1

UNIT

Introduction

CONTENTS

PART-1

Introduction : Algorithms, Analyzing Algorithms, Complexity of Algorithms.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.1. What do you mean by algorithm ? Write the characteristics of algorithm.

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 :** The algorithm must accept zero or more inputs and must produce at least one output.
2. **Definiteness :** Each step of algorithm must be clear and unambiguous.
3. **Effectiveness :** Every step must be basic and essential.
4. **Finiteness :** Total number of steps used in algorithm should be finite.

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*.

PART-2

Growth of Functions, Performance Measurements.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

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

OR

Discuss asymptotic notations in brief.

Answer

1. Asymptotic notation is a shorthand way to represent the fastest possible and slowest possible running times for an algorithm.
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) :
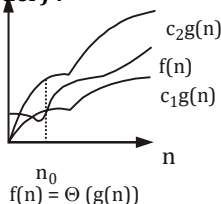

$$f(n) = \Theta(g(n))$$

Fig. 1.3.1.

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.

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 :

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

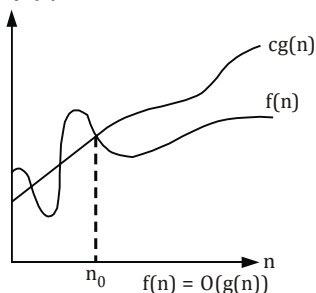*i.e.*, the set of functions which, as n gets large, grow faster than a constant time $f(n)$.

Fig. 1.3.2.

3. **и-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 $cg(n)$.
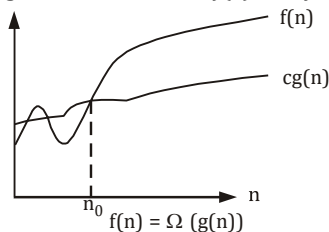

Fig. 1.3.3.

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 $0 \leq f(n) < cg(n) \ \forall \ n \geq n_0\}$

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 $0 \leq cg(n) < f(n) \ \forall \ n \geq n_0\}$

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

Answer

$$\text{If } f(n) = 100 * 2^n + n^5 + n$$
$$\text{For } n^5 \geq n$$
$$100 * 2^n + n^5 + n \leq 100 * 2^n + n^5 + n^5$$
$$\leq 100 * 2^n + 2n^5$$
$$\text{For } 2^n \geq n^5$$
$$100 * 2^n + n^5 + n \leq 100 * 2^n + 2.2^n$$
$$\leq 102 * 2^n \qquad [\because n \leq 1, n_0 = 23]$$
Thus, $\qquad f(n) = O(2^n)$

Que 1.5. 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 = Number of sub-problems in the recursion

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

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

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

Case 1 :
If it is true that : $\qquad f(n) = O(n^{\log_b a - E})$ for $E > 0$
It follows that : $\qquad T(n) = \Theta(n^{\log_b a})$

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

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

It follows that : $T(n) = \Theta\left(n^{\log_b a} \log(n)\right)$

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

and if it is also true that :

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

It follows that : $\qquad T(n) = \Theta(f(n))$

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

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 :

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

If we insert once more the values, we get :

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

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

$$\frac{1}{2} n^2 \le \frac{1}{2} n^2 \ \forall \ n \ge 1$$

So, it follows that : $\quad T(n) = \Theta\,(f(n))$

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 1.6. | **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 ?

Answer

Given that :

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2 \qquad ...(1.6.1)$$

$$T'(n) = aT'\left(\frac{n}{4}\right) + n^2 \qquad ...(1.6.2)$$

Here, eq. (1.6.1) defines the running time for algorithm A and eq. (1.6.2) 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 eq. (1.6.1) by $T(n) = aT\left(\dfrac{n}{b}\right) + f(n)$

we get, $\qquad\qquad a = 7$
$\qquad\qquad\qquad b = 2$
$\qquad\qquad\quad f(n) = n^2$

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

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

Case 1 : $\qquad\quad f(n) = O(n^{\log_2 7 - E})$
$\Rightarrow \qquad\qquad f(n) = O(n^{2.81 - E})$
$\Rightarrow \qquad\qquad f(n) = O(n^{2.81 - 0.81})$
$\Rightarrow \qquad\qquad f(n) = O(n^2)$

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

Thus, $\qquad\qquad T(n) = \theta(n^{\log_b a})$
$\Rightarrow \qquad\qquad T(n) = \theta(n^{2.81})$

Since recurrence given by eq. (1.6.1) is asymptotically bounded by θ-notation by which is used to show optimum time we have to show that recurrence given by eq. (1.6.2) is bounded by Ω-notation which shows minimum time (best case).

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

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

$\Rightarrow \qquad\qquad a = 16$
$\qquad\qquad\qquad b = 4$
$\qquad\qquad\quad f(n) = n^2$

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

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

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

Therefore, this shows that A' is asymptotically faster than A when

$a = 16$.

Que 1.7. The recurrence $T(n) = 7T(n/3) + n^2$ describes the running time of an algorithm A. Another competing algorithm B has a running time of $S(n) = a\,S(n/9) + n^2$. What is the smallest value of a such that B is asymptotically faster than A ?

AKTU 2018-19, Marks 10

Answer

Given that :

$$T(n) = 7\,T\left(\frac{n}{3}\right) + n^2 \qquad \qquad ...(1.7.1)$$

$$S'(n) = aS'\left(\frac{n}{9}\right) + n^2 \qquad \qquad ...(1.7.2)$$

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

Now, compare eq. (1.7.1) with $T(n) = a\,T\left(\dfrac{n}{b}\right) + f(n)$

we get, $a = 7, \quad b = 3$

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

$$n^{\log_b a} = n_{\log_3 7} = n^{2.81}$$

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

Case 3 : $f(n) = O(n^{\log_3 7 + \epsilon})$
$\Rightarrow$ $f(n) = O(n^{1.77 + \epsilon})$
$\Rightarrow$ $f(n) = O(n^{1.77 + 0.23})$
$\Rightarrow$ $f(n) = O(n^2)$

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

Thus, $T(n) = \theta\,f(n)$
$\Rightarrow$ $T(n) = \theta\,(n^2)$

Since recurrence (1) is asymptotically bounded by θ-notation which is used to show optimum time we have to show that recurrence given by eq. (1.7.2) is bounded by Ω-notation which shows minimum time (best case).

For the use satisfy the case 2 of Master theorem, Guess $a = 81$

$$S'(n) = f(n) = 81\,S'\left(\frac{n}{9}\right) + n^2$$

$\Rightarrow$ $a = 81, \quad b = 9$

$$f(n) = n^{\log_9 81}$$

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

Hence, case 2 of Master's theorem is satisfied.

$\Rightarrow$ $T(n) = \theta\,(n^{\log_9 81} \log n)$

Design and Analysis of Algorithms
www.aktutor.in
1–9 B (CS/IT-Sem-5)

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

Therefore, this shows that B is asymptotically faster than A when $a = 81$.

Que 1.8. Solve the following recurrences :

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

Answer

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

Let
$$m = \log n$$
$$n = 2^m$$
$$n^{1/2} = 2^{m/2} \qquad \qquad ...(1.8.2)$$

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

$$T(2^m) = T(2^{m/2}) + O(\log 2^m) \qquad \qquad ...(1.8.3)$$

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

Putting the value of $x(m)$ in eq. (1.8.3)

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

Solution of eq. (1.8.5) is given as
$$a = 1, \quad b = 2, \quad f(n) = O(m)$$
$$m^{\log_b a} = m^{\log_2 1 + E} \qquad \qquad \text{where } E = 1$$
$$x(m) = \theta(\log m)$$
$$T(n) = \theta(\log \log n)$$

Que 1.9.

i. Solve the recurrence $T(n) = 2T(n/2) + n^2 + 2n + 1$

ii. Prove that worst case running time of any comparison sort is $\nu(n \log n)$.

AKTU 2019-20, Marks 07

Answer

i. $\qquad T(n) = 2T(n/2) + n^2 + 2n + 1 \approx 2T\left(\frac{n}{2}\right) + n^2$

Compare it with $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

we have, $\qquad a = 2, b = 2, f(n) = n^2$
Now, we apply cases for Master's theorem.

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

This satisfies case 3 of Master's theorem.

$\Rightarrow \qquad f(n) = \Omega\left(n^{\log_b a + E}\right) = \Omega\left(n^{1+E}\right)$

$$= \Omega \left(n^{1+1} \right) \qquad \text{where } E = 1$$
$$= \Omega \left(n^2 \right)$$

Again $\qquad 2f\left(\dfrac{n^2}{2}\right) \le c\, f(n^2)$ ...(1.9.1)

eq. (1.9.1) is true for $c = 2$

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

$\Rightarrow \qquad\qquad T(n) = \theta\, f(n^2)$

ii. Let $T(n)$ be the time taken by merge sort to sort any array of n elements.

Therefore, $\qquad T(n) = T\left(\left\lceil \dfrac{n}{2} \right\rceil\right) + \left(\left\lceil \dfrac{n}{2} \right\rceil\right) + g(n)$

where $\qquad g(n) \in \theta(n)$

This recurrence, which becomes :

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

when n is even is a special case of our general analysis for divide-and conquer algorithms.

Compare the above given recurrence with $T(n) = aT\left(\dfrac{n}{b}\right) + f(n)$

we get $\qquad\qquad a = 2$

$\qquad\qquad\qquad b = 2$

$\qquad\qquad\qquad f(n) = g(n)$

Now we find, $\quad n^{\log_b a} = n^{\log_2 2} = n^1 = n$

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

i.e., case 2 of Master's theorem applied then

$$T(n) = \Omega\left(n^{\log_b a}\ \log n\right)$$

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

Hence, the worst case running time of merge sort is $\Omega(n \log n)$.

Que 1.10. **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. Then P is called recursive procedure.

4. 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.

5. A recursive procedure with these two properties is said to be well-defined.

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 1.11. **What is recursion tree ? Describe.**

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 in the form of tree.

Que 1.12. **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, $n - k = 1$

$$k = n - 1$$

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

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

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

$$= 2^{n-1} - 1 = O(2^n)$$
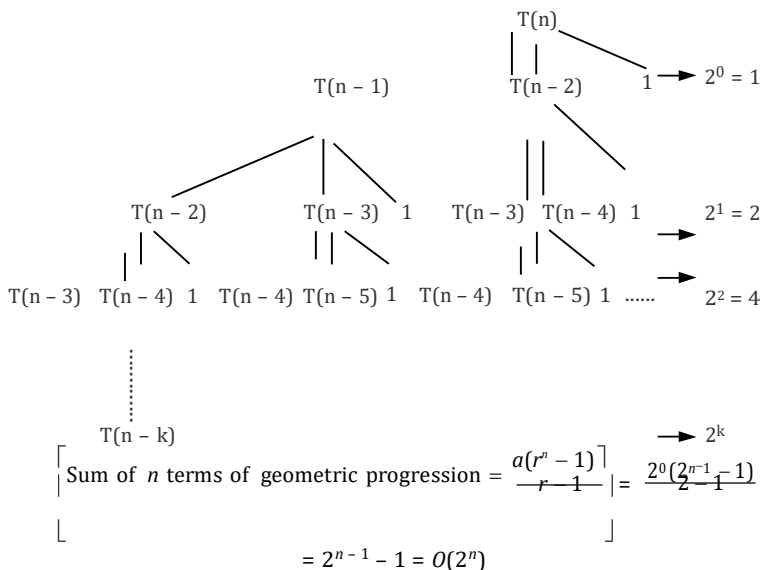
Que 1.13. | **Solve the following recurrences :**
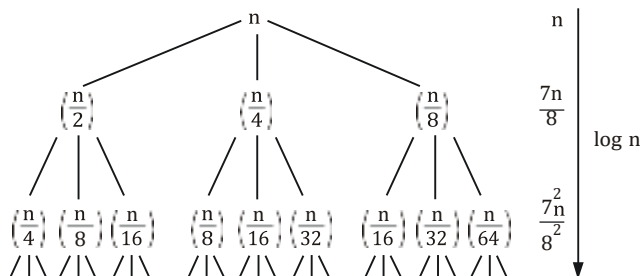
$$T(n) = T(n/2) + T(n/4) + T(n/8) + n$$

AKTU 2019-20, Marks 07

Answer

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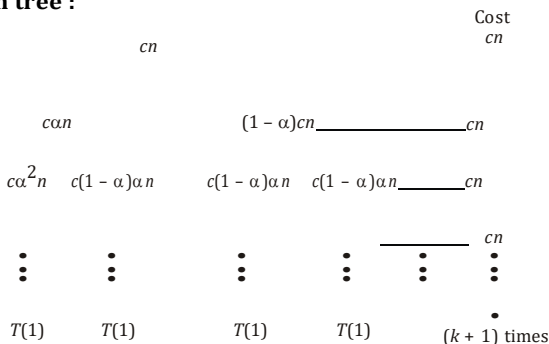$$= \Omega (n \log n)$$

Que 1.14. | **Use a recursion tree to give an asymptotically tight solution to the recurrence** $T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$, **where** α **is a constant in the range** $0 < \alpha < 1$ **and** $c > 0$ **is also a constant.**

AKTU 2018-19, Marks 07

$$T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$$

Recursion tree :

Cost
cn

cn

$c\alpha n$ $(1 - \alpha)cn$ _____ cn

$c\alpha^2 n$ $c(1 - \alpha)\alpha n$ $c(1 - \alpha)\alpha n$ $c(1 - \alpha)\alpha n$ _____ cn

$\vdots$ $\vdots$ $\vdots$ $\vdots$ $\vdots$ cn

$T(1)$ $T(1)$ $T(1)$ $T(1)$ $(k + 1)$ times

Assuming $T(1) = 1$

So $\qquad c(1 - \alpha)k\, n = 1$

$$cn = \frac{1}{(1 - \alpha)^k} = \left(\frac{1}{1 - \alpha}\right)^k$$

$$\log (n) = k \log \left(\frac{1}{1-\alpha}\right)$$

$$k = \frac{\log cn}{\log \left(\frac{1}{1 - \alpha}\right)} = \log_{\frac{1}{1-\alpha}} (cn)$$

So, $\qquad$ Total cost $= cn + cn + \ldots\ldots\ldots(k + 1)$ times $= cn(k + 1)$

$$= cn \times \log_{\frac{1}{1-\alpha}} (cn)$$

Time complexity $= O\left(n \log_{\frac{1}{1-\alpha}} (cn)\right) = O\left(n \log_{\frac{1}{1-\alpha}} n\right)$

Que 1.15.   **Solve the following by recursion tree method**

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

AKTU 2017-18, Marks 10

Answer

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

n   n

$\frac{n}{5}$ $\frac{4n}{5}$ _____ n

                                    log n

$\frac{n}{25}$ $\frac{4n}{25}$ $\frac{4n}{25}$ $\frac{16n}{25}$ n

$$\frac{n}{125} \quad \frac{4n}{125} \quad \frac{4n}{125} \quad \frac{16n}{125} \quad \frac{4n}{125} \quad \frac{16n}{125} \quad \frac{16n}{125} \quad \frac{64n}{125} \qquad n$$
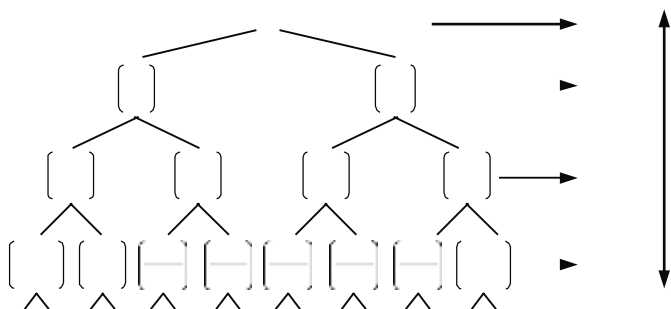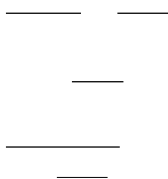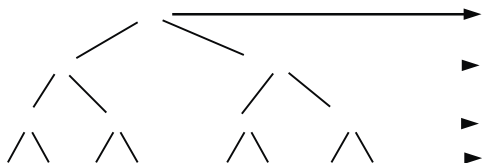
PART-3

Sorting and Order Statistic : Shell Sort, Quick Sort.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.16. 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.
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 ← i
   while $j \geq$ inc and $a[j - inc]$ > temp
   　　　$a[j]$ ← $a[j - inc]$
   　　　j ← $j -$ inc
   $a[j]$ ← 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 |
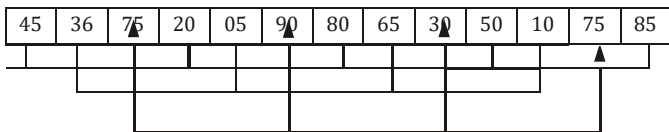
(*a*)

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 |
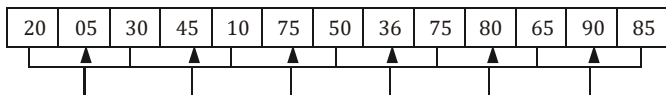
(*b*)

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 |
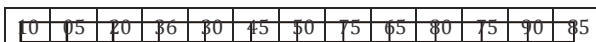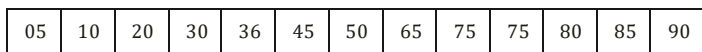
(*d*)

Fig. 1.16.1.

Que 1.17. **Describe any one of the following sorting techniques :**

i. **Selection sort**

~~**ii.** **Insertion sort**~~

Answer

i. **Selection sort (*A*) :**
   1. $n \leftarrow$ length [*A*]
   2. for $j \leftarrow 1$ to $n-1$
   3. smallest $\leftarrow j$
   4. for $i \leftarrow j + 1$ to n
   5. if $A[i] < A[\text{smallest}]$
   6. then smallest $\leftarrow i$
   7. exchange ($A[j]$, $A[\text{smallest}]$)

ii. **Insertion_Sort(*A*) :**
   1. for $j \leftarrow 2$ to length[*A*]
   2. do key $\leftarrow A[j]$
   3. Insert $A[j]$ into the sorted sequence $A[1 .... j-1]$
   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 1.18. | **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$(1 \ldots n)$
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 1.19. | **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 : Given array to be sorted

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

Sort the array A using quick sort algorithm.

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.

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

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.
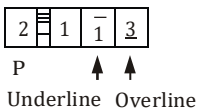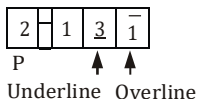
2	1	3	1

P

Underline Overline

2	1	1	3

P

Underline Overline
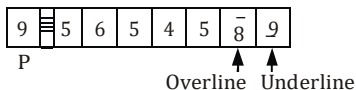
The pointers have crossed.

i.e., overline on left of underlined.
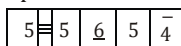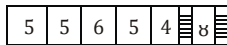Swap pivot with overline

1	1	2	3

Sorted array
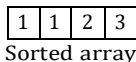
Now, for subarray 2 we apply Quick sort procedure.

9	5	6	5	4	5	8	9

P

Overline Underline

The pointer has crossed. Then swap pivot with overline.

8	5	6	5	4	5	9	9

Subarray 3   Subarray 4

8	5	6	5	4	5
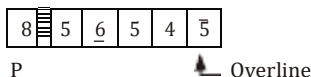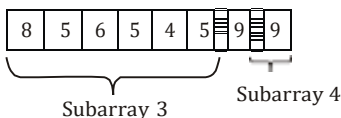
P   Overline

Swap overline with pivot.

5	5	6	5	4	8

P

5	5	6	5	4

Underline   Overline

5	5	4	5	6

Overline on left of underlined.
Swap pivot with overline.

4	5	5	5	6

P
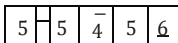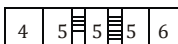
Now combine all the subarrays

Sorted array ► | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 8 | 9 | 9 |

Pivot
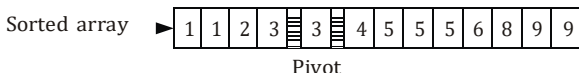
Analysis of complexity :

i.   Worst case :

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

$$T(n) = \max_{o \le q \le n-1} (T(q) + T(n - q - 1)) + \Theta(n) \quad ...(1.19.1)$$

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 eq. (1.19.1) we get

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

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

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

$$T(n) = 1/n \, (T(1) + T(n - 1) + {}^{n-1}S_{q=1} \, T(q)$$
$$T(n - q))) + \Theta(n) \quad ...(1.19.1)$$

We know from worst-case analysis

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

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

$$= 1/n \, {}^{n-1}S_{q=1} \, (T(q) + T(n - q)) + Q(n) \quad ...(1.19.2)$$

$$= 1/n \left[2 \sum_{k=1}^{n-1} (T(k)) \right] + \Theta(n)$$

$$= 2/n \sum_{k=1}^{n-1} (T(k)) + \Theta(n) \qquad \qquad ...(1.19.3)$$

6.   Solve the above recurrence using substitution method. Assume that $T(n) \leq an \, n \log n + b$ 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

$$T(n) \geq \sideset{}{_{k=1}^{n-1}}\Theta \, 2/n \, (ak \log k + b) + \Theta(n)$$
$$= 2a/n \sideset{}{_{k=1}^{n-1}}S \, k \log k - 1/8(n^2) + 2b/n$$
$$(n-1) + \Theta \, n \qquad \qquad ...(1.19.4)$$

At this point we are claiming that

$$\sideset{}{_{k=1}^{n-1}}\Theta \, k \log k \leq 1/2 \, n^2 \log n - 1/8(n^2)$$

Substituting this claim in the eq. (1.19.4), we get

$$T(n) \leq 2a/n \, [1/2n^2 \log n - 1/8(n^2)] + 2/n \, b(n-1) + \Theta(n)$$
$$\leq an \log n - an/4 + 2b + \Theta(n) \qquad ... (1.19.5)$$

In the eq. (1.19.5), $\Theta(n) + b$ 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 1.20. | **Discuss the best case and worst case complexities of quick sort algorithm in detail.**

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 :

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

And from case (2) of master theorem

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

Worst case : Refer Q. 1.19, Page 1–16B, Unit-1.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.21. | **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 |
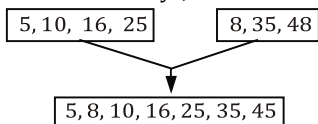2. **Consider the first part :** 10,   25,   16,   5 again divide into two sub-arrays

$$10 \quad , \quad 25 \qquad 16 \quad , \quad 5$$

| 10, 25 | | 5, 16 |

| 5, 10, 16, 25 |
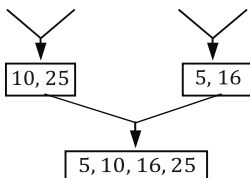
3. **Consider the second half :** 35, 48, 5 again divide into two sub-arrays

$$35 \quad , \quad 48 \qquad \qquad 8$$

| 35, 48 | 8 |

| 8, 35, 48 |
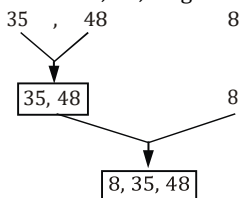
4. Merge these two sorted sub-arrays,

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

| 5, 8, 10, 16, 25, 35, 45 |

This is the sorted array.

Que 1.22. **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 :

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n = 2T\left(\frac{n}{2}\right) + n \qquad ...(1.22.1)$$

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

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn \qquad ...(1.22.2)$$

6. For the best case there are only $n/2$ comparisons hence equation (1.22.2) 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 :

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

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

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

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

PART-5

Heap Sort.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.23.   Explain heap sort algorithm with its complexity.

OR

Discuss Max-Heapify and Build-Max-Heap procedures.

Answer

1. Heap sort is a comparison based sorting technique based on binary heap data structure.
2. 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.
3. 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 $\leftarrow l$
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 ≠ 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.

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. How will you sort following array A of element using heap sort : $A = (23, 9, 18, 45, 5, 9, 1, 17, 6)$.

AKTU 2018-19, Marks 10

Answer

Given array : | 23 | 9 | 18 | 45 | 5 | 9 | 1 | 17 | 6 |
First we call Build-Max heap
heap size [A] = 9
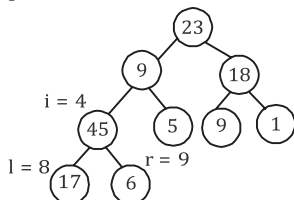
so i = 4 to 1 call MAX HEAPIFY (A, i)
i.e., first we call MAX HEAPIFY (A, 4)
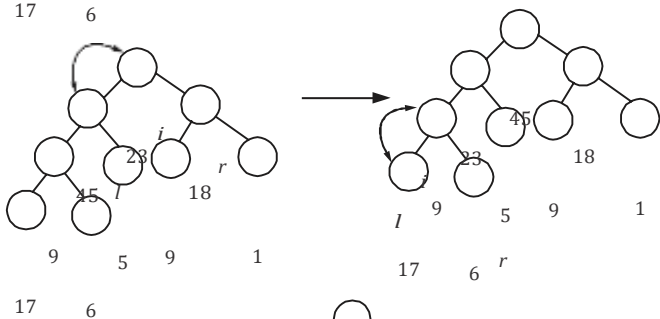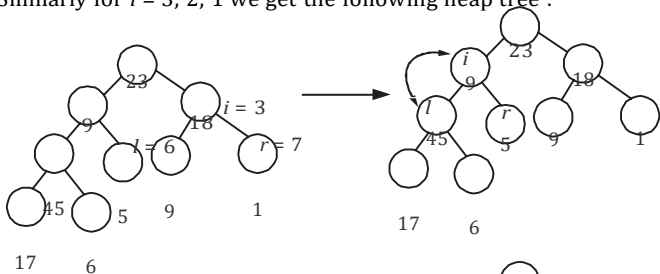$$A[l] = 7, A[i] = A[4] = 45, A[r] = 6$$
$$l \leftarrow \text{left } [4] = 2 \times 4 = 8$$
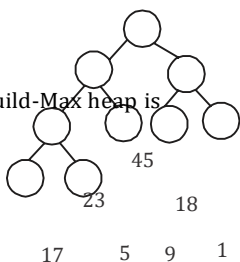$$r \leftarrow \text{right}[4] = 2 \times 4 + 1 = 9$$
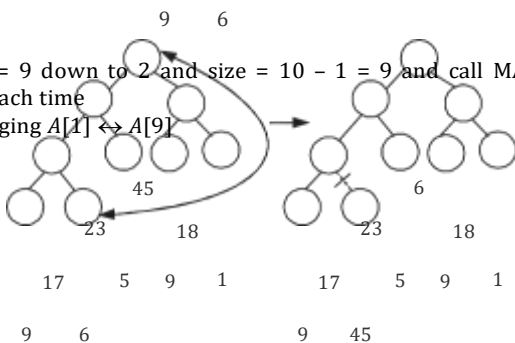$$8 \leq 9 \text{ and } A[8] = 17 < 45 \text{ (False)}$$

Then, largest ← 4.

Similarly for *i* = 3, 2, 1 we get the following heap tree :

So, final tree after Build-Max heap is

Now *i* = 9 down to 2 and size = 10 – 1 = 9 and call MAX HEAPIFY (*A*, 1) each time
Exchanging *A*[1] ↔ *A*[9]
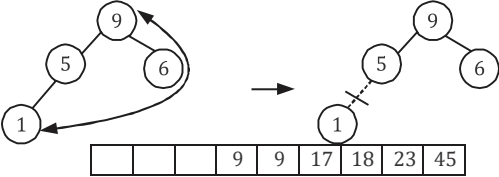
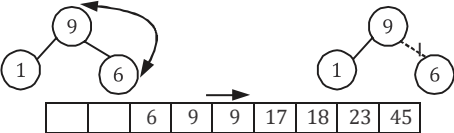|								45

Now call MAX HEAPIFY (A, 1) and
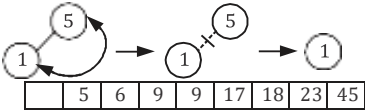Exchange $A[1]$ and $A[8]$, size = 9 – 1 = 8
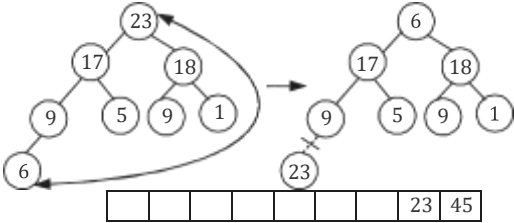
Now, call MAX HEAPIFY (A, 1), exchange A[1] and A[4] and size = 5 – 1 = 4

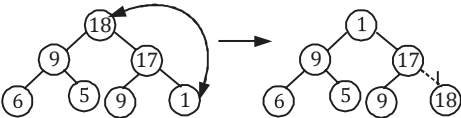Now, call MAX HEAPIFY (A, 1), exchange A[1] and A[3] and size = 4 – 1 = 3
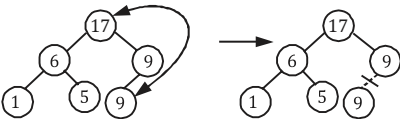

Exchange A[1] and A[2] size 3 – 1 = 2

The sorted array

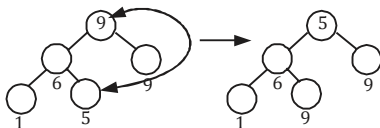| 1 | 5 | 6 | 9 | 9 | 17 | 18 | 23 | 45 |

Now call MAX HEAPIFY (A, 1) and exchange A[1] and A[7]

Now call MAX HEAPIFY (A, 1) and size = 7 – 1 = 6 exchange A[1] and A[6]

Exchange $A[1]$ and $A[5]$ and size = 6 – 1 = 5

Que 1.25. **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 and its time complexity : Refer Q. 1.23, Page 1–23B, 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 1.26. **Explain HEAP SORT on the array. Illustrate the operation HEAP SORT on the array A = {6, 14, 3, 25, 2, 10, 20, 7, 6}**
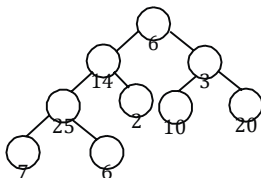
AKTU 2017-18, Marks 10

Answer

Heap sort : Refer Q. 1.23, Page 1–23B, 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 $(A, 4)$
$$Az \ [l] = 7, A \ [i] = A \ [4] = 25, A \ [r] = 6$$
$$l \leftarrow \text{left} \ [4] = 8$$
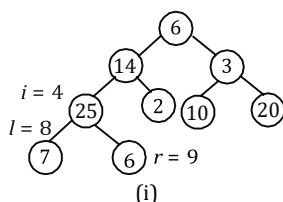$$r \leftarrow \text{right} \ [4] = 9$$
$8 \leq 9$ and $7 > 25$ (False)
Then, largest $\leftarrow 4$
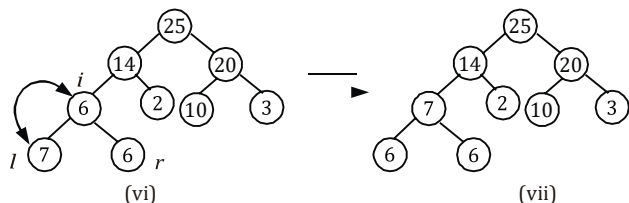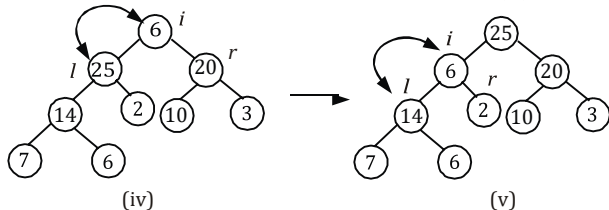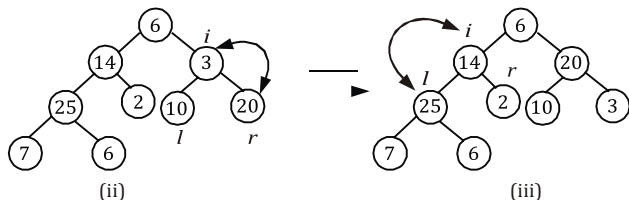$9 \leq 9$ and $6 > 25$ (False)
Then, largest = 4
$A \ [i] \leftrightarrow A \ [4]$
Now call MAX-HEAPIFY $(A, 2)$

(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] ↔ A [9]

| 6 | 14 | 20 | 7 | 2 | 10 | 3 | 6 : 25 |
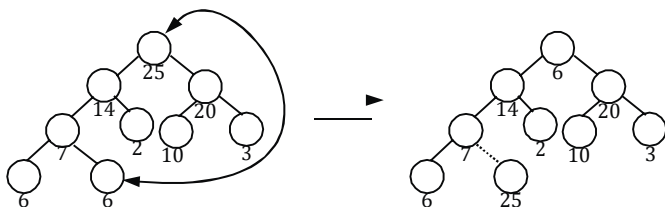
Now call MAX-HEAPIFY (A, 1) we get
Now exchange A [1] and A [8] and size = 8 – 1 = 7

| 6 | 14 | 10 | 7 | 2 | 6 | 3 : 20 |
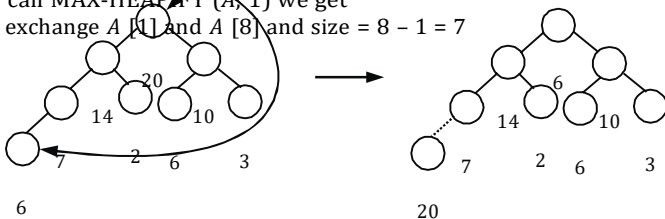
Again call MAX-HEAPIFY (A, 1), we get
exchange A [1] and A [7] and size = 7 – 1 = 6

| 3 | 7 | 10 | 6 | 2 | 6 : 14 |
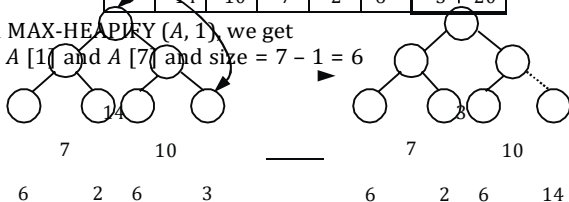
Again call MAX-HEAPIFY (A, 1), we get
exchange A [1] and A [6] and now size = 6 – 1 = 5

| 3 | 7 | 6 | 6 | 2 : 10 |
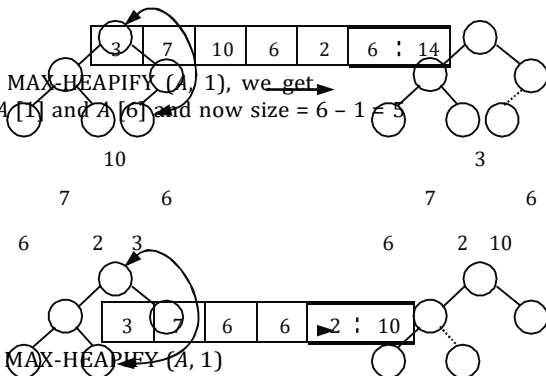
Again call MAX-HEAPIFY (A, 1)
exchange A [1] and A [5] and now size = 5 – 1 = 4

6   6   ＿＿＿   6   6

3   2   3   7

| 2 | 6 | 6 | 3 : 7 |

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

| 3 | 6 | 2 : 6 | | |
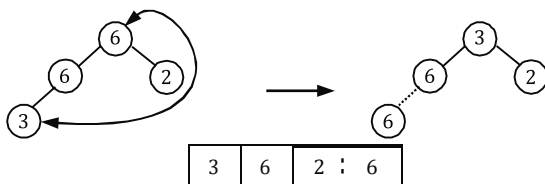
call MAX-HEAPIFY (*A*, 1)
exchange *A* [1] and *A* [3], size = 3 – 1 = 2

| 2 | 3 : 6 |
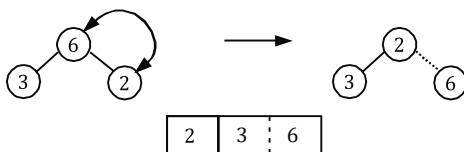
call MAX-HEAPIFY (*A*, 1)
exchange *A* [1] and *A* [2] and size = 2 – 1 = 1

| 2 : 3 |
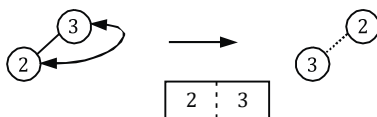
Thus, sorted array :

| 2 | 3 | 6 | 6 | 7 | 10 | 14 | 20 | 25 |

PART-6

Comparison of Sorting Algorithms, Sorting in Linear Time.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.27. | **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 1.28. **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.   do $C[i] \leftarrow 0$
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 1.29. **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}.**

Answer

Given array : 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   (∵ length [A] = 10)
                $C[A[j]] \leftarrow C[A[j]] + 1$

For j = 1

```
                                 0  1  2  3  4  5  6
$C[A[1]] \leftarrow C[1] + 1 = 0 + 1 = 1$   C  0  1  0  0  0  0  0
```
$C[1] \leftarrow 1$

For j = 2

```
                                 0  1  2  3  4  5  6
$C[A[2]] \leftarrow C[6] + 1 = 0 + 1 = 1$   C  0  1  0  0  0  0  1
```
$C[6] \leftarrow 1$

```
                                 0  1  2  3  4  5  6
Similarly for $j$ = 5, 6, 7, 8, 9, 10   C  0  1  0  3  2  2  2
```

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
$C[1] \leftarrow C[1] + C[0]$   C  0  1  0  3  2  2  2
```
$C[1] \leftarrow 1 + 0 = 1$
For i = 2

```
                          0  1  2  3  4  5  6
$C[2] \leftarrow C[2] + C[1]$   C  0  1  1  3  2  2  2
```
$C[1] \leftarrow 1 + 0 = 1$

```
                          0  1  2  3  4  5  6
Similarly for $i$ = 4, 5, 6  C  0  1  1  4  6  8  10
```

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 1.30. **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 $0 \leq A[i] \leq 1$. We also need an auxiliary array $B[0 \dots n-1]$ 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 $B[0], B[1], \dots B[n-1]$ together in order.

Que 1.31. **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 stable by nature, such as bubble sort, insertion sort, merge sort, counting sort etc.
4. Let A be an array, and let < be a strict weak ordering on the elements of A. Sorting algorithm is stable if :

    $i < j$ and $A[i] \equiv A[j]$ i.e., $A[i]$ comes before $A[j]$.
5. Stability means that equivalent elements retain their relative positions, after sorting.

    For example :
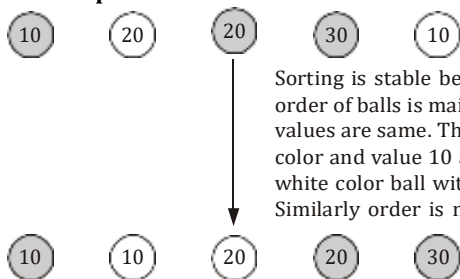
Sorting is stable because the order of balls is maintained when values are same. The bal with gray color and value 10 appears before the white color ball with value 10. Similarly order is maintained for 20.

Que 1.32. | **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. for $i \leftarrow 1$ 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 1.32.1.

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

Que 1.33. **Among Merge sort, Insertion sort and quick sort which sorting technique is the best in worst case. Apply the best one among these algorithms to sort the list *E, X, A, M, P, L, E* in alphabetic order.**

AKTU 2019-20, Marks 07

Answer

Merge sort technique is best in worst case because of its time complexity $O(n \log n)$.

Numerical :

Given : E, X, A, M, P, L, E

Pass 1 : Merge each pair of element to obtain sorted list :

| E | X | A | M | | P | L | E |

After sorting each pair, we get

| E | X | | A | M | | L | P | | E |

Pass 2 : Merge each pair to obtain the list :

| A | E | M | X | | E | L | P |

Pass 3 : Again merge the two sub arrays to obtain the list :

| A | E | E | L | M | P | X |

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. i. Solve the recurrence $T(n) = 2T(n/2) + n^2 + 2n + 1$
  ii. Prove that worst case running time of any comparison sortis и ($n \log n$).
Ans. Refer Q. 1.9.

Q. 4. Explain shell sort with example.
Ans. Refer Q. 1.16.

Q. 5. Discuss quick sort method and analyze its complexity.
Ans. Refer Q. 1.19.

Q. 6. Explain the concept of merge sort with example.
Ans. Refer Q. 1.21.

Q. 7. Write short note on heap sort algorithm with its analysis.
Ans. Refer Q. 1.23.

Q. 8. Explain HEAP SORT on the array. Illustrate the operationHEAP SORT on the array $A = \{6, 14, 3, 25, 2, 10, 20, 7, 6\}$
Ans. Refer Q. 1.26.

☺☺☺