Algorithms and Data Structures Sindi Tasellari

Homework 6

Problem 6.1 Bubble Sort & Stable and Adaptive Sorting

a) **Pseudocode:**

```
BubbleSort (A, n)
        /* Outer loop to go through the elements of the array
           You need to sort n-1 elements
           nth element would automatically be at its correct position      */
    for i = 0 to n-1
          /* make a flag to check swap was executed */
         swapped = false

         /* Inner loop for the number of comparisons
           It goes till n-1-i
           When there are n elements, we do n-1 comparisons
           -i  → the Bubble Sort will be executed only on unsorted data
           therefore, reduce the number of repetitions */
               for j = 0 to n-1-i
                   if A[j] > A[j+1]      //when the current data is bigger than next
                  /* Swapping two datas*/
                   swap (A[j], A[j+1])
                   swapped = 1          //set swap flag
                   end if
             end for

      /* we use one flag variable swapped
         it will help us see if any swap has happened or not.
         If no swap has occurred, the array requires no more processing to be sorted
         it will come out of the loop.  */
```

```
            if swapped == false
            break
            end if

      end for
   end BubbleSort
```

b)

- Worst-case: **O(n^2)**

-The worst case occurs when we want to sort in ascending order and the array is in descending order.

Proof:

- N elements times n-1 comparison; divided by two, since on average over all iterations half of the elements are compared and shifted:

$$n * (n-1) * \frac{1}{2}$$

- When multiplied, it results in:

$$\frac{1}{2} (n^2 - n)$$

- The highest power of this term is n^2 . Therefore:

$$O(n^2)$$

- Average-case: **O(n^2)**

-Average-case occurs when the elements of the array are neither ascending or descending.

Proof:

- There are half as many swap operations as there are in the worst case due to around half of the elements being in the right order when compared to their next element:

$$\frac{1}{4} (n^2 - n)$$

- Number of comparison operations:

$$\frac{1}{2} (n^2 - n * \ln(n) - (\gamma + \ln(2) - 1) * n) + O(\sqrt{n})$$

- The highest term is n^2. Therefore:

$$O(n^2)$$

- Best-case: **O(n)**

-Best-case occurs when the array is already sorted. Then there is no need for sorting.

Proof:

- n-1 comparisons must be carried out, hence:

$$O(n)$$

c)

- Insertion sort, Merge Sort and Bubble Sort are stable.

Insertion Sorting is a stable sorting. In Insertion sorting we just pick a element and places it in its correct place and we are only swapping the elements if the element is larger than the key, i.e. we are not swapping the element with key when it holds equality condition.

Merge sort maintains the position of two equals elements relative to one another.

Bubble Sort is stable because two adjacent elements are compared with each other and they are swapped only if the left element is larger than the right. Meanwhile elements with the same key cannot swap their position relative to each other.

- Heap Sort is unstable.

Heapsort is unstable because operations on the heap can change the relative order of equal items.

d)

- Insertion Sort, Bubble Sort are adaptive.

Insertion sort is adaptive : if input is already sorted then time complexity will be O(n).

Bubble sort is adaptive because for almost sorted array it gives O (n) estimation.

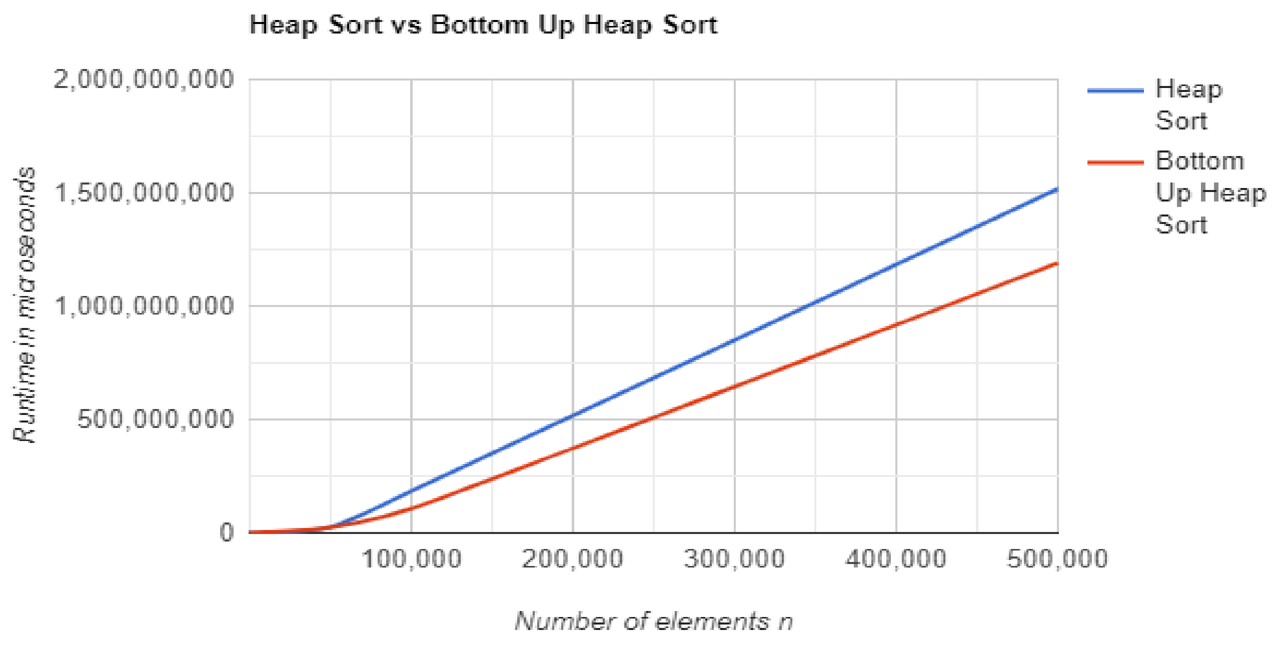
- Merge Sort, Heap Sort are not adaptive.

Merge Sort is not adaptive because the order of the elements in the input array doesn’t matter, time complexity will always be O(nlogn).

Heap Sort is not adaptive because it does not take exciting order within its input into account. Time complexity will always be O(nlogn).

Problem 6.2 Heap Sort

c) The following graph shows the runtimes of Heap Sort compared to its variant Bottom-Up Heap Sort. We observe that Bottom-Up Heap Sort is a faster method than Heap Sort for arrays with the same number of elements. Bottom-Up Heapsort is optimized to reduce the number of comparisons required. It requires fewer compare, read, and write operations than regular heapsort – regardless of the number of elements to be sorted. Thus, making it an optimized variant.


Heap Sort vs Bottom Up Heap Sort
Runtime in microseconds
Number of elements n
Heap Sort
Bottom Up Heap Sort