CONTENTS

RCS–502 : Design and Analysis of Algorithm

1

UNIT

Introduction

PART-1

Introduction : Algorithms, Analyzing Algorithms, Complexity of Algorithms, Growth of Functions, Performance Measurements.

CONCEPT OUTLINE : PART-1

- **Algorithm :** An algorithm is a sequence of computational steps that transform the input into the output.

  Input → Algorithm → Output

- **Complexity of algorithm :** Complexity of an algorithm is defined by two terms :
  i. Time complexity
  ii. Space complexity
- **Analysis of algorithm :** The analysis of an algorithm provides some basic information about that algorithm like time, space, performance etc.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.1. What do you mean by algorithm ? Write the characteristics of algorithm. **AKTU 2013-14, Marks 05**

Answer

1. An algorithm is a set of rules for carrying out calculation either by hand or on machine.
2. It is a finite step-by-step procedure to achieve a required result.
3. It is a sequence of computational steps that transform the input into the output.
4. An algorithm is a sequence of operations performed on data that have to be organized in data structures.

Characteristics of algorithm are :

1. **Input and output :** These characteristics require that an algorithm produces one or more outputs and have zero or more inputs that are externally supplied.

2. **Definiteness :** Each operation must be perfectly clear and unambiguous.
3. **Effectiveness :** This requires that each operation should be effective, *i.e.*, each step can be done by a person using pencil and paper in a finite amount of time.
4. **Termination :** This characteristic requires that an algorithm must terminate after a finite number of operations.

Que 1.2. What do you mean by analysis or complexity of an algorithm ? Give its types and cases.

Answer

Analysis/complexity of an algorithm :

The complexity of an algorithm is a function $g(n)$ that gives the upper bound of the number of operation (or running time) performed by an algorithm when the input size is n.

Types of complexity :

1. **Space complexity :** The space complexity of an algorithm is the amount of memory it needs to run to completion.
2. **Time complexity :** The time complexity of an algorithm is the amount of time it needs to run to completion.

Cases of complexity :

1. **Worst case complexity :** The running time for any given size input will be lower than the upper bound except possibly for some values of the input where the maximum is reached.
2. **Average case complexity :** The running time for any given size input will be the average number of operations over all problem instances for a given size.
3. **Best case complexity :** The best case complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n.

Que 1.3. What do you understand by asymptotic notations ? Describe important types of asymptotic notations.

AKTU 2013-14, Marks 05

OR

Discuss asymptotic notations in brief. **AKTU 2014-15, Marks 05**

Answer

1. Asymptotic notation is a shorthand way to represent 'fastest possible' and 'slowest possible' running times for an algorithm, using high and low bounds on speed.

2. It is a line that stays within bounds.
3. These are also referred to as 'best case' and 'worst case' scenarios and are used to find complexities of functions.

Notations used for analyzing complexity are :

1. **Θ-Notation (Same order) :**
   a. This notation bounds a function within constant factors.
   b. We say $f(n) = \Theta g(n)$ if there exist positive constants n_0, c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive.

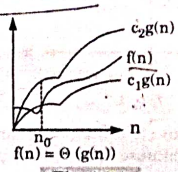


Fig. 1.3.1.

2. **O-Notation (Upper bound) :**
   a. Big-oh is formal method of expressing the upper bound of an algorithm's running time.
   b. It is the measure of the longest amount of time it could possibly take for the algorithm to complete.
   c. More formally, for non-negative functions, $f(n)$ and $g(n)$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n \geq n_0$.

$$f(n) \leq cg(n)$$

   d. Then, $f(n)$ is big-oh of $g(n)$. This is denoted as :

$$f(n) \in O(g(n))$$

i.e., the set of functions which, as n gets large, grow faster than a constant time $f(n)$.

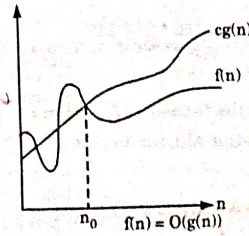


Fig. 1.3.2.

3. **Ω-Notation (Lower bound) :**
   a. This notation gives a lower bound for a function within a constant factor.
   b. We write $f(n) = \Omega g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

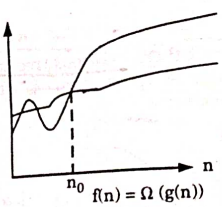


Fig. 1.3.3.

4. **Little-oh notation (o) :** It is used to denote an upper bound that is asymptotically tight because upper bound provided by O-notation is not tight.
   $o(g(n)) = \{f(n)$: for any positive constant $c > 0$, if a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)\ \forall\ n \ge n_0\}$
5. **Little omega notation (ω) :** It is used to denote lower bound that is asymptotically tight.
   $\omega(g(n)) = \{f(n)$: For any positive constant $c > 0$, if a constant $n_0 > 0$ such that $0 \le cg(n) < f(n)\ \forall\ n \ge n_0\}$

Que 1.4. **If $f(n) = 100 * 2^n + n^5 + n$, then show that $f(n) = O(2^n)$.**

Answer

If $f(n) = 100 * 2^n + n^5 + n$

For $n^5 \ge n$

$$100 * 2^n + n^5 + n \le 100 * 2^n + n^5 + n^5$$
$$\le 100 * 2^n + 2n^5$$

For $2^n \ge n^5$

$$100 * 2^n + n^5 + n \le 100 * 2^n + 2.2^n \qquad [\because\ n \le 1, n_0 = 23]$$
$$\le 102 * 2^n$$

Thus, $f(n) = O(2^n)$

Que 1.5. **Consider the following function :**

```
int SequentialSearch(int A[ ], int &x, int n)
    {
    int i;
    for (int i = 0; i < n && a[i] != x ; i++)
    if (i == n) return – 1;
    return i;
```



```
    }
```

Determine the average and worst case time complexity of the function sequential search.

Answer

Average case time complexity :

1. Let P be a probability of getting successful search and n is the total number of elements in the list.
2. The first match of the element will occur at i^{th} location. Hence probability of occurring first match is P/n for every i^{th} element.
3. The probability of getting unsuccessful search is $(1 - P)$.
4. Average case time complexity $C_{avg}(n)$ = Probability of successful search + Probability of unsuccessful search

$$= \left[1.\frac{P}{n} + 2.\frac{P}{n} + \ldots + i.\frac{P}{n}\right] + n(1-P)$$
$$= \frac{P}{n}[1 + 2 + \ldots + i] + n(1-P)$$
$$= \frac{P}{n}\frac{n(n+1)}{2} + n(1-P) \qquad (\because\ i = i{+}{+} \text{ till } i < n)$$
$$C_{avg}(n) = \frac{P(n+1)}{2} + n(1-P)$$

5. So, for unsuccessful search, $P = 0$

   Then $C_{avg}(n) = 0 + n(1 - 0)$
   $= n$

6. Thus the average case time complexity becomes equal to n for unsuccessful search.
7. For successful search,

$$P = 1$$
$$C_{avg}(n) = \frac{1(n+1)}{2} + n(1-1) = \frac{n+1}{2} + 0$$
$$C_{avg}(n) = O\left(\frac{n}{2}\right)$$

8. That means the algorithm scans about half of the elements from the list.

Worst case time complexity :

1. In this algorithm, we will consider the worst case when the element to be searched is present at n^{th} location then the algorithm will run for longest time and we will get the worst case time complexity as :
   $$C_{worst}(n) = O(n)$$
2. For any instance of input of size n, the running time will not exceed $O(n)$.

Que 1.6. **Write Master's theorem and explain with suitable examples.**

Answer

Master's theorem :

Let $T(n)$ be defined on the non-negative integers by the recurrence.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ where } a \geq 1, b > 1 \text{ are constants}$$

a = the number of sub-problems in the recursion

$1/b$ = the portion of the original problem represented by each sub-problem

$f(n)$ = the cost of dividing the problem + the cost of merging the solution

Then $T(n)$ can be bounded asymptotically as follows :

Case 1 :

If it is true that :

$$f(n) = O(n^{\log_b a - E}) \text{ for } E > 0$$

It follows that :

$$T(n) = \Theta(n^{\log_b a})$$

Example : $$T(n) = 8T\left(\frac{n}{b}\right) + 1000n^2$$

In the given formula, the variables get the following values :

$$a = 8, b = 2, f(n) = 1000n^2, \log_b a = \log_2 8 = 3$$

$$n^{\log_b a} = n^{\log_2 8} = n^3$$

$$f(n) = O(n^{\log_b a - E}) = O(n^{3-E})$$

For $E = 1$, we get

$$f(n) = O(n^{3-1}) = O(n^2)$$

Since this equation holds, the first case of the master's theorem applies to the given recurrence relation, thus resulting solution is

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$$

Case 2 :

If it is true that :

$$f(n) = \Theta(n^{\log_b a})$$

It follows that :

$$T(n) = \Theta(n^{\log_b a} \log(n))$$

Example :

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

In the given formula, the variables get the following values :

$$a = 2, b = 2, f(n) = n, \log_b a = \log_2 2 = 1$$

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

$$f(n) = \Theta(n^{\log_b a}) = \Theta(n)$$

Since this equation holds, the second case of the Master's theorem applies to the given recurrence relation, thus resulting solution is :

$$T(n) = \Theta(n^{\log_b a} \log(n)) = \Theta(n \log n)$$

Case 3 :

If it is true that :

$$f(n) = \Omega(n^{\log_b a + E}) \text{ for } E > 0$$

and if it is also true that :

if $af\left(\frac{n}{b}\right) \leq cf(n)$ for $a, c < 1$ and all sufficiently large n

It follows that :

$$T(n) = \Theta(f(n))$$

Example : $$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

In the given formula, the variables get the following values :

$$a = 2, b = 2, f(n) = n^2, \log_b a = \log_2 2 = 1$$

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

$$f(n) = \Omega(n^{\log_b a + E})$$

For $E = 1$ we get

$$f(n) = \Omega(n^{1+1}) = \Omega(n^2)$$

Since the equation holds, third case of Master's theorem is applied.

Now, we have to check for the second condition of third case, if it is true that :

$$af\left(\frac{n}{b}\right) \leq c\, f(n)$$

If we insert once more the values, we get :

$$2\left(\frac{n}{2}\right)^2 \leq cn^2 \Rightarrow \frac{1}{2}n^2 \leq cn^2$$

If we choose $c = \frac{1}{2}$, it is true that :

$$\frac{1}{2}n^2 \leq \frac{1}{2}n^2 \quad \forall\, n \geq 1$$

So, it follows that :

$$T(n) = \Theta(f(n))$$

If we insert once more the necessary values, we get :

$$T(n) \in \Theta(n^2)$$

Thus, the given recurrence relation $T(n)$ was in $\Theta(n^2)$.

Que 1.7. **The recurrence $T(n) = 7T(n/2) + n^2$ describe the running time of an algorithm A. A competing algorithm A has a running time $T'(n) = aT'(n/4) + n^2$. What is the largest integer value for a A' is asymptotically faster than A ?** **AKTU 2017-18, Marks 10**

Answer

Given that :

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2 \quad ...(1.7.1)$$

$$T'(n) = aT'\left(\frac{n}{4}\right) + n^2 \quad ...(1.7.2)$$

Here, equation (1.7.1) defines the running time for algorithm A and equation (1.7.2) defines the running time for algorithm A'. Then for finding value of a for which A' is asymptotically faster than A we find asymptotic notation for the recurrence by using Master's method.

Now, compare equation (1.7.1) by $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

we get, $a = 7$

$b = 2$

$f(n) = n^2$

$n^{\log_b a} = n^{\log_2 7} = n^{2.81}$

Now, apply cases of Master's, theorem as :

Case 1 :- $f(n) = O(n^{\log_2 7 - E})$

$\Rightarrow$ $f(n) = O(n^{2.81-E})$

$\Rightarrow$ $f(n) = O(n^{2.81-0.81})$

$\Rightarrow$ $f(n) = O(n^2)$

Hence, case 1 of Master's theorem is satisfied.

Thus, $T(n) = \theta(n^{\log_b a})$

$\Rightarrow$ $T(n) = \theta(n^{2.81})$

Since recurrence given by equation (1.7.1) is asymptotically bounded by θ-notation by which is used to show optimum time we have to show that recurrence given by equation (1.7.2) is bounded by Ω-notation which shows minimum time (best case).

For the use satisfy the case 3 of Master theorem, let $a = 16$

$$T'(n) = 16T'\left(\frac{n}{4}\right) + n^2$$

$\Rightarrow$ $a = 16$

$b = 4$

$f(n) = n^2$

$\Omega(n^{\log_b a + E}) = \Omega(n^{2+E})$

Hence, case 3 of Master's theorem is satisfied.

$\Rightarrow$ $T(n) = \theta(f(n))$

$\Rightarrow$ $T(n) = \theta(n^2)$

Therefore, this shows that A' is asymptotically faster than A when $a = 16$.

Que 1.8. **Solve the following recurrences :**

$T(n) = T(\sqrt{n}) + O(\log n)$ **AKTU 2014-15, Marks 10**

Answer

$$T(n) = T(\sqrt{n}) + O(\log n) \quad ...(1.8.1)$$

Let $m = \log_2 n$

$n = 2^m$

$n^{1/2} = 2^{m/2}$ ...(1.8.2)

Put value of $\sqrt{n}$ in equation (1.8.1) we get

$$T(2^m) = T\left(2\frac{m}{2}\right) + O(\log 2^m) \quad ...(1.8.3)$$

$$x(m) = T(2^m) \quad ...(1.8.4)$$

Putting the value of $x(m)$ in equation (1.8.3)

$$x(m) = x\left(\frac{m}{2}\right) + O(m) \quad ...(1.8.5)$$

Solution of equation (1.8.5) is given as

$a = 1, \quad b = 2, \quad f(n) = O(m)$

$f(n) = O(m^{\log, 1+E})$ where $E = 1$

$T(n) = x(m) = O(m) = O(\log n)$

Que 1.9. **What do you mean by recursion ? Explain your answer with an example.**

Answer

1. Recursion is a process of expressing a function that calls itself to perform specific operation.
2. Indirect recursion occurs when one function calls another function that then calls the first function.
3. Suppose P is a procedure containing either a call statement to itself or a call statement to a second procedure that may eventually result in a call statement back to the original procedure P.
4. Then P is called recursive procedure. So, the program will not continue to run indefinitely.
5. A recursive procedure must have the following two properties :
   a. There must be certain criteria, called base criteria, for which the procedure does not call itself.
   b. Each time the procedure does call itself, it must be closer to the criteria.
6. A recursive procedure with these two properties is said to be well-defined.
7. Similarly, a function is said to be recursively defined if the function definition refers to itself.

For example :

The factorial function may also be defined as follows :

a. If $n = 0$, then $n! = 1$.

Here, the value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value).

b. If $n > 0$, then $n! = n.(n-1)!$

Here, the value of $n!$ for arbitrary n is defined in terms of a smaller value of n which is closer to the base value 0.

Observe that this definition of $n!$ is recursive, since it refers to itself when it uses $(n-1)!$.

Que 1.10. **What is recursion tree ? Describe.**

AKTU 2013-14, Marks 05

Answer

1. Recursion tree is a pictorial representation of an iteration method, which is in the form of a tree, where at each level nodes are expanded.
2. In a recursion tree, each node represents the cost of a single subproblem.
3. Recursion trees are particularly useful when the recurrence describes the running time of a divide and conquer algorithm.
4. A recursion tree is best used to generate a good guess, which is then verified by the substitution method.
5. It is a method to analyze the complexity of an algorithm by diagramming the recursive function calls.
6. This method can be unreliable.

Que 1.11. **Solve the recurrence :**

$$T(n) = T(n-1) + T(n-2) + 1, \text{ when } T(0) = 0 \text{ and } T(1) = 1.$$

Answer

$$T(n) = T(n-1) + T(n-2) + 1$$

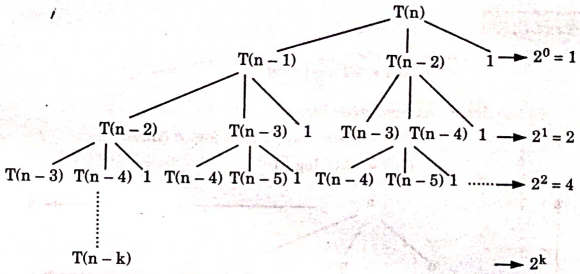
At k^{th} level, $T(1)$ will be equal to 1

when, $n - k = 1$

$$k = n - 1$$

$$= 2^0 + 2^1 + 2^2 + \ldots\ldots 2^k$$

$$= 2^0 + 2^1 + 2^2 + \ldots\ldots 2^{n-1}$$



$$\left[\text{Sum of } n \text{ terms of geometric progression} = \frac{a(r^n - 1)}{r - 1}\right] = \frac{2^0(2^{n-1} - 1)}{2 - 1}$$

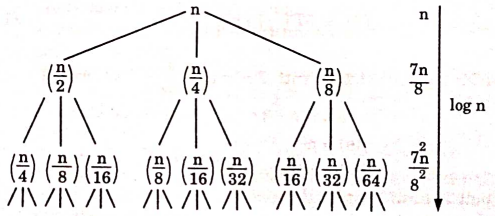
$$= 2^{n-1} - 1 = O(2^n)$$

Que 1.12. **Solve the following recurrences :**

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n$$

AKTU 2014-15, Marks 2.5

Answer



$$T(n) = n + \frac{7n}{8} + \frac{7^2 n}{8^2} + \ldots\ldots + \log n \text{ times}$$

$$= \Omega(n \log n)$$

Que 1.13. **Consider the recurrences**

$T(n) = 3T(n/3) + cn$, **and**

$T(n) = 5T(n/4) + n^2$ **where** c **is constant and** n **is the number of inputs. Find the asymptotic bounds.** **AKTU 2013-14, Marks 05**

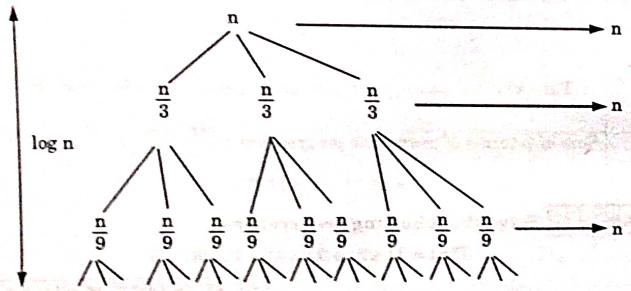
Answer

$$T(n) = 3T\left(\frac{n}{3}\right) + cn$$

we can draw recursion tree for $c \geq 1$

$$T(n) = n + n + n + \ldots\ldots + \log_3 n \text{ times}$$

$$T(n) = \Omega(n \log n)$$



$$T(n) = n + n + n + \ldots\ldots + \log_3 n \text{ times}$$

$$T(n) = \Omega(n \log n)$$

$$T(n) = 5T\left(\frac{n}{4}\right) + n^2 \qquad \ldots(1.13.1)$$

Comparing equation (1.13.1) with $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, we get

$$a = 5, b = 4$$

$$f(n) = n^2$$

$$n^{\log_b a} = n^{\log_4 5} = n^{1.160}$$

Now apply cases of Master's theorem as :

$$f(n) = \Omega(n^{\log_b a + E}) = \Omega(n^{1.160 + E})$$

$$= \Omega(n^{1.160 + 0.84}) = \Omega(n^2) \text{ where } E = 0.81$$

Hence, case 3 of Master's theorem is satisfied.

$$T(n) = \theta(f(n))$$

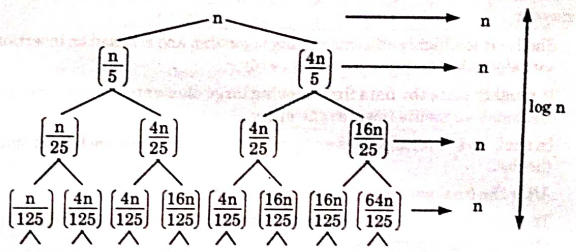
$$T(n) = \theta(n^2)$$

Que 1.14. **Solve the following by recursion tree method**

$$T(n) = n + T(n/5) + T(4n/5)$$

AKTU 2017-18, Marks 10

Answer



$$T(n) = n + n + n + \ldots\ldots\ldots\ldots + \log n \text{ times}$$

$$= \Omega(n \log n)$$

PART-2

Sorting and Order Statistic : Shell Sort, Quick Sort, Merge Sort, Heap Sort, Comparison of Sorting Algorithms, Sorting in Linear Time.

CONCEPT OUTLINE : PART-2

- **Shell Sort :** It is an algorithm that roughly sort the data first and move large elements towards one end and smaller ones towards the other.
  Complexity : $O(n^2)$
- **Heap Sort :** The heap is an array that can be viewed as a complete binary tree. The tree is filled on all levels except the lowest.
  Complexity : $O(n \log n)$
- **Merge Sort :** It works on divide and conquer approach first, it divides a list into two sublist and sort it and then combine as a new sorted one list.
  Complexity : $O(n \log n)$
- **Quick Sort :** It works on the principle of divide and conquer. It works by partitioning a given array.
  Complexity : $O(n^2)$

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.15. **Explain shell sort with example.**

Answer

1. Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm and we can code it easily.
2. It roughly sorts the data first, moving large elements towards one end and small elements towards the other.
3. In shell sort several passes over the data is performed, each finer than the last.
4. After the final pass, the data is fully sorted.
5. The shell sort does not sort the data itself, it increases the efficiency of other sorting algorithms.

Algorithm :

Input : An array a of length n with array elements numbered 0 to $n-1$.

1. inc ← round (n/2)
2. while inc > 0
3. for i = inc to $n-1$
   temp ← $a[i]$
   $j \leftarrow i$
   while $j \geq$ inc and a [j – inc] > temp
   $a[j] \leftarrow a[j - \text{inc}]$
   $j \leftarrow j - \text{inc}$
   $a[j] \leftarrow$ temp
4. inc ← round (inc/2.2)

For example :

45	36	75	20	05	90	80	65	30	50	10	75	85

The distance between the elements to be compared is 3. The subfiles generated with the distance of 3 are as follows :

Subfile 1	a[0]	a[3]	a[6]	a[9]	a[12]
Subfile 2	a[1]	a[4]	a[7]	a[10]	
Subfile 3	a[2]	a[5]	a[8]	a[11]	

Input to pass 1 with distance = 3

45	36	75	20	05	90	80	65	30	50	10	75	85

(a)

Output of pass 1 is input to pass 2 and distance = 2

20	05	30	45	10	75	50	36	75	80	65	90	85

(b)

Output of pass 2 is input to pass 3 and distance = 1

10	05	20	36	30	45	50	75	65	80	75	90	85

(c)

Output of pass 3

05	10	20	30	36	45	50	65	75	75	80	85	90

(d)

Fig. 1.15.1.

Que 1.16. **Describe any one of the following sorting techniques :**

i. Selection sort

ii. Insertion sort **AKTU 2013-14, Marks 05**

Answer

i. Selection sort (A) :

1. $n \leftarrow$ length [A]
2. for $j \leftarrow 1$ to $n-1$
3. smallest ← j
4. for $i \leftarrow j + 1$ to n
5. if A [i] < A [smallest]
6. then smallest ← i
7. exchange (A [j], A [smallest])

ii. Insertion_Sort(A) :

1. for $j \leftarrow 2$ to length[A]
2. do key ← $A[j]$
3. Insert $A[j]$ into the sorted sequence $A[1 \ldots\ldots j-1]$
4. $i \leftarrow j-1$
5. while $i > 0$ and $A[i]$ > key
6. do $A[i+1] \leftarrow A[i]$
7. $i \leftarrow i-1$
8. $A[i+1] \leftarrow$ key

Que 1.17. **Write non-deterministic algorithm for sorting.**

AKTU 2016-17, Marks 10

Answer

Non-deterministic algorithms are algorithm that, even for the same input, can exhibit different behaviours on different runs, iterations and executions.

N SORT(A, B) :

1. for $i = 1$ to n do
2. j = choice(1 . . . n)
3. if $B[j]$!= 0 then failure
4. $B[j] = A[i]$
5. endfor
6. for $i = 1$ to $n - 1$ do
7. if $B[i] < B[i + 1]$ then failure
8. endfor
9. print(B)
10. success

Que 1.18. **Explain the concepts of quick sort method and analyze its complexity with suitable example.**

AKTU 2016-17, Marks 10

Answer

Quick sort :

Quick sort works by partitioning a given array $A[p \ldots r]$ into two non-empty subarray $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ such that every key in $A[p \ldots q-1]$ is less than or equal to every key in $A[q+1 \ldots r]$. Then the two subarrays are sorted by recursive calls to quick sort.

Quick_Sort (A, p, r)

1. If $p < r$ then
2. $q \leftarrow$ Partition (A, p, r)
3. Recursive call to Quick_Sort $(A, p, q-1)$
4. Recursive call to Quick_Sort $(A, q+1, r)$

As a first step, Quick sort chooses as pivot one of the items in the array to be sorted. Then array is partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

Partition (A, p, r)

1. $x \leftarrow A[r]$
2. $i \leftarrow p-1$
3. for $j \leftarrow p$ to $r-1$

4. do if $A[j] \leq x$
5. then $i \leftarrow i+1$
6. then exchange $A[i] \leftrightarrow A[j]$
7. exchange $A[i+1] \leftrightarrow A[r]$
8. return $i+1$

Example : This example shows that how "Pivot" and "Quick sort" work suppose $A = \{3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9,\}$

Sort the array A using quick sort algorithm.

Solution : Given array to be sorted

3	1	4	1	5	9	2	6	5	3	5	8	9

Step 1 : The array is Pivoted about it first element *i.e.*, Pivot $(P) = 3$

3	1	4	1	5	9	2	6	5	3	5	8	9

P

Step 2 : Find first element larger then Pivot (make underline) and find element not larger than pivot from end make over line.

3	1	4	1	5	9	2	6	5	3	5	8	9

P Underline Overline

Step 3 : Swap these element and scan again.

3	1	3	1	5	9	2	6	5	4	5	8	9

P

Array after swapping

3	1	3	1	5	9	2	6	5	4	5	8	9

P Underline Overline

Apply swapping,

3	1	3	1	2	9	5	6	5	4	5	8	9

Again apply scanning,

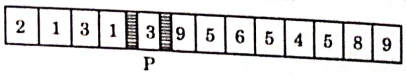
3	1	3	1	2	9	5	6	5	4	5	8	9

Overline Underline

The pointers have crossed

i.e., overline on left of underlined

Then, in this situation swap Pivot with overline.



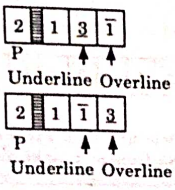
Now, Pivoting process is complete.

Step 4 : Recursively sort subarrays on each side of Pivot.

Subarray 1 : 2 1 3 1

Subarray 2 : 9 5 6 5 1 5 8 9

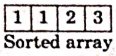
First apply Quick sort for subarray 1.



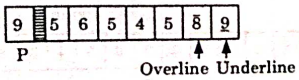
The pointers have crossed.

i.e., overline on left of underlined.

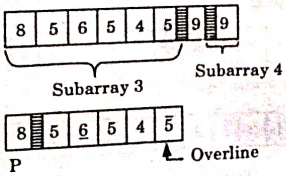
Swap pivot with overline



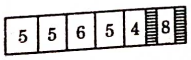
Now, for subarray 2 we apply Quick sort procedure.

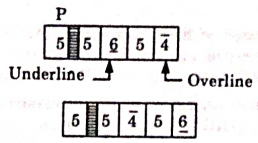


The pointer has crossed. Then swap Pivot with overline.



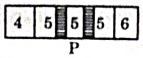
Swap overline with Pivot.



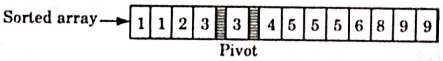


Overline on left of underlined.

Swap Pivot with overline.



Now combine all the subarrays



Analysis of complexity :

i. Worst case :

1. Let $T(n)$ be the worst case time for quick sort on input size n. We have a recurrence

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) \quad ...(1.18.1)$$

where q ranges from 0 to $n-1$, since the partition produces two regions, each having size $n-1$.

2. Now we assume that $T(n) \le cn^2$ for some constant c.

Substituting our assumption in equation (1.18.1) we get

$$T(n) \le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n)$$

$$= \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n)$$

3. Since the second derivative of expression $q^2 + (n-q-1)^2$ with respect to q is positive. Therefore, expression achieves a maximum over the range $0 \le q \le n-1$ at one of the endpoints.
4. This gives the bound

$\max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$

5. Continuing with the bounding of $T(n)$ we get

$$T(n) \le cn^2 - c(2n-1) + \Theta(n) \le cn^2$$

6. Since we can pick the constant c large enough so that the $c(2n-1)$ term dominates the $\Theta(n)$ term. We have

$$T(n) = O(n^2)$$

7. Thus, the worst case running time of quick sort is $\Theta(n^2)$.

ii. **Average case :**

1. If the split induced of RANDOMIZED_PARTITION puts constant fraction of elements on one side of the partition, then the recurrence tree has depth $\Theta(\log n)$ and $\Theta(n)$ work is performed at each level.
2. This is an intuitive argument why the average case running time of RANDOMIZED_QUICKSORT is $\Theta(n \log n)$.
3. Let $T(n)$ denotes the average time required to sort an array of n elements. A call to RANDOMIZED_ QUICKSORT with a 1 element array takes a constant time, so we have $T(1) = \Theta(1)$.
4. After the split RANDOMIZED_QUICKSORT calls itself to sort two subarrays.
5. The average time to sort an array A[1 .. q] is $T[q]$ and the average time to sort an array A[q+1 .. n] is $T[n-q]$. We have

$$T(n) = 1/n\,(T(1) + T(n-1) + {}^{n-1}S_{q=1}\,T(q) \quad T(n-q))) + \Theta(n) \qquad ...(1.18.1)$$

We know from worst-case analysis

$$T(1) = \Theta(1) \text{ and } T(n-1) = O(n^2)$$

$$T(n) = 1/n\,(\Theta(1) + O(n^2)) + 1/n\,{}^{n-1}\Theta_{q=1}\,(r(q) + T(n-q)) + Q(n)$$

$$= 1/n\,{}^{n-1}S_{q=1}\,(T(q) + T(n-q)) + Q(n) \qquad ...(1.18.2)$$

$$= 1/n\,[2\,{}^{n-1}S_{k=1}(T(k))] + \Theta(n)$$

$$= 2/n\,{}^{n-1}S_{k=1}\,(T(k)) + \Theta(n) \qquad ...(1.18.3)$$

6. Solve the above recurrence using substitution method. Assume that $T(n) \le an \log n + b$ for some constants $a > 0$ and $b > 0$.

If we can pick 'a' and 'b' large enough so that $n \log n + b > T(1)$.Then for $n > 1$, we have

$$T(n) \ge {}^{n-1}\Theta_{k=1}\,2/n\,(ak \log k + b) + \Theta(n)$$

$$= 2a/n\,{}^{n-1}S_{k=1}\,k \log k - 1/8(n^2) + 2b/n \quad (n-1) + \Theta n \qquad ...(1.18.4)$$

At this point we are claiming that ${}^{n-1}\Theta_{k=1}\,k \log k \le 1/2\,n^2 \log n - 1/8(n^2)$

Substituting this claim in the equation (1.18.4), we get

$$T(n) \le 2a/n\,[1/2n^2 \log n - 1/8(n^2)] + 2/n\,b(n-1) + \Theta(n) \qquad ...(1.18.5)$$

$$\le an \log n - an/4 + 2b + \Theta(n)$$

In the equation (1.18.5), $\Theta(n) + b$ and $an/4$ are polynomials and we can choose 'a' large enough so that $an/4$ dominates $\Theta(n) + b$.

We conclude that QUICKSORT's average running time is $\Theta(n \log n)$.

Que 1.19. Discuss the best case and worst case complexities of quick sort algorithm in detail. **AKTU 2014-15, Marks 05**

Answer

Best case :

1. The best thing that could happen in quick sort would be that each partitioning stage divides the array exactly in half.
2. In other words, the best to be a median of the keys in $A[p .. r]$ every time procedure 'Partition' is called.
3. The procedure 'Partition' always split the array to be sorted into two equal sized arrays.
4. If the procedure 'Partition' produces two regions of size $n/2$, then the recurrence relation is :

$$T(n) \le T(n/2) + T(n/2) + \Theta(n) \le 2T(n/2) + \Theta(n)$$

And from case (2) of master theorem

$$T(n) = \Theta(n \log n)$$

Worst case : Refer Q. 1.18, Page 1–17B, Unit-1.

Que 1.20. Explain the concept of merge sort with example.

AKTU 2016-17, Marks 10

Answer

1. Merge sort is a sorting algorithm that uses the idea of divide and conquer.
2. This algorithm divides the array into two halves, sorts them separately and then merges them.
3. This procedure is recursive, with the base criteria that the number of elements in the array is not more than 1.

Algorithm :

MERGE_SORT (a, p, r)

1. if $p < r$
2. then $q \leftarrow \lfloor (p+r)/2 \rfloor$
3. MERGE-SORT (A, p, q)
4. MERGE-SORT $(A, q+1, r)$
5. MERGE (A, p, q, r)

MERGE (A, p, q, r)

1. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. Create arrays L [1 .....$n_1 + 1$] and R [1.....$n_2 + 1$]
4. for $i = 1$ to n_1 do $L[i] = A\,[p + i - 1]$

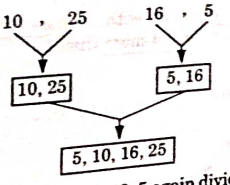
endfor

5. for $j = 1$ to n_2
   do
   $R[j] = A[q + j]$
   endfor
6. $L[n_1 + 1] = \infty$, $R[n_2 + 1] = \infty$
7. $i = 1, j = 1$
8. for $k = p$ to r
   do
   if $L[i] \leq R[j]$
   then $A[k] \leftarrow L[i]$
   $i = i + 1$
   else $A[k] = R[j]$
   $j = j + 1$
   endif
   endfor
9. exit

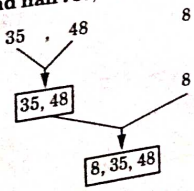
Example :

10, 25, 16, 5, 35, 48, 8

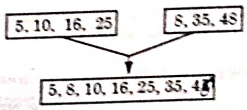
1. **Divide into two halves :** [10, 25, 16, 5] [35, 48, 8]
2. **Consider the first part :** 10, 25, 16, 5 again divide into two sub-arrays



3. **Consider the second half :** 35, 48, 5 again divide into two sub-arrays



4. Merge these two sorted sub-arrays,



This is the sorted array.

Que 1.21. **Determine the best case time complexity of merge sort algorithm.**

Answer

1. The best case of merge sort occurs when the largest element of one array is smaller than any element in the other array.
2. For this case only $n/2$ comparisons of array elements are made.
3. Merge sort comparisons are obtained by the recurrence equation of the recursive calls used in merge sort.
4. As it divides the array into half so the recurrence function is defined as :

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n = 2T\left(\frac{n}{2}\right) + n \quad \ldots(1.21.1)$$

5. By using variable k to indicate depth of the recursion, we get

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn \quad \ldots(1.21.2)$$

6. For the best case there are only $n/2$ comparisons hence equation (1.21.2) can be written as

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

7. At the last level of recursion tree

$$2^k = n$$

$$k = \log_2 n$$

8. So the recurrence function is defined as :

$$T(n) = 2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + \frac{n}{2}\log_2 n$$

$$= nT(1) + \frac{n}{2}\log_2 n = \frac{n}{2}\log_2 n + n$$

$$T(n) = O(n \log_2 n)$$

Hence, the best case complexity of merge sort is $O(n \log_2 n)$.

Que 1.22. **Explain heap sort algorithm with its analysis.**

OR

What is the running time of heap sort on an array A of length n that is already sorted in increasing order ?

OR

Discuss the complexity of Max-heapify and Build Max Heap procedures.

AKTU 2014-15, Marks 10

Answer

1. Heap sort finds the largest element and puts it at the end of array, then the second largest item is found and this process is repeated for all other elements.
2. The general approach of heap sort is as follows :
   a. From the given array, build the initial max heap.
   b. Interchange the root (maximum) element with the last element.
   c. Use repetitive downward operation from root node to rebuild the heap of size one less than the starting.
   d. Repeat step a and b until there are no more elements.

Analysis of heap sort :

Complexity of heap sort for all cases is $O(n \log_2 n)$.

MAX-HEAPIFY (A, i) :

1. $i \leftarrow$ left $[i]$
2. $r \leftarrow$ right $[i]$
3. if $l \leq$ heap-size $[A]$ and $A[l] > A[i]$
4. then largest $\leftarrow 1$
5. else largest $\leftarrow i$
6. if $r \leq$ heap-size $[A]$ and $A[r] > A$ [largest]
7. then largest $\leftarrow r$
8. if largest $\neq i$
9. then exchange $A[i] \leftrightarrow A$[largest]
10. MAX-HEAPIFY [A, largest]

HEAP-SORT(A) :

1. BUILD-MAX-HEAP (A)
2. for $i \leftarrow$ length $[A]$ down to 2
3. do exchange $A[1] \leftrightarrow A[i]$
4. heap-size $[A] \leftarrow$ heap-size $[A] - 1$
5. MAX-HEAPIFY $(A, 1)$

BUILD-MAX-HEAP (A)

1. heap-size $(A) \leftarrow$ length $[A]$
2. for $i \leftarrow$ (length $[A]/2$) down to 1 do

3. MAX-Heapify (A, i)

We can build a heap from an unordered array in linear time.

The HEAPSORT procedure takes time $O(n \log n)$ since the call to BUILD_HEAP takes time $O(n)$ and each of the $n - 1$ calls to MAX-Heapify takes time $O(\log n)$.

Que 1.23. Sort the following array using heap sort techniques : {5, 13, 2, 25, 7, 17, 20, 8, 4}. Discuss its worst case and average case time complexities.

AKTU 2013-14, Marks 05

Answer

Given array is : {5, 13, 2, 25, 7, 17, 20, 8, 4}

First we call Build-Max heap

heap size $[A] = 9$

so, $i = 4$ to 1, call MAX HEAPIFY (A, i)

i.e., first we call MAX HEAPIFY $(A, 4)$

$A[l] = 8, A[i] = 25, A[r] = 4$

$A[i] > A[l]$

$A[i] > A[r]$

Now call MAX HEAPIFY $(A, 3)$

$A[i] = 2, A[l] = 17, A[r] = 20$

$A[l] > A[i]$

$\therefore$ largest = 6

$A[r] > A$ [largest]

$20 > 17$

$\therefore$ largest = 7

largest $\neq i$

$\therefore$ $A[i] \leftrightarrow A$ [largest]

$A[i] > A[l]$

$A[i] > A[r]$

Now call MAX HEAPIFY $(A, 2)$

$A[i] < A[l]$

so, largest = 4

A [largest] $> A[r]$

$\therefore$ $i \neq$ largest, so $A[i] \leftrightarrow A$ [largest]

Now, $A[i] > A[l]$

$A[i] > A[r]$

We call MAX HEAPIFY $(A, 1)$

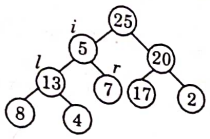
$A[i] < A[l]$

largest = 2

A [largest] $> A[r]$, and largest $\neq i$

$A[i] \leftrightarrow A$ [largest]

largest = 2, so $i = 2$



$A[i] < A[l]$, then largest = 4

A [largest] > A [r], largest ≠ i

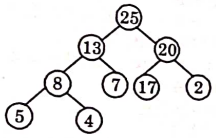
A [i] ↔ A [largest]

Now, $A[i] < A$ [l]

largest = 8, A [largest] > A [r]

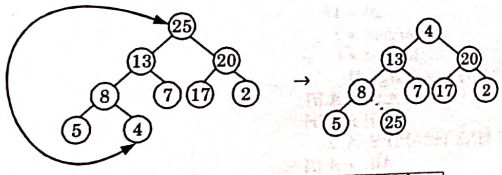
largest ≠ i, A [largest] ↔ A [i]

so final tree after Build MAX HEAPIFY



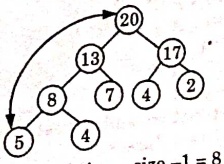
Now i = 9 down to 2, exchange A [1] and A [9] and size = size –1 and call MAX HEAPIFY (A, 1) each time.

exchanging A [1] ↔ A [9]

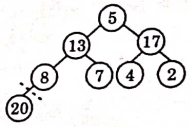


4	13	20	8	7	17	2	5 : 25

Now call MAX HEAPIFY (A, 1) we get

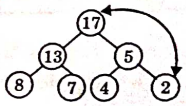


Now exchange A[1] and A[8] and size = size –1 = 8 – 1 = 7



5	13	17	8	7	4	2 : 20

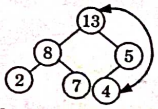
Again call MAX HEAPIFY (A, 1), we get



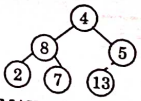
exchange A [1] and A[7] and size = size – 1 = 7 – 1 = 6

2	13	5	8	7	4 : 17

Again call MAX HEAPIFY (A, 1), we get

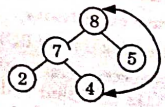


exchange A [1] and A[6] and now size = 6 –1 = 5

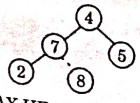


4	8	5	2	7 : 13

Again call MAX HEAPIFY (A, 1)

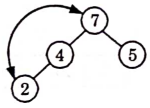


exchange A [1] and A [5] and now size = 5 –1 = 4

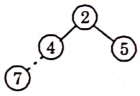


4	7	5	2 : 8

Again, call MAX HEAPIFY (A, 1)

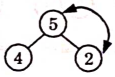


exchange A [1] and A [4] and size = 4–1 = 3



2	4	5	7

call MAX HEAPIFY (A, 1)



exchange A [1] and A [3], size = 3 – 1 = 2

2	4	5

call MAX HEAPIFY (A, 1)

exchange A [1] and A [2] and size = 2 –1 = 1

2	4

Thus, sorted array :

2	4	5	7	8	13	17	20	25

Average case and worst case complexity :

1. We have seen that the running time of BUILD-HEAP is O(n).
2. The heap sort algorithm makes a call to BUILD-HEAP for creating a (max) heap, which will take O(n) time and each of the (n – 1) calls to MAX-HEAPIFY to fix up the new heap (which is created after exchanging the root and by decreasing the heap size).
3. We know 'MAX-HEAPIFY' takes time O(log n).
4. Thus the total running time for the heap sort is O(n log n).

Que 1.24. **What is heap sort ? Apply heap sort algorithm for sorting 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Also deduce time complexity of heap sort.**

AKTU 2015-16, Marks 10

Answer

Heap sort : Refer Q. 1.22, Page 1–24B, Unit-1.

Numerical : Since the given problem is already in sorted form. So, there is no need to apply any procedure on given problem.

Que 1.25. **Explain HEAP SORT on the array. Illustrate the operation HEAP SORT on the array A = {6, 14, 3, 25, 2, 10, 20, 7, 6}**

AKTU 2017-18, Marks 10

Answer

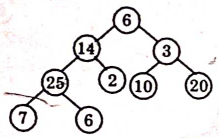
Heap sort : Refer Q. 1.22, Page 1–24B, Unit-1.

Numerical :

Originally the given array is : [6, 14, 3, 25, 2, 10, 20, 7, 6]

First we call Build-Max heap

heap size [A] = 9



so, i = 4 to 1, call MAX HEAPIFY (A, i)

i.e., first we call MAX HEAPIFY (A, 4)

A [l] = 7, A [i] = A [4] = 25, A [r] = 6

$l \leftarrow$ left [4] = 8

$r \leftarrow$ right [4] = 9

$8 \leq 9$ and $7 > 25$ (False)

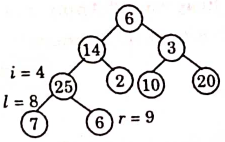
Then, largest $\leftarrow$ 4

$9 \leq 9$ and $6 > 25$ (False)

Then, largest = 4

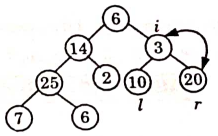
A [i] $\leftrightarrow$ A [4]

Now call MAX HEAPIFY (A, 2)

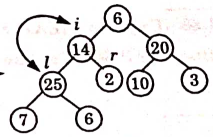


(i)

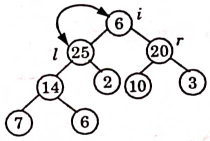
Similarly for $i = 3, 2, 1$ we get the following heap tree.



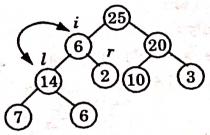
(ii)



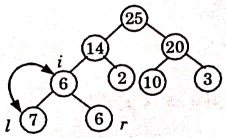
(iii)



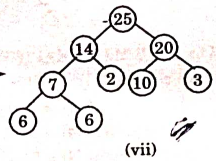
(iv)



(v)

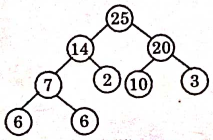


(vi)



(vii)

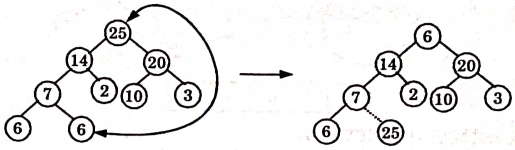
So final tree after BUILD-MAX HEAP is



(viii)

Now $i = 9$ down to 2, and size = size – 1 and call MAX HEAPIFY $(A, 1)$ each time.

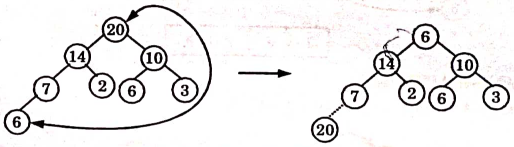
exchanging $A[1] \leftrightarrow A[9]$



6	14	20	7	2	10	3	6	25

Now call MAX HEAPIFY $(A, 1)$ we get

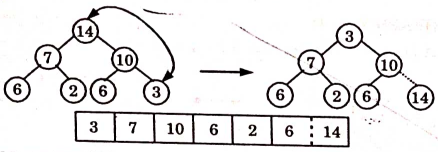
Now exchange $A[1]$ and $A[8]$ and size = 8 – 1 = 7



6	14	10	7	2	6	3	20

Again call MAX HEAPIFY $(A, 1)$, we get

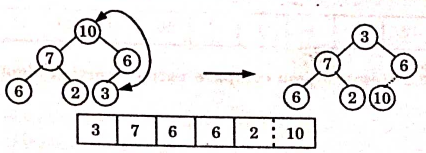
exchange $A[1]$ and $A[7]$ and size = 7 – 1 = 6



3	7	10	6	2	6	14

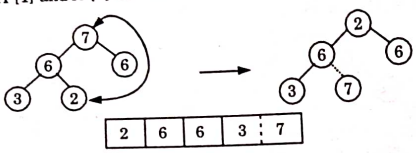
Again call MAX HEAPIFY $(A, 1)$, we get

exchange $A[1]$ and $A[6]$ and now size = 6 – 1 = 5

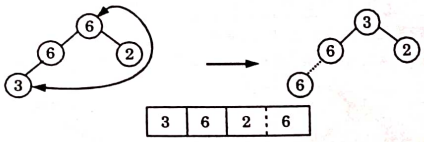


3	7	6	6	2	10

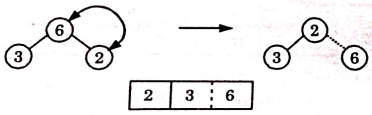
Again call MAX HEAPIFY (A, 1)
exchange A [1] and A [5] and now size = 5 – 1 = 4



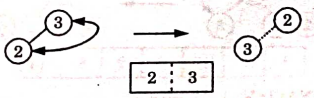
Again, call MAX HEAPIFY (A, 1)
exchange A [1] and A [4] and size = 4 – 1 = 3



call MAX HEAPIFY (A, 1)
exchange A [1] and A [3], size = 3 – 1 = 2



call MAX HEAPIFY (A, 1)
exchange A [1] and A [2] and size = 2 – 1 = 1



Thus, sorted array :

2	3	6	6	7	10	14	20	25

Que 1.26. How will you compare various sorting algorithms ?

Answer

Name	Average case	Worst case	Stable	Method	Other notes
Selection sort	$O(n^2)$	$O(n^2)$	No	Selection	Can be implemented as a stable sort
Insertion sort	$O(n^2)$	$O(n^2)$	Yes	Insertion	Average case is also $O(n + d)$, where d is the number of inversion
Shell sort	–	$O(n \log^2 n)$	No	Insertion	No extra memory required
Merge sort	$O(n \log n)$	$O(n \log n)$	Yes	Merging	Recursive, extra memory required
Heap sort	$O(n \log n)$	$O(n \log n)$	No	Selection	Recursive, extra memory required
Quick sort	$O(n \log n)$	$O(n^2)$	No	Partitioning	Recursive, based on divide conquer technique

Que 1.27. Explain the counting sort algorithm.

Answer

Counting sort is a linear time sorting algorithm used to sort items when they belong to a fixed and finite set.

Algorithm :

Counting_Sort(A, B, k)

1. let $C[0..k]$ be a new array
2. for $i \leftarrow 0$ to k
3. $C[i] \leftarrow 0$
4. for $j \leftarrow 1$ to length[A]
5. do $C[A[j]] \leftarrow C[A[j]] + 1$
   // $C[i]$ now contains the number of elements equal to i.
6. for $i \leftarrow 1$ to k
7. do $C[i] \leftarrow C[i] + C[i - 1]$
   // $C[i]$ now contains the number of elements less than or equal to i.
8. for $j \leftarrow$ length[A] down to 1
9. do $B[C[A[j]]] \leftarrow A[j]$
10. $C[A[j]] \leftarrow C[A[j]] - 1$

Que 1.28. **What is the time complexity of counting sort ? Illustrate the operation of counting sort on array $A = \{1, 6, 3, 3, 4, 5, 6, 3, 4, 5\}$.**

AKTU 2014-15, Marks 10

Answer

Time complexity of counting sort is $O(n)$.

	1	2	3	4	5	6	7	8	9	10
A	1	6	3	3	4	5	6	3	4	5

Step 1 : $i = 0$ to 6 $k = 6$ (largest element in array A)

$C[i] \leftarrow 0$

	0	1	2	3	4	5	6
C	0	0	0	0	0	0	0

Step 2 : $j = 1$ to 10 $\because$ length $[A] = 10$

$C[A[j]] \leftarrow C[A[j]] + 1$

For $j = 1$

	0	1	2	3	4	5	6
C	0	1	0	0	0	0	0

$C[A[1]] \leftarrow C[1] + 1 = 0 + 1 = 1$

$C[1] \leftarrow 1$

For $j = 2$

	0	1	2	3	4	5	6
C	0	1	0	0	0	0	1

$C[A[2]] \leftarrow C[6] + 1 = 0 + 1 = 1$

$C[6] \leftarrow 1$

	0	1	2	3	4	5	6
C	0	1	0	3	2	2	2

Similarly for $j = 5, 6, 7, 8, 9, 10$

Step 3 :

For $i = 1$ to 6

$C[i] \leftarrow C[i] + C[i-1]$

For $i = 1$

	0	1	2	3	4	5	6
C	0	1	0	3	2	2	2

$C[1] \leftarrow C[1] + C[0]$

$C[1] \leftarrow 1 + 0 = 1$

For $i = 2$

	0	1	2	3	4	5	6
C	0	1	1	3	2	2	2

$C[2] \leftarrow C[2] + C[1]$

$C[1] \leftarrow 1 + 0 = 1$

	0	1	2	3	4	5	6
C	0	1	1	4	6	8	10

Similarly for $i = 4, 5, 6$

Step 4 :

For $j = 10$ to 1

$B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

j	$A[j]$	$C[A[j]]$	$B[C[A[j]]] \leftarrow A[j]$	$C[A[j]] \leftarrow C[A[j]] - 1$
10	5	8	$B[8] \leftarrow 5$	$C[5] \leftarrow 7$
9	4	6	$B[6] \leftarrow 4$	$C[4] \leftarrow 5$
8	3	4	$B[4] \leftarrow 3$	$C[3] \leftarrow 3$
7	6	10	$B[10] \leftarrow 6$	$C[6] \leftarrow 9$
6	5	7	$B[7] \leftarrow 5$	$C[5] \leftarrow 6$
5	4	5	$B[5] \leftarrow 4$	$C[4] \leftarrow 4$
4	3	3	$B[3] \leftarrow 3$	$C[3] \leftarrow 2$
3	3	2	$B[2] \leftarrow 3$	$C[3] \leftarrow 1$
2	6	9	$B[9] \leftarrow 6$	$C[6] \leftarrow 8$
1	1	1	$B[1] \leftarrow 1$	$C[1] \leftarrow 0$

	1	2	3	4	5	6	7	8	9	10
B	1	3	3	3	4	4	5	5	6	6

Que 1.29. **Write the bucket sort algorithm.**

Answer

1. The bucket sort is used to divide the interval [0, 1] into n equal-sized sub-intervals, or bucket, and then distribute the n-input numbers into the bucket.
2. Since the inputs are uniformly distributed over [0, 1], we do not except many numbers to fall into each bucket.
3. To produce the output, simply sort the numbers in each bucket and then go through the bucket in order, listing the elements in each.
4. The code assumes that input is in n-element array A and each element in A satisfies $0 \le A[i] \le 1$. We also need an auxiliary array $B[0 \ldots n-1]$ for linked-list (buckets).

BUCKET_SORT (A)

1. $n \leftarrow$ length [A]
2. For $i \leftarrow 1$ to n
3. do Insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. For $i \leftarrow 0$ to $n-1$
5. do Sort list $B[i]$ with insertion sort
6. Concatenate the lists $B[0], B[1], \ldots B[n-1]$ together in order.

Que 1.30. **What do you mean by stable sort algorithms ? Explain it with suitable example.**

Answer

1. A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input sorted array.
2. A stable sort is one where the initial order of equal items is preserved.
3. Some sorting algorithms are naturally stable, some are unstable, and some can be made stable with care.
4. Stability is important when we want to sort by multiple fields, for example, sorting a list of task assignments first by priority and then by assignee (in other words, assignments of equal priority are sorted by assignee).
5. One easy way to do this is to sort by assignee first, then take the resulting sorted list and sort that by priority.
6. This only works if the sorting algorithm is stable otherwise; the sortedness-by-assignee of equal-priority items is not preserved.
7. For example, if sort the words "apple", "tree" and "pink" by length, then "tree", "pink", "apple" is a stable sort but "pink", "tree", apple" is not.
8. Merge sort is a very common choice of stable sorts, achieved by favouring the leftmost item in each merge step (only if item_right < item_left put item_right first).
9. Radix sort is another of the stable sorting algorithms.

Que 1.31. **Write a short note on radix sort.**

Answer

1. Radix sort is a sorting algorithm which consists of list of integers or words and each has d-digit.
2. We can start sorting on the least significant digit or on the most significant digit.
3. On the first pass entire numbers sort on the least significant digit (or most significant digit) and combine in a array.
4. Then on the second pass, the entire numbers are sorted again on the second least significant digits and combine in an array and so on.

RADIX_SORT (A, d)

1 to d do

2. use a stable sort to sort array A on digit i

   // counting sort will do the job

   The code for radix sort assumes that each element in the n-element array A has d-digits, where digit 1 is the lowest-order digit and d is the highest-order digit.

Analysis :

1. The running time depends on the table used as an intermediate sorting algorithm.
2. When each digit is in the range 1 to k, and k is not too large, COUNTING_SORT is the obvious choice.
3. In case of counting sort, each pass over n d-digit numbers takes $\Theta(n + k)$ time.
4. There are d passes, so the total time for radix sort is $\theta(n + k)$ time. There are d passes, so the total time for radix sort is $\Theta(dn + kd)$. When d is constant and $k = \Theta(n)$, the radix sort runs in linear time.

   For example : This example shows how radix sort operates on seven 3-digit number.

Table 1.31.1.

Input	1st pass	2nd pass	3rd pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

In the table 1.31.1, the first column is the input and the remaining shows the list after successive sorts on increasingly significant digits position.

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. **What do you mean by algorithm ? Write its characteristics.**

Ans: Refer Q. 1.1.

Q. 2. **Write short note on asymptotic notations.**

Ans. Refer Q. 1.3.

Q. 3. Explain shell sort with example.
Ans. Refer Q. 1.15.

Q. 4. Discuss quick sort method and analyze its complexity.
Ans. Refer Q. 1.18.

Q. 5. Explain the concept of merge sort with example.
Ans. Refer Q. 1.20.

Q. 6. Write short note on heap sort algorithm with its analysis.
Ans. Refer Q. 1.22.

Q. 7. What is radix sort ?
Ans. Refer Q. 1.31.

2 UNIT

Advanced Data Structure

PART-1

Red-Black Trees, B-trees.

CONCEPT OUTLINE : PART-1

- **Red-black tree :**
  a. A red-black tree is a binary tree where each node has colour as an extra attribute, either red or black.
  b. It is a type of self-balancing binary search tree.
- **B-tree :**
  a. B-tree is a tree data structure that keeps data sorted and allows insertion and deletion in logarithmic amortized time.
  b. In B-trees, internal nodes can have a variable number of child nodes within some predefined range.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.1. Define a red-black tree with its properties. Explain the insertion operation in a red-black tree.

Answer

Red-black tree :

A red-black tree is a binary tree where each node has colour as an extra attribute, either red or black. It is a self-balancing Binary Search Tree (BST) where every node follows following properties :

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendent leave contain the same number of black nodes.

Insertion :

i. We begin by adding the node as we do in a simple binary search tree and colouring it red.

RB-INSERT(T, z)

```
1.  y ← nil [T]
2.  x ← root [T]
3.  while x ≠ nil [T]
4.  do y ← x
5.  if key[z] < key [x]
6.  then x ← left [x]
7.  else x ← right [x]
8.  p[z] ← y
9.  if y = nil [T]
10. then root [T] ← z
11. else if key [z] < key[y]
12. then left [y]← z
13. else right [y] ← z
14. left [z] ← nil[T]
15. right [z] ← nil[T]
16. colour [z] ← RED
17. RB-INSERT-FIXUP(T, z)
```

ii. Now, for any colour violation, RB-INSERT-FIXUP procedure is used.

RB-INSERT-FIXUP(T, z)

```
1.  while colour [p[z]] = RED
2.  do if p[z] = left[p [p [z]]]
3.  then y ← right[p [p [z]]]
4.  if colour[y] = RED
5.  then colour[p [z]] ← BLACK                    ⇒ case 1
6.  colour[y] ← BLACK                             ⇒ case 1
7.  colour[p [p [z]]]← RED                        ⇒ case 1
8.  z ← p [p [z]]                                 ⇒ case 1
9.  else if z = right[p[z]]
10. then z ← p[z]                                 ⇒ case 2
11. LEFT-ROTATE(T, z)                             ⇒ case 2
12. colour[p[z]] ← BLACK                          ⇒ case 3
13. colour[p[p[z]]] ← RED                         ⇒ case 3
14. RIGHT-ROTATE(T, p[p[z]])                      ⇒ case 3
15. else (same as then clause with "right" and "left" exchanged)
16. colour[root[T]] ← BLACK
```

Cases of RB-tree for insertion :

Case 1 : z's uncle is red :

$P[z] = \text{left}[p[p[z]]]$

then uncle ← $\text{right}[p[p[z]]]$

a. change z's grandparent to red.

b. change z's uncle and parent to black.

c. change z to z's grandparent.

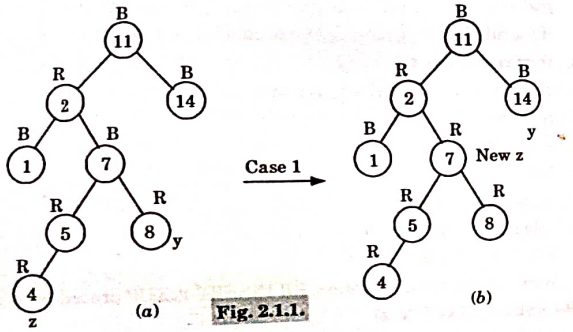


Fig. 2.1.1.

Now, in this case violation of property 4 occurs, because z's uncle y is red, then case 1 is applied.

Case 2 : z's uncle is black, z is the right of its parent :

a. Change z to z's parent.

b. Rotate z's parent left to make case 3.

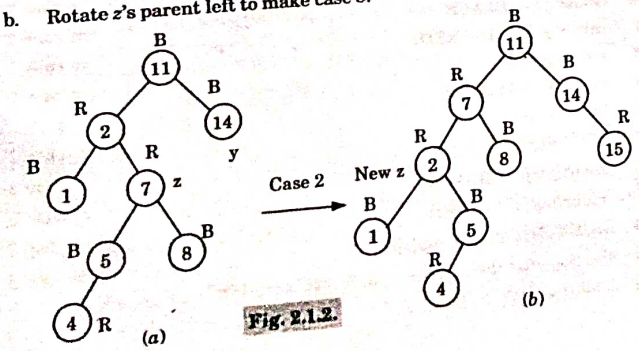


Fig. 2.1.2.

Case 3 : z's uncle is black, z is the left child of its parent :

a. Set z's parent black.

b. Set z's grandparent to red.

c. Rotate z's grandparent right.

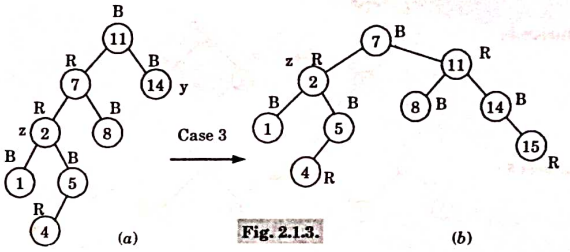


Fig. 2.1.3.

Que 2.2. **What are the advantages of red-black tree over binary search tree ? Write algorithms to insert a key in a red-black tree insert the following sequence of information in an empty red-black tree 1, 2, 3, 4, 5, 5.** **AKTU 2014-15, Marks 10**

Answer

Advantages of RB-tree over binary search tree :

1. The main advantage of red-black trees over AVL trees is that a single top-down pass may be used in both insertion and deletion operations.
2. Red-black trees are self-balancing while on the other hand, simple binary search trees are unbalanced.
3. It is particularly useful when inserts and/or deletes are relatively frequent.
4. Time complexity of red-black tree is $O(\log n)$ while on the other hand, a simple BST has time complexity of $O(n)$.

Algorithm to insert a key in a red-black tree : Refer Q. 2.1, Page 2–2B, Unit-2.

Numerical :

Insert 1 :

Insert 2 :

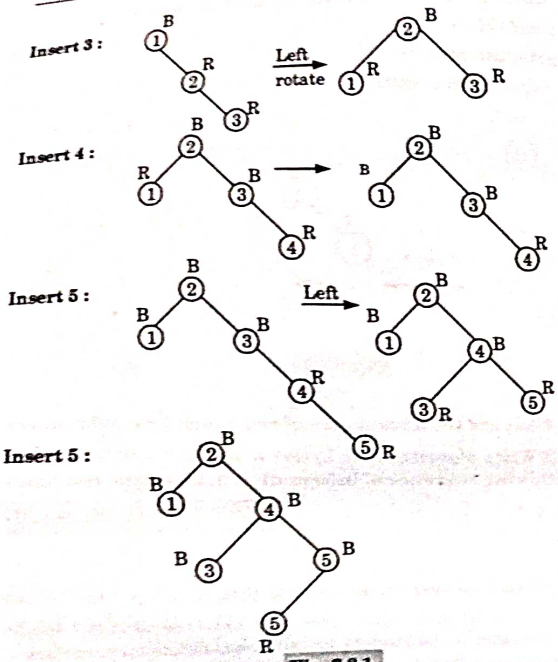


Fig. 2.2.1.

Que 2.3. Explain red-black tree. Show steps of inserting the keys 41, 38, 31, 12, 19, 8 into initially empty red-black tree.

AKTU 2013-14, Marks 10

OR

What is red-black tree ? Write an algorithm to insert a node in an empty red-black tree explain with suitable example.

AKTU 2017-18, Marks 10

Answer

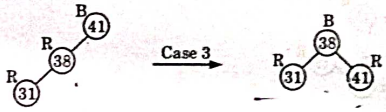
Red-black tree : Refer Q. 2.1, Page 2–2B, Unit-2.

Numerical :

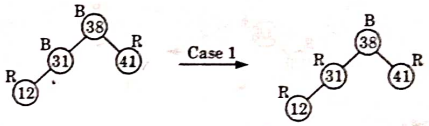
Insert 41 :

Insert 38 :

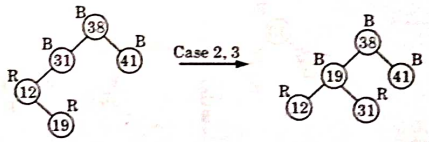
Insert 31 :



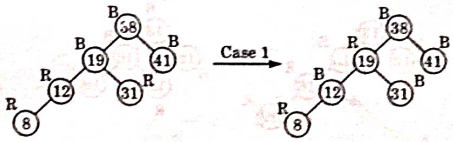
Insert 12 :



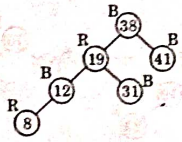
Insert 19 :



Insert 8 :



Thus final tree is



Que 2.4. **Insert the nodes 15, 13, 12, 16, 19, 23, 5, 8 in empty red-black tree and delete in the reverse order of insertion.**

AKTU 2016-17, Marks 10

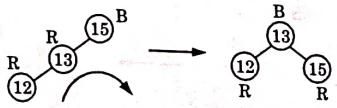
Answer

Insertion :

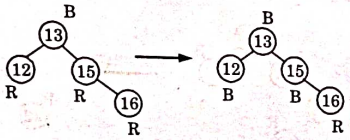
Insert 15 :

Insert 13 :

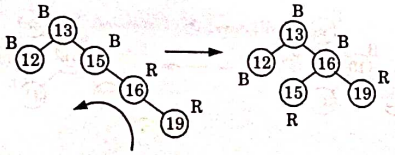
Insert 12 :



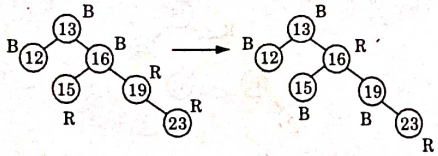
Insert 16 :



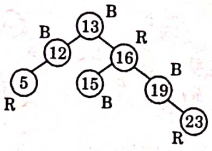
Insert 19 :



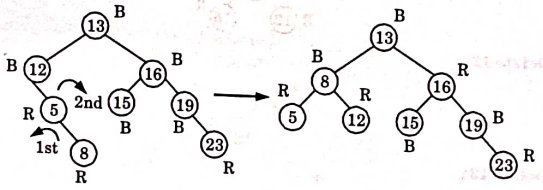
Insert 23 :



Insert 5 :

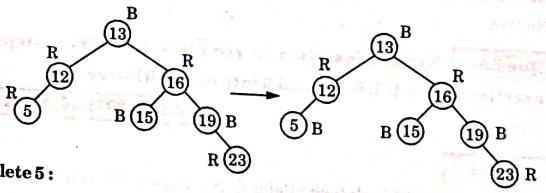


Insert 8 :

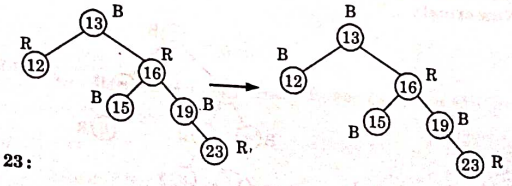


Deletion :

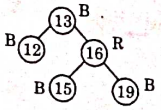
Delete 8 :



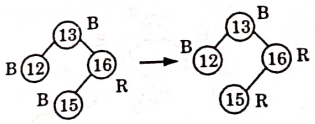
Delete 5 :



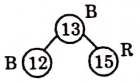
Delete 23 :



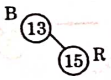
Delete 19 :



Delete 16 :



Delete 12 :



Delete 13 :

Delete 15 :

No tree

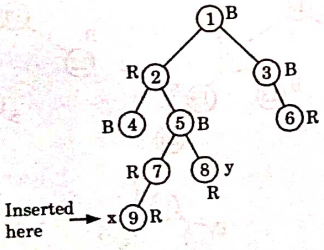
Que 2.5. **Explain insertion in red-black tree. Show steps for inserting 1, 2, 3, 4, 5, 6, 7, 8 and 9 into empty RB-tree.**

AKTU 2015-16, Marks 10

Answer

Insertion in red-black tree : Refer Q. 2.1, Page 2–2B, Unit-2.

Numerical :



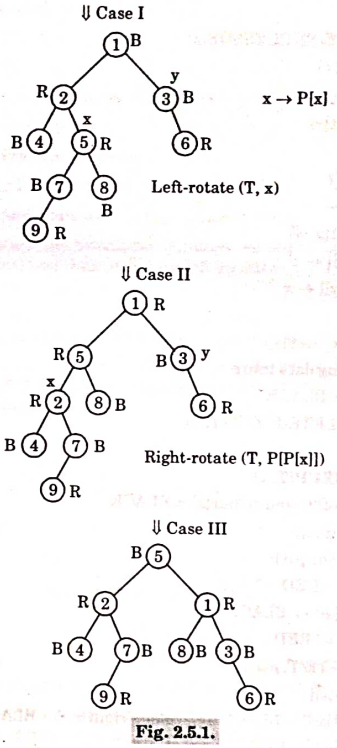


Fig. 2.5.1.

Que 2.6. **How to remove a node from RB-tree ? Discuss all cases and write down the algorithm.**

Answer

In RB-DELETE procedure, after splitting out a node, it calls an auxiliary procedure RB-DELETE-FIXUP that changes colours and performs rotations to restore the red-black properties.

RB-DELETE(T, z)

1. if left[z] = nil[T] or right[z] = nil[T]

2. then $y \leftarrow z$
3. else $y \leftarrow$ TREE-SUCCESSOR(z)
4. if left[y] $\neq$ nil[T]
5. then $x \leftarrow$ left[y]
6. else $x \leftarrow$ right[y]
7. $p[x] \leftarrow p[y]$
8. if $p[y]$ = nil[T]
9. then root[T] $\leftarrow x$
10. else if y = left[$p[y]$]
11. then left[$p[y]$] $\leftarrow x$
12. else right[$p[y]$] $\leftarrow x$
13. if $y \neq z$
14. then key[z] $\leftarrow$ key[y]
15. copy y's sibling data into z
16. if colour[y] = BLACK
17. then RB-DELETE-FIXUP(T, x)
18. return y

RB-DELETE-FIXUP(T, x)

1. while $x \neq$ root[T] and colour[x] = BLACK
2. do if x = left[$p[x]$]
3. then $w \leftarrow$ right[$p[x]$]
4. if colour[w] = RED
5. then colour[w] $\leftarrow$ BLACK ⇒ case 1
6. colour[$p[x]$] $\leftarrow$ RED ⇒ case 1
7. LEFT-ROTATE(T, $p[x]$) ⇒ case 1
8. $w \leftarrow$ right[$p[x]$] ⇒ case 1
9. if colour[left[w]] = BLACK and colour[right[w]] = BLACK
10. then colour[w] $\leftarrow$ RED ⇒ case 2
11. $x \leftarrow p[x]$ ⇒ case 2
12. else if colour[right[w]] = BLACK
13. then colour[left[w]] $\leftarrow$ BLACK ⇒ case 3
14. colour[w] $\leftarrow$ RED ⇒ case 3
15. RIGHT-ROTATE(T, w) ⇒ case 3
16. $w \leftarrow$ right[$p[x]$] ⇒ case 3
17. colour[w] $\leftarrow$ colour[$p[x]$] ⇒ case 4
18. colour[$p[x]$] $\leftarrow$ BLACK ⇒ case 4

19. colour[right[w]] $\leftarrow$ BLACK ⇒ case 4
20. LEFT-ROTATE(T, $p[x]$) ⇒ case 4
21. $x \leftarrow$ root[T] ⇒ case 4
22. else (same as then clause with "right" and "left" exchanged).
23. colour[x] $\leftarrow$ BLACK

Cases of RB-tree for deletion :

Case 1 : x's sibling w is red :

1. It occurs when node w the sibling of node x, is red.
2. Since w must have black children, we can switch the colours of w and $p[x]$ and then perform a left-rotation on $p[x]$ without violating any of the red-black properties.
3. The new sibling of x, which is one of w's children prior to the rotation, is now black, thus we have converted case 1 into case 2, 3 or 4.
4. Case 2, 3 and 4 occur when node w is black. They are distinguished by colours of w's children.

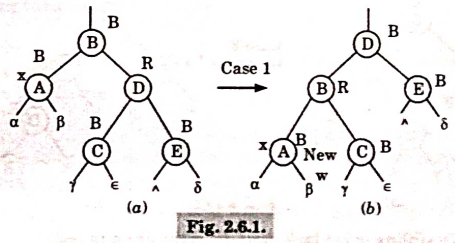


Fig. 2.6.1.

Case 2 : x's sibling w is black, and both of w's children are black :

1. Both of w's children are black. Since w is also black, we take one black of both x and w, leaving x with only one black and leaving w red.
2. For removing one black from x and w, we add an extra black to $p[x]$, which was originally either red or black.
3. We do so by repeating the while loop with $p[x]$ as the new node x.
4. If we enter in case 2 through case 1, the new node x is red and black, the original $p[x]$ was red.
5. The value c of the colour attribute of the new node x is red, and the loop terminates when it tests the loop condition. The new node x is then coloured black.

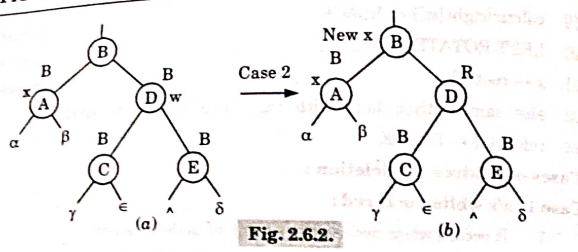


Fig. 2.6.2.

Case 3 : x's sibling w is black, w's left child is red, and w's right child is black :

1. Case 3 occurs when w is black, its left child is red and its right child is black.
2. We can switch the colours of w and its left child left[w] and then perform a right rotation on w without violating any of the red-black properties, the new sibling w of x is a black node with a red right child and thus we have transformed case 3 into case 4.

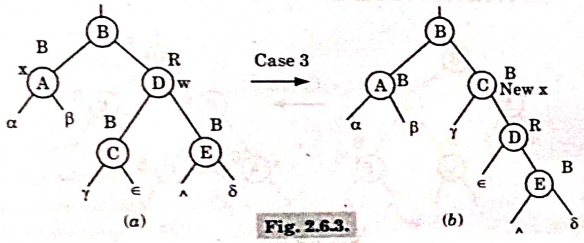


Fig. 2.6.3.

Case 4 : x's sibling w is black, and w's right child is red :

1. When node x's sibling w is black and w's right child is red.
2. By making some colour changes and performing a left rotation on $p[x]$, we can remove the extra black on x, making it singly black, without violating any of the red-black properties.

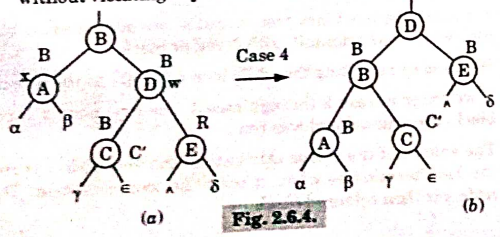


Fig. 2.6.4.

Que 2.7. Describe the properties of red-black tree. Show the red-black tree with n internal nodes has height at most $2 \log (n + 1)$.

AKTU 2014-15, Marks 05

OR

Prove the height h of a red-black tree with n internal nodes is not greater than $2 \log (n + 1)$.

Answer

Properties of red-black tree : Refer Q. 2.1, Page 2–2B, Unit-2.

1. By property 5 of RB-tree, every root-to-leaf path in the tree has the same number of black nodes, let this number be B.
2. So there are no leaves in this tree at depth less than B, which means the tree has at least as many internal nodes as a complete binary tree of height B.
3. Therefore, $n \leq 2^B - 1$. This implies $B \leq \log (n + 1)$.
4. By property 4 of RB-tree, at most every other node on a root-to-leaf path is red. Therefore, $h \leq 2B$.

   Putting these together, we have

   $$h \leq 2 \log (n + 1).$$

Que 2.8. Define a B-tree of order m. Explain the searching operation in a B-tree.

Answer

A B-tree of order m is an m-ary search tree with the following properties :

1. The root is either leaf or has atleast two children.
2. Each node, except for the root and the leaves, has between $m/2$ and m children.
3. Each path from the root to a leaf has the same length.
4. The root, each internal node and each leaf is typically a disk block.
5. Each internal node has upto $(m - 1)$ key values and upto m children.

Searching operation in a B-tree :

We adopt following convention to all operation :

a. Root is always in main memory.

b. Nodes passed to operations must have been read.

c. All operations go from root down in one pass, $O(h)$.

SEARCH(x, k)

1. $i \leftarrow 1$

2. while $i \le n[x]$ and $k > key_i[x]$
3. do $i \leftarrow i + 1$
4. if $i \le n[x]$ and $k = key_i[x]$
5. then return(x, i)
6. if leaf[x]
7. then return NIL
8. else DISK-READ($c_i[x]$)
9. return B-TREE-SEARCH ($c_i[x]$, k)

The number of disk pages accessed by B-TREE-SEARCH is $\theta(n) = \theta(\log_t n)$, where h is the height of the tree and n is the number of keys in the tree. Since $n[x] < 2t$, time taken by the while loop of lines 2-3 within each nodes is $O(t)$ and the total CPU time is $O(th) = O(t \log_t n)$

B-TREE-INSERT(T, k)

1. $r \leftarrow$ root[T]
2. if $n[r] = 2t - 1$
3. then $s \leftarrow$ ALLOCATE-NODE ( )
4. root[T] $\leftarrow S$
5. leaf[s] $\leftarrow$ FALSE
6. $n[s] \leftarrow 0$
7. $c_1[s] \leftarrow r$
8. B-TREE SPLIT CHILD(S, l, r)
9. B-TREE-INSERT-NONFULL(s, k)
10. else B-TREE-INSERT-NONFULL(r, k)

B-TREE SPLIT CHILD(x, i, y)

1. $z \leftarrow$ ALLOCATE-NODE ( )
2. leaf[z] $\leftarrow$ leaf[y]
3. $n[z] \leftarrow t - 1$
4. for $j \leftarrow 1$ to $t - 1$
5. do $key_j[z] \leftarrow key_{j+t}[y]$
6. if not leaf[y]
7. then for $j \leftarrow 1$ to t
8. do $c_j[z] \leftarrow c_{j+t}[y]$
9. $n[y] \leftarrow t - 1$

10. for $j \leftarrow n[x] + 1$ down to $i + 1$
11. do $c_{j+1}[x] \leftarrow c_j[x]$
12. $c_{i+1}[x] \leftarrow z$
13. for $j \leftarrow n[x]$ down to i
14. do $key_{j+1}[x] \leftarrow key_j[x]$
15. $key_i[x] \leftarrow key_i[y]$
16. $n[x] \leftarrow n[x] + 1$
17. DISK-WRITE[y]
18. DISK-WRITE[z]
19. DISK-WRITE[x]

The CPU time used by B-TREE SPLIT CHILD is $\theta(t)$. The procedure performs $\theta(1)$ disk operations.

B-TREE-INSERT-NONFULL(x, k)

1. $i \leftarrow n[x]$
2. if leaf[x]
3. then while $i \ge 1$ and $k < key_i[x]$
4. do $key_{i+1}[x] \leftarrow key_i[x]$
5. $i \leftarrow i - 1$
6. $key_{i+1}[x] \leftarrow k$
7. $n[x] \leftarrow n[x] + 1$
8. DISK-WRITE(x)
9. else while $i \ge 1$ and $k < key_i[x]$
10. do $i \leftarrow i - 1$
11. $i \leftarrow i + 1$
12. DISK-READ($c_i[x]$)
13. if $n[c_i[x]] = 2t - 1$
14. then B-TREE-SPLIT-CHILD($x, i, c_i[x]$)
15. if $k > key_i[x]$
16. then $i \leftarrow i + 1$
17. B-TREE INSERT NONFULL($c_i[x], k$)

The total CPU time use is $O(th) = O(t \log_t n)$

Que 2.9. **What are the characteristics of B-tree ? Write down the steps for insertion operation in B-tree.**

Answer

Characteristic of B-tree :

1. Each node of the tree, except the root node and leaves has at least $m/2$ subtrees and no more than m subtrees.
2. Root of tree has at least two subtree unless it is a leaf node.
3. All leaves of the tree are at same level.

Insertion operation in B-tree :

In a B-tree, the new element must be added only at leaf node. The insertion operation is performed as follows :

Step 1 : Check whether tree is empty.

Step 2 : If tree is empty, then create a new node with new key value and insert into the tree as a root node.

Step 3 : If tree is not empty, then find a leaf node to which the new key value can be added using binary search tree logic.

Step 4 : If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.

Step 5 : If that leaf node is already full, then split that leaf node by sending middle value to its parent node. Repeat the same until sending value is fixed into a node.

Step 6 : If the splitting is occurring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Que 2.10. **Describe a method to delete an item from B-tree.**

Answer

There are three possible cases for deletion in B-tree as follows :

Let k be the key to be deleted, x be the node containing the key.

Case 1 : If the key is already in a leaf node, and removing it does not cause that leaf node to have too few keys, then simply remove the key to be deleted. Key k is in node x and x is a leaf, simply delete k from x.

Case 2 : If key k is in node x and x is an internal node, there are three cases to consider :

a. If the child y that precedes k in node x has at least t keys (more than the minimum), then find the predecessor key k' in the subtree rooted at y. Recursively delete k' and replace k with k' in x.

b. Symmetrically, if the child z that follows k in node x has at least t keys, find the successor k' and delete and replace as before.

c. Otherwise, if both y and z have only $t - 1$ (minimum number) keys, merge k and all of z into y, so that both k and the pointer to z are removed from x, y now contains $2t - 1$ keys, and subsequently k is deleted.

Case 3 : If key k is not present in an internal node x, determine the root of the appropriate subtree that must contain k. If the root has only $t - 1$ keys, execute either of the following two cases to ensure that we descend to a node containing at least t keys. Finally, recurse to the appropriate child of x.

a. If the root has only $t - 1$ keys but has a sibling with t keys, give the root an extra key by moving a key from x to the root, moving a key from the roots immediate left or right sibling up into x, and moving the appropriate child from the sibling to x.

b. If the root and all of its siblings have $t - 1$ keys, merge the root with one sibling. This involves moving a key down from x into the new merged node to become the median key for that node.

Que 2.11. **How B-tree differs with other tree structures ? Insert the following information $F, S, Q, K, C, L, V, W, M, R, N, P, A, I, Z, E$, into an empty B-tree with degree $t = 2$.** **AKTU 2014-15, Marks 10**

Answer

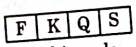
1. In B-tree, the maximum number of child nodes a non-terminal node can have is m where m is the order of the B-tree. On the other hand, other tree can have at most two subtrees or child nodes.
2. B-tree is used when data is stored in disk whereas other tree is used when data is stored in fast memory like RAM.
3. B-tree is employed in code indexing data structure in DBMS, while, other tree is employed in code optimization, Huffman coding, etc.
4. The maximum height of a B-tree is $\log mn$ (m is the order of tree and n is the number of nodes) and maximum height of other tree is $\log_2 n$ (base is 2 because it is for binary).
5. A binary tree is allowed to have zero nodes whereas any other tree must have atleast one node. Thus binary tree is really a different kind of object than any other tree.

Numerical :

$$t = 2$$
$$2t - 1 = 2 \times 2 - 1 = 3$$
$$t - 1 = 2 - 1 = 1$$

So, maximum of 3 keys and minimum of 1 key can be inserted in a node.
Now, apply insertion process as :

Insert *F, S, Q, K* :



As, there are more than 3 keys in this node.

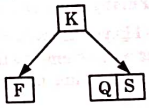
∴ Find median,

$n[x] = 4$ (even)

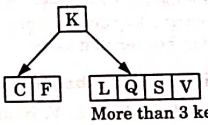
$$\text{Median} = \frac{n[x]}{2} = \frac{4}{2} = 2$$

Now, median = 2,

So, we split the node by 2nd key.

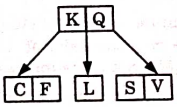


Insert *C, L, V* :

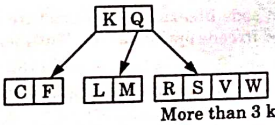


More than 3 keys
split the node from median

$$\text{Median} = \frac{n[x]}{2} = \frac{4}{2} = 2$$ (*i.e.*, 2nd key move up)

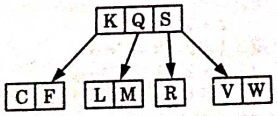


Insert *W, M, R* :

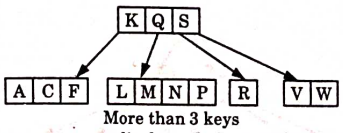


More than 3 keys
split the node from median

$$\text{Median} = \frac{n[x]}{2} = \frac{4}{2} = 2$$ (*i.e.*, 2nd key move up)

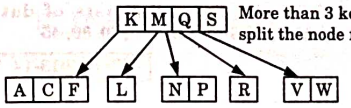


Insert *N, P, A* :



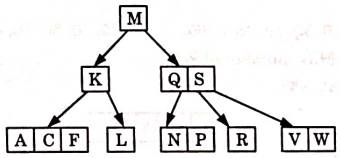
More than 3 keys
split the node from median

$$\text{Median} = \frac{n[x]}{2} = \frac{4}{2} = 2$$ (*i.e.*, 2nd key move up)

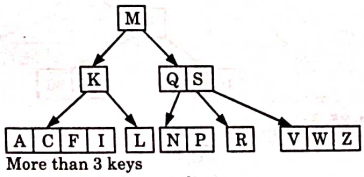


More than 3 keys
split the node from median

$$\text{Median} = \frac{n[x]}{2} = \frac{4}{2} = 2$$ (*i.e.*, 2nd key move up)

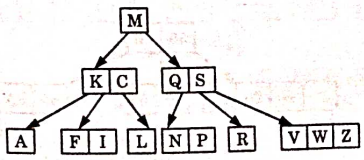


Insert *I, Z* :

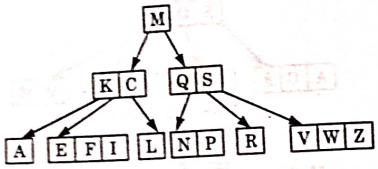


More than 3 keys
split the node from median

$$\text{Median} = \frac{n[x]}{2} = \frac{4}{2} = 2$$ (*i.e.*, 2nd key move up)



Insert *E* :



Que 2.12. **Write the characteristics of a B-tree of order *m*. Create B-tree of order 5 from the following lists of data items : 20, 30, 35, 85, 10, 55, 60, 25, 5, 65, 70, 75, 15, 40, 50, 80, 45.**

AKTU 2013-14, Marks 10

Answer

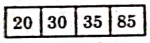
Characteristics of B-tree : Refer Q. 2.9, Page 2–18B, Unit-2.

Numerical :

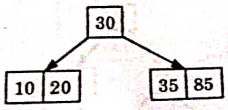
20, 30, 35, 85, 10, 55, 60, 25, 5, 65, 70, 75, 15, 40, 50, 80, 45

∵ order = 5, Max. number of keys = 4

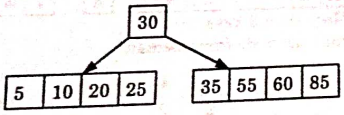
Insert 20, 30, 35, 85 :



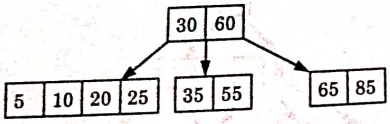
Insert 10 :



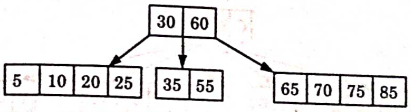
Insert 55, 60, 25, 5 :



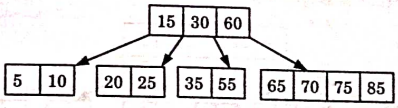
Insert 65 :



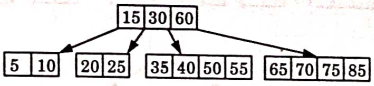
Insert 70, 75 :



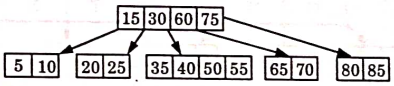
Insert 15 :



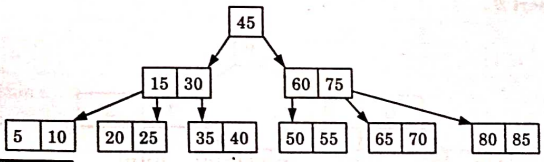
Insert 40, 50 :



Insert 80 :



Insert 45 :



Que 2.13. **Explain B-tree and insert elements *B*, *Q*, *L*, *F* into B-tree Fig. 2.13.1 then apply deletion of elements *F*, *M*, *G*, *D*, *B* on resulting B-tree.**

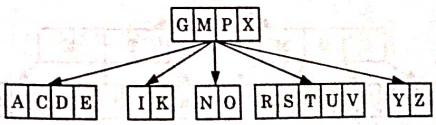


Fig. 2.13.1.

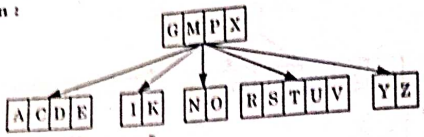
AKTU 2015-16, Marks 10

Answer

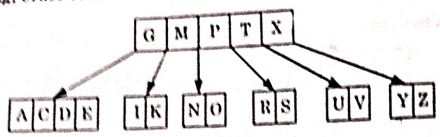
B-tree : Refer Q. 2.8, Page 2–15B, Unit-2.

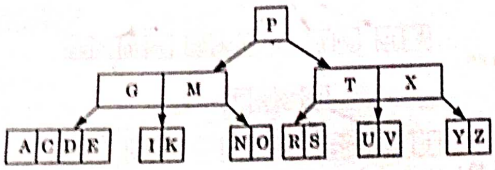
Numerical :

Insertion :

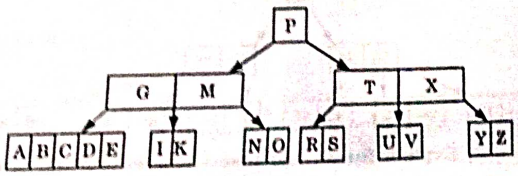


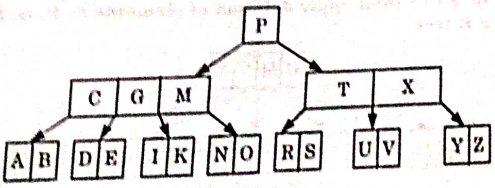
Assuming, order of B-tree = 5



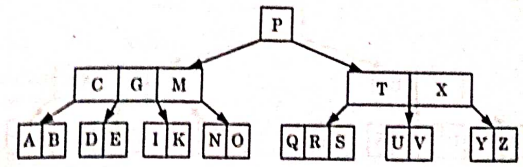


Insert *B* :

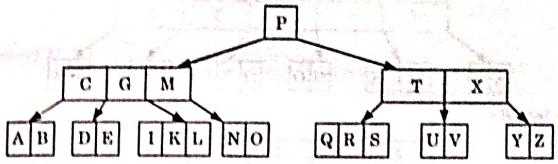




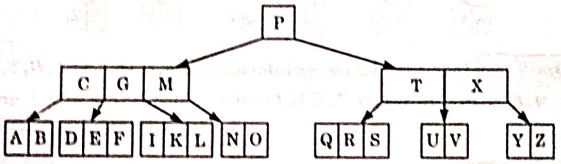
Insert *Q* :



Insert *L* :

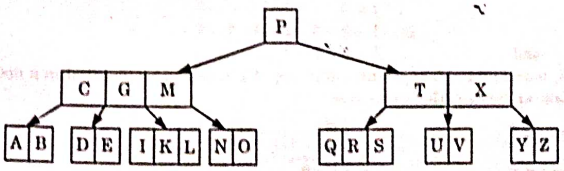


Insert *F* :

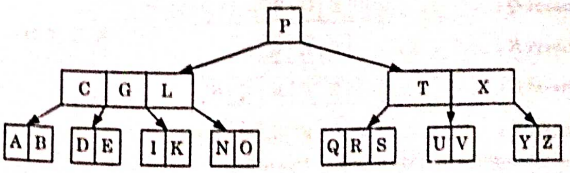


Deletion :

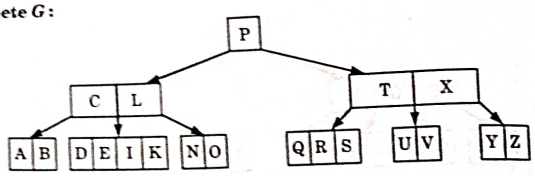
Delete *F* :



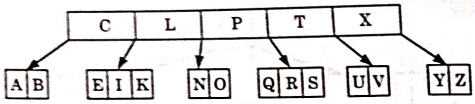
Delete *M* :



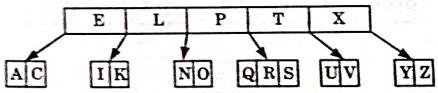
Delete *G*:



Delete *D*:



Delete *B*:



Que 2.14. **Insert the following information, *F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E, G, I* into an empty B-tree with degree *t* = 3.** **AKTU 2017-18, Marks 10**

Answer

Assume that

$$t = 3$$

$$2t - 1 = 2 \times 3 - 1 = 6 - 1 = 5$$

and $$t - 1 = 3 - 1 = 2$$

So, maximum of 5 keys and minimum of 2 keys can be inserted in a node.

Now, apply insertion process as :

Insert *F*: F

Insert *S*: F S

Insert *Q*: F Q S

Insert *K*: F K Q S

Insert *C*: C F K Q S

Insert *L*: C F K L Q S

As, there are more than 5 keys in this node.

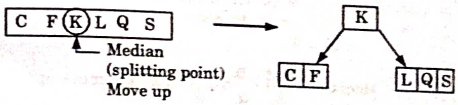
∴ Find median,

$$n[x] = 6 \text{ (even)}$$

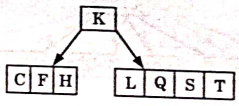
$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3$$

Now, median = 3,

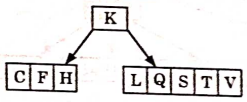
So, we split the node by 3rd key.



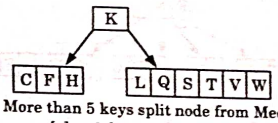
Insert *H, T*:



Insert *V*:



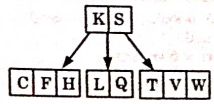
Insert *W*:



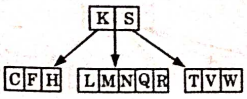
More than 5 keys split node from Median.

$n[x] = 6$ [even]

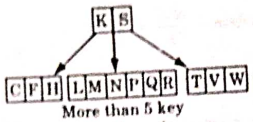
$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3$$ (*i.e.*, 3rd key move up)



Insert *M, R, N*:



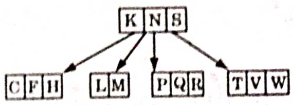
Insert *P* :



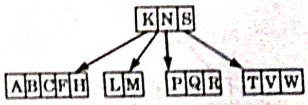
More than 5 key split the node

$n[x] = 6$

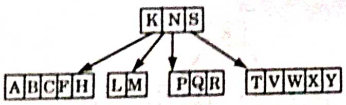
$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3 \text{ (i.e., 3rd key move up)}$$



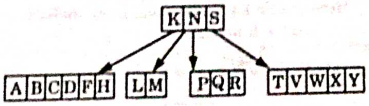
Insert *A*, *B* :



Insert *X*, *Y* :



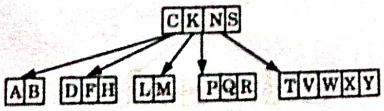
Insert *D* :



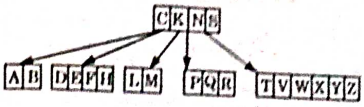
More than 5 key split the node

$n[x] = 6$ (even)

$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3 \quad \text{(i.e., 3rd key move up)}$$



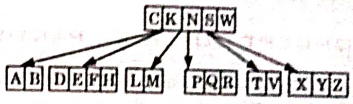
Insert *Z*, *E* :



More than 5 key split the node

$n[x] = 6$

$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3 \quad \text{(i.e., 3rd key move up)}$$



Insert *G*, *I* :

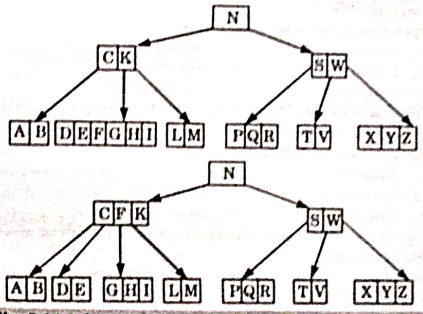


Fig. 2.14.1. Inserted all given information with degree t = 3.

Que 2.15. If $n \geq 1$, then for any n-key B-tree of height h and minimum degree $t \geq 2$, Prove that : $h \leq \log_t \frac{(n+1)}{2}$.

Answer

Proof :

1. The root contains at least one key.
2. All other nodes contain at least $t - 1$ keys.
3. There are at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^{i-1}$ nodes at depth i and $2t^{h-1}$ nodes at depth h.

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

$$= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1$$

4. So $t^h \leq (n+1)/2$ as required.
   Taking log both sides we get,
   $h \leq \log_t (n+1)/2$

PART-2

Binomial Heaps, Fibonacci Heaps, Tries, Skip List.

CONCEPT OUTLINE : PART-2

- **Binomial heap :** A binomial heap is a data structure similar to binary heap but also supporting the operation of merging two heaps quickly.
- **Operations on binomial heap :**
  i. Creation ii. Searching
  iii. Union iv. Insertion
  v. Removal vi. Decreasing
- **Fibonacci heap :** They are linked lists of heap-ordered trees. It is also a collection of trees.
- **Operations on Fibonacci heap :**
  i. Insertion ii. Union
  iii. Extraction iv. Linking
  v. Deletion iv. Decreasing
- Tries is a kind of search tree used to store dynamic or associative sets of strings.
- Skip list is a layered linked list data structure.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.16. Explain binomial heap and properties of binomial tree.

Answer

Binomial heap :

1. Binomial heap is a type of data structure which keeps data sorted and allows insertion and deletion in amortized time.
2. A binomial heap is implemented as a collection of binomial tree.

Properties of binomial tree :

1. The total number of node at order k are 2^k.
2. The height of the tree is k.
3. There are exactly $\binom{k}{i}$ *i.e.*, ${}^kC_i$ nodes at depth i for $i = 0, 1, \ldots, k$ (this is why the tree is called a "binomial" tree).
4. Root has degree k (children) and its children are $B_{k-1}, B_{k-2}, \ldots, B_0$ from left to right.

Que 2.17. What is a binomial heap ? Describe the union of binomial heap. **AKTU 2013-14, Marks 10**

OR

Define the binomial heap in detail. Write an algorithm for performing the union operation of two binomial heaps and also explain with suitable example. **AKTU 2014-15, Marks 10**

Answer

Binomial heap : Refer Q. 2.16, Page 2–30B, Unit-2.

Union of binomial heap :

1. The BINOMIAL-HEAP-UNION procedure repeatedly links binomial trees where roots have the same degree.
2. The following procedure links the B_{k-1} tree rooted at node to the B_{k-1} tree rooted at node z, that is, it makes z the parent of y. Node z thus becomes the root of a B_k tree.

   BINOMIAL-LINK (y, z)

   i p[y] ← z
   ii. sibling [y] ← child[z]
   iii. child[z] ← y
   iv. degree[z] ← degree[z] + 1
3. The BINOMIAL-HEAP-UNION procedure has two phases :
   a. The first phase, performed by the call of BINOMIAL-HEAP-MERGE, merges the root lists of binomial heaps H_1 and H_2 into a single linked list H that is sorted by degree into monotonically increasing order.
   b. The second phase links root of equal degree until at most one root remains of each degree. Because the linked list H is sorted by degree, we can perform all the like operations quickly.

BINOMIAL-HEAP-UNION(H_1, H_2)

1. H ← MAKE-BINOMIAL-HEAP ( )
2. head[H] ← BINOMIAL-HEAP-MERGE(H_1, H_2)

3. Free the objects H_1 and H_2 but not the lists they point to
4. if head[H] = NIL
5. then return H
6. prev-x ← NIL
7. x ← head[H]
8. next-x ← sibling[x]
9. while next-x ≠ NIL
10. do if (degree[x] ≠ degree[next-x]) *or*
    (sibling[next-x] ≠ NIL and degree[sibling[next-x]] = degree[x])
11. then prev-x ← x ⇒ case 1 and 2
12. x ← next-x ⇒ case 1 and 2
13. else if key[x] ≤ key[next-x]
14. then sibling[x] ← sibling[next-x] ⇒ case 3
15. BINOMIAL-LINK(next-x, x) ⇒ case 3
16. else if prev-x = NIL ⇒ case 4
17. then head[H] ← next-x ⇒ case 4
18. else sibling[prev-x] ← next-x ⇒ case 4
19. BINOMIAL-LINK(x, next-x) ⇒ case 4
20. x ← next-x ⇒ case 4
21. next-x ← sibling[x]
22. return H

BINOMIAL-HEAP-MERGE(H_1, H_2)

1. a ← head[H_1]
2. b ← head[H_2]
3. head[H_1] ← min-degree (a, b)
4. if head[H_1] = NIL
5. return
6. if head[H_1] = b
7. then b ← a
8. a ← head[H_1]
9. while b ≠ NIL
10. do if sibling[a] = NIL
11. then sibling[a] ← b
12. return
13. else if degree [sibling[a]] < degree[b]
14. then a ← sibling[a]

15. else c ← sibling[b]
16. sibling[b] ← sibling[a]
17. sibling[a] ← b
18. a ← sibling[a]
19. b ← c

There are four cases that occur while performing union on binomial heaps.

Case 1 : When degree[x] ≠ degree[next – x] = degree [sibling[next – x]], then pointers moves one position further down the root list.

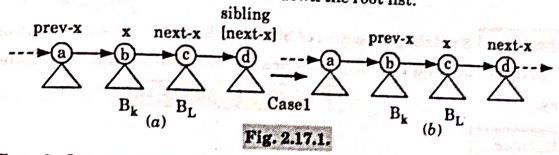


Fig. 2.17.1.

Case 2 : It occurs when x is the first of three roots of equal degree, that is, degree[x] = degree[next – x] = degree[sibling[next – x]], then again pointer move one position further down the list, and next iteration executes either case 3 or case 4.

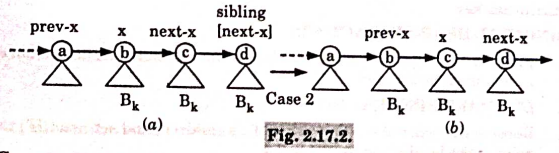


Fig. 2.17.2.

Case 3 : If degree[x] = degree[next – x] ≠ degree [sibling[next – x]] and key[x] ≤ key[next – x], we remove next – x from the root list and link it to x, creating B_{k+1} tree.

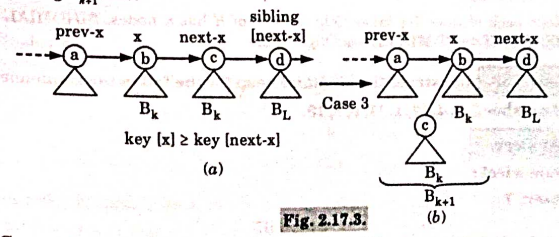


Fig. 2.17.3.

Case 4 : degree[x] = degree[next – x] ≠ degree[sibling[next – x] and key[next – x] ≤ key x, we remove x from the root list and link it to next – x, again creating a B_{k+1} tree.

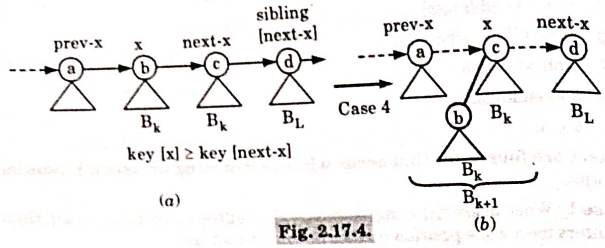


Fig. 2.17.4.

Que 2.18. **Explain properties of binomial heap. Write an algorithm to perform uniting two binomial heaps. And also to find Minimum key.** **AKTU 2017-18, Marks 10**

Answer

Properties of binomial heap : Refer Q. 2.16, Page 2–30B, Unit-2.

Algorithm for union of binomial heap : Refer Q. 2.17, Page 2–31B, Unit-2.

Minimum key :

BINOMIAL-HEAP-EXTRACT-MIN (H) :

1. Find the root x with the minimum key in the root list of H, and remove x from the root list of H.
2. $H' \leftarrow$ MAKE-BINOMIAL-HEAP( ).
3. Reverse the order of the linked list of x's children, and set head[H'] to point to the head of the resulting list.
4. $H \leftarrow$ BINOMIAL-HEAP-UNION(H, H').
5. Return x

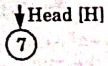
Since each of lines 1-4 takes O(lg n) time of H has n nodes, BINOMIAL-HEAP-EXTRACT-MIN runs in O(lg n) time.

Que 2.19. **Construct the binomial heap for the following sequence of number 7, 2, 4, 17, 1, 11, 6, 8, 15.**

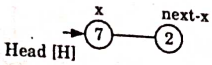
Answer

Numerical :

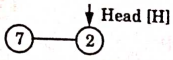
Insert 7 :



Insert 2 :

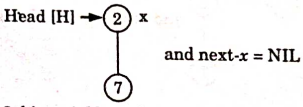


prev-x = NIL
degree [x] = 0. So, degree [x] ≠ degree [next-x] is false.
degree [next-x] = 0 and Sibling [next-x] = NIL
So, case 1 and 2 are false here.
Now key [x] = 7 and key [next-x] = 2
Now prev-x = NIL
then Head [H] ← next-x and
i.e.,

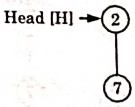


and BINOMIAL-LINK (x, next-x)
i.e.,

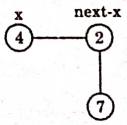
Now



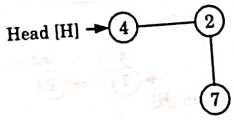
So, after inserting 2, binomial heap is



Insert 4 :

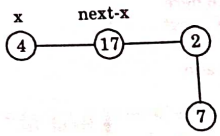


degree [x] ≠ degree [next-x]
So, Now next-x makes x and x makes prev-x.
Now next-x = NIL
So, after inserting 4, final binomial heap is :



Insert 17 :

After Binomial-Heap-Merge, we get



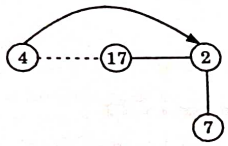
degree [x] = degree [next-x]

degree [Sibling-[next-x]] ≠ degree [x]

key [x] ≤ key [next-x]

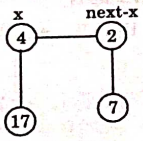
4 ≤ 17 [True]

So,



and call Binomial-Link [next-x, x]

We get



degree [x] = degree [next-x]

Sibling [next-x] = NIL

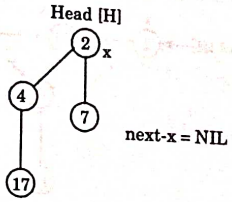
Key [x] ≤ key [next-x] [False]

prev-x = NIL then

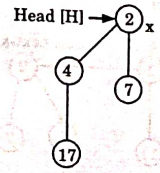
Head [H] ← [next-x]

Binomial-Link [x, next-x]

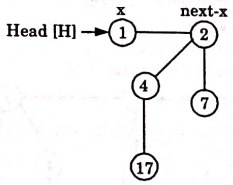
x ← next-x



So, after inserting 17, final binomial heap is :



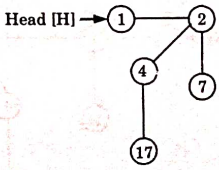
Insert 1 :



degree [x] ≠ degree [next-x]

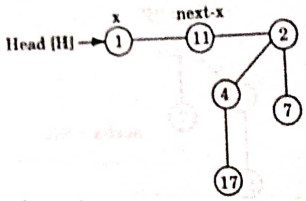
So, next-x makes x and next-x = NIL

and after inserting 1, binomial heap is :

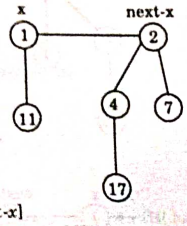


Insert 11 :

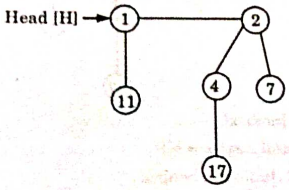
After Binomial-Heap-Merge, we get



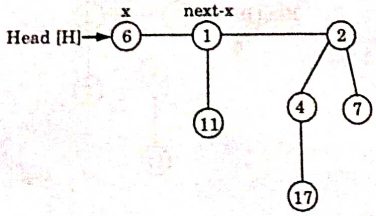
degree [x] = degree [next-x]
degree [Sibling [next-x]] ≠ degree [x]
key [x] ≤ key [next-x] [True]
So,



degree [x] ≠ degree [next-x]
So, next-x makes x and next-x = NIL
and final binomial heap after inserting 11 is

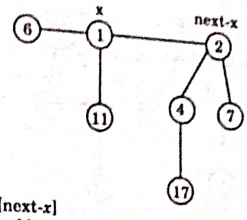


Insert 6 :

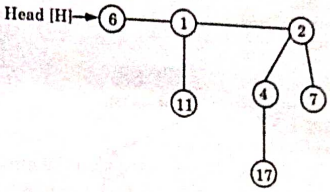


degree [x] ≠ degree [next-x]
So, next-x becomes x
Sibling [next-x] becomes next-x.

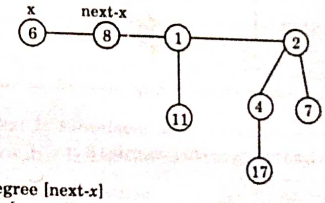
i.e.,



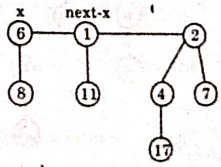
degree [x] ≠ degree [next-x]
So, no change and final heap is :



Insert 8 :

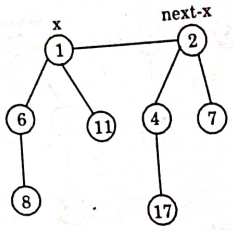


degree [x] = degree [next-x]
degree [Sibling [next-x]] ≠ degree [x]
key [x] ≤ key [next-x] [True]
So,

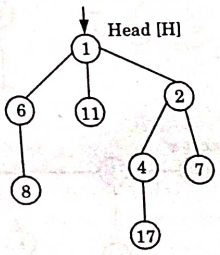


degree [x] = degree [next-x]
degree [Sibling p[next-x]] ≤ degree [x]
key [x] ≤ key [next-x] [False]
prev-x = NIL

So,

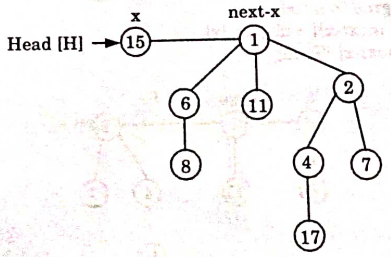


degree [x] = degree [next-x]
Sibling [next-x] = NIL
key [x] ≤ key [next-x] [True]
So, Sibling [x] = NIL.
and



next [x] = NIL
So, this is the final binomial heap after inserting 8.

Insert 15 :



degree [x] ≠ degree [next-x]
So, no change and this is the final binomial heap after inserting 15.

Que 2.20. What is a Fibonacci heap ? Discuss the applications of Fibonacci heaps.

Answer

1. A Fibonacci heap is a set of min-heap-ordered trees.
2. Trees are not ordered binomial trees, because
   a. Children of a node are unordered.
   b. Deleting nodes may destroy binomial construction.

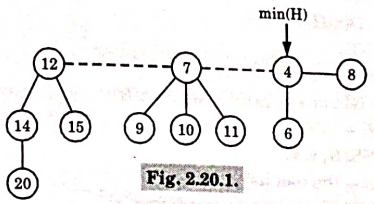


Fig. 2.20.1.

3. Fibonacci heap H is accessed by a pointer min[H] to the root of a tree containing a minimum key. This node is called the minimum node.
4. If Fibonacci heap H is empty, then min[H] = NIL.

Applications of Fibonacci heap :

1. Fibonacci heap is used for Dijkstra's algorithm because it improves the asymptotic running time of this algorithm.
2. It is used in finding the shortest path. These algorithms run in $O(n^2)$ time if the storage for nodes is maintained as a linear array.

Que 2.21. What is Fibonacci heap ? Explain CONSOLIDATE operation with suitable example for Fibonacci heap.

AKTU 2015-16, Marks 15

Answer

Fibonacci heap : Refer Q. 2.20, Page 2–40B, Unit-2.

CONSOLIDATE operation :

CONSOLIDATE(H)

1. for $i \leftarrow 0$ to $D(n[H])$
2. do $A[i] \leftarrow$ NIL
3. for each node w in the root list of H
4. do $x \leftarrow w$
5. $d \leftarrow$ degree[x]
6. while $A[d] \neq$ NIL
7. do $y \leftarrow A[d]$ ▷ Another node with the same degree as x.

8. if key[x] > key[y]
9. then exchange $x \leftrightarrow y$
10. FIB-HEAP-LINK(H, y, x)
11. $A[d] \leftarrow$ NIL
12. $d \leftarrow d + 1$
13. $A[d] \leftarrow x$
14. min[H] $\leftarrow$ NIL
15. for $i \leftarrow 0$ to $D(n[H])$
16. do if $A[i] \neq$ NIL
17. then add $A[i]$ to the root list of H
18. if min[H] = NIL or key $[A[i]]$ < key[min[H]]
19. then min[H] $\leftarrow A[i]$

FIB-HEAP-LINK(H, y, x)

1. remove y from the root list of H
2. make y a child of x, incrementing degree[x]
3. mark[y] $\leftarrow$ FALSE

Que 2.22. Define Fibonacci heap. Discuss the structure of a Fibonacci heap with the help of a diagram. Write a function for uniting two Fibonacci heaps.

Answer

Fibonacci heap : Refer Q. 2.20, Page 2–40B, Unit-2.

Structure of Fibonacci heap :

1. **Node structure ;**
   a. The field "mark" is True if the node has lost a child since the node became a child of another node.
   b. The field "degree" contains the number of children of this node. The structure contains a doubly-linked list of sibling nodes.

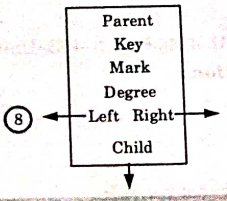


Fig. 2.22.1. Node structure.

2. **Heap structure :**
   min(H) : Fibonacci heap H is accessed by a pointer min[H] to the root of a tree containing a minimum key; this node is called the minimum node. If Fibonacci heap H is empty, then min[H] = NIL.

$n(H)$: Number of nodes in heap H

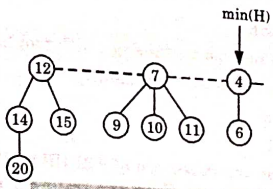


Fig. 2.22.2. Heap structure.

Function for uniting two Fibonacci heap :

Make-Heap :

MAKE-FIB-HEAP( )
allocate(H)
min(H) = NIL
$n(H) = 0$

FIB-HEAP-UNION(H_1, H_2)

1. $H \leftarrow$ MAKE-FIB-HEAP( )
2. min[H] $\leftarrow$ min[H_1]
3. Concatenate the root list of H_2 with the root list of H
4. if (min[H_1] = NIL) or (min[H_2] $\neq$ NIL and min[H_2] < min[H_1])
5. then min[H] $\leftarrow$ min[H_2]
6. $n[H] \leftarrow n[H_1] + n[H_2]$
7. Free the objects H_1 and H_2
8. return H

Que 2.23. Discuss following operations of Fibonacci heap :

i. **Make-Heap**
ii. **Insert**
iii. **Minimum**
iv. **Extract-Min**

Answer

i. **Make-Heap :** Refer Q. 2.22, Page 2–42B, Unit-2.

ii. **Insert : (H, x)**

1. degree[x] $\leftarrow$ 0
2. $p[x] \leftarrow$ NIL
3. child[x] $\leftarrow$ NIL

4. left[x] ← x
5. right[x] ← x
6. mark[x] ← FALSE
7. concatenate the root list containing x with root list H
8. if min[H] = NIL or key[x] < key[min[H]]
9. then min[H] ← x
10. $n[H] \leftarrow n[H] + 1$

To determine the amortized cost of FIB-HEAP-INSERT, Let H be the input Fibonacci heap and H' be the resulting Fibonacci heap, then $t(H) = t(H) + 1$ and $m(H') = m(H)$, and the increase in potential is,

$(t(H) + 1) + 2m(H) - (t(H) + 2m(H)) = 1$

Since the actual cost is O(1), the amortized cost is O(1) + 1 = O(1)

iii. Minimum :

The minimum node of a Fibonacci heap H is always the root node given by the pointer min[H], so we can find the minimum node in O(1) actual time. Because the potential of H does not change, the amortized cost of this operation is equal to its O(1) actual cost.

iv. FIB-HEAP-EXTRACT-MIN(H)

1. $z \leftarrow$ min[H]
2. if $z \neq$ NIL
3. then for each child x of z
4. do add x to the root list of H
5. $p[x] \leftarrow$ NIL
6. remove z from the root list of H
7. if z = right[z]
8. then min[H] ← NIL
9. else min[H] ← right[z]
10. CONSOLIDATE (H)
11. $n[H] \leftarrow n[H] - 1$
12. return z

Que 2.24. What is tries ? What are the properties of tries ?

Answer

1. A trie (digital tree / radix tree / prefix free) is a kind of search tree *i.e.*, an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings.
2. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated.

3. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string.
4. Values are not necessarily associated with every node. Rather, values tend only to be associated with leaves, and with some inner nodes that correspond to keys of interest.

Properties of a trie :

1. Tries is a multi-way tree.
2. Each node has from 1 to d children.
3. Each edge of the tree is labeled with a character.
4. Each leaf node corresponds to the stored string, which is a concatenation of characters on a path from the root to this node.

Que 2.25. Write an algorithm to search and insert a key in tries data structure.

Answer

Search a key in tries :

Trie-Search(t, $P[k..m]$) // inserts string P into t

1. if t is leaf then return true
2. else if t.child($P[k]$) = nil then return false
3. else return Trie-Search(t.child($P[k]$), $P[k + 1..m]$)

Insert a key in tries :

Trie-Insert(t, $P[k..m]$)

1. if t is not leaf then //otherwise P is already present
2. if t.child($P[k]$) = nil then

   //Create a new child of t and a "branch" starting with that child and storing $P[k..m]$
3. else Trie-Insert(t.child($P[k]$), $P[k + 1..m]$)

Que 2.26. What is skip list ? What are its properties ?

Answer

1. A skip list is built in layers.
2. The bottom layer is an ordinary ordered linked list.
3. Each higher layer acts as an "express lane", where an element in layer i appears in layer $(i + 1)$ with some fixed probability p (two commonly used values for p are ½ and ¼.).
4. On average, each element appears in $1/(1-p)$ lists, and the tallest element (usually a special head element at the front of the skip list) in all the lists.

5. The skip list contains $\log_{1/p} n$ (*i.e.*, logarithm base $1/p$ of n).

Properties of skip list :

1. Some elements, in addition to pointing to the next element, also point to elements even further down the list.
2. A level k element is a list element that has k forward pointers.
3. The first pointer points to the next element in the list, the second pointer points to the next level 2 element, and in general, the i^{th} pointer points to the next level i element.

Que 2.27. **Explain insertion, searching and deletion operation in skip list.**

Answer

Insertion in skip list :

1. We will start from highest level in the list and compare key of next node of the current node with the key to be inserted.
2. If key of next node is less than key to be inserted then we keep on moving forward on the same level.
3. If key of next node is greater than the key to be inserted then we store the pointer to current node i at update[i] and move one level down and continue our search.

At the level 0, we will definitely find a position to insert given key.

Insert(list, searchKey)

1. local update[0...MaxLevel+1]
2. x : = list → header
3. for i : = list → level down to 0 do
4. while x → forward[i] → key forward[i]
5. update[i] : = x
6. x : = x → forward[0]
7. lvl : = randomLevel()
8. if lvl > list → level then
9. for i : = list → level + 1 to lvl do
10. update[i] : = list → header
11. list → level : = lvl
12. x : = makeNode(lvl, searchKey, value)
13. for i : = 0 to level do
14. x → forward[i] : = update[i] → forward[i]
15. update[i] → forward[i] : = x

Searching in skip list :

Search(list, searchKey)

1. x : = list → header
2. loop invariant : x → key level down to 0 do
3. while x → forward[i] → key forward[i]
4. x : = x → forward[0]
5. if x → key = searchKey then return x → value
6. else return failure

Deletion in skip list :

Delete(list, searchKey)

1. local update[0..MaxLevel+1]
2. x : = list → header
3. for i : = list → level down to 0 do
4. while x → forward[i] → key forward[i]
5. update[i] : = x
6. x : = x → forward[0]
7. if x → key = searchKey then
8. for i : = 0 to list → level do
9. if update[i] → forward[i] ≠ x then break
10. update[i] → forward[i] : = x → forward[i]
11. free(x)
12. while list → level > 0 and list → header → forward[list → level] = NIL do
13. list → level : = list → level – 1

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. Define red-black tree and give its properties.

Ans: Refer Q. 2.1.

Q. 2. Explain the insertion and deletion operation in a red-black tree.

Ans: **Insertion :** Refer Q. 2.1.
Deletion : Refer Q. 2.6.

Q. 3. What do you mean by B-tree of order *m*. Explain the searching operation.

Ans. Refer Q. 2.8.

Q. 4. Explain the insertion and deletion operation in B-tree.

Ans. **Insertion operation :** Refer Q. 2.9.
Deletion operation : Refer Q. 2.10.

Q. 5. What is binomial heap ? Describe the union of binomial heap.

Ans. Refer Q. 2.17.

Q. 6. What is tries ? Give the properties of tries.

Ans. Refer Q. 2.24.

Q. 7. Explain skip list. Explain its operations.

Ans. **Skip list :** Refer Q. 2.26.
Operations : Refer Q. 2.27.

UNIT 3

Graph Algorithms

PART-1

Divide and Conquer with Examples such as Sorting, Matrix Multiplication, Convex Hull, Searching.

CONCEPT OUTLINE : PART-1

- **Strassen's algorithm for matrix multiplication :** It is an application of divide and conquer technique. Suppose we wish to compute the product $C = AB$ where each A, B and C are $n \times n$ matrices. Assuming that n is an exact power of 2. We divide each of A, B and C into four $4/2 \times 4/2$ matrices.
- **Graph :** A graph is a collection of vertices V and edges E.
- **BFS (Breadth First Search) :** BFS is an algorithm for tree searching which works on directed or undirected graphs. It is a shortest path search algorithm.
- **DFS (Depth First Search) :** It is also tree searching algorithm. In this, edges are expanded out of the most recently discovered vertex V.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.1. Write algorithm for bubble sort and selection sort.

Answer

Bubble sort :

```
Bubble_Sort (A)
1. for i → 1 to length [A]
2. for j → length [A] down to i + 1
3.        if A[j] < A[j –1]
4. Exchange (A[j], A[j – 1])
```

Selection_Sort :

```
Selection_Sort (A)
1. n ← length [A]
2.  for j ← 1 to n –1
3.    smallest ← j
4.      for i ← j + 1 to n
5.        if A[i] < A[smallest]
6.           then smallest ← i
7.  exchange (A[j] ), (A[smallest])
```


Que 3.2. What is matrix chain multiplication problem ? Describe a solution for matrix chain multiplication problem.

AKTU 2013-14, Marks 10

Answer

1. Matrix chain multiplication (or Matrix Chain Ordering Problem, MCOP) is an optimization problem that can be solved using dynamic programming.
2. MCOP helps to find the most efficient way to multiply given matrices.
3. Solution for matrix chain multiplication problem is Strassen's matrix multiplication.

Strassen's matrix multiplication :

1. It is an application of divide and conquer technique.
2. Suppose we wish to compute the product $C = AB$ where each A, B and C are $n \times n$ matrices.
3. Assuming that n is an exact power of 2. We divide each of A, B and C into four $n/2 \times n/2$ matrices.

   Rewriting the equation $C = AB$ as

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} \quad \ldots(3.2.1)$$

4. For convenience, the sub-matrices of A are labelled alphabetical from left to right, whereas those of B are labelled from top to bottom. So that matrix multiplication is performed.

   Equation (3.2.1) corresponds to the four equations :

$$r = ae + bf \quad \ldots(3.2.2)$$

$$s = ag + bh \quad \ldots(3.2.3)$$

$$t = ce + df \quad \ldots(3.2.4)$$

$$u = cg + dh \quad \ldots(3.2.5)$$

5. Each of these four equations specifies two multiplications of $n/2 \times n/2$ matrices and the addition of their $n/2 \times n/2$ products.
6. Using these equations to define a straight-forward divide and conquer strategy. We derive the following recurrence for the time $T(n)$ to multiply two $n \times n$ matrices :

$$T(n) = 8T(n/2) + \theta(n^2)$$

7. Unfortunately, this recurrence has the solution $T(n) = \Theta(n^3)$ and thus, this method is no faster than the ordinary one.

Strassen's method has four steps :

1. Divide the input matrices A and B into $n/2 \times n/2$ sub-matrices.
2. Using $\Theta(n^2)$ scalar additions and subtraction compute 14 $n/2 \times n/2$ matrices $A_1, B_1, A_2, B_2, \ldots A_7, B_7$.
3. Recursively compute the seven matrix products.
4. Compute the desired sub-matrices r, s, t, u of the result matrix C by adding and/or subtracting various combinations of the p_i matrices using only $\Theta(n^2)$ scalar additions and subtractions.

Que 3.3. Describe in detail Strassen's matrix multiplication algorithms based on divide and conquer strategies with suitable example. **AKTU 2014-15, Marks 10**

Answer

Strassen's matrix multiplication : Refer Q. 3.2, Page 3–3B, Unit-3.

Example :

Given matrices :

$$\begin{bmatrix} 2 & 9 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} 4 & 11 \\ 8 & 7 \end{bmatrix}$$

So, $\quad \text{def } a_{11} = 2, \text{def } b_{11} = 4$
$\text{def } a_{12} = 9, \text{def } b_{12} = 11$
$\text{def } a_{21} = 5, \text{def } b_{21} = 8$
$\text{def } a_{22} = 6, \text{def } b_{22} = 7$

Now calculate,

$S_1 = b_{11} - b_{22} = -3,$ $\quad S_6 = b_{11} + b_{22} = 11$
$S_2 = a_{11} + a_{12} = 11,$ $\quad S_7 = a_{12} - a_{22} = 3$
$S_3 = a_{21} + a_{22} = 11,$ $\quad S_8 = b_{21} + b_{22} = 15$
$S_4 = a_{21} - b_{11} = 4,$ $\quad S_9 = a_{11} - a_{21} = -3$
$S_5 = a_{11} + a_{22} = 8,$ $\quad S_{10} = b_{11} + b_{12} = 15$

Now, $\quad P_1 = a_{11} \times S_1 = -6,$ $\quad P_5 = S_5 \times S_6 = 88$
$P_2 = S_2 \times b_{22} = 77,$ $\quad P_6 = S_7 \times S_8 = 45$
$P_3 = S_3 \times b_{11} = 44,$ $\quad P_7 = S_9 \times S_{10} = -45$
$P_4 = a_{22} \times S_4 = 24$

Now, $\quad C_{11} = P_5 + P_4 - P_2 + P_6 = 80$
$C_{12} = P_1 + P_2 = 71$
$C_{21} = P_3 + P_4 = 68$
$C_{22} = P_5 + P_1 - P_3 - P_7 = 83$

Now matrix $= \begin{bmatrix} 80 & 71 \\ 68 & 83 \end{bmatrix}$

Que 3.4. What do you mean by graphs ? Discuss various representations of graphs.

Answer

A graph G consists of a set of vertices V together with a set E of vertex pairs of edges.

Graphs are important because any binary relation is a graph, so graphs can be used to represent essentially any relationship.

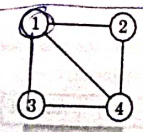


Fig. 3.4.1.

Example : A network of roads, with cities as vertices and roads between cities as edges.

Various representation of graphs :

1. **Matrix representation :** Matrices are commonly used to represent graphs for computer processing. Advantage of representing the graph in matrix lies in the fact that many results of matrix algebra can be readily applied to study the structural properties of graph from an algebraic point of view.

   a. **Adjacency matrix :**

      i. **Representation of undirected graph :**

      The adjacency matrix of a graph G with n vertices and no parallel edges is a $n \times n$ matrix $A = [a_{ij}]$ whose elements are given by

      $a_{ij} = 1$, if there is an edge between i^{th} and j^{th} vertices
      $= 0$, if there is no edge between them

      ii. **Representation of directed graph :**

      The adjacency matrix of a digraph D, with n vertices is the matrix

      $A = [a_{ij}]_{n \times n}$ in which
      $a_{ij} = 1$ if arc (v_i, v_j) is in D
      $= 0$ otherwise

   b. **Incidence matrix :**

      i. **Representation of undirected graph :**

      Consider an undirected graph $G = (V, E)$ which has n vertices and m edges all labelled. The incidence matrix $I(G) = [b_{ij}]$, is then $n \times m$ matrix, where

      $b_{ij} = 1$ when edge e_j is incident with v_i
      $= 0$ otherwise

ii. **Representation of directed graph :**
The incidence matrix $I(D) = [b_{ij}]$ of digraph D with n vertices and m edges is the $n \times m$ matrix in which.

$b_{ij} = 1$ if arc j is directed away from vertex v_i
$= -1$ if arc j is directed towards vertex v_i
$= 0$ otherwise.

2. **Linked representation :**
   a. In linked representation, the two nodes structures are used :
      i. For non-weighted graph,

INFO	Adj-list

      ii. For weighted graph,

Weight	INFO	Adj-list

      Where Adj-list is the adjacency list *i.e.*, the list of vertices which are adjacent for the corresponding node.
   b. The header nodes in each list maintain a list of all adjacent vertices of that node for which the header node is meant.

Que 3.5. **What is a bipartite graph ? How to check a graph is bipartite or not ?**

Answer

A bipartite graph is an undirected graph $G = (V, E)$ in which V can be partitioned into two sets V_1 and V_2 such that $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$.

To check graph is bipartite or not : The algorithm traverse the graph labeling the vertices 0, 1, or 2 corresponding to unvisited, partition 1 and partition 2 nodes. If an edge is detected between two vertices in the same partition, the algorithm returns.

Bipartite (G, S)

1. For each vertex $U \in V[G] - \{s\}$ do
2. Colour[u] ← WHITE
3. $d[u] \leftarrow \infty$
4. partition[u] ← 0
5. Colour[s] ← gray
6. partition[s] ← 1
7. $d[s] \leftarrow 1$
8. $Q \leftarrow [s]$
9. While Queue 'Q' is not empty do
10. $u \leftarrow$ head $[Q]$
11. for each v in Adj[u] do

12. if partition [u] = partition [v] then
13. return 0
14. else
15. if colour[v] = WHITE then
16. colour[v] ← gray
17. $d[v] \leftarrow d[u] + 1$
18. partition[v] = 3 – partition[u]
19. ENQUEUE (Q, v)
20. DEQUEUE (Q)
21. Colour[u] ← BLACK
22. Return 1

Que 3.6. **Explain DFS. Also give DFS algorithm.**

OR

Describe Depth First Search (DFS) strategy. How DFS can be used to solve the problem of unbounded trees ? Also write an algorithm.

Answer

Depth First Search Algorithm :

1. Algorithm starts at a specific vertex S in G, which becomes current vertex.
2. Then algorithm traverse graph by any edge (u, v) incident to the current vertex u.
3. If the edge (u, v) leads to an already visited vertex v, then we backtrack to current vertex u.
4. If, on other hand, edge (u, v) leads to an unvisited vertex v, then we go to v and v becomes our current vertex.
5. We proceed in this manner until we reach to "dead end". At this point we start backtracking.
6. The process terminates when backtracking leads back to the start vertex.
7. Edges leads to new vertex are called discovery or tree edges and edges lead to already visited vertex are called back edges.

How to solve the problem of unbounded tree :

1. The problem of unbounded depth of trees can be overcome by limiting the depth first search to a pre-determined depth limit 'l'.
2. This means that nodes at depth 'l' are treated as if they have no successors. This is called depth-limited search. This solves the problem of infinite-path problem.
3. However, often we do not know the depth 'd' of the goal state. If we choose $l < d$, then we will never find the solution. If we choose $l > d$, we may end up with a non-optimal solution.

4. Depth first is a special case of depth-limited search, where $l = \infty$.

Algorithm :

```
depth limit = max depth to search to;
agenda = initial state;
    if initial state is goal state then return solution;
    else
        while agenda not empty do
        take node from front of agenda;
        if depth (node) < depth limit then
            {new nodes = apply operations to node;
            add new nodes to front of agenda;
            if goal state in new nodes then return solution;}
```

Que 3.7. **Explain Breadth First Search (BFS). Give its algorithm.**

Answer

Breadth first search :

1. The general idea behind a breadth first search beginning at a starting node A is as follows :
   a. First we examine the starting node A.
   b. Then, we examine all the neighbours of A, and so on.
2. Naturally, we need to keep track of the neighbours of a node, and we need to guarantee that no node is processed more than once.
3. This is accomplished by using a queue to hold nodes that are waiting to be processed, and by using a field STATUS which tells us the current status of any node.

Algorithm : This algorithm executes a breadth first search on a graph G beginning at a starting node A.

1. Initialize all nodes to ready state (STATUS=1).
2. Put the starting node A in queue and change its status to the waiting state (STATUS = 2).
3. Repeat steps 4 and 5 until queue is empty.
4. Remove the front node N of queue. Process N and change the status of N to the processed state (STATUS = 3).
5. Add to the rear of queue all the neighbours of N that are in the ready state (STATUS=1) and change their status to the waiting state (STATUS = 2).

   [End of loop]

6. End.

Que 3.8. **Explain topological sort. Give its algorithm.**

Answer

1. A topological sort of a Directed Acyclic Graph (DAG) $\overline{G}$ is an ordering of the vertices of $\overline{G}$ such that for every edge (e_i, e_j) of $\overline{G}$ we have $i < j$.
2. A topological sort is a linear ordering of all its vertices such that if DAG $\overline{G}$ contains an edge (e_i, e_j), then e_i appears before e_j in the ordering. If DAG is cyclic then no linear ordering is possible.
3. A topological ordering is an ordering such that any directed path in DAG, G traverses vertices in increasing order.

Topological_Sort(G)

i. For each vertex find the finish time by calling DFS(G).
ii. Insert each finished vertex into the front of a linked list.
iii. Return the linked list.

Que 3.9. **Write an algorithm to test whether a given graph is connected or not.**

Answer

Test-connected (G) :

1. Choose a vertex x
2. Make a list L of vertices reachable from x, and another list K of vertices to be explored.
3. Initially, $L = K = x$.
4. while K is non-empty
5. Find and remove some vertex y in K
6. for each edge (y, z)
7. if (z is not in L)
8. Add z to both L and K
9. if L has fewer than n items
10. return disconnected
11. else return connected.

Que 3.10. **Discuss strongly connected components with its algorithm.**

Answer

1. The Strongly Connected Components (SCC) of a directed graph G are its maximal strongly connected subgraphs.
2. If each strongly connected component is contracted to a single vertex, the resulting graph is a directed acyclic graph called as condensation of G.

Kosaraju's algorithm :

Kosaraju's algorithm is an algorithm to find the strongly connected components of a directed graph.

1. Let G be a directed graph and S be an empty stack.
2. While S does not contain all vertices :
   i. Choose an arbitrary vertex v not in S. Perform a depth first search starting at v.
   ii. Each time that depth first search finishes expanding a vertex u, push u onto S.
3. Reverse the direction of all arcs to obtain the transpose graph.
4. While S is non-empty :
   i. Pop the top vertex v from S. Perform a depth first search starting at v.
   ii. The set of visited vertices will give the strongly connected component containing v; record this and remove all these vertices from the graph G and the stack S.
   iii. Equivalently, breadth first search (BFS) can be used instead of depth first search.

Que 3.11. Explain convex hull problem.

AKTU 2017-18, Marks 10

OR

Discuss convex hull. Give Graham-Scan algorithm to compute convex hull.

Answer

1. The convex hull of a set S of points in the plane is defined as the smallest convex polygon containing all the points of S.
2. The vertices of the convex hull of a set S of points form a (not necessarily proper) subset of S.

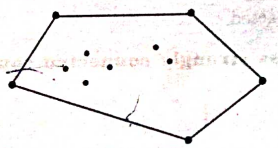


Fig. 3.11.1.

3. To check whether a particular point $p \in S$ is extreme, see each possible triplet of points and check whether p lies in the triangle formed by these three points.
4. If p lies in the triangle then it is not extreme, otherwise it is.

5. We denote the convex hull of S by CH(S). Convex hull is a convex set because the intersection of convex sets is convex and convex hull is also a convex closure.

Graham-Scan algorithm :

The procedure GRAHAM-SCAN takes as input a set Q of points, where $|Q| \geq 3$. It calls the functions Top(S), which return the point on top of stack S without changing S, and to NEXT-TO-TOP(S), which returns the point one entry below the top of stack S without changing S.

GRAHAM-SCAN(Q)

1. Let p_0 be the point in Q with the minimum y-coordinate, or the leftmost such point in case of a tie.
2. Let $\langle p_1, p_2, \ldots, p_m \rangle$ be the remaining points in Q, sorted by polar angle in counter clockwise order around p_0 (if more than one point has the same angle remove all but the one that is farthest from p_0).
3. PUSH (p_0, S)
4. PUSH (p_1, S)
5. PUSH (p_2, S)
6. for $i \leftarrow 3$ to m
7. do while the angle formed by points NEXT-To-TOP(S), Top(S), and p_i makes a non left turn.
8. do POP(S)
9. PUSH (p_i, S)
10. return S

The worst case running time of GRAHAM-SCAN is

$$T(n) = O(n)\, O(n \log n) + O(1) + O(n) = O(n \log n)$$

where $n = |Q|$

Graham's scan running time depends only on the size of the input it is independent of the size of output.

Que 3.12. Give Jarvis's March algorithm to compute convex hull.

Answer

Jarvis's march computes the convex hull of a set Q of points by a technique known as package wrapping. The algorithm runs in time $O(nh)$ where n is the number of vertices of CH(Q).

Steps for Jarvis's March algorithm :

1. First, a base point p_0 is selected, this is the point with the minimum y-coordinate.
2. Select leftmost point in case of tie.

3. The next convex hull vertices p_1 has the least polar angle with respect to the positive horizontal ray from p_0.
4. Measure in counter clockwise direction.
5. If tie, choose the farthest such point.
6. Vertices $p_2, p_3, \ldots p_k$ are picked similarly until $y_k = y_{max}$.
7. p_{i+1} has least polar angle with respect to positive ray from p_0.
8. If tie, choose the farthest such point.

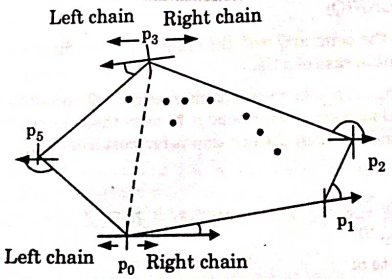


Fig. 3.12.1.

9. The sequence p_0, p_1, p_k is right chain of CH(Q).
10. To choose the left chain of CH(Q) start with p_k.
11. Choose p_{k+1} as the point with least polar angle with respect to the negative ray from p_k.
12. Again measure counterclockwise direction.
13. If tie occurs, choose the farthest such point.
14. Continue picking $p_{k+1}, p_{k+2}, \ldots p_t$ in same fashion unit obtain $p_t = p_0$.

PART-2

Greedy Methods with Examples such as Optimal Reliability Allocation, Knapsack.

CONCEPT OUTLINE : PART-2

- **Greedy algorithms :** Greedy algorithms are simple and straightforward. They are shortsighted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future.

- **Activity selection problem :** In this problem, we first choose the activity with minimum duration $(f_i - s_i)$ and schedule it. Then we left all the activities that are not compatible to this one, which means we have to select an activity which is compatible having minimum duration.
- **Knapsack problem :**
  We want to pack m items such that :
  i. The item i^{th} is worth x_i dollars and weights w_i pound.
  ii. Take as valuable a load as possible, but cannot exceed w pounds.
  iii. x_i, w_i, w are integers.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.13. Write note on the greedy algorithm.

Answer

1. Greedy algorithms are simple and straight forward.
2. Greedy algorithms are shortsighted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future.
3. Greedy algorithms are easy to invent, easy to implement and most of the time quite efficient.
4. Many problems cannot be solved correctly by greedy approach.
5. Greedy algorithms are used to solve optimization problems.

Que 3.14. What are the four functions included in greedy algorithm ? Write structure of greedy algorithm.

Answer

The greedy algorithm consists of four function :

i. A function that checks whether chosen set of items provide a solution.
ii. A function that checks the feasibility of a set.
iii. The selection function tells which of the candidates are most promising.
iv. An objective function, which does not appear explicitly, gives the value of a solution.

Structure of greedy algorithm :

1. Initially the set of chosen items is empty i.e., solution set.
2. At each step
   a. Item will be added in a solution set by using selection function.
   b. If the set would no longer be feasible
      Reject items under consideration (and is never consider again).
   c. Else if set is still feasible
      add the current item.

Que 3.15. Define activity selection problem and give its solution by using greedy approach with its correctness.

Answer

1. An activity selection is the problem of scheduling a resource among several competing activity. Given a set $S = \{1, 2, ..., n\}$ of n activities.
2. Each activity has s_i a start time, and f_i a finish time.
3. If activity i is selected, the resource is occupied in the intervals (s_i, f_i). We say i and j are compatible activities if their start and finish time does not overlap i.e., i and j compatible if $s_i \geq f_j$ and $s_j \geq f_i$.
4. The activity selection problem is, to select a maximal sized subset of mutually compatible activities.

Here we maximize the number of activities selected, but if the profit were proportional to $s_i - f_i$, this will not maximize the profit.

Greedy algorithm :

Assume that $f_1 \leq f_2 \leq ... \leq f_n$

Greedy-Activity-Selector (s, f)

1. $n \leftarrow$ length [s]
2. $A \leftarrow \{a_1\}$
3. $i \leftarrow 1$
4. for $m \leftarrow 2$ to n
5. do if $s_m \geq f_i$
6. then $A \leftarrow A \cup \{a_m\}$
7. $i \leftarrow m$
8. return A

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

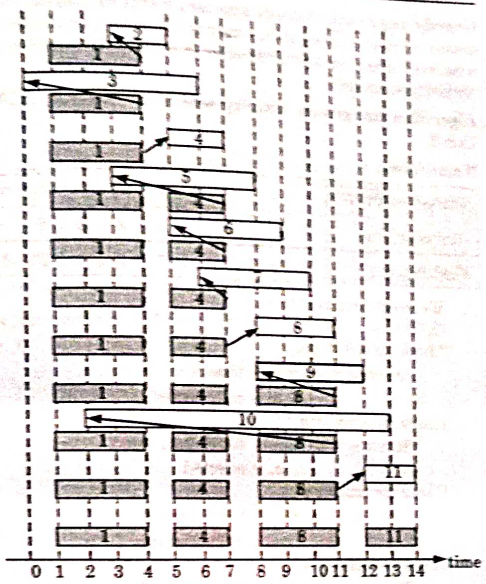


Fig. 3.15.1.

The algorithm starts with {1} and checks to see which can be added after 1, updating the global "finishing time" and comparing with each start time. The activity picked is always the first that is compatible. Greedy algorithms do not always produce optimal solutions.

Correctness : Greedy algorithm does not always produce optimal solutions but GREEDY-ACTIVITY-SELECTOR does.

Que 3.16. What are greedy algorithms ? Find a solution to the following activity selection problem using greedy technique. The starting and finishing times of 11 activities are given as follows : (2, 3) (8, 12) (12, 14) (3, 5) (0, 6) (1, 4) (6, 10) (5, 7) (3, 8) (5, 9) (8, 11)

OR

What is greedy approach ? Write an algorithm which uses this approach.

Answer

Greedy algorithm : Refer Q. 3.13, Page 3–13B, Unit-3.

Greedy approach : Greedy approach works by making the decision that seems most promising at any moment it never reconsiders this decision, whatever situation may arise later. For example consider the problem of "Activity selection".

Algorithm for greedy activity selection : Refer Q. 3.15, Page 3–14B, Unit-3.

Numerical :

Sorted activities	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
Starting time	2	1	3	0	5	3	5	6	8	8	12
Finish time	3	4	5	6	7	8	9	10	11	12	14

We select first activity a_1
(2, 3)
Check for activity a_2
Starting time of $a_2 \not\geq$ time of a_1
$\therefore$ a_2 is not selected
Check for activity a_3
Starting time of $a_3 \geq$ finish time of a_1
$\therefore$ a_3 is selected
Check for activity a_4
Starting time of $a_4 \not\geq$ finish time of a_3
$\therefore$ a_4 is not selected
Check for activity a_5
Starting time of $a_5 \geq$ finish time of a_3
$\therefore$ a_5 is selected
Check for activity a_6
Starting time of $a_6 \not\geq$ finish time of a_5
$\therefore$ a_6 is not selected
Check for activity a_7
Starting time of $a_7 \not\geq$ finish time of a_5
$\therefore$ a_5 is not selected
Check for activity a_8
Starting time of $a_8 \not\geq$ finish time of a_5
$\therefore$ a_9 is selected
Check for activity a_{10}
Starting time of $a_{10} \not\geq$ finish time of a_9
$\therefore$ a_{10} is not selected
Check for activity a_{11}
Starting time of $a_{11} \geq$ finish time of a_4
$\therefore$ a_{11} is selected.

$\therefore$ **Therefore selected activities are :**

a_1 : (2, 3)
a_3 : (3, 5)
a_5 : (5, 7)
a_9 : (8, 11)
a_{11} : (12, 14)

Que 3.17. What is "Greedy Algorithm" ? Write its pseudo code for recursive and iteration process.

Answer

Greedy algorithm : Refer Q. 3.13, Page 3–13B, Unit-3.

Greedy algorithm defined in two different forms :

i. **Pseudo code for recursive greedy algorithm :**
R_A_S (s, f, i, j)

1. $m \leftarrow i + 1$
2. while $m < j$ and $s_m < f_i$
3. do $m \leftarrow m + 1$
4. if $m < j$
5. then return $\{a_m\} \cup R\_A\_S(s, f, m, j)$
6. else return ϕ

ii. **Pseudo code for iterative greedy algorithm :**
G_A_S (s, f)

1. $n \leftarrow$ length $[s]$
2. $A \leftarrow [a_1]$
3. $i \leftarrow 1$
4. $m \leftarrow 2$ to n
5. do if $s_m \geq f_i$
6. then $A \leftarrow A \cup \{a_m\}$
7. $i \leftarrow m$
8. return A

Que 3.18. What is an optimization problem ? How greedy method can be used to solve the optimization problem ?

AKTU 2013-14, Marks 10

Answer

1. An optimization problem is the problem of finding the best solution from all feasible solutions.
2. Optimization problems can be divided into two categories depending on whether the variables are continuous or discrete.

3. There is no way in general that one can specify if a greedy algorithm will solve a particular optimization problem.
4. However if the following properties can be demonstrated, then it is probable to use greedy algorithm :
   a. **Greedy choice property :** A globally optimal solution can be arrived at by making a locally optimal greedy choice. That is, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from sub-problems.
   b. **Optimal substructure :** A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to sub-problems.

Que 3.19. What is knapsack problem ? Describe an approach used to solve the problem.

Answer

1. The knapsack problem is a problem in combinatorial optimization.
2. Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
3. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

Approach use to solve the problem :

1. In knapsack problem, we have to fill the knapsack of capacity w, with a given set of items $I_1, I_2 \ldots I_n$ having weight $w_1, w_2 \ldots w_n$ in such a manner that the total weight of items cannot exceed the capacity of knapsack and maximum possible value can be obtained.
2. Using branch and bound approach, we have a bound that none of the items can have total sum more than the capacity of knapsack and must give maximum possible value.
3. The implicit tree for this problem is a binary tree which left branch implies inclusion and right exclusion.
4. Upper bound of node can be calculated as :
   $ub = v + (W - w)(v_{i+1}/w_{i+1})$

weight	value
ub	
node	

Que 3.20. Write greedy algorithm for discrete knapsack problem.

Answer

Greedy algorithm for the discrete knapsack problem :

1. Compute value/weight ratio v_i/w_i for all items.
2. Sort the items in non-increasing order of the ratios v_i/w_i.
3. Repeat until no item is left in sorted list using following steps :
   a. If current item fits, use it.
   b. Otherwise skip this item, and proceed to next item.

Example : Knapsack problem for the following instance using greedy approach. The item can be selected or skipped completely.

Item	Weight	Value
1	7	₹49
2	3	₹12
3	4	₹42
4	5	₹30

Consider $W = 10$.

Solution : This is also called 0-1 knapsack. Either we can completely select an item or skip it. First of all we will compute value-to-weight ratio and arrange them in non-increasing order of the ratio.

Item	Weight	Value	Value/Weight	
3	4	₹42	10.5	→
1	7	₹49	7	
4	5	₹30	6	→
2	3	₹12	4	

To fulfill the capacity $W = 10$, we will have

1. add item of weight 4
2. skip item of weight 7
3. add item of weight 5
4. skip item of weight 3

Maximum value = 10.5 + 6 = 16.5

This is the solution for given instance of knapsack problem.

But the greedy algorithm does not give optimal solution always rather there is no upper bound on the accuracy of approximate solution.

Que 3.21. **Given the six items in the table below and a knapsack with weight 100, what is the solution to the knapsack problem in all concepts. *i.e.*, explain greedy all approaches and find the optimal solution.**

Item ID	Weight	Value	Value/Weight
A	100	40	.4
B	50	35	.7
C	40	20	.5
D	20	4	.2
E	10	10	1
F	10	6	.6

AKTU 2017-18, Marks 10

Answer

We can use 0 – 1 knapsack problem when the items cannot be divided into parts and fractional knapsack problem when the items can be divided into fractions.

According to 0 – 1 knapsack problem, either we select an item or reject. So the item will be selected according to value per weight.

E is selected $W = 10 < 100$

B is selected $W = 10 + 50$

$= 60 < 100$

F is selected $W = 60 + 10$

$= 70 < 100$

C cannot be selected because

$W = 70 + 40 = 110 > 100$

Hence we select *D*

$W = 70 + 20 = 90 < 100$

Total value $= 10 + 35 + 6 + 4 = 55$

According to fractional knapsack problem, we can select fraction of any item.

E is selected $W = 10 < 100$

B is selected $W = 10 + 50$

$= 60 < 100$

F is selected $W = 60 + 10$

$= 70 < 100$

If we select *C* $W = 70 + 40$

$= 110 > 100$

Hence we select the fraction of item *C* as

$$\frac{100 - W}{\text{Weight of } C} = \frac{100 - 70}{40}$$

$$= \frac{30}{40} = 0.75$$

So, $W = 0.75 \times 40 = 30$

$W = 70 + 30 = 100$

Total value $= 10 + 35 + 6 + 0.75\,(20)$

$= 10 + 35 + 6 + 15 = 66$

Que 3.22. **What is 0/1-knapsack problem ? Does greedy method effective to solve the 0/1-knapsack problem ?**

Answer

The 0/1-knapsack problem is defined as follows :

1. Given, a knapsack of capacity c and n items of weights $\{w_1, w_2, ..., w_n\}$ and profits $\{p_1, p_2, ..., p_n\}$, the objective is to choose a subset of n objects that fits into the knapsack and that maximizes the total profit.
2. Consider a knapsack (bag) with a capacity of c.
3. We select items from a list of n items.
4. Each item has both a weight of w_i and profit of p_i.
5. In a feasible solution, the sum of the weights must not exceed the knapsack capacity (c) and an optimal solution is both feasible and reaches the maximum profit.
6. An optimal packing is a feasible solution one with a maximum profit :

$$p_1x_1 + p_2x_2 + p_3x_3 + ... + p_nx_n = \sum_{i=1}^{n} p_ix_i$$

which is subjected to constraints :

$$p_1x_1 + w_2x_2 + w_3x_3 + ... + w_nx_n = \sum_{i=1}^{n} w_ix_i \le c$$

and $x_i = 1$ or 0, $1 \le i \le n$

7. We have to find the values of x_i where $x_i = 1$ if i^{th} item is packed into the knapsack and $x_i = 0$ if i^{th} item is not packed.

Greedy strategies for the knapsack problem are :

i. From the remaining items, select the item with maximum profit that fits into the knapsack.

ii. From the remaining items, select the item that has minimum weight and also fits into the knapsack.

iii. From the remaining items, select the one with maximum p_i/w_i that fits into the knapsack.

Greedy method is not effective to solve the 0/1-knapsack problem. By using greedy method we do not get optimal solution.

Que 3.23. **What is 0/1-knapsack problem? Solve the following instance using greedy approach, also write the algorithm. Knapsack capacity = 10, $P = <1, 6, 18, 22, 28>$ and $w = <1, 2, 5, 6, 7>$.**

Answer

0/1 knapsack problem : Refer Q. 3.22, Page 3–21B, Unit-3.

Numerical :

Knapsack capacity = 10

$P = < 1, 6, 18, 22, 28 >$

$W = < 1, 2, 5, 6, 7 >$

$\therefore \quad P_i = \frac{V_i}{W_i}$

$\therefore \quad V_i = P_i \times W_i$

Item	Weight	Value	Value/Weight
I_1	7	196	28
I_2	6	132	22
I_3	5	90	18
I_4	2	12	6
I_5	1	1	1

To fulfill the capacity $W = 10$

Add item of weight 7. (knapsack capacity = 10 – 7 = 3)

Skip item of weight 6 since available knapsack capacity is less than 6.

Skip item of weight 5 since available knapsack capacity is 3.

Add item of weight 2. (knapsack capacity = 3 – 2 = 1)

Add item of weight 1.

Maximum Value = 196 + 12 + 1

= 209

Que 3.24. **Consider following instance for simple knapsack problem. Find the solution using greedy method.**

$N = 8$

$P = \{11, 21, 31, 33, 43, 53, 55, 65\}$

$W = \{1, 11, 21, 23, 33, 43, 45, 55\}$

$M = 110$

AKTU 2016-17, Marks 7.5

Answer

$N = 8$

$W = \{1, 11, 21, 23, 33, 43, 45, 55\}$

$P = \{11, 21, 31, 33, 43, 53, 55, 65\}$

$M = 110$

Now, arrange the value of P_i in decreasing order

N	W_i	P_i	$V_i = W_i \times P_i$
1	1	11	11
2	11	21	231
3	21	31	651
4	23	33	759
5	33	43	1419
6	43	53	2279
7	45	55	2475
8	55	65	3575

Now, fill the knapsack according to decreasing value of P_i. First we choose item $N = 1$ whose weight is 1.

Then choose item $N = 2$ whose weight is 11.

Then choose item $N = 3$ whose weight is 21.

Now, choose item $N = 4$ whose weight is 23.

Then choose item $N = 5$ whose weight is 33.

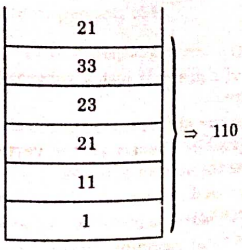
Total weight in knapsack is = 1 + 11 + 21 + 23 + 33 = 89

Now, the next item is $N = 6$ and its weight is 43, but we want only 21 because $M = 110$.

So, we choose fractional part of it, *i.e.*,

The value of fractional part of $N = 6$ is,

$$\frac{2279}{43} \times 21 = 1113$$



Thus, the maximum value is,

= 11 + 231 + 651 + 759 + 1419 + 1113

= 4184

PART-3

Minimum Spanning Trees-Prim's and Kruskal's Algorithm, Single Source Shortest Paths-Dijkstra's and Bellman-Ford Algorithm.

CONCEPT OUTLINE : PART-3

- **Minimum spanning tree :** A spanning tree of a graph is just a subgraph that contains all the vertices and does not contain any cycle.
- There are two techniques of finding minimum spanning tree which are as follows :
  a. Kruskal's algorithm
  b. Prim's algorithm
- **Single source shortest paths problem :** Given a graph $G = (V, E)$, we want to find a shortest path from a given vertex (source) $S \in V$ to every vertex $v \in V$.
- **Algorithms to single source shortest path problem :**
  i. Dijkstra's algorithm
  ii. Bellman-Ford algorithm

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.25. What do you mean by spanning tree and minimum spanning tree ?

Answer

Spanning tree :

1. A spanning tree of a graph is just a subgraph that contains all the vertices and is a tree.
2. That is, a spanning tree of a connected graph G contains all the vertices and has the edges which connect all the vertices. So, the number of edges will be 1 less the number of nodes.
3. If graph is not connected, *i.e.*, a graph with n vertices has edges less than $n - 1$ then no spanning tree is possible.
4. A graph may have many spanning trees.

Minimum spanning tree :

1. Given a connected weighted graph G, it is often desired to create a spanning tree T for G such that the sum of the weights of the tree edges in T is as small as possible.

2. Such a tree is called a minimum spanning tree and represents the "cheapest" way of connecting all the nodes in G.
3. There are number of techniques for creating a minimum spanning tree for a weighted graph but the most famous methods are Prim's and Kruskal's algorithm.

Que 3.26. Write Kruskal's algorithm to find minimum spanning tree.

Answer

i. In this algorithm, we choose an edge of G which has smallest weight among the edges of G which are not loops.

ii. This algorithm gives an acyclic subgraph T of G and the theorem given below proves that T is minimal spanning tree of G. Following steps are required :

Step 1 : Choose e_1, an edge of G, such that weight of e_1, $w(e_1)$ is as small as possible and e_1 is not a loop.

Step 2 : If edges $e_1, e_2, \ldots\ldots, e_i$ have been selected then choose an edge e_{i+1} not already chosen such that

i. the induced subgraph

$$G[\{e_1 \ldots\ldots e_{i+1}\}] \text{ is acyclic and}$$

ii. $w(e_{i+1})$ is as small as possible

Step 3 : If G has n vertices, stop after $n - 1$ edges have been chosen. Otherwise repeat step 2.

If G be a weighted connected graph in which the weight of the edges are all non-negative numbers, let T be a subgraph of G obtained by Kruskal's algorithm then, T is minimal spanning tree.

Que 3.27. Describe and compare following algorithms to determine the minimum cost spanning tree :
i. Kruskal's algorithm
ii. Prim's algorithm **AKTU 2013-14, Marks 10**

Answer

i. **Kruskal's algorithm :** Refer Q. 3.26, Page 3-25B, Unit-3.

ii. **Prim's algorithm :**

First it chooses a vertex and then chooses an edge with smallest weight incident on that vertex. The algorithm involves following steps :

Step 1 : Choose any vertex V_1 of G.

Step 2 : Choose an edge $e_1 = V_1V_2$ of G such that $V_2 \neq V_1$ and e_1 has smallest weight among the edge e of G incident with V_1.

Step 3 : If edges $e_1, e_2, \ldots\ldots, e_i$ have been chosen involving end points $V_1, V_2, \ldots\ldots\ldots, V_{i+1}$, choose an edge $e_{i+1} = V_jV_k$ with $V_j = \{V_1 \ldots\ldots V_{i+1}\}$ and $V_k \notin \{V_1 \ldots\ldots V_{i+1}\}$ such that e_{i+1} has smallest weight among the edges of G with precisely one end in $\{V_1 \ldots\ldots\ldots V_{i+1}\}$.

Step 4 : Stop after $n - 1$ edges have been chosen. Otherwise goto step 3.

Comparison :

S. No.	Kruskal's algorithm	Prim's algorithm
1.	Kruskal's algorithm initiates with an edge.	Prim's algorithm initializes with a node.
2.	Kruskal's algorithm selects the edges in a way that the position of the edge is not based on the last step.	Prim's algorithms span from one node to another.
3.	Kruskal's can function on disconnected graphs too.	In prim's algorithm, graph must be a connected graph.
4.	Kruskal's time complexity in worst case is $O(E \log E)$.	Prim's algorithm has a time complexity in worst case of $O(E \log V)$.

Que 3.28. **What do you mean by minimum spanning tree ? Write an algorithm for minimum spanning tree that may generate multiple forest trees and also explain with suitable example.**

AKTU 2014-15, Marks 10

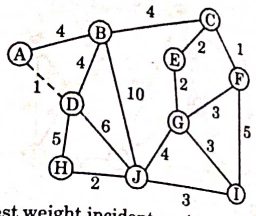
Answer

Minimum spanning tree : Refer Q. 3.25, Page 3–24B, Unit-3.

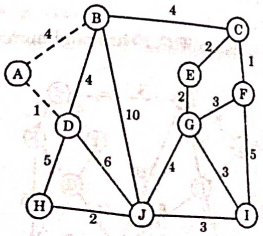
Prim's algorithm : Refer Q. 3.27, Page 3–25B, Unit-3.

Example :

According to algorithm we choose vertex A from the set $\{A, B, C, D, E, F, G, H, I, J\}$.



Now edge with smallest weight incident on A is $e = (AD)$



Now we look on weight

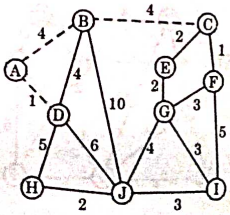
$W(A, B) = 4$

$W(D, B) = 4 \quad W(D, H) = 5$

$W(D, J) = 6$

We choose $e = AB$ since it is minimum.

$W(D, B)$ can also be chosen because it has same value.



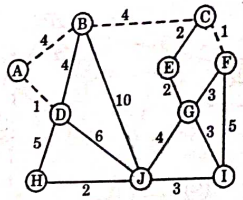
Again, $W(B, C) = 4$

$W(B, J) = 10$

$W(D, H) = 5$

$W(D, J) = 6$

We choose $e = BC$ since it has minimum value.

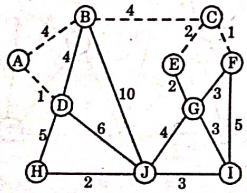


Now, $W(B, J) = 10$

$W(C, E) = 2$

$W(C, F) = 1$

We choose $e = CF$ because $W(C, F)$ has minimum value.

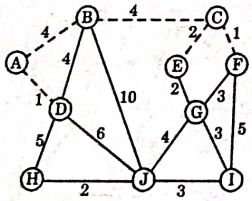


Now, $W(C, E) = 2$

$W(F, G) = 3$

$W(F, I) = 5$

We choose $e = CE$, since $W(C, E)$ is minimum.

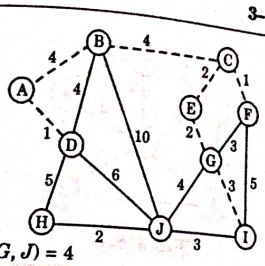


$W(E, G) = 2$

$W(F, G) = 3$

$W(F, I) = 5$

We choose $e = EG$, since $W(E, G)$ is minimum.

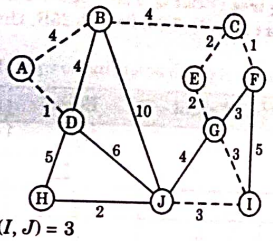


$W(G, J) = 4$

$W(G, I) = 3$

$W(F, I) = 5$

We choose $e = GI$, since $W(G, I)$ is minimum.



$W(I, J) = 3$

$W(G, J) = 4$

We choose $e = IJ$, since $W(I, J)$ is minimum

$W(J, H) = 2$

Hence, $e = JH$ will be chosen.

The final minimal spanning tree is given as :

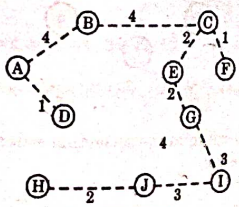


Fig. 3.28.1.

Que 3.29. **What is minimum cost spanning tree ? Explain Kruskal's algorithm and find MST of the graph. Also write its time complexity.**

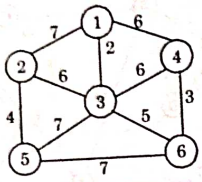


Fig. 3.29.1.

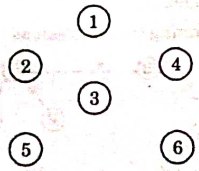
AKTU 2017-18, Marks 10

Answer

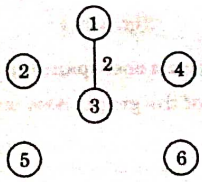
Minimum spanning tree : Refer Q. 3.25, Page 3–24B, Unit-3.
Kruskal's algorithm : Refer Q. 3.26, Page 3–25B, Unit-3.
Numerical :
Step 1 : Arrange the edge of graph according to weight in ascending order.

Edges	Weight	Edge	Weight
13	2	32	6
46	3	12	7
25	4	35	7
36	5	56	7
34	6		
41	6		

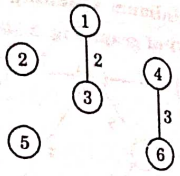
Step 2 : Now draw the vertices as given in graph,



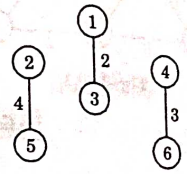
Now draw the edge according to the ascending order of weight. If any edge forms cycle, leave that edge.
Step 3 :



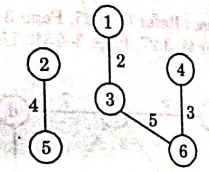
Step 4 :



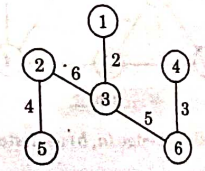
Step 5 :



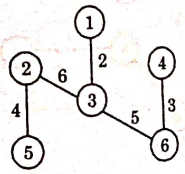
Step 6 :



Step 7 :



All the remaining edges, such as : 34, 41, 12, 35, 56 are rejected because they form cycle.
All the vertices are covered in this tree. So, the final tree with minimum cost of given graph is



Time complexity : Time complexity is $O(|E| \log |E|)$.

Que 3.30. **What is minimum spanning tree ? Explain Prim's algorithm and find MST of graph Fig. 3.30.1.**

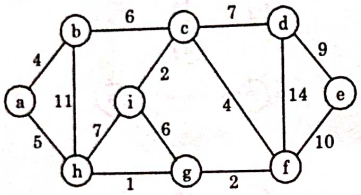


Fig. 3.30.1.

AKTU 2015-16, Marks 10

Answer

Minimum spanning tree : Refer Q. 3.25, Page 3–24B, Unit-3.
Prim's algorithm : Refer Q. 3.27, Page 3–25B, Unit-3.
Numerical :

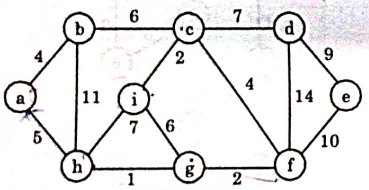
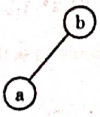
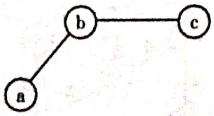


Fig. 3.30.2.

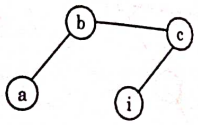
Let a be the source node. Select edge (a, b) as distance between edge (a, b) is minimum.



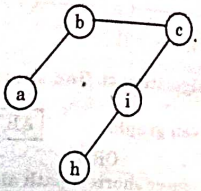
Now, select edge (b, c)



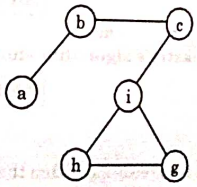
Now, select edge (c, i)



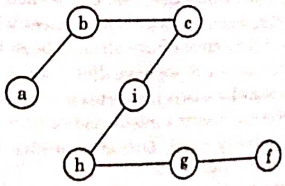
Now, select edge (i, h)



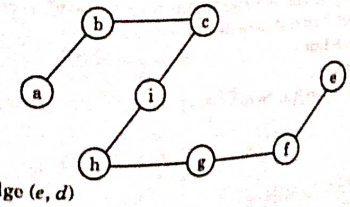
Now, select edge (h, g)



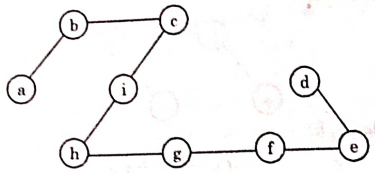
Now, select edge (g, f)



Now, select edge (f, e)



Now, select edge (e, d)



Thus, we obtained MST for Fig. 3.31.1.

Que 3.31. **Write an algorithm to find shortest path between all pairs of nodes in a given graph.** **AKTU 2013-14, Marks 10**

OR

Explain greedy single source shortest path algorithm with example. **AKTU 2015-16, Marks 10**

OR

Write short note on Dijkstra's algorithm shortest paths problems. **AKTU 2016-17, Marks 10**

Answer

1. Dijkstra's algorithm, is a greedy algorithm that solves the single source shortest path problem for a directed graph $G = (V, E)$ with non-negative edge weights, *i.e.*, we assume that $w(u, v) \geq 0$ each edge $(u, v) \in E$.
2. Dijkstra's algorithm maintains a set S of vertices whose final shortest path weights from the source s have already been determined.
3. That is, for all vertices $v \in S$, we have $d[v] = \delta(s, v)$.
4. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest path estimate, inserts u into S, and relaxes all edges leaving u.
5. We maintain a priority queue Q that contains all the vertices in $v - s$, keyed by their d values.
6. Graph G is represented by adjacency list.
7. Dijkstra's always chooses the "lightest or "closest" vertex in $V - S$ to insert into set S that it uses as a greedy strategy.

Dijkstra's algorithm :

DIJKSTRA (G, w, s)

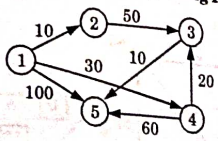
1. INITIALIZE-SINGLE-SOURCE (G, s)
2. $s \leftarrow \phi$
3. $Q \leftarrow V[G]$
4. while $Q \neq \phi$
5. do $u \leftarrow$ EXTRACT-MIN (Q)
6. $S \leftarrow S \cup \{u\}$
7. for each vertex $v \in$ Adj $[u]$

8. do RELAX (u, v, w)

RELAX (u, v, w) :

1. If $d[u] + w(u, v) < d[v]$
2. then $d[v] \leftarrow d[u] + w(u, v)$
3. $\pi[v] \leftarrow u$

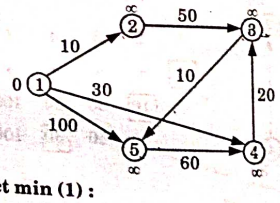
Que 3.32. **Find the shortest path in the below graph from the source vertex 1 to all other vertices by using Dijkstra's algorithm.**



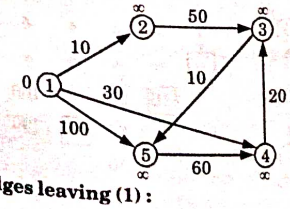
AKTU 2017-18, Marks 10

Answer

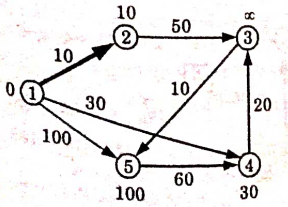
Initialize :



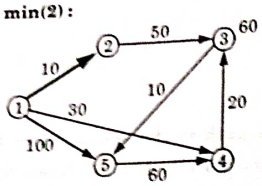
Extract min (1) :



All edges leaving (1) :

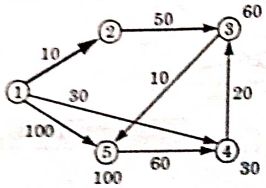


Extract min(2) :



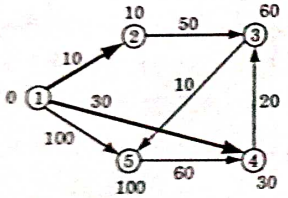
1	2	3	4	5
0	∞	∞	∞	∞
	10	60	30	100

All edges leaving (2) :



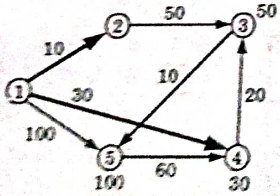
1	2	3	4	5
0	∞	∞	∞	∞
	10	60	30	100
		60	30	100

Extract min(4) :



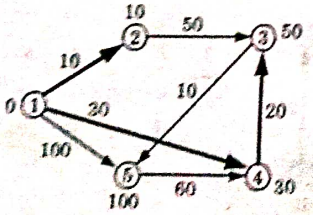
1	2	3	4	5
0	∞	∞	∞	∞
	10	60	30	100
		60	30	100

All edges leaving (4) :



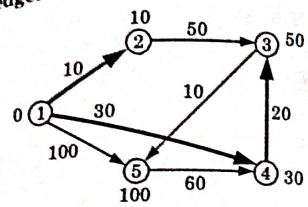
1	2	3	4	5
0	∞	∞	∞	∞
	10	60	30	100
		60	30	100
		50		

Extract min(3) :



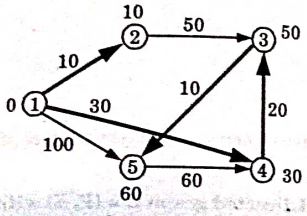
1	2	3	4	5
0	∞	∞	∞	∞
	10	∞	30	100
		60	30	100
		50		

All edges leaving (3) :



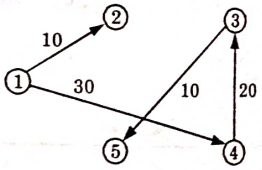
1	2	3	4	5
0	∞	∞	∞	∞
	10	∞	30	100
		60	30	100
		50		60

Extract min(5) :



1	2	3	4	5
0	∞	∞	∞	∞
	10	∞	30	100
		60	30	100
		50		60

Shortest path



Que 3.33. **State Bellman-Ford algorithm.**

AKTU 2016-17, Marks 7.5

Answer

1. Bellman-Ford algorithm finds all shortest path length from a source $s \in V$ to all $v \in V$ or determines that a negative-weight cycle exists.
2. Bellman-Ford algorithm solves the single source shortest path problem in the general case in which edges of a given digraph G can have negative weight as long as G contains no negative cycles.
3. This algorithm, uses the notation of edge relaxation but does not use with greedy method.
4. The algorithm returns boolean TRUE if the given digraph contains no negative cycles that are reachable from source vertex otherwise it returns boolean FALSE.

Bellman-Ford (G, w, s) :

1. INITIALIZE-SINGLE-SOURCE (G, s)
2. for each vertex $i \leftarrow 1$ to $V[G] - 1$ do
3. for each edge (u, v) in $E[G]$ do
4. RELAX (u, v, w)
5. For each edge (u, v) *in* $E[G]$ do
6. if $d[u] + w(u, v) < d[v]$ then
7. return FALSE
8. return TRUE

RELAX (u, v, w) :

1. If $d[u] + w(u, v) < d[v]$
2. then $d[v] \leftarrow d[u] + w(u, v)$
3. $\pi[v] \leftarrow u$

If Bellman-Ford returns true, then G forms a shortest path tree, else there exists a negative weight cycle.

Que 3.34. Given a weighted directed graph $G = (V, E)$ with source and weight function $W : E \rightarrow R$ then write an algorithm to solve a single source shortest path problem whose complexity is O (VE). Apply the same on the following graph.

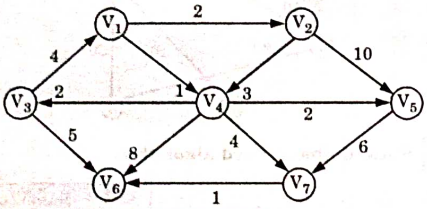


Fig. 3.34.1.

AKTU 2014-15, Marks 10

Answer

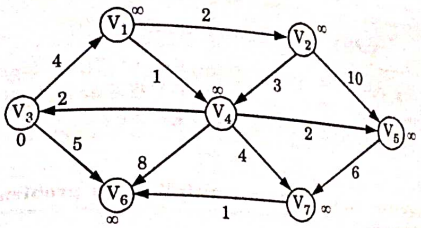
Bellman-Ford algorithm is used to solve single source shortest path problem whose complexity is O(VE).

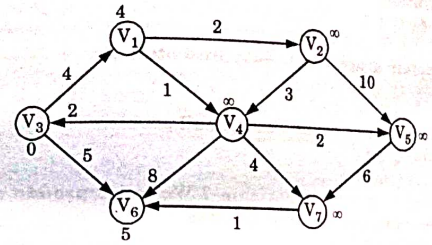
Bellman-Ford algorithm : Refer Q. 3.33, Page 3–37B, Unit-3.

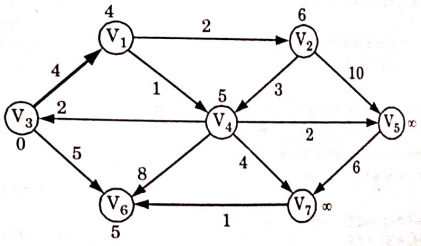
Numerical :

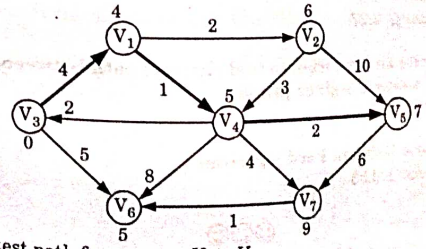
Let V_3 is source

Initialize :









Shortest path from vertex V_3 to V_5

$= d_{V_3V_1} + d_{V_1V_4} + d_{V_4V_5} = 4 + 1 + 2 = 7$

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. Discuss matrix chain multiplication problem and its solution.

Ans. Refer Q. 3.2.

Q. 2. Explain graphs with its representations.

Ans. Refer Q. 3.4.

Q. 3. Write short note on convex null problem.

Ans. Refer Q. 3.11.

Q. 4. What is greedy algorithm ? Write its pseudo code for recursive and iterative process.

Ans. Refer Q. 3.17.

Q. 5. Discuss knapsack problem.

Ans. Refer Q. 3.19.

Q. 6. Write short note on the following :

a. Minimum spanning tree

b. Kruskal's algorithm

c. Prim's algorithm

Ans.

a. Refer Q. 3.25.

b. Refer Q. 3.26.

c. Refer Q. 3.27.

Q. 7. Write an algorithm to find shortest path between all pairs of nodes in a given graph.

Ans. Refer Q. 3.31.

Q. 8. State Bellman Ford algorithm.

Ans. Refer Q. 3.33.

☺☺☺

4 UNIT Dynamic Programming, Backtracking and Branch and Bound

PART-1

Dynamic Programming with Examples such as Knapsack, All Pair Shortest Paths : Warshall's and Floyd's Algorithm, Resource Allocation Problem.

CONCEPT OUTLINE : PART-1

- **Dynamic programming :** It is a bottom-up technique. It is a stage-wise search method suitable for optimization problem whose solution may be viewed as a result of the sequence of decisions.
- **Floyd-Warshall algorithm :** It is a graph analysis algorithm for finding shortest path in a weighted directed graph.

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}, d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

- **Matrix multiplication :** To multiply two matrices, multiply the rows in the first matrix by the column of the second matrix.
- **Longest common subsequence :** Given two sequence x and y, we say that a sequence z is a common sequence of x and y if z is a subsequence of both x and y.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 4.1. What do you mean by dynamic programming ?

Answer

1. Dynamic programming is a stage-wise search method suitable for optimization problems whose solutions may be viewed as the result of a sequence of decisions.
2. The most attractive property of this strategy is that during the search of a solution it avoids full enumeration by pruning early partial decision solutions that cannot possibly lead to optimal solution.
3. In many practical situations, this strategy hits the optimal solution in a polynomial number of decision steps.
4. However, in the worst case, such a strategy may end up performing full enumeration.
5. Dynamic programming takes advantage of the duplication and arranges to solve each sub-problem only once, saving the solution (in table or something) for later use.

6. The underlying idea of dynamic programming is to avoid calculating the same stuff twice, usually by keeping a table of known results of sub-problems.
7. Unlike divide and conquer which solve the sub-problems top-down, a dynamic programming is a bottom-up technique.
8. Bottom-up means :
   a. Start with the smallest sub-problems.
   b. Combining their solutions.
   c. Obtain the solutions to sub-problems of increasing size until we arrive at the solution of the original problem.
9. It is used to solve optimization problems which often could require "testing" many possible solutions. The single best solution is called the optimal solution. Dynamic programming includes four steps :

   Step 1 : Characterize the structure of an optimal solution.

   Step 2 : Recursively define the value of an optimal solution.

   Step 3 : Compute the value of an optimal solution bottom-up.

   Step 4 : Construct an optimal solution from computed information.

Que 4.2. What is the principle of optimality ? Also give approaches in dynamic programming.

Answer

Principle of optimality : Principle of optimality states that whatever the initial state is, remaining decisions must be optimal with regard the state following from the first decision.

Approaches in dynamic programming :

There are two approaches of solving dynamic programming problems :

1. **Bottom-up approach :** Bottom-up approach simply means storing the results of certain calculations, which are then re-used later because the same calculation is a sub-problem in a larger calculation.
2. **Top-down approach :** Top-down approach involves formulating a complex calculation as a recursive series of simpler calculations.

Que 4.3. Discuss the elements of dynamic programming.

Answer

Following are the elements of dynamic programming :

1. **Optimal sub-structure :** Optimal sub-structure holds if optimal solution contains optimal solutions to sub-problems. It is often easy to show the optimal sub-problem property as follows :
   i. Split problem into sub-problems.

ii. Sub-problems must be optimal, otherwise the optimal splitting would not have been optimal.

There is usually a suitable "space" of sub-problems. Some spaces are more "natural" than others. For matrix chain multiply we choose sub-problems as sub-chains.

2. Overlapping sub-problem :

i. Overlapping sub-problem is found in those problems where bigger problems share the same smaller problems. This means, while solving larger problems through their sub-problems we find the same sub-problems in two or more different large problems. In these cases a sub-problem is usually found to be solved previously.

ii. Overlapping sub-problems can be found in Matrix Chain Multiplication (MCM) problem.

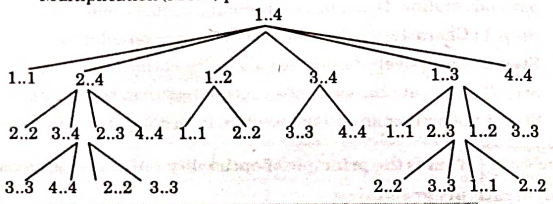


Fig. 4.3.1. The recursion tree for the computation of Recursive-Matrix-Chain (p, 1, 4).

3. Memoization :

i. The memoization technique is the method of storing values of solutions to previously solved problems.

ii. This generally means storing the values in a data structure that helps us reach them efficiently when the same problems occur during the program's execution.

iii. The data structure can be anything that helps us do that but generally a table is used.

Que 4.4. What is Matrix Chain Multiplication Problem ? Also, give a solution for matrix chain multiplication problem.

Answer

Refer Q. 3.2, Page 3–3B, Unit-3.

Que 4.5. Write down an algorithm to compute Longest Common Subsequence (LCS) of two given strings and analyze its time complexity. **AKTU 2017-18, Marks 10**

Answer

LCS-Length (X, Y) :

1. $m \leftarrow$ length[X]
2. $n \leftarrow$ length[Y]
3. for $i \leftarrow 1$ to m
4. do $c[i, 0] \leftarrow 0$
5. for $j \leftarrow 0$ to n
6. do $c[0, j] \leftarrow 0$
7. for $i \leftarrow 1$ to m
8. do for $j \leftarrow 1$ to n
9. do if $x_i = y_j$
10. then $c[i, j] \leftarrow c[i-1, j-1] + 1$
11. $b[i, j] \leftarrow$ "↖"
12. else if $c[i-1, j] \geq c[i, j-1]$
13. then $c[i, j] \leftarrow c[i-1, j]$
14. $b[i, j] \leftarrow$ "↑"
15. else $c[i, j] \leftarrow c[i, j-1]$
16. $b[i, j] \leftarrow$ "←"
17. return c and b

Note :

1. "↖" means both the same.
2. "↑" means $c[i-1, j] \geq c[i, j-1]$.
3. "←" means $c[i-1, j] < c[i, j-1]$.
4. The "↖" diagonal arrows lengthen the LCS.

Since, two for loops are present in LCS algorithm first for loop runs upto m times and second for loop runs upto n times. So, time complexity of LCS is $O(mn)$.

Que 4.6. Give the algorithm of dynamic 0/1 knapsack problem.

Answer

Knapsack problem :

```
1. Knapsack(v, w, n, W)
2. for (w = 0 to W) V[0, w] = 0
3. for (i = 1 to n)
4.       for (w = 0 to W)
5.             if (w[i] ≤ w) then
6.                   V[i, w] = max{V[i − 1, w], v[i] + V[i − 1, w − w[i]]};
7.             else V[i, w] = V[i − 1, w];
8. return V[n, W];
```

Que 4.7. **Differentiate between dynamic programming and greedy approach. What is 0/1 knapsack problem ? Solve the following instance using dynamic programming. Write the algorithm also. Knapsack Capacity = 10, P = < 1, 6, 18, 22, 28 > and w = < 1, 2, 5, 6, 7>.**

AKTU 2014-15, Marks 10

Answer

S. No.	Dynamic programming	Greedy approach
1.	Solves every optimal sub-problem.	Do not solve every optimal problem.
2.	We make a choice at each step, but the choice may depend on the solutions to sub-problem.	We make whatever choice seems best at the moment and then solve the problem.
3.	It is bottom-up approach.	It is top-down approach.
4.	Dynamic programming works when a problem has following properties : a. Optimal sub-structure b. Overlapping sub-problems	Greedy algorithm works when a problem exhibits the following properties : a. Greedy choice property b. Optimal sub-structure

0/1 knapsack problem :

The 0/1 knapsack problem is defined as follows :

1. Given, a knapsack of capacity c and n items of weights $\{w_1, w_2, ..., w_n\}$ and profits $\{p_1, p_2, ..., p_n\}$, the objective is to choose a subset of n objects that fits into the knapsack and that maximizes the total profit.
2. Consider a knapsack (bag) with a capacity of c.
3. We select items from a list of n items.
4. Each item has both a weight of w_i and profit of p_i.
5. In a feasible solution, the sum of the weights must not exceed the knapsack capacity (c) and an optimal solution is both feasible and reaches the maximum profit.
6. An optimal packing is a feasible solution one with a maximum profit,

$$p_1x_1 + p_2x_2 + p_3x_3 + ... + p_nx_n = \sum_{i=1}^{n} p_ix_i$$

which is subjected to constraints :

$$w_1x_1 + w_2x_2 + w_3x_3 + ... + w_nx_n = \sum_{i=1}^{n} w_ix_i \le c$$

and $x_i = 1$ or 0, $1 \le i \le n$

7. We have to find the values of x_i where $x_i = 1$ if i^{th} item is packed into the knapsack and $x_i = 0$ if i^{th} item is not packed.

Item	w_i	$p_i = v_i/w_i$ (Given)	v_i
I_1	1	1	1
I_2	2	6	12
I_3	5	18	90
I_4	6	22	132
I_5	7	28	196

Now, fill the knapsack according to given value of p_i.
First we choose item I_1 whose weight is 1, then choose item I_2 whose weight is 2 and item I_3 whose weight is 5.
∴ Total weight in knapsack : 1 + 2 + 5 = 8
Now, the next item is I_4 and its weight is 6, but we want only 2 [∵ W = 10]. So we choose fractional part of it *i.e.*,

The value of fractional part of I_4 is = $\frac{132}{6} \times 2 = 44$

Thus the maximum value is : = 1 + 12 + 90 + 44 = 147

$$\begin{bmatrix} 2 \\ 5 \\ 2 \\ 1 \end{bmatrix} = 10$$

Que 4.8. **Discuss knapsack problem with respect to dynamic programming approach. Find the optimal solution for given problem, w (weight set) = {5, 10, 15, 20} and W (Knapsack size) = 25 and v = {50, 60, 120, 100}.**

AKTU 2015-16, Marks 10

Answer

Knapsack problem : Refer Q. 4.7, Page 4–6B, Unit-4.
Numerical :

w = {5, 10, 15, 20}
W = 25
v = {50, 60, 120, 100}

Initially,

Item	w_i	v_i
I_1	5	50
I_2	10	60
I_3	15	120
I_4	20	100

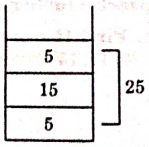
Taking value per weight ratio, *i.e.*, $p_i = v_i/w_i$

Item	w_i	v_i	$p_i = v_i/w_i$
I_1	5	50	10
I_2	10	60	6
I_3	15	120	8
I_4	20	100	5

Now, arrange the value of p_i in decreasing order.

Item	w_i	v_i	$p_i = v_i/w_i$
I_1	5	50	10
I_3	15	120	8
I_2	10	60	6
I_4	20	100	5

Now, fill the knapsack according to decreasing value of p_i.
First we choose item I_1 whose weight is 5, then choose item I_3 whose weight is 15. Now the total weight in knapsack is 5 + 15 = 20. Now, next item is I_2 and its weight is 10, but we want only 5. So, we choose fractional part of it, *i.e.*,



The value of fractional part of I_2 is,

$$= \frac{60}{10} \times 5 = 30$$

Thus, the maximum value is,

$$= 50 + 120 + 30 = 200$$

Que 4.9. Describe the Warshall's and Floyd's algorithm for finding all pairs shortest paths.

Answer

1. Floyd-Warshall algorithm is a graph analysis algorithm for finding shortest paths in a weighted, directed graph.
2. A single execution of the algorithm will find the shortest path between all pairs of vertices.

3. It does so in $\Theta(V^3)$ time, where V is the number of vertices in the graph.
4. Negative-weight edges may be present, but we shall assume that there are no negative-weight cycles.
5. The algorithm considers the "intermediate" vertices of a shortest path, where an intermediate vertex of a simple path $p = (v_1, v_2, ..., v_m)$ is any vertex of p other than v_1 or v_m, that is, any vertex in the set $\{v_2, v_3, ..., v_{m-1}\}$.
6. Let the vertices of G be $V = \{1, 2, ..., n\}$, and consider a subset $\{1, 2, ..., k\}$ of vertices for some k.
7. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, ..., k\}$, and let p be a minimum-weight path from among them.
8. Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, ..., k\}$.
   A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

Floyd-Warshall (W) :

1. $n \leftarrow$ rows $[W]$
2. $D^{(0)} \leftarrow W$
3. for $k \leftarrow 1$ to n
4. do for $i \leftarrow 1$ to n
5. do for $j \leftarrow 1$ to n
6. do $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
7. return $D^{(n)}$

Que 4.10. Apply Floyd-Warshall algorithm for constructing shortest path. Show the matrices $D^{(k)}$ and $\pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph.

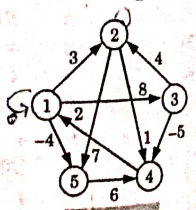


Fig. 4.10.1.

Answer

$$d_{ij}^{(k)} = \min[d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}]$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} \text{ if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} \text{ if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

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

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

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

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

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

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

Que 4.11. Describe the maximum flow and maximum flow problem. Write the suitable algorithm to calculate the maximum flow.

Answer

1. Maximum flow is a maximum feasible flow between source and sink. Here the word flow means the rate at which the material moves through underlying object. But when the material moves through some object it is very much essential to consider its capacity.
2. In maximum flow problem, we want to compute the greatest rate at which material can be moved or travelled from source to sink without violating any capacity constraints.
3. Ford-Fulkerson algorithm is one of the efficient and simple algorithm for computing maximum flow.

Ford-Fulkerson algorithm :

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 $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \text{ is in } p\}$
6. for each edge (u, v) in p
7. do $f[u, v] \leftarrow f[u, v] + c_f(p)$
8. $f[v, u] \leftarrow -f(u, v)$

PART-2

Backtracking, Branch and Bound with Examples such as Travelling Salesman Problem, Graph Colouring, N-Queen Problem, Hamiltonian Cycles and Sum of Subsets.

CONCEPT OUTLINE : PART-2

- **Backtracking :** Backtracking is a methodical way of trying out various sequences of decisions, until we found one that "works".
  Backtracking algorithm is one which backtracks with intelligence.
- **Hamiltonian circuit problem :** In a graph $G = (V, E)$, we find Hamiltonian circuit using backtracking. If we found a

Hamiltonian cycle then the search using backtracking is successful.

- **Sum-subset problem :** In this problem, we have to find a subset S' of given set $S = (S_1, S_2, S_3, \ldots, S_n)$ where the elements of set S are n positive integers.
The sum-subset problem can be solved by using backtracking approach.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 4.12. **What is backtracking ? Write general iterative algorithm for backtracking.** **AKTU 2016-17, Marks 10**

Answer

1. Backtracking is a general algorithm for finding all solutions to some computational problems.
2. Backtracking is an important tool for solving constraint satisfaction problems, such as crosswords, verbal arithmetic, and many other puzzles.
3. It is often the most convenient (if not the most efficient) technique for parsing, for the knapsack problem and other combinational optimization problem.
4. It can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution.

Iterative backtracking algorithm :

```
algorithm ibacktrack (n)
// Iterative backtracking process
// All solutions are generated in x[1 : n] and printed as soon as they are found
{
k = 1;
while (k != 0)
{
if ( there remains an untried x[k] in T(x[1], x[2], ..., x[k - 1])
and B_k(x[1], ..., x[k] ) is true )
{
if (x[1], ..., x[k] is a path to an answer node )
write (x[1 : k]);
k = k + 1;                    // Consider the next set
}
else
k = k - 1;                    // Backtrack to the previous set
}
}
```


Que 4.13. **What is backtracking ? What are the types of problems in backtracking ?**

Answer

Backtracking : Refer Q. 4.12, Page 4–12B, Unit-4.

In backtracking, the problem can be categorized into three types :

1. **Decision problem :** In decision problem we find "whether there is any feasible solution ?".
2. **Optimization problem :** In optimization problems we find "whether there exists any best solution ?".
3. **Enumeration problem :** In enumeration problem we find all the possible feasible solutions.

Que 4.14. **Write short notes on N-Queens problem.**

AKTU 2013-14, Marks 05

OR

Write pseudocode for 8-Queens problem.

AKTU 2016-17, Marks 10

Answer

1. In N-Queens problem, the idea is to place queens one by one in different columns, starting from the leftmost column.
2. When we place a queen in a column, we check for clashes with already placed queens.
3. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution.
4. If we do not find such a row due to clashes then we backtrack and return false.

Procedure for solving N-Queens problem :

1. Start from the leftmost column.
2. If all queens are placed return true.
3. Try all rows in the current column. Do following for every tried row :

a. If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.

b. If placing queen in [row, column] leads to a solution then return true.

c. If placing queen does not lead to a solution then unmark this [row, column] (backtrack) and go to step (a) to try other rows.

4. If all rows have been tried and nothing worked, return false to trigger backtracking.

Algorithm/pseudocode for *N*-Queens problem :

N-Queens are to be placed on an $n \times n$ chessboard so that no two attack *i.e.*, no two Queens are on the same row, column or diagonal.

PLACE (k, i)

1. for $j \leftarrow 1$ to $k-1$
2. do if $(x(j) = i)$ or Abs $(x[j] - i) =$ (Abs $(j - k)$)
3. then return false
4. return true

Place (k, i) returns true if a queen can be placed in the k^{th} row and i^{th} column otherwise return false.

$x[\,]$ is a global array whose first $k-1$ values have been set. Abs(r) returns the absolute value of r.

N*-Queens *(k, n)

1. for $i \leftarrow 1$ to n
2. do if PLACE (k, i)
3. then $x[k] \leftarrow i$
4. if $k = n$, then print $x[1 \ldots N]$
5. else N-Queens $(k + 1, n)$

For 8-Queen problem put $n = 8$ in the algorithm.

Que 4.15. Write an algorithm for solving *N*-Queens problem. Show the solution of 4-Queens problem using backtracking approach. **AKTU 2015-16, Marks 10**

Answer

Algorithm for *N*-Queens problem : Refer Q. 4.14, Page 4–13B, Unit-4.

4-Queens problem :

1. Suppose we have 4 × 4 chessboard with 4-queens each to be placed in non-attacking position.

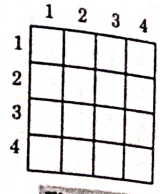


Fig. 4.15.1.

2. Now, we will place each queen on a different row such that no two queens attack each other.
3. We place the queen q_1 in the very first accept position (1, 1).
4. Now if we place queen q_2 in column 1 and 2 then the dead end is encountered.
5. Thus, the first acceptable position for queen q_2 is column 3 *i.e.*, (2, 3) but then no position is left for placing queen q_3 safely. So, we backtrack one step and place the queen q_2 in (2, 4).
6. Now, we obtain the position for placing queen q_3 which is (3, 2). But later this position lead to dead end and no place is found where queen q_3 can be placed safely.

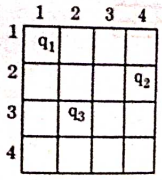


Fig. 4.15.2.

7. Then we have to backtrack till queen q_1 and place it to (1, 2) and then all the other queens are placed safely by moving queen q_2 to (2, 4), queen q_3 to (3, 1) and queen q_4 to (4, 3) *i.e.*, we get the solution < 2, 4, 1, 3>. This is one possible solution for 4-queens problem.

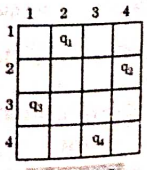


Fig. 4.15.3.

8. For other possible solution the whole method is repeated for all partial solutions. The other solution for 4-queens problem is <3, 1, 4, 2> *i.e.*,

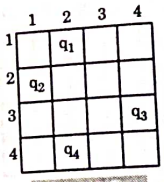


Fig. 4.15.4.

9. Now, the implicit tree for 4-queen for solution <2, 4, 1, 3> is as follows:

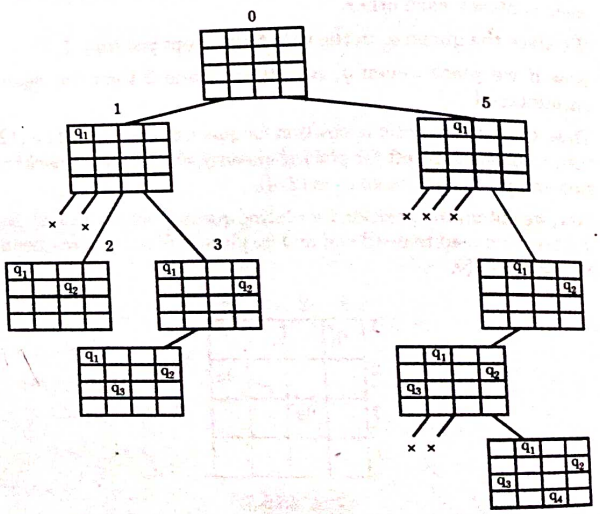


Fig. 4.15.5.

10. Fig. 4.15.5 shows the complete state space for 4-queens problem. But we can use backtracking method to generate the necessary node and stop if next node violates the rule *i.e.*, if two queens are attacking.

Que 4.16. Describe backtracking algorithm for Travelling Salesman Problem (TSP). Show that a TSP can be solved using backtracking method in the exponential time.

OR

Explain TSP (Travelling Salesman) problem with example. Write an approach to solve TSP problem.

AKTU 2015-16, 2013-14; Marks 10

Answer

Travelling Salesman Problem (TSP):

Travelling salesman problem is the problem to find the shortest possible route for a given set of cities and distance between the pair of cities that visits every city exactly once and returns to the starting point.

Backtracking algorithm for the TSP:

1. Let G be the given complete graph with positive weights on its edges.
2. Use a search tree that generates all permutations of $V = \{1 \ldots n\}$, specifically the one illustrated in Fig. 4.16.1 for the case $n = 3$.

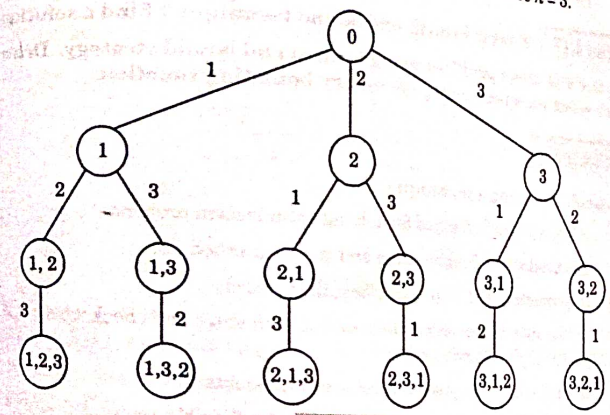


Fig. 4.16.1.

3. A node at depth i of this tree (the root is at depth 0) stores an i-permutation of $\{1, \ldots, n\}$. A leaf stores a permutation of $\{1, \ldots, n\}$, which is equivalent to saying that it stores a particular Hamiltonian cycle (tour) of G.
4. For the travelling salesman problem, we will not do any static pruning on this tree, we will do dynamic pruning, during the search.

Proof:

1. At some point during the search, let v be a non-leaf node of this tree that is just being visited, and let w be the weight of the shortest tour found to this point.
2. Let $(\pi_1 \ldots \pi_k)$ be the k-permutation of $\{1 \ldots n\}$ stored at v.

   Let, $w_v = \sum_{i=1 \ldots k-1} w_{\pi_i \pi_{i+1}}$ denote the sum of the weights on edges whose endpoints are adjacent vertices in this k-permutation.

3. Then, if $w_v \geq w$, the entire subtree of the search tree rooted at v can be pruned, i.e., not searched at all, since every leaf of this subtree represents a tour whose weight must be greater than w_v.
4. This follows from the fact that all edges in the graph have positive weights.
5. There are atmost $O(n*2^n)$ subproblem, and each one takes linear time to solve.
6. The total running time is therefore $O(n^{2}*2^n)$.
7. The time complexity is much less than $O(n!)$ but still exponential.

Hence proved.

Que 4.17. What is branch and bound technique ? Find a solution to the 4-Queens problem using branch and bound strategy. Draw the solution space using necessary bounding function.

Answer

Branch and bound technique :

1. It is a systematic method for solving optimization problems.
2. It is used where backtracking and greedy method fails.
3. It is sometimes also known as best first search.
4. In this approach we calculate bound at each stage and check whether it is able to give answer or not.
5. Branch and bound procedure requires two tools :
   a. The first one is a way of covering the feasible region by several smaller feasible sub-regions. This is known as branching.
   b. Another tool is bounding, which is a fast way of finding upper and lower bounds for the optimal solution within a feasible sub-region.

Solution to 4-Queens problem :

Basically, we have to ensure 4 things :

1. No two queens share a column.
2. No two queens share a row.
3. No two queens share a top-right to left-bottom diagonal.
4. No two queens share a top-left to bottom-right diagonal.

Number 1 is automatic because of the way we store the solution. For number 2, 3 and 4, we can perform updates in O(1) time and the whole updation process is shown in Fig. 4.17.1.

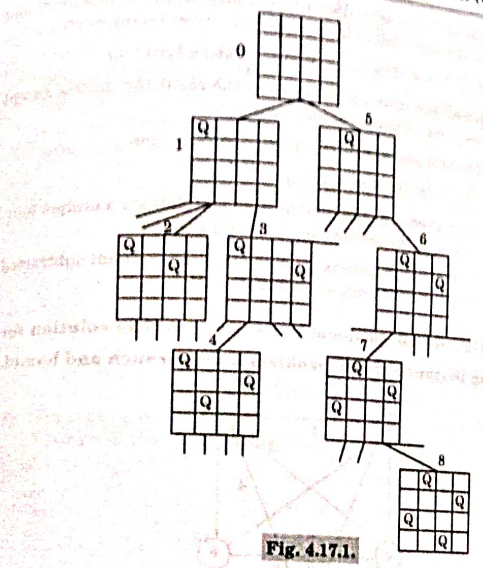


Fig. 4.17.1.

Que 4.18. Explain the working rule for Travelling Salesman Problem using branch and bound technique.

Answer

For solving travelling salesman problem we represent the solution space by a state space tree. We define three cost functions C, l and u where $l_i \leq C_i \leq u_i$ for all the nodes i of the state space tree.

Step 1 : First find the cost matrix as given cost on the edges of the graph.

Step 2 : Now, find reduced matrix by subtracting the smallest element from row i (column j) introduce a zero in to row i (column j). Repeating this process as often as needed, the cost matrix can be reduced.

Add the total amount subtracted from the column and rows and make this as the root of the state space tree.

Step 3 : Let M be the reduced cost matrix for node A and let B be a child of A such that the tree edge (A, B) corresponds to inclusion of edge (i, j) in the tour. If B is not a leaf, then the reduced cost matrix for B may be obtained by apply following given steps :

a. Change all the entries in row i, column j of M to ∞. This includes use of any more edges leaving vertex i or entering vertex j.

b. Set $M(j, 1) = \infty$. This excludes use of the edge $(j, 1)$.

c. Reduce all the rows and columns in the resulting matrix except for rows and columns containing only ∞.

Suppose T is the total amount subtracted in step (c), then

$$I(B) = l(A) + M(i, j) + T$$

For leaf nodes $l = c$ is easily computed as each leaf defines a unique tour. For the upper bound u, we assume $u_i = \infty$ for all nodes i.

Step 4 : Find the root of the node as, combine the total amount subtracted from cost matrix to find the reduced cost matrix M.

Que 4.19. Define TSP problem in detail. Find the solution for the following instance of TSP problem using branch and bound.

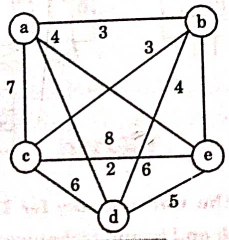


Fig. 4.19.1.

AKTU 2014-15, Marks 10

Answer

Travelling salesman problem : Refer Q. 4.16, Page 4–16B, Unit-4.

Numerical :

$$\text{Cost matrix} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} \infty & 3 & 4 & 2 & 7 \\ 3 & \infty & 4 & 6 & 3 \\ 4 & 4 & \infty & 5 & 8 \\ 2 & 6 & 5 & \infty & 6 \\ 7 & 3 & 8 & 6 & \infty \end{bmatrix} \end{array}$$

1. Reduce matrix (M) can be obtained

Subtract 2 from row 1
Subtract 3 from row 2
Subtract 4 from row 3
Subtract 2 from row 4
Subtract 3 from row 5

$$M = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} \infty & 1 & 2 & 0 & 5 \\ 0 & \infty & 1 & 3 & 0 \\ 0 & 0 & \infty & 1 & 4 \\ 0 & 4 & 3 & \infty & 4 \\ 4 & 0 & 5 & 3 & \infty \end{bmatrix} \end{array}$$

2. So, total expected cost is : $2 + 3 + 4 + 2 + 3 = 14$

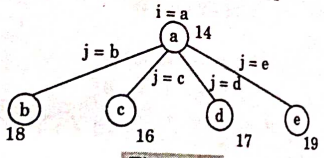


Fig. 4.19.2.

3. For edge (a, b)
Change all the elements in 1st row and 2nd column.

$$M_1 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 1 & 3 & 0 \\ 0 & \infty & \infty & 1 & 4 \\ 0 & \infty & 3 & \infty & 4 \\ 4 & \infty & 5 & 3 & \infty \end{bmatrix}$$

Set $M(j, 1) = \infty$
$\Rightarrow M_1(b, 1) = \infty$

$$M_1 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 1 & 3 & 0 \\ 0 & \infty & \infty & 1 & 4 \\ 0 & \infty & 3 & \infty & 4 \\ 1 & \infty & 2 & 0 & \infty \end{bmatrix}$$

Minimum cost

$$l(b) = l(a) + M_1(a, b) + T$$
$$= 14 + 1 + 3 = 18$$

For edge (a, c)

$$M_2 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 3 & 0 \\ 0 & 0 & \infty & 1 & 4 \\ 0 & 4 & \infty & \infty & 4 \\ 4 & 0 & \infty & 3 & \infty \end{bmatrix}$$

Set $M_2(C, 1) = \infty$

$$M_2 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 3 & 0 \\ \infty & 0 & \infty & 1 & 4 \\ 0 & 4 & \infty & \infty & 4 \\ 4 & 0 & \infty & 3 & \infty \end{bmatrix}$$

Minimum cost

$$l(c) = l(a) + M_2(a, c) + T$$
$$= 14 + 2 + 0 = 16$$

For edge (a, d)

$$M_3 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 1 & \infty & 0 \\ 0 & 0 & \infty & \infty & 4 \\ 0 & 4 & 3 & \infty & 4 \\ 4 & 0 & 5 & \infty & \infty \end{bmatrix}$$

Set $M_3(d, 1) = \infty$

$$M_3 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 1 & \infty & 0 \\ 0 & 0 & \infty & \infty & 4 \\ \infty & 4 & 3 & \infty & 4 \\ 4 & 0 & 5 & \infty & \infty \end{bmatrix}$$

Reduce matrix,

$$M_3 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 1 & \infty & 0 \\ 0 & 0 & \infty & \infty & 4 \\ \infty & 1 & 0 & \infty & 1 \\ 4 & 0 & 5 & \infty & \infty \end{bmatrix}$$

Minimum cost

$$l(d) = l(a) + M_3(a, d) + T$$
$$= 14 + 0 + 3 = 17$$

For edge (a, e)

$$M_4 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 1 & 3 & \infty \\ 0 & 0 & \infty & 1 & \infty \\ \infty & 4 & 3 & \infty & \infty \\ 4 & 0 & 5 & 3 & \infty \end{bmatrix}$$

Set $M_4(e, 1) = \infty$

$$M_4 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 1 & 3 & \infty \\ 0 & 0 & \infty & 1 & \infty \\ 0 & 4 & 3 & \infty & \infty \\ \infty & 0 & 5 & 3 & \infty \end{bmatrix}$$

Minimum cost

$$l(e) = l(a) + M_4(a, e) + T$$
$$= 14 + 5 + 0 = 19$$

We find that path (a, c) have minimum cost 16. Now, we expand the node c.

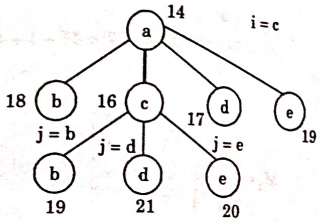


Fig. 4.19.3.

For edge (c, b)

$$M_1 = \begin{bmatrix} \infty & \infty & 2 & 0 & 5 \\ 0 & \infty & 1 & 3 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 3 & \infty & 4 \\ 4 & \infty & 5 & 3 & \infty \end{bmatrix}$$

Set $M_1(b, 1) = \infty$

$$M_1 = \begin{bmatrix} \infty & \infty & 2 & 0 & 5 \\ \infty & \infty & 1 & 3 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 3 & \infty & 4 \\ 4 & \infty & 5 & 3 & \infty \end{bmatrix}$$

Reduce matrix

$$M_1 = \begin{bmatrix} \infty & \infty & 2 & 0 & 5 \\ \infty & \infty & 1 & 3 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 3 & \infty & 4 \\ 1 & \infty & 2 & 0 & \infty \end{bmatrix}$$

Main cost $l(b) = l(c) + M_1(c, b) + T = 16 + 0 + 3 = 19$

For edge (c, d) and set $M_2(d, 1) = \infty$

$$M_2 = \begin{bmatrix} \infty & 1 & 2 & \infty & 5 \\ 0 & \infty & 1 & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 4 & 3 & \infty & 4 \\ 4 & \infty & 5 & \infty & \infty \end{bmatrix}$$

Reduce matrix $M_2 = \begin{bmatrix} \infty & 0 & 1 & \infty & 4 \\ 0 & \infty & 1 & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 1 & 0 & \infty & 1 \\ 4 & 0 & 5 & \infty & \infty \end{bmatrix}$

Main cost $l(b) = l(c) + M_2(c, b) + T = 16 + 1 + 4 = 21$

For edge (c, e)
Set $M_3(e, 1) = \infty$

$$M_3 = \begin{bmatrix} \infty & 1 & 2 & 0 & \infty \\ 0 & \infty & 1 & 3 & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 4 & 3 & \infty & \infty \\ \infty & 0 & 5 & 3 & \infty \end{bmatrix}$$

$l(e) = l(c) + M_3(c, b) + T = 16 + 4 + 0 = 20$

We find that path (c, b) have minimum cost 19. Now, we expand the node b.

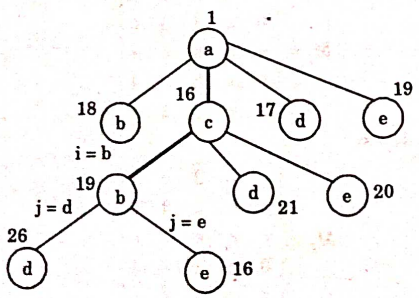


Fig. 4.19.4.

For edge (b, d) at set $M_1(d, 1) = 0$

$$M_1 = \begin{bmatrix} \infty & 1 & 2 & \infty & 5 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & 4 \\ \infty & 4 & 3 & \infty & 4 \\ 4 & 0 & 5 & \infty & \infty \end{bmatrix}$$

Reduce $M_1 = \begin{bmatrix} \infty & 0 & 1 & \infty & 4 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & 4 \\ \infty & 1 & 0 & \infty & 1 \\ 4 & 0 & 5 & \infty & \infty \end{bmatrix}$

$l(d) = l(b) + M_1(b, d) + T$
$= 19 + 3 + 4 = 26$

For (b, e) set $M_2(e, 1) = \infty$

$$M_2 = \begin{bmatrix} \infty & 1 & 2 & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & 1 & \infty \\ 0 & 4 & 3 & \infty & \infty \\ \infty & 0 & 5 & 3 & \infty \end{bmatrix}$$

$l(e) = l(b) + M_2(b, e) + T$
$= 16 + 0 + 0 + = 16$

We find that (b, d) have minimum cost 16.
Now, we expand the node e.

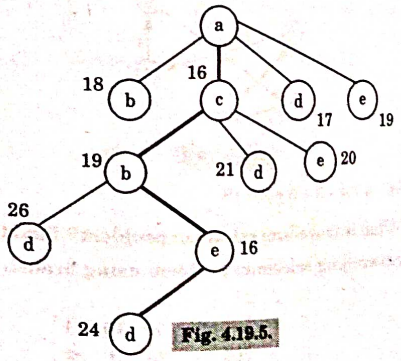


Fig. 4.19.5.

For edge (e, d)
Set $M_1(d, 1) = \infty$

$$M_1 = \begin{bmatrix} \infty & 1 & 2 & \infty & 5 \\ 0 & \infty & 1 & \infty & 0 \\ 0 & 0 & \infty & \infty & 4 \\ \infty & 4 & 3 & \infty & 4 \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Reduce matrix

$$M_1 = \begin{bmatrix} \infty & 0 & 1 & \infty & 4 \\ 0 & \infty & 1 & \infty & 0 \\ 0 & 0 & \infty & \infty & 4 \\ \infty & 1 & 0 & \infty & 1 \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

$l(d) = 4 + 16 + 4 = 24$

For edge (d, a)

Set $M_2(a, 1) = \infty$

$$M_2 = \begin{bmatrix} \infty & 1 & 2 & 0 & 5 \\ \infty & \infty & 1 & 3 & 0 \\ \infty & 0 & \infty & 1 & 4 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 5 & 3 & \infty \end{bmatrix}$$

$l(a) = l(d) + M_2(d, a) + T$

$= 24 + 0 + 0 = 24$

Therefore optimal tour for *TSP* will be

$a \rightarrow c \rightarrow b \rightarrow e \rightarrow d \rightarrow a$

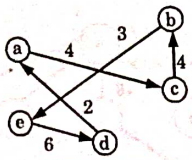


Fig. 4.19.6.

Total cost = 4 + 4 + 3 + 6 + 2 = 19

Que 4.20. What is travelling salesman problem ? Find the solution of following travelling salesman problem using branch and bound method.

$$\text{Cost matrix} = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 6 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

AKTU 2016-17, Marks 10

Answer

Travelling salesman problem : Refer Q. 4.16, Page 4–16B, Unit-4.

Numerical :

$$\text{Cost matrix} = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 6 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

1. Reduce each column and row by reducing the minimum value from each element in row and column.

Row

$$\begin{matrix} 10 \rightarrow \\ 2 \rightarrow \\ 2 \rightarrow \\ 3 \rightarrow \\ 4 \rightarrow \end{matrix} \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 3 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

Column

$$\begin{matrix} & 1 \downarrow & & 3 \downarrow & \\ \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 0 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix} = M_1$$

2. So, total expected cost is : 10 + 2 + 2 + 3 + 4 + 1 + 3 = 25.
3. We have discovered the root node V_1 so the next node to be expanded will be V_2, V_3, V_4, V_5. Obtain cost of expanding using cost matrix for node 2.
4. Change all the elements in 1st row and 2nd column.

$$M_2 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 0 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

5. Now, reducing M_2 in rows and columns, we get :

$$M_2 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 0 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

6. $\therefore$ Total cost for $M_2 = 25 + 10 + 0 = 35$

Similarly, for node 3, we have :

$$M_3 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{bmatrix}$$

7. Now, reducing M_3, we get :

$$M_3 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{bmatrix}$$

$\therefore$ Total cost for $M_3 = 25 + 17 + 0 = 42$

8. Similarly, for node 4, we have :

$$M_4 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ 15 & 3 & 0 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

9. Now, reducing M_4, we get :

$$M_4 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ 15 & 3 & 0 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

$\therefore$ Total cost = 25 + 0 + 0 = 25

10. Similarly, for node 5, we have :

$$M_5 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 0 & \infty & \infty \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

11. Now, reducing M_5, we get :

$$M_5 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 0 & \infty & \infty \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

$\therefore$ Total cost = 25 + 1 + 2 = 28

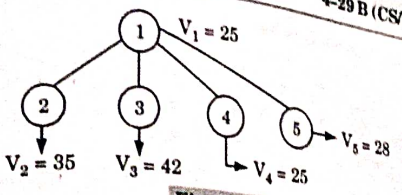


Fig. 4.20.1.

12. Now, the promising node is $V_4 = 25$. Now, we can expand V_2, V_3 and V_5. Now, the input matrix will be M_4.

13. Change all the elements in 4th row and 2nd column.

$$M_6 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

14. On reducing M_6, we get :

$$M_6 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

$\therefore$ Total cost = 25 + 3 + 0 = 28

15. Similarly, for node 3, we have :

$$M_7 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

16. On reducing M_7, we get :

$$M_7 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

$\therefore$ Total cost = 25 + 0 + 0 = 25

17. Similarly, for node 5, we have :

$$M_8 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

18. On reducing M_8, we get :

$$M_8 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

$\therefore$ Total cost = 25 + 0 + 11 = 36

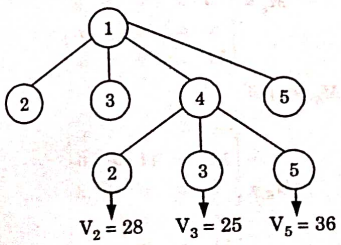


Fig. 4.20.2.

19. Now, promising node is $V_3 = 25$. Now, we can expand V_2 and V_5. Now, the input matrix will be M_7.

20. Change all the elements in 3rd row and 2nd column.

$$M_9 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix}$$

21. On reducing M_9, we get :

$$M_9 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

$\therefore$ Total cost = 25 + 3 + 0 = 28

22. Similarly, for node 5, we have :

$$M_{10} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 3 & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

23. On reducing M_{10}, we get :

$$M_{10} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

$\therefore$ Total cost = 25 + 2 + 12 + 3 = 42.

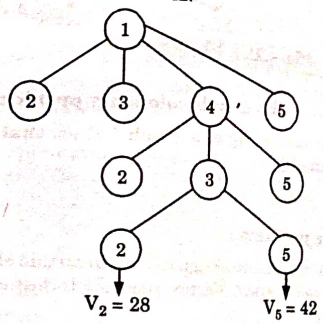


Fig. 4.20.3.

24. Here V_2 is the most promising node so next we are going to expand this node further. Now, we are left with only one node not yet traversed which is V_5.

$V_1 \xrightarrow{10} V_4 \xrightarrow{6} V_3 \xrightarrow{5} V_2 \xrightarrow{2} V_5 \xrightarrow{16} V_1$

So, total cost of traversing the graph is :

10 + 6 + 5 + 2 + 16 = 39

Que 4.21. Write short notes on graph colouring.

AKTU 2013-14, Marks 05

Answer

1. Graph colouring is a simple way of labelling graph components such as vertices, edges, and regions under some constraints.

2. In a graph, no two adjacent vertices, adjacent edges, or adjacent regions are coloured with minimum number of colours. This number is called the chromatic number and the graph is called a properly coloured graph.
3. While graph colouring, the constraints that are set on the graph are colours, order of colouring, the way of assigning colour, etc.
4. A colouring is given to a vertex or a particular region. Thus, the vertices or regions having same colours form independent sets.

Example :

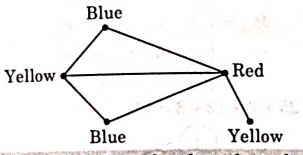


Fig. 4.21.1. Properly coloured graph.

Que 4.22. What is the graph colouring problem? What do you mean by optimal colouring of a graph ? Show that every bipartite graph is 2-colourable.

Answer

Graph colouring problem :

Graph colouring problem is to assign colours to certain elements of a graph subject to certain constraints. Vertex colouring is the most common graph colouring problem.

Vertex colouring :

1. The problem is, given m colours, find a way of colouring the vertices of a graph such that no two adjacent vertices are coloured using same colour.
2. The other graph colouring problems like Edge Colouring (No vertex is incident to two edges of same colour) and Face Colouring (Geographical Map Colouring) can be transformed into vertex colouring.

Optimal colouring :

1. In the graph colouring problem we use k colours to colour properly a given graph, but if a graph can be properly coloured with $k-1$ colours, then the k-colouring is not optimal.
2. In this way we can consider graph colouring problem as an optimization problem in which we try to find a minimum number of colour (Chromatic number G) needed to properly colour a given graph.

Proof :

1. We know that a bipartite graph is a graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V that is, U and V are independent sets.
2. If we use graph colouring method in bipartite graph then we use only two colours for the set U and V. This means that if all nodes in U is blue, and all nodes in V is green, each edge has end points of differing colours, as it required in the graph colouring problem.
3. Since the end points of every edge of bipartite graph is in different set. So only two colour is used. So every bipartite graph is 2-colourable.

Que 4.23. What is backtracking ? Discuss sum of subset problem with the help of an example. **AKTU 2017-18, Marks 10**

Answer

Backtracking : Refer Q. 4.12, Page 4–12B, Unit-4.

Sum of subset problem with example :

In the subset-sum problem we have to find a subset s' of the given set $S = (S_1, S_2, S_3, \ldots\ldots, S_n)$ where the elements of the set S are n positive integers in such a manner that $s' \in S$ and sum of the elements of subset 's" is equal to some positive integer 'X'.

Algorithm for sum-subset problem :

Subset-Sum (S, t)

1. $C \leftarrow \phi$
2. $Z \leftarrow S$
3. $K \leftarrow \phi$
4. $t_1 \leftarrow t$
5. while ($Z \neq \phi$) do
6. $K \leftarrow \max(Z)$
7. if ($K < t$) then
8. $Z \leftarrow Z - K$
9. $t_1 \leftarrow t_1 - K$
10. $C \leftarrow C \cup K$
11. else $Z \leftarrow Z - K$
12. print C // Subset Sum elements whose // Sum is equal to t_1

This procedure selects those elements of S whose sum is equal to t. Every time maximum element is found from S, if it is less than t then this element is removed from Z and also it is subtracted from t.

Example :

Given $S = <1, 2, 5, 7, 8, 10, 15, 20, 25>$ & $m = 35$

$Z \leftarrow S$, $m = 35$

$k \leftarrow \max[Z] = 25$

$\therefore$ $K < m$

i.e., $Z = Z - K$

$Z = <1, 2, 5, 7, 8, 10, 15, 20>$ & $m_1 \leftarrow m$

Subtracting K from m_1, we get
New $m_1 = m_1(\text{old}) - K = 35 - 25 = 10$
In new step,

$K \leftarrow \max[Z] = 20$
$K > m_1$
i.e., $Z = <1, 2, 5, 7, 8, 10, 15>$
In new step,
$K \leftarrow \max[Z] = 15$
$K > m_1$
i.e., $Z = <1, 2, 5, 7, 8, 10>$
In new step,
$K \leftarrow \max[Z] = 10$
$K > m_1$
i.e., $Z = <1, 2, 5, 7, 8>$
In new step,
$K \leftarrow \max[Z] = 8$
$K > m_1$
i.e., $Z = <1, 2, 5, 7> \& m_2 \leftarrow m_1$
New $m_2 = m_2(\text{old}) - K = 10 - 8 = 2$
In new step
$K \leftarrow \max[Z] = 7$
$K > m_2$
i.e., $Z = <1, 2, 5>$
In new step, $K \leftarrow \max[Z] = 5$
$K > m_2$
i.e., $Z = <1, 2>$
In new step, $K \leftarrow \max[Z] = 2$
$K > m_2$
i.e., $Z = <1>$
In new step,
$K = 1$
$K < m_2$
$\therefore$ $m_3 = 01$

Now only those numbers are needed to be selected whose sum is 01, therefore only 1 is selected from Z and rest other number found as max[Z] are subtracted from Z one by one till Z become ϕ.

Que 4.24. **Differentiate between backtracking and branch and bound approach. Write an algorithm for sum-subset problem using backtracking approach. Find all possible solution for following instance using same if $m = 30$, $S = <1, 2, 5, 7, 8, 10, 15, 20, 25>$.**

AKTU 2014-15, Marks 10

Answer

Difference between backtracking and branch and bound technique :

S. No.	Backtracking	Branch and bound
1.	It is a methodological way of trying out various sequences of decisions.	It is a systematic method for solving optimization problems.
2.	It is applied in dynamic programming technique.	It is applied where greedy and dynamic programming technique fails.
3.	It is sometimes called depth first search.	It is also known as best first search.
4.	This approach is effective for decision problem.	This approach is effective for optimization problems.
5.	This algorithm is simple and easy to understand.	This algorithm is difficult to understand.

Algorithm for sum-subset problem : Refer Q. 4.23, Page 4–33B, Unit-4.

Numerical :

Given $S = <1, 2, 5, 7, 8, 10, 15, 20, 25> \& m = 30$
$Z \leftarrow S, m = 30$
$K \leftarrow \max[Z] = 25$
$K < m$
$\therefore$ $Z = Z - K$
i.e., $Z = <1, 2, 5, 7, 8, 10, 15, 20> \& m_1 \leftarrow m$
Subtracting K from m_1, we get
New $m_1 = m_1(\text{old}) - K = 30 - 25 = 5$
In new step,

$K \leftarrow \max[Z] = 20$
$K > m_1$
i.e., $Z = <1, 2, 5, 7, 8, 10, 15>$
In new step,
$K \leftarrow \max[Z] = 15$
$K > m_1$
i.e., $Z = <1, 2, 5, 7, 8, 10>$
In new step,
$K \leftarrow \max[Z] = 10$
$K > m_1$
i.e., $Z = <1, 2, 5, 7, 8>$
In new step, $K \leftarrow \max[Z] = 8$
$K > m_1$
i.e., $Z = <1, 2, 5, 7>$

New $m_2 = m_2(\text{old}) - K = 5 - 2 = 3$
In new step, $K \leftarrow \max[Z] = 7$
$K > m_1$
$Z = <1, 2, 5>$
i.e.,
In new step, $K \leftarrow \max[Z] = 5$
$K > m_1$
$Z = <1, 2>$
i.e.,
In new step, $K \leftarrow \max[Z] = 2$
$K > m_1$
$Z = <1>$ and $m_2 \leftarrow m_1$
i.e.,
In new step, $K = 1$
$K < m_2$
$\therefore$ $Z = <\phi>$

Que 4.25. **Explain Hamiltonian circuit problem. Consider a graph $G = (V, E)$ shown in Fig. 4.25.1 and find a Hamiltonian circuit using backtracking method.**

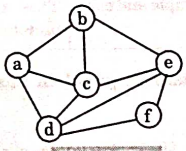


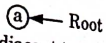
Fig. 4.25.1.

Answer

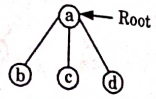
1. Given a graph $G = (V, E)$, we have to find the Hamiltonian circuit using backtracking approach.
2. We start our search from any arbitrary vertex, say '*a*'. This vertex '*a*' becomes the root of our implicit tree.
3. The first element of our partial solution is the first intermediate vertex of the Hamiltonian cycle that is to be constructed.
4. The next adjacent vertex is selected on the basis of alphabetical (or numerical) order.
5. If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex '*a*' then we say that dead end is reached.
6. In this case we backtrack one step, and again search begins by selecting another vertex and backtrack the element from the partial solution must be removed.
7. The search using backtracking is successful if a Hamiltonian cycle is obtained.

Numerical :

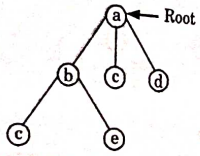
1. Firstly, we start our search with vertex '*a*', this vertex '*a*' becomes the root of our implicit tree.



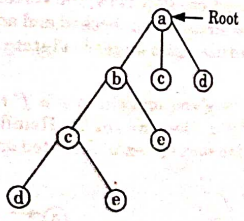
2. Next, we choose vertex '*b*' adjacent to '*a*' as it comes first in lexicographical order (*b*, *c*, *d*).



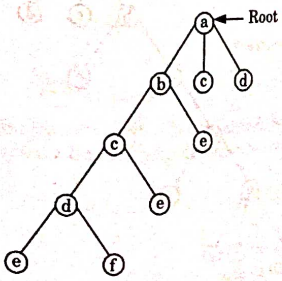
3. Next, we select '*c*' adjacent to '*b*'



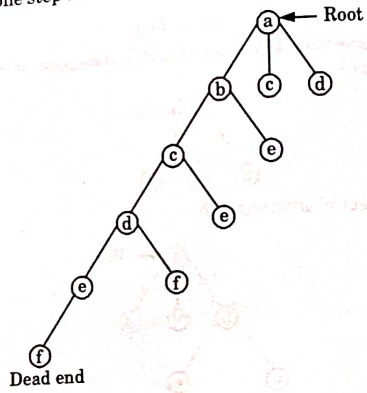
4. Next, we select '*d*' adjacent to '*c*'



5. Next, we select '*e*' adjacent to '*d*'

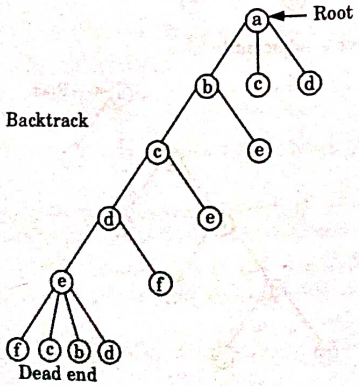


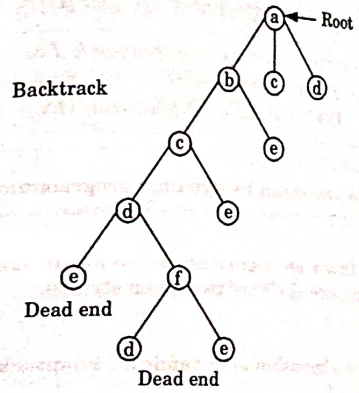
6. Next, we select vertex '*f*' adjacent to '*e*'. The vertex adjacent to '*f*' are '*d*' and '*e*' but they have already visited. Thus, we get the dead end and we backtrack one step and remove the vertex '*f*' from partial solution.



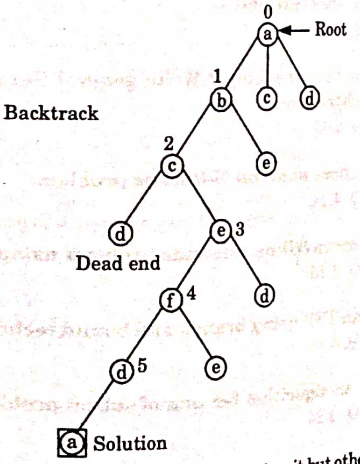
7. From backtracking, the vertex adjacent to '*e*' are '*b*', '*c*', '*d*', '*f*' from which vertex '*f*' has already been checked and '*b*', '*c*', '*d*' have already visited. So, again we backtrack one step. Now, the vertex adjacent to '*d*' are '*e*', '*f*' from which '*e*' has already been checked and adjacent of '*f*' are '*d*' and '*e*'. If '*e*' vertex visited then again we get dead state. So again we backtrack one step.

8. Now, adjacent to '*c*' is '*e*' and adjacent to '*e*' is '*f*' and adjacent to '*f*' is '*d*' and adjacent to '*d*' is '*a*'. Here, we get the Hamiltonian cycle as all the vertex other than the start vertex '*a*' is visited only once. ($a-b-c-e-f-d-a$).





Again backtrack



9. Here we have generated one Hamiltonian circuit but other Hamiltonian circuit can also be obtained by considering other vertex.

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. What do you mean by dynamic programming ?
Ans. Refer Q. 4.1.

Q. 2. Write down an algorithm to compute longest common subsequence (LCS) of two given strings.
Ans. Refer Q. 4.5.

Q. 3. Give the algorithm of dynamic 0/1 knapsack problem.
Ans. Refer Q. 4.6.

Q. 4. Describe the Warshall's and Floyd's algorithm for finding all pair shortest paths.
Ans. Refer Q. 4.9.

Q. 5. What is backtracking ? Write general iterative algorithm for backtracking.
Ans. Refer Q. 4.12.

Q. 6. Write short notes on N-Queens problem.
Ans. Refer Q. 4.14.

Q. 7. Explain travelling salesman problem using backtracking.
Ans. Refer Q. 4.16.

Q. 8. Explain TSP using branch and bound technique.
Ans. Refer Q. 4.18.

Q. 9. Write an algorithm for sum of subset problem.
Ans. Refer Q. 4.24.

5 UNIT

Selected Topics

PART-1

Algebraic Computation, Fast Fourier Transform, String Matching.

CONCEPT OUTLINE : PART-1

- **Fast fourier transform :** It computes DFT of an n-length vector in $O(n \log n)$ time. In the FFT algorithm, we apply the divide and conquer approach to polynomial evaluation.
  $A(X) = a_0 + a_1X + a_2X^2 + ... + a_{n-1}X^{n-1}$
- **String matching :** It is an approach for finding one or more, all the occurrence of a pattern in a text.
- **Various methods of string matching :**
  1. Naive string matching algorithm
  2. Rabin-Karp algorithm
  3. Knuth-Morris-Pratt algorithm

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.1. What is FFT (Fast Fourier Transformation) ? How the recursive FFT procedure works ? Explain.

AKTU 2013-14, Marks 10

Answer

1. The Fast Fourier Transform (FFT) is a algorithm that computes a Discrete Fourier Transform (DFT) of n-length vector in $O(n \log n)$ time.
2. In the FFT algorithm, we apply the divide and conquer approach to polynomial evaluation by observing that if n is even, we can divide a degree $(n-1)$ polynomial.

$$A(x) = a_0 + a_1x + a_2x^2 + ... + a_{n-1}x^{n-1}$$

into two degree $\left(\frac{n}{2}-1\right)$ polynomials.

$$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + ... + a_{n-2}x^{n/2-1}$$

$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + ... + a_{n-1}x^{n/2-1}$$

Where $A^{[0]}$ contains all the even index coefficients of A and $A^{[1]}$ contains all the odd index coefficients and we can combine these two polynomials into A, using the equation,

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \quad ...(5.1.1)$$

So that the problem of evaluating $A(x)$ at $\omega^k{}_n$ where $k = 0, 1, 2, ..., n-1$ reduces to,

i. Evaluating the degree $\left(\frac{n}{2}-1\right)$ polynomial $A^{[0]}(x)$ and $A^{[1]}(x)$ at the point $(\omega^k{}_n)^2$ *i.e.*,

$$(\omega^0{}_n)^2, (\omega^1{}_n)^2, ..., (\omega^{n-1}_n)^2.$$

because we know that if $\omega^k{}_n$ is a complex n^{th} root of unity then $(\omega^k{}_n)^2$ is a complex $\frac{n^{th}}{2}$ root of unity. Thus, we can evaluate each $A^{[0]}(x)$ and $A^{[1]}(x)$ at $(\omega^k{}_n)^2$ values.

ii. Combining the results according to the equation (5.1.1). This observation is the basis for the following procedure which computes the DFT of an n-element vector $a = (a_0, a_1, ..., a_{n-1})$ where for sake of simplicity, we assume that n is a power of 2.

FFT (a, w) :

1. $n \leftarrow$ length $[a]$ — n is a power of 2.
2. if $n = 1$
3. then return a
4. $\omega_n \leftarrow e^{2\pi i/n}$
5. $x \leftarrow \omega^0$ — x will store powers of ω initially $x = 1$.
6. $a^{[0]} \leftarrow (a_0, a_2, ...a_{n-2})$
7. $a^{[1]} \leftarrow (a_1, a_3, ... a_{n-1})$
8. $y^{[0]} \leftarrow FFT(a^{[0]}, \omega^2)$ — Recursive calls with ω^2 as $(n/2)^{th}$ root of unity.
9. $y^{[1]} \leftarrow FFT(a^{[0]}, \omega^2)$
10. for $k \leftarrow 0$ to $(n/2) - 1$
11. do $y_k \leftarrow y_k^{[0]} + x\, y_k^{[1]}$
12. $y_{k+(n/2)} \leftarrow y_k^{[0]} - xy_k^{[1]}$
13. $x \leftarrow x\omega_n$
14. return y

Line 2-3 represents the basis of recursion, the DFT of one element is the element itself. Since in this case

$$y_0 = a_0\, \omega_1{}^0 = a_0\, 1 = a_0$$

Line 6-7 defines the recursive coefficient vectors for the polynomials $A^{[0]}$ and $A^{[1]}$. $\omega = \omega^k{}_n$

Line 8-9 perform the recursive $DFT_{n/2}$ computations setting for

$$k = 0, 1, 2, ..., \frac{n}{2} - 1 \text{ i.e.,}$$

$$y_k^{[0]} = A^{[0]}(\omega_n^{2k})$$

$$y_k^{[1]} = A^{[1]}(\omega_n^{2k})$$

Lines 11-12 combine the results of the recursive $DFT_{n/2}$ calculations.

For $y_0, y_2, \cdots y_{(n/2)-1}$, line 11 yields.

$$y_k = y_k^{[0]} + \omega_n^k y_k^{[1]}$$
$$= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) = A(\omega_n^k)$$

using equation (5.1.1)

For $y_{n/2}, y_{(n/2)+1} \cdots y_{n-1}$, line 12 yields.

$$y_{k+(n/2)} = y_k^{[0]} - \omega_n^k y_k^{[1]} = y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]}$$

$[\because \omega_n^{k+(n/2)} = -\omega_n^k]$

$$= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k})$$
$$= A^{[0]}(\omega_n^{2k}\ \omega_n^n) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}\ \omega_n^n) \quad [\because \omega_n^n = 1]$$
$$= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n})$$
$$= A(\omega_n^{k+(n/2)})$$

using equation (5.1.1)

each $k = 0, 1, 2, \ldots, (n/2) - 1$.

Thus, the vector y returned by the FFT algorithm will store the values of $A(x)$ at each of the roots of unity.

Que 5.2. What is string matching ? Discuss string matching problem. Also define string, substring and proper substring.

Answer

String matching is a process of finding one or more occurrences of a pattern in a text.

String matching problem :

Given a text array $T[1 .. n]$ of n character and a pattern array $P[1 .. m]$ of m characters.

The problem is to find an integer s, called valid shift where $0 \le s < n - m$ and $T[s + 1 .... s + m] = P[1 ... m]$.

We further assume that the elements of P and T are characters drawn from a finite alphabet Σ such as $\{0, 1\}$ or $\{A, B, \ldots Z, a, b, \ldots, z\}$.

String : A string is traditionally a sequence of character, either as a literal constant or as some kind of variable.

Substring : Given a string $T[1 .. n]$, the substring is defined as $T[i .. j]$ for some $0 \le i \le j \le n - 1$, that is, the string formed by the characters in T from index j, inclusive. This means that a string is a substring of itself (simply take $i = 0$ and $j = m$).

Proper substring : The proper substring of string $T[1 .. n]$ is $T[i .. j]$ for some $0 \le i \le j \le n - 1$, that is, we must have either $i > 0$ or $j < m - 1$.

Using these definition, we can say given any string $T[1 .. n]$, the substring are

$$T[i .. j] = T[i]\ T[i + 1]\ T[i + 2] \ldots T[j]$$

for some $0 \le i \le j \le n - 1$.

And proper substrings are

$$T[i .. j] = T[i]\ T[i + 1]\ T[i + 2] \ldots T[j]$$

for some $0 \le i \le j \le n - 1$.

Note that if $i > j$, then $T[i .. j]$ is equal to the empty string or null, which has length zero. Using these notations, we can define of a given string $T[1 .. n]$ as $T[0 .. i]$ for some $0 \le i \le n - 1$ and suffix of a given string $T[1 - n]$ as $T[i .. n - 1]$ for some $0 \le i \le n - 1$.

Que 5.3. What are the different types of string matching? Explain one of them.

Answer

Basic types of string matching algorithms are :

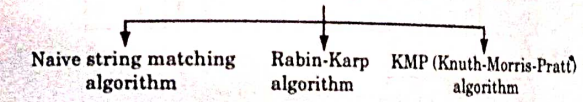


Fig. 5.3.1.

Naive string matching :

The Naive approach simply test all the possible placement of pattern $P[1 .. m]$ relative to text $T[1 .. n]$. Specifically, we try shifts $s = [0, 1, ..., n - m]$, successively and for each shift, s, compare $T[s + 1 .. s + m]$ to $P[1 .. m]$.

Naive string matcher (T, P)

1. $n \leftarrow$ length $[T]$
2. $m \leftarrow$ length $[P]$
3. for $s \leftarrow 0$ to $n - m$
4. do if $P[1 .. m] = T[s + 1 .. s + m]$
5. then print "pattern occurs with shift" s.

The Naive string matching procedure can be interpreted graphically as a sliding a pattern $P[1 .. m]$ over the text $T[1 .. m]$ and noting for which shift all of the characters in the pattern match the corresponding characters in the text.

To analyze the time of Naive matching, the given algorithm is implemented as follows, note that in this implementation, we use notation $P[1 .. j]$ to denote the substring of P from index i to index j. That is, $P[1 ... j] = P[i]\ P[i + 1] \ldots P[j]$.

Naive string matcher (T, P)

1. $n \leftarrow$ length $[T]$

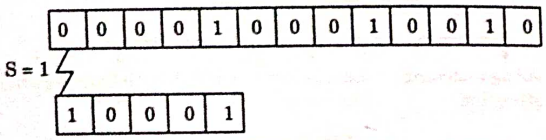
2. · $m \leftarrow$ length [P]
3. for $s \leftarrow 0$ to n–m do
4. $j \leftarrow 1$
5. while $j \leq m$ and $T[s+j] = P[j]$ do
6. $j \leftarrow j+1$
7. if $j > m$ then
8. return valid shift s
9. return no valid shift exist // *i.e.*, there is no substring of T matching P.

Que 5.4. **Show the comparisons that Naive string matcher makes for the pattern P = {10001} in the text T = {0000100010010}**

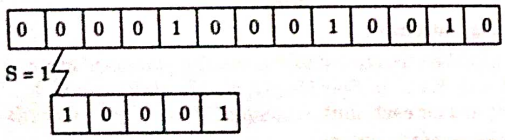
Answer

Given, $P = 10001$
$T = 0000100010010$

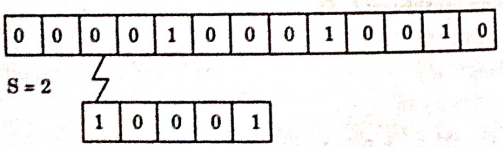
i.



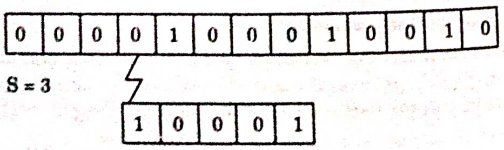
ii



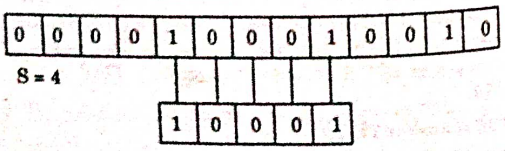
iii.



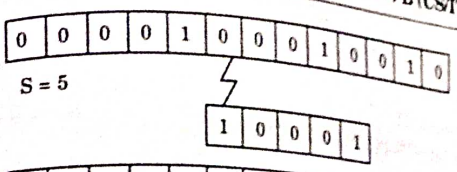
iv.



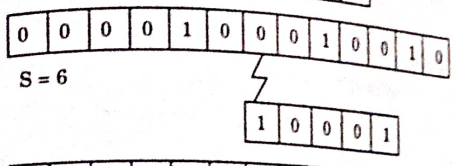
v.



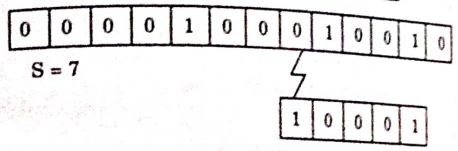
vi.



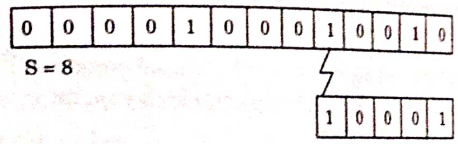
vii.



viii.



ix.



Que 5.5. **Write down Knuth-Morris-Pratt algorithm for string matching. Find the prefix function of the string *abababaca*.**

Answer

Knuth-Morris-Pratt algorithm for string matching :

COMPUTE-PREFIX-FUNCTION (P)

1. $m \leftarrow$ length [P]
2. $\pi[1] \leftarrow 0$
3. $k \leftarrow 0$
4. for $q \leftarrow 2$ to m
5. do while $k > 0$ and $P[k+1] \neq P[q]$
6. do $k \leftarrow \pi[k]$
7. if $P[k+1] = P[q]$
8. then $k \leftarrow k+1$
9. $\pi[q] \leftarrow k$
10. return π

KMP-MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION to compute π.

KMP-MATCHER (T, p)

1. $n \leftarrow$ length [T]
2. $m \leftarrow$ length [P]
3. $\pi \leftarrow$ COMPUTE-PREFIX-FUNCTION (P)
4. $q \leftarrow 0$
5. for $i \leftarrow 1$ to n
6. do while $q > 0$ and $p\ [q + 1] \neq T[i]$
7. do $q \leftarrow \pi\ [q]$
8. if $p\ [q + 1] = T[i]$
9. then $q \leftarrow q + 1$
10. if $q = m$
11. then print "pattern occurs with shift" $i - m$
12. $q \leftarrow \pi\ [q]$

Prefix function of the string *abababaabca* :

$m \leftarrow$ length [P]

$\therefore$ $m = 10$

Initially, $P[1] = 0, k = 0$

for $q \leftarrow 2$ to 10

for $q = 2$, $k \not> 0$

& $P[0 + 1] = P[2]$

$\therefore$ $\pi[2] = 0$

for $q = 3$, $k \not> 0$

& $P[0 + 1] = P[3]$

$\therefore$ $k \leftarrow k + 1 = 1$

& $\pi[3] \leftarrow 1$

for $q = 4, k > 0$

& $P[1 + 1] = P[4]$

$\therefore$ $k \leftarrow 1 + 1 = 2$

& $\pi[4] \leftarrow 2$

for $q = 5, k > 0$

& $P[2 + 1] = P[5]$

$\therefore$ $k \leftarrow 2 + 1 = 3$

& $\pi[5] \leftarrow 3$

for $q = 6, k > 0$

& $P[3 + 1] = P[6]$

$\therefore$ $k \leftarrow 3 + 1 = 4$

& $\pi[6] \leftarrow 4$

for $q = 7, k > 0$

& $P[4 + 1] = P[7]$

$\therefore$ $k \leftarrow 4 + 1 = 5$

& $\pi[7] \leftarrow 5$

for $q = 8, k > 0$

& $P[5 + 1] = P[8]$

$\therefore$ $k \leftarrow 5 + 1 = 6$

& $\pi[8] \leftarrow 6$

for $q = 9, k > 0$

& $P[6 + 1] = P[9]$

$\therefore$ $k \leftarrow \pi[k] = 6$

& $\pi[9] \leftarrow 6$

for $q = 10, k > 0$

& $P[6 + 1] = P[10]$

$\therefore$ $k \leftarrow 6 + 1 = 7$

& $\pi[10] \leftarrow 7$

String	*a*	*b*	*a*	*b*	*a*	*b*	*a*	*b*	*c*	*a*
$P[i]$	1	2	3	4	5	6	7	8	9	10
$\pi[i]$	0	0	1	2	3	4	5	6	6	7

Que 5.6. **Compute the prefix function π for the pattern $P =$ *abacab* using KNUTH-MORRIS-PRATT algorithm. Also explain Naive string matching algorithm.** **AKTU 2017-18, Marks 10**

Answer

Prefix function of the string *abacab* :

$m \leftarrow$ length [P]

$\therefore$ $m = 6$

Initially, $\pi[1] = 0, k = 0$

for $q \leftarrow 2$ to 6

for $q = 2$, $k \not> 0$

& $P[0 + 1] \neq P[2]$

$\therefore$ $\pi[2] = 0$

for $q = 3$, $k \not> 0$

& $P[0 + 1] = P[3]$

$\therefore$ $k = k + 1 = 1$

& $\pi[3] = 1$

for $q = 4, k > 0$

& $P[1 + 1] \neq P[4]$

$\therefore$ $k \leftarrow \pi[1] = 0$

& $P[1] \neq P[4]$

$\pi[4] = 0$

for $q = 5, k > 0$

& $P[0 + 1] = P[5]$

$\therefore$ $k \leftarrow 0 + 1 = 1$

& $\pi[5] = 1$

for $q = 6, k > 0$

& $P[1 + 1] = P[6]$

$\therefore$ $k \leftarrow 1 + 1 = 2$

& $\pi[6] = 2$

String	a	b	a	c	a	b
P[i]	1	2	3	4	5	6
π[i]	0	0	1	0	1	2

Naive string matching algorithm : Refer Q. 5.3, Page 5–5B, Unit-5.

Que 5.7. **What is string matching algorithm ? Write Knuth-Morris-Pratt algorithm and also calculate the prefix function for the pattern $P = ababaaca$.** **AKTU 2014-15, Marks 10**

Answer

String matching algorithm : Refer Q. 5.3, Page 5–5B, Unit-5.

Knuth-Morris-Pratt algorithm : Refer Q. 5.5, Page 5–7B, Unit-5.

Numerical :

Here pattern $P = ababaaca$

Length $|P| = 8$ so $m = 8$

Initially, $\pi(1) = 0$ and $k = 0$

For $q \leftarrow 2$ to 8

For $q = 2$ and $K \neq 0$

$P[0 + 1] \neq P[2]$.

So $\pi[2] = 0$

For $q = 3$

$P[0 + 1] = P[3]$

then $K = K + 1 = 1$

$\pi[3] = 1$

For $q = 4\ K > 0$

$P[1 + 1] = P[4]$

then $K = K + 1 = 2$

$\pi[4] = 2$

For $q = 5\ K > 0$

$P[2 + 1] = P[5]$

then $K = K + 1 = 3$

$\pi[5] = 3$

For $q = 6\ K > 0$

$P[3 + 1] \neq P[6]$

$\pi[6] = 3$

For $q = 7\ K > 0$

$P[3 + 1] \neq P[7]$

$\pi[7] = 3$

$q = 8\ K > 0$

For $P[3 + 1] \neq P[8]$

$\pi[8] = 3$

Similarly, we compute other prefix functions.

String	a	b	a	b	a	a	c	a
P[i]	1	2	3	4	5	6	7	8
π[i]	0	0	1	2	3	0	0	1

Que 5.8. **Write algorithm for Rabin-Karp method. Give a suitable example to explain it.**

OR

What is string matching algorithm ? Explain Rabin-Karp method with examples. **AKTU 2015-16, Marks 10**

Answer

String matching algorithm : Refer Q. 5.3, Page 5–5B, Unit-1.

The Rabin-Karp algorithm :

The Rabin-Karp algorithm states that if two strings are equal, their hash values are also equal. This also uses elementary number-theoretic notions such as the equivalence of two numbers module a third.

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. $p \leftarrow 0$
5. $t_0 \leftarrow 0$
6. for $i \leftarrow 1$ to m
7. do $p \leftarrow (dp + p\ [i]) \bmod q$
8. $t_0 \leftarrow (dt_0 + T[i]) \bmod q$
9. for $s \leftarrow 0$ to $n–m$
10. do if $p = t_s$
11. then if $p\ [1....m] = T[s + 1.....s + m]$
12. The "pattern occurs with shift" s
13. if $s < n - m$
14. then $t_{s+1} \leftarrow (d(t_0 - T\ [s + 1]\ h) + T[s + m + 1])$ mod q.

Example of Rabin-Karp method : Working modulo $q = 11$, how many spurious hits does the Rabin-Karp matcher encounter in the text $T = 3141592653589793$ when looking for the pattern $p = 26$

Given, $p = 26$ and $q = 11$

Now we divide 26 by 11 *i.e.*,
Remainder is 4 and $m = 2$.
We know m denotes the length of p.

T	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

Now we divide 31 by 11, and get remainder is 9.

Similarly, 14 by 11 and get remainder is 3.

So, continue this step till last *i.e.*, 93 is divided by 11 and get remainder is 5.
After that we will store all remainder in a table.

9	3	8	4	4	4	4	10	9	2	3	1	9	2	5

Now we find valid matching.

3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

Valid matching

9	3	8	4	4	4	4	10	9	2	3	1	9	2	5

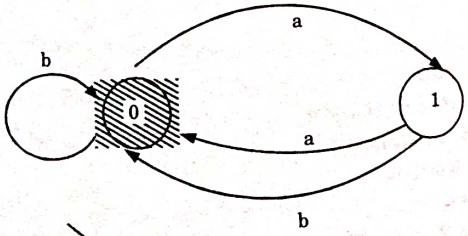
Spurious hit

The number of spurious hits is 3.

Que 5.9. Discuss string matching with finite automata.

Answer

The string matching automaton is very efficient *i.e.*, it examines each character in the text exactly once and reports all the valid shifts in $O(n)$ time.



State	a	b
0	1	0
1	0	0

Given pattern : a^{2k+1}
Input string = abaaa
Start state : 0
Terminate state : 1

Fig. 5.9.1. An automaton.

The basic idea is to build a automaton in which,

1. Each character in the pattern has a state.
2. Each match sends the automaton into a new state.
3. If all the characters in the pattern have been matched, the automaton enters the accepting state.
4. Otherwise, the automaton will return to a suitable state according to the current state and the input character such that this returned state reflects the maximum advantage we can take from the previous matching.
5. The matching takes $O(n)$ time since each character is examined once.

The construction of the string matching automaton is based on the given pattern. The time of this construction may be $O(m^3|S|)$.

Finite automata :

A finite automaton M is a 5-tuple $(Q, q_o, A, \Sigma, \delta)$ where,

1. Q is a finite set of states
2. $q_o \in Q$ is start state
3. $A \sqsubseteq Q$ is a distinguished set of accepting states
4. Σ is a finite input alphabet
5. δ is a function from $Q \times \Sigma$ into Q called transition function of M

FINITE-AUTOMATON-MATCHER (T, δ, m)

1. $n \leftarrow$ length $[T]$
2. $q \leftarrow 0$
3. for $i \leftarrow 1$ to n
4. do $q \leftarrow \delta(q, T[i])$
5. if $q = m$
6. then $s \leftarrow i - m$
7. print "Pattern occurs with shift" s.

COMPUTE-TRANSITION-FUNCTION (P, Σ)

1. $m \leftarrow$ length $[P]$
3. do for each character $a \in \Sigma^*$
4. do $k \leftarrow \min(m + 1, q + 2)$
5. repeat $k \leftarrow k - 1$
6. until
7. $\delta(q, \alpha) \leftarrow k$
8. return δ

PART-2

Theory of NP-Completeness, Approximation Algorithms, Randomized Algorithm.

CONCEPT OUTLINE : PART-2

- **NP-completeness :** A language $L \subseteq \{0, 1\}^*$ is NP-complete if it satisfies two properties :
  1. $L \in$ NP and
  2. For every $L' \in$ NP, $L' \leq L$.
- **Satisfiability (SAT) :** A Boolean formula is satisfiable, if there is a way to assign truth values to the variables, such that the final results is 1.
- **Approximation algorithm :** An approximate algorithm is a way of dealing with NP-completeness for optimization problem. This algorithm is also known as Heuristic algorithm.
- **Randomized algorithm :** It is the problem which determines the primality to access a source of independent, unbiased bits, to use random bits to complete its computation.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.10. **Discuss the problem classes P, NP and NP-complete.**

AKTU 2013-14, Marks 10

Answer

P : Class P are the problems which can be solved in polynomial time, which take time like $O(n)$, $O(n^2)$, $O(n^3)$.

Example : Finding maximum element in an array or to check whether a string is palindrome or not. So, there are many problems which can be solved in polynomial time.

NP : Class NP are the problem which cannot be solved in polynomial time like TSP(travelling salesman problem).

Example : Subset sum problem is best example of NP in which given a set of numbers, does there exist a subset whose sum is zero, but NP problems are checkable in polynomial time means that given a solution of a problem, we can check that whether the solution is correct or not in polynomial time.

NP-complete : The group of problems which are both in NP and NP-hard are known as NP-complete problem.

Now suppose we have a NP-complete problem R and it is reducible to Q then Q is at least as hard as R and since R is an NP-hard problem, therefore Q will also be at least NP-hard, it may be NP-complete also.

Que 5.11. **Discuss the problem classes P, NP and NP-complete with class relationship.**

AKTU 2017-18, Marks 10

Answer

1. The notion of NP-hardness plays an important role in the relationship between the complexity classes P and NP.
2. It is also often used to define the complexity class NP-complete which is the intersection of NP and NP-hard.
3. Consequently class NP-hard can be understood as the class of problems that are NP-complete or harder.

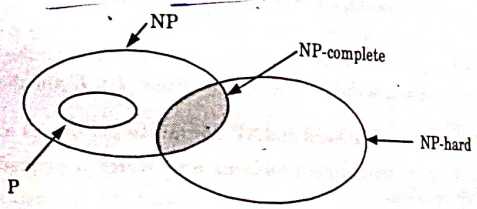


Fig. 5.11.1. Relationship among P, NP, NP-complete and NP-hard problems.

4. There are no polynomial time algorithms for NP-hard problems.
5. A problem being in NP means that the problem is "easy" (in a certain specific sense), whereas a problem being NP-hard means that the problem is "difficult" (in another specific sense).
6. A problem being in NP and a problem being NP-hard are not mutually exclusive. When a problem is both in NP and NP-hard, we say that the problem is NP-complete.
7. All problems in NP can be solved deterministically in time $O(2n)$, or in time $2^{O(n)}$ for that matter.
8. An example of an NP-hard problem is the decision problem subset-sum. Given a set of integers, does any non-empty subset of them add up to zero ? *i.e.*, a yes or no question, and happens to be NP-complete.
9. There are, also decision problems that are NP-hard but not NP-complete.
10. For example, in the halting problem "given a program and its input, will it run forever" *i.e.*, yes or no question, so this is a decision problem. It is case to prove that the halting problem is NP-hard but not NP-complete.

Que 5.12. **What is NP-completeness ?**

Answer

A language $L \subseteq \{0,1\}^*$ is NP-complete if it satisfies the following two properties :

i. $L \in NP$; and

ii. For every $L' \leq_p L$

NP-hard : If a language L satisfies property (ii), but not necessarily property (i), we say that L is NP-hard.

NP-complete : We use the notation $L \in NPC$ to denote that L is NP-complete.

Theorem : If any NP-complete problem is polynomial time solvable, then P = NP. If any problem in NP is not polynomial time solvable, then all NP complete problems are not polynomial time solvable.

Proof : Suppose that $L \in P$ and also that $L \in NPC$. For any $L' \in NP$, we have $L' \in_p L$ by property (ii) of the definition of NP-completeness. We know if $L' \leq_p L$ then $L \in P$ implies $L' \in P$, which proves the first statement.

To prove the second statement, suppose that there exists and $L \in NP$ such that $L \notin P$. Let $L' \in NPC$ be any NP-complete language, and for the purpose of contradiction, assume that $L' \in P$. But then we have $L \leq_p L'$ and thus $L \in P$.

Que 5.13. Explain NP-hard and NP-complete problems and also define the polynomial time problems and write a procedure to solve NP-problems.

OR

Write short note on NP-hard and NP-complete problems.

Answer

NP-hard problem :

1. We say that a decision problem P_i is NP-hard if every problem in NP is polynomial time reducible to P_i.
2. In symbols,

   P_i is NP-hard if, for every $P_j \in NP$, $P_j \xrightarrow{Poly} P_i$.
3. This does not require P_i to be in NP.
4. Highly informally, it means that P_i is 'as hard as' all the problem in NP.
5. If P_i can be solved in polynomial time, then all problems in NP.
6. Existence of a polynomial time algorithm for an NP-hard problem implies the existence of polynomial solution for every problem in NP.

NP-complete problem :

1. There are many problems for which no polynomial time algorithms is known.

2. Some of these problems are travelling salesman problem, optimal graph colouring, the Knapsack problem, Hamiltonian cycles, integer programming, finding the longest simple path in a graph, and satisfying a Boolean formula.
3. These problems belongs to an interesting class of problems called the "NP-complete" problems, whose status is unknown.
4. The NP-complete problems are traceable *i.e.*, require a super polynomial time.

Polynomial time problem :

An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is $O(n^k)$ for some non-negative integer k where n is the complexity of input.

Polynomial time verifiable algorithm : A polynomial time algorithm A is said to be polynomial time verifiable if it has following properties :

1. The input to A consists of an instance I of X (X is a decision problem) and a string S such that the length of S is bounded by some polynomial in the size of I.
2. The output of A is either yes or no.
3. If I is a negative instance of X, then the output of A is "no" regardless of the value of S.
4. If I is a positive instance of X, then there is at least one choice of S for which A output "yes".

Procedure to solve NP-problems :

1. The class NP is the set of all decision problems that have instances that are solvable in polynomial time using a non-deterministic turing machine.
2. In a non-deterministic turing machine, in contrast to a deterministic turing machine, for each state, several rules with different actions can be applied.
3. Non-deterministic turing machine branches into many copies that are represented by a computational tree in which there are different computational paths.
4. The class NP corresponds to a non-deterministic turing machine that guesses the computational path that represents the solution.
5. By doing so, it guesses the instances of the decision problem.
6. In the second step, a deterministic turing machine verifies whether the guessed instance leads to a "yes" answer.
7. It is easy to verify whether a solution is valid or not. This statement does not mean that finding a solution is easy.

Que 5.14. Differentiate NP-complete with NP-hard.

AKTU 2016-17, Marks 10

Answer

S. No.	NP-complete	NP-hard
1.	An NP-complete problems is one to which every other polynomial-time non-deterministic algorithm can be reduced in polynomial time.	NP-hard problems is one to which an NP-complete problem is Turing-reducible.
2.	NP-complete problems does not corresponds to an NP-hard problem.	NP-hard problems correspond to an NP-complete problem.
3.	NP-complete problems are exclusively decision problem.	NP-hard problems need not to be decision problem.
4.	NP-complete problems have to be in NP-hard and also in NP.	NP-hard problems do not have to be in NP.
5.	**For example :** 3-SAT vertex cover problem is NP-complete.	**For example :** Halting problem is NP-hard.

Que 5.15. Discuss NP-complete problem and also explain minimum vertex cover problem in context to NP-completeness.

Answer

NP-complete problem : Refer Q. 5.13, Page 5–16B, Unit-5.

Minimum vertex cover problem :

1. A vertex cover of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex of the set.
2. The problem of finding a minimum vertex cover is a classical optimization problem in computer science and is a typical example of an NP-hard optimization problem that has an approximation algorithm.
3. Its decision version, the vertex cover problem, was one of Karp's 21 NP-complete problems and is therefore a classical NP-complete problem in computational complexity theory.
4. Furthermore, the vertex cover problem is fixed-parameter tractable and a central problem in parameterized complexity theory.
5. The minimum vertex cover problem can be formulated as a half-integral linear program whose dual linear program is the maximum matching problem.

6. Formally, a vertex cover V' of undirected graph $G = (V, E)$ is a subset of V such that $uv \in V' \vee v \in V'$ *i.e.*, it is a set of vertices V' where every edge has at least one endpoint in the vertex cover V'. Such a set is said to cover the edges of G.

   Example : Fig. 5.15.1 shows examples of vertex covers, with some vertex cover V' marked in dark.

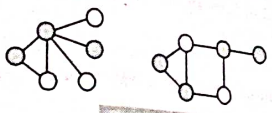


Fig. 5.15.1.

7. A minimum vertex cover is a vertex cover of smallest possible size.
8. The vertex cover number τ is the size of a minimum cover, *i.e.*, $\tau = |V'|$. The Fig. 5.15.2 shows examples of minimum vertex covers in the previous graphs.

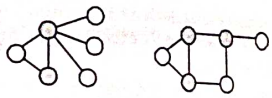


Fig. 5.15.2.

Que 5.16. Discuss different types of NP-complete problem.

Answer

Types of NP-complete problems :

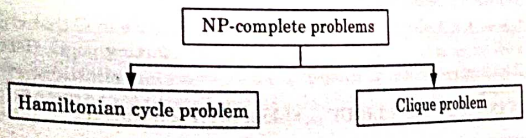


Fig. 5.16.1.

Hamiltonian cycle problem :

1. A Hamiltonian cycle of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V.
2. A graph that contains a Hamiltonian cycle is said to be Hamiltonian, otherwise it is said to be non-Hamiltonian.

The clique problem :

1. A clique 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.
2. The size of a clique is the number of vertices it contains.

CLIQUE = {<G, K> : G is a graph with a clique of size K}

3. The clique problem is the optimization problem of finding a clique of maximum size in a graph.
4. As a decision problem, we ask simply whether a clique of a given size k exists in the graph.

Que 5.17. **Show that Hamiltonian circuit is NP-complete.**

Answer

Theorem : Hamiltonian circuit (HC) is NP-complete.

Proof :

1. Let us define a non-deterministic algorithm A that takes, as input, a graph G encoded as an adjacency list in binary notation, with the vertices numbered 1 to N.
2. We define A to first iteratively call the choose method to determine a sequence S of $N + 1$ numbers from 1 to N.
3. Then, we have A to check that each number from 1 to N appears exactly once in S (for example, by sorting S), except for the first and last numbers in S, which should be the same.
4. Then, we verify that a sequence S defines a cycle of vertices and edges in G.
5. A binary encoding of the sequence S is clearly of size at most n, where n is the size of the input. Moreover, both of the checks made on the sequence S can be done in polynomial time in n.
6. Observe that if there is a cycle in G that visits each vertex of G exactly once, returning to its starting vertex, then there is a sequence S for which A will output "yes."
7. Likewise, if A outputs "yes," then it has found a cycle in G that visits each vertex of G exactly once, returning to its starting point. Hence, Hamiltonian circuit is NP-complete.

Que 5.18. **Show that CLIQUE problem is NP-complete.**

Answer

Problem : The CLIQUE problem is defined as $\{< G, k >\mid G$ is a graph with a k-clique$\}$. Show that CLIQUE is NP-complete.

Proof :

1. First, to show that CLIQUE is in NP. Given an instance of $< G, k >$ and a k-clique we can easily verify in $O(n^2)$ time that we do, in fact, have a k-clique.
2. Now, we want to show that 3-SAT is CLIQUE. Let F be a boolean formula in CNF.
3. For each literal in F we will make a vertex in the graph *i.e.*,

   $(x_1 + \bar{x}_2 + x_3)\ (\bar{x}_1 + x_2 + \bar{x}_3)$ has 6 vertices.

   Let k be the number of clauses in F.
4. We will connect each vertex to all of the other vertices that are logically compatible except for the ones that are in the same clause.
5. Now, if we have a satisfiable assignment we will have a k-clique because the satisfying vertices will all be connected to one another.
6. Thus, we can use CLIQUE to solve 3-SAT so CLIQUE is NP-complete.

Que 5.19. **Define different complexity classes in detail with suitable example. Show that TSP problem is NP-complete.**

AKTU 2014-15, Marks 10

Answer

Different complexity classes :

There are some complexity classes involving randomized algorithms :

1. **Randomized polynomial time (RP) :** The class RP consists of all languages L that have a randomized algorithm A running in worst case polynomial time such that for any input x in Σ^*

$$x \in L \Rightarrow P[A(x) \text{ accepts}] \geq \frac{1}{2}$$

$$x \notin L \Rightarrow P[A(x) \text{ accepts}] = 0$$

   Independent repetitions of the algorithms can be used to reduce the probability of error to exponentially small.
2. **Zero-error probabilistic polynomial time (ZPP) :** The class ZPP is the class of languages which have Las Vegas algorithms running in expected polynomial time.

$$\text{ZPP} = \text{RP} \cap \text{co-RP}$$

   where a language L is in co-X where X is a complexity class if and only if its complement $\Sigma^* - L$ is in X.
3. **Probabilistic polynomial time (PP) :** The class PP consists of all languages L that have a randomized algorithm A running in worst case polynomial time such that for any input x in Σ^*.

$$x \in L \Rightarrow P[A(x) \text{ accepts}] \geq \frac{1}{2}$$

$$x \notin L \Rightarrow P[A(x) \text{ accepts}] < \frac{1}{2}$$

   To reduce the error probability, we cannot repeat the algorithm several times on the same input and produce the output which occurs in the majority of those trials.
4. **Bounded-error probabilistic polynomial time (BPP) :** The class BPP consists of all languages that have a randomized algorithm A running in worst case polynomial time such that for any input x in Σ^*.

$$x \in L \Rightarrow P[A(x) \text{ accepts}] \geq \frac{3}{4}$$

$$x \notin L \Rightarrow P[A(x) \text{ accepts}] \leq \frac{1}{4}$$

For this class of algorithms, the error probability can be reduced to $1/2n$ with only a polynomial number of iterations.

For a given a graph $G = (V, E)$ and a number k, does there exist a tour C on G such that the sum of the edge weights for edges in C is less than or equal to k.

Part 1 : TSP is in NP.

Proof :

1. Let a hint S be a sequence of vertices $V = v_1, \ldots, v_n$.
2. We then check two things :
   a. First we check that every edge traversed by adjacent vertices is an edge in G (usually not required, since G is traditionally a complete graph), such that the sum of these edge weights is less than or equal to k.
   b. Secondly we check that every vertex in G is in V, which assures that every node has been traversed.
3. We accept S if and only if S satisfies these two questions, otherwise reject.
4. Both of these checks are clearly polynomial, thus our algorithm forms a verifier with hint S, and TSP is consequently in NP.

Part 2 : TSP is NP-Hard.

Proof :

1. To show that TSP is NP-Hard, we must show that every problem y in NP reduces to TSP in polynomial time.
2. To do this, consider the decision version of Hamiltonian Cycle (HC).
3. Take $G = (V, E)$, set all edge weights equal to 1, and let $k = |V| = n$, that is, k equals the number of nodes in G.
4. Any edge not originally in G then receives a weight of 2 (traditionally TSP is on a complete graph, so we need to add in these extra edges).
5. Then pass this modified graph into TSP, asking if there exists a tour on G with cost at most k. If the answer to TSP is YES, then HC is YES. Likewise if TSP is NO, then HC is NO.

First direction : HC has a YES answer => TSP has a YES answer.

Proof :

1. If HC has a YES answer, then there exists a simple cycle C that visits every node exactly once, thus C has n edges.
2. Since every edge has weight 1 in the corresponding TSP instance for the edges that are in the HC graph, there is a Tour of weight n. Since $k = n$, and given that there is a tour of weight n, it follows that TSP has a YES answer.

Second direction : HC has a NO answer => TSP has a NO answer.

Proof :

1. If HC has a NO answer, then there does not exist a simple cycle C in G that visits every vertex exactly once. Now suppose TSP has a YES answer.
2. Then there is a tour that visits every vertex once with weight at most k.
3. Since the tour requires every node be traversed, there are n edges, and since $k = n$, every edge traversed must have weight 1, implying that these edges are in the HC graph. Then take this tour and traverse the same edges in the HC instance. This forms a Hamiltonian Cycle, a contradiction.

This concludes Part 2. Since we have shown that TSP is both in NP and NP-Hard, we have that TSP is NP-Complete.

Que 5.20. Prove that three colouring problem is NP-complete.

AKTU 2016-17, Marks 10

Answer

1. To show the problem is in NP, let us take a graph $G(V, E)$ and a colouring c, and checks in $O(n^2)$ time whether c is a proper colouring by checking if the end points of every edge $e \in E$ have different colours.
2. To show that 3-COLOURING is NP-hard, we give a polytime reduction from 3-SAT to 3-COLOURING.
3. That is, given an instance ϕ of 3-SAT, we will construct an instance of 3-COLOURING (*i.e.*, a graph $G(V, E)$) where G is 3-colourable iff ϕ is satisfiable.
4. Let ϕ be a 3-SAT instance and $C_1, C_2, \ldots, C_m$ be the clauses of ϕ defined over the variables $\{x_1, x_2, \ldots, x_n\}$.
5. The graph $G(V, E)$ that we will construct needs to capture two things :
   a. Somehow establish the truth assignment for $x_1, x_2, \ldots, x_n$ via the colours of the vertices of G; and
   b. Somehow capture the satisfiability of every clause C_i in ϕ.
6. To achieve these two goals, we will first create a triangle in G with three vertices $\{T, F, B\}$ where T stands for True, F for False and B for Base.
7. Consider $\{T, F, B\}$ as the set of colours that we will use to colour (label) the vertices of G.
8. Since this triangle is part of G, we need 3 colours to colour G.
9. Now we add two vertices $v_i, \bar{v}_i$ for every literal x_i and create a triangle B, $v_i, \bar{v}_i$ for every $(v_i, \bar{v}_i)$ pair, as shown in Fig. 5.20.1.

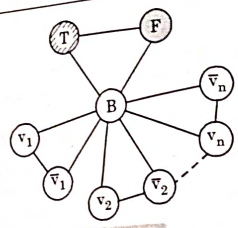


Fig. 5.20.1.

10. This construction captures the truth assignment of the literals.
11. Since if G is 3-colourable, then either v_i or $\bar{v}_i$ gets the colour T, and we interpret this as the truth assignment to v_i.
12. Now we need to add constraints to G to capture the satisfiability of the clauses of ϕ.
13. To do so, we introduce the Clause Satisfiability Gadget, (the OR-gadget). For a clause $C_i = (a \vee b \vee c)$, we need to express the OR of its literals using our colours $\{T, F, B\}$.
14. We achieve this by creating a small gadget graph that we connect to the literals of the clause. The OR-gadget is constructed as follows :

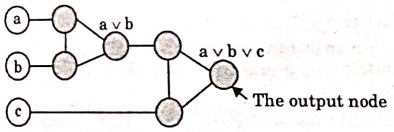


Fig. 5.20.2.

15. Consider this gadget graph as a circuit whose output is the node labeled $a \vee b \vee c$. We basically want this node to be coloured T if C_i is satisfied and F otherwise.
16. This is a two step construction : The node labelled $a \vee b$ captures the output of $(a \vee b)$ and we repeat the same operation for $((a \vee b) \vee c)$. If we play around with some assignments to a, b, c, we will notice that the gadget satisfies the following properties :
    a. If a, b, c are all coloured F in a 3-colouring, then the output node of the OR-gadget has to be coloured F. Thus capturing the unsatisfiability of the clause $C_i = (a \vee b \vee c)$.
    b. If one of a, b, c is coloured T, then there exists a valid 3-colouring of the OR-gadget where the output node is coloured T. Thus again capturing the satisfiability of the clause.
17. Once we add the OR-gadget of every C_i in ϕ, we connect the output node

of every gadget to the Base vertex and to the False vertex of the initial triangle, as follows :

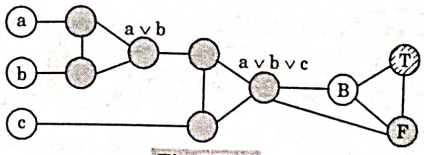


Fig. 5.20.3.

18. Now we prove that our initial 3-SAT instance ϕ is satisfiable if and only the graph G as constructed above is 3-colourable. Suppose ϕ is satisfiable and let $(x_1^*, x_2^*, ..., x_n^*)$ be the satisfying assignment.
19. If x_i^* is assigned True, we colour v_i with T and $\bar{v}_i$ with F (recall they are connected to the Base vertex, coloured B, so this is a valid colouring).
20. Since ϕ is satisfiable, every clause $C_i = (a \vee b \vee c)$ must be satisfiable, *i.e.*, at least of a, b, c is set to True. By the property of the OR-gadget, we know that the gadget corresponding to C_i can be 3-coloured so that the output node is coloured T.
21. And because the output node is adjacent to the False and Base vertices of the initial triangle only, this is a proper 3-colouring.
22. Conversely, suppose G is 3-colourable. We construct an assignment of the literals of ϕ by setting x_i to True if v_i is coloured T and vice versa.
23. Now consider this assignment is not a satisfying assignment to ϕ, then this means there exists at least one clause $C_i = (a \vee b \vee c)$ that was not satisfiable.
24. That is, all of a, b, c were set to False. But if this is the case, then the output node of corresponding OR-gadget of C_i must be coloured F.
25. But this output node is adjacent to the False vertex coloured F; thus contradicting the 3-colourability of G.
26. To conclude, we have shown that 3-COLOURING is in NP and that it is NP-hard by giving a reduction from 3-SAT.
27. Therefore 3-COLOURING is NP-complete.

Que 5.21. Prove that P is the subset of NP.

Answer

To prove : P is the subset of NP.

Proof :

1. If $L \in P$, then $L \in$ NP, as there is a polynomial time algorithm to decide L, this algorithm can easily be converted into a row argument verification algorithm that simply ignores any exception and accepts exactly those input strings it determines to be in L.

2. Thus, $P \subseteq NP$.

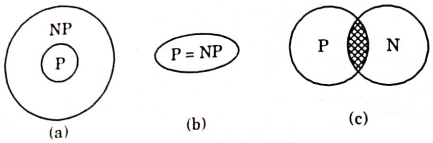


Fig. 5.21.1. *P* Vs NP.

Que 5.22. **Describe approximation algorithm in detail. How it differ with deterministic algorithm. Show that TSP is 2 approximate.**

OR

Explain approximation algorithms with suitable examples.

AKTU 2015-16, 2017-18; Marks 10

Answer

Approximation algorithm :

1. An approximation algorithm is a way of dealing with NP-completeness for optimization problem. This technique does not guarantee the best solution.
2. The best of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at most polynomial time.
3. Let $c(i)$ be the cost of solution produced by approximate algorithm and $c^*(i)$ be the cost of optimal solution for some optimization problem instance i.
4. For minimization and maximization problem, we are interested in finding a solution of a given instance i in the set of feasible solutions, such that $c(i) / c^*(i)$ and $c^*(i) / c(i)$ be as small as possible respectively.
5. We say that an approximation algorithm for the given problem instance i, has a ratio bound of $p(n)$ if for any input of size n, the cost c of the solution produced by the approximation algorithm is within a factor of $p(n)$ of the cost c^* of an optimal solution. That is

$$\max(c(i) / c^*(i), c^*(i) / c(i)) \leq p(n)$$

The definition applies for both minimization and maximization problems.

6. $p(n)$ is always greater than or equal to 1. If solution produced by approximation algorithm is true optimal solution then clearly we have $p(n) = 1$.
7. For a minimization problem, $0 < c^*(i) < c(i)$, and the ratio $c(i) / c^*(i)$ gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution.

8. Similarly, for a maximization problem, $0 < c(i) \leq c^*(i)$, and the ratio $c^*(i) / c(i)$ gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution.

Difference between deterministic algorithm and approximation algorithm :

S. No.	Deterministic algorithm	Approximation algorithm
1.	It does not deal with optimization problem.	It deals with optimization problem.
2.	It has initial and final step.	It does not have initial or final state.
3.	It require finite state machine.	It does not require finite state machine.
4.	It fails to deliver a result.	It gives an optimal result.
5.	It does not apply to maximization or minimization problem.	It applies to maximization and minimization problem.

Proof :

TSP is 2-approximate :

Let H^* denote the optimal tour. Observe that a TSP with one edge removed is a spanning tree (not necessarily MST).

It implies that the weight of the MST 'T' is in lower bound on the cost of an optimal tour.

$$c(T) \leq c(H^*)$$

A "Full" walk, W, traverse every edge of MST, T, exactly twice. That is,

$$c(W) = 2c(T)$$

which means

$$c(W) \leq 2c(H^*)$$

and we have

$$c(W) / c(H^*) \leq p(n) = 2$$

That is, the cost of walk, $c(W)$, of the solution produced by the algorithm is within a factor of $p(n) = 2$ of the cost $c(H^*)$ of an optimal solution.

Que 5.23. **Explain vertex cover problem with algorithm and analysis.**

Answer

A vertex cover of an undirected graph $G = (V, E)$ is a subset of $V' \subseteq V$ such that if edge $(u, v) \in G$ then $u \in V'$ or $v \in V'$ (or both).

Problem : Find a vertex cover of maximum size in a given undirected graph.

This optimal vertex cover is the optimization version of an NP-Complete problem but it is not too hard to find a vertex cover that is near optimal.

Approx-vertex-cover (G : Graph)

1. $c \leftarrow \phi$
2. $E' \leftarrow E[G]$
3. while E' is not empty
4. do Let (u, v) be an arbitrary edge of E'
5. $c \leftarrow c \cup \{u, v\}$
6. Remove from E' every edge incident on either u or v
7. return c

Analysis : It is easy to see that the running time of this algorithm is $O(V + E)$, using adjacency list to represent E'.

Que 5.24. Describe approximation algorithm in detail. What is the approximation ratio ? Show that vertex cover problem is 2-approximate. **AKTU 2014-15, Marks 10**

Answer

Approximation algorithm : Refer Q. 5.22, Page 5–26B, Unit-5.

Proof :

Vertex cover problem is 2-approximate :

Goal : Since this is a minimization problem, we are interested in smallest possible c / c^*. Specifically we want to show $c / c^* = 2 = p(n)$.

In other words, we want to show that Approx-Vertex-Cover algorithm returns a vertex-cover that is almost twice the size of an optimal cover.

Proof : Let the set c and c^* be the sets output by Approx-Vertex-Cover and Optimal-Vertex-Cover respectively. Also, let A be the set of edges.

Because, we have added both vertices, we get $c = 2|A|$ but Optimal-Vertex-Cover would have added one of two.

$$c / c^* \leq p(n) = 2.$$

Formally, since no two edge in A are covered by the same vertex from c^* and the lower bound :

$$|c^*| \geq A \qquad ...(5.24.1)$$

on the size of an Optimal-Vertex-Cover.

Now, we pick both end points yielding an upper bound on the size of Vertex-Cover :

$$|c| \leq 2|A|$$

Since, upper bound is an exact in this case, we have

$$|c| = 2|A| \qquad ...(5.24.2)$$

Take $|c|/2 = |A|$ and put it in equation (5.24.1)

$$|c^*| \geq |c|/2$$

$$|c^*|/|c| \geq 1/2$$

$|c^*|/|c| \leq 2 = p(n)$ Hence the theorem proved.

Que 5.25. Explain Travelling Salesman Problem (TSP) with the triangle inequality.

Answer

Problem : Given a complete graph with weights on the edges, find a cycle of least total weight that visits each vertex exactly once. When the cost function satisfies the triangle inequality, we can design an approximate algorithm for TSP that returns a tour whose cost is not more than twice the cost of an optimal tour.

APPROX-TSP-TOUR (G, c) :

1. Select a vertex $r \in V[G]$ to be a "root" vertex.
2. Compute a minimum spanning tree T for G from root r using MST-PRIM (G, c, r).
3. Let L be the list of vertices visited in a pre-order tree walk of T.
4. Return the Hamiltonian cycle H that visits the vertices in the order L.

Outline of an approx-TSP tour : First, compute a MST (minimum spanning tree) whose weight is a lower bound on the length of an optimal TSP tour. Then, use MST to build a tour whose cost is no more than twice that of MST's weight as long as the cost function satisfies triangle inequality.

Que 5.26. Write short notes on the following using approximation algorithm with example.

i. Nearest neighbour

ii. Multifragment heuristic

Answer

i. Nearest neighbour :

The following well-known greedy algorithm is based on the nearest-neighbour heuristic *i.e.*, always go next to the nearest unvisited city.

Step 1 : Choose an arbitrary city as the start.

Step 2 : Repeat the following operation until all the cities have been visited : go to the unvisited city nearest the one visited last (ties can be broken arbitrarily).

Step 3 : Return to the starting city.

Example :

1. For the instance represented by the graph in Fig. 5.26.1, with a as the starting vertex, the nearest-neighbour algorithm yields the tour (Hamiltonian circuit) s_a: $a - b - c - d - a$ of length 10.
2. The optimal solution, as can be easily checked by exhaustive search, is the tour s^*: $a - b - d - c - a$ of length 8. Thus, the accuracy ratio of this approximation is

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{10}{8} = 1.25$$

i.e., tour s_a is 25 % longer than optimal tour s^*.

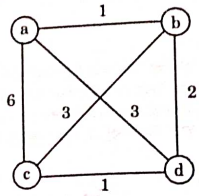


Fig. 5.26.1. Instance of the traveling salesman problem.

3. Unfortunately, except for its simplicity, not many good things can be said about the nearest-neighbour algorithm.
4. In particular, nothing can be said in general about the accuracy of solutions obtained by this algorithm because it can force us to traverse a very long edge on the last leg of the tour.
5. Indeed, if we change the weight of edge (a, d) from 6 to an arbitrary large number $w \geq 6$ in given example, the algorithm will still yield the tour $a-b-c-d-a$ of length $4+w$, and the optimal solution will still be $a-b-d-c-a$ of length 8. Hence,

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{4+w}{8}$$

which can be made as large as we wish by choosing an appropriately large value of w. Hence, $RA = \infty$ for this algorithm.

ii. Multifragment heuristic :

Another natural greedy algorithm for the traveling salesman problem considers it as the problem of finding a minimum-weight collection of edges in a given complete weighted graph so that all the vertices have degree 2.

Step 1 : Sort the edges in increasing order of their weights. (Ties can be broken arbitrarily.) Initialize the set of tour edges to be constructed to the empty set.

Step 2 : Repeat this step n times, where n is the number of cities in the instance being solved : add the next edge on the sorted edge list to the set of tour edges, provided this addition does not create a vertex of degree 3 or a cycle of length less than n; otherwise, skip the edge.

Step 3 : Return the set of tour edges.

Example :

1. Applying the algorithm to the graph in Fig. 5.26.1 yields $\{(a, b), (c, d), (b, c), (a, d)\}$.
2. There is, however, a very important subset of instances, called Euclidean, for which we can make a non-trivial assertion about the accuracy of both the nearest-neighbour and multifragment-heuristic algorithms.

3. These are the instances in which intercity distances satisfy the following natural conditions :
   a. **Triangle inequality :** $d[i, j] \leq d[i, k] + d[k, j]$ for any triple of cities i, j, and k (the distance between cities i and j cannot exceed the length of a two-leg path from i to some intermediate city k to j).
   b. **Symmetry :** $d[i, j] = d[j, i]$ for any pair of cities i and j (the distance from i to j is the same as the distance from j to i).
4. A substantial majority of practical applications of the traveling salesman problem are its Euclidean instances.
5. They include, in particular, geometric ones, where cities correspond to points in the plane and distances are computed by the standard Euclidean formula.
6. Although the performance ratios of the nearest-neighbour and multifragment-heuristic algorithms remain unbounded for Euclidean instances, their accuracy ratios satisfy the following inequality for any such instance with $n \geq 2$ cities :

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{4+w}{8}$$

where $f(s_a)$ and $f(s^*)$ are the lengths of the heuristic tour and shortest tour.

Que 5.27. Write short notes on randomized algorithms.

AKTU 2013-14, Marks 10

OR

Explain randomized algorithms. **AKTU 2017-18, Marks 05**

Answer

1. A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased bits and it is then allowed to use these random bits to influence its computation.
2. An algorithm is randomized if its output is determined by the input as well as the values produced by a random number generator.
3. A randomized algorithm makes use of a randomizer such as a random number generator.
4. The execution time of a randomized algorithm could also vary from run to run for the same input.
5. The algorithm typically uses the random bits as an auxiliary input to guide its behaviour in the hope of achieving good performance in the "average case".
6. Randomized algorithms are particularly useful when it faces a malicious attacker who deliberately tries to feed a bad input to the algorithm.

Randomized algorithm are categorized into two classes :

i. **Las Vegas algorithm :** This algorithm always produces the same output for the same input. The execution time of Las Vegas algorithm depends on the output of the randomizer.

ii. **Monte Carlo algorithm :**

a. In this algorithm output might differ from run to run for the same input.

b. Consider any problem for which there are only two possible answers, say yes and no.

c. If a Monte Carlo algorithm is used to solve such a problem then the algorithm might give incorrect answers depending on the output of the randomizer.

d. Then the requirement is that the probability of an incorrect answer from a Monte Carlos algorithm be low.

Que 5.28. Write a short note on randomized algorithm. Write its merits, and applications.

Answer

Randomized algorithm : Refer Q. 5.27, Page 5–31B, Unit-5.

Merits :

1. Simple.
2. High efficiency.
3. Better complexity bounds.
4. Random selection of good and bad choices.
5. Cost efficient.

Applications :

1. Randomized quick sort algorithm
2. Randomized minimum-cut algorithm
3. Randomized algorithm for N-Queens problem
4. Randomized algorithm for majority element

Que 5.29. Write the EUCLID'S GCD algorithm. Compute gcd (99, 78) with EXTENDED-EUCLID.

Answer

Euclid's GCD algorithm :

The inputs a and b are arbitrary non-negative integers.

EUCLID (a, b) {

if ($b == 0$)

then return a;

else return EUCLID (b, a mod b); }

EXTEUCLID (a, b) {

//returns a triple (d, x, y) such that $d = \gcd(a, b)$


```
//d = = (a × x + b × y)
if (b = = 0) return (a, 1, 0);
(d1, x1, y1) = EXTEUCLID (b, a % b);
d = d1;
x = y1;
y = x1 – (a div b) × y1;   //div = integer division
return (d, x, y);
}
```

Let $a = 99$ and $b = 78$

a	b	$[a/b]$	d	x	y
99	78	1	3	–11	14
78	21	3	3	3	–11
21	15	1	3	–2	3
15	6	2	3	1	–2
6	3	2	3	0	1
3	0	–	3	1	0

i. In the 5th receive ($a = 6$, $b = 3$), values from the 6th call ($b = 0$) has $d_1 = 3$, $x_1 = 1$ and $y_1 = 0$. Still within the 5th call we calculate that $d = d_1 = 3$, $x = y_1 = 0$ and $y = x_1 - (a \text{ div } b) \times y_1 = 1 - 2 \times 0 = 1$.

ii. In the 4th receive ($a = 15, b = 6$), the values $d_1 = 3, x_1 = 0$ and $y_1 = 1$ from the 5th call, then compute $x = y_1 = 1$ and

$y = x_1 - (a \text{ div } b) \times y_1 = 0 - 2 \times 1 = -2$.

iii. In the 3rd receive ($a = 21, b = 15$), $x = -2$ and

$y = x_1 - (a \text{ div } b) \times y_1 = 1 - (1 \times -2) = 3$.

iv. In the 2nd receive ($a = 78, b = 21$), $x = 3$ and

$y = x_1 - (a \text{ div } b) \times y_1 = (-2) - 3 \times 3 = -11$.

v. In the 1st receive ($a = 99, b = 78$), $x = -11$ and

$y = x_1 - (a \text{ div } b) \times y_1 = 3 - 1 \times (-11) = 14$.

vi. The call EXTEUCLID (99, 78) return (3, – 11, 14), so gcd (99, 78) = 3 and gcd (99, 78) = 3 = 99 × (–11) + 78 × 14.

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. What is Fast Fourier Transformation and how it works ?

Ans. Refer Q. 5.1.

Q. 2. Explain the following string matching algorithms :
a. Naive string matching
b. Rabin-Karp algorithm
c. Knuth-Morris-Pratt algorithm

Ans.
a. Refer Q. 5.3.
b. Refer Q. 5.8.
c. Refer Q. 5.5.

Q. 3. Discuss string matching with finite automata.
Ans. Refer Q. 5.9.

Q. 4. Discuss the problem classes P, NP and NP-complete.
Ans. Refer Q. 5.10.

Q. 5. Differentiate NP-complete with NP-hard.
Ans. Refer Q. 5.14.

Q. 6. Show that CUQUE NP-complete.
Ans. Refer Q. 5.18.

Q. 7. Explain the following :
a. Approximation algorithm
b. Randomized algorithm

Ans.
a. Refer Q. 5.22.
b. Refer Q. 5.27.