This assignment is **due on March 25** and should be submitted on Gradescope. All submitted work must be *done individually* without consulting someone else's solutions in accordance with the University's "Academic Dishonesty and Plagiarism" policies.

As a first step go to the last page and read the section: Advice on how to do the assignment.

Problem 1. (10 points)
Using O-notation, upperbound the running time of the following algorithm, where A is an array containing n integers.

```
1: function ALGORITHM(A)
2:     $result \leftarrow 1$
3:     for $i \leftarrow 0; i < n;$ i++ do
4:         for $j \leftarrow 0; j < 25;$ j++ do
5:             $result \leftarrow result * A[i]$
6:     return $result$
```

Solution 1. Line 2, 4, 5, and 6 take $O(1)$ time (the for loop is always executed exactly 25 times for each i). The loop on line 3 loops over n elements, so line 3-5 take $O(n)$ time. Thus the total running time is $O(n)$.

Problem 2. (25 points)
We want to design a list-like data structure that supports the following operations on integers.

- ADDTOFRONT(e): Adds a new element e at the front of the list.
- ADDTOBACK(e): Adds a new element e at the end of the list.
- REMOVEFROMFRONT(): Removes the first element from the list and returns it.
- REMOVEFROMBACK(): Removes the last element from the list and returns it.
- SETELEMENT(i, e): Sets the element of the i-th element of the list to e.

Each operation should run in $O(1)$ time. You can assume that we know that the list will never contain more than k elements (k is **not** a constant and should **not** show up in your running time).

Example execution:

ADDTOBACK(0)	[0]
ADDTOFRONT(-1)	[-1,0]
ADDTOFRONT(2)	[2,-1,0]
SETELEMENT(1,6)	[2,6,0]
REMOVEFROMBACK()	[2,6], returns 0
REMOVEFROMFRONT()	[6], returns 2

a) Design a data structure that supports the above operations in the required time.

b) Briefly argue the correctness of your data structure and operations.

c) Analyse the running time of your operations.

Solution 2.

a) Since we need to support setting the element of arbitrary positions/indices in constant time, we'll use an array A. As the list will have length at most k, we'll use an array of length k. To allow us to insert both ends of the array, we'll use a cyclic scheme similar to what we saw for queues. We keep track of two variables *start* and *size*, both are initially 0.

When we insert or remove at the end of the list, we only need to update *size*. If we insert at the front, we need to update both *start* and *size*: inserting at the front requires us to insert before the current first element, which we can do by moving *start* one index (modulo k) and inserting the element at the new starting index of the list. We update *size* to ensure we know where the list ends and how many elements we're storing. Setting an element now becomes a question of computing the correct index in the array and setting the corresponding element.

```
1: function addToFront($e$)
2:     $start \leftarrow start - 1 \mod k$
3:     $A[start] \leftarrow e$
4:     $size \leftarrow size + 1$
```

```
1: function addToBack($e$)
2:     $A[start + size \mod k] \leftarrow e$
3:     $size \leftarrow size + 1$
```

```
1: function removeFromFront()
2:     if $size = 0$ then
3:         return "error"
4:     $e \leftarrow A[start]$
5:     $start \leftarrow start + 1 \mod k$
6:     $size \leftarrow size - 1$
7:     return $e$
```

```
1: function removeFromBack()
2:     if $size = 0$ then
3:         return "error"
4:     $e \leftarrow A[start + size - 1 \mod k]$
5:     $size \leftarrow size - 1$
6:     return $e$
```

```
1: function setElement($i, e$)
2:     if $i < 0$ or $i \geq size$ then
3:         return "error"
4:     $A[start + i \mod k] \leftarrow e$
```

b) To show correctness, we first show that *start* always contains the starting index in A of our list and that *size* indeed correctly maintains the size of the list. Together these imply that $start + size$ is the index immediately after the end of the list.

We note that we need to update start only if we add or remove an element from the start of the list. Both of these operations indeed move *start* forward or backward one position as appropriate and thus *start* is maintained correctly. Since we increment *size* when we add an element and decrement it when we remove one, *size* is indeed maintained correctly.

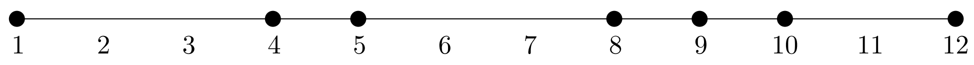
Now that we know that *start* and *size* are correct, we can see that we know where the start and end of the list are at all time. Hence, by inserting and removing the elements at those positions (or just before them), the first four operations are correct. Similarly, the i-th element can be found at index $start + i \mod k$, so as long as i is less than *size* we are accessing a valid element in our list and thus our operation is correct.

c) All operations perform a constant number of elementary operations (assignments, comparisons, simple math, throwing errors) and hence the running time of each operation is $O(1)$ time.

Problem 3. (25 points)
We are given $n \geq 2$ wireless sensors modelled as points on a 1D line and every point has a distinct location modelled as a positive x-coordinate. These sensors are given as a *sorted* array A. Each wireless sensor has the same broadcast radius r. Two wireless sensors can send messages to each other if their locations are at most distance r apart.

Your task is to design an algorithm that returns all pairs of sensors that can communicate with each other, possibly by having their messages forwarded via some of the intermediate sensors. For full marks your algorithm needs to run in $O(n^2)$ time.

Example:



$A : [1, 4, 5, 8, 9, 10, 12], r = 2$: return $\{\{1, 2\}, \{3, 4\}, \{3, 5\}, \{3, 6\}, \{4, 5\}, \{4, 6\}, \{5, 6\}\}$. Note that $A[3]$ and $A[6]$ communicate via $A[5]$.

a) Design an algorithm that solves the problem.

b) Briefly argue the correctness of your algorithm.

c) Analyse the running time of your algorithm.

Solution 3.

a) We first observe that there are a quadratic number of potential pairs that need to be reported, so we can't spend too much time to find a single pair. We also observe that $A[i]$ and $A[j]$ ($i < j$) can communicate if and only if all sensors $A[k]$ and $A[l]$ ($i \leq k < l \leq j$) can communicate with each other. In particular this means that every pair of consecutive sensors in $[i : j]$ can send messages to each other. We'll argue that this observation is correct later.

So we'll keep track of two indices *start* and *current* indicating the range such that every pair of consecutive sensors in $A[start]$ to $A[current]$ can send messages to each other. We repeatedly check whether $A[current]$ can communicate with $A[current + 1]$ and if so we report all pairs with the left sensor in $[start : current]$ and the right sensor being $current + 1$. If $A[current]$ can't communicate with $A[current + 1]$, we update both *start* and *current* to be $current + 1$ and start a new block of sensors where every sensor can communicate with every other sensor.

```
function REPORTCOMMUNICATION($A$, $r$)
    $result \leftarrow \emptyset$
    $start \leftarrow 0$
    $current \leftarrow 0$
    while $current + 1 < n$ do
        if $A[current + 1] - A[current] \leq r$ then
            for $i \leftarrow start; i \leq current;$ i++ do
                $result \leftarrow result \cup \{i, current + 1\}$
            $current \leftarrow current + 1$
        else
            $start \leftarrow current + 1$
            $current \leftarrow current + 1$
    return $result$
```

b) We start by proving our observation: two sensors $A[i]$ and $A[j]$ can communicate if and only if every pair of consecutive sensors between $A[i]$ and $A[j]$ can communicate. It is clear that if every pair of consecutive sensors between $A[i]$ and $A[j]$ can communicate, then $A[i]$ and $A[j]$ can communicate by repeatedly forwarding the message to the next sensor in the sequence.

   To prove the statement in the other direction, we prove the contrapositive: if there is a pair of consecutive sensors between $A[i]$ and $A[j]$ that can't communicate, then $A[i]$ and $A[j]$ can't communicate. Let $A[k]$ and $A[l]$ be sensors in $i \leq k < l \leq j$ such that $A[k]$ and $A[l]$ are more than r apart. Since A is sorted, this implies that no sensor to the left of $A[k]$ can communicate with any sensor to the right of $A[k]$, since their distance is strictly larger than that between $A[k]$ and $A[l]$. Analogously, no sensor to the right of $A[l]$ can communicate with any sensor to the left of $A[l]$. In particular this means that $A[i]$ and $A[j]$ can't communicate with each other.

   Hence, it suffices to check consecutive pairs and report all combinations of sensors until we find a pair that can't communicate with each other (at which point, we start a new block of sensors that can communicate with each other).

c) Line 2-4, 6, and 8-13 take $O(1)$ time, if we pass *result* by reference instead of copying it over explicitly on line 13 (we can also copy it in $O(n^2)$ time, if preferred). For the set, we can simply use a linked list and add the new pair at either end in $O(1)$ time. The loop on line 5 is executed n times, as each of the cases increments *current*. The loop on line 7 is executed at most n times per iteration of the while-loop. So the running time of the algorithm is upper bounded by $n \cdot n \cdot O(1) = O(n^2)$ time.

Advice on how to do the assignment

- Assignments should be typed and submitted as pdf (no handwriting).
- Start by typing your student ID at the top of the first page of your submission. Do **not** type your name.
- Submit only your answers to the questions. Do **not** copy the questions.
- When designing an algorithm or data structure, it might help you (and us) if you briefly describe your general idea, and after that you might want to develop and elaborate on details. If we don't see/understand your general idea, we cannot give you points for it.
- Be careful with giving multiple or alternative answers. If you give multiple answers, then we will give you marks only for "your worst answer", as this indicates how well you understood the question.
- Some of the questions are very easy (with the help of the lecture notes or book). You can use the material presented in the lecture or book without proving it. You do not need to write more than necessary (see comment above).
- When giving answers to questions, always prove/explain/motivate your answers.
- When giving an algorithm as an answer, the algorithm does not have to be given as (pseudo-)code.
- If you do give (pseudo-)code, then you still have to explain your code and your ideas in plain English.
- Unless otherwise stated, we always ask about worst-case analysis, worst case running times etc.
- As done in the lecture, and as it is typical for an algorithms course, we are interested in the most efficient algorithms and data structures.
- If you use further resources (books, scientific papers, the internet,...) to formulate your answers, then add references to your sources.
- If you refer to a result in a scientific paper or on the web you need to explain the results to show that you understand the results, and how it was proven.