# ADS
# Homework 4

Aashsh Paudel

March 8, 2019

## Problem 1

```
def bubblesort(arr, n(=length)):
    swapped = True        # n  times
    #Here, we loop through all the values
    for i in 1 -> n && swapped:        # $n + 1$ times
        '''
        We introduce swapped variable because we want to make our algorithm
        adaptive.
        '''
        swapped = False          # n times
        '''
        We loop through 1 to n-i
        For this algorithm loop invariant is the data from n-i to n
        is always sorted. And we continue to bring the small values from
        beginning to end with every loop.
        '''
        for j in 1 -> n-i:         # $\sum_{j=1}^{n-i+1} t_j$

            '''
            Here we try to achieve what is said above about pushing large values
            Backwards and pulling smaller values forward. To achieve this,
            We just swap the two elements if they are in different order.
            '''
            if arr[j] > arr[j+1]:     # $\sum_{j=1}^{n-i} t_j$
                SWAP(arr[j], arr[j+1])  # $\sum_{j=1}^{n-i} t_j$
                swapped = True          # $\sum_{j=1}^{n-i} t_j$

                '''
                Here, $t_j$ is the number of times while loop is executed!
                '''
```

## (b)

Consider, the sequence of numbers: $\mathbf{a_1}, \mathbf{a_2}, \mathbf{a_3}, \ldots.., \mathbf{a_n}$
**Best case:** $a_1 < a_2 < \ldots. < a_n$
**Time complexity**
The time complexity at the best case is: $\theta(n)$
***Proof:***
Dry run in case of $a_1 < a_2 < \ldots < a_n$:
swapped = True
first for loop executed (1st time)
swapped = False
second for loop executed (n times)
/*Program executes*/

Thus the program solely depends upon the number of elements(n+1) and thus
The time complexity $\theta(n)$

**Worst case:** $a_1 > a_2 > \ldots. > a_n$
**Time complexity**
The time complexity at the worst case is:
$mathbftheta(n^2)$
***Proof:***
In worst case, $t_j = n, n-1, n-2, n-3, n-4, \ldots 2, 1$
Total number of times the code inside inner loop runs (as per the comment on the pseudo code)

$$= \sum_{j=1}^{n-i+1} t_j$$
$$= n + (n-1) + (n-2) + (n-3) + \ldots + 2 + 1$$
$$= \frac{(n)(n+1)}{2}$$

Thus, the time complexity $\theta(n^2)$

**Average case:** Normally everything else! (Randomly placed data)
**Time complexity**
The time complexity at the average case is:
$mathbftheta(n^2)$
***Proof:***
In average case, $t_j = n, n-1, n-2, n-3, n-4, \ldots n/2$
(We say that this is the average of all cases)
Total number of times the code inside inner loop runs (as per the comment on

the pseudo code)

$$= \sum_{j=1}^{(n-i+1)/2} t_j$$
$$= n + (n-1) + (n-2) + (n-3) + ... + (n/2)$$
$$= \frac{(n)(2n-1)}{4}$$

Thus, the time complexity $\theta(n^2)$

## (c)

Consider the sequence of numbers $a_1, a_2, a_3, ..., R, ..., S, ..., a_{n-1}, a_n$
, where $R = S$
**Insertion sort is stable!**
Say variable i loops through our sequence. In this sense we can imagine i as a cursor(in fact an index) that points to each value of the sequence. Loop invariant technique says us that the sequence left of the cursor is sorted. Consider the cursor is at S!
At that time, the sequence will look like $a_{j1}, a_{j2}, ...R, ..., S, ...$
This is because we do not move any value ahead than the cursor itself. Thus, until this time there is no chance R will be later in the series than S. At this point the S as it is key will keep being compared until this comparision happens: $isR > S?(= False)$. Thus, the new series will be $a_{j1}, a_{j2}, ...R, S, ...$
Thus, Insertion sort is stable.

**Merge Sort is stable!**
Say after n merges we put ourselves into this situation:
Merge($[a_1, a_2, ..R, ..a_j], [a_{j+1}, ...S, ...a_k]$)
Until this, we do not have to worry about R and S switching places because merge sort makes exchanges in array only during merging time and not during splitting. And we shall not worry about time hereafter because After these two nodes have been merged one will never come infront of other.
At this step, the comparision will go like this:
Is $(a_1 <= a_{j+1})$?
. . . . .
. . . . .
. . . . .
Is $(R <= S)$ (=True)
Thus, the R will be picked up first and put first in the array and only after that S (! hopefully if there does not lie T (=S) after R).
Thus, merge sort is stable.

**Heap sort is not stable!**
Consider the sequence $20(a), 20(b), a_1, a_2, ..., a_n$ which is already in max-heap. The next step will be $a_n, 20(n, a_1, ..., a_n - 1)$ and 20(a) will be added last in the ouput array. Thus, 20(a) will be later than 20(b) in any case. Thus, the stability is not maintained.
Thus, heap sort is not stable.

**Bubble sort is stable!** Say i loops through all the elements in a sequence where j loops from 1 to $n - i$.
The cursor j (infact an index) will point to R and S time and again and swap R with the value right to it and swap S with the value left to it (consider there lies no value (say T) between R and S st. R=S=T).
With the process, R and S will encounter each other and the following comparision will be made:
Is $(R > S)$? (= False). Thus the swapping will not be made! Hence R always stays infront of S.
Hence, Bubble sort is stable!

**(d)**

Consider the sequence $a_1, a_2, a_3, ..., a_n = K$(say).

**Insertion sort is adaptive!**
Consider the subsequence $a_1, a_2, ..., a_j$ which is in sorted order.
Say a loops 2-¿j while b loops a ¡- 2. From the insertion sort we know that b loops iff K[a-1] ¿ K[a] which is False for our subsequence. Thus the second loop (loop b) will not be executed while i moves from $a_1$ to $a_j$ the second loop will not run giving the time complexity only $\theta(n)$. From this case of subsequence, we can say that insertion sort is adaptive.

**Merge sort is not adaptive**
Consider the subsequence $a_j, a_2, ..., a_k$ which is in sorted order.
After the dividing the sequences is over, we will have leaves such as: $a_1, ...a_j, a_{j+1}, ..., a_k, ..., a_n$.
And merging will happen with our normal merge algorithm with nor regards of the already sorted subsequence whatsoever! As this is the case for all the sorted subsequences, we can say that merge sort is not adaptive.

**Heap sort is not adaptive**
Consider the subsequence $a_{j_1}, a_2, a_3, ..., a_{j_n}$ which is pre-sorted.
While building the max-heap, the heapsort will ignore the sorted-ness of this array regardless of its position in the greater sequence. The algorithm will choose the $M > (a_1, a_2, ....)$, where $M$ might not belong to pre-sorted sequence. And it will choose $M_1, M_2 > (a_1, a_2, ....)$, where $M_1, M_2$ might not belong to pre-sorted order, and this continues.
Thus, while building the max-heap heap sort does not take into account of pre-sorted array. And every other procedure will run as it will be!

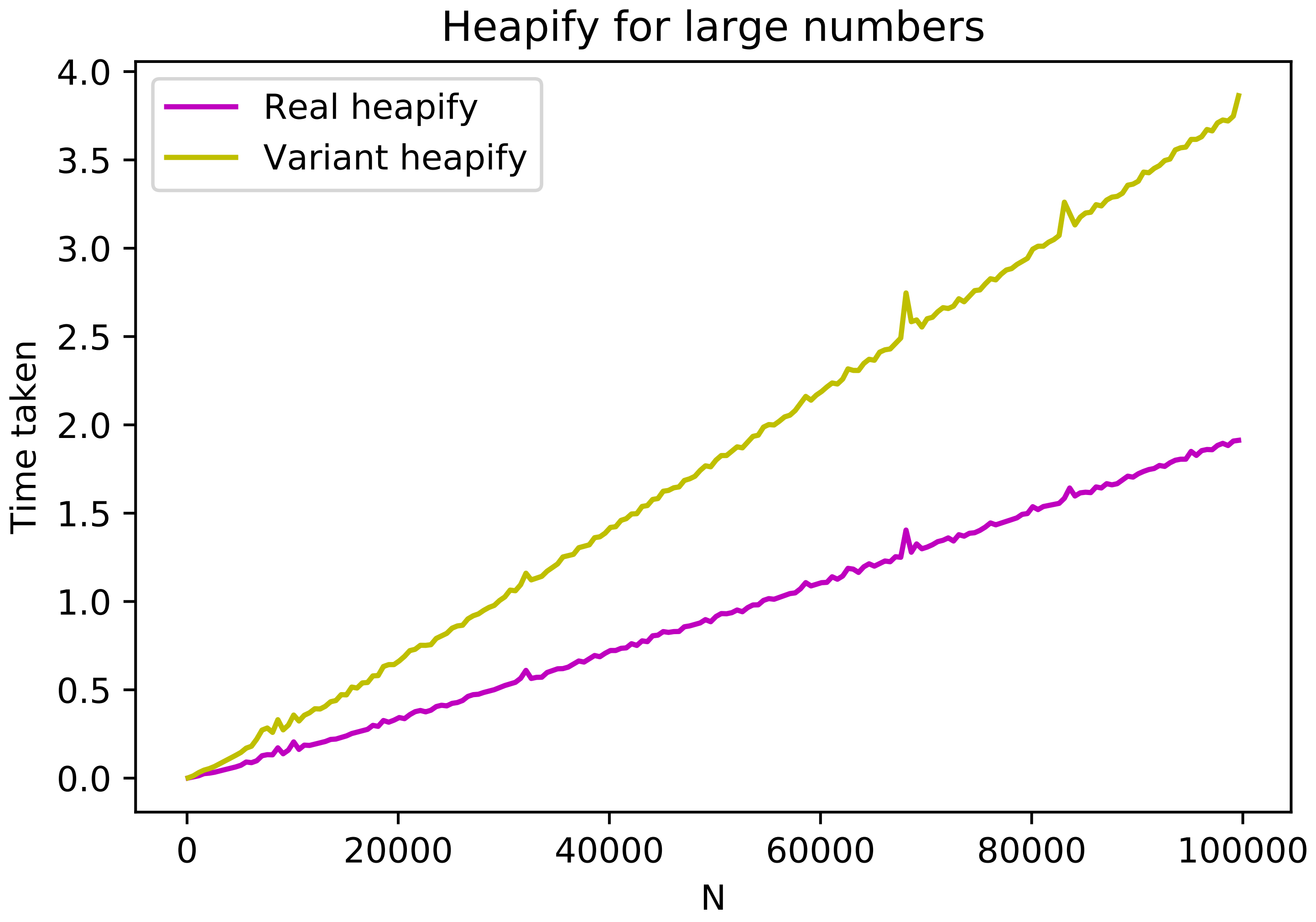
Thus, heap sort is not adaptive.
**Bubble sort is adaptive**
"'The explanation might reference the solution in **(a)**, **(b)**"' The best case of heap sort if $\theta(n)$ while worst case is $\theta(n^2)$
This cleary shows it used the pre-sorted sequence as advantage which makes it adaptive

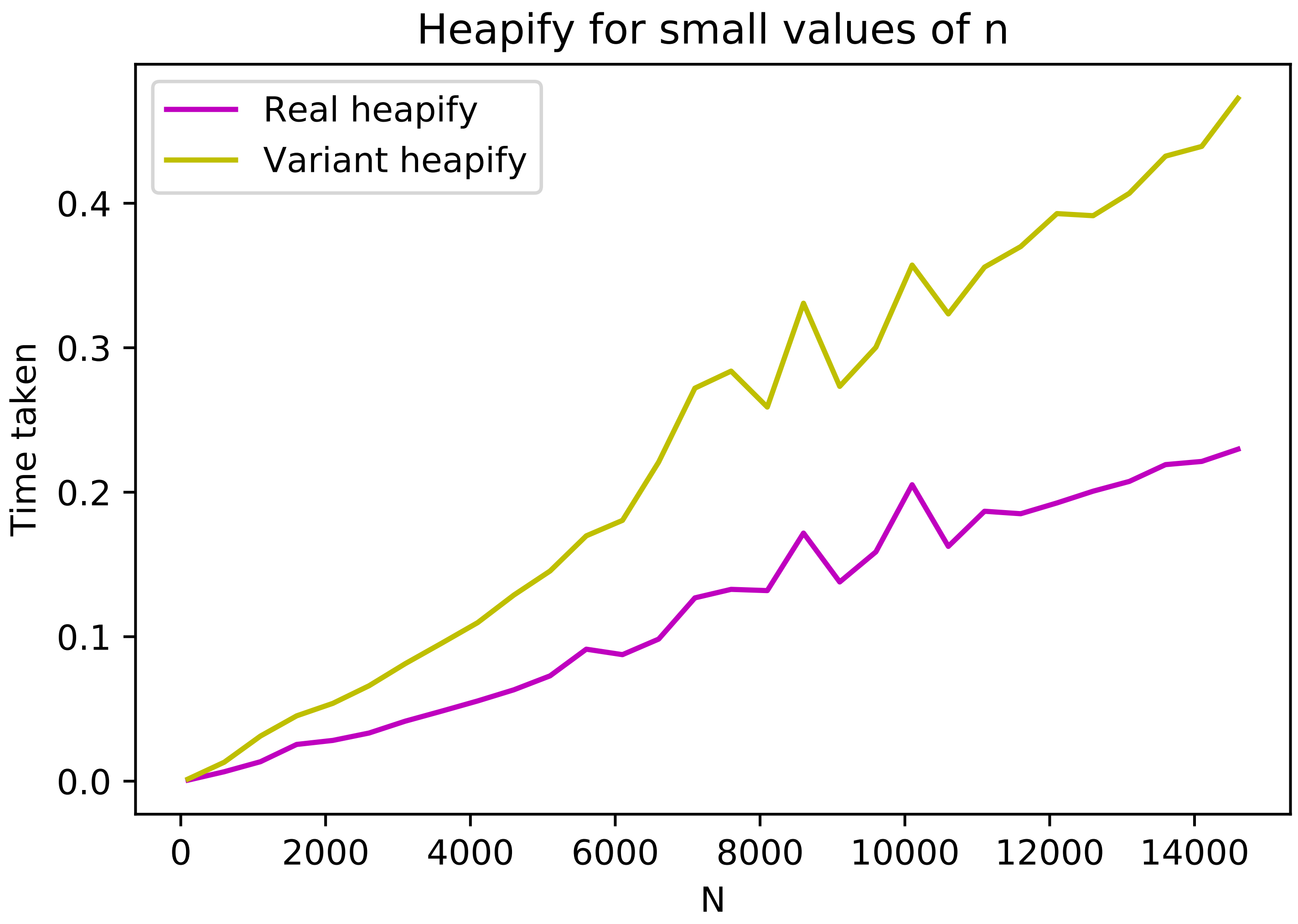
## Problem 2

The code for algorithms and code for plotting with log file is attached with the zip file!

### (c)

The pdf for large values of n and small values of n is given below:


Heapify for large numbers
Real heapify
Variant heapify
Time taken
N
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
20000
40000
60000
80000
100000


Heapify for small values of n
Real heapify
Variant heapify
Time taken
0.0
0.1
0.2
0.3
0.4
N
0
2000
4000
6000
8000
10000
12000
14000