CSCI-570 Homework 2

Chunho Lin* (3226964170)
chunholi@usc.edu

Due Date: Tuesday, September 23rd, 11:59pm

1. **Problem 1.**

   When discussing the implementation of binary heaps we said that the data is stored in an array. Since we did not impose any limits on the size of the heap, it means that we have to use an unbounded array. However, when discussing the runtime of the operations, we did not consider the cost of resizing (copying the entries) the array. Show that the cost of `build` in the online case (i.e. inserting n entries into an empty heap) is $O(n \cdot logn)$ even with the cost of copying taken into account. For the sake of simplicity you may assume that we store the root of the heap at index 0, instead of leaving it empty. Provide detailed calculations for the runtime.

   [Hint: revisit the cost analysis of unbounded arrays.]

   [4 points]

   Proof. When we build a binary heap in online case by inserting n entries into an empty heap, we need to consider two types of costs: the cost of insertions and the cost of expansions the array.

$$\text{Total Cost } T(n) = \text{Cost of Insertions} + \text{Cost of Expansions}$$

   For the cost of insertion:

$$\text{Cost of Insertions} = log1 + log2 + log3 + \cdots + logn$$

   For the cost of expansions:

$$\text{Cost of Expansions} = 2^0 + 2^1 + 2^2 + \cdots + 2^{logn}$$

   Total Cost:

$$
\begin{aligned}
T(n) &= \big(\log 1 + \log 2 + \log 3 + \cdots + \log n\big) + \big(2^0 + 2^1 + 2^2 + \cdots + 2^{\log n}\big) \\
&= \sum_{k=1}^{n} \log k \; + \sum_{k=0}^{\log n} 2^k \\
&= \log(n!) \; + \; \frac{2^{\log n + 1} - 1}{2 - 1} \qquad \text{(geometric series sum)} \\
&= \log(n!) \; + \; 2^{\log n + 1} - 1 \\
&= \log(n!) \; + \; 2 \cdot 2^{\log n} - 1 \\
&= \log(n!) \; + \; 2n - 1.
\end{aligned}
$$

Using the bound $\log(n!) < n \log n$, we obtain

$$T(n) < n \log n + 2n - 1 \; \in \; O(n \log n).$$

Conclusion. The cost of `build` in the online case is $O(n \cdot logn)$ even with the cost of copying taken into account.

■

2. **Problem 2.**

Suppose you have two binary min-heaps, A and B, with a total of n elements between them. You want to know whether A and B have a key in common. Devise an algorithm that solves this problem in $O(n \cdot logn)$ time and besides the two binary heaps it only uses an array of length 2.

[Hint: first come up with a $O(n \cdot logn)$ algorithm
without the restriction on the size of extra storage.]

[4 points]

Proof. We maintain an auxiliary array of length two: `arr[0]` will hold the current root of heap A, and `arr[1]` will hold the current root of heap B. The algorithm proceeds as follows:

1. Begin with two nonempty min-heaps A and B.
2. At each step, copy the root of heap A into `arr[0]` and the root of heap B into `arr[1]`.
3. Compare the two values:
   - If they are equal, then a common key has been found, and the algorithm terminates successfully.
   - If the value in `arr[0]` is smaller, remove the root of heap A (i.e., perform `deleteMin` on A).
   - If the value in `arr[1]` is smaller, remove the root of heap B.
4. Continue this process until at least one of the heaps becomes empty.
5. If no equality has been found by the time one heap is exhausted, then no common key exists, and the algorithm returns `False`.

Correctness. The algorithm works because both heaps are min-heaps, so the smallest elements are always available at the root. By comparing the roots at each step, we ensure that no potential common key is skipped: if the roots are equal, we have found a common key; if not, the smaller root cannot possibly appear later in the other heap, so it is safely removed. This guarantees that all possible matches are checked.

Time Complexity. Each comparison and deletion takes $O(\log n)$ time. In the worst case, we may perform up to n deletions (when all elements are distinct). Hence, the overall time complexity is $O(n \log n)$. ■

3. **Problem 3.**

You are given a "k-sorted array": this is an almost-sorted array, where each of the elements are misplaced by less than k positions from their correct location. For example, $A = [1, 2, 3, 6, 4, 5, 7]$ is a 3-sorted array, because the elements 4, 5, and 6 are misplaced by 1, 1, and 2 positions respectively from their correct locations. Design an $O(n \cdot logk)$ algorithm to sort the array and analyze the runtime.

[Hint: using a small heap will be helpful.]

[5 points]

Proof. We are given a k-sorted array, i.e., each element is at most k positions away from its correct sorted position. To exploit this property, we use a min-heap with size of $k + 1$:

1. Initialize an empty min-heap.
2. Insert the first $k + 1$ elements of the array into the heap.
3. For each index i from 0 to $n - 1$:
   i. Perform a `deleteMin` on the heap. This removes the smallest element from the heap and returns it. Place this value into the output array at position i.
   ii. If there are still elements left in the input array, insert the next element (at index $i + k + 1$) into the heap.
4. After all input elements have been considered, repeatedly apply `deleteMin` on the heap until it becomes empty, appending each result to the output array.

Correctness. Because each element is at most k positions from its sorted position, the next smallest element must lie within the current $k + 1$ candidates in the heap. The `deleteMin` operation always returns this globally smallest candidate, ensuring elements are placed in the correct sorted order.

Time Complexity.

- Building the initial heap with $k + 1$ elements takes $O(k)$.
- Each `deleteMin` and `insert` operation takes $O(\log k)$.
- There are n `deleteMin` operations and at most n `insert` operations.

Thus the total running time is

$$O(k) + O(n \log k) = O(n \log k).$$

■

4. **Problem 4.**

   Write a divide and conquer algorithm that finds the largest difference between any two elements of an unsorted array of n numbers.

   Example: in case of $A = [3, 7.65, -2, 3, -0.85, 9.15, -1.5]$
   the algorithm should return 11.15.

   Your algorithm should run in $O(n)$ time. To achieve this, the algorithm needs $T(n) = 2 \cdot T(\frac{n}{2}) + O(1)$ as its runtime recurrence relation. After providing the algorithm, justify that its runtime is indeed $O(n)$ by unrolling its recursion tree.

   [5 points]

Proof. The divide and conquer algorithm to find the largest difference works as follows:

1. **Base case:** If the array has only one or two elements, return the difference between the maximum and minimum elements (0 for one element).
2. **Divide:** Split the array into two halves.
3. **Conquer:** Recursively find:
   - The maximum and minimum elements in the left half
   - The maximum and minimum elements in the right half
4. **Combine:** The largest difference is the maximum of:
   - The largest difference within the left half
   - The largest difference within the right half
   - The difference between the maximum of both halves and the minimum of both halves

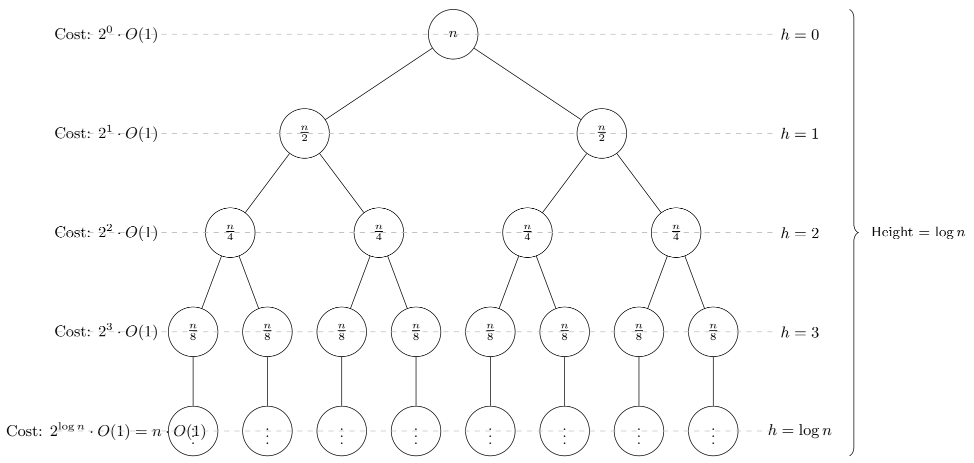

Figure 1: Recursion tree for the divide and conquer algorithm

Recurrence Relation: $T(n) = 2 \cdot T(\frac{n}{2}) + O(1)$

Time Complexity:

From the recursion tree analysis:

- **Height:** $\log n$ levels
- **Cost per level:** Each level has 2^i nodes, each doing $O(1)$ work
- **Level 0:** $1 \cdot O(1) = O(1)$
- **Level 1:** $2 \cdot O(1) = O(1)$
- **Level 2:** $4 \cdot O(1) = O(1)$
- $\vdots$
- **Level** $\log n$**:** $2^{\log n} \cdot O(1) = n \cdot O(1) = O(n)$

Total Cost:

$$T(n) = \sum_{i=0}^{\log n} 2^i \cdot O(1) = O(1) \cdot \sum_{i=0}^{\log n} 2^i = O(1) \cdot (2^{\log n + 1} - 1) = O(1) \cdot (2n - 1) = O(n)$$

Therefore, the algorithm runs in $O(n)$ time as required.

■