**Homework Assignment Week 3: Sorting**

Hyuntaek Oh

ohhyun@oregonstate.edu

CS 514\_400 Algorithm

October 21, 2024

1. **a) Implement the *heapSort* algorithm.**

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명 텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**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 , which is largest index in the code, the running time is the time to fix up the relationships among the elements . In addition, the time to run *Max-Heapify* on a subtree rooted at one of the children of node since the algorithm call recursively *Max-Heapify.* The children’s subtrees each have size at most . Thus, we can describe the running time of *Max-Heapify* by the recurrence:

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

Therefore, the running time of *Max-Heapify* is

1. *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 , which is approximately .

An -element heap has height and at most nodes of any height . The time required by *Max-Heapify* function when called on a node of height is . We can express the total cost of *Building Max-Heap* as being bounded from above by . Since goes 2 if , we can conclude:

Thus, with tighter bound, the running time of *Building Max-Heap* is .

1. *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 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 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 .

From the analysis above, we can conclude:

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

We use the theoretical time complexity of *Heap* Sort, which is , from the analysis.

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input size of** |  | **A long list of random numbers** | **An already sorted array** | **A reversely sorted array** |
|  | 9965.78 | 0.0019 | 0.0021 | 0.0009 |
|  | 132877.12 | 0.0269 | 0.0260 | 0.0227 |
|  | 1660964.05 | 0.3616 | 0.3854 | 0.3435 |
|  | 19931568.57 | 5.8377 | 4.6809 | 4.3847 |
|  | 232434966.64 | 81.1724 | 59.7248 | 68.5521 |

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

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** |
| to | 13.52 | 12.20 | 13.28 |
| to | 13.40 | 14.83 | 13.74 |
| to | 16.14 | 12.14 | 12.76 |
| to | 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 .

1. **Argue the correctness of *heapSort* algorithm using the following loop invariant: At the start of each iteration of the for loop, the subarray is a max-heap containing the smallest elements of , and the subarray contains the largest elements of , 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 is a max-heap containing the smallest elements of , and the subarray contains the largest elements of , 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 using max-heapify function. The function of building max-heap guarantees that is a valid max-heap, meaning that for every element , the property and holds. Because of the property, the root node is the largest element in the given array. Without sorting at this point, the entire array satisfies the max-heap property, satisfying the subarray is a valid max-heap.

**Maintenance**:

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

**Termination**:

The loop terminates when , meaning that there is only smallest element left in the heap. The subarray contains the largest elements, which are already sorted. is the remainder of the heap and already placed as the smallest element. The entire array is sorted in non-decreasing order since is the smallest and the next smallest element follow it in increasing order until reaching .

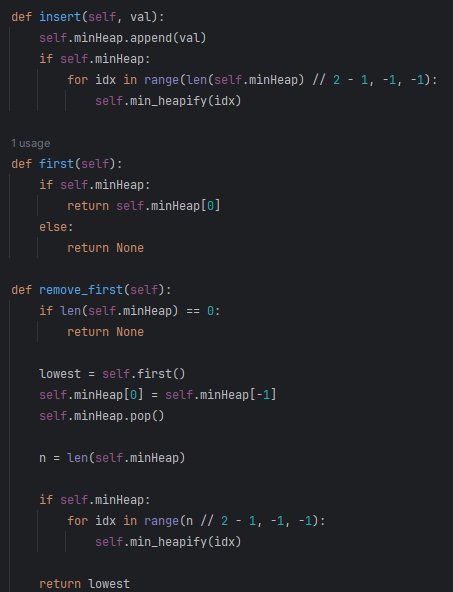
1. **Implement a *minPriorityQueue* using a min-heap. Given a heap, the *MinPriorityQueue* acts as follows:**

• *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

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

**Fig 2.** minPriorityQueue function



**Fig 3.** insert(), first(), and remove\_first() functions