COMP 2402 W24 Lab 5 Specifications

PQs & Graphs & The Programming Interview (& Lists & Sets & Maps!)

Prelab: due on brightspace by Friday March 22nd, 3:00pm (no lates)
Programming: due on gradescope by Wednesday March 27th, 3:00pm (24h late ok)
Postlab: due on brightspace.ca by Wednesday April 3rd, 3:00pm (no lates)
Topic focus: Lec 1 - 19

Changes from Labs 1-4

Lab 5 is structured slightly differently than Labs 1-4 as its focus is more on algorithms than on data structure implementation. While you will still need to make careful choices about which data structures to use based on how you want to store and manipulate data, the algorithms that use the data structures are non-trivial (not that previous labs were trivial! But these might "feel different".)

Another focus of the back part of the course is on the skills needed for the "programming interview." While there is only one lecture explicitly on the programming interview, the principles therein have been on display throughout this course. You should always be trying the problem out on examples, breaking the problems into manageable (and testable!) chunks, **trying to find an exhaustive (just-get-it-done correct) solution**, and improving upon that solution in increments. As someone who used to conduct programming interviews, I can say with certainty that I was looking for these problem solving skills in my candidates and would make a note when a step was missing. Moreover, these will be very important for working on the Lab 5 problems. They are very good practice for "the programming interview," if I do say so myself (and I do 😛).

Finally, most of the problems in this lab are throwbacks to problems from previous labs. This gives you an opportunity to review those problems, and if you weren't able to get them the first time around, it gives you an opportunity and an excuse to go back to look over the debrief and sample solutions and get it the second time around.

There are still no hidden tests. In a sense, the autograder variability serves as a "hidden test" in that you sometimes have to look at your code and determine its complexity (gasp!) and decide whether it matches the desired complexity rather than relying on the occasionally flaky tests.

As an incentive to start this lab early, I will give +0.1 bonus points for each part that is completed (16/16) a week in advance (by Thursday, March 21 3pm.) See the pinned piazza post labeled "Bonus Point Opportunities" to see how bonus is computed in this course.

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 5 aims to improve your

1. Implementation Skills
   While there isn't a specific data structure implementation on this lab, you will need to review the implementations of NAStrand (Labs 1 and 2) and PhyloTree (Lab 4), as well as AdjacencyLists (for Reconstruct).
2. Design Skills (i.e. Programming Interview Skills)
   Find a "get it done" solution first, then make iterative improvements. Break your problem into modular sub-problems that can be solved, tested, and debugged individually.
3. Critical Thinking
   Demonstrate a solid understanding of the pros and cons of all the data structures seen so far (including different implementations), especially the PriorityQueue, Graph, Map, and Set, and Deque. 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 a variety of problems that involve lists, sets, maps, priority queues, trees, and graphs, as well as iterative versions of recursive algorithms.
5. Error Handling
   Implement appropriate error handling and boundary checks, when appropriate.
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. (16 points) `PhyloTree.likelyTree` implementation
   b. (16 points) `Bond` implementation
   c. (16 points) `DecodeA` implementation
   d. (16 points) `DecodeB` implementation
   e. (16 points) `Reconstruct` 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

Workflow

In a perfect world, this is how you would complete Lab 5:

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 5 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 5 (replace `Lab1/l1` with `Lab5/l5`). The file structure and many of the files will be the same, with new and different files as well.

Programming Components

Programming Notes

1. For some problems, some of java's data structures are available to you. Here is java's official documentation for the relevant data structures:
    - TreeSet
    - TreeMap
    - HashMap
    - HashSet
    - PriorityQueue
2. If you want to store elements in a java `Collection` but use their "reverse" sorted order, many offer a constructor that takes in a `Comparator`, and if you pass in `Collections.reverseOrder()` that will set the default Comparator to be the flip of what the default is. (e.g. any Collection of Comparables will offer this kind of constructor.)
3. The Programming Notes section of Lab 4 has information about writing your own Comparators or Comparable classes, if you decide you need them.
4. Note that you should not need recursion for any of the problems. You can/should run all tests with the `-Xss144k` flag (or, the smallest heap size your local machine will allow.) Sometimes you'll pass such tests even with recursion.
5. You can turn a `char c` into a `String` using `""+c` or `String.valueOf(c)`

Practice [80 marks]

Bond (Bond and RNAStrand, Revisited) [16 marks]

Method Signature

```
public int bond(InputGenerator<Character> gen)
```

Method Behaviour & Notes

Returns the maximum number of bonds that an RNA sequence given by input sequence over `A`, `C`, `G`, `U` could form, where bonds have to satisfy the following 4 properties:

1. every character is bonded with at most one other character;
2. `A` bonds with `U` and `C` bonds with `G`, in either order (they are valid RNA pairs according to Lab 1's isPair);
3. there are no "tight turns" in the bonds, i.e. if `(i,j)` are the indices of a bond then `j-i>4`
4. the bonds cannot "cross", i.e. if `(i,j)` and `(k,m)` are indices of two bonds, then you cannot have `i < k < j < m`; see the examples below for a visual.

The maximum number of bonds in the sequence $b_i, b_{i+1}, \ldots, b_{j-1}, b_j$ is given by the following recurrence:

$\texttt{bonds(i,j)} = 0$ if $\texttt{j-i} \leq 4$

$\texttt{bonds(i,j)} = \max_{\{i \leq t < j\}} [\texttt{bonds(i,t-1)} + \texttt{bonds(t+1,j-1)} + 1]$ if $\texttt{isPair}(b_i, b_t)$

$\texttt{bonds(i,j)} = \texttt{bonds(i,j-1)}$ if $\texttt{!isPair}(b_i, b_j)$

We want to return `bonds(0,n-1)`, where `n` is the length of the input sequence.

The recurrence above can be turned into a recursive method in a straightforward way; your job is to correctly implement it iteratively by unwinding the recursion (start with the base cases, i.e. short intervals of j-i then increase the interval length.)

You might consider first programming a get-it-done recursive solution, and then unwinding the recursion to an iterative solution.

Note that your only allowable import is `RNAStrand`; you will use (and submit) one of your `RNAStrand/NAStrand` implementations (from either Lab 1 or Lab 2) with the modification that it should be in package comp2402w24l5 (and you need fewer implemented methods). Once you figure out how you're accessing your RNAStrand, you will want to make the choice that is best for your time and space complexity.

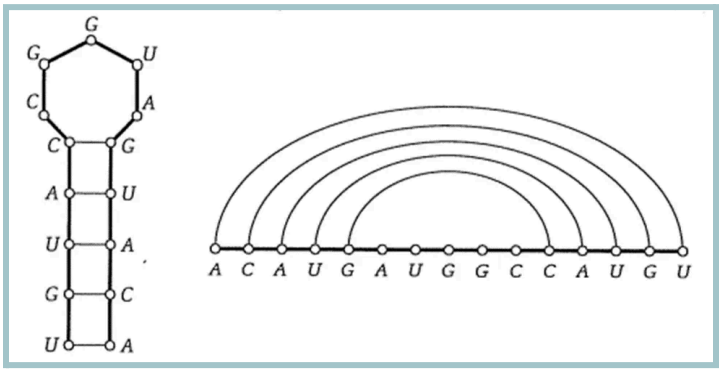
Desired Complexity

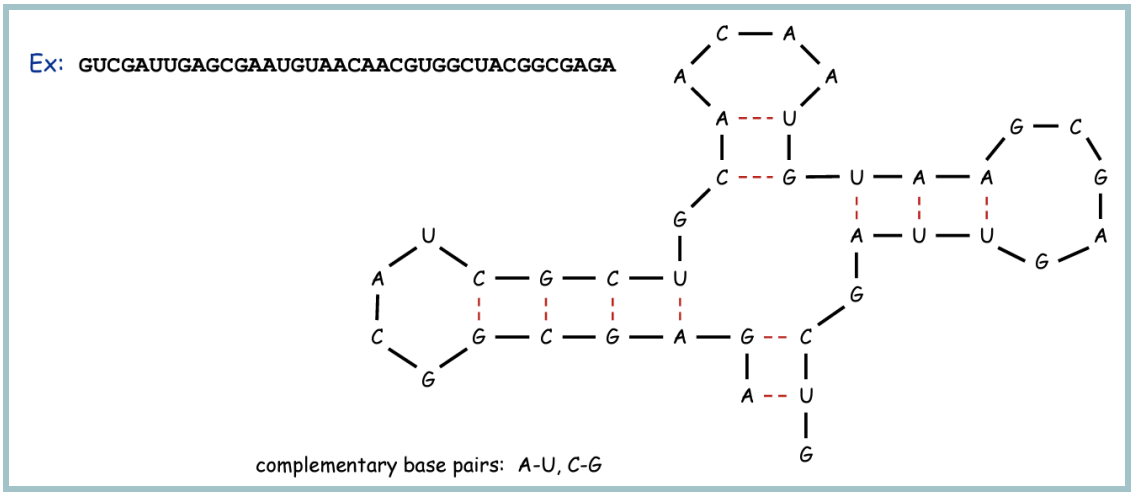
$O(n^3)$ time, where n is the number of bases generated.

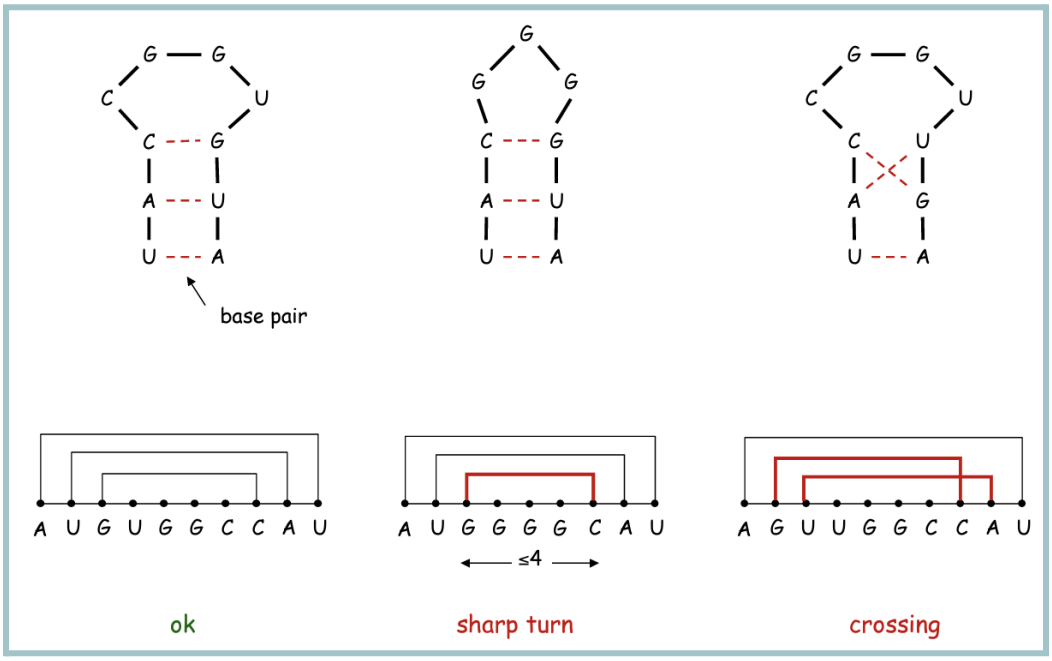
$O(n^2)$ space.

Examples

input	bonds	bond indices	output	n
ACAU			0	4
ACCAU			0	5
ACCCAU	ACCCAU |____|	(0,5)	1	6
UGAUGGCC	UGAUGGCC |____|	(1,6)	1	8
UGAUGGCC	UGAUGGCC |_____|	(1,7)	1	8
ACCAUUGAUGGCC	ACCAUUGAUGGCC |____||____|	(0,5),(6,11)	2	13
ACCAUUGAUGGCC	ACCAUUGAUGGCC |____||_____|	(0,5),(6,12)	2	13
ACCAUUGAUGGCC	ACCAUUGAUGGCC ||____| | |_____ _|	(0,8),(1,6)	2	13
ACCAUUGAUGGCC	ACCAUUGAUGGCC | |____| | |_____|	(1,10),(3,8)	2	13
ACAUGAUGGCCAUGU	see diagram below	(0,14),(1,13),(2,12),(3,11),(4,10)	5	15
ACAUGAUGGCCAUGU	see diagram below	(0,14),(1,13),(2,12),(3,11),(4,9)	5	15
GUCGUA…	see diagram below			
(no chars)				0







(source)

Testing & Autograder

There are limited local tests in `Bond.main` and `tests/BondTest.java`; see Lab 1 for testing instructions. Submit `Bond.java` and your preferred `RNAStrand.java` and `NAStrand.java` to gradescope; see Lab 1 for submission instructions.

LikelyTree (PhyloTree, Revisited) [16 marks]

In this problem you will implement one more method in `PhyloTree` from Lab 4. In order to get this working you will first need working versions of `addChild`, the array-based `PhyloTree` constructor, and `computeSets`. If you did not get these working on Lab 4, now is your chance to look at the debrief document and solutions videos to get a working version!

Method Signature

```
public void likelyTree()
```

For this problem, you may assume each `Node` has a length-`k` String of `DNABases` (or is `null`).
(You can use the provided class field `DNABases = {A,C,G,T}` here.)

Method Behaviour & Notes

Initially, all the leaves will be non-`null`; the internal nodes may or may not be `null`.
Given the current tree, construct the most likely "family" tree for the given list of leaf nodes by overwriting the Strings at the internal nodes.
Do not use recursion.
Throws an `IllegalArgumentException` if any of the leaves are `null`.

Use the following algorithm to compute the most likely "family" tree for each index i:

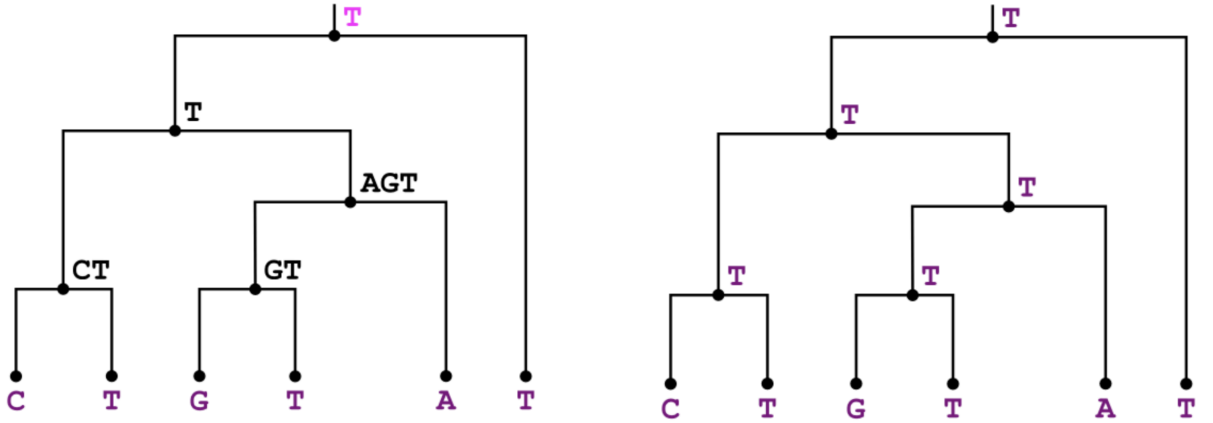
```
for each node v, compute its set of possible bases for index i
for each node v in pre-order (working from root downwards)
     if v is the root
          set v's index i to be the smallest element in v's set
     else if the v.parent's index i is in v's set
          set v's index i to be the same as v.parent's index i
     else
          set v's index i to be the smallest element in v's set
```

Desired Complexity

`O(nk²)` time, where `n` is the number of nodes in the tree, and `k>0` is the length of the non-null Strings in the tree.
`O(1)` space.

Examples

Tree (pre)	k	method	Likely Tree with [ComputeSets]
AT	2	likelyTree	AT[A,T]
null AC AG	2	likelyTree	AC[A,CG] AC[A,C] AG[A,G]
null A C	1	likelyTree	A[AC] A[A] C[C]
TT CC GG AC CA AT GT	2	likelyTree	AA[A,ACT] AA[AC,AC] AT[AG,T] AC[A,C] CA[C,A] AT[A,T] GT[G,T]
A G null	1	likelyTree	IllegalArgumentException (leaf is null)



computeSets likelyTree [source]

Testing & Autograder

There are limited local tests in `PhyloTree.main` and `tests/PhyloTreeTest.java`; see Lab 1 for testing instructions. Submit `PhyloTree.java` to gradescope; see Lab 1 for submission instructions. Note: I've left all the relevant tests from Lab 4 in the local tests and the autograder but you will need to add likelyTree to your Lab 4 code in order to pass the Lab 4 tests.

DecodeA [16 marks]

Method Signature

```
public static int decodeA(InputGenerator<Integer> gen, int k)
```

Method Behaviour & Notes

Returns an integer decoding of a given input sequence over positive Integers, produced as follows:

> Out of the $\leq k$ most recent **distinct** integers **prior** to the current one, consider the integer whose most recent occurrence is farthest back (i.e. its most recent index is smallest). Multiply this integer by the current index in the generated input, and add that to the output decoding.

You may assume $k>0$.

Note: First try to find an exhaustive ("get it done") solution to the subproblem of computing the integer whose most recent occurrence is the farthest back. Then worry about computing the sum. Then worry about improving its time and space complexity.

Desired Complexity

> $O(n \log d)$ time, where n is the number of integers generated and $d=\min\{k,$ number of distinct elements in the input$\}$
> $O(d)$ space.

Examples

integers generated by `gen`	`k`	decoded integers + indices	decoded value	output	`n`
`[1,1,1,1]`	3	`[1,1,1]+` `1-2-3` ←indices	$1\cdot1+1\cdot2+1\cdot3$ $=6$	6	4
`[1,2,3,4]`	1	`[1,2,3]+` `1-2-3` ←indices	$1\cdot1+2\cdot2+3\cdot3$ $=1+4+9$	14	4
`[1,2,3,4]`	2	`[1,1,2]+` `1-2-3`	$1\cdot1+1\cdot2+2\cdot3$ $=1+2+6$	9	4
`[1,2,1,3,2]`	2	`[1,1,2,1]+` `1-2-3-4`	$1\cdot1+1\cdot2+2\cdot3+1\cdot4$ $=1+2+6+4$	13	5
`[1,2,1,3,2]`	3	`[1,1,2,2]+` `1-2-3-4`	$1\cdot1+1\cdot2+2\cdot3+2\cdot4$ $=1+2+6+8$	17	5
`[1,2,3,1,2,3]`	3	`[1,1,1,2,3]`	$1\cdot1+1\cdot2+1\cdot3+2\cdot4+3\cdot5$	29	6

`[1,1,1,1,2,2,2,2,3,3,3,3]`	12	`[1,1,1,1,1,1,1,1,1,1,1,1]`	`1(1+2+3+...+11)`	`66`	12
`[1,1,1,1,2,2,2,2,3,3,3]`	12	`[1,1,1,1,1,1,1,1,1,1]`	`1(1+2+3+...+10)`	`55`	11
`[1,1,1,1,2,2,2,2,1,3,3,3]`	12	`[1,1,1,1,1,1,1,1,2,2,2]`	`1(1+2+...+8)+2(9+10+11)`	`96`	12
`[1,1,1,2,2,2,3,3,3,4,4,4]`	3	`[1,1,1,1,1,1,1,1,1,2,2]`	`1(1+2+...+9)+2(10+11)`	`87`	12
`[1,2,1,3,4,1,2,1,2,3,4]`	4	`[1,1,2,2,2,2,3,3,3,4]`	`1(1+2)+2(3+4+5+6)+3(7+8+9)+4·10`	`151`	11
(no chars)	> 0			`0`	0

- see the piazza post “DecodeA worked examples” for some of these worked out in detail.

Testing & Autograder

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

DecodeB [16 marks]

Method Signature

```
public static int decodeB(InputGenerator<Integer> gen, int k)
```

Method Behaviour & Notes

Returns an integer decoding of a given input sequence over positive Integers, produced as follows:

> For each integer in the sequence, examine up to `k` integers just before it. That is, if there are less then `k` characters before the current character, consider the maximum number of characters in this interval. Consider the integer that occurs most frequently in this interval, multiply this frequency by the current index, and add that product to the decoding.

You may assume `k>0`.

I would recommend first writing an "exhaustive" (just get-it-done) solution without worrying about efficiency. Then, incrementally replace parts of your exhaustive program with the appropriate data structures that will help you solve parts of the problem faster.

Desired Complexity

> `O(n log k)` time.
> `O(k)` space.

Examples

integers generated by `gen`	`k`	frequencies + non-zero indices	decoded value	output	`n`
`[1,1,1,1]`	3	`[1,2,3]+` `1-2-3` ←indices	`1·1+2·2+3·3`	`14`	4
`[1,2,3,4]`	1	`[1,1,1]+` `1-2-3` ←indices	`1·1+1·2+1·3`	`6`	4
`[1,2,3,4]`	2	`[1,1,1]+` `1-2-3`	`1·1+1·2+1·3`	`6`	4
`[1,2,1,3,2]`	2	`[1,1,1,1]+` `1-2-3-4`	`1·1+1·2+1·3+1·4`	`10`	5
`[1,2,1,3,2]`	3	`[1,1,2,1]+` `1-2-3-4`	`1·1+1·2+2·3+1·4`	`13`	5
`[1,2,3,1,2,3]`	3	`[1,1,1,1,1]`	`1·1+1·2+1·3+1·4+1·5`	`15`	6
`[1,1,1,1,2,2,2,2`	12	`[1,2,3,4,4,4,`	`1·1+2·2+3·3+4(4+5...+1`	`254`	12

`,3,3,3,3]`		`4,4,4,4,4]`	`1)`		
`111222333444`	3	`111222333444` `012322322322`	`1·1+2·2+3·3+2·4+...+2·11`	`149`	12
(no chars)	> 0			`0`	0

Testing & Autograder

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

Reconstruct [16 marks]

Method Signature

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

Method Behaviour & Notes

Most of Lab 3 and 4's Interface problems involve processing genes (i.e. substrings of length k) of a given input sequence. For *this* problem, you are given a list of genes (substrings of length k), and are asked to reconstruct the original sequence that could've produced exactly those substrings of length k. For example, gene list [ABC, BCD] (and k=3) could only have been produced from ABCD; similarly for the gene list [BCD, ABC]. You will return the original sequence as a String; if you were to then run your Lab 3-4 methods on this String as an input sequence, the genes produced would be exactly the gene list. The list is provided to you by the input generator with the genes in the list concatenated one after the other (in no particular order in relation to the output String.) That is, the input of length n will always be a multiple of k, and will represent n/k distinct genes. Examples below.

We will use a directed (multi-) graph G=(V,E) to solve this problem as follows:

- for each gene $c_1c_2…c_k$, add two nodes $c_1c_2…c_{k-1}$ and $c_2c_3…c_k$ to V. Add an edge between them from $c_1c_2…c_{k-1}$ to $c_2c_3…c_k$; this *edge* uniquely represents the gene $c_1c_2…c_k$.

Notes:

- If there are m genes in the input list, there will be m edges in the graph (including possibly self-loops and multi-edges.)
- If a gene occurs multiple times in the list, we may have multiple edges between the same two vertices. No worries, ods.AdjacencyLists handles this automatically. It is also accepting of self-loops, no changes necessary.
- A path v, w, u represents an output String that puts gene (v,w) followed by gene (w,u), which overlap on all but their first and last characters (because they have w in common.)
- To solve the problem, you need to find a path in this graph that traverses all the edges (i.e. "uses" all the genes) exactly once! This is known as an Eulerian path.
- In this directed graph, if there is a vertex with its out-degree one bigger than its in-degree, this is where your path must start; otherwise, you can start at any vertex.
- Construct the path by starting at the start vertex, then repeatedly traverse edges that have not already been used. The details of this algorithm are given below.
- There may be multiple possible correct outputs; any correct output will pass the tests.

```
Finding an Eulerian Path in a graph G=(V,E)
// start from start vx v, select unused edge.
// continue until all edges are used.
find a vertex s with outdegree(s)=indegree(s)+1
path = []
while( true )
     if this is the first iteration, start at v=s
```

```
otherwise, start at any vertex with outdegree(v)>0
stack = []
push v onto stack
while stack is not empty
   let u = top of stack
   if deg(u) > 0
       push any neighbour of u onto stack
   else
       pop stack
       push u onto path
path is reverse order of Eulerian path
```

Desired Complexity

$O(k^2(n-k))$ time, where n=length of output (input is $n-k+1$ genes, length k each.)
$O(k(n-k))$ space.

Examples

input generated by `gen`	k	gene list	graph	output	n
ATGTGT	3	ATG,TGT	AT->TG->GT	ATGT	4
TGTATG	3	TGT,ATG	AT->TG->GT	ATGT	4
ATGTTGTA	4	ATGT,TGTA	ATG->TGT->GTA	ATGTA	5
ATGTGTGTATATATGTGA	3	ATG,TGT,GTA,TAT,ATG,TGA	AT-2->TG->GT->TA \ \ / \ GA / _____/	ATGTATGA	8
AAAAAAAT	4	AAAA,AAAT	AAA->AAT \/	AAAAT	5
AAAAAA	3	AAA,AAA	AA-2->__/	AAAA	4
(no chars)	> 0				0

* note that the “2”s on top of edges means there are two edges represented. Some of the edges are missing arrows because ASCII art has its limitations, but if you can’t figure it out ask on piazza.

Testing & Autograder

There are limited local tests in `Reconstruct.main` and `tests/Reconstruct.java`; see Lab 1 for testing instructions. Unlike the other parts, this problem has a validate method that will

validate your algorithm on any sequence! See the main method for usage. Submit `Reconstruct.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 `comp2402w2415`

Gradescope Autograder

See the Gradescope Autograder section of Lab 1; be sure to replace Lab 1 with Lab 5 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.

Bond

Bond correctness test edge case empty strand (1/1)

```
Running test (4) Bond correctness test edge case empty strand
        Test passed.
        Time taken: 0:00:00.655945
```

Bond correctness random (1/1)

```
Running test (5) Bond correctness random
        Test passed.
        Time taken: 0:00:00.783066
```

Bond O(n^3) time n=200 (1/1)

```
Running test (6) Bond O(n^3) time n=200
        Test passed.
        Time taken: 0:00:00.844407
```

Bond O(n^3) time test n=400 (2/2)

```
Running test (7) Bond O(n^3) time test n=400
        Test passed.
        Time taken: 0:00:01.238860
```

Bond O(n^3) time n=600 (5/5)

```
Running test (8) Bond O(n^3) time n=600
        Test passed.
        Time taken: 0:00:02.427411
```

Bond O(n^2) space n=1,000 (1/1)

```
Running test (9) Bond O(n^2) space n=1,000
        Test passed.
        Time taken: 0:00:03.195917
```

Bond O(n^2) space n=1,500 and no recursion (3/3)

```
Running test (10) Bond O(n^2) space n=1,500 and no recursion
        Test passed.
        Time taken: 0:00:09.546222
```

LikelyTree

likelyTree correctness all leaves same (0.5/0.5)

```
Running test (27) likelyTree correctness all leaves same
        Test passed.
        Time taken: 0:00:00.976374
```

likelyTree correctness all sets disjoint (0.5/0.5)

```
Running test (28) likelyTree correctness all sets disjoint
        Test passed.
        Time taken: 0:00:01.082819
```

likelyTree correctness sets partially overlap (0.5/0.5)

```
Running test (29) likelyTree correctness sets partially overlap
        Test passed.
        Time taken: 0:00:01.115940
```

likelyTree correctness all leaves same random structure (0.5/0.5)

```
Running test (30) likelyTree correctness all leaves same random structure
        Test passed.
        Time taken: 0:00:01.064671
```

likelyTree correctness random assignment (0.5/0.5)

```
Running test (31) likelyTree correctness random assignment
        Test passed.
        Time taken: 0:00:01.023805
```

likelyTree exceptions (0.5/0.5)

```
Running test (32) likelyTree exceptions
        Test passed.
        Time taken: 0:00:00.951030
```

likelyTree correctness various (1/1)

```
Running test (33) likelyTree correctness various
        Test passed.
        Time taken: 0:00:01.099896
```

likelyTree O(n^2) time (1/1)

```
Running test (34) likelyTree O(n^2) time
        Test passed.
        Time taken: 0:00:01.029908
```

likelyTree O(nk) time (4/4)

```
Running test (35) likelyTree O(nk) time
        Test passed.
        Time taken: 0:00:01.846737
```

likelyTree O(1) space (3/3)

```
Running test (36) likelyTree O(1) space
        Test passed.
        Time taken: 0:00:06.009900
```

likelyTree O(1) call stack (no recursion) (4/4)

```
Running test (37) likelyTree O(1) call stack (no recursion)
        Test passed.
        Time taken: 0:00:02.021862
```

DecodeA

DecodeA correctness random (1/1)

```
Running test (5) DecodeA correctness random
        Test passed.
        Time taken: 0:00:00.801212
```

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

```
Running test (6) DecodeA O(n^2) time n~5,000, k~O(1), d~O(1)
        Test passed.
        Time taken: 0:00:00.860040
```

DecodeA O(nk) time n~100,000, k~100, d~100 (0.5/0.5)

```
Running test (7) DecodeA O(nk) time n~100,000, k~100, d~100
        Test passed.
        Time taken: 0:00:01.374291
```

DecodeA O(nk) time n~15,000, k~O(n), d~100 (0.5/0.5)

```
Running test (8) DecodeA O(nk) time n~15,000, k~O(n), d~100
        Test passed.
        Time taken: 0:00:00.848915
```

DecodeA O(n log k) time n~400,000, k~O(n), d~O(n) (2/2)

```
Running test (9) DecodeA O(n log k) time n~400,000, k~O(n), d~O(n)
        Test passed.
        Time taken: 0:00:01.996641
```

DecodeA O(n log d) time n~2,000,000, k~O(n), d~O(1) (5/5)

```
Running test (10) DecodeA O(n log d) time n~2,000,000, k~O(n), d~O(1)
        Test passed.
        Time taken: 0:00:01.616122
```

DecodeA O(n) space n~10,000, k, d=O(n) (0/0)

```
Running test (11) DecodeA O(n) space n~10,000, k, d=O(n)
        Test passed.
        Time taken: 0:00:00.849630
```

DecodeA O(k) space n=1,000,000, k~O(1), d~O(1) (1/1)

```
Running test (12) DecodeA O(k) space n=1,000,000, k~O(1), d~O(1)
        Test passed.
        Time taken: 0:00:01.550324
```

DecodeA O(d) space n=1,000,000, k~O(n), d~O(1) (3/3)

```
Running test (13) DecodeA O(d) space n=1,000,000, k~O(n), d~O(1)
        Test passed.
        Time taken: 0:00:00.946934
```

DecodeB

DecodeB correctness random (1/1)

```
Running test (6) DecodeB correctness random
        Test passed.
        Time taken: 0:00:00.935715
```

DecodeB O(n^2) time n~6,000, k~O(1) (0/0)

```
Running test (7) DecodeB O(n^2) time n~6,000, k~O(1)
        Test passed.
        Time taken: 0:00:00.911034
```

DecodeB O(nk^2) time n~10,000, k~60 (1/1)

```
Running test (8) DecodeB O(nk^2) time n~10,000, k~60
        Test passed.
        Time taken: 0:00:01.113089
```

DecodeB O(nk) time n~20,000, k~120 (4/4)

```
Running test (9) DecodeB O(nk) time n~20,000, k~120
        Test passed.
        Time taken: 0:00:01.921803
```

Challenge Level: DecodeB O(n log k) time n~45,000, k~O(n) (3/3)

```
Running test (10) Challenge Level: DecodeB O(n log k) time n~45,000, k~O(n)
        Test passed.
        Time taken: 0:00:01.990080
```

DecodeB O(n) space n~10,000, k=O(1) (1/1)

```
Running test (11) DecodeB O(n) space n~10,000, k=O(1)
        Test passed.
        Time taken: 0:00:01.000150
```

DecodeB O(k) space n=1,000,000, k~O(1) (3/3)

```
Running test (12) DecodeB O(k) space n=1,000,000, k~O(1)
        Test passed.
        Time taken: 0:00:05.024285
```

Reconstruct

Reconstruct correctness not unique, shuffled, output not unique (1/1)

```
Running test (6) Reconstruct correctness not unique, shuffled, output not unique
        Test passed.
        Time taken: 0:00:01.151309
```

Reconstruct correctness all same character (1/1)

```
Running test (7) Reconstruct correctness all same character
        Test passed.
        Time taken: 0:00:00.864388
```

Reconstruct O(n^2) time n~6,000, k~O(1) (0.5/0.5)

```
Running test (8) Reconstruct O(n^2) time n~6,000, k~O(1)
        Test passed.
        Time taken: 0:00:01.155770
```

Reconstruct O(nk^2) time n~10,000, k~60 (1.5/1.5)

```
Running test (9) Reconstruct O(nk^2) time n~10,000, k~60
        Test passed.
        Time taken: 0:00:01.429966
```

Reconstruct O((n-k)k^2) time n~20,000, k~120 (6/6)

```
Running test (10) Reconstruct O((n-k)k^2) time n~20,000, k~120
        Test passed.
        Time taken: 0:00:03.169389
```

Reconstruct O(n) space n~10,000, k=O(1) (1/1)

```
Running test (11) Reconstruct O(n) space n~10,000, k=O(1)
        Test passed.
        Time taken: 0:00:01.221999
```

Reconstruct O(k(n-k)) space n=20,000, k~O(1) (3/3)

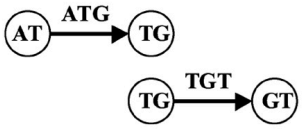
```
Running test (12) Reconstruct O(k(n-k)) space n=20,000, k~O(1)
        Test passed.
        Time taken: 0:00:01.538469
```

Eulerian Paths in various graphs

1. ATGT

   ATG

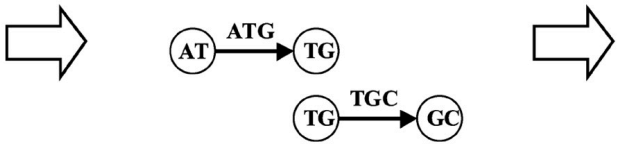
   TGT

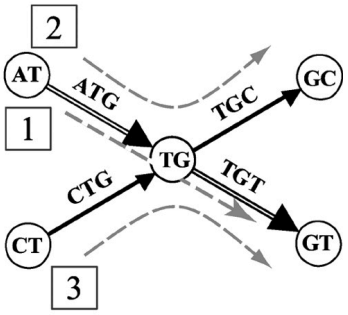


2. ATGC

   ATG

   TGC

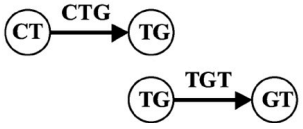




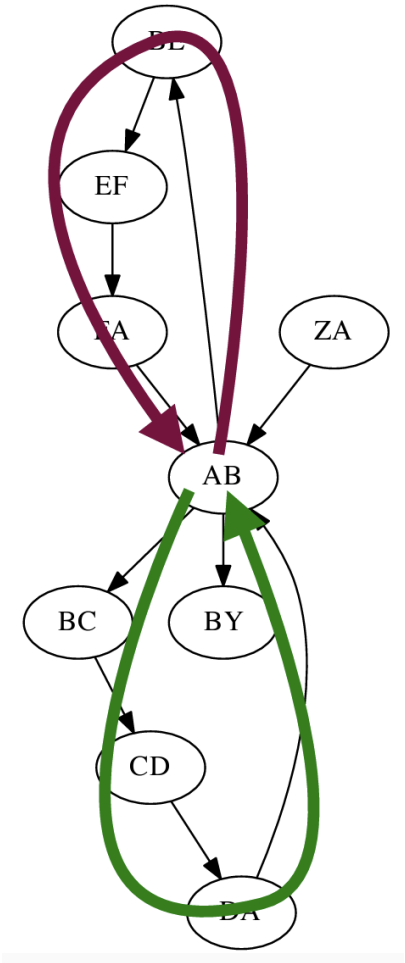
3. CTGT

   CTG

   TGT



source: https://www.pnas.org/doi/10.1073/pnas.0409240102



Right: graph for ZABCDABEFABY, $k = 3$

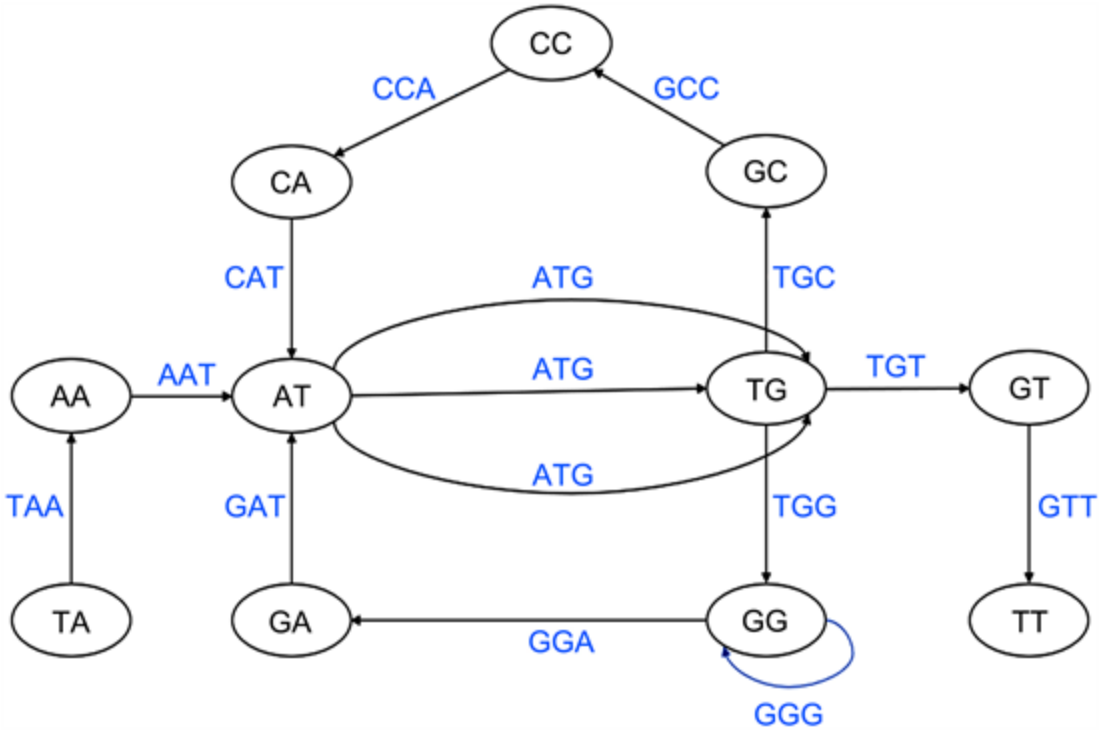
Alternative Eulerian walks:

ZA → AB → BE → EF → FA → AB → BC → CD → DA → AB → BY

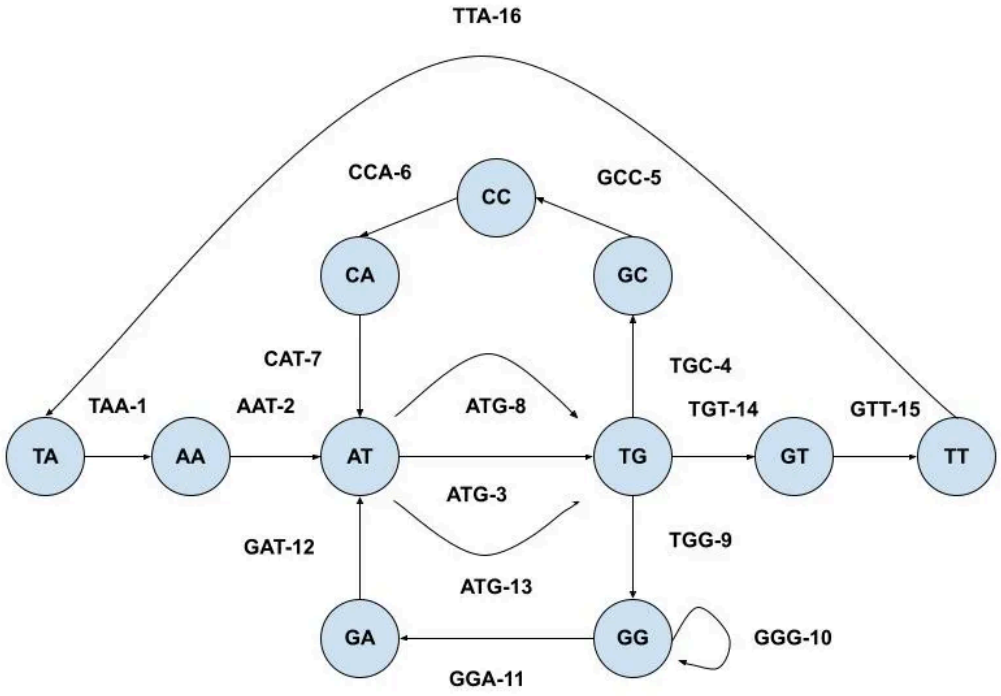
ZA → AB → BC → CD → DA → AB → BE → EF → FA → AB → BY

These correspond to two edge-disjoint directed cycles joined by node AB

AB is a repeat: ZABCDABEFABY



source: https://www.scirp.org/journal/paperinformation?paperid=124339



source:
https://towardsdatascience.com/how-to-sequence-a-human-genome-a-bioinformatics-approach-ae64481cec7b