CSC 2014 Java Bootcamp

Lecture 8
Algorithms & Big O Analysis

ANALYZING ALGORITHMS

Algorithms

What is an algorithm?

An algorithm is a finite set of precise instructions for performing a computation or for solving a problem.

Properties of algorithms:

- **Input** from a specified set,
- **Output** from a specified set (solution),
- **Definiteness** of every step in the computation,
- **Correctness** of output for every possible input,
- **Finiteness** of the number of calculation steps,
- **Effectiveness** of each calculation step and
- **Generality** for a class of problems.

Algorithm Examples

Example: an algorithm that finds the maximum element in a finite sequence

```
procedure max(a_1, a_2, ..., a_n: integers)
max := a_1
for i := 2 to n
  if max < a_i then max := a_i
{max is the largest element}
```

Running Time Analysis

- Reasoning about an algorithm's speed
- "Does it work fast enough for my needs?"
- "How much longer when the input gets larger?"
- "Which algorithm is fastest?"
Elapsed Time vs. No. of Operations

Q: Why not just use a stopwatch?
A: Elapsed time depends on independent factors

Number of operations carried out is the same for two runs of the same code with the same arguments -- no matter what the environment might be.

Analyzing Programs

- Count operations, not time
  - operations is “small step”
  - e.g., a single program statement; an arithmetic operation; assignment to a variable; etc.
- No. of operations depends on the input
  - the larger the problem to solve, the larger the number of operations

Complexity

- In examining algorithm efficiency we must understand the idea of complexity
  - Space Complexity
  - Time Complexity

Space Complexity

- When memory was expensive we focused on making programs as space efficient as possible and developed schemes to make memory appear larger than it really was (virtual memory and memory paging schemes)
- Space complexity is still important in the field of embedded computing (hand held devices like smart phones, tablets, etc.)

Time Complexity

- Is the algorithm “fast enough” for my needs
- How much longer will the algorithm take if I increase the amount of data it must process
- Given a set of algorithms that accomplish the same thing, which is the right one to choose
Algorithm Efficiency

- A measure of the amount of resources consumed in solving a problem of size n
  - Time
  - Space

- Benchmarking: implement algorithm,
  - Run with some specific input and measure time taken
  - Better for comparing performance of processors than for comparing performance of algorithms

- Big Oh (asymptotic analysis)
  - Associates n, the problem size, with t, the processing time required to solve the problem

Cases to examine

- Best case
  - If the algorithm is executed, the fewest number of instructions are executed

- Average case
  - Executing the algorithm produces path lengths that will on average be the same

- Worst case
  - Executing the algorithm produces path lengths that are always a maximum

Big-O Notation

- The magnitude of the number of operations
- Less precise than the exact number
- More useful for comparing two algorithms as input grows larger
- Rough idea: “Term in the formula which grows most quickly”

**BIG O NOTATION**

**Big-O Notation**

- Quadratic Time
  - Largest term no more than
  - “Big-O of n-squared”
  - Doubling the input increases the number of operations approximately 4 times or less
  - E.g.:
    - Method 2(100) = 10,200
    - Method 2(200) = 40,400

\[ O(n^2) \quad c \cdot n^2 \]

- Linear Time
  - Largest term no more than
  - “Big-O of n”
  - Doubling the input increases the number of operations approximately 2 times or less
  - E.g.:
    - Method 1(100) = 300
    - Method 1(200) = 600

\[ O(n) \quad c \cdot n \]
**Big-O Notation**

- **Logarithmic Time**
  - largest term no more than
  - “big-O of log n”
  - doubling the input increases the running time by a fixed number of operations
  - e.g.: Method 3(100) = 3
  - Method 3(1000) = 4

\[ O(\log n) \quad c \cdot \log(n) \]

---

**Common Big Os**

- constant \( O(1) \)
- logarithmic \( O(\log_2 n) \) or \( O(\log n) \)
- linear \( O(n) \)
- \( n \log n \) \( O(n \log n) \) or \( O(n \log n) \)
- quadratic \( O(n^2) \)
- cubic \( O(n^3) \)
- exponential \( O(2^n) \)
- factorial \( O(n!) \)

---

**Growth Rates Compared**

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>( T(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^n )</td>
<td>2 8 16 512 2048</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>1 4 16 64 256</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>1 8 64 512 4096</td>
</tr>
<tr>
<td>( n \log_2 n )</td>
<td>0 2 8 24 64</td>
</tr>
<tr>
<td>( n )</td>
<td>1 2 4 8 16</td>
</tr>
<tr>
<td>( \log n )</td>
<td>0 1 2 3 4</td>
</tr>
<tr>
<td>( n = 1 )</td>
<td>1 1 1 1 1</td>
</tr>
<tr>
<td>( n = 2 )</td>
<td>1 1 1 1 1</td>
</tr>
<tr>
<td>( n = 4 )</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>( n = 8 )</td>
<td>1 1</td>
</tr>
<tr>
<td>( n = 16 )</td>
<td>1</td>
</tr>
<tr>
<td>( n = 32 )</td>
<td>1</td>
</tr>
</tbody>
</table>

---

**Comparing Growth Rates**

---

**EXAMPLES**

Add two numbers

```java
int a, b, c;

c = a + b;
System.out.println("Sum is " + c);
```
**Find a specific value (search)**

```java
int n; // number of values
int[] values = new int[n]; // random values
int find; // value to find
for (int i = 0; i < n; i++)
    if (values[i] == find)
        System.out.println("Found at "+ i);
```

**Find the maximum**

```java
int n; // number of values
int[] values = new int[n]; // random values
int max = values[0];
for (int i = 1; i < n; i++)
    if (values[i] > max)
        max = values[i];
System.out.println("Max value is: "+ max);
```

**Put in order (selection sort)**

```java
int min, temp;
for (int i = 0; i < list.length - 1; i++)
    min = i;
    for (int j = i + 1; j < list.length; j++)
        if (list[j] < list[min])
            min = j;
    temp = list[i]; // swap
    list[i] = list[min]; // i and min
    list[min] = temp; // values
```

**Find a sorted value (binary search)**

```java
int low = 0;
int high = values.length - 1;
int mid;
while (high >= low)
    {
        mid = (low + high) / 2;
        if (target < values[mid])
            high = mid - 1;
        else if (target == values[mid])
            System.out.println("Found at "+ mid);
        else
            low = mid + 1;
    }
```

**Examples of algorithm complexity**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>Arithmetic, Comparison, Variable declaration, Assignment, Invoking a method or function</td>
</tr>
<tr>
<td>log N</td>
<td>Binary search, Insert, delete into heap or BST</td>
</tr>
<tr>
<td>Linear</td>
<td>Iterate over N elements, Concatenate two strings</td>
</tr>
<tr>
<td>N log N</td>
<td>Quicksort, Mergesort, FFT</td>
</tr>
<tr>
<td>N²</td>
<td>All pairs of N elements, Allocate N-by-N array</td>
</tr>
<tr>
<td>N³</td>
<td>All triples of N elements, N-by-N matrix multiplication</td>
</tr>
<tr>
<td>2ⁿ</td>
<td>Number of N-bit integers, All subsets of N elements, Discs moved in Towers of Hanoi</td>
</tr>
<tr>
<td>N!</td>
<td>All permutations of N elements</td>
</tr>
</tbody>
</table>
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.

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.

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

- 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

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:
  ```java
  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

Interface
(Methods and Constants)

- Class that uses an ADT
- Class that implements an ADT
- Data Structure