Exam: Virus App

By Joel Pillar-Rogers Completed: 30 June 2020

(i) Describe how you would implement such a program, outlining the data structures and algorithms that you would use. Give examples of how the infected person and their contacts would be represented in your program and how the contacts would be found (15 marks, 1-2 pages).

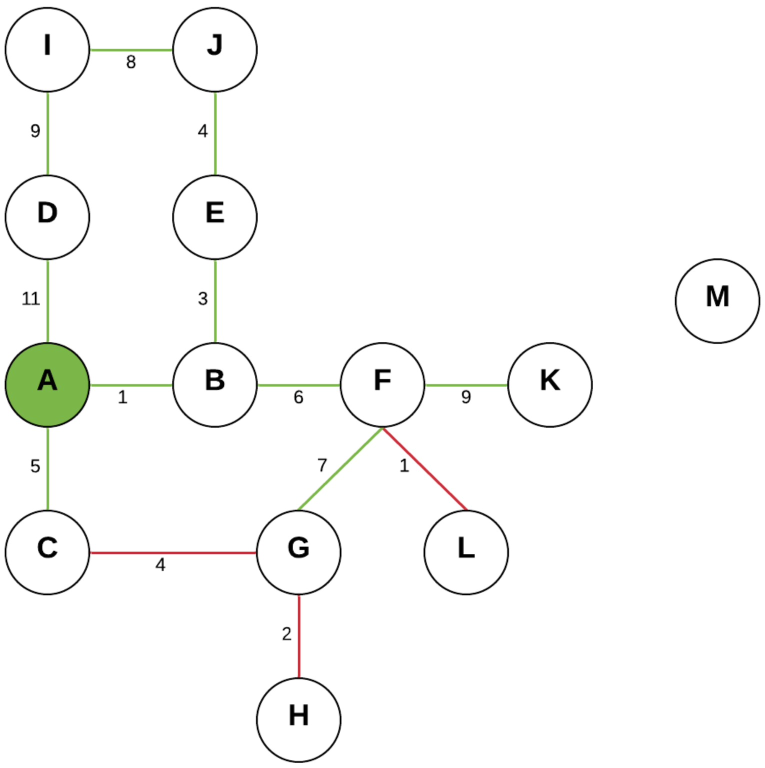
This is a graph problem. A simple implementation would have each person represented by a vertex, and each interaction with another person represented by an edge. You could then use a graph discovery algorithm, such as Depth-First Search or Breadth-First Search to find which people will need to be contacted for testing.

This specific scenario is complicated by the time dimension. The date and time of interactions matter and should be included in the analysis to create the most efficient contact tracing application. Without considering time, you would conduct a lot of unnecessary testing, because the number of people to test increases exponentially as you expand the graph.

An example will illustrate these points.

Suppose person A tested positive to the virus and assume the period they were infectious is the prior 2 weeks. Everyone they've come within 1.5 metres of in that time will need to be tested, which includes people B, C and D. Their contacts will also need to be tested, provided they interacted with them after interacting with A. These include E, F, G, H, I, J, K, L.

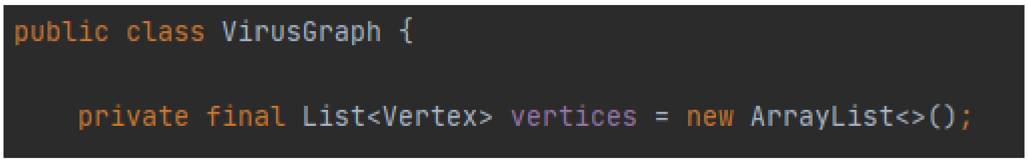
I've mapped out these interactions in the graph below and included the day of interaction (from day 1 to 14) along each edge (note: these aren't edge weights). This isn't a directed graph, because traversal is allowed in both directions. However, adding the day of interaction does prevent the virus moving along certain edges, so in some ways it operates like a directed graph. I've also included a contact, M, to demonstrate a person who didn't interact with anyone and won't ever be contacted.



This is a contact tracing graph for person A, who tested positive to the virus. The green edges indicate the path the virus could have followed, considering the day of interaction. The virus will never travel along red edges for this same reason. In this graph, H and L do not need to be tested, because they contacted other people before those others could have been infected. This also means, even without M, this is not a connected graph, as you can't reach every vertex from every other vertex.

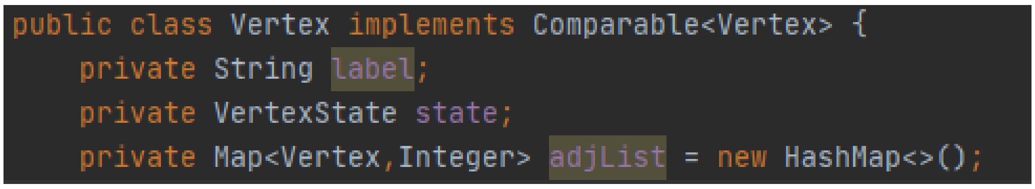
To implement this in code, you need data structures to store vertices (people) and edges (interactions) and an algorithm to traverse the graph. I did a quick implementation, based on the Labs and Assignments. The main alteration was for the *days* variable along each edge.

For **data structures**, my VirusGraph has an ArrayList that holds Vertex objects, which represent a person. Each Vertex contains a HashMap data structure, which holds the people contacted and day of contact for that person over the past 14 days.



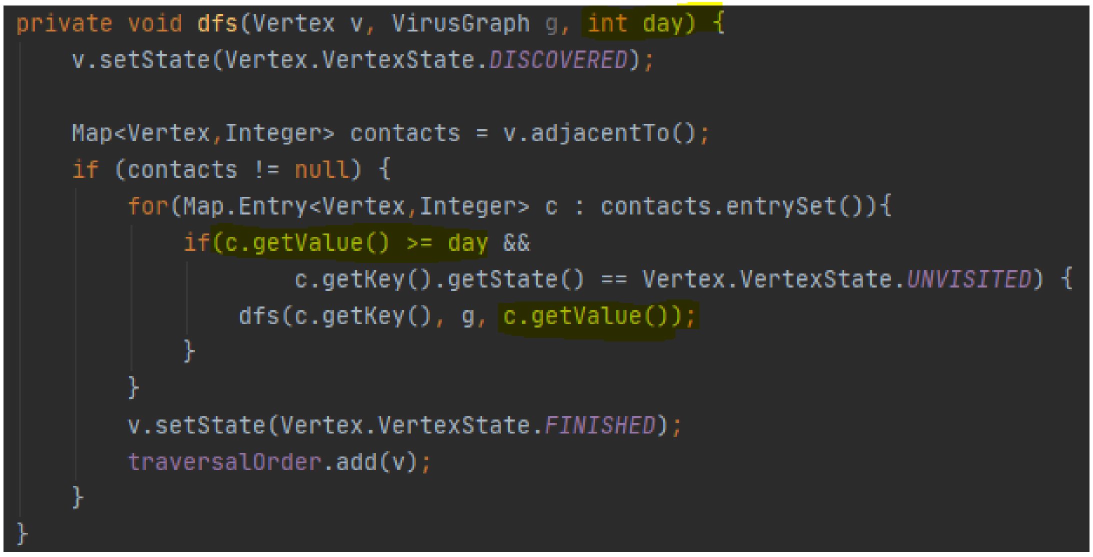
```java
public class VirusGraph {

    private final List<Vertex> vertices = new ArrayList<>();
```



```java
public class Vertex implements Comparable<Vertex> {
    private String label;
    private VertexState state;
    private Map<Vertex,Integer> adjList = new HashMap<>();
```

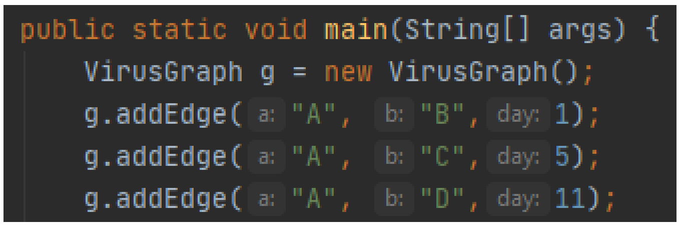
Each Vertex also saves a VertexState, which is used by the Depth-First Search algorithm to determine if it has been previously visited. I used the Depth-First Search **algorithm** from Lab03, with the alteration to also consider if the interaction happened earlier than the day the first person may have been infected.



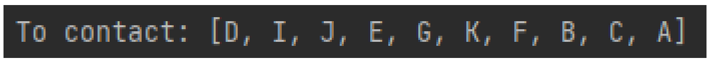
```java
private void dfs(Vertex v, VirusGraph g, int day) {
    v.setState(Vertex.VertexState.DISCOVERED);

    Map<Vertex,Integer> contacts = v.adjacentTo();
    if (contacts != null) {
        for(Map.Entry<Vertex,Integer> c : contacts.entrySet()){
            if(c.getValue() >= day &&
                    c.getKey().getState() == Vertex.VertexState.UNVISITED) {
                dfs(c.getKey(), g, c.getValue());
            }
        }
        v.setState(Vertex.VertexState.FINISHED);
        traversalOrder.add(v);
    }
}
```

Adding the edges and running the program produced the following sequence of people to be contacted:



```java
public static void main(String[] args) {
    VirusGraph g = new VirusGraph();
    g.addEdge( a: "A",  b: "B", day: 1);
    g.addEdge( a: "A",  b: "C", day: 5);
    g.addEdge( a: "A",  b: "D", day: 11);
```



```
To contact: [D, I, J, E, G, K, F, B, C, A]
```

Tracing through the graph above, you can see this is the expected traversal order of a Depth-First Search, which recursively moves as far down a path as it can, choosing the first lexicographical Vertex each time, until it can't go any further, then retraces it's steps. In the graph above, this order would be A->B->E->J->I->D and then back out to B->F->G, then F->K, then back out to A->C. In fact, person D could be infected from two sources, which isn't captured in my graph (it doesn't need to be). D could get the virus directly from A or indirectly from I.

(ii) How would you initialize the data structure? What is the time and memory complexity of doing this (O(n), O(m))? Your solution should be both time and memory efficient. Justify any trade-offs considered (15 marks, 1-2 pages).

There are two data structures suitable for this problem: a HashTable and a Trie. The HashTable uses a function to Hash the alphanumeric code to index directly into a table of contact info. The Trie is an offline tree data structure that pre-processes all the target strings, in this case the alphanumeric codes, and allows very fast retrieval of them.

The **creation time** of each data structure is O(S), where S is the total length of all strings. The HashTable will need to hash each code to place the related contact information object in the correct position. This takes O(m) each time, where m is the length of the code. This can sometimes result in collisions, which increase in probability as the load factor of the underlying array increases. Rehashing to decrease the load factor can be expensive but is amortised to very little over all operations. The Trie will need to add new nodes in the tree, and edges to the previous nodes, as it encounters each character in every code string, resulting in O(m) for each code and O(S) time to create the whole structure.

Both data structures can **retrieve** data in O(m) **time.** The HashTable just needs to rehash a given code to find and retrieve the contact info if it exists in the HashTable. Parsing the code string and hashing it takes O(m) time. The Trie structure will work through the tree structure, choosing the next character from the available edges of each node until it reaches a terminal node. This will also be O(m) time, as there are m characters in the code.

In terms of **memory**, the HashTable will store an entry for every person, which is O(n). The Trie will store every character of every code, which is O(S), where S is the length of all codes. However, in practice it will be closer to O(n), as I explain below.

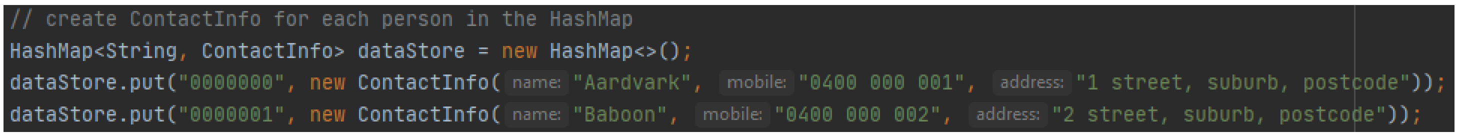
Assuming codes assigned to people increment for each user, as the Trie fills up, it is unlikely to have long uninterrupted strings shared by lots of codes (i.e. it will branch quickly). For example, assuming capitalisation doesn't matter, there are 26 letters and 10 numbers, so 36 possibilities for each character in the code. Log_{36} of 5 billion is 6.23, or 7 characters needed to represent that many unique codes (put another way, 36^6 = 2 billion, 36^7 = 78 billion). Codes might run from 0000000 to ZZZZZZZ but could all fit in 2ZZZZZZ (3 * 2 billion = 6 billion). Keeping them compacted in this space makes the Trie efficient without making it a Tree. It also means every edge is shared by the maximum number of codes.

In this setup, every node will have 36 edges to other nodes (aside from the first nodes), and each of those nodes will have 36 edges to other nodes, and so on, increasing exponentially. The final level of nodes, representing the last digit in the fixed-length code, will have 5 billion nodes. Each level before that will have logarithmically less nodes (e.g. O(n + log(n) + log(log(n)) +…) which reduces to O(n).

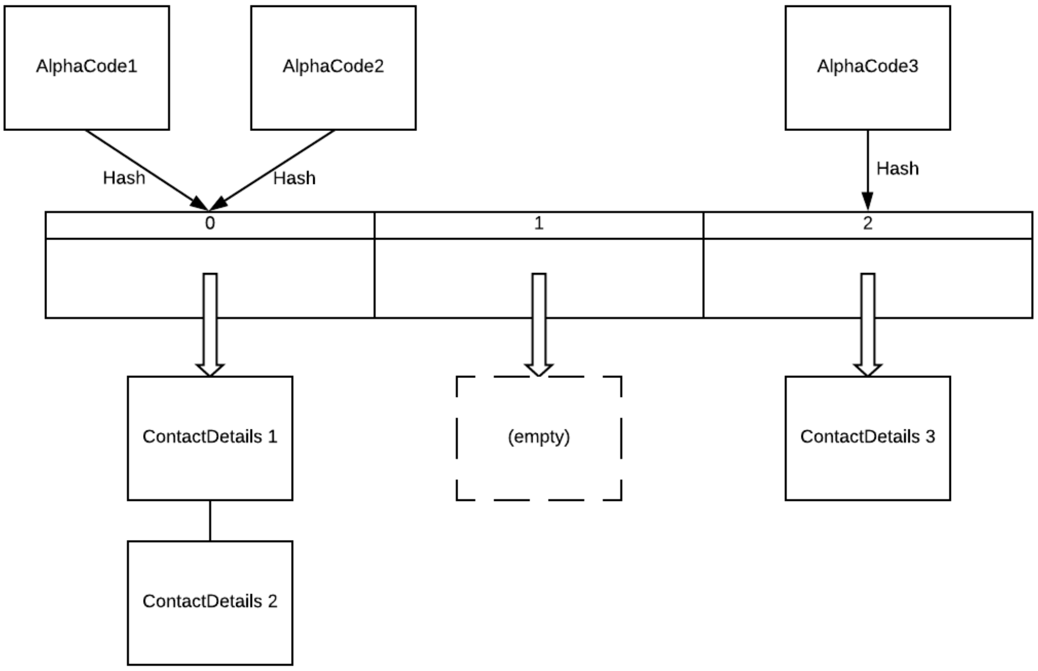
According to the lecture slides, a HashTable is probably faster than a Trie for this type of data. There is also no benefit from *prefix queries*, or *alphabetical sorting*, and the Trie is unable to return the correct contact *details after 1 step* in this case. Although the the retrieval speed will be blindingly fast for both, the HashTable probably has *better time and space complexity*, and is *simpler to implement* with standard libraries, so it seems like the right choice here.

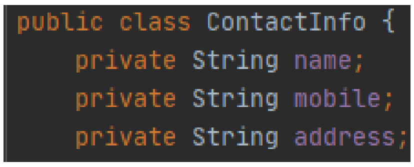
The Java HashMap uses separate chaining to deal with collisions. This means that each element in the array is actually a list of the objects intended to be stored there. If two codes hash to the same value, both objects are stored in the list at that position in the array (see figure below).

I implemented a HashMap to store contact details in my Java program. I created a new class to store contact details, called ContactInfo. The HashMap uses Strings for keys and ContactInfo for values. I also added an alphanumeric code to each Vertex/Person in my VirusGraph. Currently, I use string literals to add ContactInfo into the HashMap. In a real scenario, this would probably involve a data import or a GUI frontend, with alphanumeric codes generated automatically. Also, for this example I'm saving all these data in volatile RAM, instead of in secondary storage.

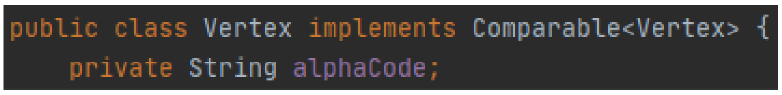

```
// create ContactInfo for each person in the HashMap
HashMap<String, ContactInfo> dataStore = new HashMap<>();
dataStore.put("0000000", new ContactInfo( name: "Aardvark", mobile: "0400 000 001", address: "1 street, suburb, postcode"));
dataStore.put("0000001", new ContactInfo( name: "Baboon", mobile: "0400 000 002", address: "2 street, suburb, postcode"));
```

HashMap example implementation in Java, with Separate Chaining

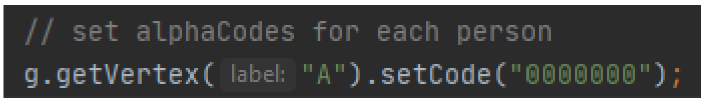




```
public class ContactInfo {
    private String name;
    private String mobile;
    private String address;
```

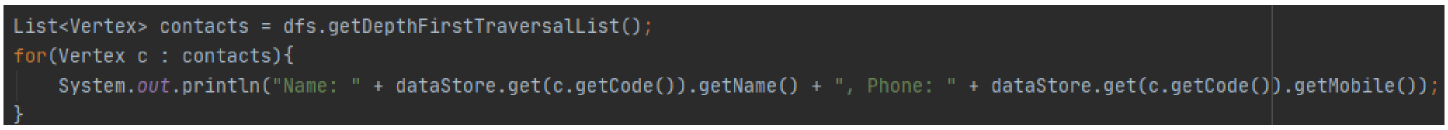


```
public class Vertex implements Comparable<Vertex> {
    private String alphaCode;
```

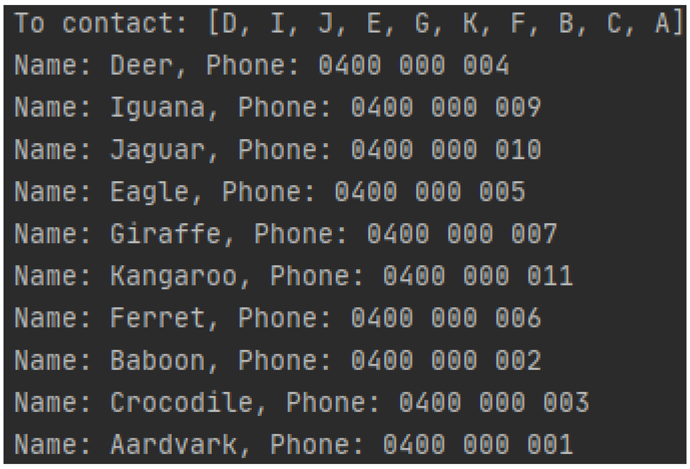


```
// set alphaCodes for each person
g.getVertex( label: "A").setCode("0000000");
```

My code now printed out the names and phone numbers of all the people to be contacted as a result of Person A testing positive to the virus. It used the alpha code from each Vertex to rehash into the HashMap and retrieve the related details. Retrieving each set of details is an O(m) operation, as the code string is rehashed and indexed into the hash table.



```
List<Vertex> contacts = dfs.getDepthFirstTraversalList();
for(Vertex c : contacts){
    System.out.println("Name: " + dataStore.get(c.getCode()).getName() + ", Phone: " + dataStore.get(c.getCode()).getMobile());
}
```



```
To contact: [D, I, J, E, G, K, F, B, C, A]
Name: Deer, Phone: 0400 000 004
Name: Iguana, Phone: 0400 000 009
Name: Jaguar, Phone: 0400 000 010
Name: Eagle, Phone: 0400 000 005
Name: Giraffe, Phone: 0400 000 007
Name: Kangaroo, Phone: 0400 000 011
Name: Ferret, Phone: 0400 000 006
Name: Baboon, Phone: 0400 000 002
Name: Crocodile, Phone: 0400 000 003
Name: Aardvark, Phone: 0400 000 001
```

(iii) How would you test each of the unknown DNA sequences from the saliva against the newly discovered virus DNA sequence? (10 marks, ½-1 page)

This is an approximate string-matching problem. The protein sequences from the saliva need to be compared with the protein sequence of the virus DNA. Saliva sequences within a specified edit distance of the virus sequence could be considered a positive result / match due to virus mutation, missing start or finish proteins or some other minor corruption from the testing process.

I considered whether to use a Longest Common Substring (LCS), but the scenario makes it seem like there could be lots of errors and sequence fragments in the saliva sequences so an Approximate String Matching (ASM) technique might be better.

ASM allows comparison of various substrings to a target text, in this case the virus DNA. It applies Dynamic Programming, the idea that a table of solved subproblems can be used to solve larger problems, to the Levenshtein Edit Distance algorithm, which establishes the fewest edits needed to turn a pattern into a substring of a target text.

Below are the first 60 proteins of the virus sequence, with a random example saliva sequence. To populate the table, I worked across each row, before moving to the next row. This makes it an O(mn) operation, where m is the length of the saliva pattern and n is the length of the target virus sequence. At each cell, I calculated the minimum value of deleting (down, +1), inserting (right, +1), or substituting (diagonal, +1) if the row character didn't match the column character (diagonal, +0). Using a maximum distance of 1 along the bottom row, there are three possible candidates for a positive result from this saliva test. Retracing through the algorithm returns the following possible matches.

1. aggtct -> aggtt (delete the 'c')
2. aggtct -> aggttt (swap 'c' for 't')
3. aggtct -> agatct (swap 'g' for 'a')

ASM - Saliva sequence to Virus sequence[1]

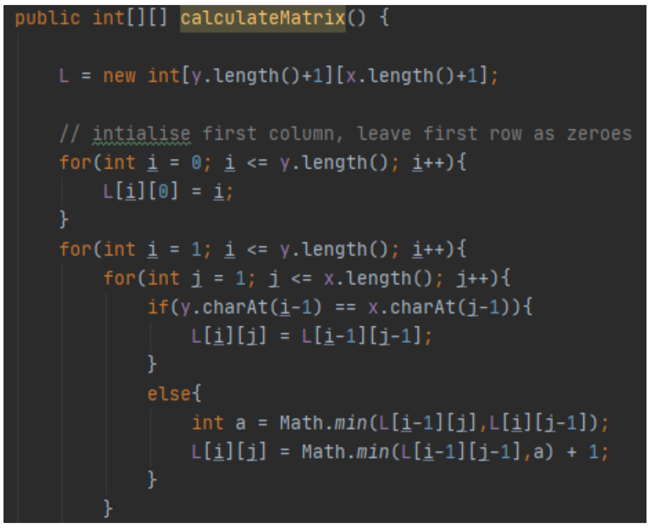
		-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
			a	t	t	a	a	a	g	g	t	t	t	a	t	a	c	c	t	t	c	c	c	a	g	g	t	a	a	c	a	a
-1		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	a	1	0	1	1	0	0	0	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	0	1	1	1	0	0	1	0	0
1	g	2	1	1	2	1	1	1	0	1	2	2	2	1	1	1	1	2	2	2	2	2	2	1	0	1	2	1	1	1	1	1
2	g	3	2	2	2	2	2	2	1	0	1	2	3	2	2	2	2	2	3	3	3	3	3	2	1	0	1	2	2	2	2	2
3	t	4	3	2	2	3	3	3	2	1	0	1	2	3	2	3	3	3	2	3	4	4	4	3	2	1	0	1	2	3	3	3
4	c	5	4	3	3	3	4	4	3	2	1	1	2	3	3	3	3	3	3	3	3	4	4	4	3	2	1	1	2	2	3	4
5	t	6	5	4	3	4	4	5	4	3	2	1	1	2	3	4	4	4	3	3	4	4	5	5	4	3	2	2	2	3	3	4

		30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
		a	c	c	a	a	c	c	a	a	c	t	t	t	c	g	a	t	c	t	c	t	t	g	t	a	g	a	t	c	t
-1		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	a	0	1	1	0	0	1	1	0	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	0	1	0	1	1	1
1	g	1	1	2	1	1	1	2	1	1	1	2	2	2	2	1	1	1	2	2	2	2	2	1	2	1	0	1	1	2	2
2	g	2	2	2	2	2	2	2	2	2	2	2	3	3	3	2	2	2	2	3	3	3	3	2	2	2	1	1	2	2	3
3	t	3	3	3	3	3	3	3	3	3	3	2	2	3	4	3	3	2	3	2	3	3	3	3	2	3	2	2	1	2	2
4	c	4	3	3	4	4	3	3	4	4	3	3	3	3	3	4	4	3	2	3	2	3	4	4	3	4	3	3	2	1	2
5	t	5	4	4	4	5	4	4	4	5	4	3	3	3	4	4	5	4	3	2	3	2	3	4	4	4	4	4	3	2	1

This process could be repeated for each saliva protein sequence, potentially combining several fragments to meet a minimum confidence level that a test is positive.

The algorithm to calculate this is similar to the LCS algorithm from Lab03, without the first column initialised to 0. However, instead of adding values to find the maximum substring, the minimum edit is added, if any is needed. The algorithm is to the right and creates the same matrix as above.

Retrieving the matched sequences could also be implemented in a similar way to LCS but starting with a search along the bottom row to find edit distances below a certain value.



```java
public int[][] calculateMatrix() {

    L = new int[y.length()+1][x.length()+1];

    // intialise first column, leave first row as zeroes
    for(int i = 0; i <= y.length(); i++){
        L[i][0] = i;
    }
    for(int i = 1; i <= y.length(); i++){
        for(int j = 1; j <= x.length(); j++){
            if(y.charAt(i-1) == x.charAt(j-1)){
                L[i][j] = L[i-1][j-1];
            }
            else{
                int a = Math.min(L[i-1][j],L[i][j-1]);
                L[i][j] = Math.min(L[i-1][j-1],a) + 1;
            }
        }
```

[1] COVID-19 genome sequence: https://www.ncbi.nlm.nih.gov/nuccore/MN908947

(iv) Discuss what parts of your code can be parallelized and the issues associated with doing so. For example, if you have access to a 1000-core supercomputer, will you see a 1000 times speed up? (10 Marks, ½-1 page)

When looking for areas of the program to speed up with parallelization, there are two main considerations:

1. Which parts currently take the longest time (i.e. in O(n))
2. Which parts safely access memory (and which don't)

The graph building process at first seems fraught with memory access issues. I don't see an easy way to use multiple threads to build a graph. However, if you had many positive test results to run contact tracing on, it should be possible to create a new thread for each graph built. This could provide a speed boost if the contact chains were extensive.

Likewise, with the HashMap contact data store, as the process adds new hashes and ContactInfo objects to a shared memory location it wouldn't be ideal for parallelizing (conflicts would be a mess). However, since building this data store is the largest and slowest operation in my program (O(S)), it's worth investigating where threading can safely help. If the range of alphanumerical codes where broken into segments, then a separate thread could handle each segment. Given there are 5 billion codes to add, this amounts to 5 million codes for each of the 1000 cores. As Java uses separate chaining, it should be possible to Lock the LinkedList stored in an array element while adding a hash value and ContactInfo object to it.

You could also use multiple threads for retrieving contact details from the data store after it's built. However, these operations are so fast already there wouldn't be much gain here (it could make the operation slower given thread overhead; see below).

A definite gain for threading would be in approximate string matching. The target virus sequence is fixed, so there's no memory conflict issues. You could run millions of sequence fragments against the target sequence on multiple threads (e.g. with a 1000-core thread pool) to get much improved testing times. Each test sequence could be put in a queue which the threads take from whenever they are free (or wait if the queue is empty).

There are many reasons why you wouldn't get a 1000x speed boost when using a 1000-core supercomputer for these tasks:

1. Other users or processes may be using the physical processors, limiting the time slices scheduled for your threads on each processor.
2. Depending on your language and underlying architecture, your runtime may not implement many-to-many threading to processors. You may be funnelled into just a few cores.
3. Starting, maintaining and killing threads requires some CPU overhead time. This can be mostly mitigated with a thread pool, however.
4. Thread priorities could be interfering, particularly if another program is given high priority. This could cause starvation and deadlock, if your program can't get access to certain system resources.
5. Context switching takes time as the local cache memory and instruction pipeline for a core is flushed, in preparation for another program to switch onto that core.
6. If the task granularity is very fine, thread waking/starting/switching may take longer than the actual task to complete. This might be the case with retrieving ContactInfo from the data store, for example.
7. If you have implemented locking or synchronisation, for example with the data store HashMap, threads could start lining up for access, becoming more serial in behaviour.

My VirusApp code on GitHub: https://github.com/jakroth/VirusApp