

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.

- Definiteness** : Each operation must be perfectly clear and unambiguous.
- 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.
- 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 :

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

Cases of complexity :

- 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.
- 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.
- 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

- 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.

- It is a line that stays within bounds.
- 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) :**

- This notation bounds a function within constant factors.
- 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 $c_1g(n)$ and $c_2g(n)$ inclusive.

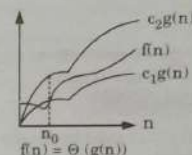


Fig. 1.3.1.

2. **O -Notation (Upper bound) :**

- Big-oh is formal method of expressing the upper bound of an algorithm's running time.
- It is the measure of the longest amount of time it could possibly take for the algorithm to complete.
- 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 $n \geq n_0$,

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

- 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 $f(n)$.

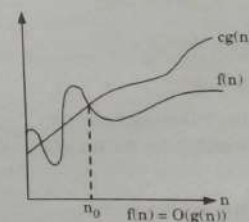


Fig. 1.3.2.

3. Ω -Notation (Lower bound) :

- This notation gives a lower bound for a function within a constant factor.
- 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 $cg(n)$.

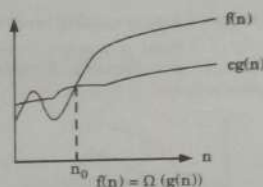


Fig. 1.3.3.

- 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.

$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ if a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \forall n \geq n_0\}$

- Little omega notation (ω)** : It is used to denote lower bound that is asymptotically tight.

$\omega(g(n)) = \{f(n) : \text{For any positive constant } c > 0, \text{ if a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \forall n \geq n_0\}$

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

Answer

$$\begin{aligned} \text{If } f(n) &= 100 \cdot 2^n + n^5 + n \\ \text{For } n^5 &\geq n \\ 100 \cdot 2^n + n^5 + n &\leq 100 \cdot 2^n + n^5 + n^5 \\ &\leq 100 \cdot 2^n + 2n^5 \end{aligned}$$

$$\begin{aligned} \text{For } 2^n &\geq n^5 \\ 100 \cdot 2^n + n^5 + n &\leq 100 \cdot 2^n + 2 \cdot 2^n \\ &\leq 102 \cdot 2^n \end{aligned}$$

$$\text{Thus, } f(n) = O(2^n)$$

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 :

- Let P be a probability of getting successful search and n is the total number of elements in the list.
- 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.
- The probability of getting unsuccessful search is $(1 - P)$.
- Average case time complexity $C_{\text{avg}}(n) = \text{Probability of successful search} + \text{Probability of unsuccessful search}$

$$= \left[1 \cdot \frac{P}{n} + 2 \cdot \frac{P}{n} + \dots + i \cdot \frac{P}{n} \right] + n(1 - P)$$

$$= \frac{P}{n} [1 + 2 + \dots + i] + n(1 - P)$$

$$= \frac{P \cdot n(n+1)}{2n} + n(1 - P) \quad (\because i = i++ \text{ till } i < n)$$

$$C_{\text{avg}}(n) = \frac{P(n+1)}{2} + n(1 - P)$$

- So, for unsuccessful search, $P = 0$

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

$$= n$$

- Thus the average case time complexity becomes equal to n for unsuccessful search.

- For successful search,

$$P = 1$$

$$C_{\text{avg}}(n) = \frac{1(n+1)}{2} + n(1 - 1) = \frac{n+1}{2} + 0$$

$$C_{\text{avg}}(n) = O\left(\frac{n}{2}\right)$$

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

Worst case time complexity :

- 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 :

$$C_{\text{worst}}(n) = O(n)$$

- 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 :

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

It follows that :

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

Example :

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

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

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

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

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

For $\epsilon = 1$, 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 :

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

and if it is also true that :

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

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 :

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

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

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

For $\epsilon = 1$ 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 :

$$af\left(\frac{n}{b}\right) \leq cf(n)$$

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 :

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

$$T(n) = aT\left(\frac{n}{b}\right) + n^c \quad \dots(1.7.2)$$

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

$$\begin{aligned} \text{we get,} \quad a &= 7 \\ b &= 2 \\ f(n) &= n^2 \end{aligned}$$

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

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

$$\begin{aligned} \text{Case 1:} \quad f(n) &= O(n^{\log_2 7 - \epsilon}) \\ &= O(n^{2.81 - \epsilon}) \\ &= O(n^{2.81 - 0.81}) \\ &= O(n^2) \end{aligned}$$

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

$$\begin{aligned} \text{Thus,} \quad T(n) &= \theta(n^{\log_2 7}) \\ &= \theta(n^{2.81}) \end{aligned}$$

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 $a = 16$

$$T(n) = 16T\left(\frac{n}{4}\right) + n^2$$

$$\begin{aligned} \Rightarrow \quad a &= 16 \\ b &= 4 \\ f(n) &= n^2 \end{aligned}$$

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

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

$$\begin{aligned} \Rightarrow \quad T(n) &= \theta(f(n)) \\ &= \theta(n^2) \end{aligned}$$

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

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

Let

$$\begin{aligned} m &= \log_2 n \\ n &= 2^m \\ n^{1/2} &= 2^{m/2} \end{aligned} \quad \dots(1.8.2)$$

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

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

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

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

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

Solution of equation (1.8.5) is given as

$$\begin{aligned} a &= 1, \quad b = 2, \quad f(n) = O(m) \\ f(n) &= O(m^{\log_2 1 + E}) \quad \text{where } E = 1 \\ T(n) = x(m) &= O(m) = O(\log n) \end{aligned}$$

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 $n! = n \cdot (n-1)!$

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

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

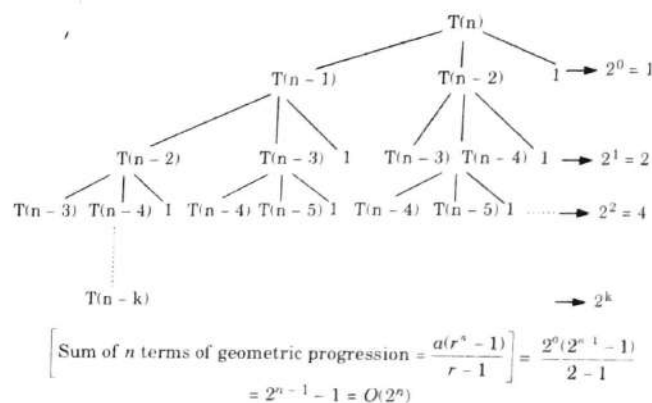
At k^{th} level, $T(1)$ will be equal to 1

when, $n - k = 1$

$$k = n - 1$$

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

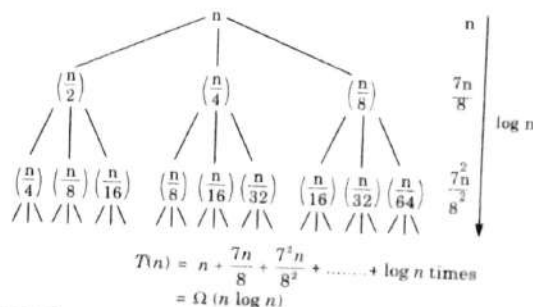
$$= 2^0 + 2^1 + 2^2 + \dots + 2^{n-1}$$



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

Que 1.13. Consider the recurrences

$$T(n) = 3T(n/3) + cn, \text{ and}$$

$T(n) = 5T(n/4) + n^2$ where c is constant and n is the number of inputs. Find the asymptotic bounds.

AKTU 2013-14, Marks 05

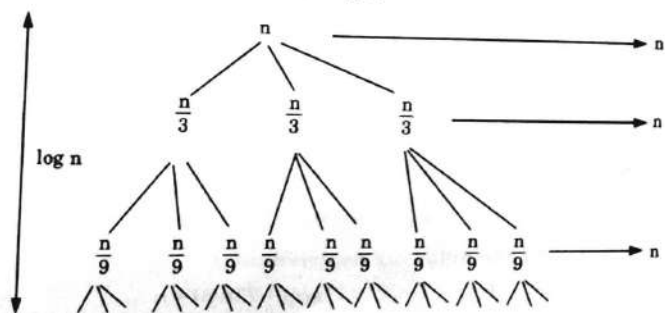
Answer

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

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

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

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



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

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

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

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.

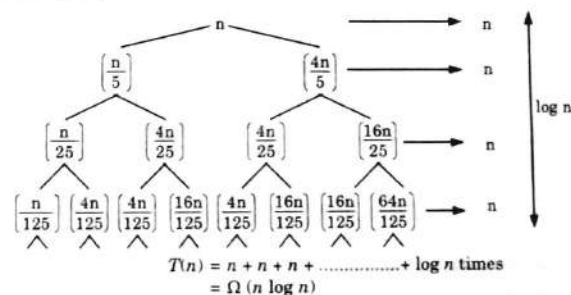
$$T(n) = \Theta(f(n))$$

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

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

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

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

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

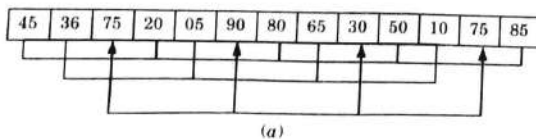
For example :

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	$a[2]$	$a[5]$	$a[8]$	$a[11]$	

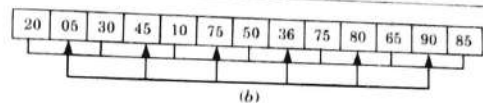
Input to pass 1 with distance = 3



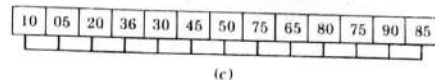
Output of pass 1 is input to pass 2 and distance = 2

1-16 B (CS/IT-Sem-5)

Introduction



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



Output of pass 3

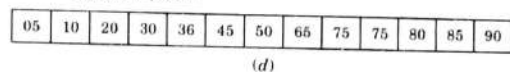


Fig. 1.15.1.

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

ii. Insertion_Sort(A) :

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

AKTU 2016-17, Marks 10

Answer

NSORT(A, B):

- ```

1 for $i = 1$ to n do
2 $j = \text{choice}(1 \dots n)$
3 if $B[j] \neq 0$ 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

```

its complexity with suitable example.

**Answer**

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

1. If  $p < r$  then
2.  $q \leftarrow \text{Partition}(A, p, r)$
3. Recursive call to **Quick\_Sort** ( $A, p, q - 1$ )
4. Recursive call to **Quick\_Sort** ( $A, q + 1, r$ )

Partition  $(A, p, r)$ 

1.  $x \leftarrow A[j]$
2.  $j \leftarrow p - 1$
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 $i + 1$

```

**Solution :** Given array to be sorted

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


  
 P

these element and swap it

Array after swapping

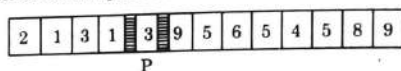
Underline      Overline

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.



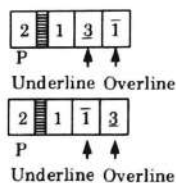
Now, Pivoting process is complete.

**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]

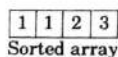
First apply Quick sort for subarray 1.



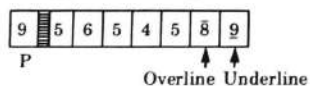
The pointers have crossed.

i.e., overline on left of underlined.

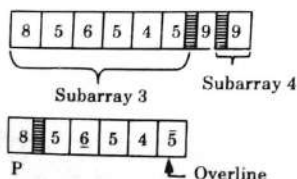
Swap pivot with overline



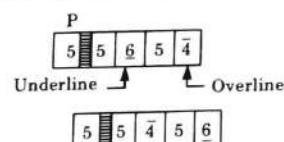
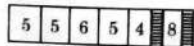
Now, for subarray 2 we apply Quick sort procedure.



The pointer has crossed. Then swap Pivot with overline.

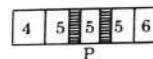


Swap overline with Pivot.

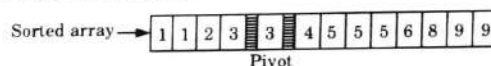


Overline on left of underlined.

Swap Pivot with overline.



Now combine all the subarrays



**Analysis of complexity :**

i. **Worst case :**

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

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

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

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

Substituting our assumption in equation (1.18.1) we get

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= \max_{0 \leq q \leq n-1} (q^2 + (n-q-1)^2) + \Theta(n) \end{aligned}$$

- 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  $0 \leq q \leq n-1$  at one of the endpoints.
- This gives the bound

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

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

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

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

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

- 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

$$T(n) = 1/n (T(1) + T(n-1) + \dots + T(n-q)) + \Theta(n) \quad \dots(1.18.1)$$

We know from worst-case analysis

$$\begin{aligned} T(1) &= \Theta(1) \text{ and } T(n-1) = O(n^2) \\ T(n) &= 1/n (\Theta(1) + O(n^2)) + 1/n \sum_{q=1}^{n-1} (T(q) + T(n-q)) + Q(n) \quad \dots(1.18.2) \\ &= 1/n \sum_{q=1}^{n-1} (T(q) + T(n-q)) + Q(n) \\ &= 1/n [2 \sum_{k=1}^{n-1} (T(k))] + \Theta(n) \\ &= 2/n \sum_{k=1}^{n-1} (T(k)) + \Theta(n) \quad \dots(1.18.3) \end{aligned}$$

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

$$\begin{aligned} T(n) &\geq \sum_{k=1}^{n-1} \Theta_{k=1}^{n-1} 2/n (ak \log k + b) + \Theta(n) \\ &= 2a/n \sum_{k=1}^{n-1} k \log k - 1/8(n^2) + 2b/n \\ &\quad (n-1) + \Theta(n) \quad \dots(1.18.4) \end{aligned}$$

At this point we are claiming that

$$\sum_{k=1}^{n-1} k \log k \leq 1/2 n^2 \log n - 1/8(n^2)$$

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

$$T(n) \leq 2a/n [1/2 n^2 \log n - 1/8(n^2)] + 2/n b(n-1) + \Theta(n) \quad \dots(1.18.5)$$

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

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 :

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

And from case (2) of master theorem

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

Worst case : Refer Q. 1.18, 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  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$
  4. for  $i = 1$  to  $n_1$   
do  
 $L[i] = A[p + i - 1]$

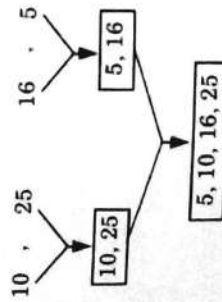


- endfor  
 5. for  $j = 1$  to  $n_2$   
   do  
    $R[j] = A[q + j]$   
   endfor  
 6.  $L[n_1 + 1] = \infty, R[n_2 + 1] = \infty$   
 7.  $i = 1, j = 1$   
 8. for  $k = p$  to  $r$   
   do  
   if  $L[i] \leq R[j]$   
   then  $A[k] \leftarrow L[i]$   
        $i = i + 1$   
   else  $A[k] = R[j]$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

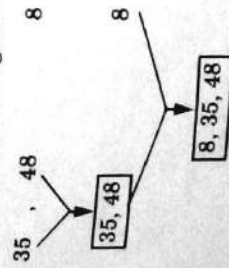
**Example:**

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



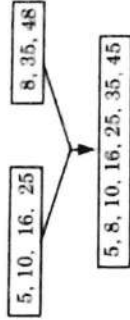
3. Consider the second half : 35, 48, 5 again divide into two sub-arrays



4. Merge these two sorted sub-arrays,

### 1-24 B (CS/IT-Sem-5)

#### Introduction



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 :

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

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

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

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 :

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

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

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

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

**HEAP-SORT( $A$ ) :**

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

**BUILD-MAX-HEAP ( $A$ )**

1. heap-size ( $A$ )  $\leftarrow \text{length}[A]$
2. for  $i \leftarrow (\text{length}[A]/2)$  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 ( $A, 3$ )

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

$A[l] > A[i]$

largest = 6

$A[r] > A[\text{largest}]$

$20 > 17$

largest = 7

largest  $\neq i$

$A[i] \leftrightarrow A[\text{largest}]$

$A[i] > A[l]$

$A[i] > A[r]$

Now call MAX HEAPIFY ( $A, 2$ )

$A[i] < A[l]$

so,

largest = 4

$A[\text{largest}] > A[r]$

Now,

$i \neq \text{largest}$ , so  $A[i] \leftrightarrow A[\text{largest}]$

$A[i] > A[l]$

$A[i] > A[r]$

We call MAX HEAPIFY ( $A, 1$ )

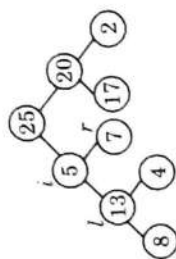
$A[i] < A[l]$

largest = 2

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

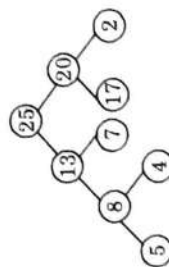
$A[i] \leftrightarrow A[\text{largest}]$

largest = 2, so  $i = 2$

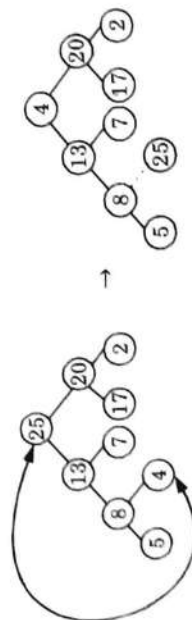


$A[i] < A[l]$ , then  $\text{largest} = l$   
 $A[\text{largest}] > A[r]$ ,  $\text{largest} = r$   
 $A[l] \leftrightarrow A[\text{largest}]$   
 Now,  
 $A[l] < A[l]$   
 $\text{largest} = 8, A[\text{largest}] > A[r]$   
 $\text{largest} = i, A[\text{largest}] \leftrightarrow A[l]$

so final tree after Build MAX HEAPIFY

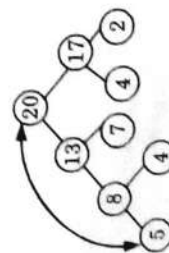


Now  $i = 9$  down to 2, exchange  $A[1]$  and  $A[9]$  and  $\text{size} = \text{size} - 1$  and call MAX HEAPIFY ( $A, 1$ ) each time.  
 exchanging  $A[1] \leftrightarrow A[9]$

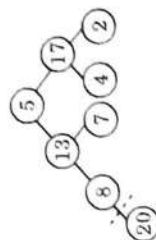


|   |    |    |   |   |    |   |   |    |
|---|----|----|---|---|----|---|---|----|
| 4 | 13 | 20 | 8 | 7 | 17 | 2 | 5 | 25 |
|---|----|----|---|---|----|---|---|----|

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

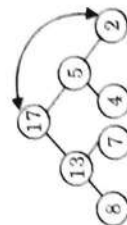


Now exchange  $A[1]$  and  $A[8]$  and  $\text{size} = \text{size} - 1 = 8 - 1 = 7$

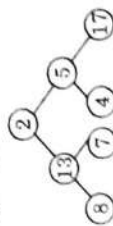


|   |    |    |   |   |   |   |    |
|---|----|----|---|---|---|---|----|
| 5 | 13 | 17 | 8 | 7 | 4 | 2 | 20 |
|---|----|----|---|---|---|---|----|

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

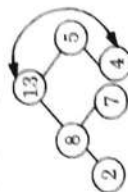


exchange  $A[1]$  and  $A[7]$  and  $\text{size} = \text{size} - 1 = 7 - 1 = 6$

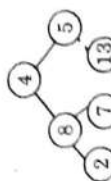


|   |    |   |   |   |   |    |
|---|----|---|---|---|---|----|
| 2 | 13 | 5 | 8 | 7 | 4 | 17 |
|---|----|---|---|---|---|----|

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

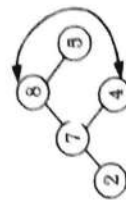


exchange  $A[1]$  and  $A[6]$  and now  $\text{size} = 6 - 1 = 5$

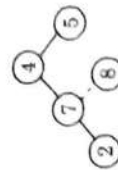


|   |   |   |   |   |    |
|---|---|---|---|---|----|
| 4 | 8 | 5 | 2 | 7 | 13 |
|---|---|---|---|---|----|

Again call MAX HEAPIFY ( $A, 1$ )



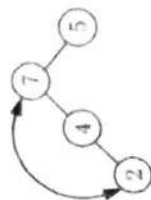
exchange  $A[1]$  and  $A[5]$  and now  $\text{size} = 5 - 1 = 4$



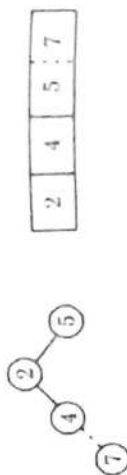
|   |   |   |   |   |
|---|---|---|---|---|
| 4 | 7 | 5 | 2 | 8 |
|---|---|---|---|---|

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

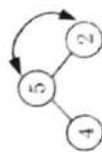




exchange  $A[1]$  and  $A[4]$  and size =  $4 - 1 = 3$



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 | 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**

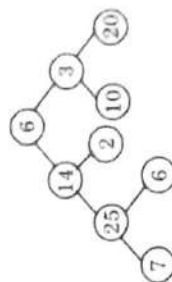
**Heap sort :** Refer Q. 1.22, Page 1-24B, Unit-1

**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 ( $A, 4$ )

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

$i \leftarrow \text{left}[4] = 8$

$r \leftarrow \text{right}[4] = 9$

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

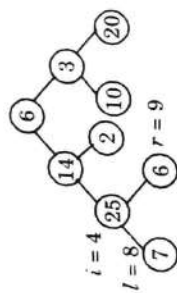
Then, largest  $\leftarrow 4$

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

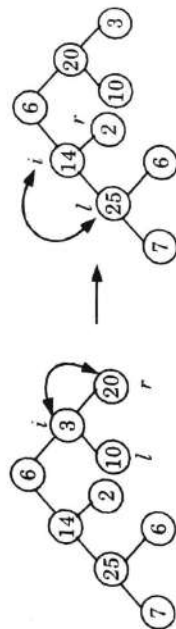
Then, largest = 4

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

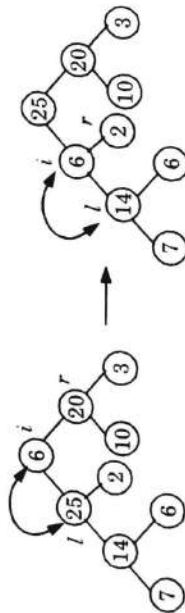
Now call MAX HEAPIFY ( $A, 2$ )



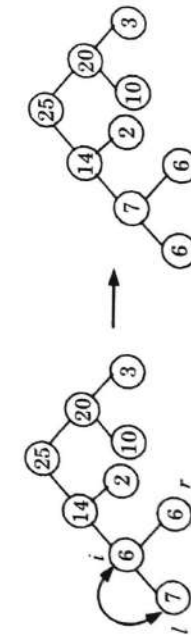
(i)

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

(ii)

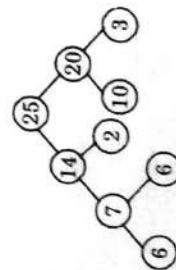


(iv)



(vi)

So final tree after BUILD-MAX HEAP is

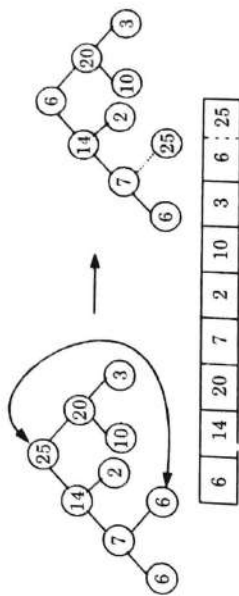


(viii)

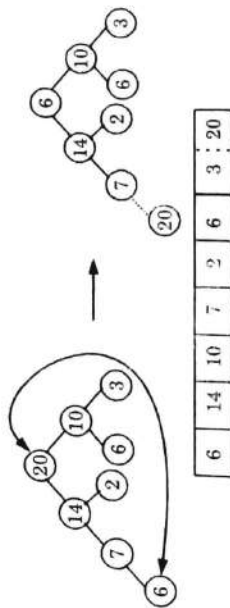
## I-32 B (CS/IT-Sem-5)

Introduction

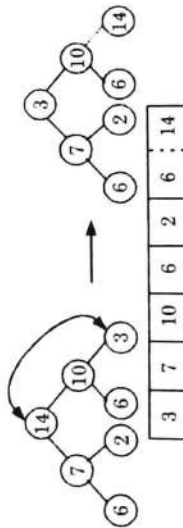
Now  $i = 9$  down to 2, and  $\text{size} = \text{size} - 1$  and call MAX HEAPIFY (A, 1) each time exchanging  $A[1] \leftrightarrow A[9]$



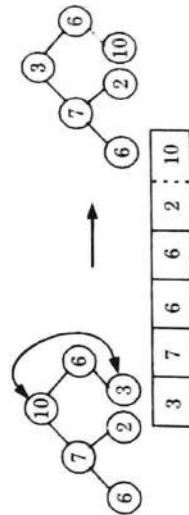
Now call MAX HEAPIFY (A, 1) we get

Now exchange  $A[1]$  and  $A[8]$  and  $\text{size} = 8 - 1 = 7$ 

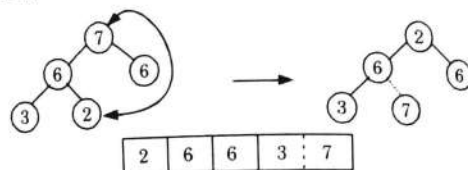
Again call MAX HEAPIFY (A, 1), we get

exchange  $A[1]$  and  $A[7]$  and  $\text{size} = 7 - 1 = 6$ 

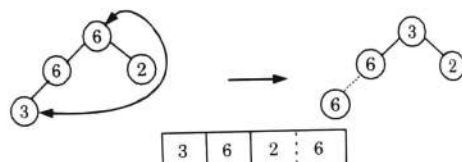
Again call MAX HEAPIFY (A, 1), we get

exchange  $A[1]$  and  $A[6]$  and now  $\text{size} = 6 - 1 = 5$ 

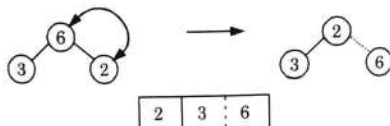
Again call MAX HEAPIFY (A, 1)  
exchange A [1] and A [5] and now size = 5 - 1 = 4



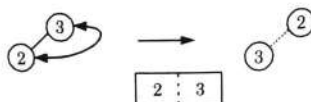
Again, call MAX HEAPIFY (A, 1)  
exchange A [1] and A [4] and size = 4 - 1 = 3



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 $O(n \cdot d)$ , 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  $\text{length}[A]$
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  $j \leftarrow \text{length}[A]$  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  $10$   $\therefore$  length  $[A] = 10$   
 $C[A[j]] \leftarrow C[A[j]] + 1$

For  $j = 1$

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

$C[1] \leftarrow 1$

For  $j = 2$

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

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

**Step 3:**

For  $i = 1$  to  $6$

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

For  $i = 1$

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

For  $i = 2$

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

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

**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 expect 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.  $n \leftarrow \text{length}[A]$
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 1.30.** 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. for  $i \leftarrow 1$  to  $d$  do

### 1-38 B (CS/IT-Sem-5)

Introduction

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 | 1 <sup>st</sup> pass | 2 <sup>nd</sup> pass | 3 <sup>rd</sup> 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.

