# Assignment One – 20%

**Algorithms and Data Structures – COMP3506/7505 – Semester 2, 2025**

**Due: 4pm on Friday September 5 (Week 6)**

**Summary**

The main objective of this assignment is to get your hands dirty with the implementation of fundamental data structures and algorithms to solve computational problems. These data structures will also come in handy for your second assignment, so you should take your time to think about your implementations and try to make them as efficient as possible.

## 1 Scary Warnings

You should read this specification in its entirety. There are some aspects that will catch you out if not. For example, Section 7 has some critical information on expectations for referencing your code, details on some required files you must submit, and a list of penalties that can be applied to your assignment solution. Please be aware.

## 2 Getting Started

Before we get into the nitty gritty, we will discuss the skeleton codebase that will form the basis of your implementations, and provide some rules that must be followed when implementing your solutions.

### 2.1 Codebase

The codebase contains a number of data structures stubs that you should implement, as well as some test programs that allow your code to be tested. Figure 1 shows a snapshot of the project directory tree with the different files categorized.

```
├── src/main/java/uq/comp3506/a1/
│           ├─DNAStructure.java    // Part Three
│           ├─Problems.java        // Part Two
│           └─ structures          // Part One
│               ├── BitVector.java
│               ├── DoublyLinkedList.java
│               ├── DynamicArray.java
│               └── ListInterface.java
└── test  // Not assessed, but you should submit them anyway
    ├── checkstyle.xml
    ├── TestDoublyLinkedList.java
    ├── TestDynamicArray.java
    ├── TestBitVector.java
    ├── TestDNAStructure.java
    └── TestProblems.java
```

**Figure 1** The provided directory tree. Pink represents files that contain implementations but are not executable (in the `src/` directory), and Blue represents executable files (in the `test` directory).

The codebase is hosted on Bitbucket if you would like to clone it directly and/or track any changes: `https://bitbucket.org/JMMackenzie/3506-7505-a1-25/`

## 2.2 Implementation

The following list outlines some important information regarding the skeleton code, and your implementation. If you have any doubts, please ask on Ed discussion.

- The code is written in Java and, in particular, using Java 21. As such, you will need to install a version of JDK21 – we recommend using OpenJDK or Temurin. The EAIT student server, `moss`, has OpenJDK 21 installed by default. We will support students using `moss` for the development and testing of your assignment, but you can use your own system if you wish. We do not care which IDE or build system you use, so long as your code runs on the autograder correctly.
- You are not allowed to use pre-built data structures beyond primitives – this is an algorithms and data structures course, after all. This means you may not import `java.util.Collections` or use any pre-built structures from `java.util.*` including, but not limited to, `ArrayList`, `LinkedList`, `HashMap`, `HashSet`, and so on. You cannot use `java.util.stream`, `Arrays.sort`, or other similar data structure utilities. Please use your common sense – the goal of this course is for you to learn how these fundamental data structures work under the hood. If you are in doubt, please implement the utility yourself, or check with us.
- You are allowed to use Java primitive types like `int`, `char`, `boolean`, static arrays (like `int[]`, `Object[]`), control structures, classes, methods, constructors, generics, `System.out`, `Math`, and basic exception handling. You may also use `System.arraycopy` if convenient. You can also use boxed primitives like `Integer` or `Boolean` as well as `String` objects.[1]
- You must use explicit import statements. That is, instead of `import java.util.*;` you must use `import java.util.Random`, for example.

## 2.3 Testing

We have provided very basic skeleton test files in the `test/` directory. You can make use of these however you see fit. You may like to use fully fledged testing toolkits like `JUnit`, but the choice is yours. We **will not assess** your test code, but you are encouraged to submit your tests nonetheless. There is a detailed guide on how to execute tests in Section 6.

## 3 Task 1: Data Structures (8 marks)

We'll start off by implementing some fundamental data structures. You should write your own tests. We *will* try to break your code via (hidden) corner cases. You have been warned.

### Doubly Linked List (2 marks)

Your first task is to debug and optimise a *doubly linked list* — your first "pointer-based" data structure. To get started, look at the `DoublyLinkedList.java` file. You will notice that this file contains two classes: the `Node` type – a nested class which stores a *data* payload, as well

[1] Recall that if you want to use generics, you cannot use primitive types.

as a reference to the *next* node – and the DoublyLinkedList class, which tracks the *head* and *tail* of the list, as well as the number of nodes in the list, and is used to implement the list functionality.

Carefully read the methods and constructor to understand the code. Then, write some tests to determine whether the functionality is working as expected. Identify any bugs, and fix them. There may be inefficient implementations provided – you need to identify these, and re-implement (overwrite) them to make them more efficient. You will also notice that there may be some changes or modifications required to the data structures to support the necessary operations – feel free to add member variables or functions, but please do not change the names of the provided functions as these will be used for marking.

## Dynamic Array (3 marks)

Unlike the linked list discussed above, which can store nodes at any abitrary location in memory, we often prefer to have data items stored contiguously (consecutively in memory), allowing us to access an element $x$ at some index $i$ in $\mathcal{O}(1)$ (constant time). One such way to achieve this is through the use of static arrays, but they cannot have new elements added. Enter the dynamic array – called an `ArrayList` in Java – which can grow by appending new elements to the end.

The file `DynamicArray.java` contains another skeleton for you to implement.[2] The internal data storage will be handled by a static array of `Object` types (supporting generics), and will need to grow if space runs out. You should store your data in a member variable called `data`, and you can add any other required member variables to your DynamicArray object. Each function that needs to be supported is provided as a stub. Your implementations should be efficient and correct, and we have provided annotations to describe the expected complexity. In this assignment, we have given you a slightly trickier ADT than the classic *append-only array* discussed in the lectures. In particular, you must support *prepend* operations — that is, allowing an element $x$ to be placed at the *front* of the array — in $\mathcal{O}(1)$ amortized time.

## Bitvector (3 marks)

In some applications, it is useful to track the state of a collection of objects using simple Boolean (`true` or `false`) flags. A naïve way to do this is to simply use a (dynamic) array, storing `bool` or `Boolean` types as the underlying data. While a Boolean variable could, theoretically, be represented by a single bit, they are more commonly represented by a byte (8 bits) or a full machine word (32 or 64 bits), meaning that we waste a lot of space with this approach.

An alternative approach is to use a *bitvector*, which stores an array of $b$-bit integers to represent each item — unset bits (value 0) represent `false`, and set bits (value 1) represent `true`. Clearly, this approach uses up to $64\times$ less space than the naïve approach, as a single $b = 64$ bit integer can track the state of 64 items.

The file `BitVector.java` contains another skeleton for you to implement. To keep things simple, we will only support *statically sized* bitvectors, meaning that the size will be provided to the constructor, and the bitvector will not need to grow or shrink. As such, you should use a simple array of primitive `long` types to track the bits.

[2] We call it a DynamicArray to distinguish from Java's own ArrayList class.

Each function that needs to be supported is provided as a stub. Your implementations should be efficient and correct, and we have provided annotations to describe the expected complexity.

### Reading Binary Numbers

In this course, we use assume the least significant bit is the rightmost bit in a binary number, and that binary numbers are read from right to left. For example, the number 15 is represented as `000...01111`, and the number 17 is represented as `000...10001`.

### Signed Integers

In Java, all integers are signed. This means one bit is reserved to denote whether a number of positive or negative. Java, as such, supports two types of bit-shifting operations:

- `>>` is an arithmetic shift – the sign bit is preserved.
- `>>>` is a logical shift – the sign bit is ignored.

Since we are implementing a bitvector, the sign of a number is meaningless, as we are just using the bits to represent some Boolean flag. So, you should always treat our operations as *logical* shifts, and not as arithmetic shifts – you will want to use the logical right shift. Note that there is no logical left shift, and the `<<` operator will behave as expected.

### Bitvector Operations: Shift

The shift operator handles both left and right shifts, depending on the sign of the dist parameter. If the dist parameter is positive, we do a left shift by dist. A left shift moves all bits in the bitvector left by dist positions, replacing empty positions with 0 bits. For example, the following demonstrates the before (top) and after (bottom) of a left shift by dist $= 2$:

```
1011000100011
1100010001100
```

Notice that the two *most significant* bits have fallen off (the leftmost `10` on the first bitvector). The right shifts work the same way, but we move the bits dist positions to the right (and the least significant bits will fall off).

### Bitvector Operations: Rotate

The rotate operator works exactly the same way as the shift operator, except it ensures that any bits that fall off the end are *rotated* back onto the start of the bitvector. Using the same example as above with dist $= 2$:

```
1011000100011
1100010001110
```

**NOTE:** This is intended to be completed by both COMP3506 and COMP7505 students.

### Bitvector Operations: Popcount and Runcount – COMP7505 Only

The popcount operator returns the number of "on" (`1`) bits in the bitvector. The runcount operator returns the length of the longest run of "on" (`1`) bits in the bitvector. For example, given the following bitvector, the popcount would be 6, and the runcount would be 2:

```
1011000100011
```

## 4 Task 2: Algorithmic Problem Solving (8 Marks)

Next, we are going to work on solving problems algorithmically. These are designed to build your problem solving skills. Some of them may appear tricky at first; you are encouraged to sit down and think about them (a pen and a piece of paper will help). Do not be afraid to get creative, as there may be multiple ways to solve each problem. Each problem will be assessed on three tiers of tests, and you will receive higher marks for faster solutions. See the `Problems.java` file for more details, and test with `TestProblems.java` (you will need to implement your own tests).

### Short Runs (1 mark)

Run length encoding (RLE) is a simple algorithm used to reduce the size of strings with lots of repetitive characters. When repeat characters are found, they are replaced with their run length. Given a `String` $S$, you need to return the run length encoded version. There is one small twist, however: When a run exceeds 9 characters, you should start a new run, as we only need to use single-digits to track our run lengths. For example:

- HELLOOOO → H1E1L2O4
- AAAAAAAA → A8
- VERYSAD → V1E1R1Y1S1A1D1
- AAAAAAAAAAAA → A9A3

### Arithmetic Rules (1 mark)

You are playing a new game called *"Arithmetic Rules!"* The rules of the game are simple: You are given an array of integers $A$ and a natural number $t$ that represents the number of turns you can take. You start with an initial score of 0. On each turn, you may choose *any* element $e$ from $A$, add its value to your score, and then replace $e$ with $e + 1$ in the same position. You must return the *maximum* score that can be achieved within exactly $t$ turns.

### SQRT Happens (1 mark)

You are given a natural number $n$, and a positive real number, $\varepsilon$. You must, as quickly as possible, compute $x = \sqrt{n}$. For simplicity, you may return an $\varepsilon$-approximate value of $x$ – that is, $|n - x^2| \leq \varepsilon$. However, you are *not* allowed to use any mathematics utilities like `Math.sqrt` or `Math.pow`. For the avoidance of doubt, you must only use simple arithmetic operations like `+, -, *, /`. Sorry, but sqrt happens...

- $n = 10, \varepsilon = 0.1 \rightarrow x = 3.16$
- $n = 10, \varepsilon = 0.1 \rightarrow x = 3.17$
- $n = 17, \varepsilon = 0.001 \rightarrow x = 4.1231$

## Space Oddity (1 mark)

You are given an unsorted array $A$ containing $n$ integers in the range $[0, 2^{32} - 1]$. You need to return the largest integer that appears in $A$ an odd number of times, or $-1$ if no such integer exists.

- $A = [1, 5, 2, 4, 6, 5, 1, 5, 5, 2, 5] \to 6$
- $A = [1, 1, 5, 5, 2, 3, 2, 3] \to -1$
- $A = [9, 9, 1, 5, 1, 9, 1, 9] \to 5$

Hint: You should use one of your data structures from part one to help you solve this problem efficiently!

## Frea-$k$-y Numbers (2 marks)

A natural number is $k$-freaky if it can be represented as the sum of unique non-negative powers of $k$. For example:

- 17 is 4-freaky because $4^0 + 4^2 = 17$;
- 128 is 2-freaky because $2^7 = 128$;
- 11 is 10-freaky because $10^0 + 10^1 = 11$.

Given integers $m$, $n$, and $k$, you must return the *number of* $k$-freaky numbers in the range $[m, n]$.

## Life is Sweet (2 marks)

You are given a rectangular chocolate bar (wow, thanks) that has $n \times m$ single squares. You only want to eat $k$ squares, so you might need to break the bar:

- You can break any rectangular piece of the bar into two pieces;
- You can only break the bar between existing squares;
- The cost of breaking is equal to the square of the length of the break.
- You can eat exactly $k$ squares of chocolate if, after all breaks, there is a set of pieces that can be combined to give exactly $k$ squares.
- The remaining block's shape (the leftover $n \times m - k$ squares) does not matter.

Given $m, n$, and $k$, all of which are integers, you must find the minimum total cost of breaking the chocolate.

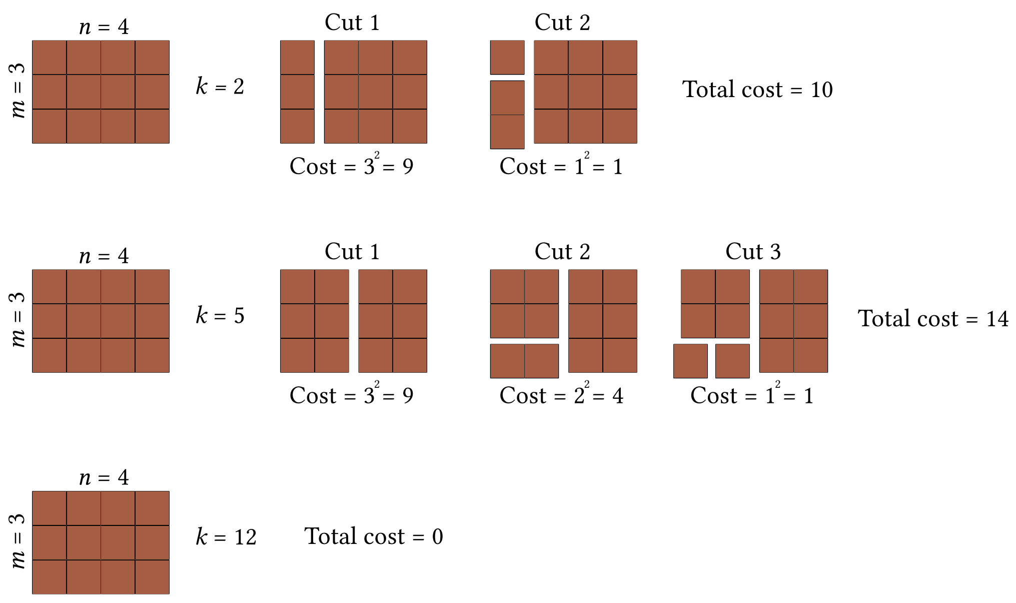

**Figure 2** Three examples of chocolate bar breaking.

## 5 Task 3: Data Structure Design (4 marks)

You are an algorithms specialist working at SIGBUS, a biology company that specialises in futuristic applications of genomic sequencing. As part of a new project, SIGBUS are working on a sequence analysis tool. The lead Bioinformatician, Barry Malloc, has provided you with the following overview of the data and the system requirements. Your job is to design and implement an appropriate data structure – and related algorithms – to match Barry's requirements. Barry has kindly placed the required time bounds in the required methods — you should carefully consider these when designing your data structure.

### DNA Data

DNA data is represented as a string $S$ of length $|S|$ over an alphabet $\Sigma = \{\texttt{A, C, G, T}\}$. Each character represents a different base (*Adenine*, *Cytosine*, *Guanine*, and *Thymine*). For example, a sequence $S$ with $|S| = 32$ might look like

$S =$ `GTCGTGAAGTCGGTTCCTTCAATGGTTAAACC`.

### Required Functionality

We will be working on a *stream* of DNA data as it is read from a high-throughput sequencing device, one character at a time. We need to keep track of the most recent window of characters, with the size of the window denoted $w$, with $w$ provided to the constructor of your data structure. As such, your data structure must be dynamic: it should be able to efficiently remove the least recent character (the character leaving the window), and insert the most recent character (the one entering the window). At any instant in time, we may query your data structure to find out important information about the current state of the window. The full list of operations that must be supported is provided here:

- `isFull()` – returns `true` if the structure is holding a full window ($w$ characters), `false` otherwise.
- `slide(c)` – adds `c` to the end of the window, and, if the window is full, removes the first (least recent) character;
- `count(c)` – returns the number of times character $c$ occurs in the current window;
- `countRepeats(k)` – returns the number of unique sequences of length $k$ that are repeated inside the current window;
- `hasPalindrome(k)` – returns `true` if the window contains a palindrome of length $k$, `false` otherwise;
- `stringify()` – converts the elements in the window/container to a string, read left to right.

You may want or need to use one or more of the structures implemented in part one to support your final data structure implementation, but the choice of design is yours. Implement your solution in `DNAStructure.java`, and your tests in `TestDNAStructure.java`. Further information on the more exotic query types is provided below.

#### countRepeats

Suppose you have a current window $w$, as follows, and $k = 3$. Then, the `countRepeats` query would return 2, because there are two unique $k$-mers that are repeated: `TAT` and `ATA`.

$w =$ `CCTATAGGTATACATA`.

Note that it doesn't matter how many times each $k$-mer is repeated, we just want to know how many unique $k$-mers repeat inside $w$.

### hasPalindrome

A *palindrome* is a string that reads the same way forwards as it does in reverse. For example, `did`, `anna`, `kayak`, and `tacocat` are all palindromes.

In genomics, however, things are not so simple: A DNA sequence is said to be a palindrome if it is equal to its *reverse complement*. The pair `A` and `T` is said to be *complementary*, as is the pair `C` and `G`. The sequence `CATG` is a palindrome, because the complement is `GTAC`; if we reverse `GTAC`, we get `CATG`! Other examples of palindromes include `ACCTAGGT` and `TTAA`.

## 6 Writing and Running Tests

In this course, we do not require to use any specific development environment or toolchain (other than Java 21, as outlined earlier in the document). This means, the way in which you run your code/tests will depend entirely on the development environment you use. The following example assumes a Unix, CLI-based environment. It also works on Windows, with the exception that the `ls` command doesn't run (use `dir` instead). There are guides for IntelliJ on the LMS that may come in handy. Please don't hesitate to ask for help on Ed if you would like to use a specific IDE – I'm sure someone will be able to give you some tips if need be.

We assume you are in the root directory of this project, as follows:

```
barry@temple:~/a1$ ls
src statement.txt test
```

Firstly, I am going to create a new directory called `out/` that will hold the compiled class files:

```
barry@temple:~/a1$ mkdir out
```

Next, I will compile the functionality in the `src/` directory, storing the compiled results in `out/`. Note that I will compile all .java files in both the `src/main/java/uq/comp3506/a1/` and in the `src/main/java/uq/comp3506/a1/structures/` directory:

```
barry@temple:~/a1$ javac -d out src/main/java/uq/comp3506/a1/*.java
    src/main/java/uq/comp3506/a1/structures/*.java
```

Next, I will compile a single test file:

```
barry@temple:~/a1$ javac -cp out -d out test/TestDoublyLinkedList.java
```

Finally, we can run the test as follows:

```
barry@temple:~/a1$ java -cp out TestDoublyLinkedList
Testing DoublyLinkedList Class...
Success!
```

Note that the JVM will not evaluate assertions by default. You will need to pass the `-ea` flag to your program call to ensure assertions are checked:

```
barry@temple:~/a1$ java -ea -cp out TestDoublyLinkedList
```

### Style Checking

If you want to check the style of your code, you can download the appropriate `checkstyle` jar and run it as follows:

```
barry@temple:~/a1$ java -jar checkstyle.jar -c test/checkstyle.xml src
Starting audit...
[WARN] src/main/java/uq/comp3506/a1/DNAStructure.java:8:14:
    Abbreviation in name ...[truncated]
Audit done.
```

For the above test, I used `checkstyle-10.26.1-all.jar` which you can grab here: https://github.com/checkstyle/checkstyle/releases/ You can also set up checkstyle as an IDE plugin. Up to you!

## 7 Assessment

This section briefly describes how your assignment will be assessed.

### Mark Allocation

Marks will be provided based on an extensive (hidden) set of unit tests. These tests will do their best to break your data structure in terms of time, space, and/or correctness, so you need to pay careful attention to the efficiency and the validity of your code. Each test passed will carry some weight, and your autograder score will be computed based on the outcome of the test suite. If you did not rigorously test your programs/code, you should go back and do so – results of hidden tests will be made public after the assignment due date.

#### Penalties

Penalties will be computed and removed from your overall final mark as follows:

- If your code does not adhere to the style guide provided on the Blackboard (`Course Resources > Java Resources`) – the same style file as provided in the `test/` directory – you will lose 2 marks. Note, however, that we have some leniency built in, and the autograder will inform you of whether you are being penalised or not;
- If your submission does not include the `statement.txt` file, or if the statement is empty, you will lose 20 marks, no exceptions – see the instructions regarding the statement below.
- If you make more than 50 submissions to Gradescope, you will lose 5 marks;
- If you make more than 80 submissions to Gradescope, you will lose a further 5 marks;
- If you make more than 100 submissions to Gradescope, you will lose a further 5 marks.

### Plagiarism and Generative AI

If you want to actually learn something in this course, our recommendation is that you avoid using Generative AI tools: You need to think about what you are doing, and why, in order to put the theory (what we talk about in the lectures and tutorials) into practical knowledge that you can use, and this is often what makes things "click" when learning. Mindlessly lifting code from an AI engine won't teach you how to solve algorithms problems, and if you're not caught here, you'll be caught soon enough by prospective employers.

If you are still tempted, note that we will be running your assignments through sophisticated software similarity checking systems against a number of samples including including your classmates and our own solutions (including a number that have been developed with AI assistance). If we believe you may have used AI extensively in your solution, you may be called in for an interview to walk through your code. Note also that the final exam may contain questions or scenarios derived from those presented in the assignment work, so cheating could weaken your chances of successfully passing the exam.

#### AI Statement

As part of your submission, you must edit the `statement.txt` file to provide attribution to any sources or tools used to help you with your assignment, including any prompts provided to AI tooling. If you did not use any such tooling, you can make a statement outlining that fact. Failing to submit this file, or submitting an empty file, will result in a 20 mark penalty.

**Please refer to the referencing guide for assessments** located in the `Assessment` directory on Blackboard for more information.

### Interviews

We will be conducting assignment interviews, where a subset of students will be invited to walk through their assignment submission with a member of staff. Your final assignment score will be weighted according to your demonstrated understanding of your submission, where the weight will be assigned in the range $[0.0, 1.0]$. Note that students who are not selected for interviews will have this weight set to 1.0 by default. For example:

- Student A scores 18 marks. They are not interviewed. Their final score remains at 18 marks.
- Student B scores 12 marks. They are interviewed, and they are judged to have a full understanding of their submission giving them a weighting of 1.0, resulting in 12 marks.
- Student C scores 19 marks. They are interviewed, and they are judged to have a mostly complete understanding of their submission, but are lacking understanding of some parts of their algorithms, giving them a weighting of 0.75. This means their final mark is $0.75 \times 19 = 14.25$.

Students may be selected for an interview according to their reported AI usage (or lack thereof) as noted in their statement file, their assignment similarity with other students and/or AI generated solutions, their mark as compared to their past performance, or totally randomly.

## Submission Information

You need to submit your solution to *Gradescope* under the *Assignment 1: Autograder* link in your dashboard. Please use the appropriate link as there is a separate submission for 3506 and 7505 students. Once you submit your solution, a series of tests will be conducted and the results of the public tests will be provided to you. However, the assessment will also include a number of additional *hidden* tests, so you should make sure you test your solutions extensively.

## Submission Structure

The easiest way to submit your solution is to submit a `.zip` file. The autograder expects a specific directory structure for your solution, and the tests will fail if you do not use this structure. In particular, you should use the same structure as the skeleton codebase that was provided. This means submitting a zip file which, when unpacked, contains a `src` and `test` directory, with the `statement.txt` file in the project root directory. A common mistake is having the project packaged in a nested directory – this will not work.

## 8 Resources

We provide a number of useful git and/or unix resources to help you get started, as well as some information about the IntelliJ IDE, and a Java style guide. Please go onto the Blackboard LMS and see the `Course Resources` directory for more information. We will also hold assignment help sessions. Please refer to the Blackboard for more information.

## 9 Specification Changelog

Note that the codebase is released on Bitbucket, so changes to the code skeleton can be found there. Below is just concerned with the specification itself.

- V1.0: Initial release.
- V1.1: Clarification: Bitvector rotate is for **all** students, not just COMP7505.
- V1.2: Clarification: Fixed wrong example for “Space oddity” and clarified style checking.