SPRING MID SEMESTER EXAMINATION-2017

Design & Analysis of Algorithms

[CS-3001]

Full Marks: 25 **Time: 2 Hours**

Answer any five questions including question No.1 which is compulsory.
The figures in the margin indicate full marks.
Candidates are required to give their answers in their own words as far as practicable and all parts of a question should be answered at one place only.

SOLUTION & EVALUATION SCHEME

Q1 Answer the following questions: (1 x 5)

a) Consider the following C function.

```
int FUN( int x, int n)
{
   if ( n==0)
       return 1;
   else if(n==1)
       return x;
   else if (n%2==0)
          return FUN( x * x, n/2);
   else
          return FUN( x * x, n/2 ) * x ;
}
```

i) Write down the recurrence T(n) for the above function **FUN(int x, int n)**.

ii) Derive the worst case Time Complexity of **FUN(int x, int n)**

Scheme

- Correct recurrence for T(n) – 0.5 Mark
- Worst case time complexity (only answer) – 0.5 Mark

Answer:

i) T(n) = T(n/2) + c, where c is constant.

ii) Worst case time complexity = O (log n)

b) Do as directed.

i) State true or false.

f(n) = O(f(n))

ii) Fill in the blanks.

_____ is the best case Time Complexity to find out the smallest element in a binary MAX-HEAP containing **n** numbers.

Scheme

- Only answer – each 0.5 Mark

Answer:

i) True

ii) O(n) (Smallest element can be found from n/2+1 to n in max-heap A[1..n]

c) What is the solution of the following recurrence?

$T(n) = 9T(n/3) + n^2 \log n$

Scheme

- Correct answer for T(n) : 1 Mark
- Incorrect answer, but some valid steps : 0.5 Mark

Answer:

$\Theta(n^2 \text{loglog } n)$

d) Consider a complete binary tree where the left and the right sub-trees of the root are MIN-HEAPs. What is upper bound Time Complexity to convert the tree into a MIN-HEAP?

Scheme

- Correct answer : 1 Mark
- Wrong answer : Zero Mark

Answer:

The upper bound Time Complexity = $O(\log_2 n)$

(In the worst case applying MIN-HEAPIFY(1) will terminate at leaf that visits the heights of the tree.)

e) Match the following Algorithms with its best case Time Complexity

Insertion Sort	$O(n^2)$
Quick-sort	O(log n)
Simple Bubble Sort	O(n)
Binary-Search	O(n log n)

Scheme

- Each Correct Matching – 0.25 Mark

Answer:

Insertion Sort	O(n)
Quick-sort	O(n log n)
Simple Bubble Sort	$O(n^2)$
Binary-Search	O(log n)

Q2 a) Describe the asymptotic efficiency of Algorithms with suitable examples. (2.5)

Scheme

- Explanation of asymptotic efficiency of Algorithms : 1.5 Mark
- Examples on asymptotic notation : 1 Mark

Answer:

- To analyze the efficiency of an algorithm, it is not necessary to conduct a detailed analysis of the running time. The asymptotic analysis is the method

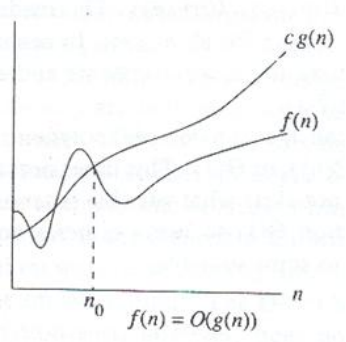
that efficiently describes the efficiency of an algorithm.

- The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared.
- Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.
    - The asymptotic run time of an algorithm gives a simple and machine independent, characterization of its complexity.
    - The notations works well to compare algorithm efficiencies because we want to say that the growth of effort of a given algorithm approximates the shape of a standard function.

- The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.
    i) O - Notation (Big-Oh Notation) : Asymptotic upper bound
    ii) Ω - Notation (Big-Omega Notation) : Asymptotic lower bound
    iii) Θ - Notation (Theta Notation) : Asymptotic tight bound

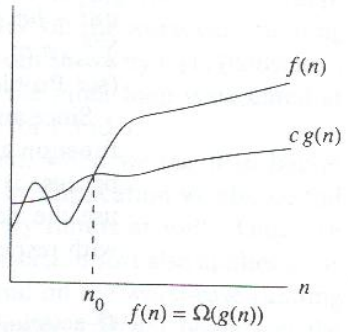
i) O - Notation (Big-Oh Notation)

- It represents the upper bound of the resources required to solve a problem.(worst case running time)
- **Definition:** Formally it is defined as
  For any two functions f(n) and g(n), which are non-negative for all $n \geq 0$, f(n) is said to be g(n), f(n) = O(g(n)), if there exists two positive constants c and n_0 such that
  $$0 \leq f(n) \leq c\, g(n) \text{ for all } n \geq n_0$$
- Less formally, this means that for all sufficiently big n, the running time of the algorithm is less than g(n) multiplied by some constant. For all values n to the right of n_0, the value of the function f(n) is on or below g(n).



ii) Ω - Notation (Big-Omega Notation)

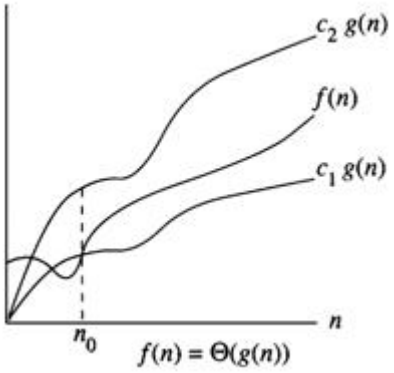
- **Definition:** For any two functions f(n) and g(n), which are non-negative for all $n \geq 0$, f(n) is said to be g(n), f(n) = Ω (g(n)), if there exists two positive constants c and n0 such that
  $$0 \leq cg(n) \leq f(n) \text{ for all } n \geq n0$$



iii) Θ - Notation (Theta Notation)

- **Definition:** For any two functions f(n) and g(n), which are non-negative for all n ≥ 0, f(n) is said to be g(n), f(n) = Θ (g(n)), if there exists positive constants c1, c2 and n0 such that

  $0 \leq c1g(n) \leq f(n) \leq c2g(n)$ for all $n \geq n0$



b) Solve the following recurrences. (2.5)

a) T(n) = 3T(n/2) + n where n>1 and T(1)=1

b) T(n) = 2T(n-1) + 1/n, T(0) = 1

Scheme

- Correct recurrence for T(n) – 0.5 Mark
- Worst case time complexity (only answer) – 0.5 Mark

Answer:

a) $T(n)=O(n^{\log_2 3})=O(n^{1.5})= O(n^2)$

Explanation

Given recurrence T(n) = 3T(n/2) + n. Solving this recurrence by master theorem.

Step-1: Here, a = 3, b = 2, f(n) = n

$n^{\log_b a} = n^{\log_2 3} = n^{1.5}$

Comparing $n^{\log_b a}$ with f(n), $n^{\log_b a}$ is found asymptotically larger than f(n), so we guess case-1 of master theorem.

Step-2: Confirming the guess

As per case-1, if $f(n)=O(n^{\log_b a-\epsilon})$, then $T(n)=O(n^{\log_b a})$

Let $f(n)=O(n^{\log_b a-\epsilon})$ is true

⇨ $f(n) \le c\, n^{\log_b a-\epsilon}$

⇨ $n \le c\, n^{1.5-\epsilon}$

Taking value of $\epsilon=0.5$ and $c=1$, the above inequality is valid, so the assumption is true.

Now, as per the case-2 the solution of recurrence is

$T(n)=O(n^{\log_b a})=O(n^{\log_2 3})=O(n^{1.5})= O(n^2)$

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

Explanation

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

$= 2\{2T(n-2) +1/(n-1)) +1/n$

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

$= 2^2\{2T(n-3) +1/(n-2)\} +2/(n-1) +1/n$

$= 2^3T(n-3) +2^2/(n-2) +2/(n-1) +1/n$

…….

…….

$=2^iT(n-i) +\{2^{i-1}/(n-i+1) + 2^{i-2}/(n-i+2) + \ldots\ldots +2^2/(n-2) +2/(n-1) +1/n\}$

Now $n-i=0 \Rightarrow i=n$

$=2^nT(0) +\{2^{n-1}/1 + 2^{n-2}/2 + \ldots\ldots +2^2/(n-2) +2/(n-1) +1/n\}$

$=2^n +\{2^{n-1} +$ Some value around $2^{n-1}\}$

$T(n) \le c2^n$

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

Q3 a) Given an unsorted array A[1..n], where odd indexed elements are sorted in ascending order and the even indexed elements are sorted in descending order. Design an Algorithm to sort the array in O(n) worst-case time, in ascending order. (2.5)

Scheme

- Correct algorithm with O(n) worst-case time : 2.5 Marks
- Correct algorithm with other than O(n) worst-case time : 1.5 Marks
- Algorithm approaches to answer (not fully correct) : 0.5-1.5 Marks

Answer:

```
ARRAY-MERGE(A,n)
{
  k←1, l←1
  for i←1 to n
  {
     if(i%2!=0)
        O[k++]←A[i]
```

```
        else
            E[l++]←A[i]
    }
    k←k-1, l←l-1
    /*now k is the number of odd numbers stored in array O and l is the number of even numbers stored in  array E*/
    MERGE(O, k, E, l, A); //Merge sorted array O, E to A
}

/*Merge procedure to merge sorted array A and B to C. A[1..m] sorted in ascending order, B[1…n] sorted in descending order, the C[1..m+n] is required to sort in ascending order*/
MERGE(A, m, B, n, C)
{
  i←1, j←n, k←1;
  while(i≤m and j≥1)
  {
     if(A[i]<B[j])
     {
        C[k]=A[i]
        k←k+1
        i←i+1
     }
     else
     {
       C[k]=B[j]
       k←k+1
       j←j-1
     }
  }
  while(i≤m)
  {
        C[k]←A[i]
        k←k+1
        i←i+1
 }
  while(j≥1)
  {
        C[k]←B[j]
        k←k+1
```

```
        j←j-1
    }
}
```

b) Write the algorithm to build a MAX-HEAP? Describe your Algorithm to build a MAX-HEAP from the array A = {5, 7, 8, 2, 1, 0, 3, 9, 4, 5, 6} in a step by step process. Derive the time complexity of building a MAX-HEAP. (2.5)

Scheme

- BUILD-MAX-HEAP algorithm : 1 Mark
- Derivation of Time Complexity : 0.5 Mark
- Description of algorithm step by step on given example : 1 Mark

Answer:

```
BUILD-MAX-HEAP(A, n)
{
   for i ← n/2 down to 1
       MAX-HEAPIFY(A, i, n)
}
```

```
MAX-HEAPIFY(A, i, n)
{
    l ← LEFT(i )
   r ← RIGHT(i )
   if l ≤ n and A[l] > A[i ]
         largest ←l
   else largest ←i
   if r ≤ n and A[r ] > A[largest]
        largest ←r
   if largest ≠ i
   {
      exchange A[i ] ↔ A[largest]
     MAX-HEAPIFY(A, largest, n)
   }
}
```

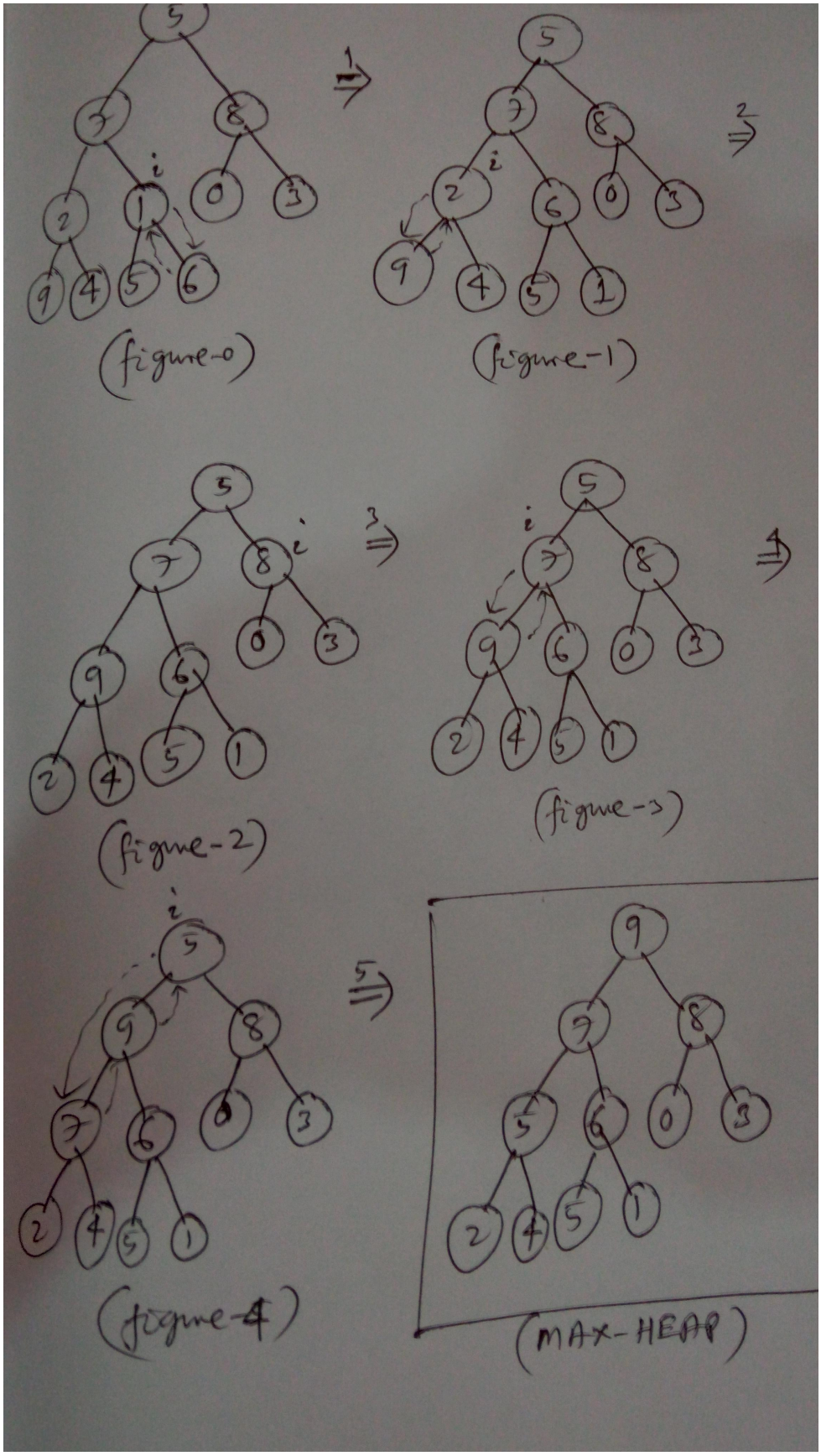
Time Complexity of BUILD-MAX-HEAP

Simple bound: O(n) calls to MAX-HEAPIFY, each of which takes O(log n) time ⇒ O(n log n).

Tighter analysis: T(n) = O(n)

Description of algorithm step by step on given array A = {5, 7, 8, 2, 1, 0, 3, 9, 4, 5, 6} to build a MAX-HEAP

i denotes the index of the node where MAX-HEAPIFY(i) is applied.


(figure-0)
(figure-1)
(figure-2)
(figure-3)
(figure-4)
(MAX-HEAP)

Q4 a) Write the recursive Binary-Search Algorithm. Find out its recurrence and discuss its best case and worst case time complexities. Also identify the lines of the Algorithms that support each part of Divide, Conquer and Combine paradigm respectively. (2.5)

Scheme

- Correct Recursive Binary Search algorithm : 1 Mark
- Writing Recurrence for this algorithm : 0.5 Mark
- Finding best & worst case time complexity : 0.5 Mark
- Identification of lines….: 0.5 Mark

Answer

```
REC-BINARY-SEARCH(A, lb, ub, key)
{
  if (lb≤ub)
  {
     mid←(lb+ub)/2;          ---------------------------- (1)
     if(key==A[mid])
        return mid;       //successful search
     else if(key<A[mid]
          REC-BINARY-SEARCH(A, lb, mid-1, key);    ----------(2)
     else
          REC-BINARY-SEARCH(A, mid+1, ub, key);  -----------(3)
  }
  return -1;   //Indication of unsuccessful search
}
```

Time complexity of this algorithm (in terms of recurrence)

Let T(n) is the time complexity for REC-BINARY-SEARCH(A, 1, n, key)

Best case: If the element to be searched is found in the mid position of the array then it will take O(1) time.

Worst case: If the element to be searched is found in the last attempt or may not found, the its time complexity can be represented as $T(n) = T(n/2) + 1$

Solving this recurrence we will get worst case time complexity=O(log n)

Identification of Line Numbers

Line number marked as (1) is the divide part, line number (2) and (3) are conquer parts, there is no combine part of Divide, Conquer and Combine paradigm respectively for binary search.

b) Write the PARTITION() Algorithm of Quick-Sort. Describe in a step by step (2.5)

process to get the pass1 result of PARTITION() by taking last element as pivot on the following array elements.

13, 12, 10, 18, 14, 20, 11, 18, 14, 15, 15

Describe the Time complexity of PARTITION() Algorithm.

Scheme

- PARTITION Algorithm: 1 Mark
- Description of Time Complexity : 0.5 Mark
- Representation of intermediate steps of pass-1 : 1 Mark

Answer:

PARTITION Algorithm:

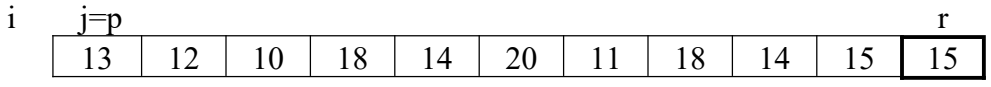
```
PARTITION(A, p, r)
{
    x ← A[r]
    i ← p-1
    for j ← p to r-1
    {
       if A[j] ≤ x
       {
           i ← i+1
           A[i] ↔A[j]
       }
    }
    i ← i+1
    A[i] ↔A[r]
    return i
}
```

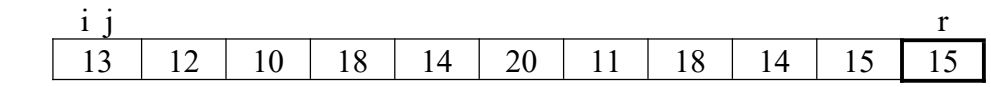
Description of Time Complexity

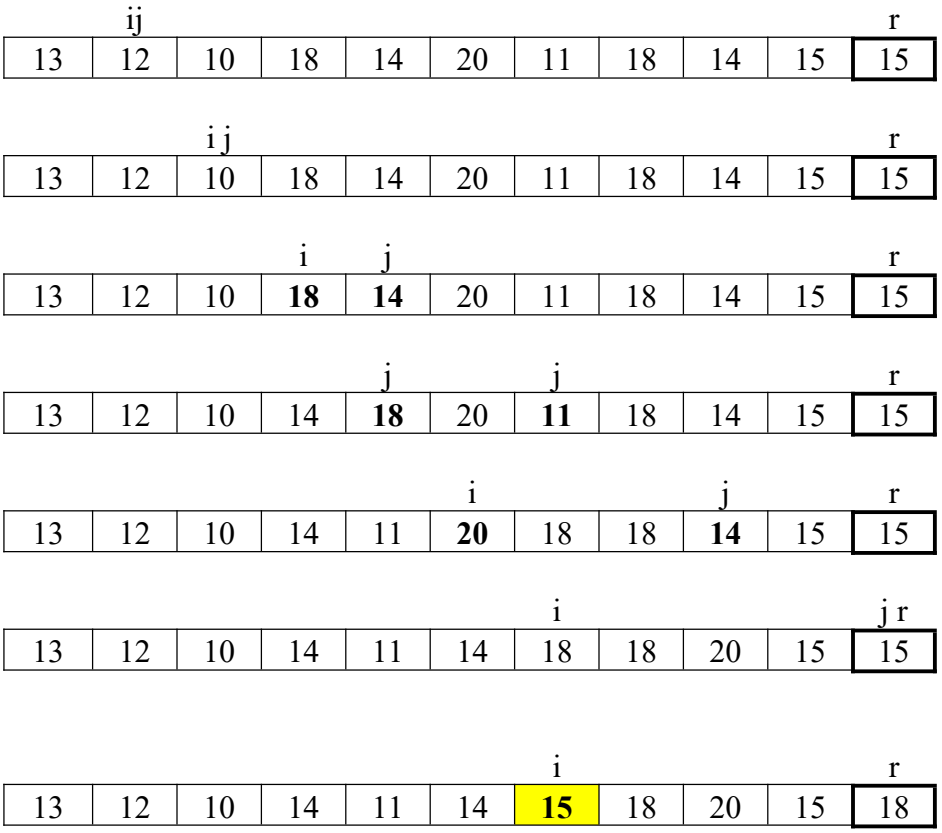
If p=1 and r=n and T(n) is the time complexity for PARTITION(A, 1, n), then T(n) can be described as T(n) = O(n)

Representation of intermediate steps of pass-1

Given Data: 13, 12, 10, 18, 14, 20, 11, 18, 14, 15, 15







Q5 a) Write an Algorithm MAX-MIN(A, max, min) to find out the maximum and minimum elements of an array A by using Divide-and-Conquer strategy. Compare this with the STRAIGHT-MAX-MIN(A, max, min) Algorithm in terms of number of comparisions. (2.5)

Scheme

- Correct algorithm : 1.5 Mark
- Comparison : 1 Mark

Answer:

/* A[1..n] is a array of n elements. Parameters p and r are integers, represents lower bound and upper bound of the array/subarray $1 \le p \le r \le n$.

```
STRAIGHT-MAX-MIN(A, p, r, max, min)
{
   max := min := a[p];
   for i := p+1 to r
   {
       if(a[i] > max) then max := a[i];
       if(a[i] < min) then min := a[i];
   }
```

```
}
```

/* A[1..n] is a array of n elements. Parameters p and r are integers, represents lower bound and upper bound of the array/subarray 1≤p≤r≤n.

```
DIVIDE-MAX-MIN(A, p, r, max, min)
{
    if (p==r) then max := min := a[p]; // if array contains one element
    else if (p==r-1) then // if array contains two elements
    {
        if (a[p] < a[r]) then max := a[r]; min := a[p];
       else max := a[p]; min := a[r];
    }
    else // if array contains more than two elements
    {
         //Divide into two subproblems
          q ← (p + r )/2;
         // Solve the sub-problems.
         MAX-MIN( A, p, q, max, min );
         MAX-MIN( q+1,  r, max1, min1 );
         // Combine the solutions.
         if (max1 > max) then max := max1;
         if (min1 < min) then min := min1;
    }
}
```

Time Complexity of Divide-Conquer Max-Min

If T(n) represents the time complexity, then the resulting recurrence relation is

$$T(n) = \begin{cases} 0 & n=1 \\ 1 & n=2 \\ 2T(n/2) + 2 & n>2 \end{cases}$$

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

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

$= 2^2T(n/2^2) + 2^2 + 2$

$= 2^2\{2T(n/2^3) + 2\} + 2^2 + 2$

$= 2^3T(n/2^3) + 2^3 + 2^2 + 2$

.....

.....

$= 2^{k-1}T(n/2^{k-1}) + (2^{k-1} + 2^{k-2} + 2^{k-3} + ...+ 2^3 + 2^2 + 2^1)$

If $n=2^k$, then

$= 2^{k-1}T(2) + 2(2^{k-2} + 2^{k-3} + ...+ 2^2 + 2^1 + 2^0)$

$= 2^{k-1} + 2 (2^{k-2} + 2^{k-3} + ...+ 2^2 + 2^1 + 2^0)$

$= 2^{k-1} + 2\sum_{i=0}^{k-2} 2^i$

$= 2^{k-1} + 2(2^{k-1} - 1)$

$= 2^{k-1} + 2^k - 2$

$= 2^k/2 + 2^k - 2 = n/2 + n - 2 = 3n/2 - 2 = O(n)$

3n/2 – 2 is the best, average, worst case number of comparison when n is a power of two

Comparisons with Straight Forward Method

. The straight max-min algorithm requires 2n-2 element comparisons in the best, average, and worst cases. Compared with the 2n – 2 comparisons for the Straight Forward method, Divide & Conquer max-min algorithm is a saving of 25% in comparisons.

b) Write the Algorithm for the procedure HEAP-EXTRACT-MAX(A), where the procedure removes and returns the element of MAX-HEAP with largest key. Illustrate the operation of HEAP-EXTRACT-MAX on the Heap A={1, 3, 2, 6, 4, 5, 7, 8, 9} (2.5)

Scheme

- HEAP-EXTRACT-MAX(A) algorithm : 1 Mark
- Explanation of the operation of this algorithm through given example: 1.5 Mark

Answer:

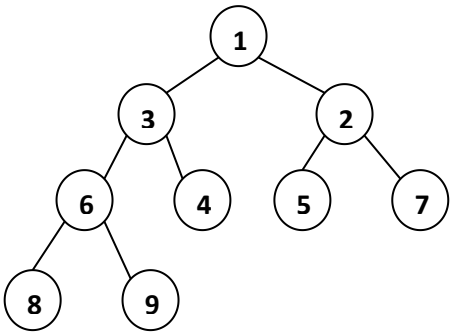
/*Algorithm to remove and returns the largest element of max-heap A*/

```
HEAP-EXTRACT-MAX(A, n)
{
    if n < 1
       then error-heap-underflow"
    max ← A[1]
    A[1] ← A[n]
    MAX-HEAPIFY(A, 1, n – 1)    //remakes heap
    return max
}
```

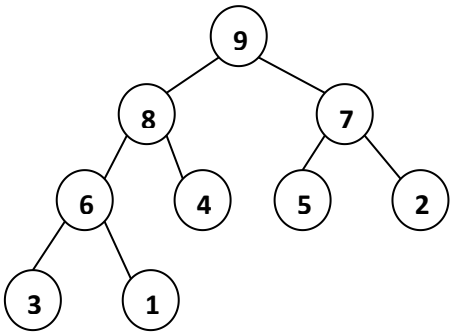
Illustrate of operation of HEAP-EXTRACT-MAX on the Heap A={1, 3, 2, 6, 4, 5, 7, 8, 9}

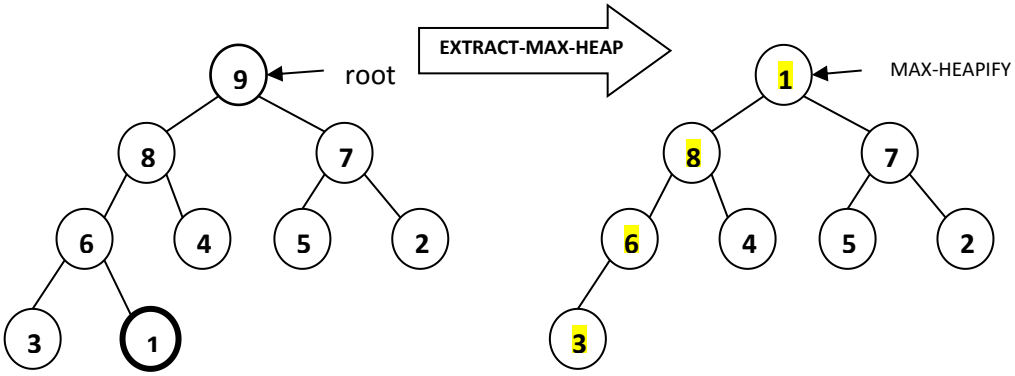
- The given heap is not a max-heap, it is a min-heap.
- First convert the given min-heap to max-heap, then apply the above algorithm.
- Method: Apply MAX-HEAPIFY(A, i, n) for i=n/2 down to 1.

Given Min-Heap

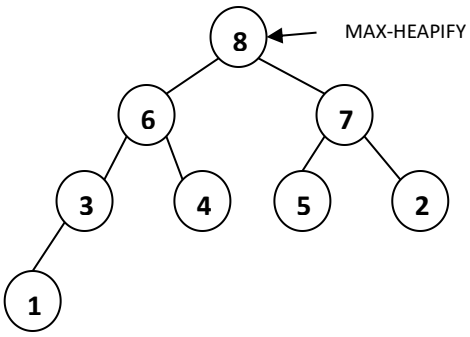


Max-Heap after conversion





(Now re-building the heap)



After the operation of HEAP-EXTRACT-MAX on the Heap A={1, 3, 2, 6, 4, 5, 7, 8, 9}, the algorithm will return the largest value as 9 and the array (max-heap) becomes A={8, 6, 7, 3, 4, 5, 2, 1}

(Final heap after the operation of EXTRACT-MAX-HEAP)

Q6 a) Given a set S of **n** integers and another integer **x**, determine whether or not there exist two elements in S whose sum is exactly **x**. Describe a Θ(nlogn) time Algorithm for the above problem. (2.5)

Scheme

- Correct Algorithm with Θ(nlogn) time complexity : 2.5 Mark
- Correct Algorithm without Θ(nlogn) time complexity :1.5 Mark

- Algorithm approaches to answer (partially correct) : 0.5-1.5 Marks

Answer:

/* Input: An array A and a value x. Output: A boolean value indicating if there is two elements in A whose sum is x.*/

```
CHECKSUMS(A, n, x)
{
   MERGE-SORT(A,n);   ---------- O(n log n)
   lb←i, ub←n;
   while(lb<ub)                                  ┐
   {                                             │
        if (A[lb] + A[ub] == x)                  │
            return TRUE;                         │
       else if (A[lb] + A[ub] < x )              │  O(n)
                lb←lb+1;                         │
        else    ub←ub-1;                         │
   }                                             ┘
   return FALSE;
}
```

Overall time Complexity

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

b) Explain the Master Theorem with suitable examples. (2.5)

Scheme

- Defining master theorem : 1 Mark
- Examples on master theorem : 1.5 Mark

Answer:

- The Master Theorem applies to recurrences of the following form:
  $T(n) = aT(n/b) + f(n)$
  where $a \geq 1$ and $b > 1$ are constants and f(n) is an asymptotically positive function. T(n) is defined on the non-negative integers by the recurrence.
  T(n) can be bounded asymptotically as follows: There are 3 cases:
  a) Case-1: If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$,
  then $\mathbf{T(n) = \Theta(n^{\log_b a})}$
  b) Case- 2: If $f(n) = \Theta(n^{\log_b a})$, then $\mathbf{T(n) = \Theta(n^{\log_b a} \log n)}$
  c) Case-3: If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, and : $af(n/b) \leq cf(n)$, then $T(n) = \Theta(f(n))$, for some constant $c < 1$ and all sufficiently large n, then $\mathbf{T(n) = \Theta(f(n))}$

- **Examples:**
- **Example-1 (Case-1):**
  Solve the the recurrence $T(n)=9T(n/3)+n$
  Sol^n: For this recurrence, we have a=9, b=3, f(n)=n, and thus we have that $n^{\log_3 9}= n^2$ which is asymptotically larger than f(n), we guess case-1. Since $f(n)= O(n^{\log_3 9-\epsilon})$, where $\epsilon=1$, we can apply case-1 of the master theorem and conclude that the solution is $T(n)= \Theta(n^2)$
- **Example-2 (Case-2)**
  Solve the recurrence $T(n)=T(2n/3)+1$
  For this recurrence, we have a=1, b=3/2, f(n)=1, and thus we have that $n^{\log_{3/2} 1}= n^0=1$ which is same as f(n). So case-2 applies, and thus the solution to the recurrence is $T(n)= \Theta(\log n)$
- **Example-3(Case-3)**
  Solve the the recurrence $T(n)=3T(n/4)+n \log n$
  For this recurrence, we have a=3, b=4, f(n)=n log n, and thus we have that $n^{\log_4 3}= n^{0.793}=1$. Comparing $n^{\log_b a}$ with f(n), f(n) is seems to be larger. So case-3 may be applied. Since (n)= $O(n^{\log_4 3+\epsilon})$, where $\epsilon=0.2$, Also $af(n/b)=3(n/4)\log(n/4)\leq(3/4)n \log n = cf(n)$ for c=3/4. So we conclude it is case-3 and thus the solution to the recurrence is $T(n)= \Theta(n \log n)$
- **Example-4**
  The master theorem does not apply to the recurrence $T(n)=2T(n/2) + n \log n$ even though it has the proper form : a=2, b=2, f(n)=n log n, and $n^{\log_b a}=n$. It might seem that case-3 should apply. Since f(n)=n log n is asymptotically larger than $n^{\log_b a}=n$. The problem is that it is not polynomially lager. The ratio $f(n)/n^{\log_b a}=n \log n/n=\log n$ is asymptotically less than n^{ϵ} for any positive constant ϵ.

==========XXXXXXXX==========