CS106X Instructor: Cynthia Lee

Autumn 2015 October 27, 2015

MIDTERM EXAM – SOLUTIONS

---

1. **ADTs (34pts).**

```
string buildRepetitionPattern(const string& word); // used in (a) and (b)

// Code for Part (a)

Map<string, Vector<string>> getPatterns(const Lexicon &english) {
  Map<string, Vector<string>> allPatterns;
  for(const string& word : english) {
      allPatterns[buildRepetitionPattern(word)].add(word);
  }
  return allPatterns;
}

string buildRepetitionPattern(const string& word) {
  Map<char, char> subChars;
  string pattern;
  char subChar = 'A';
  for (int i = 0; i < word.length(); i++) {
    if (subChars.containsKey(word[i])) {
      pattern += subChars[word[i]];
    } else {
      pattern += subChar;
      subChars[word[i]] = subChar;
      subChar++;
    }
  }
  return pattern;
}

// Code for Part (b)

bool satisfiesSubstitution(const string& origin, const string& targt,
                           const Map<char, char>& knownSubstitutions);

Vector<string> getCandidatePlaintext(string ciphertextWord,
          const Map<string, Vector<string>> &patternToWords,
          const Map<char, char> &knownSubstitutions) {
  Map<char, char> reverseSubstitutions;
  for (const char& ch : knownSubstitutions) {
```

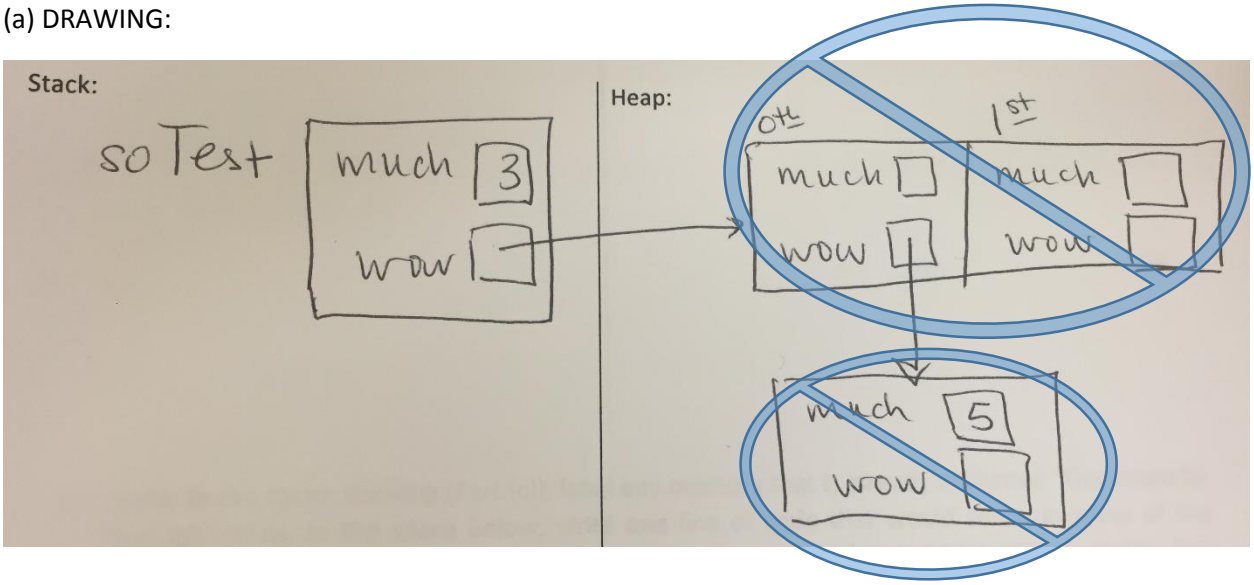
```
    reverseSubstitutions[knownSubstitutions[ch]] = ch;
  }
  Vector<string> candidatePlaintext =
                              patternToWords[buildRepetitionPattern(cipherTextWord)];
  Vector<string> filteredPlaintext;
  for (const string& candidate : candidatePlaintext) {
    if (satisfiesSubstitution(candidate, cipherTextWord, knownSubstitutions) &&
        satisfiesSubstitution(cipherTextWord, candidate, reverseSubstitutions)) {
      filteredPlaintext.add(candidate);
    }
  }
  return filteredPlaintext;
}

bool satisfiesSubstitution(const string& origin, const string& target,
                           const Map<char, char>& knownSubstitutions) {
  for (int i = 0; i < origin.length(); i++) {
    if (knownSubstitutions.containsKey(origin[i]) &&
        knownSubstitutions[origin[i]] != target[i]) {
      return false;
    }
  }
  return true;
}
```

---

2. **Pointers and Memory (18pts).**

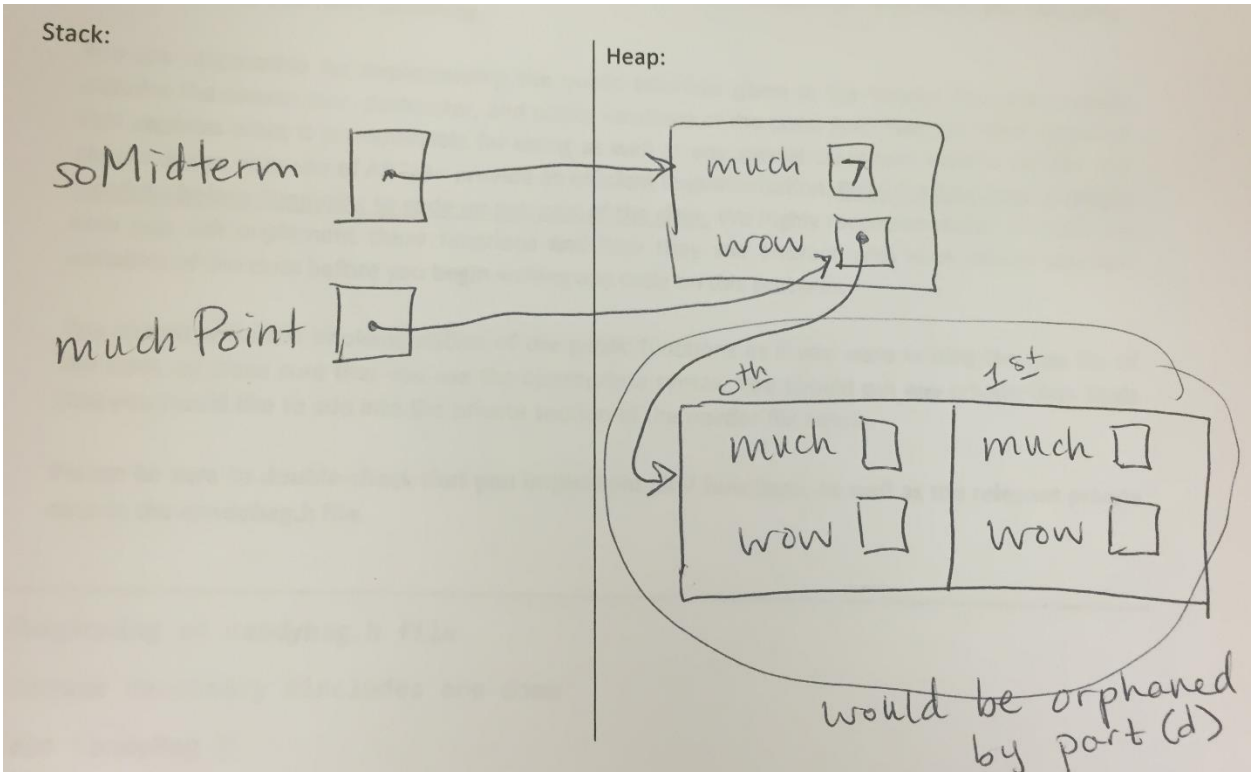
(a) DRAWING:



(b) CODE:

```
delete soTest.wow[0].wow;
delete[] soTest.wow;
```

(c) DRAWING:



(d) CODE: (there are several correct solutions, these are two possibilities)

```
(*muchPoint) = NULL;
/* or */
soMidterm->wow = NULL;
```

---

3. **Classes (30pts).**

```
// In candybag.h

private:
    Map<CandyBar, int> candyCount;
    Map<string, Set<double>> candyWeights; // Avoid dups
    double totalWeight;
    int numCandyBars;
    double weightLimit;

    double weightOfType(const string& type);
};
// End of candybag.h file
```

```
// Beginning of candybag.cpp file

#include "candybag.h"

// O(1)
CandyBag::CandyBag(const double& weightLimit, const double& bagWeight) {
    totalWeight = bagWeight;
    numCandyBars = 0;
    this->weightLimit = weightLimit;
}

// O(1)
CandyBag::~CandyBag() { }

// O(Number of Keys)
Vector<string> CandyBag::getCandyBarTypes() const {
    return candyWeights.keys();
}

// O(log N)
void CandyBag::addCandyBar(const string& type, const double& weight) {
    if (weight + totalWeight > weightLimit) {
        throw "I'm sorry, I cannot seem to hold more candy.";
    }

    candyCount[CandyBar(type, weight)]++;
    candyWeights[type] += weight;
    totalWeight += weight;
    numCandyBars++;
}

// O(log N)
int CandyBag::getNumOfDistinctCandyTypes(const string& type) {
    return candyWeights.get(type).size(); // .get() is safe to use here
}

double CandyBag::weightOfType(const string& type) {
    double totalWeight = 0.0;

    // .get() will return a default value if the map does not contain the key type
    for (const double& weight : candyWeights.get(type)) {
        totalWeight += weight * candyCount[CandyBar(type, weight)];
    }

    return totalWeight;
}
```

```
// Requires a Set for efficiency
double CandyBag::getWeightOfCandyTypes(const Set<string>& types) {
    double weight = 0.0;

    for (const string& type : types) {
        weight += weightOfType(type);
    }

    return weight;
}

// O(log N)
CandyBag::CandyBar CandyBag::eatCandyBar() {
    if (numCandyBars == 0) {
        throw "Drat, no candy left!";
    }

    CandyBar result;
    // Get the first element in O(log N) time
    for (const CandyBar& candy : candyCount) {
        result = candy;
        break;
    }

    candyCount[result]--;
    if (candyCount[result] == 0) {
        candyCount.remove(result);
        if (candyWeights[result.type].size() == 1) {
            candyWeights.remove(result.type);
        } else {
            candyWeights[result.type].remove(result.weight);
        }
    }

    totalWeight -= result.weight;
    numCandyBars--;

    return result;
}
```

---

4. Recursion (26pts).

```
Set<Point> magicWand(const Grid<int> &image, Point pixel, int threshold)  {
    Set<Point> allPoints;
    magicWand(image, pixel, image[pixel.row][pixel.col], threshold, allPoints);
    return allPoints;
}

void magicWand(const Grid<int> &image, Point pixel, int origColor, int threshold,
Set<Point>& allPoints)  {
```

```
    if (allPoints.contains(pixel) || !image.inBounds(pixel.row, pixel.col) ||
        pixelDiff(image[pixel.row][pixel.col], origColor) >= threshold) {
        return;
    }
    allPoints.add(pixel);
    magicWand(image, Point(pixel.row+1, pixel.col), origColor, threshold, allPoints);
    magicWand(image, Point(pixel.row-1, pixel.col), origColor, threshold, allPoints);
    magicWand(image, Point(pixel.row, pixel.col+1), origColor, threshold, allPoints);
    magicWand(image, Point(pixel.row, pixel.col-1), origColor, threshold, allPoints);
}
```

---

5. **Big-O (12pts).**

O(N^2)
O(N^2)
O(N^2)
O(logN)