Assignment 4

Dushan Terzikj

04.03.2018

Problem 1:

a)

```
bubble_sort(arr, len) // arr is the array to be sorted, len is the length of the array
1       swapped ← true // helper variable to indicate whether a swap was performed
2       while swapped = true // repeat while no swap operations are needed (i.e., until array is sorted)
3               swapped ← false // assume that the array is sorted
4               for i ← 1 to len-1
5                       if arr[i] > arr[i+1] // check if they are in the wrong order
6                               swap(arr[i], arr[i+1])
7                               swapped ← true // a swap operation was performed
```

b) Please find it attached below since it is handwritten.

c)

i) **Insertion sort:** Stable. The reason for this is that the comparison loop in insertion sort looks like this:

```
while j > 0 and arr[j] > key // before the loop j ← current index - 1 and arr is the array
        swap(arr[j], arr[j+1])
        j ← j-1
```

Therefore if we have an array [... R, S ....] and current index is on the position of R, first it will place R to its right place in the left sorted subarray. The next iteration, S will be moved to the left until *j* becomes the position of R. According to the loop condition and insertion sort algorithm, S will be placed on the (position of R) + 1, therefore making R appear before S after sorting.

ii) **Merge Sort:** Unstable. Say we have an array [... R, S …] and that array might be divided in more than one way depending on the length of that array:

1) **Case 1:** The array is divided [... R, S] and [...] therefore maintaining R before S after sorting.
2) **Case 2:** [... R, S, …] → [...] and [R, S ....] therefore maintaining R before S after sorting.
3) **Case 3:** [... R, S, ...] → [... R] and [S ...]. After inner sorting, comes the merge part. If we have the conditions:

```
if left[curr_left_idx] < right[curr_right_idx]
        put left inside big array
else
        put right inside big array
```

With this way, the final array will look like this → [... S, R, ...], therefore R does not appear before S like it was before sorting. This can be fixed to be stable if instead of < we use <= in our conditions. **Technically, whether it is stable or not, depends on the implementation.**

iii) **Heapsort:** Unstable. The reason for this is that during the creation of the heap structure, all the positions of elements are lost. There might be some examples that heapsort produces a stable sorted sequence, but that is not the case for most examples.

iv) **Bubble sort:** Stable. Since the comparison condition is the following:

```
if arr[j] > arr[j + 1]
        Swap
```

If arr[j] = R and arr[j+1] = S, they won't be swapped since R = S. It would have been unstable if we use >= instead of >, but that would not make sense.

d)

i) **Insertion sort: Adaptive.** The reason for this is that bubble sort moves the current element to the left until it finds an element that is less than the current element. If the array (or part of it) is already sorted, then the "pushing to the left" loop does not do as much iterations as unsorted parts. Additionally, the best case running time of Insertion sort is $\Omega(n)$ and the reason for that is already sorted subarrays.

ii) **Merge Sort: Not adaptive.** No matter if the array is already sorted, division of the array takes O(lgN) time and the merging takes O(N) time, therefore the total run-time is O(NlgN).

iii) **Heapsort: Not adaptive.** The sortedness of the array does not matter after building the heap, since the structure of the original array is destroyed after building the heap.

iv) **Bubble sort: Adaptive.** If part of the array is sorted, the number of swap operations is decreased, therefore making the run-time faster. In bubble sort case, if the array is already sorted, the running time flushes from $\Theta(n^2)$ *to* $\Theta(n)$

Problem 2:

Please find the solutions for this problem into the files:

- 'Heapsort.cpp' → top-down heapsort implementation with STDIN and STDOUT
- 'Bottom_up_heapsort.cpp' → bottom-up heap sort implementation with STDIN and STDOUT
- 'Plot.png' → plot for running time for both bottom-up and top-down approaches

Both approaches also have generator source files. Those source files were used to generate randomized sequences in order to gather data for the plot. To compile the programs type 'make' into a Unix-like terminal.

a) Check 'heapsort.cpp'
b) Check 'bottom_up_heapsort.cpp'
c) Check 'plot.png'. **Orange line is bottom-up, blue line is top-down.** From the plot we can see that bottom-up approach is slightly faster, that is because when we restore the heap structure after we pop the root, we sometimes do less operations than h = lgN.