CSC 016 Midterm (Sample)

Code reading 1

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.

small . big

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:	

Code reading 2

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

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

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

d) `HashMap map;`
n^2 `for (int i = 1; i <= N * N; i++) {` n^2
  `map.put(i, i * i + 1);`
`}`
`HashSet set;`
n^2 `for (int k : map) {`
  `set.add(~~map[k~~]);`
`}`
`cout << "done!" << endl;`
$n^2 + n^2$
	O(n^2)

Code writing 1

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

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

`Vector v1 {1, 2, 3};`
`Vector 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.

Code writing 2

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.