CSC 505, Spring 2018 ***Homework #2***

due Friday, 23 February 2018, at 11:45 PM

Student Name	**Student ID**
Akshay Nalwaya	200159155
K. Sai Sri Harsha	200207493

Note: If you are working with the same partner as on Homework 1 and Project 1, there is no need to register. If you will be working with a new partner, have both you and your partner post to Piazza by Friday, February 16.

When you submit your assignment solutions please take the time to indicate the starting page of each answer. To make life easy for the TA's put the answer to each problem on a separate page (you may put multiple parts of the same problem on one page)

[Total points: 30]

1. [8 points, 4 points each part] *Understanding Heapsort*

   Suppose that, at the beginning of Heapsort, the array has $A[i] = i$ for $i = 1,...2^k-1$ for some k. Consider only what happens during the BuildHeap, a.k.a, MakeHeap phase. In answering each of the following questions, give a *rigorous proof* either by describing the situation clearly and deriving a recurrence, or via an induction argument.

(a) What is the *exact* number of comparisons in this case, as a function of k? Hint: Come up with a recurrence and solve it.

(b) How many of the Heapify calls in the main loop recurse all the way to the bottom level.
   Recall that the BuildHeap algorithm is
   for i = floor(n/2) **to** 1 **do** Heapify(i) **end do**
   The question refers to the floor(n/2) calls in the loop, most of which will make additional recursive calls to Heapify, but not all will recurse all the way to the bottom.

Sol.

The tree will be of the form

Since there are 2^{k-1} elements, it is evident that tree will be a complete binary tree of depth k (assuming root node is level 1)

The recurrence relation for counting the exact number of comparisons will be of the form:
T(k) = 2(k-1) + 2*T(k-1)
where, the quantity 2(k-1) denotes the number of comparisons occurred during the current node's recursive ***Heapify(i)*** call while the T(k-1) is the number of comparisons for one child and since it's a binary tree, we have 2*T(k-1). The number of comparisons is same for both children because it's a complete binary tree.

Solving the recurrence, ***T(k) = 2*T(k-1) + 2(k-1)***

T(k) = 2T(k-1) + 2(k-1)
= 2[2T(n-2) + 2(k-2)] + 2(k-1)
= 4T(k-2) + 4(k-2) + 2(k-1)
...
So, for i^{th} level, it can be generalized as
$= 2^i T(k-i) + \sum_{i=1}^{k} 2^i(k-i)$
...
Hence, for all the levels we can conclude the value of T(k) as,

$T(k) = 2^k T(1) + \sum_{i=1}^{k} 2^i(k-i)$
Since there are no comparisons at the last level, so, ***T(1) = T(0) = 0***
Hence, ***T(k)*** = $\sum_{i=1}^{k} 2^i(k-i)$

T(k) = $k\sum_{i=1}^{k} 2^i - \sum_{i=1}^{k} i2^i$
= $k\sum_{i=1}^{k} 2^i - \sum_{i=1}^{k} i2^i + k - k$ *[Adding and subtracting k to start summation from i = 0]*
= $k\sum_{i=0}^{k} 2^i - \sum_{i=1}^{k} i2^i - k$
The first part of above equation is GP while the second is an AGP, so applying corresponding formulae from the class notes, we get,

$= k(2^{k+1} - 1) - (2 - (k+1)2^{k+1} + k2^{k+2}) - k$
$= k * 2^{k+1} - k - 2 + (k+1)2^{k+1} - k2^{k+2} - k$
$= 2^{k+1} - 2(k+1)$

Hence, ***T(k) = 2^{k+1} - 2(k+1)*** gives the exact number of comparisons.

(b) Since all the numbers are arranged in ascending order, so when heapify(i) is called, for each of the node i, we can find some child node which is greater than this node, so it has to go till bottom of the tree. And since there are a total of floor(n/2) iterations in the loop, there will be a total of floor(n/2) recursive calls to heapify(i) which recurse all the way to the bottom level.

2. [6 points] ***Understanding Quicksort***

Suppose list-based Quicksort – the kind where the elements with equal keys are put in a separate list and are not involved in any recursive calls, is used to sort elements with keys in the range 1,...,*k* for some integer *k*.

Give a Θ bound, as a function of both *n* and *k*, on the worst case number of comparisons for Quicksort in this situation. You have to prove both a big-oh and a big-omega bound. For the former you need an upper bound, for the latter you need to describe an example that works for any value of *n* and *k*.

For the big-oh bound you cannot presume to know what the worst case looks like – your argument has to be general, applying to all cases.

Sol.

Number of comparisons for quicksort T(N) = T(N-1) + (N-1)

As the elements with equal keys are placed in a separate list and are ignored for the further recursive calls, the total number of recursive calls are now reduced to 'k'. This is because all the equal keys are ignored, and only distinct elements are considered during the recursion.

To come up with a general recursion for list-based quicksort where equal elements are ignored, let's consider the following scenario:

In an array of 'N' elements, let's pick a pivot which is repeated 'p' times in the initial array. Now, all the 'p' elements are removed from the array and the remaining array is sorted recursively. Here, the value of 'p' can range from 1 through N-1. For each recursive call with 'N' elements in the array, we get 'N-1' comparisons. Thus, our recurrence relation goes as follows:

$$T(N) = T(N - p) + N - 1$$

In worst case, maximum number of comparisons in a single recursive call happen when 'p' takes the least possible value till the last recursive call. In other cases, the number of recursive calls is much lesser as the total input size itself is less since we ignore the elements with equal keys.

$$T(N) = T(0) + N\text{-}k + \ldots. + N\text{-}3 + N\text{-}2 + N\text{-}1$$

$$T(N) = kN - (1 + 2 + 3 + \ldots.. + k)$$

$$T(N) = kN - k*(k+1)/2$$

Therefore, **$T(N) = \theta(n*k)$**

Proof:

Part A: First, let's consider the part T(N) = O(n*k)

From the definition of big-Oh, we have

$$T(N) \leq c_1 * n * k \ (c_1 > 0 \text{ and } n > n_0)$$

$$\Rightarrow \ kN - (k * (k+1))/2 \leq c_1 * n * k$$

$$\Rightarrow \ n - (k + 1)/2 \leq c_1 * n$$

Upon substituting n_0 and for a given c_1, we obtain

$$\Rightarrow n_0 \geq (k+1)/(2 * (1 - c_1))$$

Assuming $c_1 = 0.5$, we obtain $n_0 \geq (k+1)$.

Hence, we can conclude that **T(N) = O(n * k)**

Part B: Let's consider the part T(N) = Ω(n * k)

From the definition of big-Omega, we have

$$T(N) \geq c_1 * n * k \ (c_1 > 0 \text{ and } n > n_0)$$

$$\Rightarrow \ kN - (k * (k+1))/2 \geq c_1 * n * k$$

$$\Rightarrow \ n - (k + 1)/2 \geq c_1 * n$$

$$\Rightarrow \ -(k + 1)/2 \geq (c_1-1) * n$$

$$\Rightarrow \ 1 - (k+1)/2n \geq c_1$$

$c_1 > 0$ when $n_0 \geq (k+1)/2$. This is true as $n \geq k$

Hence, we can conclude that **T(N) = Ω(n * k)**

Therefore, T(N) = θ(n*k)

3. [8 points] ***Adapting bin sort to solve a new problem***

Assume you are given an array A with n not necessarily sorted numbers (need not be integers). Describe an algorithm for finding the largest gap, that is, two numbers $A[i]$ and $A[j]$ such that $A[i] < A[j]$ and there is no $A[k]$ in the array such that $A[i] < A[k] < A[j]$, and so that $A[j] - A[i]$ is maximal. Your algorithm has to run in time O(n). Hint: *Let a be the minimum number in the array and b be the maximum. Divide the range of numbers into intervals of length $g = (b - a)/n$. Here g is not necessarily an integer.*

Please pay heed to the following instructions for describing an algorithm (taken from Erik Demaine): Try to be concise, correct, and complete. To avoid deductions, you should provide

(a) a textual description of the algorithm, and, if helpful, pseudocode;

Sol.

Description: Firstly, preprocess the input array in the following manner:

Create 'n' buckets of equal size starting from minimum element 'a' through the maximum element 'b'. Each of the bucket should be of size g = (b – a)/n.

Bucket No.	Minimum element	Maximum element
1	a	a + g
2	a + g	a + 2*g
…		
I	a + (i-1)*g	a + i*g
…		
N	a + (n-1)*g	b

Now, iterate through the input array and update the minimum and maximum element of the corresponding bucket.

The maximum difference between two adjacent numbers (after sorting) will always be from two adjacent non-empty buckets. (Please find the proof of correctness of this assertion in part (c) of this question).

Pseudocode:

// Required Data Structures

Struct **Bucket** contains:

    Bucket_min // minimum value in the bucket

    Bucket_max // maximum value in the bucket

end

Function GET_MAX_DIFFERENCE(A):

1. b = MAX_VAL(A) // obtains the maximum value in the array 'A'
2. a = MIN_VAL(A) // obtains the minimum value in the array 'A'
3. Compute g = (b – a)/A.size
4. Initialize the buckets, **Bucket bucket[A.size]**
5. FOR index in 1 to A.size:
   a. bucketIndex <- Floor((A[index] – a)/g)
   b. if(bucket[bucketIndex].Bucket_min > A[index]):
      i. bucket[bucketIndex].Bucket_min = A[index]
   c. if(bucket[bucketIndex].Bucket_max < A[index]):
      i. bucket[bucketIndex].Bucket_max = A[index]
6. Initialize max_diff <- 0
7. Initialize start <- 1
8. FOR index in 2 to bucket.size:
   a. if(bucket[index] is not empty):
      i. if(max_diff < bucket[index].Bucket_min – bucket[start].Bucket_max):
         max_diff = bucket[index].Bucket_min – bucket[start].Bucket_max
      ii. start = index
9. Return (max_diff)

(b) at least one worked example or diagram to illustrate how your algorithm works;

Sol: Let the array be = [3,18,16,27,5,4]

Size of the array = 6

Minimum element(a) = 3

Maximum element(b) = 27

Bucket size(g) = (b-a)/n = (27-3)/6 = 4

Bucket No.	Bucket_min	Bucket_max
1	3	5
2	NULL	NULL
3	NULL	NULL
4	16	18
5	NULL	NULL
6	27	27

Gap between adjacent non-empty buckets:

Bucket 1 and Bucket 4 = 16 – 5 = 11

Bucket 4 and Bucket 6 = 27 – 18 = 9

Maximum gap = 11

(c) proof (or other indication) of the correctness of the algorithm;

Sol:

Proof of correctness:

Assertion: The maximum difference between two adjacent numbers (after sorting) will always be from two adjacent non-empty buckets.

Case 1: This assertion comes from the Pigeonhole principle which states that if there are 'n' elements and 'n' buckets, then at most one element can be placed in each bucket such that none of the buckets remain empty. Clearly, in this arrangement, we find only one element in each bucket and hence the maximum difference will be between numbers from two adjacent non-empty buckets.

Case 2: In a different arrangement, if there are multiple elements in a single bucket, then we can find at least one bucket which remains empty. Hence, even in this case, the maximum difference between two numbers will be from two adjacent non-empty buckets which is greater than 'g' as opposed to the maximum difference between elements in the same bucket which will be less than 'g'. Since n > 2 and 'a' and 'b' are placed in the first and last bucket respectively, there won't occur a case where all the elements are filled up in a single bucket.

(d) an analysis of the time (and, if relevant, space) complexity of the algorithm.

Sol:

Time Complexity analysis:

1) The first 'for loop' in the above pseudocode runs from 1 till A.size = O(n)
2) The second 'for loop' in the pseudocode runs from 1 till bucket.size = O(n) (since the number of buckets is equal to the number of elements in the array)
3) All the other statement run in O(1) time.
4) Hence, the total time complexity of the solution = O(n)

Space Complexity analysis:

1) Total number of buckets = n
2) Size of each bucket = O(1). (Each bucket stores only two values - min and max elements of the bucket)
3) Hence, the total space complexity = O(n)

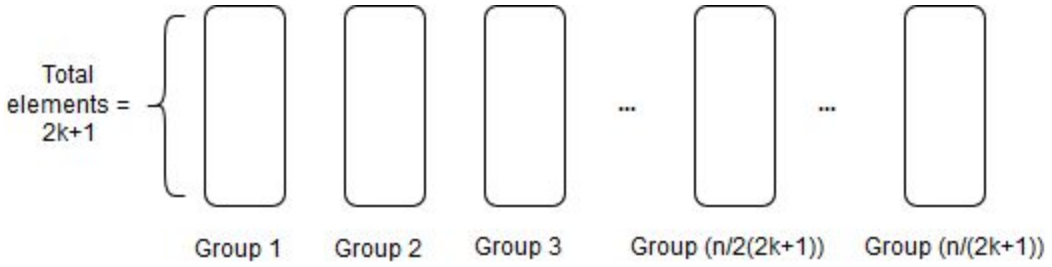
4. [8 points, 4 points each part] ***Understanding the analysis of the linear time median algorithm***

The worst-case linear-time median algorithm, a.k.a., the big five algorithm, is based on putting elements into groups of five. There is nothing magic about the number five. Other numbers can be made to work as well.

- **(a) Prove that the algorithm will still yield a linear number of comparisons for groups of any odd number > 5.**
- **(b) Prove that the algorithm will *not* yield a linear number of comparisons if groups of three are used. Here you have to prove that the number of comparisons is described by a function $f(n) \in \omega(n)$.**

Sol.

Let there be odd number of elements in each group, denoted by $m = (2k + 1)$ for some positive value of k.
Set $S = \{X_1, X_2, X_3, ..., X_n\}$ be the set of all elements X_i
S is partitioned into (n/m) groups of size m each.



We find median of each group which will be the central element when group is sorted. As, there are $(2k+1)$ elements in each group, the median element will be located at the $(k+1)^{th}$ position.
Since, size of each group is very small, we can ignore the time complexity of this step and assume it to be ***O(1)***.
We now have total (n/m) median values,
Next step is to find the median value of these medians, which will have time complexity of ***O(n/m)***
We finally use this element as our pivot element.

The idea is to partition the elements around the pivot such that all the elements which are lesser than the pivot at on the left side while the greater ones are on the right side. The set of elements to the left of pivot are denoted by S_L and those to the right are denoted by S_R.

Lower Bounds for S_L and S_R,

We know that at-least all the elements till the pivot elements in each of the groups before the center group (i.e., the group containing median of medians) will surely be less than the median of median value, so there are k elements in each of the $(n/2m)$ groups, which gives us the lower bound for number of elements in S_L as,

$$|S_L| \geq (k+1)\left(\frac{n}{2m}\right) = \frac{(k+1)n}{2(2k+1)}$$

Using the same rationale,

$$|S_R| \geq (k+1)\left(\frac{n}{2m}\right) = \frac{(k+1)n}{2(2k+1)}$$

Upper Bounds for S_L and S_R,

To determine the maximum number of elements in S_L we can say that it can have all the elements present throughout the groups except the ones in S_R, i.e., max value of S_L can be obtained by subtracting the least number of elements that S_R will have from the total elements n.

$$|S_L| \leq n - |S_R| = n - \frac{(k+1)n}{2(2k+1)} = \frac{n(3k+1)}{2(2k+1)}$$

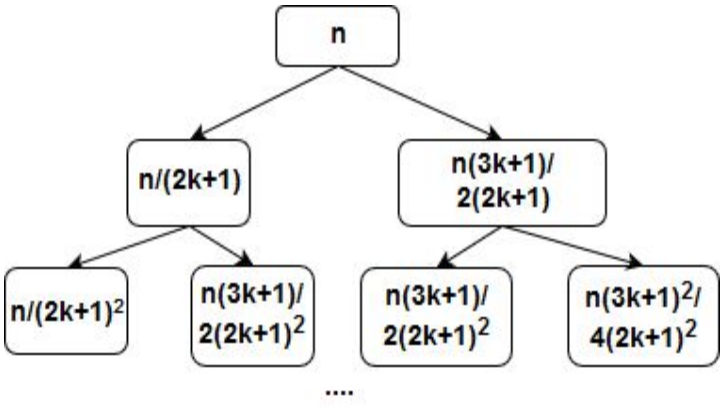
Similarly,

$$|S_R| \leq n - |S_L| = n - \frac{(k+1)n}{2(2k+1)} = \frac{n(3k+1)}{2(2k+1)}$$

Hence, recursive expression of the problem can be written as,

$$\mathbf{T(n) \leq O(n) + T\left(\frac{n}{(2k+1)}\right) + T\left(\frac{n(3k+1)}{2(2k+1)}\right)} \quad \textbf{…… eq. (1)}$$

But, this is not in a format where we can apply Master's Theorem directly,
So, using tree recursion method to simplify this expression,



$= \mathbf{n}$

$= \frac{n}{2(2k+1)} + \frac{n(3k+1)}{2(2k+1)} = \frac{3n(k+1)}{2(2k+1)}$

$= \frac{n}{(2k+1)^2} + \frac{n(3k+1)}{2(2k+1)^2} + \frac{n(3k+1)}{2(2k+1)^2} + \frac{n(3k+1)^2}{4(2k+1)^2}$

$= \left[\frac{3n(k+1)}{2(2k+1)}\right]^2$

Hence, for i^{th} level, we can write the total value will be $= \left[\frac{3n(k+1)}{2(2k+1)}\right]^i$ for i = 1 to k

The general expression of T(n) is

$$\mathbf{T(n) = \sum_{i=0}^{n-1} \left[\frac{3n(k+1)}{2(2k+1)}\right]^i + O(n)} \quad \textbf{……. eq. (2)}$$

(a) Since, it is given that groups are of odd numbers greater than 5,

$2k+1 > 5$

$k > 2$

$4k > 3k + 2$ *[Adding 3k on both sides]*

$4k+2 > 3k + 4$ *[Adding 2 on both sides]*

$2(2k+1) > 3k+4$

So, it is easy to deduce that $2(2k+1) > 3k+3$

Hence, ***2(2k+1) > 3(k+1)***

These are numerator and denominator of the expression in the summation in eq. (2)

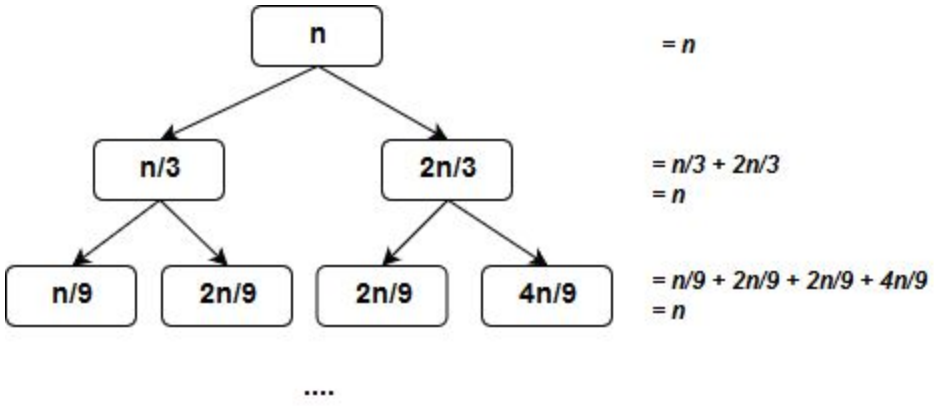
So, it is evident that denominator is always greater than the numerator for all allowed values of k.

Hence, the term O(n) in T(n) will always be greater than $\sum_{i=0}^{n-1} [\frac{3n(k+1)}{2(2k+1)}]^i$, so it can be concluded that the algorithm will yield a linear number of comparisons for groups of any odd number > 5.

(b) For groups of 3,
2k+1 = 3
So, k = 1
Substituting k=1 in numerator and denominator of equation (1), we get,

Substituting k = 1 in eq. (1), we get,
$T(n) \leq O(n) + T(\frac{n}{3}) + T(\frac{2n}{3})$

The following tree can be constructed using this recurrence relation,



Let the number of levels be *l*.
Number of levels in the tree will be $l = (log_3 n)$,
So, $T(n) = c * \sum_{i=1}^{l} n$
$= c * n * l$
$= \mathbf{c * n * \log_3 n}$

T(n) can be written as,
$T(n) = (\frac{c}{lg\ 3}) * n * lg\ n$
$= c' * n * lg\ n$ *[where c' = (c/lg 3)]*

So, we can conclude that total number of comparisons is ***O(n*lg n)***
With the constant ***c' > 0, and n > 0***

We can see that, $\lim_{n \to \infty} (n * lg\ n)$ tends to ∞

Also, $\lim_{n\to\infty}(n)$ tends to ∞

So, using L'Hopital Rule,

$\lim_{n\to\infty} \frac{n * lgn}{n} = \lim_{n\to\infty}(lg\ n)$

This limit tends to ∞,

So, we can say that ***T(n) ϵ ω(n) and the value of constants is c' > 0, and n > 0***