Algorithms and Data Structures

Spring 2019

Assignment 4

Date: March 7, 2019

Taiyr Begeyev

Problem 4.1 Bubble Sort & Stable and Adaptive Sorting

(a) Bubble Sort is a sorting algorithm that works by repeatedly iterating through the list to be sorted, comparing each pair of adjacent items, and swapping them if they are in the wrong order. This is repeated until no swaps are needed, which indicates that the list is sorted. Write down the Bubble Sort algorithm in pseudocode including comments to explain the steps and/or actions.

```
function bubbleSort(A)
    // define the length of the array
    n = A.length
    // repeatedly iterate through the list until no swaps
    // are needed
    repeat
        swapped = false
        // compare each pair of adjacent items
        for i = 1 to n - 1 inclusive do
            // if the next element is greater that the
            // current one, then swap elements
            if (A[i] > A[i + 1]) then
                swap(A[i], A[i + 1])
                swapped = true
    until not swapped
```

(b) Determine and prove the asymptotic worst-case, average-case, and best-case time complexity of Bubble Sort.

The worst case: The worst case occurs when the list is in a descending order(reverse sorted). Let's figure out the worst-case runtime complexity by counting how many times the inner block repeats. At iteration 0, the block runs for $n-1-0$ times. At iteration 1, the block runs for $n-1-1$ times. At iteration $n-1-(n-1)=0$ times. So in total, the block runs $\sum_{i=0}^{n-1}(n-i-1)$ times.

$$\sum_{i=0}^{n-1}(n-i-1) = n^2 - n - \sum_{i=0}^{n-1} i = n^2 - n - \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Therefore, $T(n) = O(n^2)$.

The average case: It is the same as for the worst-case, because the entire array needs to be iterated for every element. $T(n) = O(n^2)$.

The best case: The best case occurs when the list in an ascending order(already sorted). Since all elements are less or equal than the following one, that means that no swapping happens. So for n elements this algorithm performs only n - 1 comparisons. Therefore, $T(n) = O(n)$.

(c) Stable sorting algorithms maintain the relative order of records with equal keys (i.e., values). Thus, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list. Which of the sorting algorithms Insertion Sort, Merge Sort, Heap Sort, and Bubble Sort are stable? Explain your answers

Insertion Sort: Stable sort. It does not change the relative order of elements with equal keys. In other words, repeated elements are in the same order as they appear in the input.
Merge Sort: Stable sort. When merging two halves, we just make sure to use "$L \leq R$" so that we prefer left-half values over right-half values, if they are equal.
Heap Sort: Unstable Sort. The final sequence of the results from Heapsort comes from removing items from the created heap. Any information about the ordering of the items in the original sequence was lost during the heap creation stage, which came first.
Bubble Sort: Stable Sort. Because we swap only if the following element is strictly less than the current one in our Pseudocode.

(d) A sorting algorithm is adaptive, if it takes advantage of existing order in its input. Thus, it benefits from the pre-sortedness in the input sequence and sorts faster. Which of the sorting algorithms Insertion Sort, Merge Sort, Heap Sort, and Bubble Sort are adaptive? Explain your answers.

In the context of sorting, adaptive algorithms are algorithms with better best-case than its worst-case and average case. In other words, it benefits from pre-sortedness. A non-adaptive algorithm takes the same amount of time for a given number of values no matter what. Therefore we can conclude that:
Insertion Sort: Adaptive. Insertion sort's complexity is $O(n^2)$ on worst-case and average case. On the almost sorted arrays insertion sort shows better performance, up to $\Omega(n)$ in case of applying insertion sort to a sorted array.
Merge Sort: Non-adaptive. The time complexity of all cases is $O(nlogn)$. The number of merge operations doesn't chnage.
Heap Sort: Non-adaptive. Number of heapify call doesn't change.
Bubble Sort: Adaptive. It certainly benefits from pre-sorted list. Bubble Sort's complexity is $O(n^2)$ on worst-case and average case. On the almost sorted array insertion sort shows better performance, up to $\Omega(n)$ in case of apllying bubble sort to a sorted array.

Problem 4.2 Heap Sort

(a) Implement the Heap Sort algorithm as presented in the lecture.

```
/**
 * @brief lets the value at arr[i] float down, so that subtree
 * rooted at index i obeys the max-heap property, where parent is
 * greater than any of its children
 *
 * @param array arr
 * @param i - index into the array
* */
void maxHeapify(vector<int>& arr, const int& i){
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    int largest;

    if (left < heapSize && arr[left] > arr[i])
        largest = left;
    else
        largest = i;

    if (right < heapSize && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        maxHeapify(arr, largest);
    }
}

/**
 * @brief in a bottom-up manner make the  arr[1..n] have the max
 * heap property
 *
 * @param arr
*/
void buildMaxHeap(vector<int>& arr){
    heapSize = arr.size();
    for (int i = (arr.size() / 2) - 1; i >= 0; i--)
```

```
        maxHeapify(arr, i);
}

/**
 * @brief sorts the array
 *
 * Start by using buildMaxHeap funnction to build a max-heap. Since
 * the max element of the array is stored at the root, we can put it
 * into its correct position by exchaning it with arr[i]. If we now
 * discard node n from the heap     and we can do so by simply decrementing
 * heapSize     we observe that the children of the root remain max-heaps,
 * but the new root element might violate the max-heap property.
 * All we need to do to restore the max-heap property, however, is call
 * maxHeapify(arr, i, 0). The heapsort algorithm then repeats this process
 * for the max-heap of size n - 1 down to a heap of size 1
 *
 * @param arr
*/
void heapSort(vector<int>& arr){
    buildMaxHeap(arr);
    for (int i = heapSize - 1; i >= 1; i--) {
        swap(arr[0], arr[i]);
        heapSize--;
        maxHeapify(arr, 0);
    }
}
```

(b)

```
/**
 * @brief bottom-way approach. floats the new root all the way down to a
 * leaf level. Then, it checks whether that was actually correct and if
 * not fixes the max-heap by moving the element up again.
 *
 * @param arr the heap array
 * @param i the current root of the subtree
*/
void newApproach(vector<int>& arr, const int& i) {
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    int largest = 0;
    if(left < heapSize && arr[left] > arr[largest])
        largest = left;
    if(right < heapSize && arr[right] > arr[largest])
        largest = right;

    if(largest == 0){
        swap(arr[0], arr[i]);
        return;
    }

    newApproach(arr, largest);
    swap(arr[0], arr[i]);
}

void heapSortModified(vector<int>& arr) {
    buildMaxHeap(arr);
    for(int i = arr.size() - 1; i >= 1; i--){
        swap(arr[0], arr[i]);
```

```
            heapSize--;
            newApproach(arr, 0);
        }
    }
```