CSC263 Winter 2021 Problem Set 1

Xiao Owen Hu & Eric Zhu

February 11, 2021

Question 1

Since there can be at most one 21 (no repeats) and that it's equally likely to be in any of the n positions or not in the list at all, there are $n+1$ equally likely positions for 21.

Therefore, probability p of 21 at any position is $\frac{1}{n+1}$ which means it is a uniform distribution of $n+1$ cases.

Calculating the expected runtime where $t(k)$ is the runtime of 21 being in position k:

$$\begin{aligned}
E[t] &= \sum_{\text{All cases}} Pr(\text{each case}) \cdot t(\text{each case}) \\
&= \sum_{k=1}^{n} Pr(\text{21 at position k}) \cdot 2k + Pr(\text{21 not in list}) \cdot (2n+1) \\
&= \sum_{k=1}^{n} p \cdot 2k + p \cdot (2n+1) \\
&= 2p \cdot \frac{(n+1)n}{2} + p \cdot (2n+1) \\
&= p(n^2 + 3n + 1) \\
&= \frac{n^2 + 3n + 1}{n+1} \\
&= \Theta(n)
\end{aligned}$$

Since, this is a uniform distribution, $Pr(\text{21 at position k})$ and $Pr(\text{21 not in list})$ are both $p = \frac{1}{n+1}$. And since we are analyzing the same function as the one from the lecture, the runtime $t(\text{21 at position k}) = 2k$ and $t(\text{21 not in list}) = 2n+1$ as justified in "Lecture Extras: Average-Case Runtime Analysis of hasTwentyOne".

Question 2

Best Case:

From looking at code of `all_groups_legal`, we see that there is a possible early return at line 5. So for our best case analysis, we will consider the input family of arrays of submissions we will call *submissions* of size $n \in \mathbb{N}$, such that the first element in *submissions* will cause the program to execute line 5. In other words, the first element in *submissions* contains a submission where the two partners are from different sections. Then, it's a certainty that `all_groups_legal` with only execute 5 constant time operations, which gives us a best case runtime of $\Theta(1)$.

Worst Case:

From looking at code of `all_groups_legal`, we see that there is a possible early return at line 5. So for our worst case analysis, we will consider the input family of arrays of submissions we will each call *submissions* of size $n \in \mathbb{N}$, such that the first element in *submissions* will cause the for loop to complete and exit the program through line 6. Specifically, our input family consists of arrays that have submissions from partners from the only same section, which includes students working alone.

We can then be certain with our input family that line 4 will never execute for every element in *submissions*. Additionally, we see that in each iteration, we have 4 constant time operations, leading us to exactly $4n$ operations using this input family. Since $4n \in \Theta(n)$, our worst case runtime is $\Theta(n)$.

Average Case:

In the average case, we'll consider our input family to be the entire universe of possible inputs, i.e., all submission arrays of length n, where $n \in \mathbb{N}$. We find that there can be two classes of groups: those that conform to the group rules (single partner groups and those within the same class) which we can call "good groups" and those that don't which we can call "bad groups". From there, we can classify our inputs into arrays that only have submissions with "good groups" or arrays that have at least one "bad group".

We know from our worst case analysis that our worst case running time occurs when we have all "good groups". So when we have submissions of only "good groups", we have a running time of $\Theta(n)$ as we concluded in our worst case running time.

From our best case analysis, we were able to derive the $\Theta(1)$ running time because we realized that if there was a "bad group" at the first index, we could have an early return. So if there exists a submission belonging to a "bad group" in our array, we will exit at the index of that submission. It follows then that we have two cases of submissions: a "bad submission", which is one from a "bad group", and a "good submission", which is one from

a "good group". It's logical to denote $k \in \mathbb{N}$, the index of the first "bad submission" our array of submissions. Note that $0 \leq k \leq n-1$. Again assuming that we do have at least one "bad submission" in our array, we'll have to execute the for loop exactly k times. In this case we won't execute line 5 until the k^{th} iteration, so for $k-1$ iterations of the for loop we'll execute lines 1-4, leading us to 4 constant time operations. So there will be $4(k-1)$ operations for $k-1$ iterations. On our k^{th} iteration, we'll execute lines 1-5, leading us to 5 constant time operations. Adding these up, we get $4k-1+5 \iff 4k+1$ operations in total.

We'll introduce discrete random variable T, the running time of our program `all_groups_legal`. T takes the following possible values, i.e., the support of T is defined as:

$$T = \begin{cases} n \text{ if there are no "bad submissions"} \\ 4k+1 \text{ otherwise, where } k \text{ is } k \text{ as defined above} \end{cases}$$

To find the probability mass function of T, we'll consider the two distinct cases from our support of T, i.e., n and $4k+1$. Since the early exit is dependent on the index of the first "bad submission", we can define $p \in \mathbb{R}$, where $0 \leq p \leq 1$; and so p is our probability that we get a "bad submission". To make the rest of our analysis flow logically, we can consider the case where $T = n$.

In the case that $T = n$, it is necessarily true that we have no "bad submissions", i.e., for each of our n elements in our array, we do not execute line 5. So we have a probability of $1-p$. Since there are n independent elements in our array, we have a total probability of $\underbrace{(1-p)(1-p)(1-p)...(1-p)}_{(1-p) \text{ multiplied n times}} = (1-p)^n$.

Next, considering $4k+1$, we'll take inspiration from the case that $T = n$. Suppose our first "bad submission" comes at index $k-1$, our last element in our array. In this case we'd have a probability of $\underbrace{(1-p)(1-p)(1-p)...(p)}_{n \text{ elements}} = (1-p)^{n-1}p$. Let's do this a couple more times:

1. In the case that our first "bad submission" comes in our second to last element, we'd get a probability of $\underbrace{(1-p)(1-p)(1-p)...(p)}_{n-1 \text{ elements}} = (1-p)^{n-2}p$.
2. In the case that our first "bad submission" comes in our third to last element, we'd get a probability of $\underbrace{(1-p)(1-p)(1-p)...(p)}_{n-2 \text{ elements}} = (1-p)^{n-3}p$.

Notice how our exponents above our $(1-p)$ term is exactly $k-1$; this is because if our first "bad submission" is at k, we must have $k-1$ "good submissions" first. Since we have

exactly n elements, we could've done this for all n elements, but it'd be smarter to generate a summation in terms of k:

$$\sum_{k=1}^{n}(1-p)^{k-1}p$$

So we've now created our probability mass function, and all that's left is to find the expected value:

$$\begin{aligned}
\mathbb{E}[T] &= (\sum_{k=1}^{n} Pr(\text{first "bad submission" at k}) \cdot T(\text{first "bad submission" at k})) \\
&\quad + Pr(\text{no "bad submission"}) \cdot T(\text{no "bad submissions"}) \\
&= (\sum_{k=1}^{n}(1-p)^{k-1}p \cdot (4k+1)) + (1-p)^n \cdot n \\
&= (\sum_{k=1}^{n}(1-p)^{k-1}p \cdot 4k) + (\sum_{k=1}^{n}(1-p)^{k-1}p \cdot 1) + (1-p)^n \cdot n \\
&= (\frac{4p}{1-p})\sum_{k=1}^{n} k(1-p)^k + \frac{p}{1-p}\sum_{k=1}^{n}(1-p)^k + (1-p)^n \cdot n \\
&= (\frac{4p}{1-p})(\frac{(1-p)^{n+1}(n((1-p)-1)-1)+(1-p)}{(1-(1-p))^2} + \frac{p}{1-p}\sum_{k=1}^{n}(1-p)^k + (1-p)^n \cdot n \\
&= (\frac{4p}{1-p})(\frac{(1-p)^{n+1}(n((1-p)-1)-1)+(1-p)}{(1-(1-p))^2} + \frac{p}{1-p}\frac{(1-p)^{n+1}-1}{(1-p)-1} + (1-p)^n \cdot n \\
&= (\frac{4p}{1-p})(\frac{(1-p)^{n+1}(n((1-p)-1)-1)+(1-p)}{(1-(1-p))^2} - \frac{p}{1-p}\frac{(1-p)^{n+1}-1}{p} + (1-p)^n \cdot n \\
&= (\frac{4p}{1-p})(\frac{(1-p)^{n+1}(n(-p)-1)+(1-p)}{(1-(1-p))^2} - \frac{p}{1-p}\frac{(1-p)^{n+1}-1}{p} + (1-p)^n \cdot n
\end{aligned}$$

Since our expression above is in terms of p, we need to define p in order to obtain a sensible Θ bound for our expected runtime. The first key thing to note is that we're provided section sizes, and the probability of working alone. As such, our probability p is the probability of a "bad group" as a "bad submission" has an equivalent probability as these events are equivalent. So then, our probability of a "bad group" first entails a group, i.e., 0.8 chance, and then that group being bad. In more mathematical terms, we'll want to multiply our chance of having a group by the proportion of "bad groups".

Since we have a total of 700 students, we have a total of $\binom{700}{2}$= 244650 combinations of

groups. Out of these combinations of groups, we have 163200 bad groups, which we can calculate by computing $244650 - 2 \cdot \binom{240}{2} - \binom{220}{2}$, assuming that students are no more likely to choose partners across all sections. In other words, we subtract the number of "good groups" from each class by the total number of two person groups across all 3 sections. So our proportion bad groups is therefore ($163200/244650 = 0.667075$), which gives us a probability of $p = 0.8 \cdot 0.667075 = 0.53366$.

Let's also consider the more realistic scenario, where students are much more likely to pair with students within their own class, e.g., students are likely to coordinate which sections and classes to take and students are more likely to get to know students within their section. So then, it follows that the amount of "bad groups" we got in our first scenario would be less, but we don't have much information on how to specify that by the assignment handout. A reasonable guess would be that half the number of "bad groups" would now be good groups (81600 vs 163200), which gives us that the proportion of "bad groups" is now $\frac{81600}{244650} = 0.33538$, which seems to be reasonable. So our two values for p come out to be $0.53366, 0.33538$.

We'll now use our two values of p (0.53366 and 0.333538) in order to find a concise Θ bound for the runtime of the program. Recall our expression for the expected value of the runtime:

$$\mathbb{E}[T] = (\frac{4p}{1-p})(\frac{(1-p)^{n+1}(n(-p)-1)+(1-p)}{(1-(1-p))^2} - \frac{p}{1-p}\frac{(1-p)^{n+1}-1}{p} + (1-p)^n \cdot n$$

First, when $p = 0.53366$, we find our expression for the expected value of the runtime is:

$$\begin{aligned}
\mathbb{E}[T]_{p=0.53366} &= (\frac{4p}{1-p})(\frac{(1-p)^{n+1}(n(1-p)-1)-1)+(1-p)}{(1-(1-p))^2} - \frac{p}{1-p}\frac{(1-p)^{n+1}-1}{p} + (1-p)^n \cdot n \\
&= (\frac{4 \cdot 0.53366}{1-0.53366})(\frac{(1-0.53366)^{n+1}(n((1-0.53366)-1)-1)+(1-0.53366)}{(1-(1-0.53366))^2} \\
&- \frac{0.53366}{1-0.53366}\frac{(1-0.53366)^{n+1}-1}{0.53366} + (1-0.53366)^n \cdot n \\
&= (n + 7.49541(n(-0.53366) - 1.13342)) \cdot (0.46634)^n + 9.63977
\end{aligned}$$

Taking the expression $(n + 7.49541(n(-0.53366) - 1.13342)) \cdot (0.46634)^n + 9.63977$, we'll realize that after dropping the constant terms (in order to get a Θ expression), we have $(-3n - 8.49545) \cdot (0.46634)^n$, which we notice that as $n \to \infty$, the expression goes to 0 since $0.46634 < 1$. So similar to the `hasTwentyOne` example from class, we conclude that asymptotically, as $n \to \infty$, our average case runtime approaches 9.63977.

Second, when $p = 0.333538$, we find our expression for the expected value of the runtime is:

$$\begin{aligned}
\mathbb{E}[T]_{p=0.333538} &= (\frac{4p}{1-p})(\frac{(1-p)^{n+1}(n(1-p)-1)-1)+(1-p)}{(1-(1-p))^2} - \frac{p}{1-p}\frac{(1-p)^{n+1}-1}{p} + (1-p)^n \cdot n \\
&= (\frac{4 \cdot 0.333538}{1-0.333538})(\frac{(1-0.333538)^{n+1}(n((1-0.333538)-1)-1)+(1-0.333538)}{(1-(1-0.333538))^2} \\
&- \frac{0.333538}{1-0.333538}\frac{(1-0.333538)^{n+1}-1}{0.333538} + (1-0.333538)^n \cdot n \\
&= (n + 11.9926(n(-0.333538) - 1.08338)) \cdot (0.666462)^n + 13.4931
\end{aligned}$$

Taking the expression $(n + 11.9926(n(-0.333538) - 1.08338)) \cdot (0.666462)^n + 13.4931$, we'll realize that after dropping the constant terms (in order to get a Θ expression), we have $(-3n - 12.9925) \cdot (0.666462)^n$, which we notice that as $n \to \infty$, the expression goes to 0 since $0.666462 < 1$. So similar to the `hasTwentyOne` example from class, we conclude that asymptotically, as $n \to \infty$, our average case runtime approaches 13.4931.

Question 3

(a)

For this algorithm, we will essentially use the merge function of a merge sort (as we have learned in CSC148). We first create an array of size n to store the merged array. We keep two pointers for the two arrays, one pointer for the merged array and start at the 0th index. Then, we compare the values of the two elements at the pointers, assign the larger of the two elements to the merged array at the pointer and increment the pointer to the larger element and the pointer to the merged array. We repeat this process until one of the two pointers reaches the end of their respective length, at which point we check if the pointer to the other array has reached the end - if not, we push the rest of the elements of that array onto the merged array by incrementing the merged array's pointer and assigning each value to the pointer location.

(b)

```
    def merge_sorted_arrays(Array arrayA[a], Array arrayB[b]):
        i = 0 # pointer for arrayA
        j = 0 # pointer for arrayB
        k = 0 # pointer for merged
        Array merged[a+b]
        while i < a and j < b:
            if arrayA[i] > arrayB[j]:
                merged[k] = arrayA[i]
                i++
            else:
                merged[k] = arrayB[j]
                j++
            k++
        if i < a:
            for h = i to a-1:
                merged[k] = arrayA[h]
                k++
        else:
            for h = j to b-1:
                merged[k] = arrayB[h]
                k++
```

(c)

WTP: Worst-case runtime of `merge_sorted_arrays` is $O(n)$ element comparisons.

The function `merge_sorted_arrays` has inputs `arrayA` and `arrayB` of lengths a and b respectively and that the input size $n = a + b$. Assume `arrayA` and `arrayB` are sorted in decreasing order and contain no duplicates. Inside the function, the only element comparison is on line 7 inside the while loop.

Line 5 contains the while loop condition that terminates the loop when `i >= len(arrayA)` or `j >= len(arrayB)`. Since either i or j will increment inside the while loop, in the worst case, both arrays have the same length and i and j are both incremented to $i = a - 1$ and $j = b - 1$ before terminating after the next increment/iteration. In that case, the loop will iterate for a total of $a + b - 1$ or $n - 1$ times. Since each iteration has one element comparison, there will be a total of $n - 1$ element comparisons.

After the while loop, there is no more element comparisons, therefore the total number of element comparisons is $n - 1 = O(n)$.

(d)

The best-case running time of the merge algorithm would be $min(a, b)$ element comparisons where $min(a, b)$ stands for the smaller number of a and b or a if $a = b$.

Let a, b be arbitrary large integers, want to describe an input family of array A and array B of lengths a and b respectively.

If $a > b$, let array A contain the values $[a, a - 1, a - 2, ..., 1]$ and array B contain the values $[a + b, a + b - 1, ..., a + 1]$. Since every element of array B is larger than the first element of array A which is a, the entire array B will be iterated while the pointer for array A stays at index 0 and the while loop will terminate after b iterations when $j = b$. So the total number of element comparison is $min(a, b) = b$.

If $a \leq b$, let array B contain the values $[b, b - 1, b - 2, ..., 1]$ and array A contain the values $[b + a, b + a - 1, ..., b + 1]$. Since every element of array A is larger than the first element of array B which is b, the entire array A will be iterated while the pointer for array B stays at index 0 and the while loop will terminate after a iterations when $i = a$. So the total number of element comparison is also $min(a, b) = a$.

No other input could require fewer comparisons because the while loop will only terminate after the pointer $i = a$ or the pointer $j = b$, and since i and j start at 0 and increment once every iteration, there will be at least $min(a, b)$ loop iterations and consequently at least $min(a, b)$ element comparisons.

(e)

Let X be the resulting sorted array of the algorithm and assume that every input combination of A and B that could have led to creating X is equally likely.

To calculate the expected runtime, we divide the cases according to the loop exit states. First, we divide them into those that exit the loop on $i = a$ and those that exit on $j = b$:

$$\begin{aligned} E[t] &= \sum_{\text{All cases}} Pr(\text{each case}) \cdot t(\text{each case}) \\ &= \sum_{\text{exits on } i=a} Pr(\text{each case}) \cdot t(\text{each case}) \\ &+ \sum_{\text{exits on } j=b} Pr(\text{each case}) \cdot t(\text{each case}) \end{aligned}$$

Then, for those exiting on $i = a$, the cases can be divided into the possible values of j on exit. Since all input combinations are equally likely, we will use p for uniform probability for now and the runtime equals the number of while loop iterations which is $a + j$, since i and j are only incremented once each iteration.

$$\begin{aligned} \sum_{\text{exits on } i=a} Pr(\text{case}) \cdot t(\text{case}) &= \sum_{j=0}^{b-1} Pr(\text{case}) \cdot t(\text{case}) \\ &= \sum_{j=0}^{b-1} p(a+j) = \sum_{j=0}^{b-1} pa + \sum_{j=0}^{b-1} pj \\ &= pab + p\sum_{j=0}^{b-1} j \\ &= p(ab + \frac{(b-1)b}{2}) \end{aligned}$$

Similarly, for those exiting on $j = b$, the cases can be divided into the possible values of i on exit:

$$\begin{aligned} \sum_{\text{exits on } j=b} Pr(\text{case}) \cdot t(\text{case}) &= \sum_{i=0}^{a-1} Pr(\text{case}) \cdot t(\text{case}) \\ &= \sum_{i=0}^{a-1} p(b+i) = \sum_{i=0}^{a-1} pb + \sum_{i=0}^{a-1} pi \\ &= pab + p\sum_{i=0}^{a-1} i \\ &= p(ab + \frac{(a-1)a}{2}) \end{aligned}$$

Now, going back to the original calculation of $E[t]$ and adding the two together we get:

$$\begin{aligned} E[t] &= p(ab + \frac{(b-1)b}{2}) + p(ab + \frac{(a-1)a}{2}) \\ &= p(2ab + \frac{(b-1)b}{2} + \frac{(a-1)a}{2}) \\ &= \frac{p(4ab + b^2 - b + a^2 - a)}{2} \\ &= \frac{p[(a+b)^2 - (a+b) + 2ab]}{2} \\ &= \frac{p(n^2 - n + 2ab)}{2} \end{aligned}$$

Since all cases that result in the sorted array X are equally likely to occur and that there are b number of cases for loop exiting on $i = a$, a number of cases for loop exiting on $j = b$, and two cases that do not enter the loop where either A or B are empty, there are a total $a + b + 2 = n + 2$ number of equally likely cases. Therefore, $p = \frac{1}{n+2}$ and the expected value or the average case runtime $E[t] = \frac{n^2 - n + 2ab}{2(n+2)} = \Theta(n)$

(f)

In this algorithm, we will first create an array of size k to store a max heap of at most k nodes. Then, we will insert the first element of all k sorted arrays onto the max heap (line 7). We will also create an array of size n to store the return array and a temporary array of size $n - k$ to store all the un-inserted elements (line 8-9). We will also create two pointers for the return array and the temporary array.

Then, we will do `extract_max` on the max heap, assign the max number to the pointer location of the return array and increment the pointer. After each `extract_max`, we will also insert another element of the input arrays onto the heap. Since there are $n - k$ total elements that needed to be inserted, we will iterate through the $n - k$ elements of the temporary array (line 11) while extracting max, assigning them to the pointer location of the return array (line 12) while incrementing the pointer each time and inserting the rest of the elements onto the max heap (line 13).

After $n - k$ iterations, we will repeatedly do `extract_max` on the rest of the k elements of the max heap and assigning them to the pointer location of the return array (line 15-16) while incrementing the pointer each time.

Pseudo-code:

```
1        def merged_k_arrays(array1, array2, ..., arrayk):
2            MaxHeap mheap[k]
```

```
3           Array ret_arr[n]
4           Array temp_arr[n-k]
5
6           ret_arr_pointer = 0
7           temp_arr_pointer = 0
8
9           for arr in [array1, array2, ..., arrayk]:
10              mheap.insert(arr[0])
11              for i = 1 to len(arr):
12                  temp_arr[temp_arr_pointer] = arr[i]
13                  temp_arr_pointer++
14
15          for element in temp_arr:
16              ret_arr[ret_arr_pointer] = mheap.extract_max()
17              ret_arr_pointer++
18              mheap.insert(element)
19
20          for i = 0 to k:
21              ret_arr[ret_arr_pointer] = mheap.extract_max()
22              ret_arr_pointer++
23
24          return ret_arr
```

Since each input array is sorted in descending order, the initial max heap and the max heap after each insertion of array element contains the largest number of every input array that has not been pushed onto the return array. In other words, every time before we extract max, the max number of the heap will always be larger than every element that has not been on the heap.

Therefore when performing an `extract_max`, the max number is always the largest number of the entire input minus the previous max numbers that have been pushed onto the return array. So after we have extracted max $n-k$ times and inserted the rest of the $n-k$ elements onto the heap, the return array contains the $n-k$ largest numbers in descending order. Then, after we repeatedly extract max of the remaining heap and push onto the return array, we will have a sorted array of the entire input.

Worst-case runtime:

The heap insert on line 7 contains the first element comparison. It will take $\sum_{i=1}^{k} log(i)$ element comparisons in the worst case. To simplify the calculation, we will use a worse runtime $\sum_{i=1}^{k} log(k) = klog(k)$.

Line 11-13 contains a for loop that iterates $n-k$ times and performs a heap `extract_max` and `insert`. Since we know that both `extract_max` and `insert` take $\Theta(log(k))$ operations where k is the number of nodes on the heap, the worst-case runtime of each loop iteration is $2log(k)$. So the worst-case runtime of the entire for loop is $2(n-k)log(k)$.

Line 15-16 contains a for loop that iterates k times and performs a single `extract_max` each

iteration. As above, this mean that the worst-case runtime of the entire for loop is $klog(k)$.

Therefore, in total, the worst-case runtime of the function is:

$$klog(k) + 2(n-k)log(k) + klog(k) = 2nlog(k) = \Theta(nlog(k))$$

Question 4

(a)

```
1    def efficient_insertion_point(Node v , Node w):
2        if v.parent is None: # only root
3            v.left = w
4        elif v.parent.right is None: # empty sibling
5            v.parent.right = w
6        else:
7            # v.parent is none iff heap is complete bt
8            # else we have a v being part of some complete bt subtree
9            while v.parent is not None and v.parent.right is v:
10               v = v.parent
11           if v.parent is not None: # second case, i.e., v is part of
                 some complete bt
12               v = v.parent.right
13           # go down hugging left side
14           while v.left is not None:
15               v = v.left
16           v.left = w
17       return
```

(b)

Our best case running time will be $\Theta(1)$.

Consider an input of size k where k is an arbitrary large number, we'll define input family I_k, an input family of heaps represented by binary trees with k elements where k is an even number. If k is an even number, then the number of elements excluding the root $k-1$ is an odd number. Since every full depth has an even number of elements, this means that there is an odd number of nodes at depth h where $h = \lfloor log(k) \rfloor$, in other words the last "layer" of the heap represented by a binary tree.

If the last depth has an odd number of elements, that means the last node is a left child and does not have a sibling. So the algorithm will step into line 4 and exit. That is a total of one operation which means the best case running time is $\Theta(1)$.

No other input family of arbitrary large size k has a better runtime, because if k is an odd number, then the last depth will have an even number of nodes following the same logic above. This means that the last node will be the right child so it will step into the else block on line 6 and it will iterate through the while loops depending on the input size k. More precisely, the number of iterations will depend on the number of depths traversed which will be in terms of $log(k)$. Therefore, the running time will no longer be constant.

We conclude that the best case runtime is $\Theta(1)$.

(c)

Our worst case running time will be $\Theta(logn)$.

Examining our program `efficient_insertion_point`, we see that the only parts of the program that are dependent on the size of the heap, n, are the two while loops on lines 9 and 14, which implies that in our worst case, we must get these two loops to run to examine a runtime that is dependent on heap size, i.e., not constant time.

Note that we are not considering the trivial cases of n here (such as 1 or 0), as we wish to find an asymptotic bound for the worst case running time.

To get these two loops to execute, let's define input family I_n, where $n \in \mathbb{N}$, that is comprised of heaps of height h, where there are an even number of nodes at depth h. Note: v starts out at depth h by construction of a heap, and using the nearly complete binary tree property of a heap, we have that $h = \lfloor logn \rfloor$. Further, v will be the right child of some node at depth $h - 1 = \lfloor logn \rfloor - 1$.

Since v is the right child of some node at depth $h - 1$, w cannot be v's sibling. Thus, we have to enter the else branch on line 6. Next, we'll have to enter the while loop on line 9 because we consider n to be large and since v is at depth h, v will have a non None parent, and since v is defined to be the right child of its parent, we have that `v.parent.right` is v.

Examining how many iterations the while loop on line 9 takes, we first note that the only operation in the while loop is `v = v.parent` and so v moves over at most 2^d nodes for some depth d by getting reassigned to its parent, so given our heap of even size n, we would take $\Theta(logn)$ time to traverse the heap using our while loop. This is also just a property of a heap which is a nearly complete binary tree, i.e., $h = \lfloor logn \rfloor$, meaning there are h "layers" and if we traverse one node at each "layer" we will have a runtime of $\Theta(logn)$ due to $\Theta(logn)$ iterations with constant time operations at each "layer", which is exactly what our while loops do.

We can be more precise using a constant, c, of the height of the heap and a constant, k, the count of iterations of the while loop (the number of reassignments to v), i.e., $\exists c, k \in \mathbb{N}, c \cdot \lfloor logn \rfloor = k$. Solving for c, we get $c = \frac{k}{\lfloor logn \rfloor}$, and more succinctly expressed, c is the fraction of the height of our heap that v travels for some k iterations of the while loop. Alternatively, k can also be considered the change in depth v gets as a result of the while loop. After k iterations of the while loop we'll end up on line 11, which considers the case that the heap is a complete binary tree, i.e., we will enter the if statement on line 11 if the heap is not a complete binary tree because `v.parent` isn't `None`. In either case, v must travel down to a leaf node due to the while loop's condition on line 14. We know that in both

cases, the while loop will execute because we are not at a leaf node given that v is now at a depth of $h - k$, so `v.left` isn't `None`. Consequently, to travel down to a leaf node, we must travel k depths again and always going down the left branch, i.e., k iterations of the while loop on line 14. Note that in the case the heap isn't a complete binary tree, we will actually need to just travel $k - 1$ depths to reach a leaf node due to line 12. Recall that we express k as $k = c \cdot \lfloor logn \rfloor$, i.e., a constant fraction of the height, which is the exact same expression as the first while loop. This expression also holds for the case that we have $k - 1$ depths to travel to get to a leaf. So both while loops can therefore by the definition of big Θ, get a runtime of $c \lfloor logn \rfloor \in \Theta(logn)$.

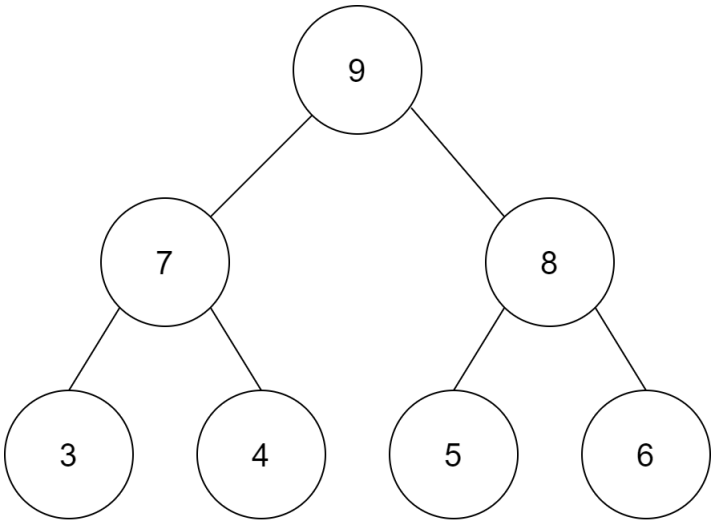
Then we note that we've are able to add the big Θ bounds for the while loop runtimes together, i.e., $\Theta(logn) + \Theta(logn) \in \Theta(logn)$. We also will have the other constant time operations (as we've accounted for all the heap size dependent operations of our program). Since they will add up to some constant $C \in \mathbb{N}$, they'll have a runtime of $C \in \Theta(1)$. Adding that to our runtime for our two loops, we get $\Theta(logn) + \Theta(1) \in \Theta(logn)$, our program's runtime.

Finally, no other input family is able to generate a runtime worse than our input family I_n because as stated in the beginning of this analysis, the only heap size dependent (non constant time) part of the program are the two sequential while loops. These while loops, as analysed above, each have a runtime of at worst $\Theta(logn)$ in the case that we traverse the entire height of the heap in each while loop, i.e., the most iterations of the while loop. So as a theta bound for the runtime, we'd have $2\Theta(logn) + C \in \Theta(logn)$, where $C \in \mathbb{N}$, like above, are the count of constant time operations in the program (also the runtime as they are constant time). Since the runtime for all heaps in I_n is in $\Theta(logn)$, we have that no other input family can be worse than I_n.

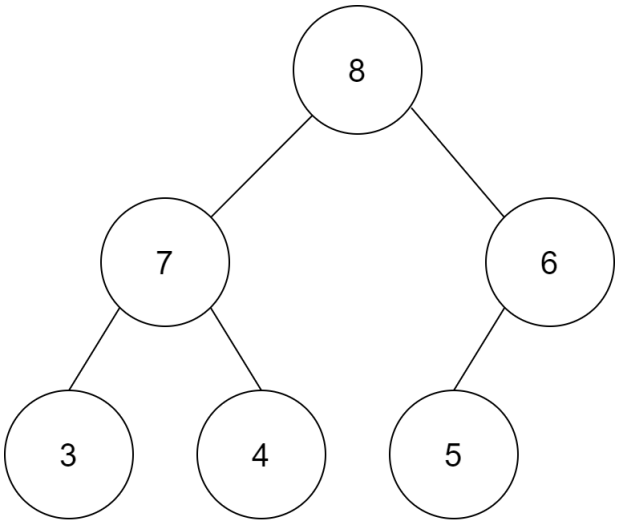
Question 5

(1)

Original max heap:

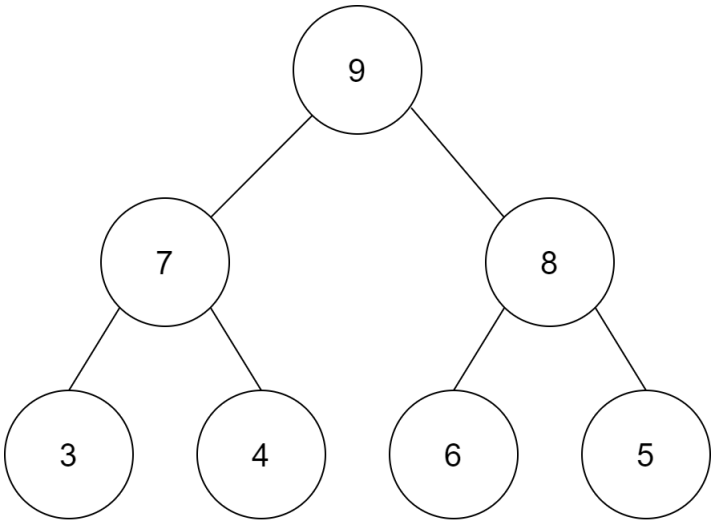


Intermediate heap, after ExtractMax:

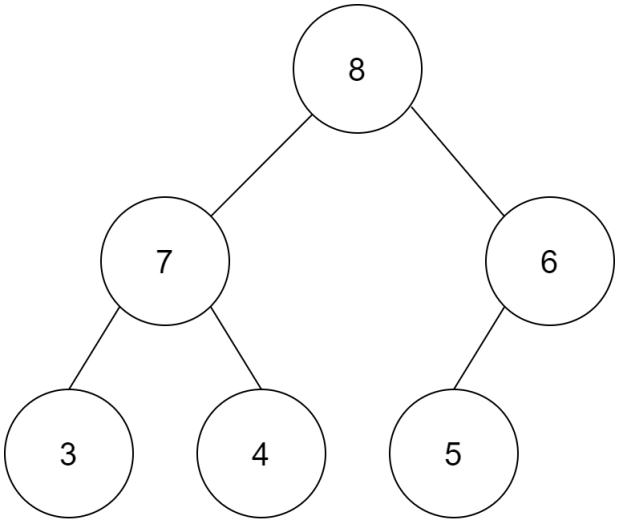


(2)

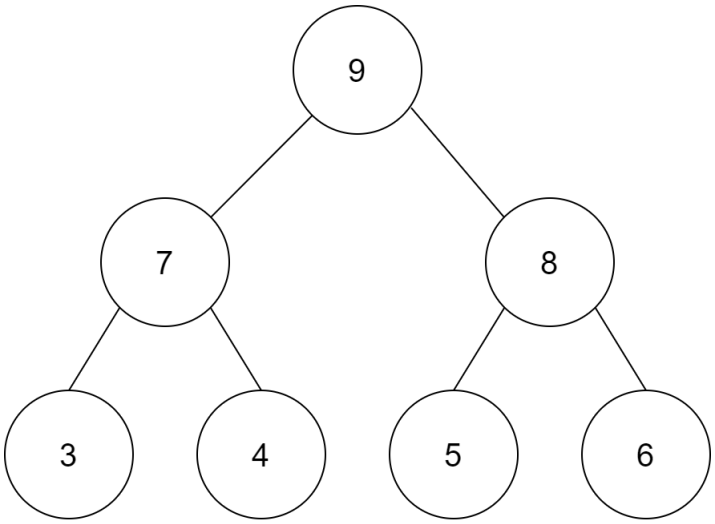
Original max heap:



Intermediate heap, after ExtractMax:



Final heap, after insert:



(3)

For an arbitrary heap H with no duplicates, we define our definition of it being a DIS max heap as:

$P(H) : MaxHeap(H) \wedge (\forall n \in \{N_1, N_2, ..., N_h\}, (n \text{ has a sibling}) \implies (n > \text{n's sibling}))$

where $MaxHeap(H)$ means that H is a max heap and $N_1, N_2, ..., N_h$ are nodes on the path from the root to the last node, excluding the root node and including the last node.

In other words:

H must be a max heap, and for all nodes on the path from the root to the last node, excluding the root node, $N_1, N_2, ..., N_h$ where the subscript stands for their depths and N_h is the last node, if they have a sibling node, they have to be greater than their sibling.

(4)

For an arbitrary max heap H_M with no duplicates, define:

$IsDIS(H_M) : H_M$ is a DIS max heap.

We want to prove that our description from (3) describes all DIS max heaps and does not hold for max heaps that are not DIS, in other words:

WTP: $\forall H_M, P(H_M) \iff IsDIS(H_M)$

Pf:

Let H_M be an arbitrary max heap with no duplicates.

Proving $P(H_M) \implies IsDIS(H_M)$:

Assume $P(H_M)$ stands.

If the height of H_M is 0, it will only have the root node, so extracting max and re-inserting will preserve the max heap. If the height of H_M is greater than 0, consider the following.

We'll refer to the nodes on the path from the root to the last node as $N_0, N_1, N_2, ..., N_h$ where h is the height of the tree, and N_i is the node on the path at depth i. So N_0 is the root node and N_h is the last node.

We'll also call the nodes that are the siblings of the above nodes $n_1, n_2, ..., n_h$ where n_i is the sibling of N_i. n_0 does not exist since N_0 is the root and n_h might not exist depending on the position of N_h. This means N_{i+1} is the parent of N_i and n_i.

When we perform an ExtractMax, the root node will first be replaced by the value of the last node N_h and the last node will be removed. Since we know that $P(H_M)$ stands, when we bubble down N_h from the root, it will follow the path from the root to the position of the last node.

This is because when we bubble down the path $N_1, N_2, ..., N_h$, we will swap with the larger of the two children as there are no duplicates. Since each node N_i on the path is larger than its sibling n_i, N_h will keep swapping with the values of N_i and so the bubbling down will follow that path down to the previous location of N_{h-1}. Then, $N_1, N_2, ..., N_{h-1}$ will all have been shifted up to the previous locations of $N_0, N_1, N_2, ..., N_{h-2}$. And if the heap has height 1, there is no path to bubble down along, so N_h will simply stay at the root before the next step.

If N_h was previously a left child, then N_{h-1} will no longer have any children, and if N_h was previously a right child, then N_h will be greater than n_h because of $P(H_M)$. Therefore, in both cases, after bubbling down and shifting the parent nodes up, N_h will stay at the previous location of N_{h-1}.

Then, when we insert the previous max back to the heap, it will first be inserted at the previous location of N_h and then bubbled up. This means that regardless of whether N_h was a left or right child, it will return to its original location after it swaps with the previous max.

Previous max is still the largest number of the heap since no other number was inserted, so it will bubble back up to the top along the path from the root to the previous last node (as that is where the previous max was inserted at). Since the nodes on the path

$N_1, N_2, ..., N_{h-1}$ was shifted up to the previous locations of $N_0, N_1, N_2, ..., N_{h-2}$ when we extracted max, they will now be shifted back down along the same path back to their original position when we bubble up the previous max.

Therefore, H_0 will return to the root node, $N_1, N_2, ..., N_{h-1}$ returned to their original positions, and N_h also returned to its original location, therefore H_M is preserved and $IsDIS(H_M)$.

Proving $IsDIS(H_M) \implies P(H_M)$:

We will prove its contrapositive $\neg P(H_M) \implies \neg IsDIS(H_M)$.

We'll refer to the nodes on the path from the root to the last node as $N_0, N_1, N_2, ..., N_h$ where h is the height of the tree, and N_i is the node on the path at depth i. So N_0 is the root node and N_h is the last node.

We'll also call the nodes that are the siblings of the above nodes $n_1, n_2, ..., n_h$ where n_i is the sibling of N_i. n_0 does not exist since N_0 is the root and n_h might not exist depending on the position of N_h. This means N_{i+1} is the parent of N_i and n_i.

Assume $\neg P(H_M)$, that means along the path from the root to the last node $N_1, N_2, ..., N_h$, there is at least one node such that its value is less than its sibling. Let N_x be that node.

We want to prove that this H_M is not a DIS max heap. To do this, we will trace the ExtractMax and Insert operations.

When we call ExtractMax, the last node will be removed and N_h will temporarily replace the root node, then it will be bubbled down. When it bubbles down, it will swap with the larger node of the two children. Therefore, it will follow the path of $N_1, N_2, ..., N_h$ until it reaches N_x. Since $N_x < n_x$, instead of swapping with N_x, N_h will swap with n_x and so N_h will no longer stay on the same path and stay at the previous location of n_x or bubble further down the subtree of n_x if applicable. This means that the old values of $N_1, N_2, ..., N_{x-1}$ will occupy $N_0, N_1, N_2, ..., N_{x-2}$ respectively, while the previous value of n_x will occupy N_{x-1} and $N_x, N_{x+1}, ..., N_{h-1}$ will stay at the same locations.

Then, when we insert the previous max N_0 back into the heap, it will be temporarily placed at the previous location of N_h and start to bubble up. Since N_0 is still the largest number of the heap, it will be bubbled all the way up along the same path. This means that $N_x, N_{x+1}, ..., N_{h-1}$ will now occupy $N_{x+1}, N_{x+2}, ..., N_h$ respectively while the old values of $N_1, N_2, ..., N_{x-1}$ will return to their original positions and n_x will now occupy the original location of N_x.

Therefore, the nodes $N_x, N_{x+1}, ..., N_{h-1}$ will now be in different locations as where they started, N_h will be at the previous location of n_x of lower down its subtree, and n_x will be at the previous position of N_x.

In the case that N_x is the last node N_h, then the nodes $N_x, ..., N_{h-1}$ that would have changed positions will no longer be applicable but N_x will still have swapped positions with n_x, so the heap will still be different. (as illustrated in 5.2)

Therefore, if $P(H_M)$ does not hold, then H_M is not a DIS max heap.

Since we have showed that $P(H_M) \implies IsDIS(H_M)$ and $IsDIS(H_M) \implies P(H_M)$, we have proved $P(H_M) \iff IsDIS(H_M)$.

■