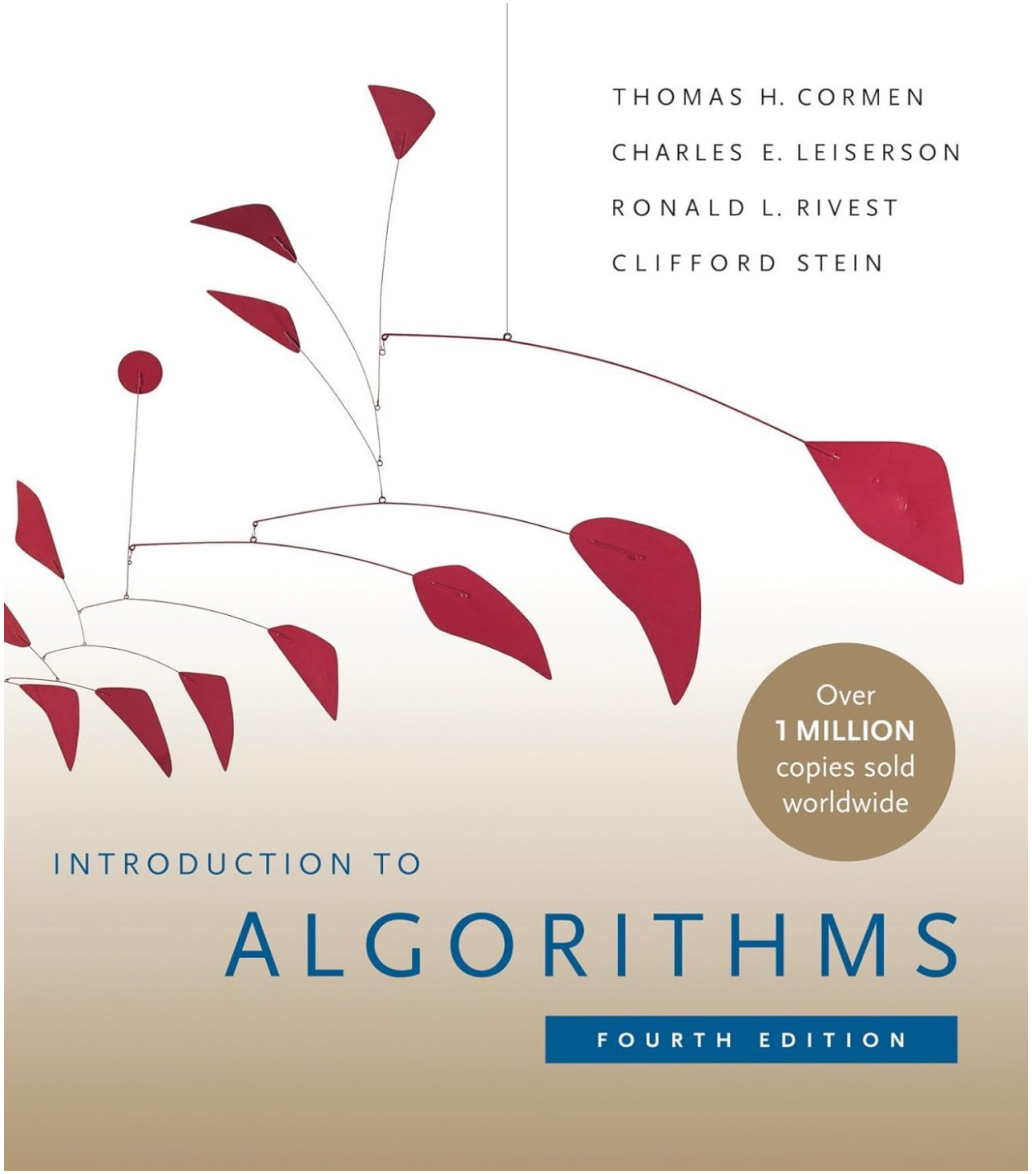
Introduction to Algorithms

Introduction

The Course

- Purpose: an introduction to the design and analysis of algorithms
  - Not a math course
  - Implementation the algorithms!
- Textbook: *Introduction to Algorithms*, Cormen, Leiserson, Rivest, Stein
  - An excellent reference you should own

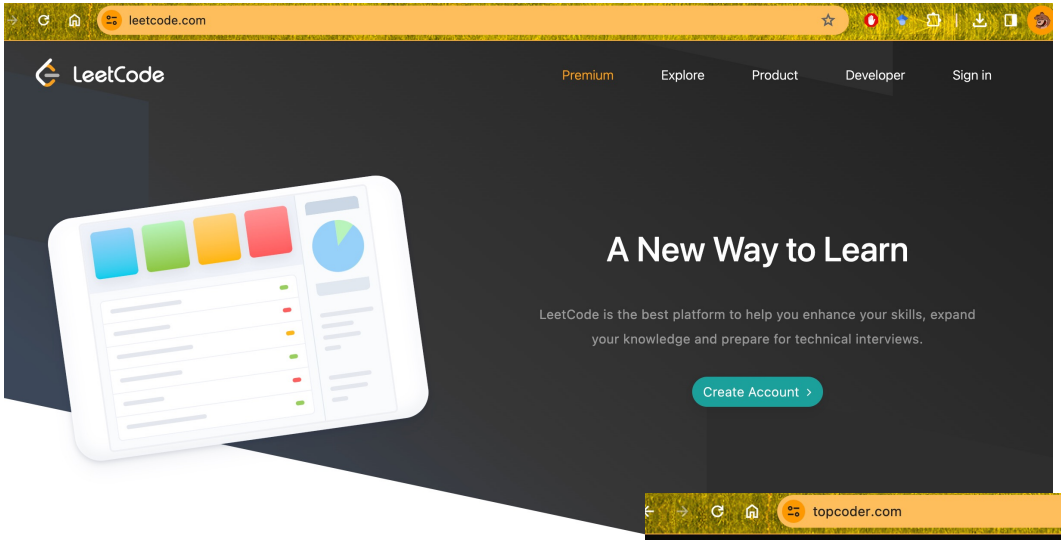
Textbook


THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN
Over
1 MILLION
copies sold
worldwide
INTRODUCTION TO
ALGORITHMS
FOURTH EDITION

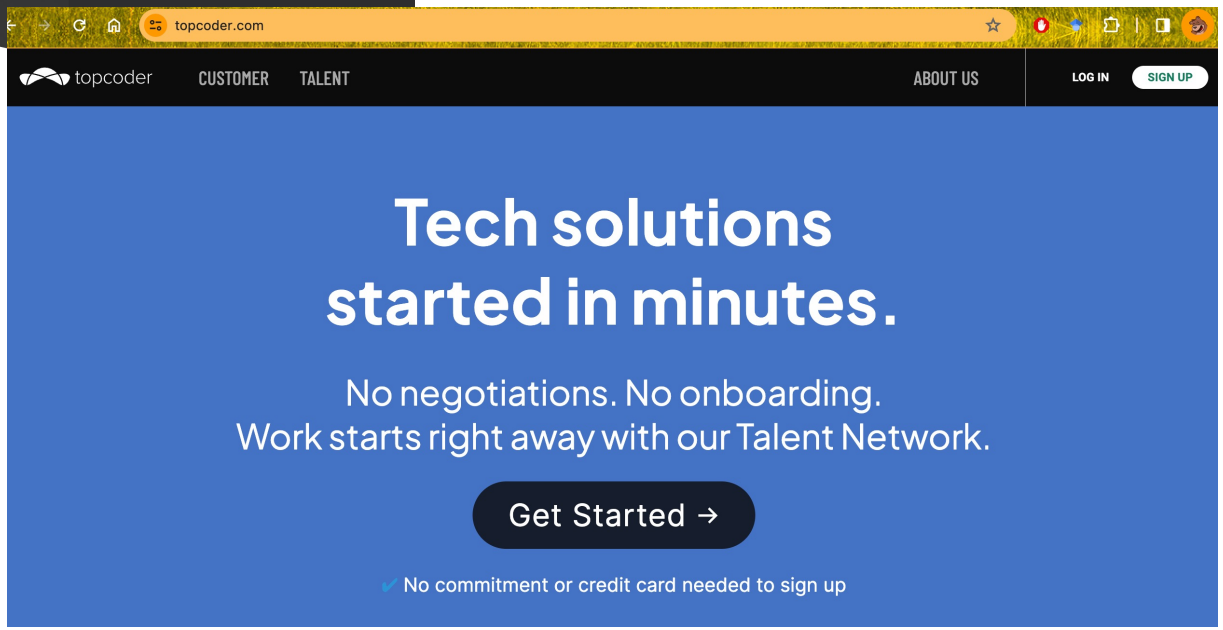
C++ / Data Structure

- C++ Primer
- Python
- Stack Overflow: https://stackoverflow.com/
- https://www.geeksforgeeks.org/introduction-to-algorithms/

Online Programming Learning

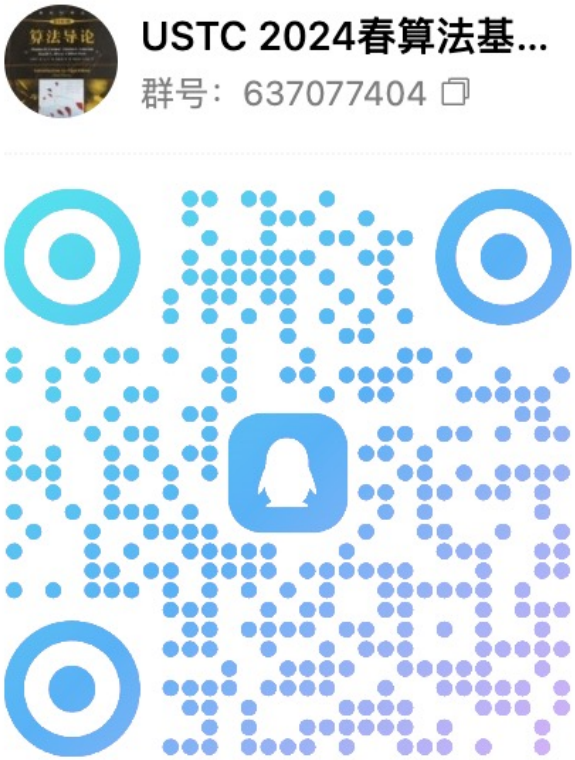


Start Exploring



The Course

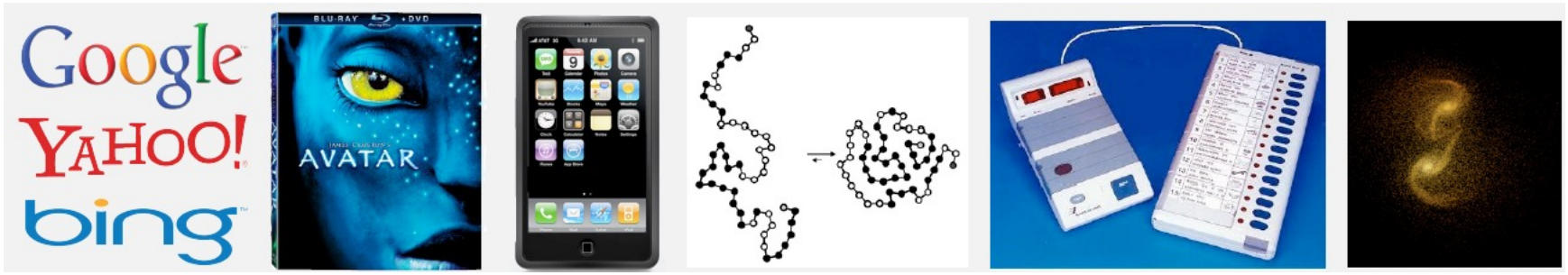
- Grading policy:
  - Homework&Programming: 40%
  - Final: 60%
- Online Judge:
  - https://oj.algustc.com/
- QQ Group: 637077404、
- 两位助教：
  - 陈纳川
  - 王鹏翔



Why study algorithms?

Their impact is broad and far-reaching.

- Internet. Web search, packet routing, distributed file sharing, ...
- Biology. Human genome project, protein folding, …
- Computers. Circuit layout, file system, compilers, …
- Computer graphics. Movies, video games, virtual reality, …
- Security. Cell phones, e-commerce, voting machines, …
- Multimedia. MP3, JPG, DivX, HDTV, face recognition, …
- Social networks. Recommendations, news feeds, advertisements, …
- Physics. N-body simulation, particle collision simulation, …
- ……



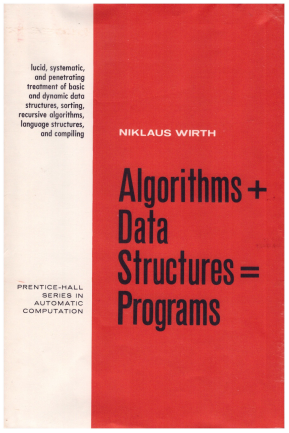
Why study algorithms?

To become a proficient programmer.

" I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships. "

— Linus Torvalds (creator of Linux)

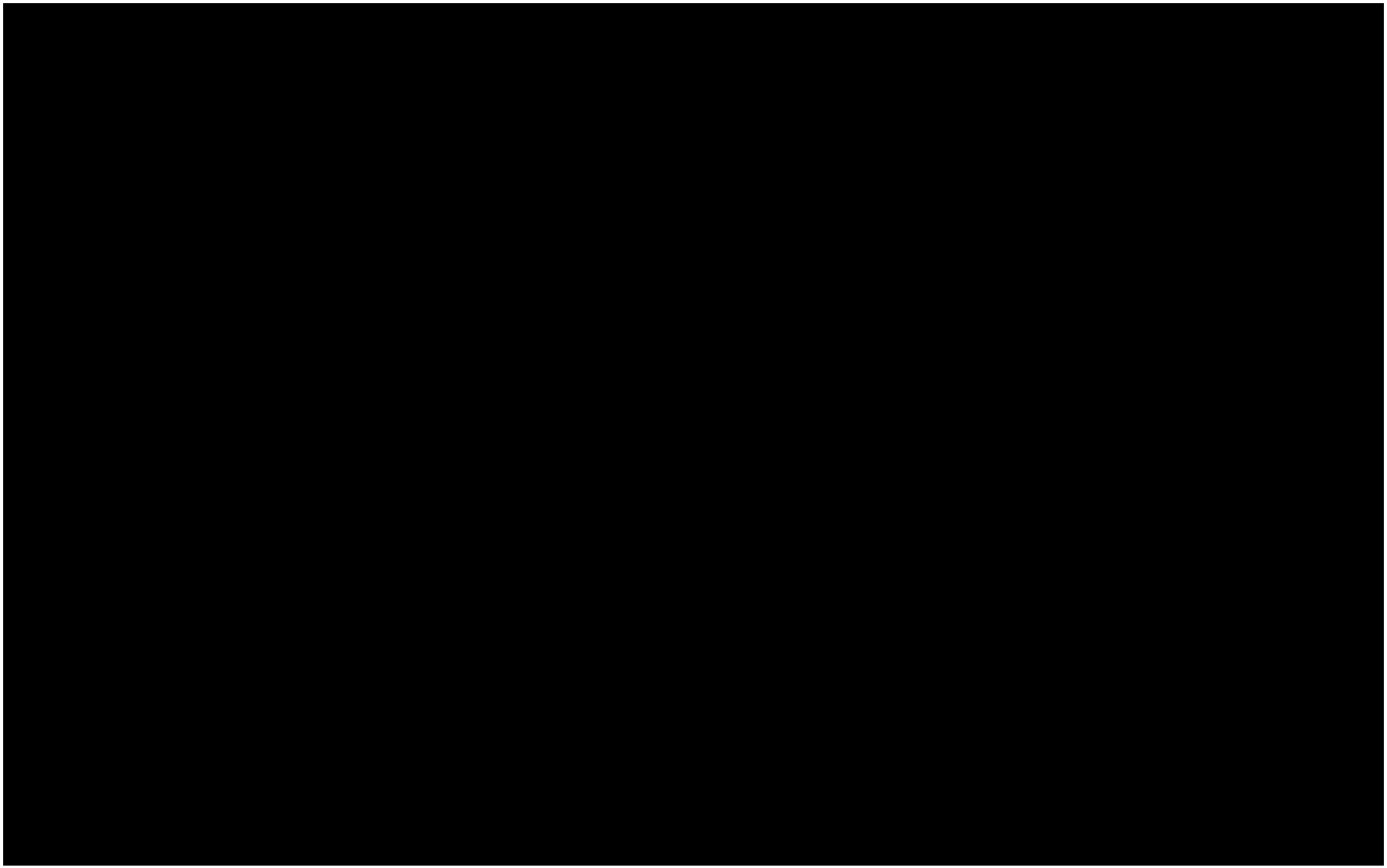
" Algorithms + Data Structures = Programs. " — Niklaus Wirth



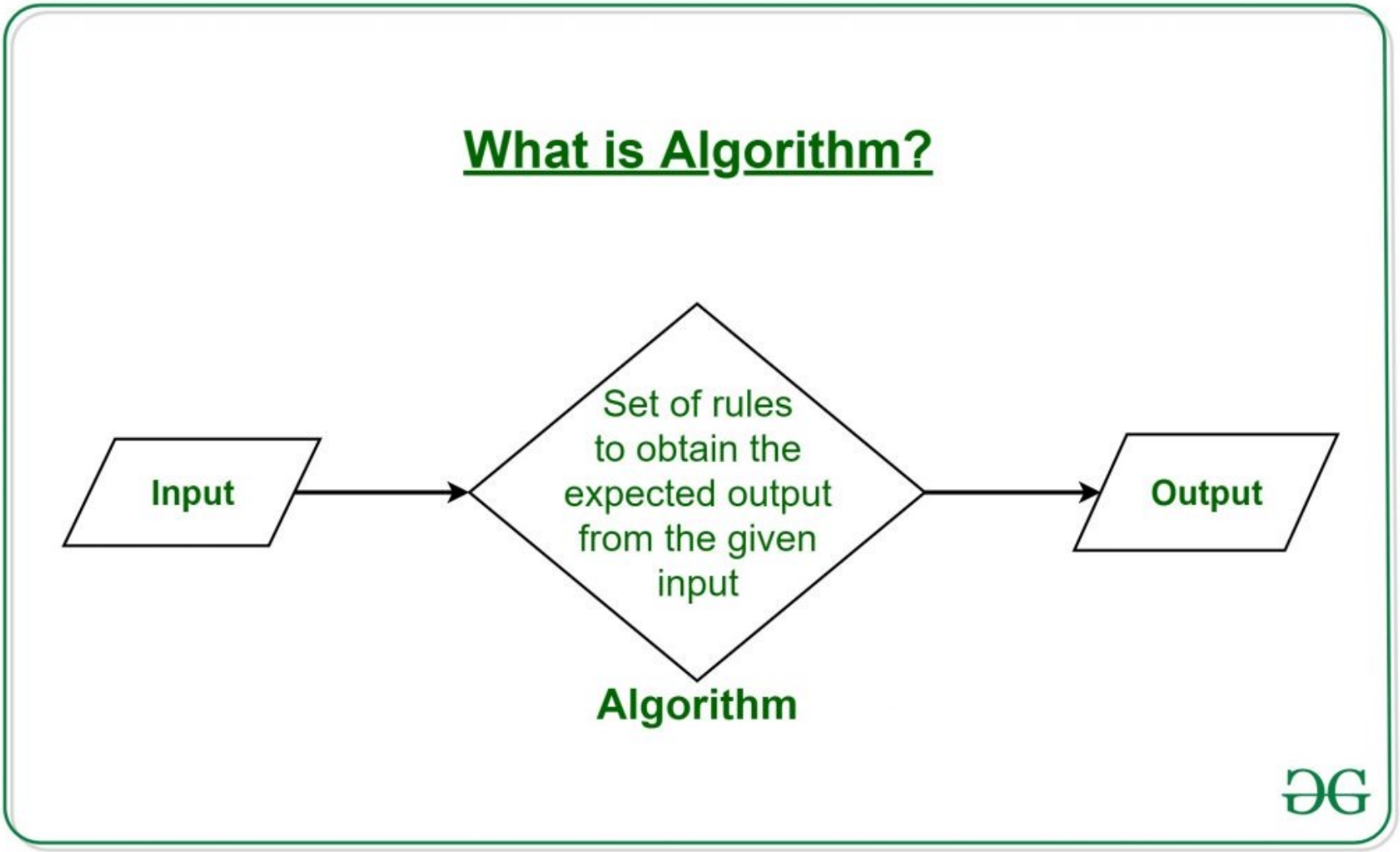
Why study algorithms?

For fun and profit.

What is an algorithm and why should you care



Algorithm Definition



Algorithm Definition

- A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.
  - From Google
- An algorithm is a specification of a behavioral process. It consists of a finite set of instructions that govern behavior step-by-step
  - Shackelford, Russell L. in Introduction to Computing and Algorithms

Notice

- Notice the term finite. Algorithms should lead to an eventual solution.
- Step by step process. Each step should do one logical action.

Algorithms

- Algorithms are addressed to some audience. Consider:
  - A set of instructions for building a child's bicycle.
  - A program for class scheduling.
  - An algorithm that will run on a computer system to calculate student GPA's.

Good vs. Bad Algorithms

- All algorithms will have input, perform a process, and produce output.
- A good algorithm should be:
  - Simple - *relative*
  - Complete – account for all inputs & cases
  - Correct (Right)
  - should have appropriate levels of Abstraction. – *grouping steps into a single module*
  - Precise
  - Mnemonic - SCRAP

Precision

- Precision means that there is only one way to interpret the instruction. Unambiguous
- Words like “maybe”, “sometimes” and “occasionally” have no business in a well developed algorithm.
- Instead of “maybe”, we can specify the exact circumstances in which an action will be carried out.

Simplicity

- Simple can be defined as having no **unnecessary** steps and no **unnecessary** complexity. (*You may lose points if your algorithm contains unnecessary steps*)
- Each step of a well developed algorithm should carry out one logical step of the process.
  - Avoid something like: "Take 2nd right *after* you exit at King Street"

Algorithm Design Process

- Analyze the problem and develop the specification.
- Design the solution
  - Test the solution as part of the design steps.
- Implement the program (code the program)
- Test the program
- Validate the program (further extensive testing) to insure it works under all circumstances.

Induction

- Suppose
  - S(k) is true for fixed constant k
    - Often k = 0
  - S(n) ⇒ S(n+1) for all n >= k
- Then S(n) is true for all n >= k

Proof By Induction

- Claim:S(n) is true for all n >= k
- Basis:
  - Show formula is true when n = k
- Inductive hypothesis:
  - Assume formula is true for an arbitrary n
- Step:
  - Show that formula is then true for n+1

Induction Example: Gaussian Closed Form

- Prove $1 + 2 + 3 + \ldots + n = n(n+1)/2$
  - Basis:
    - If $n = 0$, then $0 = 0(0+1)/2$
  - Inductive hypothesis:
    - Assume $1 + 2 + 3 + \ldots + n = n(n+1)/2$
  - Step (show true for n+1):

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

    $= n(n+1)/2 + n+1 = [n(n+1) + 2(n+1)]/2$

    $= (n+1)(n+2)/2 = (n+1)(n+1 + 1)/2$

Induction Example: Geometric Closed Form

- Prove $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for all $a \neq 1$
  - Basis: show that $a^0 = (a^{0+1} - 1)/(a - 1)$

    $a^0 = 1 = (a^1 - 1)/(a - 1)$
  - Inductive hypothesis:
    - Assume $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$
  - Step (show true for n+1):

    $a^0 + a^1 + \ldots + a^{n+1} = a^0 + a^1 + \ldots + a^n + a^{n+1}$

    $= (a^{n+1} - 1)/(a - 1) + a^{n+1} = (a^{n+1+1} - 1)/(a - 1)$

Induction

- We've been using *weak induction*
- *Strong induction* also holds
  - Basis: show S(0)
  - Hypothesis: assume S(k) holds for arbitrary k <= n
  - Step: Show S(n+1) follows
- Another variation:
  - Basis: show S(0), S(1)
  - Hypothesis: assume S(n) and S(n+1) are true
  - Step: show S(n+2) follows

Asymptotic Performance

- In this course, we care most about *asymptotic performance*
  - How does the algorithm behave as the problem size gets very large?
    - Running time
    - Memory/storage requirements
    - Bandwidth/power requirements.

Analysis of Algorithms

- Analysis is performed with respect to a computational model
- We will usually use a generic uniprocessor random-access machine (RAM)
  - All memory equally expensive to access
  - No concurrent operations
  - All reasonable instructions take unit time
    - Except, of course, function calls

Reasons to analyze algorithms

- Predict performance
- Compare algorithms
- Provide guarantees
- Understand theoretical basis

Their impact is broad and far-reaching: *avoid performance bugs.*

client gets poor performance because programmer did not understand performance characteristics

Input Size

- Time and space complexity
  - This is generally a function of the input size
    - E.g., sorting, multiplication
  - How we characterize input size depends:
    - Sorting: number of input items
    - Multiplication: total number of bits
    - Graph algorithms: number of nodes & edges
    - Etc

Running Time

- Number of primitive steps that are executed
  - Except for time of executing a function call most statements roughly require the same amount of time
    - y = m * x + b
    - c = 5 / 9 * (t - 32)
    - z = f(x) + g(y)
- We can be more exact if need be

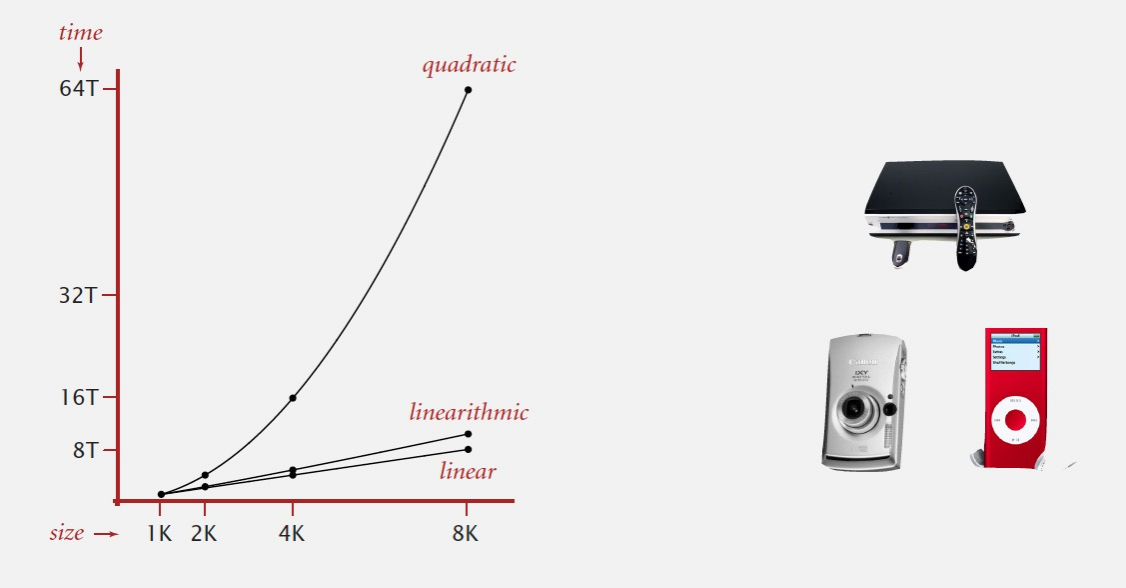
Analysis

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee
- Average case
  - Provides the expected running time
  - Very useful, but treat with care: what is "average" ?
    - Random (equally likely) inputs
    - Real-life inputs

Some algorithmic successes

Discrete Fourier transform.

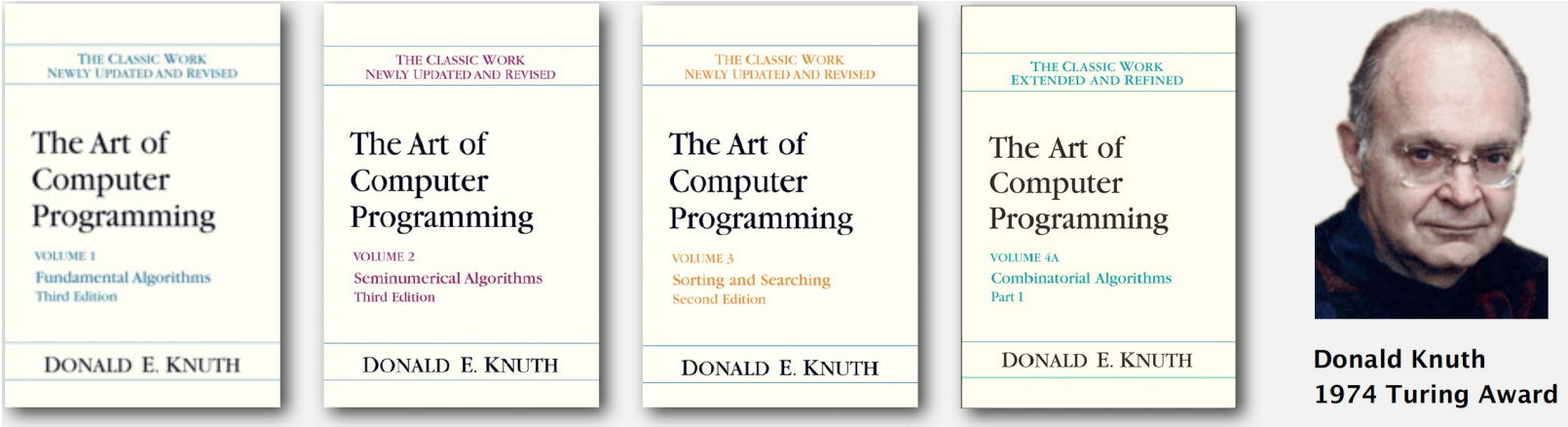
- Break down waveform of N samples into components.
- Applications: DVD, JPEG, MRI, astrophysics, …
- Brute force: N^2 steps.
- FFT algorithm: Nlog N steps, enables new technology.



Mathematical models for running time

Total running time: *sum of cost $\times$ frequency for all operations.*

- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.



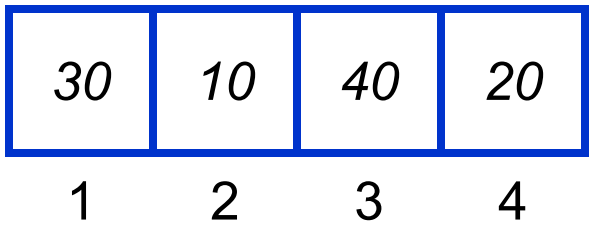
http://product.dangdang.com/1190486604.html

In principle, accurate mathematical models are available.

An Example: Insertion Sort

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

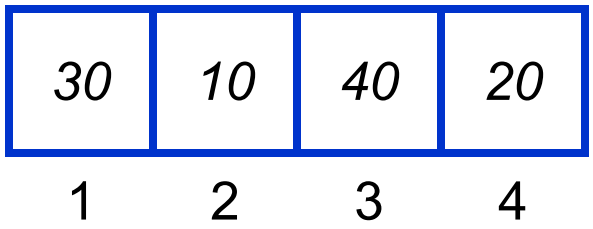


$i = \emptyset \quad j = \emptyset \quad key = \emptyset$

$A[j] = \emptyset \qquad A[j+1] = \emptyset$

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

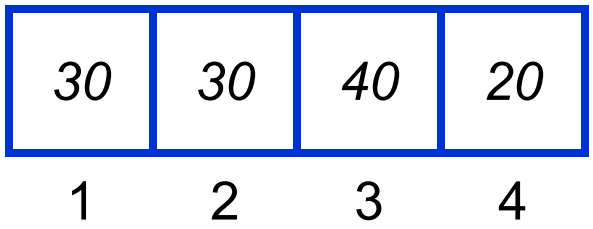
An Example: Insertion Sort



i = 2	j = 1	key = 10
A[j] = 30		A[j+1] = 10

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

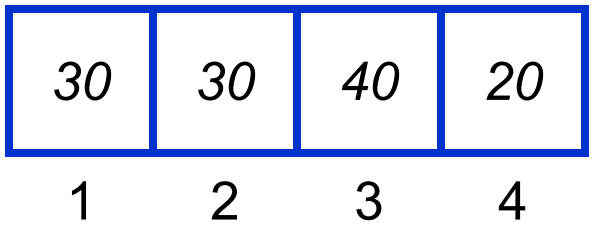
An Example: Insertion Sort



i = 2	j = 1	key = 10
A[j] = 30		A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

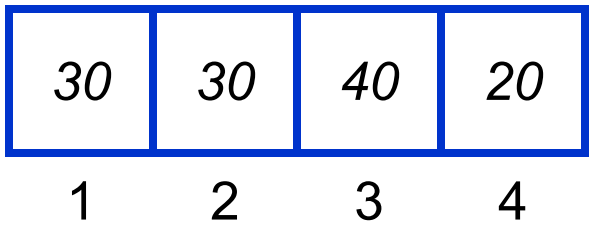
An Example: Insertion Sort



i = 2	j = 1	key = 10
A[j] = 30		A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

An Example: Insertion Sort

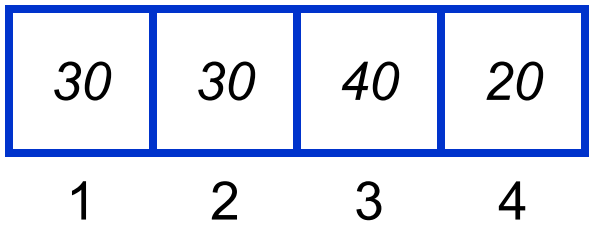


$i = 2 \quad j = 0 \quad key = 10$

$A[j] = \emptyset \qquad A[j+1] = 30$

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

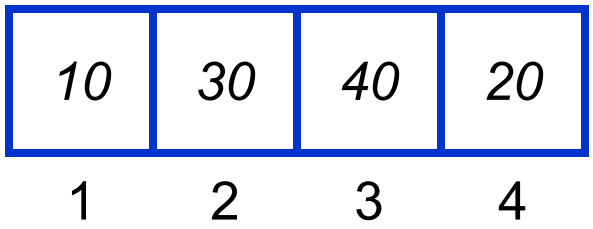
An Example: Insertion Sort



$i = 2$	$j = 0$	$key = 10$
$A[j] = \emptyset$		$A[j+1] = 30$

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

An Example: Insertion Sort

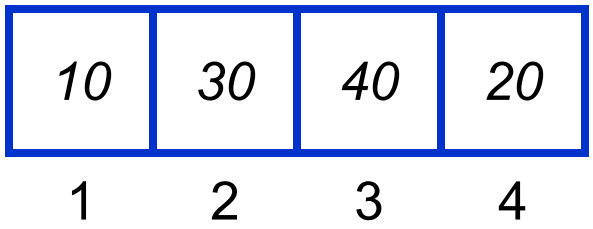


$i = 2 \quad j = 0 \quad key = 10$

$A[j] = \emptyset \qquad A[j+1] = 10$

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

An Example: Insertion Sort

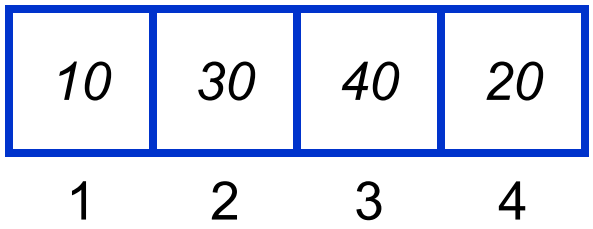


$i = 3 \quad j = 0 \quad key = 10$

$A[j] = \emptyset \qquad A[j+1] = 10$

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

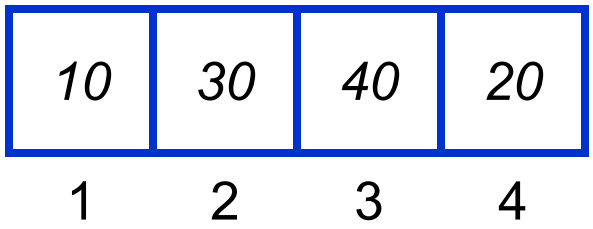
An Example: Insertion Sort



$i = 3$	$j = 0$	$key = 40$
$A[j] = \emptyset$		$A[j+1] = 10$

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

An Example: Insertion Sort

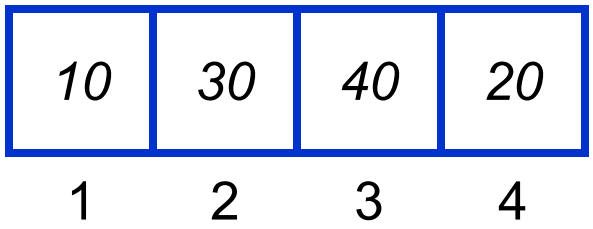


$i = 3 \quad j = 0 \quad key = 40$

$A[j] = \emptyset \qquad A[j+1] = 10$

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

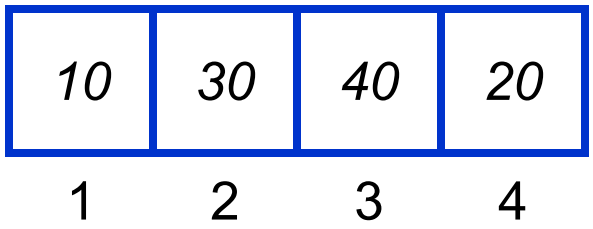
An Example: Insertion Sort



i = 3	j = 2	key = 40
A[j] = 30		A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

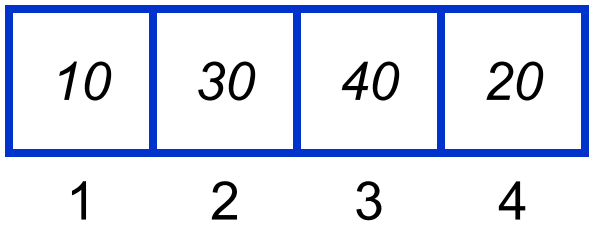
An Example: Insertion Sort



i = 3	j = 2	key = 40
A[j] = 30		A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

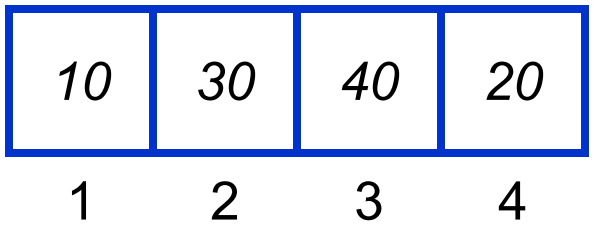
An Example: Insertion Sort



i = 3	j = 2	key = 40
A[j] = 30		A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

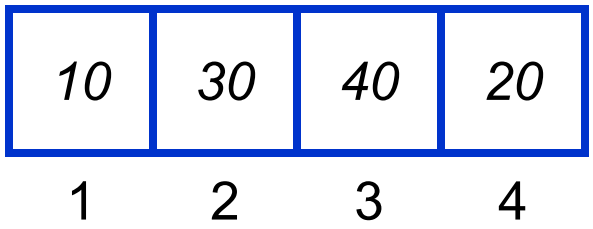
An Example: Insertion Sort



i = 4 j = 2	key = 40
A[j] = 30	A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

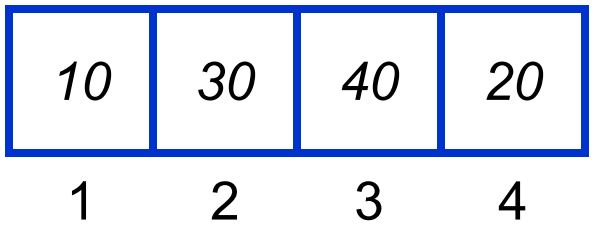
An Example: Insertion Sort



i = 4 j = 2	key = 20
A[j] = 30	A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

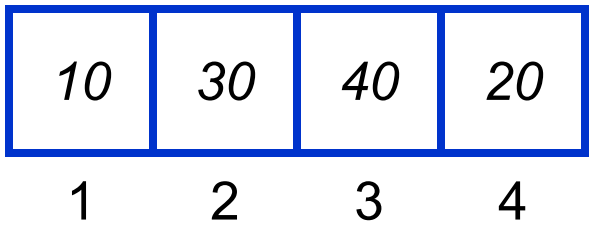
An Example: Insertion Sort



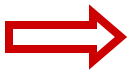
i = 4	j = 2	key = 20
A[j] = 30		A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

An Example: Insertion Sort

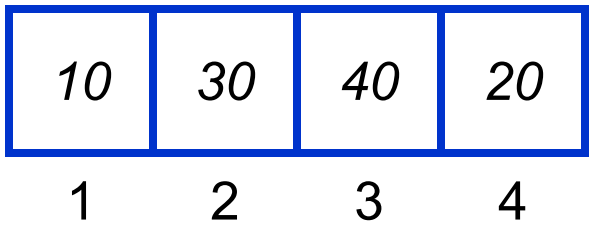


i = 4	j = 3	key = 20
A[j] = 40		A[j+1] = 20



```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

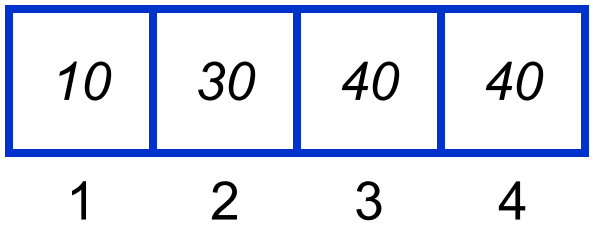
An Example: Insertion Sort



i = 4	j = 3	key = 20
A[j] = 40		A[j+1] = 20

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

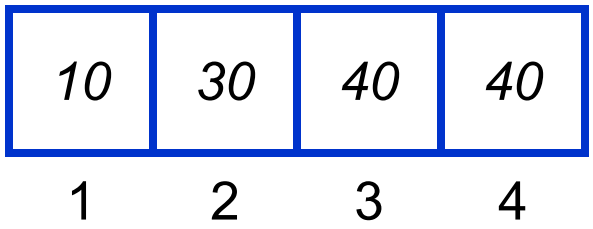
An Example: Insertion Sort



i = 4	j = 3	key = 20
A[j] = 40		A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

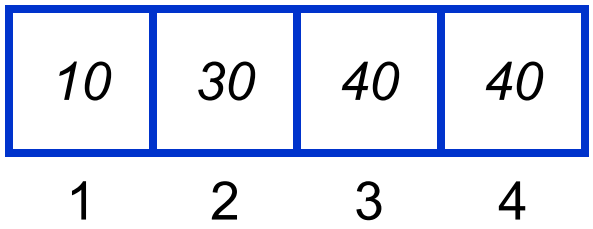
An Example: Insertion Sort



i = 4	j = 3	key = 20
A[j] = 40		A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

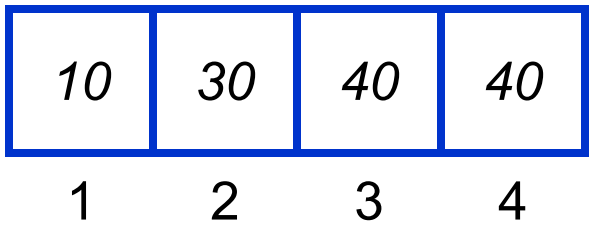
An Example: Insertion Sort



i = 4	j = 3	key = 20
A[j] = 40		A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

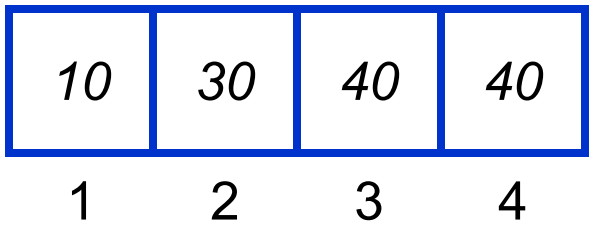
An Example: Insertion Sort



i = 4	j = 2	key = 20
A[j] = 30		A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

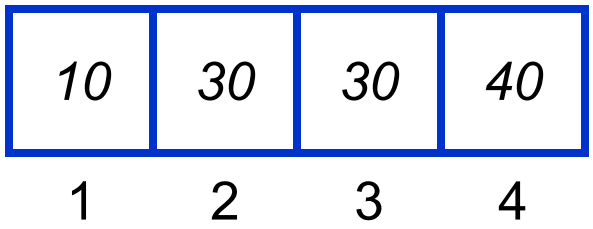
An Example: Insertion Sort



i = 4	j = 2	key = 20
A[j] = 30		A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

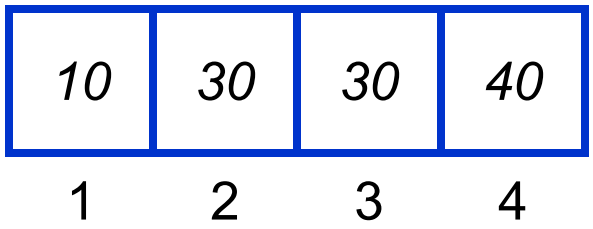
An Example: Insertion Sort



i = 4	j = 2	key = 20
A[j] = 30		A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

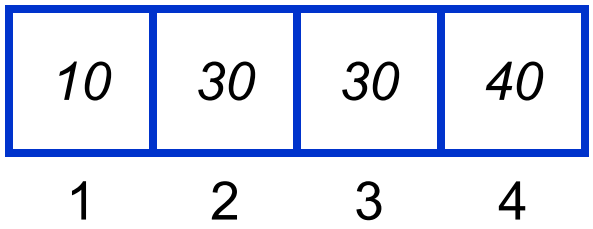
An Example: Insertion Sort



i = 4 j = 2	key = 20
A[j] = 30	A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

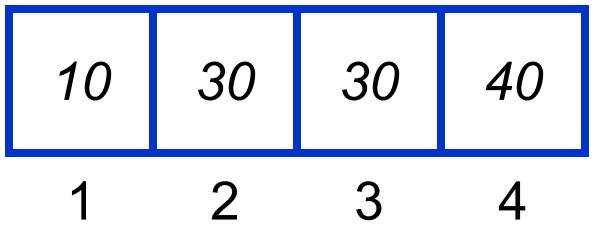
An Example: Insertion Sort



i = 4	j = 1	key = 20
A[j] = 10		A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

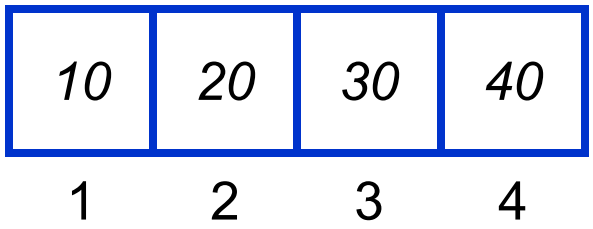
An Example: Insertion Sort



i = 4	j = 1	key = 20
A[j] = 10		A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

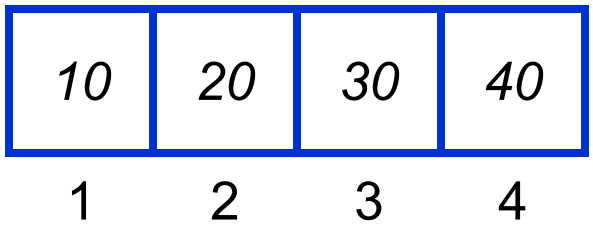
An Example: Insertion Sort



i = 4	j = 1	key = 20
A[j] = 10		A[j+1] = 20

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

An Example: Insertion Sort



i = 4	j = 1	key = 20
A[j] = 10		A[j+1] = 20

```
InsertionSort(A, n) {
   for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
   }
}
```

Done!

Animating Insertion Sort

- Check out the Animator, a java applet at:

- ***https://courses.cs.vt.edu/csonline/Algorithms/Lessons/InsertionCardSort***

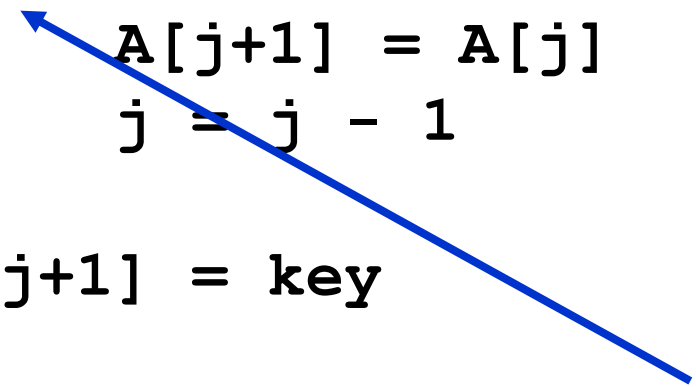
Insertion Sort

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

What is the precondition for this loop?

Insertion Sort

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
      }
      A[j+1] = key
  }
}
```



How many times will this loop execute?

Insertion Sort

Statement	Effort
`InsertionSort(A, n) {`	
`  for i = 2 to n {`	$c_1 n$
`  key = A[i]`	$c_2(n-1)$
`  j = i - 1;`	$c_3(n-1)$
`  while (j > 0) and (A[j] > key) {`	$c_4 T$
`  A[j+1] = A[j]`	$c_5(T-(n-1))$
`  j = j - 1`	$c_6(T-(n-1))$
`  }`	0
`  A[j+1] = key`	$c_7(n-1)$
`  }`	0
`}`	

$T = t_2 + t_3 + \ldots + t_n$ where t_i is number of while expression evaluations for the i^{th} for loop iteration

Analyzing Insertion Sort

- $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 T + c_5(T - (n-1)) + c_6(T - (n-1)) + c_7(n-1)$

  $= c_8 T + c_9 n + c_{10}$
- What can T be?
  - Best case -- inner loop body never executed
    - $t_i = 1$ ➔ T(n) is a linear function
  - Worst case -- inner loop body executed for all previous elements
    - $t_i = i$ ➔ T(n) is a quadratic function
  - Average case
    - ???

Common order-of-growth classifications

order of growth	name	typical code framework	description	example	$T(2N)$ / $T(N)$
1	**constant**	`a = b + c;`	statement	add two numbers	1
$\log N$	**logarithmic**	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~ 1
N	**linear**	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
$N \log N$	**linearithmic**	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	**quadratic**	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs	4
N^3	**cubic**	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }`	triple loop	check all triples	8
2^N	**exponential**	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

Upper Bound Notation

- We say InsertionSort's run time is $O(n^2)$
  - Properly we should say run time is *in* $O(n^2)$
  - Read O as "Big-O" (you'll also hear it as "order")
- In general a function
  - f(n) is O(g(n)) if there exist positive constants c and n_0 such that $f(n) \leq c \times g(n)$ for all $n \geq n_0$
- Formally
  - $O(g(n)) = \{ f(n): \exists$ positive constants c and n_0 such that $f(n) \leq c \times g(n) \forall\ n \geq n_0 \}$

- Question
  - Is InsertionSort $O(n^3)$?
  - Is InsertionSort $O(n)$?

Big O Fact

- A polynomial of degree k is $O(n^k)$
- Proof:
  - Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$
    - Let $a_i = | b_i |$
  - $f(n) \leq a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$

$$\leq \quad n^k \sum a_i \frac{n^i}{n^k} \quad \leq \quad n^k \sum a_i \quad \leq \quad cn^k$$

Lower Bound Notation

- We say InsertionSort's run time is $\Omega(n)$
- In general a function
  - f(n) is $\Omega(g(n))$ if $\exists$ positive constants c and n_0 such that $0 \leq c \times g(n) \leq \forall f(n) \quad n \geq n_0$
- Proof:
  - Suppose run time is an + b
    - Assume a and b are positive (what if b is negative?)
  - an $\leq$ an + b

Asymptotic Tight Bound

- A function f(n) is $\Theta(g(n))$ if $\exists$ positive constants c_1, c_2, and n_0 such that

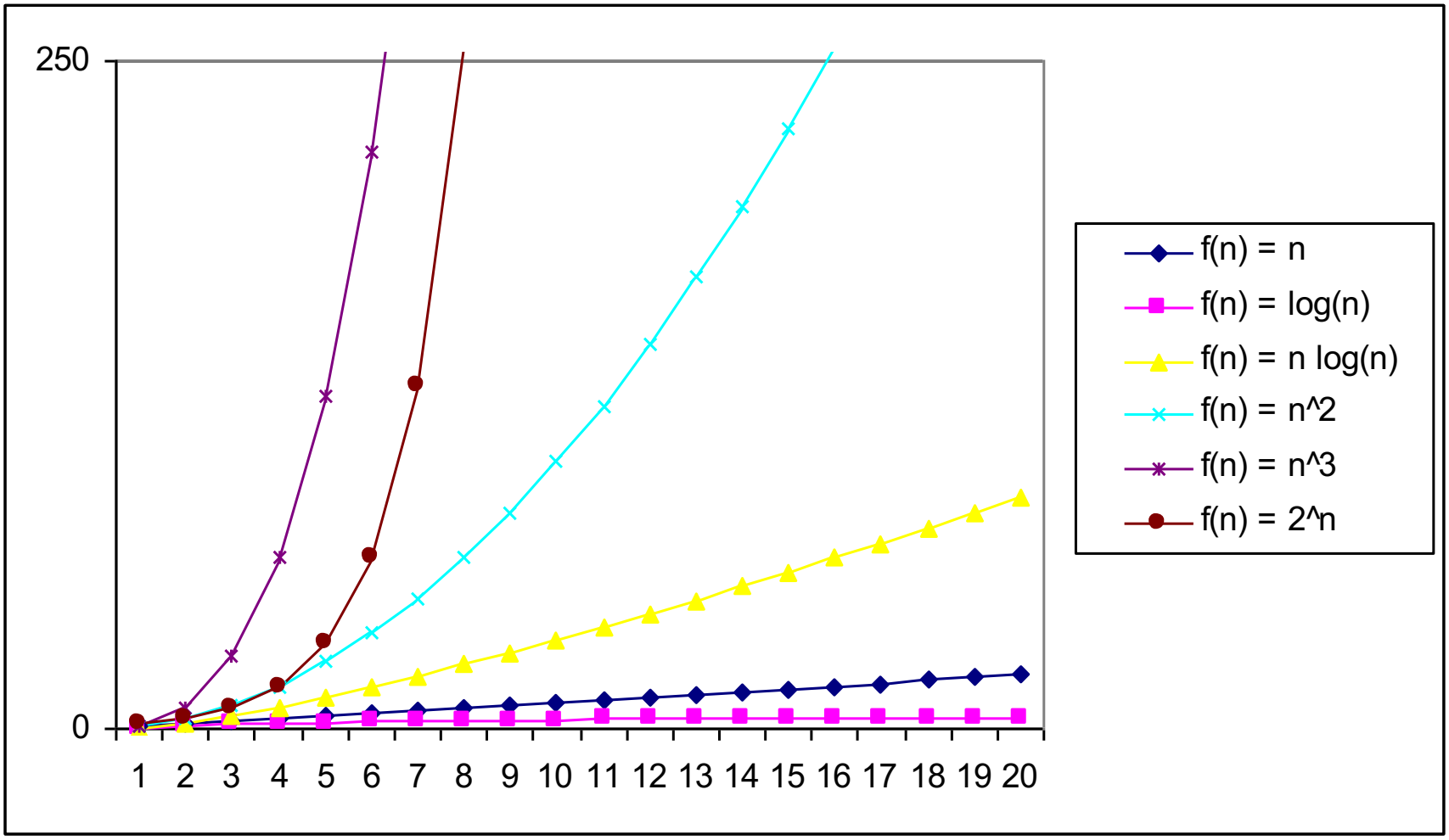
$$c_1\, g(n) \leq f(n) \leq c_2\, g(n)\ \forall\ n \geq n_0$$

- Theorem
  - f(n) is $\Theta(g(n))$ iff f(n) is both $O(g(n))$ and $\Omega(g(n))$

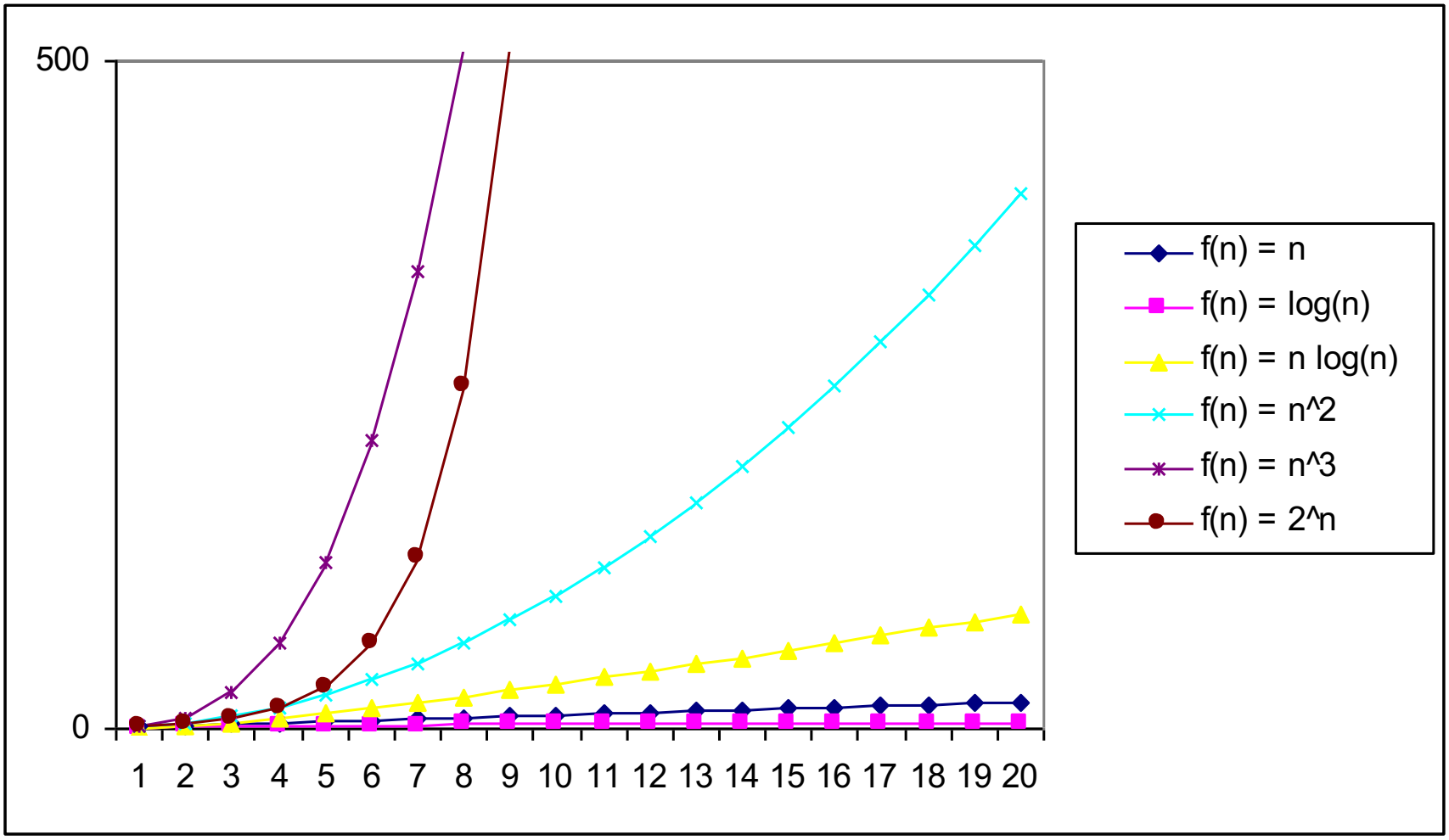
Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ $\vdots$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ $\vdots$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$ $\vdots$	develop lower bounds

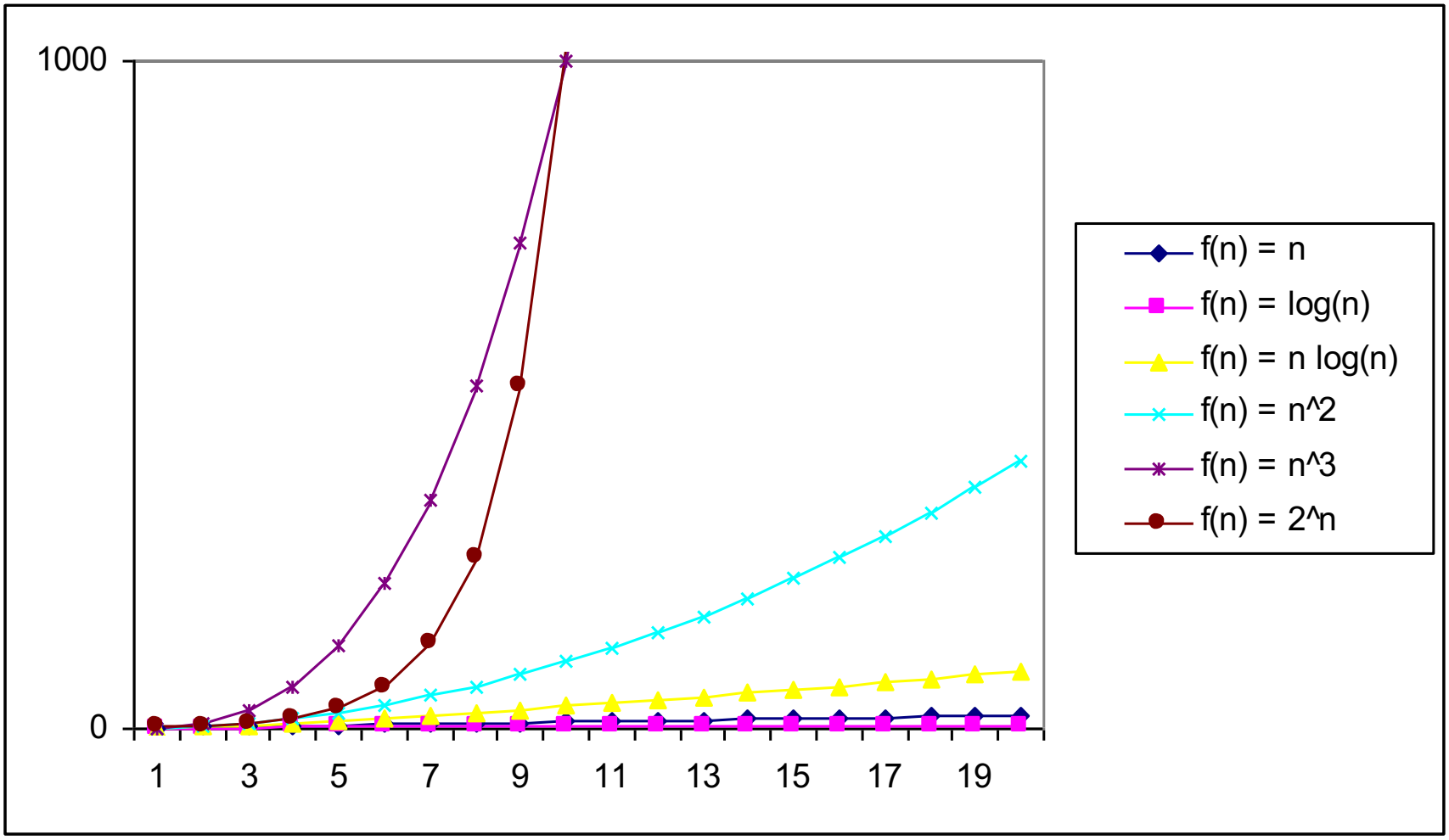
Practical Complexity



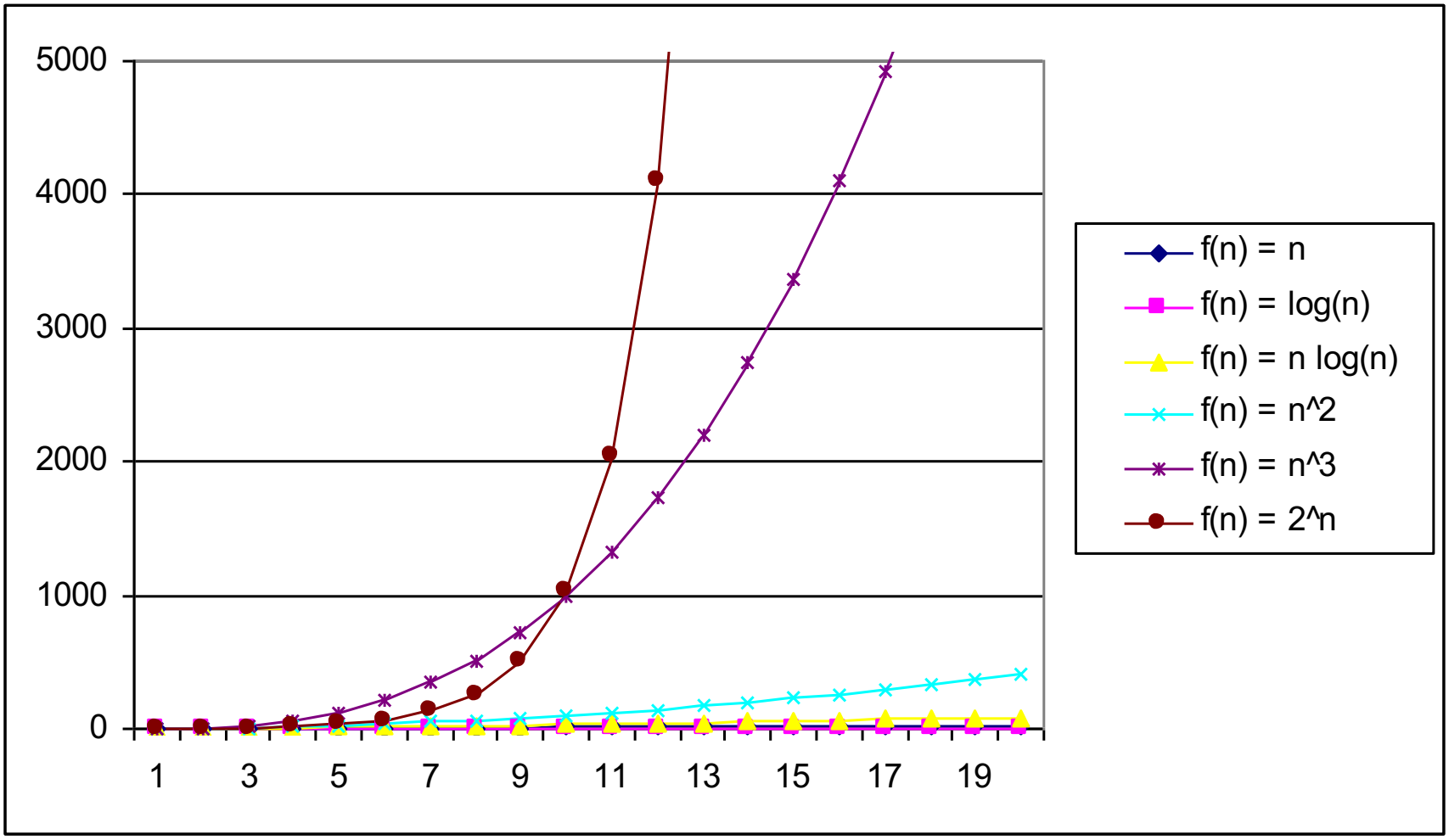
Practical Complexity



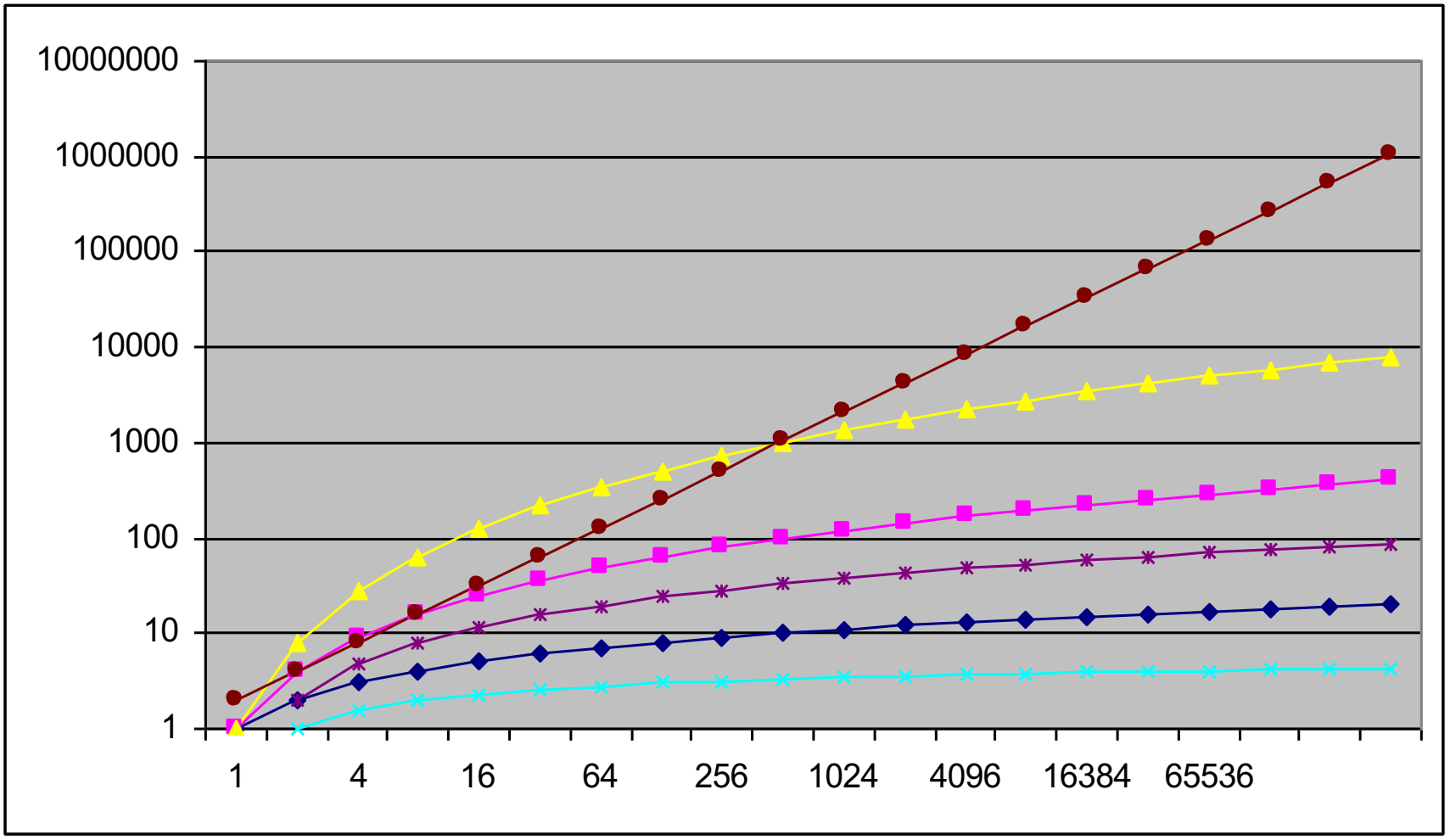
Practical Complexity



Practical Complexity



Practical Complexity



Other Asymptotic Notations

- A function f(n) is o(g(n)) if $\exists$ positive constants c and n_0 such that

  $f(n) < c\ g(n) \quad \forall n \geq n_0$

- A function f(n) is ω(g(n)) if $\exists$ positive constants c and n_0 such that

  $c\ g(n) < f(n) \quad \forall n \geq n_0$

- Intuitively,
  - o() is like <
  - ω() is like >
  - Θ() is like =
  - O() is like $\leq$
  - Ω() is like $\geq$

Types of analyses

Best case. *Lower bound on cost.*

- Determined by "easiest" input.
- Provides a goal for all inputs.

Worst case. *Upper bound on cost.*

- Determined by "most difficult" input.
- Provides a guarantee for all inputs.

Average case. *Expected cost for random input.*

- Need a model for "random" input.
- Provides a way to predict performance.

Memory: Basics

Bit. 0 or 1.

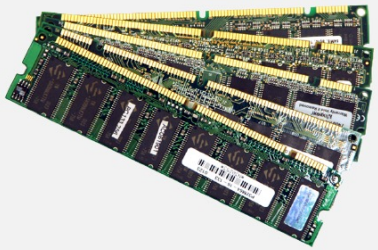
Byte. 8 bits.

Megabyte (MB). 1 million or 2^{20} bytes.

Gigabyte (GB). 1 billion or 2^{30} bytes.

NIST

most computer scientists



64-bit machine. We assume a 64-bit machine with 8-byte pointers.

- Can address more memory.
- Pointers use more space.

some JVMs "compress" ordinary object pointers to 4 bytes to avoid this cost

Typical memory usage for primitive types and arrays

type	bytes
boolean	1
byte	1
char	2
int	4
float	4
long	8
double	8

primitive types

type	bytes
char[]	$2N + 24$
int[]	$4N + 24$
double[]	$8N + 24$

one-dimensional arrays

type	bytes
char[][]	$\sim 2MN$
int[][]	$\sim 4MN$
double[][]	$\sim 8MN$

two-dimensional arrays

Typical memory usage summary

Total memory usage for a data type value:

- Primitive type: 4 bytes for `int`, 8 bytes for `double`, ...
- Object reference: 8 bytes.
- Array: 24 bytes + memory for each array entry.
- Object: 16 bytes + memory for each instance variable.
- Padding: round up to multiple of 8 bytes.

+ 8 extra bytes per inner class object
(for reference to enclosing class)

Shallow memory usage: Don't count referenced objects.

Deep memory usage: If array entry or instance variable is a reference, count memory (recursively) for referenced object.