CS 106X Midterm Exam - Question Booklet
Fall 2018
Lecturer: Nick Troccoli

You may not use any internet devices. You will be graded on functionality—but good style saves time and helps graders understand what you were attempting. You have 120 minutes. We hope this exam is an exciting journey.

Note: DO NOT WRITE in this booklet. Only work in the answer booklet will be graded.

Problem 1: Code Reading (20 points)

(Note: you may need to scroll right to view the entire question description).

For both of the calls to the following recursive function below, indicate the **state of the Vector `v` and the Set `s`** that were passed to the function, as well as what value is **returned**.

```
int mystery(Vector<int>& v, Set<int> s, int i, int j) {
    if (i < 0 || j < 0 || i >= v.size() || j >= v.size()) {
        return 0;
    }

    s.add(v[i]);
    if (i == j) {
        return mystery(v, s, i, j + 1);
    } else if (v[i] > v[j]) {
        v[j] += v[i];
        return v[i] + mystery(v, s, i + 1, j);
    } else {
        v[i] += v[j];
        return v[j] + mystery(v, s, i, j + 1);
    }
}
```

a) Call:

```
Vector<int> v = {10, 5, 7};
Set<int> s;
int returnValue = mystery(v, s, 0, 2);
```

b) Call:

```
Vector<int> v = {3, 1, 5};
Set<int> s;
int returnValue = mystery(v, s, 0, 1);
```

Problem 2: Code Reading (20 points)

(Note: you may need to scroll right to view the entire question description).

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 text area.

a)

```
int sum = 0;
for (int i = 0; i < N; i++) {
    for (int j = i; j < N; j++) {
        for (int k = 0; k <  N / 2; k += 2) {
            sum++;
        }
    }
}

for (int i = 0; i < N; i++) {
    cout << i << endl;
}
```

---

b)

```
test2(N);       /* calculate runtime of this line, assuming definition below */
...
void test2(int x) {
    if (x <= 0) return;
    cout << x << endl;
    test2(x - 2);
}
```

---

c)

```
HashMap<int, int> map;
for (int i = 0; i < N; i++) {
    map[i] = i;
}

Set<int> nums;
for (int key : map) {
    nums += key;
}
```

---

d)

```
Stack<int> s;
```

```
for (int i = 0; i < 10000; i++) {
    s.push(N);
}

Queue<int> q;
while (!s.isEmpty()) {
    q.enqueue(s.pop());
}

while (!q.isEmpty()) {
    cout << q.dequeue() << endl;
}
```

Problem 3: Code Writing (20 points)

(Note: you may need to scroll right to view the entire question description).

You are working at a costume store that is gearing up for a busy Halloween. They are interested to find out some statistics about their customers' purchases, and noticing that you are in CS106X, they have enlisted your help.

Your job is to write a function **`Map<string, int> uniqueCostumeCustomers(ifstream& file);`** that reads in a file containing customer purchases and returns a map from each costume to the *number of unique customers that purchased that costume*.

Your function accepts as its parameter a reference to an input file of type **`ifstream`** representing customer orders. The box below shows example contents of such a file. Each line of the input file will contain information about one order. First will be the name of the customer who made the order, contained within double quotes. Then, that name is followed by one or more costumes that they bought as part of that order, each separated by a single space and contained within double quotes. Note that the same customer may make multiple orders (e.g. in the file below, Zach made 2 orders, each on its own line).

```
"Nick Troccoli" "Legend of Zelda" "Pikachu" "Legend of Zelda"
"Zach Birnholz" "Darth Vader" "Legend of Zelda"
"Lucy" "Legend of Zelda" "Darth Vader"
"Nancy" "Hermione Granger" "Pumpkin"
"Zach Birnholz" "Pumpkin" "Legend of Zelda"
"Nick Troccoli" "Pikachu"
```

In this case, the following map would be returned (the order of the key-value pairs does not matter). Note that even though "Legend of Zelda" appeared in 4 orders, it was only purchased by 3 *unique customers*. Similarly, "Pikachu" was only purchased by 1 unique customer. Also note that purchasing the same costume multiple times in an order is permitted, but this would not influence the number of unique customers purchasing the item:

```
// key              ->   value
   Legend of Zelda  ->   3
   Pumpkin          ->   2
   Pikachu          ->   1
   Darth Vader      ->   2
   Hermione Granger ->   1
```

Assumptions: You may **assume valid file input**, that the file exists and exactly follows the format shown. The file may be empty, in which case you should return an empty map. You may assume that each order contains at least one costume. Do not assume the length or number of tokens (words) in either the customer or costume names.

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.

Problem 4: Code Writing (20 points)

(Note: you may need to scroll right to view the entire question description).

Write a **recursive** function named **`combineSortedStacks`** that accepts two sorted (in increasing order) stacks of integers **`s1`** and **`s2`** by reference, and returns a new stack of integers containing all of the values from **`s1`** and **`s2`** sorted in increasing order. The following table shows several calls to your function and their expected return values.

Call	Returns

`// bottom to top`
`Stack<int> s1 = {9, 7, 5, 3, 1};`
`Stack<int> s2 = {8, 6, 4, 2, 0};`
`combineSortedStacks(s1, s2);`
	`{9, 8, 7, 6, 5, 4, 3, 2, 1, 0} <- top`

`// bottom to top`
`Stack<int> s1 = {1, 1};`
`Stack<int> s2 = {2, 2};`
`combineSortedStacks(s1, s2);`
	`{2, 2, 1, 1} <- top`

`// bottom to top`
`Stack<int> s1 = {9, 6, 2, 1};`
`Stack<int> s2 = {8, 7, 5, 4, 3, 0};`
`combineSortedStacks(s1, s2);`
	`{9, 8, 7, 6, 5, 4, 3, 2, 1, 0}; <- top`

Note that in all cases, "sorted in increasing order" means that the smallest element should be at the **top** of the stack, and the largest element should be at the bottom.

Your function is allowed to modify the state of the passed-in stacks, so they do not have to be the same at the end as when they were passed in.

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

- **You may use only one auxiliary data structure**, which is the Stack that you return from your function. You may not make any other data structures like additional **`Queues`**, **`Stacks`**, **`Vector`**, **`Map`**, **`Set`**, array, strings, etc.
- **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.

Problem 5: Code Writing (20 points)

(Note: you may need to scroll right to view the entire question description).

We need your help to assign discussion sections to next quarter's CS106X students! We have information about each student's section preferences, as well as the maximum size of each discussion section. Your job is to **assign each student to a section such that the overall unhappiness of the class is minimized**. Specifically, write the following recursive function named **`assignSections`**:

```
SectionAssignments assignSections(HashMap<string, Vector<int>>& preferences,
                                  Vector<int>& maxSectionSizes);
```

`SectionAssignments` is a struct defined as follows:

```
struct SectionAssignments {
    HashMap<string, int> assignments;
    int overallUnhappiness;
};
```

The first element of this struct is a map from student name to which section number they are assigned. In this problem, we represent sections as numbers, such as section #0, section #1, etc. The second element is the overall unhappiness that results from these section assignments.

The parameters to your function are defined as follows:

- **`preferences`**: a map from student name to a list of their section preferences. This list is formatted such that index *i* is their *i+1th* choice. For instance, if **`preferences["Ali"]`** is **`[4, 3, 0, 2]`**, this means that section #4 is Ali's first choice, section #3 is Ali's second choice, and so on. Each student may have 0 or more preferences, and you can assume that each student ranks a single section at most once. For example, Ali's section preferences could not be [4, 4, 4, 1].
- **`maxSectionSizes`**: a vector of the maximum capacities of each of the offered sections (all capacities are >= 0). For example, if this was [10, 12, 13, 8] then this means that section #0 has maximum size 10, section #1 has maximum size 12, and so on. You may assume that, if a student ranks a certain section, that it will be included in the section sizes vector (e.g. you can assume that if Ali ranks section #4, that this vector will be at least size 5).

Unhappiness is calculated as follows: unhappiness increases by *i* each time a student is assigned their *i+1th* choice. What this means is that if Ali gets his first choice, there is no unhappiness (yay!). If Ali gets his second choice, there is +1 unhappiness. If Ali gets his third choice, there is +2 unhappiness, and so on.

You must find the section assignments for each student that minimize their overall unhappiness. If there are multiple answers with the same best unhappiness, you may choose any of them. Note that you can **only** assign a student to a section that is included in their preferences. If you cannot find any possible section assignment with a given set of parameters, you should return a **`SectionAssignments`** struct containing an empty map and **`INT_MAX`** unhappiness (sad times).

As an example, let's say we have the following parameters passed in:

```
HashMap<string, Vector<int>> prefs;
prefs["Jack"] = {0};
prefs["Aleksander"] = {0, 1, 2};
```

```
prefs["Rachel"] = {1, 0, 2};
Vector<int> space = {1, 1, 1};
SectionAssignments s = assignSections(prefs, space);
```

In this case, Jack would be assigned section 0, Aleksander section 2, and Rachel section 1, for an overall unhappiness of 2. This is because if Aleksander was put in section 1 and Rachel in section 2, the overall unhappiness would be 1 from Aleksander and 2 from Rachel = 3 total.

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.
- When your function returns to the caller, the state of the HashMap and Vector 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.

Problem 6: Code Writing (20 points)

(Note: you may need to scroll right to view the entire question description).

Write a function **`swapK`** that accepts a reference to a queue of integers and an integer **`k`** as parameters and modifies the queue by flipping pairs of the first *k* elements. That is, if we pass in the following queue with k=4:

```
front {1, 2, 3, 4, 5, 6, 7, 8} back
```

the queue should be modified to contain:

```
front {2, 1, 4, 3, 5, 6, 7, 8} back
```

If k is odd, then the final element in the k-sequence should be left in place. For example, with the following queue and k=7:

```
front {1, 2, 3, 4, 5, 6, 7, 8} back
```

the queue should be modified to contain:

```
front {2, 1, 4, 3, 6, 5, 7, 8} back
```

A k of 0 or 1 should therefore not modify the passed-in queue. If the k passed in is larger than the queue size, you should throw an error by calling the **`error()`** function and passing in an error string as a parameter.

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

- You may **not** use any auxiliary data structures like additional **`Queues`**, **`Stacks`**, **`Vector`**, **`Map`**, **`Set`**, array, strings, etc., though you may have as many simple variables (int, string, etc.) as you like.
- Your solution should run in ***O(N)* time**, where *N* is the number of elements of the queue.

As part of this problem, you must also write **2 unit tests** to test the functionality of the function. You should include a comment above each unit test explaining what it is testing and why it is a useful test case. As a reminder, you can add a new test case using the following syntax:


```
// comment explaining test
```

```
ADD_TEST("my test name here") {
    // my test code here
}
```

Inside your test, you can use the **`expect(expression)`** function to test functionality. It accepts a boolean expression and, if the expression evaluates to **`false`**, the test will fail. If the expression evaluates to **`true`**, the test will pass. You can call **`expect()`** multiple times in a single test, and the test will pass only if all of the calls' parameters evaluate to **`true`**.