# CS519 Algorithms Design, Analysis, and Implementation
## Homework 3
## Aashish Adhikari

### 1. Implement Heap sort. (You could see pages 151--160 of CLRS for a description and analysis, but note that the arrays are indexed starting with 1).

### You write three routines, Maxheapify which heapifies the input array, BuildMaxHeap, which builds the MaxHeap and finally HeapSort, which sorts the input array. The testing program tests HeapSort which should use the other two.

**Solution**

```
def Maxheapify(array_to_maxheapify, length_of_passed_array, element_index_to_heapify):


    '''Store the parent's index in the temp variable'''
    temp_largest_among_parent_left_right = element_index_to_heapify
    '''Find the indices for the children'''
    left_child_index = 2 * element_index_to_heapify + 1
    right_child_index = left_child_index + 1



    '''Check if left child is larger than the parent'''
    if left_child_index < length_of_passed_array:
        if array_to_maxheapify[left_child_index] >
array_to_maxheapify[temp_largest_among_parent_left_right]:
            temp_largest_among_parent_left_right = left_child_index



    '''Check if the right child is larger than both, parent and the left child'''
    if right_child_index < length_of_passed_array:
        if array_to_maxheapify[right_child_index] >
array_to_maxheapify[temp_largest_among_parent_left_right]:
```

```
        temp_largest_among_parent_left_right = right_child_index


    '''In case the parent was not the largest, swap parent with the largest and heapify the index swapped with the parent'''

    if temp_largest_among_parent_left_right != element_index_to_heapify:

        array_to_maxheapify[element_index_to_heapify], array_to_maxheapify[temp_largest_among_parent_left_right] = array_to_maxheapify[temp_largest_among_parent_left_right], array_to_maxheapify[element_index_to_heapify]

        Maxheapify(array_to_maxheapify, length_of_passed_array, temp_largest_among_parent_left_right)


def BuildMaxHeap(arr):


    for i in range(int(len(arr)/2), -1, -1): #upto -1 since range() stop one step earlier

        Maxheapify(arr, len(arr), i)



def HeapSort(arr):

    no_of_elements_in_array = len(arr)

    BuildMaxHeap(arr)


    print("MaxHeap built as: ", arr, " Now, sorting the maxheap.\n")


    for i in range(no_of_elements_in_array-1, 0, -1):

        '''Exchange the last element with the largest element that is on top of the heap.'''

        arr[i], arr[0] = arr[0], arr[i]

        print("After the root element is swapped with the last unsorted element, array looks like: ",arr)

        Maxheapify(arr, i, 0)
```

```
    print("After maxheapifying, the array looks like this. ",arr)

return arr
```

2. **Consider two *sorted* arrays *A* of size *m* and *B* of size *n.***

- **(a) Design an efficient (*O(log m + log n)*) Divide-and-Conquer algorithm to find the *k*'th element in the merged array. To be efficient, you should do this without actually merging the two arrays.**

Solution

Consider we have two sorted arrays A and B with lengths m and n respectively.

Algorithm:

**STEP (A,B)**

If A[0] > B[m]:

$k^{th}$ element = B[k]

Elseif B[0] > A[n]:

$k^{th}$ element = A[k]

Else:

Divide A into A1[0…...,m/2] and A2[(m/2)+1…….m] and B into

B1[0…...n/2] andbA2[(n/2)+1…….n]

If |A1| + |B1| >= k:

**A ← A1, B ← B1, m ← m/2, n ← n/2**

**STEP(A,B)**

Else:

**K ← K - |A1| - |B1|**

**A← A2, B ← B2, m ← m/2, n ← n/2**

**STEP(A,B)**

- **(b) Prove that the time complexity is *O(log n + log m).***

**Solution**

- We start with two arrays of sizes m and n.
- In the worst case, at each step, we are reducing the size of each array by half. I.e., m/n and n/2.
- This is similar to binary search. The maximum number of times that a binary search can go for input size k is log k.
- However, instead of nesting one search inside another, we are doing these steps sequentially. I.e., the size of the input is reducing together. Hence, For these two different operations, we add the time complexities. I.e., O(logm + logn)
- Variation of the input : (m,n) → (m/2 , n/2) → (m/4 , n/4) + …… which becomes O(log m + log n) in the worst case.

**3. Report how long you worked on**

- **Coding Heapsort**
- **Making it work**
- **Answering the different parts of question 2.**

Solution

It took me about an hour to code the heap sort. I made some mistake while running the loop and was getting error for some input. But my code gave “time limit exceeded” output for the last few output cases and I spent hours trying to learn better ways - things like iterative heap sort and implemented them. Still the last test case exceeded the time limit. But then again after I ran the original code a few hours later, the same code worked for all test cases. So I kind of spent a lot of time trying to make things faster

while my code was fine the whole time and it was not working within the time limit earlier because of a busy server.

For the second question, coming up with a divide and conquer strategy for this question took about 30 minutes most probably because of the hint. Proving was also straightforward.

**Note**: The server is returning “correct output on all cases” and “time limit exceeds” for the same code stochastically. So I am assuming that either the input is generated randomly to test the code or the server gets slow when used by other students to run their code. Hence I am attaching the response screen to illustrate that the submission works. Also, the names of the variables in the code submitted in the python file and in this file vary since I refactored the names to see if that helps the “time limit” problem.