Homework Assignment Week 3

Sorting

Name : Woonki Kim

Email : Kimwoon@oregonstate.edu

1. a) Implement the `heapSort` algorithm. Given an input array, the `heapSort` algorithm acts as follows:

- **Builds a max-heap from the input array (instead of the min-heap provided in the exploration).**
- **Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.**
- **Discard this last node (knowing that it is in its correct place) by decreasing the heap size, and calling MAX-HEAPIFY on the new (possibly incorrectly-placed) root.**
- **Repeat this discarding process until only one node (the smallest element) remains, and therefore is in the correct place in the array.**

```
def max_heapify(arr, n,i ):
    max = i
    l = 2*i + 1
    r = 2*i + 2

    if l<n and arr[l]>arr[max]:
        max = l

    if r<n and arr[r]>arr[max]:
```

```
        max = r

    if max != i:
        arr[i], arr[max] = arr[max], arr[i]
        max_heapify(arr, n, max)


def build_max_heap(arr):
    n = len(arr)
    for i in range(n//2, -1, -1):
        max_heapify(arr, n, i)

def heapSort(arr):
    n = len(arr)

    build_max_heap(arr)

    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        max_heapify(arr, i,0)

    return 0
```

b) Derive the time complexity of the `heapSort` algorithm and provide test cases. Test your implementation using:

- **A long list of random numbers**
- **An already sorted array**
- **A reversely sorted array**

Compare the time taken in each case to the theoretical time complexity and report any differences.

1. Deriving the time complexity of the heapSort.

Max-heapify function:

- initial steps:

  Computing l and r and comparing max with left and right children takes constant time $O(1)$.

- Recursive:

  If the max value is one of the children, max_heapify function swaps values and makes a recursive call to the subtree which has child as a root. In the worst case, the subtree's size is approximately $\frac{2n}{3}$. Which means max_heapify function calls itself recursively with updated input n to $\frac{2n}{3}$ in worst case.

  Thus, $T(n) = T(\frac{2n}{3})$

- Total time estimated:

  $T(n) = T\left(\frac{2}{3}n\right) + O(1)$

- Analyze Time complexity using recurrence relation(Master Theorem).
  - General Form of Mater Theorem:

    $T(n) = aT\left(\frac{n}{b}\right) + f(n^d)$

    Since $a = 1 \geq 1$, $b = \frac{3}{2} > 1$, $f(n^d) = n^0 = 1 > 0$, we can use master theorem.
  - $\log_b a = \log_{\frac{3}{2}} 1 = 0$, $d = 0$
  - By case 2 of the Master Theorem applies when $\log_b a = d$. In this case, the time complexity is:

    $T(n) = O(n^d \log n) = O(\log n)$

Build-max-heap function:

- Simple Analysis:

  As we have proven above, max_heapify function takes $O(\log n)$ time, where n is the number of elements in the heap. Since the total number of build_max_heap function calling max_heapify is n times, build_max_heap function's time complexity is $O(n \log n)$.

- Tighter Analysis introduced in our textbook *"Introduction to Algorithms by Cormen, Leiserson, Rivest, Stein, 3rd Edition".*
  - The main idea of tighter time complexity bound is that as height increases, nodes that are in height decreases.
  - Height of heap with n elements: $\lfloor \log n \rfloor$
  - Number of nodes in height h: $N(h) \leq \frac{n}{2^{h+1}}$
  - Time complexity of max_heapify at a node of height h : $O(h)$
  - $T(n) = \sum_{h=0}^{\lfloor \log n \rfloor}$(number of nodes at height h $\times$ (cost of max_heapify at height h)
    $= \sum_{h=0}^{\lfloor \log n \rfloor} \left(\frac{n}{2^{h+1}} \times O(h) \right)$

    $= n \sum_{h=0}^{\lfloor \log n \rfloor} \left(\frac{h}{2^{h+1}} \right)$ *(since n is considered constant in the summation)*

    $= n \sum_{h=0}^{\lfloor \log n \rfloor} \left(\frac{1}{2} \frac{1}{2^h} \right)$

    $= n \sum_{h=0}^{\lfloor \log n \rfloor} \left(\frac{1}{2} \times 2 \right)$ *(since,* $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$*)*

    $= O(n \cdot 1) = O(n)$

    Thus, with tighter bound, build_max_heap function's time complexity is:

    $O(n)$

heapSort function:

- heapSort function calls build_max_heap function to build heap:

  $O(n)$

- heapSort function iterates $n-1$ times of swapping and calling max_heapify. Since swapping takes constant time:

  $n \cdot O(\log n) = O(n \log n)$

- Total:

  $O(n) + O(n \log n) = O(n \log n)$ *(since* $O(n \log n)$ *dominates* $O(n)$*)*

2. Time measured

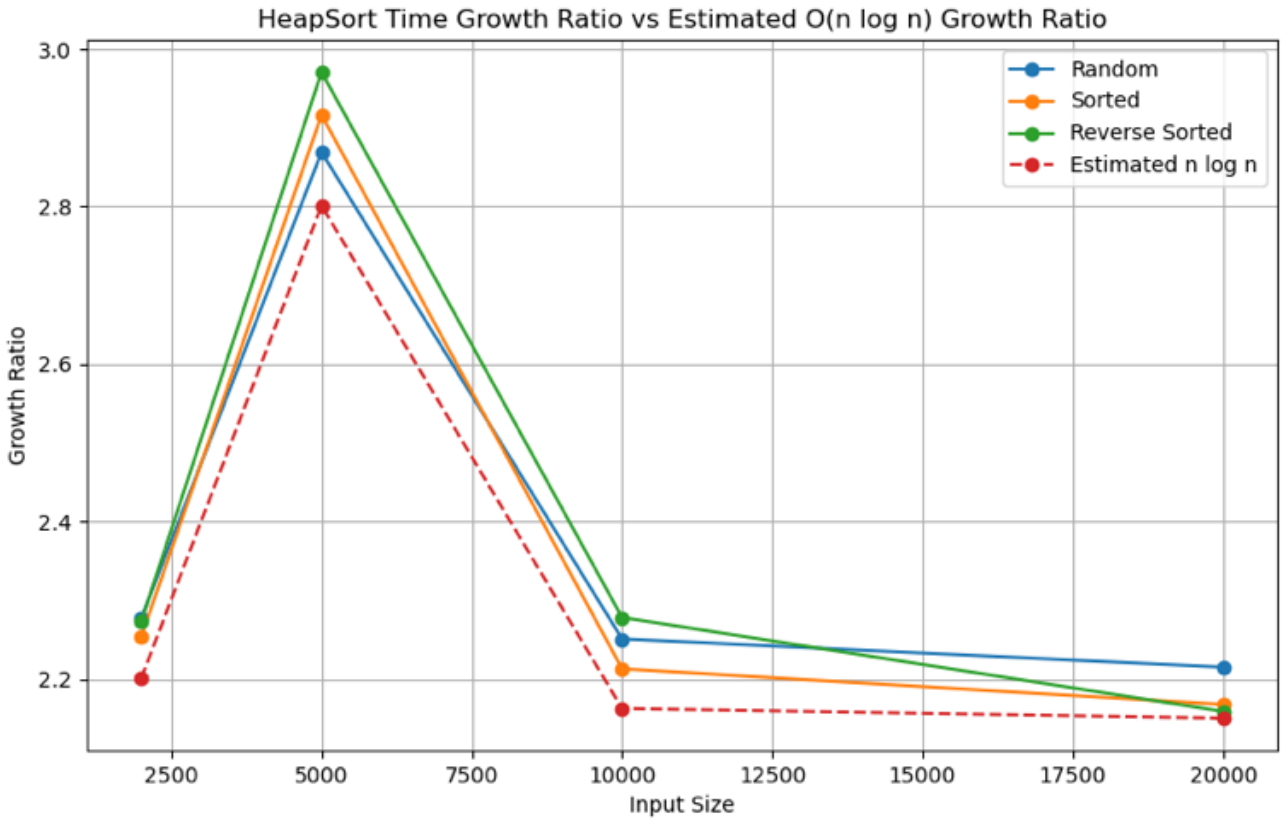
Estimation of $n \log n$ and time measured for random, sorted, reverse sorted array.

Input Size	Estimated n log n (s)	Random Time (s)	Sorted Time (s)	Reverse Sorted Time (s)
1000	9965.784285	0.001271	0.001268	0.001107
2000	21931.568569	0.002662	0.002864	0.002407
5000	61438.561898	0.007712	0.007967	0.006977
10000	132877.123795	0.016719	0.017688	0.015812
20000	285754.247591	0.037890	0.038399	0.034484

Growth ratios of $n \log n$ and for random, sorted, reverse sorted array.

Input Size	Estimated Growth Ratio	Random Growth Ratio	Sorted Growth Ratio	Reverse Growth Ratio
2000	2.20	2.09	2.26	2.17
5000	2.80	2.90	2.78	2.90
10000	2.16	2.17	2.22	2.27
20000	2.15	2.27	2.17	2.18

Plot of growth rate



3. Comparing the time taken in each case to the theoretical time complexity and report any differences.

As shown in the plot of growth rates, each array follows a growth pattern similar to the $n \log n$, which is same with my theoretical analysis. This tendency is different compared to quick_sort and merge_sort we discussed in previous lecture, since they are affected by how the array is sorted. And here's my explanation:

> Heapsort takes the same time for sorted, random, and reverse-sorted arrays because it is based on heap operations, not input order. The first step, building the heap, always takes $O(n)$ time, without getting affected by the array's initial state. Then, during sorting, it repeatedly finds the maximum element and max_heapifies, which always takes $O(n \log n)$time. Unlike other sorting algorithms, heapSort is not affected by how array is sorted. Since the heap structure is unaffected by input order, the overall time complexity remains $O(n \log n)$ in all cases.
>
> To summarize, heapSort always start by building heap structure which always takes O(n) times.

After that the other processes are going to be exactly same for sorted, random, and reverse-sorted array.

2) Argue the correctness of `heapSort` algorithm using the following loop invariant: At the start of each iteration of the for loop, the subarray $A[1..i]$ 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. 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 $arr[1...i]$ is a max-heap containing the i smallest elements of $arr[i..n]$, and the subarray $arr[i+1...n]$ contains the $n-i$ largest elements of $arr[1...n]$ sorted.

Initialization:

Since we assumed that build_max_heap is correct, after calling build_max_heap function, $arr[1...n]$ is a valid max-heap. Which is equivalent of saying that, subarray $arr[1...n]$ which is not sorted yet, is a max-heap containing the n smallest elements. At this point there is no sorted array yet.

Thus the invariant holds.

Maintenance:

- Subarray $arr[i+1...n]$

  During the for loop, largest element in $arr[1..i]$, which is $arr[1]$, is swapped with the last unsorted element, which is $arr[i]$. After this swap, the subarray $arr[i+1...n]$ conatains $n-i$ largest elements in sorted order.

- Subarray $arr[1...i]$

  Now, there is a big possibility of $arr[1...i]$ being invalid max-heap because the root element has changed.

  So what we do next is to call max_heapify function to make $arr[1...i]$ valid max-heap. Since we assume that max_heapify function is correct, the subarray $arr[1...i]$ is once again a valid max-heap.

  For every iteration of loop, starting from $i = n - 1$ and decrementing i by one, we send $n - i$ largest element to sorted subarray part($arr[i + 1...n]$) and consistently building max_heap with max_heapify function, which makes subarray of $arr[1...i]$ to be max_heap while holding i smallest elements of $arr[1...n]$.

Thus the invariant holds.

Termination:

When loop terminates, subarray $arr[2..n]$ is sorted and only one elements is in the heap which is $arr[1]$. Since we have proven in maintenance that subarray $arr[2...n]$ holds $n - 2$ largest elements that is sorted, and the fact that $arr[1]$ holds $1st$ smallest elements, the whole array can be said to be sorted.

Thus the invariant holds.

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/mimimum priority value

```
class MinPriorityQueue:
```

```
    def __init__(self):
        self.heap = []

    def min_heapify(self, n, i):
        min = i
        l = 2 * i + 1
        r = 2 * i + 2

        if l < n and self.heap[l] < self.heap[min]:
            min = l

        if r < n and self.heap[r] < self.heap[min]:
            min = r

        if min != i:
            self.heap[i], self.heap[min] = self.heap[min], self.heap[i]
            self.min_heapify(n, min)

    def insert(self, element):
        self.heap.append(element)
        index = len(self.heap) - 1

        while index > 0:
            parent_index = (index - 1) // 2
            if self.heap[parent_index] > self.heap[index]:
                self.heap[index], self.heap[parent_index] = self.heap[p
arent_index], self.heap[index]
                index = parent_index
            else:
                break

    def first(self):
        if len(self.heap) > 0:
            return self.heap[0]
        return None
```

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

        min_element = self.heap[0]
        last_element = self.heap.pop()

        if len(self.heap) > 0:
            self.heap[0] = last_element
            self.min_heapify(len(self.heap), 0)

        return min_element
```