COMP 2402 W24 Lab 3 Specifications

(Some) SSet Implementations

Prelab: due on brightspace by Friday February 16th, 3:00pm (no lates)
Programming: due on gradescope by Wednesday February 28th, 3:00pm (24h late ok)
Postlab: due on brightspace.ca by Wednesday March 6th, 3:00pm (no lates)
Topic focus: Lec 1 - 11

Feedback on Lab 1 Feedback

Hi all, thank you to the 83 (!!) of you that took the time to provide feedback after Lab 1; I know you are all very busy! I am reading over your feedback carefully, and making some adjustments to the course in response. You can read more about your feedback and the changes I will and not make (and why) here.

Changes from Lab 1 & Lab 2

The specifications are still highly structured in the same way as Labs 1 and 2, but as with Lab 2, a lot of the non-programming details are just links to previous documentation. This is so that you don't feel you have to read such a long document unless you need the resources (in which case, you can follow the links to them.)

I have added a section at the very end: Autograder Runtimes for Sample Solutions, which provides screenshots of the autograder as it ran on sample solutions. This is to give you a sense of some runtimes you might strive for. For what it's worth, I ran this tests during Lab 2 crunch times.

Any randomization in the autograder is now "fixed," as in, the input was randomly generated initially by me, but then fixed for the autograder. As such, everyone is being tested on exactly the same input every single time. Any variability in the autograder can now definitely be attributed to load/gradescope rather than variability in the tests (which I suspect was negligible anyways, but just to be safe.)

There are still no hidden tests. I will wait to see if the variability of the autograder is resolved before I go hiding any tests. In some sense, the variability of the autograder is serving as a "hidden test" in the sense that you have to sometimes look at your code and determine its time/space complexity and decide whether it matches the desired complexity rather than relying on the occasionally flaky tests.

The GeneSet autograder has more structured feedback; more info in Geneset.

Lab Objectives

The labs in this course are meant to give you the opportunity to practice with the topics of this course in a way that is challenging yet also manageable. At times you may struggle and at others it may seem more straight-forward; keep trying and practicing and you will improve.

Specifically, Lab 3 aims to improve your

1. Implementation Skills
   Implement operations (add, remove, find, etc.) for linked-list-based data structures similar to the SkiplistList and SkiplistSet (and, SLList and DLList, to some extent.)
2. Design Skills
   Use object-oriented programming concepts such as abstraction and polymorphism to provide flexibility in your choices of which interface implementation to use.
3. Critical Thinking
   Demonstrate a solid understanding of the pros and cons of various implementations of the list (array-based, linked-list-based, and SkiplistList-based) and of the situations when a sorted set might be a better choice than a list. This includes considering the time- and space-complexity of various operations in different data structures.
4. Algorithmic Thinking
   Apply algorithmic thinking to design efficient algorithms for common linked-list-based operations, such as findPredNode(i), iteration over nodes using pointers and integers, careful pointer manipulation, and using nested structures to store and access data.
5. Error Handling
   Implement appropriate error handling and boundary checks to ensure the robustness of the data structures.
6. Testing
   Determine the necessary tests to ensure your algorithms are correct and efficient.
7. Planning
   Maintain or adopt good academic and programming habits through the pre-lab.
8. Reflection
   Reflect on your choice of data structures and algorithms through the post-lab.

Assignment Components

Details of each component follow later in the specifications.

1. (6.7 points) Prelab (complete on brightspace)
2. (80 points) Programming portion (submit on gradescope.ca)
   a. (40 points) `GeneSet` Implementation
   b. (10 points) `JumbleC` implementation
   c. (10 points) `Genes` Implementation
   d. (10 points) `GenesOrder` implementation
   e. (10 points) `Mutations` implementation
3. (13.3 points) Postlab (complete on brightspace)

Grading Criteria & Submission Guidelines

See the Grading Criteria and Submission Guidelines of Lab 1; they are the same.

Collaboration & Academic Integrity

1. Individual work is expected. Any collaboration should be explicitly mentioned and acknowledged at the top of each file. It is **okay to discuss high-level approaches** with **your peers, or low-level syntax-type questions**, but you must construct your solution on your own (as in: you have to formulate the *code* of your solution on your own.) Consider the analogy of writing an essay. You might talk with a peer about the high-level concepts of your thesis, or you might ask them about grammar or even phrasing of individual sentences. But you should not be writing the essay sentence-by-sentence with someone else's help; in the end you have to sit down and write that thing on your own.
2. Plagiarism will result in severe consequences. **Ensure that all code and documentation are your own work**. Do not send code to or receive code from any source except for course staff or the textbook, even if you change a thing here or there. It helps to keep the analogy of an essay in mind; it is not okay to take a paragraph from a friend and then rearrange some of the words or replace some with a thesaurus. It's not even okay to paraphrase each sentence. It is not okay to send your essay to a peer. Similarly here, you cannot start with code that is not yours and then "make it your own" with minor edits. Automated tools for detecting plagiarism will be employed in this course.
3. The same restrictions apply to **AI programmers** (such as chatGPT, copilot). You can use them to help with basic syntax (e.g. "spelling" and "grammar") or to understand broad concepts (e.g. getting feedback on a thesis) but you have to formulate your solution in code on your own (e.g. you have to write that essay yourself.)
4. Note that **contract cheating sites** are known, unauthorized, and regularly monitored. Some of these services employ misleading advertising practices and have a high risk of blackmail and extortion.
5. Every student should be familiar with the Carleton University student academic integrity policy. Academic integrity is upheld in this course to the best of Prof Alexa's abilities, as it protects the students that put in the effort to work on coursework within the allowable parameters. Potential violations must be reported to the Dean of Academic Integrity. If you ever have **questions** about what is or is not allowable regarding academic integrity, **please do not hesitate to reach out to course staff**. We are happy to answer.

Copyright

Prof Alexa is the exclusive owner of copyright and intellectual property of all course materials, including the labs. You may use course materials for your own educational use. **You may not reproduce or distribute course materials publicly for commercial purposes**, or allow others to, without express written consent.

Workflow

In a perfect world, this is how you would complete Lab 3:

1. Attend or watch the relevant lectures that are listed in the heading of this document.
2. Read the lab objectives listed on the second page of this document. For each data structure listed there, review its important algorithms as well as the time- and space-complexity of its methods. This should give you some pros and cons for each.
3. Carefully read each problem detailed in the Programming section of this document.
    a. Make sure you understand the problem.
    b. Try the given examples by hand to get a better understanding of the problem.
    c. Pay special attention to any special cases or edge cases.
    d. Try more examples of your own devising if you need them.
    e. Do not start programming yet!
    f. Consider attending or watching the lab's workshop video, posted on brightspace.
4. Once you have completed steps 1-3, you are ready to do the prelab (brightspace).
5. Complete the programming portion of the lab.
    a. Take this one problem at a time. Any order should be okay.
    b. Remember what you learned in Steps 1-4 as you brainstorm solutions.
    c. Test locally at frequent intervals. Do not write a whole program then test it afterwards. See the section on Local Tests to help you here.
    d. Submit to gradescope.ca whenever you have made good progress, but do not use gradescope.ca as your only tests. Gradescope keeps your most recent score unless you select a different submission to be active. See the section on the Gradescope Autograder to help you debug here.
    e. If you're stuck on a problem for more than 30 minutes, ask for help using the How to Get Help section. Move on to something else until help has arrived.
6. Once the late programming deadline has passed, complete the post-lab (brightspace).
    a. If you did well on the programming portion, this is not meant to take too long.
    b. There are resources available to help you posted on brightspace under the Lab 2 module. There is a solutions walk-through video for each part, and also a debrief document where I walk through the problem solving process and learning engagement I was hoping you would experience. You might consider looking at these before completing the prelab if you had trouble with any of the programming parts.
    c. You do not have to have completed the programming parts in order to do the postlab. If you were stuck on a problem, this is an opportunity to look at the sample solution videos, to figure out what went wrong for you, and to still learn what you were meant to learn.

Coding Environment Setup

Lab 1's Coding Environment Setup will work with Lab 3 (replace `Lab1/l1` with `Lab3/l3`). The file structure and many of the files will be the same, with new and different files as well.

Programming Components

Programming Notes

1. String compareTo might be helpful to compare strings lexicographically (alphabetically). Comparing elements can take more time for certain types of elements than others. Comparing two integers or doubles is basically O(1) time, as is comparing two Characters. But comparing two Strings requires potentially looping through both strings from front to back, which takes time linear in the length of the String. While Java has optimized String comparison as much as possible, there is only so much it can do if two strings of length k are exactly the same except for one character (say, the last one.)

2. Recall from Lab 2 that repeated String appends (i.e. the + operator) is costly and can often be avoided by using StringBuffer and its append method.

3. Note that the ods SSet interface has more than just the find operation, but also findLT and findGT that may be of use in this lab. Also you might take a look at its iterator.

4. AbstractList is the abstract java List interface. All of our textbook (ods) List implementations implement this interface (either implementing it explicitly, as the ArrayDeque does, or by implementing a subclass as the DLList does.) This allows us to choose different implementations of the same interface; we know there is a guarantee that we can call the methods and they will have the promised behaviour, but we do not need to know the inner workings of the implementation (or, which implementation you choose to use.) This is polymorphism and abstraction at work.

5. In order to make a new array of type Node in java, you need to do something like this:
   ```
   next = (Node [])Array.newInstance(Node.class, h+1);
   ```
   Anytime you use this you will need to add `@SuppressWarnings("unchecked")` prior to the method or inner class that creates the array. You can see examples of this scattered throughout the `GeneSet` class, should you need to do it yourself.

Implementation Practice [40 marks]

GeneSet [40 marks]

`GeneSet` represents a set of Strings (where a String can ostensibly represent a gene; no biology knowledge necessary for the lab). Your task is to implement the following methods according to the specifications below, using the starter code and lecture to guide you. The provided implementations are incomplete and incorrect, but should be enough to compile and run the (failing) local and autograder tests.

The data structure you use is a greatly simplified version of a SkiplistSSet, with randomization removed. Instead of determining the heights of elements via "coin flips," we will determine the heights of elements based on the pattern of search paths we construct as the SSet gets used.

As with a `SkiplistSSet`, a `GeneSet` is a dummy `Node` that links to a singly-linked list of `Nodes`. Each node `u` has its data (String `s`) and an array `next` of `u.height+1` pointers to its next nodes at levels `0, 1, 2, …, u.height`. That is, `u.next[r]` represents `u`'s "shortcut" at level-`r`, as with the `SkiplistSSet`. For example, a loop that would iterate over just the level `r` shortcuts would look something like this:

```
u = dummy;
while( u.next[r] != null )
    u = u.next[r]
```

As with a `SkiplistSSet`, not every node will have shortcuts at every level; a node's height represents the highest level `r` for which it has shortcuts; in other words, it is `u.next.length-1` (since heights are 0-based.)

Generally, we want our levels and heights to satisfy the four properties below:

1. all elements are in level `0` of our skiplist, in sorted order;
2. the elements at level `r` are all contained at level `r-1` (but probably not vice-versa);
3. the number of elements at level `r` is about half the number of elements at level `r-1`; and
4. when searching for an element `x`, we take at most ~1 shortcut "right" per level before dropping down to the next lower level.

In lecture, we saw the first saw the "fixed/deterministic" skiplist that allowed us to maintain these properties, but not without some restructuring cost on an add/remove. We then saw the SkiplistSSet that used randomization to get properties 3 and 4 in expectation (and maintained properties 1 and 2 always.)

In this lab, we will also aim to always maintain properties 1 and 2, and we will try to maintain 3 and 4 in a "just-in-time" basis. In particular, we will focus on property 4: if, in an `add`, `remove`, or `find` operation, we take two shortcuts at some level `r` (in "violation" of property 4), we will "just-in-time" add in a shortcut to level `r+1` that jumps over the two shortcuts we just took. The idea is that this new shortcut can't be used on the current operation, but it *can* be used on future

operations, and hopefully the cost of adding in the new shortcut can be amortized over the adds and removes that made the shortcut necessary.

This is the general idea. Let's get to some details.

Inner Class

`Node`		(Represents a Node in our set.)
`String`	`s`	The non-empty `String` in this `Node`.
`Node[]`	`next`	The `Node`s to which this `Node` has shortcuts.
`int`	`height()`	Returns the height of this `Node` (it's `next.length-1`)

Fields

- `Node` `dummy` — The dummy `Node` for the set (also known as a sentinel).
- `int` `n` — The number of elements in the set.
- `int` `h` — The maximum height of any element (`Node`) in the set.

Implemented Methods & Constructor

- `GeneSet()`
  Initializes the fields so that this set is empty (just a dummy `Node` of length 32).
  Note that the length of the dummy `Node` is not the same as the height of the `GeneSet`.

- `public int size()`
  Returns the number of elements in the set.
  See the `main` method for examples of how to use `size`.

- `public void clear()`
  Clears the set so that it contains no elements.
  See the `main` method for examples of how to use `clear`.

- `public String toString()`
  Returns a `String` representation of the set that is helpful for debugging purposes.
  See the `main` method for examples of how to use the `toString` method.

- `protected Node findPredNode(String s)`
  Returns the `Node` that precedes element `s` in the skiplist. This is for use in the iterator, so please do not modify it.

- `public Iterator<String> iterator()`

Returns an `Iterator` that iterates over the elements (Strings) of the set.
See the `main` method for examples of how to use the iterator.

- `public int[] getHeights()`
  Returns an array of length `n+1` with the heights of the `n+1` nodes (including the dummy node.) This is for testing purposes, so please do not modify it.
  See the `main` method for examples of how to use the `getHeights` method.

- `public Node getNode(int i)`
  Returns the i^{th} element of our set stored in level 0 of the skiplist. This is for testing purposes, so please do not modify it.
  See the `main` method for examples of how to use the `getNode` method.

Methods For You to Implement

Note: For all of these methods, I recommend getting them to work with just level 0 and level 1. The autograder has tests for solutions working with just those levels. Get the kinks out for levels 0 and 1 before proceeding to more levels. There are even some tests with just levels 0, 1, and 2, so that is another next step to take after getting levels 0 and 1 working.

- `public void resize(Node u, int newHeight)`
  Resizes the `u.next` array to have length `newHeight`, if `newHeight>u.height` is greater than its current length. Does nothing if `newHeight≤u.height`.
  Update the `height` of the GeneSet (`h`) if necessary.
  This would typically be a private method, but it is public so that the autograder can test it.

If you've implemented `resize`, it is a very good idea to test this locally and on the autograder before proceeding. Bugs in `resize` will be hard to track down later.

The `add`, `remove` and `find` methods will all share similar logic that finds the predecessor node of input `s`; usually I'd recommend working on a `findPredNode` method whose logic you can reuse in `add`, `remove` and `find`, we would not be able to test such a method until we have an `add` method that allows us to construct a set in the first place! As such, I recommend you work on `add` first, but then remember that the logic of add may be easily transferred to the other methods.

- `public boolean add(String s) throws IllegalArgumentException`
  Adds the element `s` to the set, if it is not already there.
  Throws an `IllegalArgumentException` if `s` is the empty string or `null`.
  Returns true if the element is newly added; false if it was already in the set.

At any point if you traverse two existing level-`r` shortcuts (i.e. `next[r]` pointers), add a shortcut over those two edges at level `(r+1)`. This may require resizing various nodes.

If you've implemented `add(s)`, it is a very good idea to test this locally and on the autograder before proceeding. If your `add` isn't working, none of the other methods are likely to work.

- `public String find(String s) throws IllegalArgumentException`
  Returns the smallest element in the set that is ≥ `s` (in alphabetical order); returns `null` if no such element exists (i.e. if `s` > all elements of the set.)
  Throws an `IllegalArgumentException` if `s` is the empty string or `null`.
  At any point if you traverse two existing level-`r` shortcuts (i.e. `next[r]` pointers), add a shortcut over those two edges at level `(r+1)`. This may require resizing various nodes.

- `public boolean remove(String s) throws IllegalArgumentException`
  Removes `s` from the set; returns true if `s` was removed, false if `s` was not in the set.
  Throws an `IllegalArgumentException` if `s` is the empty string or `null`.
  At any point if you traverse two existing level-`r` shortcuts (i.e. `next[r]` pointers), add a shortcut over those two edges at level `(r+1)`. This may require resizing various nodes.

If you've implemented `find` and `remove` it is a very good idea to test this locally and on the autograder before proceeding.

- `public boolean equals(Object o)`
  Returns whether this `GeneSet` is equal to `o`'s `GeneSet`.
  Returns false if `o` is not an instance of `GeneSet`.
  This should not only check whether the sets contain the same elements, but also whether they contain Nodes of the same heights, and the same shortcuts.

Desired Complexity

In the following table, let

- `n` denote the number of elements in our set;
- `h` denote the maximum height of any node.

marks	method	time complexity	(extra) space complexity
8	`resize(u,t)`	`O(t)`	`O(t)`
8	`add(s)`	$O(h+n/2^h)$ [A] `[O(nh) occasionally, not often]`	`O(n)` `[occasionally]`
8	`find(s)`	$O(h+n/2^h)$ [A] `[O(nh) occasionally, not often]`	`O(n)` `[occasionally]`

8	remove(s)	O(h+n/2^h)[A] [O(nh) occasionally, not often]	O(n) [occasionally]
8	equals(o)	O(n)	O(1)

Examples (admire my ascii art, please)

Resize

Node u	u.height (pre)	method	t=u.height(post)	Node u (post)
_ _ |_|-> |s||_|->	1	resize(u,1)	1	_ _ |_|-> |s||_|->
_ _ |_|-> |s||_|->	1	resize(u,0)	1	_ _ |_|-> |s||_|->
_ _ |_|-> |s||_|->	1	resize(u,2)	2	_ |_|-> _ |_|-> |s||_|->

Add

GeneSet	method	returns	GeneSet (post)
L0: |	add(A)	true	L0: |-A
L0: |-A	add(A)	false	L0: |-A
L0: |-A	add(C)	true	L0: |-A-C
L0: |-A-C	add(E)	true	L1: |---C-- L0: |-A-C-E
L1: |---C-- L0: |-A-C-E	add(G)	true	L1: |---C---- L0: |-A-C-E-G
L1: |---C------ L0: |-A-C-E-G-I	add(I)	false	L1: |---C---G-- L0: |-A-C-E-G-I
L1: |---C---G-- L0: |-A-C-E-G-I	add(K)	true	L2: |-------G---- L1: |---C---G--- L0: |-A-C-E-G-I-K

L0: |-K	add(I)	true	L0: |-I-K
L0: |-I-K	add(G)	true	L0: |-G-I-K
L0: |-A-C-E-G-I-K	add(L)	true	L1: |---C---G---K-- L0: |-A-C-E-G-I-K-L
L2: |-------K L1: |-------K L0: |-C-E-I-K	add(K)	false	L2: |-------K L1: |-------K L0: |-C-E-I-K
L2: |-------K L1: |-------K L0: |-C-E-I-K	add(L)	true	L2: |-------K L1: |-------K L0: |-C-E-I-K-L
L0: |-A-C-E-G-I-K	add("")	IllegalArgumentException	

Find

GeneSet	method	returns	GeneSet (post)
L0: |	find(A)	null	L0: |
L0: |-A	find(A)	A	L0: |-A
L0: |-A-C	find(B)	C	L0: |-A-C
L0: |-A-C-E-G-I-K	find(C)	C	L1: |---C-------- L0: |-A-C-E-G-I-K
L1: |---C-------- L0: |-A-C-E-G-I-K	find(L)	null	L1: |---C---G---K L0: |-A-C-E-G-I-K
L1: |---C---G---K L0: |-A-C-E-G-I-K	find(L)	null	L2: |-------G---- L1: |---C---G---K L0: |-A-C-E-G-I-K
L0: |-A-C-E-G-I-K	find("")	IllegalArgumentException	

Remove

GeneSet	method	returns	GeneSet (post)
L0: |	remove(A)	false	L0: |
L0: |-A	remove(A)	true	L0: |
L0: |-A-C-E-G-I-K	remove(B)	false	L0: |-A-C-E-G-I-K
L0: |-A-C-E-G-I-K	remove(A)	true	L0: |-C-E-G-I-K
L0: |-C-E-G-I-K	remove(E)	true	L0: |-C-E-G-I-K

L0: |-C-E-G-I-K	remove(G)	true	L1: |---E---- L0: |-C-E-I-K
L1: |---E---- L0: |-C-E-I-K	remove(L)	false	L1: |---E---K L0: |-C-E-I-K
L1: |---E---K L0: |-C-E-I-K	remove(L)	false	L2: |-------K L1: |---E---K L0: |-C-E-I-K
L2: |-------K L1: |---E---K L0: |-C-E-I-K	remove(K)	true	L1: |---E-- L0: |-C-E-I
L2: |-------K L1: |-------K L0: |-C-E-I-K	remove(K)	true	L1: |---E-- L0: |-C-E-I
L0: |-C-E-I-K	remove("")	IllegalArgumentException	

Equals

GeneSet this	**GeneSet o**	**method**	**returns**
L0: |	null	this.equals(o)	false
L0: |	"ABCD" <non-GeneSet>	this.equals(o)	false
L0: |	L0: |	this.equals(o)	true
L0: |-A	L0: |	this.equals(o)	false
L0: |-A	L0: |-B	this.equals(o)	false
L0: |-A-C	L1: |---C L0: |-A-C	this.equals(o)	false

Interface Practice [40 marks]

The 4 problems in this section ask you to choose the best data structures (interfaces), if any, to solve each problem correctly and efficiently (with regards to time and space). As with Labs 1 and 2, the characters generated contain any combination and number of the four characters `A`, `U`, `C`, `G`. Ensure your implementation is efficient and handles different scenarios gracefully.

JumbleC (extends Jumble) (a.k.a. Jumble Returns)

Method Signature

```
public static String jumble(InputGenerator<Character> gen)
```

Method Behaviour

Returns a jumbled version of the input sequence generated by `InputGenerator<Character> gen` *after it has been jumbled* as follows:

1. Initially the "cursor" is at index 0.
2. As characters are generated, add them at the cursor.
3. If an `AUC` is generated, "rewind" the cursor $\lfloor \frac{n}{2} \rfloor$ positions, where `n` is the number of characters generated thus far, or to 0 if this would move the cursor left of 0.
4. If an `AUG` is generated, "fast forward" the cursor $\lfloor \frac{n}{4} \rfloor$ positions, where `n` is the number of characters generated thus far, or to `n` if this would move the cursor right of `n`.

Desired Complexity & Constraints

`O(n log n)`[A] time, where `n` is the number of characters generated.
`O(n log n)` space.

Examples

characters generated by `gen`	output	`n`	`d`
`UUUAUCG`	`UUUGAUC`	7	1
`UUAUCGGGAUGUU`	`UUAGGGAUGUCUU`	13	2
`UAUCAUCAUCAUC`	`AUCUAAUCAUCUC`	13	4
(no characters)	`""`	0	0

Testing & Autograder

There are limited local tests in `JumbleC.main` and `tests/JumbleCTest.java`; see Lab 1 for testing instructions. Submit `JumbleC.java` to gradescope; see Lab 1 for submission instructions.

Genes

Method Signature

```
public static int genes(InputGenerator<Character> gen, int k)
```

Method Behaviour

Returns the number of different genes (substrands) of length `k > 0` in the sequence generated by a given `InputGenerator`.

Desired Complexity & Notes

> $O(k^2n)^A$ time, where `n` is the number of characters generated.
> $O(kd)$ space, where `k` is the integer input parameter, and `d` is the solution.
> Since there are 4 bases, $d \leq 4^k$ and thus $\log d = O(k)$.
> Even within these bounds, do your best to minimize wasteful time or space operation in order to pass all the performance tests.

Examples

characters generated by `gen`	`k`	output (`d`)	`n`
AAAA	3	1	4
AAAAU	3	2	5
UGCGAAUAUA	3	7	10
UUUUCCCCAAAA	12	1	12
UUUUCCCCAAA	12	0	11
(no characters)	> 0	0	0

Testing & Autograder

There are limited local tests in `Genes.main` and `tests/GenesTest.java`; see Lab 1 for testing instructions. Submit `Genes.java` to gradescope; see Lab 1 for submission instructions.

GenesOrder

Method Signature

```
public static AbstractList<String> genesOrder(InputGenerator<Character>
gen, int k)
```

Method Behaviour & Notes

Returns a list of the different genes (substrands) of length `k > 0` in the sequence generated by a given `InputGenerator`, in the order they are first encountered.

An AbstractList is an abstract java interface that is implemented by all the ods List implementations (e.g. ArrayStack, ArrayDeque, DLList, SkiplistList, etc). Using AbstractList<String> as the return type means you have the flexibility to choose which of these List implementations best suits your code and the desired complexity requirements.

Desired Complexity & Notes

$O(k^2n)^A$ time, where n is the number of characters generated.
$O(kd)$ space, where k is the integer input parameter, and d is the solution.
Since there are 4 bases, $d \leq 4^k$ and thus $\log d = O(k)$.
Even within these bounds, do your best to minimize wasteful time or space operation in order to pass all the performance tests.

Examples

characters generated by `gen`	`k`	output (`d`)	`n`
`AAAA`	3	`AAA`	4
`AUAAA`	3	`AUA,UAA,AAA`	5
`UGCGAAUAUA`	3	`UGC,GCG,CGA…`	10
`UUUUCCCCAAAA`	12	`UUUUCCCCAAAA`	12
(no characters)	> 0	`0`	0

Testing & Autograder

There are limited local tests in `GenesOrder.main` and `tests/GenesOrderTest.java`; see Lab 1 for testing instructions. Submit `GenesOrder.java` to gradescope; see Lab 1 for submission instructions.

Mutations

Method Signature

```
public static int mutations(InputGenerator<Character> gen, int k)
```

Method Behaviour

Returns the number of different “closest mutations” computed over all substrands of length `k > 0` in the sequence generated by a given `InputGenerator`.
Given a strand of length `k`, its “closest mutation” is the nearest already-existing strand of length `k` just following it alphabetically, if it exists. For example

in list `[AAA]` the closest mutation to `AAA` is non-existent
in list `[AAA,CAA]` the closest mutation to `AAA` is `CAA`
in list `[AAA,CAA,ACA]` the closest mutation to `AAA` is `ACA`

Desired Complexity & Notes

$O(nk^2)^A$ time, where `n` is the number of characters generated.
$O(kd)$ space, where `k` is the integer input parameter, and `d` is the number of different strands of length `k` (same as in `Genes`).
Since there are 4 bases, $d \leq 4^k$ and thus $\log d = O(k)$.
Even within these bounds, do your best to minimize wasteful time or space operation in order to pass all the performance tests.

Examples

characters generated by `gen`	`k`	output	`n`
`AAA`	3	`0`	3
`AAAA`	3	`0`	4
`CAAA`	3	`1`	4
`CAAACAAA`	3	`2`	8
`ACGUACGUACGU`	3	`3`	12
(no characters)	> 0	`0`	0

Testing & Autograder

There are limited local tests in `Mutations.main` and `tests/MutationsTest.java`; see Lab 1 for testing instructions. Submit `Mutations.java` to gradescope; see Lab 1 for submission instructions.

Local Tests

See the Local Tests section of Lab 1; be sure to replace `comp2402w2411` with `comp2402w2413`!

Gradescope Autograder

See the Gradescope Autograder section of Lab 1; be sure to replace Lab 1 with Lab 3 where necessary.

How to Get Help

See the How to Get Help section of Lab 1. It will be updated with any new resources rather than duplicating information here.

Common Errors & Fixes

See the Common Errors & Fixes section of Lab 1. It will be updated with any new errors rather than duplicating information here.

Glossary

There is a glossary at the top of the Problem Solving & Programming Tips document. If you can't find a term listed there, please post publicly on piazza so that everyone can benefit from the answer!

Autograder Runtimes for Sample Solutions

To give you a sense for the autograder runtimes for "perfect scores" you can see some of the screenshots that follow.

GeneSet: I've made some adjustments to the autograder since making the screenshots below, by increasing the time bound on the correctness and space tests.

- all non-performance tests now have 3 seconds, which should be plenty of time to check for correctness (this is up from 2 seconds)
- all space performance tests now have 6 seconds, which should be plenty of time to check for space usage without failing due to time (although they are related: frequently an overly-space-intensive program will take too much time, as in order to use space you take time to do so. Keep that in mind if you *do* fail the space test due to time.)

Resize

resize correctness random heights between 0 and 31 (1/1)

```
Running test (4) resize correctness random heights between 0 and 31
        Test passed.
        Time taken: 0:00:01.053982
```

resize O(t^2) time (0.5/0.5)

```
Running test (5) resize O(t^2) time
        Test passed.
        Time taken: 0:00:01.077595
```

resize O(t) time (3.5/3.5)

```
Running test (6) resize O(t) time
        Test passed.
        Time taken: 0:00:01.406459
```

resize O(n) space (0.25/0.25)

```
Running test (7) resize O(n) space
        Test passed.
        Time taken: 0:00:01.069665
```

resize O(t) space (1.75/1.75)

```
Running test (8) resize O(t) space
        Test passed.
        Time taken: 0:00:02.881831
```

Space test has 4 seconds

Add

add O(n) time when add-to-back (0.25/0.25)

```
Running test (16) add O(n) time when add-to-back
        Test passed.
        Time taken: 0:00:01.155887
```

add O(h+n/2^h) time when add-to-back (1/1)

```
Running test (17) add O(h+n/2^h) time when add-to-back
        Test passed.
        Time taken: 0:00:01.875235
```

This test has 3 seconds

add O(n) time when add-to-middle (0.25/0.25)

```
Running test (18) add O(n) time when add-to-middle
        Test passed.
        Time taken: 0:00:01.194026
```

add O(h+n/2^h) time when add-to-middle (2.25/2.25)

```
Running test (19) add O(h+n/2^h) time when add-to-middle
        Test passed.
        Time taken: 0:00:02.250498
```

This test has 3 seconds

add O(n) total space (1.25/1.25)

```
Running test (20) add O(n) total space
        Test passed.
        Time taken: 0:00:01.922929
```

Space test has 4 seconds

Find

find O(n) time when find-at-back (0.25/0.25)

```
Running test (28) find O(n) time when find-at-back
        Test passed.
        Time taken: 0:00:01.149607
```

find O(h+n/2^h) time when find-at-back (1/1)

```
Running test (29) find O(h+n/2^h) time when find-at-back
        Test passed.
        Time taken: 0:00:01.921316
```

This test has 3 seconds

find O(n) time when find-in-middle (0.25/0.25)

```
Running test (30) find O(n) time when find-in-middle
        Test passed.
        Time taken: 0:00:01.081890
```

find O(h+n/2^h) time when find-in-middle (2.25/2.25)

```
Running test (31) find O(h+n/2^h) time when find-in-middle
        Test passed.
        Time taken: 0:00:02.406737
```

This test has 3 seconds

find O(n) total space (1.25/1.25)

```
Running test (32) find O(n) total space
        Test passed.
        Time taken: 0:00:01.863457
```

Space test has 4 seconds

Remove

remove O(n) time when remove-at-back (0.25/0.25)

```
Running test (40) remove O(n) time when remove-at-back
        Test passed.
        Time taken: 0:00:01.263236
```

remove O(h+n/2^h) time when remove-at-back (1/1)

```
Running test (41) remove O(h+n/2^h) time when remove-at-back
        Test passed.
        Time taken: 0:00:01.868628
```

This test has 3 seconds

remove O(n) time when remove-in-middle (0.25/0.25)

```
Running test (42) remove O(n) time when remove-in-middle
        Test passed.
        Time taken: 0:00:01.276897
```

remove O(h+n/2^h) time when remove-in-middle (2.25/2.25)

```
Running test (43) remove O(h+n/2^h) time when remove-in-middle
        Test passed.
        Time taken: 0:00:02.445577
```

This test has 3 seconds

remove O(n) total space (1.25/1.25)

```
Running test (44) remove O(n) total space
        Test passed.
        Time taken: 0:00:02.199136
```

Space test has 4 seconds

Equals

equals correctness when elements added and removed in reverse sorted order (0.5/0.5)

```
Running test (50) equals correctness when elements added and removed in reverse sor
        Test passed.
        Time taken: 0:00:01.058567
```

equals correctness random operations and order (0.5/0.5)

```
Running test (51) equals correctness random operations and order
        Test passed.
        Time taken: 0:00:01.223747
```

equals O(n^2) time (0.5/0.5)

All the time tests have 2 seconds

```
Running test (52) equals O(n^2) time
        Test passed.
        Time taken: 0:00:01.087960
```

equals O(n) time (3/3)

```
Running test (53) equals O(n) time
        Test passed.
        Time taken: 0:00:01.724112
```

equals O(1) extra space (1/1)

```
Running test (54) equals O(1) extra space
        Test passed.
        Time taken: 0:00:02.426291
```

Space test has 4 seconds

JumbleC

JumbleC O(n log n) time when stay near front, n~300,000 (0.25/0.25)

```
Running test (6) JumbleC O(n log n) time when stay near front, n~300,000
        Test passed.
        Time taken: 0:00:01.868529
```

JumbleC O(n log n) time when cursor near middle, d~n~1,000,000 (4/4)

```
Running test (7) JumbleC O(n log n) time when cursor near middle, d~n~1,000,000
        Test passed.
        Time taken: 0:00:01.718239
```

JumbleC O(n log n) time, cursor random, d~n~80,000 (2/2)

```
Running test (8) JumbleC O(n log n) time, cursor random, d~n~80,000
        Test passed.
        Time taken: 0:00:00.882546
```

these tests all get 3 seconds

JumbleC O(n log n) time, cursor random, d~n~1,000,000 (2/2)

```
Running test (9) JumbleC O(n log n) time, cursor random, d~n~1,000,000
        Test passed.
        Time taken: 0:00:01.789612
```

Jumble O(n log n) time when d=O(1), n=1,000,000 (1/1)

```
Running test (10) Jumble O(n log n) time when d=O(1), n=1,000,000
        Test passed.
        Time taken: 0:00:02.130860
```

Jumble O(n log n) space n=1,000,000 (0.25/0.25)

```
Running test (11) Jumble O(n log n) space n=1,000,000
        Test passed.
        Time taken: 0:00:02.038431
```

Genes

Genes O(n^2) time n~5,000, k~O(n), d~O(1) (0/0)

```
Running test (6) Genes O(n^2) time n~5,000, k~O(n), d~O(1)
        Test passed.
        Time taken: 0:00:00.730620
```

Genes O(kn) time n~100,000, k~1,000, d~O(1) (1/1)

```
Running test (7) Genes O(kn) time n~100,000, k~1,000, d~O(1)
        Test passed.
        Time taken: 0:00:01.128484
```

Genes O(kn) time n~15,000, k~O(n), d~O(n) (2/2)

```
Running test (8) Genes O(kn) time n~15,000, k~O(n), d~O(n)
        Test passed.
        Time taken: 0:00:02.356121
```

this test gets 3 seconds

Genes O(kn) time n~20,000, k~O(n), d~O(n) (2/2)

```
Running test (9) Genes O(kn) time n~20,000, k~O(n), d~O(n)
        Test passed.
        Time taken: 0:00:01.926914
```

this test gets 3 seconds

Genes O(n) space n=10,000, k, d=O(1) (0.5/0.5)

```
Running test (10) Genes O(n) space n=10,000, k, d=O(1)
        Test passed.
        Time taken: 0:00:00.749685
```

Genes O(kd) space n=1,000,000, k, d=O(1) (2/2)

```
Running test (11) Genes O(kd) space n=1,000,000, k, d=O(1)
        Test passed.
        Time taken: 0:00:01.246362
```

GenesOrder

GenesOrder O(n^2) time n~5,000, k~O(n), d~O(1) (0/0)

```
Running test (6) GenesOrder O(n^2) time n~5,000, k~O(n), d~O(1)
        Test passed.
        Time taken: 0:00:00.834384
```

GenesOrder O(kn) time n~100,000, k~1,000, d~O(1) (1/1)

```
Running test (7) GenesOrder O(kn) time n~100,000, k~1,000, d~O(1)
        Test passed.
        Time taken: 0:00:01.375378
```

GenesOrder O(kn) time n~15,000, k~O(n), d~O(n) (2/2)

```
Running test (8) GenesOrder O(kn) time n~15,000, k~O(n), d~O(n)
        Test passed.
        Time taken: 0:00:02.701102
```

GenesOrder O(kn) time n~20,000, k~O(n), d~O(n) (2/2)

```
Running test (9) GenesOrder O(kn) time n~20,000, k~O(n), d~O(n)
        Test passed.
        Time taken: 0:00:02.225428
```

GenesOrder O(n) space n=10,000, k, d=O(1) (0.5/0.5)

```
Running test (10) GenesOrder O(n) space n=10,000, k, d=O(1)
        Test passed.
        Time taken: 0:00:00.793853
```

GenesOrder O(kd) space n=1,000,000, k, d=O(1) (2/2)

```
Running test (11) GenesOrder O(kd) space n=1,000,000, k, d=O(1)
        Test passed.
        Time taken: 0:00:01.762986
```

Mutations

Mutations O(n^2) time n~5,000, k~O(n), d~O(1) (0/0)

```
Running test (6) Mutations O(n^2) time n~5,000, k~O(n), d~O(1)
        Test passed.
        Time taken: 0:00:00.749174
```

Mutations O(kn) time n~100,000, k~1,000, d~O(1) (1/1)

```
Running test (7) Mutations O(kn) time n~100,000, k~1,000, d~O(1)
        Test passed.
        Time taken: 0:00:01.243316
```

Mutations O(kn) time n~15,000, k~O(n), d~O(n) (2/2)

```
Running test (8) Mutations O(kn) time n~15,000, k~O(n), d~O(n)
        Test passed.
        Time taken: 0:00:02.586516
```

Mutations O(kn) time n~20,000, k~O(n), d~O(n) (2/2)

```
Running test (9) Mutations O(kn) time n~20,000, k~O(n), d~O(n)
        Test passed.
        Time taken: 0:00:02.023389
```

Mutations O(n) space n=10,000, k, d=O(1) (0.5/0.5)

```
Running test (10) Mutations O(n) space n=10,000, k, d=O(1)
        Test passed.
        Time taken: 0:00:00.746378
```

Mutations O(kd) space n=1,000,000, k, d=O(1) (2/2)

```
Running test (11) Mutations O(kd) space n=1,000,000, k, d=O(1)
        Test passed.
        Time taken: 0:00:01.541654
```