Assignment One – 15%

Algorithms and Data Structures – COMP3506/7505 – Semester 2, 2024

Due: 3pm on Friday August 23 (week 5)

Summary

The main objective of this assignment is to get your hands dirty with some simple data structures and algorithms to solve basic 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 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.

1.1 Codebase

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

```
├── test_structures.py
├── test_warmup.py
├── test_kmers.py
├── structures
│   ├── dynamic_array.py
│   ├── linked_list.py
│   └── bit_vector.py
├── warmup
│   └── warmup.py
└─── malloclabs
    ├── kmer_structure.py
    ├── analysis.txt
    └── generate_dna.py
```

Figure 1 The directory tree is organized by task. Blue represents directories, Teal represents files that contain implementations (but are not executable), and Orange represents executable files.

1.2 Implementation Rules

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 Python and, in particular, should be executed with Python 3 or higher. The EAIT student server, `moss`, has Python 3.11.* installed by default. We recommend using `moss` for the development and testing of your assignment, but you can use your own system if you wish.

- You are not allowed to use built-in methods or data structures – this is an algorithms and data structures course, after all. If you want to use a `dict` (aka `{}`), you will need to implement that yourself. Lists can be used as "dumb arrays" by manually allocating space like `myArray = [None] * 10` but you may not use built-ins like `append, clear, count, copy, extend, index, insert, pop, remove, reverse, sort, min, max,` and so on. List slicing is also prohibited, as are functions like `sorted, len, reversed, zip`. Be sensible – if you need the functionality provided by these methods, you may implement them yourself. Similarly, don't use any other collections or structures such as `set` or `tuple` (for example; `mytup = ("abc", 123)`).
- You are not allowed to use libraries such as `numpy`, `pandas`, `scipy`, `collections`, and so on.
- Exceptions: The only additional libraries you can use are `random` and `math`. You are allowed to use `range` and `enumerate` to handle looping. You can also use for item in my_list looping over simple lists.

2 Task 1: Data Structures (5 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.

Task 1.1: Doubly Linked List (1.5 Marks)

Your first task is to implement a *doubly linked list* — your first "pointer-based" data structure. To get started, look at the `linked_list.py` file. You will notice that this file contains two classes: the Node type, which stores a *data* payload, as well as a reference to the *next* node; and the DoublyLinkedList type which tracks the *head* and *tail* of the list, as well as the number of nodes in the list.

A basic set of functions that you need to support are provided as function templates, and you will need to implement them. 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.

You will need to implement your own tests and run them using:
`python3 test_structures.py --linkedlist`.

Task 1.2: Dynamic Array (2 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 constant time. One such way to achieve this is through the use of a dynamic array.

The file `dynamic_array.py` contains another skeleton for you to implement. You should store your data in self._data, and you can add any other 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, worst case.

You will need to implement your own tests and run them using:

```
python3 test_structures.py --dynamicarray
```

Task 1.3: Bitvector (1.5 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 types as the underlying data. However, Boolean types are usually represented by a machine word (32 or 64 bits), meaning that we waste a lot of space with this approach.[1]

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 64× less space than the naïve approach, as a single $b = 64$ bit integer can track the state of 64 items.

The file `bit_vector.py` contains another skeleton for you to implement. Note that it uses a composition based design where a DynamicArray object is used to store the underlying data. 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. We describe two of the more exotic operations that should be supported in more detail below.

You will need to implement your own tests and run them using:
`python3 test_structures.py --bitvector`.

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
```

[1] For the interested student: Most statically typed languages would represent a Boolean value as a machine word (that is, 32 or 64 bits) for convenience. Python is a dynamically typed language, and each object actually carries a bunch of metadata around with it, and may be something like 24 *bytes*. However, every `True` or `False` used is merely a reference to one of these two objects (there is only real `True` and one real `False` object instantiated in the runtime). Nonetheless, here, we will pretend that each `Bool` usually costs a machine word. In C, for example, a bool would use 8 bits (one byte), the minimal addressable size.

3 Task 2: Algorithmic Thinking Warm-Up (5 Marks)

Next, we are going to work on some simple *warm up* problems. 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 — see the `warmup.py` file for more details, and test with `test_warmup.py` (you need to implement your own tests).

3.1 The Main Character

You are given a string S. You need to simply return the *first position* of a repeated character (indexed from zero), or -1 if there are no repeats.

- $S =$ `hello` → 3.
- $S =$ `world` → -1.
- $S =$ `algorithmsarefun` → 10.
- $S =$ `ooooohigetitnow` → 1.

Here's the catch: S can be built from an alphabet containing 2^{32} possible characters, represented as integers in the range $[0, 2^{32} - 1]$. Hint: You should use one of your data structures from part one to help you solve this problem efficiently!

3.2 Sum-Thing Odd

You are given an unsorted list L containing n *unique* integers. Let $\min(L)$ and $\max(L)$ represent the smallest and largest integers in L. Your task is to find the *sum* of the *missing odd integers* in the range $[\min(L), \max(L)]$.

- Consider $L = \langle 6, 4, 9 \rangle$: $\min(L) = 4$ and $\max(L) = 9$. Your range of interest is thus $[4, 9]$. The sum of the missing odd integers will be $5 + 7 = 12$.
- Consider $L = \langle 10, 1, 7, 17 \rangle$: $\min(L) = 1$ and $\max(L) = 17$. Your range of interest is $[1, 17]$. The sum of missing odds in this range will be $3 + 5 + 9 + 11 + 13 + 15 = 56$.

3.3 It's cool, k?

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

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

Given n and k, you must return the nth largest k-cool natural number. Two alternative ways to say this:

- Return the nth smallest number that can be created by summing non-negative integer powers of k; or
- If you created all combinations of sums of powers of k and sorted them into ascending order, we want you to return the nth one.

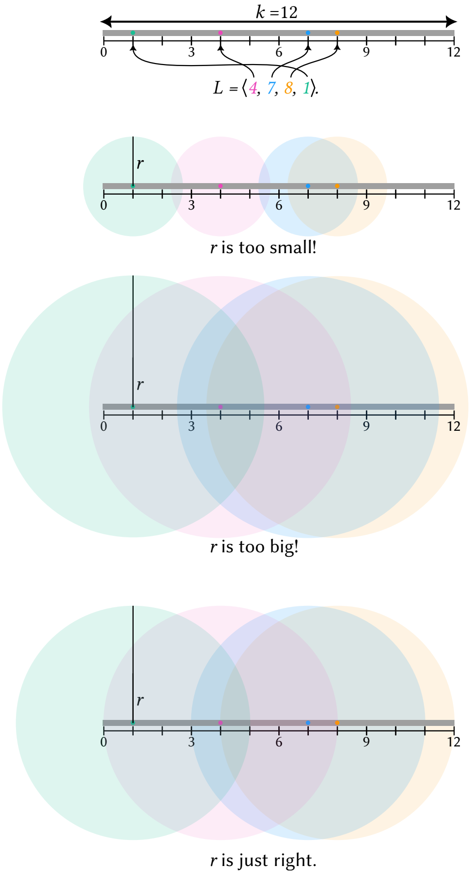
Clearly, the *first* k-cool number is always 1 ($k^0 = 1$) and the *second* k-cool number is always $k^1 = k$. Since we will test some very large numbers (up to $10^{1,000,000}$), you should return the number modulo $10^{16} + 61$.

3.4 Your Number is Up

Alice and Bob are playing a numbers game. The game starts with an unsorted list of integers, L, with $n = |L|$ guaranteed to be even. On each turn, they both remove one number from L. If Alice removes an even number, it is added to her score. If she removes an odd number, her score remains unchanged. The opposite is true for Bob (removing an odd number adds to his score; removing an even number keeps his score unchanged). After n turns, the array is exhausted, and the winner is determined by the player with the largest score — A draw is also possible. Given L, you must return the winner, and their score, assuming both Alice and Bob play optimally. You can assume that Alice always gets the first turn.

3.5 Getting Lit

A straight road of length k is illuminated by n light poles. Each light pole is represented by a natural number i, where i represents the total distance from the start of the road to the given pole. Each light is capable of illuminating a radius r. Given an unsorted list L of poles, you need to return the minimal radius r such that the entire length of the road, k, is illuminated. You may assume that the *width* of the road is infinitely small.[2]



r is too small!

r is too big!

r is just right.

Figure 2 A sketch of *getting lit* with $n = 4$ light poles.

[2] Yes, we know it's not a very useful road, but it makes your assignment easier. See also: `https://w.wiki/Ajs5`

4 Task 3: Problem Solving with Data Structures (5 marks)

You are an algorithms specialist working at `SIGSEGV`$^{\text{TM}}$, a world leader in high performance algorithmic solutions. You have been contracted by a bioinformatics company called `MallocLabs` who require a bespoke system to help them deal with a growing amount of genomic data they need to index. Their 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 function stubs — you should carefully consider these when designing your data structure.

4.1 Data Representation

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`.

Since sequences can be very long, we can break them up into k-mers, all possible substrings of length k. For example, there are 10 individual 23-mers of S:

```
GTCGTGAAGTCGGTTCCTTCAAT
TCGTGAAGTCGGTTCCTTCAATG
CGTGAAGTCGGTTCCTTCAATGG
GTGAAGTCGGTTCCTTCAATGGT
TGAAGTCGGTTCCTTCAATGGTT
GAAGTCGGTTCCTTCAATGGTTA
AAGTCGGTTCCTTCAATGGTTAA
AGTCGGTTCCTTCAATGGTTAAA
GTCGGTTCCTTCAATGGTTAAAC
TCGGTTCCTTCAATGGTTAAACC
```

In general, a sequence of length $|S|$ will contain $|S|-k+1$ k-mers, and there are a total of $|\Sigma|^k$ unique possible k-mers (in our case, $|\Sigma| = 4$). You are provided with a tool `generate_dna.py` that can generate n sequences of length $|S|$ for you to experiment with; it is probably easiest to simply write them out to a file, and use the file as input to your testing program.

4.2 Required Functionality

At run-time, your program will be given two arguments that specify a path to a file containing DNA sequences, as well as the value of k we are interested in working with. For example, you might be given a file containing $50,000$ sequences of length 200, and $k = 31$. We will always use `str` types to represent k-mers. The data structure used to solve the following requirements is up to you, and should be designed based on the functionality requested. You may need to use one or more of the structures implemented in part one for example, but the final choice is yours. Implement in `kmer_structure.py` and test with `test_kmers.py` (you need to implement your own tests).

Storage and Modification: 2 marks

The first set of functions you need to support allow for reading and modifying data. They are specified as follows:

- read: Given a file containing DNA sequences, break them into individual k-mers, and store them in your data structure;
- batch_insert(L): Given a list of k-mers L, insert them into your data structure;
- batch_delete(L): Given a list of k-mers L, delete *all* occurences of them from your data structure.

Note that there may be some duplicate k-mers in your data structure. You must keep track of duplicates and their frequency, as these will be required for answering some query types in the next section.

Queries: 3 marks

Your data structure also needs to support the following query types.

- freqgeq(n): Return a list of *unique* k-mers that occur at least n times;
- count(q): Return the number of times a k-mer q occurs;
- countgeq(q): Return the *total* number of k-mers that are $\geq q$; that is, you need to sum the frequencies of all k-mers lexicographically greater than or equal to q;
- compatible(q): Return the *total* number of k-mers that are *compatible* with q.

We provide some further information on the compatibility query as follows. A given k-mer q is called *compatible* with k-mer b if the last *two* characters in q are the complement of the first *two* characters in b. In genomics, the pair `A` and `T` is complementary, as is the pair `C` and `G`. So, for example, `CCTGATG` is compatible with `ACTTGCG`:

```
q = CCTGATG
         ||
         ACTTGCG
```

Note that we always assume we are matching the *end* of the input query k-mer q with the start of all other k-mers.

Analysis: 2 marks (COMP7505 Students Only)

If you are a COMP7505 student, you must also answer the questions posed in the plain text file called analysis.txt (inside the `malloclabs` directory).COMP3506 students are encouraged to do this too, but they will not be assessed on this component. Please keep your answers succinct, but make sure to include all details that may be relevant. If in doubt, err on the side of *more detail*.

5 Assessment

This section briefly describes how your assignment will be assessed.

5.1 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 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! As the famous poet Ice Cube once said: *check yourself before you wreck yourself.*

The marks (percentages) provided in each task above are indicative of the total score available for each part, but marks may be taken off for poor coding style including lack of commenting, inefficient solutions, and incorrect solutions. Our code quality checks are not as strict as PEP8, but we assume typical best practices are used such as informative variable and function names, commenting, and breaking long lines. While the overall grade/score will be calculated mathematically, an indicative rubric is provided as follows:

- **Excellent:** Passes at least 90% of test cases, failing only sophisticated or tricky tests; well structured and commented code; appropriate design choices; appropriate application of data structures/algorithms for solving Tasks 2/3.
- **Good:** Passes at least 80% of test cases, failing one or two simple tests; well structured and commented code; good design choices with some minor improvements possible; good application of data structures/algorithms for solving Task 2/3 with some minor improvements possible.
- **Satisfactory:** Passes at least 70% of test cases; code is reasonably well structured with some comments; most design choices are reasonable but significant room for improvement; reasonable application of data structures/algorithms for solving Task 2/3, but significant improvements possible.
- **Poor:** Passes less than 70% of test cases; code is difficult to read, not well structured, or lacks comments; design choices do not demonstrate a sound understanding of the desired functionality; little or no suitable application of data structures or algorithms towards solving Task 2/3.

5.2 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.

As part of your submission, you must create a file called `statement.txt`. In that file, you must 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 will yield you zero marks.

6 Submission

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. You may resubmit as often as you like before the deadline, but we are imposing a limit of *ten submissions per 24 hour period*. Please write your own tests!

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. You should also have the `statement.txt` and `analysis.txt` (for COMP7505 students). Submissions without the `statement.txt` will be given zero marks, and the autograder will notify you of this.

7 Resources

We provide a number of useful git and/or unix resources to help you get started. Please go onto the Blackboard LMS and see the Learning Resources > Resources directory for more information.

8 Changelog

- V1.0: Initial release.
- V2.0: Release with code; changed directory structure to match code. Clarified k-cool further. Fixed an unfortunate typo.