Algorithm Efficiency, Big O Notation, and Role of Data Structures/ADTs

- Algorithm Efficiency
- Big O Notation
- Role of Data Structures
- Abstract Data Types
- Reading: L&C 1.2, 2.1-2.4
Algorithm Efficiency

• Let’s look at the following algorithm for initializing the values in an array:

```java
final int N = 500;
int [] counts = new int[N];
for (int i=0; i<counts.length; i++)
    counts[i] = 0;
```

• The length of time the algorithm takes to execute depends on the value of N
Algorithm Efficiency

• In that algorithm, we have one loop that processes all of the elements in the array
• Intuitively:
  – If N was half of its value, we would expect the algorithm to take half the time
  – If N was twice its value, we would expect the algorithm to take twice the time
• That is true and we say that the algorithm efficiency relative to N is linear
Algorithm Efficiency

• Let’s look at another algorithm for initializing the values in a different array:

```java
final int N = 500;
int [][] counts = new int[N][N];
for (int i=0; i<counts.length; i++)
    for (int j=0; j<counts[i].length; j++)
        counts[i][j] = 0;
```

• The length of time the algorithm takes to execute still depends on the value of N
Algorithm Efficiency

• However, in the second algorithm, we have two nested loops to process the elements in the two dimensional array

• Intuitively:
  – If N is half its value, we would expect the algorithm to take one quarter the time
  – If N is twice its value, we would expect the algorithm to take quadruple the time

• That is true and we say that the algorithm efficiency relative to N is quadratic
Big-O Notation

- We use a shorthand mathematical notation to describe the efficiency of an algorithm relative to any parameter n as its “Order” or Big-O
  - We can say that the first algorithm is $O(n)$
  - We can say that the second algorithm is $O(n^2)$
- For any algorithm that has a function $g(n)$ of the parameter n that describes its length of time to execute, we can say the algorithm is $O(g(n))$
- We only include the fastest growing term and ignore any multiplying by or adding of constants
Eight Growth Functions

• Eight functions $O(n)$ that occur frequently in the analysis of algorithms (in order of increasing rate of growth relative to $n$):
  – Constant $\approx 1$
  – Logarithmic $\approx \log n$
  – Linear $\approx n$
  – Log Linear $\approx n \log n$
  – Quadratic $\approx n^2$
  – Cubic $\approx n^3$
  – Exponential $\approx 2^n$
  – Exhaustive Search $\approx n!$
## Growth Rates Compared

<table>
<thead>
<tr>
<th></th>
<th>n=1</th>
<th>n=2</th>
<th>n=4</th>
<th>n=8</th>
<th>n=16</th>
<th>n=32</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(\log n)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>(n)</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>(n\log n)</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>160</td>
</tr>
<tr>
<td>(n^2)</td>
<td>1</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>1024</td>
</tr>
<tr>
<td>(n^3)</td>
<td>1</td>
<td>8</td>
<td>64</td>
<td>512</td>
<td>4096</td>
<td>32768</td>
</tr>
<tr>
<td>(2^n)</td>
<td>2</td>
<td>4</td>
<td>16</td>
<td>256</td>
<td>65536</td>
<td>4294967296</td>
</tr>
<tr>
<td>(n!)</td>
<td>1</td>
<td>2</td>
<td>24</td>
<td>40320</td>
<td>20.9T</td>
<td>Don’t ask!</td>
</tr>
</tbody>
</table>
Travelling Salesman Problem Joke

Brute-Force Solution: $O(n!)$

Dynamic Programming Algorithms: $O(n^2 2^n)$

Selling on eBay: $O(1)$

Still working on your route?

Shut the hell up.
Big-O for a Problem

- $O(g(n))$ for a *problem* means there is some $O(g(n))$ algorithm that solves the problem
- Don’t assume that the specific algorithm that you are currently using is the best solution for the problem
- There may be other correct algorithms that grow at a smaller rate with increasing $n$
- Many times, the goal is to find an algorithm with the smallest possible growth rate
Role of Data Structures

- That brings up the topic of the structure of the data on which the algorithm operates.
- If we are using an algorithm manually on some amount of data, we intuitively try to organize the data in a way that minimizes the number of steps that we need to take.
- Publishers offer dictionaries with the words listed in alphabetical order to minimize the length of time it takes us to look up a word.
Role of Data Structures

• We can do the same thing for algorithms in our computer programs
• Example: Finding a numeric value in a list
• If we assume that the list is unordered, we must search from the beginning to the end
• On average, we will search half the list
• Worst case, we will search the entire list
• Algorithm is $O(n)$, where $n$ is size of array
Role of Data Structures

• Find a match with value in an unordered list

```java
int [] list = {7, 2, 9, 5, 6, 4};

for (int i=0; i<list.length, i++)
    if (value == list[i])
        statement; // found it
// didn’t find it
```
Role of Data Structures

• If we assume that the list is ordered, we can still search the entire list from the beginning to the end to determine if we have a match
• But, we do not need to search that way
• Because the values are in numerical order, we can use a binary search algorithm
• Like the old parlor game “Twenty Questions”
• Algorithm is $O(\log_2 n)$, where $n$ is size of array
Role of Data Structures

• Find a match with value in an ordered list
  
  ```java
  int [] list = {2, 4, 5, 6, 7, 9};
  int min = 0, max = list.length-1;
  while (min <= max) {
    if (value == list[(min+max)/2])
      statement;  // found it
    else
      if (value < list[(min+max)/2])
        max = (min+max)/2 - 1;
      else
        min = (min+max)/2 + 1;
  }
  statement;  // didn’t find it
  ```
Role of Data Structures

• The difference in the structure of the data between an unordered list and an ordered list can be used to reduce algorithm Big-O
• This is the role of data structures and why we study them
• We need to be as clever in organizing our data efficiently as we are in figuring out an algorithm for processing it efficiently
Role of Data Structures

• It may take more time to complete one iteration of the second loop (due to more code in the body of the loop), but the growth rate of time taken with increasing size of the array is less

• As n gets large, the growth rate eventually dominates the resulting time taken

• That is why we ignore multiplication by or addition of constants in Big-O notation
Abstract Data Types (ADT’s)

• A data type is a set of values and operations that can be performed on those values
• The Java primitive data types (e.g. int) have values and operations defined in Java itself
• An Abstract Data Type (ADT) is a data type that has values and operations that are not defined in the language itself
• In Java, an ADT is implemented using a class or an interface
Abstract Data Types (ADT’s)

• The code for Arrays.sort is designed to sort an array of Comparable objects:
  public static void sort (Comparable [ ] data)
• The Comparable interface defines an ADT
• There are no objects of Comparable “class”
• There are objects of classes that implement the Comparable interface (like the class in CS110 project 3 last semester)
• Arrays.sort only uses methods defined in the Comparable interface, i.e. compareTo()
ADT’s and Data Structures

• An Abstract Data Type is a programming construct used to implement a data structure
  – It is a class with methods for organizing and accessing the data that the ADT encapsulates
  – The type of data structure should be hidden by the API (the methods) of the ADT