This practice exam is based on the actual midterm exam from Cynthia's Spring 2014 class. It did not include a classes problem (which you should expect this quarter), and the memory/pointers problem covered material that we covered differently this quarter and needed to be replaced. So I added a classes problem and a linked nodes problem from one of Marty Stepp's practice exams. Because the exam now has one more problem than it did before, this exam is somewhat longer than what you should expect for the actual midterm exam. --CBL

CS106B Instructor: Cynthia Lee
Spring 2016 Practice Exam

PRACTICE MIDTERM EXAM #2

NAME (LAST, FIRST): _____

SUNET ID: _____@stanford.edu___

Problem	1	2	3	4	4	TOTAL
Topic	ADTs	Linked Nodes	Recursion	Classes	Short Answer	
Score						
Possible	20	20	20	20	20	100

Instructions:

- The time for this exam is **2 hours**, or 120 minutes. There are 100 points possible, so about 1-1.5 minutes per point. This should give you a general sense of pacing, though each person will have different problems that challenge them more or less than others.
- Use of anything other than a pencil, eraser, pen, one 8.5x11 page (two sides) of notes, and the official textbook is prohibited. In particular, **no** computers or digital devices of any kind are permitted. Blank scratch paper may be provided by proctors and does not need to be turned in.
- PLEASE rip off the pages of library reference in the back of the exam and do not turn them in.
- Please do NOT staple or otherwise insert new pages into the exam. Changing the number of pages in the exam confuses our automatic scanning system. Thanks.
- SCPD and OAE: Please call or text Cynthia @ 760-845-7489 if you have a question.

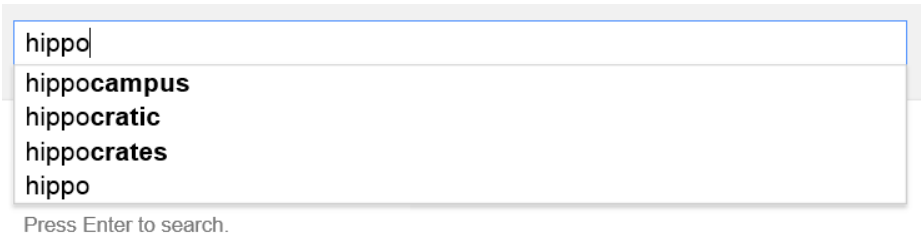
Please sign *before* you begin:

I agree to abide by the spirit and letter of the Honor Code, and to follow the instructions above.

_____ _____

(Signature) (Date)

1. **ADTs (20pts).** For this problem, you will write an autocomplete feature for typing on a phone or in the Google search box. You will suggest words that complete what the user has typed so far, like in this example:



You should write two functions, with the following signatures:

```
void calculatePrefixes(Lexicon& english, Map<string,Set<string>>& prefixes);
```

- Given a Lexicon, you will populate a Map (you may assume it is initially empty) with all prefixes found in all the words in the Lexicon.
- Each prefix maps to a Set of strings that are all the words starting with that prefix. (This is a very space-inefficient way to store this information—we'll learn something better on that count later in the quarter—but it will be fast.)
- Usually we say that it's not a good idea to loop over a Lexicon, but in this problem, because we are processing all the words, you will want to loop over the Lexicon.
- We will not consider the empty string to be a valid prefix.
- Example: Assume (for this example only) that the Lexicon only contains these 3 words: "bat" "cat" "bag". Then when the function returns, the `prefixes` map should contain the following mappings: "b"→{"bat","bag"}, "c"→{"cat"}, "ba"→{"bat","bag"}, "ca"→{"cat"}, "bat" →{"bat"}, "bag"→{"bag"}, "cat"->{"cat"}.

```
void printSuggestions(Map<string,Set<string>>& prefixes, string typing);
```

- Given the map (as created in the `calculatePrefixes` function), and a string `typing` that is what the user has typed so far, print to cout a list of words starting with the prefix `typing` (if there are none, do not print anything to cout). Each word should print on its own line.
- Example: Using the same example as above, if `typing` is "ba", then the output of your program should be as follows:

```
bag
bat
```

- You may print the words in any order.

Note: one page of space is provided for you to write each function, but this is likely much more space than you will actually need.

```
void calculatePrefixes(Lexicon& english, Map<string,Set<string>>& prefixes){

}
```

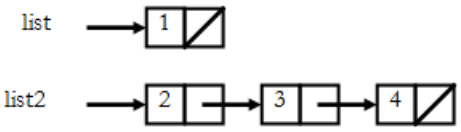
```
void printSuggestions(Map<string,Set<string>>& prefixes, string typing){

}
```

2. **Linked Nodes (20pts) [by Marty Stepp].** Examine the following code. Write the code that will turn the "before" picture into the "after" picture by modifying links between the nodes shown and/or creating new nodes as needed. There may be more than one way to write the code, but you are NOT allowed to change any existing node's data field value. You also should not create new ListNode objects unless necessary to add new values to the chain, but you may create a single ListNode* pointer variable to point to any existing node if you like.

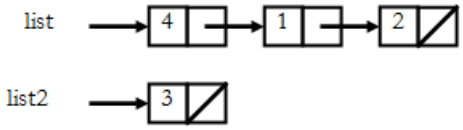
   If a pointer variable does not appear in the "After" picture, it doesn't matter what value it has after the changes are made. If a *node* does not appear in the "After" picture, you must free its memory to avoid a memory leak.

   To help maximize partial credit in case you make mistakes, we suggest that you include optional comments with your code that describe the links you are trying to change, as shown in the solution code for the linked list section handout.

   Before



   After



   Assume that you are using the ListNode structure as defined in lecture and section:

```
struct ListNode {
   int data; // data stored in this node
   ListNode* next; // a link to the next node in the list
   ListNode(int data, ListNode* next) { ... } // constructor
};
```

3. **Recursion (20pts).** After the midterm is over, you plan to drive to Reno with some friends to relax for the weekend. You haven't ever been to a casino, and you'd like to make some notes for yourself to help you get started playing the card game Blackjack. On the most basic level, the objective in Blackjack is to gather cards that sum to 21 points. Each card with a number contributes that many points to the sum, and the Ace can be either 1 point or 11 points (your choice). We will ignore face cards (the Jack, Queen, and King). For this problem, we will represent a card as an int, and the values will be between 1 and 10 as follows:

Card	Represented by int value	Points
Ace	1	1 or 11
2-10	2-10	2 = 2 points, 3 = 3 points, etc.

Write a function with the following signature:

```
void sum21(Vector<int>& availableCards, Vector<Vector<int>>& waysToSum)
```

It will find all the ways of making a sum of 21 with the cards provided in availableCards.

- `availableCards` is a list of the cards available to you. You may only use each card once, but the same number may appear in the vector more than once to reflect more than one card of that number is available
    - Example: if the vector contains 2, 2, 2, 2, 3, 10, then you could return the whole list as one way of summing to 21, but you could not return a solution of 2, 2, 2, 2, 2, 2, 2, 2, 2, 3. The latter sums to 21, but uses more than the available 2 cards.
    - If you wish, you may destructively modify availableCards. In other words, it doesn't need to be in the same condition you received it in when you return.
- `waysToSum` is a vector, initially empty, that you should populate as the output of your function. Your output should include every combination of the available cards that sums to 21.
    - **It is acceptable for there to be repetition in your waysToSum result. In other words, your result may contain two vectors that are exactly the same or some permutation (order rearrangement) of each other, but you do <u>not</u> need to go out of your way to generate every permutation.**

- Your solution must use recursion. The provided function signature may be used as a wrapper that calls a separate helper recursive function.
- You may not use any global variables.

Example:

- Input:
    - `availableCards = {2,1,5,5,2,3,3}`
- Output:
    - `waysToSum = {{5,5,1}, {1,2,2,5,5,3,3}, {1,2,2,3,3}, {5,5,1}}`
- Notes:
    - Two of the combinations use the Ace as 11 points, and one combination uses the Ace as 1 point.
    - There is a repetition of one of the combinations, which is neither banned nor required.

```
void sum21(Vector<int>& availableCards, Vector<Vector<int>>& waysToSum){

}
```

4. Classes (20pts) [by Marty Stepp].

Write a member function `maxCount` to be added to the `ArrayList` class from lecture. Your function should examine the elements of the list and return the number of occurrences of the most frequently occurring value in the list of integers. For this problem, you should assume that the elements in the list are in sorted order. Because the list will be sorted, all duplicates will be grouped together, which will make it easier to count duplicates. For example, suppose that an `ArrayList` called list stores the following sequence of values:

{1, 3, 4, 7, 7, 7, 7, 9, 9, 11, 13, 14, 14, 14, 16, 16, 18, 19, 19, 19}

This list has some values that occur just once or twice, some values that occur three times (14, 19) and a single value that occurs four times (7). Therefore, the client's call of `list.maxCount()` should return 4 to indicate that the most frequently occurring value occurs 4 times. It is possible that there will be a tie for the most frequently occurring value, but that doesn't affect the outcome because you are just returning the count, not the value. If there are no duplicates in the list, then every value will occur exactly once and the maximum would be 1. If the list is empty, you should return 0.

You may call other member functions of `ArrayList` if you like, but your function should not modify the state of the list. Its header should be declared in such a way as to indicate to the client that this function does not modify the list.

In the space below, write the member function's body as it would appear in `ArrayList.cpp`. You do not need to write the function's header as it would appear in `ArrayList.h`. Write only your member function, not the rest of the class. Remember that you are adding this as a member function to the ArrayList class from lecture:

```
class ArrayList {
public:
    void add(int value);
    void clear();
    int get(int index) const;
    void insert(int index, int value);
    bool isEmpty() const;
    void remove(int index);
    void set(int index, int value);
    int size() const;
    string toString() const;
}
```

```
private:
    int* elements;
    int mysize;
    int capacity;
    void checkCapacity();
    void checkIndex(int i, int min, int max);
```

5. **Short Answer (20pts).**

a. Which of the following data structures allow you to retrieve any element in the data structure at any time? (circle ALL that apply)

Vector Stack Queue Grid

b. T/F: You cannot have a pointer that points to memory on the stack. (circle ONE)

TRUE FALSE

c. T/F: One of the base cases in our recursive Binary Search algorithm from class (recall it looks for a number in a sorted Vector of numbers) was that there are only three numbers left to search, so we check each one to see if it is the number we're looking for. (circle ONE)

TRUE FALSE

d. What is the best Big-O characterization of the performance of this code, in terms of N, where N is the size of the vector? (Write your answer on the line.)

```
for (int i=3; i<data.size()/5; i++){
      cout << data[i] << endl;
}
```

O(_____)

e. What is the best Big-O characterization of the performance of this code, in terms of N, where N is the size of the vector? (Write your answer on the line.)

```
for (int i=0; i<data.size(); i+=(data.size()/5)){
      cout << data[i] << endl;
}
```

O(_____)

f. What is the best Big-O characterization of the performance of this code, in terms of N, where N is the size of the vector data? (Write your answer on the line.)

```
for (int i=data.size()-1; i>=0; i-=3){
    for (int j=0; j<data.size(); j+=3){
        cout << data[i] << data[j] << endl;
    }
}
```

O(_____)

g. (6pts) What are two different uses for the & symbol in C++ code? Write one short sentence each, naming and explaining the use. (Please only list two.)

h. What are two different reasons you might use pass by reference in when writing a function in C++? Write 1-2 short sentences each, explaining the use. (Please only list two.)

i. (0pts - Optional) What is the most interesting thing you've learned in class so far?

Summary of Relevant Data Types

We tried to include the most relevant member functions and operators for the exam, but not all are listed. You are free to use ones not listed here that you know exist. ***You do <u>not</u> need to do #include for these.***

```
class string {
 bool empty() const;
 int size() const;
 int find(char ch) const;
 int find(char ch, int start) const;
 string substr(int start) const;
 string substr(int start, int length) const;
 char& operator[](int index);
 const char& operator[](int index) const;
};

class Vector {
 bool isEmpty() const;
 int size() const;
 void add(const Type& elem); // operator+= used similarly
 void insert(int pos, const Type& elem);
 void remove(int pos);
 Type& operator[](int pos);
};

class Grid {
 int numRows() const;
 int numCols() const;
 bool inBounds(int row, int col) const;
 Type get(int row, int col) const; // cascade of operator[] also works
 void set(int row, int col, const Type& elem);
};

class Stack {
 bool isEmpty() const;
 void push(const Type& elem);
 Type pop();
};

class Queue {
 bool isEmpty() const;
 void enqueue(const Type& elem);
 Type dequeue();
};
```

```
class Map {
 bool isEmpty() const;
 int size() const;
 void put(const Key& key, const Value& value);
 bool containsKey(const Key& key) const;
 Value get(const Key& key) const;
 Value& operator[](const Key& key);
};
```
Example for loop: `for (Key key : mymap){…}`

```
class Set {
 bool isEmpty() const;
 int size() const;
 void add(const Type& elem); // operator+= also adds elements
 bool contains(const Type& elem) const;
};
```
Example for loop: `for (Type elem : mymap){…}`

```
class Lexicon {
  int size() const;
  bool isEmpty() const;
  void clear();
  void add(std::string word);
  bool contains(std::string word) const;
  bool containsPrefix(std::string prefix) const;
};
```
Example for loop: `for (string str : english){…}`