This document gives a number of hints helping you to solve written assignments for this unit of study. We start by explaining some general strategies for reading the problems and then look at how to approach and structure your solution. These tips are accompanied by an actual assignment problem from a previous year. At the end, there are some notes on how the assignments are marked.

Reading the problem. When reading a question it's good to ask yourself the following questions, as they'll help you determine the most important information contained in the question.

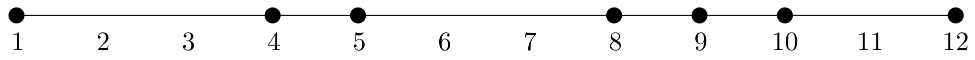
1. What are the parameters? For example, n is typically the number of elements in the input, but there might be other parameters.
2. What are we told about the input? For example, are they integers, are they positive or can there be negative numbers, are they sorted, are they distinct, etc.
3. What are we supposed to compute?
4. What are the requirements in terms of running time, space, and/or technique used?

It's usually a good idea to take a look at the provided example to see if what you think you should compute is actually what the question asks you to compute.

Problem 1. 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 the locations of 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 $\{\{4,5\}, \{8,9\}, \{8,10\}, \{8,12\}, \{9,10\}, \{9,12\}, \{10,12\}\}$. Note that $A[3]$ (with x-coordinate 8) and $A[6]$ (with x-coordinate 12) communicate via $A[5]$ (with x-coordinate 10).

(a) Design an algorithm that solves the problem.

(b) Briefly argue the correctness of your algorithm.

(c) Analyse the running time of your algorithm.

Reading Problem 1. For the above example problem, we could answer our four questions as follows:

1. The question has two parameters: n (the number of sensors) and r (the communication radius).

2. We know that $n \geq 2$, that all sensors have distinct locations, and that they are given in sorted order in our input array A. We also know that r is identical for all sensors.

3. We need to compute all pairs of sensors that can communicate with each other and the message is allowed to be forwarded via intermediate nodes. We are supposed to output the locations of the sensors involved in each pair.

4. Our algorithm is supposed to run in $O(n^2)$ time (slower algorithms can still receive partial marks). There are no space requirements, though you can't access more than quadratic space in quadratic time. There is no restriction on the technique or data structure we're supposed to use. We also observe that r doesn't occur in the running time, so we should ensure that our running time doesn't depend on it (do **not** assume that r is a constant).

Coming up with a solution. It's typically a good idea to come up with a few small example instances of the problem that you can solve by hand. By doing so, you might observe some useful properties of the problem that help you to design a solution for it. For the example problem, we could for example do the following:

$A = [1, 3, 5, 7, 9, 11, 13]$, $r = 2$: We should report every pair of sensors, since they can all communicate with each other by forwarding the message using all intermediate sensors.

$A = [1, 3, 5, 7, 9, 11, 13]$, $r = 1$: We should report nothing, since no pairs are within range of each other.

$A = [1, 2, 5, 7, 9, 12, 13]$, $r = 2$: We should report $\{1, 2\}$, $\{5, 7\}$, $\{5, 9\}$, $\{7, 9\}$, and $\{12, 13\}$.

$A = [1, 2, 5, 7, 9, 12, 13]$, $r = 1$: We should report $\{1, 2\}$ and $\{12, 13\}$.

From the above we might observe that when we report $\{A[i], A[j]\}$ ($0 \leq i < j < n$), we also report all $\{A[k], A[l]\}$ ($i \leq k < l \leq j$). That might be a useful property to keep in mind for our algorithm.

Structuring your solution. Algorithm and data structure design problems typically ask you to split your solution in three parts: (a) the description of the approach, (b) an argument explaining why the approach is correct, and (c) a running time analysis of the approach. For data structure problems, (c) might ask you to analyze both the time and space of the approach.

(a) Describing your approach: When describing your approach, it's generally a good idea to explain the high-level idea behind what you're doing before going into detail. For algorithms, this is typically the observation that lead to your idea. For data structures, this is usually the data structure that you're planning to base your approach on along with a description of what this data structure is supposed to store. Following this, you should describe your approach in enough detail that you can argue its correctness and analyze its running time. It's always a good idea to start your description with a plain English explanation. You can use pseudocode to clarify things, but the main ideas should be clear from reading your plain English explanation. Using pseudocode is always optional, so you're never forced to add this. When describing your data structure, having a separate paragraph for each of the required operations typically gives your description a clear structure, making it easier to read (this also applies to the correctness argument and running time analysis).

(b) Arguing correctness: When arguing the correctness of your approach, be careful that you focus on **why** your approach is correct, instead of repeating only what it does. Another common pitfall is to try to argue correctness by showing that your approach works on a limited set of examples. Since there are an infinite number of possible instances, no finite set of examples can be used to argue that your approach works for every possible instance. Instead, you can try one of the following techniques (though arguing correctness without them can also be correct):

- Proof by contradiction: assume that what you're trying to prove isn't true and show that this leads to a contradiction.
- Proof by contraposition: if you're trying to prove that some predicate p implies some predicate q, it might be worth seeing if proving that the negation of q implies the negation of p is easier to argue (i.e., instead of proving $p \implies q$ you prove $\neg q \implies \neg p$).
- Using an invariant: the invariant describes some condition that you want to be true throughout the execution. You need to show that the invariant holds initially (i.e., before the execution starts) and that your approach maintains it. This is typically used for data structures, where you show that if the invariant holds before an operation is executed, the operation has the desired result **and** the invariant still holds after the operation is executed.
- Using an exchange argument. This is mostly useful when arguing that your algorithm finds an optimal solution, for example in greedy algorithms.
- Using a lower bound argument. This is mostly useful when arguing that your algorithm finds an optimal solution, for example in greedy algorithms.
- Proof by induction. This is useful when your algorithm builds a solution using recursion, for example in divide and conquer algorithms.

If you used an observation to design your algorithm, arguing that this observation is correct is also a good idea.

(c) Analyzing the running time: This is usually considered the easiest part by students and the main things to keep in mind are to analyze everything your approach does (i.e., don't skip the parts that don't influence the running time, as we need to see that you understand that they don't) and make sure that you quote the correct running times of the data structure operations, if you use any.

For our example problem from earlier, the solution could looks something like this:

Solution 1.

(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's 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 8-12 take $O(1)$ time. Line 13 can be done in $O(n^2)$ time by copying over *result* explicitly (there are at most $O(n^2)$ pairs to report and outputting a single one takes $O(1)$ time), but line can also be done in $O(1)$ time if we pass *result* by reference instead[1]. 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.

On marking. When marking your solutions, we typically consider the following things: correctness, efficiency, clarity of your description, clarity and correctness of your correctness argument, clarity and correctness of your running time analysis.

If you can't come up with an algorithm that satisfies all requirements, *it's usually better to come up with a slower correct algorithm than a fast incorrect algorithm*. After all, when you're asked to solve a problem, we're interested in correct solutions and giving an incorrect solution very fast is less useful. Note that depending on how much slower than required your approach is, different penalties apply, depending on how easy the problem is to solve in the time you use. For example, if your data structure is supposed to return the maximum of all integers stored in it in $O(1)$ time, you'd get very few points for needing $O(n)$ time to do so, as that running time can be achieved by just iterating through all integers. You'd get more points for an $O(\log n)$ time solution, as this is more difficult to obtain and you're closer to the requirements.

It's also a good idea to avoid mixing in concepts from the lecture that aren't related to the question. Adding unrelated concepts gives the impression that you don't understand what is and isn't related and hence is likely to cost you marks. Also avoid giving multiple answers, since the least correct one shows your level of understanding and thus giving multiple answers is highly likely to cost you marks.

Finally, it's also important to be able to describe your approach and arguments in a concise fashion, so this is taken into consideration as well, though only in a very minor form. We typically consider a solution too long when it's double the length (or more) of the example solutions that we use for marking and that will be released to you afterwards. A general rule of thumb is that the example solutions will be around 2 pages long, so you should aim for at most 4 pages for your description, correctness argument, and analysis.

[1]Either option is fine and this part is not strictly required, but we include it for completeness.