2025 Algorithm hw1

109062327 Hsu, Hung-Che

March 2025

Problem 1

Let $f(n)$, $g(n)$, and $h(n)$ be functions that are positive for all sufficiently large n.

(a) Prove that $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Proof:

($\Rightarrow$) **Assume $f(n) = \Theta(g(n))$:**
By the definition of Θ, there exist positive constants c_1, c_2 and an integer n_0 such that for all $n \geq n_0$,

$$c_1 \dot{g}(n) \leq f(n) \leq c_2 \dot{g}(n).$$

Thus, taking $c = c_2$, we have $f(n) \leq c\, g(n)$, which shows that $f(n) = O(g(n))$. Similarly, taking $c = c_1$, we have

$$f(n) \geq c_1\, g(n),$$

which implies $f(n) = \Omega(g(n))$.

($\Leftarrow$) **Assume $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$:**
Since $f(n) = O(g(n))$, there exist a constant $c_2 > 0$ and an integer n_1 such that for all $n \geq n_1$,

$$f(n) \leq c_2\, g(n).$$

Also, since $f(n) = \Omega(g(n))$, there exist a constant $c_1 > 0$ and an integer n_2 such that for all $n \geq n_2$,

$$f(n) \geq c_1\, g(n).$$

Let $n_0 = \max\{n_1, n_2\}$, then for all $n \geq n_0$, we have

$$c_1\, g(n) \leq f(n) \leq c_2\, g(n),$$

which is precisely the definition of $f(n) = \Theta(g(n))$.

Thus, we have proven that

$$f(n) = \Theta(g(n)) \iff \Big(f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))\Big).$$

(b) Prove that if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

Since $f(n) = O(g(n))$, there exists a constant $c_1 > 0$ and an integer n_1 such that for all $n \geq n_1$,

$$f(n) \leq c_1\, g(n).$$

Similarly, since $g(n) = O(h(n))$, there exists a constant $c_2 > 0$ and an integer n_2 such that for all $n \geq n_2$,

$$g(n) \leq c_2\, h(n).$$

Let $n_0 = \max\{n_1, n_2\}$, then for all $n \geq n_0$,

$$f(n) \leq c_1\, g(n) \leq c_1 c_2\, h(n).$$

This shows that $f(n) = O(h(n))$. □

Problem 2: Time Complexity Analysis of Binary Search

Given a sorted list $a_1, a_2, \ldots, a_n$ (with $a_1 \leq a_2 \leq \cdots \leq a_n$), we analyze the time complexity of the binary search to determine whether a given number X is present in the list. For simplicity, assume that $n = 2^k - 1$, where k is a positive integer.

Best-case Time Complexity

If the target X is equal to the middle element in the first comparison, then only one comparison is needed. Therefore, the best-case time complexity is

$$O(1).$$

Worst-case Time Complexity

The binary search halves the search interval at each step. In the worst case, the number of comparisons equals the height of the binary search tree. For $n = 2^k - 1$, the height is exactly $k \approx \log_2(n+1)$, so the worst-case time complexity is

$$O(\log_2 n).$$

Average-case Time Complexity

Let $T(n)$ denote the average number of comparisons required to search among n elements. Since each comparison approximately halves the search interval, we obtain the recurrence

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

Solving this recurrence yields

$$T(n) = O(\log_2 n).$$

Alternatively, by analyzing the average depth of the nodes in the binary search tree, we can show that the average number of comparisons is $\Theta(\log_2 n)$. Thus, the complexity of the average case time is

$$O(\log_2 n).$$

Problem 3: Lower Bound for Finding the Second Largest Element

Show that finding the second largest element in a list of n numbers requires at least

$$n - 2 + \lceil \log_2 n \rceil$$

comparisons.

Proof Idea:

1. First, use the tournament method (i.e., pairwise comparisons) to find the largest element, which requires $n - 1$ comparisons.
2. In the tournament process, the largest element is compared against a series of opponents. In a balanced tournament tree, the height is at least $\lceil \log_2 n \rceil$, meaning that the maximum element has been compared with at least $\lceil \log_2 n \rceil$ elements.
3. The second largest element must be among these $\lceil \log_2 n \rceil$ candidates. To find the largest among these candidates, at least $\lceil \log_2 n \rceil - 1$ comparisons are necessary.

Therefore, the total number of comparisons is at least

$$(n - 1) + (\lceil \log_2 n \rceil - 1) = n - 2 + \lceil \log_2 n \rceil.$$

Problem 4: Heap Sort in a Sequence of Nine Numbers

Given the sequence:

$$1,\ 3,\ 5,\ 7,\ 9,\ 8,\ 6,\ 4,\ 2,$$

we are required to:

- (10%) Construct the initial heap (using a max-heap).
- (10%) Show the heap states during the heap sort process.

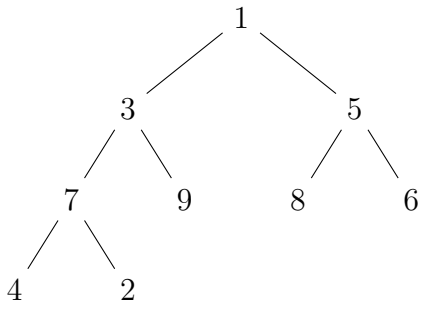
Step 1: Building the Initial Max-Heap

Consider the array representation (using 1-indexing):

$$A = [1,\ 3,\ 5,\ 7,\ 9,\ 8,\ 6,\ 4,\ 2].$$

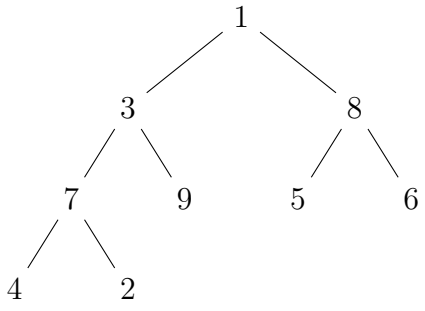
Following the `build-heap` procedure, we start from the last internal node at $\lfloor n/2 \rfloor = 4$ and apply `heapify` for each node.

(i) Node 4:
The value at node 4 is 7, with children $A[8] = 4$ and $A[9] = 2$. Since $7 \geq 4$ and $7 \geq 2$, no adjustment is needed.



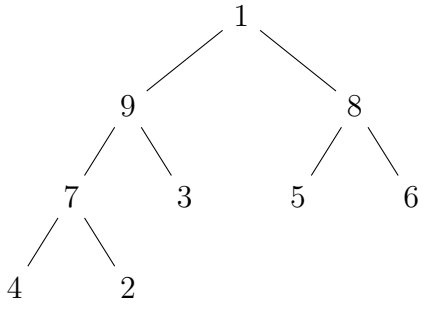
(ii) Node 3:
The value at node 3 is 5, with children $A[6] = 8$ and $A[7] = 6$. The larger child is $A[6] = 8$, and since $8 > 5$, swap them:

$$A = [1,\ 3,\ 8,\ 7,\ 9,\ 5,\ 6,\ 4,\ 2].$$



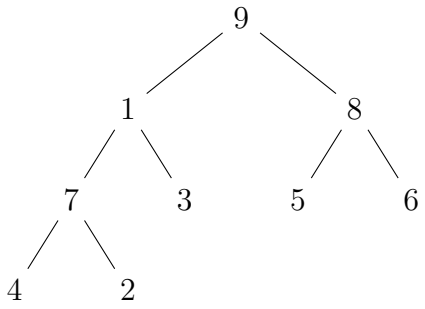
(iii) Node 2:
The value at node 2 is 3, with children $A[4] = 7$ and $A[5] = 9$. The larger child is $A[5] = 9$, so swap:

$$A = [1,\ 9,\ 8,\ 7,\ 3,\ 5,\ 6,\ 4,\ 2].$$



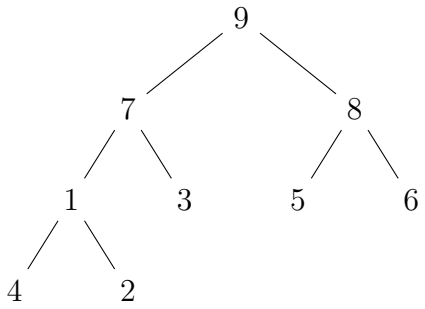
(iv) Node 1:
The value at node 1 is 1, with children $A[2] = 9$ and $A[3] = 8$. The larger child is $A[2] = 9$, so swap:

$$A = [9,\ 1,\ 8,\ 7,\ 3,\ 5,\ 6,\ 4,\ 2].$$



Now, apply `heapify` at the new position of 1 (node 2), whose children are $A[4] = 7$ and $A[5] = 3$. The larger child is $A[4] = 7$, so swap:

$$A = [9,\ 7,\ 8,\ 1,\ 3,\ 5,\ 6,\ 4,\ 2].$$

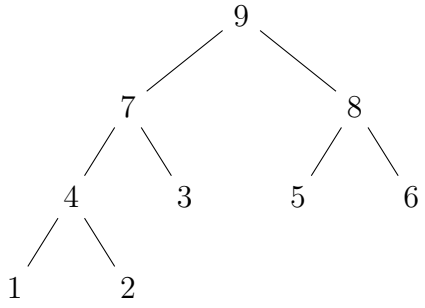


Then, `heapify` at node 4 (now with value 1) with children $A[8] = 4$ and $A[9] = 2$. The larger child is $A[8] = 4$, so swap:

$$A = [9,\ 7,\ 8,\ 4,\ 3,\ 5,\ 6,\ 1,\ 2].$$

The initial max-heap is:

$$\boxed{[9,\ 7,\ 8,\ 4,\ 3,\ 5,\ 6,\ 1,\ 2]}.$$



Step 2: Heap Sort Process

According to Heap Sort, in each iteration, we swap the root (maximum element) with the last element of the heap, reduce the heap size by one, and then perform `heapify` on the root. The key steps are as follows:

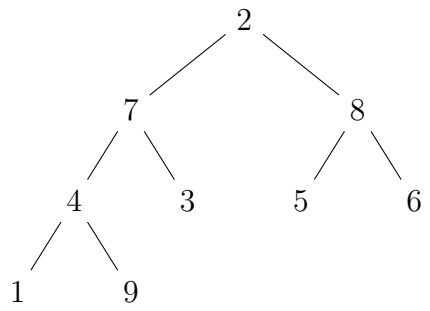
Initial Max-Heap:

$$A = [9,\ 7,\ 8,\ 4,\ 3,\ 5,\ 6,\ 1,\ 2].$$

Step 2.1:

Swap $A[1]$ and $A[9]$: swap 9 and 2, obtaining

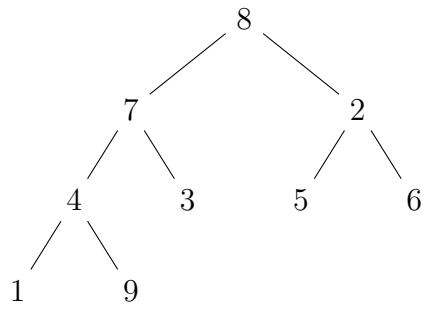
$$A = [2,\ 7,\ 8,\ 4,\ 3,\ 5,\ 6,\ 1,\ 9].$$



Perform `heapify` on $A[1 \ldots 8]$:

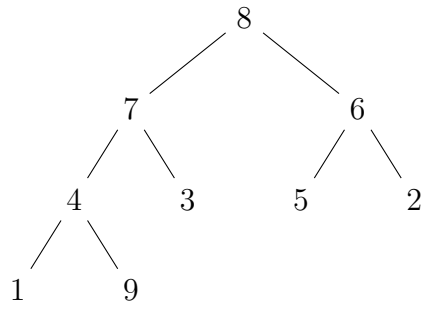
- Compare root 2 with its children 7 and 8, swap with the larger child 8:

$$A = [8,\ 7,\ 2,\ 4,\ 3,\ 5,\ 6,\ 1,\ 9].$$



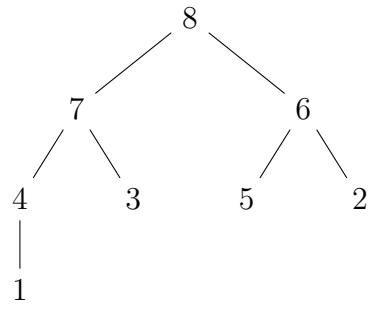
- At position 3 (value 2), compare with children $A[6] = 5$ and $A[7] = 6$, swap with 6:

$$A = [8,\ 7,\ 6,\ 4,\ 3,\ 5,\ 2,\ 1,\ 9].$$



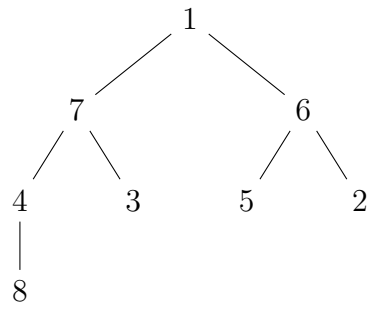
The heap now is:

$$\boxed{[8,\ 7,\ 6,\ 4,\ 3,\ 5,\ 2,\ 1]} \quad \text{(Sorted portion: 9).}$$



Step 2.2:
Swap $A[1]$ and $A[8]$: swap 8 and 1,

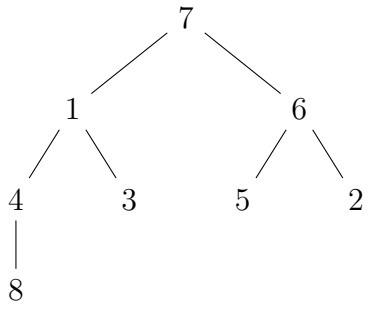
$$A = [1,\ 7,\ 6,\ 4,\ 3,\ 5,\ 2,\ 8,\ 9].$$



Perform `heapify` on $A[1 \ldots 7]$:

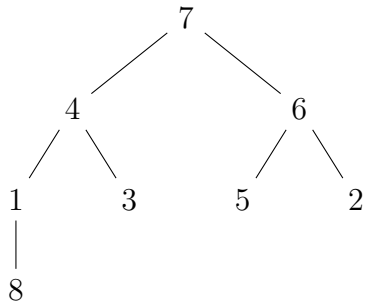
- Compare root 1 with children 7 and 6, swap with 7:

$$A = [7,\ 1,\ 6,\ 4,\ 3,\ 5,\ 2,\ 8,\ 9].$$



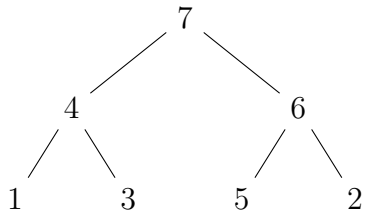
- At position 2 (value 1), compare with children $A[4] = 4$ and $A[5] = 3$, swap with 4:

$$A = [7,\ 4,\ 6,\ 1,\ 3,\ 5,\ 2,\ 8,\ 9].$$



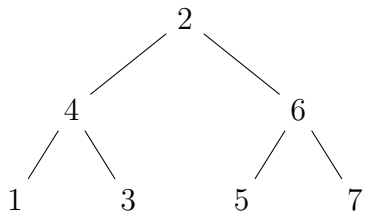
The heap becomes:

$$\boxed{[7,\ 4,\ 6,\ 1,\ 3,\ 5,\ 2]} \quad \text{(Sorted portion: } 8, 9\text{)}.$$



Step 2.3:
Swap $A[1]$ and $A[7]$: swap 7 and 2,

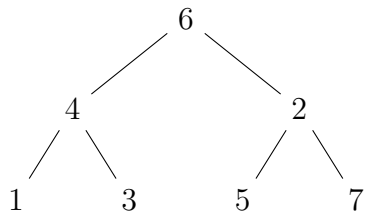
$$A = [2,\ 4,\ 6,\ 1,\ 3,\ 5,\ 7,\ 8,\ 9].$$



Perform `heapify` on $A[1 \ldots 6]$:

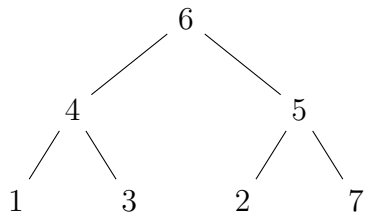
- Compare root 2 with children 4 and 6, swap with 6:

$$A = [6,\ 4,\ 2,\ 1,\ 3,\ 5,\ 7,\ 8,\ 9].$$



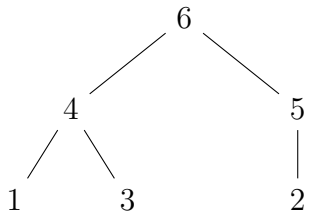
- At position 3 (value 2), compare with child $A[6] = 5$, swap with 5:

$$A = [6,\ 4,\ 5,\ 1,\ 3,\ 2,\ 7,\ 8,\ 9].$$



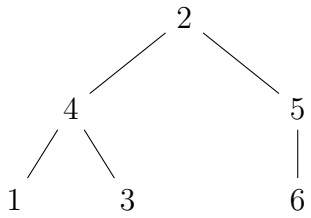
The heap becomes:

$\boxed{[6,\ 4,\ 5,\ 1,\ 3,\ 2]}$ (Sorted portion: $7, 8, 9$).



Step 2.4:
Swap $A[1]$ and $A[6]$: swap 6 and 2,

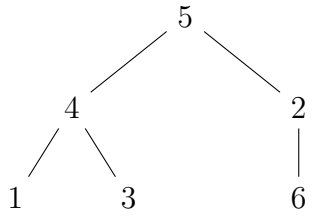
$$A = [2,\ 4,\ 5,\ 1,\ 3,\ 6,\ 7,\ 8,\ 9].$$



Perform `heapify` on $A[1 \ldots 5]$:

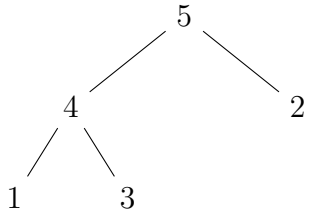
- Compare root 2 with children 4 and 5, swap with 5:

$$A = [5,\ 4,\ 2,\ 1,\ 3,\ 6,\ 7,\ 8,\ 9].$$



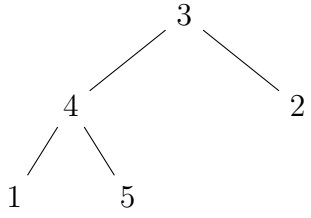
The heap becomes:

$$\boxed{[5,\ 4,\ 2,\ 1,\ 3]} \quad \text{(Sorted portion: } 6, 7, 8, 9).$$



Step 2.5:
Swap $A[1]$ and $A[5]$: swap 5 and 3,

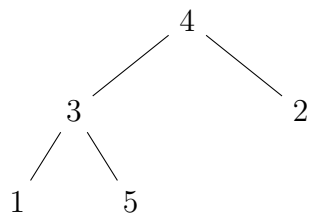
$$A = [3,\ 4,\ 2,\ 1,\ 5,\ 6,\ 7,\ 8,\ 9].$$



Perform `heapify` on $A[1 \ldots 4]$:

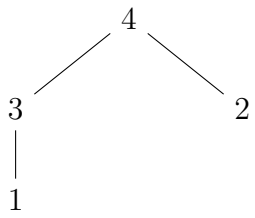
- Compare root 3 with children 4 and 2, swap with 4:

$$A = [4,\ 3,\ 2,\ 1,\ 5,\ 6,\ 7,\ 8,\ 9].$$



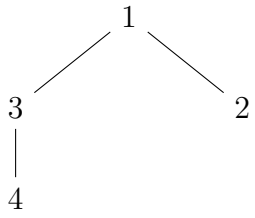
The heap becomes:

$\boxed{[4,\ 3,\ 2,\ 1]}$ (Sorted portion: $5, 6, 7, 8, 9$).



Step 2.6:
Swap $A[1]$ and $A[4]$: swap 4 and 1,

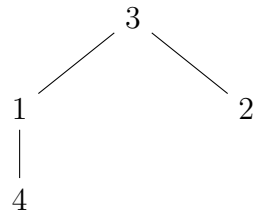
$$A = [1,\ 3,\ 2,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9].$$



Perform `heapify` on $A[1 \ldots 3]$:

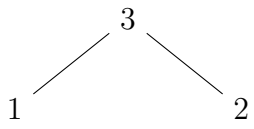
- Compare root 1 with children 3 and 2, swap with 3:

$$A = [3,\ 1,\ 2,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9].$$



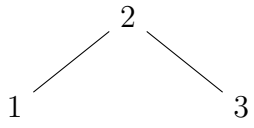
The heap becomes:

$$\boxed{[3,\ 1,\ 2]} \quad \text{(Sorted portion: } 4, 5, 6, 7, 8, 9\text{)}.$$



Step 2.7:
Swap $A[1]$ and $A[3]$: swap 3 and 2,

$$A = [2,\ 1,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9].$$



Perform `heapify` on $A[1 \ldots 2]$:

- Compare root 2 with its only child 1; no swap is needed.

The heap becomes:

$$\boxed{[2,\ 1]} \quad \text{(Sorted portion: } 3, 4, 5, 6, 7, 8, 9\text{)}.$$

Step 2.8:
Finally, swap $A[1]$ and $A[2]$: swap 2 and 1,

$$A = [1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9].$$

The array is now fully sorted in increasing order.

Final Result: The sorted sequence is

$$\boxed{1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9.}$$