CS106B

Winter 2023

Practice Midterm Exam 3

This is a printable version of the third set of practice problems from the practice midterm exam. It's designed to have the same format and style as the actual CS106B midterm.

Solutions can be found by visiting the Additional Practice Problems resource on the CS106B website and looking up solutions to the corresponding problems.

Lexicon

```
Lexicon lex; Lexicon english(filename);
lex.addWord(word);

bool present = lex.contains(word);
bool pref = lex.containsPrefix(prefix);

int numElems = lex.size();
bool empty = lex.isEmpty();

lex.clear();

/* Elements visited in sorted order. */
for (string word: lex) { … }
```

Map

```
Map<K, V> map = {{k1, v1}, … {kn, vn}};

cout << map[key] << endl; // Autoinserts
map[key] = value; // Autoinserts

bool present = map.containsKey(key);
int numKeys = map.size();
bool empty = map.isEmpty();

map.remove(key);
map.clear();

Vector<K> keys = map.keys();
K key = map.firstKey();

/* Keys visited in ascending order. */
for (K key: map) { … }
```

Stack

```
stack.push(elem);

T val = stack.pop();   // Removes top
T val = stack.peek();  // Looks at top

int numElems = stack.size();
bool empty = stack.isEmpty();

stack.clear();
```

Queue

```
queue.enqueue(elem);

T val = queue.dequeue(); // Removes front
T val = queue.peek();   // Looks at front

int numElems = queue.size();
bool empty = queue.isEmpty();

queue.clear();
```

Set

```
Set<T> set = {v1, v2, …, vn};

set.add(elem); set += elem;

Set<T> s = set - elem; // or + elem

bool present = set.contains(elem);
set.remove(x); set -= x; set -= set2;

Set<T> unionSet = s1 + s2;
Set<T> intersectSet = s1 * s2;
Set<T> difference = s1 - s2;

T elem = set.first();

int numElems = set.size();
bool empty = set.isEmpty();

set.clear();

/* Visited in ascending order. */
for (T elem: set) { … }
```

Vector

```
Vector<T> vec = {v1, v2, …, vn};

vec[index]; // Read/write

vec.add(elem);  vec += elem;

vec.insert(index, elem);

vec.indexOf(elem); // index or -1

vec.remove(index);
vec.clear();

int numElems = vec.size();
bool empty = vec.isEmpty();

Vector<T> v = vec.subList(start, nElems);

/* Visited in sequential order. */
for (T elem: vec) { … }
```

string

```
str[index]; // Read/write

str.substr(start);
str.substr(start, numChars);

str.find(c); // index or string::npos
str.find(c, startIndex);

str += ch; str += otherStr;
str.erase(index, length);

/* Visited in sequential order. */
for (char ch: str) { … }
```

Grid

```
Grid<T> grid(nRows, nCols);
Grid<T> grid(nRows, nCols, fillValue);

int nRows = grid.numRows();
int nCols = grid.numCols();

if (grid.inBounds(row, col)) { … }

grid[row][col] = value;
cout << grid[row][col] << endl;

/* Visited left-to-right, top-to-bottom */
for (T elem: grid) { … }
```

Problem One: Minesweeper

Minesweeper is a puzzle game that's shipped with Microsoft Windows almost since its inception. (I have memories as a kid playing Minesweeper on Windows 3.1. Ah, the nostalgia.) In Minesweeper, you're presented a grid of squares, some of which contain mines. Your goal is to locate the mines. Clicking on a square will do one of two things:

- If it's a mine, you lose the game.
- Otherwise, the square will be updated to tell you how many mines are adjacent to the square, where "adjacent" means "adjacent up, down, left, right, or diagonally."

Here's an example grid showing where the mines are, as well as the numbers indicating how many mines are adjacent to each square. Following the convention from the original game, I've left blank all the squares that are adjacent to zero mines.

Write a function

```
Grid<int> minesAdjacentTo(const Grid<bool>& mines);
```

that takes as input a `Grid<bool>` indicating where mines are located, then returns a `Grid<int>` that says how many mines are adjacent to each location. As a special case, each square containing a mine should be set to -1, since in Minesweeper you never see how many mines are adjacent to a cell that contains a mine. If a square doesn't contain a mine and isn't adjacent to any mines, that square should hold the value 0.

```
Grid<int> minesAdjacentTo(const Grid<bool>& mines) {
```

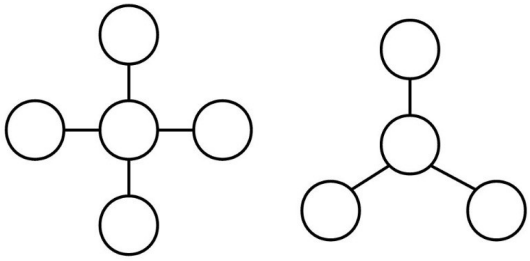
We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There's additional space for your answer on the next page.

Additional space for your answer to Problem One, if you need it.

Problem Two: No More Counting Dollars…

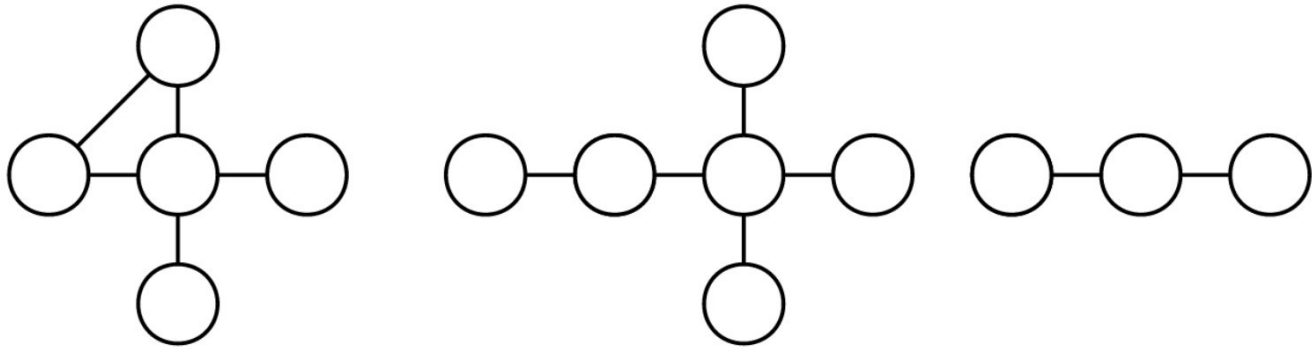
You have a road network for a country represented as a `Map<string, Set<string>>`. Each key is the name of a city, and the associated value consists of all the cities that are adjacent to that city. You can assume that the road network is bidirectional: if *A* is adjacent to *B*, then *B* is adjacent to *A*. You can also assume no city is adjacent to itself.

A ***star*** is a cluster of four or more cities arranged as follows: there's a single city in the center that's connected to all the other cities in the cluster, and every other city in the cluster is only connected to the central city. Here are some stars shown below:



Stars commonly arise in road networks in smaller island nations: you have a main, central city (often a capital city) and a bunch of smaller, outlying towns.

Here are some examples of groups of cities that aren't stars. The group on the left isn't a star because two of the peripheral cities are connected to one another (and therefore not just the central city). The group in the center isn't a star because one of the peripheral cities is connected to a city besides the central one. Finally, the group on the right isn't a star because it doesn't have the minimum required number of nodes.



If you have a country that consists of a large archipelago, you might find that its road network consists of multiple different independent stars. In fact, the number of stars in a road network is a rough proxy for how decentralized that country is.

Your task is to write a function

```
int countStarsIn(const Map<string, Set<string>>& network);
```

that takes as input the road network, then returns the number of stars in the network. You don't need to worry about efficiency, but do be careful not to count the same star multiple times.

```
int countStarsIn(const Map<string, Set<string>>& network) {
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There's additional space for your answer on the next page.

Additional space for your answer to Problem Two, if you need it.

Problem Three: Building an Index

Larger textbooks or reference books usually include an ***index***, a series of pages in which different terms are listed along with the page numbers on which those terms occur.

How might we represent an index in software? Given the text of the book, our goal is to create an object that's good at answering this question:

☞ ***Given a word, return the page numbers of each page containing that word.*** ☜

This problem has two parts.

First, using the C++ containers we've covered so far, tell us which type you think is best-suited for representing an index, given that the index needs to be able to easily answer the above question. (For example, you could say `string`, `Set<double>`, `Vector<Map<Stack<int>, Queue<char>>>`, etc.) Then, briefly justify your answer. Your justification should take at most fifty words; anything longer than that will receive no credit.

Some notes on this problem:

- Remember that you want to give back the *page numbers* on which the word occurs, not the pages themselves.
- If you can think of multiple answers that all sound good, choose only one as your answer to this question. If you list multiple answers, we will not award any points.

The type I would pick is _____.

Justification (at most fifty words):

We will grade whatever you write in this box.
Anything written elsewhere will be considered scratch work.

Your next task is to write a function

```
YourType makeIndexFor(const Vector<string>& pages, int maxFrequency);
```

that takes as input the pages of a book, represented as a `Vector<string>`. That is, `pages[0]` represents the first page of the book, `pages[1]` represents the second page of the book, etc. This function then returns the index, represented as an object of the type you mentioned in your answer above. As a reminder, the type you chose should be good at solving this problem:

☞ ***Given a word, return the page numbers of each page containing that word.*** ☜

There's one little detail we need to address: there are many words in a book that we wouldn't want to include an in index. For example, the word "a" is so common that including it in an index would take up a ton of space without actually contributing anything useful. To account for this, *the index you return should only include words that appear on at most `maxFrequency` pages*.

As a reminder, the elements in the `Vector` are individual pages in the book, while your index should be working at the level of individual words in the book. Feel free to use this `tokenize` function:

```
Vector<string> tokenize(const string& str);
```

This function takes as input a string representing the contents of a page, then returns a `Vector<string>` consisting of all the words on that page.

Some notes on this problem:

- Remember that your goal is to produce an index that tells you the *page numbers* of each page on which a given word occurs, not the contents of that page.
- For simplicity's sake, let's assume that page numbering starts at Page 0, not Page 1. That makes it easier to use a `Vector<string>` to represent the pages of the book.
- For full credit, please use our `tokenize` function rather than other approaches like the library function `stringSplit` or the helper type `TokenScanner`.

Feel free to use the rest of this page for scratch work. Write your answer on the next pages.

```
Vector<string> tokenize(const string& str); // Provided to you


/* Fill in the blank below with your return type, then implement this function. */
_____________________________ makeIndexFor(const Vector<string>& pages,
                                           int maxFrequency) {
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There's additional space for your answer on the next page.

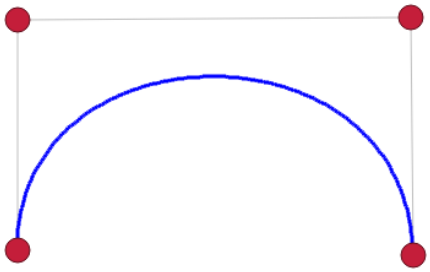
Additional space for your answer to Problem Three, if you need it.

Problem Four: Cubic Bézier Curves

Cubic Bézier curves are a family of curves that are used extensively in design, architecture, and computer graphics. They provide a rich, natural way of representing smooth curves and are used in both 2D and 3D graphics.

To draw a cubic Bézier curve, we begin with four points p_0, p_1, p_2, and p_3, which are referred to as the ***control points***. The curve will begin at p_0 and end at p_3. The curve is shaped so that it starts off tangent to the line from p_0 to p_1 and ends tangent to the lin from p_2 to p_3.

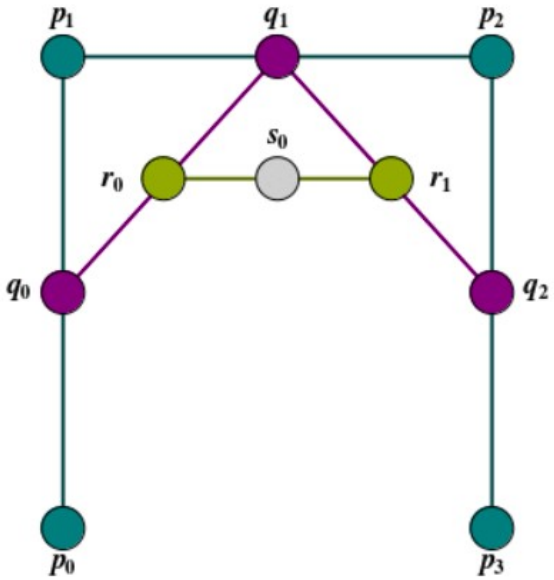
Here's a sample Bézier curve, where p_0 is in the bottom left, p_1 is in the top left, p_2 is in the top right, and p_3 is in the bottom right.



There are many algorithms for drawing cubic Bézier curves, and, conveniently, one of them works recursively. Like our recursive tree from lecture, the Bézier curve is parameterized by an order, where the higher the order, the more detail is present in the curve. An order-0 Bézier curve is just a straight line from p_0 to p_3. Otherwise, compute the following six points:

- Compute the midpoint of p_0 and p_1, calling this point q_0. Similarly, let q_1 be the midpoint of p_1 and p_2, and let q_2 be the midpoint of p_2 and p_3.
- Compute the midpoint of q_0 and q_1, calling this point r_0. Similarly, let r_1 be the midpoint of q_1 and q_2.
- Compute the midpoint of r_0 and r_1, calling this point s_0.

This is shown here:



(Continued on the next page…)

Once we've done this, we recursively draw an order-(n-1) cubic Bézier curve using the control points p_0, q_0, r_0, and s_0 and another order-(n-1) cubic Bézier curve using the control points s_0, r_1, q_2, and p_3.

Write a function

```
void drawBezierCurve(const GPoint& p0,
                     const GPoint& p1,
                     const GPoint& p2,
                     const GPoint& p3,
                     int order);
```

that draws a Bézier curve of the given order using the given control points. You can assume you have access to a function

```
void drawLine(const GPoint& p0, const GPoint& p1);
```

that draws a line between the two indicated points.

The `GPoint` type is a `struct` that has two fields `x` and `y`. You can use it like this:

```
GPoint pt = { 137, 42 }; // (137, 42)
cout << pt.x << endl;    // 137
cout << pt.y << endl;    // 42
```

If the order parameter to `drawBezierCurve` is negative, call error to report an error.

```
void drawBezierCurve(const GPoint& p0,
                     const GPoint& p1,
                     const GPoint& p2,
                     const GPoint& p3,
                     int order); {
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There's additional space for your answer on the next page.

Additional space for your answer to Problem Four, if you need it.

Problem Five: Social Distancing

There's a checkout line at a local grocery store. It's one meter wide, some number of meters long, and subdivided into one-meter-by-one-meter squares. Each of those squares is either empty or has a person standing in it. We can represent the state of the checkout line as a string made of the characters `'P'` and `'.'`, where `'P'` represents a square with a person in it and `'.'` represents a square with no one in it.

Social distancing rules require there to be at least two meters of empty space between each pair of people. So, for example, the string:

```
P..P....P........P
```

has everyone obeying social distancing rules, while the string

```
P..P..P.P
```

does not. Similarly, the string

```
P..P....PP....P
```

does not obey social distancing rules.

Your task is to write a function

```
Set<string> safeArrangementsOf(int lineLength, int numPeople);
```

that takes as input the length of the line (in meters) and the number of people in the line, then returns all arrangements of the checkout line that maintain the social distancing requirements. For example, calling `safeArrangementsOf(12, 4)` would return these strings:

```
P..P..P..P..
P..P..P...P.
P..P..P....P
P..P...P..P.
P..P...P...P
P..P....P..P
P...P..P..P.
P...P..P...P
P...P...P..P
P....P..P..P
.P..P..P..P.
.P..P..P...P
.P..P...P..P
.P...P..P..P
..P..P..P..P
```

On the other hand, calling `safeArrangementsOf(8, 4)` would return an empty set, since there's no way to fit four people into a line eight meters long while ensuring everyone is at least two meters apart. (Do you see why?)

```
Set<string> safeArrangementsOf(int lineLength, int numPeople) {
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There's additional space for your answer on the next page.

Additional space for your answer to Problem Five, if you need it.