Programming Methodology II

Homework 2a

Student: Adil Hydari, adil.hydari@rutgers.edu
Professor: Yao Liu

Problem 1

1. Prove or disprove the assertions below.

   (a) $2^{2n} = O(2^n)$
   To find if $2^{2n} = O(2^n)$, we analyze the growth rates of the functions.
   Def. of O:
   A function $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that:

   $$0 \leq f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

   Let $f(n) = 2^{2n}$ and $g(n) = 2^n$.

   $$2^{2n} = (2^2)^n = 4^n$$

   We need to find constants $c > 0$ and n_0 such that:

   $$4^n \leq c \cdot 2^n \quad \forall n \geq n_0$$

   Dividing both sides by 2^n:

   $$2^n \leq c \quad \forall n \geq n_0$$

   However, 2^n is not bounded by anything, so it will continue to increase as n increases. This means that there is no constant c exists that satisfies the inequality.
   Therefore:
   2^{2n} is $\neq O(2^n)$.

   (b) $\max(100n^2, 50n^3) = \Omega(n^3)$
   To find if $\max(100n^2, 50n^3) = \Omega(n^3)$, we can use the Def. of Omega.
   $f(n)$ is $\Omega(g(n))$ if there exist positive constants c and n_0 such that (from example):

   $$f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$$

   $f(n) = \max(100n^2, 50n^3)$ and $g(n) = n^3$.
   When $n \geq 2$:

   $$50n^3 \geq 100n^2$$

   Thus, $n \geq 2$:

   $$\max(100n^2, 50n^3) = 50n^3$$

   If we choose $c = 50$ and $n_0 = 2$:

   $$50n^3 \geq 50n^3 \quad \forall n \geq 2$$

   Therefore:
   $\max(100n^2, 50n^3) = \Omega(n^3)$.

Problem 2

Consider the following functions:

$$\begin{aligned}
f_1(n) &= n, \\
f_2(n) &= n \cdot \log n, \\
f_3(n) &= \begin{cases} n \cdot \log n & \text{if } n < 10, \\ n & \text{if } n \geq 10, \end{cases} \\
f_4(n) &= n^2, \\
f_5(n) &= 2^n, \\
f_6(n) &= 2^{2n}.
\end{aligned}$$

Please state if the following assertions are True or False.

(a) $f_1 = O(f_2)$

(b) $f_1 = O(f_3)$

(c) $f_1 = \Theta(f_3)$

(d) $f_2 = \Theta(f_3)$

(e) $f_1 = O(f_4)$

(f) $f_2 = O(f_3)$

(g) $f_3 = O(f_1)$

(h) $f_4 = O(f_2)$

(i) $f_4 = O(f_5)$

(j) $f_6 = O(f_5)$

Solutions:

(a) $f_1 = O(f_2)$

To find if $f_1(n) = O(f_2(n))$, we use the Def. of Big-O:

A function $f(n) = O(g(n))$ if there are positive constants c and n_0 such that:

$$0 \leq f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

$f(n) = n$ and $g(n) = n \log n$.

Find constants $c > 0$ and n_0 such that:

$$n \leq c \cdot n \log n \quad \forall n \geq n_0$$

Dividing both sides by n:

$$1 \leq c \cdot \log n \quad \forall n \geq n_0$$

If we choose $c = 1$ and $n_0 = 2$ (since the log of 2 is 1):

$$1 \leq 1 \cdot \log n \quad \forall n \geq 2$$

Therefore:

$f_1(n) = O(f_2(n))$ is True.

(b) $f_1 = O(f_3)$

$$f_3(n) = \begin{cases} n \cdot \log n & \text{if } n < 10, \\ n & \text{if } n \geq 10. \end{cases}$$

For $n \geq 10$:

$$f_3(n) = n$$

Thus, $f_1(n) = n \leq c \cdot n$ for $c \geq 1$.

For $n < 10$:

$$f_3(n) = n \cdot \log n$$

Since $n \cdot \log n \geq n$ for $n \geq 2$:

$$n \leq c \cdot n \log n$$

Choose $c = 1$, as $n \leq n \log n$ for $n \geq 2$. For $n = 1$, $\log 1 = 0$, but since $f_3(1) = 0$, we can choose a larger c.

Therefore:

$f_1(n) = O(f_3(n))$ is True.

(c) $f_1 = \Theta(f_3)$

To find $f_1(n) = \Theta(f_3(n))$, both $f_1(n) = O(f_3(n))$ and $f_1(n) = \Omega(f_3(n))$ must be true.

From part B, we have $f_1(n) = O(f_3(n))$.

Now, find if $f_1(n) = \Omega(f_3(n))$:

$f(n) = \Omega(g(n))$ if there are positive constants c and n_0 such that:

$$f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$$

For $n \geq 10$:

$$f_3(n) = n \implies f_1(n) = n \geq c \cdot n \quad \forall n \geq n_0$$

Choose $c = 1$.

For $n < 10$:

$$f_3(n) = n \cdot \log n$$

$f_1(n) = n$ and $n \cdot \log n > n$ for $n > 1$.

Thus, there is no constant $c > 1$ that satisfies $n \geq c \cdot n \log n$ for $n < 10$.

Therefore:

$f_1(n) \neq \Omega(f_3(n))$, $f_1(n) = \Theta(f_3(n))$ is False

(d) $f_2 = \Theta(f_3)$

To find if $f_2(n) = \Theta(f_3(n))$, both $f_2(n) = O(f_3(n))$ and $f_2(n) = \Omega(f_3(n))$ must be true.

1. $f_2(n) = O(f_3(n))$**:**

For $n \geq 10$:

$$f_3(n) = n$$

$$f_2(n) = n \log n \leq c \cdot n \quad \text{needs} \quad \log n \leq c$$

However, $\log n$ grows without bound as $n \to \infty$, so no constant c exists for this.

2. $f_2(n) = \Omega(f_3(n))$**:**

For $n \geq 10$:

$$f_3(n) = n$$

$$f_2(n) = n \log n \geq c \cdot n \quad \text{for} \quad c = 1, \quad \forall n \geq 10$$

Therefore:

$f_2(n) \neq O(f_3(n))$. $f_2(n) = \Theta(f_3(n))$ is False.

(e) $f_1 = O(f_4)$

$f_1(n) = O(f_4(n))$.

Using the Big-O def.:

Constants $c > 0$ and n_0 such that:

$$n \leq c \cdot n^2 \quad \forall n \geq n_0$$

Divide both sides by n :

$$1 \leq c \cdot n \quad \forall n \geq n_0$$

Choose $c = 1$ and $n_0 = 1$:

$$1 \leq 1 \cdot n \quad \forall n \geq 1$$

Which holds true for $n \geq 1$.

Therefore:

$f_1(n) = O(f_4(n))$ is True.

(f) $f_2 = O(f_3)$

$f_2(n) = O(f_3(n))$.

Using the Big-O def.:

Constants $c > 0$ and n_0 such that:

$$n \log n \leq c \cdot f_3(n) \quad \forall n \geq n_0$$

For $n < 10$:

$$f_3(n) = n \log n$$

Thus:

$$n \log n \leq c \cdot n \log n \quad \text{is satisfied for any } c \geq 1$$

For $n \geq 10$:

$$f_3(n) = n$$

Thus:

$$n \log n \leq c \cdot n \quad \Rightarrow \quad \log n \leq c \quad \forall n \geq 10$$

$\log n$ grows without bound as n increases. Therefore, no constant c exists that satisfies $\log n \leq c$ for all $n \geq n_0$.

Therefore:

There is no constant c that satisfies the Big-O condition. $n \geq n_0$, $f_2(n) = O(f_3(n))$ is False.

(g) $f_3 = O(f_1)$

$f_3(n) = O(f_1(n))$.

Using the Big-O def.:

Constants $c > 0$ and n_0 such that:

$$f_3(n) \leq c \cdot f_1(n) \quad \forall n \geq n_0$$

For $n < 10$:

$$f_3(n) = n \log n$$

The maximum value occurs at $n = 10$:

$$f_3(10) = 10 \cdot \log 10 \approx 10 \cdot 1 = 10$$

For $n \geq 10$:

$$f_3(n) = n$$

Thus:

$$n \leq c \cdot n \quad \text{for } c \geq 1$$

$c = 10$ and $n_0 = 1$

$$f_3(n) \leq 10 \cdot n \quad \forall n \geq 1$$

Therefore:

$f_3(n) = O(f_1(n))$ is True.

(h) $f_4 = O(f_2)$

$f_4(n) = O(f_2(n))$.

Using the Big-O def.:

Constants $c > 0$ and n_0 such that:

$$n^2 \leq c \cdot n \log n \quad \forall n \geq n_0$$

Divide both sides by n:

$$n \leq c \cdot \log n \quad \forall n \geq n_0$$

As n increases, n grows much faster than $\log n$. So:

$$\lim_{n\to\infty} \frac{n}{\log n} = \infty$$

This shows that n eventually exceeds any multiple of $\log n$, making the inequality $n \leq c \cdot \log n$ impossible to satisfy for a large n.

Therefore:

No such constant c exists to satisfy the Big-O. $n \geq n_0$, $f_4(n) = O(f_2(n))$ is False.

(i) $f_4 = O(f_5)$

$f_4(n) = O(f_5(n))$.

Using the Big-O def.:

Constants $c > 0$ and n_0 such that:

$$n^2 \leq c \cdot 2^n \quad \forall n \geq n_0$$

For $n \geq 5$:

$$2^n \geq n^3 > n^2$$

We can choose $c = 1$ and $n_0 = 5$:

$$n^2 \leq 2^n \quad \forall n \geq 5$$

Therefore:

$f_4(n) = O(f_5(n))$ is True.

(j) $f_6 = O(f_5)$

$f_6(n) = O(f_5(n))$.

Using the Big-O def.:

Constants $c > 0$ and n_0 such that:

$$2^{2n} \leq c \cdot 2^n \quad \forall n \geq n_0$$

Then:

$$2^{2n} = (2^n)^2 = 4^n \quad \text{and} \quad 4^n \leq c \cdot 2^n \quad \Rightarrow \quad 2^n \leq c \quad \forall n \geq n_0$$

As n increases, 2^n grows exponentially without bound. No constant c exists that satisfies $2^n \leq c$ for $n \geq n_0$.

Therefore:

$n \geq n_0$, $f_6(n) = O(f_5(n))$ is False.

Problem 3

SELECTIONSORT is another algorithm for sorting numbers in an array $A[1:n]$:

```
SELECTIONSORT(A, n)
for i =1 to n - 1
    minIndex = i
    for j = i + 1 to n
        if A[j] < A[minIndex]
            minIndex = j
    if minIndex \neq i
        swap A[i] with A[minIndex]
```

(a) Show that the running time of SELECTIONSORT is $O(n^2)$.

Big-O Def. :

A function $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that:

$$f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

Analyzing the Algo. :

SELECTIONSORT consists of two nested loops:

(a) The outer loop runs from $i = 1$ to $n - 1$, executing $n - 1$ times.

(b) The inner loop runs from $j = i + 1$ to n, executing $n - i$ times for each i.

The total # of Comparisons is the sum of the inner loop executions:

$$\sum_{i=1}^{n-1}(n-i) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$$

Big-O Bound:

$$f(n) = \frac{n(n-1)}{2} \leq \frac{n^2}{2} \leq c \cdot n^2 \quad \text{for } c = \frac{1}{2} \text{ and } n_0 = 1$$

Therefore:

The running time of SELECTIONSORT is $O(n^2)$.

(b) Show that the running time of SELECTIONSORT is $\Omega(n^2)$.

Big-Omega Def. :

Function $f(n) = \Omega(g(n))$ if there are positive constants c and n_0 such that:

$$f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$$

Analyzing the Algo. :

SELECTIONSORT has two nested loops with a total of $\frac{n(n-1)}{2}$ comparisons.

Big-Omega Bound:

$$f(n) = \frac{n(n-1)}{2} \geq \frac{n(n-1)}{2} \geq \frac{n^2}{4} \quad \text{for } n \geq 2$$

Choose $c = \frac{1}{4}$ and $n_0 = 2$:

$$f(n) \geq \frac{1}{4} \cdot n^2 \quad \forall n \geq 2$$

Therefore:

The running time of SELECTIONSORT is $\Omega(n^2)$.

(c) Show that the running time of SELECTIONSORT is $\Theta(n^2)$.

Big-Theta Def. :

Function $f(n) = \Theta(g(n))$ if it is both $O(g(n))$ and $\Omega(g(n))$. And from parts (a) and (b), we have:

$$f(n) = O(n^2) \quad \text{and} \quad f(n) = \Omega(n^2)$$

Therefore:

The running time of SELECTIONSORT is $\Theta(n^2)$.

Problem 4

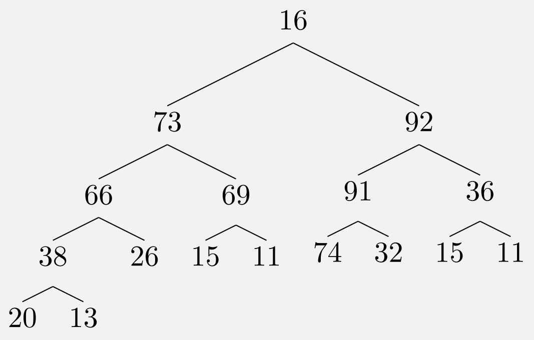
Is an array that is in sorted (increasing) order a min-heap? Please provide your reasoning.
To determine whether a sorted (increasing order) array satisfies the properties of a min-heap, we must recall the definition of a min-heap.
Def. of a Min-Heap:
For every node i other than the root node, the value of $i \geq$ parent. The parent of a node at i is at $\lfloor \frac{i}{2} \rfloor$. The children of the node at i are at $2i$ and $2i + 1$.
For every parent node $A[i]$, Greater than or Equal to $A[2i]$ and $A[2i + 1]$
Therefore:
Since a sorted and increasing order array displays that every parent node is less than or equal to its child nodes, the array satisfies min-heap.

Problem 5

Please illustrate the process of MAX-HEAPIFY(A,1) on array $A = [16, 73, 92, 66, 69, 91, 36, 38, 20, 13, 26, 74, 32, 15, 11]$ using the "binary tree view".
Initial Array:

$$A = [16, 73, 92, 66, 69, 91, 36, 38, 20, 13, 26, 74, 32, 15, 11]$$

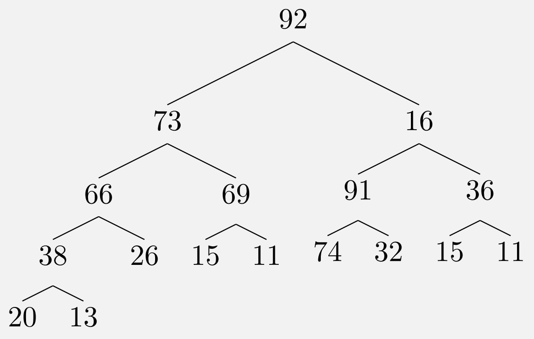
Initial Binary Tree Representation:



Step 1: Compare $A[1] = 16$ with its children $A[2] = 73$ and $A[3] = 92$.
The largest value among the parent and children is 92 at index 3. Swap 16 with 92.
Array After Swap:

$$A = [92, 73, 16, 66, 69, 91, 36, 38, 20, 13, 26, 74, 32, 15, 11]$$

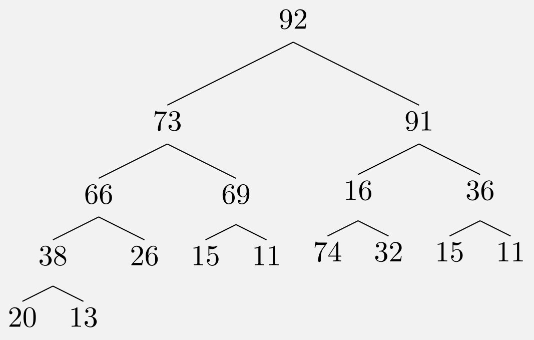
Binary Tree After Step 1:



Step 2: Heapify the Subtree at Index 3 (Value 16).
Compare $A[3] = 16$ with its children $A[6] = 91$ and $A[7] = 36$. Swap 16 with 91.
Array After Swap:

$$A = [92, 73, 91, 66, 69, 16, 36, 38, 20, 13, 26, 74, 32, 15, 11]$$

Binary Tree After Step 2:

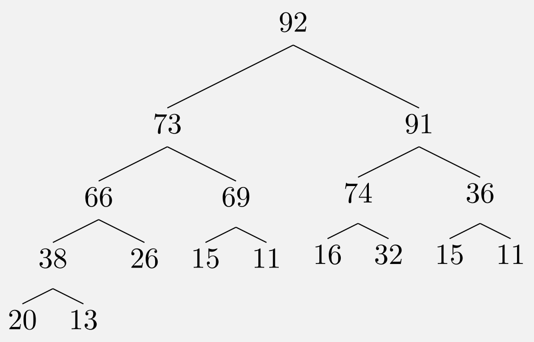


Step 3: Heapify the Subtree at Index 6 (Value 16).
Compare $A[6] = 16$ with its children $A[12] = 74$ and $A[13] = 32$. Swap 16 with 74.

Final Array After All Swaps:

$$A = [92, 73, 91, 66, 69, 74, 36, 38, 20, 13, 26, 16, 32, 15, 11]$$

Final Binary Tree After Step 3:



The largest element 92 is at the root, and all parent nodes being greater than or equal to their respective children, the max-heap build is finished.

Problem 6

Problem 6. (10 points)
Please illustrate the process of BUILD-MAX-HEAP on array $A = [7, 40, 65, 91, 42, 69, 97, 11, 9, 28, 25, 19, 35, 2, 48]$ by showing A after each iteration of the for loop (not the "binary tree view").

```
BUILD-MAX-HEAP(A, n)
1 A.heap_size = n
2 for i =  [n/2] down to 1
3     MAX-HEAPIFY(A, i)
```

$$A = [7, 40, 65, 91, 42, 69, 97, 11, 9, 28, 25, 19, 35, 2, 48]$$

(i) **Iteration $i = 7$:**
Action: `MAX-HEAPIFY(A, 7)`.

$$\text{Parent} = A[7] = 97$$
$$\text{Left Child} = A[14] = 2$$
$$\text{Right Child} = A[15] = 48$$

Since $97 \geq 2$ and $97 \geq 48$, no swaps are needed.

Array After Iteration 7:

$$A = [7, 40, 65, 91, 42, 69, 97, 11, 9, 28, 25, 19, 35, 2, 48]$$

(ii) **Iteration $i = 6$:**
Action: `MAX-HEAPIFY(A, 6)`.

$$\begin{aligned}\text{Parent} &= A[6] = 69\\ \text{Left Child} &= A[12] = 19\\ \text{Right Child} &= A[13] = 35\end{aligned}$$

Since $69 \geq 19$ and $69 \geq 35$, no swaps are needed.

Array After Iteration 6:

$$A = [7, 40, 65, 91, 42, 69, 97, 11, 9, 28, 25, 19, 35, 2, 48]$$

(iii) **Iteration $i = 5$:**
Action: `MAX-HEAPIFY(A, 5)`.

$$\begin{aligned}\text{Parent} &= A[5] = 42\\ \text{Left Child} &= A[10] = 28\\ \text{Right Child} &= A[11] = 25\end{aligned}$$

Since $42 \geq 28$ and $42 \geq 25$, no swaps are needed.

Array After Iteration 5:

$$A = [7, 40, 65, 91, 42, 69, 97, 11, 9, 28, 25, 19, 35, 2, 48]$$

(iv) **Iteration $i = 4$:**
Action: `MAX-HEAPIFY(A, 4)`.

$$\begin{aligned}\text{Parent} &= A[4] = 91\\ \text{Left Child} &= A[8] = 11\\ \text{Right Child} &= A[9] = 9\end{aligned}$$

Since $91 \geq 11$ and $91 \geq 9$, no swaps are needed.

Array After Iteration 4:

$$A = [7, 40, 65, 91, 42, 69, 97, 11, 9, 28, 25, 19, 35, 2, 48]$$

(v) **Iteration $i = 3$:**
Action: `MAX-HEAPIFY(A, 3)`.

$$\begin{aligned}\text{Parent} &= A[3] = 65\\ \text{Left Child} &= A[6] = 69\\ \text{Right Child} &= A[7] = 97\end{aligned}$$

The largest among $65, 69, 97$ is 97 at index 7. Swap $A[3]$ with $A[7]$.

Array After Swap:

$$A = [7, 40, 97, 91, 42, 69, 65, 11, 9, 28, 25, 19, 35, 2, 48]$$

Heapify Subtree at $i = 7$:

$$\text{Parent} = A[7] = 65$$
$$\text{Left Child} = A[14] = 2$$
$$\text{Right Child} = A[15] = 48$$

Since $65 \geq 2$ and $65 \geq 48$, no further swaps are needed.

Array After Iteration 3:

$$A = [7, 40, 97, 91, 42, 69, 65, 11, 9, 28, 25, 19, 35, 2, 48]$$

(vi) **Iteration $i = 2$:**
Action: `MAX-HEAPIFY(A, 2)`.

$$\text{Parent} = A[2] = 40$$
$$\text{Left Child} = A[4] = 91$$
$$\text{Right Child} = A[5] = 42$$

The largest among $40, 91, 42$ is 91 at index 4. Swap $A[2]$ with $A[4]$.

Array After Swap:

$$A = [7, 91, 97, 40, 42, 69, 65, 11, 9, 28, 25, 19, 35, 2, 48]$$

Heapify Subtree at $i = 4$:

$$\text{Parent} = A[4] = 40$$
$$\text{Left Child} = A[8] = 11$$
$$\text{Right Child} = A[9] = 9$$

Since $40 \geq 11$ and $40 \geq 9$, no further swaps are needed.

Array After Iteration 2:

$$A = [7, 91, 97, 40, 42, 69, 65, 11, 9, 28, 25, 19, 35, 2, 48]$$

(vii) **Iteration $i = 1$:**
Action: `MAX-HEAPIFY(A, 1)`.

$$\text{Parent} = A[1] = 7$$
$$\text{Left Child} = A[2] = 91$$
$$\text{Right Child} = A[3] = 97$$

The largest among $7, 91, 97$ is 97 at index 3. Swap $A[1]$ with $A[3]$.

Array After Swap:

$$A = [97, 91, 7, 40, 42, 69, 65, 11, 9, 28, 25, 19, 35, 2, 48]$$

Heapify Subtree at $i = 3$:

$$\text{Parent} = A[3] = 7$$
$$\text{Left Child} = A[6] = 69$$
$$\text{Right Child} = A[7] = 65$$

The largest among $7, 69, 65$ is 69 at index 6. Swap $A[3]$ with $A[6]$.

Array After Swap:

$$A = [97, 91, 69, 40, 42, 7, 65, 11, 9, 28, 25, 19, 35, 2, 48]$$

Heapify Subtree at $i = 6$:

$$\text{Parent} = A[6] = 7$$
$$\text{Left Child} = A[12] = 19$$
$$\text{Right Child} = A[13] = 35$$

The largest among $7, 19, 35$ is 35 at index 13. Swap $A[6]$ with $A[13]$.

Array After Swap:

$$A = [97, 91, 69, 40, 42, 35, 65, 11, 9, 28, 25, 19, 7, 2, 48]$$

Heapify Subtree at $i = 13$:

$$\text{Parent} = A[13] = 7$$
$$\text{Left Child} = A[26] = \text{N/A}$$
$$\text{Right Child} = A[27] = \text{N/A}$$

No more children, no further swaps are needed.

Final Array After BUILD-MAX-HEAP:

$$A = [97, 91, 69, 40, 42, 35, 65, 11, 9, 28, 25, 19, 7, 2, 48]$$

Problem 7

In the for loop of the BUILD-MAX-HEAP procedure above, index i decreases from $\lfloor \frac{n}{2} \rfloor$ to 1. Can we modify it so that i increases from 1 to $\lfloor \frac{n}{2} \rfloor$ while still ensuring a correct max heap is built? Please provide your explanation.

Explanation:

The BUILD-MAX-HEAP is made to construct a max heap from an unordered array by making sure that each subtree satisfies the max-heap properties. The standard implementation processes the nodes in a "bottom-up" way, starting from the last non-leaf node and moving upwards to the root.

1. **Heapify Deps. :**
   - When heapifying a node, it is essential that its children are already max heaps. Processing nodes from bottom to the top makes sure that by the time a parent

node is heapified, all of its subtrees have already been heapified.

- If we were to process nodes from top to bottom, we would end up having to heapify parent nodes before the children. This means that we would get situations where the children are not yet max heaps.

Conclusion:

Modifying the ‘BUILD-MAX-HEAP‘ procedure to iterate i from 1 to $\lfloor \frac{n}{2} \rfloor$ would change the heapify order, leading to an incorrect max heap. Therefore, the loop must continue to decrement i from $\lfloor \frac{n}{2} \rfloor$ down to 1 to make sure the max heap happens correctly.

Problem 8

Problem 8. (10 points)
Using the example in the slides as a model, please illustrate the process of HEAPSORT with $A = [88, 43, 76, 22, 25, 17, 23]$. Here, `BUILD-MAX-HEAP` has already been executed, and A is a max-heap.
Max heap array:

$$A = [88, 43, 76, 22, 25, 17, 23]$$

Algo. PsuedoCode:

```
HEAPSORT(A, n)
1 BUILD-MAX-HEAP(A, n)
2 for i = n downto 2
3     exchange A[1] with A[i]
4     A.heap_size = A.heap_size - 1
5     MAX-HEAPIFY(A, 1)
```

HEAPSORT:

(i) **Iteration 1 ($i = 7$):**

Action: Swap the root $A[1] = 88$ with the final element $A[7] = 23$.

$$A = [23, 43, 76, 22, 25, 17, 88]$$

Heap Size Reduction: heap_size = 6

Apply `MAX-HEAPIFY(A, 1)`:

- Parent: $A[1] = 23$
- Left Child: $A[2] = 43$
- Right Child: $A[3] = 76$

Largest Value: $A[3] = 76$

Action: Swap $A[1]$ with $A[3]$.

$$A = [76, 43, 23, 22, 25, 17, 88]$$

Heap After Iteration 1:

$$A = [76, 43, 23, 22, 25, 17, 88]$$

(ii) **Iteration 2 ($i = 6$):**

Action: Swap the root $A[1] = 76$ with the final element in the heap $A[6] = 17$.

$$A = [17, 43, 23, 22, 25, 76, 88]$$

Heap Size Reduction: heap_size = 5

Apply `MAX-HEAPIFY(A, 1)`:

- Parent: $A[1] = 17$
- Left Child: $A[2] = 43$
- Right Child: $A[3] = 23$

Largest Value: $A[2] = 43$

Action: Swap $A[1]$ with $A[2]$.

$$A = [43, 17, 23, 22, 25, 76, 88]$$

Heap After Iteration 2:

$$A = [43, 17, 23, 22, 25, 76, 88]$$

(iii) **Iteration 3 ($i = 5$):**

Action: Swap the root $A[1] = 43$ with the final element in the heap $A[5] = 25$.

$$A = [25, 17, 23, 22, 43, 76, 88]$$

Heap Size Reduction: heap_size = 4

Apply `MAX-HEAPIFY(A, 1)`:

- Parent: $A[1] = 25$
- Left Child: $A[2] = 17$
- Right Child: $A[3] = 23$

Largest Value: $A[1] = 25$ (No swap needed)

Heap After Iteration 3:

$$A = [25, 17, 23, 22, 43, 76, 88]$$

(iv) **Iteration 4 ($i = 4$):**

Action: Swap the root $A[1] = 25$ with the final element in the heap $A[4] = 22$.

$$A = [22, 17, 23, 25, 43, 76, 88]$$

Heap Size Reduction: heap_size = 3

Apply `MAX-HEAPIFY(A, 1)`:

- Parent: $A[1] = 22$
- Left Child: $A[2] = 17$
- Right Child: $A[3] = 23$

Largest Value: $A[3] = 23$

Action: Swap $A[1]$ with $A[3]$.

$$A = [23, 17, 22, 25, 43, 76, 88]$$

Heap After Iteration 4:

$$A = [23, 17, 22, 25, 43, 76, 88]$$

(v) **Iteration 5 ($i = 3$):**

Action: Swap the root $A[1] = 23$ with the final element in the heap $A[3] = 22$.

$$A = [22, 17, 23, 25, 43, 76, 88]$$

Heap Size Reduction: heap_size = 2

Apply `MAX-HEAPIFY(A, 1)`:

- Parent: $A[1] = 22$
- Left Child: $A[2] = 17$
- Right Child: $A[3] =$ N/A (heap size is 2)

Largest Value: $A[1] = 22$ (No swap)

Heap After Iteration 5:

$$A = [22, 17, 23, 25, 43, 76, 88]$$

(vi) **Iteration 6 ($i = 2$):**

Action: Swap the root $A[1] = 22$ with the final element in the heap $A[2] = 17$.

$$A = [17, 22, 23, 25, 43, 76, 88]$$

Heap Size Reduction: heap_size = 1

Apply `MAX-HEAPIFY(A, 1)`:

- Parent: $A[1] = 17$
- Left Child: $A[2] =$ N/A (heap size is 1)

- Right Child: $A[3] = \text{N/A}$

Largest Value: $A[1] = 17$ (No swap needed)

Heap After Iteration 6:

$$A = [17, 22, 23, 25, 43, 76, 88]$$

Final Sorted Array:

$$A = [17, 22, 23, 25, 43, 76, 88]$$

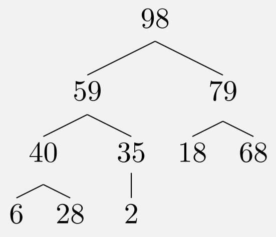
Problem 9

Problem 9. (10 points)
Suppose that the objects in a max-priority queue are just keys. Illustrate the operation of `MAX-HEAP-INSERT(A, 70)` on the heap $A = [98, 59, 79, 40, 35, 18, 68, 6, 28, 2]$ using the "binary tree view".

Given Max-Heap Array:

$$A = [98, 59, 79, 40, 35, 18, 68, 6, 28, 2]$$

Initial Binary Tree Representation:



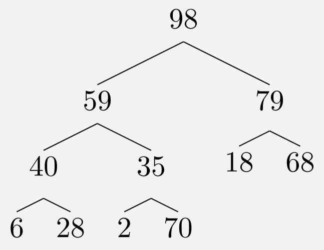
Function Call: `MAX-HEAP-INSERT(A, 70)`

(i) **Insert the New Key at the End of the Array:**

- Increase the heap size by 1.
- Insert 70 at index 11.

$$A = [98, 59, 79, 40, 35, 18, 68, 6, 28, 2, 70]$$

Binary Tree After Insertion:

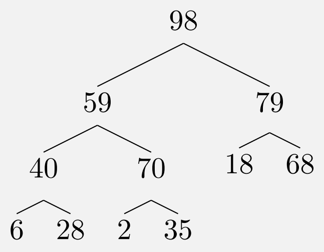


(ii) **Move the Inserted Key from top to bottom:**

- Step 1: Compare 70 with its parent.
- Index of 70: 11
- Parent Index: $\lfloor \frac{11}{2} \rfloor = 5$
- Parent Key: $A[5] = 35$
- Comparison: $70 > 35$
- Action: Swap 70 with 35.

$$A = [98, 59, 79, 40, 70, 18, 68, 6, 28, 2, 35]$$

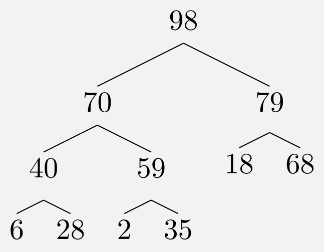
Binary Tree After First Swap:



- Step 2: Compare 70 with its new parent.
- Index of 70: 5
- Parent Index: $\lfloor \frac{5}{2} \rfloor = 2$
- Parent Key: $A[2] = 59$
- Comparison: $70 > 59$
- Action: Swap 70 with 59.

$$A = [98, 70, 79, 40, 59, 18, 68, 6, 28, 2, 35]$$

Binary Tree After Second Swap:

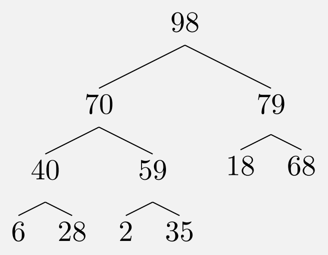


- Step 3: Compare 70 with its new parent.
- Index of 70: 2
- Parent Index: $\lfloor \frac{2}{2} \rfloor = 1$
- Parent Key: $A[1] = 98$
- Comparison: $70 < 98$
- Action: No swap needed. Heapify complete.

Final Array After `MAX-HEAP-INSERT`:

$$A = [98, 70, 79, 40, 59, 18, 68, 6, 28, 2, 35]$$

Final Binary Tree Structure:



The final max-heap array is:

$$A = [98, 70, 79, 40, 59, 18, 68, 6, 28, 2, 35]$$