

CS 106X Autumn 2018 Practice Midterm Exam
ANSWER KEY

Based on Autumn 2017 CS 106X Midterm Exam Solutions

Copyright © Stanford University, Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.

1. Code Reading

a)
mystery("OXIDIZE")

after : "EIIDXZ0"
return: 3

b)
mystery("TCTTCGTCCGAACCAGA")

after : "ACACCCACCAGTGTTGT"
return: 10

2. Code Reading

- a. $O(N^2)$
- b. $O(N)$
- c. $O(N \log N)$
- d. $O(N^2)$

3.

Every code-writing problem can be solved in multiple ways. Here is one example solution.

```
Map<string, double> gerrymanderingRatios(ifstream& file) {
    Map<string, double> ratios;
    int totalCityCount = 0;
    int totalVoteCount = 0;
    Map<string, int> voteCountsTotal;
    Map<string, int> majorityPartyCount;

    // read the district data from the file
    string line;
    while (getline(file, line)) {
        // read the data about one district
        istringstream input(line);    // "District1 A B A A B"
        string district;
        input >> district;
        totalCityCount++;

        // count the votes and figure out the winning majority party
        string party;
        string maxParty = "";
        Map<string, int> voteCountsCity;
        while (input >> party) {
            totalVoteCount++;
            voteCountsCity[party]++;
            voteCountsTotal[party]++;
            if (maxParty == "" || voteCountsCity[party] > voteCountsCity[maxParty]) {
                maxParty = party;
            }
        }
        majorityPartyCount[maxParty]++;
    }

    // compute the gerrymandering ratio for each party
    for (string party : voteCountsTotal) {
        // = (% of districts with this party as majority) / (% of total votes)
        double pctOfDistricts = (double) majorityPartyCount[party] / totalCityCount;
        double pctOfTotalVotes = (double) voteCountsTotal[party] / totalVoteCount;
        ratios[party] = pctOfDistricts / pctOfTotalVotes;
    }
}

return ratios;
}
```

```

// solution 1
int countOccurrencesHelper(Vector<int>& v1, Vector<int>& v2, int i1, int i2) {
    if (i2 >= v2.size()) {
        // reached end of v2; found a match
        return 1 + countOccurrencesHelper(v1, v2, i1 + 1, /* i2 */ 0);
    } else if (i1 + i2 >= v1.size()) {
        // base case: reached end of v1; stop
        return 0;
    } else if (v1[i1 + i2] == v2[i2]) {
        // matches so far; keep going
        return countOccurrencesHelper(v1, v2, i1, i2 + 1);
    } else {
        // elements don't match; start over at next index
        return countOccurrencesHelper(v1, v2, i1 + 1, /* i2 */ 0);
    }
}

int countOccurrences(Vector<int>& v1, Vector<int>& v2) {
    return countOccurrencesHelper(v1, v2, 0, 0);
}

=====
// solution 2
// returns true if v2 is found in v1 starting at index i
bool isMatch(Vector<int>& v1, Vector<int>& v2, int i, int j) {
    if (j >= v2.size()) {
        return true;
    } else if (i >= v1.size()) {
        return false;
    } else {
        return v1[i] == v2[j] && isMatch(v1, v2, i + 1, j + 1);
    }
}

void countOccurrencesHelper(Vector<int>& v1, Vector<int>& v2, int i, int& total) {
    if (i > v1.size() - v2.size()) {
        // base case: finished looking at whole array
        return;
    } else {
        // recursive case: first check for a match starting at this index
        if (isMatch(v1, v2, i, 0)) {
            total++;
        }
        // now check for matches starting in the rest of v1
        countOccurrencesHelper(v1, v2, i + 1, total);
    }
}

int countOccurrences(Vector<int>& v1, Vector<int>& v2) {
    int total = 0;
    countOccurrencesHelper(v1, v2, 0, total);
    return total;
}

```

5.

```
bool cryptaSolver(Cryptarithm& crypt) {
    Set<char> letters;
    for (char c : (crypt.op1 + crypt.op2 + crypt.sum)) { letters.add(c); }
    Set<int> digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    return cryptaSolverHelper(crypt, letters, digits);
}

bool cryptaSolverHelper(Cryptarithm& crypt, Set<char>& letters, Set<int>& digits) {
    if (letters.isEmpty()) {
        // base case: check whether puzzle solution works
        if (stringToInteger(crypt.op1) + stringToInteger(crypt.op2)
            == stringToInteger(crypt.sum)) {
            cout << crypt << endl << endl;
            return true;
        }
    } else {
        // recursive case: process the next letter
        char letter = letters.first();
        letters.remove(letter);

        for (int i = 0; i <= 9; i++) {
            if (digits.contains(i)) {
                digits.remove(i);                                // choose
                crypt.replace(letter, i);
                if (cryptaSolverHelper(crypt, letters, digits)) { // explore
                    return true;
                }
                crypt.unreplace(letter, i);                      // un-choose
                digits.add(i);
            }
        }
        letters.add(letter);
    }
    return false;
}
```

6.

```
// using an auxiliary stack
bool isSorted(Stack<int>& s) {
    if (s.size() < 2) {
        return true;
    }
    bool sorted = true;
    int prev = s.pop();
    Stack<int> backup;
    backup.push(prev);
    while (!s.isEmpty()) {
        int curr = s.pop();
        backup.push(curr);
        if (prev > curr) {
            sorted = false;
        }
        prev = curr;
    }
    while (!backup.isEmpty()) { // restore s
        s.push(backup.pop());
    }
    return sorted;
}
```

Unit testing is a technique where you write a series of short, distinct tests that each cover a small, isolated part of a function or class's functionality. Unit testing helps isolate and catch bugs, and also outline the expected behavior of code.

Here are two essential unit tests:

```
ADD_TEST("Test one empty list") {
    Vector<int> v1 = {1};
    Vector<int> v2;
    expect(matchCount(v1, v2) == 0);
}
```

This test covers an essential edge case for this function, which is when one of the two lists is empty (in which case, as outlined in the spec, the function should return 0).

```
ADD_TEST("Test two equal short lists") {
    Vector<int> v1 = {1, 4, 5};
    Vector<int> v2 = v1;
    expect(matchCount(v1, v2) == v1.size());
}
```

This test covers the case when both lists are identical, testing the important equality checking in this function. In this case, the match count should be the length of one of the lists.