CSCI 335
Software Design and Analysis III

Algorithm Analysis – Mathematical Background

Chapter 2

Exponents

$$a^x \cdot a^y = a^{x+y}$$
$$a^x \div a^y = a^{x-y}$$
$$(a^x)^y = a^{xy}$$

- Laws of exponents
- Derived from the basic definitions of multiplication, division, and exponentiation.

Logarithms

$$\log xy = \log x + \log y$$
$$\log x/y = \log x - \log y$$
$$\log x^y = y \log x$$
$$\log_a x = \frac{\log_b x}{\log_b a}$$

- Rules of logarithms
- Derived from the rules of exponents

Logarithms

$$\log x^y = y \log x$$

Proof.

Let $x = e^z$.

$x^y = (e^z)^y = e^{yz}$ by the rules of logarithms.

$\log e^{yz} = yz = y \log x$ by substitution.

□

Logarithms

$$\log_a x = \frac{\log_b x}{\log_b a}$$

Proof.

Let $x = a^z$.

$\log_b x = \log_b a^z = z \log_b a$ by the rules of logarithms.

But $z = \log_a x = \frac{\log_b x}{\log_b a}$

□

Modulo Operation

- Euclidean division is defined by the equation:

$$a = qb + r$$

$$a \text{ div } b = q$$
$$a \bmod b = r = a - qb$$

- Note that $a \bmod b = a$ for $1 \leq a < b$, not 0!
- For example $3 \bmod 11 = 3 \neq 0$.

Series

Math Background: Series

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

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}$$

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} \approxeq \frac{N^2}{2}$$

$$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$$

Big-Oh Notation

- We adopt special notation to define **upper bounds** and **lower bounds** on functions
- In CS, usually the functions we are bounding are running times, memory requirements.
- We will refer to the running time as *T(N)*

Definitions

- For *N* greater than some constant, we have the following definitions:

Upper bound on T(N)

$$T(N) = O(f(N)) \leftarrow T(N) \leq cf(N)$$

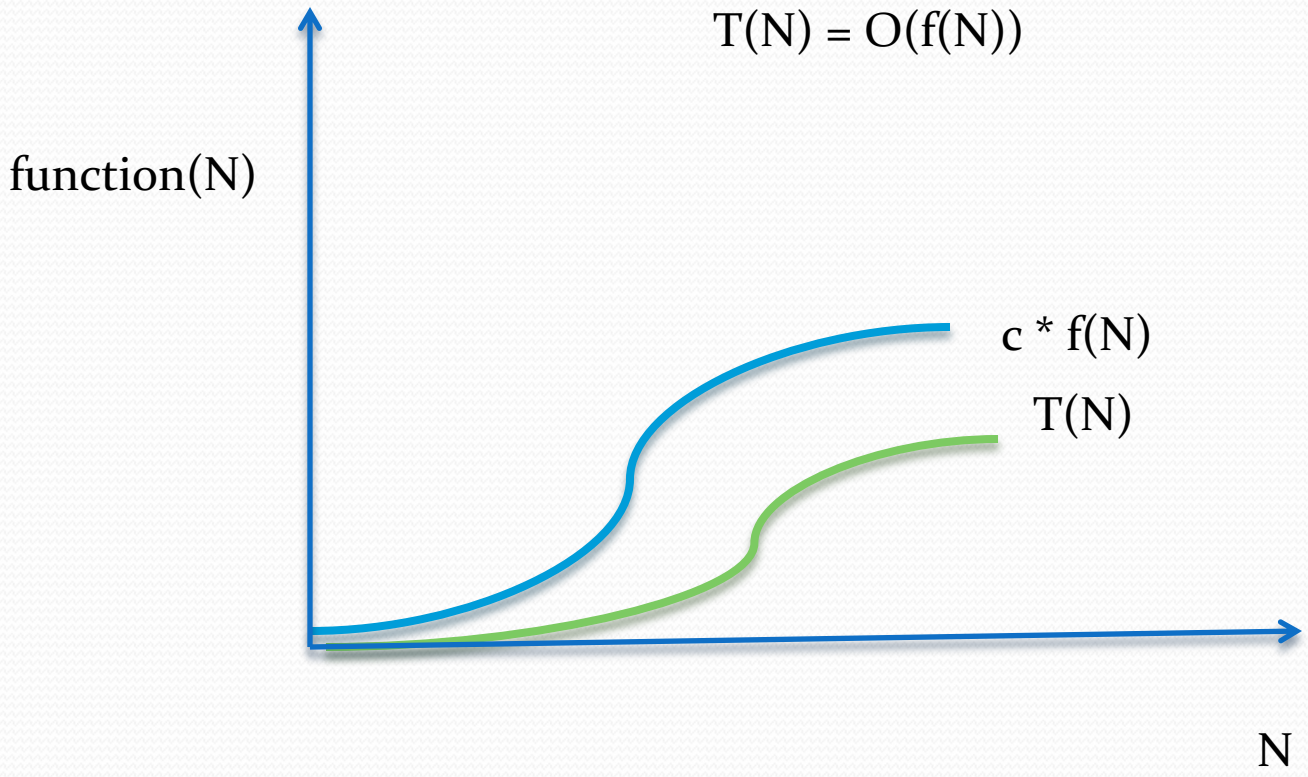
Lower bound on T(N)

$$T(N) = \Omega(g(N)) \leftarrow T(N) \geq cg(N)$$

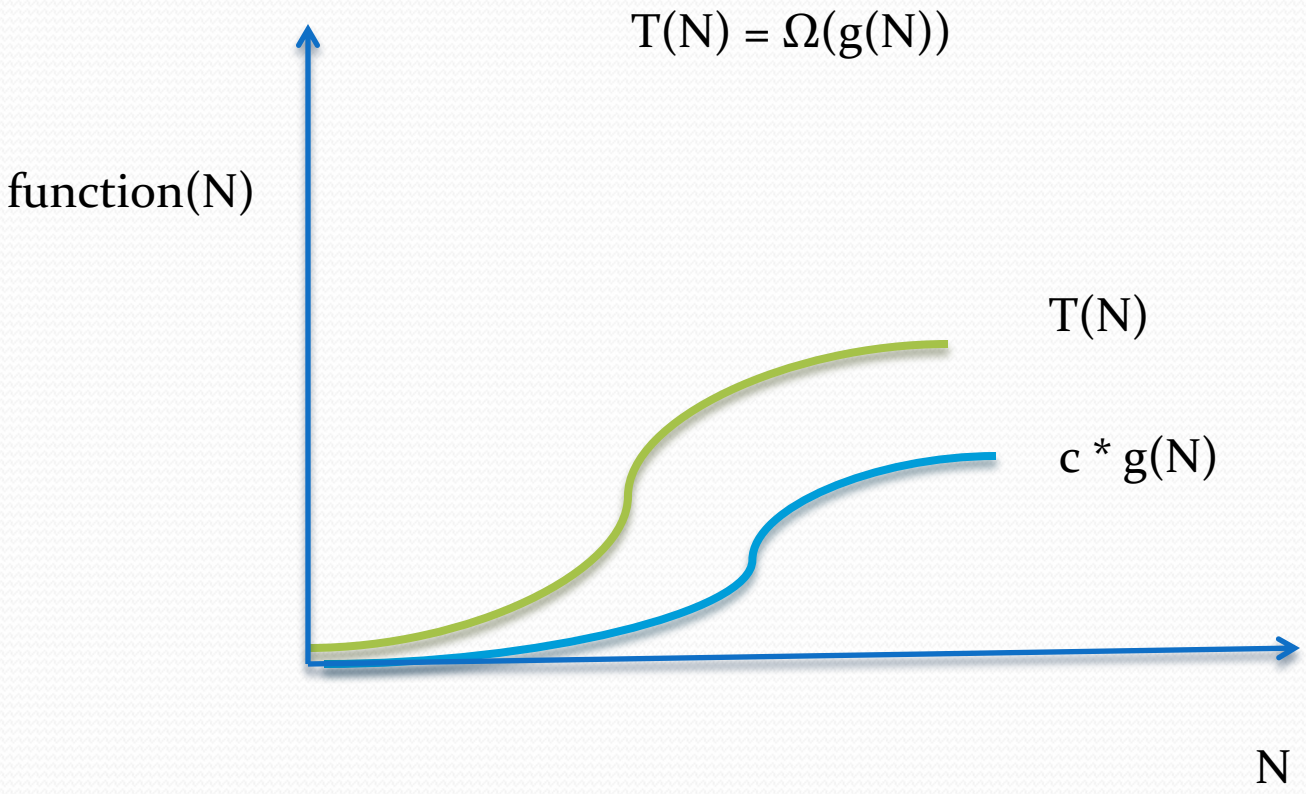
Tight bound on T(N)

$$T(N) = \Theta(h(N)) \leftarrow \begin{array}{c} T(N) = O(h(N)), \\ T(N) = \Omega(h(N)) \end{array}$$

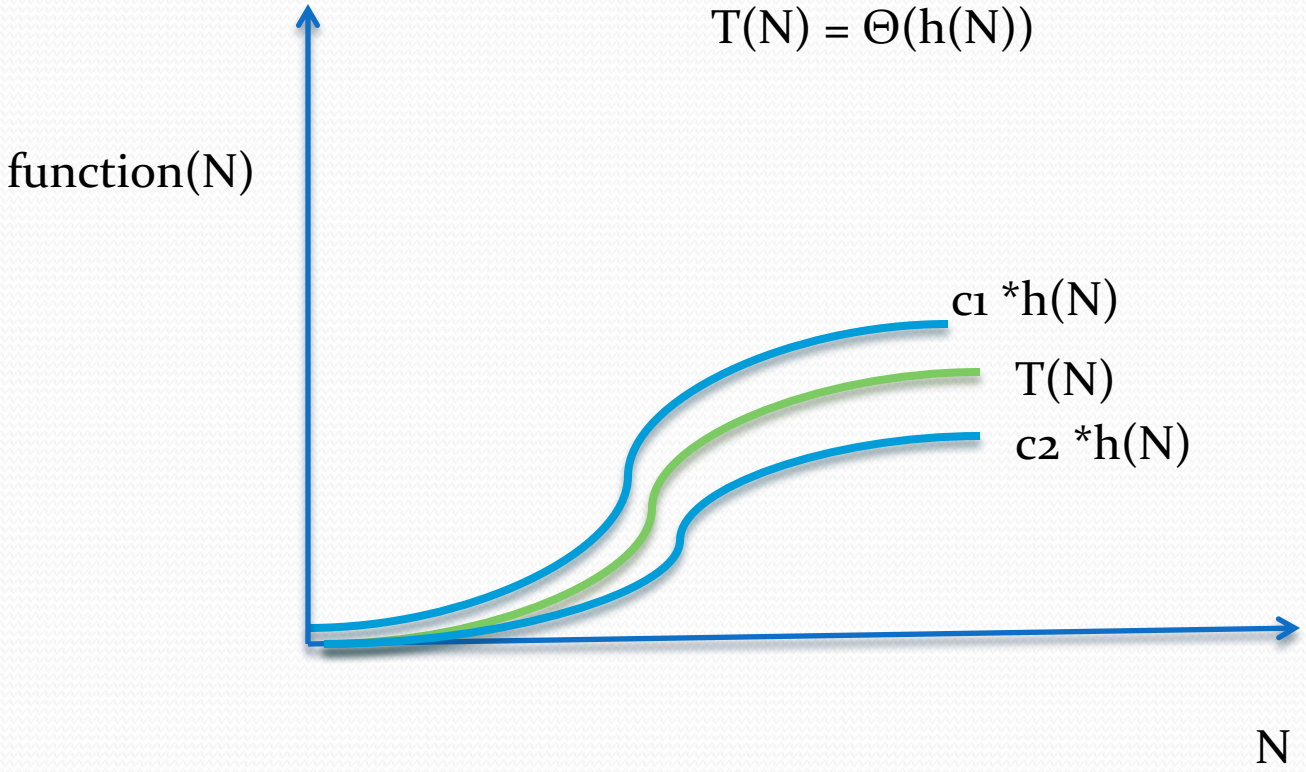
Examples: N = O(N^2), 4 * N^2 + N = O(N ^2), logN = O(N)



Examples: N^2 = Ω (N^2), N = Ω (logN)



Examples: 5* N^3 + N = Θ (N ^3), 0.2 * logN + loglogN= Θ (logN)



Definitions

- Alternately, O(f(N)) can be thought of as meaning

$$T(N) = O(f(N)) \leftarrow \lim_{N \to \infty} f(N) \geq \lim_{N \to \infty} T(N)$$

Relative Rates of Growth

- Big-Oh notation is also referred to as **asymptotic** analysis, for this reason.

A final definition

- Little-Oh:

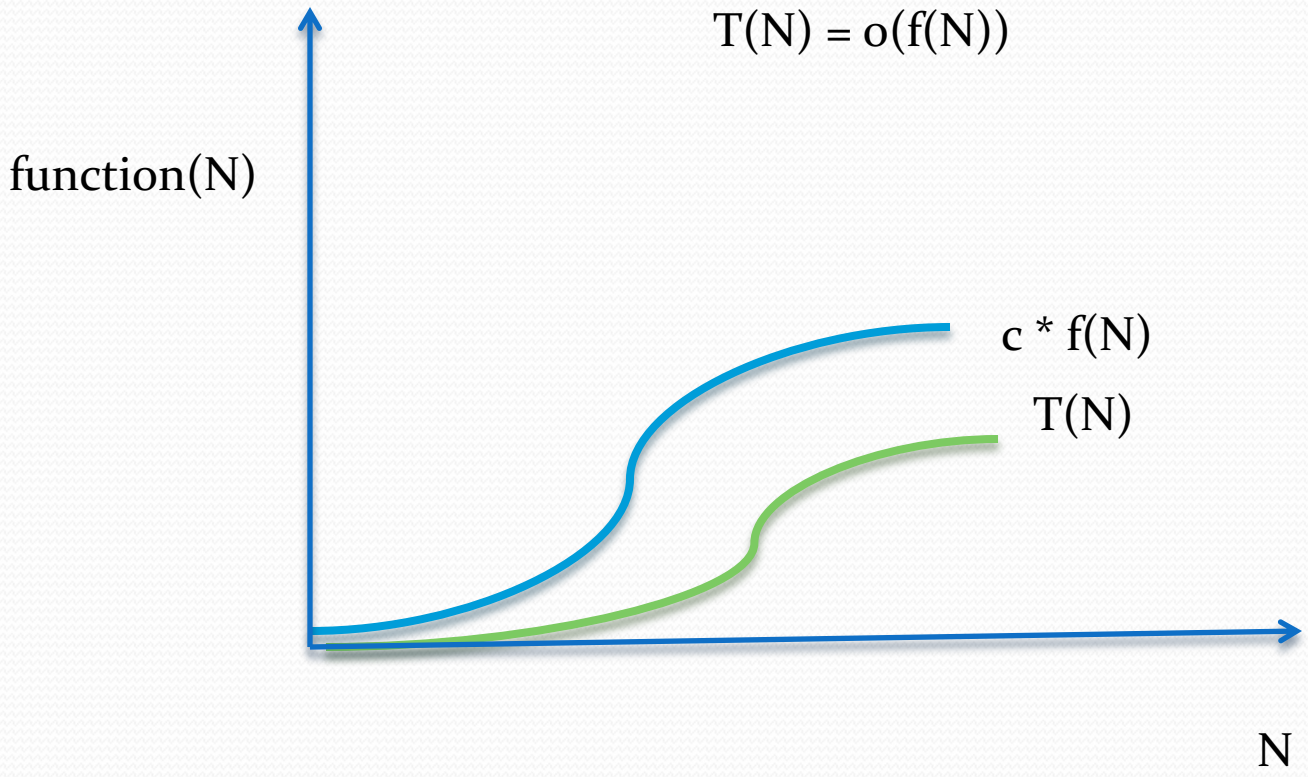
$T(N) = o(p(N))$ if for all constants c there exist an n0:
$T(N) < cp(N)$ when $N > no$.

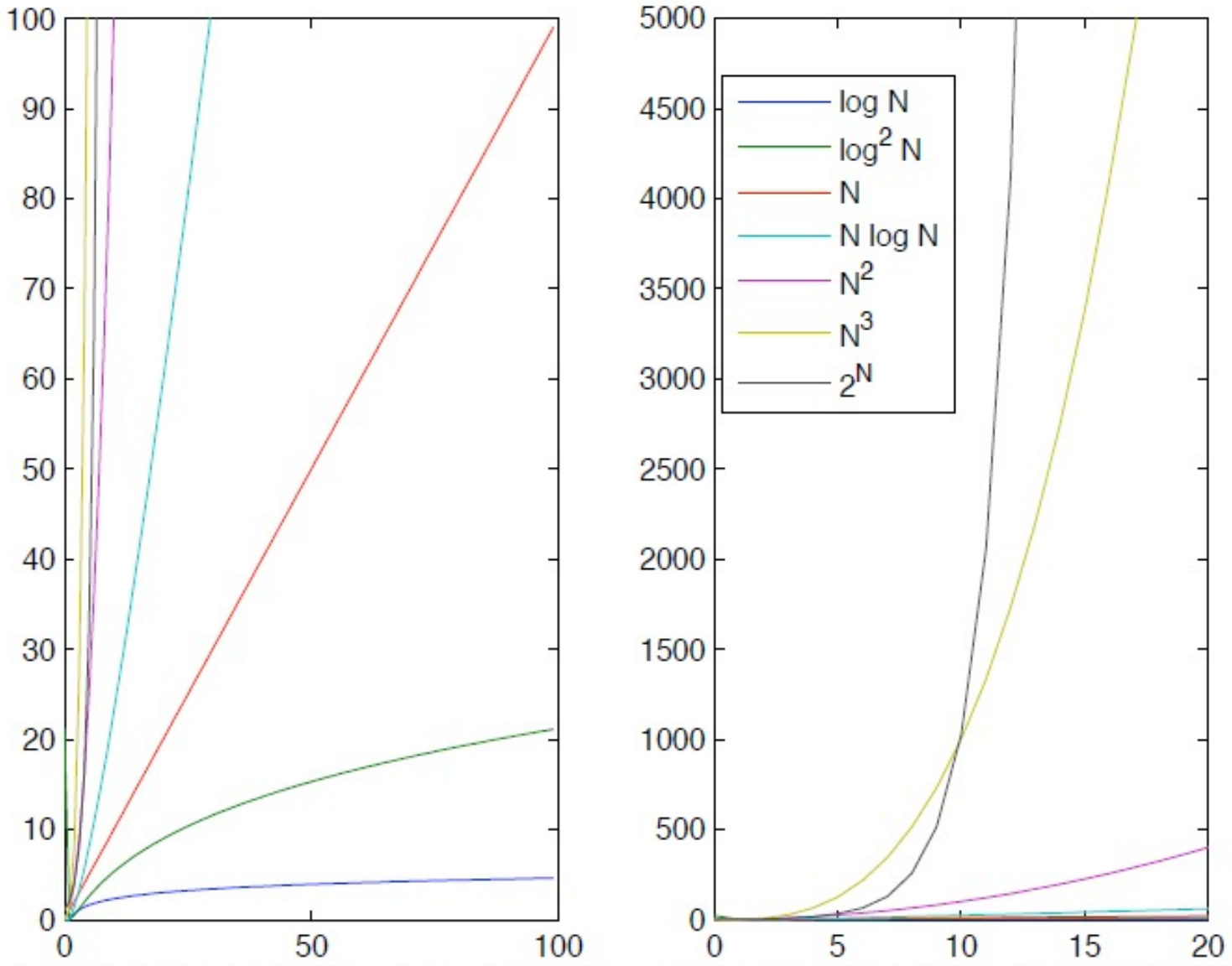
i.e. $T(N) = o(p(N))$ if
$T(N) = O(p(N))$ and $T(N) \neq \Theta(p(N))$.

The growth rate of $T(N)$ is less ($<$) than the growth rate of $p(N)$.

In $O()$ we had $\leq$ and in Θ we had both $\leq$ and $\geq$

Examples: N = o(N^2), ~~4 * N^2 + N = o(N^2)~~, logN = o(N)




100
90
80
70
60
50
40
30
20
10
0
0
50
100
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0
5
10
15
20
log N
log^2 N
N
N log N
N^2
N^3
2^N

T(N) = 2 * N

T(N) = o (N) X not true

Because T(N) = THETA(N)

T(N) = O(N)

T(N) = o(N^3). <. BETTER DESCRIPTION

T(N) = O (N^3). <= VALID AND TRUE

4 * logN + 10 = O(logN)

4 *logN + 10 = THETA(logN)

4 *logN + 10 = o (logN) XX WRONG XX

4 * logN + 10 = o(N) CORRECT

4 * logN + 10 = O(N) CORRECT, BUT NOT AS USEFUL

Comparing Growth Rates

$T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$

then **RULE 1**

(a) $T_1(N) + T_2(N) = O(f(N) + g(N))$

(b) $T_1(N)T_2(N) = O(f(N)g(N))$

- **Rule 2:**

  If $T(N)$ is a polynomial of degree k, then $T(N)=\Theta(N^k)$.

- **Rule 3:**

  $\log^k(N) = O(N)$ for any constant k.

Using the Notation

- So we do not write $O(2N^2)$ or $O(N^2 + N)$, but instead just $O(N^2)$. Lower order terms do not effect growth rate in the limit.

- If it is known that $T(N) = O(N^2)$ then even though it would be true to write $T(N) = O(N^3)$, it would not be considered a good estimate for the growth rate of $T(N)$

Using limits

In order to determine the relative growth rate of two functions f(N) and g(N) we can compute the limit:

$$\lim_{N \to \infty} f(N)/g(N)$$

If the limit is

Zero: f(N) = o(g(N))
C ≠ o : f(N) = Θ (g(N)
∞ : g(N) = o(f(N))
Oscillation: no relation

L'Hôpital's Rule

$$\text{If } \lim_{N\to\infty} f(N) = \infty \text{ and } \lim_{N\to\infty} g(N) = \infty$$

$$\lim_{N\to\infty} \frac{f(N)}{g(N)} = \lim_{N\to\infty} \frac{f'(N)}{g'(N)}$$

Where $f'(N)$ and $g'(N)$ are derivatives of $f(n)$ and $g(n)$ respectively.

f(N) = N^2 vs g(N) = N

f(N) / g(N) = N lim->infinity = ?

i.e.

---

f(N) = logN vs g(N) = N

f(N) / g(N) = logN / N

Lim -> infinity ?

(1/N) / 1 -> 1/N lim = Zero

i.e.

Typical growth rates

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Model of computation

- Model needed for algorithm analysis
  - Note algorithm not C++/code analysis
- One unit of time corresponds to simple instructions:
  - Addition, multiplication, comparison, assignment.
- Fixed size (32-bit) integers.
- Infinite memory
  - Interest in algorithm itself, not performance in any particular machine.

What to analyze

- Running time (Time complexity)
- Memory usage (Space complexity)
- What kind of input?
  - Worst-case running time
  - Best-case running time
  - Average-case running time
    - Sometimes it is hard to define what is the average input.
- Size of the input problem, N.
  - If input is an array, N is the size of the array.
  - If input is a number, N is the number of bits used to represent it.

Simple example

- Simple example. Running time?

```
// @n: a positive integer
// @returns 1^3 + 2^3 + ... + n^3
//    Will return 0, if n is smaller than 1.
int SumOfCubes(int n) {
  int sum_of_cubes = 0;
  for (int i = 1; i <= n; ++i)
    sum_of_cubes += i * i * i;
  return sum_of_cubes;
}
```

Simple example

- Simple example. Running time?

```
// @n: a positive integer
// @returns 1^3 + 2^3 + ... + n^3
//    Will return 0, if n is smaller than 1.
int SumOfCubes(int n) {
  int sum_of_cubes = 0;                    1
  for (int i = 1; i <= n; ++i)          n times
    sum_of_cubes += i * i * i;             4
  return sum_of_cubes;                     1
}
```

Simple example

- Simple example. Running time?

```
// @n: a positive integer
// @returns 1^3 + 2^3 + ... + n^3
//    Will return 0, if n is smaller than 1.
int SumOfCubes(int n) {
   int sum_of_cubes = 0;                    1
   for (int i = 1; i <= n; ++i)          n times
      sum_of_cubes += i * i * i;            4
   return sum_of_cubes;                     1
}
                                          _____

                        F(n) = 1 + 4 * n + 1 = O(n)
```

Recursion

- Factorial:

```
// @n: an integer
// @return n * (n – 1) * … * 1, if n >= 2
//          1, otherwise.
long Factorial(int n) {
  if (n <= 1)
     return 1;
  else
     return n * Factorial(n – 1);
}
```

- Running time?

Recursion

- Factorial:

```
// @n: an integer
// @return n * (n – 1) * … * 1, if n >= 2
//          1, otherwise.
long Factorial(int n) {
  if (n <= 1)                                  1
     return 1;                                 1
  else
     return n * Factorial(n – 1);              1 + ?
}
```

- Running time?

Recursion

- Factorial running time:

$$T(n) = 1 + T(n-1) \text{ for } n > 1,\ T(1)=2$$

Recursion

- Factorial running time:

$T(n) = 1 + T(n-1)$ for $n > 1$, $T(1)=2$

$T(n) = 1 + T(n-1) = 1 + (1 + T(n-2)) = \ldots =$

$= 1+(1(+\ldots(1+T(n-k))..)$ (k 1's)

$\Rightarrow T(n) = k + T(n-k) = (n-1) + T(n-(n-1)) = (n-1) + T(1) = (n-1)+2 = n+1 \Rightarrow T(n) = n + 1 \Rightarrow T(n) = O(n)$.

Linear algorithm.

Recursion

- Fibonacci (bad example):

```
// @n: an integer.
// @return the Fibonacci number of n.
long Fibonacci(int n) {
    if (n <= 1)
        return 1;
    else
        return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

- Running time?

Recursion

- Fibonacci (bad example):
- Running time: $T(n) = T(n-1) + T(n-2) + 2$, $T(0)=T(1)=1$

- We can prove (by induction) that $T(n) > 1.5^n$ for $n>2$

  $\Rightarrow T(n) = ?\ (1.5^n)$ $O/\Theta/\Omega/o$

- Section 1.2.5 proves that $T(n) < (5/3)^n$

  $\Rightarrow T(n) = ?\ ((5/3)^n)$ $O/\Theta/\Omega/o$

Fibonacci

N	T(n)	1.5^n
0	1	1.00
1	1	1.50
2	4	2.25
3	7	3.38
4	13	5.06

…

We can see than T(n) > 1.5^n for small values of n

Induction:

T(n)=T(n-1)+T(n-2)+2>T(n-1)+T(n-2)>(ind. Hyp.) 1.5^(n-1)+1.5^(n-2)=
2.5*1.5^(n-2)>2.25*1.5^(n-2)=1.5^2*1.5^(n-2)=1.5^n.

Exponential: Really bad result! (Use iteration instead)

Maximum Subsequence Problem

Given (possibly negative) integers A_1, ..., A_N, find the maximum value of $\sum_{k=i}^{j} A_k$ for $i, j = 1, \ldots, N, i \leq j$

- Given a sequence of integers (possibly negative), find the subsequence whose sum is the maximum

-2	11	-4	13	-5	-2

Many algorithms to solve this simple problem

When input size is small, brute force works fine.

Algorithm 1 (brute force)

```
// @a: a vector of integers.
// @return the maximum positive subsequence sum. If the
//    maximum sum is smaller than 0, return 0.
//    Brute force approach.
int MaxSubsequenceSum1(const vector<int> &a) {
    int max_sum = 0;
    for (size_t i = 0; i < a.size(); ++i)
      for (size_t j = i; j < a.size(); ++j) {
            int current_sum = 0;
            for (size_t k = i; k <= j; ++k)
                current_sum += a[k];
            if (current_sum > max_sum)
                max_sum = current_sum;
      }
    return max_sum;
}
```

Algorithm 1 (brute force)

```
// @a: a vector of integers.
// @return the maximum positive subsequence sum. If the
//    maximum sum is smaller than 0, return 0.
//    Brute force approach.
int MaxSubsequenceSum1(const vector<int> &a) {
    int max_sum = 0;
    for (size_t i = 0; i < a.size(); ++i)
      for (size_t j = i; j < a.size(); ++j) {
            int current_sum = 0;
            for (size_t k = i; k <= j; ++k)
                current_sum += a[k];
            if (current_sum > max_sum)
                max_sum = current_sum;
      }
    return max_sum;
}
```

- Check all possible subsequences
- N starting places, average N/2 lengths to check, average N/4 numbers to add, so $O(N^3)$. Redundant work.
- For N=1,000,000 this is 10^{18} about 58 days at 2Ghz (assuming 2billion operations per second)

Algorithm 2 (brute force)

```
// @a: a vector of integers.
// @return the maximum positive subsequence sum. If the
//    maximum sum is smaller than 0, return 0.
//    Brute force approach (slightly better).
int MaxSubsequenceSum2(const vector<int> &a) {
    int max_sum = 0;
    for (size_t i = 0; i < a.size(); ++i) {
      int current_sum = 0;
      for (size_t j = i; j < a.size(); ++j) {
          current_sum += a[j];
          if (current_sum > max_sum)
             max_sum = current_sum;
      }
    }
    return max_sum;
}
```

Algorithm 2 (brute force)

```
// @a: a vector of integers.
// @return the maximum positive subsequence sum. If the
//    maximum sum is smaller than 0, return 0.
//    Brute force approach (slightly better).
int MaxSubsequenceSum2(const vector<int> &a) {
    int max_sum = 0;
    for (size_t i = 0; i < a.size(); ++i) {
      int current_sum = 0;
      for (size_t j = i; j < a.size(); ++j) {
          current_sum += a[j];
          if (current_sum > max_sum)
             max_sum = current_sum;
      }
    }
    return max_sum;
}
```

- Removed one loop
- $O(N^2)$.
- About 8 seconds at 2Ghz

Algorithm 3 (recursive)

- Maximum subsequence is either entirely in first half, or entirely in second half, or is patch of best last half of first half with best first half of last half.
- Example:

First half				Second half			
4	-3	5	-2	-1	2	6	-2

Best first half: ?

Best second half: ?

Best last part of first half: ?

Best first part of second half: ?

Result: ?

Code

```
// @a: a vector of integers.
// @left: a left index.
// @right: a right index.
// It is assumed that left <= right. It is also assumed that right is not
//  outside of the boundaries of the vector.
// @return the maximum positive subsequence sum of items
//     a[left], a[left + 1], … , a[right].
// Recursive solution.
int MaxSubsequenceSum3(const vector<int> &a, size_t left, size_t right) {
    if (left > right) abort();  // Invalid.
    if (right >= a.size()) abort();  // Invalid.

    if (left == right)  // Base case.
       return a[left] > 0 ? a[left]: 0;

    const size_t center = (left + right) / 2;
    const int max_left_sum = MaxSubsequenceSum3(a, left, center);
    const int max_right_sum = MaxSubsequenceSum3(a, center + 1, right);

    // Compute maximum of left part when you always end at center.
    int max_left_border_sum = 0, left_border_sum = 0;
    for (int i = center; i >= left; --i) {
       left_border_sum += a[i];
       if (left_border_sum > max_left_border_sum)
             max_left_border_sum = left_border_sum;
    }
    // continues in next page…
```

```
 // Compute maximum of right part when you always start at center + 1.
 int max_right_border_sum = 0, right_border_sum = 0;
 for (int i = center + 1; i <= right; ++i) {
       right_border_sum += a[i];
       if (right_border_sum > max_right_border_sum)
           max_right_border_sum = right_border_sum;
    }

 return MaxOfThree(max_left_sum, max_right_sum,
                   max_right_border_sum + max_left_border_sum);
}  // End of MaxSubsequenceSum3.

// Driver routine, that calls the recursive one.
int MaxSubsequenceSum3Driver(const vector<int> &a) {
  return MaxSubsequenceSum3(a, 0, a.size() – 1);
}
```

```
// @a: a vector of integers.
// @left: a left index.
// @right: a right index.
// It is assumed that left <= right. It is also assumed that right is not
//  outside of the boundaries of the vector.
// @return the maximum positive subsequence sum of items
//     a[left], a[left + 1], … , a[right].
// Recursive solution. Alternative version. Is this better?
int MaxSubsequenceSum3(const vector<int> &a, size_t left, size_t right) {
    if (left > right) abort();  // Invalid.
    if (right >= a.size()) abort();  // Invalid.

    if (left == right)  // Base case.
       return a[left] > 0 ? a[left]: 0;

    const size_t center = (left + right) / 2;
    const int max_left_sum = MaxSubsequenceSum3(a, left, center);
    const int max_right_sum = MaxSubsequenceSum3(a, center + 1, right);
    // Compute maximum of left part when you always end at center.
    int max_left_border_sum = 0, left_border_sum = 0;
    for (int i = center; i >= left; --i) {
       left_border_sum += a[i];
       if (left_border_sum > max_left_border_sum)
             max_left_border_sum = left_border_sum;
    }
    // Continues in next page…
```

```
// Compute maximum of right part when you always start at center + 1.
int max_right_border_sum = 0, right_border_sum = 0;
for (int i = center + 1; i <= right; ++i) {
      right_border_sum += a[i];
      if (right_border_sum > max_right_border_sum)
          max_right_border_sum = right_border_sum;
   }

return MaxOfThree(MaxSubsequenceSum3(a, left, center),
                  MaxSubsequenceSum3(a, center + 1, right),
                  max_right_border_sum + max_left_border_sum);
}  // End of MaxSubsequenceSum3 (alternative version).
```

Running time analysis

- T(1) = 1 (base case)
- T(N) = 2 T(N/2) + O(N) (why ?)
- Simpler (note that we always ignore constants)

T(N) = 2 T(N/2) + N (assume N = 2^k)

Running time analysis (expand)

T(N) = 2 T(N/2) + N (assume N = 2^k)
T(N/2) = 2 T(N/4) + N/2
…
T(N/2^i) = 2 T(N/2^(i+1)) + N/2^i
…
T(N/2^(k-1)) = 2 T(N/2^k) + N/2^k-1)
T(2) = 2 T(1) + 2

2^(k) = N => k = log(N)

So T(N) = 2 * (2*T(N/4) + N/2) + N = 2^2T(N/4)+N + N=
2^2 (2*T(N/8)+N/4) + N + N = 2^3 T(N/8)+N+N+N= 2^3T(N/2^3)+3N …..
T(N) = 2^kT(N/2^k)+kN, but 2^k=N and k=log(N)
Therefore
T(N) = N*T(1)+log(N)*N = N + N*log(N) =**O (N*logN)**
(0.003 secs on earlier example)

Algorithm 4 (linear version)

```
// @a: a vector of integers.
// @return the maximum positive subsequence sum. If the
//     maximum sum is smaller than 0, return 0.
//     Linear version.
int MaxSubsequenceSum4(const vector<int> &a) {
    int max_sum = 0, current_sum = 0;
    for (size_t i = 0; i < a.size(); ++i) {
      current_sum += a[i];
      if (current_sum > max_sum) max_sum = current_sum;
      else if (current_sum < 0) current_sum = 0;
    }
    return max_sum;
}
// Run on example: -2, 10, 3, -4, -8, -2, 10, 2, -5
```

-2, 10, 3, -4, -8, -2, 10, 2, -5

Algorithm 4 (Linear)

- Why does it work?
- Running time is obvious, but correctness is not.
- 0.0005 secs on earlier example

Algorithm 4 (example)

- Consider this sequence:

-2, 10, 3, -4, -8, -2, 10, 2, -5, …

Append 3, 20 to the above sequence and continue…

- Algorithm 4 is an example of an **on-line algorithm**
  - Data can be read sequentially with no need to store it in main memory (i.e. you could apply the algorithm write off the disk, or as it arrives from the internet, or through a sensor)
  - At any point in time the algorithm provides the current best solution to the problem.

Four different algorithms

Input Size	Algorithm Time			
	1	2	3	4
	$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
$N = 10$	0.000009	0.000004	0.000006	0.000003
$N = 100$	0.002580	0.000109	0.000045	0.000006
$N = 1{,}000$	2.281013	0.010203	0.000485	0.000031
$N = 10{,}000$	NA	1.2329	0.005712	0.000317
$N = 100{,}000$	NA	135	0.064618	0.003206

Note, that time required to read input is included in the above analysis.

Binary Search

- Given an integer X and integers A_0, … , A_{N-1}, which are **presorted** an already in memory, find i such that A_i=X, or return i=-1 if X is not in the input
- Code
- Running time?

```
// Performs the standard binary search using two comparisons per level.
// @a: input vector of elements. Assumes sorted vector.
// @x: item we are searching for.
// @return index where item is found or -1 if not found.
template <typename Comparable>
int BinarySearch(const vector<Comparable> &a, const Comparable &x) {
    int low  = 0;
    int high = a.size() - 1;
    while (low <= high) {
      const int mid = (low + high) / 2;
      if (a[mid] == x) return mid;  // Found.
        if (a[mid] < x)
          low = mid + 1;
        else
          high = mid - 1;
    }
    return -1;
}
```

Binary Search

- Running time?
  - $T(N) = 1 + T(N/2)$, $T(1) = 1$

  - $T(N) = 1 + T(N/2) \quad = 1 + 1 + T(N/2^2) = 2+T(N/2^2) =$

    $= 2+1+T(N/2^3) = 3+T(N/2^3) \quad = \ldots = \mathbf{k+T(N/2^k)} = \ldots$
  - If N is a power of 2 (i.e. $N=2^k$ with $k = \log(N)$) we will have:

    $T(N)=\mathbf{k+T(N/2^k)} = k+T(1)=\log(N)+1 \Rightarrow$

$$\mathbf{T(N) = O(\log N)}$$

Exponentiation ?

- Compute X^N, for positive N
- Naïve algorithm: (N-1) multiplications
- Smart algorithm O(logN)
- How?

Fast Exponentiation

$$X^0 = 1$$
$$X^1 = X$$
$$X^N = \left(X^{N/2}\right)^2, \text{if } N \text{ is even}$$
$$X^N = X\left(X^{(N-1)/2}\right)^2, \text{if } N \text{ is odd}$$

$$T(N) = \begin{cases} T\left(\frac{N}{2}\right) + 1, & \text{even } N \\ T\left(\frac{N-1}{2}\right) + 2, & \text{odd } N \end{cases}$$

Example

$$X^{62} = (X^{31})^2, X^{31} = X(X^{15})^2, X^{15} = X(X^7)^2, \ldots$$

```
// @x: a number.
// @n: a positive integer.
// @return x^n.
long Power(long x, unsigned int n) {
    if (n == 0) return 1;
    if (n == 1) return x;
    if (n % 2 == 0)
        return Power(x * x, n / 2);
    else
        return x * Power(x * x, n / 2);
}
```

Exponentiation

```
// Can we replace recursive call
//      Power(x * x, n / 2)
// by one of the following?
return Power(Power(x, 2), n / 2);
return Power(Power(x, n / 2), 2);
return Power(x, n / 2) * Power(x, n / 2);
```

Exponentiation

```
// Endless loop.
return Power(Power(x, 2), n /2);
return Power(Power(x, n / 2), 2);

// Non-logarithmic runtime. Why?
Return Power(x, n / 2) * Power(x, n / 2);
```

Problem 2.14

- Evaluate $p(x) = a_0x^0 + a_1\,x^1 + \cdots + a_{N-1}\,x^{N-1}$
- Running time if we use naïve exponentiation ?
- Running time if we use fast exponentiation?
- How about using the algorithm below?

Horner's Method:

```
poly=0;
for (i = n ; i>=0; i--)
     poly = x * poly + a[i];
```