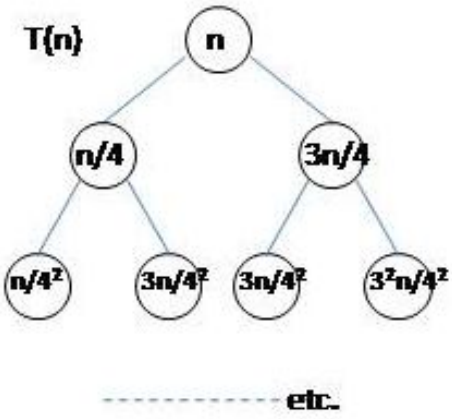
ESO207: Data Structures and Algorithms	*Time : 90min.*
Quiz: 1	*Max. Marks: 100*

Problem 1.

a. Derive the solution to the recurrence relation using the recursion tree method.

$$T(n) = T(n/4) + T(3n/4) + \Theta(n)$$

State any assumptions that you have made in deriving the answer. (10)



The unrolling of the recursion tree follows as in the above figure. However, the unrolled tree is asymmetric. Up to and including $\log_4 n$ levels of unrolling , the level wise sum of the costs of the nodes are n each. After $\log_4 n$ steps, the leftmost node will contribute $n/4^{\log n} = 1$. So from $\log_4 n + 1$ steps, till $\log_{4/3} n$ steps, the nodes from the left will stop contributing, and the level wise sum of each level is less than n. After $\log_{4/3} n$ steps, the recursion halts, as all problem sizes have become 1 or 0. Hence, $T(n) \leq n \log_{4/3} n$ and $T(n) \geq n \log_4 n$. Hence $T(n) = \Theta(n \log n)$. We have assumed that n is a power of 4, although the answer really does not depend on it.

b. Draw the array $[25, 17, 20, 14, 17, 18, 20, 1, 2, 3, 4, 5, 6, 7]$ as a heap structure (binary tree). Is it a max-heap? Argue your answer. (5)

The tree representation of the heap is shown in Figure 1. As we see, the max-heap property is satisfied at all nodes. Hence it is a max-heap.

Problem 2. Given an array $A[1, 2, \ldots, n]$ of *distinct* numbers that is sorted in ascending order, design an $O(\log n)$ time divide-and-conquer algorithm to determine whether there is an index satisfying $A[i] = i$. Return one such index if it exists. Write the pseudo-code, argue the correctness and give an analysis of time complexity. $(8 + 9 + 8 = 25)$

Let i be any index. If $A[i] = i$, then we can return i. Otherwise, there are two cases, $A[i] > i$. For any $k = i + 1, \ldots, n$, $A[i + k] > A[i] + k$ (since the numbers are distinct), which is $> i + k$. Hence, any possibility of a j such that $A[j] = j$ may occur only in $A[1, \ldots, i - 1]$. Analogously, if $A[i] < i$, then any possibility of a j such that $A[j] = j$ may occur only in $A[i + 1, \ldots, n]$. Choose i to be the mid-point of the array segment in order to obtain nearly equal sized partitions. The algorithm is given below.

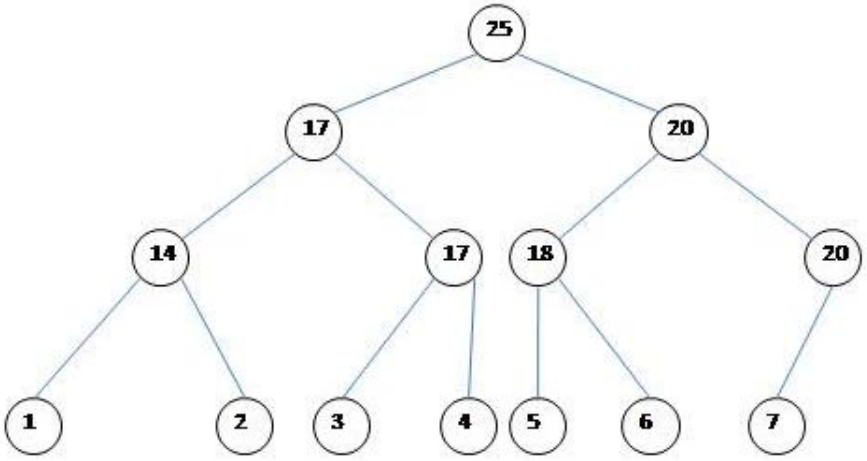


Figure 1: Heap Question 1 (b.)

```
FIXPOINT(A, p, r)
1.   if p > r return "No Fixpoint"
2.   else
3.        i = ⌈(p + r)/2⌉
4.        if A[i] == i
5.             return i
6.        elseif A[i] > i
7.             FIXPOINT(A, p, i − 1)
8.        else FIXPOINT (A, i + 1, r)
```

In each recursive step, a constant number of operations are performed and either the function terminates or the range $r - p + 1$ is reduced by half. So, the recurrence equation is $T(n) \leq T(\lfloor n/2 \rfloor) + \Theta(1)$, whose solution is $T(n) = O(\log n)$. Worst case is $\Theta(\log n)$ if the item is not found in the array.

Problem 3. The following pseudo-code claims to build a heap out of the array segment $A[1, \ldots, n]$ consisting of n numbers. Here MAX_HEAPIFY(A, i, n) assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max heaps, assuming heapsize is n. MAX_HEAPIFY(A, i, n) ensures that the binary tree rooted at i is a max heap. (25)

```
BUILD_MAX_HEAP(A, i, n)
// attempts to create a max heap rooted at node i in the array A[1, . . . , n].
1.   if LEFT(i) > n return ; // if i is a leaf node, then it is a 1-element heap
2.   else
3.        BUILD_MAX_HEAP(A, LEFT(i), n)
4.        BUILD_MAX_HEAP(A, RIGHT(i), n)
5.        MAX_HEAPIFY(A, i, n)
```

The top level call is BUILD_MAX_HEAP$(A, 1, n)$.

a Argue that a call to the BUILD_MAX_HEAP$(A, 1, n)$ procedure given above correctly builds a heap out of $A[1, \ldots, n]$? (8)

b Derive a recurrence relation for the time complexity of the BUILD_MAX_HEAP$(A, 1, n)$ procedure in $O(\cdot)$ given above. (8)

c Solve the recurrence relation to obtain the $O(\cdot)$ time complexity of the BUILD_MAX_HEAP$(A, 1, n)$ procedure given above. (9)

Assume that for all nodes at a height h in the heap, the procedure BUILD_MAX_HEAP correctly builds a heap. The base case is when $h = 0$, which corresponds to leaf nodes, and holds iff LEFT$(i) > n$. Otherwise, at a node i of height $h \geq 1$, first, BUILD_MAX_HEAP is called on each of the left and right children of i, and by the induction hypothesis, they now satisfy the max-heap property. Thereafter, Max_Heapify can now be applied as its pre-conditions are met, and it transforms the tree rooted at i to a max-heap.

For a node at height h, MAX_HEAPIFY takes $O(h)$ time. In general, for a heap of n nodes, the height of the root is at most $\log n$. Let n_L be the number of nodes in the left subtree and n_R denote the number of nodes in the right sub-tree. This gives the recurrence relation $T(n) = T(n_L) + T(n_R) + O(\log n)$. Now $n_L + n_R = n - 1$ and $n_L \geq n_R$ and $n_L \leq \lfloor 2n/3 \rfloor$ and $n_R \leq \lfloor n/3 \rfloor$. So, the worst case can be written as $T(n) = T(2n/3) + T(n/3) + O(\log n)$.

The solution is $T(n) = O(n)$. Let the recurrence be $T(n) = T(n/3) + T(2n/3) + d \log n$. By substitution, let $T(n) \leq cn - d \log n - e$. Then,

$$\begin{aligned} & T(n/3) + T(2n/3) + d \log n \\ &= c(n/3) - d \log(n/3) - e + c(2n/3) + d \log(2n/3) - e + d \log n \\ &= cn - 2d \log n - 2e + d \log(9/2) + d \log n \\ &= cn - d \log n + d \log(9/2) - 2e \end{aligned}$$

Choose e so that $d \log(9/2) - 2e \leq -e$, or $e \geq d \log(9/2)$. Hence, $T(n) \leq cn + d \log n + e$, for some constants, c, d, e, and hence, $T(n) = O(n)$.

Problem 4. Multiplying long integers. (35)

The problem is to multiply two n-bit numbers x and y, where n is very long, and does not fit within a word or two of a given computer. Dividing the problem, we can split x and y each into two halves as follows.

$$x = \boxed{x_L}\boxed{x_R} = 2^{n/2}x_L + x_R$$
$$y = \boxed{y_L}\boxed{y_R} = 2^{n/2}y_L + y_R$$

Now

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R \quad (1)$$

We can compute $x_L y_L$, $x_R y_R$, $x_L y_R$ and $x_R y_L$ as products of $n/2$-bit numbers. The product xy is then the addition and/or subtraction of suitable n-bit numbers. Assume that addition/subtraction of n-bit numbers can be done in $O(n)$ time. Note that multiplying an n-bit number by 2^k is just left shifting by k bits, and can be done in $O(n)$ time on most hardware.

a. Design a divide-and-conquer method (write pseudo-code) for multiplying two n-bit integers using the method outlined above. Argue correctness. (7)

b. Write the recurrence relation for the time complexity of this method and solve it. (8)

Soln. . The algorithm computes four products of $n/2$-bit numbers, $x_L y_L$, $x_L y_R$, $x_R y_L$ and $x_R y_R$ and then uses the equation Eqn. (1) to combine them to obtain xy. The number of operations satisfies the recurrence relation $T(n) = 4T(n/2) + \Theta(n)$, whose solution is $T(n) = \Theta(n^2)$.

The identity $bc + ad = (a+b)(c+d) - ac - ad$ allows one design an improved divide and conquer method. Using the identity, the middle term of the *RHS* of Eqn. (1) $(x_L y_R + x_R y_L)$ can be computed instead as $(x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$. Thus, the product xy can be computed using the multiplication of just *three* $n/2$ bit numbers (actually $n/2 + 1$-bit numbers), namely, $x_L y_L$, $x_R y_R$ and $(x_L + x_R)(y_L + y_R)$, followed by addition, substraction and left shifting of some n-bit numbers.

a. Write pseudo-code for the efficient divide and conquer algorithm outlined above. Argue correctness. (8)

b. Derive a recurrence relation for the number of operations required to compute the multiplication of two n-bit numbers (assume n is large). (8)

c. Solve the recurrence relation. State any assumptions that you make, if any. (4)

Soln. The pseudocode is essentially described above, there are three multiplications of $n/2 + 1$-bit numbers, $t_1 = x_L y_L$, $t_2 = x_R y_R$ and $t_3 = (x_L + x_R)(y_L + y_R)$. Then,

$$xy = 2^n t_1 + 2^{n/2}(t_3 - t_1 - t_2) + t_2$$

So the recurrence equation is $T(n) = 3T(n/2) + \Theta(n)$, whose solution is $n^{\log_2 3} \approx n^{1.51}$.

(*Note:* For the second part, if you have an even better method than the improved divide and conquer algorithm outlined above, you can explain the same, along with a correctness argument and time analysis.