CS 106X, Autumn 2018
Practice Midterm Exam (Based on Autumn 2017 CS 106X midterm exam)

Your Name: _____

Section Leader: _____

Honor Code: *I hereby agree to follow both the letter and the spirit of the Stanford Honor Code. I have not received any assistance on this exam, nor will I give any. The answers I am submitting are my own work. I agree not to talk about the exam contents to anyone until a solution key is posted by the instructor.*

Signature: _____ ← **YOU MUST SIGN HERE!**

Rules: *(same as posted previously to class web site)*

- This exam is to be completed by each student **individually**, with no assistance from other students.
- You have **2 hours** (120 minutes) to complete this exam.
- This test is **open-book**, but **closed notes. You may not use any paper resources.**
- You may not use any computing devices, including calculators, cell phones, iPads, or music players.
- Unless otherwise indicated, your code will be graded on proper behavior/output, not on style.
- On code-writing problems, you do not need to write a complete program, nor `#include` statements. Write only the code (function, etc.) specified in the problem statement.
- Unless otherwise specified, you can write **helper functions** to implement the required behavior. When asked to write a function, do not declare any **global variables**.
- Do not abbreviate code, such as writing ditto marks ("") or dot-dot-dot marks (...).
- If you wrote your answer on a back page or attached paper, please **label this** clearly to the grader.
- Follow the Stanford **Honor Code** on this exam and correct/report anyone who does not do so.

Good luck! You can do it!

Many thanks to colleagues for problem ideas:
Jerry Cain, Cynthia Lee, Nick Parlante, Chris Piech, Stuart Reges, Eric Roberts, Mehran Sahami, Keith Schwarz, and more.

#	Description	Earned	Max	Grader (initial)
1	Code Reading		10	
2	Code Reading		10	
3	Code Writing		10	
4	Code Writing		10	
5	Code Writing		10	
6	Code Writing		6	
7	Code Writing and Short Answer		4	
	TOTAL		**60**	

1. Code Reading

For both of the calls to the following recursive function below, indicate the **state of the string s** that was passed to the function, as well as what value is **returned**. Recall that when one `char` value is compared to another, they are compared by ASCII value, which amounts to alphabetical ordering.

The following listing of the alphabet may help you: `ABCDEFGHIJKLMNOPQRSTUVWXYZ`

```
int mysteryX(string& s, int i, int j, char k) {
    if (i >= j) {
        return i;
    } else if (s[i] < k) {
        return mysteryX(s, i + 1, j, k);
    } else if (s[j] > k) {
        return mysteryX(s, i, j - 1, k);
    } else {
        int temp = s[i];
        s[i] = s[j];
        s[j] = temp;
        return mysteryX(s, i + 1, j - 1, k);
    }
}
```

a) call:	`//          0123456`
	`string s = "OXIDIZE";`
	`mysteryX(s, 0, 6, 'K')`

state of string s after call:	
returns:	

b) call:	`//          01234567890123456`
	`string s = "TCTTCGTCCGAACCAGA";`
	`mysteryX(s, 0, 16, 'F')`

state of string s after call:	
returns:	

2. Code Reading

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*. (In other words, the algorithm's runtime growth rate as *N* grows.)Write a simple expression that gives only a power of *N*, not an exact calculation like $O(2N^3 + 4N + 14)$. Write your answer in the blanks on the right side.

Question	Answer

a)
`Vector<int> v;`
`for (int i = 1; i <= N; i++) {`
`  v.insert(0, 2 * i);`
`}`
`HashSet<int> s;`
`for (int k : v) {`
`  s.add(k);`
`}`
`cout << "done!" << endl;`
	$O($_____$)$

b)
`int sum = 0;`
`for (int i = 1; i <= 100000; i++) {`
`  for (int j = 1; j <= i; j++) {`
`  for (int k = 1; k <= N; k++) {`
`  sum++;`
`  }`
`  }`
`}`
`for (int x = 1; x <= N; x += 2) {`
`  sum++;`
`}`
`cout << sum << endl;`
	$O($_____$)$

c)
`Queue<int> q;`
`for (int i = 1; i <= 4 * N; i++) {`
`  q.enqueue(i);`
`}`
 	
`Map<int, int> map;`
`while (!q.isEmpty()) {`
`  int k = q.dequeue();`
`  map[k] = -2 * k;`
`}`
`cout << "done!" << endl;`
	$O($_____$)$

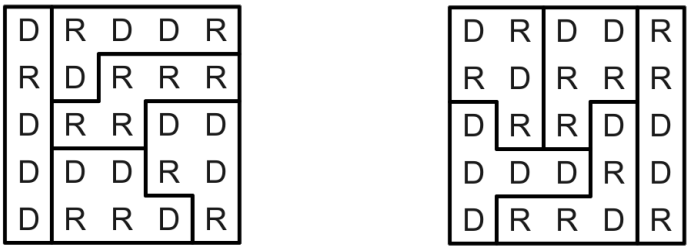
d)
`HashMap<int, int> map;`
`for (int i = 1; i <= N * N; i++) {`
`  map.put(i, i * i + 1);`
`}`
`HashSet<int> set;`
`for (int k : map) {`
`  set.add(map[k]);`
`}`
`cout << "done!" << endl;`
	$O($_____$)$

3. Code Writing - based on a problem idea from Keith Schwarz

Write a function named **`gerrymanderingRatios`** that examines an input file of voting data to calculate whether voting districts have been created in a way that gives an unfair advantage to a particular political party.

Districting is the process by which a US state is divided into voting regions called districts. During an election, the candidate who earns the largest number of votes in a district (the "plurality" of votes) is elected as that district's representative. **Gerrymandering** is a malicious process of trying to draw district boundaries in such a way as to give one party more representatives than their fair share. For example, if you spread the voters from a given party too thinly across many districts, they will not have enough votes to win a majority in any district. Similarly, if you too tightly pack the voters from a political party all into a small number of districts, all votes over the needed plurality (e.g. 50% in a two-party system) are wasted.

In the figure below, the square area of Democrat (D) and Republican (R) voters are divided into 5-person districts. The districting at left leads to 4/5 of the districts won by Democrats, while the districting at right is the opposite, awarding 4/5 of the districts to Republicans. These both seem unfair given the nearly equal total count of voters from the two parties.



gerrymandered toward D (left) and toward R (right)

For this problem let us define the "Gerrymandering ratio" as a measure of a given political party's performance in a given districting arrangement. It is computed by dividing the % of districts that party won by that party's % of the total vote. A ratio of **1.0** would be perfect representation. If the ratio is far above 1.0 for a given party, it implies that the districting was gerrymandered to favor that party. Similarly, if the ratio is far below 1.0, that party might be disadvantaged.

Gerrymandering Ratio for a Party = $\frac{\text{(Percent of districts where that Party won a majority of the vote)}}{\text{(Percent of total votes won by that Party)}}$

Your function accepts as its parameter a reference to an input file of type `ifstream` representing voting data. The box above at right shows an example contents of such a file. Each line of the input file will contains information about one voting district. First will be a single word token representing that district's name, followed by each individual vote from that district, each separated by a single space. The votes will be single uppercase characters indicating the political party of interest, such as D or R.

```
District1 D R D D D
District2 R D R R
DISTRICT3 R R R R R
District4 D D R R D R D
district5 D D R D R
```

Your job is to calculate the gerrymandering ratio for every political party that is represented in the data set. You should return this information as a `Map` from each political party name (string) to that party's gerrymandering ratio (a real number). For example, if the input file stream for the file above on the right is passed to your function, your code should determine that Democrats make up 12/26 (46.1%) of the total voters, but Dems won 3/5 (60%) of the districts by majority vote. This means that the gerrymandering ratio for "D" party is 60 / 46.1, or roughly 1.3. You can perform a similar calculation for Republicans. Based on this information, your function would return the following map:

```
{"D":1.3, "R":0.742857}
```

Assumptions: You may **assume valid file input**, that the file exists and exactly follows the format shown, with party votes in uppercase. You may assume that there is at least one line of district data in the file and that every district has at least one voter, though the districts might not all be the same size. You may assume that there are at least 2 political parties represented, though there might be more than 2 parties. Do not assume that the parties will be named "D" and "R".

Constraints: Choose an **efficient** solution. Choose data structures intelligently and use them properly. While your code is not required to have any particular Big-Oh, you may lose points if your code is extremely inefficient. Do not read the file more than once. Do not define custom classes/structs; solve this problem using the Stanford collections.

Write your answer on the next page.

3. Writing Space

4. Code Writing

Write a **recursive** function named **countOccurrences** that accepts two vectors of integers v1 and v2 by reference, and returns an integer indicating the number of times that the contents of v2 appear in v1. The contents must be consecutive elements and must occur in the same relative order. The following table shows several calls to your function and their expected return values. The occurrences are underlined in the first call as an illustration.

Call	Returns

`Vector<int> v1 {1, 4, 2, 4, 2, 1, 4, 2, 9, 1, 4, 2, 0, 1, 4, 2};`
`Vector<int> v2 {1, 4, 2};`
`countOccurrences(v1, v2)`
	4

`Vector<int> v1 {8, 8, 8, 4, 8, 8, 8, 8, 2, 8, 1, 8, 7, 8, 8};`
`Vector<int> v2 {8, 8};`
`countOccurrences(v1, v2)`
	6

`Vector<int> v1 {1, 2, 3};`
`Vector<int> v2 {1, 2, 3, 4};`
`countOccurrences(v1, v2)`
	0

Note that occurrences of v2 in v1 can partially **overlap**. For example, in the second call above, there is an occurrence of {8, 8} starting at index 0 in v1, and an overlapping occurrence that starts at index 1. The range of indexes 4-7 contains 3 occurrences of v2: one that starts at index 4, one that starts at index 5, and one that starts at index 6.

When your function returns to the caller, the state of the two vectors passed in must be the same as when your function started. Your function should either not modify the vectors that are passed in, or if it does do so, it should restore their state before returning.

You may assume that v2 is non-empty. If v1 is empty, it does not contain any v2 occurrences, so you should return 0.

Constraints: For full credit, obey the following restrictions. A solution that disobeys them can get partial credit.

- **Do not create or use any auxiliary data structures** like additional Queues, Stacks, Vector, Map, Set, array, strings, etc. You should also not call functions that return multi-element regions of a vector, such as sublist.
- **Do not use any loops**; you must use recursion.
- Do not declare any **global variables**.
- You *can* declare as many primitive variables like ints as you like.
- You *are* allowed to define other "**helper**" functions if you like; they are subject to these same constraints.

Write your answer on the next page.

4. Writing Space

5. Code Writing - based on a problem idea from Julie Zelenski

Write a recursive function named **cryptaSolver** that solves Cryptarithm puzzles as described on this page. In a Cryptarithm puzzle, an "addition problem" is posed between two words that "add up" to produce a third word as their sum. For example, consider the puzzle below at left, along with one correct solution to the puzzle at right:

```
  LOVE                    5970
+ HATE          ==>     + 4260
 PEACE                   10230

puzzle                solution
```

The idea is that the letters are each stand-ins for individual **digit values** from 0-9 inclusive. The goal of the puzzle is to see if there is a mapping of letters to numbers that produces a correct mathematical equation where the sum of the first two operands equals the third value. For example, if the letter L is replaced by 5, and O by 9, and V by 7, and so on, you can produce the equation shown above at right, which is a correct solution because 5970 + 4260 does equal 10230.

When substituting letters for numbers, all occurrences of a given letter must map to the same integer value. For example, in our example above, the E in LOVE must become the same integer value as the E in HATE and the two Es in PEACE. No two letters can substitute for the same digit; if the letter H becomes 4, no other letter can also become 4 in that puzzle. It is okay for a leading digit to be a zero, such as if L, H, or P were assigned to the value 0 in the puzzle above.

Your function will be passed a Cryptarithm puzzle by reference as a structure that contains the three strings (two operands and sum), along with two methods for replacing or un-replacing all occurrences of a given letter with a given integer in all three strings. For example, if you call `puzzle.replace('X', 4);` on the structure, all occurrences of the letter `'X'` will become occurrences of the number `4` in the three strings in the puzzle. The struct also defines a `<<` operator that prints the puzzle in the format shown above. Below is the struct's declaration along with an example call to your function:

```
struct Cryptarithm {    // provided
    string op1, op2, sum;
    void replace(char c, int i);
    void unreplace(char c, int i);
};
```

```
// calling your function
Cryptarithm puzzle {"LOVE", "HATE", "PEACE"};
cryptaSolver(puzzle);
```

Your code should stop exploring once it finds and prints any **single correct solution**. A correct solution is defined as one where all characters have been substituted for unique integer values and where the sum of `op1` and `op2` is exactly equal to `sum`. If there is no solution to the given puzzle, your function should produce no output.

The Stanford library functions `stringToInteger` and/or `integerToString` may help you when solving this problem.

Assumptions: You may assume that the puzzle's op1, op2, and sum values are non empty strings that contain only uppercase letters, though the strings might not be the same length. You may assume that the puzzle is always an addition puzzle where you are trying to find a mapping where op1 + op2 is equal to sum, and doesn't involve any other math operations.

Efficiency: While your code is not required to have any particular Big-Oh, you may lose points if your code is extremely inefficient, repeatedly revisits the same path many times, and so on. You should not explore obviously invalid paths, such as trying to assign the same digit value to two or more different letters. In a very clever implementation it would be possible to improve efficiency by checking partial results, such as replacing only the ones digit and seeing if those add up properly before proceeding. It is not required nor recommended to make these kinds of arithmetic optimizations. You can create a full mapping of all the letters in the puzzle to integers before checking whether the mapping produces a correct solution. You should also choose reasonable collections that efficiently implement the operations you need.

Constraints: For full credit, obey the following restrictions. A solution that disobeys them can get partial credit.

- Do not declare any **global variables**.
- You are allowed to use **collections** as needed to help you solve the problem.
- Your code can contain **loops** as needed, but to receive full credit, your overall algorithm must be recursive.
- You are allowed to define other "**helper**" functions if you like; they are subject to these same constraints.

Write your answer on the next page.

5. Writing Space

6. Code Writing

Write a function **`isSorted`** that accepts a reference to a stack of integers as a parameter and returns `true` if the elements in the stack occur in ascending (non-decreasing) order from top to bottom, else `false`. That is, the smallest element should be on top, growing larger toward the bottom. For example, passing the following stack should return `true`:

```
bottom {20, 20, 17, 11, 8, 8, 3, 2} top
```

The following stack is *not* sorted (the 15 is out of place), so passing it to your function should return a result of `false`:

```
bottom {18, 12, 15, 6, 1} top
```

An empty or one-element stack is considered to be sorted.

When your function returns, the stack should be in the **same state** as when it was passed in. In other words, if your function modifies the stack, you must restore it before returning.

For full credit, obey the following restrictions in your solution. A solution that disobeys them can get partial credit.

- You may use **one queue or one stack** (but not both) as auxiliary storage.
- You may not use other structures (arrays, lists, etc.), but you can have as many simple variables as you like.
- Your solution should run in **O(*N*) time**, where *N* is the number of elements of the stack.

Write your answer on the next page.

6. Writing Space

7. Code Writing and Short Answer

Answer the following questions on the next page.

1) **What is Unit Testing, and why is it useful in software development? Explain in 2-3 sentences.**

2) **Examine the specified function behavior below, and write two unique Unit Tests for this function using the testing functions described in class that test distinct cases. For each test, explain in 2-3 sentences why it is an essential test.**

`matchCount` is a recursive function that accepts two references to vectors of integers as parameters and returns the number of elements that match between them. Two elements match if they occur at the same index in both vectors and have equal values. For example, given the two vectors shown below, `matchCount(v1, v2)` would compare as follows:

```
v1: {2, 5, 0, 3, 8, 9, 1, 1, 0, 7}
     |  |  |  |  |  |  |
v2: {2, 5, 3, 0, 8, 4, 1}
```

The function would return 4 in this case because 4 of these pairs match (2-2, 5-5, 8-8, and 1-1). If either vector is empty, by definition it has no matches with the other vector, so the function would return 0. When the code is done running, the two vectors will have the same contents as when the call began.

7. Writing Space

Additional Writing Space