Midterm Practice Problems

Here’s a collection of practice problems you can work through to prepare for the upcoming midterm exam. These questions are taken from past CS106B midterms, which, it should be noted, were not given in the same format as the upcoming exam. However, we think that they’re nonetheless excellent practice for the upcoming exam.

1) Containers and Flightless Birds

Consider the following functions, **dodo** and **kiwi**, which differ only in the highlighted areas:

```
string dodo(const string& str, int n) {
  Queue<char> storage;

  /* Visit the characters in str in the
   * order in which they appear.
   */
  for (char ch: str) {
    storage.enqueue(ch);
  }

  for (int i = 0; i < n; i++) {
    char ch = storage.dequeue();
    storage.enqueue(ch);
  }

  /* If you don't initialize a string
   * in C++, it defaults to empty.
   */
  string result;
  while (!storage.isEmpty()) {
    result += storage.dequeue();
  }

  return result;
}
```

```
string kiwi(const string& str, int n) {
  Stack<char> storage;

  /* Visit the characters in str in the
   * order in which they appear.
   */
  for (char ch: str) {
    storage.push(ch);
  }

  for (int i = 0; i < n; i++) {
    char ch = storage.pop();
    storage.push(ch);
  }

  /* If you don't initialize a string
   * in C++, it defaults to empty.
   */
  string result;
  while (!storage.isEmpty()) {
    result += storage.pop();
  }

  return result;
}
```

Answer each of the following questions and *fill in the appropriate blanks*. No other justification is needed.

i. How many strings s are there where dodo(s, 3) returns "penguin"?

- 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 strings s are there where kiwi(s, 3) returns "penguin"?

- 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 n are there for which dodo("kakapo", n) returns "kakapo"?

- There are no choices of n for which this happens.
- There is exactly one choice of n for which this happens, namely n = ________.
- There are two or more choices of n for which this happens, such as ________ and ________.

iv. How many choices of n are there for which kiwi("kakapo", n) returns "kakapo"?

- There are no choices of n for which this happens.
- There is exactly one choice of n for which this happens, namely n = ________.
- There are two or more choices of n for which this happens, such as ________ and ________.

Solution

2) 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.

Part 1: Selecting a Container

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: ____________________.

Here is my justification (at most fifty words):

Part 2: Creating the Index

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 the **`tokenize`** function that we gave you in Assignment 3:

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

Just to make sure you didn't miss it, ***your index should only contain words***, not spaces, punctuation, numbers, etc. Similarly to Assignment 3, you can assume that a string is a word if its first character is a letter. You can check if a **`char`** is a letter by using the **`isalpha`** function.

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**.

Solution

3) 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.

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 "________".

Solution

4) Doing the Splits

Your ultimate goal in this problem is to write a function `Set<Vector<string>> splitsOf(const string& str);` that takes as input a string, then returns all ways of splitting that string into a sequence of nonempty strings called **pieces**. For example, given the string "RUBY", you'd return a `Set` containing these `Vector<string>`s; notice that all letters are in the same relative order as in the original string:

```
{"R", "U", "B", "Y"}
{"R", "U", "BY"}
{"R", "UB", "Y"}
{"R", "UBY"}
{"RU", "B", "Y"}
{"RU", "BY"}
{"RUB", "Y"}
{"RUBY"}
```

If you take any one of these `Vector`s and glue the pieces together from left to right, you'll get back the original string "RUBY". Moreover, every possible way of splitting "RUBY" into pieces is included here. Each character from the original string will end up in exactly one piece, and no pieces are empty.

Given the string "TOPAZ", you'd return a `Set` containing all of these `Vector<string>`s:

```
{"T", "O", "P", "A", "Z"}
{"T", "O", "P", "AZ"}
{"T", "O", "PA", "Z"}
{"T", "O", "PAZ"}
{"T", "OP", "A", "Z"}
{"T", "OP", "AZ"}
{"T", "OPA", "Z"}
{"T", "OPAZ"}
{"TO", "P", "A", "Z"}
{"TO", "P", "AZ"}
{"TO", "PA", "Z"}
{"TO", "PAZ"}
{"TOP", "A", "Z"}
{"TOP", "AZ"}
{"TOPA", "Z"}
{"TOPAZ"}
```

As before, notice that picking one of these `Vector`s and gluing the pieces in that `Vector` back together will always give you "TOPAZ".

The decision tree for listing subsets is found by repeatedly considering answers to questions of the form "should I include or exclude this element?" The decision tree for listing permutations is found by repeatedly considering answers to questions of the form "which element should I choose next?" In this problem, the decision tree is found by repeatedly considering answers to this question: **How many characters from the front of the string should I include in the next piece?** Based on this insight, draw the decision tree for listing all splits of the string "JET" along the lines of the decision trees we drew in class. At a minimum, please be sure to do the following:

- Label each entry in the decision tree with the arguments to the recursive call it corresponds to.
- Label each arrow in the decision tree with what choice it corresponds to. Now, implement the `splitsOf` function. For full credit, your implementation must be recursive and match the decision tree that you drew in the first part of this problem.

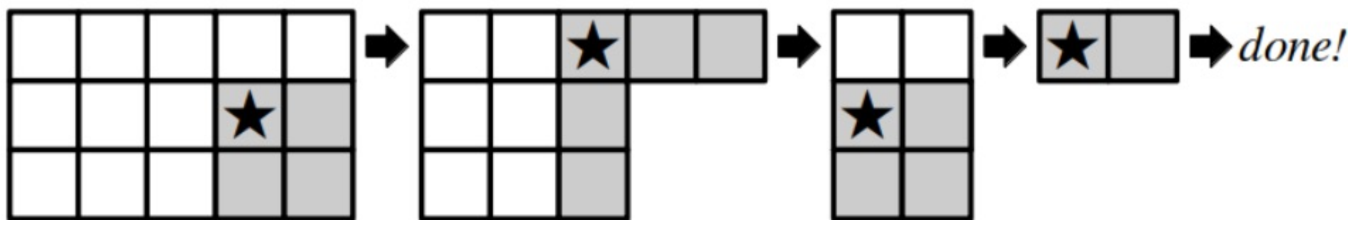
Solution

5) Eating a Chocolate Bar

You have a chocolate bar that's subdivided into individual squares. You decide to eat the bar according tothe following rule:

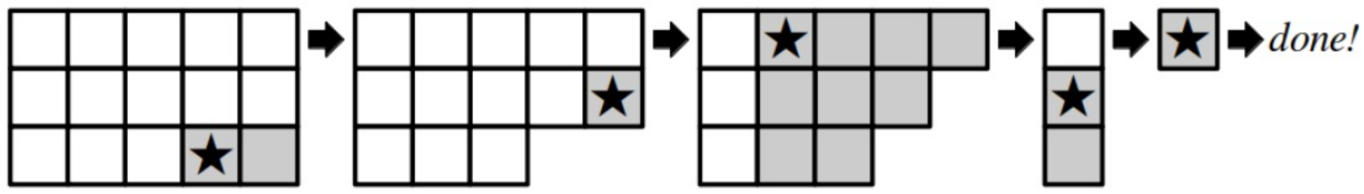
if you choose to eat one of the chocolate squares, you have to also eat every square below and/or to the right of that square.

For example, here's one of the many ways you could eat a 3 x 5 chocolate bar while obeying the rule. The star at each step indicates the square chosen out of the chocolate bar, and the gray squares indicate which squares must also be eaten in order to comply with the above rule.



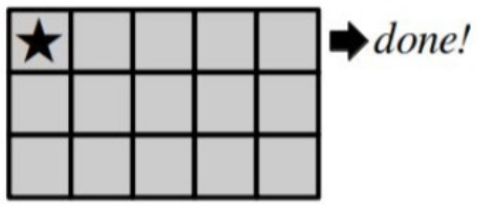
The particular choice of the starred square at each step was completely arbitrary, but once a starred square is picked the choice of grayed-out squares is forced. You have to eat the starred square, plus each square that's to the right of that square, below that square, or both.

The above route isn't only one way to eat the chocolate bar. Here's another:



As before, there's no particular pattern to how the starred squares were chosen, but once we know which square is starred the choice of gray squares is forced.

One more example. If you are so hungry you can't wait, you could eat the chocolate bar this way:



This eats the entire bar in a single bite. Ah, gluttony.

Your task is to write a function

```
int waysToEat(int numRows, int numCols);
```

that returns the number of different ways you could eat a **numRows × numCols** chocolate bar while obeying the above rule. Your **waysToEat** function will almost certainly be a wrapper around a different recursive function that actually computes the result. Think about what information you need to keep track of and what type or types you might use to represent that information. Some notes on this problem:

- You can assume that **numRows** and **numCols** cannot be negative and don't need to worry about what happens in this case. However, **numRows** and/or **numCols** may be zero.

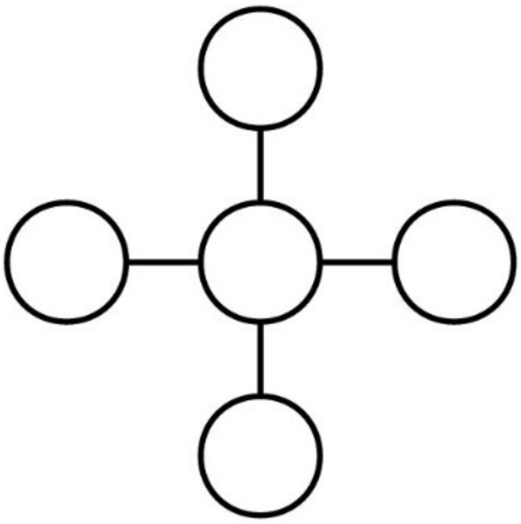
- You are encouraged to write as many helper functions as you'd like. To save you time, you don't need to write function prototypes.
- The chocolate bar never flips or rotates; the square in the upper-left corner will stay there until you've eaten the whole bar.
- You do not need to worry about efficiency.
- Your solution must be recursive. That's kinda what we're testing here.

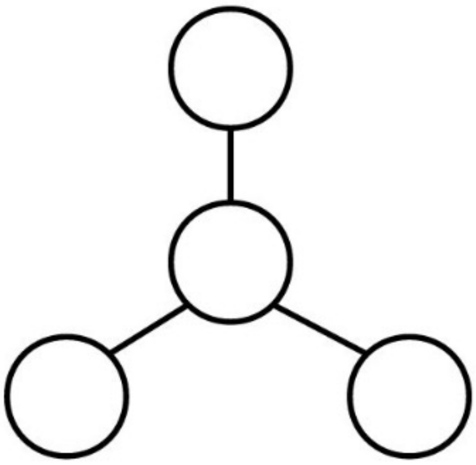
Solution

6) No More Counting Dollars, We'll Be…

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.

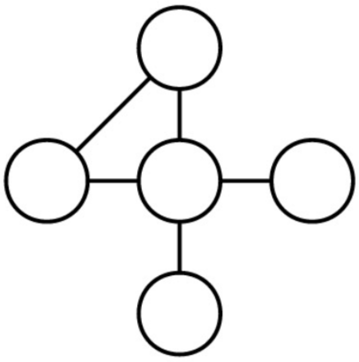
A **star** is a cluster of four or more cities arranged as follows: there's a single city in the center that's connected to all the other cities in the cluster, and every other city in the cluster is only connected to the central city. Here are some stars shown below:

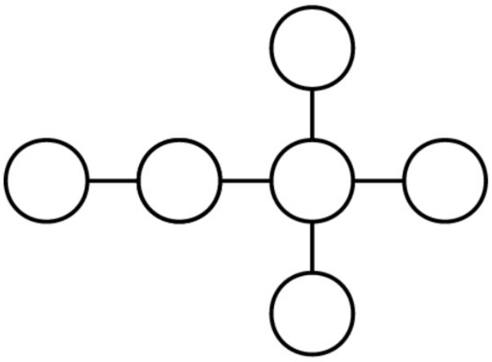


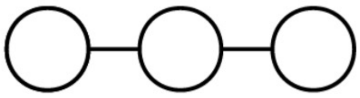


Stars commonly arise in road networks in smaller island nations: you have a main, central city (often acapital city) and a bunch of smaller, outlying towns.

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(constMap<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.

Solution

7) Proofreading a Report

You've been working on preparing a report as part of a larger team. Each person on the team has been tasked with writing a different section of the report. To make sure that the final product looks good and is ready to go, your team has decided to have each person in the team proofread a section that they didn't themselves write.

There are a lot of ways to do this. For example, suppose your team has five members conveniently named A, B, C, D, and E. One option would be to have A read B's section, B read C's section, C read D's section,D read E's section, and E read A's section, with everyone proofreading in a big ring. Another option would be to have A and B each proofread the other's section, then have C proofread D's section, D read E's section, and E read C's section. A third option would be to have A read E's work, E read C's work, and C read A's work, then to have B and D proofread each other's work. The only restrictions are that (1) each section needs to be proofread by exactly one person and (2) no person is allowed to proofread their ownwork.

Write a function

that lists off all ways that everyone can be assigned a person's work to check so that no person is assigned to check their own work. For example, given the five people listed above, this function might print the following output:

A checks B, B checks A, C checks E, D checks C, E checks D
A checks B, B checks A, C checks D, D checks E, E checks C
A checks C, B checks A, C checks B, D checks E, E checks D
(... many, many lines skipped ...)
A checks C, B checks E, C checks D, D checks B, E checks A
A checks D, B checks C, C checks E, D checks B, E checks A
A checks E, B checks C, C checks D, D checks B, E checks A

Some notes on this problem:

- You're free to list the proofreading assignments in any order that you'd like. However, you should *make sure that you don't list the same assignment twice*.
- Your function should print all the arrangements it finds to **cout**. It shouldn't return anything.
- Your solution needs to be recursive – that's kinda what we're testing here.
- While in general you don't need to worry about efficiency, you should *not* implement this function by listing all possible permutations of the original group of people and then checking each one to see whether someone is assigned to themselves. That ends up being a bit too slow to be practical.
- *Your output doesn't have to have the exact same format as ours*. As long as you print out something that makes clear who's supposed to proofread what, you should be good to go. In case it helps, youmay want to take advantage of the fact that you can use **cout** to directly print out a container class (**Vector, Set, Map, Lexicon, etc.**). For example, if you have a **Map** named **myMap**, the C++ code __cout « myMap « endl; __ prints out all the key/value pairs.
- As a hint, focus on any one person in the group. You know that they're going to have to proofread some section. Consider exploring each possible way they could do so.

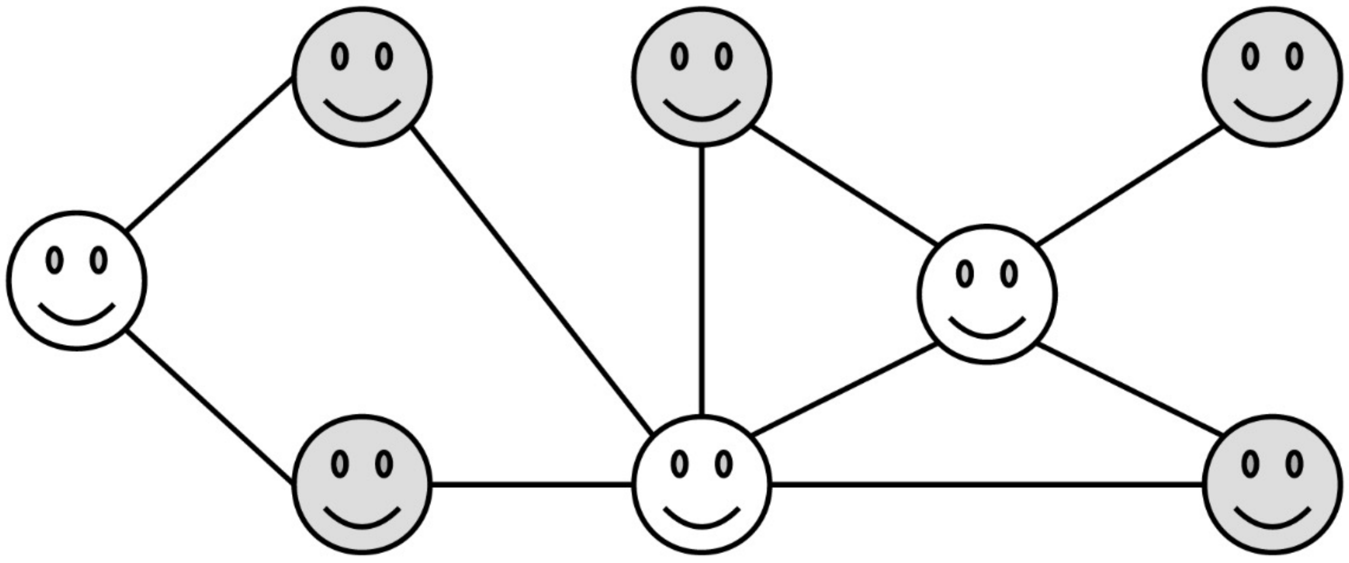
```
void listAllProofreadingArrangements(const Set<string>& people);
```

Solution

8) Avoiding Sampling Bias

One of the risks that comes up when conducting field surveys is *sampling bias*, that you accidentally survey a bunch of people with similar backgrounds, tastes, and life experiences and therefore end up with a highly skewed view of what people think, like, and feel. There are a number of ways to try to control for this. One option that's viable given online social networks is to find a group of people of which no two are Facebook friends, then administer the survey to them. Since two people who are a part of some similar organization or group are likely to be Facebook friends, this way of sampling people ensures that you geta fairly wide distribution.

For example, in the social network shown below (with lines representing friendships), the folks shaded in gray would be an unbiased group, as no two of them are friends of one another.



Your task is to write a function

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

that takes as input a **Map** representing Facebook friendships (described later) and returns the largest group of people you can survey, subject to the restriction that you can't survey any two people who are friends.

The **network** parameter represents the social network. Each key is a person, and each person's associ-ated value is the set of all the people they're Facebook friends with. You can assume that friendship issymmetric, so if person *A* is a friend of person *B*, then person *B* is a friend of person *A*. Similarly, you can assume that no one is friends with themselves.

Some other things to keep in mind:

- You need to use recursion to solve this problem – that's what we're testing here.
- Your solution must not work by simply generating all possible groups of people and then checking at the end which ones are valid (i.e. whether no two people in the group are Facebook friends). This approach is far too slow to be practical.

Solution

9) Predecessor Maps

Google Translate is powered, in large part, by a technique called **word2vec** that builds an understanding of a language by looking at the context in which each word occurs.

Imagine you have a word like "well." There are certain words that might reasonably appear immediately before the word “well” in a sentence, like “feeling,” “going,” “reads,” etc., and some words that that are highly unlikely to appear before “well,” like “cake,” “circumspection,” and “election.” The idea behind word2vec is to find connections between words by looking for pairs of words that have similar sets of words preceding them. Those words likely have some kind of connection between them, and the rest of the logic in word2vec works by trying to discover what those connections are.

Your task is to write a function `Map<string, Lexicon> predecessorMap(istream& input);` that takes as input an istream& containing the contents of a file, then returns a `Map<string, Lexicon>` that associates each word in the file with all the words that appeared directly before that word. For example, given JFK’s quote “Ask not what your country can do for you; ask what you can do for your country,” your function should return a `Map` with these key/value pairs:

"not" : { "ask" }
"what" : { "not", "ask" }
"your" : { "what", "for" }
"country" : { "your" }
"can" : { "country", "you" }
"do" : { "can" }
"for" : { "do" }
"you" : { "for", "what" }
"ask" : { "you" }

Notice that although the word “ask” appears twice in the quote, the first time it appears it’s the first word in the file and so nothing precedes The second time it appears, it’s preceded by some whitespace and a semicolon, but before that is the word “you,” which is what ultimately appears in the `Lexicon`.Some notes on this problem:

- You can assume that a token counts as a word if its first character is a letter. You can use the `isalpha` function to check if a character is a letter.
- You can assume you have access to the function `Vector<string> tokenize(const string& input);` that we provided you in Assignment 3.
- Your code should be case-insensitive, so it should return the same result regardless of the capitalization of the words in the file. The capitalization of the keys in the map is completely up to you.
- Your code should completely ignore non-word tokens (whitespace, punctuation, quotation marks, etc.) and just look at the words it encounters.
- It is not guaranteed that the file has any words in it, and there’s no upper bound on how big the file can be.

Solution

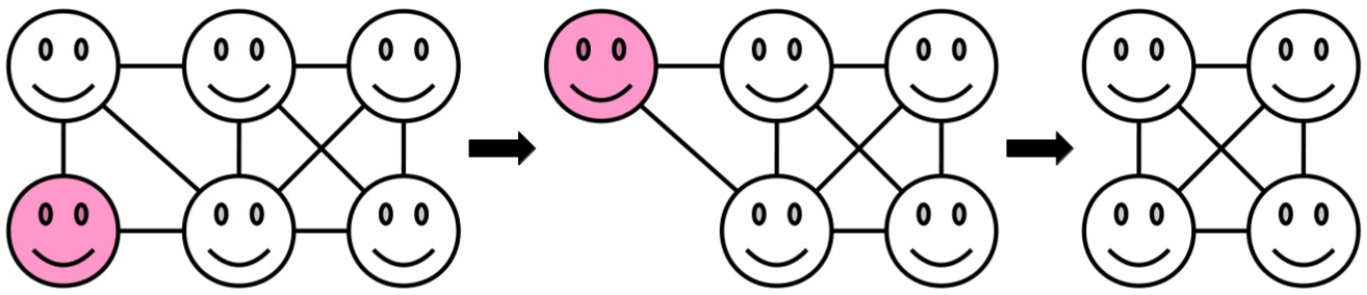
10) The Core

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*.

Some notes on this problem:

- You're free to implement this one either iteratively or recursively. It's up to you to decide.
- Normally, we'd ask you to do error-checking, but for the purposes of this problem you can assumethat $k \geq 0$ and you don't need to validate this.
- It's possible that someone in the network has no friends. That person would be represented as a key in the map associated with an empty set.
- You can assume no one is their own friend.
- While you don't need to come up with the most efficient solution possible, you should avoid solution routes that are unnecessarily slow. For example, don't list off all possible sets of people, then check whether each of them is the *k-core*.
- In case it helps, you can assume no person's name is the empty string.

Solution

11) Cosmic Care Packaages

At present, it costs around $5,000 to place one kilogram of material into orbit. This means that if you're floating on the International Space Station, you probably shouldn't expect that much out of your meals. They're calibrated to be nutritionally complete and lightweight, and while there's some effort made to provide nice things like "flavor" and "texture," that's not always possible.

Let's imagine that you want to be nice and send a cosmic care package up to the ISS with some sweets for the crew to share. Every penny matters at five grand a kilo, so you'll want to be very sure that what you send up into the vast abyss will actually make the folks there happy. You have a list of the desserts that each individual ISS crew member would enjoy. What's the smallest collection of treats you could send up that would ensure everyone gets something they like?

For example, suppose the ISS crew members have the following preferences, with each row representing one person's cravings:

{ "Pumpkin pie", "Revani", "Sufganiyot" }
{ "Castella", "Kheer", "Melktert" }
{ "Po'e", "Tangyuan", "Tres leches" }

{ "Apfelstrudel", "Tangyuan" }
{ "Alfajores", "Brigadeiro", "Revani" }
{ "Baklava", "Revani" }
{ "Melktert" }
{ "Brownies", "Cannolis", "Castella", "Po'e", "Revani" }
{ "Cendol", "Kashata", "Tangyuan" }
{ "Baklava", "Castella", "Kheer", "Revani" }

The smallest care package that would cheer up the whole crew would have three items: revani, melktert,and tangyuan. Anything smaller than this will leave someone unhappy.

Write a function:

```
Set<string> smallestCarePackageFor(const Vector<Set<string>>& prefer-ences);
```

that takes as input a list containing sets of what treats each crew member would like, then returns the smallest care package that would make everyone on the ISS happy.

While in principle you could solve this problem by listing off all possible subsets of treats and seeing which ones satisfy everyone's sweet tooths, this approach is probably not going to be very fast if there area lot of different choices for sweets. Instead, use the following approach: pick a person who hasn't yet had anything sent up that would make them happy, then consider each way you could send them something that would fill them with joy.

Some notes on this problem:

- Yep, you guessed it! You have to do this one recursively.
- To clarify, if you send up some item to space, you can assume there's enough of it for everyone on the space station to share. Sending up revani, for example, means sending up enough revani for each crew member to have a piece.
- Each person's preferences will contain at least one item. There is no fixed upper size to how many preferences each person can have.
- If there are multiple care packages that are tied for the smallest, you can return any of them.
- In case it helps, you can assume no treat's name is the empty string.

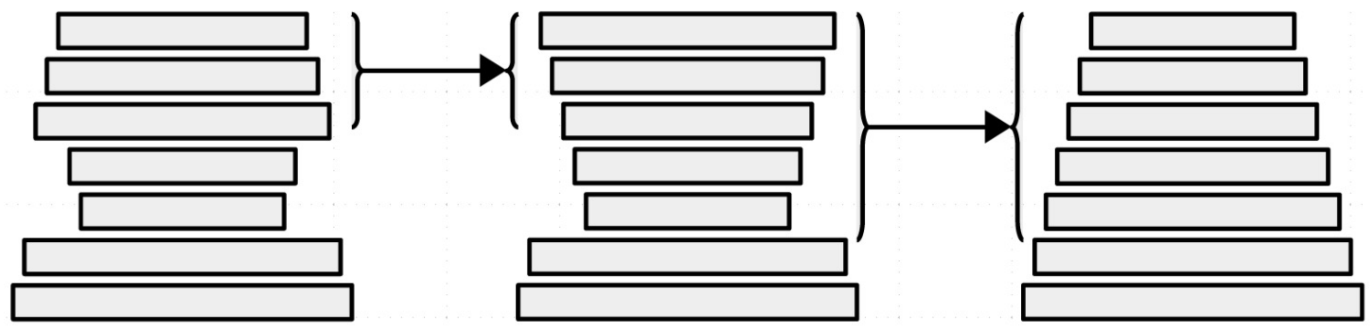
Solution

12) Mmmmm... Pancakes!

Bill Gates is famous for a number of things – for his work at Microsoft, for his wide-scale philanthropy through the Bill and Melinda Gates Foundation, and, oddly enough, for a paper he co-authored about flip-ping pancakes.

Imagine you have a stack of pancakes of different diameters. In an ideal pancake stack, the kind of pancake stack you'd see someone photograph and put on Instagram, the pancakes would be stacked in decreasing order of size from bottom to top. That way, all the delicious syrup and melted butter you put ontop of the pancakes can drip down onto the rest of the stack. Otherwise, you'll have a bigger pancake ontop of a smaller one, serving as an edible umbrella that shields the lower pancakes from all the deliciousness you're trying to spread onto them.

The problem with trying to get a stack of pancakes in sorted order is that it's really, really hard to pull a single pancake out of a stack and move it somewhere else in the stack. On the other hand, it's actually not that difficult to slip a spatula under one of the pancakes, pick up that pancake and all the pancakes above it, then flip that part of the stack over. For example, in the illustration below, we can flip the top three pancakes in the stack over, then flip the top five pancakes of the resulting stack over:



Your task is to write a function

```
bool canSortStack(Stack<double> pancakes, int numFlips, Vector<int>& flipsMade);
```

that accepts as input a `Stack<double>` representing the stack of pancakes (with each pancake represented by its width in cm) and a number of flips, then returns whether it's possible to get the stack of pancakes into sorted order making at most the specified number of flips. If so, your function should fill in the **flipsMade** out parameter with the sequence of flips that you made, with each flip specified as how many pancakes off the top of the stack you flipped. For example, the above illustration would correspond to the flip sequence {3, 5}, since we first flipped the top three pancakes, then the top five pancakes.

Here are some notes on this problem:

- Treat this as a backtracking problem – try all possible series of flips you can make and see whether you can find one that gets everything into sorted order. As a result, don't worry about efficiency.
- Your solution must be recursive – again, that's what we're trying to test.
- You can assume the pancakes all have different widths and that those widths are positive.
- The **Stack** here is ordered the way the pancakes are – the topmost pancake is at the top of the stack, and the bottommost pancake is at the bottom.

You can assume **flipsMade** is empty when the function is first called, and its contents can be whatever you'd like them to be if you can't sort the pancake stack in the given number of flips.

Solution

13) PetsMatch

Suppose that you own a pet store and want to help customers find their ideal pet. To do so, you create a list of all the pets in the store, along with all of the adjectives which best describe them. For example, you might have this list of pets and their adjectives:

```
Ada           Happy, Loving, Fuzzy, Big, Tricksy
Babbage       Loving, Tiny, Energetic, Quiet
Lucy          Happy, Loving, Big
Larry         Happy, Fuzzy, Tiny, Tricksy
Abby          Loving, Big, Energetic
Luba          Happy, Loving, Tiny, Quiet, Tricksy
Mitzy         Fuzzy, Big, Energetic, Quiet
```

If a customer gives you a list of adjectives, you can then recommend to them every pet that has all of those adjectives. For example, given the adjectives "Happy" and "Tricksy," you could recommend Ada, Larry, and Luba. Given the adjective "Fuzzy," you could recommend Ada, Larry, and Mitzy. However, given the adjectives "Energetic," "Quiet," and "Fuzzy," and "Tiny," you could not recommend any pets at all.

Write a function:

```
Set<string> petsMatching(const Map<string, Set<string>>& adjectiveMap,
                         const Vector<string>& requirements);
```

that accepts as input a **Map<string, Set<string>>** associating each pet with the adjectives that best describe it, along with a customer's requested adjectives (represented by a **Vector<string>**), then returns a **Set<string>**holding all of the pets that match those adjectives.

There might not be any pets that match the requirements, in which case your function should return an empty set. You can assume that all adjectives have the same capitalization, so you don't need to worry about case-sensitivity. Also, if a client does not include any adjectives at all in their requirements, you should return a set containing all the pets, since it is true that each pet matches all zero requirements. Finally, your function should not modify any of the input parameters.

Solution

14) Shrinkable Words Revisited

Recall from lecture that a shrinkable word is a word that can be reduced down to a single letter by removing one letter at a time, at each step leaving a valid English word.

In lecture, we wrote a function **isShrinkableWord** that determined whether a given string was a shrinkable word. Initially, this function just returned true or false. This meant that if a word was indeed shrinkable, we would have to take on faith that the word could indeed be reduced down to a single letter.

To help explain why a word was shrinkable, our second version of the function additionally produced a **Stack<string>** showing the series of words one would go through while shrinking the word all the way down to a single letter. (In reality, our lecture example used a **Vector<string>** rather than a **Stack<string>**, but a **Stack<string>** is actually a better choice.) Let's call a sequence of this sort a shrinking sequence.

However, let's suppose that you're still skeptical that the **Stack<string>** produced by the function actually is a legal shrinking sequence for the word. To be more thoroughly convinced that a word is shrinkable, you decide to write a function that can check whether a given **Stack<string>** is indeed a shrinking sequence for a given word.

Write a function:

```
bool isShrinkingSequence(const string& word,
                         const Lexicon& english,
                         Stack<string> path);
```

that accepts as input a word, a **Lexicon** containing all words in English, and a **Stack<string>** containing an alleged shrinking sequence for that word, then returns whether the **Stack<string>** is indeed a legal shrinking sequence for that word. For example, given the word "pirate" and the stack

```
 pirate
  irate
  rate
  rat
  at
  a
---------
```

Your function would return **true**. However, given any of these stacks:

```
 pirate           avast
  irate
  rate
```

```
pirate
rate
at
---------
```

```
---------
```

Your function would return **`false`** (the first stack is not a shrinking sequence because **`"te"`** and **`"e"`** are not words; the second is a legal shrinking sequence, but not for the word **`"pirate"`**; the third is not a shrinking sequence because it skips words of length 1, 3, and 5; and the fourth is not a shrinking sequence because it contains no words at all).

You can assume that **`word`** and all the words in **`path`** consist solely of lower-case letters.

Feel free to tear out this page as a reference, and write your solution on the next page.

Solution

15) Ddeeddoouubblliinngg

(This excellent problem by Eric Roberts.)

In the early part of the 20th century, there was considerable interest in both England and the United States in simplifying the rules used for spelling English words, which has always been a difficult proposition. One suggestion advanced as part of this movement was the removal of all doubled letters from words. If this were done, no one would have to remember that the name of the Stanford student union is spelled “Tresidder,” even though the incorrect spelling “Tressider” occurs at least as often. If doubled letters were banned, everyone could agree on “Tresider.”

Write a ***recursive*** function

```
string removeDoubledLetters(const string& str);
```

that takes a string as its argument and returns a new string with any consecutive substring consisting of repeated copies of the same letter replaced by a single copy letter of that letter. For example, if you call

```
removeDoubledLetters("tresidder")
```

your function should return the string **`"tresider"`**. Similarly, if you call

```
removeDoubledLetters("bookkeeper")
```

your function should return **`"bokeper"`**. And because your function compresses strings of multiple letters into a single copy, calling

```
removeDoubledLetters("zzz")
```

should return **`"z"`**.

In writing your solution, you should keep the following points in mind:

- Your function should not try to consider the case of the letters. For example, calling the function on the name "Lloyd" should return the argument unchanged because 'L' and 'l' are different letters.
- Your function must be purely recursive and may not make use of any iterative constructs such as for or while.

Solution

16) A Question of Balance

Your job in this problem is to write a function

```
Set<string> balancedStringsOfLength(int n);
```

that accepts as input a nonnegative number n, then returns a **`Set<string>`** containing all strings of exactly n pairs of balanced parentheses.

As examples, here is the one string of one pair of balanced parentheses:

()

Here are the two strings of two pairs of balanced parentheses:

(())   ()()

Here are all five strings of three pairs of balanced parentheses:

((()))   (()())   (())()   ()(())   ()()()

And here are the fourteen strings of four pairs of balanced parentheses:

(((())))	((()()))	((())())	(()(()))	(()()())
((()))()	(()())()	(())(())	(())()()	()((()))
()(()())	()(())()	()()(())	()()()()	

As a hint, you might find the following observation useful: any string of n > 0 pairs of balanced parentheses can be split apart as follows:

(*some-string-of-balanced-parentheses*) *another-string-of-balanced-parentheses*

You might find it useful to try splitting apart some of the above strings this way to see if you understand why this works.

Write your solution on the next page, and feel free to tear out this page as a reference.

Solution