Weekly Assignment 3: Solution
Total: 100

CS 2500: Algorithms

Due Date: October 7, 2024 at 11.59 PM

1 Problems

1.1 Max Heap Operation

1. **Initial Array (1-based indexing):**

Index:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Array A:	27	17	3	16	13	10	1	5	7	12	4	8	9	0

Step 1 - Identify the Subtree Root:

- The function `Max-Heapify(A, 3)` starts at index 3.
- The element at index 3 is 3.

Step 2 - Identify Left and Right Children:

- Left child of 3 is at index $2 \times 3 = 6 \rightarrow 10$.
- Right child of 3 is at index $2 \times 3 + 1 = 7 \rightarrow 1$.
- Compare 3, 10, and 1. The largest value is 10.

Step 3 - Swap and Continue:

- Swap 3 with 10.
- New array:
$$[27, 17, 10, 16, 13, 3, 1, 5, 7, 12, 4, 8, 9, 0]$$

Step 4 - Recursively Max-Heapify Downward:

- Now, apply `Max-Heapify(A, 6)` since 3 was moved down to index 6.
- Left child of 3 is at index $2 \times 6 = 12 \rightarrow 8$.
- Right child of 3 is at index $2 \times 6 + 1 = 13 \rightarrow 9$.
- Compare 3, 8, and 9. The largest value is 9.

Step 5 - Swap and Continue:

- Swap 3 with 9.
- New array:
$$[27, 17, 10, 16, 13, 9, 1, 5, 7, 12, 4, 8, 3, 0]$$

Step 6 - Recursively Max-Heapify Downward (No More Changes Needed):

- Apply `Max-Heapify(A, 13)`:
  - Left and right children indices (26 and 27) are out of bounds.
  - No further action needed.

Final Result:

$$[27, 17, 10, 16, 13, 9, 1, 5, 7, 12, 4, 8, 3, 0]$$

This is the updated max-heap after applying `Max-Heapify(A, 3)`.

2. The algorithm for `Min-Heapify` is given below:

Algorithm 1 `Min-Heapify(A, i)`

```
left = $2 \times i$                                                   // Index of left child
right = $2 \times i + 1$                                              // Index of right child
smallest = $i$
if left $\leq$ heap-size[A] and A[left] $<$ A[smallest] then
  smallest = left
end if
if right $\leq$ heap-size[A] and A[right] $<$ A[smallest] then
  smallest = right
end if
if smallest $\neq i$ then
  swap A[i] with A[smallest]
  Min-Heapify(A, smallest)
end if
```

Explanation:

- **Input**: Array `A` and index `i`.
- **Objective**: Maintain the min-heap property at node `i` by ensuring `A[i]` is smaller than or equal to its children.
- **Steps**:
  - Compare the element at index `i` with its left and right children.
  - Identify the smallest among them.
  - If `A[i]` is not the smallest, swap `A[i]` with the smallest child.
  - Recursively call `Min-Heapify` on the affected subtree.

Running Time Comparison The running time of `Min-Heapify` is the same as that of `Max-Heapify`. In both algorithms, the primary operation involves traversing down the height of the tree, and the maximum number of recursive calls is proportional to the height of the heap.

- For a heap of size n, the height of the heap is $O(\log n)$.
- Thus, the running time of both `Min-Heapify` and `Max-Heapify` is $O(\log n)$.

3. If the element `A[i]` is already larger than its children, calling `Max-Heapify(A, i)` will have no effect on the heap. Since `Max-Heapify` only swaps `A[i]` with its largest child when `A[i]` is smaller, no swaps or recursive calls will be made. As a result, the heap structure will remain unchanged, and the function will terminate after checking the conditions. This ensures that the max-heap property is preserved.

   In terms of performance, this scenario leads to the best-case running time of $O(1)$, as no additional recursive calls are needed.

4. When calling `Max-Heapify(A, i)` for $i >$ `A.heap-size`$/2$, the index i corresponds to a position in the array that is a leaf node in the heap. In a binary heap, all nodes with indices greater than $\lfloor$`heap-size`$/2\rfloor$ are leaf nodes because they do not have any children.

   Since leaf nodes do not have any children, calling `Max-Heapify(A, i)` on a leaf node will not cause any comparisons, swaps, or recursive calls. The function will simply check that there are no children to compare with and terminate immediately. Therefore, the heap remains unchanged, and no modifications occur.

1.2 Build Heap Operation

1. **Loop Invariant:** At the start of each iteration of the `for` loop, for every index j such that $j > i$ (where i is the current loop variable), the subtree rooted at j is a max-heap.

   Proof

   (a) **Initialization:** Before the first iteration of the loop, $i = \lfloor n/2 \rfloor$. By this point, all nodes with indices greater than $\lfloor n/2 \rfloor$ are leaf nodes because of the properties of a binary heap. Since leaf nodes are trivially max-heaps (as they have no children), the loop invariant holds true at the start.

   (b) **Maintenance:** Assume the loop invariant holds at the beginning of the i-th iteration. During the iteration, the algorithm calls `Max-Heapify(A, i)`.

   - The function `Max-Heapify` ensures that the subtree rooted at i satisfies the max-heap property, assuming that the subtrees rooted at its children are already max-heaps.
   - By the loop invariant, we know that all subtrees rooted at nodes with indices greater than i are already max-heaps.
   - Therefore, after the call to `Max-Heapify(A, i)`, the subtree rooted at i will also be a max-heap, and the loop invariant continues to hold.

   (c) **Termination:** The loop runs until $i = 1$. When the loop terminates, $i = 0$, which is out of the loop range. By the loop invariant, all subtrees rooted at every index from 1 to n are max-heaps. Since the root node $A[1]$ is also now a max-heap, the entire array satisfies the max-heap property, which means that `Build-Max-Heap` has correctly converted the array into a max-heap.

   Conclusion: The loop invariant holds at the beginning, is maintained throughout the loop, and ensures that when the algorithm terminates, the entire array A is a valid max-heap. Thus, the algorithm `Build-Max-Heap` is correct.

2. **Initial Array**

$$A = [5, 3, 17, 10, 84, 19, 6, 22, 9]$$

Index:	1	2	3	4	5	6	7	8	9
Array:	5	3	17	10	84	19	6	22	9

   Steps of `Build-Max-Heap`
   Heap Size: Set A.heap_size $= 9$. The algorithm starts from $i = \lfloor n/2 \rfloor = \lfloor 9/2 \rfloor = 4$ and proceeds backward to 1, calling `Max-Heapify` at each step.

   i. `Max-Heapify` at $i = 4$

   - Current node: 10 (index 4)
   - Left child: 22 (index 8)
   - Right child: 9 (index 9)
   - Largest: 22 (index 8)
   - Swap 10 and 22.

$$[5, 3, 17, 22, 84, 19, 6, 10, 9]$$

ii. `Max-Heapify` at $i = 3$

- Current node: 17 (index 3)
- Left child: 19 (index 6)
- Right child: 6 (index 7)
- Largest: 19 (index 6)
- Swap 17 and 19.

$$[5, 3, 19, 22, 84, 17, 6, 10, 9]$$

iii. `Max-Heapify` at $i = 2$

- Current node: 3 (index 2)
- Left child: 22 (index 4)
- Right child: 84 (index 5)
- Largest: 84 (index 5)
- Swap 3 and 84.

$$[5, 84, 19, 22, 3, 17, 6, 10, 9]$$

- Apply `Max-Heapify` at $i = 5$ because 3 was moved down.
- Current node: 3 (index 5)
- Left child and right child do not exist (indices 10 and 11).
- No swap needed.

iv. `Max-Heapify` at $i = 1$

- Current node: 5 (index 1)
- Left child: 84 (index 2)
- Right child: 19 (index 3)
- Largest: 84 (index 2)
- Swap 5 and 84.

$$[84, 5, 19, 22, 3, 17, 6, 10, 9]$$

- Apply `Max-Heapify` at $i = 2$ because 5 was moved down.
- Current node: 5 (index 2)
- Left child: 22 (index 4)
- Right child: 3 (index 5)
- Largest: 22 (index 4)
- Swap 5 and 22.

$$[84, 22, 19, 5, 3, 17, 6, 10, 9]$$

- Apply `Max-Heapify` at $i = 4$ because 5 was moved down.
- Current node: 5 (index 4)
- Left child: 10 (index 8)
- Right child: 9 (index 9)
- Largest: 10 (index 8)

- Swap 5 and 10.

$$[84, 22, 19, 10, 3, 17, 6, 5, 9]$$

Final Result The resulting max-heap after running `Build-Max-Heap` is:

$$[84, 22, 19, 10, 3, 17, 6, 5, 9]$$

The array has been successfully converted into a max-heap where each parent node is greater than or equal to its children.

3. **Proof of (a)**
Statement: There are at most $\lceil \frac{n}{2^{h+1}} \rceil$ elements with a height h in a heap with n elements.

Proof:

(a) In a binary heap, the height h of a node is defined as the number of edges on the longest path from the node to a leaf.

(b) The height of the heap is $\lfloor \log n \rfloor$, where n is the total number of elements in the heap.

(c) The height h can range from 0 (leaf nodes) to $\lfloor \log n \rfloor$ (root node).

(d) At height h, the maximum number of nodes that can exist at this height is 2^h because a binary heap can have at most 2^h nodes at each level h.

(e) To find how many nodes are at height h, we need to consider how many nodes are at least h edges away from the leaf nodes.

(f) In a complete binary tree, there are $n - \lceil n/2^{h+1} \rceil$ nodes at height greater than h, leaving at most $\lceil n/2^{h+1} \rceil$ nodes at height h.

Thus, the maximum number of elements at height h is at most $\lceil \frac{n}{2^{h+1}} \rceil$.

Proof of (b)
Statement: $\lceil \frac{n}{2^{h+1}} \rceil \geq \frac{1}{2} \quad \forall h \in [0, \lfloor \log n \rfloor]$.

Proof:

(a) From (a), we have $\lceil \frac{n}{2^{h+1}} \rceil$ representing the number of elements at height h in the heap.

(b) Since h can range from 0 up to $\lfloor \log n \rfloor$, let's analyze the expression $\frac{n}{2^{h+1}}$.

(c) When $h = 0$ (height of the leaf level), we get:

$$\frac{n}{2^{0+1}} = \frac{n}{2}$$

Therefore, $\lceil \frac{n}{2} \rceil \geq \frac{n}{2}$, which is at least $\frac{1}{2}$ for $n \geq 1$.

(d) For other values of h, $\frac{n}{2^{h+1}}$ decreases as h increases, but it will always remain positive and at least $\frac{1}{2}$.

(e) Since $\lceil x \rceil$ rounds up x to the nearest integer, $\lceil \frac{n}{2^{h+1}} \rceil$ is always at least $\frac{1}{2}$.

Therefore, $\lceil \frac{n}{2^{h+1}} \rceil \geq \frac{1}{2}$ for all $h \in [0, \lfloor \log n \rfloor]$.

1.3 Heap Sort

1. **Step 1:** `Build-Max-Heap` We first convert the array into a max-heap using `Build-Max-Heap`. The process begins at $i = \lfloor n/2 \rfloor = \lfloor 9/2 \rfloor = 4$ and continues down to 1.

   - `Max-Heapify` at $i = 4$:
     - Current node: 25, Left child: 8, Right child: 4
     - No change needed.

- `Max-Heapify` at $i = 3$:
  - Current node: 2, Left child: 17, Right child: 20
  - Swap 2 and 20.
  - Updated Array: $[5, 13, 20, 25, 7, 17, 2, 8, 4]$
- `Max-Heapify` at $i = 2$:
  - Current node: 13, Left child: 25, Right child: 7
  - Swap 13 and 25.
  - Updated Array: $[5, 25, 20, 13, 7, 17, 2, 8, 4]$
- `Max-Heapify` at $i = 1$:
  - Current node: 5, Left child: 25, Right child: 20
  - Swap 5 and 25.
  - Apply `Max-Heapify` at $i = 2$ (position of 5).
  - Swap 5 and 13.
  - Final Max-Heap: $[25, 13, 20, 8, 7, 17, 2, 5, 4]$

Step 2: `Heap-Sort` Extraction Process

- Swap 25 and 4. Reduce heap size to 8. Apply `Max-Heapify` at $i = 1$.

$$[20, 13, 17, 8, 7, 4, 2, 5, 25]$$

- Swap 20 and 5. Reduce heap size to 7. Apply `Max-Heapify` at $i = 1$.

$$[17, 13, 5, 8, 7, 4, 2, 20, 25]$$

- Swap 17 and 2. Reduce heap size to 6. Apply `Max-Heapify` at $i = 1$.

$$[13, 8, 5, 2, 7, 4, 17, 20, 25]$$

- Swap 13 and 4. Reduce heap size to 5. Apply `Max-Heapify` at $i = 1$.

$$[8, 7, 5, 2, 4, 13, 17, 20, 25]$$

- Swap 8 and 4. Reduce heap size to 4. Apply `Max-Heapify` at $i = 1$.

$$[7, 4, 5, 2, 8, 13, 17, 20, 25]$$

- Swap 7 and 2. Reduce heap size to 3. Apply `Max-Heapify` at $i = 1$.

$$[5, 4, 2, 7, 8, 13, 17, 20, 25]$$

- Swap 5 and 2. Reduce heap size to 2. Apply `Max-Heapify` at $i = 1$.

$$[4, 2, 5, 7, 8, 13, 17, 20, 25]$$

- Swap 4 and 2. Reduce heap size to 1.

$$[2, 4, 5, 7, 8, 13, 17, 20, 25]$$

Final Sorted Array After completing the `Heap-Sort` process, the final sorted array is:

$$[2, 4, 5, 7, 8, 13, 17, 20, 25]$$

2. **Loop Invariant:** At the start of each iteration of the `for` loop (lines 2-5) in `Heap-Sort`, the subarray `A[1:i]` is a max-heap containing the i smallest elements of `A[1:n]`, and the subarray `A[i+1:n]` contains the $n - i$ largest elements of `A[1:n]`, sorted in ascending order.

   Proof

   (a) **Initialization:**
   - Before the first iteration of the `for` loop, the algorithm calls `Build-Max-Heap(A)`, which transforms the entire array `A[1:n]` into a max-heap.
   - Therefore, at the start of the loop (when $i = n$), `A[1:n]` is a valid max-heap, and there are no sorted elements yet. This satisfies the loop invariant since there are 0 smallest elements in `A[1:n]`, and the n largest elements are not yet sorted but are organized within the max-heap structure.

   (b) **Maintenance:**
   - At the beginning of each iteration, the largest element in the max-heap `A[1:i]` is at `A[1]` (the root of the heap). This element is one of the $n - i + 1$ largest elements of `A[1:n]`.
   - The `for` loop performs the following steps:
     i. Swap `A[1]` (the largest element in the heap) with `A[i]`, moving it to the end of the array. Now `A[i]` is the i-th largest element in `A[1:n]` and is in its final, sorted position.
     ii. Reduce the heap size by 1 (i.e., i is decremented).
     iii. Call `Max-Heapify(A, 1)` to restore the max-heap property for the subarray `A[1:i-1]`.
   - After these operations, `A[1:i-1]` is again a max-heap containing the $i - 1$ smallest elements of `A[1:n]`, and `A[i:n]` now contains the $n - i + 1$ largest elements of `A[1:n]`, sorted in ascending order.
   - Therefore, the loop invariant is maintained at the start of each iteration.

   (c) **Termination:**
   - The loop runs until $i = 1$. At this point, all elements in the array have been moved to their correct, sorted positions. Specifically, `A[2:n]` is sorted in ascending order, and `A[1]` is the smallest element in the array.
   - The loop invariant guarantees that when the loop terminates, the entire array `A[1:n]` is sorted.

   Conclusion The loop invariant holds true at the start, is maintained throughout each iteration of the `for` loop, and ensures that the array `A[1:n]` is sorted when the loop terminates. Therefore, the `Heap-Sort` algorithm correctly sorts the array.

3. In both scenarios (increasing order and decreasing order), the running time of `Heap-Sort` is: $O(n \log n)$. The initial order of the array does not affect the overall time complexity because `Heap-Sort` always performs `Build-Max-Heap` and subsequent extractions in the same manner.

1.4 Merge Sort

1. The algorithm is given below:

Algorithm 2 Merge-With-Sentinels(a, b, m, n)

```
a[m + 1] =                        // Place a large sentinel value at the end of array a
b[n + 1] =                        // Place a large sentinel value at the end of array b
i = 1, j = 1
for k = 1 to m + n do
  if a[i]  b[j] then
    c[k] = a[i]
    i = i + 1
  else
    c[k] = b[j]
    j = j + 1
  end if
end for
```

Explanation

- **Sentinel Values:** The sentinel values (∞) at the end of `a` and `b` ensure that once we exhaust one array, the remaining elements from the other array can be easily merged without additional boundary checks. This simplifies the logic within the loop.
- **Pointers Initialization:** `i` and `j` are pointers initialized to the start of arrays `a` and `b`, respectively.
- **Merge Process:** Iterate `k` from 1 to `m + n`, which is the total length of the output array `c`.
  - If `a[i]` is smaller, place `a[i]` into `c[k]` and increment `i`.
  - Otherwise, place `b[j]` into `c[k]` and increment `j`.

Running Time The running time of this algorithm is $O(m + n)$, as each element from arrays `a` and `b` is processed exactly once.

2. **Key Idea:** We can merge ℓ sorted sequences $X_1, X_2, \ldots, X_\ell$ using a divide-and-conquer approach. The idea is to recursively divide the sequences into groups, merge them, and combine them until a single sorted sequence is obtained. The algorithms are shown in Algorithm 2 and 2.

Algorithm 3 Merge-Multiple-Sequences($X[1 : \ell]$)

```
if $\ell == 1$ then
  return $X[1]$                                   // Only one sequence, already sorted
end if
if $\ell == 2$ then
  return Merge-Two-Sorted-Arrays(X[1], X[2])      // Merge two sequences directly
end if
mid = $\ell$ / 2
left = Merge-Multiple-Sequences(X[1:mid])
right = Merge-Multiple-Sequences(X[mid + 1:$\ell$])
return Merge-Two-Sorted-Arrays(left, right)
```

Algorithm 4 `Merge-Two-Sorted-Arrays(A, B)`

```
Create an empty array $C$.
i = 1, j = 1
while i $\leq$ |A| and j $\leq$ |B| do
  if A[i] $\leq$ B[j] then
    Append A[i] to $C$
    i = i + 1
  else
    Append B[j] to $C$
    j = j + 1
  end if
end while
Append any remaining elements of $A$ or $B$ to $C$.
return $C$
```

Time Complexity Analysis

- **Total Number of Levels:** The recursive approach divides the ℓ sequences into halves at each level, leading to $\log \ell$ levels of recursion.
- **Work Done at Each Level:** At each level, we merge n elements (total across all sequences), which requires $O(n)$ operations.
- **Overall Running Time:** $O(n \log \ell)$