Topic Number 2

Efficiency – Complexity

Algorithm Analysis

"bit twiddling: 1. (pejorative) An exercise in tuning (see tune) in which incredible amounts of time and effort go to produce little noticeable improvement, often with the result that the code becomes incomprehensible."
- The Hackers Dictionary, version 4.4.7

Clicker Question 1

"My program finds all the primes between 2 and 1,000,000,000 in 1.37 seconds."
- how good is this solution?

A. Good
B. Bad
C. It depends

Efficiency

- Computer Scientists don’t just write programs.
- They also analyze them.
- How efficient is a program?
  - How much time does it take program to complete?
  - How much memory does a program use?
  - How do these change as the amount of data changes?

Technique

- Informal approach for this class
  - more formal techniques in theory classes
- Many simplifications
  - view algorithms as Java programs
  - count executable statements in program or method
  - find number of statements as function of the amount of data
  - focus on the dominant term in the function
Counting Statements

```java
int x; // one statement
x = 12; // one statement
int y = z * x + 3 % 5 * x / i; // 1
x++; // one statement
boolean p = x < y && y % 2 == 0 ||
    z >= y * x; // 1
int[] list = new int[100]; // 100
list[0] = x * x + y * y; // 1
```

Clicker Question 2

- What is output by the following code?
  ```java
  int total = 0;
  for(int i = 0; i < 13; i++)
      for(int j = 0; j < 11; j++)
          total += 2;
  System.out.println( total );
  ```

  A. 24  
  B. 120  
  C. 143  
  D. 286  
  E. 338

Clicker Question 3

- What is output when method `sample` is called?
  ```java
  public static void sample(int n, int m) {
      int total = 0;
      for(int i = 0; i < n; i++)
          for(int j = 0; j < m; j++)
              total += 5;
      System.out.println( total );
  }
  ```

  A. 5  
  B. n * m  
  C. n * m * 5  
  D. \( n^m \)  
  E. \( (n \times m)^5 \)

Example

```java
public int total(int[] values) {
    int result = 0;
    for(int i = 0; i < values.length; i++)
        result += values[i];
    return result;
}
```

- How many statements are executed by method `total` as a function of `values.length`?

- Let \( N = \text{values.length} \)
  - \( N \) is commonly used as a variable that denotes the amount of data
Counting Up Statements

- int result = 0;  
- int i = 0;  
- i < values.length; N + 1  
- i++ N  
- result += values[i]; N  
- return total;  
- T(N) = 3N + 4  
- T(N) is the number of executable statements in method total as function of values.length  

Another Simplification

- When determining complexity of an algorithm we want to simplify things  
  - hide some details to make comparisons easier  
- Like assigning your grade for course  
  - At the end of CS314 your transcript won’t list all the details of your performance in the course  
  - it won’t list scores on all assignments, quizzes, and tests  
  - simply a letter grade, B- or A or D+  
- So we focus on the dominant term from the function and ignore the coefficient

Big O

- The most common method and notation for discussing the execution time of algorithms is Big O, also spoken Order  
- Big O is the asymptotic execution time of the algorithm  
- Big O is an upper bounds  
- It is a mathematical tool  
- Hide a lot of unimportant details by assigning a simple grade (function) to algorithms

Formal Definition of Big O

- T(N) is $O(F(N))$ if there are positive constants $c$ and $N_0$ such that $T(N) \leq cF(N)$ when $N \geq N_0$  
  - $N$ is the size of the data set the algorithm works on  
  - $T(N)$ is a function that characterizes the actual running time of the algorithm  
  - $F(N)$ is a function that characterizes an upper bounds on $T(N)$. It is a limit on the running time of the algorithm. (The typical Big functions table)  
  - $c$ and $N_0$ are constants
What it Means

- T(N) is the actual growth rate of the algorithm
  - can be equated to the number of executable statements in a program or chunk of code
- F(N) is the function that bounds the growth rate
  - may be upper or lower bound
- T(N) may not necessarily equal F(N)
  - constants and lesser terms ignored because it is a bounding function

Showing O(N) is Correct

- Recall the formal definition of Big O
  - T(N) is O(F(N)) if there are positive constants c and N_0 such that T(N) ≤ cF(N) when N > N_0
- Recall method total, T(N) = 3N + 4
  - show method total is O(N).
  - F(N) is N
- We need to choose constants c and N_0
- how about c = 4, N_0 = 5?

Typical Big O Functions – "Grades"

<table>
<thead>
<tr>
<th>Function</th>
<th>Common Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>N!</td>
<td>factorial</td>
</tr>
<tr>
<td>2^N</td>
<td>Exponential</td>
</tr>
<tr>
<td>N^d, d &gt; 3</td>
<td>Polynomial</td>
</tr>
<tr>
<td>N^3</td>
<td>Cubic</td>
</tr>
<tr>
<td>N^2</td>
<td>Quadratic</td>
</tr>
<tr>
<td>N√N</td>
<td>N Square root N</td>
</tr>
<tr>
<td>N log N</td>
<td>N log N</td>
</tr>
<tr>
<td>N</td>
<td>Linear</td>
</tr>
<tr>
<td>√N</td>
<td>Root - n</td>
</tr>
<tr>
<td>log N</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>1</td>
<td>Constant</td>
</tr>
</tbody>
</table>
Clicker Question 4

Which of the following is true?
A. Method total is O(N)
B. Method total is O(N^2)
C. Method total is O(N!)
D. Method total is O(N^N)
E. All of the above are true

Just Count Loops, Right?

// assume mat is a 2d array of booleans
// assume mat is square with N rows,
// and N columns

int numThings = 0;
for(int r = row - 1; r <= row + 1; r++)
    for(int c = col - 1; c <= col + 1; c++)
        if( mat[r][c] )
            numThings++;

What is the order of the above code?
A. O(1)  B. O(N)  C. O(N^2)  D. O(N^3)  E. O(N^{1/2})

It is Not Just Counting Loops

// Second example from previous slide could be
// rewritten as follows:

int numThings = 0;
if( mat[r-1][c-1] ) numThings++;
if( mat[r-1][c] ) numThings++;
if( mat[r-1][c+1] ) numThings++;
if( mat[r][c-1] ) numThings++;
if( mat[r][c] ) numThings++;
if( mat[r][c+1] ) numThings++;
if( mat[r+1][c-1] ) numThings++;
if( mat[r+1][c] ) numThings++;
if( mat[r+1][c+1] ) numThings++;

Sidetrack, the logarithm

Thanks to Dr. Math

3^2 = 9

likewise log_3 9 = 2
  "The log to the base 3 of 9 is 2."

The way to think about log is:
  "the log to the base x of y is the number you can
   raise x to to get y."
  Say to yourself "The log is the exponent." (and say
  it over and over until you believe it.)
  In CS we work with base 2 logs, a lot

log_2 32 = ?  log_2 8 = ?  log_2 1024 = ?  log_{10} 1000 = ?
When Do Logarithms Occur

- Algorithms have a logarithmic term when they use a divide and conquer technique
- the data set keeps getting divided by 2

```java
public int foo(int n) {
    // pre n > 0
    int total = 0;
    while(n > 0) {
        n = n / 2;
        total++;
    }
    return total;
}
```

What is the order of the above code?

A. O(1)  
B. O(log N)  
C. O(N)  
D. O(N log N)  
E. O(N^2)

Dealing with other methods

- What do I do about method calls?
  ```java
double sum = 0.0;
for(int i = 0; i < n; i++)
    sum += Math.sqrt(i);
  ```

  - Long way
    - go to that method or constructor and count statements
  
  - Short way
    - substitute the simplified Big O function for that method.
    - if Math.sqrt is constant time, O(1), simply count
      ```java
      sum += Math.sqrt(i);
      ```
      as one statement.

Dealing With Other Methods

```java
public int foo(int[] list) {
    int total = 0;
    for(int i = 0; i < list.length; i++)
        total += countDups(list[i], list);
    return total;
}
```

What is the Big O of `foo`?

A. O(1)  
B. O(N)  
C. O(N log N)  
D. O(N^2)  
E. O(N!)

Independent Loops

```java
// from the Matrix class
public void scale(int factor) {
    for(int r = 0; r < numRows(); r++)
        for(int c = 0; c < numCols(); c++)
            iCells[r][c] *= factor;
}
```

Assume an `numRows()` = N and `numCols()` = N.
In other words, a square Matrix.
`numRows` and `numCols` are O(1)

What is the T(N)? What is the Big O?

A. O(1)  
B. O(N)  
C. O(N log N)  
D. O(N^2)  
E. O(N!)
Significant Improvement – Algorithm with Smaller Big O function

- Problem: Given an array of ints replace any element equal to 0 with the maximum positive value to the right of that element. (if no positive value to the right, leave unchanged.)

Given:
[0, 9, 0, 13, 0, 0, 7, 1, -1, 0, 1, 0]

Becomes:
[13, 9, 13, 13, 7, 7, 7, 1, -1, 1, 1, 0]

Replace Zeros – Typical Solution

```java
public void replace0s(int[] data){
    for(int i = 0; i < data.length - 1; i++)
        if( data[i] == 0 ) {
            int max = 0;
            for(int j = i+1; j<data.length; j++)
                max = Math.max(max, data[j]);
            data[i] = max;
        }
}
```

Assume all values are zeros. (worst case)

Example of a dependent loops.

Replace Zeros – Alternate Solution

```java
public void replace0s(int[] data){
    int max =
        Math.max(0, data[data.length - 1]);
    int start = data.length - 2;
    for(int i = start; i >= 0; i--){
        if( data[i] == 0 )
            data[i] = max;
        else
            max = Math.max(max, data[i]);
    }
}
```

Big O of this approach?
A. O(1)   B. O(N)   C. O(NlogN)
D. O(N^2)   E. O(N!)

A Useful Proportion

- Since F(N) is characterizes the running time of an algorithm the following proportion should hold true:
F(N_0) / F(N_1) ~ = time_0 / time_1

- An algorithm that is O(N^2) takes 3 seconds to run given 10,000 pieces of data.
  - How long do you expect it to take when there are 30,000 pieces of data?
  - common mistake
  - logarithms?
Why Use Big O?

- As we build data structures Big O is the tool we will use to decide under what conditions one data structure is better than another
- Think about performance when there is a lot of data.
  - "It worked so well with small data sets..."
  - Joel Spolsky, Schlemiel the painter's Algorithm
- Lots of trade offs
  - some data structures good for certain types of problems, bad for other types
  - often able to trade SPACE for TIME.
  - Faster solution that uses more space
  - Slower solution that uses less space

Big O Space

- Big O could be used to specify how much space is needed for a particular algorithm
  - in other words how many variables are needed
- Often there is a time – space tradeoff
  - can often take less time if willing to use more memory
  - can often use less memory if willing to take longer
  - truly beautiful solutions take less time and space

The biggest difference between time and space is that you can't reuse time. - Merrick Furst

Quantifiers on Big O

- It is often useful to discuss different cases for an algorithm
- Best Case: what is the best we can hope for?
  - least interesting
- Average Case (a.k.a. expected running time): what usually happens with the algorithm?
- Worst Case: what is the worst we can expect of the algorithm?
  - very interesting to compare this to the average case

Best, Average, Worst Case

- To Determine the best, average, and worst case Big O we must make assumptions about the data set
- Best case -> what are the properties of the data set that will lead to the fewest number of executable statements (steps in the algorithm)
- Worst case -> what are the properties of the data set that will lead to the largest number of executable statements
- Average case -> Usually this means assuming the data is randomly distributed
  - or if I ran the algorithm a large number of times with different sets of data what would the average amount of work be for those runs?
Another Example

```java
public double minimum(double[] values)
{
    int n = values.length;
    double minValue = values[0];
    for(int i = 1; i < n; i++)
        if(values[i] < minValue)
            minValue = values[i];
    return minValue;
}
```

- T(N)? F(N)? Big O? Best case? Worst Case? Average Case?
- If no other information, assume asking average case

Example of Dominance

- Look at an extreme example. Assume the actual number as a function of the amount of data is:
  \[ \frac{N^2}{10000} + 2N\log_{10} N + 100000 \]
- Is it plausible to say the \( N^2 \) term dominates even though it is divided by 10000 and that the algorithm is \( O(N^2) \)?
- What if we separate the equation into \( \left( \frac{N^2}{10000} \right) \) and \( (2N \log_{10} N + 100000) \) and graph the results.

Summing Execution Times

- For large values of \( N \) the \( N^2 \) term dominates so the algorithm is \( O(N^2) \)
- When does it make sense to use a computer?

Comparing Grades

- Assume we have a problem
- Algorithm A solves the problem correctly and is \( O(N^2) \)
- Algorithm B solves the same problem correctly and is \( O(N \log_2 N) \)
- Which algorithm is faster?
- One of the assumptions of Big O is that the data set is large.
- The "grades" should be accurate tools if this is true
Running Times

- Assume N = 100,000 and processor speed is 1,000,000,000 operations per second

<table>
<thead>
<tr>
<th>Function</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^N$</td>
<td>$3.2 \times 10^{30086}$ years</td>
</tr>
<tr>
<td>$N^4$</td>
<td>3171 years</td>
</tr>
<tr>
<td>$N^3$</td>
<td>11.6 days</td>
</tr>
<tr>
<td>$N^2$</td>
<td>10 seconds</td>
</tr>
<tr>
<td>$N \sqrt{N}$</td>
<td>0.032 seconds</td>
</tr>
<tr>
<td>$N \log N$</td>
<td>0.0017 seconds</td>
</tr>
<tr>
<td>$\sqrt{N}$</td>
<td>$3.2 \times 10^2$ seconds</td>
</tr>
<tr>
<td>$\log N$</td>
<td>$1.2 \times 10^4$ seconds</td>
</tr>
</tbody>
</table>

Theory to Practice OR

Dykstra says: "Pictures are for the Weak."

<table>
<thead>
<tr>
<th></th>
<th>1000</th>
<th>2000</th>
<th>4000</th>
<th>8000</th>
<th>16000</th>
<th>32000</th>
<th>64000</th>
<th>128K</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(N)</td>
<td>$2.2\times 10^6$</td>
<td>$2.7\times 10^6$</td>
<td>$5.4\times 10^5$</td>
<td>$4.2\times 10^5$</td>
<td>$6.8\times 10^4$</td>
<td>$1.2\times 10^4$</td>
<td>$2.3\times 10^4$</td>
<td>$5.1\times 10^4$</td>
</tr>
<tr>
<td>O(NlogN)</td>
<td>$8.5\times 10^5$</td>
<td>$1.9\times 10^5$</td>
<td>$3.7\times 10^4$</td>
<td>$4.7\times 10^4$</td>
<td>$1.0\times 10^3$</td>
<td>$2.1\times 10^3$</td>
<td>$4.6\times 10^3$</td>
<td>$1.2\times 10^2$</td>
</tr>
<tr>
<td>O($N^{3/2}$)</td>
<td>$3.5\times 10^6$</td>
<td>$6.9\times 10^4$</td>
<td>$1.7\times 10^3$</td>
<td>$5.0\times 10^3$</td>
<td>$1.4\times 10^2$</td>
<td>$3.8\times 10^2$</td>
<td>$0.11$</td>
<td>$0.30$</td>
</tr>
<tr>
<td>O($N^2$) ind.</td>
<td>$3.4\times 10^3$</td>
<td>$1.4\times 10^3$</td>
<td>$4.4\times 10^3$</td>
<td>$0.22$</td>
<td>$0.86$</td>
<td>$3.45$</td>
<td>$13.79$</td>
<td>(55)</td>
</tr>
<tr>
<td>O($N^2$) dep.</td>
<td>$1.8\times 10^3$</td>
<td>$7.1\times 10^2$</td>
<td>$2.7\times 10^2$</td>
<td>$0.11$</td>
<td>$0.43$</td>
<td>$1.73$</td>
<td>$6.90$</td>
<td>(27.6)</td>
</tr>
<tr>
<td>O($N^3$)</td>
<td>3.40</td>
<td>27.26</td>
<td>(218)</td>
<td>(1745) min.</td>
<td>(13,957) min</td>
<td>(112k) 31 hrs</td>
<td>(896k) 10 days</td>
<td>(7.2m) 80 days</td>
</tr>
</tbody>
</table>

Change between Data Points

<table>
<thead>
<tr>
<th></th>
<th>1000</th>
<th>2000</th>
<th>4000</th>
<th>8000</th>
<th>16000</th>
<th>32000</th>
<th>64000</th>
<th>128K</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(N)</td>
<td>-</td>
<td>1.21</td>
<td>2.02</td>
<td>0.78</td>
<td>1.62</td>
<td>1.76</td>
<td>1.89</td>
<td>2.24</td>
</tr>
<tr>
<td>O(NlogN)</td>
<td>-</td>
<td>2.18</td>
<td>1.99</td>
<td>1.27</td>
<td>2.13</td>
<td>2.15</td>
<td>2.15</td>
<td>2.71</td>
</tr>
<tr>
<td>O($N^{3/2}$)</td>
<td>-</td>
<td>1.98</td>
<td>2.48</td>
<td>2.87</td>
<td>2.79</td>
<td>2.76</td>
<td>2.85</td>
<td>2.79</td>
</tr>
<tr>
<td>O($N^2$) ind</td>
<td>-</td>
<td>4.06</td>
<td>3.98</td>
<td>3.94</td>
<td>3.99</td>
<td>4.00</td>
<td>3.99</td>
<td>-</td>
</tr>
<tr>
<td>O($N^2$) dep</td>
<td>-</td>
<td>4.00</td>
<td>3.82</td>
<td>3.97</td>
<td>4.00</td>
<td>4.01</td>
<td>3.98</td>
<td>-</td>
</tr>
<tr>
<td>O($N^3$)</td>
<td>-</td>
<td>8.03</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Value obtained by $T_{nx} / T_{nx-1}$

Okay, Pictures

Results on a 2GHz laptop

<table>
<thead>
<tr>
<th>Value of N</th>
<th>0</th>
<th>5000</th>
<th>10000</th>
<th>15000</th>
<th>20000</th>
<th>25000</th>
<th>30000</th>
<th>35000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>0.0</td>
<td>1.0</td>
<td>2.0</td>
<td>3.0</td>
<td>4.0</td>
<td>5.0</td>
<td>6.0</td>
<td>7.0</td>
</tr>
</tbody>
</table>
Put a Cap on Time

No O(N^2) Data

Just O(N) and O(NlogN)

Just O(N)
10^9 instructions/sec, runtimes

<table>
<thead>
<tr>
<th>N</th>
<th>O(log N)</th>
<th>O(N)</th>
<th>O(N log N)</th>
<th>O(N^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.0000000003</td>
<td>0.00000001</td>
<td>0.000000033</td>
<td>0.000001</td>
</tr>
<tr>
<td>100</td>
<td>0.000000007</td>
<td>0.00000010</td>
<td>0.000000664</td>
<td>0.001000</td>
</tr>
<tr>
<td>1,000</td>
<td>0.000000010</td>
<td>0.00000100</td>
<td>0.00010000</td>
<td>0.001</td>
</tr>
<tr>
<td>10,000</td>
<td>0.000000013</td>
<td>0.0001000</td>
<td>0.00132900</td>
<td>0.1 min</td>
</tr>
<tr>
<td>100,000</td>
<td>0.000000017</td>
<td>0.0010000</td>
<td>0.001661000</td>
<td>10 seconds</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.000000020</td>
<td>0.001</td>
<td>0.0199</td>
<td>16.7 minutes</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>0.000000030</td>
<td>1.0 second</td>
<td>30 seconds</td>
<td>31.7 years</td>
</tr>
</tbody>
</table>

Formal Definition of Big O (repeated)

- T(N) is O( F(N) ) if there are positive constants c and N_0 such that T(N) ≤ cF(N) when N ≥ N_0
  - N is the size of the data set the algorithm works on
  - T(N) is a function that characterizes the *actual* running time of the algorithm
  - F(N) is a function that characterizes an upper bounds on T(N). It is a limit on the running time of the algorithm
  - c and N_0 are constants

More on the Formal Definition

- There is a point N_0 such that for all values of N that are past this point, T(N) is bounded by some multiple of F(N)
- Thus if T(N) of the algorithm is O( N^2 ) then, ignoring constants, at some point we can *bound* the running time by a quadratic function.
- Given a *linear* algorithm it is *technically correct* to say the running time is O(N^2). O(N) is a more precise answer as to the Big O of the linear algorithm
  - thus the caveat “pick the most restrictive function” in Big O type questions.

What it All Means

- T(N) is the actual growth rate of the algorithm
  - can be equated to the number of executable statements in a program or chunk of code
- F(N) is the function that bounds the growth rate
  - may be upper or lower bound
- T(N) may not necessarily equal F(N)
  - constants and lesser terms ignored because it is a *bounding function*
**Other Algorithmic Analysis Tools**

- **Big Omega** \( T(N) \) is \( \Omega( F(N) ) \) if there are positive constants \( c \) and \( N_0 \) such that \( T(N) \geq cF( N ) \) when \( N \geq N_0 \)
  - Big O is similar to less than or equal, an upper bound
  - Big Omega is similar to greater than or equal, a lower bound

- **Big Theta** \( T(N) \) is \( \theta( F(N) ) \) if and only if \( T(N) \) is \( O( F(N) ) \) and \( T( N ) \) is \( \Omega( F(N) ) \).
  - Big Theta is similar to equals

**Relative Rates of Growth**

<table>
<thead>
<tr>
<th>Analysis Type</th>
<th>Mathematical Expression</th>
<th>Relative Rates of Growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big O</td>
<td>( T(N) = O( F(N) ) )</td>
<td>( T(N) \leq F(N) )</td>
</tr>
<tr>
<td>Big ( \Omega )</td>
<td>( T(N) = \Omega( F(N) ) )</td>
<td>( T(N) \geq F(N) )</td>
</tr>
<tr>
<td>Big ( \theta )</td>
<td>( T(N) = \theta( F(N) ) )</td>
<td>( T(N) = F(N) )</td>
</tr>
</tbody>
</table>

*In spite of the additional precision offered by Big Theta, Big O is more commonly used, except by researchers in the algorithms analysis field* - Mark Weiss