Mathematical Background and Running Time Analysis

EECS 233

Big-O intuition

$T(N) = \boldsymbol{O}(f(N))$ if there are positive constants c and n_0 such that
$T(N) \leq cf(N)$ for all $N \geq n_0$

```
int sum(float[] array) {
    for (int i = 0; i <= array.length; i++)
        array[i] = array[i]*5;
}
```

```
int sum(float[] array) {
    for (int i = 0; i <= array.length; i++)
        array[i] = array[i]/0.2;
}
```

```
int sum(float[] array) {
    float[] backup = new float[array.length];
    for (int i = 0; i <= array.length; i++){
        for (int j = 0; j <= array.length; j++)
            backup[j] = array[j];

        array[i] = array[i]*5;
        if(Float.isInfinite(array[i]){
            for (int j = 0; j <= array.length; j++)
                array[j] = backup[j];
            print("Numbers too large!");
            break;
        }
    }
}
```

Function Growth Rates: Mathematical Definitions.

Consider positive functions T(N) and f(N).

- $T(N) = \boldsymbol{O}(f(N))$ if there are positive constants c and n_0 such that
  $T(N) \leq cf(N)$ for all $N \geq n_0$
  - Example: $10*N^2+10000 = \boldsymbol{O}(N^3)$

- $T(N) = \boldsymbol{\Omega}(f(N))$ if there are positive constants c and n_0 such that
  $T(N) \geq cf(N)$ for all $N \geq n_0$
  - Example: $0.0001*N^3 = \boldsymbol{\Omega}(N^2)$

- $T(N) = \boldsymbol{\theta}(f(N))$ iff $T(N) = \mathbf{O}(f(N))$ and $T(N) = \boldsymbol{\Omega}(f(N))$
  - Example: $0.001*N^2+10000*N = \boldsymbol{\theta}(N^2)$

- $T(N) = \boldsymbol{o}(f(N))$ if for *all* constants c there exists an n_0 such that
  $T(N) < cf(N)$ for all $N > n_0$.
  or
  $T(N) = \boldsymbol{o}(f(N))$ iff $T(N) = \boldsymbol{O}(f(N))$ and $T(N) \neq \boldsymbol{\Omega}(f(N))$
  - Example: $10*N^2+10000 = \boldsymbol{o}(N^3)$

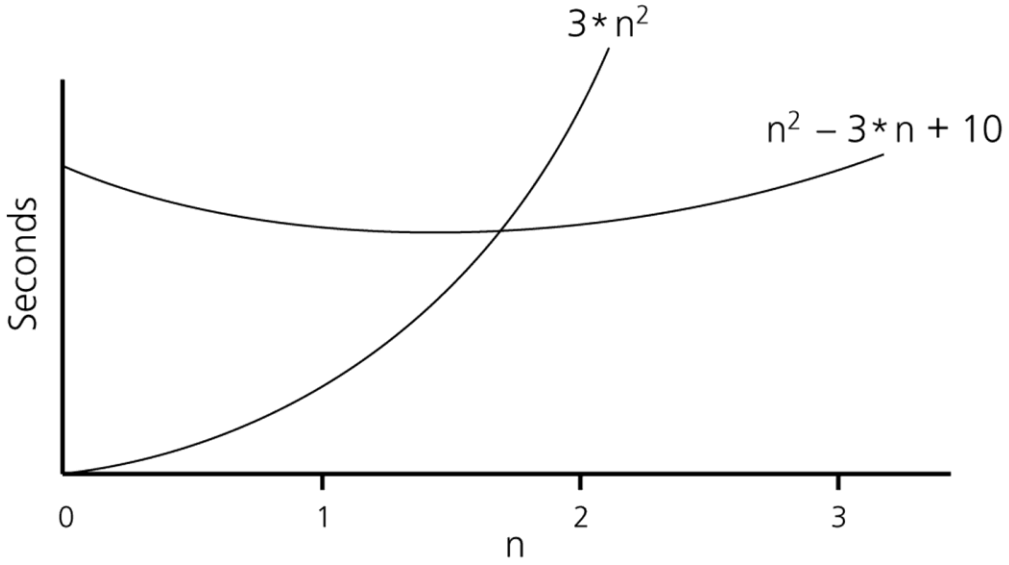
Function Growth Rates: Mathematical Definitions.

- Big-O expression, T(N) = O(f(N)) says that T(N) grows NOT FASTER than f(N)
- Big-omega says that T(N) grows AT LEAST AS SLOW (and maybe faster) as f(N)
- Theta says the two functions grow at the same rate
- Little-o says that T(N) grows strictly slower than f(N)

Example

- Show that $T(n) = n^2 - 3 \cdot n + 10$ is order of $O(n^2)$
  - Show that there exist constants *c* and n_0 that satisfy the condition

Try $c = 3$ and $n_0 = 2$



How to Determine the Relative Growth Rate?

- If $\lim_{N->\infty} T(N) / f(N)$
  - $= 0$: $T(N) = \boldsymbol{o}(f(N))$ (and $O(f(N))$ for sure)
  - $= c$!= 0: $T(N) = \boldsymbol{\Theta}(f(N))$
  - = infinity: $f(N) = \boldsymbol{o}(T(N))$

Big-O Toolbox
(for positive monotonic functions)

- Constants do not matter!
  - If $T(N) = O(f(N) + c))$ then $T(N) = O(f(N))$
  - If $T(N) = O(c*f(N))$ then $T(N) = O(f(N))$
  - If $T(N) + c = O(f(N))$ then $T(N) = O(f(N))$
  - If $T(N)*c = O(f(N)$ then $T(N) = O(f(N))$
- Algebraic properties:
  - If $T1(N) = O(f(N))$ and $T2(N) = O(g(N))$ then $T1(N)+T2(N) = O(f(N) + g(N))$
  - If $T1(N) = O(f(N))$ and $T2(N) = O(g(N))$ then $T1(N)*T2(N) = O(f(N) * g(N))$
  - If $T1(N) = O(f(N))$ and $g(x)$ is monotonic, then $g(T1(N)) = O(g(f(N))$
- Dominated terms do not matter:
  - If $T(N) = O(f(N) + g(N))$ and $g(N) = o(f(N))$ then $T(N) = O(f(N))$

Misconceptions

From http://rob-bell.net/2009/06/a-beginners-guide-to-big

A Beginners' Guide to Big O Notation

Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the **worst-case** scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.

From http://www.perlmonks.org/?node_id=573138

What Is The Big-O

This tutorial covers the Big-O as it relates to computer science. If y
thinking of something else (perhaps Fridays in the Chatterbox), you
reading now. Simply put, it describes how the algorithm scales (*per*
worst case scenario as it is is run with more input.

What The Big-O Is Good For

The good news is that the Big-O belongs to an entire family of notation. This tutorial will not cover it but family members include describing the average and best cases.

From http://www.perlmonks.org/?node_id=227909

An informal introduction to O(N) notation

Quite often, when an algorithm's growth rate is characterized by some mix of orders, the dominant order is shown, and the rest are dropped. $O(N^2)$ might really mean $O(N^2 + N)$.

Logarithm Properties

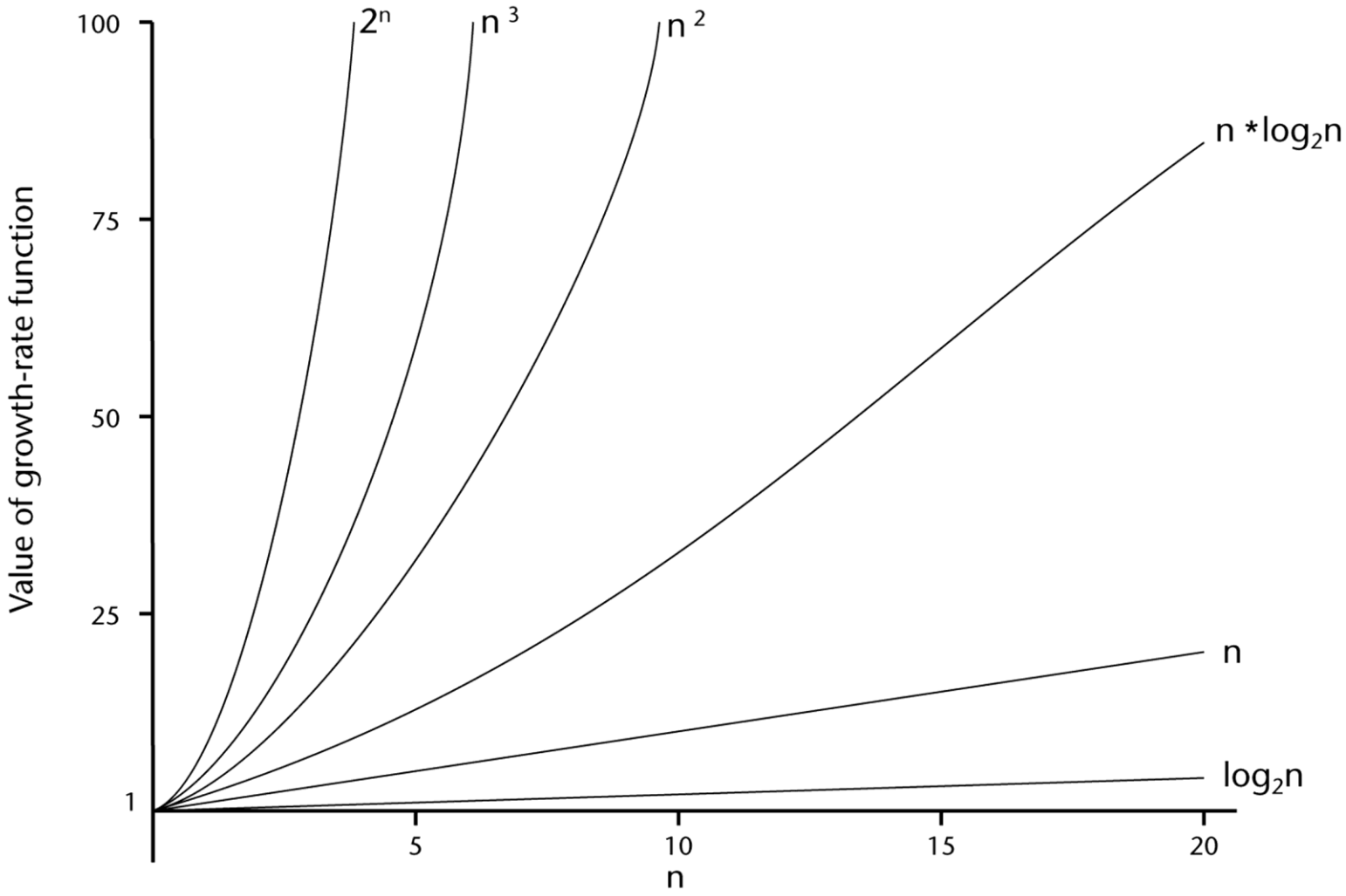
- Non-negative, monotonic function for N ≥ 1
- Sub linear growth: $\log(X) = o(X)$
- Basic equivalencies:

$$\log(cd) = \log(c) + \log(d)$$
$$\log(c/d) = \log(c) - \log(d)$$
$$\log(c^d) = d\log(c)$$
$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}$$

Corollary: logarithm grows slower than *any* polynomial!

$$\log(N) = o(N^c) \Rightarrow \log(\log(N)) = o(\log(N^c) = o(c\log(N))$$
$$\Rightarrow \boldsymbol{\log X = o(cX) = o(X)}$$

A Comparison of Growth-Rate Functions



Some Useful Mathematical Equalities

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n*(n+1)}{2} \approx \frac{n^2}{2}$$

$$\sum_{i=1}^{n} i^2 = 1 + 4 + \ldots + n^2 = \frac{n*(n+1)*(2n+1)}{6} \approx \frac{n^3}{3}$$

$$\sum_{i=0}^{n-1} 2^i = 0 + 1 + 2 + \ldots + 2^{n-1} = 2^n - 1$$

Relative Growth Rates

- Examples (which grows faster?)
  - 1000000 versus 0.01*sqrt(N)
  - log(N) versus sqrt(N)
  - $N\log(N)$ versus $N^{1.001}$
  - N^3 versus 10000*N^2
  - $\log^2(N)$ versus 10*log(N^5)
  - 2*$\log_2(N)$ versus $\log_3(N)$
  - N*2^N versus 3^N

$N^{1.001} = N * N^{0.001}$

$10*\log(N^5) = 50*\log(N)$

$$\log_3(N) = \frac{\log_2(N)}{\log_2(3)}$$

$3^N = 1.5^N * 2^N$

Algorithm Analysis

- Models and Assumptions
  - Consider rather abstract algorithm (a procedure or method)
  - Ignore the details/specifics of a computer
  - Assume sequential process (a sequence of instructions)
  - Ignore small constant factors (e.g., differences among “primitive” instructions)
  - Ignore language differences (e.g., C++ versus Java)

- Average-case versus worst-case performance
  - Example: finding a person in phonebook using sequential search
    - Best-case?
    - Worst-case?
    - Average-case?

How to Calculate Running Time?

- A simple example: compute $f(N) = \sum_{i=1}^{N} i^3$

```
public static int sum(int n)
{
  int partialSum = 0;
  for (int i = 1; i <= n; i++)
    partialSum += i*i*i;
  return partialSum;
}
```

- Assume each operation takes unit time, total time is "around" 6N+4
- *Time complexity* $O(N)$
- More accurately $\Theta(N)$ in this example

General Rules

- Simple statement: constant

  i++; i<n; etc.

- Simple loops: # of iterations times the cost of the loop body

```
i = 0; While (i < n) { … i++; …}
```

- Nested loops: (the product of # of iterations of outer and inner loops) times (the cost of the inner loop body)

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

- Consecutive statements: count the more expensive one

```
i = 0;
while (i < n) { … i++; …}
for (i=0; i < n; i++)
    for(j=0; j < n; j++)
      k++;
```

- If/else statement: count the more expensive branch (for worst-case analysis)

Running time in Recursive Methods

- Recursion often makes analysis more difficult.
- A simple one:

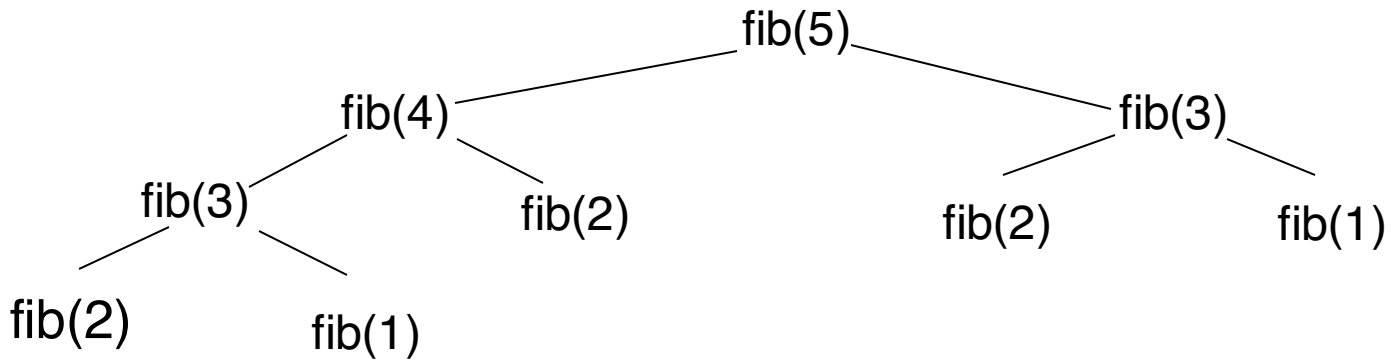
```
public static long factorial (int n)
{
  if (n <= 1 )
    return 1;
  else
    return n * factorial ( n - 1 );
}
```

- Count the number of recursive calls (and the cost of each call)

Recursion – Fibonacci Example

```
public static long fib( int n )
{
   if (n <= 1)
      return 1;
   else
      return fib(n - 1) + fib(n - 2);
}
```

- Redundant calculation of the same Fibonacci numbers, results in exponential running time



An Example: Maximum Subsequence Sum

- Given an array of (possibly negative) integers A[1…N], find the maximum sum of a sub-array A[i…j].

  example: 4 5 -6 -6 2 3 5 -8 -1 4 3 7 -2