ADS Homework 6

Name - Tewodros Adane
Email - tadane@jacobs-university.de

Problem 6.1 – Bubble Sort & Stable and Adaptive Sorting

a) The pseudocode of Bubble Sort:

```
Bubble-Sort(A)
    // repeatedly iterates through the array n times
    for i = 1 to A.length
        // used to check if swaps were performed in the
        current iteration
        isSwapped = FALSE
        // repeatedly iterates n − i times because after i
        iterations the last i elements are sorted
        for j = 1 to A.length − i
            // swaps elements if the next element is smaller a
            and set isSwapped to true
            if A[j] > A[j + 1] then
                swap(A[j], A[j + 1])
                isSwapped = TRUE
        // if no swaps in current iteration then the array is
        already sorted and the algorithm terminates
        if !isSwapped then
            break
```

b) Asymptotic worst-case: In the worst-case the input would be an array A, sorted in reverse. This will take the most swaps because it would need to swap the first element will all elements in the array until A.length - i.

	Cost	times
`for i = 1 to A.length`	c_1	n
`  isSwapped = FALSE`	c_2	n - 1
`  for j = 1 to A.length − i`	c_3	$\sum_{i=1}^{n} t_i$
`  if A[j] > A[j + 1] then`	c_4	$\sum_{i=1}^{n}(t_i - 1)$
`  swap(A[j], A[j + 1])`	c_5	$\sum_{i=1}^{n}(t_i - 1)$
`  isSwapped = TRUE`	c_6	$\sum_{i=1}^{n}(t_i - 1)$
`  if !isSwapped then`	c_7	n - 1
`  break`	c_8	1

The sum of the product of the costs and times is the running time complexity of the algorithm. c_8 is only run once because after that the program terminates.

$$T(n) = c_1 n + c_2(n-1) + c_3\sum_{i=1}^{n} t_i + c_4\sum_{i=1}^{n}(t_i - 1) + c_5\sum_{i=1}^{n}(t_i - 1) + \\ + c_6\sum_{i=1}^{n}(t_i - 1) + c_7(n-1) + c_8$$

For the asymptotic worst-case, the value of t_i is equal to i because for all values of i, there are going to be i swaps, b/c all next items are less than the previous item.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}\ , and\ \sum_{i=1}^{n} i - 1 = \frac{n(n-1)}{2}$$

Therefore,

$$T(n) = c_1 n + c_2(n-1) + c_3\frac{n(n+1)}{2} + c_4\frac{n(n-1)}{2} + c_5\frac{n(n-1)}{2} + c_6\frac{n(n-1)}{2} \\ + c_7(n-1) + c_8$$

$$T(n) = \frac{(c_3 + c_4 + c_5 + c_6)}{2}n^2 + \frac{(2c_1 + 2c_2 + c_3 - c_4 - c_5 - c_6 + 2c_7)}{2}n \\ + (c_8 - c_2 - c_7)$$

Because the complexity depends on the size of the input instead of the constants (the cost of each step), the complexity would be bounded by the highest term.

$$T(n) = O(n^2)$$

Asymptotic best-case: In the best-case the input would be an array A, sorted in ascending order. This will not perform any swaps because for all element the adjacent element would be larger. The algorithm would also exit both loops if there have been no swaps in the current iteration.

	Cost	times
for i = 1 to *A*.length	c_1	n
isSwapped = FALSE	c_2	n - 1
for j = 1 to *A*.length − i	c_3	$\sum_{i=1}^{n} t_i$
if *A*[j] > *A*[j + 1] **then**	c_4	$\sum_{i=1}^{n}(t_i - 1)$
swap(*A*[j], *A*[j + 1])	c_5	$\sum_{i=1}^{n}(t_i - 1)$
isSwapped = TRUE	c_6	$\sum_{i=1}^{n}(t_i - 1)$
if !isSwapped **then**	c_7	n - 1
break	c_8	1

The sum of the product of the costs and times is the running time complexity of the algorithm. c_8 is only run once because after that the program terminates.

$$T(n) = c_1 n + c_2(n-1) + c_3\sum_{i=1}^{n} t_i + c_4\sum_{i=1}^{n}(t_i - 1) + c_5\sum_{i=1}^{n}(t_i - 1) + \\ + c_6\sum_{i=1}^{n}(t_i - 1) + c_7(n-1) + c_8$$

For the asymptotic best-case, the value of t_i is equal to 1 because after one pass through the input the program quits because no swaps are needed.

$$\sum_{i=1}^{n} 1 = n, and \sum_{i=1}^{n} 1 - 1 = 0$$

Therefore,

$$T(n) = c_1 n + c_2(n-1) + c_3 n + c_4(0) + c_5(0) + c_6(0) + c_7(n-1) + c_8$$

$$T(n) = (c_1 + c_2 + c_3 + c_7)n + (c_8 - c_2 - c_7)$$

Because the complexity depends on the size of the input instead of the constants (the cost of each step), the complexity would be bounded by the highest term.

$$T(n) = O(n)$$

Asymptotic average-case: In the average-case the input would be an array A, the contents of which are randomly ordered. For this we could make an assumption that on average-case it will swap half as much as in the worst-case, this was chosen to be somewhat in the middle of the best and worst cases.

	Cost	times
for i = 1 to *A*.length	c_1	n
isSwapped = FALSE	c_2	n - 1
for j = 1 to *A*.length − i	c_3	$\sum_{i=1}^{n} t_i$
if *A*[j] > *A*[j + 1] **then**	c_4	$\sum_{i=1}^{n}(t_i - 1)$
swap(*A*[j], *A*[j + 1])	c_5	$\sum_{i=1}^{n}(t_i - 1)$
isSwapped = TRUE	c_6	$\sum_{i=1}^{n}(t_i - 1)$
if !isSwapped **then**	c_7	n - 1
break	c_8	1

The sum of the product of the costs and times is the running time complexity of the algorithm. c_8 is only run once because after that the program terminates.

$$T(n) = c_1 n + c_2(n-1) + c_3\sum_{i=1}^{n} t_i + c_4\sum_{i=1}^{n}(t_i - 1) + c_5\sum_{i=1}^{n}(t_i - 1) + \\ + c_6\sum_{i=1}^{n}(t_i - 1) + c_7(n-1) + c_8$$

For the asymptotic worst-case, the value of t_i is equal to $\frac{i}{2}$ because for some values of i the algorithm will perform swaps and for others it will not.

$$\sum_{i=1}^{n}\frac{i}{2} = \frac{n(n+1)}{4}\ , and\ \ \sum_{i=1}^{n}\frac{i}{2} - 1 = \frac{n(n-1)}{4}$$

Therefore,

$$T(n) = c_1 n + c_2(n-1) + c_3\frac{n(n+1)}{4} + c_4\frac{n(n-1)}{4} + c_5\frac{n(n-1)}{4} + c_6\frac{n(n-1)}{4} \\ + c_7(n-1) + c_8$$

$$T(n) = \frac{(c_3 + c_4 + c_5 + c_6)}{4}n^2 + \frac{(4c_1 + 4c_2 + c_3 - c_4 - c_5 - c_6 + 4c_7)}{4}n \\ + (c_8 - c_2 - c_7)$$

Because the complexity depends on the size of the input instead of the constants (the cost of each step), the complexity would be bounded by the highest term.

$$T(n) = O(n^2)$$

c) Insertion Sort is stable since it will only insert elements at the position where every element to the right is larger. So, if two elements have equal keys, they'll be put in their position in the order they come in.

   Merge Sort is stable. This is because in the merging step if two elements have the same key the one from the left sub-array (which is the one that comes first in the array) is given precedence.

   Heap Sort is unstable, because when we swap the max element with the last element in the heap, the last element might now come before other elements of the same key and by the end the array might be sorted in a different order than the original.

   Bubble Sort is stable. As it will only swap two adjacent elements if the element to the right is smaller. If they are equal, there wouldn't be any swaps, thus maintaining their relative order.

d) Insertion Sort is adaptive. If the input is already sorted, the algorithm will only have to check once and add the element to the end. This significantly reduces the comparisons, and it will have a time complexity of O(n). Even if only some part of the input is sorted the algorithm will still have better performance.

   Merge Sort is not adaptive because no matter the input's pre-sortedness the algorithm will divide the input and merge them the same way and wouldn't save any unnecessary comparisons.

   Heap sort is not adaptive because it sorts by finding the max element and moving it to the end to sort. And it also does not use the input being sorted is not utilized to select the max element.

   Bubble Sort is adaptive. This is because if the input is sorted, then the algorithm won't have to swap any elements. So, if no swaps are performed in one iteration, it means all the elements are in their correct place and it terminates.

Problem 6.2 – Heap Sort

a) The implementation of heap sort in C++ is shown below:

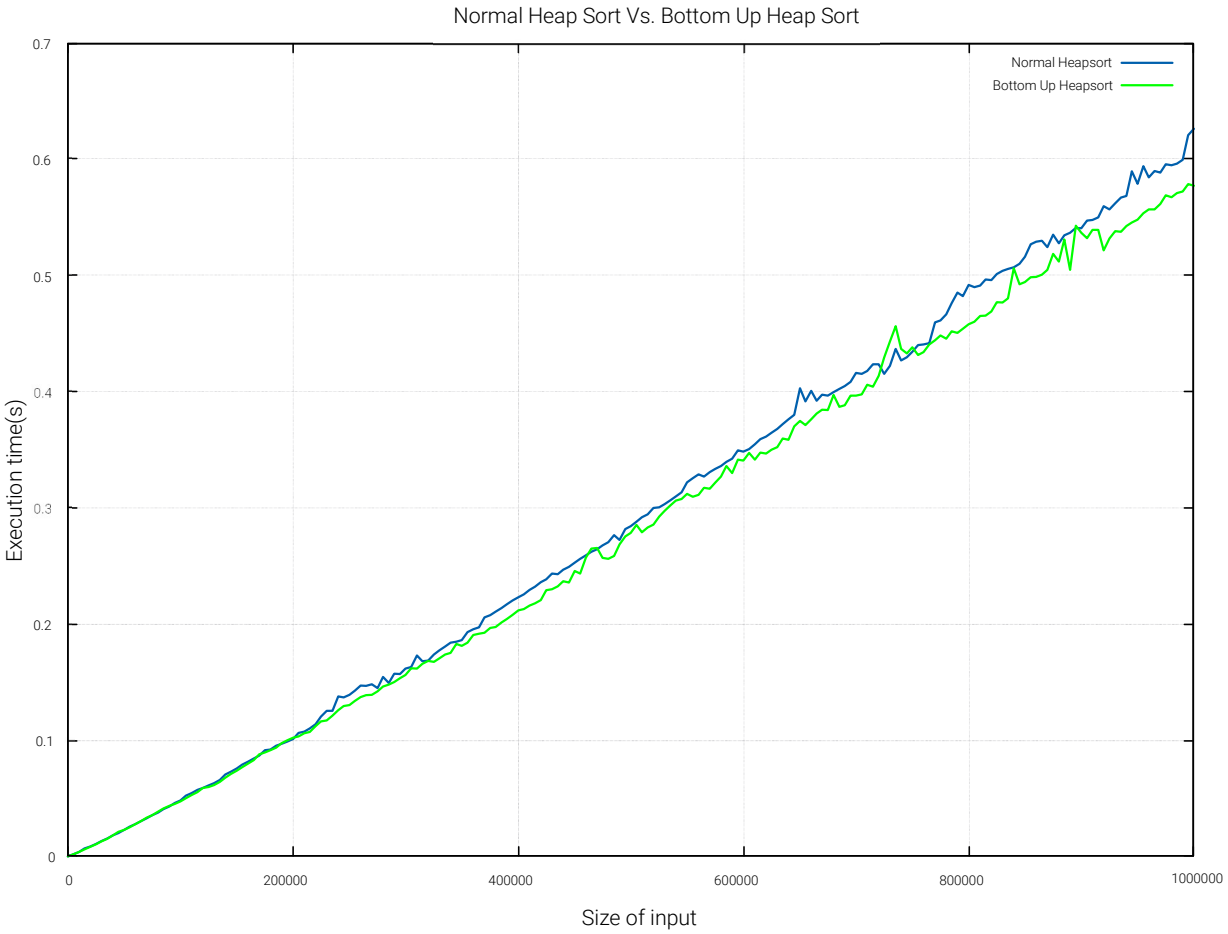
```
void Heap::maxHeapify(int i) {
    int l = left(i);
    int r = right(i);
    int largest = i;

    if (l <= heapsize and heap[l] > heap[i])
        largest = l;

    if (r <= heapsize and heap[r] > heap[largest])
        largest = r;

    if (largest != i) {
        std::swap(heap[i], heap[largest]);
        maxHeapify(largest);
    }
}

void Heap::buildMaxHeap() {
    heapsize = heap.size() - 1;
    for (int i = (heap.size() / 2) - 1; i >= 0; i--)
        maxHeapify(i);
}

void Heap::heapSort() {
    buildMaxHeap();
    for (int i = heapsize; i >= 0; i--) {
        std::swap(heap[0], heap[i]);
        heapsize--;
        maxHeapify(0);
    }
}
```

b) The implementation of bottom up heap sort in C++ is shown below:

```
int Heap::leafSearch(int i) {
    int j = i;
    // goes from i to the leaf, following the path of largest child
    while (right(j) <= heapsize) {
        if (heap[right(j)] > heap[left(j)])
            j = right(j);
        else
            j = left(j);
    }
    // if we stop at an element with only one child and the index of
    the child is in the heap, we set j to that position
    if (left(j) <= heapsize)
        j = left(j);
    return j;
}

void Heap::siftDown(int i) {
    int j = leafSearch(i);
    // incase moving the element up is needed
    while (heap[i] > heap[j])
        j = parent(j);

    // swap the element starting from the bottom and going up
    int x = heap[j];
    heap[j] = heap[i];
    while (j > i) {
        std::swap(x, heap[parent(j)]);
        j = parent(j);
    }
}

void Heap::bottomUpHeapSort() {
    buildMaxHeap();
    for (int i = heapsize; i >= 0; i--) {
        std::swap(heap[0], heap[i]);
        heapsize--;
        siftDown(0);
    }
}
```

Source: Wikipedia

c) The following graph represents the comparison of execution times between the normal heapsort and the bottom-up heapsort for arrays of size 0 to 1,000,000 with an interval of 5000.



From the graph above, it can be observed that for large input sizes bottom-up heap sort takes less time than the normal heap sort. This happens because when searching for the place to put the root element bottom-up heap sort only makes one comparison per level, while max-Heapify makes two comparisons. For small inputs the tree is going to have only a few levels therefore it doesn't show much difference. However, for large inputs the height of the tree is going to be large and every comparison we save on each level adds up and we get a better performing algorithm.