HOMEWORK 2 – ANALYSIS OF ALGORITHMS

DATHRIKA SAICHARITHA

02010404

REFERENCE TEXTBOOK: Cormen, Thomas H. - Introduction to Algorithms (3rd ed.)

1. Exercise 6.5-8: The operation HEAP-DELETE (A, i) deletes the item in node i from heap A. Give an implementation of HEAP-DELETE that runs in O(log n) time for an n-element max-heap.

   (Binary) Heap must be a nearly complete binary tree. Tree is completely filled on all levels except lowest, which is filled from left to right.

   There are two kinds of binary heaps: max-heaps and min-heaps.

   In a max-heap, the max-heap property is that for every node i other than the root, A[PARENT(I)]>A[i].

   IMPLEMENTATION:

   - Use an array A to represent the heap with n = A.length and A.heapsize.
   - We start storing from index 1.
   - Replace the value of the node to be deleted (at position i) by the value of the last element in the heap.
   - Decrement the heap count.
   - Check if i > 1 and if the parent of element at index i is greater than the value at i then call MAX-HEAPIFY to restore the max heap property.
   - Else use HEAP-INCREASE-KEY to repeatedly compare element to its parent and exchange their keys if the element is larger and terminate if the element is smaller.

   Algorithm is as follows:

   - Let HEAP-DELETE(A,i) be the function to delete a node at position i.
   - In this function we first check two conditions that is if the node at i is greater than A.heapsize (represents how many elements in the heap are stored in array A) or node at i is less than 1.
   - If either of the conditions is true it returns to error message.
   - Then we store the value of A[i] in temperory variable, let this be x.
   - Also, replace the value A[i] with the with the last element in the heap (i.e., with the value at A[A.heapsize])
   - Now check two conditions that is if (i>1)and (A[PARENT(i)] > A[i]). If these both conditions are true MAX-HEAPIFY(A, i) else HEAP-INCREASE-KEY(A, i, A[i])
   - After checking if condition return the value of X which is the value of the deleted node at position i.
   - The Running time is O (log n) since we are traversing till the bottom of the heap, and we know that that the number of levels in the Heap is logn, so we have to check and shift only a maximum of log n times.

- And in the algorithm, we are either using HEAP-INCREASE-KEY or MAX-HEAPIFY both take O (log n).

2. Exercise 6.5-9: Give an O (n log k) time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (Hint: Use a minheap for k-way merging.)

   Given an array of K sorted lists. Here list is a linked list, so we need to return one output linked list.

   IMPLEMENTATION:

   i) Create a min-heap (of size k) Place a pointer on all the first elements of each sorted list. We use this pointer to access an element from the min-heap and find which linked list it belongs to.
   ii) insert the first elements of each sorted list which the pointer is present into the min-heap.
   iii) Extract the smallest element at the root of the min-heap and insert it into the output list.
   iv) If we have same elements in two or more sorted lists extract them and place them in output list.
   v) Update the current pointer.
   vi) Min-heapify the min-heap and repeat the process till the min-heap is not null.
   vii) Output list is the result we require.

   For example: Let us take 3 sorted lists.
   List 1: [5, 8, 10]
   List 2: [2, 4, 8]
   List 3: [3, 5, 9]
   Place a pointer on all the first elements (supposedly smallest elements) in each list.
   Insert these elements in min-heap and min-heapify
   Heap: [2, 3, 5]
   Extract the Root node of the heap and place in output list
   Output list: [2]
   Element 2 is from list2, so move the pointer of the list2 from 2 to 4 and insert 4 into min-heap and min-heapify.
   Heap: [3, 4, 5]
   Extract the Root node of the heap and place in output list
   Output list: [2, 3]
   Element 3 is from list 3, so now move the pointer to right and insert the value 5 into min-heap and min-heapify.
   Heap: [4, 5, 5]
   Extract the Root node of the heap and place in output list
   Output list: [2, 3, 4]
   Move the pointer in list 2 to right and insert this value 8 into min-heap and min-heapify.

Heap: [5, 5, 8]
Extract 5 and place them in output list. Now we have two 5's so, extract both
Output list: [2, 3, 4, 5, 5]
In this way move the pointer of the lists from which the element is extracted, to the right and insert the values in min-heap and min-heapify and extract the root node and insert into output list.
On repeating this process:
Heap: [8, 8, 9]
Output list: [2, 3, 4, 5, 5, 8, 8]
Heap: [9, 10]
Output list: [2, 3, 4, 5, 5, 8, 8, 9]
Heap: [10]
Output list: [2, 3, 4, 5, 5, 8, 8, 9, 10]
Min-heap is empty, so we end the process and result is the output list.

3. Exercise 6.1: Exercise Show that there is an algorithm that produces the k smallest elements of an unsorted set of n elements in time O (n + k log n). Be careful: To do this problem correctly, you have to do two things:
   1. State the algorithm carefully and prove that it does what it is supposed to do. (The proof can be very simple.)
   2. Prove that the algorithm runs in time O (n + k log n)
   We can use min-heap to get kth smallest element of an unsorted set of n elements in time O (n+klog n).
   IMPLEMENTATION:
      (a) Build a min-heap of size n of all elements.
      (b) Traverse the min-heap (K-1) times by Extracting the minimum elements & performing heapify operation k times.
      (c) Return to the root value which is the kth smallest element.

   PSUEDO CODE:

      Kth-smallest (A, n, k):

         1. BUILD-MINHEAP (A, n)
         2. For i =1 downto k-1
         3. EXTRACT-MIN(A)
         4. Return A [1]

   TIME COMPLEXITY:

   =Building min-heap of n elements + extracting min elements K-1 times

   = O(n) + (K-1) *log(n)

   = O (n + k log n)

SPACE COMPLEXITY IS O (1)

If we use max heap, we have O (n log k) time and O (1) space.

4. Exercise 7.3-2: When RANDOMIZED-QUICKSORT runs, how many calls are made to the random number generator RANDOM in the worst case? How about in the best case? Give your answer in terms of θnotation.

   Every time 'Randomized partition' is called, RANDOM is also called. so, if we can estimate the calls made for 'Randomized partition' in 'Randomized-quicksort'. we can know how many random number generators RANDOM is called.
   WORST CASE:
   If each pivot consistently ands on either of two extremes, n-1 calls are made to Random Number Generator (RNG). We have one with 0 elementsand the recursive call on array of size 0 returns T(0) = θ(n). The partitioning costs θ(1) time.
   Recurrence of running time of randomized partition is

$$T(n) = T(n-1) + T(0) + \theta(1)$$

   $T(0) = C1$
   $T(n) = T(n-1) + c2$
   $T(n) = T(n-k) + kc2$
   $if\ k = n\ we\ get\ T(n) = T(n-n) + nc2$
   If we sum the costs at each level of recursion we get

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

   BEST CASE:
   Partition divides into two subproblems, each of size no more than n/2.Time to solve two partitioning's of size n/2 is 2T(n/2). So, recurrence for the running time is then

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(1)$$

   $using\ masters\ theorem\ we\ get\ T(n) = \theta\left(n^{\log_2 2}\right) = \ \theta(n^1) = \theta(n)$

5. on Tail-Recursive-Quicksort
   a. Argue that TAIL-RECURSIVE-QUICKSORT(A, 1, A.length) correctly sorts the array A.
      TAIL-RECURSIVE-QUICKSORT performs same operations as Quick-sort. Since the Quicksort algorithm has already been proven, we can use it to prove that tail-recursive-quicksort correctly sorts the array A.
      Quick sort calls itself recursively the left and right sub-arrays (A[p, q-1] and A[q+1,r]) which will be further partitioned.
      Tai-Recursive-Quicksort – initially recursively calls left sub array A[p,q-1] then increment p to q+1 and with while loop, we partition A[q+1,r] and recursively calls itself.

Thus, both recursions and partitioning is done on sub arrays similarly. So, TRQ correctly sorts the array A.

b. Describe a scenario in which TAIL-RECURSIVE-QUICKSORT's stack depth is $\theta(n)$ on an n-element input array.
TRQ has stack depth $\theta(n)$ when partition return a value where q=r.
this satifies when there is an array which is already sorted.
Example for the sequence of calls :

TAIL-RECURSIVE-QUICKSORT(A,p,n)
TAIL-RECURSIVE-QUICKSORT(A,p,n-1)
TAIL-RECURSIVE-QUICKSORT(A,p,n-2)
TAIL-RECURSIVE-QUICKSORT(A,p,n-2)
.
.
TAIL-RECURSIVE-QUICKSORT(A,p,1)

Worst case here will be when q<r at every point so we will have stack depth of $\theta(n)$.

c. Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is $\theta(lg\, n)$. Maintain the $O(n\, lg\, n)$ expected running time of the algorithm.

T-R-Q-new (A, p, r)

```
i.    If (p < r )then
ii.     q = PARTITION (A, p, r)
iii.    if [q < floor ((p + r) / 2)] then
iv.       T-R-Q-new (A, p, q - 1)
v.        p = q + 1
vi.     else
vii.      T-R-Q-new (A, q + 1, r)
viii.     r = q - 1
```

6. Problem 7-4: Assume that c ≥ 0, and assume you had some kind of super-hardware that, when given two lists of length n that are sorted, merges them into one sorted list, and takes only n^csteps.
(a) Write down a recursive algorithm that uses this hardware to sort lists of length n.
The divide and conquer concept efficiently used in merge sort. It works by splitting a huge array into smaller subarrays, which are then sorted recursively. Once both halves are sorted, the algorithm would use the super-hardware to merge the two lists together into single sorted list.

The fundamental strategy entails halving the array at each level, comparing the nodes of the two subarrays, and adding a node with a lesser value to the output array. The merging of the entire array is handled following this recursion.
As a result, the run time is O. (n log n)
The method that employs this strategy can sort the list significantly more quickly if you have super hardware to merge two sorted lists as quickly as feasible.
The list or array would be divided into two equal parts, and each hardware would then be sorted using the super hardware.
It will use extremely powerful technology to combine the two lists or arrays once the two parts have been sorted into a single sorted list.

Mergesort has two functions: mergesort and merge.

Let array A
Split into half we get left and right sub arrays let them be L and R
Let pointers k, i, j points to array A and Left and right subarrays.
Psuedo code:

```
Merge(L, R, A){
    nL = length(L)
    nR = length(R )
    I = j = k = 0
    while(i<nL and j<nR){
            If ( L[i] <= R[i] ){
                    A[k] = L[i]
                    i+=1    }
            else {
                    A[k] = R[j]
                    j+=1    }

            K+=1            }

    while ( I < nL ){

            A[k] = L[i]

            i++

            k++     }

    while (j<nR){

            A[k] = R[j]

            j++

            k++     }
```

MERGE-SORT(A)

a. n = length(A)
b. if (n < 2)
    i. return
c. mid = n/2
d. left = array of size(mid)
e. right = array of size (n-mid)
f. for i = 0 to mid – 1
    i. left [i] = A [i]
g. for i = mid to n - 1
    i. right [i - mid] = A [i]
h. MERGE-SORT(left)
i. MERGE-SORT(right)
j. MERGE-SORT(left, right, A)

(b) Write down a recurrence to describe the run time.

The recurrence of this algorithm would be $T(n) = 2T\left(n/2\right) + cn.$

Where $T(n)$ is run time of merge sort.

$2T\left(n/2\right)$ time for breaking it down to sub halves

cn time for merging

On solving :

Let $T(n) = 2T\left(n/2\right) + cn.$ -eq(1)

$T(n / 2) = 2T(n / 4) + cn / 2$ -eq(2)

$T(n / 4) = 2T(n / 8) + cn / 4$ -eq(3)

Eq(2) in eq(1) we get

$T(n) = 2[2T(n / 4) + cn / 2] + cn.$

$T(n) = 4T\left(n/4\right) + cn + cn$

$T(n) = 4T\left(n/4\right) + 2cn$ -eq(4)

Eq(3) in eq(4) we get

$T(n) = 8T\left(n/8\right) + 3cn$

From this we can write $T(n) = 2^k.T\left(n/2^k\right) + kcn$

This gives the run time of $O(cn)$

(c) For what values of c does this algorithm perform substantially better than O (n log n)? Why is it highly implausible that this kind of super-hardware could exist for these values of c?

This algorithm would perform substanstially better than O(n log n) for values where c is less than half(1/2).

This implies that the sorting of n items list is done in less than n/2 steps, which is physically not possible. That is why it highly implausible that this kind of super-hardware could exist for these values of c.

7. In a binary tree, a leaf node is a node whose left and right children are both nil. The depth of the tree is the maximum number of edges between the root node and any leaf node. Show that if a binary tree has depth n, then it has at most 2^nleaf nodes.

   We will use mathematical induction.

   Let X be set of integers n >= 0.

   Base case: the tree has one node if n = 0 then (root node) and have no leaf nodes.

   Hence there is 1 leaf node (which is $2^n = 2^0 = 1$ if n = 0) and 1∈ X

   Suppose for integer k>=0, all integers 0 through k are in X. That is when a binary tree has depth M with M<=k it has at most 2^Mleaf nodes.

   Let T be a binary tree with k depth. If T has the maximum number of depths, T consists of a root and two nonempty subtrees, say X1 and X2. Let X1 and X2 have M1 and M2 depth respectively. Since M1 and M2 are between 0 and K, each is in X by inductive assumption.

   Since all the leaves of T must be leaves of X1 and X2, the number of leaves in T is at most $2^{k-1} + 2^{k-1}$ which is 2^k. Therefore k is in X.

   Hence by mathematical induction X= [1,∞].