CS106B

Winter 2023

Practice Midterm Exam 1

This is a printable version of the first 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: Where's Waldo?

Where's Waldo? is a series of children's puzzle books. Each page consists of a picture of a huge crowd of people, and the goal is to locate an iconically-dressed man named Waldo.

In this problem, we'll deal with a simplified version of this problem. You'll be given a Grid<char>, and your objective will be to find Waldo, who will be represented as follows:

```
 o
 +
 ^
```

That is, you'll have Waldo's head, torso, and legs. Waldo will always be standing up in this orientation. For example, Waldo is hidden in this image. His feet are one row above the bottom and he's more or less centered horizontally.

```
]0Muz$Y]77*cM-8bcT0x+INC`D3pC3Wo1B9vyykoLL_K2z"k'4*H{(SQbmZ7U:`*=%c4@~?*}Jr_8'`o
q@!?hyZ?^<0bb|/oI+y^)Q~OLdxxx2CeGrq2Vy1+&<!/?CB(9nH^~A2rN3AuZ18$<o|6m^*u"YqZ~`[!
6:($uLn.RKw6u+U+/+UcaZ9#82/DsGU:=C^rc'Kf0*DR,.J.W(y<&S}6"W&}uiDh<vf:H3#LD>5doS5}
:JJ2l9kyhmt-W^apa[}+QK[e1{\s*{3[CatLt*h9bf0-W'IL~\C-\h&LV<EKbZQz#js`M4NAV,g=9E%x
q3#dm7E^QH4G/_6Kupqb8=DrI9>#]&UaE'b6:q30m=$J\{l#,$^S$XfA3B[x1k.O%;Y>CN,YtD_wio1j
226/^{y%?-I`d||:GP;$#2RR(*]-ZW9qv{YrR-}b_|W)=\je{M_N*T/*6,o':7CJK#Ib_N/'kz(Mrs4Z
&uA&ACkex"JEsk_DjD6smK]JhnWv%,-.}^g?!kH7WL#]GM0u(.GD~=[B7"-ciHNWbp;KNdc%hcMl>\g*
r#M>E{jrsA[zehuAnFh(brw6@e2(M:,$ROrXN{^JcTiBeLziBDN`[l2Z^[*$b,RtC]'@!+?B>o&9I2Pr
IYh`!cr7*6y(7bS`%OH1bO&"]-"b)NZ52+j`[rly=?V=0H}O]dL#<`3q#?#=$;X7OA\S5Y2/h<KFOuu5
!Ink6FR2Y)y7-2|IuY$i=k/KXe`nNZr+3iHQ6kSD>vv:E{~~Pw?V1{om=}a0kzDFlp(O{I`:Y&\mOxY$
Y{IX{O<CjP-vJa{*%sp&yI,p9AC]dIJ>PdIAMCpos2lDt_`6g14r{FdC^w?5Dj]Z!zosI=,HZg1e6/nv
ldV\0YSuU7RcTV-EFQ&XW0\`.|_$PtsZ6M%~id={AGB2P"gt*-rV4d:zR^w@0*vXk8@;YL~5S"-eJj7o
U~2jgUx]tor\7@"Ek=&`ndlwUUXme*J26J9Hc8`j|DbGJ;\V$XrzW'-\#\K8~Cgz:![yN%]#u/*pnj%0
S)qJBi1TK2)Y"4'4b%E_I8`a"@M{?Yo!5Yjq\d=>0r.-+lQnzSNsRgCHKO\d=9D7[DC(=cz1GrJ,rt{P
.c]HkfY1$Pn3*o?F;sCOEEuZQQjOnSZ~K_wgyQtQaj;F^OC<Gx$nA3X_n))YB73PvHakcL(]Rn|qM9{B
p%Op3TLp]d7+:B,ttA(lIzc[uO,BQq2-h.cY[{4Xi'lT]oAZBSlcIVj{,V(xSH[o="xRt?*>c[J'')V!
r}rOz*cyyihcUQ1|ZmM|Z}0g-$5Nk~E0i|e?2bbEzoGNOVUAPiDpG5t-hIkFnau\2Ei=4e<KvR{sH*>@
:g#|a+SkCtwW#;:1a(^[Qr5Cb/4I@&ErYv!PJOb]V+F5X>{4SkVJr=\,nEXd4V,e@.&qUww1k'Oj4=Px
@|uYp^3]Xk6%R1aXMyZ"X`}v|z_2.<f@pT$N@\xq&^b]g8QF@!I!&4uzi1~L]`j.!!cHRilB"(@H+9sV
]OjYdh86;'ilarr>P0JOqqG_c@YD)!C_j5S&Q4hBlvTl:gp:#eF,}[y(s2R@qM0MU[D-&XT/|EKTZaO+
```

There may be one Waldo in an image, or multiple Waldos, or perhaps none.

Your task is to write a function

```
Vector<GridLocation> findWaldosIn(const Grid<char>& picture);
```

that takes as input a `Grid<char>` representing the puzzle, then returns a `Vector<GridLocation>` that holds the locations of each Waldo's torso (the + character making up the Waldo).

```
Vector<GridLocation> findWaldosIn(const Grid<char>& picture) {
```

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.

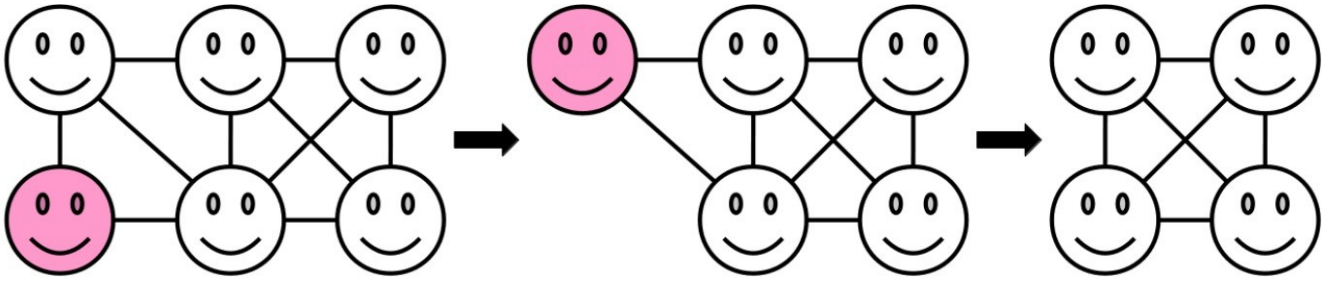
Problem Two: The Core

Social networks – Facebook, LinkedIn, Instagram, etc. – are popular *because* they're popular. The more people that are on a network, the more valuable it is to join that network. This means that, generally speaking, once a network starts growing, it will tend to keep growing. At the same time, this means that if many people start leaving a social network, it will tend to cause other people to leave as well.

Researchers who study social networks, many of whom work in our CS department, have methods to quantify how resilient social networks are as people begin exiting. One way to do this is to look at the ***k-core*** of the network. Given a number *k*, you can find the *k*-core of a social network as follows:

- If everyone has at least *k* friends on the network, then the *k*-core consists of everyone on that network.
- Otherwise, there's some person who has fewer than *k* friends on the network. Delete them from the network and repeat this process.

For example, here's how we'd find the 3-core of the social network shown to the left. At each step, we pick a person with fewer than three friends in the network (shown in red) and remove them. We stop once everyone has three or more friends, and the remaining people form the 3-core.



Intuitively, the *k*-core of a social network represents the people who are unlikely to leave – they're tightly integrated into the network, and the people they're friends with are tightly integrated as well.

Your task is to write a function:

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

that takes as input a social network (described below) and a number *k*, then returns the set of people in the *k*-core of that network.

Here, the social network is represented as a `Map<string, Set<string>>`, where each key represents a person and each value is the set of people they're friends with. You can assume that each person listed as a friend actually exists in the network and that friendship is symmetric: if *A* is friends with *B*, then *B* is also friends with *A*.

In case it helps, you can assume no person's name is the empty string.

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

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: Mongolian Recursion

Consider the following recursive functions, `selenge` and `zavkhan`, which differ only in their last lines.

```
string selenge(const string& str) {
  if (str.length() <= 1) {
    return str;
  } else {
    string temp;

    /* Append these characters to the
     * temp string.
     */
    temp += str[1];
    temp += str[0];

    /* This next line means "get the
     * string formed by dropping the
     * first two characters from str."
     */
    string next = str.substr(2);

    /* The next line differentiates
     * selenge from zavkhan.
     */
    return temp + selenge(next);
  }
}
```

```
string zavkhan(const string& str) {
  if (str.length() <= 1) {
    return str;
  } else {
    string temp;

    /* Append these characters to the
     * temp string.
     */
    temp += str[1];
    temp += str[0];

    /* This next line means "get the
     * string formed by dropping the
     * first two characters from str."
     */
    string next = str.substr(2);

    /* The next line differentiates
     * zavkhan from selenge.
     */
    return zavkhan(next) + temp;
  }
}
```

Answer each of the following questions and fill in the appropriate blanks. No other justification is needed. Incorrect answers will not receive points, but otherwise there is no penalty for an incorrect guess.

i. How many choices of string `s` are there where `selenge(s)` returns `"bulgan"`?

- ☐ There are no choices of `s` for which this happens.
- ☐ There is exactly one choice of `s` where this happens, namely `s` = "_____".
- ☐ There are two or more choices of `s` where this happens, such as "_____" and "_____".

ii. How many choices of string `s` are there where `zavkhan(s)` returns `"bulgan"`?

- ☐ There are no choices of `s` for which this happens.
- ☐ There is exactly one choice of `s` where this happens, namely `s` = "_____".
- ☐ There are two or more choices of `s` where this happens, such as "_____" and "_____".

iii. How many choices of string `s` are there where `selenge(s)` returns `"khovd"`?

- ☐ There are no choices of `s` for which this happens.
- ☐ There is exactly one choice of `s` where this happens, namely `s` = "_____".
- ☐ There are two or more choices of `s` where this happens, such as "_____" and "_____".

iv. How many choices of string `s` are there where `zavkhan(s)` returns `"khovd"`?

- ☐ There are no choices of `s` for which this happens.
- ☐ There is exactly one choice of `s` where this happens, namely `s` = "_____".
- ☐ There are two or more choices of `s` where this happens, such as "_____" and "_____".

Problem Four: Parking Randomly

In this problem, you'll code up a simulation of cars parking along a curb that yields a surprising result.

Let's suppose you have a curb that is exactly *n* car lengths long. Someone comes along and parks at a uniformly random point along the curve where they fit. Then, another person comes along and parks at a uniformly random remaining point on the curve where they can fit (without hitting the first car), then another, then another, etc. For the sake of simplicity, we'll assume that a car can fit anywhere on the curb where there's at least one car length of space. (This isn't realistic, as anyone who's tried parallel parking in a dense city can tell you! But it'll make things easier for us to work with.)

At some point, the curb will be so filled with cars that no more cars can park. In an ideal scenario, given a curb of length *n*, you'd be able to fit *n* cars. But with cars arriving randomly, this isn't the case, since gaps between the cars will take up space unnecessarily.

Write a function

```
int parkRandomly(double length);
```

that takes as input the length of the curb, measured in car lengths, then parks cars along the curb uniformly at random until no cars fit. The function should then return the total number of cars that were able to park along the curb.

Once you've done that, write a function

```
double averageParkingDensity(double length);
```

that runs a large number of simulations of cars parking along a curb that is length car lengths long, then returns the ratio between the average number of cars that actually could park along the curb and the maximum number of cars that could conceivably park there.

```
int parkRandomly(double length) {
```

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.

```
double averageParkingDensity(double length) {
```

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

Problem Five: Splitting the Bill

You've gone out for coffees 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 who ordered something, 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: $2, B: $1, C: $1
A: $2, B: $0, C: $1
…
A: $0, B: $1, C: $3
A: $0, B: $0, C: $4

Some notes on this problem:

- The total amount owed will always be nonnegative. If the total owed is negative, you should use the error() function to report an error.
- There is always at least one person in the set of people. If not, you should report an error.
- 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) {
```

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.