

# Algorithm Analysis

Algorithm Analysis

Textbook Ch 2,3

# Outline

- Justification for analysis
- Landau symbols
- Run time of programs
- Best-, worst-, and average-case

# Comparing algorithms

Suppose we have two algorithms. How can we tell which is better?

We could implement both algorithms, run them both

- Expensive and error prone

Preferably, we should analyze them mathematically

- *Algorithm analysis*

# Example

- Find a item in a sorted array of length  $N$
- Algorithm 1: Linear search (check each item from left to right)
  - Do you use this approach when looking up a word in a dictionary?
- Algorithm 2: Binary search

key is 11

```
key < 50
```

low                          mid                          high

↓                          ↓                          ↓

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12]

list    2    4    7    10   11   45   **50**   59   60   66   69   70   79

key > 7

low                  mid                  high

↓                    ↓                    ↓

[0] [1] [2] [3] [4] [5]

list   

2	4	7	10	11	45
---	---	---	----	----	----

```
key == 11
```

low mid high

[3] [4] [5]

list | 10 11 45

# Implementation

## Algorithm 1: Linear search

```
int lfind(int key, int a[], int n)
{
    if (n==0) return -1;
    if (key == a[n-1]) return n-1;
    return lfind(key, a, n-1);
}
```

## Algorithm 2: Binary search ([demo](#))

```
int bfind(int key, int a[], int left, int right)
{
    if (left+1 == right) return -1;
    int m = (left + right) / 2;
    if (key == a[m]) return m;
    if (key < a[m]) return bfind(key, a, left, m);
    else return bfind(key, a, m, right);
}
```

# Empirical comparison

```
// create an array [0,1,2,...,n-1]
```

```
for (i=0; i<n; i++) a[i] = i;
```

```
// search each item in the array
```

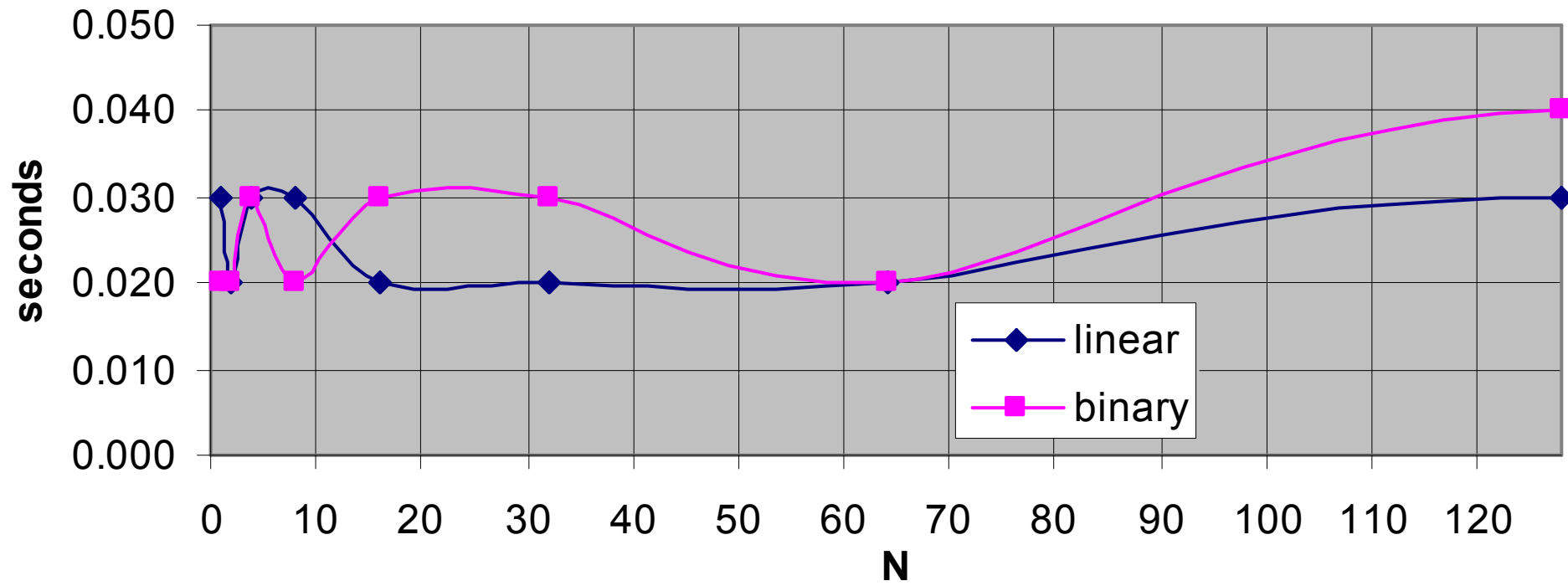
```
for (i=0; i<n; i++) lfind(i,a,n);
```

or

bfind(i,a,-1,n)

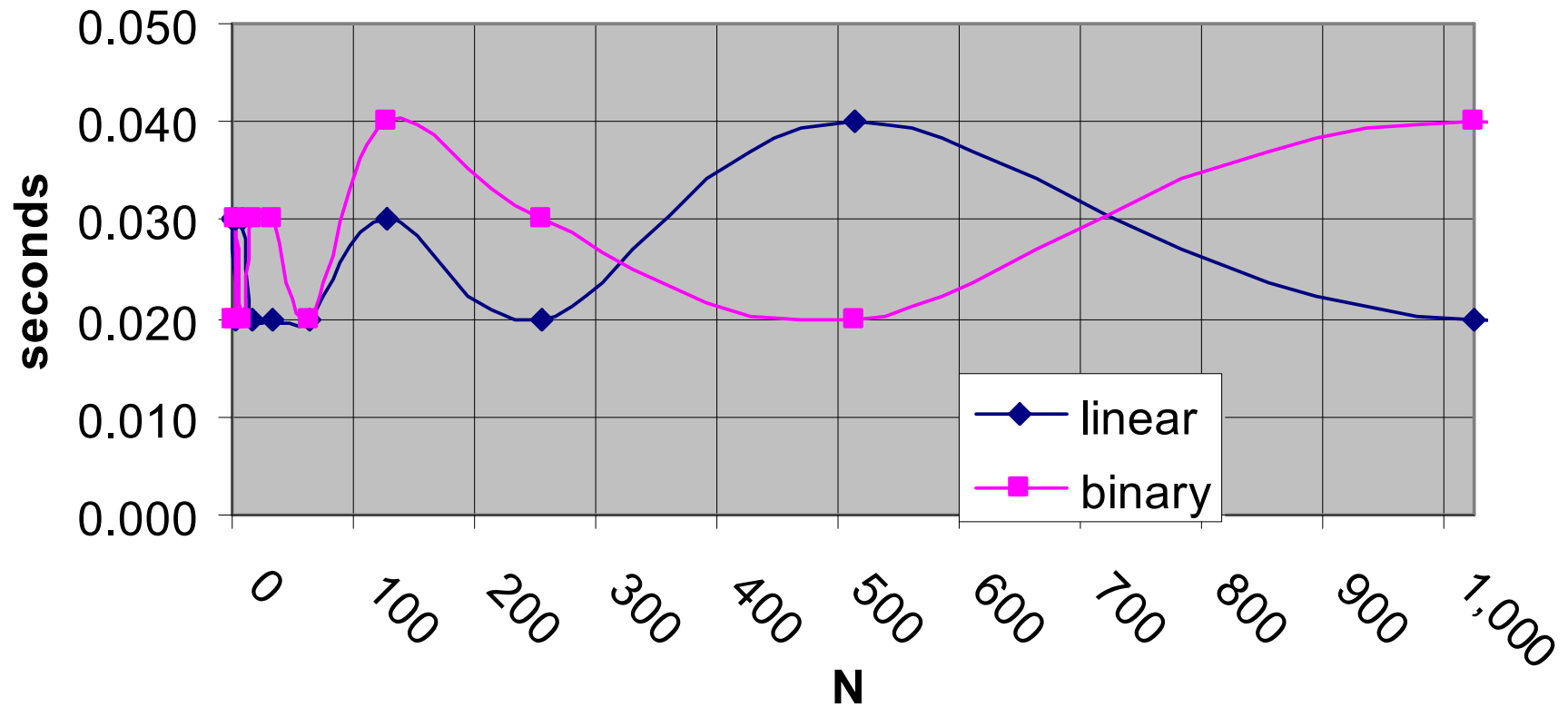
# Empirical comparison

## linear vs binary search



# Empirical comparison

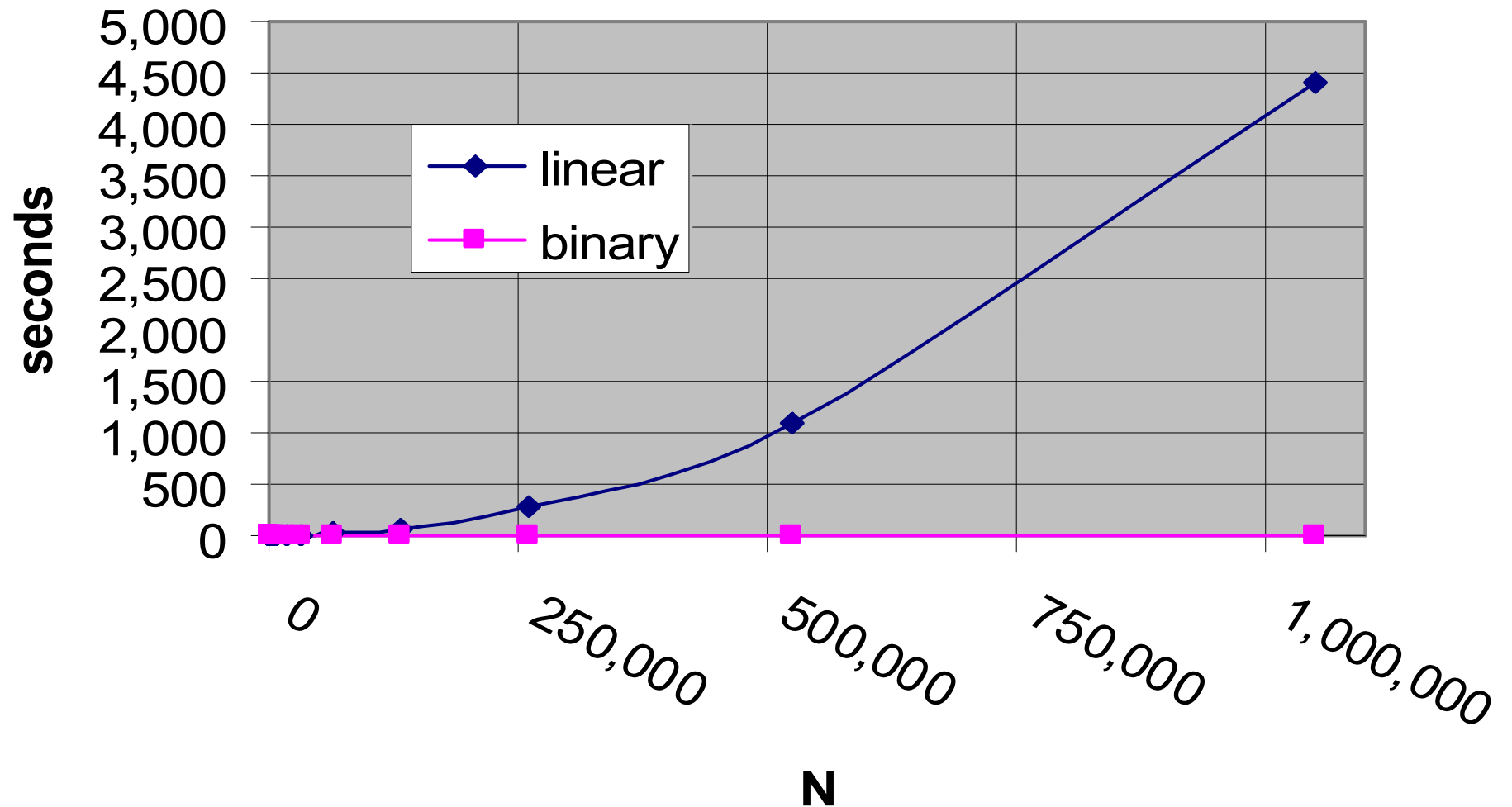
## linear vs binary search





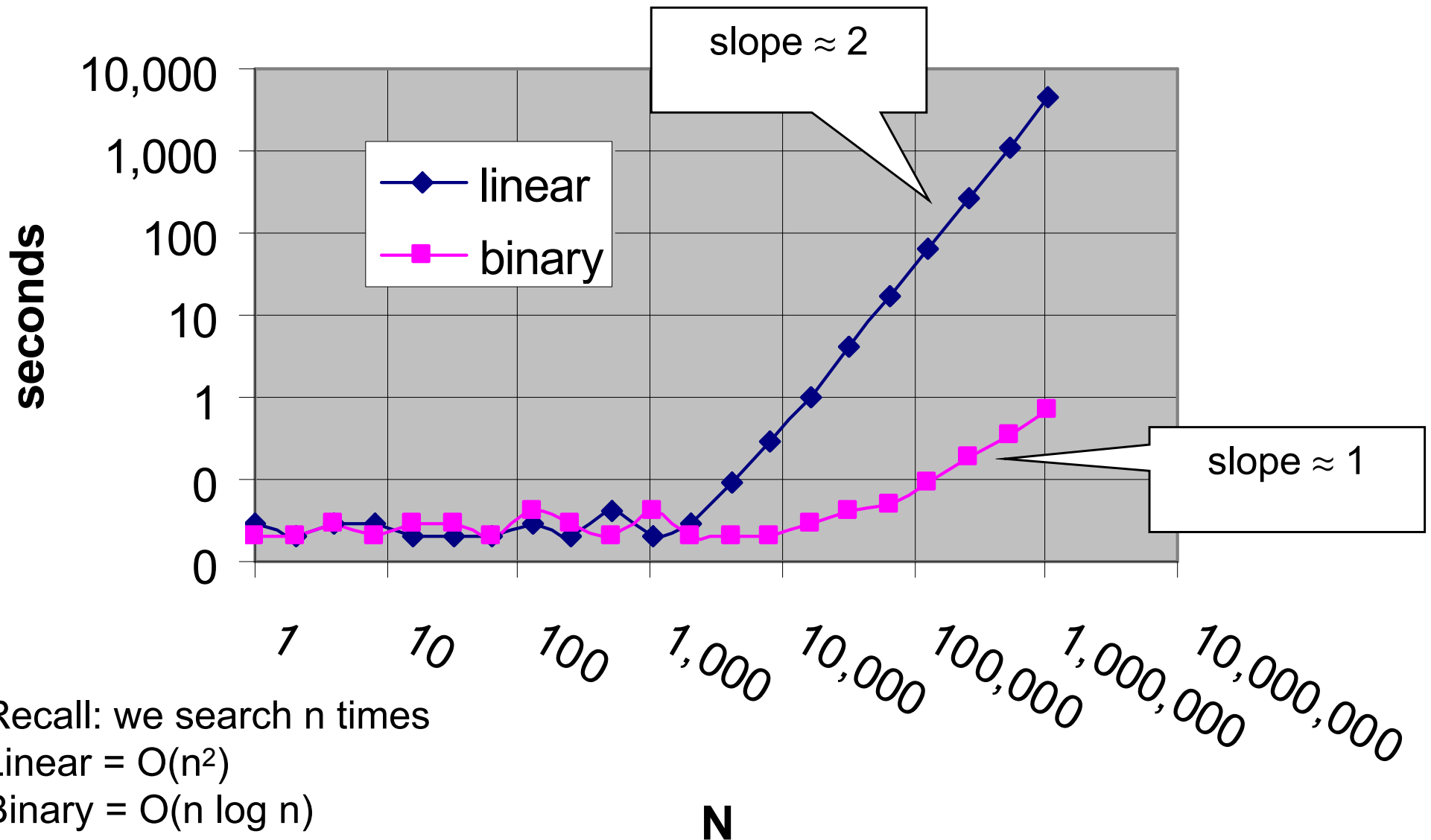
# Empirical comparison

## linear vs binary search



# Empirical comparison

## linear vs binary search - log/log plot



# Analytical comparison

- Linear search
  - $O(n)$
- Binary search
  - $O(\log n)$
- So binary search is better than linear search

# Asymptotic Analysis

In general, we will always analyze algorithms with respect to one or more variables

We will begin with one variable:

- The number of items  $n$  currently stored in an array or other data structure
- The number of items expected to be stored in an array or other data structure
- The dimensions of an  $n \times n$  matrix

Examples with multiple variables:

- Dealing with  $n$  objects stored in  $m$  memory locations
- Multiplying a  $k \times m$  and an  $m \times n$  matrix
- Dealing with sparse matrices of size  $n \times n$  with  $m$  non-zero entries

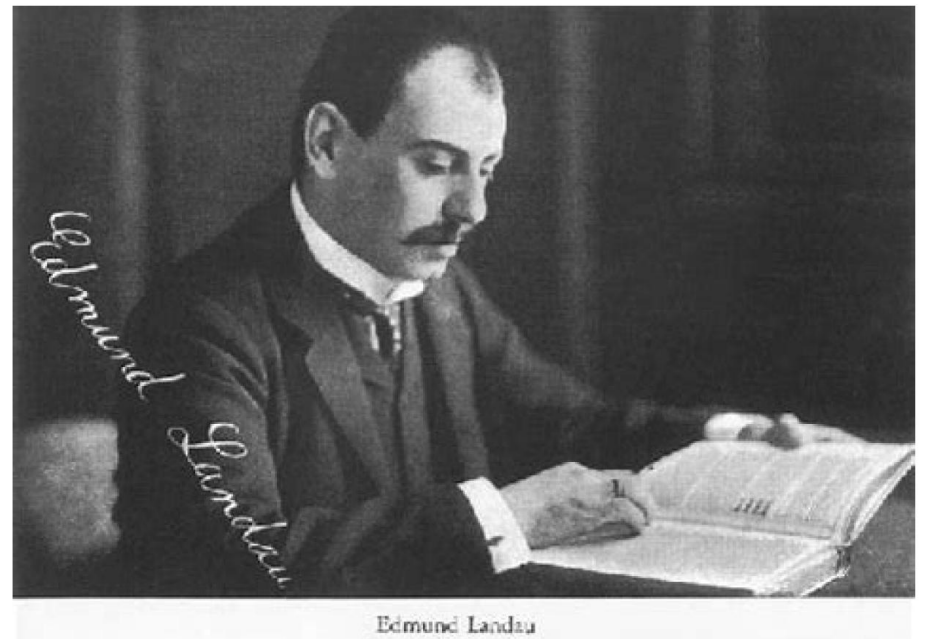
# Asymptotic Analysis

Given an algorithm, we want to describe its computational cost mathematically and in a machine-independent way

For this, we need Landau symbols (a.k.a. Big-O notation) and the associated asymptotic analysis

# Outline

- Justification for analysis
- Landau symbols
- Run time of programs
- Best-, worst-, and average-case

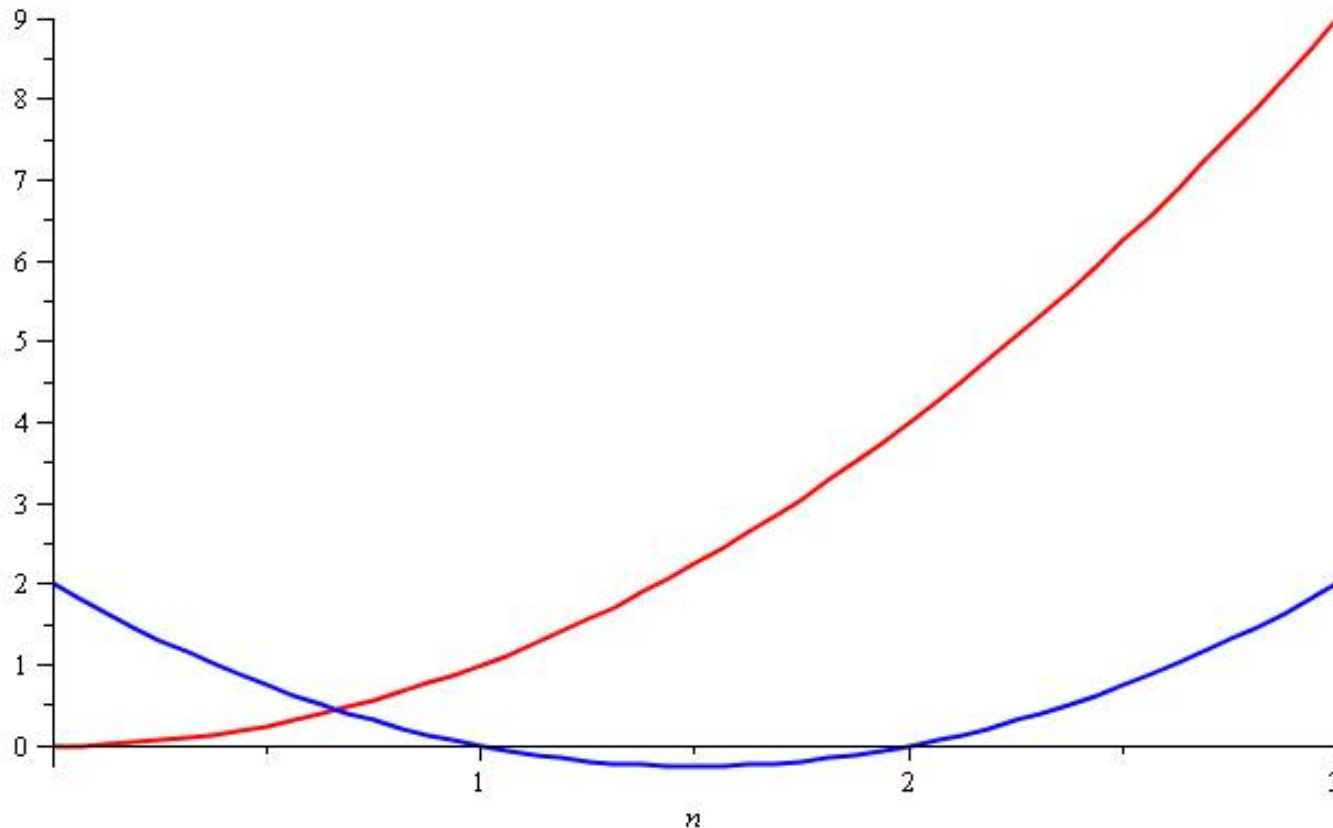


# 二次的 Quadratic Growth

Consider the two functions

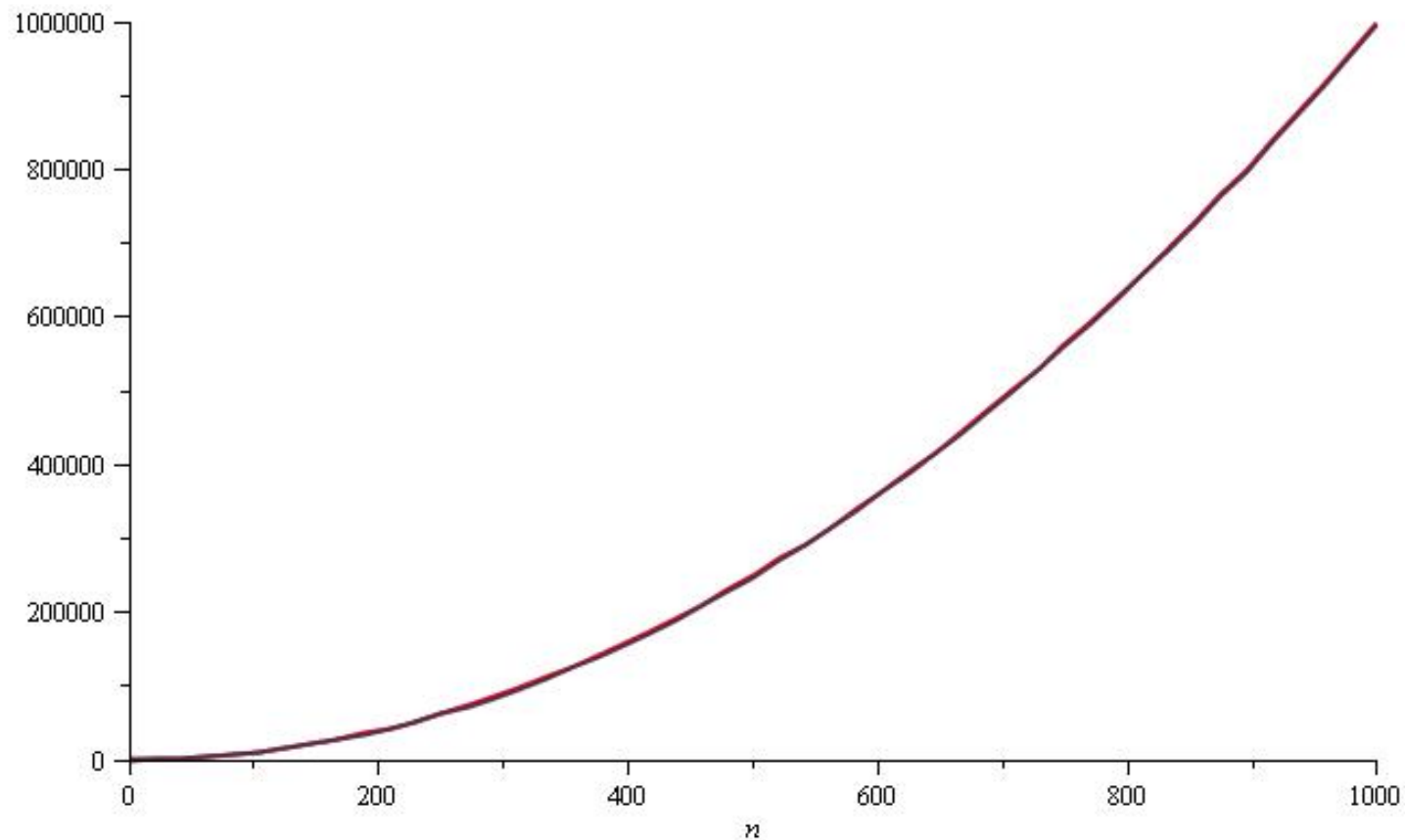
$$f(n) = n^2 \text{ and } g(n) = n^2 - 3n + 2$$

Around  $n = 0$ , they look very different



# Quadratic Growth

Yet on the range  $n = [0, 1000]$ , they are (relatively) indistinguishable:





# Quadratic Growth

The absolute difference is large, for example,

$$f(1000) = 1\,000\,000$$

$$g(1000) = 997\,002$$

but the relative difference is very small

$$\left| \frac{f(1000) - g(1000)}{f(1000)} \right| = 0.002998 < 0.3\%$$

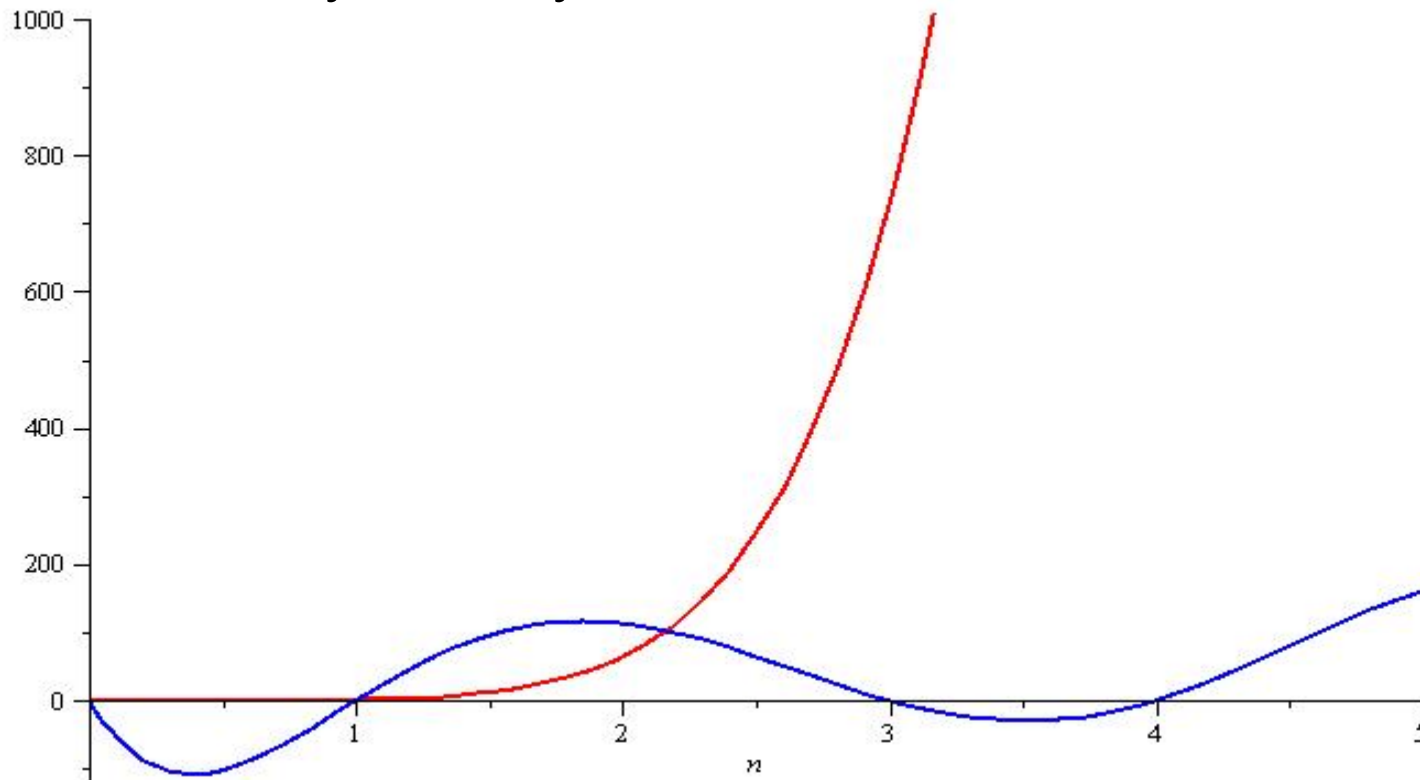
and this difference goes to zero as  $n \rightarrow \infty$

# Polynomial Growth

To demonstrate with another example,

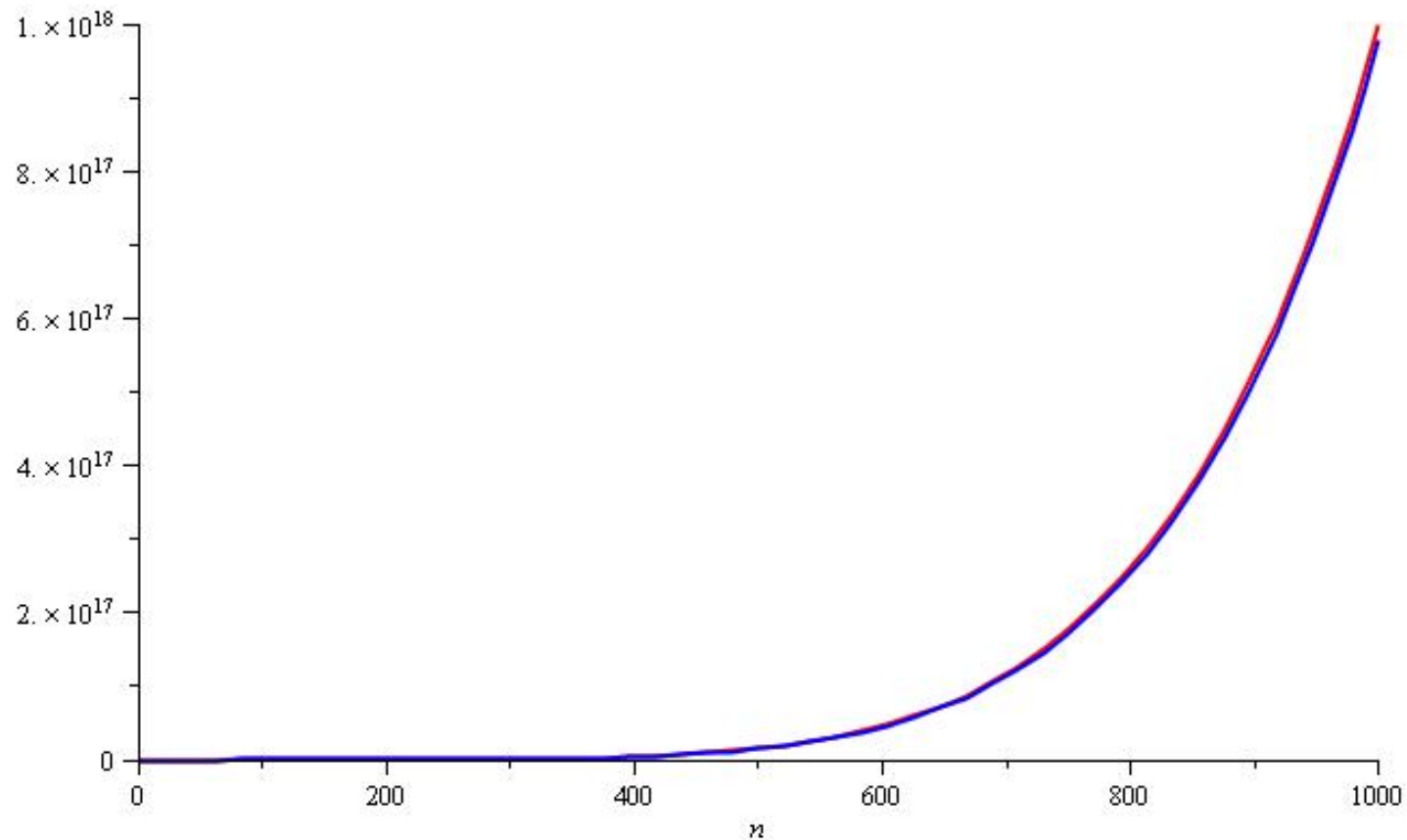
$$f(n) = n^6 \quad \text{and} \quad g(n) = n^6 - 23n^5 + 193n^4 - 729n^3 + 1206n^2 - 648n$$

Around  $n = 0$ , they are very different



# Polynomial Growth

Still, around  $n = 1000$ , the relative difference is less than 3%



# Polynomial Growth

The justification for both pairs of polynomials being similar is that, in both cases, they each had the same leading term:

$n^2$  in the first case,  $n^6$  in the second

What if the coefficients of the leading terms were different?

- In this case, both functions would exhibit the same rate of growth, however, one would always be **proportionally larger**

However, if the two functions describe the run-time of two algorithms

- We can always run the slower algorithm on a faster computer to make them equally fast

In contrast: **can we make linear search equally fast to binary search by using a faster computer (say, an Ultimate Laptop)?**

# Weak ordering

Consider the following definitions:

- We will consider two functions to be equivalent,  $f \sim g$ , if

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c \text{ where } 0 < c < \infty$$

- We will state that  $f < g$  if  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$

For functions we are interested in, these define a weak ordering

# Weak ordering

Let  $f(n)$  and  $g(n)$  describe the run-time of two algorithms

- If  $f(n) \sim g(n)$ , then it is **always possible** to improve the performance of one function over the other by purchasing a faster computer
- If  $f(n) < g(n)$ , then you can **never** purchase a computer fast enough so that the second function always runs in less time than the first

# Some Assumptions

We will make some assumptions:

- Our functions will describe the time or memory required to solve a problem of size  $n$
- We are restricting ourselves to certain functions:
  - They are defined for  $n \geq 0$
  - They are strictly positive for all  $n$ 
    - In fact,  $f(n) > c$  for some value  $c > 0$
    - That is, any problem requires at least one instruction and byte
  - They are increasing (monotonic increasing)

# Landau Symbols

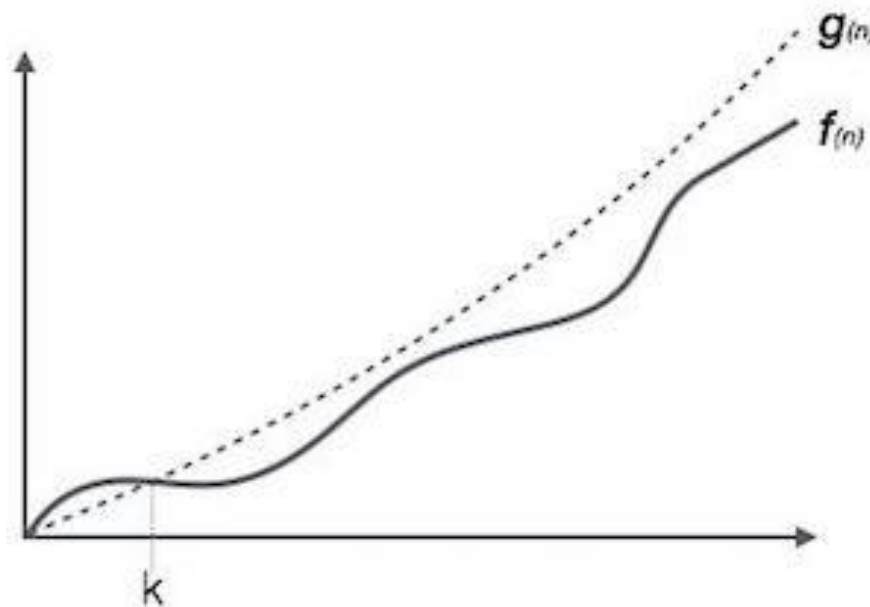
Better known as big O notation

A function  $f(n) = \mathbf{O}(g(n))$  if there exists  $k$  and  $c$  such that

$$f(n) < c g(n)$$

whenever  $n > k$

- The function  $f(n)$  has a rate of growth no greater than that of  $g(n)$





# Landau Symbols

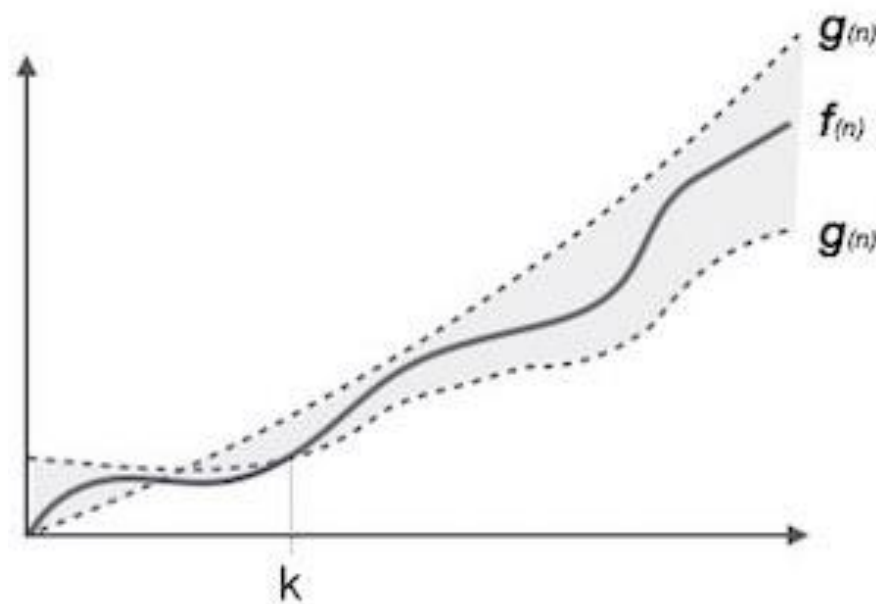
Another Landau symbol is  $\Theta$

A function  $f(n) = \Theta(g(n))$  if there exist positive  $k$ ,  $c_1$ , and  $c_2$  such that

$$c_1 g(n) < f(n) < c_2 g(n)$$

whenever  $n > k$

- The function  $f(n)$  has a rate of growth equal to that of  $g(n)$



# Landau Symbols

If  $f(n)$  and  $g(n)$  are polynomials of the **same degree** with positive leading coefficients:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c \quad \text{where} \quad 0 < c < \infty$$

From the definition, this means given  $c > \varepsilon > 0$  there

exists a  $k > 0$  such that  $\left| \frac{f(n)}{g(n)} - c \right| < \varepsilon$  whenever  $n > k$

That is,

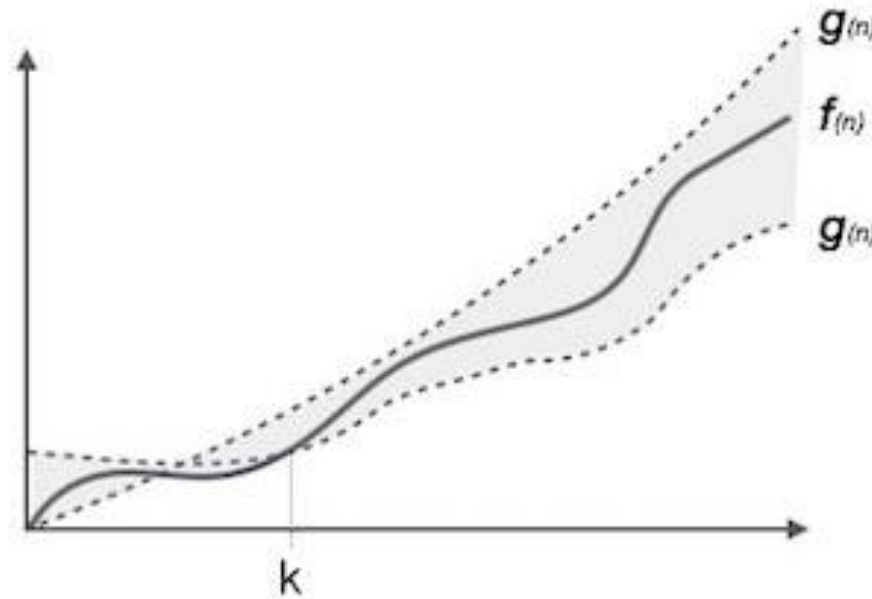
$$c - \varepsilon < \frac{f(n)}{g(n)} < c + \varepsilon$$

$$g(n)(c - \varepsilon) < f(n) < g(n)(c + \varepsilon)$$

# Landau Symbols

If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$  where  $0 < c < \infty$ , it follows that  $f(n) = \Theta(g(n))$

$$g(n)(c - \varepsilon) < f(n) < g(n)(c + \varepsilon)$$



# Landau Symbols

We have a similar definition for **O**:

If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$  where  $0 \leq c < \infty$ , it follows that  **$f(n) = O(g(n))$**

There are other possibilities we would like to describe:

If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ , we will say  **$f(n) = o(g(n))$**

- The function  $f(n)$  has a rate of growth less than that of  $g(n)$

We would also like to describe the opposite cases:

- The function  $f(n)$  has a rate of growth greater than that of  $g(n)$
- The function  $f(n)$  has a rate of growth greater than or equal to that of  $g(n)$

# Landau Symbols

We will at times use five possible descriptions

$$f(n) = \mathbf{o}(g(n)) \qquad \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = \mathbf{O}(g(n)) \qquad \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) = \mathbf{\Theta}(g(n)) \qquad 0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) = \mathbf{\Omega}(g(n)) \qquad \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$$

$$f(n) = \mathbf{\omega}(g(n)) \qquad \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$




# Landau Symbols

Graphically, we can summarize these as follows:

We say  $f(n) =$

$\mathbf{O}(g(n))$	$\mathbf{\Omega}(g(n))$
$\mathbf{o}(g(n))$	$\mathbf{\Theta}(g(n))$
$\mathbf{\omega}(g(n))$	

if  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} =$

		
$0$	$0 < c < \infty$	$\infty$

# Landau Symbols

For the functions we are interested in, it can be said that

$f(n) = \mathbf{O}(g(n))$  is equivalent to  $f(n) = \Theta(g(n))$  or  $f(n) = \mathbf{o}(g(n))$

and

$f(n) = \Omega(g(n))$  is equivalent to  $f(n) = \Theta(g(n))$  or  $f(n) = \omega(g(n))$

# Landau Symbols

Some other observations we can make are:

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

$$f(n) = \mathbf{O}(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$

$$f(n) = \mathbf{o}(g(n)) \Leftrightarrow g(n) = \omega(f(n))$$



# Big- $\Theta$ as an Equivalence Relation

If we look at the first relationship, we notice that  $f(n) = \Theta(g(n))$  seems to describe an equivalence relation:

1.  $f(n) = \Theta(g(n))$  if and only if  $g(n) = \Theta(f(n))$
2.  $f(n) = \Theta(f(n))$
3. If  $f(n) = \Theta(g(n))$  and  $g(n) = \Theta(h(n))$ , it follows that  $f(n) = \Theta(h(n))$

Consequently, we can group all functions into equivalence classes, where all functions within one class are big-theta  $\Theta$  of each other

# Big- $\Theta$ as an Equivalence Relation

For example, all of

$$\begin{array}{lll} n^2 & 100000 n^2 - 4 n + 19 & n^2 + 1000000 \\ 323 n^2 - 4 n \ln(n) + 43 n + 10 & & 42n^2 + 32 \\ & n^2 + 61 n \ln^2(n) + 7n + 14 \ln^3(n) + \ln(n) & \end{array}$$

are big- $\Theta$  of each other

$$E.g., 42n^2 + 32 = \Theta( 323 n^2 - 4 n \ln(n) + 43 n + 10 )$$

# Big- $\Theta$ as an Equivalence Relation

We will select just one element to represent the entire class of these functions:  $n^2$

- We could choose any function, but this is the simplest

# Big- $\Theta$ as an Equivalence Relation

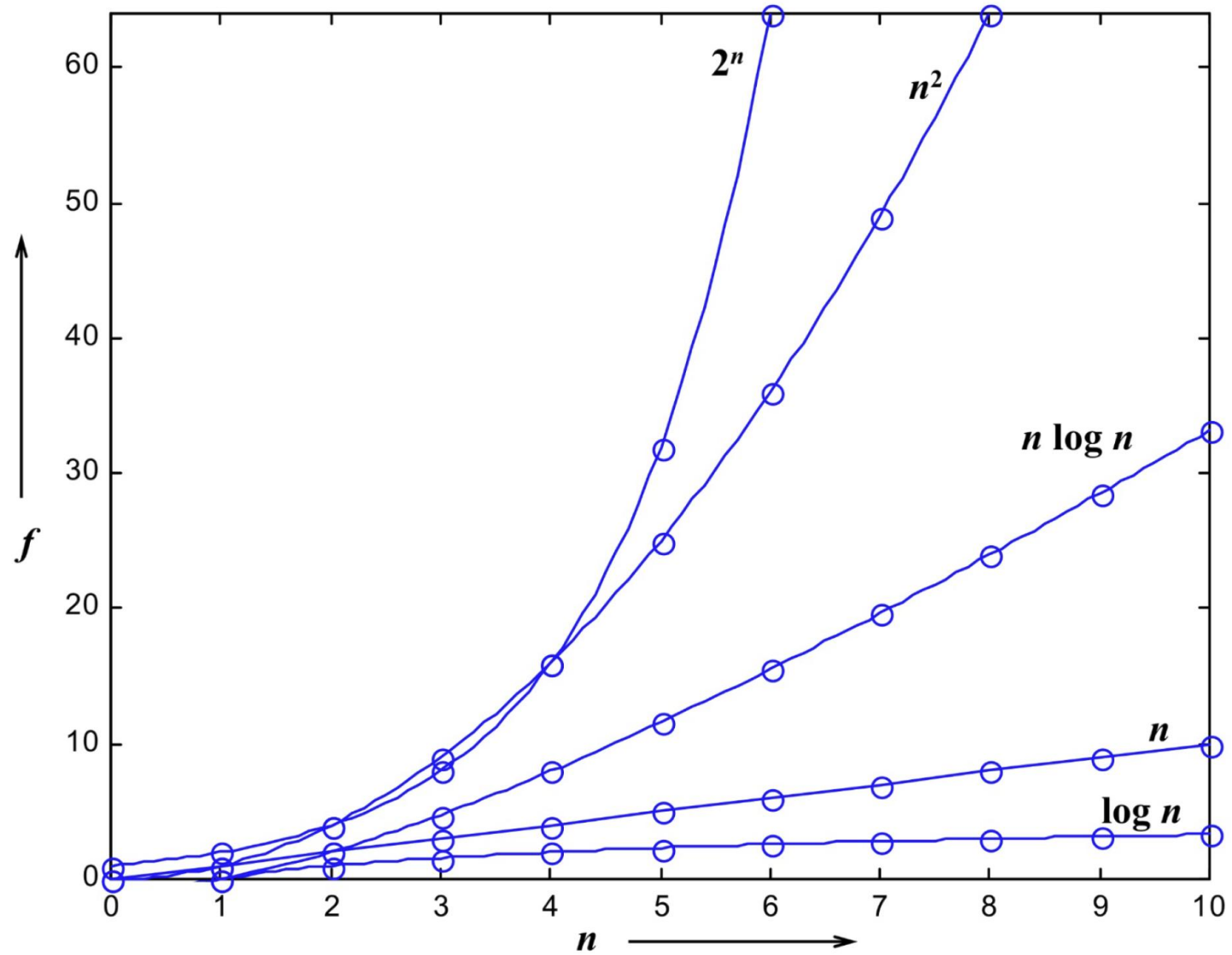
The most common classes are given names:

$\Theta(1)$	constant
$\Theta(\ln(n))$	logarithmic
$\Theta(n)$	linear
$\Theta(n \ln(n))$	“ $n \log n$ ”
$\Theta(n^2)$	quadratic
$\Theta(n^3)$	cubic
$2^n, e^n, 4^n, \dots$	exponential

# Empirical comparison

	1	2	4	8	16	32
<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>
<b><math>\log n</math></b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
<b><math>n</math></b>	<b>1</b>	<b>2</b>	<b>4</b>	<b>8</b>	<b>16</b>	<b>32</b>
<b><math>n \log n</math></b>	<b>0</b>	<b>2</b>	<b>8</b>	<b>24</b>	<b>64</b>	<b>160</b>
<b><math>n^2</math></b>	<b>1</b>	<b>4</b>	<b>16</b>	<b>64</b>	<b>256</b>	<b>1024</b>
<b><math>n^3</math></b>	<b>1</b>	<b>8</b>	<b>64</b>	<b>512</b>	<b>4096</b>	<b>32768</b>
<b><math>2^n</math></b>	<b>2</b>	<b>4</b>	<b>16</b>	<b>256</b>	<b>65536</b>	<b>4294967296</b>
<b><math>n !</math></b>	<b>1</b>	<b>2</b>	<b>24</b>	<b>40326</b>	<b>2092278988000</b>	<b><math>26313 \times 10^{33}</math></b>

# Empirical comparison plot



# Logarithms and Exponentials

Recall that all **logarithms are scalar multiples of each other**

- Therefore  $\log_b(n) = \Theta(\ln(n))$  for any base  $b$

On the other hand, there is no single equivalence class for exponential functions:

- If  $1 < a < b$ ,  $\lim_{n \rightarrow \infty} \frac{a^n}{b^n} = \lim_{n \rightarrow \infty} \left(\frac{a}{b}\right)^n = 0$
- Therefore  $a^n = o(b^n)$

But any exponentially growing function is almost universally undesirable to have!

# Logarithms and Exponentials

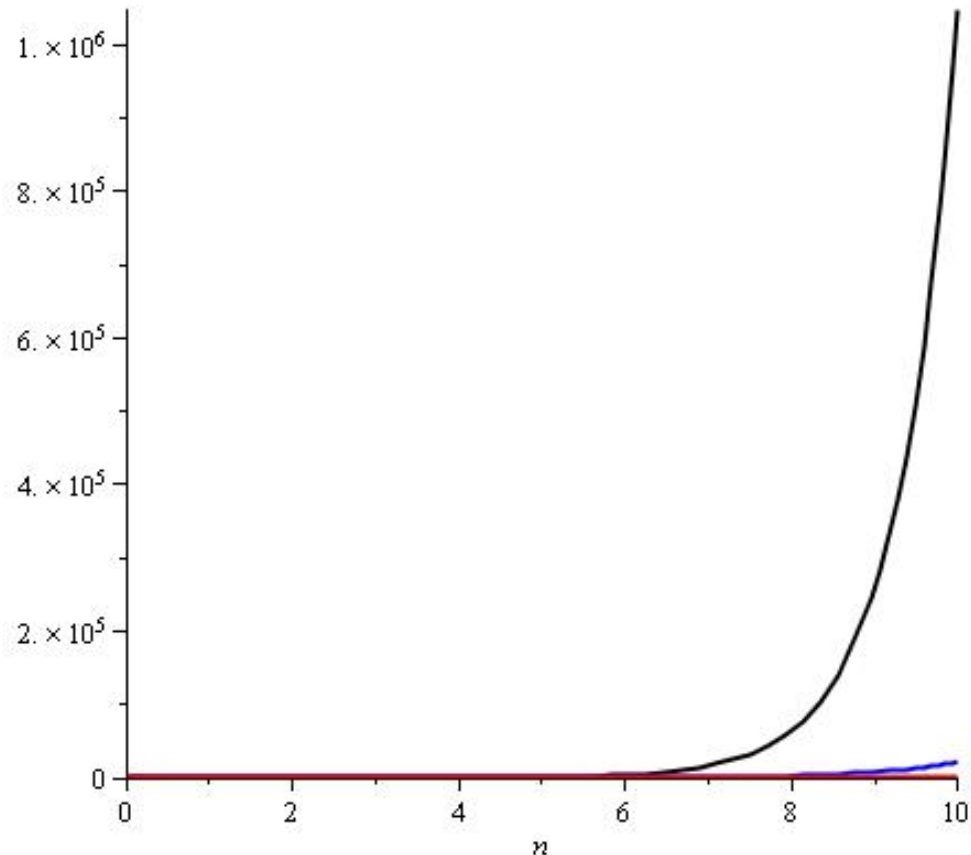
Plotting  $2^n$ ,  $e^n$ , and  $4^n$  on the range  $[1, 10]$  already shows how significantly different the functions grow

Note:

$$2^{10} = 1024$$

$$e^{10} \approx 22\,026$$

$$4^{10} = 1\,048\,576$$





# Little-o as a Weak Ordering

We can show that, for example

$$\ln(n) = o(n^p)$$

for any  $p > 0$

Proof: Using l'Hôpital's rule, we have

$$\lim_{n \rightarrow \infty} \frac{\ln(n)}{n^p} = \lim_{n \rightarrow \infty} \frac{1/n}{pn^{p-1}} = \lim_{n \rightarrow \infty} \frac{1}{pn^p} = \frac{1}{p} \lim_{n \rightarrow \infty} n^{-p} = 0$$

Conversely,  $1 = o(\ln(n))$

# Little-o as a Weak Ordering

If  $p$  and  $q$  are real positive numbers where  $p < q$

- It follows that  $n^p = o(n^q)$
- For example, matrix-matrix multiplication is  $\Theta(n^3)$  but a refined algorithm is  $\Theta(n^{\lg(7)})$  where  $\lg(7) \approx 2.81$
- Also,  $n^p = o(\ln(n)n^p)$ , but  $\ln(n)n^p = o(n^q)$ 
  - $n^p$  has a slower rate of growth than  $\ln(n)n^p$ , but
  - $\ln(n)n^p$  has a slower rate of growth than  $n^q$  for  $p < q$
  - Ex:  $n \ln n = o(n^{1.0000000000000001})$

# Little-o as a Weak Ordering

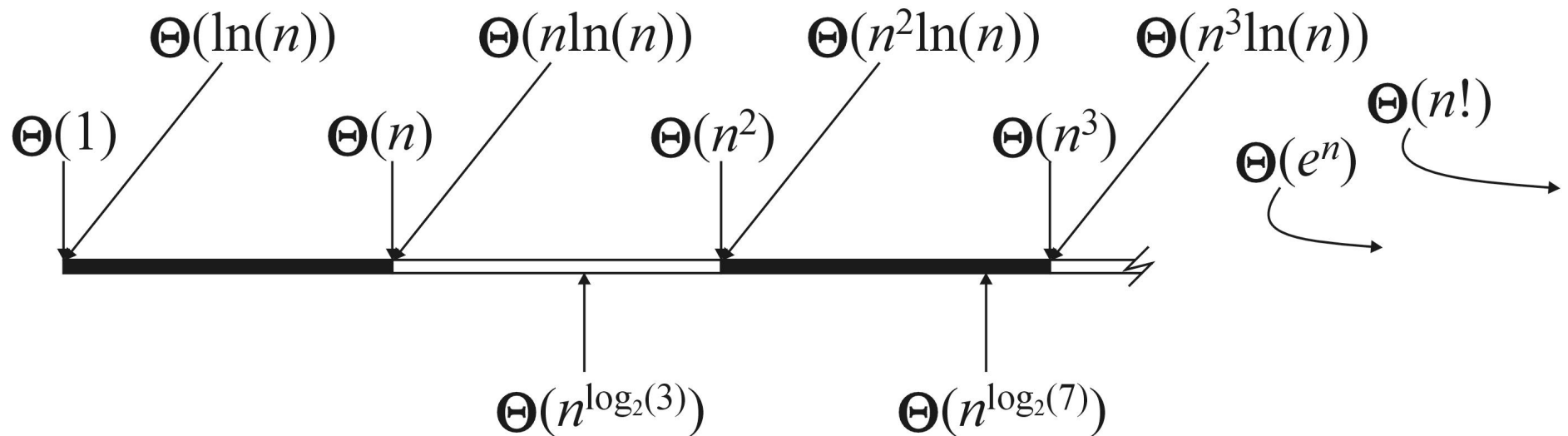
If we restrict ourselves to functions  $f(n)$  which are  $\Theta(n^p)$  and  $\Theta(\ln(n)n^p)$ , we note:

- It is never true that  $f(n) = o(f(n))$
- If  $f(n) \neq \Theta(g(n))$ , it follows that either
$$f(n) = o(g(n)) \text{ or } g(n) = o(f(n))$$
- If  $f(n) = o(g(n))$  and  $g(n) = o(h(n))$ , it follows that  $f(n) = o(h(n))$

This defines a weak ordering!

# Little-o as a Weak Ordering

Graphically, we can show this relationship by marking these against the real line



# Outline

- Justification for analysis
- Landau symbols
- Run time of programs
- Best-, worst-, and average-case

# Algorithms Analysis

The goal of algorithm analysis is to determine the asymptotic run time or memory requirements based on various parameters

We will use Landau symbols to describe the complexity of algorithms. E.g.,

- Given an array of size  $n$ :
  - Selection sort requires  $\Theta(n^2)$  time
  - Merge sort, quick sort, and heap sort all require  $\Theta(n \ln(n))$  time
- However:
  - Merge sort requires  $\Theta(n)$  additional memory
  - Quick sort requires  $\Theta(\ln(n))$  additional memory
  - Heap sort requires  $\Theta(1)$  additional memory

# Algorithms Analysis

An algorithm is said to have *polynomial time complexity* if its run-time may be described by  $O(n^d)$  for some fixed  $d \geq 0$

- We will consider such algorithms to be *efficient*

Problems that have no known polynomial-time algorithms are said to be *intractable*

- Traveling salesman problem: find the shortest path that visits  $n$  cities
- Best run time:  $\Theta(n^2 2^n)$

In general, you don't want to implement exponential-time or exponential-memory algorithms

# Algorithms Analysis

To properly investigate the determination of run times asymptotically:

- We will begin with machine instructions and basic operations
- Control statements
- Conditional-controlled loops
- Functions
- Recursive functions



# Machine Instructions

Given any processor, it is capable of performing only a limited number of operations

These operations are called *instructions*

The collection of instructions is called the *instruction set*

- The exact set of instructions differs between processors
- MIPS, ARM, x86, 6800, 68k

Any instruction runs in a fixed amount of time (an integral number of CPU cycles)

# Machine Instructions

Any instruction runs in a fixed amount of time (an integral number of CPU cycles)

An example on the Coldfire is:

0x06870000000F

which adds 15 to the 7<sup>th</sup> data register

As humans are not good at hex, this can be programmed in assembly language as

ADDI.L #15, D7

- More in ECE 222

# Machine Instructions

Assembly language has an almost one-to-one translation to machine instructions

- Assembly language is a low-level programming language

Other programming languages are higher-level:

Fortran, Pascal, Matlab, Java, C++, and C#

The adjective “high” refers to the level of abstraction:

- Java, C++, and C# have abstractions such as OO
- Matlab and Fortran have operations which do not map to relatively small number of machine instructions:

```
>> 1.27^2.9                % 1.27**2.9 in Fortran  
2.0000036616123606774
```

# Machine Instructions

The C programming language (C++ without objects and other abstractions) can be referred to as a mid-level programming language

- There is abstraction, but the language is closely tied to the standard capabilities
- There is a closer relationship between operators and machine instructions

Consider the operation **a** += **b**;

- Assume that the compiler has already has the value of the variable **a** in register **D1** and perhaps **b** is a variable stored at the location stored in address register **A1**, this is then converted to the single instruction

**ADD (A1) , D1**

# Operators

There is a close relationship between basic operations and machine instructions, so we may assume each operation requires a fixed number of CPU cycles, i.e.,  $\Theta(1)$  time:

- Variable assignment `=`
- Integer operations `+ - * / % ++ --`
- Logical operations `&& || !`
- Bitwise operations `& | ^ ~`
- Relational operations `== != < <= >= >`
- Memory allocation and deallocation `new delete`

# Operators

Of these, memory allocation and deallocation are the slowest by a significant factor

- A quick test on eceunix shows a factor of over 100
- They require communication with the operation system
- This does not account for the time required to call the constructor and destructor

Note that after memory is allocated, the constructor is run

- The constructor may not run in  $\Theta(1)$  time

# Blocks of Operations

Each operation runs in  $\Theta(1)$  time and therefore any fixed number of operations also run in  $\Theta(1)$  time, for example:

```
// Swap variables a and b  
int tmp = a;  
a = b;  
b = tmp;
```

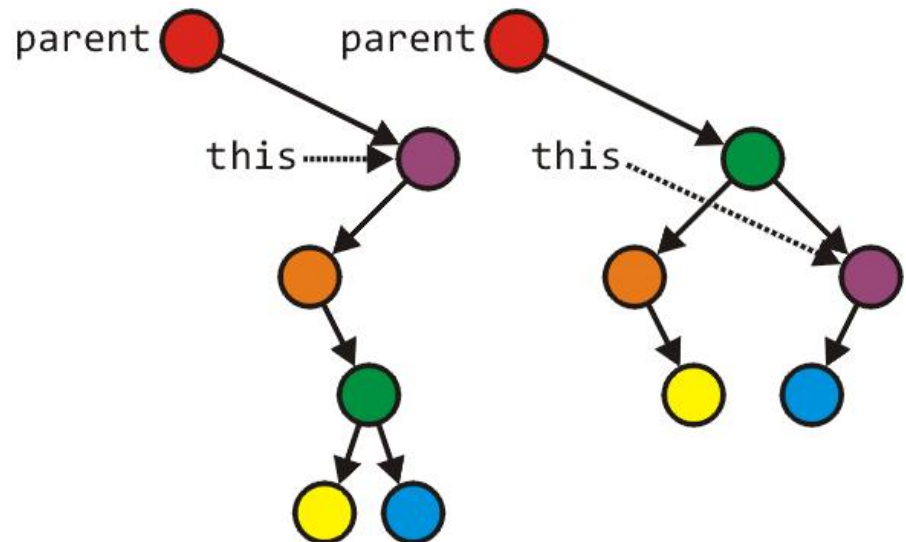
# Blocks of Operations

Seldom will you find large blocks of operations without any additional control statements

This example rearranges an AVL tree structure

```
Tree_node *lrl = left->right->left;  
Tree_node *lrr = left->right->right;  
parent = left->right;  
parent->left = left;  
parent->right = this;  
left->right = lrl;  
left = lrr;
```

Run time:  $\Theta(1)$





# Blocks in Sequence

Suppose you have now analyzed a number of blocks of code run in sequence

```
template <typename T>
void update_capacity( int delta ) {
    T *array_old = array;
    int capacity_old = array_capacity;
    array_capacity += delta;
    array = new T[array_capacity];

    for ( int i = 0; i < capacity_old; ++i ) {
        array[i] = array_old[i];
    }

    delete[] array_old;
}
```

$\Theta(1)$

$\Theta(n)$

$\Theta(1)$

To calculate the total run time, add the entries:  $\Theta(1 + n + 1) = \Theta(n)$

# Blocks in Sequence

```
template <int M, int N>
```

```
Matrix<M, N> &Matrix<M, N>::operator= ( Matrix<M, N> const &A ) {
```

```
    if ( &A == this ) {
```

```
        return *this;
```

$\Theta(1)$

$$\Theta(1 + 1 + \min(M, N) + M + n + 1) \\ = \Theta(M + n)$$

```
    }
```

```
    if ( capacity != A.capacity ) {
```

```
        delete [] column_index;
```

```
        delete [] off_diagonal;
```

```
        capacity = A.capacity;
```

```
        column_index = new int[capacity];
```

```
        off_diagonal = new double[capacity];
```

$\Theta(1)$

```
    }
```

```
    for ( int i = 0; i < minMN; ++i ) {
```

```
        diagonal[i] = A.diagonal[i];
```

$\Theta(\min(M, N))$

```
    }
```

```
    for ( int i = 0; i <= M; ++i ) {
```

```
        row_index[i] = A.row_index[i];
```

$\Theta(M)$

```
    }
```

```
    for ( int i = 0; i < A.size(); ++i ) {
```

```
        column_index[i] = A.column_index[i];
```

```
        off_diagonal[i] = A.off_diagonal[i];
```

$\Theta(n)$

```
    }
```

```
    return *this;
```

$\Theta(1)$

```
}
```

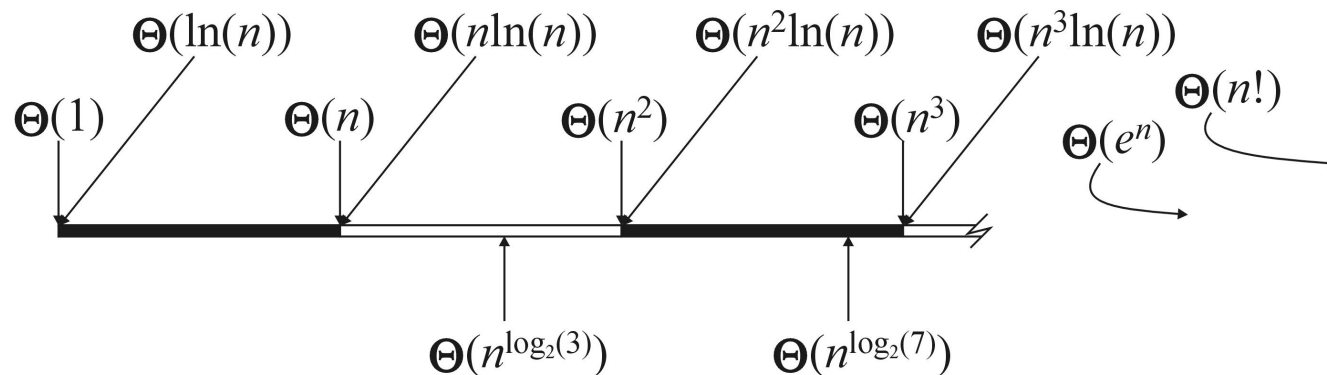
- Note that  $\min(M, N) \leq M$
- We cannot say anything about  $M$  and  $n$ . As a convention, we keep both.

# Blocks in Sequence

Other examples:

- Run three blocks of code which are  $\Theta(1)$ ,  $\Theta(n^2)$ , and  $\Theta(n)$   
Total run time  $\Theta(1 + n^2 + n) = \Theta(n^2)$
- Run two blocks of code which are  $\Theta(n \ln(n))$ , and  $\Theta(n^{1.5})$   
Total run time  $\Theta(n \ln(n) + n^{1.5}) = \Theta(n^{1.5})$

Recall this linear ordering from the previous topic



- When considering a sum, take the dominant term

# Blocks in Sequence

What if we have **both big O and big Theta**?

- if the leading term is big- $\Theta$ , then the result must be big- $\Theta$ , otherwise
- if the leading term is big- $O$ , we can say the result is big- $O$

For example,

$$O(n) + O(n^2) + O(n^4) = O(n + n^2 + n^4) = O(n^4)$$

$$O(n) + \Theta(n^2) = \Theta(n^2)$$

$$O(n^2) + \Theta(n) = O(n^2)$$

$$O(n^2) + \Theta(n^2) = \Theta(n^2)$$

# Control Statements

Next, we will look at the following control statements

These are statements which potentially alter the execution of instructions

- Conditional statements  
if, switch
- Condition-controlled loops  
for, while, do-while
- Count-controlled loops  
for i from 1 to 10 do ... end do;      # Maple
- Collection-controlled loops  
foreach ( int i in array ) { ... }      // C#

# Control Statements

Given any collection of nested control statements, it is always necessary to work inside out

- Determine the run times of the inner-most statements and work your way out

```
for(i=0; i<n; i++) {  
    // do something...  
    for(j=0; j<m; j++) {  
        // do something else...  
    }  
}
```

# Control Statements

Given

```
if ( condition ) {  
    // true body  
} else {  
    // false body  
}
```

The run time of a conditional statement is:

- the run time of the condition (the test), plus
- the run time of the body which is run

In most cases, the run time of the condition is  $\Theta(1)$

# Control Statements

In some cases, it is easy to determine which statement must be run:

```
int factorial ( int n ) {  
    if ( n == 0 ) {  
        return 1;  
    } else {  
        return n * factorial ( n - 1 );  
    }  
}
```



# Control Statements

In others, it is less obvious

- Find the maximum entry in an array:

```
int find_max( int *array, int n ) {  
    max = array[0];  
  
    for ( int i = 1; i < n; ++i ) {  
        if ( array[i] > max ) {  
            max = array[i];  
        }  
    }  
  
    return max;  
}
```

# Control Statements

If we had information about the distribution of the entries of the array, we may be able to determine it

- if the list is sorted (ascending) it will always be run
- if the list is sorted (descending) it will never be run
- if the list is randomly distributed, then??? We don't know.

# Control Statements

- Conditional

if C then S1 else S2

- Suppose you are doing a big O analysis

$\text{Time}(C) + \text{Max}(\text{Time}(S1), \text{Time}(S2))$  or

$\text{Time}(C) + \text{Time}(S1) + \text{Time}(S2)$  ?

# Conditional Statements

Consider this example

```
void Disjoint_sets::clear() {  
    if ( sets == n ) {  
        return;  
    }
```

$\Theta(1)$

```
    max_height = 0;  
    num_disjoint_sets = n;
```

$\Theta(1)$

```
    for ( int i = 0; i < n; ++i ) {  
        parent[i] = i;  
        tree_height[i] = 0;  
    }
```

$\Theta(n)$

```
}
```

$$T_{\text{clear}}(n) = \begin{cases} \cdot \cdot (1) & \text{sets} = n \\ \cdot \cdot (n) & \text{otherwise} \end{cases}$$

# Condition-controlled Loops

The C++ for loop is a condition controlled statement:

```
for ( int i = 0; i < N; ++i ) {  
    // ...  
}
```

is identical to

```
int i = 0;           // initialization  
while ( i < N ) {    // condition  
    // ...  
    ++i;            // increment  
}
```

# Condition-controlled Loops

The initialization, condition, and increment usually are single statements running in  $\Theta(1)$

```
for ( int i = 0; i < N; ++i ) {  
    // ...  
}
```

# Condition-controlled Loops

If the body does not depend on the variable (in this example,  $i$ ), then the run time of

```
for ( int i = 0; i < n; ++i ) {  
    // code which is Theta(f(n))  
}
```

is  $\Theta(n f(n))$

If the body is  $\mathbf{O}(f(n))$ , then the run time of the loop is  $\mathbf{O}(n f(n))$

# Condition-controlled Loops

For example,

```
int sum = 0;
for ( int i = 0; i < n; ++i ) {
    sum += 1;    // Theta(1)
}
```

This code has run time

$$\Theta(n \cdot 1) = \Theta(n)$$



# Condition-controlled Loops

Another example

```
int sum = 0;
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < n; ++j ) {
        sum += 1;      // Theta(1)
    }
}
```

The previous example showed that the inner loop is  $\Theta(n)$ , thus the outer loop is

$$\Theta(n \cdot n) = \Theta(n^2)$$

# Analysis of Repetition Statements

Suppose with each loop, we use a linear search an array of size  $m$ :

```
for ( int i = 0; i < n; ++i ) {  
    // search through an array of size m  
    // O( m );  
}
```

The inner loop is  $O(m)$  and thus the outer loop is

$O(\textcolor{red}{n} m)$

# Condition-controlled Loops

Another example

```
int sum = 0;
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < i; ++j ) {
        sum += i + j;
    }
}
```

The inner loop is  $\Theta(i)$ , hence the outer is

$$\Theta\left(\sum_{i=0}^{n-1} i\right) = \Theta\left(\frac{n(n-1)}{2}\right) = \Theta(n^2)$$

# Analysis of Repetition Statements

Final example:

```
int sum = 0;
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < i; ++j ) {
        for ( int k = 0; k < j; ++k ) {
            sum += i + j + k;
        }
    }
}
```

From inside to out:

$\Theta(1)$

$\Theta(j)$

$\Theta(i^2)$

$\Theta(n^3)$

# Control Statements

Switch statements appear to be nested if statements:

```
switch( i ) {  
    case 1:  /* do stuff */ break;  
    case 2:  /* do other stuff */ break;  
    case 3:  /* do even more stuff */ break;  
    case 4:  /* well, do stuff */ break;  
    case 5:  /* tired yet? */ break;  
    default: /* do default stuff */  
}
```

# Control Statements

Thus, a switch statement would appear to run in  $O(n)$  time where  $n$  is the number of cases, the same as nested if statements

- Why then not use:

```
if ( i == 1 ) { /* do stuff */ }  
else if ( i == 2 ) { /* do other stuff */ }  
else if ( i == 3 ) { /* do even more stuff */ }  
else if ( i == 4 ) { /* well, do stuff */ }  
else if ( i == 5 ) { /* tired yet? */ }  
else { /* do default stuff */ }
```

# Control Statements

Question:

Why would you introduce something into programming language which is redundant?

There are reasons for this:

- your name is Larry Wall and you are creating the Perl (**not** PERL) programming language
- you are introducing software engineering constructs, for example, classes

# Control Statements

However, switch statements were included in the original C language... why?

First, you may recall that the cases must be actual values, either:

- integers
- characters

For example, you cannot have a case with a variable, e.g.,

```
case n: /* do something */ break; //bad
```



# Control Statements

The compiler looks at the different cases and calculates an appropriate jump

For example, assume:

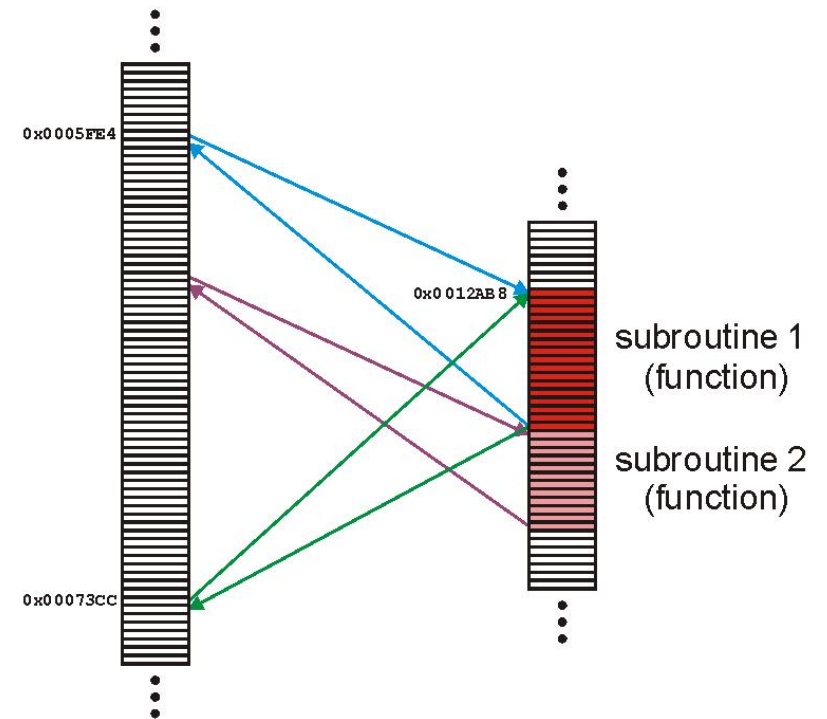
- the cases are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
- each case requires a maximum of 24 bytes (for example, six instructions)

Then the compiler simply makes a jump size based on the variable, jumping ahead either 0, 24, 48, 72, ..., or 240 instructions

# Functions

A function (or subroutine) is code which has been separated out:

- repeated operations
  - e.g., mathematical functions
- to group related tasks
  - e.g., initialization



# Functions

Because a subroutine (function) can be called from anywhere, we must:

- prepare the appropriate environment
- deal with arguments (parameters)
- jump to the subroutine
- execute the subroutine
- deal with the return value
- clean up

We will assume that the overhead required to make a function call and to return is  $\Theta(1)$ .

# Functions

Fortunately, this is such a common task that all modern processors have instructions that perform most of these steps in one instruction

Thus, we will assume that the overhead required to make a function call and to return is  $\Theta(1)$  and

- We will discuss this later (stacks/ECE 222)

# Functions

Because any function requires the overhead of a function call and return, we will always assume that

$$T_f = \Omega(1)$$

That is, it is impossible for any function call to have a zero run time

# Functions

Given a function  $f(n)$  (the run time of which depends on  $n$ ) we will associate the run time of  $f(n)$  by some function  $T_f(n)$

- We may write this as  $T(n)$
- This includes the time required to both call and return from the function

# Functions

Consider this function:

```
void Disjoint_sets::set_union( int m, int n ) {
```

```
    m = find( m );
```

```
    n = find( n );
```

```
    if ( m == n ) {
```

```
        return;
```

```
    }
```

```
    --num_disjoint_sets;
```

```
    if ( tree_height[m] >= tree_height[n] ) {
```

```
        parent[n] = m;
```

```
        if ( tree_height[m] == tree_height[n] ) {
```

```
            ++( tree_height[m] );
```

```
            max_height = std::max( max_height, tree_height[m] );
```

```
        }
```

```
    } else {
```

```
        parent[m] = n;
```

```
    }
```

```
}
```

$$T_{\text{set\_union}} = 2T_{\text{find}} + \Theta(1)$$

$2T_{\text{find}}$

$\Theta(1)$

# Recursive Functions

A function is relatively simple (and boring) if it simply performs operations and calls other functions

Most interesting functions designed to solve problems usually end up calling themselves

- Such a function is said to be *recursive*



# Recursive Functions

As an example, we could implement the factorial function recursively:

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

$$T_1(1)$$

$$T_1(n-1) + T_1(1)$$

# Recursive Functions

The analysis of the run time of this function yields a recurrence relation:

$$T_1(n) = T_1(n - 1) + \Theta(1) \quad T_1(1) = \Theta(1)$$

This recurrence relation has Landau symbols...

- Replace each Landau symbol with a representative function:

$$T_1(n) = T_1(n - 1) + 1 \quad T_1(1) = 1$$

- Then it is easy to prove that  $T_1(n) = \Theta(n)$

# Recursive Functions

Suppose we want to sort a array of  $n$  items

We could:

- go through the list and find the largest item
- swap the last entry in the list with that largest item
- then, go on and sort the rest of the array

This is called *selection sort*

# Recursive Functions

```
void sort( int * array, int n ) {  
    if ( n <= 1 ) {  
        return;                // special case: 0 or 1 items are always sorted  
    }  
  
    int posn = 0;               // assume the first entry is the smallest  
    int max = array[posn];  
  
    for ( int i = 1; i < n; ++i ) { // search through the remaining entries  
        if ( array[i] > max ) { // if a larger one is found  
            posn = i;           // update both the position and value  
            max = array[posn];  
        }  
    }  
  
    int tmp = array[n - 1];      // swap the largest entry with the last  
    array[n - 1] = array[posn];  
    array[posn] = tmp;  
  
    sort( array, n - 1 );       // sort everything else  
}
```

# Recursive Functions

We could call this function as follows:

```
int array[7] = {5, 8, 3, 6, 2, 4, 7};  
sort( array, 7 ); // sort an array of seven items
```

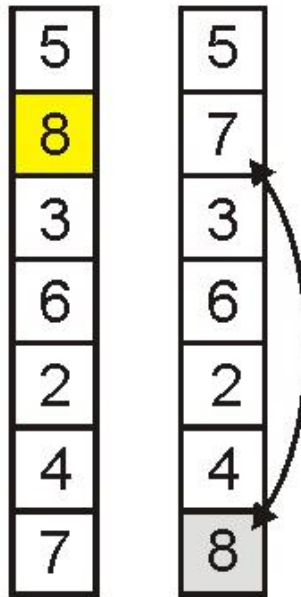
array

5
8
3
6
2
4
7

# Recursive Functions

The first call finds the largest element

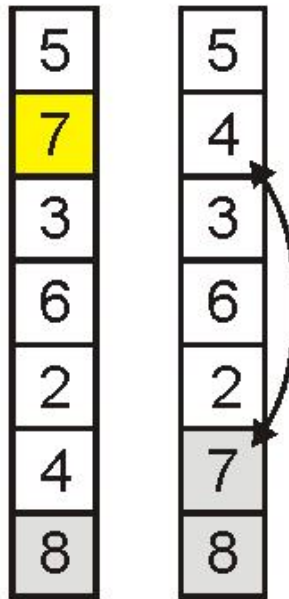
```
sort( array, 7 )
```



# Recursive Functions

The next call finds the 2<sup>nd</sup>-largest element

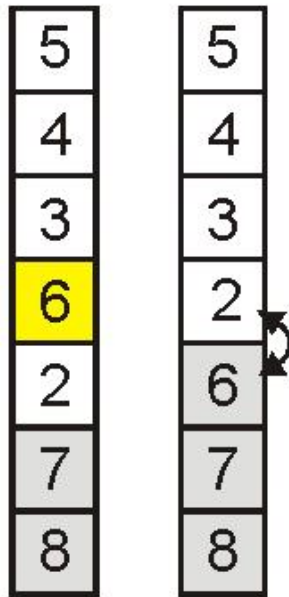
```
sort( array, 6 )
```



# Recursive Functions

The third finds the 3<sup>rd</sup>-largest

```
sort( array, 5 )
```

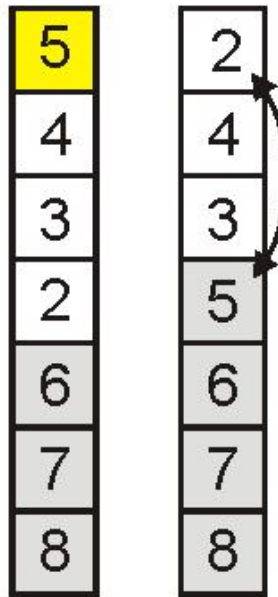




# Recursive Functions

And the 4<sup>th</sup>

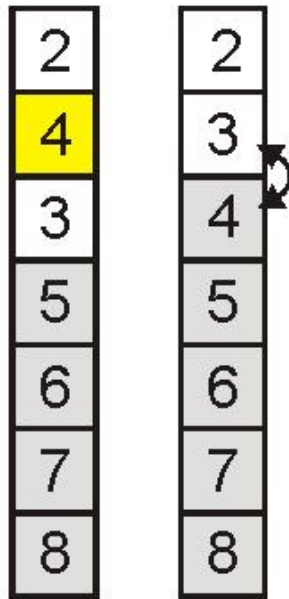
```
sort( array, 4 )
```



# Recursive Functions

And the 5<sup>th</sup>

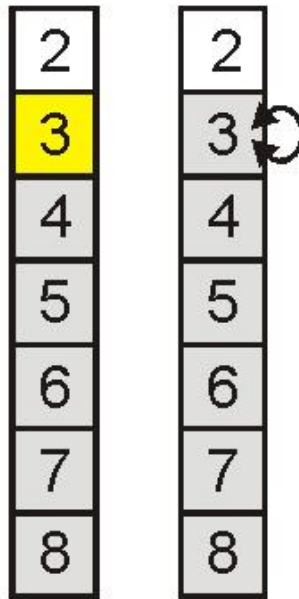
```
sort( array, 3 )
```



# Recursive Functions

Finally the 6<sup>th</sup>

```
sort( array, 2 )
```



# Recursive Functions

And the array is sorted:

```
sort( array, 1 )
```

2
3
4
5
6
7
8

# Recursive Functions

Analyzing the function, we get:

```
void sort( int * array, int n ) {  
    if ( n <= 1 ) {  
        return;  
    }  
  
    int posn = 0;  
    int max = array[posn];  
  
    for ( int i = 1; i < n; ++i ) {  
        if ( array[i] > max ) {  
            posn = i;  
            max = array[posn];  
        }  
    }  
  
    int tmp = array[n - 1];  
    array[n - 1] = array[posn];  
    array[posn] = tmp;  
  
    sort( array, n - 1 );  
}
```

Diagram illustrating the time complexity analysis of the recursive function:

- $T(0) = T(1) = \Theta(1)$  (Base case)
- $\Theta(1)$  (Initialization of `posn` and `max`)
- $\Theta(1)$  (Inner loop body)
- $\Theta(n)$  (Outer loop)
- $\Theta(1)$  (Swapping elements)
- $T(n-1)$  (Recursive call)

Recurrence relation:

$$T(n) = \Theta(1) + \Theta(n) + \Theta(1) + T(n-1)$$
$$= T(n-1) + \Theta(n)$$

# Recursive Functions

Thus, replacing each Landau symbol with a representative, we are required to solve the recurrence relation

$$T(n) = T(n - 1) + n \quad T(1) = 1$$

The easy way to solve this is with Maple:

```
> rsolve( {T(n) = T(n - 1) + n, T(1) = 1}, T(n) );
```

$$-1 - n + (n + 1) \left( \frac{n}{2} + 1 \right)$$

```
> expand( % );
```

$$\frac{1}{2}n + \frac{1}{2}n^2$$

# Recursive Functions

Consequently, the sorting routine has the run time

$$T(n) = \Theta(n^2)$$

To see this by hand, consider the following

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= (T(n-2) + (n-1)) + n \\ &= T(n-2) + n + (n-1) \\ &= T(n-3) + n + (n-1) + (n-2) \\ &\vdots \\ &= T(1) + \sum_{i=2}^n i = 1 + \sum_{i=2}^n i = \sum_{i=1}^n i = \frac{n(n+1)}{2} \end{aligned}$$

# Recursive Functions

Now consider binary search of a sorted list:

- Check the middle entry
- If we do not find it, check either the left- or right-hand side, as appropriate

Thus,  $T(n) = T((n - 1)/2) + \Theta(1)$

Also, if  $n = 1$ , then  $T(1) = \Theta(1)$



# Recursive Functions

Thus we have to solve:

$$T(n) = \begin{cases} 1 & n = 1 \\ T\left(\frac{n-1}{2}\right) + 1 & n > 1 \end{cases}$$

Assume  $n = 2^k - 1$  where  $k$  is an integer

Then  $(n - 1)/2 = (2^k - 1 - 1)/2 = 2^{k-1} - 1$

# Recursive Functions

Thus, we can write

$$\begin{aligned}T(n) &= T(2^k - 1) \\&= T\left(\frac{2^k - 1 - 1}{2}\right) + 1 \\&= T(2^{k-1} - 1) + 1 \\&= T\left(\frac{2^{k-1} - 1 - 1}{2}\right) + 1 + 1 \\&= T(2^{k-2} - 1) + 2 \\&\vdots\end{aligned}$$

# Recursive Functions

Notice the pattern with one more step:

$$\begin{aligned} &= T(2^{k-1} - 1) + 1 \\ &= T\left(\frac{2^{k-1} - 1 - 1}{2}\right) + 1 + 1 \\ &= T(2^{k-2} - 1) + 2 \\ &= T(2^{k-3} - 1) + 3 \\ &\vdots \end{aligned}$$

# Recursive Functions

Thus, in general, we may deduce that after  $k - 1$  steps:

$$\begin{aligned}T(n) &= T(2^k - 1) \\&= T(2^{k-(k-1)} - 1) + k - 1 \\&= T(1) + k - 1 = k\end{aligned}$$

because  $T(1) = 1$

# Recursive Functions

Thus,  $T(n) = k$ , but  $n = 2^k - 1$

Therefore  $k = \lg(n + 1)$

Recall that  $f(n) = \Theta(g(n))$  if  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$  for  $0 < c < \infty$

$$\lim_{n \rightarrow \infty} \frac{\lg(n+1)}{\ln(n)} = \lim_{n \rightarrow \infty} \frac{\frac{1}{(n+1)\ln(2)}}{\frac{1}{n}} = \lim_{n \rightarrow \infty} \frac{n}{(n+1)\ln(2)} = \frac{1}{\ln(2)}$$

Thus,  $T(n) = \Theta(\lg(n + 1)) = \Theta(\ln(n))$

# Outline

- Justification for analysis
- Landau symbols
- Run time of programs
- Best-, worst-, and average-case

# Cases

When determining the run time of an algorithm, because the data may not be deterministic, we may be interested in:

- Best-case run time
- Average-case run time
- Worst-case run time

In many cases, these will be significantly different

# Cases

Searching a list linearly is simple enough

We will count the number of comparisons

- Best case:
  - The first element is the one we're looking for:  $O(1)$
- Worst case:
  - The last element is the one we're looking for, or it is not in the list:  $O(n)$
- Average case?
  - We need some information about the list...



# Cases

Assume the item we are looking for is in the list and equally likely distributed

If the list is of size  $n$ , then there is a  $1/n$  chance of it being in the  $i$ th location

Thus, we sum

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

which is  $\mathbf{O}(n)$

# Cases

Suppose we have a different distribution:

- there is a 50% chance that the element is the first
- for each subsequent element, the probability is reduced by  $\frac{1}{2}$

We could write:

$$\sum_{i=1}^n \frac{i}{2^i} < \sum_{i=1}^{\infty} \frac{i}{2^i} = 2$$

which is  $\mathbf{O}(1)$

# Cases

- Best-case run time
  - Not so useful
- Average-case run time
  - Need to choose a distribution over input instances
  - Average-case analysis may tell us more about the choice of distributions than about the algorithm itself.
- Worst-case run time
  - Most widely used to capture efficiency in practice.
  - Draconian view, but hard to find effective alternative.
  - Exceptions: some worst-case exponential-time algorithms are widely used because the worst-case instances seem to be rare.
    - E.g., the simplex algorithm

# Cases

Previously, we had an example where we were looking for the number of times a particular assignment statement was executed:

```
int find_max( int * array, int n ) {  
    max = array[0];  
  
    for ( int i = 1; i < n; ++i ) {  
        if ( array[i] > max ) {  
            max = array[i];  
        }  
    }  
  
    return max;  
}
```

# Cases

This example is taken from Preiss

- The best case was once (first element is largest)
- The worst case was  $n$  times

For the average case, we must consider:

- What is the probability that the  $i^{\text{th}}$  object is the largest of the first  $i$  objects?

# Cases

To consider this question, we must assume that elements in the array are evenly distributed

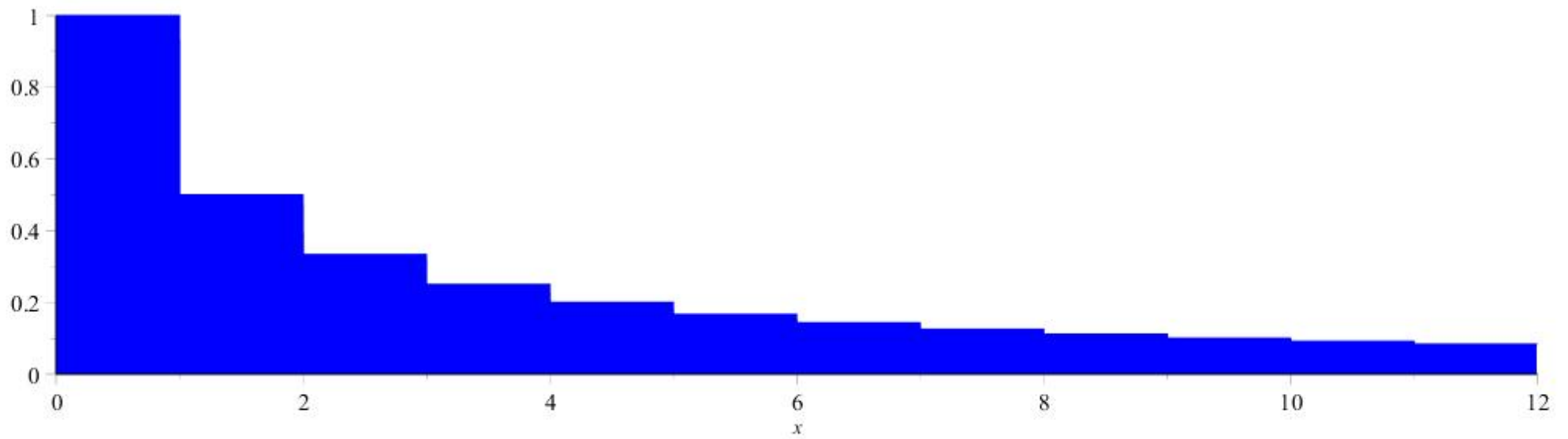
Thus, given a sub-list of size  $k$ , the probability that any one element is the largest is  $1/k$

Thus, given a value  $i$ , there are  $i + 1$  objects, hence

$$\sum_{i=0}^{n-1} \frac{1}{i+1} = \sum_{i=1}^n \frac{1}{i} = ?$$

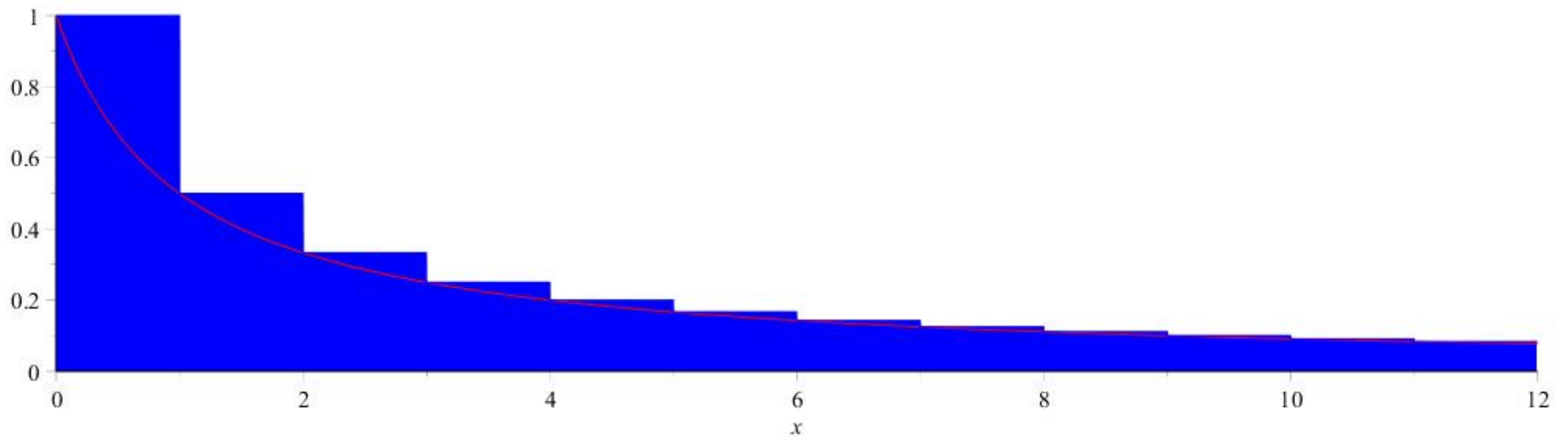
# Cases

We can approximate the sum by an integral – what is the area under:



# Cases

We can approximate this by the  $1/(x+1)$  integrated from 0 to  $n$





# Cases

From calculus:

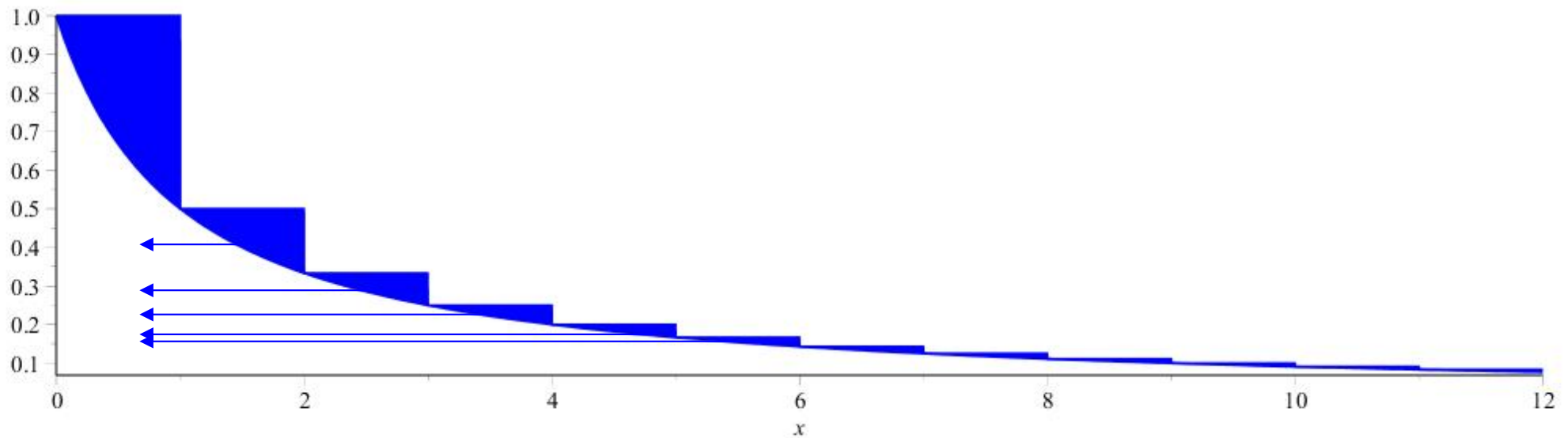
$$\int_0^n \frac{1}{x+1} dx = \int_1^{n+1} \frac{1}{x} dx = \ln(x) \Big|_1^{n+1} = \ln(n+1) - \ln(1) = \ln(n+1)$$

How about the error? Our approximation would be useless if the error was  $\mathbf{O}(n)$

# Cases

Consider the following image which highlights the errors

- The errors can be *fit* into the box  $[0, 1] \times [0, 1]$

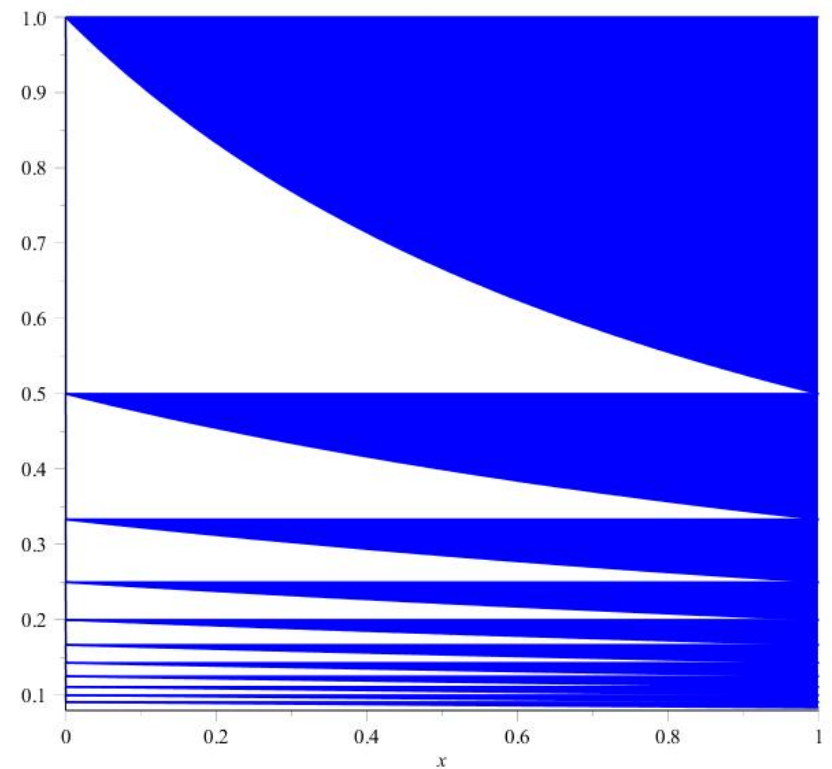


# Cases

Consequently, the error must be  $< 1$

In fact, it converges to  $\gamma \approx 0.57721566490$

– Therefore, the error is  $\theta(1)$



# Cases

Thus, the number of times that the assignment statement will be executed, assuming an even distribution is  $\mathbf{O}(\ln(n))$

# Cases

Thus, the total run of:

```
int find_max( int *array, int n ) {  
    max = array[0];  
  
    for ( int i = 1; i < n; ++i ) {  
        if ( array[i] > max ) {  
            max = array[i];  
        }  
    }  
  
    return max;  
}
```

is  $\therefore \left( 1 + \sum_{i=1}^{n-1} \left( 1 + \frac{1}{i+1} \right) \right) = \therefore (1 + n + \ln(n)) = \therefore (n)$

# Summary

- Justification for analysis
- Landau symbols
  - $o$   $O$   $\Theta$   $\Omega$   $\omega$
- Run time of programs
  - Basic operations
  - Control statements
  - Conditional-controlled loops
  - Functions
  - Recursive functions
- Best-, worst-, and average-case