CS106B
Spring 2017 May 4th, 2017

CS106B Midterm KEY

This is a closed note, closed-book exam. You are allowed **one back-and-front page of notes, and the reference sheet posted on the course website.** You may not use any laptops, cell phones, or internet devices of any sort, unless you are taking the exam on a laptop, which must only be used for the exam. You will be graded on functionality—but good style helps graders understand what you were attempting. You do not need to #include any libraries and you do not need to forward declare any functions. You have 2 hours. We hope this exam is an exciting journey.

Last Name:	____________________
First Name:	____________________
Sunet ID (eg jdoe):	____________________
Section Leader:	____________________

I accept the letter and spirit of the honor code. I've neither given nor received aid on this exam. I pledge to write more neatly than I ever have in my entire life.

(signed) ____

		Score	Grader
1. Algorithm Analysis and Big-O	[10]	______	______
2. Stacks and/or Queues	[12]	______	______
3. Recursion Tracing	[10]	______	______
4. ADTs	[12]	______	______
5. Traveling Salesman Problem	[15]	______	______
Practice Midterm Bonus	[1]	______	
Total	**[59]**	______	______

Question 1: Algorithm Analysis and Big O (10 Points)

Give a tight bound of the nearest runtime complexity class for each of the following code fragments in Big-Oh notation, in terms of variable *N*. (Write the growth rate as *N* grows.) Write a simple expression that gives only a power of *N*, such as O(N^2) or O(log *N*), *not* an exact calculation like O($2N^3 + 4N + 14$). Write your answer in the blanks on the right side.

Question | **Answer (2 points each)**

```
a)
int sum = 0;
for (int i = 0; i < N; i++) {
    sum++;
}
for (int i = 100*N; i >= 0; i--) {
    sum++;
}
cout << sum << endl;
```

O(______N_______)

```
b)
int sum = 0;
for (int i = 1; i < N - 2; i++) {
    for (int j = 0; j < N * 3; j += 2) {
        for (int k = 0; k < 1000; k++) {
            sum++;
        }
    }
}
cout << sum << endl;
```

O(______N^2_______)

```
c)
Vector<int> v;
for (int i = 0; i < N; i++) {
    v.add(i);
}
while (!v.isEmpty()) {
    v.remove(0);
}
cout << "done!" << endl;
```

O(______N^2_______)

```
d)
Set<int> set;
for (int i = 0; i < N/2; i++) {
    set.add(i);
}
Stack<int> stack;
for (int i = 0; i < N/2; i++) {
    set.remove(i);
    stack.push(i);
}

cout << "done!" << endl;
```

O(______N log N_______)

```
e)
Queue<int> queue;
for (int i = 1; i <= N; i++) {
    queue.enqueue(i * i);
}
HashMap<int, int> map;
while (!queue.isEmpty()) {
    int k = queue.dequeue();
    map.put(k, N * N);
}
cout << "done!" << endl;
```

O(______N_______)

Question 2: Stacks and / or Queues (12 Points)

For this problem, you will be given a sequence consisting of the letters 'I' and 'D' where 'I' denotes an *increasing* sequence and 'D' denotes a *decreasing sequence* of numbers. Here are some examples:

Sequence	➔	Output
`IIDDIDID`	➔	125437698
`IDIDII`	➔	1325467
`DDDD`	➔	54321
`IIII`	➔	12345
`I`	➔	12
`D`	➔	21

Note that a sequence of *n* characters produces a number with *n*+1 digits, because the 'I' or 'D' represent the nature of the sequence from one number to the next.

Write the following function, which takes a string sequence and returns a string that represents the minimum number without repeating any digits:

```
string decode(string seq);
```

Notes:

1. You must use a stack and / or a queue in your solution in a meaningful way.
2. For full credit, your solution should run in worst-case O(n) time (you can still receive most of the points for a non-O(n) solution).
3. You should only use the digits 1-9 (not 0).
4. You may use the Stanford library function `string integerToString(int i)` if you need to.
5. You do not need to #include any Stanford or Standard libraries (e.g., if you use a stack or a queue, we will assume the appropriate libraries are included).
6. The minimum length of the input sequence will be one character, and the maximum length of the input sequence string will be eight characters.

Please put your answer to question 2 here:

```
string decode(string seq) {

    // create a stack to hold the values
    Stack<int> s;

    // create a string for the result
    string result;

    // we need to iterate n+1 times
    for (int i=0; i <= (int)seq.length(); i++) {
       s.push(i+1);
       // if we have processed all characters or the character
       // is an 'I'
       if (i == (int)seq.length() || seq[i] == 'I') {
           // process the entire stack
           while (!s.isEmpty()) {
               // pop and add it to the solution
               result += integerToString(s.pop());
           }
       }
    }
    return result;
}
```

Question 3: Recursion Tracing (10 points)

For each of the calls to the following recursive function below, indicate what output is produced:

```
void recursionMystery(int n) {
    if (n <= 1) {
        cout << "*";
    } else if (n == 2) {
        recursionMystery(n - 1);
        cout << "*";
    } else {
        cout << "(";
        recursionMystery(n - 2);
        cout << ")";
    }
}
```

Call	Output (2 points each)
a) `recursionMystery(2);`	`**`
b) `recursionMystery (3);`	`(*)`
c) `recursionMystery (4);`	`(**)`
d) `recursionMystery (6);`	`((**))`
e) `recursionMystery (9);`	`((((*))))`

Question 4: ADTs (12 points)

Consider the following function:

```
void collectionMystery(const Map<string, string>& m) {
    Set<string> s;
    for (string key : m) {
        if (m[key] != key) {
            s.add(m[key]);
        } else {
            s.remove(m[key]);
        }
    }
    cout << s << endl;
}
```

Note: remember that a map stores keys in order (e.g., "cat" is stored before "dog").

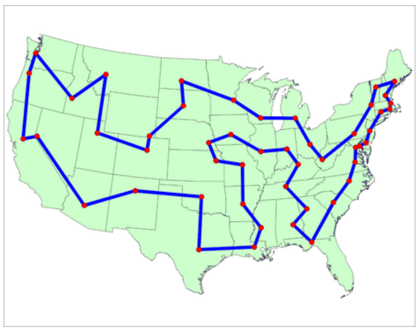
Write the output produced by the function when passed each of the following maps:

Map	Output (3 points each)
a) `{"cast":"plaster", "house":"brick", "sheep":"wool", "wool":"wool"}`	`{"brick", "plaster"}`
b) `{"ball":"blue", "corn":"yellow", "emerald":"green", "grass":"green", "winkie":"yellow"}`	`{"blue", "green", "yellow"}`
c) `{"apple":"peach", "corn":"apple", "peach":"peach", "pie":"fruit", "potato":"peach"}`	`{"apple", "fruit", "peach"}`
d) `{"cat":"cat", "corgi":"dog", "emu":"animal", "lab":"lair", "lair":"lair", "nyan":"cat"}`	`{"animal", "cat", "dog"}`

Note: For part (d), "crash" is acceptable, because we were unclear that you can attempt to remove from an empty set without an error (the set just doesn't do anything).

We did not take off points for having the output out of order, but we did take points off for duplicate items in the set.

Question 5: The Traveling Salesman Problem (15 points)



There is a famous problem in computer science and mathematics called the *Traveling Salesman Problem*. It is stated as follows:

"A salesman must visit *n* cities in a given area. Given the list of cities and the distances between them, what is the shortest possible route that visits each city exactly once and returns to the original city?"

This problem is in the computational category called *NP-Complete*, which means that there is no known way to find an efficient solution. That is the bad news. The good news is that we have a method for finding *all* solutions to the problem (however inefficient), and choosing the correct one: recursive backtracking!

Assume all cities are numbered from 0 to n-1, and you have a `Grid<double> distance` where `distance[r][c]` is the distance between city `r` and city `c`. Here is an example of a 4x4 grid with cities 0-3 (indexes in bold, distances in plain text):

	0	**1**	**2**	**3**
0	0	10	15	20
1	10	0	35	25
2	15	35	0	30
3	20	25	30	0

The solution to the Traveling Salesman Problem starting and ending at city 0 is:

0 -> 1 -> 3 -> 2 -> 0
for a total trip distance of 10 + 25 + 30 + 15 = 80

Write the following function, which returns a `Vector<int> bestRoute` of the **best possible route** between the cities, which has the original city as its first element and its last element, for a complete path:

```
Vector<int> bestRoute(Grid<double> &distance, int startCity);
```

You are allowed to create any helper functions you need, and your solution should recursively check all possible paths between all cities, starting from the first city.

Notes:

1. You can assume you have access to the following function, which calculates the total distance of a `Vector<int> route`:

   ```
   double totalRouteDistance(Grid<double> &distance,
                              Vector<int> &route);
   ```

2. You may use the constant `DBL_MAX` (the largest possible double) to indicate an infinite distance.

Please put your answer to question 5 here:

```
Vector<int> bestRoute(Grid<double> &distance, int startCity) {
    Vector<int> currentRoute;
    // add startCity to best
    currentRoute.add(startCity);
    Set<int> leftToVisit;

    // populate leftToVisit with cities;
    for (int i=0; i < distance.numCols(); i++) { // same as numRows()
        if (i != startCity) { // don't add start city
            leftToVisit.add(i);
        }
    }
    Vector<int> bestOverall;
    bestRouteHelper(distance, currentRoute, leftToVisit, bestOverall);
    return bestOverall;
}

void bestRouteHelper(Grid<double> &distance, Vector<int> &currentRoute,
                Set<int> &leftToVisit, Vector<int> &bestOverall) {
    // base case
    if (leftToVisit.isEmpty()) {
        // add start city to end
        currentRoute.add(currentRoute[0]);

        double currentDist = totalRouteDistance(distance, currentRoute);
        double bestDist = totalRouteDistance(distance, bestOverall);

        if (bestOverall.isEmpty() || currentDist < bestDist) {
            bestOverall = currentRoute;
        }

    } else {

        for (int currentCity : leftToVisit) {
            // go through all remaining cities, removing one at a time

            Set<int> nextLeftToVisit = leftToVisit - currentCity;
            Vector<int> nextRoute = currentRoute;
            nextRoute += currentCity;

            // recursively calculate the best route with the new city
            bestRouteHelper(distance, nextRoute, nextLeftToVisit,
bestOverall);

        }

    }
}
```

Extra space for question 5:

```
(alternate solution)
Vector<int> bestRoute(Grid<double> &distance, int startCity) {
    Vector<int> currentRoute;
    // add startCity to best
    currentRoute.add(startCity);
    Vector<int> leftToVisit;

    // populate leftToVisit with cities;
    for (int i=0; i < distance.numCols(); i++) { // same as numRows()
        if (i != startCity) { // don't add start city
            leftToVisit += i;
        }
    }
    Vector<int> bestOverall;
    bestRoute(distance, currentRoute, leftToVisit, bestOverall);
    return bestOverall;
}

double bestRoute(Grid<double> &distance, Vector<int> &currentRoute,
                 Vector<int> &leftToVisit, Vector<int> &bestOverall) {
    double minDist;
    // base case
    if (leftToVisit.isEmpty()) {
        // return the current route's distance,
        // and consider this the best overall route
        minDist = totalRouteDistance(distance, currentRoute);
        bestOverall = currentRoute;

        // add start city to bestOverall
        int startCity = bestOverall[0];
        minDist += distance[bestOverall[bestOverall.size()-1]][startCity];
        bestOverall.add(startCity);
    } else {
        minDist = DBL_MAX;
        Vector<int> localBestRoute = bestOverall; // keep local best

        for (int i = 0; i < leftToVisit.size(); i++) {
            // go through all remaining cities, removing one at a time
            int currentCity = leftToVisit[i];
            // add it to the current route
            currentRoute.add(currentCity);

            // remove it from the cities we still have to visit
            leftToVisit.remove(i);

            // recursively calculate the best route with the new city
            double newDist = bestRoute(distance, currentRoute,
                                       leftToVisit, bestOverall);

            // update minDist and the best route
            if (newDist < minDist) {
                minDist = newDist;
                localBestRoute = bestOverall;
            }
            // backtrack:
            // replace the city in the cities we have to visit
            // and remove it from the current route
            leftToVisit.insert(i,currentCity);
            currentRoute.remove(currentRoute.size()-1);
        }
        // once we finish the loop, we have a best route
        bestOverall = localBestRoute;
    }
    return minDist;
}
```

------------total route distance func (did not have to write):

```
double totalRouteDistance(Grid<double> &distance, Vector<int> &route) {
    int routeSize = route.size();
    if (routeSize == 0) {
        return DBL_MAX;
    }
    double totalDistance = 0;
    for (int i=0; i < routeSize-1; i++) {
        int city1 = route[i];
        int city2 = route[i+1];
        totalDistance += distance[city1][city2];
    }
    return totalDistance;
}
```