**Algorithms and Data Structures**

**CH08-320201**

**Homework 4**

Bubble Sort, Heap Sort & Stable and Adaptive Sorting

**Problem 4.1**

1. **Heap Sort Pseudo Code**

\*/

Bubble Sort Pseudo Code

\*/

repeat

swapped = false

for i = 1 to length(A) - 1 inclusive do:

/\* if this pair is out of order \*/

if A[i-1] > A[i] then

/\* swap them and remember something changed \*/

swap( A[i-1], A[i] )

swapped = true

end if

end for

until not swapped

1. **Asymptotic Complexity of Bubble Sort**

The Bubble Sort algorithm when implemented employs a for loop and an if statement that swaps elements to formulate a sorted array.

Considering the best case for Bubble Sort, if an already sorted array is passed through Bubble Sort the loop will run for a total of *(n – 1)* times with a constant *C* as the time required for the if loop. Since the constant *C* is ignored, the time complexity shall depend on the *(n – 1)* which will satisfy the big-O Notation and since for larger values of *n,* the time complexity shall be completely dependent on *n,* the time complexity for the best case shall be;

*Since c1 \* f(n) >= g(n)*

*f(n) = O(n)*

For the worst case complexity for Bubble Sort, the algorithm will run through the maximum amount of time for a reversed array. This means that the algorithm will run for a total of (*n – 1) + (n – 2) + … + 3 + 2 + 1* times before sorting the reversed array.

Thus, the time complexity for the worst case will be reduced to (*n2-n)/ 2,* where, as n will increase the time complexity will depend on *n2.*

*f(n) = O(n2)*

For the average case, we assume that a semi-sorted array or a randomized array is passed through Bubble Sort. In that case the run-time will still resemble that of the worst case, since it shall be the average of the best and worst case and time complexity will still depend on the quadratic part of the function.

1. **Stability of Sorting Algorithms.**

Stability of sorting algorithms depends on the consistent maintenance of index values of identical elements after passing an array through a sorting algorithm.

This relates to not changing the orientation of the array which only happens in Heap Sort because of an algorithm specific index that is assigned to the different nodes in the Heap. Thus it is likely that identical elements do not appear in a sorted array in the same general position as they were in an unsorted array.

For Merge Sort, Insertion Sort, and Bubble Sort, the general position remains the same due to a simple comparison that is done using an if statement which compares the index *A[i]* with *A[i + 1]* that does not change the orientation of the original elements in the unsorted array as the algorithm sifts through the array in a general top to bottom or bottom to top approach, sorting the elements as the key is incremented.

1. **Adaptive Algorithms**

Adaptive Algorithms use already sorted parts of an array to their advantage by leaving them as be and focusing on the rest of the unsorted parts of an array. Adaptive Algorithms, thus, iterate through the array and leave already sorted parts of the array untouched and move along to place the block of sorted elements in position that the algorithm deems fit. Insertion, Bubble, and Heap Sort all iterate through the array before performing algorithm specific operations on the array. However, Merge Sort specifically divides the array without checking for any sorted segments of the input array at initial invocation. Thus, with the exception of Merge Sort, the sorting algorithms mentioned in the question are adaptive.

void split(int arr[], int sub[], int low, int high)

{

//Split part of mergesort

if (low < high - 1)

{

int mid = (low + high) / 2;

split(arr, sub, low, mid);

split(arr, sub, mid, high);

Merge Sort code from HW2 that shows the division part of the merge sort algorithm which is then passed to the merge function that sorts the divided arrays.

However, Heap Sort is also considered to being “not really adaptive”. This is because both phases are adaptive at the least. In the nearly sorted case, the heapify phase destroys the original order. In the reversed case, the heapify phase is as fast as possible since the array starts in heap order, but then the sort-down phase is typical. In the few unique keys case, there is some speedup but not as much.[[1]](#footnote-1)

1. https://www.toptal.com/developers/sorting-algorithms/heap-sort [↑](#footnote-ref-1)