

# Efficiency, Complexity, Algorithm Analysis

Instructor: Jeeho Ryoo

# 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?
  - What is the difference between the average case and worst case efficiency if any?

# Technique

- Informal approach for this class
  - more formal techniques in theory classes
- **How many computations will this program (method, algorithm) perform to get the answer?**
- Many simplifications
  - view algorithms as C programs
  - **determine by analysis the total number executable statements (computations) in program or method as a function of the amount of data**
  - focus on the *dominant term* in the function
  - $T(N) = 17.5N^3 + 25N^2 + 35N + 251$  **IS ORDER  $N^3$**

# Counting Statements

```
int total(int[] values) {  
    int result = 0;  
    int n = sizeof(values)/sizeof(int)  
    for (int i = 0; i <n; i++)  
        result += values[i];  
    return result;  
}
```

- ▶ How many statements are executed by method `total` as a function of  $n$
- ▶ Let  $N = n$ 
  - ▶  $N$  is commonly used as a variable that denotes the amount of data

# Counting Statements

```
int result = 0 1
int i = 0 1
int n = sizeof(values)/sizeof(int) 1
i < n N + 1
i++ N
result += values[i] N
return total; 1
```

$$T(N) = 3N + 4$$

$T(N)$  is the number of executable statements in method `total`  
as function of  $n$

# Simplification

- When determining complexity of an algorithm we want to simplify things
  - ignore some details to make comparisons easier
- Like assigning your grade for course
  - At the end of COMP2510 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 % grade
- 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
  - In other words, how does the running time of the algorithm grow as a function of the amount of input data?
- 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

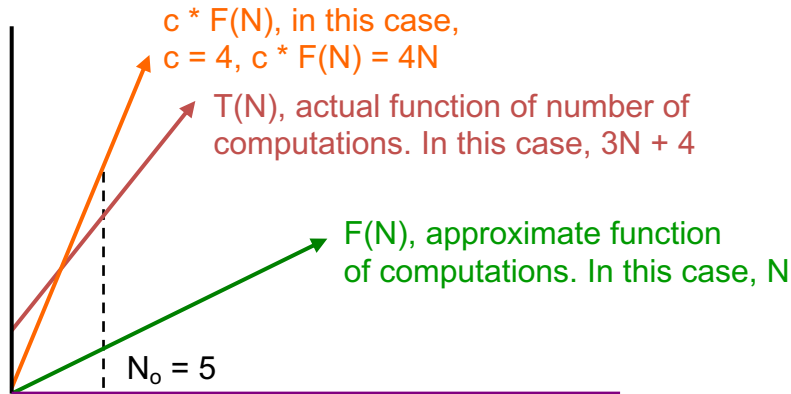
- $T(N)$  is the actual growth rate of the algorithm
  - can be equated to the number of executable statements in a program or chunk of code
- $F(N)$  is the function that bounds the growth rate
  - may be upper or lower bound
- $T(N)$  may not necessarily equal  $F(N)$ 
  - constants and lesser terms 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  $T(N) \leq cF(N)$  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  $c = 4$ ,  $N_0 = 5$  ?

# Graphical Representation

vertical axis: time for algorithm to complete. (simplified to number of executable statements)




horizontal axis:  $N$ , number of elements in data set

# Typical Big O Functions

Function	Common Name
$N!$	factorial
$2^N$	Exponential
$N^d, d > 3$	Polynomial
$N^3$	Cubic
$N^2$	Quadratic
$N * \text{Sqrt}(N)$	N Square root N
$N \log N$	$N \log N$
$N$	Linear
$\text{Sqrt}(N)$	Root - n
$\log N$	Logarithmic
1	Constant

Running time grows 'quickly' with more input.



Running time grows 'slowly' with more input.

## Showing Order More Formally ...

- Show  $10N^2 + 15N$  is  $O(N^2)$
- Break into terms.
- $10N^2 \leq 10N^2$
- $15N \leq 15N^2$  for  $N \geq 1$  (Now add)
- $10N^2 + 15N \leq 10N^2 + 15N^2$  for  $N \geq 1$
- $10N^2 + 15N \leq 25N^2$  for  $N \geq 1$
- $c = 25, N_0 = 1$
- Note, the choices for  $c$  and  $N_0$  are not unique

## Showing Order More Formally

- Show  $10N^2 + 15N$  is  $O(N^2)$
- Break into terms.
- $10N^2 \leq 10N^2$
- $15N \leq 15N^2$  for  $N \geq 1$  (Now add)
- $10N^2 + 15N \leq 10N^2 + 15N^2$  for  $N \geq 1$
- $10N^2 + 15N \leq 25N^2$  for  $N \geq 1$
- $c = 25$ ,  $N_0 = 1$
- Note, the choices for  $c$  and  $N_0$  are not unique

## Dealing With Other Methods

```
int foo(int[] data) {  
    int total = 0;  
    for (int i = 0; i < n; i++)  
        total += countDups(data[i], data);  
    return total;  
}  
// method countDups is O(N) where N is the  
// length of the array it is passed
```

What is the Big O of `foo`?

# Independent Loops

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

`numRows()` returns number of rows in the matrix `iCells`

`numCols()` returns number of columns in the matrix `iCells`

Assume `iCells` is an  $N$  by  $N$  square matrix.

Assume `numRows` and `numCols` are  $O(1)$

What is the Big  $O$ ?



## Just Count Loops, Right?

```
// Assume mat is a 2d array of booleans.  
// Assume mat is square with N rows,  
// and N columns.  
void count(int[][] mat, int row, int col) {  
  
    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 Big O?

## 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 tend to have a logarithmic term the size of the data set keeps getting divided by 2

```
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?

## 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

```
void replace0s(int[] data){
    for(int i = 0; i < n; i++){
        if (data[i] == 0) {
            int max = 0;
            for(int j = i+1; j<n; j++)
                max = max(max, data[j]);
            data[i] = max;
        }
    }
}
```

What is the order of the above code?

## Replace Zeros – Improvement

## Question

- Is  $O(N)$  really that much faster than  $O(N^2)$ ?
  1. never
  2. always
  3. Typically
- Depends on the actual functions and the value of  $N$ .
- $1000N + 250$  compared to  $N^2 + 10$
- When do we use mechanized computation?
- $N = 100,000$
- $100,000,250 < 10,000,000,010$  ( $10^8 < 10^{10}$ )

# 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



# Empirical Analysis

Assumptions:

Laptop:  $10^8$  ops

Supercomputer:  $10^{12}$  ops

	N <sup>2</sup> algorithm			NlogN algorithm		
	thousand	million	billion	thousand	million	billion
Laptop	instant	2.8hr	317yrs	instant	1sec	18sec
Super computers	instant	1sec	1wk	instant	instant	instant

# 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, but a good exercise
  - **Don't assume no data. Amount of data is still variable, possibly quite large**
- 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

```
double minimum(double[] values) {  
    int n = sizeof(values)/sizeof(values[0]);  
    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

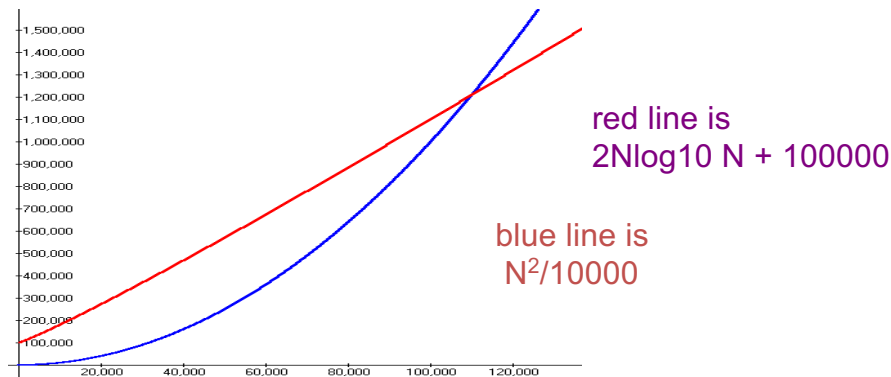
## Example of Dominance

- Look at an extreme example. Assume the actual number as a function of the amount of data is:

$$N^2/10000 + 2N\log_{10} N + 100000$$

- 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  $(N^2/10000)$  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?

# Running Times

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

Function	Running Time
$2^N$	$3.2 \times 10^{30,086}$ years
$N^4$	3171 years
$N^3$	11.6 days
$N^2$	10 seconds
$N\sqrt{N}$	0.032 seconds
$N \log N$	0.0017 seconds
$N$	0.0001 seconds
$\sqrt{N}$	$3.2 \times 10^{-7}$ seconds
$\log N$	$1.2 \times 10^{-8}$ seconds