# Practice CS106B Midterm Exam

This exam is closed-book and closed-computer. You may have a double-sided, 8.5” × 11” sheet of notes with you when you take this exam. You may not have any other notes with you during the exam. You may not use any electronic devices (laptops, cell phones, etc.) during the course of this exam without prior authorization from the course staff. Please write all of your solutions on this physical copy of the exam.

In all questions, you may include functions, classes, or other definitions that have been developed in the course, either by writing the `#include` line for the appropriate header or by giving the name of the function or class and the handout, chapter number, or lecture in which that definition appears.

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you to get partial credit if they help us determine what you were trying to do. You do not need to worry about efficiency unless a problem specifically requires an efficient solution.

This practice exam is completely optional and does not contribute to your overall grade in the course. However, we think that taking the time to work through it under realistic conditions is an excellent way to prepare for the actual exam and figure out where you need to focus your studying.

You have three hours to complete this practice exam. There are 40 total points.

| Question | Points | Graders |
|---|---|---|
| (1) Container Classes | / 8 | |
| (2) Recursive Enumeration | / 8 | |
| (3) Recursive Optimization | / 8 | |
| (4) Recursive Backtracking | / 8 | |
| (5) Big-O and Efficiency | / 8 | |
| | **/ 40** | |

## Problem One: Container Classes (8 Points)

***Shrinkable Word Ladders*** ***(Recommended time: 30 minutes)***

In lecture, we explored the question of finding shrinkable words. If you'll recall, a ***shrinkable word*** is a word that can be reduced down to a single-letter word by deleting one letter at a time, leaving what remains a word at each step. For example, the word "startling" is a shrinkable word, since we get this sequence of words back:

```
startling
starting
staring
string
sting
sing
sin
in
i
```

A sequence of words like this one is called a ***shrinking sequence***, since it shows how to shrink the initial word down to a single-letter word.

We solved this problem in class by using recursive backtracking, but that's by no means the only possible way that we could have done so. In fact, there's another very elegant way to solve this problem that uses the breadth-first search algorithm that you saw in the Word Ladders assignment. Here's the pseudocode for breadth-first search, slightly modified to fit the parameters of this problem:

```
create an empty queue;
add the start word to the end of the queue;

while (the queue is not empty) {
   dequeue the first shrinking sequence from the queue;

   if (the final word in this shrinking sequence is a single letter) {
       return this shrinking sequence as a solution;
   }

   for (each word that can be made by deleting a single letter) {
      if (that word has not already been used in a shrinking sequence) {
         create a copy of the current shrinking sequence;
         add the new word to the end of the copy;
         add the new shrinking sequence to the end of the queue;
      }
   }
}
return that no shrinking sequence exists;
```

Your task is to write a function

```
bool isShrinkable(const string& word, const Lexicon& english,
                  Vector<string>& shrinkingSequence)
```

that uses breadth-first search to determine whether the given word is shrinkable. If it is, your function should fill in the `shrinkingSequence` parameter with one possible shrinking sequence starting at the initial word and ending at a single-letter word. Some notes on this problem:

- You are required to use the breadth-first search algorithm. Recursive solutions aren't permitted.
- If there are many different shrinking sequences possible, you can pick any one of them.
- The input is not guaranteed to be an English word.

```
bool isShrinkable(const string& word, const Lexicon& english,
                  Vector<string>& shrinkingSequence) {
```

*(extra space for your answer to Problem One, if you need it.)*

## Problem Two: Recursive Enumeration (8 Points)

***Splitting the Bill*** ***(Recommended time: 45 minutes)***

You've gone out to dinner with a bunch of your friends and the waiter has just brought back the bill. How should you pay for it? One option would be to draw straws and have the loser pay for the whole thing. Another option would be to have everyone pay evenly. A third option would be to have everyone pay for just what they ordered. And then there are a ton of other options that we haven't even listed here!

Your task is to write a function

```
void listPossiblePayments(int total, const Set<string>& people);
```

that takes as input a total amount of money to pay (in dollars) and a set of all the people at the dinner, then lists off every possible way you could split the bill, assuming everyone pays a whole number of dollars. For example, if the bill was $4 and there were three people at the lunch (call them *A*, *B*, and *C*), your function might list off these options:

```
A: $4,    B: $0,    C: $0
A: $3,    B: $1,    C: $0
A: $3,    B: $0,    C: $1
A: $2,    B: $2,    C: $0
                …
A: $0,    B: $1,    C: $3
A: $0,    B: $0,    C: $4
```

Some notes on this problem:

- The total amount to pay will never be negative, and there will always be at least one person who has to pay.
- You can list off the possible payment options in any order that you'd like. Just don't list the same option twice.
- The output you produce should indicate which person pays which amount, but aside from that it doesn't have to exactly match the format listed above. Anything that correctly reports the payment amounts will get the job done.

```
void listPossiblePayments(int total, const Set<string>& people) {
```

*(extra space for your answer to Problem Two, if you need it.)*

## Problem Three: Recursive Optimization (8 Points)

***Drills, Baby, Drills!*** ***(Recommended time: 40 minutes)***

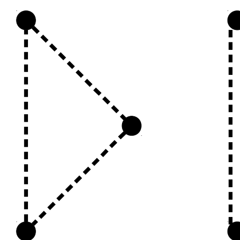
On Assignment 3, you wrote a function

```
Vector<DrillSite> bestDrillRouteFor(const Vector<DrillSite>& sites);
```

that takes as input a list of drill sites, then returns the fastest route for a robot to drill all of the holes and return back to its starting point. Here, `DrillSite` was a `struct` defined as follows:

```
struct DrillSite {
    string name; // Name of the drill site, for testing purposes
    GPoint pt;   // Where it is
};
```

Now, imagine you have *two independent robots* that can move and drill independently of one another. Consider the drill sites to the right, assuming that the top-left point is at (0cm, 0cm) and the bottom-right corner is at (6cm, 6cm). If you divide the points between the robots as shown here, the first robot moves about 14.5cm and the second robot a total of 12cm. Assuming the drills move at one centimeter per second, this means that the robots will collectively drill all the holes after 14.5 seconds (since we have to wait for both robots to finish drilling). This is a big improvement compared to a single robot, which would take a total of 26.5 seconds to finish.



Your job is to write a function

```
DrillRoutes bestDoubleDrillRouteFor(const Vector<DrillSite>& sites);
```

that takes as input a list of drill sites, then returns back the routes that each of the two independent robots should drill in order to minimize the completion time. Here, `DrillRoutes` is defined as

```
struct DrillRoutes {
    Vector<DrillSite> forRobotOne;
    Vector<DrillSite> forRobotTwo;
};
```

where `forRobotOne` is the list of routes that the first robot should drill, in order, and `forRobotTwo` is the list of routes that the second robot should drill. Some notes on this problem:

- You can assume that you have access to the `bestDrillRouteFor` function you wrote on Assignment 3 and that it works correctly. You ***should not*** try to reimplement that function here – we already asked you to write it once and that was probably plenty. ☺
- It's never optimal to have both robots drill the same hole.
- The two robots don't need to split the number of points perfectly in half.
- The total completion time for the two robots is defined as the *longer* of the two times that the robots will be working for. If the first robot finishes in one second and the second finishes in fifteen hours, the completion time for that split is fifteen hours.
- You can assume you have access to a function

  ```
  double drillRouteLength(const Vector<DrillSite>& path)
  ```

  that tells you how long it will take for a single drill to drill at all the sites in path and then return back to the starting point.
- You don't need to worry about the robots' paths crossing one another. Assume they'll never collide with one another.
- Your solution needs to use recursion to solve this problem. Again, that's what we're specifically testing here. ☺

```
struct DrillRoutes {
    Vector<DrillSite> forRobotOne;
    Vector<DrillSite> forRobotTwo;
};

/* Reports the total length of a route that drills all the sites in the
 * specified order. You do not need to implement this function.
 */
double drillRouteLength(const Vector<DrillSite>& path);

/* Given a list of sites that need to be drilled, returns the optimal order in
 * which they should be drilled. You do not need to implement this function.
 */
Vector<DrillSite> bestDrillRouteFor(const Vector<DrillSite>& sites);


DrillRoutes bestDoubleDrillRouteFor(const Vector<DrillSite>& sites) {
```

*(extra space for your answer to Problem Three, if you need it.)*

## Problem Four: Recursive Backtracking (8 Points)

***A Team-Building Exercise*** ***(Recommended time: 45 minutes)***

You are the volunteer coordinator for an educational outreach nonprofit. Your nonprofit sends groups of volunteers out to different schools to help provide tutoring and mentorship to local students. Specifically, you'll split everyone into groups of equal size, then send each group out to a different school.

The unfortunate reality, though, is that not all your volunteers necessarily get along very well with one another, and to ensure that you have the best impact on the students you want to make sure that all the people within the teams you assemble will get along well.

Write a function

```
bool canSplitMerrily(const Set<string>& people, int teamSize,
                     Vector<Set<string>>& teams)
```

that takes as input a set of all the people to break into groups of size `teamSize`, then returns whether it's possible to split everyone into teams in a way that makes everyone happy. If so, you should fill in the `teams` outparameter with one possible way of splitting everyone into teams.

In the course of writing this function, you should assume you have access to a function

```
bool areHappyTogether(const string& p1, const string& p2);
```

that takes in a pair of people and reports whether they'd be happy working together.

In writing this function, you can assume the following:

- Every team has to have size *exactly equal* to `teamSize`, no more, and no less.
- If each pair of people in a group are happy working together, then everyone in the group collectively will be happy working together.
- You can assume that the `teams` parameter is empty when the function is called, and its contents are irrelevant if your function returns false.
- The final order in which you return back the list of teams doesn't matter.
- You can assume the team size is at least one.

```
/* Given two people, returns whether they'd be happy working together.
 * This function is provided to you; you don't need to implement it.
 */
bool areHappyTogether(const string& p1, const string& p2);


bool canSplitMerrily(const Set<string>& people, int teamSize,
                     Vector<Set<string>>& teams) {
```

*(extra space for your answer to Problem Four, if you need it.)*

## Problem Five: Big-O and Efficiency (8 Points)

***The "Science" Part of Computer Science*** ***(Recommended time: 20 minutes)***

As part of a project you're working on in a research lab, you need to do some sort of data analysis (say, Analysis $X$) on a large data set. You find three software packages available online and try running them on some data sets you have lying around, which are of different sizes. When you run the three software packages on data sets of size 100, you get back the following runtimes:

- Package $A$ takes 124s.
- Package $B$ takes 11s.
- Package $C$ takes 21s.

You remember hearing that one of these software packages runs in time $O(n)$, one runs in time $O(n^2)$, and one runs in time $O(n^3)$, but for the life of you you can't remember which one is which.

i. **(3 Points)** Based on the above data, do you have enough information to make an educated guess about which algorithm runs in time $O(n)$, which runs in time $O(n^2)$, and which runs in time $O(n^3)$? If so, explain which algorithm has which runtime and why you have enough information to make the guess. If not, explain why you don't have enough information to make an educated guess.

Now, imagine you run the software packages on inputs of size 200, 300, 400, 500, 600, 700, and 800. Here's what you find:

| | Package *A* | Package *B* | Package *C* |
|---|---|---|---|
| 100 elements | 124s | 11s | 21s |
| 200 elements | 243s | 20s | 79s |
| 300 elements | 370s | 68s | 181s |
| 400 elements | 506s | 161s | 324s |
| 500 elements | 631s | 314s | 504s |
| 600 elements | 757s | 544s | 724s |
| 700 elements | 880s | 860s | 982s |
| 800 elements | 1013s | 1280s | 1276s |

ii. **(3 Points)** Based on the above data, do you have enough information to make an educated guess about which algorithm runs in time O($n$), which runs in time O($n^2$), and which runs in time O($n^3$)? If so, explain which algorithm has which runtime and why you have enough information to make the guess. If not, explain why you don't have enough information to make an educated guess.

iii. **(2 Points)** Based on the data you have available to you, if you had to make an educated guess about which software package you'd recommend to someone with a data set of size 5,000, which one would you recommend? Why?

# C++ Library Reference Sheet

**Lexicon**
```
Lexicon lex; Lexicon english(filename);
lex.addWord(word);
bool present = lex.contains(word);
bool pref = lex.containsPrefix(p);
int numElems = lex.size();
bool empty = lex.isEmpty();
lex.clear();
```

**Map**
```
Map<K, V> map = {{k1, v1}, … {kn, vn}};
map[key] = value; // Autoinsert
bool present = map.containsKey(key);
int numKeys = map.size();
bool empty = map.isEmpty();
map.remove(key);
map.clear();
Vector<K> keys = map.keys();
```

**Stack**
```
stack.push(elem);
T val = stack.pop();
T val = stack.top();
int numElems = stack.size();
bool empty = stack.isEmpty();
stack.clear();
```

**Queue**
```
queue.enqueue(elem);
T val = queue.dequeue();
T val = queue.peek();
int numElems = queue.size();
bool empty = queue.isEmpty();
queue.clear();
```

**Set**
```
Set<T> set = {v1, v2, …, vn};
set.add(elem);
set += 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();
```

**Vector**
```
Vector<T> vec = {v1, v2, …, vn};
vec.add(elem);
vec += elem;
vec.insert(index, elem);
vec.remove(index);
vec.clear();
vec[index]; // Read/write
int numElems = vec.size();
bool empty = vec.isEmpty();
vec.subList(start, numElems);
```

**TokenScanner**
```
TokenScanner scanner(source);
while (scanner.hasMoreTokens()) {
   string token = scanner.nextToken();
   …
}
scanner.addWordCharacters(chars);
```

**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);
```

**ifstream**
```
input.open(filename);
input >> val;
getline(input, line);
```

**GWindow**
```
GWindow window(width, height);
gw.drawLine(x0, y0, x1, y1);
pt = gw.drawPolarLine(x, y, r, theta);
```

**GPoint**
```
double x = pt.getX();
double y = pt.getY();
```

**General Utility Functions**
```
int getInteger(optional-prompt);
double getReal(optional-prompt);
string getLine(optional-prompt);
int randomInteger(lowInclusive,
                  highInclusive);
double randomReal(lowInclusive,
                  highExclusive);
error(message);
x = max(val1, val2); y = min(val1, val2);
stringToInteger(str); stringToReal(str);
integerToString(intVal);
realToString(realVal);
```