Homework Assignment Week 3: Sorting

Hyuntaek Oh

ohhyun@oregonstate.edu

CS 514_400 Algorithm

October 21, 2024

1. a) Implement the *heapSort* algorithm.

```python
def heapSort(arr):

    # Length of input array
    n = len(arr)

    # Build Max-Heap
    for i in range(n // 2 - 1, -1, -1):
        max_heapify(arr, n, i)

    # For sorting, repeat discarding process
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        max_heapify(arr, i, 0)

    return arr
```

```python
def max_heapify(arr, n, i):
    largest_idx = i
    left_idx = (2 * i) + 1
    right_idx = (2 * i) + 2

    # If left is larger than largest, assign it largest
    if left_idx < n and arr[left_idx] > arr[largest_idx]:
        largest_idx = left_idx

    # If right is larger than largest, assign it largest
    if right_idx < n and arr[right_idx] > arr[largest_idx]:
        largest_idx = right_idx

    # If changed, do max_heapify again
    if largest_idx != i:
        arr[largest_idx], arr[i] = arr[i], arr[largest_idx]
        max_heapify(arr, n, largest_idx)
```

Fig 1. *heapSort* and *max_heapify* functions

b) Derive the time complexity of the *heapSort* algorithm

The time complexity of the heapsort algorithm can be derived from analysis of its three parts: *Max-Heapify, Building Max-heap*, and *Sorting heap* parts.

1) Max-Heapify

For a tree rooted at a given node i, which is largest index in the code, the running time is the $\theta(1)$ time to fix up the relationships among the elements $arr[left\ idx], arr[largest], arr[right\ idx]$. In addition, the time to run *Max-Heapify* on a subtree rooted at one of the children of node i since the algorithm call recursively *Max-Heapify.* The children's subtrees each have size at most $\frac{2n}{3}$. Thus, we can describe the running time of *Max-Heapify* by the recurrence:

$$T(n) \leq T\left(\frac{2n}{3}\right) + \theta(1)$$

After applying the case 2 of the master theorem to the inequality above, we get $T(n) = O(logn)$. The height of heap-tree is dominant since the size of input n can affect the algorithm and the heap is based on binary tree structure, which is $logn$.

Therefore, the running time of *Max-Heapify* is $O(logn)$

2) *Building Max-Heap*

Building Max-Heap function includes *Max-Heapify* function to maintain the heap property that the value in a parent node is larger than ones in its child nodes. The number of calling *Max-Heapify* function depends on internal nodes in a heap of size n, which is approximately $n/2$.

An n-element heap has height $\lfloor \lg n \rfloor$ and at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of any height h. The time required by *Max-Heapify* function when called on a node of height h is $O(h)$. We can express the total cost of *Building Max-Heap* as being bounded from above by $\sum_{h=0}^{\lfloor lgn \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil h$. Since $\sum_{h=0}^{\lfloor lgn \rfloor} \frac{h}{2^h}$ goes 2 if $h \to \infty$, we can conclude:

$$\sum_{h=0}^{\lfloor lgn \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathrm{h} \cong \mathrm{O(n)}$$

Thus, with tighter bound, the running time of *Building Max-Heap* is $\mathrm{O(n)}$.

3) *Heap Sort*

The function of *Heap Sorting* starts by calling the *Building Max-Heap* function to build a max-heap on the input array, which has $\mathrm{O(n)}$ time complexity.

Since the maximum element of the array is stored at the root node, *Heap Sort* can place it into its correct final position by exchanging it with the last element in the array. After changing the first and last element, the last element should be maximum element, meaning that the last index is sorted and the rest of the elements in the array is needed to be sorted. This process keeps going until reaching the root index, meaning that the time complexity is $\mathrm{O(n)}$ until finishing iterations.

However, the new element at the root node violates the property of max-heap, so the algorithm needs to call *Max-Heapify* function, which has $\mathrm{O(logn)}$.

From the analysis above, we can conclude:

$$O(n) + [O(n) * O(logn)] = O(nlogn)$$

Compare the time taken in each case to the theoretical time complexity.

We use the theoretical time complexity of *Heap* Sort, which is $O(nlogn)$, from the analysis.

The table below is the time taken in each case with respect to the input size n.

Input size of n	$nlogn$	A long list of random numbers	An already sorted array	A reversely sorted array
10^3	9965.78	0.0019	0.0021	0.0009
10^4	132877.12	0.0269	0.0260	0.0227
10^5	1660964.05	0.3616	0.3854	0.3435
10^6	19931568.57	5.8377	4.6809	4.3847
10^7	232434966.64	81.1724	59.7248	68.5521

Table 1. The time taken in three cases based on the input size of n

In the table above: -'input size of n' is the size of an array for experiment. –'nlogn' represents the "theoretical" running time calculated based on the time complexity of *Heap* Sort. –'A long list of random numbers', 'An already sorted array', and 'A reversely sorted array' represent the actual running time for a given n. To see the relationship between the increase of input size and running time in *Heap* Sort algorithm, we use the table below that shows growth rate in each case step by step.

Input size change	Random list Growth rate	Sorted list Growth rate	Reverse sorted list Growth rate
10^3to10^4	13.52	12.20	13.28
10^4to10^5	13.40	14.83	13.74
10^5to10^6	16.14	12.14	12.76
10^6to10^7	13.91	12.76	15.64

Table 2. Growth rate of three cases vs. input size change

From the table, it seems that the ratio stays around 13~16 for random list, stays around 12~14 for sorted list, and stays around 12-15 for reverse sorted list. As can be seen in Table 2, there are some small flunctuations because of laptop setting. However, the growth rartes are relatively consistent across all lists and fall within the expected range of $O(nlogn)$.

2. Argue the correctness of *heapSort* algorithm using the following loop invariant: At the start of each iteration of the for loop, the subarray $A[1 \dots i]$ is a max-heap containing the i smallest elements of $A[1 \dots n]$, and the subarray $A[i+1 \dots n]$ contains the $n-i$ largest elements of $A[1 \dots n]$, sorted. Assume that both BUILD-MAX-HEAP

and MAX-HEAPIFY are correct when constructing your argument.

loop invariant: At the start of each iteration of the for loop, the subarray $A[1 \ldots i]$ is a max-heap containing the i smallest elements of $A[1 \ldots n]$, and the subarray $A[i+1 \ldots n]$ contains the $n-i$ largest elements of $A[1 \ldots n]$, sorted.

For proof of the correctness of the algorithm, we need to explore three stages: Initialization, Maintenance, and termination.

Initialization:

Before entering the for loop, the first step is to build a max-heap from the entire array $A[1 \ldots n]$ using max-heapify function. The function of building max-heap guarantees that $A[1 \ldots n]$ is a valid max-heap, meaning that for every element $A[k]$, the property $A[k] \geq A[left(k)]$ and $A[k] \geq A[right(k)]$ holds. Because of the property, the root node $A[1]$ is the largest element in the given array. Without sorting at this point, the entire array satisfies the max-heap property, satisfying the subarray $A[1 \ldots n]$ is a valid max-heap.

Maintenance:

After building max-heap, the algorithm swaps the first and last elements in the subarray $A[1 \ldots i]$, which are $A[1]$ and $A[i]$ respectively and reduces the size of the heap by one, placing the largest element from heap into sorted position $A[i+1 \ldots n]$. Then, it calls max-heapify function to maintain the property of max-heap with reduced length of the subarray $A[1 \ldots i-1]$. Since the number of elements in the heap decreases by 1, the sorted array $A[i+1 \ldots n]$ grows by 1 element, containing the $n-i$ largest elements in decreasing order.

Termination:

The loop terminates when $i = 1$, meaning that there is only smallest element left in the heap. The subarray $A[2 \ldots n]$ contains the $n-1$ largest elements, which are already sorted. $A[1]$ is the remainder of the heap and already placed as the smallest element. The entire array $A[1 \ldots n]$ is sorted in non-decreasing order since $A[1]$ is the smallest and the next smallest element follow it in increasing order until reaching $A[n]$.

3. Implement a *minPriorityQueue* using a min-heap. Given a heap, the *MinPriorityQueue* acts as follows:

• *insert()* -insert an element with a specified priority value
• *first()* – return the element with the lowest/minimum priority value (the 'first' element in the priority queue)
• *remove_first()* – remove (and return) the element with the lowest/minimum

prirority value

```python
class minPriorityQueue:
    def __init__(self):
        self.minHeap = []

    3 usages
    def min_heapify(self, idx):
        smallest = idx
        left = (2 * idx) + 1
        right = (2 * idx) + 2

        if left < len(self.minHeap) and self.minHeap[left] < self.minHeap[smallest]:
            smallest = left

        if right < len(self.minHeap) and self.minHeap[right] < self.minHeap[smallest]:
            smallest = right

        if smallest != idx:
            self.minHeap[idx], self.minHeap[smallest] = self.minHeap[smallest], self.minHeap[idx]
            self.min_heapify(smallest)
```

Fig 2. minPriorityQueue function

```python
    def insert(self, val):
        self.minHeap.append(val)
        if self.minHeap:
            for idx in range(len(self.minHeap) // 2 - 1, -1, -1):
                self.min_heapify(idx)

    1 usage
    def first(self):
        if self.minHeap:
            return self.minHeap[0]
        else:
            return None

    def remove_first(self):
        if len(self.minHeap) == 0:
            return None

        lowest = self.first()
        self.minHeap[0] = self.minHeap[-1]
        self.minHeap.pop()

        n = len(self.minHeap)

        if self.minHeap:
            for idx in range(n // 2 - 1, -1, -1):
                self.min_heapify(idx)

        return lowest
```

Fig 3. insert(), first(), and remove_first() functions