Fall 2015

CIS 121

Data Structures and Algorithms with Java

Recitation Notes Compendium

CIS 121—Data Structures and Algorithms with Java - Fall 2015

Lab 1 — Monday, August 31/Tuesday, September 1

Learning Goals

During this lab, you will:

- approach problems with consideration for runtime complexity
- apply sorting as a tool for solving some real problems
- consider tradeoffs between different solutions to problems
- get a feel for the sort of problem solving that this course will focus on

Problem 0: General Approaches

When faced with these sorts of algorithmic problems, there are some general considerations to take:

- Is there a known algorithm that solves this problem, or a related problem?
- Which kinds of data structures fit the operations we need to do?
- Do we have bounds on our input or on our runtime requirement?
- Does the problem give us additional information that might help us do better?
- Ignoring efficiency considerations, what is the very first working solution you can come up with?

Problem 1: Unique Character Strings

Given a string of length n, determine whether it contains all unique characters – or, whether any characters are used more than once.

Food for Thought:

- If you do it as fast as possible, how much extra space is needed?
- If you do it with no extra space, how quickly can you do it?
- What if you aren't worried about destroying the input string?

Problem 2: Anagram Strings

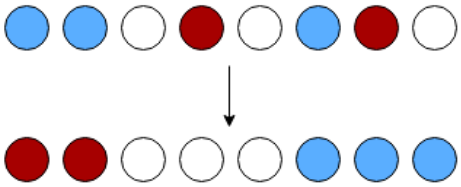
Given two strings, determine whether they are anagrams of each other.

Example:
`TOMMARVOLORIDDLE` $\longleftrightarrow$ `IAMLORDVOLDEMORT`
The above are anagrams of each other!
(Sorry if we spoiled book 2 of Harry Potter for you...)

- If you do it as fast possible, how much extra space is needed?
- If you do it with no extra space, how quickly can you do it?
- What if you aren't worried about destroying the input strings?

Problem 3: Dutch National Flag Problem

Given n balls of three colors – say red, white, and blue – arranged randomly in a line, sort them as quickly as possible into contiguous groups of red, white, and blue, with the groups in that order.



- How fast can generic sorts do this?
- What additional information do we have, beyond what generic sorts assume?
- Can we sort it in-place in $O(n)$ time?

Problem 4: Row/Column-sorted Matrix Membership

Given an m by n matrix of integers, where each row and each column of the matrix is sorted, determine whether an integer k exists somewhere in the matrix.

- How could we do this if we didn't know the rows and columns were sorted?
- How does knowing the rows and columns are sorted let us do this faster?
- What is our optimal runtime?

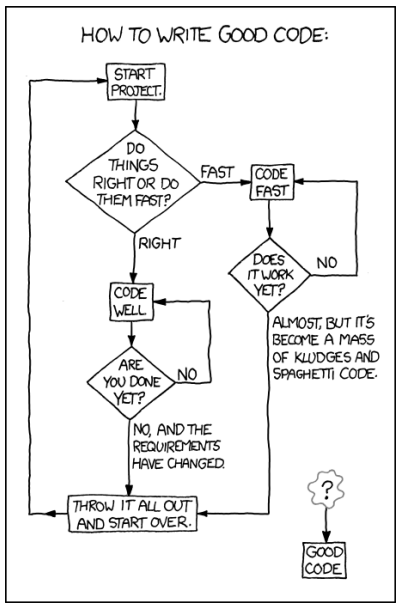
CIS 121—Data Structures and Algorithms with Java - Fall 2015

Lab 1 — Monday, August 31/Tuesday, September 1

Introduction

Agenda

- Self/Student Introductions!
- Brief discussion of course and breadth of its material.
  - Algorithms
  - Data Structures
  - Runtime Complexity
- Where will this course take you?
  - Interview preparation, jobs
  - Smart, informed coding choices and profiling
  - Start to doubt the world around you. Beware of black box code!
  - Theory-land
- Course caveats (at least in past semesters):
  - Lots of math, for Big Oh proofs/statistical analysis later on.
  - Java, JUnit (If you don't feel comfortable, please let me know! I will try to help.)
  - Less hand-holding.
  - Walk the walk! If you don't do the work, you're going to have a hard time.



XKCD: 844

Problem 1

Unique Character Strings: Given a string of length n, determine whether it contains all unique characters - or, whether any characters are used more than once.

Tips: Guess the lower-bound! What is the limit on the fastest runtime algorithm that anyone can formulate for this problem?

Naive Solution. Iterate through each character c in the string s. For each such c, iterate starting from the next character to the end of the string. If a duplicate character appears, return false. Otherwise, return true.

```
1   //naive solution
2   boolean hasNoRepeats(String s) {
3       for (int i = 0; i < s.length(); i++) {
4           for (int j = i+1; j < s.length(); j++) {
5               if (s.charAt(i) == s.charAt(j)) {
6                   return false;
7               }
8           }
9       }
10      return true;
11  }
```

Runtime: Quadratic.

How might you guess that you could improve this runtime? Notice how this algorithm will repeat itself by examining the same parts of the string over and over again. Trying to avoid pointless repetition is a key to success.

Alternate solution. Suppose the string s only contains characters from the English alphabet. Create a boolean array B of size 26, with all values initially false. Iterate through s and examine each character in order. Suppose that at any particular point in the iteration WLOG you have the ith character in the alphabet. Check if $B[i]$ is true. If it is, return false. Otherwise, set it to true and continue. At the end of the iteration, return true.

Runtime: Linear (in terms n). Space usage is constant.

Suggested improvements: Use a bitstring of maximal size $\log m$ instead of an array. (m is size of the alphabet.) Example solution in C shown on the next page, **for the 240 folk.** (It's ok if you don't understand. C is not covered in this course!)

Alternate solution 2. Iterate through each character of the string and add each to a set S. If S already contains the character, return false. Otherwise, return true.

Runtime: It depends! How is the set implemented? Is it a black box? If you don't know the runtime of set add/contains, you can't use this result.

```c
// pr0.c
// Implementation of alternate solution 1.
// READ IF CURIOUS. NOT COVERED IN COURSE MATERIAL.
// Returns 1 if string has no repeat characters.
// Otherwise, returns 0 if it finds any.
//
// @param s - string of lowercase alphabetical chars

int hasNoRepeats(char *s) {
    unsigned int b = 0;

    for (int i = 0; i < strlen(s); i++) {
        int offset = s[i] - 'a';

        if (1 & b >> offset) {  //Examine the ith bit.
            return 0;           //If set, duplicate found!
        }

        b |= (1 << offset);     //Mark bit as found.
    }

    return 1;
}
```

Problem 2

Anagram Strings: Given two strings, determine whether they are anagrams of each other.

TOMMARVOLORIDDLE $\longleftrightarrow$ IAMLORDVOLDEMORT

The above are anagrams of each other!
(Sorry if I spoiled book 2 of Harry Potter for you...)

Solution 1. Suppose the two strings are s_1 and s_2 and we are allowed to modify them. Compare their lengths. If unequal, return false. Compare the strings. If they are equal, return true. Iterate through s_1. For each character c, try removing c from s_2. If $c \notin s_2$, return false. When finished, if s_2 is not empty, return false. Return true otherwise.

Runtime: Quadratic. At each iteration we search in s_2, doing $n + (n-1) + (n-2) + ... + 1 = \frac{n \cdot (n+1)}{2}$ total work in the worst case.

Solution 2. Sort both strings and compare them for equality.

Runtime: It depends. There are variety of sorting algorithms out there, from the speedy quicksort to the *infamous* bogosort! Using quicksort or any comparison based sort, the lower bound will be linear-logarithmic $(n \log n)$. You'll learn more about this later in the course.

Solution 3. Suppose the two strings are s_1 and s_2 and that both have a fixed alphabet (English). Create two arrays, A and B and initialize their entries to 0. Iterate through s_1. For each character c, if it is the ith letter in the alphabet, increment $A[i]$. Repeat for s_2, using B instead. When finished, iterate through A and B at the same time, comparing the values of the entries. If for any i, $A[i] \neq B[i]$, return false. Otherwise, return true when finished.

Runtime: Linear, and constant space. Observe that the first loop runs n times. The second loop runs a constant number of times, also doing a constant amount of work.

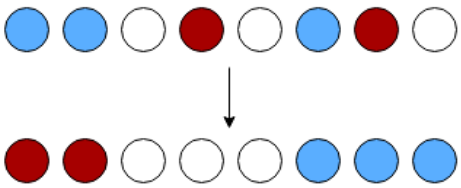
Suggested improvements: To improve space usage, instead of having two arrays - create one. Each time a character shows up in s_1, increment by 1. Each time a character appears in s_2, decrement by 1. In the

second loop, check for nonzero entries.

```
/**
 * Returns whether two strings are anagrams of each other.
 *
 * @param s1, s2 - the input strings. Assumed to contain
 *                 only letters of the English alphabet.
 */
boolean areAnagrams(String s1, String s2) {
    if (s1.length() != s2.length()) {
        return false;
    }
    if (s1.equals(s2)) {
        return true;
    }

    int A[] = new int[26], B[] = new int[26];
    for (int i = 0; i < s1.length(); i++) { //Compute the histograms of s1, s2.
        A[s1.charAt(i) - 'a']++;
        B[s2.charAt(i) - 'a']++;
    }

    for (int i = 0; i < 26; i++) {          //Compare histograms.
        if (A[i] != B[i]) {
            return false;
        }
    }

    return true;
}
```

Problem 3

Dutch National Flag: Given n balls of three colors - say red, white, and blue - arranged randomly in a line, sort them as quickly as possible into contiguous groups of red, white, and blue, with the groups in that order.



How fast can generic sorts do this?

Selection and insertion sort can do this in quadratic time. Not exactly the best choice.

Mergesort can solve this in linear-logarithmic time... but not in-place.

Quicksort can do the sort in average-case linear-logarithmic time and worst-case quadratic time, in-place.

Linear-time algorithm. Suppose the array A is given as an input, which contains the initial ordering of the balls. Assume that red balls are represented as 0's, white 1's, and blue 2's. Create an array B of size 3, where the entries are initialized to 0. Iterate through A. For each occurrence of a colored ball, increment the corresponding entry in B. When finished, iterate through B. For $i = 0$, output $B[0]$ red balls - and so forth.

```
    //
    int[] dutchNationalFlagSort(int[] A) {
        int B[] = new int[3];

        for (int b : A) {
            B[b]++;
        }

        int count = 0;
        for (int i = 0; i < B.length; i++) {
            for (int j = 0; j < B[i]; j++) {
                A[count++] = i;
            }
        }

        return A;
    }
```

The above is an implementation of **counting-sort**, which you'll learn later on (probably in 320.)