CSC263 Assignment #1

Divij Sanjanwala, Vaibhav Holani

February 11 2021

Question 1

Average case analysis

Let n be the number of positions in the array.
I will be doing analysis on the number of comparisons.

From our assumption, there is at most one 21 in the input list L.
Thus,

$$P(21 \text{ at } k^{th} \text{ position in L}) = \frac{1}{n}$$

<u>Case 1</u>: Exactly one 21 in input list L.

If 21 is at the k^{th} position there will be k loop iterations before $j.key == 21$ and as each loop iteration performs 2 comparisons, there will be $2k + 1$ comparisons.

Thus,

$$\begin{aligned} avg\_runtime &= \sum_{k=1}^{n} (2k) \cdot P(21 \text{ at } k^{th} \text{ position in L}) \\ &= \frac{2}{n} \cdot \sum_{k=1}^{n} k \quad = \frac{2}{n} \cdot \frac{n \cdot (n+1)}{2} \\ &= n + 1 \end{aligned}$$

<u>Case 2</u>: 21 not in input list L.

Here, the loop on line 2 will execute till j==None and this would happen when file reaches the end of the list (when line 3 sets j to None). However, after all the loop iterations which will make $2n$ comparisons. But the loop would end on the $2n + 1^{th}$ comparison where j != None would be false and loop would terminate.

Thus, let the probability of 21 being in the list be p.

<u>Conclusion</u>: The average case will be the sum of case 1 and 2

$$avg = (1 - p) \cdot (2n + 1) + (p) \cdot n + 1$$

Note: $T_{avg}(n) \in \Theta(n)$

Question 2

We will do the analysis on the number of loop iterations.

Let n be the input size.

Best Case Analysis

Assuming, at-least one submission was submitted n ≥ 1.
The best-case scenario will occur when the first line in the input file refers to the submission of 2 students where submission partner 1 is not in the same section as the submission partner 2.

Here, the condition on line 4 will be true and loop will terminate as the function returns.

Thus, there will be only one loop iteration.

$\therefore$ best case $\in \Theta(1)$

Worst Case Analysis

The worst case scenario will occur when when every submission in the input file refers to a submission submitted by only one student (CASE 1) or by two students in the same section(2).

In (CASE 1), submission.partner1 == submission.partner2
This implies the condition on line will be true as same student will always have the same section.

In (CASE 2),
As both the students belong to the same section, the evaluation on line 23 will return the same section.

Thus, in both cases, the condition on line 4 will be true and the function will not return early. As the function does not return early, the loop will iterate n times. Thus n operations.
Note: worst case $\in \Theta(n)$

Average-case Analysis

Let $n \in \mathbb{N}$
Let the submission file L be a list containing n submissions.
Let k be the position at which the first "across section" group appears in L.
Let p be the probability that a group contains an across-section partnership at a point in L.

Case 1: No across section group in L.

In this case, condition on line 4 will be true for each submission and the function will not return early. Thus having n loop iterations.

$$P(\text{No across section group in L}) \cdot n = (1-p)^n \cdot n$$

Case 2: First across section group at the k^{th} position in L.

In this case, condition on line 4 will be true on the k^{th} iteration as the submission would refer to the element at k^{th} position.
Thus, assuming $0 < p < 1$,

$$\begin{aligned} T_{avg}(n) &= \sum_{k=1}^{n} k \cdot P(\text{across section group at the } k^{th} \text{ position}) \\ &= \sum_{k=1}^{n} k \cdot (1-p)^{k-1} \cdot p \\ &= \frac{p}{1-p} \cdot \frac{(1-p)^{n+1} \cdot (n(-p)-1) + (1-p)}{(1-1+p)^2} \\ &= \frac{1}{p} \cdot ((1-p)^n \cdot (-1-np) + 1) \end{aligned}$$

Conclusion: The average case will be the sum of case 1 and 2

$$\begin{aligned} T_{avg}(n) &= \frac{1}{p} \cdot ((1-p)^n \cdot (-1-np) + 1) + (1-p)^n \cdot n \\ &= ... = (1-p)^n \cdot (\frac{-1}{p}) + \frac{1}{p} \end{aligned}$$

Note: $T_{avg}(n) \in \Theta(1)$ as $(1-p) < 1$ and as $n \to \infty$, $(1-p)^n \to 0$.

Calculating P

Here, as the probability of a student working alone is 0.2 and as the probability matches across all sections.

Then, $0.2 \cdot 700 = 140$ students would work alone (48, 48 and 44 students in the class respectively.) Thus, the number of students who would work in groups with another student is $700 - 140 = 560$ and students left in the classes would be 192, 192 and 176.

Situation 1

In these 560 students, as there are equal chances of any student choosing another student at random(regardless of the section), we can calculate the probability across section by multiplying the probability of a student belonging in a section to the probability of the student's partner being in another section doing .

$$P(\text{student working in across-section group}) = \left(\frac{192}{560} \times \frac{192 + 176)}{559}\right) \times 2 + \frac{176}{560} \times \frac{2 \times 192}{559} \approx 0.6673$$

Situation 2

Assume a student is 50% more likely to work with someone from their section than working with someone from another section. I picked this possibility as while students may know other students from all over different sections, they would like to pair with students in the same section as they are being taught by the same professor and as many times different profs like to teach the same concept in different ways which can create confusion as to which to way to follow to produce a better solution.

In this case, $P(\text{student working in the same section}) = 1.5 \cdot P(\text{across sections}) = 1.5p$.

Thus, as $P(\text{same section}) + P(\text{Across sections}) = 1$ as these events are complements. Therefore,

$$1.5p + p = 1$$
$$p = \frac{1}{2.5} \approx 0.4$$

Question 3

Part a)

We have 2 arrays A, B.
We know length of A is a and length of B is b, A and B are sorted in descending order and a + b = n.

The algorithm will contain a while loop condition which will stop iterating when all the elements in the loop of one of the lists have been iterated over and stored in the return array X.

The current position in array A, B, X (the merged list) will be tracked by i, j, k respectively all starting with an initial value of 0. As both A and B are sorted, the while loop would contain an if condition which will evaluate A[a] and B[b] and insert the bigger one to X[k] and increment i or j accordingly which would ensure the array is sorted in descending order at after each iteration.

After that, all the elements left in array that is not empty would be added to the array, as the element left in the array would be smaller than the elements in X. These elements would be added to the back of X and we would have X sorted in descending order.

Part b)

```
def merge(A, B):
    n = 0
    i = 0
    j = 0
    X = []
    while i < a and j < b:
        if A[i] > B[j]:
            X[n] = A[i]
                i += 1
        else:
            X[n] = B[j]
                j += 1
        n += 1
    X = X + A[i:] + B[j:]
```

Part c)

Here,
Let A and B be sorted arrays with length a and b and $a + b = n$.
Let A be [n, n - 2, n - 4,...] such that A[i] = $n - 2i$ for $0 \leq i < a$
Let B be [n - 1, n - 3, n - 5, ...] and B[j] = $n - 2j - 1$ for $0 \leq j < b$

Thus, in the lists define above, for $1 \leq k < min(a, b)$, B[k] < A[k] < B[k - 1]

Here, as A[0] (= n) > B[0] (= n - 1), the condition on line 6 would be true and i will be incremented to 1 by line 8. Now at i = 1, we know that A[1] < B[0], if B[0] exists.

Thus, on the next loop iteration condition on line 6 will be False, thus j would be incremented to 1 by line 11 and again by the definition of the list, A[1] > if B[0], if B[0] exists.

We will observe a similar pattern till there are no elements left in the smaller list.
Thus, there will be at-least $a + b - 1 (= max(a, b) + min(a, b) - 1)$ loop iterations = $n - 1$ iterations and as each iteration performs 1 element comparison, we get $n - 1$ comparisons.

Also, as loop iterates till i < a and j < b, the maximum number of loop iterations will be achieved when i = a and j = b - 1 (wlog). Also, as either the value of i or j is only incremented by 1 in each iteration, we can have at-most $a + b - 1$ iterations = $n - 1$ iterations which imply, $n - 1$ comparisons.

Note: worst case $\in \mathcal{O}(n)$

Part d)

Let A and B be sorted arrays with length a and b respectively.
Let $n = a + b$.
The best case will be the case where all the elements in the list of smaller length would be bigger than all the elements in the other larger list. As in this case, the loop will only execute $min(a, b)$ times and as there is only one element comparison per iteration, we would have $min(a, b)$ comparisons.
Moreover, as the loop must execute till all the elements in one of the list have been iterated. The loop needs to execute at least min(a,b) times which would imply at-least $min(a, b)$ comparisons.
Here, WLOG lets take a to be the list smaller in length.
Thus, $a = n - b$, best case $\in \mathcal{O}(n)$.

Part e)

The average case analysis can be divided into 2 cases: where list A and B are the first list to become empty.

Note: There are a total number of 2^n possible ways to merge List A and B together.

Case WLOG: List A is the first list to become empty
Let $X = X_1, X_2, X_3....X_n$ be the list formed after merging A and B.

Generally, in the case where X_i is the smallest element in List A, the loop would terminate after i comparisons. There would be $n - i$ elements in X before the i^{th} position which may be a combination elements from List A and B. Thus, there would be 2^{n-i} ways of writing that part of X. Additionally, we know to reach the i^{th} we would have i iterations and as each iteration would perform 1 element comparison, there would i comparisons.

Therefore,

$$P(X_i\text{is the smallest element in A}) = \frac{2^{i-1}}{2^n}$$

and with each having a runtime of i comparisons.
The average case in this case would be -

$$\begin{aligned} T_{avg}(n) &= \sum_{i=1}^{n-1} \frac{2^{i-1}}{2^n} \cdot i \\ &= \frac{1}{2^{n+1}} \cdot \sum_{i=1}^{n-1} 2^i \cdot i \\ &= \frac{1}{2^{n+1}} \cdot \frac{2^n(n-2)+2}{(-1)^2} \\ &= \frac{n-2}{2} + \frac{1}{2^n} \end{aligned}$$

Thus, we only considered one case with WLOG, multiplying it by 2 lets us consider the case where B is the first list to become empty. Thus,

$$T_{avg}(n) = \frac{1}{2^{n-1}} + (n-2)$$

Part f)

Let n be the sum of length of all k lists.

Assigning important variables -
- An array X of size of n
- An array y of size k
- An integer variable curr = 0

The answer is divided into 4 steps:
Step 1:
We remove the first element from each of the k lists and store them in an unsorted array **y** and keep track of the array of each element was removed from (Used in Step 3).

Step 2:
We call max-heapify on **y**. The order in which we call it is bottom up. max-heapify is only called on nodes that are not leaves.
Now, the array **y** is a valid max heap.

Step 3:
Now we remove the element y_{old} at the 0^{th} index in **y** and add it at position X[curr] and increment the value of curr by 1. Here, we extract the new first element y_{new} from the k^{th} list y_{old} was removed from, kept track of by step 1. Now we insert y_{new} at the 0^{th} index in **y** and then call max-heapify on the y_{new} once.
We repeat step 3 till one of the k lists becomes empty.

Step 4:
When one of the k lists become empty, we remove the last element in **y** and put it at the 0^{th} index in **y** to restore heap structure and reduce the heapsize by 1. Now we, call max-heapify once on the first element of **y** to restore heap property.
Repeat the cycle of step 3 and 4 till all of the k lists become empty.

Worst case analysis
In the worst case,
Step 2 would make at-most $\left\lfloor \frac{k}{2} \right\rfloor$ calls to max-heapify as we only call it on nodes that are not leaves and each call would make at most $log(k)$ comparisons which would be the height of a max heap. Thus, step 2 would make at-most $\left\lfloor \frac{k}{2} \right\rfloor \cdot log(k)$ comparisons.
Step 3 would make at-most n calls to max-heapify as there n elements in all the lists and each call would make at most $log(k)$ comparisons (same reasoning as above). Thus, step 3 would make at-most $n \cdot log(k)$ comparisons.
Step 4 would make at most k calls to max-heapify as there are only k lists. Thus, using the same reasoning as above, step 3 would make at-most $k \cdot log(k)$ comparisons.

In total, the number of comparisons is $\left\lfloor\frac{k}{2}\right\rfloor \cdot log(k) + n \cdot log(k) + k \cdot log(k)$
Simplifying we get,

$$\begin{aligned}
&= \left\lfloor\frac{k}{2}\right\rfloor \cdot log(k) + n \cdot log(k) + k \cdot log(k) \\
&= (\left\lfloor\frac{k}{2}\right\rfloor + n + k) \cdot log(k) \\
&\leq (2k + n) \cdot log(k)
\end{aligned}$$

This quantity would be in $\mathcal{O}(n \log k)$

Question 4

Part a)

First, check what kind of child v is in-order to locate itself on the tree.
We will return the node above the point where w - the last new node - gets inserted.

1. If the node identified has a left child insert w in the right.

2. If it has no children insert left child.

```
def function(v: Node, w: Node) -> Node:
# If we have a single node
    if v.parent == null:
        w = v.left
        return v
# If we have v as a left-leaf
    if v.parent.right == null:
        w = v.parent
        return w

    while v.parent.right == v and v.parent != null:
        v = v.parent

# If the node traversal reaches the root node.
    if v.parent == null:
        while v.left =! null:
            v = v.left
        w = v
        return w

# When the traversing reaches the left child of the root
    R = v.parent.right

# Traverses towards the bottom of the tree
# till the sub-tree has no left child.
    while R.left != null:
        R = R.left

    w = R
    return w
```

Part b)

Best-case running time analysis

Here, we will do analysis on the algorithm as we traverse the nodes along the binary tree. It will be majorly focused on the specific input cases that result in the best-case of the algorithm.

Lemma 1: According to the properties of heaps, v ie: the last-node will always be a leaf since it is the last attached node of the nearly-complete binary tree.

The best-case running time will occur on a specific input where v ie: the reference to the current last node, is the left child ie: left-leaf (according to Lemma 1) in the nearly-complete binary search tree and its parents' right child is empty.

Line 6 and **Line 7** of the algorithm in part a) refers to the scenario when the the last attached node of a nearly-complete-binary tree is left child and its parent's right child is empty. This is checked by **Line 6** where, if the parent of the v (left-child / left-most leaf) has no right-child, then **Line 7** will execute and w (the insertion point) gets assigned to the parent of v.

It gets assigned to the parent of v because, we need to find the point of insertion (w) of the new node w, v's parent is where the new node will be attached.

We **know** that this input is the best-case as, the child can otherwise only be a right child.

Note: We can also have a single node and have similar running-time to this input, however taking this particular input adds value to the best-case analysis - thus we chose to explain this rather.

If v is a right-child, then algorithm will run the while loop to reach the top of the tree and then traverse downwards till it finds an empty node to find the insertion point (w).

This can happen in many ways.

First, if the tree is a complete binary tree, or it has v as the any right-child/right-leaf. Although, in each of the scenario the while loop executes and the algorithm traverses up and then down - thus making its run-time more than $\Theta(1)$, ie: in terms of $\Theta(\log(n))$ which is elaborated upon in the worst-case scenario.

To conclude, through this best case scenario, Lines 6 - 8 get executed, returning w, making the the best case a series of 3 constant time operations.

Thus in the best-case the running-time is $\Theta(1)$.

■

Part c)

Worst-case running time analysis

For the **worst-case analysis**, we will have to consider specific cases that make the algorithm traverse the most along and down the tree.

For the worst-case, we consider the case where v ie: the reference to the current last node, is the right-child ie: the right-leaf (according Lemma 1 in part b) in the nearly-complete binary tree.
Here, since we have a right-child/right-leaf, we know from the property of heaps that the left-child of the parent is **not-empty**.

Since it isn't empty, we require to traverse along the tree to identify the exact place where to insert the new tree.

Analysis:

Now, as our right child is not empty based on our specific input, the if-condition in **Line 2** fails to execute as having a right-child ie: v, confirms that we have a parent. Thus we jump to **Line 6**.

For **Line 6**, since the right-child of v's parents' is v itself, proving that v is not null, the if-condition in the **Line 6** also fails.

Continuing to the while loop in **Line 10**, we basically keep traversing up until the parent of v that gets reassigned every iteration of the while loop is not null.

Here is where we distinguish between a complete and a nearly complete binary tree.

Edge-case 1: If we had a complete binary tree, the v would be assigned to the root node of the whole tree.

Edge-case 2: Correspondingly, if we had a nearly complete binary tree with v as the reference to the current last node ie: the right-leaf, then the algorithm would traverse up-to the parent whose right child has no children thus we can find a position of insertion beneath it.

However, a complete binary tree ie: ***Edge-case 1*** causes the highest traversal along the tree among the two edge cases.

Thus, considering the worst-case as a complete binary tree.
As we consider the while loop in **Line 10**, the **number of iterations** depend on the height of the tree.

Thus, if we have $k = 2^n - 1$ number of nodes in the tree, running the while-loop in **Line 10** would traverse up the tree based on the height of the tree, which would be,

$$k - 1 = 2^n$$
$$n = \log_2(k - 1)$$

Therefore, **Line 10** would have $n = \log_2(k - 1)$ number of iterations of the while loop until it reaches the top of the tree ie: the root node.

Since it reaches the root node, the if-condition on **Line 14** evaluates to **True** and **Lines 15 - 18** are executed.

The while-loop in Line 15 does the same thing, however it traverses down until the end of the tree.

Thus, line 15 again has $n = \log_2(k - 1)$ number of iterations.

Once the while-loop terminates, it executes **Lines 16 - 18** which assigns the w as the left-leaf as the point of insertion and returns w it.

To conclude, since we traversed up and down the tree, we have $2(n = \log_2(k - 1))$ iterations + 3 constant-time operations.

Thus, we are in a **worst-case running time** of

$$\begin{aligned} &\Theta(2(\log_2(k-1)+3) \\ &\Theta(\log_2(k-1)+3) \\ &= \Theta(\log_2(k)) \end{aligned}$$

Thus, here we considered the input that would have the most number of iterations of the algorithm and most amount of traversals along and down the heap. Other possible input-cases such as: single node, left-leaf etc, were eliminated based on lesser running time while analysing **Lines 2 - 9**.

Therefore, a complete-tree has the worst-case running time of our algorithm at $\Theta(\log(k))$ based on the analysis above.

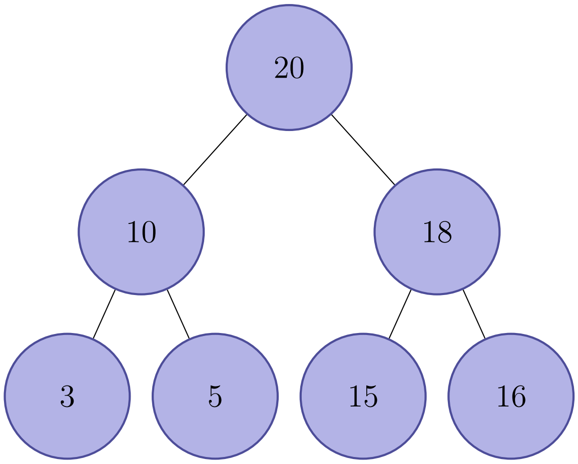
Note: Since we wanted to consider a complete-binary-tree in our worst-case, and **Lines 20 - 29** doesn't consider the traversal up till the root node of the tree, we don't consider them in our worst-case. Although based on intuition from the analysis above, they might turn out to have a similar running-time.

■

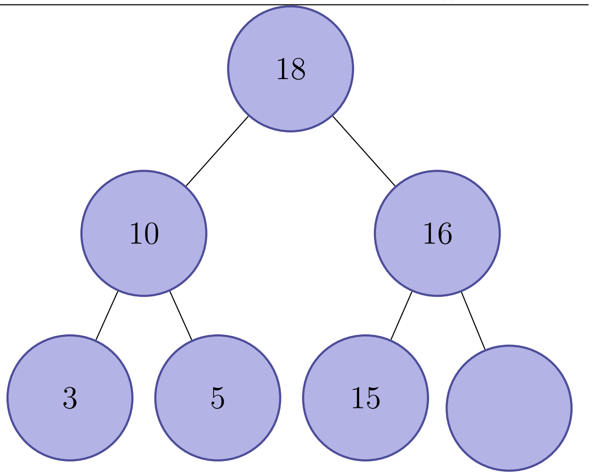
Question 5

Part 1)

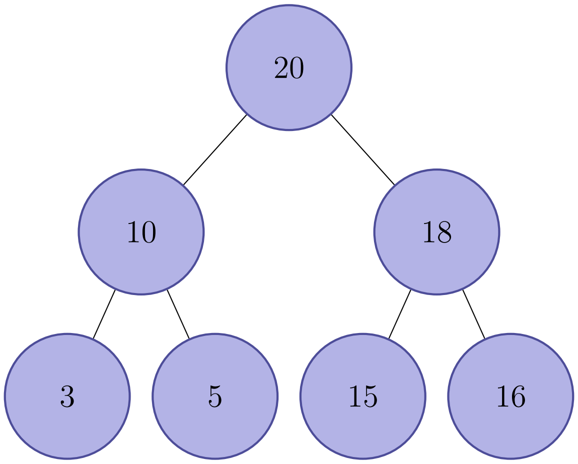
Original Heap



Intermediate Heap

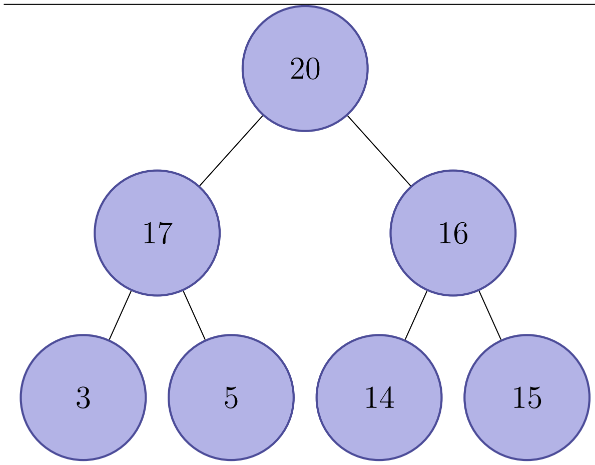


Final Heap

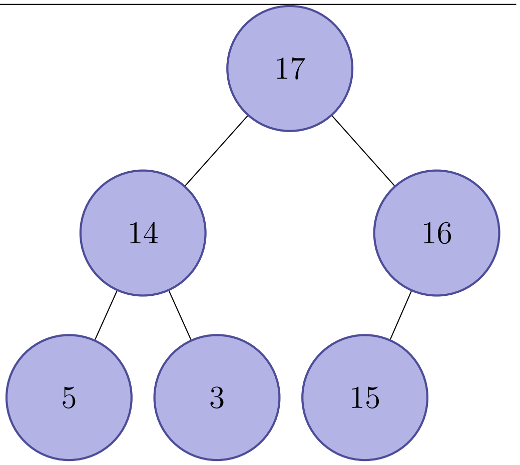


Part 2)

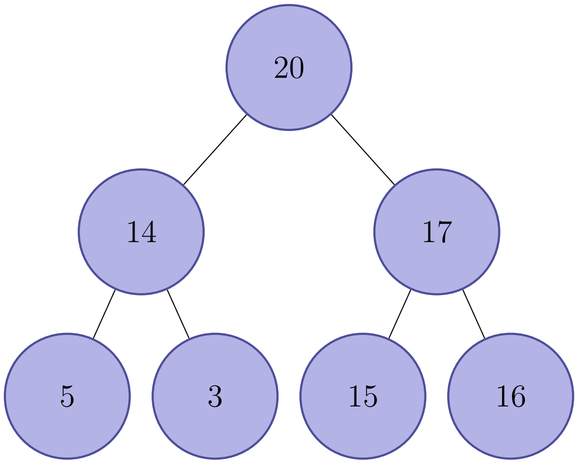
Original Heap



Intermediate Heap



Final Heap



Part 3)

The nodes from the right-most/left-most node to the root have values greater than than the values of their siblings.

Thus, based on the example from the Part 1, along the path from 16 to 20, all the nodes have their values greater than the value their siblings contain ie: $16 > 15$ and $18 > 10$.

Part 4)

The above property defined of a Delete-Insert-Stable tree ensures that the when we Extract the Max of the heap, the swapping ie: the reordering/bubbling-down of nodes happen in way that only the left-side of the heap is altered.

As only one side of the heap is used for bubbling-down, this ensures that when the Max node is re-inserted in the heap, the exchanged nodes exactly bubble up and attain their original position.

This property maintains that exactly same nodes are bubbled down and bubbled up when a Max node is extracted and re-inserted respectively. Therefore, this maintains the property and the requirements of the Delete-Insert-Stable max heap.

■