CS106B Final Practice 2

You have 3 hours to complete this exam

This is a closed book, closed computer exam. You are allowed only 1 page (8.5x11, both sides) of notes, and no other aids (and you are not allowed magnifying devices other than regular glasses). You don't need to **`#include`** any libraries, and you needn't use assert to guard against any errors. Understand that the majority of points are awarded for concepts taught in CS106B, and not prior classes. You don't get many points for for-loop syntax, but you certainly get points for proper use of Stanford library containers, efficiency of your code, and solution of the problem.

You will be given a reference sheet. If you need extra space for scratch and/or problem responses, use extra scratch paper. We will not accept any work done on scratch paper.

NAME: ____

SUNet: ____
(e.g., cgregg)

I accept the letter and spirit of the honor code. I will neither give nor receive unauthorized aid on this exam.

signed ____

Problem 1: 15pts

Problem 2: 25pts

Problem 3: 15pts

Problem 4: 15pts

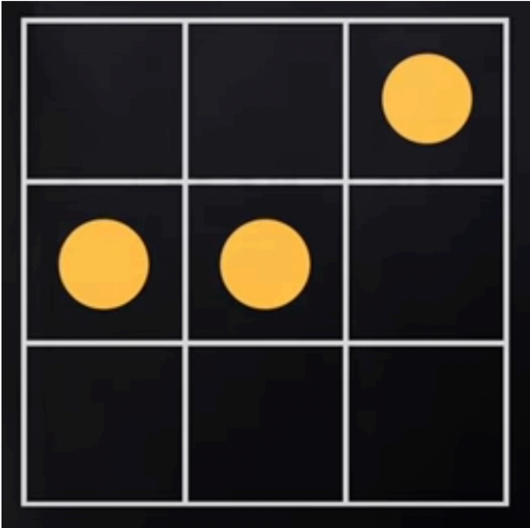
Problem 5: 15pts

Problem 6: 15pts

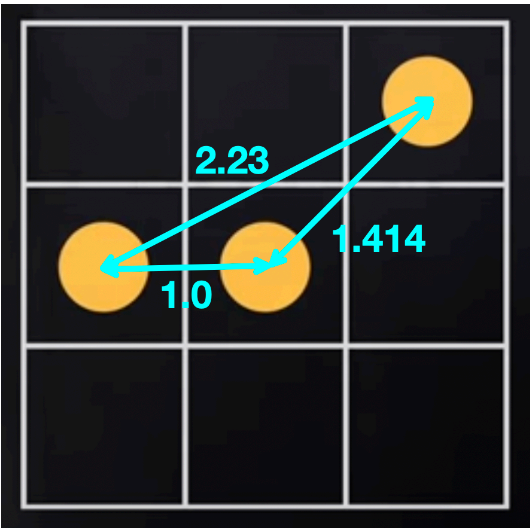
Total: 100 points

Problem 1 – Unique Grid Distance Puzzle, 15 points

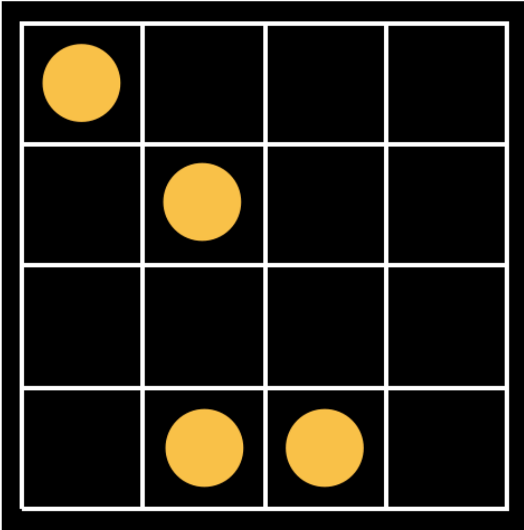
Consider a puzzle that places n circles on an n x n grid, with the intent to have distinct distances between the center of all of the circles. For example, here is a 3 x 3 grid with three circles:



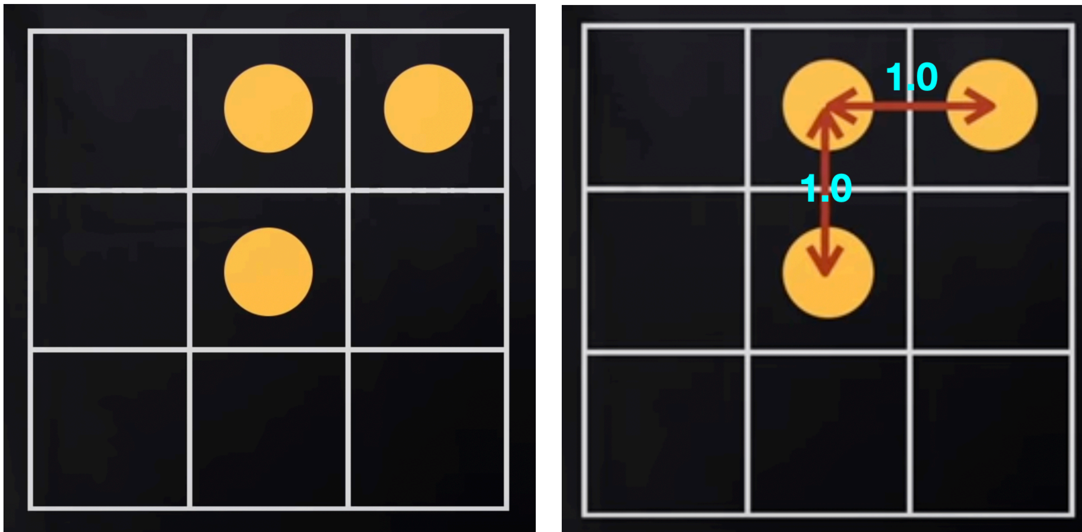
The above circle placement is a solution to the puzzle becuase the circle centers are placed such that no two circles are the same distance apart:



Here is another solution for four circles on a 4 x 4 grid:



However, the following grid is not a solution, because the circle located at row=0, col=1 is the same distance to both of the other circles, as shown on the right:



For this problem, you are to write two functions. The first function, **isUniqueSolution** determines if a vector of **Grid-Locations** consists of circles that are uniquely distanced from each other. You can assume you have a function called **euclidianDistance** that returns the distance between two **GridLocations** (as a double):

```
bool isUniqueSolution(Vector<GridLocation> circles);
```

The second function is a recursive helper function, **findUniqueDistancesHelper** which populates a

Set<Vector<GridLocation>> with all possible solutions for an **n** x **n** grid with **n** circles. A solution is a vector of **n** **GridLocations** that will return **true** when passed into **isUniqueSolution()**.

```
void findUniqueDistancesHelper(const int n,
                 Vector<GridLocation> &circles,
                 Set<Vector<GridLocation>> &solutions,
                 int row, int col);
```

The helper function will be called as follows:

```
Set<Vector<GridLocation>> findUniqueDistances(int n) {
    // n is the number of circles we need to place
    // on a grid of n x n locations
    Vector<GridLocation> circles;
    Set<Vector<GridLocation>> solutions;
    findUniqueDistancesHelper(n,
                  circles,
                  solutions,
                  0, 0);
    return solutions;
}
```

Notes:

- You do not need to use any **`Grids`** for this solution.
- Recall that a **`GridLocation`** has a **`row`** and **`col`** field, and you can create a **`GridLocation`** as follows:

```
GridLocation loc = {1, 2}; // a location at row 1, col 2.
```

- All of the solutions in the set should have **`n`** **`GridLocations`**.
- Circles cannot go on top of each other. In other words, solutions will always have **`n`** distinct circles on the grid in different locations.

Write your solution below:

```
bool isUniqueSolution(Vector<GridLocation> circles) {









}


void findUniqueDistancesHelper(const int n,
                 Vector<GridLocation> &circles,
                 Set<Vector<GridLocation>> &solutions,
                 int row, int col) {
```

Problem 2 – Classes and Dynamic Memory, 15 points

Your task is to write a **Section** class. This class will store information on a single discussion section in CS106B. You can assume you have access to the following **Student struct**:

```
struct Student {
  string name;
  int suid;

  /*
   * This Vector contains the section preferences
   * of a student as string. You can assume that
   * there are >= 0 elements in this Vector.
   * The Vector is formatted such that index 0
   * holds the student's top preference, index
   * 1 holds the students 2nd preference, etc.
   */
  Vector<string> preferences;
};
```

Here's the interface for Section, which stores all the students enrolled in that section as a dynamically-allocated array of **Student**s.

```
// Section.h file

class Section {

public:

  /*
   * Constructor
   * ------------
   * The constructor takes in a maximum capacity, and a
   * meeting time.
   * You can assume the capacity of the section will
   * never change.
   */
  Section(int capacity, string time);

  /*
   * Destructor
   * ----------
   * Cleans up any memory associated with the object.
   */
  ~Section();

  /*
   * Method: tryAddStudent
   * ---------------------
   * Takes in a student and returns true/false based on
   * whether or not the student can be enrolled in this
   * section. A student can be enrolled in the section if:
   *
```

```
   *  - The section time is in the student's
   *    top 3 preferences
   *  - The section is not full
   *
   * Notes:
   * - Students should be added to the internal array in
   * ascending order of preference.
   *
   * - You can assume one student won't try to enroll twice
   * in the same section.
   *
   * - This method must run in a worst-case Big-O of O(n)
   * where n is the number of elements in the internal array
   *
   * - You can assume that all section time strings given to
   * the class methods are formatted the same way (i.e. the
   * time given to the constructor is formatted the same way
   * as the times in the student's preferences)
   */
  bool tryAddStudent(Student s);

  /*
   * Method: getStudent
   * ------------------
   * Takes in an index and returns the student at that
   * index. Throws an error if the index is invalid.
   */
  Student getStudent(int index);

  /*
   * Provided Method: swap
   * ---------------------
   * Takes in two indices and swaps the Students at those
   * two indices in the internal array. You can assume
   * this function is implemented and fully function.
   *
   * YOU DO NOT NEED TO IMPLEMENT THIS FUNCTION
   */
  void swap(int index1, int index2);

private:
  /* TODO */
};
```

1. Write the **`private`** section of the header file
2. Write the full class implementation (i.e. what would go in the **`Section.cpp`** file)

   Note: you can assume that all section time strings given to the class methods are formatted the same way.

Write your solution below:

```
// Section.h

class Section {

public:
  // ... see above

private:









};
// Section.cpp

#include "Section.h"

Section::Section(int capacity, string time) {





}

Section::~Section() {




}
```

```
bool Section::tryAddStudent(Student s) {









}

Student Section::getStudent(int index) {


}
```

Problem 3 – Linked Lists, 15 points

Your task is to write a function called **listDivisibleBy** that takes in a linked list and returns a linked list comprised only of values in the input that are divisible by a certain number.

Say we have a linked list as follows:

4 -> 7 -> 20 -> 8 -> 5 -> 0

called **front**. Calling **listDivisibleBy(front, 4)** would return the following list:

4 -> 20 -> 8 -> 0

and **front** would be edited to be:

7 -> 5

Here's the full function signature you'll be writing:

```
ListNode* listDivisibleBy(ListNode*& front, int divisor);
```

Some notes on this problem:

- Your solution should not allocate any new **ListNode**s. As such, you shouldn't use **new** anywhere in your solution.
- The nodes in the returned list should be in the same order they were in in the original list, and the nodes remaining in the original list should also be in the same order they were originally in.
- Be aware that if the first value in the original list is divisible by the number, **front** will be a different pointer when the function closes (and also note that **front** has been passed into the function by reference).
- You can assume the **divisor** parameter won't be 0
- 0 is divisible by all numbers. The only exception is that 0 is not divisible by 0, but this doesn't matter here since **divisor** will never be 0
- If all numbers in the input list are divisible by the **divisor**, **front** should be **nullptr** at the end of the function
- Here is the **ListNode** struct, for reference:

```
   struct ListNode {
     int data;           /* Data stored in the node. */
     ListNode* next;     /* Pointer to next node in the list. */

     // default constructor, does not initialize
     ListNode() {}

     // two-arg constructor
     ListNode(int d, ListNode* n) {
         data = d;
         next = n;
     }

 };
```

Write your solution below

```
ListNode* listDivisibleBy(ListNode*& front, int divisor) {
```

Problem 4 – Trees, 20 Points

There are 3 parts to this problem. All three parts are based on the following **TreeNode** definition:

```
struct TreeNode {
    int data;
    TreeNode *left;
    TreeNode *right;

    // default constructor does not initialize
    TreeNode() {}

    // 3-arg constructor sets fields from arguments
    TreeNode(int d, TreeNode* l, TreeNode* r) {
        data = d;
        left = l;
        right = r;
    }
};
```

A tree is defined by its root. There are no pre-defined tree functions (e.g., you do not have **findMin()** or **add()**, etc.).

Part A: Difference of Max and Min in a BST

For this problem, write a function that returns the difference of the maximum and minimum values in a Binary Search Tree (BST). Your function must be O(log n) for full credit:

```
int diffMaxMinBst(TreeNode *root);
```

Notes:

- This function should not be recursive, and it cannot have any helper functions.
- As an example, if the maximum value in the tree is 42, and the minimum value is 11, then the function should return 31.
- Negative values in the tree are allowed.
- Remember that Big O disregards constants, so O(2 log n) is equivalent to O(log n).

Write your function below:

```
int diffMaxMinBst(TreeNode *root);
```

Part B: Difference of Max and Min in a non-BST

For this problem, write a function that returns the difference of the maximum and minimum values in a non-Binary Search Tree. You cannot assume any orderings of the nodes in the tree. Your function must be O(n) for full credit:

```
int diffMaxMinNonBst(TreeNode *root);
```

Notes:

- You are welcome to write one or more helper functions for this problem.

Write your function below (you may put your helper below that):

```
int diffMaxMinNonBst(TreeNode *root) {
```

Part C: Create a complete, balanced BST from a sorted vector

For this problem, write a function that creates a complete, balanced Binary Search Tree (BST) from a sorted vector. Recall that a complete tree is a tree that has all of its levels filled from left-to-right except for (possibly) the last level.

```
TreeNode *balancedBstFromSortedVec(Vector sortedVec);
```

Notes:

- The solution does not require any helper functions, though you are welcome to write one or more helper functions for this problem if you wish.

Write your function below:

```
TreeNode *balancedBstFromSortedVec(Vector sortedVec) {
```

Problem 5 – Hashing, 9 Points

There are three questions to answer for this problem.

Question A: Hashing – multiple choice, choose all answers that are correct

A good hash function suitable for a hash table should:

A. Be deterministic
B. Be reversible
C. Create a good distribution
D. Be fast, e.g., O(1)
E. Be fast, e.g., O(log n)

Question B: Hashing – multiple choice, choose the best answer

What does it mean for a hash table to use chaining?

A. Each hash bucket holds a linked list for elements that collide when hashed.
B. The next hash is based on the previous valued added to the hash table, so collisions are avoided.
C. All values in the table are linked together with hash nodes, avoiding collisions.
D. The hash function for each bucket depends on the function for the previous bucket, limiting collisions.
E. The hash values form a increasing or decreasing sequence based on the hash table direction, meaning that collisions can be differentiated from each other.

Question C: Hashing – multiple choice, choose all answers that are correct

Chaining is not the only way to build a hash table. Another type of hash table collision handling is called open addressing where an algorithm is used to find an open position in the hash table if the original location is already filled. The simplest open addressing algorithm is called "linear probing," whereby once a hash value is found and there is a collision, the algorithm walks along the buckets one-by-one, looking for an empty bucket. If the end of the array is reached, the search continues at the beginning of the array. Like chaining, both the key and value are stored in the hash table.

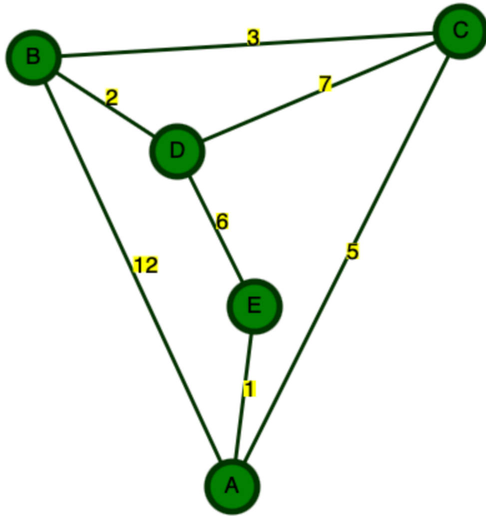
Which of the following would be true of an open addressing hash table with linear probing?

A. Unlike a hash table that uses chaining, an open addressing table can fill up, and must be expanded when the table becomes full.
B. A disadvantage of the linear probing is that hashed values can end up in "clusters," near each other in the table. I.e., many values will end up next to each other in the table.
C. Open addressing guarantees O(1) time, unlike chaining, which can have O(n) behavior.
D. Open addressing would keep the keys in sorted order, unlike chaining.

Problem 6 – BFS, Dijkstra's Algorithm, and A* Algorithm, 9 Points

Note: The reference sheet has pseudocode for all three algorithms mentioned in this problem.

There are three questions to answer for this problem. They are all are based on the following graph:



Question A: Breadth First Search – multiple choice, choose the best answer.

If a breadth first search (BFS) was run to determine the shortest path from node A to node B in the graph, what would the algorithm return?

A. A->B
B. A->E->D->B
C. A->C->B
D. A->C->D->B

Question B: Dijkstra's Algorithm – multiple choice, choose the best answer.

If Dijkstra's Algorothm was run to determine the shortest path from node A to node B in the graph, what would the algorithm return?

A. A->B
B. A->E->D->B
C. A->C->B
D. A->C->D->B

Question C: A* Algorithm – multiple choice, choose the best answer.

If an A* Algorithm was run to determine the shortest path from node A to node B in the graph, what would the algorithm return?

A. A->B
B. A->E->D->B
C. A->C->B
D. A->C->D->B