CS106B Handout 27
Winter 2017 March 20, 2017

CS106B Final 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. Please write all of your solutions on this physical copy of the exam.

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 exam is "self-contained" in the sense that if you're expected to reference code from the lectures, textbook, assignments, or section handouts, we'll explicitly mention what that code is and provide sufficient context for you to use it. There's a reference sheet at the back of the exam detailing the library functions and classes we've discussed so far.

SUNetID: _____

Last Name: _____

First Name: _____

Section Leader: _____

I accept both the letter and the spirit of the Honor Code. I have not received any unpermitted assistance on this test, nor will I give any. I do not have any advance knowledge of what questions will be asked on this exam. My answers are my own work. Finally, I understand that the Honor Code requires me to report any violations of the Honor Code that I witness during this exam.

(signed) _____

You have three hours to complete this exam. There are 49 total points.

Question	Points	Grader
(1) Words of Encouragement	/ 1	
(2) Recursive Problem-Solving	/ 12	
(3) Linear Structures	/ 12	
(4) Tree Structures	/ 12	
(5) Graphs and Graph Algorithms	/ 12	
	/ 49	

It's been a pleasure teaching CS106B this quarter. Best of luck on the final exam!

Problem One: Words of Encouragement (1 Point)

Hi Everybody!

You've come quite a long way since the start of the quarter. Over the past ten weeks, we've gone through a whirlwind tour of container classes, recursive problem-solving, recursive optimization, recursive backtracking, sorting algorithms, big-O notation, memory management, dynamic arrays, linked lists, binary search trees, data compression, hashing, graph representations, and graph algorithms. That's a ton of material, so congratulations on making it this far! You've worked hard to get where you are now, and I wanted to thank you for putting in so much time and effort.

The questions on this exam are designed for us to see how much you've learned over the past ten weeks. Ten weeks ago you wouldn't have been able to understand most of these questions, let alone answer them. Treat this exam as an opportunity to show us what you've learned over the course of this quarter. Give it your best shot – ***you can do this!***

Good luck!

-Keith

For a free point, write the phrase "Got it!" on the line below.

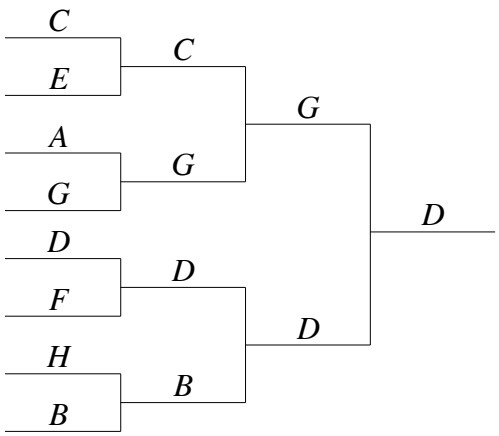
Problem Two: Recursive Problem-Solving (12 Points)

Rigging a Tournament ***(Recommended time: 45 minutes)***

This question has two parts.

This question explores how to use recursion to determine who would win an elimination tournament and, somewhat mischievously, how to set up a tournament so that your favorite player ends up winning.

A ***tournament bracket*** is a type of tournament structure for a group of players. The players are lined up in some initial order (here, ***C, E, A, G, D, F, H, B***, as you can see on the left column). The players are paired off by their positions, with the first player competing against the second, the third player competing against the fourth, etc. The winner of each game advances to the next round, and the loser is eliminated. For example, in the first round, player *C* won her game against player *E*, player *A* lost his game against player *G*, player *D* won her game against player *F*, and player *H* lost his game against player *B*. Those players are again paired off, making sure to preserve their relative ordering. Thus players *C* and *G* and players *D* and *B* face off in the second round, with players *G* and *D* winning and advancing to the next round. Finally, players *G* and *D* face off, and player *D* emerges victorious. Since she's the last player remaining, player *D* is the overall winner of the tournament.



Your task in the first part of this problem is to write a ***recursive*** function

```
string overallWinnerOf(const Vector<string>& initialOrder);
```

that takes as input a vector representing the ordering of the players in the initial tournament bracket, then returns the name of player who ends up winning the overall tournament.

In the course of implementing this function, assume you have access to a helper function

```
string winnerOf(const string& p1, const string& p2);
```

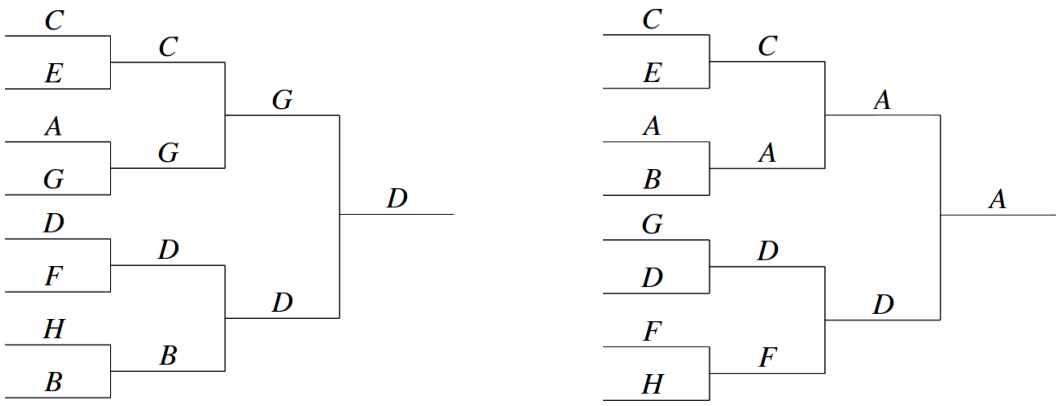
that takes as input the names of two players, then returns which of those two players would win in a direct matchup.

Some notes on this problem:

- You can assume the number of players is a perfect power of two (for example, 1, 2, 4, 8, 16, 32, 64, etc.), so there will never be a case where there's an "odd player out" who isn't assigned to play a game in some round. This also means you'll never get a list of zero players.
- You should not make any assumptions about how a matchup between two players would go based on previous matchups. To determine how a match would go, call the `winnerOf` function.
- This part of the problem ***must*** be implemented recursively – that's what we're testing here. ☺

```
/* You can assume you have access to this function. */
string winnerOf(const string& p1, const string& p2);


string overallWinnerOf(const Vector<string>& initialOrder) {
```

Changing the initial order of players in a tournament can change the outcome of that tournament. For example, imagine that player *A* is a very strong player who would win against every player except player *G*. In the tournament bracket shown to the left, player *A* immediately gets eliminated from the tournament. On the other hand, in the tournament bracket shown to the right, player *A* ends up winning the entire tournament, since player *G* get eliminated before she gets a chance to play a game against player *A*.



Because the winner of a tournament depends on the player ordering, in some cases it is possible to "rig" the outcome of a tournament by changing the initial player ordering. For example, if you were a huge fan of player *D* and wanted her to be the overall winner, and if you knew in advance which opponents player *D* would win against, you could try different orderings and come up with the bracket to the left. If you wanted player *A* to be the overall winner, then you could set up the players in the order to the right. In some cases, there's nothing you can do to ensure someone will win the tournament. For example, a player who you know will lose every game they play will always lose their first game and be eliminated.

Your task is to write a function

```
bool canRigFor(const string& player, const Set<string>& allPlayers,
               Vector<string>& initialOrder);
```

that takes as input the name of a player and a set containing the names of all the players in the tournament, then returns whether there's some initial ordering of the players that will cause that player to be the overall winner. If so, your function should fill in `initialOrder` with one such possible ordering.

Some notes on this problem:

- You should make all the same assumptions about the input as in the first part of this problem: the number of players is always going to be a perfect power of two, that you should use the `winnerOf` function to determine who would win in a matchup, etc.
- Feel free to use the `overallWinnerOf` function from part (i) of this function in the course of solving this problem, even if you weren't able to get a working solution.
- Don't worry about efficiency. We're expecting you to use brute force here, and no creative optimizations are necessary.
- This part of the problem must be done recursively. Again, that's what we're aiming to test here.

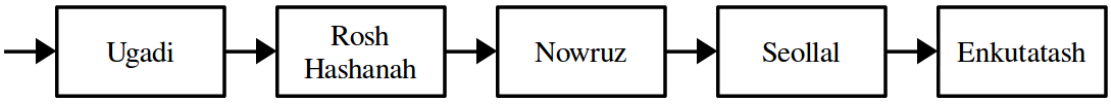
```
/* Assume you have access to these functions and that they work correctly. */
string winnerOf(const string& p1, const string& p2);
string overallWinnerOf(const Vector<string>& initialOrder);


bool canRigFor(const string& player, const Set<string>& allPlayers,
               Vector<string>& initialOrder) {
```

Problem Three: Linear Structures (12 Points)

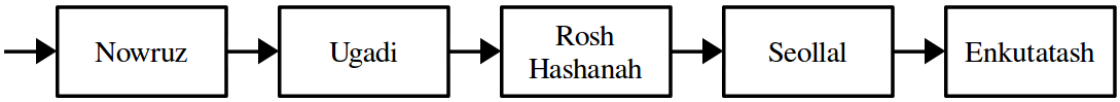
Self-Organizing Lists ***(Recommended time: 45 minutes)***

YouTube and Facebook have tons of data (literally, if you weigh all the disk drives they use to store things), though most of that data is rarely accessed. When you visit YouTube, for example, the videos that will show up will likely be newer videos or extremely popular older videos, rather than random videos from a long time ago. Your Facebook feed is specifically populated with newer entries, though you can still access the older ones if you're willing to scroll long enough.

More generally, data sets are not accessed uniformly, and there's a good chance that if some piece of data is accessed once, it's going to be accessed again soon. We can use this insight to implement the set abstraction in a way that speeds up lookups of recently-accessed elements. Internally, we'll store the elements in our set in an unsorted, singly-linked list. Whenever we insert a new element, we'll put it at the front of the list. Additionally, and critically, whenever we *look up* an element, we will reorder the list by moving that element to the front. For example, imagine our set holds the strings Ugadi, Rosh Hashanah, Nowruz, Seollal, and Enkutatash in the following order:

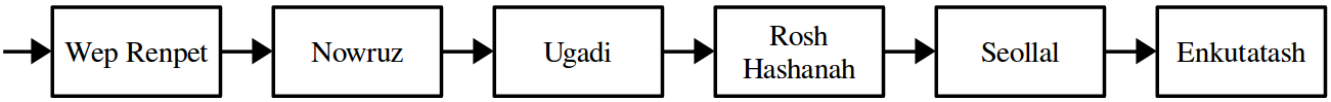


If we look up Nowruz, we'd move the cell containing Nowruz to the front of the list, as shown here:

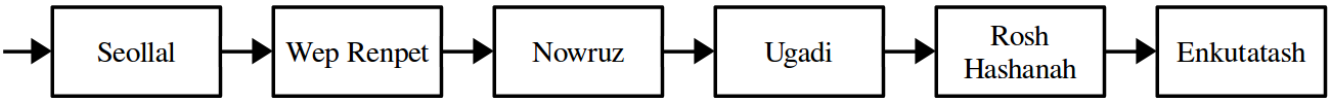


If we now do a look up for Nowruz again, since it's at the front of the list, we'll find it instantly, without having to scan anything else in the list.

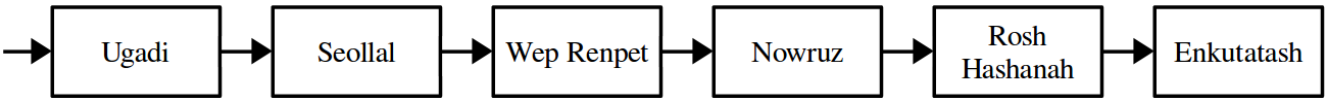
If we now insert the new element Wep Renpet, we'd insert it at the front of the list, as shown here:



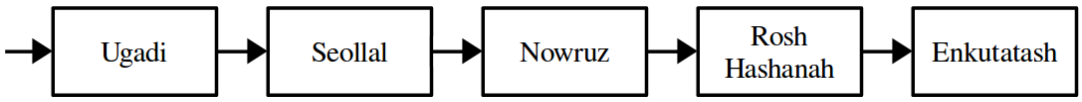
Now, if we do a lookup for Seollal, we'd reorder the list as follows:



If we do an insertion to add Ugadi, since it's already present in the set, we just move it to the front of the list, rather than adding another copy. This is shown here:



Finally, to remove an element from the list, we'd just delete the indicated cell out of the list. For example, deleting Wep Renpet would make the list look like this:



Your task is to implement this idea as a type called `MoveToFrontSet`. The interface is given at the bottom of this page. You're responsible for implementing a constructor and destructor and for implementing the `contains`, `add`, and `remove` member functions.

Some notes on this problem:

- Your implementation ***must*** use a singly-linked list, not a doubly-linked list, but aside from that you can represent the linked list however you'd like.
- When doing a move-to-front, you ***must*** actually rearrange the cells in the list into the appropriate order. Although it might be tempting to simply swap around the strings stored within those cells, this is significantly less efficient than rewiring pointers, especially for long lists.
- You're welcome to add any number of private helper data members, member functions, or member types that you'd like, but you must not modify the public interface provided to you.
- Your implementations of the member functions in this class do not need to be as efficient as humanly possible, but you should avoid operations that are unnecessarily slow or that use an unreasonable amount of auxiliary memory.

As a hint, you may find it useful to have your `add` and `remove` implementations call your `contains` member function and use the fact that it reorganizes the list for you.

```
class MoveToFrontSet {
public:
    MoveToFrontSet();  // Creates an empty set
    ~MoveToFrontSet(); // Cleans up all memory allocated

    bool contains(const string& str); // Returns whether str is present.
    void add(const string& str);      // Adds str if it doesn't already exist.
    void remove(const string& str);   // Removes str if it exists.

private:
    /* Add anything here that you'd like! */

};
```

```
/* Initializes the set so that it's empty. */
MoveToFrontSet::MoveToFrontSet() {









}

/* Cleans up all memory allocated by the set. */
MoveToFrontSet::~MoveToFrontSet() {









}
```

```
/* Returns whether the specified element is in the set. If so, reorders the list so
 * that the element is now at the front. If not, the list order is unchanged.
 */
bool MoveToFrontSet::contains(const string& str) {
```

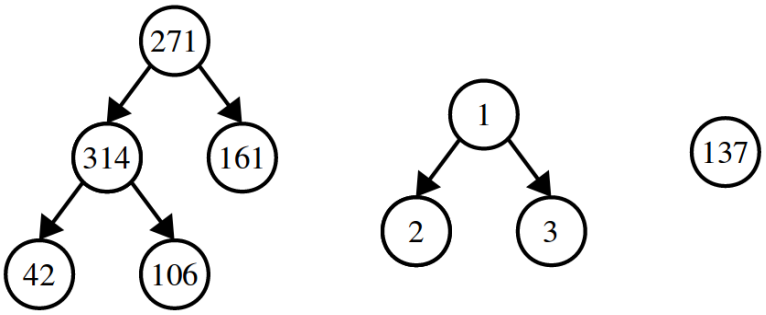
```
/* Adds the specified element to the list, if it doesn't already exist. Either way,
 * the element should end up at the front of the list.
 */
void MoveToFrontSet::add(const string& str) {









}
/* Removes the specified element from the set. If that element doesn't exist, this
 * function should have no effect and should not reorder anything.
 */
void MoveToFrontSet::remove(const string& str) {









}
```

Problem Four: Tree Structures (12 Points)

Agglomerative Clustering ***(Recommended time: 45 minutes)***

This question has two parts.

A ***full binary tree*** is a binary tree where each node either has two children or no children. The Huffman encoding trees you worked with Assignment 6 were full binary trees, as are these:



Note that full binary trees are not necessarily binary *search* trees.

Let's imagine that we have a type representing a node in a full binary tree, which is shown here:

```
struct Node {
    double value;  // The value stored in this node

    Node* left;    // Standard left and right child pointers
    Node* right;
};
```

Your first task in this problem is to write a function

```
Set<double> leavesOf(Node* root);
```

that takes as input a pointer to the root of a full binary tree, then returns a set of all the values stored in the *leaves* of that tree. For example, calling this function on the leftmost tree above would return a set containing {42, 106, 161}, calling this function on the tree in the middle would return {2, 3}, and calling this function on the tree on the right would return {137}.

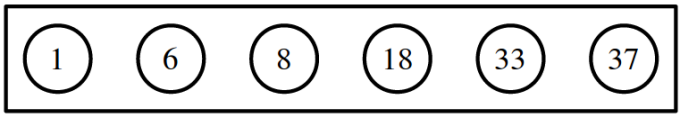
Some notes:

- You can assume that the pointer to the root of the tree is not null.
- You should completely ignore the values stored at the intermediary nodes.

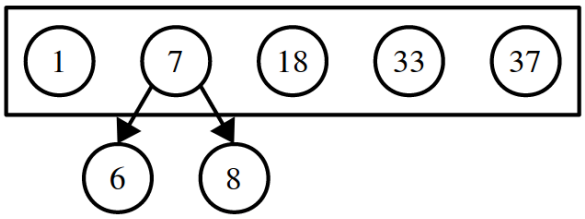
```
struct Node {
    double value;  // The value stored in this node

    Node* left;    // Standard left and right child pointers
    Node* right;
};

Set<double> leavesOf(Node* root) {
```

The second part of this problem explores an algorithm called ***agglomerative clustering*** that, given a collection of data points, groups similar data points together into clusters of similar values. The algorithm is similar to Huffman encoding in that it works by starting with a bunch of singleton nodes and assembling them into full binary trees. For the purposes of this problem, we'll cluster a group of `doubles`.

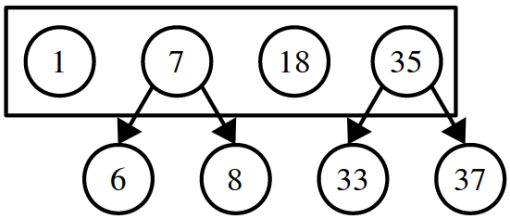
The first step in running agglomerative clustering is to create singleton trees for each of the data points. For example, given the numbers 1, 6, 8, 18, 33, and 37, we'd begin with the following trees:



We now choose the two trees whose root nodes' values are closest to one another. Here, we pick the trees holding 6 and 8. We then merge those two trees into a single tree, and give the root of the new tree the average value of all its leaves. Here, the leaves hold 6 and 8, so the new tree has root value 7:

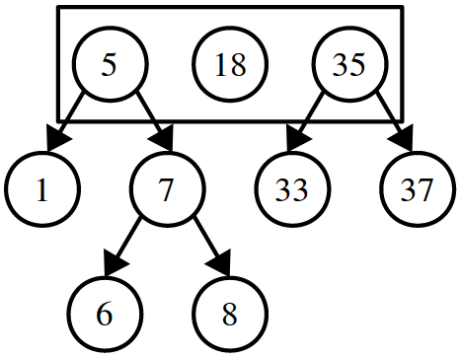


We repeat this process, again selecting the two trees whose root values are as close as possible. In this case, that would be 33 and 37, which get merged into a new tree:



Notice that the new root node has the value 35, the average value of its leaves.

On this next step, we'll find that the trees with the two closest roots are the ones with roots 1 and 7. We merge those trees into a new tree. As before, the root of this tree is then assigned the average value of all its leaves. The leaves have values 1, 6, and 8, so the new root gets the value 5. Here's the result:



As with Huffman encoding, each step in this process reduces the number of trees by one. Unlike Huffman encoding, though, we'll stop this algorithm before it has reduced everything down to a single tree; we'll have a separate parameter that tells us how many trees to hand back.

If we stop here, we have three clusters: the cluster {1, 6, 8}, the cluster {18}, and the cluster {33, 37}, which you can see by looking at the leaves of the resulting trees.

Write a function

```
Set<Node*> cluster(const Set<double>& values, int numClusters);
```

that takes as input a set of values and a desired number of clusters, then runs the agglomerative clustering algorithm described on the previous page to form the specified number of clusters. This function should then return a `Set<Node*>` containing pointers to the roots of the trees formed this way. Some notes:

- You can assume the number of clusters is less than or equal to the number of values and greater than or equal to one (the algorithm only works on values in those ranges).
- Unlike Huffman encoding, ***do not use a priority queue to determine which trees to merge;*** it doesn't work well here. It's fine to iterate over all pairs of trees to see which two are closest.
- If there's a tie between which pair of tree roots is closest, break the tie however you'd like.
- You'll almost certainly want to use the `leavesOf` function you wrote in the first part of this function in the course of writing up your solution.

```
/* You can assume this function works correctly. */
Set<double> leavesOf(Node* root);

/* Feel free to use this helper function to determine the absolute value of the
 * difference between two numbers.
 */
double distance(double a, double b);

Set<Node*> cluster(const Set<double>& values, int numClusters) {
```

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

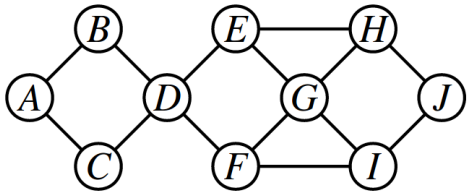
Problem Five: Graphs and Graph Algorithms (12 Points)

Finding All Shortest Paths ***(Recommended time: 45 minutes)***

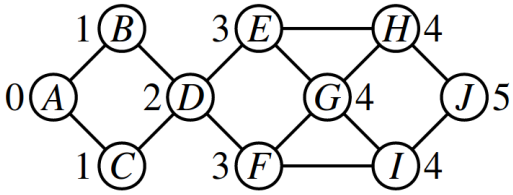
This question has two parts.

If you want to find a minimum-hop path between two nodes in an undirected graph, you can use breadth-first search. We say ***a*** minimum-hop path rather than ***the*** minimum-hop because it's possible for there to be many different paths between two nodes in a graph that are all tied for the shortest length, and running BFS will only find a single one of them. This problem explores one way to find all the shortest paths between two nodes in a graph.

Suppose we want to find all the shortest paths from node *A* to node *J* in this graph:



Our first step is to compute the distances from node *A* to each other node in the graph, which we can do by using a modified breadth-first search. Here's the result of doing that in the above graph:



Notice that distance is measured by the number of *hops*, not the number of *nodes encountered*.

The next step in the algorithm is to do a search in the reverse direction, starting at node *J* and ending with node *A*. The key observation is that ***a path from J to A is a shortest path precisely if the distance to node A decreases with each step of the path***. For example, the path

$$J \to H \to E \to D \to B \to A$$

is a shortest path from *J* to *A*, and we can see this because at each step the distance decreases by one. To find all the shortest paths, we'll use a (recursive) exhaustive search starting at *J* for all paths back to *A* that have this property. This will turn up these paths:

$$J \to H \to E \to D \to B \to A$$
$$J \to H \to E \to D \to C \to A$$
$$J \to I \to F \to D \to B \to A$$
$$J \to I \to F \to D \to C \to A$$

Take a few seconds to confirm that each of these paths does indeed get closer to node *A* at each step.

Each of these paths is a shortest path from node *A* to node *J*, just written in reverse. We can then reverse these paths to get the list of all possible shortest paths from *A* to *J*, as shown here:

$$A \to B \to D \to E \to H \to J$$
$$A \to C \to D \to E \to H \to J$$
$$A \to B \to D \to F \to I \to J$$
$$A \to C \to D \to F \to I \to J$$

To recap: first, we compute (using BFS) the distance from node *A* to each other node in the graph, then we find (recursively) all the shortest paths from node *J* to node *A*, reversing each path we find.

In this problem, we'll use the following types to represent graphs and paths:

```
using Graph = Map<string, Set<string>>;
using Path  = Vector<string>;
```

The first step in the algorithm is to compute the lengths of the shortest paths from the start node to each reachable node in the graph. Below is an implementation of BFS that works by maintaining a queue of paths. Your task is to ***insert new code*** into this function so that it correctly returns a `Map<string, int>` storing the lengths of the shortest paths from the start node to each other (reachable) node in the graph. You ***must not*** delete, reorder, or edit any of the existing code. Some notes:

- If there isn't a path from the start node to some destination node in the graph, then the destination node shouldn't be put into the resulting map.
- The length of a path is measured by the number of *hops taken*, not the number of *nodes visited*.

```
Map<string, int> distancesFrom(const string& start, const Graph& graph) {

    Queue<Path> worklist;

    worklist.enqueue({ start });

    Set<string> visited = { start };

    while (!worklist.isEmpty()) {

        Path   path = worklist.dequeue();

        string last = path[path.size() - 1];

        for (string next: graph[last]) {

            if (!visited.contains(next)) {

                visited.add(next);

                Path nextPath = path;

                nextPath += next;

                worklist.enqueue(nextPath);

            }

        }

    }

}
```

With that helper function written, your next task is to write a function

```
Vector<Path> allShortestPathsBetween(const string& start, const string& end,
                                     const Graph& graph);
```

that takes as input a start and end node and an undirected graph, then returns all the shortest paths from the start node to the end node. Use the following algorithm to solve this problem:

- Compute the distances from the start node to each other node in the graph, which you should do by calling the function you updated last page.
- Using a ***recursive*** exhaustive search, find all paths from the end node to the start node such that the distance to the start node decreases at each step.
- Each of those paths goes the wrong direction (from the end node to the start node), so reverse each path and return the final collection of paths found this way.

Some notes:

- ***You must not implement your exhaustive search using a breadth-first search***. We've already given you code for BFS on the previous page, so it would be kinda silly if we wanted you to code up yet another BFS. Instead, implement this search recursively.
- Your exhaustive search does not have to be as efficient as humanly possible, but it should not have any gross inefficiencies in it. Don't, for example, list off every possible path in the graph and then check at the end which of them are shortest paths.
- You can assume the start and end nodes are present in the graph.
- As an edge case, if the start and end nodes are the same, you should represent a path from the start node to the end node as a path just containing the start node.
- You don't need to consider the case where there are no paths from the start node to the end node.

```
using Graph = Map<string, Set<string>>;
using Path  = Vector<string>;

/* Assume this function works properly. */
Map<string, int> distancesFrom(const string& start, const Graph& graph);


Vector<Path> allShortestPathsBetween(const string& start, const string& end,
                                     const Graph& graph) {
```

(Extra space for your answer to Problem Five, if you need it.)

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);
```

Graph Algorithms Reference Sheet

```
breadth-first-search() {
  make a queue of nodes.
  enqueue the start node.
  color the start node yellow.

  while (the queue is not empty) {
    dequeue a node from the queue.
    color that node green.

    for (each neighboring node) {
      if (that node is gray) {
        color the node yellow.
        enqueue it.
      }
    }
  }
}
```

```
dijkstra's-algorithm() {
  make a priority queue of nodes.
  enqueue the start node at distance 0.
  color the start node yellow.

  while (the queue is not empty) {
    dequeue a node from the queue.
    if (that node isn't green) {
      color that node green.

      for (each neighboring node) {
        if (that node is not green) {
          color the node yellow.
          enqueue it at the new distance.
        }
      }
    }
  }
}
```

```
aStarSearch() {
  make a priority queue of nodes.
  enqueue the start node at distance 0.
  color the start node yellow.

  while (the queue is not empty) {
    dequeue a node from the queue.
    if (that node isn't green) {
      color that node green.

      for (each neighboring node) {
        if (that node is not green) {
          color the node yellow.
          enqueue it at the new distance plus the heuristic.
        }
      }
    }
  }
}
```

```
kruskals-algorithm() {
  remove all edges from the graph.
  put each node into its own cluster.
  for (each edge, in increasing order of cost) {
    if (the edge's endpoints are in different clusters) {
       add that edge back to the graph.
       merge those two clusters.
    }
  }
  return the edges added back.
}
```