst-case running time is $\Theta((n-m+1)m)$.

TRACTABLE AND INTRACTABLE PROBLEMS

☞ Chapter at a Glance

- **P-NP Class:**
  Deterministic algorithm: An algorithm having a property that the result of every operation is uniquely defined is called a deterministic algorithm.
  Non-Deterministic Algorithm: An algorithm containing certain operation whose outcomes are not uniquely defined but are limited to specified set of possibilities, is called a non-deterministic algorithm.
  The **P** versus **NP** problem is to determine whether every language accepted by some nondeterministic algorithm in polynomial time is also accepted by some (deterministic) algorithm in polynomial time.
- **NP- Complete problem:**
  The NP-Complete problems are an interesting class of problems whose status is unknown. The properties of NP-Complete problem are:
  - No polynomial-time algorithm has been discovered for solving an NP-Complete problem
  - No polynomial lower bound in respect of time overhead has been proved for any NP-Complete problem.

Multiple Choice Type Questions

1. Which one is true of the following? **[WBUT 2008]**
a) All NP hard problems are NP compete
b) All NP complete problems are NP hard
c) Some NP complete problem are NP hard
d) none of these
Answer: (b)

2. Traveling salesman problem belongs to **[WBUT 2009]**
a) P class b) NP class c) NP- Hard d) NP-complete class
Answer: (d)

3. Consider the following statements: **[WBUT 2009]**
NP hard problem is a subset of NP complete problem.
Which of the following alternatives is true.
a) I- True, II-False b) Both True
c) Both False d) I- False, II- True
Answer: (d)

4. Which one is true of the following? **[WBUT 2010, 2013]**

a) All NP hard problems are NP complete
b) All NP complete problems are NP hard
c) Some NP complete problems are NP hard
d) None of these

Answer: (b)

5. Travelling Salesman Problem is **[WBUT 2012, 2013]**

a) NP Hard b) NP c) NP Complete d) none of these

Answer: (a)

6. Which one is true of the following? **[WBUT 2014]**

a) all NP hard problems are NP complete
b) all NP complete problems are NP hard
c) some NP complete problems are NP hard
d) none of these

Answer: (c)

Short Answer Type Questions

1. Define classes P, NP and NP complete. **[WBUT 2008]**

Answer:

Class P and Class NP:

Refer to Question No. 1(a) of Long Answer Type Questions.

Class NP complete:

Refer to Question No. 1(c) of Long Answer Type Questions.

2. Describe circuit satisfiability problem and prove that circuit-SAT is in NP. **[WBUT 2008]**

OR,

Define circuit satisfiability problem and prove that circuit SAT is in class NP. **[WBUT 2014]**

Answer:

Proving that circuit-SAT is in NP, The algorithm V() takes in input the description of a circuit C and a sequence of n Boolean values $x_1, x_2, \ldots, x_n$ and V (C, $x_1, \ldots, x_n$) = C($x_1, \ldots, x_n$). i.e. V simulates or evaluates the circuit.

Now we have to prove that for every decision problem A in NP, we can find a reduction from A to CSAT. This is a difficult result to prove, and it is impossible to prove it really formally without introducing the Turing machine model of computation. We will prove the result based on the following fact, of which we only give an informal proof.

Suppose A is a decision problem that is solvable in p(n) time by some program P, where n is the length of the input. Also assume that the input is represented as a sequence of bits.
Then, for every fixed n, there is a circuit C_n of size about
$O((p(n)^2).(\log p(n))^{O(1)})$
such that for every input $x = (x_1,\ldots, x_n)$ of length n, we have
$A(x) = C_n(x_1,\ldots, x_n)$
That is, circuit C_n solves problem A on all the inputs of length n.
Furthermore, there exists an efficient algorithm (running in time polynomial in p(n)) that on input n and the description of P produces C_n.

3. Describe Clique Decision Problem (CDP). [WBUT 2009, 2012]

Answer:

A ***Clique Decision Problem (CDP)*** in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E. In other words, a clique is a complete subgraph of G. The ***size*** of a clique is the number of vertices it contains. The ***clique problem*** is the optimization problem of finding a clique of maximum size in a graph. As a decision problem, we ask simply whether a clique of a given size k exists in the graph. The formal definition is
CLIQUE = $\{\langle G, k\rangle : G$ is a graph with a clique of size $k\}$.

A I algorithm for determining whether a graph $G = (V, E)$ with $|V|$ vertices has a clique of size k is to list all k-subsets of V, and check each one to see whether it forms a clique. The running time of this algorithm is $\Omega(k^2(^{|v|}C_k))$, which is polynomial if k is a constant. In general, however, k could be near $|V|$, in which case the algorithm runs in super-polynomial time. As one might suspect, an efficient algorithm for the clique problem is unlikely to exist. *K.*

4. What do you mean by non-deterministic algorithms? [WBUT 2009, 2012, 2016]
OR,
What is nondeterministic algorithm? Explain with example. [WBUT 2014]

Answer:

In the theory of computation, a **nondeterministic algorithm** is an algorithm with one or more choice points where multiple different continuations are possible, without any specification of which one will be taken. A particular execution of such an algorithm picks a choice whenever such a point is reached. Thus, different execution paths of the algorithm arise when it is applied to the same input / initial state, and these paths, when they terminate, generally produce different output / end in different final states.
It is not random, as in randomized algorithm, or indeterminate. Rather it has the super computational characteristic of choosing the optimal behavior. Conceptually, a nondeterministic algorithm could run on a deterministic computer with an unlimited number of parallel processors. Each time more than one step is possible, new processes are instantly forked to try all of them. When one successfully finishes, that result is returned. Thus, the computation is as fast as if it always chooses the right step.

This notion is defined for theoretical analysis. Nondeterministic algorithms compute the same class of functions as deterministic algorithms, but the complexity may be much less. Every nondeterministic algorithm can be turned into a deterministic algorithm, possibly with exponential slow down.
Such an algorithm has *three phases*

- ***Nondeterministic "guessing" phase*:** In this phase arbitrary string, ***s***, is written starting at some place in memory (***s*** may differ for each time it is run)
- ***Deterministic "verifying" phase*:** In this phase a normal algorithm will consider the input to the decision problem & ***s*** may return true/false.
- ***Output phase*:** It verifies phase outputs. If it is then output is true; else no output.

Example:
Merge Sort,
Spanning Tree

5. Show that 2SAT is in *P* but 3SAT is NP-complete. **[WBUT 2010]**

Answer:

2SAT is in *P*

To show that 2SAT $\in$ P, we outline a polynomial time algorithm for solving 2SAT.
The idea is to construct a directed graph G(F) = (V, E) with

(i) $V = \{x, \bar{x} \mid F \text{ contains the literal } x\}$

(ii) $E = \{(\bar{x}, y), (\bar{y}, x) \mid (x \vee y) \in F\} \subseteq (V \times V)$

We interpret the (directed) edges of G as implications. Because of the following equivalence

$$x \vee y \equiv (x \rightarrow y) \wedge (y \rightarrow x)$$

An assignment of F corresponds to an assignment of G. An assignment is satisfying iff no implication modeled through an edge is violated. G is satisfiable iff for no x there exists a path from x to x or from x to x in G. In other words: in order to be satisfying, x must not imply x, and x must not imply x.

Thus, the following algorithm solves 2SAT (with input size n = |F|):
1) Construct the graph G(F), which has O(n) vertices and edges.
2) For each variable x, search a path from x to $\bar{x}$. Effort: $O(n^2)$
3) If no such paths exist, return 'yes', otherwise return 'no'
This algorithm has complexity $O(n^2)$.

3SAT is NP-complete

A version of the SAT problem in which every clause has 3 literals is called 3SAT.
Let the given instance of SAT contain the collection $C = \{c_1, c_2, \ldots, c_m\}$ of clauses over the variables $U = \{u_1, u_2, \ldots, u_n\}$. We construct a collection C_0 of 3 literal clauses over the variables U_0 which consist of the original variables plus sets of additional variables as follows:

Replace each clause $c_i \in C$ by a collection of 3 literal clauses over the variables which appear in c_i plus some additional variables which appear only in these 3 literal clauses. The exact procedure depends on the number of literals in c_i.
Let $c_i \in C$ be given by $\{z_1, z_2, \ldots, z_k\}$ where the z_j's are literals over U.
If $k = 1$ $c_i = \{z_1\}$, use two additional variables $\{y_{i,1}, y_{i,2}\}$. Form the collection
$C_i' = \{\{z_1, y_{i,1}, y_{i,2}\}, \{z_1, y_{i,1}, y_{i,2}\}, \{z_1, y_{i,1}, y_{i,2}\}, \{z_1, y_{i,1}, y_{i,2}\}\}$
$k = 2$ $c_i = \{z_1, z_2\}$, use one additional variable $\{y_{i,1}\}$.
Form the collection $C_i' = \{\{z_1, z_2, y_{i,1}\}, \{z_1, z_2, y_{i,1}\}\}$
$k = 3$ $c_i = \{z_1, z_2, z_3\}$. No additional variables. $C_i' = c_i$.
$k > 3$ Use the additional variables $\{y_{i,1}, y_{i,2}, \ldots, y_{i,k-3}\}$. Form the collection
$C_i' = \{\{z_1, z_2, y_{i,1}\}, \{y_{i,1}, z_3, y_{i,2}\}, \{y_{i,2}, z_4, y_{i,3}\}, \{y_{i,3}, z_5, y_{i,4}\}, \ldots, \{y_{i,k-3}, z_{k-1}, z_k\}\}$
To show that this procedure forms a polynomial reduction, firstly we observe that the number of 3 literal clauses in C' is bounded by a polynomial in nm which leads to the procedure being a polynomial time one. Secondly, whenever C is satisfiable, then the C' clauses formed in the cases $k \leq 3$ are automatically satisfied (for arbitrary assignment of any additional variables) while in the case $k > 3$

(a) If z_1 or z_2 is true, assign all additional variables the truth value "false". In this case, the first literal in each relevant clause is true.
(b) If z_{k-1} or z_k is true, assign all additional variables the truth value "true". In this case, the third literal in each relevant clause is true.
(c) Otherwise, if z_l is true, assign $y_{i,j}$ the value "true" when $1 \leq j \leq l-2$ and the value "false" when $l-1 \leq j \leq k-3$. In this case, the third literal in each relevant clause preceding the one which includes z_l is true while the first literal in each relevant clause succeeding the one which includes z_l is true.

Thus all the clauses in C' are satisfied. Conversely, if all the clauses in C' are satisfied by an truth assignment to U' (which includes the original variables), then C is satisfied by the same truth assignment restricted to U.
Thus we conclude that C' is satisfied if and only if C is satisfied.

6. Show that 3-colouring problem is NP-complete. **[WBUT 2010]**

Answer:

(1) 3-Coloring is in NP because for each node a color from (1, 2, 3). Check if for each edge (u, v), the color of *u* is different from that of *v*
(2) Reduce 3-SAT to 3-COLORING.
(3) Given a 3-SAT formula of m clauses on n variables $x_1, x_2, \ldots, x_n$,

We construct a graph G as follows. We have
(a) a vertex v_i for each variable x_i,
(b) a vertex v_i' for the negation of each variable x_i,
(c) 5 vertices j_1-j_5 for each clause j,

(d) 3 special vertices: T, F, R

We would like T, F, and R to be forced to different colors, so we will add edges between them to form a triangle. For the remaining nodes, and node that is colored the same color as T/F/R will be called colored TRUE/FALSE/RED, respectively.

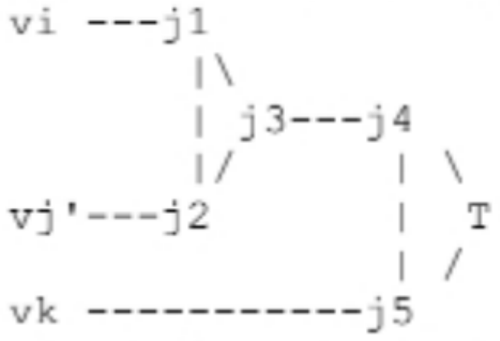
We would like the edges to enforce the constraints on satisfying assignments.

Constraint: For all i, exactly one of v_i and v_i' is colored TRUE and one is colored FALSE.

Edges: for each i, form a triangle between v_i, v_i', and R.

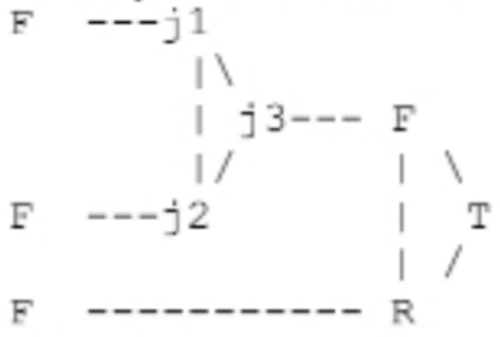
Constraint: For each clause j, at least one of the literals in the clause is colored TRUE.

Edges: for each clause j, say = (x_i or not(x_j) or x_k), we have the following gadget.



Claim: If each of v_i, v_j', and v_k is colored TRUE or FALSE, then gadget is 3-colorable iff at least one of v_i, v_j', and v_k is colored TRUE.

Proof: If v_i, v_j', and v_k are all colored false, then we are forced to the following colors:



But then j_1, j_2, j_3 all must be colored different colors and NONE can be colored F, so there is no legal coloring.

The remainder of the proof considers the 7 possible combinations of coloring v_i, v_j', and v_k such that at least one is colored TRUE and the rest are colored FALSE, and shows that a_3 coloring exists in each case.

As an example, if v_k is colored TRUE but v_i and v_j' are colored FALSE, we have the following legal 3-coloring:

The other cases are similar and were presented in class. The construction takes polynomial time. Follows from the above arguments thus 3-COLORING is NP-complete.

7. State satisfiability problem. State Cook's theorem in connection with P and NP problem. **[WBUT 2015]**

Answer:

1st part:

SATISFIABILITY, or SAT, is a problem of great practical importance, with applications ranging from chip testing and computer design to image analysis and software engineering. It is also a canonical hard problem. Here's what an instance of SAT looks like:

$(x \vee y \vee z)(x \vee \bar{y})(y \vee \bar{z})(z \vee \bar{x})(\bar{x} \vee \bar{y} \vee \bar{z})$

This is a *Boolean formula in conjunctive normal form (CNF)*. It is a collection of *clauses* (the parentheses), each consisting of the disjunction (logical *or*, denoted $\vee$) of several *literals*, where a literal is either a Boolean variable (such as x) or the negation of one (such as $\bar{x}$).

A *satisfying truth assignment* is an assignment of false or true to each variable so that every clause contains a literal whose value is true. The SAT problem is the following: given a Boolean formula in conjunctive normal form, either find a satisfying truth assignment or else report that none exists.

2nd part:

The *Cook's Theorem* gives a proof that the problem SAT is NP-Complete, via the technique of showing that any problem in NP may be reduced to it.

Before proving the theorem, we give a formal definition of the SAT problem (Satisfiability Problem): Given a Boolean formula φ in CNF (Conjuctive Normal Form), is the formula satisfiable? In other words, we are given some Boolean formula φ with n Boolean variables x1, x2, . . . , xn, $\varphi = C1 \wedge C2 \wedge \cdots \wedge Ck$, where each of the Ci is a clause of the form $(t_{i1} \vee t_{i2} \vee \ldots \vee t_{il})$, with each l_{ij} is a literal drawn from the set $\{x_1, x_2, \ldots, x_n, \bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\}$. We need to decide whether or not the exists some setting of the $x_1, x_2, \ldots, x_n$ that cause the formula φ to be satisfied (take on a Boolean 1 value of true).

Cook-Levin Theorem: SAT is NP-complete.

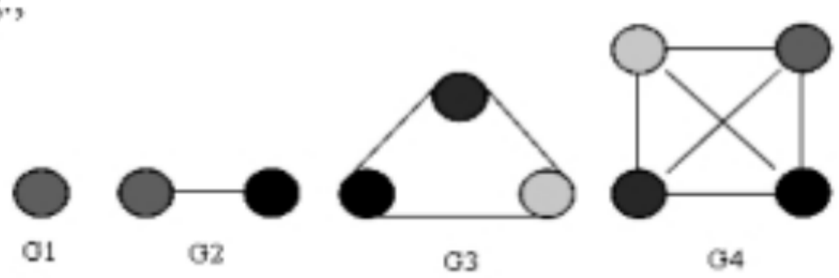
Statement: Suppose L is a NP problem, then L has a polynomial time verifier V :

1. If $x \in L$, $\exists$ witness y, V (x, y) = 1
2. If $x \notin L$, $\forall$ witness y, V (x, y) = 0

We can build a circuit with polynomial size for the verifier V, since the verifier runs in polynomial time (note that this fact is nontrivial; however, it is left to the reader to verify that it is true). The circuit contains AND, OR and NOT gates. The circuit has |x| + |y| sources, where |x| of them are hardcoded to the values of the bits in x and the rest |y| are variables.

8. What do you mean by chromatic number of a graph? **[WBUT 2017]**

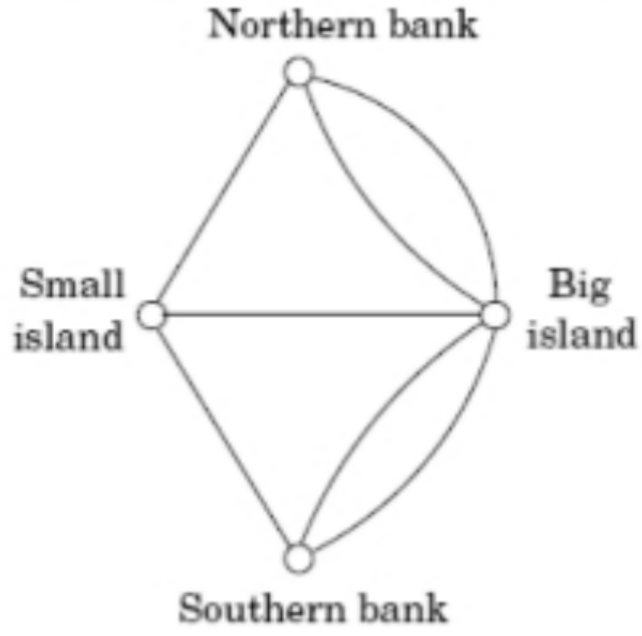
Answer:
The least number of color needed to color a graph is called its chromatic number. The chromatic number, say X of a complete graph G is X(G) = d+1, where d is the degree of the vertex. So if we consider a graph G is a complete graph, i.e., each node is connected to all other nodes of the graph then total number of colors required to color all the nodes of the graph is d+1.e.g.,



In the above figure each graph like G1, G2, G3 or G4 requires (d+1) colors to color all the nodes, where d is the degree of each node.

9. Can we define Euler graph with SAT problem? **[MODEL QUESTION]**

Answer:
In the summer of 1735 Leonhard Euler, the famous Swiss mathematician, was walking across the bridges of the East Prussian town of Konigsberg. After a while, he noticed in frustration that, no matter where he started his walk from, no matter how cleverly he continued, it was impossible to cross each bridge exactly once. And from this silly ambition, the field of graph theory was born. Euler identified at once the map of the park into a graph whose vertices are the four land masses (two islands, two banks) and whose edges are the seven bridges:



This graph has multiple edges between two vertices. A feature we have not been allowing so far in our discussion, but one that is meaningful for this particular problem, since each bridge must be accounted for separately. We are looking for a path that goes through each edge exactly once (the path is allowed to repeat vertices). In other words, we are asking this question:
When can a graph be drawn without lifting the pencil from the paper?
The answer discovered by Euler is simple, elegant, and intuitive: If and only if (a) the graph is connected and (b) every vertex, with the possible exception of two vertices (the

start and final vertices of the walk), has even degree. This is why Konigsberg's park was impossible to traverse: all four vertices have odd degree.
To put it in terms of our present concerns, let us define a search problem called EULER PATH: Given a graph, find a path that contains each edge exactly once. It follows from Euler's observation, and a little more thinking, that this search problem can be solved in polynomial time.

Long Answer Type Questions

1. a) Define the classes P and NP. **[WBUT 2005, 2009, 2016]**

Answer:

Class P:

Let A be an algorithm to solve a problem X. We say that A is deterministic if, when presented with an instance of the problem X, it has only one choice in each step throughout its execution. Thus, if A is run again and again on the same input instance, its output never changes.
The class of decision problem P consists of those decision problems whose solution can be obtained using a deterministic algorithm that runs in polynomial number of steps, i.e., in $O(n^k)$ steps, for some nonnegative integer k, where n is the input size.

Class NP:

The class NP consists of those problems for which there exists a deterministic algorithm A, which presents with a claimed solution to an instance of the problem. This is able to verify its correctness in polynomial time. That is, if the claimed solution leads to a yes answer, there is a way to verify this solution in polynomial time. For example, in the hamiltonian-cycle problem, given a directed graph $G = (V, E)$, a certificate would be a sequence $\langle v_1, v_2, v_3, ..., v_{|V|} \rangle$ of $|V|$ vertices. It is easy to check in polynomial time that $(v_i, v_{i+1}) \in E$ for $i = 1, 2, 3, ..., |V| - 1$ and that $(v_{|V|}, v_1) \in E$ as well. Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time without even being given a certificate. We will formalize this notion later in this chapter, but for now we can believe that $P \in NP$. The open question is whether or not P is a proper subset of NP.

b) Discuss what you mean by Polynomial Reductions. **[WBUT 2005, 2009, 2016]**

Answer:

The root of NP-Completeness is reducibility. Informally, a problem P can be reduced to another problem Q if any instance of P can be "easily rephrased" as an instance of Q, the solution to which provides a solution to the instance of P. This rephrasing is called transformation. If P reduces to Q, P is "no harder to solve" than Q.

Suppose,
P1 : is an unknown problem (easy/hard ?)
P2 : is known to be easy

If we can reduce P1 to P2, that is, we can easily solve P1 using
P2 as a subroutine then P2 is easy
One of the example of reduction is that solving linear equations is reducible to solving quadratic equations.

e.g.,

P: Given a set of Booleans, is at least one TRUE.
Q: Given a set of integers.
Transformation: $(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_n)$ where $y_i = 1$ if x_i = TRUE, $y_i = 0$ if x_i = FALSE

c) Discuss diagrammatically the relations among P class, NP class, NP-hard and NP-complete. [WBUT 2005, 2009]

OR,

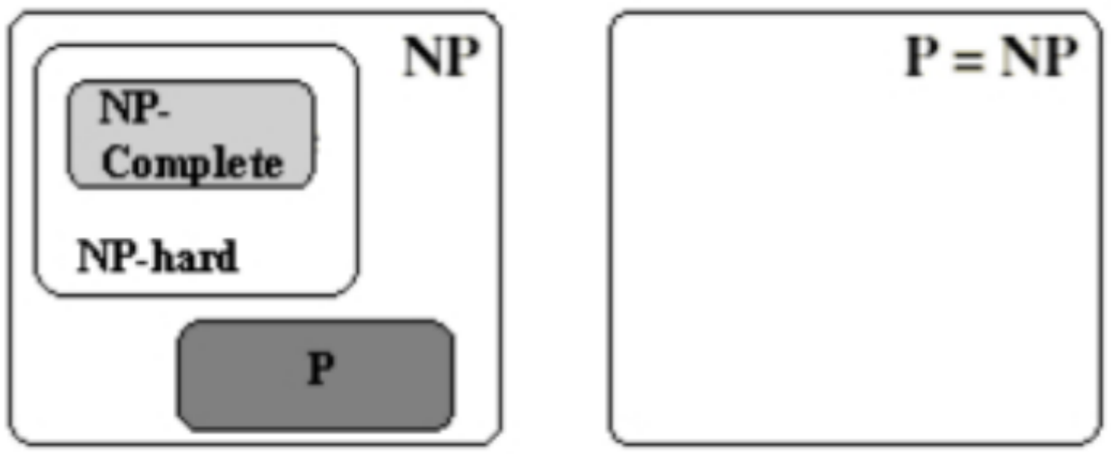
Define classes P, NP, NP hard and NP-complete and also explain their relationship diagrammatically. [WBUT 2014]

OR,

Define P-class, NP-class, NP-complete class and NP-hard class of problems. What is the relation between them? [WBUT 2015, 2016]

Answer:

- We say that a decision problem Pi is NP-complete if it is NP-hard and
  - it is also in the class NP itself.
  - in symbols, Pi is NP-complete if Pi is NP-hard and Pi $\in$ NP.
- Highly informally, it means that Pi is one of the hardest problems in NP.
- So the NP-complete problems form a set of problems that may or may not be intractable but, whether intractable or not, are all, in some sense, of equivalent complexity.
- If anyone ever shows that an NP-complete problem is tractable, then
  - Every NP-complete problem is also tractable indeed,
  - Every problem in NP is tractable and so P = NP.
- If anyone ever shows that an NP-complete problem is intractable, then
  - Every NP-complete problem is also intractable and, of course, P $\neq$ NP.
  - So there are two possibilities as shown in figure:



- We don't know which the case is, but this gives Computer Scientists a clear line of attack. It makes sense to focus efforts on the NP-complete problems; they all stand or fall together.

- So these sound like very significant problems in our theory.

2. Prove that CDP (Clique Decision Problem) is NP-Complete.

[WBUT 2006, 2009, 2012]

Answer:

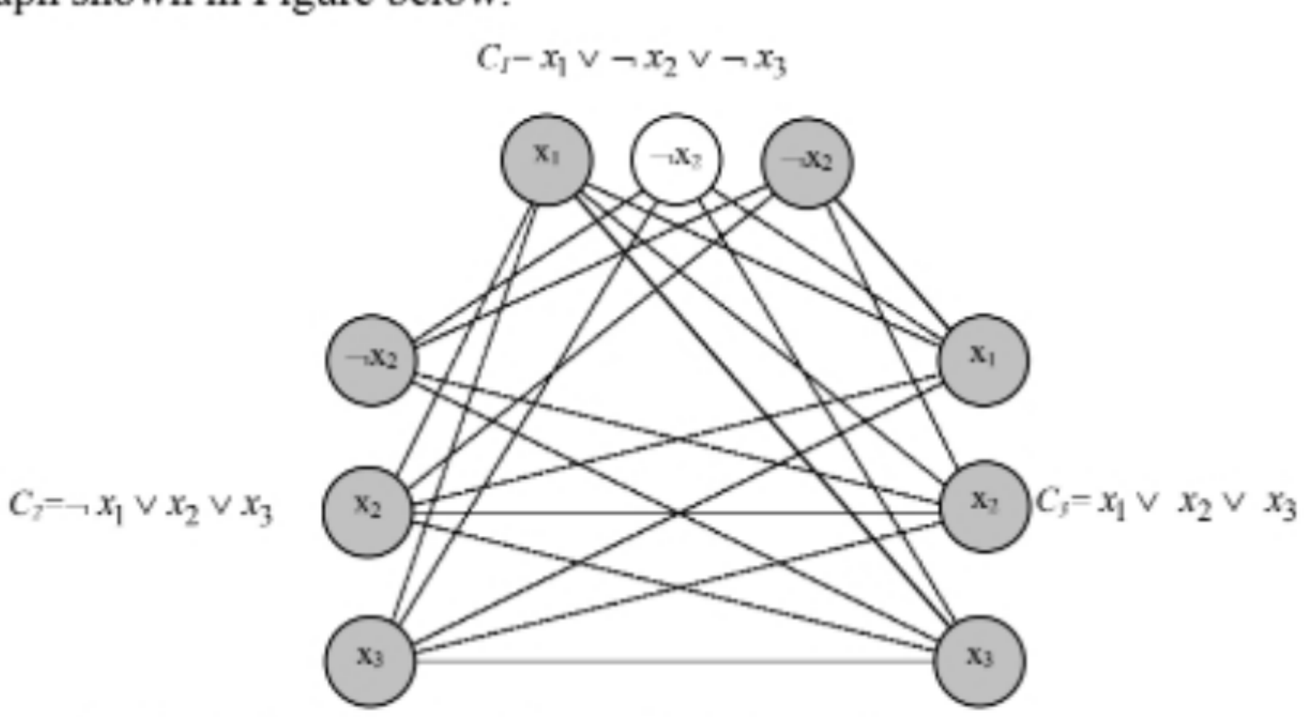
To show that CLIQUE $\in$ NP, for a given graph $G = (V, E)$, we use the set $V' \subseteq V$ of vertices in the clique as a certificate for G. Checking whether V' is a clique can be accomplished in polynomial time by checking whether, for each pair $u, v \in V'$, the edge (u, v) belongs to E.

We next prove that 3-CNF-SAT $\leq_P$ CLIQUE, which shows that the clique problem is NP-hard. The reduction algorithm begins with an instance of 3-CNF-SAT. Let $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be a Boolean formula in 3-CNF with k clauses. For $r = 1, 2, \ldots, k$, each clause C_r has exactly three distinct literals l_1^r, l_2^r, and l_3^r We shall construct a graph G such that φ is satisfiable if and only if G has a clique of size k.

The graph $G = (V, E)$ is constructed as follows. For each clause $(l_1^r \vee l_2^r \vee l_3^r)$ in φ, we place a triple of vertices v_1^r, v_2^r, and v_3^r into V. We put an edge between two vertices v_i^r and v_j^s if both of the following hold:

- v_i^r and v_j^s are in different triples, that is, $r \neq s$, and
- their corresponding literals are ***consistent***, that is, l_i^r is not the negation of l_j^s.

This graph can easily be computed from φ in polynomial time. As an example of this construction, if we have $\varphi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$, then G is the graph shown in Figure below.



The graph G derived from the 3-CNF formula $\varphi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$ $C_2 = (\neg x_1 \vee x_2 \vee x_3)$ and $C_3 = (x_1 \vee x_2 \vee x_3)$, in reducing 3-CNF-SAT to CLIQUE. A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and x_1 may be either 0 or 1. This assignment satisfies C_1 with $\neg x_2$, and it satisfies C_2 and C_3 with x_3, corresponding to the clique with lightly shaded vertices.

We must show that this transformation of φ into G is a reduction. First, suppose that φ has a satisfying assignment. Then each clause C_r contains at least one literal l_i^r that is

assigned 1, and each such literal corresponds to a vertex v_i^r. Picking one such "true" literal from each clause yields a set V' of k vertices. We claim that V' is a clique. For any two vertices $v_i^r, v_j^s \in V'$, where $r \neq s$, both corresponding literals are mapped to 1 by the given satisfying assignment, and thus the literals cannot be complements. Thus, by the construction of G, the edge (v_i^r, v_j^s) belongs to E.

Conversely, suppose that G has a clique V' of size k. No edges in G connect vertices in the same triple, and so V' contains exactly one vertex per triple. We can assign 1 to each literal l_i^r such that $v_i^r \in V'$ without fear of assigning 1 to both a literal and its complement, since G contains no edges between inconsistent literals. Each clause is satisfied, and so φ is satisfied.

3. a) What is Non-deterministic algorithm? Differentiate between Deterministic and Non-Deterministic algorithm.

b) Write algorithm to sort an array using Deterministic and Non-Deterministic technique. Compare the two techniques and show that the time complexity of non-deterministic technique is better than Deterministic.

c) Describe *P* class, NP class, NP hard and NP complete class and describe their relationships. **[WBUT 2011]**

Answer:

a) ***Refer to Question No. 4 of Short Answer Type Questions.***

b) There are many deterministic algorithm for sorting an array, quicksort is one of them. The following procedures together form the quick sort algorithm.

```
QuickSort(A, p, r)
{
  if p < r then
  {
    q ← Partition (A, p, r);
    Quick Sort (A, p, q);
    Quick Sort (A, q + 1, r);
  }
}
```

Algorithm for partition

Partition (A, p, r)

```
{
  initialize x ← A[p];
  i ← p-1, j ← r+1;
  while (i < j)
  {
    while (A[--j] > x)
    {
  while (A[++i] < x)
  {
    if (i < j) then
```

```
                Exchange (A[i], A[j]);
        }
      }
    }
  return j;
}
```

Exchange(A, I, j)

```
{
 p ← A[i];
 A[i] ← A[j];
 A[j] ← p;
}
```

Partition selects the first key x ← *A*[*p*] as a pivot key about which the array is partitioned. If x ≤ A[*p*], *A[p]* will move towards the left of the pivot, otherwise if x ≥ *A*[*p*], *A*[*p*] will move towards the right of the pivot key x.

Complexity Analysis

- ***Best case***
  - In the best case, the pivot is in the middle position of the array.
  - To simplify the equations, we assume that the two sub arrays are each exactly half the length of the original one. So, we get $T(n) = 2T(n/2) + cn$, $c>0$ constant (independent of n) and $n \geq 2$ with $T(1) = 1$.
  - This is very similar to the formula for Merge sort, and a similar analysis leads to $T(n) = cn \log_2 n + n$ which is $O(n \log_2 n)$.

- ***Average case:*** We assume that each of the sizes of the left partitions are equally likely, and hence each has probability 1/n.
  - With this assumption, the average value of T(i), and hence also of T(n–i–1), is (T(0) + T(1) + … + T(n–1))/n
  - Naturally, our recurrence relation becomes T(n) = 2(T(0) + T(1) + … + T(n–1))/n + cn
  - Multiplying both sides by n we find $nT(n) = 2(T(0) + T(1) + \ldots + T(n-1)) + cn^2$
  - Substitution of n by n–1 gives $(n-1)T(n-1) = 2(T(0) + T(1) + \ldots + T(n-2)) + c(n-1)^2$
  - Subtracting the last equation from the previous one, we get nT(n) – (n–1)T(n–1) = 2T(n–1) + 2cn – c
  - Rearranging and ignoring constant c, we arrive at nT(n) = (n+1)T(n–1) + 2cn
  - Division through out by n(n+1) gives T(n)/(n+1) = T(n–1)/n + 2c/(n+1)
  - Hence, T(n–1)/n = T(n–2)/(n–1) + 2c/n
    Similarly T(2)/3 = T(1)/2 + 2c/3
  - Thus
    T(n)/(n+1) = T(1)/2 + 2c(1/3 + ¼ + … + 1/(n+1))

- The sum in brackets is about $\log_e(n+1) + \gamma - 3/2$, where γ is Euler's constant, which is approximately 0.577.
- So, $T(n)/(n+1) = O(\log_2 n)$ and thus $T(n) = O(n \log_2 n)$

- ***Worst case:*** In quick sort technique, the worst case condition arises when the elements of the array are already sorted.
  - If the pivot is always the smallest element, then always $I = 0$
  - We ignore the term $T(0)=1$, so the recurrence relation is $T(n) = T(n-1) + cn$
  - So, $T(n-1)=T(n-2)+c(n-1)$ and so on until we get $T(2)=T(1)+c(2)$
  - Substituting backwards, we get $T(n) = T(1) + c(n + \ldots + 2) = O(n^2)$

It may be noted that this case happens if we always take the pivot to be the first element in the array and the array is already sorted.

Let A[i], $1 \le i \le n$, be an unsorted array of positive integers. The nondeterministic algorithm NSort(A, n) sorts the number into nondecreasing order and then outputs them in this order. An auxiliary array B[1:n] is used for conbience. Line 4 initializes B to zero though any value different from all the A[i] will do. In the for loop of lines 5 to 10, each A[i] is assigned to a position in B. Line 7 nondeterministically determines this position. Line 8 ascertaines that B[j] has not already been used. Thus, the order of the numbers in B is some permutation of the initial order in A. The for loop of lines 11 and 12 verifies that B is sorted in nondecreasing order. A successful completion is achieved if and only if the numbers ar coutput in nondecreasing order. Since there is always a set of choices at line 7 for such an output order, algorithm NSort is a sorting algorithm. Its complexity is O(n). All deterministic sorting algorithms must have a complecity $\Omega(n \log n)$.

```
1   Algorithm NSort(A, n)
2   // Sort n positive integers
3   {
4         for i: = 1 to n do B[i]: = 0, // Initialize B[].
5         for i: = 1 to n do
6        {
7                 j:= Choice(1, n);
8                 if B[j] ≠0then Failure();
9                 B[j]: = A[i];
10       }
11       For i = 1 to n -1 do // Verify order.
12               If B[i]>B[i+1]then Failure();
13      Write ([B1:n]);
14      Success}
15   }
```

c) *Refer to Question No. 1 of Long Answer Type Questions.*

4. What do you mean by deterministic and non-deterministic algorithm? Write a non-deterministic algorithm for searching an element from a given list of real numbers. Also, specify its time complexity. [WBUT 2015]

Answer:

1st part:

The algorithm where result is uniquely defined is a termed the deterministic algorithms. Such algorithms agree with the way programs are executed on a computer. In a theoretical framework we can remove this restriction on the outcome of every operation. We can allow algorithms to contain operations whose outcomes are not uniquely defined but are limited to specified sets of possibilities.

The machine executing such operations is allowed to choose anyone of these outcomes subject to a termination condition.
This leads to the concepts of a nondeterministic algorithm.
To specify such algorithms, we introduce three new functions:
(a) Choose (s) arbitrarily chooses one of the elements of set s.
(b) Failure ( ) signals an unsuccessful completion.
(c) Success ( ) signals a successful completion.
The assignment statement x: = choice (l, n) could result in x being assigned anyone of the integers in the range (l, n) there is no rule specifying how this choice is to be made. The failure*1 and success () signals are used to define a computation of the algorithm. These statements cannot be used to effect a return. Whenever there is a set of choices that leads to a successful completion. Then one such set of choices is always made and the algorithm terminates successfully.

2nd part:
A non-deterministic algorithm for searching an element from a given list of real numbers:
Let *A[1..n]* be an array and *x* be the target element to search for.
We can write the following nondeterministic search algorithm:

```
N_SEARCH(A, x)
{
        j = choice(1 . . . n)
        if A[j] = x then
              print(j)
              success
        endif
        print('0')
        failure
}
```

Time complexity: O(n).

5. Write short notes on the following:
a) Clique decision problem **[WBUT 2003, 2015]**
b) NP-hard and NP-complete classes **[WBUT 2017]**
c) Importance of P, NP classes **[MODEL QUESTION]**

Answer:

a) Clique decision problem:

Refer to Question No. 3 of Short Answer Type Questions.

b) NP-hard and NP-complete classes:

Refer to Question No. 1(c) of Long Answer Type Questions.

c) Importance of P, NP classes:

The importance of the **P** vs **NP** questions stems from the successful theories of **NP**-completeness and complexity-based cryptography, as well as the potentially stunning practical consequences of a constructive proof of **P** =**NP**.

The theory of **NP**-completeness has its roots in computability theory. The computability precursors of the classes **P** and **NP** are the classes of decidable and c.e. (computably enumerable) languages, respectively. We say that a language L is c.e. (or semi-decidable) if and only if L = L (M) for some Turing machine M. We say that L is decidable if and only if L = L (M) for some Turing machine M which satisfies the condition that M halts on all input strings w. There is an equivalent definition of c.e. which brings out its analogy with **NP**, namely L is c.e. if and only if there is a computable "checking relation" R(x, y) such that $L = \{x \mid \exists y\, R(x, y)\}$.

Using the notation $\langle M \rangle$ to denote a string describing a Turing machine M, we define the Halting Problem HP as follows:

$HP = \{\langle M \rangle \mid M \text{ is a Turing machine which halts on input } \langle M \rangle\}$

Turing used a simple diagonal argument to show that HP is not decidable. On the other hand, it is not hard to show that HP is c.e. Of central importance in computability theory is the notion of reducibility.

APPROXIMATION ALGORITHMS

☞ Chapter at a Glance

- **Approximation Algorithm:**
  Suppose you need to solve NP-hard problem X. So we have to find a polynomial algorithm to solve it. So, we must sacrifice one of three desired features:
  - Find out exact optimal solution.
  - Solve the problem in polynomial time to ensure time feasibility.
  - Provide solutions to arbitrary instances of the problem.

  Now applying approximation algorithm, the following issues are achieved:
  - Guaranteed polynomial time execution.
  - Guaranteed to find "high quality" near optimal solution within feasible time.

  But there is an additional responsibly that we need to prove how close the value of the solution is to the optimum, without even knowing what optimum value is.
- **Examples of approximation algorithms for NP-Complete problem**
  Now we discuss about the following examples of approximation algorithms for **NP-Complete** problems
  - Approximation ratio
  - Polynomial-Time Approximation Schemes
  - 2-Approximation for Vertex Cover
  - 2-Approximation for TSP special case
  - log n-Approximation for Set Cover

Long Answer Type Questions

1. Write short notes on the following:
a) Approximation algorithms **[WBUT 2008, 2012, 2014]**
b) Approximation schemes **[WBUT 2009, 2016]**
c) Vertex Cover Algorithm **[WBUT 2014, 2016]**
d) Randomized algorithms **[MODEL QUESTION]**
e) NP Complete problem **[MODEL QUESTION]**

Answer:

a) Approximation algorithms:

If a problem is NP-complete, we are unlikely to find a polynomial-time algorithm for solving it. There are at least three approaches to getting around NP-completeness. First, if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory. Second, we may be able to isolate important special cases that are solvable in polynomial time. Third, it may still be possible to find *near-optimal* solutions in

polynomial time (either in the worst case or on average). In practice, near-optimality is often good enough. An algorithm that returns near-optimal solutions is called an ***approximation algorithm***.

Suppose that we are working on an optimization problem in which each potential solution has a positive cost, and we wish to find a near-optimal solution. Depending on the problem, an optimal solution may be defined as one with maximum possible cost or one with minimum possible cost; that is, the problem may be either maximization or a minimization problem.

We say that an algorithm for a problem has an ***approximation ratio*** of $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

We also call an algorithm that achieves an approximation ratio of $\rho(n)$ a ρ(n)-approximation algorithm. The definitions of approximation ratio and of $\rho(n)$-approximation algorithm apply for both minimization and maximization problems. For a maximization problem, $0 < C \leq C^*$, and the ratio C^*/C gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution. Similarly, for a minimization problem, $0 < C^* \leq C$, and the ratio C/C^* gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution. Since all solutions are assumed to have positive cost, these ratios are always well defined. The approximation ratio of an approximation algorithm is never less than 1, since C/C^* 003C; 1 implies $C^*/C > 1$. Therefore, a 1-approximation algorithm produces an optimal solution, and an approximation algorithm with a large approximation ratio may return a solution that is much worse than optimal.

b) Approximation schemes:

Now applying approximation algorithm, the following issues are achieved:

- Guaranteed polynomial time execution.
- Guaranteed to find "high quality" near optimal solution within feasible time.

But there is an additional responsibly that we need to prove how close the value of the solution is to the optimum, without even knowing what optimum value is.

Different types of approximation algorithms and schemes

There are different types of approximation algorithms and schemes described below:

ρ-approximation algorithm:

- An algorithm A for problem P that runs in polynomial time.
- For every problem instance, A outputs a feasible solution within ratio P of true optimum for that instance.

Polynomial-time approximation scheme (PTAS):

- A family of approximation algorithms $\{A_\varepsilon: \varepsilon > 0\}$ for a problem P.
- A_ε is a $(1 + \varepsilon)$ - approximation algorithm for P.

- A_ε is runs in time polynomial in input size for a fixed ε.

Fully polynomial-time approximation scheme (FPTAS):
PTAS where A_ε is runs in time polynomial both in input size and 1/ε.

c) Vertex Cover Algorithm:
Given a G = (V, E), find a minimum subset C ⊆ V , such that C "covers" all edges in E, i.e., every edge ∈ E is incident to at least one vertex in C.

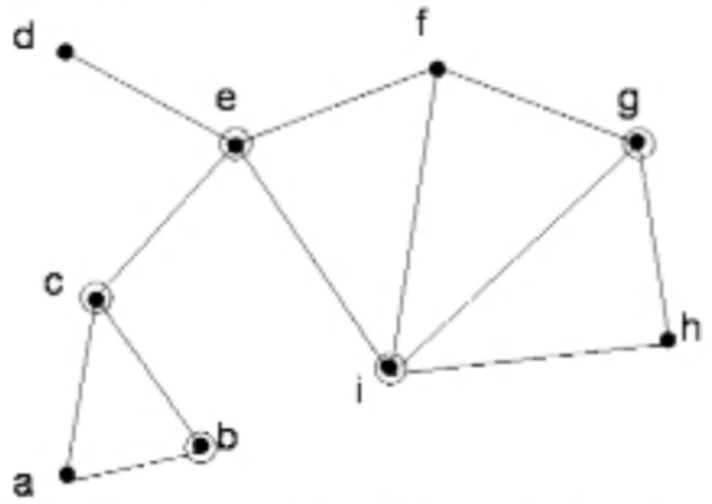


Fig.: An instance of Vertex Cover problem. An optimal vertex cover is {b, c, e, i, g}.

Algorithm 1: Approx-Vertex-Cover (G)

```
{
1 C ← φ
2 while E ≠ φ;
      pick any {u, v} ∈ E
      C  ← C ⋃ {u, v}
      delete all edges incident to either u or v
return C
}
```

As it turns out, this is the best approximation algorithm known for vertex cover. It is an open problem to either do better or prove that this is a lower bound.

d) Randomized algorithms:
A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic. The algorithm typically uses uniformly random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random bits. An algorithm that uses the random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm. For example, in Randomized Quick Sort, we use random number to pick the next pivot (or we randomly shuffle the array).

Analysis of Randomized Algorithm:
Some randomized algorithms have deterministic time complexity. For example, this implementation of Karger's algorithm has time complexity as O(E). Such algorithms are called Monte Carlo Algorithms and are easier to analyse for worst case. On the other hand, time complexity of other randomized algorithms (other than Las Vegas) is

dependent on value of random variable. Such Randomized algorithms are called Las Vegas Algorithms. These algorithms are typically analysed for expected worst case. To compute expected time taken in worst case, all possible values of the used random variable needs to be considered in worst case and time taken by every possible value needs to be evaluated. Average of all evaluated times is the expected worst case time complexity.

Examples:

For example consider below a randomized version of QuickSort.

/ Sorts an array arr[low..high]

randQuickSort(arr[], low, high)

1. If low >= high, then EXIT.

2. While pivot 'x' is not a Central Pivot.

(i) Choose uniformly at random a number from [low..high].
Let the randomly picked number number be **x**.

(ii) Count elements in arr[low..high] that are smaller than arr[x]. Let this count be **sc**.

(iii) Count elements in arr[low..high] that are greater than arr[x]. Let this count be **gc**.

(iv) Let **n** = (high-low+1). If sc >= n/4 and gc >= n/4, then x is a central pivot.

3. Partition arr[low..high] around the pivot x.

4. // Recur for smaller elements
randQuickSort(arr, low, sc-1)

5. // Recur for greater elements
randQuickSort(arr, high-gc+1, high)

The important thing in our analysis is, time taken by step 2 is O(n).

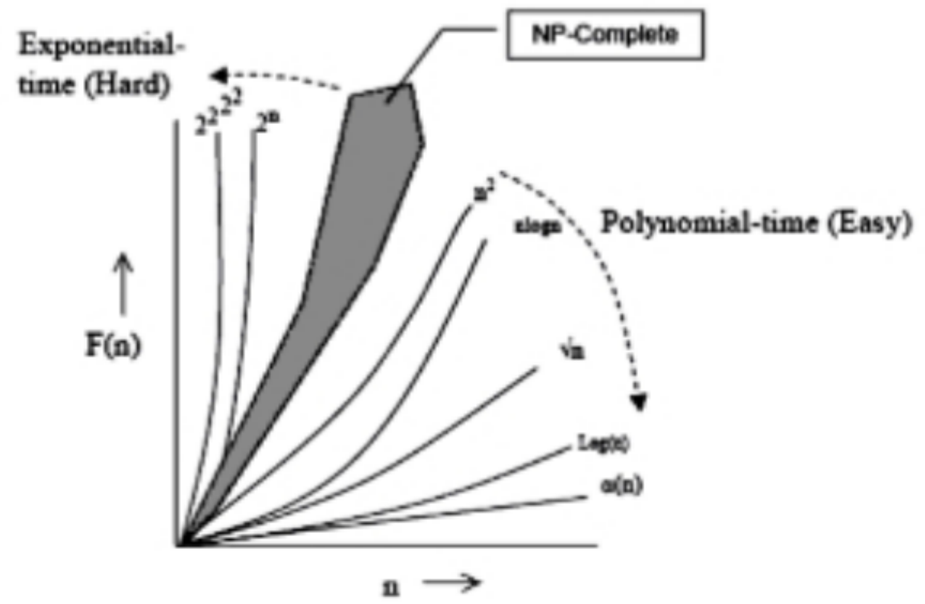
e) NP Complete problem:

The NP-Complete problems are an interesting class of problems whose status is unknown. The properties of NP-Complete problem are-

- No polynomial-time algorithm has been discovered for solving an NP-Complete problem
- No polynomial lower bound in respect of time overhead has been proved for any NP-Complete problem.

In figure 1 we have shown the complexity of algorithms.

POPULAR PUBLICATIONS



NP class

The notation **NP** stands for "nondeterministic polynomial time", since originally **NP** was defined in terms of nondeterministic machines (that is, machines that have more than one possible move from a given configuration). However now it is customary to give an equivalent definition using the notion of a checking relation, which is simply a binary relation $R \subseteq \Sigma^* \times \Sigma_1^*$ for some finite alphabets Σ and Σ_1. We associate with each such relation R a language L_R over $\Sigma \cup \Sigma_1 \cup \{\#\}$ defined by

$$L_R = \{ w \# y \mid R(w, y)\}$$

Where the symbol # is not in Σ. We say that *R* is *polynomial-time* if and only if $L_R \in \mathbf{P}$.

Now we define the class **NP** of languages by the condition that a language *L* over Σ is in **NP** if and only if there is $k \in N$ (N is the set of natural number) and a polynomial-time checking relation *R* such that for all $w \in \Sigma^*$,

$$w \in L \Leftrightarrow \exists\, y(|y| \leq |w|^k \text{ and } R(w, y))$$

Where $|w|$ and $|y|$ denote the lengths of *w* and *y*, respectively.

For example, in the Hamiltonian-cycle problem, given a directed graph $G = (V, E)$, a certificate would be a sequence $(v_1, v_2, v_3, ..., v_{|V|})$ of $|V|$ vertices. It is easy to check in polynomial time that $(v_i, v_{i+1}) \in E$ for $i = 1, 2, 3, ..., |V| - 1$ and that $(v_{|V|}, v_1) \in E$ as well.

Is P = NP?

It is easy to see that the answer is independent of the size of the alphabet Σ.(we assume $|\Sigma| \geq 2$), since strings over an alphabet of any fixed size can be efficiently coded by strings over a binary alphabet. (For $|\Sigma| = 1$ the problem is still open, although it is possible that **P** = **NP** in this case but not in the general case).

It is trivial to show that $\mathbf{P} \subseteq \mathbf{NP}$, since for each language *L* over Σ, if $L \in \mathbf{P}$ then we can define the polynomial-time checking relation $R \subseteq \Sigma^* \cup \Sigma^*$ by

$R(w, y) \Leftrightarrow w \in L$, for all $w, y \in \Sigma^*$.

Here are two simple examples, using decimal notation to code natural numbers:

The set of perfect squares is in **P** and the set of composite numbers is in **NP** (and not known to be in **P**). For the latter, the associated polynomial time checking relation *R* is given by

$R(\bar{a}, \bar{b}) \Leftrightarrow 1 < b < a$ and $b \mid a$

In general the decimal notation for a natural number c is denoted by c.

MISCELLANEOUS

Multiple Choice Type Questions

1. Which of the following is solved by using Branch and Bound method?

a) Knapsack Problem **b) Hamiltonian Problem [WBUT 2015]**

c) Travelling Salesman Problem **d) 15-Puzzle Problem**

Answer: (d)

2. By applying Strassen's algorithm we can multiply two $n \times n$ matrices in [WBUT 2017]

a) $O(n^3)$ **time** **b)** $O(n)$ **time** **c)** $O(n^n)$ **time** **d)** $O(n^{\log 7})$ **time**

Answer: (d)

Short Answer Type Questions

1. Prove that if $f(n) = a_m n^m + a_{m-1} n^{n-1} + ... + a_1 n + a_0$, then $f(n) = O(n^m)$.

[WBUT 2007]

Answer:

Let, $p(n) = a_m n^m + a_{m-1} n^{m-1} + ... + a_2 n^2 + a_1 n + a_0$

$$\leq \sum_{i=0}^{m} |a_i| n^i$$

$$\leq n^m \sum_{i=0}^{m} |a_i| n^{i-m}$$

$$\leq n^m \sum_{i=0}^{m} |a_i| \qquad \text{where, } n \geq 1$$

So, $p(n) \leq n^m \sum_{i=0}^{m} |a_i| = n^m (a_0 + a_1 + ... + a_m) = n^m . A$

Where, $A = (a_0 + a_1 + ... + a_m)$

Now, $p(n) \leq A.\ n^m$, which shows that $p(n) = O(n^m)$

2. Write down DFT algorithm and explain its method of execution. [WBUT 2008]

Answer:

We wish to evaluate a polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$ of degree-bound n at $\omega_n^o, \omega_n^1, \ldots, \omega_n^{n-1}$ (that is, at the n complex nth roots of unity).Without loss of generality, we assume that n is a power of 2, since a given degree-bound can always be raised -we can always add new high-order zero coefficients as necessary. We assume that A is given in coefficient form: $a = (a_0, a_1, \ldots, a_{n-1})$. Let us define the results y_k, for $k = 0, 1, \ldots, n-1$, by

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

The vector $y = (y_0, y_1, \ldots, y_{n-1})$ is the ***Discrete Fourier Transform (DFT)*** of the coefficient vector $a = (a_0, a_1, \ldots, a_{n-1})$.

We also write $y = \mathrm{DFT}_n(a)$

3. Write the FFT algorithm and find the computational complexity of this algorithm. [WBUT 2010]

Answer: ***Refer to Question No. 4(b) of Long Answer Type Questions.***

4. What is meant by LUP decomposition? [WBUT 2013]

Answer:

The idea behind LUP decomposition is to find three n × n matrices L, U, and P such that,

PA = LU

Where,

- L is a unit lower-triangular matrix,
- U is an upper-triangular matrix, and
- P is a permutation matrix.

We call matrices L, U, and P satisfying LUP decomposition of the matrix A. We shall show that every nonsingular matrix A possesses such a decomposition.

The advantage of computing an LUP decomposition for the matrix A is that linear systems can be solved more readily when they are triangular, as is the case for both matrices L and U. Having found an LUP decomposition for A, we can solve the equation Ax = b by solving only triangular linear systems, as follows.

Multiplying both sides of Ax = b by P yields the equivalent equation PAx = Pb.

Using our decomposition, we obtain

LUx = Pb

We can now solve this equation by solving two triangular linear systems. Let us define y = Ux, where x is the desired solution vector. First, we solve the lower triangular system Ly = Pb for the unknown vector y by a method called "forward substitution." Having solved for y, we then solve the upper-triangular system

Ux = y

for the unknown x by a method called "back substitution." The vector x is our solution to Ax = b, since the permutation matrix P is invertible:

$Ax = P^{-1}LUx$

$= P^{-1}Ly$
$= P^{-1}Pb$
$= b.$

5. Discuss Strassen's matrix multiplication procedure and show that the time complexity is reduced from the conventional multiplication. [WBUT 2017]

Answer:

Refer to Question No. 1(1st & 2nd Part) of Long Answer Type Questions.

6. Write a recursive algorithm for finding maximum and minimum from a list of elements. Also find the complexity of your algorithm. [WBUT 2018]

Answer:

Refer to Question No. 2 of Long Answer Type Questions.

Long Answer Type Questions

1. Discuss the procedure for Strassen's matrix multiplication to evaluate the product of 'n' matrices. Find the resulting recurrence relation for the same and analyze its time-complexity. Is this method an improvement over the conventional matrix multiplication method? If so, why? [WBUT 2007, 2011, 2012, 2013]

Answer:

We have seen that the divide-and-conquer approach can reduce the number of one-digit multiplications in multiplying two integers; we should not be surprised that a similar feat can be accomplished for multiplying matrices. Such an algorithm was published by V. Strassen in 1969. The principal insight of the algorithm lies in the discovery that we can find the product C of two 2-by-2 matrices A and B with just seven multiplications as opposed to the eight required by the brute-force algorithm. This is accomplished by using the following formulas:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

where,

$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$
$m_2 = (a_{10} + a_{11}) * b_{00}$
$m_3 = a_{00} * (b_{01} - b_{11})$
$m_4 = a_{11} * (b_{10} - b_{00})$
$m_5 = (a_{00} + a_{01}) * b_{11}$
$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$
$m_7 = (a_{10} - a_{11}) * (b_{10} + b_{11})$

Thus, to multiply two 2-by-2 matrices, Strassen's algorithm makes seven multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires eight multiplications and four additions. These numbers should not lead us to multiplying 2-by-

2 matrices by Strassen's algorithm. Its importance stems from its *asymptotic* superiority as matrix order *n* goes to infinity.

Let *A* and *B* be two *n-by-n* matrices where *n* is a power of two. (If *n* is not a power of two, matrices can be padded with rows and columns of zeros.) We can divide *A, B,* and their product C into four *n* /2-by-n /2 sub-matrices each as follows:

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

It is not difficult to verify that one can treat these sub-matrices as numbers to get the correct product. For example, C_{00} can be computed either as $A_{00} * B_{00} + A_{01} * B_{10}$ or as $M_1 + M_4 - M_5 + M_7$ where M_1, M_4, M_5, and M_7 are found by Strassen's formulas, with the numbers replaced by the corresponding sub-matrices. If the seven products of n /2-by-n /2 matrices are computed recursively by the same method, we have Strassen's algorithm for matrix multiplication.

Let us evaluate the asymptotic efficiency of this algorithm. If *M (n)* is the number of multiplications made by Strassen's algorithm in multiplying two *n-by-n*matrices (where *n* is a power of2), we get the following recurrence relation for it:

$M(n) = 7M(n/2)$ for $n > 1$, $M(1) = 1$.

Since $n = 2^k$

$$M(2k) = 7M(2k-l) = 7[7\ M(2k-2)] = 72M(2k-2) = \ldots$$
$$= 7i\ M(2k\text{-}i)\ldots = 7k\ M(2k-k) = 7k$$

Since $k = log_2\ n$,

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$$

Since this saving in the number of multiplications was achieved at the expense of making extra additions, we must check the number of additions *A* (n) made by Strassen's algorithm. To multiply two matrices of order n > 1, the algorithm needs to multiply seven matrices of order *n/2* and make 18 additions of matrices of size *n/2;* when *n* = 1, no additions are made since two numbers are simply multiplied. These observations yield the following recurrence relation:

$$A(n) = 7\ A(n/2) + 18(n/2)^2 \quad \text{for} \quad n > 1, \quad A(1) = 0.$$

Though one can obtain a closed-form solution to this recurrence, here we simply establish the solution's order of growth. According to the Master Theorem, $A(n) \in \Theta(n^{\log_2 7})$. In other words, the number of additions has the same order of growth as the number of multiplications. This puts Strassen's algorithm in $\Theta(n^{\log_2 7})$, which is a better efficiency class than $\Theta(n^3)$ of the brute-force method.

2. The solution of recursive MAXMIN problem is based on some assumptions. Briefly state the assumptions and its effect on the algorithm in comparison the reality. **[WBUT 2011]**

Answer:

The divide and conquer algorithm we develop for this problem is motivated by the following observation. Suppose we knew the maximum and minimum element in both of the roughly n/2 sized partitions of an n-element (*n*>2) list. Then in order to find the

maximum and minimum element of the entire list we simply need to see which of the two maximum elements is the larger, and which of the two minimums is the smaller. We assume that in a 1-element list the sole element is both the maximum and the minimum element. With this in mind we present the following pseudo-code for the max/min problem.

```
procedure maxmin(A[1...n] of numbers) -> (min, max)
begin
if (n == 1)
      return (A[1], A[1])
else if (n == 2)
      if( A[1] < A[2])
            return (A[1], A[2])
      else
            return (A[2], A[1])
else
      (max_left, min_left) = maxmin(A[1...(n/2)])
      (max_right, min_right) = maxmin(A[(n/2 +1)...n])
if (max_left < max_right)
      max = max_right
else
      max = max_left
if (min_left < min_right)
      max = min_left
else
      min = min_right
return (min, max)
end
```

Let $T(n)$ be the number of comparisons performed by the maxmin procedure. When $n=1$ clearly there are no comparisons. Thus we have $T(1)=0$. Similarly, $T(2)=1$. Otherwise when $n>2$ clearly

$$T(n)=2T(n/2)+2$$

Since maxmin performs two recursive calls on partitions of roughly half of the total size of the list and then makes two further comparisons to sort out the max/min for the entire list. (Of course, to be pedantic there should be floors and ceilings in the recursive function, and something should be said about the fact that the following proof is only for n which are powers of two and how this implies the general result. This is omitted.)

We next show that $T(n)=\left\lceil \frac{3n}{2} \right\rceil -2$ for all n which are powers of 2. The proof is by induction on n.

Base Case:

$(n=2)$: form the recursive definition we have that $T(2)=1$. Similarly, we have that $\left[\frac{3\times 2}{2}\right]-2=3-2=1$, thus verifying the base case.

Inductive Step:

Let $n>2$ and $n=2^j$ for some integer $j\geq 2$. Assume that $T(k)=\left[\frac{3k}{2}\right]-2$ for all $k=2^j$ for some integer $j\geq 2$ and $k<n$. We want to show that this assumption implies that $T(n)=\left[\frac{3n}{2}\right]-2$ for all positive n which are powers of 2.

We start with the given recurrence

$$T(n)=2T(n/2)+2=2\left(\left[\frac{3(n/2)}{2}\right]-2\right)+2$$

By the inductive hypothesis with $k=n/2$. This gives

$$T(n)=2\left[\frac{3(n/2)}{2}\right]-4+2=\left[\frac{3n}{2}\right]-2$$

showing the desired result. By the principle of mathematical induction we are done.

3. Explain how do you attempt to solve 15-puzzle problem using branch and bound strategy. Draw a portion of the state space generated by it. [WBUT 2012]

Answer:

The 15-puzzle has 15 tiles or squares, which are numbered from 1 to 15 that are placed in a 4 x 4 box leaving one position out of the 16 empty and which lay in a square frame. The goal is to reposition the tiles from a given arbitrary starting arrangement by sliding them one at a time into the configuration shown below in figure 1. The only legal moves are ones in which a tile adjacent to the empty spot (ES) is moved to ES. Each move creates an arrangement of the tiles. These arrangements are called the states of tile puzzle. For some initial arrangements, this goal arrangement is possible (admissible), but for others, it may not (inadmissible).

11	5	12	14
15	2		9
13	7	6	1
3	10	4	8

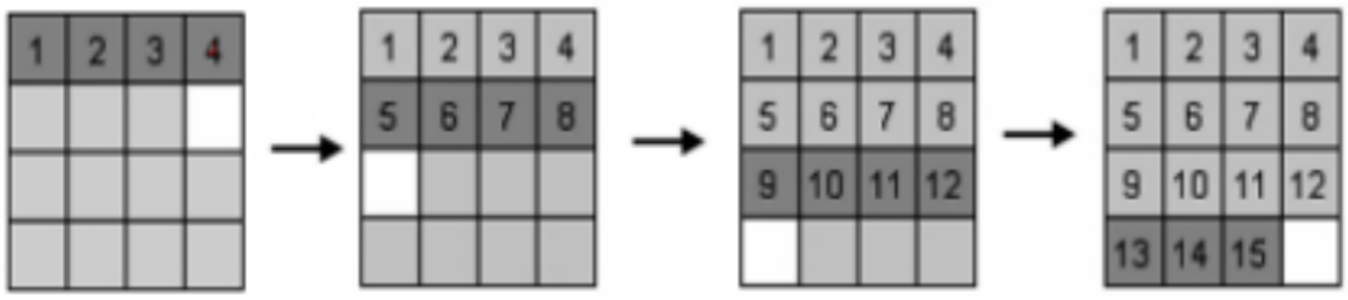
Initial state

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Goal state

Fig: 1

Figure 2 shows the procedure and orders how we can arrange the tiles in such a way to reach the goal state.



Step by step movement of squares to arrange the goal state

Fig: 2

Condition to reach the goal state from initial state

We first assign an index (an integer number) to each tile of the frame. The numbering starts from top position of the frame and move row wise from left to right. So, the goal state of above figure 1 shows the numbering of the each tile in the frame and empty position is numbered by 16.

Let k be the position of tile k according to the frame position of goal state and P(k) is the position index of the tile k in the initial position without considering the empty tile, i.e. tile 16.

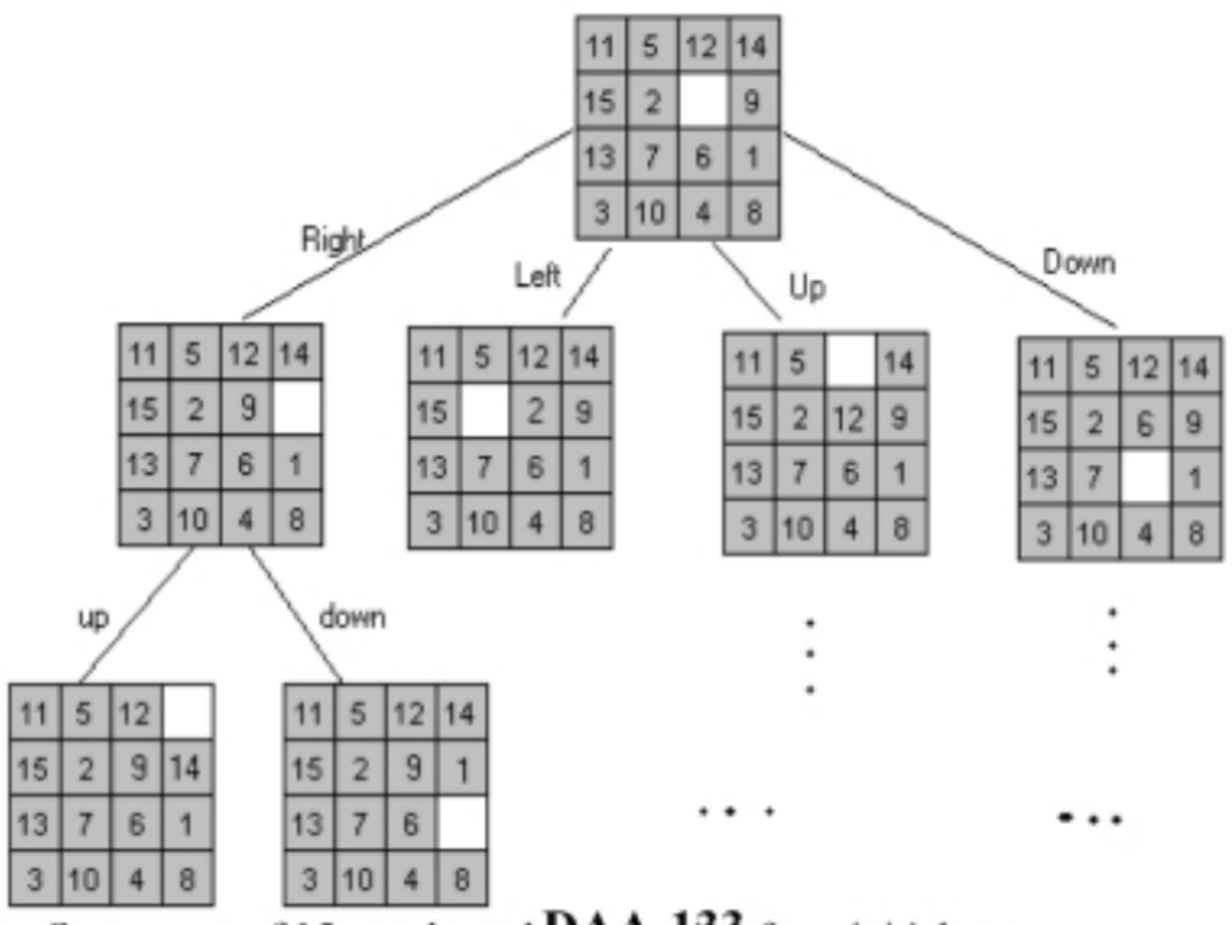
Now, let l(k) be the number of tiles t where,

$t < k$ … (1)

and $P(k) > P(t)$ …(2)

State space is all the possible combination of the tiles within the frame. So, all the possible movements of the tiles are in the state space. Naturally, goal state is also present in the state space, if the goal state can be reached from the initial state.

Now, we can construct the tree of state space starting from initial state. Here all the nodes of the tree represent one of the possible movements from the initial state. In the state space of figure 3, we consider the movement of different tiles with respect to the empty tile.



State space of 15 puzzle problem starting from initial state

Fig: 3

The above state space denotes the all possible arrangement of the tiles within the frame starting from initial state. We can remove those states from the state space which are not reachable to the goal state. So, only those leave nodes are to be considered from a parent node, which can step forward to the goal state, i.e. those configurations which satisfy equation $\sum_{i=1}^{16} L(k) + e$ is even.

4. Write short notes on the following:

a) FFT **[WBUT 2004, 2009]**

b) Strassen's matrix multiplication **[WBUT 2008, 2012, 2014]**

c) Amortized Analysis **[WBUT 2013]**

d) 15-puzzle problem **[WBUT 2013]**

Answer:

a) FFT:

There are several ways to calculate the Discrete Fourier Transform (DFT), such as solving simultaneous linear equations or the *correlation* method. The Fast Fourier Transform (FFT) is another method for calculating the DFT. While it produces the same result as the other approaches, it is incredibly more efficient, often reducing the computation time.

There exists an efficient algorithm to compute the Discrete Fourier Transform and its inverse; it is called Fast Fourier Transform (FFT). So, the Fast Fourier Transform has the following properties:

- The FFT is based on the divide and conquer principle
- It has the time complexity O(n log n) , instead of $O(n^2)$ of DFT.
- It is only applicable for composite sizes, i.e. $n = n_1 n_2$ (and often implemented for powers of 2)

If n is even, we can divide a polynomial

$p(x) = a_0 + a_1 x^1 + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$ into polynomials

$p^{even}(x) = a_0 + a_2 x + a_4 x^2 \ldots + a_{n-2} x^{(n/2)-1}$

$p^{odd}(x) = a_1 + a_3 x + a_5 x^2 \ldots + a_{n-1} x^{(n/2)-1}$

And we can write

$p(x) = p^{even}(x^2) + x.p^{odd}(x^2)$

Fast Fourier Transform (FFT) algorithm.

Input: An n-length coefficient vector a= [a_0, a_1, …,a_n] and a primitive n –th root of unity ω , where n is a power of 2.

Output: A vector y of values of the polynomial for a at the n-th roots of unity.

FFT (a, ω)

{

Step 1 if n = 1 then

Step 1.1 return y= a;
Step 2 x← ω^0
// x will store powers of ω, so initially x = 1.
// Divide Step, which separates even and odd indices
Step 3 $a^{even} \leftarrow [a_0, a_2, a_4, \ldots, a_{n-2}]$
Step 4 $a^{odd} \leftarrow [a_1, a_3, a_5, \ldots, a_{n-1}]$
// Recursive calls with ω^2 as (n/2)-th root of unity, by the
// reduction property
Step 5 $y^{even} \leftarrow FFT(a^{even}, \omega^2)$
Step 6 $y^{odd} \leftarrow FFT(a^{odd}, \omega^2)$
// Combine Step using $x = \omega^i$
Step 7 for i ← 0 to (n/2)-1 do
Step 7.1 $y_i \leftarrow y_i^{even} + x.y_i^{odd}$
Step 7.2 $y_{i+n/2} \leftarrow y_i^{even} - x.y_i^{odd}$
// Use reflexive property
Step 7.3 x← x.ω
Step 8 return y;

b) Strassen's matrix multiplication:

Strassen's matrix multiplication algorithm consists in reducing the number of multiplication at the expense of increasing the number of additions and subtractions. In sort, this algorithm uses 7 multiplications and 18 additions of n/2 x n/2 matrices.

Let

$$A=\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \text{and } B=\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

Be two 2 × 2 matrices. To compute the matrix product

$$C=\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}=\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

We first compute the following products

$$d_1=(a_{11}+a_{22})(b_{11}+b_{22})$$
$$d_2=(a_{21}+a_{22})b_{11}$$
$$d_3=a_{11}(b_{12}-b_{22})$$
$$d_4=a_{22}(b_{21}-b_{11})$$
$$d_5=(a_{11}+a_{12})b_{22}$$
$$d_6=(a_{21}-a_{11})(b_{11}+b_{12})$$
$$d_1=(a_{21}-a_{22})(b_{21}+b_{22})$$

Next, we compute C from the equation

$$C=\begin{pmatrix} d_1+d_4-d_5+d_7 & d_3+d_5 \\ d_2+d_4 & d_1+d_3-d_2+d_6 \end{pmatrix}$$

Since commutatively of scalar products are not used here, the above formula holds for matrices as well.

Time complexity: The number of additions used is 18 and the number of multiplications is 7. This gives rise to the following recurrence for the running time.

$$T(n)=\begin{cases} m & \text{if n}=1 \\ 7T(n/2)+18(n/2)^2 a & \text{if n}\geq 2, \end{cases}$$

$$T(n)=\begin{cases} m & \text{if } n=1 \\ 7T(n/2)+(9a/2)n^2 & \text{if } n\geq 2, \end{cases}$$

Assuming, that n is a power of 2, then we can write,

$$T(n)=\left(m+\frac{(9a/2)2^2}{7-2^2}\right)n^{\log 7}-\left(\frac{(9a/2)2^2}{7-2^2}\right)n^2 = mn^{\log 7}+6an^{\log 7}-6an^2$$

i.e., the time complexity is $\Theta(n^{\log 7})=O(n^{2.81})$

c) Amortized Analysis:

Amortized analysis is a tool for analyzing algorithms that perform a sequence of similar operations. It can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation within the sequence might be expensive. In a sequence of operations the worst case does not occur often in each operation - some operations may be cheap, some may be expensive. Therefore, a traditional worst-case per operation analysis can give overly pessimistic bound. The amortized approach is going to assign an "artificial cost" to each operation in the sequence, called the amortized cost of an operation. It requires that the total real cost of the sequence should be bounded by the total of the amortized costs of all the operations.

Three methods are used in amortized analysis

1. Aggregate Method (or brute force)
2. Accounting Method (or the banker's method)
3. Potential Method (or the physicist's method)

d) 15-Puzzle Problem: ***Refer to Question No. 4 of Long Answer Type Questions.***