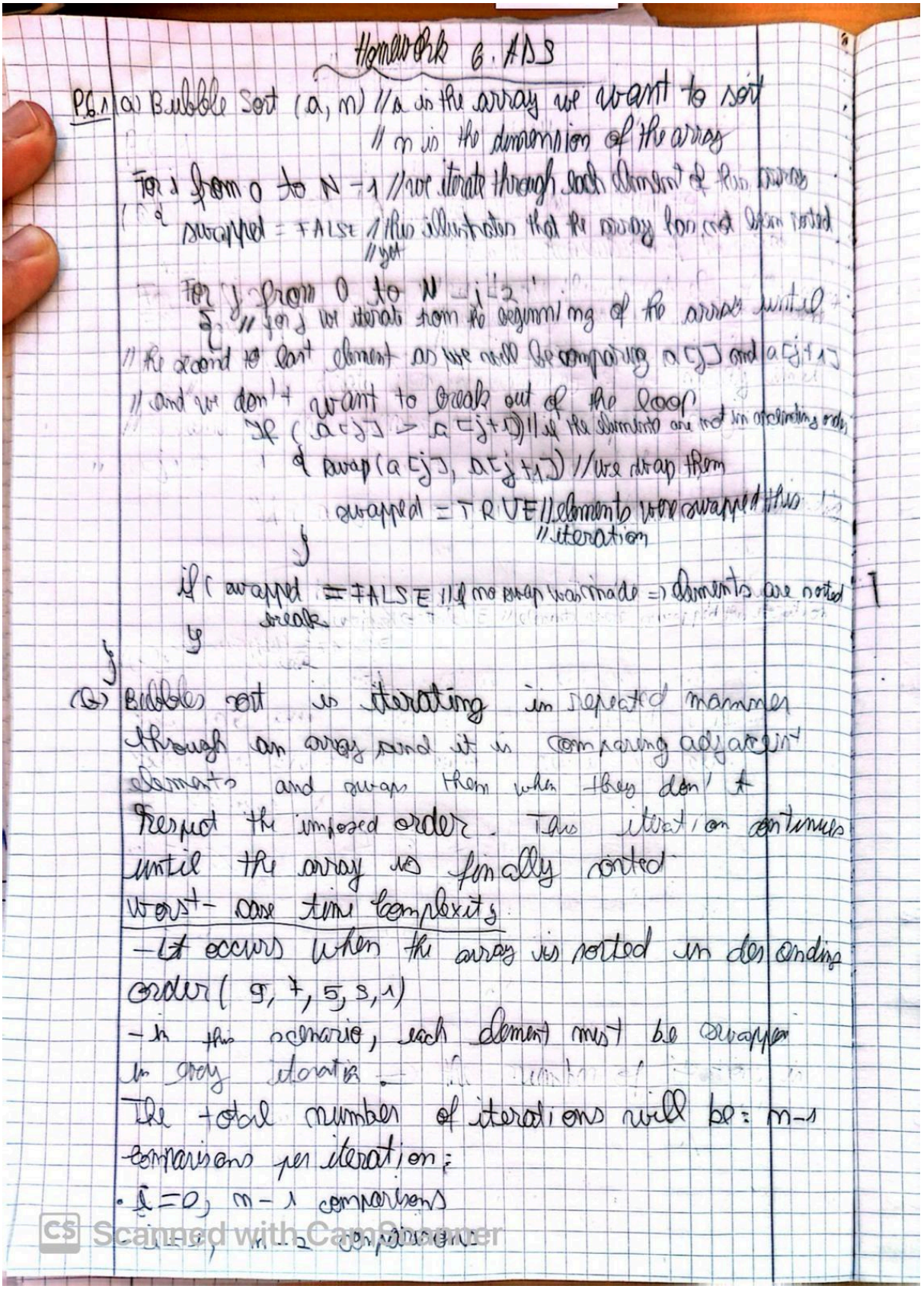
Homework 6

Hw6 1



Homework 6. ADS

Pb.1 a) Bubble Sort (a, n) // a is the array we want to sort
// n is the dimension of the array
for i from 0 to N-1 // we iterate through each element of the array
{ swapped = FALSE // this illustrates that the array has not been sorted yet
for j from 0 to N - i - 2 //
{ // for j we iterate from the beginning of the array until
// the second to last element as we will be comparing a[j] and a[j+1]
// and we don't want to break out of the loop
if (a[j] > a[j+1]) // if the elements are not in ascending order
{ swap(a[j], a[j+1]) // we swap them
swapped = TRUE // elements were swapped this iteration
}
if (swapped == FALSE // if no swap was made => elements are sorted
break
}
}

b) Bubble sort is iterating in repeated manner through an array and it is comparing adjacent elements and swaps them when they don't respect the imposed order. This iteration continues until the array is finally sorted.

Worst-case time complexity:
- It occurs when the array is sorted in descending order (9, 7, 5, 3, 1)
- In this scenario, each element must be swapped in every iteration.
The total number of iterations will be: n-1
comparisons per iteration:
- i = 0, n - 1 comparisons
- i = 1, n - 2 comparisons

i = n-1 → 1 comparison made

Total number of comparisons:

$(n-1)+(n-2)+\dots+2+1 = \frac{(n-1)(n-1+1)}{2}$

$= \frac{n(n-1)}{2} = \frac{n^2-n}{2} \Rightarrow$ time complexity of $O(n^2)$

Average-case time complexity:

occurs when elements are in random order

- On average, in a random array, after the first pass, roughly half of the elements will have been "moved" into their final position

The number of comparisons remains

$(n-1)+(n-2)+\dots+1 = O(n^2)$

Best-case time complexity:

- The list is already sorted $\Rightarrow$ no need for swaps
- The first step will make n-1 comparisons and no swaps will occur. Since no swaps occurred the algorithm breaks and the total number of comparisons = $n-1 \Rightarrow O(n)$ time complexity

(c) Insertion Sort:

- Elements are directly inserted into their correct position one by one. The equal elements are never swapped unnecessarily, thus their order remains unchanged $\Rightarrow$ STABLE

Merge Sort:

- Merge Sort is known to break the list into smaller parts, proceeds to sort and then merge them, while equal elements are kept in the same order.

In the case of which two equal elements come from different parts, the one from the left is always placed first, in this way stability being maintained ⇒ STABLE

Heap Sort

- it uses a heap (complete binary tree that satisfies the heap property: either: 1) Max-Heap (the parent node ≥ its children) or 2) Min-Heap (the parent node ≤ its children)) and it extracts the maximum or minimum element in repeated manner.
- during this process (heapification) elements may be rearranged in a way that changes the relative order of the equal values ⇒ NOT stable

Bubble Sort

→ it repeatedly swaps adjacent elements only when they are out of order ⇒ that the equal elements are never swapped ⇒ the relative order is preserved)
⇒ STABLE,

d) Insertion Sort is adaptive because

- In the best case (the case in which the array is already sorted), it runs in O(n) time complexity because it proceeds to check each element once, making no swaps.

Bubble sort is adaptive when we have the optimization (the additional swapped variable that checks whether a swap has been made or not). In the best case, we stop early when no swaps are being made ⇒ it can run in O(n) for a sorted array

without the optimization, the algorithm always runs in $O(N^2)$ => without the optimization bubble sort is not adaptive

Merge Sort is NOT adaptive because it always takes $O(N \log N)$ time, even if the input is already sorted. It keeps splitting and merging subarrays regardless of the state of the array (sorted or unsorted)

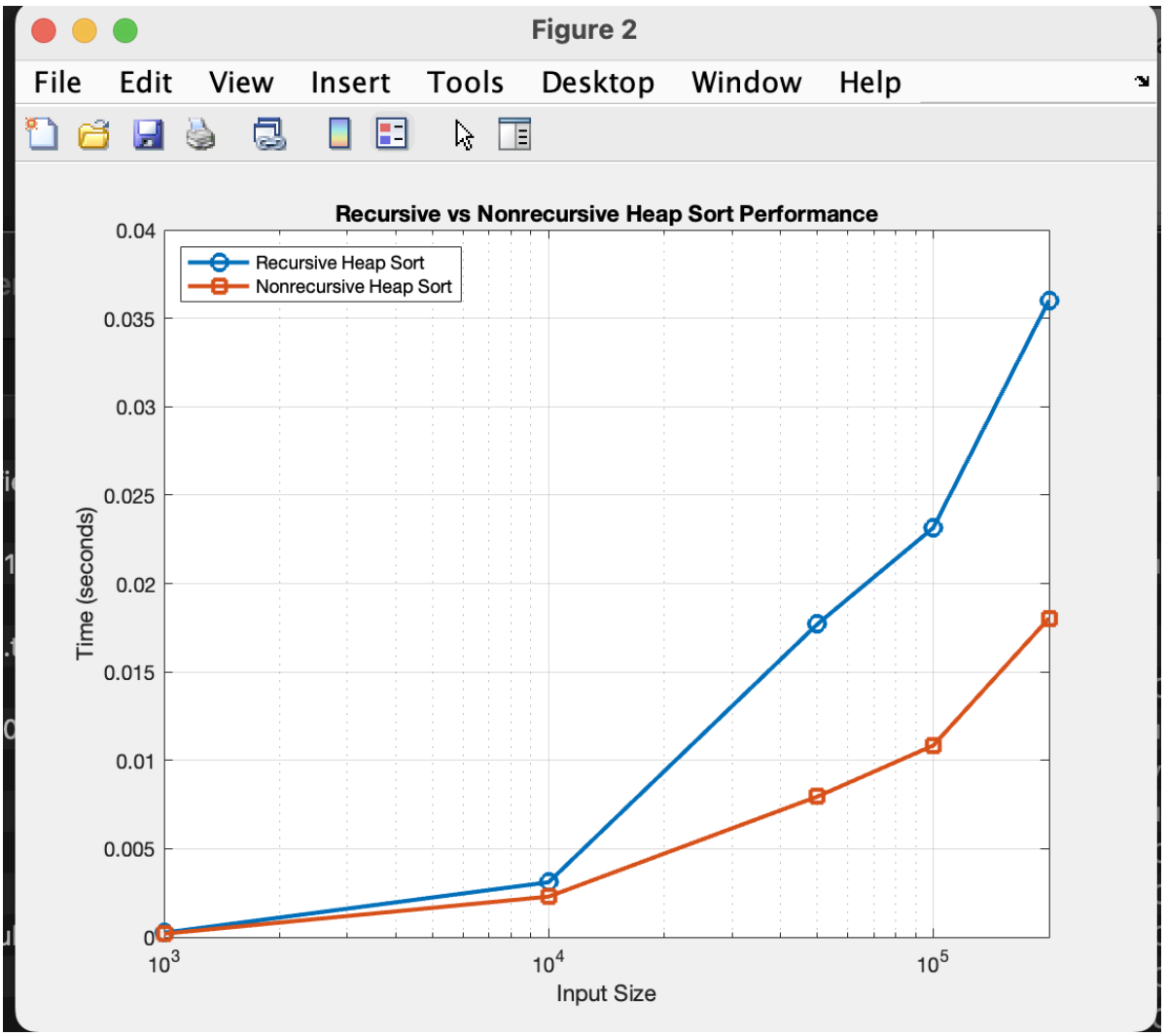
Heap Sort is Not adaptive because it keeps building a heap and then proceeds to perform the heapify operation, no matter how sorted the input was.

HW6 2

(a) See code implementation

(b) See code implementation

(c)



Comparison of Recursive and Non-Recursive Heap Sort :

Recursive Heap Sort: utilizes recursion to maintain heap structure, leading to O(n log n) time complexity but O(lg n) extra space due to recursion.Slower for large inputs due to function call overhead.

Non-Recursive Heap Sort: uses iteration instead of recursion, maintaining O(n log n) time complexity but with O(1) space. More efficient for large datasets.

Conclusion: The non-recursive version is generally faster and more memory-efficient.