Questions and/or Exercises to work out and turn in:

Grading Guidelines:

A right answer will get full credit when:

1. It is right (worth 25%)
2. It is right **AND** neatly presented making it easy and pleasant to read. (worth an **extra** 15%)
3. There is an **obvious and clear link** between 1) the information provided in the exercise and in class and 2) the final answer. A clear link is built by properly writing, justifying, and documenting an answer (worth an **extra** 60%).
4. Calculation mistakes will be minimally penalized (2 to 5% of full credit) while errors on units will be more heavily penalized.

You are welcome/encouraged to discuss exercises with other students or the instructor. But, ultimately, **personal** writing is expected.

* USE THIS FILE AS THE STARTING DOCUMENT YOU WILL TURN IN. **DO NOT DELETE ANYTHING FROM THIS FILE:** JUST **INSERT** EACH ANSWER RIGHT AFTER THE RELATED QUESTION.
* IF USING HAND WRITING (STRONGLY DISCOURAGED), **USE THIS FILE** BY CREATING SUFFICIENT SPACE AND WRITE IN YOUR ANSWERS.
* FAILING TO FOLLOW TURN IN DIRECTIONS /GUIDELINES WILL COST **A 30% PENALTY.**

Objectives of this assignment:

* to use and manipulate the concepts presented in this module
* to use and manipulate the definitions of O(g(n)), Ω(g(n)), and Θ(g(n))
* to get familiar with the “order” of usual functions: polynomials, square root, logarithms, exponentiatl...

What you need to do:

Answer the questions and/or solve the exercises described below.

The objective of this assignment is to study a naïve sorting algorithm: NaïveSort. You must prove the correctness of NaïveSort and evaluate its performance.

NaïveSort works by repeatedly swapping adjacent elements that are out of order. Below is its pseudocode of the simplified NaïveSort algorithm.

**NaïveSort**(A)

1 for j = A.length downto 2

2 for i = j-1 downto 1

3 if A[j]< A[i]

4 exchange A[j] with A[i]

Part I (45 points) Correctness

a. Let A denote the output of **NaïveSort** (A). To prove that **NaïveSort** is correct, we need to prove that it terminates and that

A[1] ≤ A[2] … A[n] , **(2.3)**

where n = A.length. In order to show that **NaïveSort** actually sorts, what else do we need to prove?

Answer the next two questions b. and c. to prove inequality (2.3).

There are two main things that need to be proven here. The first thing is that the sorted array A is correctly sorted and then that there are no missing elements, or duplicates for that matter as well.

b. (20 points) State precisely a **loop invariant** for the for loop in **lines 2–4**, and prove that this loop invariant holds. The loop invariant must be useful, i.e., it must help prove Property 2.3. Your proof should use the structure of the loop invariant proof presented in this chapter:

(5 points) **Loop invariant** is: answer here .....

The loop invariant that I choose to represent is within the inner loop of the NaiveSort algorithm of array A. For a fixed j, for any k such that I < k <= j. A[k] is greater than or equal to all the elements in positions from 1 to variable i.

(3 points) **Initialization**: answer here .....

The idea of this step is to set up the variables “i” and “j” and upon initialization there will be no element between these two. So before the loop starts for a given “j”, i = j – 1. So for the initial case we will have k = j. This means that A[j] is already equal to itself, which should satisfy the proof of initializing.

(8 points) **Maintenance**: answer here .....

The maintenance step includes decrementing the variable “i”, which in turn will place A[j] at the correct position relative to the next element. So, when we have A[j] < A[i], we will swap A[j] with the element located at A[i]. To prove that we now “maintain” the loop invariant after the swap, for any k where variable “i” < k <= j. This will also include the swapped A[j], A[k] will still be greater than or equal to all elements in the final positions already from 1 to variable “i” – 1.

(4 points) **Termination**: answer here .....

The loop will break out when variable “i” becomes equal to 1. A[j] has been compared and then swapped, if needed, for every element before it. This will show A[j] is in its final, correct position(in-place position) in the sorted array version. This proves that A[j] >= to all other elements with regard from 1 to

j – 1. This will show a maintained sorted order up to j.

c. Using the termination condition of the loop invariant proved in part (b), state a **loop invariant** for the for loop in **lines 1–4** that will help you to prove inequality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter:

(6 points) **Loop invariant** is: answer here .....

The outer loop will consist of all the smallest elements within the original array and will be sorted in non-decreasing order. This loop invariant, at the start of each iteration for some j, where 2 <= j <= n for every element in the subarray A[j..n], given it is already in it’s final position (in-place). For any two indices i and k such that j <= k <= i <=n and it holds for A[k] <= A[i].

(4 points) **Initialization**: answer here .....

Before the loop begins, when j = n, the subarray A[j..n] only has one element and can be considered sorted.

(10 points) **Maintenance**: answer here .....

Now assuming the loop invariant will hold before an iteration into the loop for some j. The inner loop operation should hold true based on the above observation. A[j] is placed in its correctly sorted position relative to the elements in the array A[j..n]. Then by swapping A[j] with elements before it if A[j] < A[i] for any i that is greater than j (i > j). This then shows that A[j] is always less than or equal to any element A[k] for j < k < n. After the decrement of j the sorted subarray from A[j..n] to A[j-1..n].

(5 points) **Termination**: answer here .....

Simply when the variable “j” is equal to 2 (j = 2) is when the loop will terminate and then array A will be considered sorted in non-decreasing order (increasing order). This will mean that A[2..n] is relative and sorted to the next positioned element A[3..n] and that A[1..n] is also positioned (in-place) because it is the only element to not be sorted, which would always be at the beginning of the this sorted array A. This demonstrates that NaïveSort successfully sorts the array according to the pseudocode.

Part II (45 points) Running Time of NaïveSort

1. (2 points) What is the input size? answer here .....

The input size will be whatever the value for the variable “n”. This will be equal to the length of array A that is plugged into the NaiveSort algorithm.

1. (3 points) What is the operation that you will count? answer here .....

The total number of comparisons between all elements to determine if a swap will be needed.

1. (40 points) Let T(n) be the running time of NaïveSort. Derive the asymptotic bound for T(n).

answer here ..... We expect that you use a table like the previous homework assignment to derive the time complexity and follow the same steps. If you have any doubt, ask.

|  |  |  |
| --- | --- | --- |
| j(Outer Loop Index) | i (Range for inner loop) | Comparisons per j |
| **n** | **n-1 to 1** | **n-1** |
| **n-1** | **n-2 to 1** | **n-2** |
| **..** | **..** | **..** |
| **3** | **2 to 1** | **2** |
| **2** | **1 to 1** | **1** |

The total number of comparisons will equal the running time T(n). From the table above we can derive the formula:

T(n) = (n – 1) + (n-2) +..+ 2 + 1

T(n) = (n(n-1))/2

T(n) = O(n2)

This can be seen as the general running time for most all algorithms that use comparisons to sort an array.

Part III (10 points) Space Complexity

1. (2 points) Is NaïveSort an *in place* algorithm? answer here .....

After each element has been moved or replaced it will not move again. This means that this is an in-place algorithm.

1. (8 points) What is the space complexity of NaïveSort if we do not count the space used by the input.

answer here .....

If the space that is used by the input array is not counted as more space since it is not growing. The space complexity comes then mostly from for the variables for iteration(i and j) and then space for the swaps when needed. These do not scale more in relation to the input size so the space complexity is O(1), which means there is considered to be a constant amount of additional space.

**Appendix**: Grading: What is an OBVIOUS and CLEAR LINK?

Here is an example to explain what an **obvious and clear link** is and how we grade your work.

Consider the following problem:

"(100 points) John travels from Auburn to Atlanta in his car at a speed of 60 mph. Leaving at 8am, at what time will John reach Atlanta".

Here are the answers of three students and their scores:

* **Student 1** answers: "9:48am". Student 1 will get 25 points.
* **Student 2**answers : "John will reach Atlanta at 9:48am". Student 2 will get 25+15 = 40 points
* **Student 3** answers: "The time t to travel a distance d at speed v is equal to d/v = d/60mph. The problem does not provide the distance d from Auburn to Atlanta. Based on GoogleMaps, the distance from Auburn to Atlanta is approximately 108 miles (**document is attached**).



Therefore, the time t = 108 miles/60mph \* 60 minutes/hour= 108 minutes. Since John left at 8am, he will then reach Atlanta at 8am + 108 minutes = 8 am + 60 minutes + 48 minutes = 9:48".

**Student 3** will get 25 + 15 + 60 = 100 points

Do you see the **direct** **link** going from the data provided in the question to the final answer, using general knowledge/formula and documents?.... Can you now solve the following problem and get 100 points?

"(100 points) Alice travels from Auburn to Atlanta in her car at a speed of 60 mph. Leaving at 8am, at what time will Alice reach Atlanta assuming that she had a flat tire that delayed her 30 minutes".