CS106B Practice Exam

Exam Facts:

When: Tuesday, October 30th from 7:00 – 10:00 p.m.
Where: Last Names A through L: Braun Auditorium (138 students, 292 seats)
Last Names M through Z: Building 320, Room 105 (107 students, 242 seats)

Note this practice exam is much longer than your will be.

Coverage

Your first exam covers everything up through recursive backtracking and memoization. All students are expected to take the exam at the above time unless something has already been worked out.

The exam is closed-book, closed-reader, closed-computer. We will not be especially picky about syntax or other conceptually shallow ideas. We are looking for a clear understanding of the core programming concepts in C++. As needed, we will present the prototypes of functions and methods we expect to be relevant to the correct answer.

Writing code on paper in a relatively short time period is not quite the same as working with the compiler and a keyboard. I recommend that you practice writing out solutions to these practice problems—starting with a blank sheet of paper—until you're certain you can write code without a computer to guide you.

Problem 1: Acronyms [courtesy of Julie Zelenski]

An acronym is an abbreviation formed from the first letter of each of a series of words, such as NATO [North American Treaty Organization] or HTML [Hyper Text Markup Language]. In this problem, you are to write two functions that operate on a table of acronyms. An acronym data file contains a list of expanded acronyms, one per line, such as shown here:

```
American Automobile Association
Internal Revenue Service
Standard Query Language
Parent Teacher Association
Animal Acupuncture Academy
Inertial Reference System
Abdominal Aortic Aneurysm
National Direct Student Loan
```

The terms within each expansion are separated by at least one and possibly more spaces. All terms within an expansion are capitalized and thus the acronym formed is uppercase.

a) The **`buildMap`** function reads an acronym data file and builds a map from acronyms to expansions. The two parameters to the function are a correctly opened **`ifstream`** and an empty map. The function should fill the map with entries where acronym (e.g. **`"IRS"`**) is the key, and the associated value is its expansion(s) (e.g., **`"Internal Revenue Service"`** and "Inertial Reference System"). An acronym may have more than one expansion (e.g., there are three options for "AAA" in the above file) and thus the value for each key is a vector, where each entry is a possible expansion. (The **`explode`** function from the Twitter example, Handout 12, will be helpful here.)

```
static void buildMap(ifstream& infile, Map<string, Vector<string> >& map);
```

b) Although using acronyms can be convenient, it can be confusing when one acronym has many different expansions. Given a map such as the one created by the function from part (a), the **`percentConfusing`** function returns the percentage of acronyms in the map that have more than one expansion. For a map constructed from the data file shown on the previous page, there are five unique acronyms, of which two have multiple expansions, so the returned percentage would be 40% (expressed as a double that's **`0.40`**).

```
static double percentConfusing(Map<string, Vector<string> >& map);
```

Problem 2: HTML Entitles

HTML is a language that helps decorate plain text with tags that serve as instructions on how to style the text within a web browser. Tags take the form of strings that begin with **`'<'`** and end with **`'>'`**. Some of the more common tags are **`<b>`** (for styling text as boldface), **`<u>`** (for underlining text), and **`<i>`** (for italicizing text).

Because **'<'** and **'>'** are so prevalent in HTML, it's difficult to include actual < and > characters in plain text because the browser might confuse them as markup. As a result, the following plain text:

< and > are popular relational operators.

might give the impression that there's a leading HTML **and** tag. You could argue that browsers should just know that **and** isn't a genuine tag, but as it turns out, HTML documents can define their own tags, and in principle **<and>** might be a legitimate tag type within a particular document.

The solution is to replace the plain text **<**'s and the >'s with **HTML entities**—strings used to represent characters in a document that otherwise have special meaning. In standard HTML, < is represented by **"<"** (without the double quotes), and > is represented by **">"**. And because **&** has special meaning now, there's also an HTML entity for it: **"&"**. In fact, there are a whole slew of HTML entities beyond these three. Here are just a few examples:

<	**<**
>	**>**
&	**&**
"	**"**
'	**'**
¢	**¢**

Every single HTML entity begins with **&** and ends with **;**, and is associated with some single character in (what we're simplifying it to be) the ASCII character set.

To be clear:

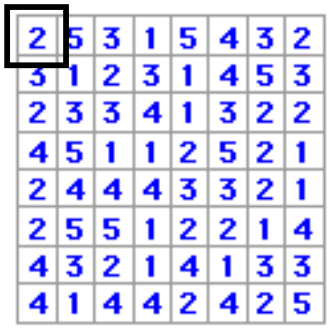
John said, "Hi!" is really **John said, "Hi!"**
Cold Stone's sundaes: 49¢! is really **Cold Stone's sundaes: 49¢!**
A&P Supermarket is really **A&P Supermarket**
x < 7 && y >= 10 is really **x < 7 && y >= 10**

You're to write a function called **restorePlainText**, which takes a string and a map of HTML entity strings to characters, and returns a copy of the incoming string, save that all of its HTML entities have been replaced with the characters they represent. The **Map<string, char>** is keyed on the HTML entity (without the leading **&** and trailing **;** so that strings like **"lt"** and **"amp"** are keys, not **"<"** and **"&"**) and each maps to the character it represents (i.e. **'<'** or **'&'**). If the string contains what appears to be an HTML entity not in the **Map<string, char>**, then that entity is left in the string as is. You can assume the string is otherwise well formed, in that every opening **&** is eventually balanced by a closing **;**, and that there aren't any unpaired ampersands or semicolons.

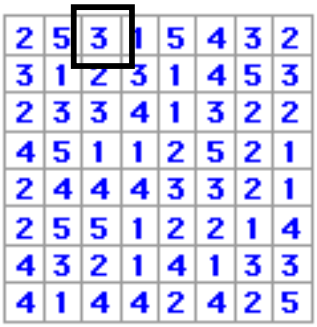
```
static string restoreString(const string& str, Map<string, char>& entityMap);
```

Problem 3: Pascal's Travels

You're given a grid of positive integers to represent a game board, where the [0, 0] entry is the upper left corner. The number in each location is the number of squares you can advance in any of the four primary compass directions, provided that move doesn't take you off the board. You're interested in the total number of distinct ways one could travel from the upper left corner to the lower right corner, given the constraint that no single path should ever visit the same location twice.

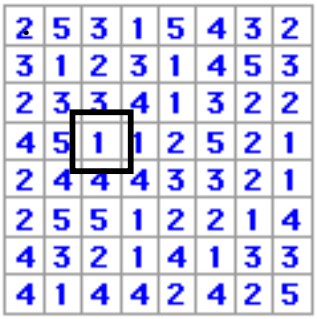


Consider the initial game board to the left, and notice that the upper left corner is occupied by a 2. That means you can take either two steps to the right, or two steps down (but not two steps to the left or above, because that would carry you off the board). Suppose you opt to go right so that you find yourself in the configuration to the right.

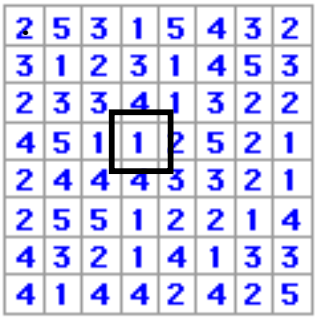


From there, you could continue along as follows:

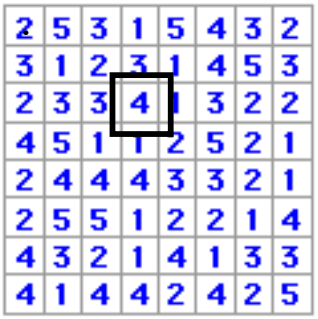
1



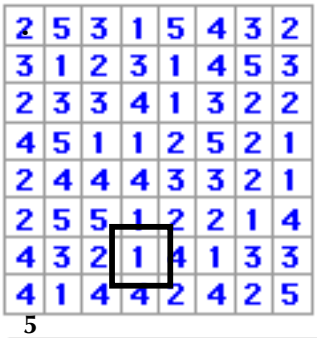
2



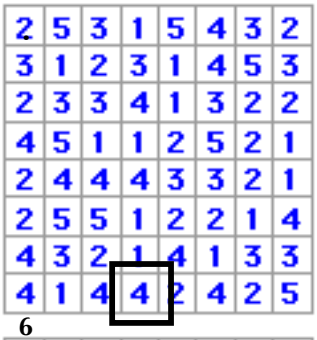
3



4



5



6

So, this series of moves illustrates just one of potentially several paths you could take from upper left to lower right. Your job here is to write a function called **`countPaths`**, which takes a **`Grid`** of integers and computes the total number of ways to travel to the lower right corner of the board. Note that you never want to count the same path twice, but two paths are considered to be distinct even if they share a common sub-path. And because you want to prevent cycles, you should change the value at any given location to a zero as a way of marking that you've been there. Just be sure to restore the original value as you exit the recursive call. You'll want to write a wrapper function and some utility functions to decide it you're on the board or not.

```
static int countPaths(Grid<int>& board);
```

Problem 4: Longest Increasing Subsequence

Given a string of lowercase letters, write a function **`longestIncreasingSubsequence`** that returns the longest increasing subsequence. Recall that a subsequence of a string is the same string, except that an arbitrary number of its characters have been removed (and the order of the characters that remain is preserved.) By increasing, we mean that as a character sequence the string is sorted from low to high.

Given the string **`"aiemckgobjfndlhp"`**, each of the following is a subsequence:

`"mjp"`
`"acgjlp"`
`"aiemckgobjfndlhp"`
`"aemckobjfnlp"`
`"emjfnp"`
`"aegjnp"`

Of the six subsequences above, only **`"acgjlp"`** and **`"aegjnp"`** are increasing, since as character arrays they are sorted from low to high. And because there aren't any increasing subsequences of length 7 or more (an exhaustive search demonstrates this—just trust us), **`"acgjlp"`** and **`"aegjnp"`** are each longest increasing subsequences.

Using this and the next page, write a recursive function that takes a string and returns the longest increasing subsequence. If there are two or more such subsequences, your function can return any one of them.

```
static string longestIncreasingSubsequence(const string& str);
```

Problem 5: Polydivisible Numbers

Polydivisible numbers are positive integers such that the first k digits, when taken as a number, are divisible by k, for all reasonable values of k. For example, 8076 is a polydivisible number, because:

- 8 is divisible by 1
- 80 is divisible by 2
- 807 is divisible by 3
- 8076 is divisible by 4

Other examples of polydivisible numbers: 30080, 1652588, 444402009, 80480408404, and 76245056107220 are all examples of polydivisible numbers. 3608528850368400786036725, at 25 digits, is the largest polydivisible number.

Write a recursive procedure **`generatePolydivisibleNumbers`** that procedurally constructs all polydivisible numbers and implants them in a **`Set<int>`** passed in by reference. You should assume there's no limit to how big an **`int`** can be, and you can also

assume that the number of polydivisibles is finite. Your implementation must make direct, relevant use of the observation that

10 * n + d (where 0 <= d < 10) can only be polydivisible if n is.

Implement to the following interface:

```
static void generatePolydivisibleNumbers(Set<int>& numbers);
```

Problem 6: Sanitizing Strings

Write a recursive procedure called **`sanitize`**, which takes a string of text and a **`Vector<string>`** of banned substrings and returns the shortest string that can be generated by eliminating all illegal substrings. For instance, if the only banned substring is **`abc`**, then the string **`babaabcbcc`** can be sanitized down to **`b`**, via:

babaabcbcc
babcc
babc
b

Note that each string in the stack is the same at the one above it, except that the underlined substrings have been spliced out.

A more elaborate example illustrates that **`baacabacaaabcaaabbaa`** can be edited down to **`baa`** if the set of banned substrings includes **`ac`**, **`ab`**, and **`caa`**, as illustrated by:

baacabacaaabcaaabbaa
baacabacaaabcaabaa
baacacaaabcaabaa
baacaabcaabaa
baabcaabaa
baabcaaa
bacaaa
babaa
baa

Write a recursive procedure called **`sanitize`** that computes and returns the **smallest** string that can be generated by an optimal series of substring eliminations.

```
static string sanitize(const string& str, Vector<string>& substrings);
```

Problem 7: Recursive Backtracking and Scheduling Movies

The Academy Awards are fast approaching (Seth MacFarlane will host them on February 24th, 2013), and you've been so busy with school and summer internships that you've not

seen a single movie all year. You've decided to set aside a single Saturday to see a predefined list of nominated movies, and you're hoping there's some way to fit them all in on any given day. Given a dictionary of movies (each bundling the title, the length in minutes, and its various start times into a single **struct**) and the titles you want to see, write a function that decides whether or not it's possible to see everything you want to see in any given day. For simplicity, assume that you can attend a movie that begins at the same time another movie ends. Write a function that returns **true** if and only if it's possible to see all the movies you want to see, and **false** otherwise. Notice that you needn't generate the schedule itself—just a yes or no as to whether a schedule exists.

Your routine should return as soon as it can produce a **true** or **false**, as all of our recursive backtracking examples have.

```
struct movie {
   string name;
   int duration;           // in minutes
   Vector<int> showTimes;  // each in minutes since midnight
};

static bool canSchedule(Vector<string>& titles, Map<string, movie>& schedule);
```