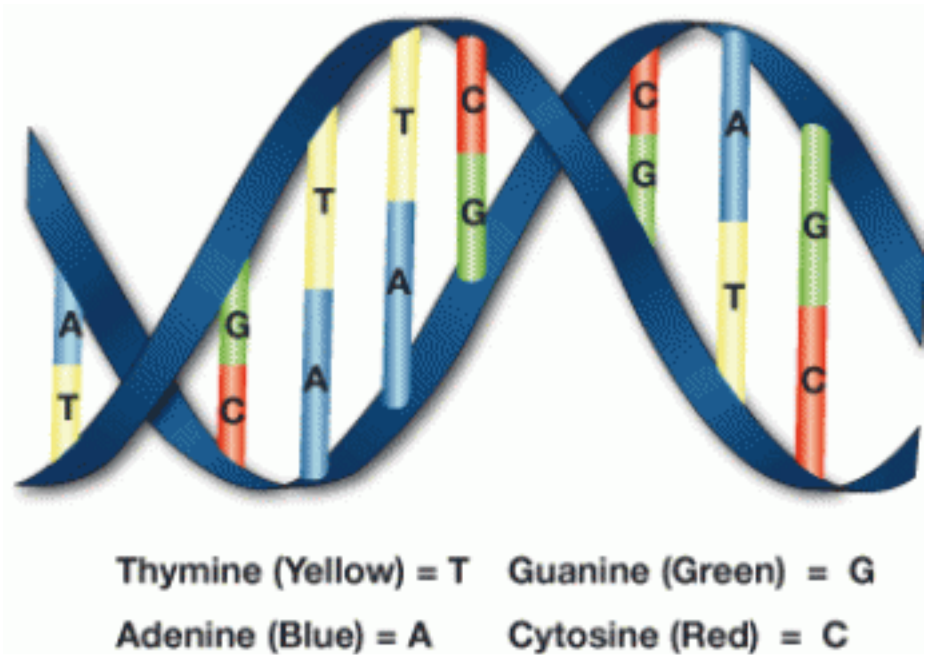
Project 4
Gee-nomics

Time due: 11 PM Thursday, March 14



Before writing a single line of code, you MUST first read AND THEN RE-READ the *Requirements and Other Thoughts* section.

Introduction

Did you read the text in red on the previous page?

Before writing a single line of code, you **must** first read **and then re-read** the *Requirements and Other Thoughts* section. Print out that page, tape it to your wall or on the mattress above your bunk bed, etc. And read it over and over (it's nearly as exciting as Carey's novel, The Florentine Deception).

The NachenSmall Software Corporation has been contacted by the U.S. Centers for Disease Control (CDC) and asked to create a software program to help its researchers better search through and compare the DNA of bacteria responsible for common illnesses. The thought is that if the CDCs researchers have better tools to process the DNA of various organisms, they can more quickly identify new bacterial strains and create cures.

Virtually all organisms on the face of the earth (except for perhaps RNA-based viruses) are DNA-based. DNA, which stands for deoxyribonucleic acid, is a chemical strand comprised of long sequences of just four different chemical units: adenine, cytosine, thymine, and guanine. These four chemical units (called *bases* or *nucleotides*) are usually referred to by their letter abbreviations, A, C, T and G. Organisms from viruses and bacteria to humans have long strands of DNA, comprised of thousands to billions of these simple DNA bases. Each organism is uniquely defined by its particular strand of DNA, and with the exception of identical twins and organisms that clone themselves, no two organisms share the same sequence of A, C, T and G bases.

The DNA of a particular organism is called its *genome*. Each genome is comprised of up to tens of thousands of different *genes*, and each gene is made up of hundreds or thousands of these individual A, C, T and G DNA bases. You can think of each gene as a paragraph made up of only As, Cs, Ts and Gs. Each gene is responsible for one or more aspects of the organism's biological processes such as the organism's eye color, its height, how the organism produces energy from its food sources, etc.

For example, here are the first 60 DNA bases (of the nearly 3 million total DNA bases) of the organism *Halobacterium jilantaiense*:

```
GGTTCTCAATGAATGGCAAGAGCTTCAACCCGACAACTCCGTCCTCACCGGCCGACAAT
```

Imagine how complex even a single-celled organism is if its genetic code is made up of nearly 3 million of these bases.

Different organisms (even of different species) often share many of the same or related genes. For example, humans and some ape species share up to 99% of the same DNA! Even humans and mice have many genes in common. This is important for a number of reasons:

- We can test drugs on an organism (e.g., a mouse) with similar genetics to ours and predict whether the drugs will be toxic or likely work in humans.
- We can deduce how a drug will work on one infection that is closely related to another infection.
- We can understand how two different types of organisms are related and how they may be related to a common ancestor.

While organisms of the same species share 99+% of the same basic genetic code, each individual has unique mutations in their genes that cause them to differ slightly from others. For example, David may have a slightly different sequence of As, Cs, Ts and Gs in a gene responsible for eye color than Carey, causing David's eyes to be brown and Carey's eyes to be green.

If two sequences of DNA bases differ by only a single DNA base, this is called a SNiP or *single nucleotide polymorphism*. For example, consider our original DNA sequence from *Halobacterium jilantaiense*:

```
GGTTCTCAATGAATGGCAAGAGCTTCAACCCGACAACTCCGTCCTCACCGGCCGACAAT
```

Here are four different SNiPs of our original sequence:

```
GGTTCTCTATGAATGGCAAGAGCTTCAACCCGACAACTCCGTCCTCACCGGCCGACAAT
GGTTCTCAATGAATGGCAAGAGCCTCAACCCGACAACTCCGTCCTCACCGGCCGACAAT
GGTTCTCAATGAATGGCAAGAGCTTCAACCCGACAACTCCGTCCTCACCTGCCGACAAT
GGTTCTCAATGAATGGCAAGAGCTTCAACCCGACGACTCCGTCCTCACCGGCCGACAAT
```

As you can see, each of the four sequences above differ by just one base from our original DNA sequence. A SNiP in a gene responsible for hair color might change one's hair from brown to blonde, or a SNiP in a gene responsible for metabolism might impact how quickly you put on weight. Or in many cases, a SNiP might have no observable impact on an organism.

What Do You Need to Do?

In this project, you're going to build three classes that can be used to process genetic data. These three classes will be used together to enable the following use cases:

- Maintain a library of genomes from multiple organisms; the user can add new genomes to this library.
- Allow the user to search the library for a specified DNA sequence and identify all genomes in the library that contain that DNA sequence or any SNiP of it.
- Allow the user to present the genome/DNA of a new organism and quickly identify all genomes in the library that have a high percentage of matching DNA with the presented genome.

Here is an overview of the classes you must build:

Class #1: You need to build a class named *Trie* that implements a templated, Trie-based multi-map:

You need to create a new template-based multi-map class, based on the *Trie* data structure (which we'll explain in more detail in the *Trie* section below). It can be used to map any C++ string of characters to one or more values (of the templated type). Unlike the STL map and STL multi-map which require you to specify the types of both the key and the value, as in:

```
std::multimap<string, int> someVariable;
```

the type of the key in our Trie-based multi-map is always (implicitly) a C++ **string**:

```
Trie<int> someVariable;  // Like std::multimap<string, int> above
```

The challenging thing about this multi-map class is that when the user searches for a key string (e.g. "ACTGGTCA"), not only must the object provide all values associated with the exact, searched-for key string, but it **must** also be able to return all values associated with any string in the multi-map which differs by at most one character from the searched for key string. So, when searching a *Trie,* the user can request that the class return either (a) only exact matches, or (b) both exact matches AND matches that differ by exactly one character (except that the first character of the search string may not differ)!

So suppose our multi-map object holds the following mappings:

"ABCD" → {1, 2, 3}
"AXCD" → {4, 5}
"AXYD" → {6, 7}

If we were searching for exact matches, then:

- Searching for "ABCD" would return {1, 2, 3}
- Searching for "ABCE" would return {}

If we were searching in a mode that allows both exact matches and mismatches, then:

- Searching for "ABCD" would return the following values: {1, 2, 3, 4, 5} since our search term matches ABCD exactly (yielding 1,2, and 3), and differs from AXCD by exactly one letter (yielding 4 and 5). Our search term is two letters away from AXYD so its results would not be returned.
- Searching for "AXYQ" would return the following values: {6, 7} since our search term differs from AXYD by exactly one character, but differs from the other strings in the multi-map by at least two characters.

- Searching for “AXCD” would return the following values: {1, 2, 3, 4, 5, 6, 7} because our search term exactly matches AXCD and mismatches by exactly one character the other two strings. So results from all three keys would be returned.
- Searching for “QBCD” would match none of the above strings, because we require that the *first* character of the searched-for string must always match exactly (no mismatches are allowed on the first character).

You **must not** use any STL containers other than the STL string, vector, or list classes to implement your *Trie* class.

Class #2: You need to build a class named *Genome*:

You’ll have to build a class named *Genome* that can be used to load organisms' genomes from a data file and, once loaded, allow a user of the class to (a) obtain an organism’s name, and (b) extract any subsequence of an organism’s DNA genome.

The user can call a function declared in the *Genome* class to load genome data from a specially-formatted genomic data file (we’ll provide a number of these data files for you to test with). The genomic data file contains both the name of the organism (e.g., Ferroglobus_placidus) and DNA sequences of the organism, consisting of millions of As, Cs, Ts and Gs. The function returns a vector of *Genome* objects, one for each DNA sequence specified in the file.

Once the user has loaded the genomic data into *Genome* objects, the user of a *Genome* object can ask the object to extract various subsequences of the DNA from that genome for processing. For example, let’s assume you have *Genome* object into which was loaded a genome like this:

ATAGGTACACATATGTATATATATATATATA…*3 million more bases...*

You could then ask the *Genome* object to give you the DNA sequence starting at position 3 that is 12 bases long (positions start at 0). Your object would return “GGTACACATATG”.

The object also enables its user to request the name of the organism the genome is associated with, e.g., “Ferroglobus_placidus”.

We’ll provide you with genome data files in a pre-defined format and tell you how to open, read and interpret the DNA data from those files to enable you to implement this class.

You **may** use any STL container classes you like to implement your *Genome* class.

Class #3: You need to build a class named *GenomeMatcher*:

You'll build a class called *GenomeMatcher* that maintains a library of *Genome* objects and allows the user of the class to:

- Add a new organism's *Genome* to the library
- Search all the genomes held in the library for a given DNA sequence, e.g. "Find the names of all genomes in the library whose DNA sequence contains 'ACCATGGATTACA' or some SNiP of that sequence such as ACCATGAATTACA', and tell me at what offset(s) these located sequences were found in each genome."
- Search the library to identify all genomes in the library whose DNA contains at least T% overlap with a given *Genome*, where the threshold T is specified by the user (e.g., T = 15%). This might be used to identify organisms whose genomes are closely related to a queried organism's genome.

You **may** use any STL container classes you like to implement your *GenomeMatcher* class. **However, your *GenomeMatcher* class MUST use your *Trie* class to index and search through the DNA bases of the genomes in its library.**

What Will We Provide?

We'll provide you with a number of text files containing genomes of common archaea[1].

We'll also provide you with a simple main driver program in file *main.cpp* which allows you to run some simple tests on your project as you build it.

Finally, we'll provide you with a header file named *provided.h* that will declare and implement the two classes *Genome*, and *GenomeMatcher*. Stop and think about that sentence. Is it saying that we're writing most of this project's code for you? Well, no.

Here's what we're doing in a nutshell:

1. We provide a *Genome* class in *provided.h* and all it does is forward all method calls you make to it to a class called *GenomeImpl* that you will write.
   a. Our Genome class holds a pointer to a *GenomeImpl* object as its only data member.
   b. When the user calls a method like *Genome::name()*, our *Genome* class's *name()* method just forwards that call to the *name()* function in the *GenomeImpl* class that you write.

[1] Archaea are single-celled organisms with no cell nucleus. Archaea are one of the three domains of life, the other two being Eukarya (which includes animals, plants, fungi, and protozoa) and Bacteria.

2. We provide a *GenomeMatcher* class in *provided.h* and all it does is forward all method calls you make to it to a class called *GenomeMatcherImpl* that you will write.
    a. Our *GenomeMatcher* class holds a pointer to a *GenomeMatcherImpl* object as its only data member.
    b. When the user calls a method like *GenomeMatcher::findGenomesWithThisDNA()*, our *GenomeMatcher* class's *findGenomesWithThisDNA()* method just forwards that call to the *findGenomesWithThisDNA()* function in the *GenomeMatcherImpl* class that you write.

So, to summarize, the code in each of these provided classes simply delegates work to a corresponding class that you will write. Let's see how this will work, and then explain why we're doing it this way.

The *provided.h* file that we give you will have the following code (edited to make this example simpler) and that you MUST NOT modify:

```
class GenomeImpl;

class Genome
{
public:
    Genome(const std::string& nm, const std::string& sequence);
    ~Genome();
    ...
    std::string name() const;
    bool extract(int position, int length, std::string& fragment) const;
    ...
private:
    GenomeImpl* m_impl;
};
...
```

Notice how our provided *Genome* class declaration in *provided.h* contains only one data member, a pointer to a *GenomeImpl* object, which is a class that you must write yourself in the *Genome.cpp* file; you'll similarly implement a *GenomeMatche* class in of a file named *GenomeMatcher.cpp*. You will submit the *Genome.cpp* and *GenomeMatcher.cpp* files as part of your solution to this project.

Below is a skeleton version of *Genome.cpp* that you will need to modify and turn in; you'll do the same thing with a *GenomeMatcher.cpp* file. Notice that the skeleton file has two parts: The first part is the definition of your new *GenomeImpl* class that you will write. The second part of the file contains the implementations of our *Genome* class member functions, which you shouldn't modify. Our *Genome* member functions simply delegate their work to your corresponding *GenomeImpl* functions[2]:

[2] This is an example of what is called the pimpl idiom (for "**p**ointer-to-**impl**ementation").

```
  // Genome.cpp file, skeleton version that we provide
#include "provided.h"
#include <string>
using namespace std;

// Part #1: You must modify the code below to implement your GenomeImpl class.
// You may add additional headers, using statements, functions (including a
// destructor if necessary), classes, etc.

class GenomeImpl
{
public:
    GenomeImpl(const std::string& nm, const std::string& sequence);
    ...
    string name() const;
    bool extract(int position, int length, string& fragment) const;
    ...
      // You may implement this however you want, provided that Genome objects
      // behave as required by our spec.
};

// Part 2: Implementation of our provided Genome class that delegates everything
// to GenomeImpl. You shouldn't change anything below this line.

Genome::Genome(const std::string& nm, const std::string& sequence)
{
    m_impl = new GenomeImpl(nm, sequence);  // create a new implementation object
};

Genome::~Genome()
{
    delete m_impl;  // destroy the implementation object
}

...

string Genome::name() const
{
      // delegate work to the implementation object
    return m_impl->name();
}

bool Genome::extract(int position,
                     int length,
                     string& fragment) const
{
      // delegate work to the implementation object
    return m_impl->extract(position, length, fragment);
}

...
```

There's an important restriction we will place on you: Other than *Genome.cpp* itself, no source file that you turn in may contain the name *GenomeImpl* class or that of any new helper functions/classes/structs you might introduce in your *Genome.cpp* class. So some other file whose code wants to use a genome object **must not** have:

```
void f()
{
    GenomeImpl g(..., ...);  // No! Other files must not use the name
                             // GenomeImpl
    ...
}
```

but instead should have:

```
void f()
{
    Genome g(..., ...);
    ...
    string s = g.name();
    ...
}
```

When the second snippet of code above constructs a *Genome* object, the *Genome* class code that we provide creates a *GenomeImpl* object for itself and has code you write construct yet. When the above code calls *Genome::name()*, the code that we provide in *Genome::name()* calls the *GenomeImpl::name()* function that you write and returns the value that your *GenomeImpl::name()* function returns. So all that our class does is call your class's methods!

Why are we having you implement a separate *GenomeImpl* class instead of just implementing the *Genome* class directly? And why can't your other code use your *GenomeImpl* class directly, but instead must use our *Genome* class to delegate work to your *GenomeImpl* class? For two reasons: one that restricts you and one that helps you.

How this restricts you: We want to make sure you don't introduce a public function into one of the required classes, e.g. *Genome*, that you then call in another, e.g. *GenomeMatcher*; we want you to use the interface we provided. By having none of your implementation go in *Genome*, we can put the *Genome* class in a file *provided.h* that you will not turn in. We will build our tester using the *provided.h* we gave you, so it's useless for you to modify it, since we'll never use or even see those modifications. All of your implementation related to *Genome* will go into *Genome.cpp*. By forbidding you to mention *GenomeImpl* outside of *Genome.cpp* (which is easy for us to detect), code in other files can't use a *GenomeImpl* (whose interface you might have extended) instead of a *Genome*. (Of course, your not being able to change the interface for *Genome* means you can't introduce changes that might keep our test framework from being able to work with your code.)

How this helps you: Suppose you cannot figure out how to write a correct *GenomeImpl* class and your implementation has some bugs. If your *GenomeMatcherImpl* class would work correctly if it used a correctly implemented *Genome*, but fails with your incorrectly implemented one, woud you lose points on *GenomeMatcher* too? No, because to test your *GenomeMatcherImpl* implementation, we will build *your* implementation together with *our* correct implementations of the other classes. This is much easier for us to do when a class's interface and implementation are separated the way we're doing it.

The *Trie* class template isn't subject to this separation, since template implementations aren't put in a separate .cpp file; they go in the header file. Your other classes will use the *Trie* class template directly; there's no need for a separate implementation class.

Details: The Classes You MUST Write

You **must** write correct versions of the following classes to obtain full credit on this project. Your classes **must** work correctly with our provided code, and you **must not** modify our provided *main.cpp* or *provided.h* to make them work with your code. Doing so will result in a **zero score** on that part of the project.

Trie Class Template

You must write a class template named *Trie* that implements a multi-map ADT using a trie data structure. You read that right – we said trie! A trie is a tree-based data structure which we will explain in more detail below. Your *Trie* class **must** be implemented using a trie to get credit for this part of the project.

Your *Trie* multi-map class template **must** have the following public interface:

```
template <class ValueType>
class Trie
{
public:
    Trie();
    ~Trie();
    void reset();
    void insert(const std::string& key, const ValueType& value);
    std::vector<ValueType> find(const std::string& key, bool exactMatchOnly) const;
};
```

You **must not** add any additional public member functions or data members to this class. You may add as many *private* member functions or data members as you like.

You **must not** use any STL containers other than the STL string, vector, or list classes to implement your *Trie* class.

With a standard STL *map* or *multimap,* when you instantiate the template you must specify both the type of the key and the type of the value, e.g.:

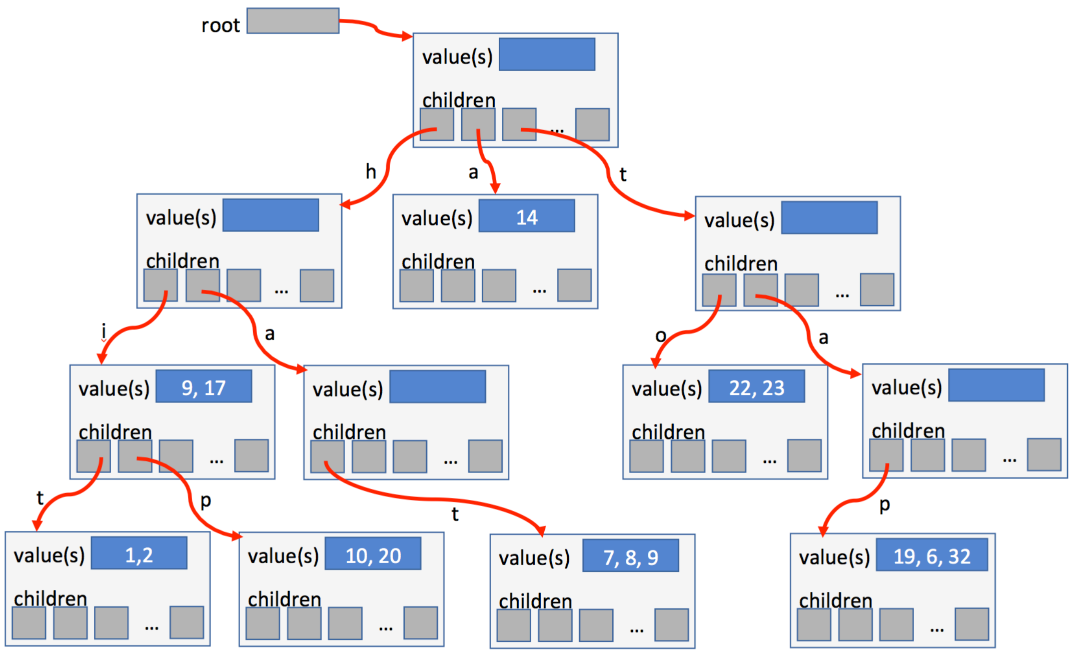
```
        std::multimap<std::string, int> myMap;   // maps strings to ints
```

In contrast, when you instantiate our *Trie* class template, you specify only the type of the value. The key is always implicitly an *std::string*. Here's an example of how you might use our *Trie* class template to define a variable:

```
    Trie<int> trie;  // This is like std::multimap<std::string,int> trie;
```

What is a Trie?

A trie is a special type of tree in which each node can have zero or more values and zero or more children (potentially hundreds of children). Unlike a regular tree node which simply contains a value and a pointer to each child, in a trie node, each child pointer comes with an associated label (e.g., typically a character like 'h' or 'a'):



When we search through a trie for a string, we search one character at a time by following the labeled child pointers from the root node until we reach the proper node in the trie, or find that a node does not have a child pointer with the proper label (which means that the string we're searching for was not in the trie).

For example, let's consider the diagram above. If we want to find the value(s) associated with the string "hip", we would start at the root node and take the child pointer labeled 'h'. This leads us to the leftmost node on the second row. From there, we'd take the child pointer labeled 'i' to the leftmost node on the third row. Finally, we'd take the child pointer labeled 'p' to the second node on the last row of the diagram. Here we reach the end of our searched-for string, and find that the values associated with "hip" are {10, 20}.

Imagine if we were to search instead for “at”. In this case, we’d again start at the root, and then take the child pointer labeled with an ‘a’ leading us to the second node on the second row. From here we’d like to find a child pointer labeled with a ‘t’ but no such child pointer exists, so we’ve hit a dead end and the word “at” is discovered not to be in our trie.

Finally, what if we search for the word “hi”. Again, we would start at the root node and take the child pointer labeled ‘h’. This leads us to the left-most node on the second row. From there, we’d take the child pointer labeled ‘i’ to the left-most node on the third row. Since we’ve hit the end of our searched-for string, we’ve found our matching node. This node contains the values {9, 17} associated with the string “hi”, even though the node is not a leaf node.

Tries are useful when we have a situation where many of our keys share the same common prefixes. For example, there are probably thousands of DNA sequences within a typical genome that begin with the string “**ACTAAG**” such as “**ACTAAG**AAT…”, “**ACTAAG**GTA…”, “**ACTAAG**TCA…” and so on. A trie data structure exploits this prefix repetition in order to reduce the amount of storage needed to store the complete set of data. In contrast, with a traditional binary tree, we’d end up repeatedly storing the same repetitive prefix string multiple times across multiple nodes.

There are a number of different ways to represent children (remember each child of a node has both a label and a pointer to a child node) in trie nodes, and you can use any approach that you like for this project. Some approaches optimize space at the expense of runtime, while other approaches optimize runtime at the expense of space. As with regular trees, you’ll use some sort of struct or class to represent your trie nodes.

Similarly, there are many ways to store the multiple values that might be held within a given trie node. For instance, you could have each trie node point to a (potentially empty) linked list of values, simply store a *vector* in each node, etc.

Trie() and ~Trie()

You **must** implement a constructor and destructor for your *Trie* class. It should create a root trie node with no children and no values, and set the root pointer (if any) to point to this root node. This method **must** run in O(1) time.

The destructor **must** free all memory associated with trie. This method **must** run in O(N) time where N is the number of nodes in the trie.

void reset()

Your trie’s *reset()* method **must** free all of the memory associated with the current trie, then allocate a new empty trie with a single root node (with no children or values). This method **must** run in O(N) time where N is the number of nodes in the trie.

void insert(const string& key, const ValueType& value)

The *insert()* method associates the specified *key* with the specified *value* in your trie, e.g. "GATTACA" → 42 by adding the appropriate nodes (if any are required) to the trie, and then adding the specified value (e.g., 42) to the existing set of values in the appropriate node. A given key inserted in the trie may map to one or more values, e.g., "GATTACA" → {42, 17, 32, 42, 42, 19, 17}, and *those values may include duplicates*. Here's how you might define a trie variable that maps strings to ints and insert a few items:

```
Trie<int> trie;  // This is like std::multimap<std::string,int> trie;

trie.insert("GATTACA", 42);  // GATTACA → {42}
trie.insert("GATTACA", 17);  // GATTACA → {42, 17}
trie.insert("GATTACA", 42);  // GATTACA → {42, 17, 42}
...
trie.insert("GCTTACA", 30);  // GCTTACA → {30}
```

There are a number of ways you can associate a given trie node with one or more values, including having a vector or list of values associated with each trie node. You can choose any approach you wish.

Your *insert()* function **must** run in O(LC) time where L is the length of the inserted key and C is the average number of children per node in your trie. With a clever approach, you may be able to get your *insert()* function to run in O(L) time, but this is not required.

Hint: Don't try to deal with mismatch-handling in your insert() method – just insert strings into your trie exactly as they are passed into your insert() method. Instead deal with searching for mismatches in your find() method.

std::vector<ValueType> find(const std::string& key,
bool exactMatchOnly)
const

The *find()* method is used to search for the values associated with a given key string, e.g., "GATTACA".

If *exactMatchOnly* is *true*, the *find()* method must return a vector containing all of the values associated with the exact *key* string specified. There is no required ordering of the returned values. If no values are associated with the specified key, then the *find()* method should return an empty vector.

In the tree depicted in the diagram several pages above, searching for "hit" with an *exactMatchOnly* of true would return values of {1, 2} in any order you choose:

```
std::vector<int> result1 = trie.find("hit", true);  // returns {1, 2} or {2, 1}
```

If *exactMatchOnly* is *false*, the *find()* method must return a vector containing all of the values associated with both the exact search term as well as all values associated with keys in the trie that:

1. Match the first character of the search term exactly, and
2. Have a single mismatching character (i.e., are a SNiP) of the specified key anywhere past the first character.

There is no required ordering of the returned values. If no values are associated with the specified key or SNiPs of that key, then the *find()* method should return an empty vector.

In the tree depicted in the diagram above, searching for "hit" with an *exactMatchOnly* of false would return values of {1, 2, 10, 20, 7, 8, 9} in any order you choose:

```
std::vector<int> result2 = trie.find("hit", false);  // returns {1, 2, 10, 20, 7, 8, 9}
```

Why {1, 2, 10, 20, 7, 8, 9}? The {1, 2} are associated with the exact match, "hit", the {10, 20} are associated with the SNiP of "hi**p**", and the {7, 8, 9} are associated with the SNiP of "h**a**t". (If instead of {7, 8, 9} "hat" were associated with {10, 2, 5, 10}, then the resulting vector would be {1, 2, 10, 20, 10, 2, 5, 10} in any order.

In the tree depicted in the diagram above, searching for "sit" with an *exactMatchOnly* of false would return an empty vector:

```
std::vector<int> result3 = trie.find("sit", false);  // returns {}
```

Why? Because no nodes in our trie begin with 's', and the *find()* method must only returns (a) exact matches or (b) SNiPs where the first letter of the searched for string is an exact match with some string in the trie. So even while "sit" is technically a SNiP of "hit" since the first letters differ, the results associated with "hit" in the trie would not be returned.

If *exactMatchOnly* is *true*, your *find()* function **must** run in O(LC+V) time where L is the length of the searched-for key, C is the average number of children per node in your trie, and V is the size of the returned vector. If *exactMatchOnly* is *false*, your *find()* function **must** run in O(L^2 C^2+V) time.

Hint: Recursion is your friend! With some clever coding you can easily find exact matches and single-character mismatches with the same code!

Genome Class

You must implement a class named *Genome* that can be used to hold an organism's complete genome and allow you to extract DNA subsequences from various positions in the genome. Your *Genome* class **must** have the following public interface:

```
class Genome
{
public:
       Genome(const std::string& nm, const std::string& sequence);
       ~Genome();  // Declare a destructor if you need to write one
       static bool load(std::istream& genomeSource, std::vector<Genome>& genomes);
       int length() const;
       std::string name() const;
       bool extract(int position, int length, std::string& fragment) const;
};
```

You **must not** add any additional public member functions or data members to this class other than a destructor, should you need one. You may add as many *private* member functions or data members as you like.

Genome(const std::string& nm, const std::string& sequence)

The *Genome* constructor must initialize a new *Genome* object with the indicated name and DNA sequence. You implementation may assume the sequence contains at least one character, and all characters in the sequence are upper case A, C, T, G, or N (we'll explain N later). It should run in O(S) time, where S is the length of the longer string.

~Genome()

If the compiler-generated destructor would not do the right thing, you must declare and implement a destructor. This spec imposes no requirements on its time complexity.

static bool load(std::istream& genomeSource, std::vector<Genome>& genomes)

The *Genome::load()* method is responsible for loading the genomic data from one of our provided genome text files. Notice that you pass in to this method an istream object, which is associated with a genomic data file. (For more information on what an *istream* is and how to open data files in C++, see the File I/O writeup on the class web site.) This function fills the vector second parameter with *Genome* objects, one for each genome specified in the data file.

This method is a *static member function*, which means that it is not generally called for a particular *Genome* object (e.g., `g.load(...)` or `gp->load(...)`), but instead is called for the class itself, on no particular object (i.e., `Genome::load(...)`). Static member functions are never passed a `this` pointer that would point to one particular object.[3] Static member functions are often used when you have some functionality that might otherwise be an ordinary global function, not a member of any class, that bears a strong relationship with one particular class. Loading genomes seems pretty tightly associated with the *Genome* type, so making it a static member function of *Genome* is appropriate.

[3] C++ actually allows you to say `g.load(...)` or `gp->load(...)`, but it ignores the `g` or `gp` (since there's no `this` pointer to pass it to) and acts as if you said `Genome::load(...)`.

Here's how a user might use your class's *load()* method:

```
#include <iostream>
#include <fstream>  // needed to open files
#include <string>
#include <vector>

void somefunc()
{
	  // Specify the full path and name of the gene data file on your hard drive.
	string filename = "c:/genomes/Ferroplasma_acidarmanus.txt";

	  // Open the data file and get a ifstream object that can be used to read its
	  // contents.
	ifstream strm(filename);
	if (!strm)
	{
		cout << "Cannot open " << filename << endl;
		return;
	}

	vector<Genome> vg;
	bool success = Genome::load(strm, vg);  // Load the data via the stream.

	if (success)
	{
		cout << "Loaded " << vg.size() << " genomes successfully:" << endl;
		for (int k = 0; k != vg.size(); k++)
			cout << vg[k].name() << endl;
	}
	else
		cout << "Error loading genome data" << endl;

}  // destructor for ifstream closes the file
```

Hint: Don't worry about the complexities of C++ input streams! The only input function your load() method needs is std::getline()!

Genomics Data File Format

Your *load()* method must load genetic data from a genomics data file in the FASTA format universally used in the bioinformatics field. The customary filename suffix for these files is *.fna* (for FASTA Nucleic Acids), but for our data files, we replaced *.fna* with *.txt* for ease of use on some operating systems.

A FASTA file is a text file that contains genetic information for an organism, with one or more entries for portions of that organism's DNA (often each corresponding to a gene); the information for each portion is a name (e.g., *Halorubrum chaoviator NODE 103*) and a long sequence of DNA bases (made up of As, Cs, Ts and Gs) for that organism. These genomics data files are produced by research labs around the world.

You can find hundreds of FASTA files documenting hundreds of different genomes on ftp://ftp.ncbi.nlm.nih.gov/genomes/refseq/bacteria/. We've provided eight of them for you to work with (which should be enough for you to get the project working, so there's no need to go to the website unless you're curious).

Should you go to the website, you can browse the various folders on the site for all files with an extension of *.fna.gz, e.g., Ferroglobus_placidus.fna.gz*. (Note: You might have to click down a few folder levels to find these fna.gz files.) The "gz" means that the file is compressed to save disk space. You must first decompress these gz files (e.g., using a decompression tool like Winzip in Windows, or just by double clicking on the file on your Mac desktop) and this will produce the full genome file, e.g., *Ferroglobus_placidus.fna*, which is the file containing the raw genetic information for each organism.

FASTA files have the following general format:

```
>One genome name
Up to 80 DNA bases, e.g., GGTTCTCAATGAATGGCAAGAGCTTCAACCCGACAACTCCGTCCTCACCGGCCGACAATCGTTCGCGCAGTCCCAACACG
Next up-to-80 DNA bases, e.g.,  ACCGAATCGTCGGGGGAGAGATACTCGATTCCCTGATGATCCATCGCGAATTCCCGGGAGAAGGTTACGGGTCCACGCTG
Next up-to-80 bases
…  // thousands of more lines of up to 80 bases
>Another genome name
Up to 80 bases
Next up-to-80 bases
Next up-to-80 bases
…  // thousands of more lines of up to 80 bases
>Yet another genome name
Up to 80 bases
Next up-to-80 bases
…  // thousands of more lines of up to 80 bases
```

The file consists of one or more occurrences of

1. A line starting with a greater-than sign (>) with the name, followed by
2. One or more lines consisting of between 1 and 80 upper or lower case letters A, C, T, G, or N (we'll explain N later).

Here's an actual example from the FASTA file for *Halobacterium jilantaiense*, which we provide to you in our project zip file:

```
>NZ_FOJA01000002.1 Halobacterium jilantaiense strain CGMCC 1.5337, whole genome shotgun sequence
GGTTCTCAATGAATGGCAAGAGCTTCAACCCGACAACTCCGTCCTCACCGGCCGACAATCGTTCGCGCAGTCCCAACACG
ACCGAATCGTCGGGGGAGAGATACTCGATTCCCTGATGATCCATCGCGAATTCCCGGGAGAAGGTTACGGGTCCACGCTG
GGTTTCGGCACCAGTCTCTGAGAGCGACGCTGGCAGGAAGACATCATACAGCTGGTTACCACGCTTCTCAAGGCGCCCAC
CGAAAGCTTCCAACCCCCGCTCGACAAAGGCCCGGATATCGGCTTCCGTCCCAAACACGTCGGCCGATTCATCCATTATA
TCCTGAATCCGCTCGCGACTCTCAGCGTCGAACGTGCTTGGATCGATCAGGCTACGGTCGTACCACTCAAGCAGAGTCTG
CTCACGCTCTTCCATGAGCTCCTCTAACTCCGCCGCGGTCGCTTCGGGGGGCTCCTCGTTCTGGAGCGACCGCATGATCA
GGTCGTCGATGTTGATGTCGTCAAGCATCCCCAGCACGTCTGCGGTCGACCCAACCTGGGAGCGAATATTCTCGACCTTG
// thousands more lines just like those above
>NZ_FOJA01000001.1 Halobacterium jilantaiense strain CGMCC 1.5337, whole genome shotgun sequence
ATTTTGCCGCCGCTCCAGCAGTTCGGCATCATCACCGGACTCACCATCATCTACGCGTTCCTCGCCAGCGTGCTCGTACT
CCCCAGCCTGCTCGTCATCTGGACGAGGTACCTCGGTCCGAGCGTCGACGAGTCCACGACGAACGTCGACGCAGCGACGC
CCACACCGGAGGACTGACCGATGGACGAAACTGACGCGATAGACGCGATGGGGGAACTCGGCCTCACCCAGTACGAGGCC
CAGGTGTTCATCGCGCTCCAGCAGTTGGGTGTCGCGTCAGCGAGCGAAATCGGTCAGGCGACCGACGTTCCCCGGTCGCA
GGTGTACGGCACCGCGGAGTCGCTCGAAGAACGCGGCCTCGTCGAAGTGCAGCAGTCGAACCCGATTCGGTACCGTCCGG
TCGGACTAGAGGAGGCACAGCAGACGCTCCGCGAGCAGTACGAGACGCACCAGCGCAACGCCTTCGACTACCTGGAGTCG
GTCCAGCAGCAGCCACACAGTGCGGAACAGCAAGAAGACATCTGGACCGTCCGCGGGTCGGACCACATCGACACGCGGGT
CGAGCAGCTGGCCGCCGACGCGACCGAGCGCGTCGTCTACGGCTGTGGCGGCGCGCTGTTCGACGAGCGGACCGCCGAGA
// thousands more lines just like those above
```

When your *load()* method runs, it must repeatedly:

1. Extract the genome name from a line in the file that begins with a greater than sign; everything following the greater-than sign (and excluding the newline at the end of the line) is the name, so for the line

   ```
   >NZ_FOJA01000002.1 Halobacterium jilantaiense strain CGMCC 1.5337, whole genome shotgun sequence
   ```

   the name would be

   ```
   NZ_FOJA01000001.1 Halobacterium jilantaiense strain CGMCC 1.5337, whole genome shotgun sequence
   ```

2. Extract the sequence of DNA bases for that name from the file into one concatenated string. The bases are on the lines following the name up to but not including the next line starting with a greater-than sign (or the end of the file). So if a file started with

   ```
   >Caenorhabditis elegans
   AGGATGCAGGAGAAATCCAGGCCCAGTAGCATTT
   TGTTCAGTAGCAAGATCAGCAAAC
   GGCAGCACCACACAGTGACTGAGATGTGAAA
   >Dictyostelium discoideum
   CCCATGCATACATACGCACAGCGCATTCAACGACTCAGCATCACAGCAAGGTTTAGTAT
   ...
   ```

   the concatenated sequence corresponding to the name Caenorhabditis elegans would be

   ```
   AGGATGCAGGAGAAATCCAGGCCCAGTAGCATTTTGTTCAGTAGCAAGATCAGCAAACGGCAGCACCACACAGTGACTGAGATGTGAAA
   ...
   ```

   In the concatenated sequence, every base letter must be upper case, even if the letter appeared in lower case in the file.

3. Create a *Genome* object with the extracted name and DNA sequence, and add it to the vector of *Genome*s that is the second parameter.

You'll notice that in addition to As, Cs, Ts and Gs, you'll also occasionally see 'N' characters in some genomes, e.g.:

```
GCTCGGNACACATCCGCCGCGGACGGGACGGGATTCGGGCTGTCGATTGTCTCACAGATCGTCGACGTACATGACTGGGA
```

The real meaning of the N character is that in this specific spot of the genome, any of the bases (A, C, T or G) may be found in typical organisms. For the purpose of this project, we'll simply treat N as if it were just another possible base, treating it no differently from A, C, T, or G.

If the FASTA file is properly formatted, this method must return *true*, and regardless of the orginal contents of the second vector parameter, the vector must end up containing all and only the *Genome* objects corresponding to the genomes specified in the file. If the FASTA file is improperly formatted, the function must return *false*, and the vector may be in any state you like: unchanged, empty, containing some of the genomes from the file, or whatever. Improper formatting includes things like not starting with a name line, non-name lines containing any characters other than upper or lower case A C T G N, no base lines after a name line, empty lines, or a line starting with a greater-than character but containing no other characters.

The method must run in O(N) time where N is the number of characters in the loaded genome file.

int length() const;

The *length()* method returns the complete length of the DNA sequence, specifically the total count of As, Cs, Ts, Gs (and Ns) found in the sequence held by the object. This method must run in O(1) time.

std::string name() const;

This method returns the name of the genome, e.g.,

```
NZ_FOJA01000001.1 Halobacterium jilantaiense strain CGMCC 1.5337, whole genome shotgun sequence
```

It must run in O(S) time, where S is the length of the name.

bool extract(int position, int length, std::string& fragment) const;

The *extract()* method must set *fragment* to a copy of a portion of the *Genome*'s DNA sequence: the substring *length* characters long starting at *position* (where the first character of the sequence is at position 0). For example, if a *Genome* object were created with an 80-base DNA sequence:

```
Genome g("oryx",
"GCTCGGNACACATCCGCCGCGGACGGGACGGGATTCGGGCTGTCGATTGTCTCACAGATCGTCGACGTACATGACTGGGA");
```

Then calling the *extract()* method would produce the following results:

```
string f1, f2, f3;
bool result1 = g.extract(0, 5, f1);  // result1 = true, f1 = "GCTCG";
bool result2 = g.extract(74, 6, f2); // result2 = true, f2 = "CTGGGA";
bool result3 = g.extract(74, 7, f3); // result3 = false, f3 is unchanged
```

The *extract()* method must return *true* if it successfully extracts a string of the specified length, and *false* otherwise (e.g., you try to extract a string that goes past the end of the genome); if extract returns *false*, *fragment* must remain unchanged.

This method must run in O(S) time where S is the length of the extracted sequence.

GenomeMatcher Class

A *GenomeMatcher* object is responsible for maintaining a library of genomes and allowing you to search through these genomes for DNA sequences, or identify genomes in the library that are related to a queried genome. Your *GenomeMatcher* class **must** have the following public interface:

```
class GenomeMatcher
{
public:
       GenomeMatcher(int minSearchLength);
       ~GenomeMatcher();  // Declare a destructor if you need to write one
       void addGenome(const Genome& genome);
       int minimumSearchLength() const;
       bool findGenomesWithThisDNA(const std::string& fragment,
                                   int minimumLength,
                                   bool exactMatchOnly,
                                   std::vector<DNAMatch>& matches) const;
       bool findRelatedGenomes(const Genome& query,
                               int fragmentMatchLength,
                               bool exactMatchOnly,
                               double matchPercentThreshold,
                               std::vector<GenomeMatch>& results) const;
};
```

You **must not** add any additional public member functions or data members to this class other than a destructor, should you need one. You may add as many *private* member functions or data members as you like.

Your *GenomeMatcher* implementation **must** use your *Trie* class template in the implementation of all data structures that hold DNA sequences. It may use any other STL container classes so long as these are not used to hold DNA sequence data (i.e., strings of As, Cs, Ts, Gs, or Ns). It may also use any functions from <algorithm>.

GenomeMatcher(int minSearchLength)

The *GenomeMatcher* constructor takes a single argument which specifies the minimum length of a DNA sequence that the user can later search for within the genome library. (Your constructor should save this into a data member for later use.) For example, if the user were to pass in a value of six for the minimum search length, then the shortest DNA sequence the user could search for using the *findGenomesWithThisDNA()* method would be six bases long, e.g. "GATTAC". Attempting to search for shorter sequences like "GATTA" or "ACTG" would result in *findGenomesWithThisDNA()* returning *false*.

This method must run in O(1) time.

~GenomeMatcher()

If the compiler-generated destructor would not do the right thing, you must declare and implement a destructor. This spec imposes no requirements on its time complexity.

void addGenome(const Genome& genome);

The *addGenome()* method is used to add a new genome to the library of genomes maintained by your *GenomeMatcher* object. Once a genome has been added, any time the user searches for a DNA sequence using the *findGenomesWithThisDNA()* method or searches for related genomes using the *findRelatedGenomes()* method, your *GenomeMatcher* object must search through the genomes in the library for matches.

This method must do two things:

1. Add the genome to a collection of genomes (e.g., a vector or list) held by the *GenomeMatcher* object.
2. Index the DNA sequences of the newly-added genome by adding ***every*** substring of length *minSearchLength* (the value that was passed into your constructor) of that genome's DNA sequence into a *Trie* maintained by the *GenomeMatcher*.

Every *GenomeMatcher* object **must** maintain a *Trie* that maps every DNA sequence of length *minSearchLength* from every added genome to a list of all those genomes that contain that sequence, and the position of every match in each such genome.

For example, imagine the user added the following (really short) genomes using the *addGenome()* method:

Genome 1: ACTG
Genome 2: TCGACT
Genome 3: TCTCG

And further assume that *minSearchLength* was set to 3 during construction. Then your *addGenome()* method would put the following strings into its *Trie*.

For Genome 1:

ACT → (Genome 1, position 0)
CTG → (Genome 1, position 1)

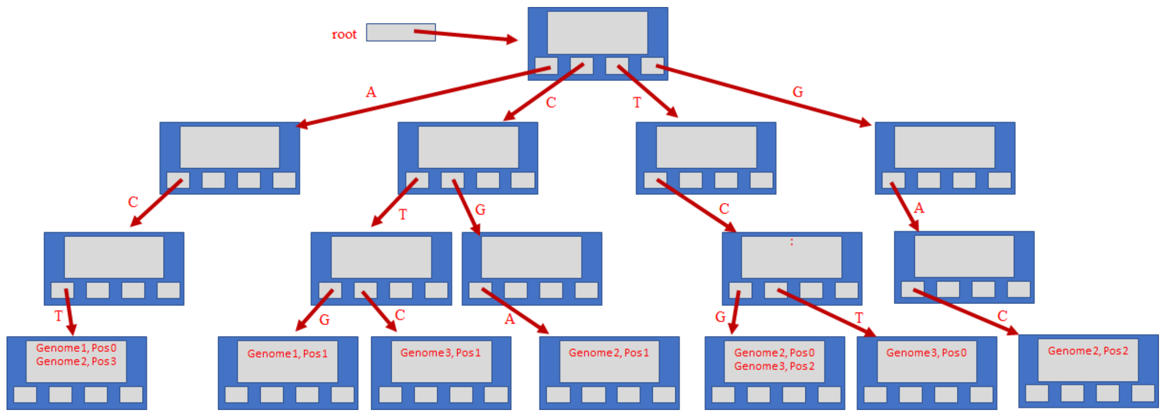
For Genome 2:

TCG → (Genome 2, position 0)
CGA → (Genome 2, position 1)
GAC → (Genome 2, position 2)
ACT → (Genome 2, position 3)

For Genome 3:

> TCT → (Genome 3, position 0)
> CTC → (Genome 3, position 1)
> TCG → (Genome 3, position 2)

And your completed *Trie* would look like this:



This method must run in O(L* N) time where L is the *GenomeMatcher*'s *minSearchLength* and N is the length of the added *Genome*'s DNA sequence.

int minimumSearchLength() const;

This method must return the minimum search length passed to the constructor so the user of the class can determine the minimum length of strings that can be searched for. It must run in O(1) time.

```
bool findGenomesWithThisDNA(const std::string& fragment,
                            int minimumLength,
                            bool exactMatchOnly,
                            std::vector<DNAMatch>& matches) const;
```

The *findGenomesWithThisDNA()* method is used to find all genomes in the library that contain a specified DNA *fragment* (e.g., "GATTACA"), or potentially one or more of its SNiPs (e.g. "GCTTACA", "GATTATA"), which are *minimumLength* or more bases long. The method must pass back an STL *vector* of the *longest match* found from each matching genome, and if more than one match for the *fragment* is found in a particular genome, then only the longest match must be returned for that genome. If two or more matches from the same genome have the same length, then the match that is found earliest in the genome must be returned. This function is case-sensitive, so searching for "GaTtAca" will not match "GATTACA".

We say that a *match* exists between *fragment* and a segment of a genome's DNA sequence when

1. The segment in the genome is identical to the first N characters of *fragment*, for some N ≥ *minimumLength*, **or**
2. The *exactMatchOnly* parameter is *false*, and the segment in the genome is a SNiP of the first N characters of *fragment*, for some N ≥ *minimumLength*. For our purposes, one sequence is a SNiP of another if
   - they are the same length, and
   - except for one position, the corresponding characters at that position of the two sequences are identical, and
   - the position of the sole mismatch is not the first position (position 0).

   Examples: If *exactMatchOnly* is *false*, *minimumLength* is 4, and fragment is "ACTG", then "ATTG" would be a SNiP of that fragment, as would "ACTA" (the mismatch could occur at the last position). However, "ATTA" would not be a SNiP (more than one mismatch in the first *minimumLength* characters), and "TCTG" would not be (the mismatch can't be in the first position).

The *findGenomesWIthThisDNA()* method must return *false* if

1. *fragment*'s length is less than *minimumLength*, or
2. *minimumLength* is less than the *minSearchLength* value passed to the GnomeMatcher's constructor, or
3. There are no matches between *fragment* and any segment of any genome in the *GenomeMatcher* object's library.

Otherwise, the method must return true (indicating that at least one match was found).

Note that the searched-for *fragment* might be longer than the *minSearchLength* used to build the *Trie* that holds your DNA prefixes. Your *findGenomesWithThisDNA()* method therefore MUST be able to return both exact (and potentially SNiP) matches that are longer than those held in the *Trie*, and SNiP mismatches may be found anywhere (except for the first DNA base of the *fragment*), not just in the portion of the match within the *Trie*. This means that you'll have to use your *Trie* to locate the proper locations in your *Genomes* where there may be potential matches (of just the *fragment*'s prefix of length *minSearchLength*), but then *extract()* more bases from each *Genome* from those location(s) to verify that you can match *minimumLength* or more characters.

For example, in the example above for *addGenome()*, Genome 2 is "TCGACT", and the trie shown in the diagram was built for a *minSearchLength* of 3. Suppose we call findGenomesWithThisDNA with *minimumLength* being 4, and *exactMatchOnly* being *false*. Then searching for CGACT, CTACT or CGACG should all return Genome 2, Position 1, since:

1. If *fragment* is "CGACT", we have an exact match of length 5, with "CGA" found in the trie as an exact match, and the remainder of the letters "CT" matching exactly.

2. If *fragment* is "CTACT", we have a SNiP of length 5 (with the mismatch on the second character), with “CTA” found in the trie (as a partial match with “CGA” in the trie), and the remainder of the letters “CT” matching exactly.
3. If *fragment* is "CGACG", we have a SNiP of length 5 (with the mismatch on the fifth character), with “CGA” found in the trie as an exact match, and the remainder of the letters “CG” being identical except in one position.

If the *findGenomesWithThisDNA()* method returns *true*, it must set the parameter *matches*, a *vector* containing exactly one *DNAMatch* struct for each of and only the genomes containing a match, where the *DNAMatch* struct looks like this:

```
struct DNAMatch
{
	std::string genomeName;
	int position;
	int length;
};
```

If *findGenomesWithThisDNA()* returns *false*, the *matches* vector may be in any state you like: unchanged, empty, or whatever.

This method must run in O(H*F) time, where F is the length of *fragment*, and H is the number of distinct hits across all genomes where the prefix of length *minSearchLength* of *fragment* (or a SNiP of the *fragment*, if *exactMatchOnly* is *false*) was found. So if *fragment* were “ACTGTTTT”, F would be 8. If, when we searched our trie (supposing that minSearchLength were 4), we found that the prefix sequence of length 4, “ACTG”, existed at 3 different places in Genome 1, at 10 different places in Genome 2, and at 5 different places in Genome 3, then H would be 18 (3+10+5).

Given the following *Genome*s added to a *GenomeMatcher* object with a *minSearchLength* value of 4 (meaning your *Trie* only indexes sequences of length 4):

```
Genome 1: CGGTGTACNACGACTGGGGATAGAATATCTTGACGTCGTACCGGTTGTAGTCGTTCGACCGAAGGGTTCCGCGCCAGTAC
Genome 2: TAACAGAGCGGTNATATTGTTACGAATCACGTGCGAGACTTAGAGCCAGAATATGAAGTAGTGATTCAGCAACCAAGCGG
Genome 3: TTTTGAGCCAGCGACGCGGCTTGCTTAACGAAGCGGAAGAGTAGGTTGGACACATTNGGCGGCACAGCGCTTTTGAGCCA

          01234567890123456789012345678901234567890123456789012345678901234567890123456789
                    1         2         3         4         5         6         7
```

Our *findGenomesWithThisDNA()* method would return the following results:

```
std::vector<DNAMatch> matches;
bool result;

result = findGenomesWithThisDNA("GAAG", 4, true, matches);
```

result: true; matches:
Genome 1 of length 4 at position 60
Genome 2 of length 4 at position 54
Genome 3 of length 4 at position 29

```
result = findGenomesWithThisDNA("GAATAC", 4, true, matches);
```

result: true; matches:
 Genome 1 of length 5 at position 22
 Genome 2 of length 5 at position 48

```
result = findGenomesWithThisDNA("GAATAC", 6, true, matches);
```

result: false; matches: none

```
result = findGenomesWithThisDNA("GAATAC", 6, false, matches);
```

result: true; matches:
 Genome 1 of length 6 at position 22
 Genome 2 of length 6 at position 48

```
result = findGenomesWithThisDNA("GTATAT", 6, false, matches);
```

result: true; matches:
 Genome 1 of length 6 at position 22
 Genome 2 of length 6 at position 48

```
result = findGenomesWithThisDNA("GAATACG", 6, false, matches);
```

result: true; matches:
 Genome 1 of length 6 at position 22
 Genome 2 of length 7 at position 48

```
result = findGenomesWithThisDNA("GAAGGGTT", 5, false, matches);
```

result: true; matches:
 Genome 1 of length 8 at position 60
 Genome 3 of length 7 at position 35
 Genome 2 of length 5 at position 54

```
result = findGenomesWithThisDNA("GAAGGGTT", 6, false, matches);
```

result: true; matches:
 Genome 1 of length 8 at position 60
 Genome 3 of length 7 at position 35

```
result = findGenomesWithThisDNA("ACGTGCGAGACTTAGAGCC", 12, false, matches);
```

result: true; matches:
 Genome 2 of length 19 at position 28

```
result = findGenomesWithThisDNA("ACGTGCGAGACTTAGAGCG", 12, false, matches)
```

result: true; matches:
 Genome 2 of length 19 at position 28

```
result = findGenomesWithThisDNA("GAAG", 3, true, matches);
```

result: false; matches: none

```
result = findGenomesWithThisDNA("GAAG", 5, true, matches);
```

result: false; matches: none

```
bool findRelatedGenomes(const Genome& query,
                        int fragmentMatchLength,
                        bool exactMatchOnly,
                        double matchPercentThreshold,
                        std::vector<GenomeMatch>& results) const;
```

The *findRelatedGenomes()* method compares a passed-in *query* genome for a new organism against all genomes currently held in a *GenomeMatcher* object's library and passes back a *vector* of all genomes that contain more than *matchPercentThreshold* of the base sequences of length *fragmentMatchLength* from the *query* genome. It returns *true* if one or more genomes in the library were close enough matches, and *false* if no close matches were located. The method also must return false if the value *fragmentMatchLength* is less than the value of *minSearchLength* passed into the *GenomeMatcher* constructor.

The method **MUST** use the following algorithm to get full credit:

We will consider sequences of length *fragmentMatchLength* from the *query* genome starting at positions 0, 1* *fragmentMatchLength*,2* *fragmentMatchLength*, etc. (e.g., if *fragmentMatchLength* were 12, the start positions would be 0, 12, 24, 36). If the length of the *query* genome is not a multiple of *fragmentMatchLength*, we ignore the final sequence that is shorter than *fragmentMatchLength*. Let S be the number of sequences we will consider. For example, if the *query* genome were 800 bases long and *fragmentMatchLength* were 12, then since 800/12 is 66.6667, S will be 66 (since we ignore the final 8 base long sequence).

For each such sequence:

1. Extract that sequence from the queried genome.
2. Search for the extracted sequence across all genomes in the library (using *findGenomesWithThisDNA()*), allowing SNiP matches if *exactMatchOnly* is *false*).
3. If a match is found in one or more genomes in the library, then for each such genome, increase the count of matches found thus far for it.

For each genome g in the library that contained at least one matching sequence from the query genome:

1. Compute the percentage p of sequences from the query genome that were found in genome g by dividing the number of matching sequences found in genome g by S (e.g., if S is 66, and 15 of the 66 sequences were found

in the genome, then 15/66 or 22.73% of the sequences from the *query* genome were found in that genome, so p will be 22.73).

2. If p is greater than or equal to the *matchPercentThreshold* parameter (a percentage in the range 0 though 100), then g is a matching genome.

Return from the function with the *results* vector containing all matching genomes, ordered in descending order by the match proportion p, and breaking ties by the genome name in ascending alphabetical order. The results vector contains *GenomeMatch* objects, where the GenomeMatch structure looks like this:

```
struct GenomeMatch
{
	std::string genomeName;
	double percentMatch;	// 0 to 100
};
```

This method must run in O(Q * X) time, where Q is the length in DNA bases of the *query* sequence (e.g., 3 million bases), and X is the function in the big-O of your *findGenomesWithThisDNA()* method.

Test Harness

We have graciously provided you with a simple test harness (in *main.cpp*) that lets you test your overall Gee-nomics implementation. You can compile this *main.cpp* file with your source files to build a complete working test program. You can then run this program from the Windows command line or the shell running in a macOS Terminal window or under Linux.

Our test harness lets you:

- Add a genome a *GenomeMatcher*'s library by typing a name and a DNA sequence.
- Load a genome data file into the library
- Load all of our demonstration genomes into the library at once to save you time ☺
- Search for an exact DNA match like CGTTAGAG without any mismatching bases
- Search for an DNA match like CGTTAGAG allowing one mismatching base, e.g., searching for CGTTAGAG could match CGTTAGGG within a genome
- Type a DNA sequence and identify all genomes in the library that are close matches of that genome.
- Specify a genome data file and identify all genomes in the library that are close matches to that genome.

When you run our test harness, it might look like this:

```
Welcome to the Gee-nomics test harness!
The genome library is initially empty, with a default minSearchLength of 10
        Commands:
         c - create new genome library      s - find matching SNiPs
         a - add one genome manually        r - find related genomes (manual)
         l - load one data file             f - find related genomes (file)
         d - load all provided data files   ? - show this menu
         e - find matches exactly           q - quit
Enter command: a
Enter name: yeti
Enter DNA sequence: ACGTACGTAAAACCCCGGGGTTTTNANANANANA
Enter command: e
Enter DNA sequence for which to find exact matches: AAAACCCCGGGGTTNN
Enter minimum sequence match length: 12
1 matches of AAAACCCCGGGGTTNN found:
  length 14 position 8 in yeti
Enter command: e
Enter DNA sequence for which to find exact matches: CCCCAAAATTTT
Enter minimum sequence match length: 10
No  matches of CCCCAAAATTTT were found.
Enter command: s
Enter DNA sequence for which to find exact matches and SNiPs: AAAACCTCGGGGTTNN
Enter minimum sequence match length: 12
1 matches and/or SniPs of AAAACCTCGGGGTTNN found:
  length 14 position 8 in yeti
Enter command: c
Enter minimum search length (3-100): 4
Enter command: a
Enter name: sasquatch
Enter DNA sequence: GGGGTTTTAAAACCCCACGTACGTACGTNANANANA
Enter command: r
Enter DNA sequence: AAATCCCTGGGGTTTTNANA
Enter match percentage threshold (0-100): 20
Require (e)xact match or allow (S)NiPs (e or s): s
    1 related genomes were found:
  50.00%  sasquatch
Enter command: c
Enter minimum search length (3-100): 10
Enter command: d
Loaded 1 genomes from Ferroplasma_acidarmanus.txt
Loaded 2 genomes from Halobacterium_jilantaiense.txt
Loaded 105 genomes from Halorubrum_chaoviator.txt
Loaded 83 genomes from Halorubrum_californiense.txt
Loaded 55 genomes from Halorientalis_regularis.txt
Loaded 121 genomes from Halorientalis_persicus.txt
Loaded 1 genomes from Ferroglobus_placidus.txt
Loaded 1 genomes from Desulfurococcus_mucosus.txt
Enter command: e
Enter DNA sequence for which to find exact matches: ACGAATCACGTGCGAGA
Enter minimum sequence match length: 11
2 matches of ACGAATCACGTGCGAGA found:
  length 17 position 568 in NZ_AOJK01000080.1 Halorubrum californiensis DSM 19288
contig_80, whole genome shotgun sequence
  length 12 position 1977 in NZ_FOCX01000065.1 Halorientalis persicus strain IBRC-
M 10043, whole genome shotgun sequence
Enter command: q
```

Requirements and Other Thoughts

Make sure to read this entire section before beginning your project!

1. You should backup your code to a flash drive or an online repository like Dropbox or Google drive frequently (e.g., after successfully creating a new function).

   If you come to us and complain that your computer crashed and you lost all of your work, we'll ask you where your backups are.
2. In Visual C++, make sure to change your project from UNICODE to Multi Byte Character set, by going to Project → Properties → Configuration Properties → General → Character Set
3. No matter what you do, and how much you finish, make sure your project builds and at least runs (even if it crashes after a while). WHATEVER YOU DO, don't turn in code that doesn't build.
4. Whatever you do, DO NOT MODIFY OUR PROVIDED HEADER FILE IN ANY WAY! YOU WILL NOT TURN IT IN, SO ANY MODIFICATIONS WILL NOT BE SEEN BY OUR GRADING TOOL!
5. The entire project can be completed in less than 400 lines of C++ code beyond what we've already written for you, so if your program is getting much larger than this, talk to a TA – you're probably doing something wrong.
6. Be sure to make use of the C++ STL container classes ***AND*** algorithm functions where we permit it – it can make things much easier for you!
7. If you need to define your own comparison functions, feel free to do so! However you MUST place these functions within one of your own source files that you will submit as part of the project. You MUST NOT modify our provided header file!!
8. Before you write a line of code for a class, think through what data structures and algorithms you'll need to solve the problem. Plan before you program!
9. Don't make your program overly complex – use the simplest data structures possible that meet the requirements.
10. You MUST NOT modify any of the code in the files we provide you that you will not turn in; since you're not turning them in, we will not see those changes. We will incorporate the required files that you turn in into a project with special test versions of the other files.
11. Make sure to implement and test each class independently of the others that depend on it. Once you get the simplest class coded, get it to compile and test it with a number of different unit tests. Only once you have your first class working should you advance to the next class.
12. Try your best to meet our big-O requirements for each method in this spec. If you can't figure out how, then solve the problem in a simpler, less efficient way, and move on. Then come back and improve the efficiency of your implementation later if you have time.

If you don't think you'll be able to finish this project, then take some shortcuts. For example, implement the *Trie* class first by using the STL's map or unordered_map class to do all of the hard work for exact matching. Then use it to get your *GenomeMatcher* class working. Once you get your genome matcher working, then go back and implement your full *Trie* class.

You can still get a good amount of partial credit if you implement most of the project. Why? Because if you fail to complete a class (e.g., *Trie*), we will provide a correct version of that class and test it with the rest of your program. If you implemented the rest of the program properly, it should work perfectly with our version of the class you couldn't get working, and we can give you credit for those parts of the project you completed correctly.

But whatever you do, make sure that ALL CODE THAT YOU TURN IN BUILDS without errors with both g32 and either Visual C++ or clang++! For full credit, your code must run correctly under both g32 and either Visual C++ or clang++.

What to Turn In

You will turn in **four** files:

Trie.h	Contains your trie map class template implementation
Genome.cpp	Contains your Genome class implementation
GenomeMatcher.cpp	Contains your GenomeMatcher class implementation
report.docx, report.doc, or report.txt	Contains your report

You must submit a brief (You're welcome!) report that describes:

1. Whether any of your classes have known bugs or other problems that we should know about. For example, if you didn't finish the *GenomeMatcher::findRelatedGenomes()* method or it has bugs, tell us.
2. Whether or not each method satisfies our big-O requirements, and if not, what you did instead and what the big-O is for your version.
3. How two of your methods work — use high-level pseuocode to describe them:
   - *Trie*'s *find()* method
   - *GenomeMatcher*'s *findGenomesWithThisDNA()* method

Grading

- 95% of your grade will be assigned based on the correctness of your solution.
- 5% of your grade will be based on your report.

Good luck!