Fall 2015

CIS 121

Data Structures and Algorithms with Java

Recitation Notes Compendium

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

Introduction Lab — 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

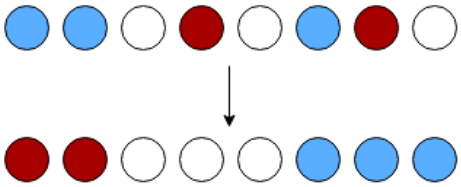
Given two strings, determine whether they are anagrams of each other.

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

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

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.

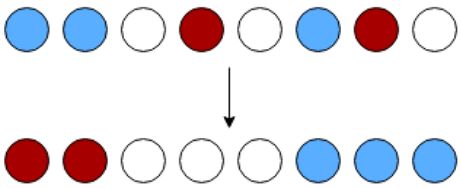
Suggested improvements: To improve space usage, instead of having two arrays - create one. Each time a character shows up in s_1, increment by 1. Each time a character appears in s_2, decrement by 1. In the

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 (in CIS 320).

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?*

  Solution 1. If we can not presume anything about the structure of the matrix, we have no choice but to check each element manually! This of course leads to worst case $O(mn)$ time.

```
        public boolean containsK(int[][] A, int k) {
            for (int i = 0; i < A.length; i++)
                for (int j = 0; j < A[0].length; j++)
                    if (A[i][j] == k)
                        return true;
            return false;
        }

```

- *How does knowing the rows and columns are sorted let us do this faster?*

  If we know that the rows and columns are ordered (suppose in ascending order), this allows us to rule out certain parts of the matrix when searching.

  Solution 2. Compare k to the top-right element of the matrix, c. If $k = c$, return true. If $k > c$, then we can rule out the top row because it is sorted and c is its largest element. If $k < c$, then we can rule out the right column, since it is sorted and c is the smallest element. Repeat until either k is found or every row/column is eliminated without reaching the $k = c$ case. Return false.

- *What is our optimal runtime?*

  The optimal runtime of the above algorithm is $O(m + n)$. At each iterative step, we eliminate either a row or a column and decreasing the overall problem size. Therefore, the worst case runtime is bounded by the total number of rows and columns.

```java
/**
 * Recursive implementation of linear-time search for
 * an element $k$ in $A$.
 * @param A - 2D array of integers, the input matrix
 * Guaranteed to have sorted rows and columns.
 * @param k - number to search for
 * @return whether or not $k$ is an element of $A$
 */
public boolean containsK(int[][] A, int k) {
    return containsKHelper(A, 0, A[0].length-1, k);
}

/**
 * Recursive helper function for `containsK`.
 * Receives a 'submatrix' of the original that shrinks
 * as the algorithm eliminates rows/columns in it search.
 * @param A - 2D array of integers
 * @param minRow - the minimum row of the submatrix
 * @param maxCol - the maximum column of the submatrix
 * @param k - number to search for
 * @return whether or not $k$ is an element of $A$
 */
public boolean containsKHelper(int[][] A, int minRow, int maxCol, int k) {
    // Base case.
    if (minRow >= A.length || maxCol < 0) {
        return false;
    }

    int c = A[minRow][maxCol];
    if (k == c)
        return true;
    else if (k > c)
        return containsK(A, minRow+1, maxCol, k);
    else
        return containsK(A, minRow, maxCol-1, k);
}
```

```java
/**
 * Iterative implementation of linear-time search
 * for an element $k$ in $A$.
 */
public boolean containsK(int[][] A, int k) {
    int minRow = 0, maxCol = A[0].length-1;

    while (minRow < A.length && maxCol >= 0) {
        int c = A[minRow][maxCol];
        if (k == c)
            return true;
        else if (k > c)
            minRow++;
        else
            maxCol--;
    }
    return false;
}
```

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

Big-Oh Lab — TBD

Learning Goals

During this lab, you will:

- review Bachmann-Landau notation
- examine certain functions and their relative asymptotic growth rates
- examine the runtime complexity of code
- prove Bachmann-Landau relations

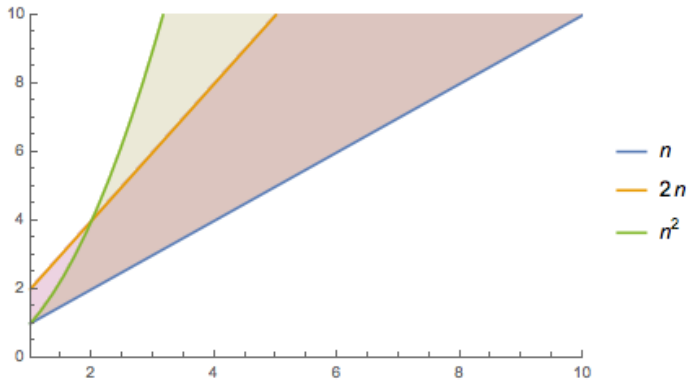
Big-Oh and Bachmann-Landau Notation

In class, you have started to discuss Big Oh and other ways of classifying functions and algorithms. These notations belong to what is commonly referred to as the *Bachmann-Landau* family of notations.

Big-Oh Notation

Definition. $f(n) = O(g(n))$ if there exist constants n_0 and $c > 0$ s.t. $f(i) \leq cg(i)$ for all $i \geq n_0$.

Simplified: If $f(n)$ is $O(g(n))$, $g(n)$ is an asymptotic upper bound for $f(n)$.

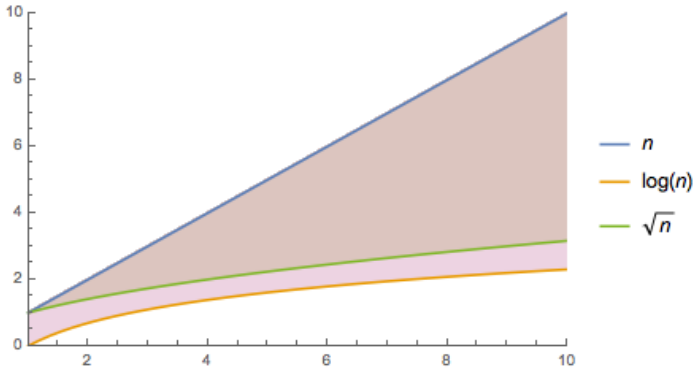


$n = O(n)$, $n = O(n^2)$

Big-Omega Notation

Definition. $f(n) = \Omega(g(n))$ if there exist constants n_0 and $c > 0$ s.t. $f(i) \geq cg(i)$ for all $i \geq n_0$.

Simplified: If $f(n)$ is $\Omega(g(n))$, $g(n)$ is an asymptotic lower bound for $f(n)$.

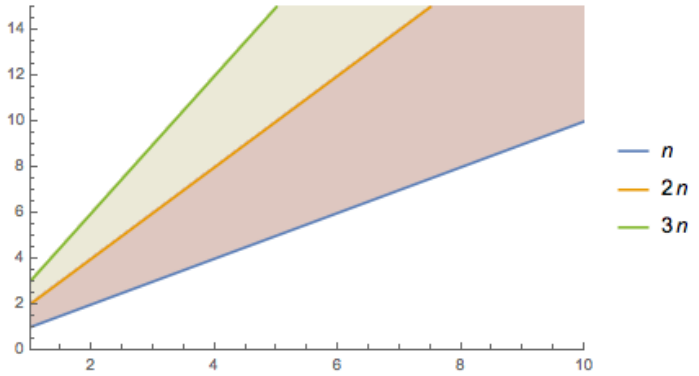


$n = \Omega(\lg n)$, $n = \Omega(\sqrt{n})$

Big-Theta Notation

Definition. $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Simplified: If $f(n)$ is $\Theta(g(n))$, $g(n)$ is an asymptotic *tight* bound for $f(n)$.



$n = \Theta(n)$. Every function is Big-Θ of itself.

As a protip, it is also good to note that the Bachmann-Landau notations refer to *classes of functions*. When you read $f(n) = O(g(n))$, this is equivalent to the statement:

$$f(n) \in O(g(n))$$

. Specifically, $f(n)$ is in the class of functions which are asymptotically bounded above by $g(n)$. Likewise, Big-Ω and Big-Θ both reflect classes of functions.

Stirling's Approximation

$$n! \sim (\frac{n}{e})^n \sqrt{2\pi n}$$

Lab Problems

Problem 1

Order the following functions such that if f precedes g, then $f(n)$ is $O(g(n))$.

$$\sqrt{n},\ n,\ n^{1.5},\ n^2,\ n\lg n,\ n\lg\lg n,\ n\lg n^2,\ 2^{n/2},\ 2^n,\ \lg(n!),\ n^2\lg n,\ n^3,\ 2^{2^n}$$

Problem 2

Provide a runtime analysis of the following loop:

```
for(int i = 0; i < n; i++)
    for (int j = i; j <= n; j++)
        for (int k = i; k <= j; k++)
            sum++;
```

Problem 3

In this problem, you are **not** allowed to use the theorems about Big-Oh stated in the lecture notes. Your proof should follow exclusively from the definition of Big-Oh.

Prove or disprove the following statement:

$$f(n) + g(n) \text{ is } \Theta(\max\{f(n), g(n)\}), \text{ where } f, g : R \to R^+.$$

Problem 4

Prove or disprove the following statement:

$$2^n \text{ is } O(n!).$$

Problem 5

Provide a runtime analysis of the following loop:

```
for (int i = 2; i < n; i = i*i)
    for (int j = 1; j < Math.sqrt(i); j = j+j)
        System.out.println("*");
```

Problem 6

Prove or disprove the following statement:

$$\lg(n!) \text{ is } \Theta(n\lg n).$$

Logarithms Cheat Sheet

Exponential terms appear very frequently in the study of algorithms and their runtimes. Therefore, logarithms are very useful when manipulating exponential terms in Big-Oh proofs! It is therefore advised that you become very familiar with logs.

Here's a little cheat-sheet for you to refresh your memory!

Properties of Logarithms

$$\log(c \cdot f(n)) = \log c + \log f(n)$$

$$\log x^y = y \log x$$

$$\log a + \log b = \log(ab)$$

$$\log a - \log b = \log(a/b)$$

$$\log_b x = \frac{\log_c x}{\log_c b}$$

$$\log 2^n = n$$

$$\log n! = \log[n \cdot (n-1)...2 \cdot 1] = \log n + \log(n-1) + ... + \log 1 = \sum_{i=1}^{n} \log i$$

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

Master Theorem Lab — TBD

Learning Goals

During this lab, you will:

- review the Simplified Master Theorem
- solve recurrences by iteration and using the S.M.T.
- identify recurrences that can not be solved using the S.M.T.

Simplified Master Theorem

The **master theorem** is a powerful tool in the analysis and classification of recurrences. It may be used to easily *classify* recurrences that might otherwise be very time-consuming!

Simplified Master Theorem

Given a recurrence $T(n)$ of the form,

$$T(n) = \begin{cases} c & n < c_1 \\ aT(n/b) + \Theta(n^i) & n \geq c_1 \end{cases}$$

Case I: If $a > b^i$ then $T(n) = \Theta(n^{\log_b a})$.

Case II: If $a = b^i$ then $T(n) = \Theta(n^i \log_b n)$.

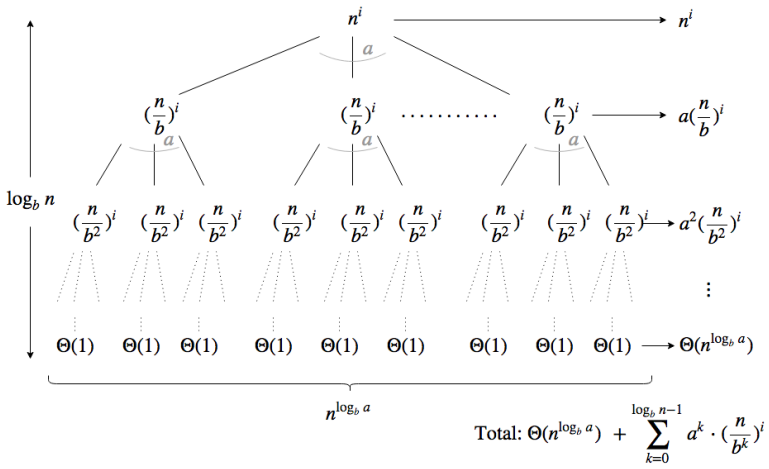
Case III: If $a < b^i$ then $T(n) = \Theta(n^i)$.

This probably seems very magical and hand-wavy. But we're computer scientists, so let's delve in and figure this out. Let us assume here that n is some power of b.

You don't need to be a wizard to understand the master theorem!

Understanding the S.M.T.

As shown in the diagram on the next page, at the kth level of iteration, there are a^k subdivisions of $(n/b^k)^i$ work. Therefore, at the bottom-most level there are $a^{\log_b n}$ subdivisions of $(n/b^{\log_b n})^i = 1$ work. Recall that by the properties of logarithms, $a^{\log_b n} = n^{\log_b a}$ — so we can switch the base and the contents of the log.



Depicted above is a tree-representation of the work performed at each level of iteration in the recurrence.

We can therefore write the total amount of work represented by the recurrence $T(n)$ as,

$$\text{Total Work: } \Theta(n^{\log_b a}) + \sum_{k=0}^{\log_b n-1} a^k \cdot (\frac{n}{b^k})^i$$

So what does this mean in the three cases shown in the simplified master theorem?

In the case where $a > b^i$, the work done at the leaves *heavily* outgrows that done at the root. That is, $n^{\log_b a}$ is the dominating term in the sum and the total work is therefore $\Theta(n^{\log_b a})$.

In the case where $a = b^i$, the work done at each level is the same and the total work is just the height of the tree multiplied by the work at each level: $\Theta(n^i \log_b n)$.

In the case where $a < b^i$, then the work done at each subsequent level decreases with respect to the root, and the work done at the root dominates: $\Theta(n^i)$.

Tada! By drawing the recurrence tree and summing the total work performed at each level, we were able to find general expressions for the recurrence solutions for each case of the S.M.T.

Problems