Assignment 2: Using Collection Classes

Swallowing a hundred rivers, as it is virtuous vast is the sea;
Towering thousands of feet high, as it is desireless the cliff's mighty.

— Couplet, *Lin Zexu* (1785-1850), Qing Dynasty

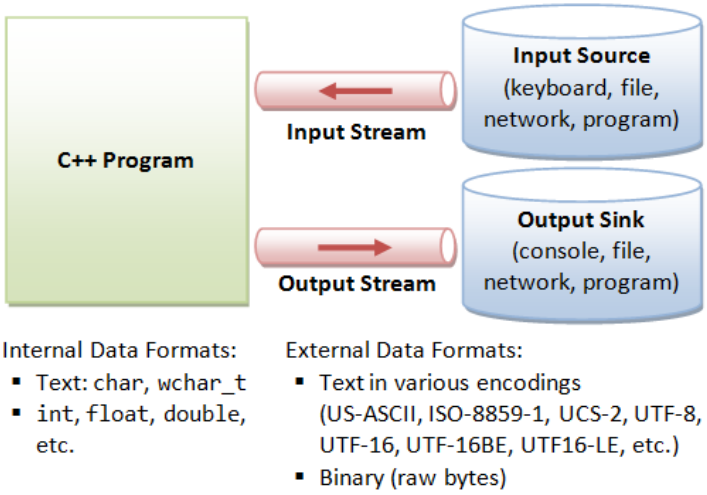
Now that you've been introduced to the handy Stanford C++ class library, it's time to put these objects to work! In your role as a client of these Abstraction Data Types (ADTs) the low-level details abstracted away, you can put your energy toward solving more interesting problems. In this assignment, your job is to write five short client programs that use these classes to do nifty things. The tasks may sound a little daunting at first, but given the power tools in your arsenal, each requires only a page or two of code. Let's hear it for abstraction!

The assignment has several purposes:

1. To let you experience more fully the joy of using powerful library classes.
2. To stress the notion of abstraction as a mechanism for managing data and providing functionality without revealing the representational details.
3. To increase your familiarity with using C++ class templates.
4. To give you some practice with classic data structures such as the stack, queue, vector, map, and lexicon.

Problem 0 Deduplication and Sorting of Data in File

C/C++ IO are based on streams, which are sequence of bytes flowing in and out of the programs (just like water and oil flowing through a pipe). In input operations, data bytes flow from an input source (such as keyboard, file, network or another program) into the program. In output operations, data bytes flow from the program to an output sink (such as console, file, network or another program). Streams acts as some intermediaries between the programs and the actual IO devices, in such the way that frees the programmers from handling the actual devices, so as to archive device independent IO operations.



C++ provides both the formatted and unformatted IO functions. In formatted or high-level IO, bytes are grouped and converted to types such as int, double, string or user-defined types. In unformatted or low-level IO, bytes are treated as raw bytes and unconverted. Formatted IO operations are supported via overloading the stream insertion (<<) and stream extraction (>>) operators, which presents a consistent public IO interface.

To perform input and output, a C++ program:

1. Construct a stream object.
2. Connect (Associate) the stream object to an actual I/O device (e.g., keyboard, console, file, network, another program).
3. Perform input/output operations on the stream, via the functions defined in the stream's pubic interface in a device independent manner. Some functions convert the data between the external format and internal format (formatted IO); while other does not (unformatted or binary IO).
4. Disconnect (Dissociate) the stream to the actual IO device (e.g., close the file).
5. Free the stream object.

Now there is a **text format file** (named as "***raw-data.txt***") containing several lines of integers. Your program can accomplish the following tasks:

Task 1: Read the integers in ***raw-data.txt*** sequentially, put them into a ***set***, and then iterate through the ***set*** using range-based loops. Output the integers to a binary file named as ***sorted-data.bin***, achieving the effect of deduplication and sorting.

Task 2: Read out the integers in ***sorted-data.bin***, find out the median and output to the screen.

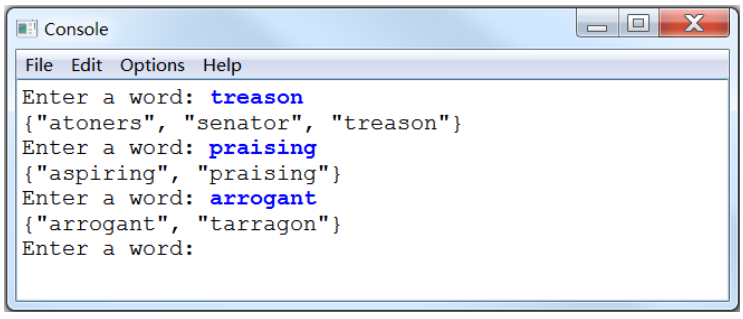
Note: Starter files contain ***raw-data.txt***, and your program needs to create ***sorted-data.bin***.

Problem 1 Anagram Clusters

Two words are ***anagrams*** of one another if the letters in one can be rearranged into the other. For examples:

"senator" and "treason"
"praising" and "aspiring"
"arrogant" and "tarragon"

Nifty fact: two words are anagrams if you get the same string when you write the letters in those words in sorted order. For example, "praising" and "aspiring" are anagrams because, in both cases, you get the same string as "aiignprs" if you sort the letters in the two words.



Write the program group all words in English into "clusters" of words that are all anagrams of one another.

You can use a Map<string, lexicon>. ***Each key is a string of letters in sorted order described in the slides 05 for Lecture. Each value is the collection of English words that have those letters in that order.***
The starter project for this problem includes a copy of the `EnglishWords.txt` file described in Lecture. Before you write the Anagram Clusters program, you might experiment with a simpler program that uses the lexicon in simpler ways. For example, you might write a program that prints out all the English words that by specific first letters.

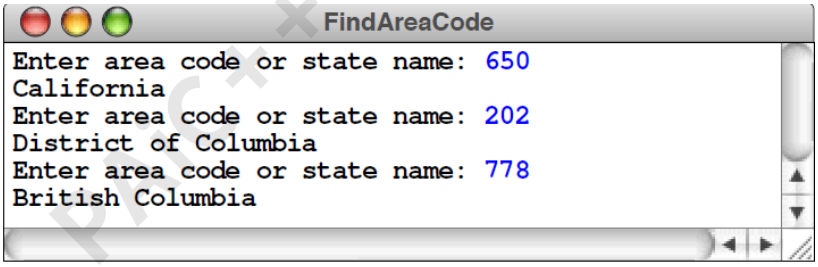
Problem 2 Area Codes

Telephone numbers in the United States and Canada are organized into various three-digit ***area codes***. A single state or province will often have many area codes, but a single area code will not cross a state boundary. This rule makes it possible to list the geographical locations of each area code in a data file. For this problem, assume that you have access to the file **AreaCodes.txt**, which lists all the area codes paired with their locations as illustrated by the first ten lines of that file:

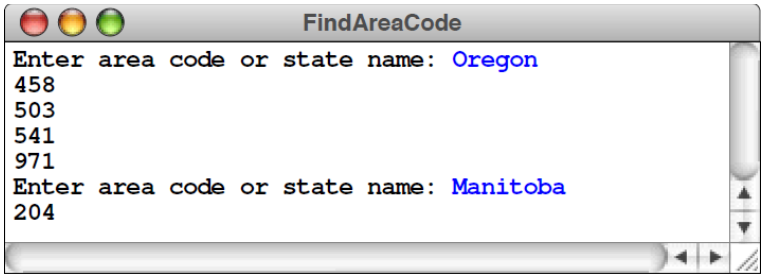
AreaCodes.txt

```
201-New Jersey
202-District of Columbia
203-Connecticut
204-Manitoba
205-Alabama
206-Washington
207-Maine
208-Idaho
209-California
210-Texas
```

Write the code necessary to read this file into a Map<int, string>, where the key is the area code and the value is the location. Once you've read in the data, write a main program that repeatedly asks the user for an area code and then looks up the corresponding location, as illustrated in the following sample run:



As the prompt suggests, however, your program should also allow users to enter the name of a state or province and have the program list all the area codes that serve that area, as illustrated by the following sample run:



Extension: When you wrote the **FindAreaCode** program for the previous exercise, it is likely that you generated the list of area codes for a state by looping through the entire map and printing out any area codes that mapped to that state. Although this strategy is fine for small maps like the area code example,

efficiency might become an issue in working with much larger collections of data. This strategy also feels uncomfortably asymmetric. When you want to translate an area code to a state name, you ask the map and it gives you the answer immediately; translating in the opposite direction requires a lot more work.

What you would like to do is invert the map so that you could perform lookup operations in either direction. You can't, however, declare the inverted map as a **Map<string, int>**, because there is often more than one area code associated with a state. An **int** can't hold all the necessary information. A better strategy is to make the inverted map a **Map< string, Vector<int> >** that maps each state name to a vector of the area codes that serve that state. Rewrite the **FindAreaCode** program so that it creates an inverted map after reading in the data file and then uses that map to list the area codes for a state.

Problem 3 Word Ladders

A ***word ladder*** is a connection from one word to another formed by changing one letter at a time with the constraint that at each step the sequence of letters still forms a valid word. For example, here is a word ladder connecting `"code"` to `"data"`.

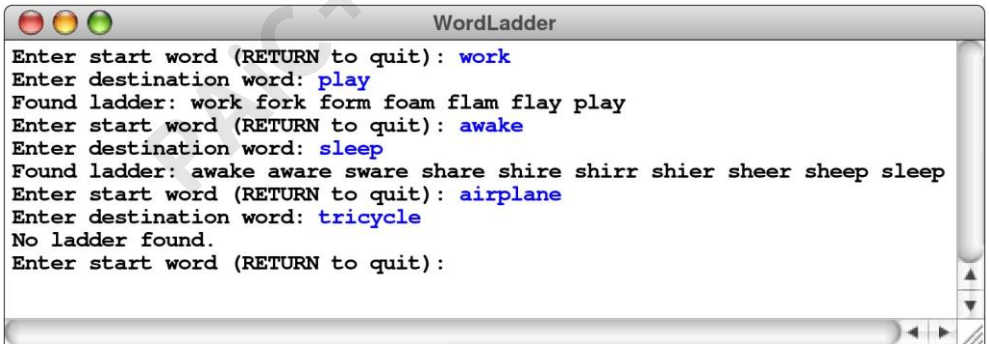
`code → core → care → dare → date → data`

That word ladder, however, is not the shortest possible one. Although the words may be a little less familiar, the following ladder is one step shorter:

`code → cade → cate → date → data`

Imagine the word ladder from `"boost"` to `"happy"`, the connections given by Donald E. Knuth is `"boost" → "boast" → "beast" → "least" → "leapt" → "leaps" → "leads" → "lends" → "lands" → "hands" → "handy" → "hardy" → "harpy" → "happy"`. It may not be the shortest one, but may be appropriate one for this year. (https://www-cs-faculty.stanford.edu/~knuth/news.html)

Your job in this problem is to write a program that finds a minimal word ladder between two words. Your code will make use of several of the ADTs from Chapter 5, along with a powerful algorithm called breadth-first search to find the shortest such sequence. Here, for example, is a sample run of the word-ladder program in operation:



A sketch of the word ladder implementation

Finding a word ladder is a specific instance of a ***shortest-path problem***, in which the challenge is to find the shortest path from a starting position to a goal. Shortest-path problems come up in a variety of situations such as routing packets in the Internet, robot motion planning, determining proximity in social networks, comparing gene mutations, and more.

One strategy for finding a shortest path is the classic algorithm known as ***breadth-first search***, which is a search process that expands outward from the starting position, considering first all possible solutions that are one step away from the start, then all possible solutions that are two steps away, and so on, until an actual solution is found. Because you check all the paths of length 1 before you check any of length 2, the first successful path you encounter must be as short as any other.

For word ladders, the breadth-first strategy starts by examining those ladders that are one step away from the original word, which means that only one letter has been changed. If any of these single-step changes reach the destination word, you're done. If not, you can then move on to check all ladders that are two steps away from the original, which means that two letters have been changed. In computer science, each step in such a process is called a ***hop***.

The breadth-first algorithm is typically implemented by using a queue to store partial ladders that represent possibilities to explore. The ladders are enqueued in order of increasing length. The first elements enqueued are all the one-hop ladders, followed by the two-hop ladders, and so on. Because queues guarantee first-in/first-out processing, these partial word ladders will be dequeued in order of increasing length.

To get the process started, you simply add a ladder consisting of only the start word to the queue. From then on, the algorithm operates by dequeueing the ladder from the front of the queue and determining whether it ends at the goal. If it does, you have a complete ladder, which must be minimal. If not, you take that partial ladder and extend it to reach words that are one additional hop away, and enqueue those extended ladders, where they will be examined later. If you exhaust the queue of possibilities without having found a completed ladder, you can conclude that no ladder exists.

It is possible to make the algorithm considerably more concrete by implementing it in ***pseudocode***, which is simply a combination of actual code and English. The pseudocode for the word-ladder problem appears as following.

```
Create an empty queue.
Add the start word to the end of the queue.
while (the queue is not empty) {
    Dequeue the first ladder from the queue.
    if (the final word in this ladder is the destination word){
        Return this ladder as the solution.
    }
    for (each word in the lexicon of English words that differs by one
        letter){ if (that word has not already been used in a ladder) {
            Create a copy of the current ladder.
            Add the new word to the end of the
            copy. Add the new ladder to the end of
            the queue.
        }
    }
}
Report that no word ladder exists.
```

As is generally the case with pseudocode, several of the operations that are expressed in English need to be fleshed out a bit. For example, the loop that reads

```
for (each word in the lexicon of English words that differs by one letter)
```

is a conceptual description of the code that belongs there. It is, in fact, unlikely that this idea will correspond to a single for loop in the final version of the code. The basic idea, however, should still make sense. What you need to do is iterate over all the words that differ from the current word by one letter. One strategy for doing so is to use two nested loops; one that goes through each character position in the word and one that loops through the letters of the alphabet, replacing the character in that index position with each of the 26 letters in turn. Each time you generate a word using this process, you need to look it up in the lexicon of English words to make sure that it is actually a legal word.

Another issue that is a bit subtle is the restriction that you not reuse words that have been included in

a previous ladder. One advantage of making this check is that doing so reduces the need to explore redundant paths. For example, suppose that you have previously added the partial ladder

```
cat → cot → cog
```

to the queue and that you are now processing the ladder

```
cat → cot → con
```

One of the words that is one hop away from `con`, of course, is `cog`, so you might be tempted to enqueue the ladder

```
cat → cot → con → cog
```

Doing so, however, is unnecessary. If there is a word ladder that begins with these four words, then there must be a shorter one that, in effect, cuts out the middleman by eliminating the unnecessary word `con`. In fact, as soon as you've enqueued a ladder ending with a specific word, you never have to enqueue that word again.

The simplest way to implement this strategy is to keep track of the words that have been used in any ladder (which you can easily do using another lexicon) and ignore those words when they come up again. Keeping track of what words you've used also eliminates the possibility of getting trapped in an infinite loop by building a circular ladder, such as

```
cat → cot → cog → bog → bag → bat → cat
```

One of the other questions you will need to resolve is what data structure you should use to represent word ladders. Conceptually, each ladder is just an ordered list of words, which should make your mind scream out "Vector!" (Given that all the growth is at one end, stacks are also a possibility, but vectors will be more convenient when you are trying to print out the results.) The individual components of the `Vector` are of type `string`.

Implementing the application

At this point, you have everything you need to start writing the actual C++ code to get this project done. It's all about leveraging the class library—you'll find your job is just to coordinate the activities of various different queues, vectors, and lexicons necessary to get the job done. The finished assignment requires less than a page of code, so it's not a question of typing in statements until your fingers get tired. It will, however, certainly help to think carefully about the problem before you actually begin that typing.

As always, it helps to plan your implementation strategy in phases rather than try to get everything working at once. Here, for example, is one possible breakdown of the tasks:

- *Task 1—Try out the demo program.* Play with the demo just for fun and to see how it works from a user's perspective.
- *Task 2—Read over the descriptions of the classes you'll need.* For this part of the assignment, the classes you need from Chapter 5 are `Vector`, `Queue`, and `Lexicon`. If you have a good sense of how those classes work before you start coding, things will go *much* more smoothly than they will if you try to learn how they work on the fly.
- *Task 3—Think carefully about your algorithm and data-structure design.* Be sure you understand the breadth-first algorithm and what data types you will be using.
- *Task 4—Play around with the lexicon.* The starter project for this problem includes a copy of the `EnglishWords.txt` file described in Lecture. Before you write the word ladder application, you might experiment with a simpler program that uses the lexicon in simpler ways. For example, you might write a program that reads in a word and then prints out all the English words that are one letter

away.

- *Task 5—Implement the breadth-first search algorithm.* Now you're ready for the meaty part. The code is not long, but it is dense, and all those templates will conspire to trip you up. We recommend writing some test code to set up a small dictionary (with just ten words or so) to make it easier for you to test and trace your algorithm while you are in development. Test your program using the large dictionary only after you know it works in the small test environment.

Note that breadth-first search is not the most efficient algorithm for generating minimal word ladders. As the lengths of the partial word ladders increase, the size of the queue grows exponentially, leading to exorbitant memory usage when the ladder length is long and tying up your computer for quite a while examining them all.

Problem 4 Rising Tides

Global sea levels have been rising, and the most recent data suggest that the rate at which sea levels are rising is increasing. This means that city planners in coastal areas need to start designing developments so that an extra meter of water doesn't flood people out of their homes.

Your task in this part of the assignment is to build a tool that models flooding due to sea level rise. To do so, we're going to model terrains as grids of doubles, where each doublerepresents the altitude of a particular square region on Earth. Higher values indicate higher elevations, while lower values indicate lower elevations. For example, take a look at the three grids to the right. Before moving on, take a minute to think over the following questions, which you don't need to submit. Which picture represents a small hill? Which one represents a long, sloping incline? Which one represents a lowland area surrounded by levees?

0	1	2	3	4	2	1
1	2	3	4	5	4	2
3	4	5	6	6	5	4
2	4	5	7	5	3	2
1	2	4	5	3	2	1
0	1	2	3	1	1	1
0	0	1	2	1	1	1

-1	0	0	4	0	0	1
0	0	4	0	-1	-1	0
0	4	0	0	0	0	0
4	0	-1	-1	0	0	3
0	-1	-2	-1	0	3	0
0	0	-1	0	3	0	0
0	0	0	3	0	0	-1

0	1	2	3	4	5	6
0	1	2	3	4	5	5
0	1	2	3	3	4	4
0	0	1	2	3	3	3
0	0	1	1	2	2	2
-1	-1	0	0	1	1	1
-2	-1	0	0	0	0	0

We can model the flow of water as follows. We'll imagine that there's a water source somewhere in the world and that we have a known height for the water. Water will then flow anywhere it can reach by moving in the four cardinal directions (up/down/left/right) without moving to a location at a higher elevation than the initial water height. For example, suppose that the upper-left corner of each of the three above worlds is the water source. Here's what would be underwater given several different water heights:

Water source at top-left corner

Height: 0m

0	1	2	3	4	2	1
1	2	3	4	5	4	2
3	4	5	6	6	5	4
2	4	5	7	5	3	2
1	2	4	5	3	2	1
0	1	2	3	1	1	1
0	0	1	2	1	1	1

-1	0	0	4	0	0	1
0	0	4	0	-1	-1	0
0	4	0	0	0	0	0
4	0	-1	-1	0	0	3
0	-1	-2	-1	0	3	0
0	0	-1	0	3	0	0
0	0	0	3	0	0	-1

0	1	2	3	4	5	6
0	1	2	3	4	5	5
0	1	2	3	3	4	4
0	0	1	2	3	3	3
0	0	1	1	2	2	2
-1	-1	0	0	1	1	1
-2	-1	0	0	0	0	0

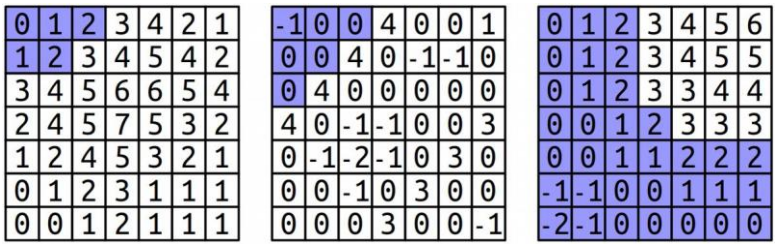
Height: 1m

0	1	2	3	4	2	1
1	2	3	4	5	4	2
3	4	5	6	6	5	4
2	4	5	7	5	3	2
1	2	4	5	3	2	1
0	1	2	3	1	1	1
0	0	1	2	1	1	1

-1	0	0	4	0	0	1
0	0	4	0	-1	-1	0
0	4	0	0	0	0	0
4	0	-1	-1	0	0	3
0	-1	-2	-1	0	3	0
0	0	-1	0	3	0	0
0	0	0	3	0	0	-1

0	1	2	3	4	5	6
0	1	2	3	4	5	5
0	1	2	3	3	4	4
0	0	1	2	3	3	3
0	0	1	1	2	2	2
-1	-1	0	0	1	1	1
-2	-1	0	0	0	0	0

Height: 2m

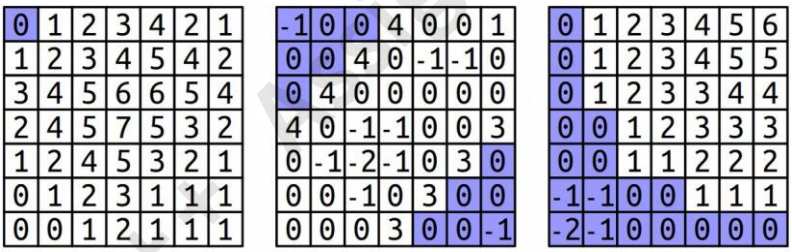


A few things to notice here. First, notice that the water height is independent of the height of the terrain at its starting point. For example, in the bottom row, the water height is always two meters, even though the terrain height of the upper-left corner is either 0m or -1m, depending on the world. Second, in the terrain used in the middle column, notice that the water stays above the upper diagonal line of 4's, since we assume water can only move up, down, left, and right and therefore can't move diagonally through the gaps. Although there's a lot of terrain below the water height, it doesn't end up under water until the height reaches that of the barrier.

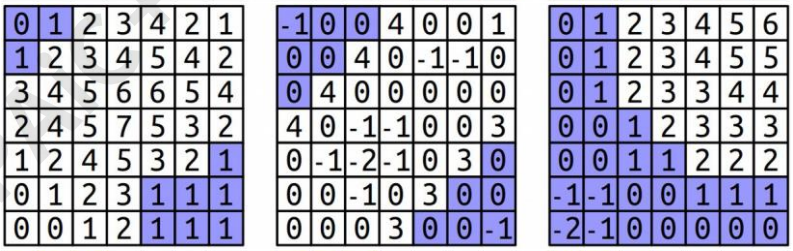
It's possible that a particular grid has multiple different water sources. This might happen, for example, if we were looking at a zoomed-in region of the San Francisco Peninsula, we might have water to both the east and west of the region of land in the middle, and so we'd need to account for the possibility that the water level is rising on both sides. Here's another set of images, this time showing where the water would be in the sample worlds above assume that both the top-left and bottom-right corner are water sources. (We'll assume each water source has the same height.)

Water sources at top-left and bottom-right corners

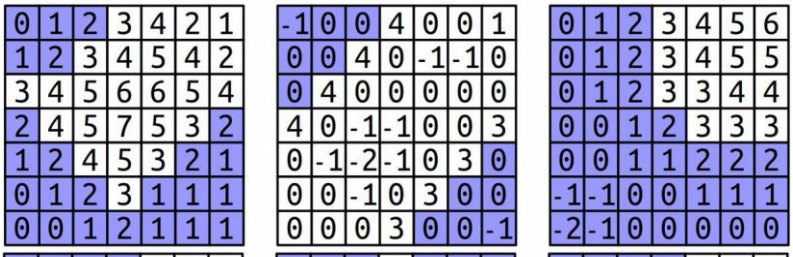
Height: 0m



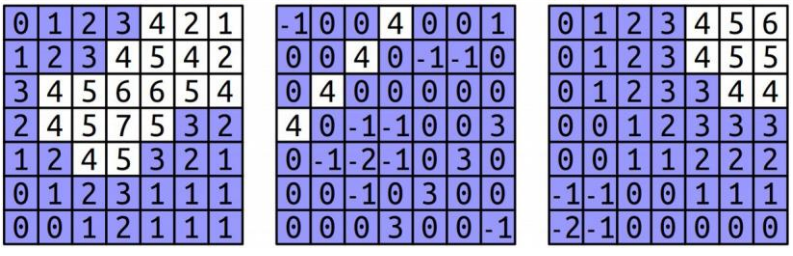
Height: 1m



Height: 2m



Height: 3m



Notice that the water overtops the levees in the central world, completely flooding the area, as soon as the water height reaches three meters. The water line never changes, regardless of the current elevation. As such, water will never flood across cells at a higher elevation than the water line, but will flood across cells at the same height or below the water line

Your task is to implement a function

```
Grid<bool> floodedRegionsIn(const Grid<double>&terrain,
                            const Vector<GridLocation>& sources,double height);
```

that takes as input a terrain (given as a Grid<**double**>), a list of locations of water sources (represented as a Vector<GridLocation>; more on GridLocation later), and the height of the water level, then returns a Grid<**bool**> indicating, for each spot in the terrain, whether it's under water (**true**) or above the water (**false**).

You may have noticed that we're making use of the **GridLocation** type. This is a type representing a position in a Grid. You can create a **GridLocation** and access its row and column using this syntax:

```
GridLocation location;
location.row =137;
location.col = 42;
GridLocation otherLocation = { 106, 103 }; // Row 106, Column 103
otherLocation.row++;
cout << otherLocation.col << endl;
```

Now that we've talked about the types involved here, let's address how to solve this problem. How, exactly, do you determine what's going to be underwater? Doing so requires you to determine which grid locations are both (1) below the water level and (2) places water can flow to from one of the sources.

Fortunately, there's a beautiful algorithm you can use to solve this problem called ***breadth-first search***. The idea is to simulate having the water flow out from each of the sources at greater and greater distances. First, you consider the water sources themselves. Then, you visit each location one step away from the water sources. Then, you visit each location two steps away from the water sources, then three steps, four steps, etc. In that way, the algorithm ends up eventually finding all places the water can flow to, and it does so fairly quickly!

Breadth-first search is typically implemented by using a ***queue*** that will process every flooded location. The idea is the following: we begin by enqueuing each water source, or at least the sources that aren't above the water line. That means that the queue ends up holding all the flooded locations zero steps away from the sources. We'll then enqueue a next group of elements, corresponding to all the flooded locations one step away from the sources. Then we'll enqueue all flooded locations two steps away from the sources, then three steps away, etc. Eventually, this process will end up visiting every flooded location in the map.

Let's make this a bit more concrete. To get the process started, you add to the queue each of the individual water sources that happen to be at least as high as the grid cell they're located in. From then on, the algorithm operates by dequeuing a location from the front of the queue. Once you have that location, you look at each of the location's neighbors in the four cardinal directions. For each of those neighbors, if that neighbor is already flooded, you don't need to do anything because the search has already considered that square. Similarly, if that neighbor is above the water line, there's nothing to do because that square shouldn't end up under water. However, if neither of those conditions hold, you then add the neighbor to the queue,

meaning "I'm going to process this one later on." By delaying processing that neighbor, you get to incrementally explore at greater and greater distances from the water sources. The process stops once the queue is empty; when that happens, every location that needs to be considered will have been visited.

At each step in this process, you're removing the location from the queue that's been sitting there the longest. Take a minute to convince yourself that this means that you'll first visit everything zero steps from a water source, then one step from a water source, then two steps, etc.

To spell out the individual steps of the algorithm, here's some ***pseudocode*** for breadth-first search.

```
create an empty queue;
for (each water source at or below the water level) {
    flood that square;
    add that square to the queue;
}

while (the queue is not empty) {
    dequeue a position from the front of the queue;

    for (each square adjacent to the position in a cardinal direction) {
        if (that square at or below the water level and isn't yet flooded) {
            flood that square;
            add that square to the queue;
        }
    }
}
```


As is generally the case with pseudocode, several of the operations that are expressed in English need to be fleshed out a bit. For example, the loop that reads

for (*each square adjacent to the position in a cardinal direction***)**

is a conceptual description of the code that belongs there. It's up to you to determine how to code this up; this might be a loop, or it might be several loops, or it may be something totally different. The basic idea, however, should still make sense. What you need to do is iterate over all the locations that are one position away from the current position. How you do that is up to you.

It's important to ensure that the code you write works correctly here, and to help you with that we've provided a suite of test data with the starter files. These test data look at a few examples, but they aren't exhaustive.

To summarize, here's what you need to do for this assignment:

1. Implement the ***floodedRegionsIn*** function in ***RisingTides.cpp***. Your code should use the breadth-first search algorithm outlined above in pseudocode to determine which regions are under water. Water flows up, down, left, and right and will submerge any region whose height is less than or equal to the global water level passed in as a parameter. Test as you go.

2. Add in at least one custom test data into ***RisingTides.cpp***– preferably, not one from the assignment handout – and see how things go.

Some notes on this problem:

- Need a refresher on ***Grid*** type? Check the Stanford C++ Library Documentation.
- The initial height of the water at a source may be below the level of the terrain there. If that happens, that water source doesn't flood anything, including its initial terrain cell. (This is an edge

case where both the "flood it" and "don't flood it" options end up being weird, and for consistency we decided to tiebreak in the "don't flood it" direction.)

- The heights of adjacent cells in the grid may have no relation to one another. For example, if you have the topography of Preikestolen in Norway, you might have a cell of altitude 0m immediately adjacent to a cell of altitude 604m. If you have a low-resolution scan of Mount Whitney and Death Valley, you might have a cell of altitude 4,421m next to a cell of altitude -85m.

Once you're tested all the data, please try "**final test data**" to see your code in action! Some of these sample data are large, and it might take a while for that flood-fill to finish. However, it should probably take no more than 30 seconds for the program to finish running your code. If you find that it's taking more than 30 seconds to load a terrain, it may mean that you have an inefficiency somewhere in your code. Make sure that you aren't passing large objects into functions by value (that is, if you're passing around a Grid or Queue or something like that, it should be done by reference or const reference)