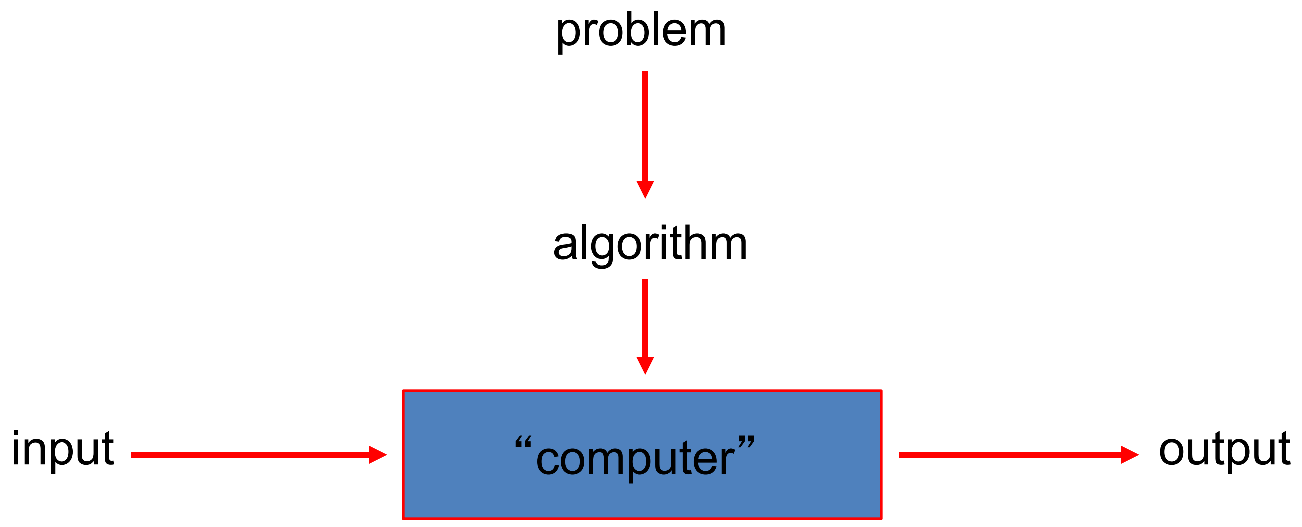
CPSC 5031: Data Structures & Algorithms

Lecture 2: Analysis of Algorithm Efficiency

(Levitin, Chapter 1, 2.1-2.6)

What is an algorithm?

An ***algorithm*** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



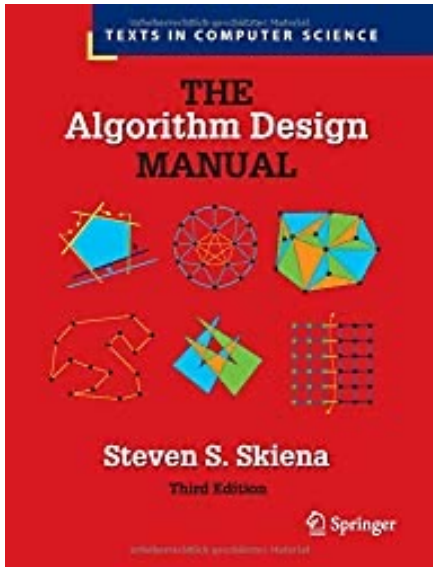
Why study algorithms?

- Theoretical importance
  - the core of computer science
- Practical importance
  - A practitioner's toolkit of known algorithms
  - Framework for designing and analyzing algorithms for new problems

Two main issues related to algorithms

- How to design algorithms

- How to analyze algorithm efficiency



https://algorist.com/

Analysis of algorithms

- How good is the algorithm?
  - time efficiency
  - space efficiency

- Does there exist a better algorithm?
  - lower bounds
  - optimality

Algorithm Efficiency

- Given computing resources and input data
  - how fast does an algorithm run?
    - Time efficiency: amount of time required to accomplish the task
    - Our focus
  - How much memory is required?
    - Space efficiency: amount of memory required
    - Deals with the extra space the algorithm requires

Time Efficiency

- Time efficiency depends on :
  - size of input
  - speed of machine
  - quality of source code
  - quality of compiler

$\Bigg\}$ These vary from one platform to another

So, we cannot express time efficiency meaningfully in real time units such as seconds!

Empirical analysis of time efficiency

- Select a specific (typical) sample of inputs
- Use physical unit of time (e.g., milliseconds)

  or

  Count actual number of basic operation’s executions
- Analyze the empirical data
- Limitation: results dependent on the particular computer and the sample of inputs

Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*

- *Basic operation*: the operation that contributes most towards the running time of the algorithm

Input size

$$T(n) \approx c_{op} C(n)$$

running time

execution time for basic operation

Number of times basic operation is executed

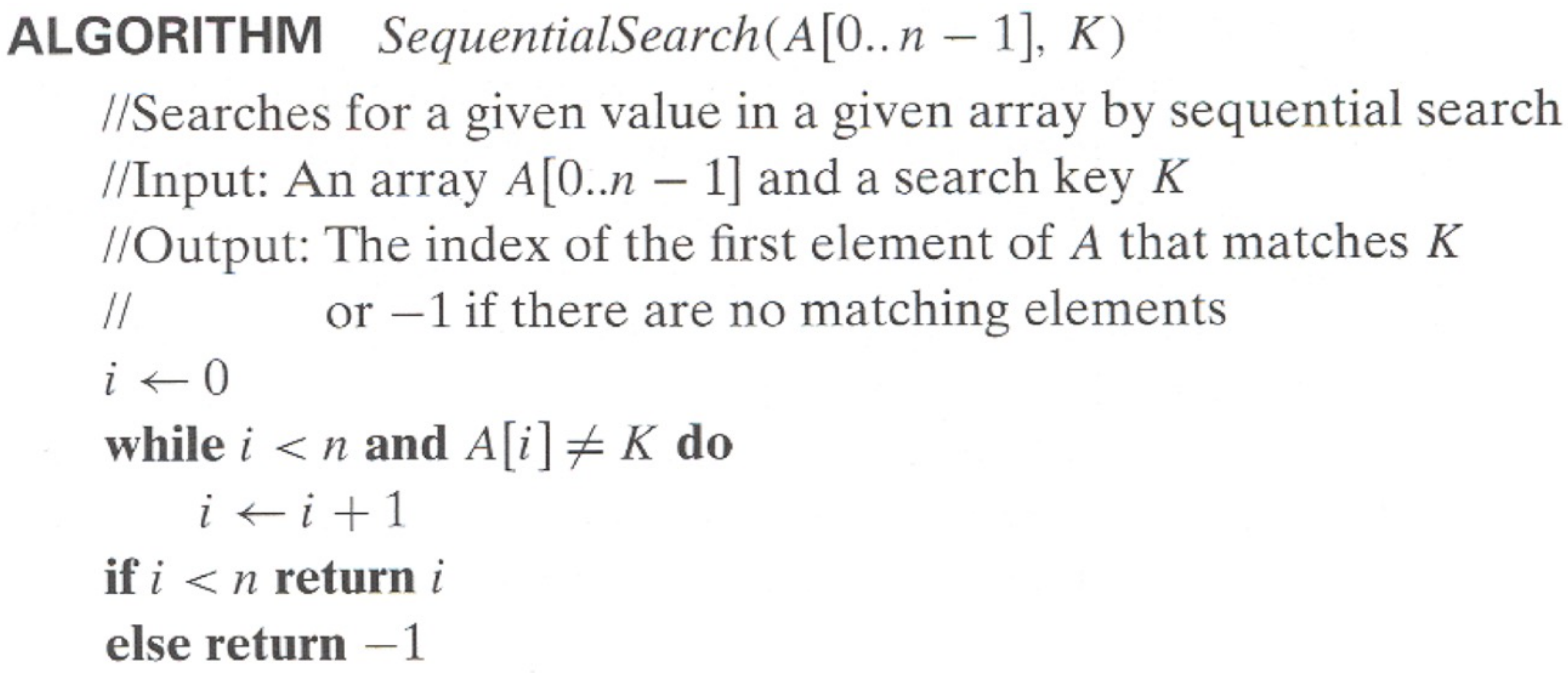
Input size and basic operation examples

Problem	***Input size measure***	***Basic operation***
Searching for key in a list of *n* items	Number of list’s items, i.e. *n*	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer *n*	*n*’ size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

Time Efficiency

- T(n) = (approximated by) number of times the basic operation is executed.
- Not only depends on the input size *n*, but also depends on the arrangement of the input items
  - Best case: not informative
  - Average case: difficult to determine
  - Worst case: is used to measure an algorithm’s performance

Example: Sequential Search



```
ALGORITHM SequentialSearch($A[0..n-1]$, $K$)
    //Searches for a given value in a given array by sequential search
    //Input: An array $A[0..n-1]$ and a search key $K$
    //Output: The index of the first element of $A$ that matches $K$
    //        or $-1$ if there are no matching elements
    $i \leftarrow 0$
    while $i < n$ and $A[i] \neq K$ do
        $i \leftarrow i + 1$
    if $i < n$ return $i$
    else return $-1$
```

- Best case T(n)
- Worst case T(n)
- Average case T(n)
  - Assume success search probability of *p*

Order of Growth

- Established framework for analyzing *$T(n)$*
- Order of growth as $n \rightarrow \infty$
  - Highest-order term is what counts
    - Remember, we are doing asymptotic analysis
    - As the input size grows larger it is the high order term that dominates

Asymptotic Order of Growth

A way of comparing functions that ignores constant factors and small input sizes

- O(g(n)): class of functions f(n) that grow no faster than g(n)   [worst case]
- Θ(g(n)): class of functions f(n) that grow at same rate as g(n) [avg case]
- Ω(g(n)): class of functions f(n) that grow at least as fast as g(n) [best case]

Big-oh

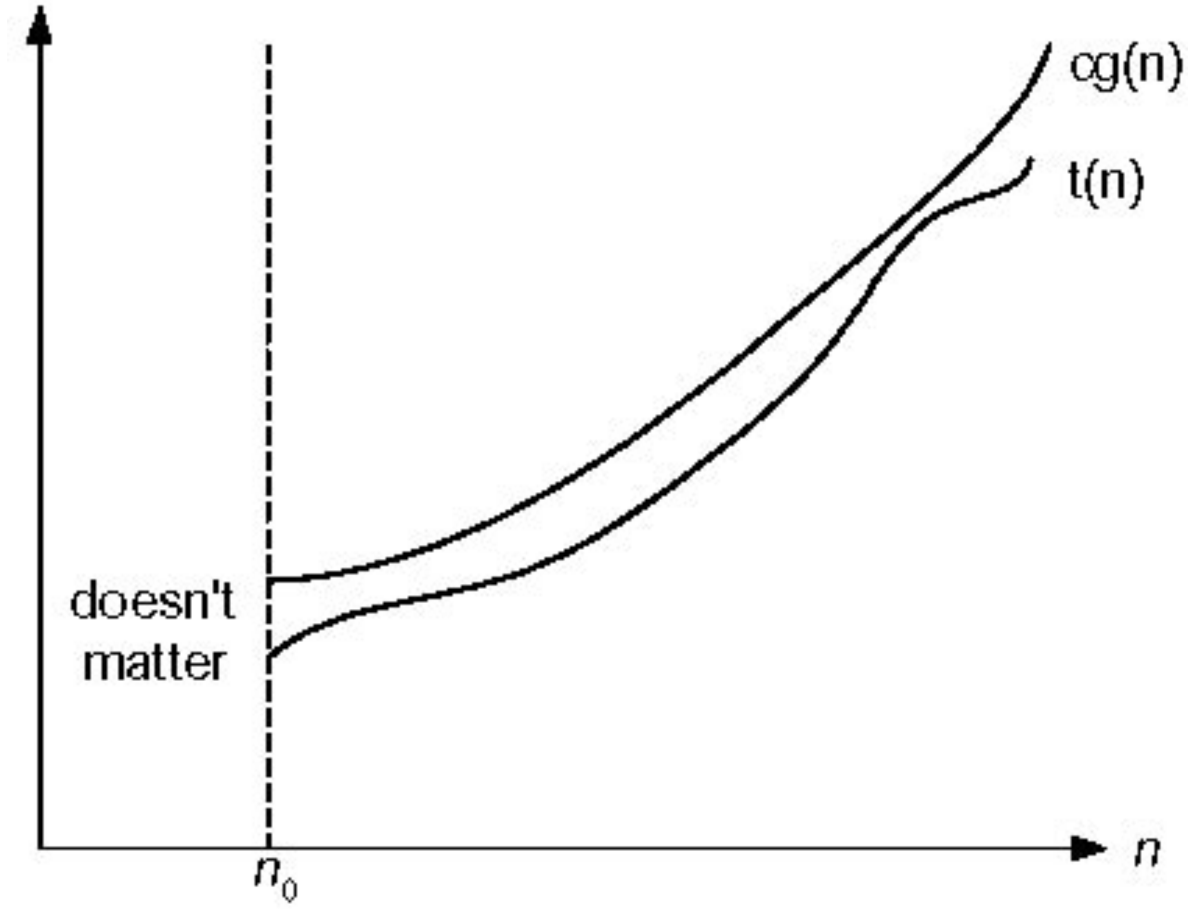


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-omega

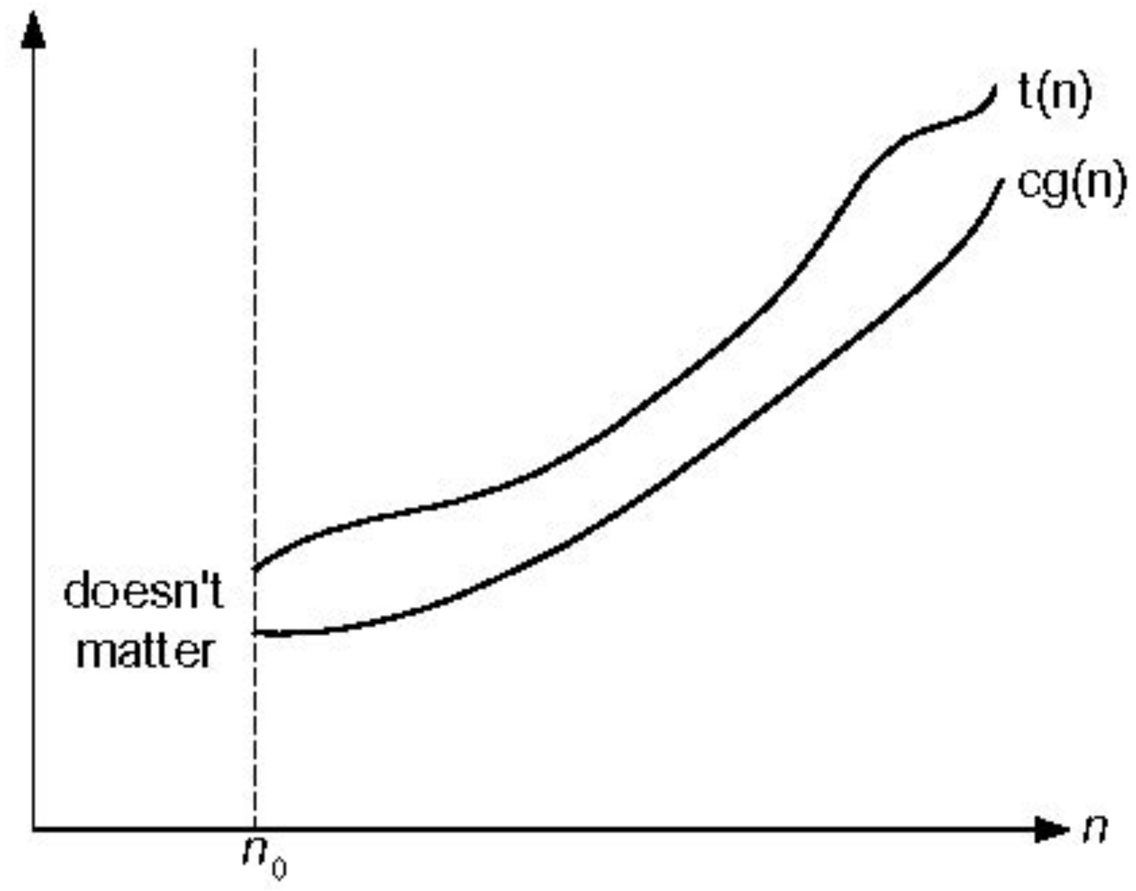


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-theta

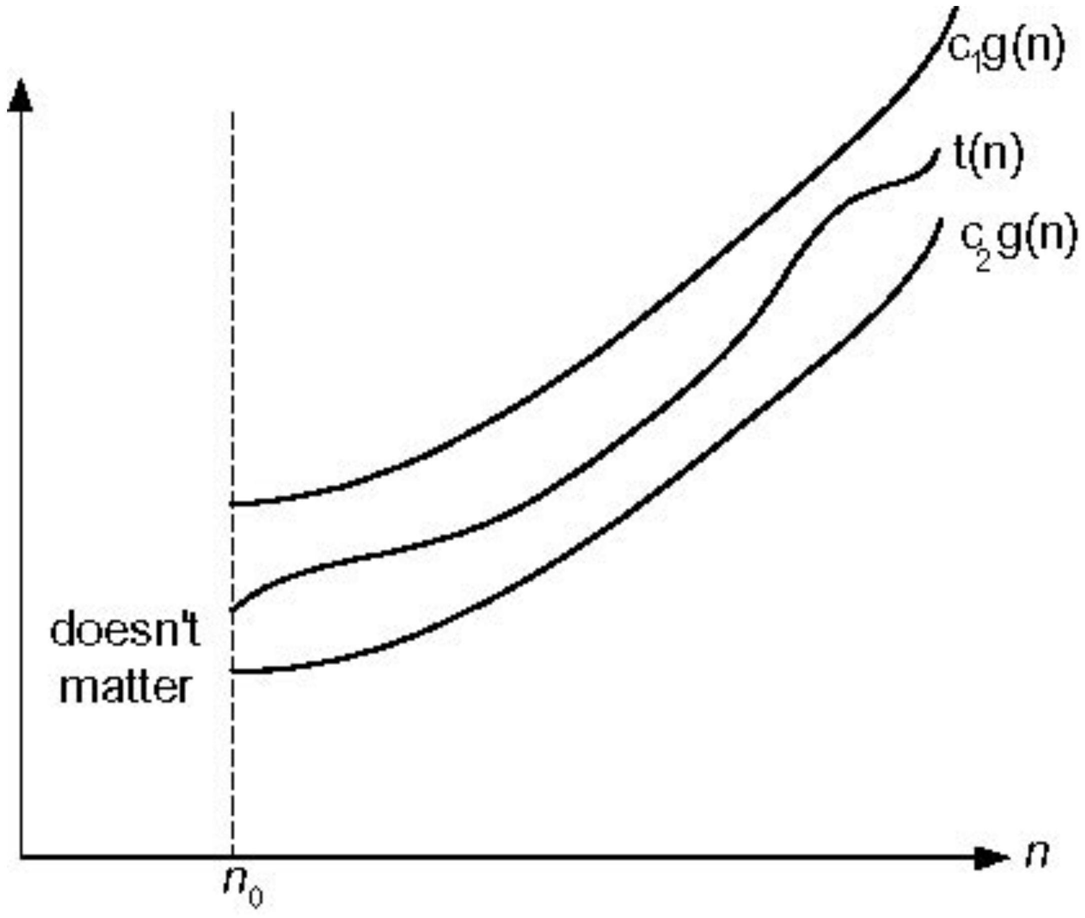


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Upper Bound Notation

- In general a function
  - *f(n)* is *O(g(n))* if there exist positive constants *c* and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
  - Order of growth of *f(n)* $\leq$ order of growth of *g(n)* (within constant multiple)
- Formally
  - $O(g(n)) = \{ f(n)$: $\exists$ positive constants *c* and n_0 such that $f(n) \leq c \cdot g(n)\ \forall\ n \geq n_0$

Big-Oh

- Examples
  - $n(n-1)+1$ is $O(n^2)$
  - $10n$ is $O(n^2)$
  - $5n+20$ is $O(n)$

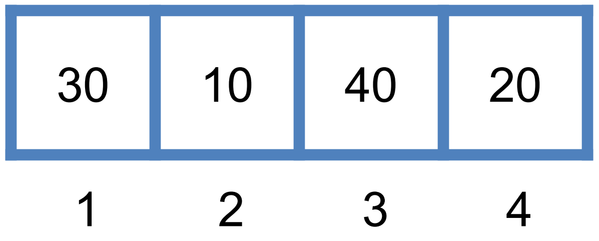
An Example: Insertion Sort

6 5 3 1 8 7 2 4

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

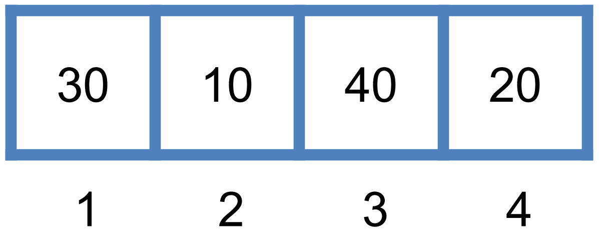


$i = \varnothing \quad j = \varnothing \quad key = \varnothing$

$A[j] = \varnothing \quad A[j+1] = \varnothing$

```
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

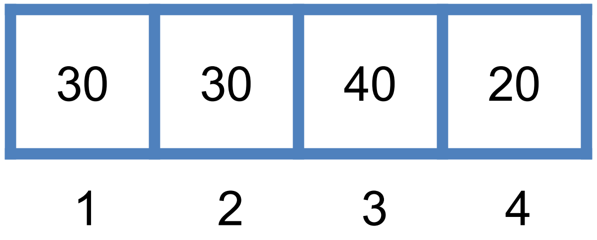


$i = 2$ $j = 1$ $key = 10$

$A[j] = 30$ $A[j+1] = 10$

```
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

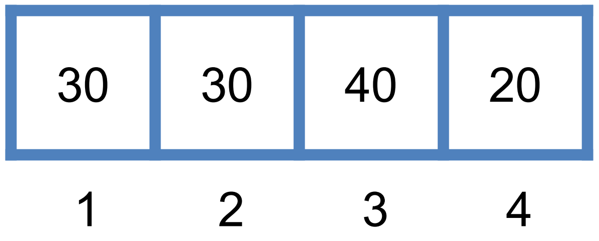


$i = 2$ $j = 1$ $key = 10$

$A[j] = 30$ $A[j+1] = 30$

```
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

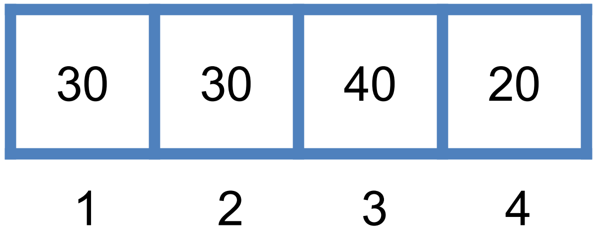
30	30	40	20
1	2	3	4

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

$A[j] = \varnothing \quad A[j+1] = 30$

```
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

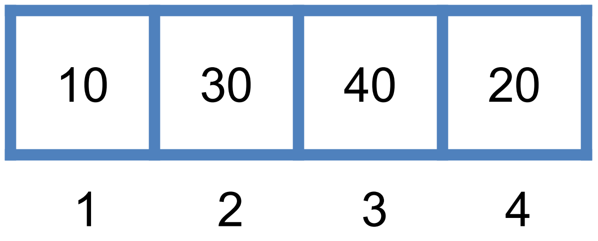


$i = 2 \quad j = 0 \quad key = 10$

$A[j] = \varnothing \quad A[j+1] = 30$

```
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

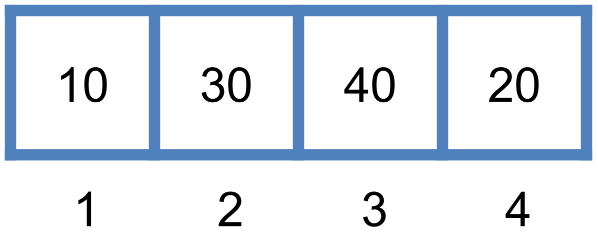


$i = 2$ $j = 0$ $key = 10$

$A[j] = \varnothing$ $A[j+1] = 10$

```
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
   }
}
```

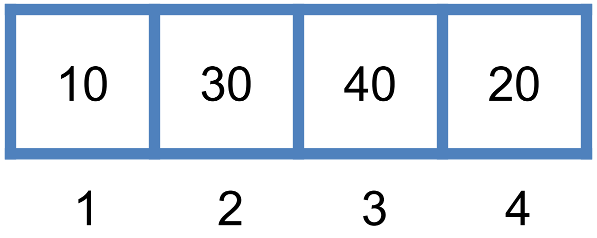
An Example: Insertion Sort



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

```
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

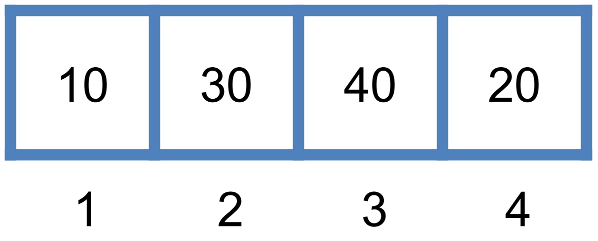


$i = 3$ $j = 0$ $key = 40$

$A[j] = \varnothing$ $A[j+1] = 10$

```
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

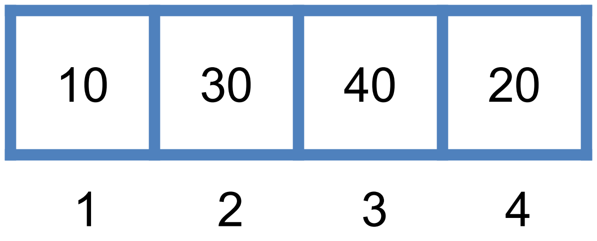


$i = 3$ $j = 0$ $key = 40$

$A[j] = \varnothing$ $A[j+1] = 10$

```
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
  }
}
```

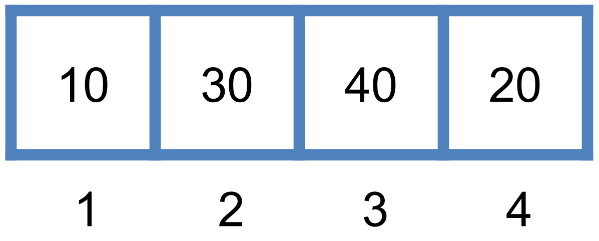
An Example: Insertion Sort



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

```
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

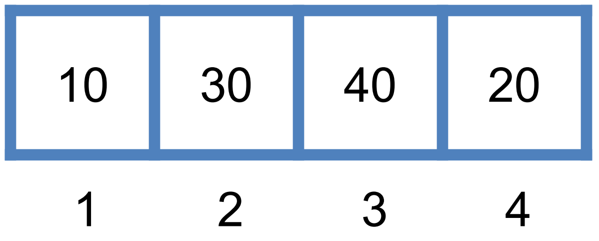


$i = 3 \quad j = 2 \quad key = 40$

$A[j] = 30 \quad A[j+1] = 40$

```
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

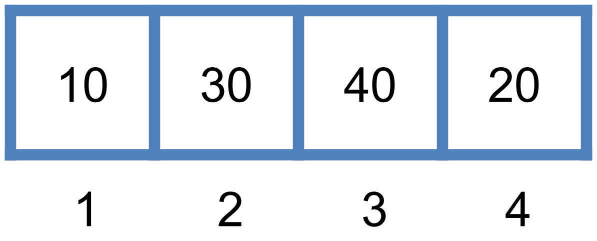


$i = 3 \quad j = 2 \quad key = 40$

$A[j] = 30 \quad A[j+1] = 40$

```
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

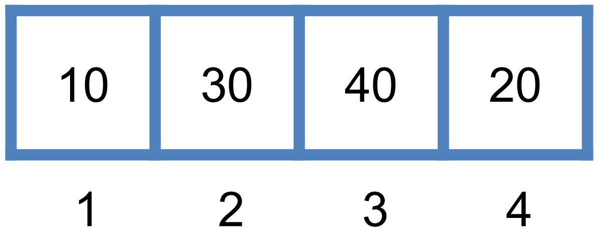


$i = 4 \quad j = 2 \quad key = 40$

$A[j] = 30 \quad A[j+1] = 40$

```
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

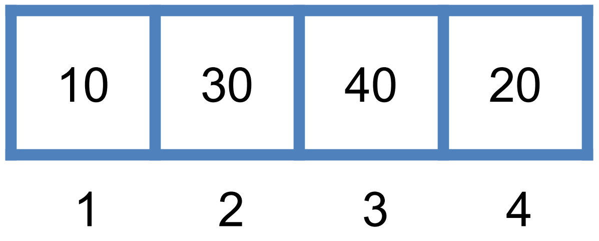


$i = 4$ $j = 2$ $key = 20$

$A[j] = 30$ $A[j+1] = 40$

```
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

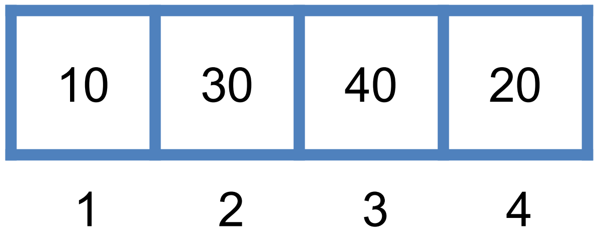


$i = 4$ $j = 2$ $key = 20$

$A[j] = 30$ $A[j+1] = 40$

```
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

10	30	40	20
1	2	3	4

$i = 4$ $j = 3$ $key = 20$

$A[j] = 40$ $A[j+1] = 20$

```
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
   }
}
```

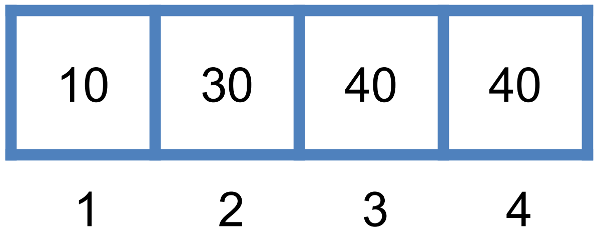
An Example: Insertion Sort

10	30	40	40
1	2	3	4

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

```
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

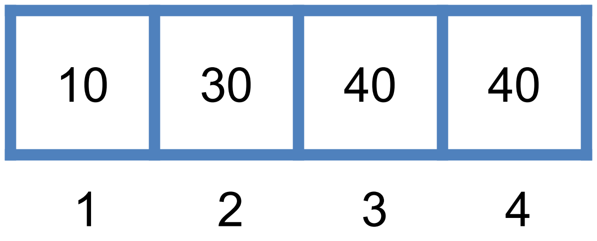


$i = 4$ $j = 3$ $key = 20$

$A[j] = 40$ $A[j+1] = 40$

```
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

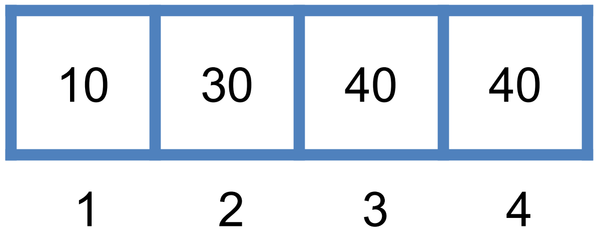


i = 4 j = 3 key = 20

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
   }
}
```

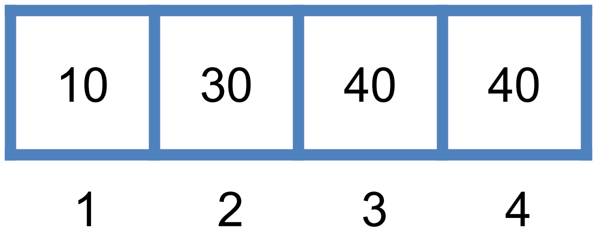
An Example: Insertion Sort



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

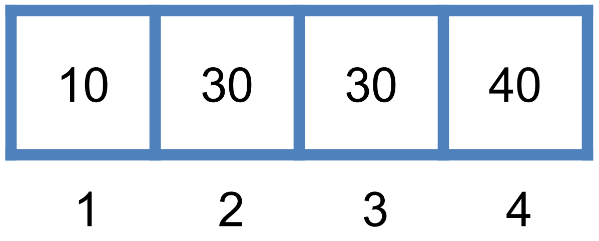


$i = 4$ $j = 2$ $key = 20$

$A[j] = 30$ $A[j+1] = 40$

```
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
   }
}
```

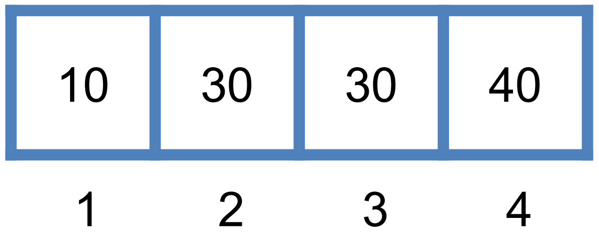
An Example: Insertion Sort



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

```
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

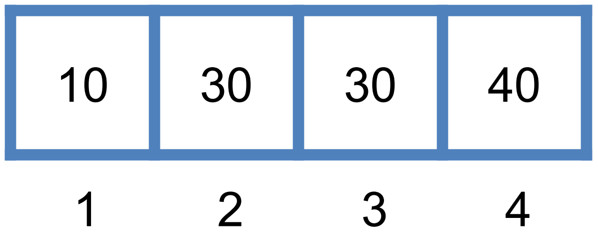


i = 4 j = 2 key = 20

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
   }
}
```

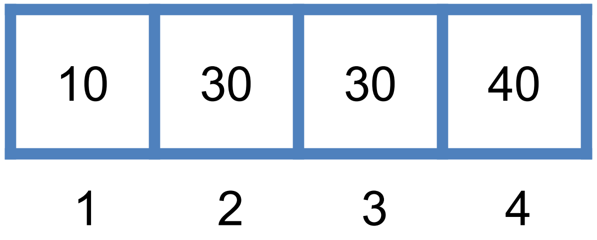
An Example: Insertion Sort



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

```
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
   }
}
```

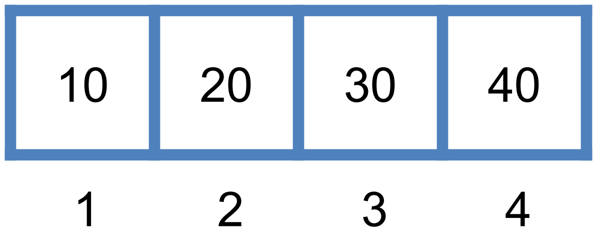
An Example: Insertion Sort



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

```
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
   }
}
```

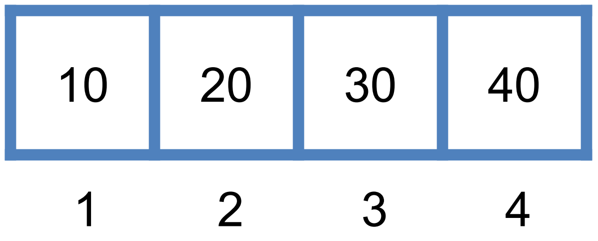
An Example: Insertion Sort



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

```
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!

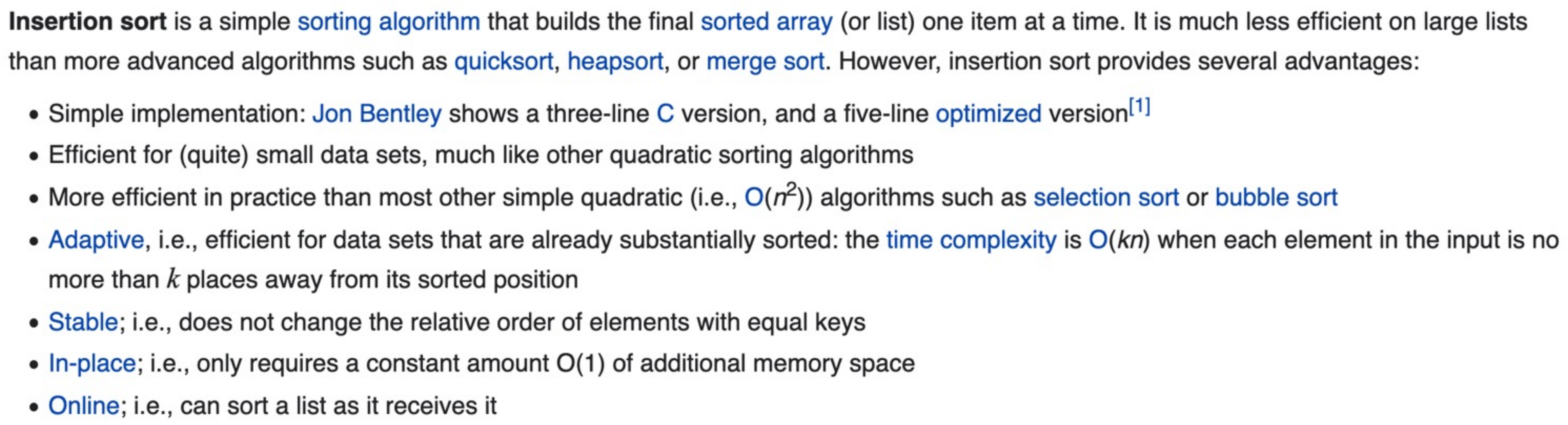
Insertion Sort

```
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?

T(n) for Insertion Sort

- Worst case?
- Best case?



Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

- Simple implementation: Jon Bentley shows a three-line C version, and a five-line optimized version[1]
- Efficient for (quite) small data sets, much like other quadratic sorting algorithms
- More efficient in practice than most other simple quadratic (i.e., $O(n^2)$) algorithms such as selection sort or bubble sort
- Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is $O(kn)$ when each element in the input is no more than k places away from its sorted position
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount O(1) of additional memory space
- Online; i.e., can sort a list as it receives it

https://en.wikipedia.org/wiki/Insertion_sort

Insertion Sort Is O(n^2)

- Proof
  - Suppose runtime is $an^2 + bn + c$
    - If any of a, b, and c are less than 0 replace the constant with its absolute value
  - $an^2 + bn + c \quad \leq (a + b + c)n^2 + (a + b + c)n + (a + b + c)$
    $\leq 3(a + b + c)n^2$ for $n \geq 1$
  - Let $c' = 3(a + b + c)$ and let $n_0 = 1$
- Question
  - Is InsertionSort O(n^3)?
  - Is InsertionSort O(n)?

Big-Oh 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 $a_i = | b_i |$
  - $f(n) \leq a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$

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

Lower Bound Notation

- In general a function
  - $f(n)$ is $\Omega(g(n))$ if $\exists$ positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n) \quad \forall\ n \geq n_0$
- Proof:
  - Suppose run time is $an + b$
    - Assume a and b are positive (what if b is negative?)
  - $an \leq an + b$
  - Example: $n^3 = \Omega(n^2)$

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 Θ(g(n)) iff f(n) is both O(g(n)) and Ω(g(n))
  - Proof
- Engineering
  - Drop low-order items; ignore leading constants
  - Example: $3n^3 + 90n^2 - 5n + 9999 = \Theta(n^3)$

Using limits for comparing orders of growth

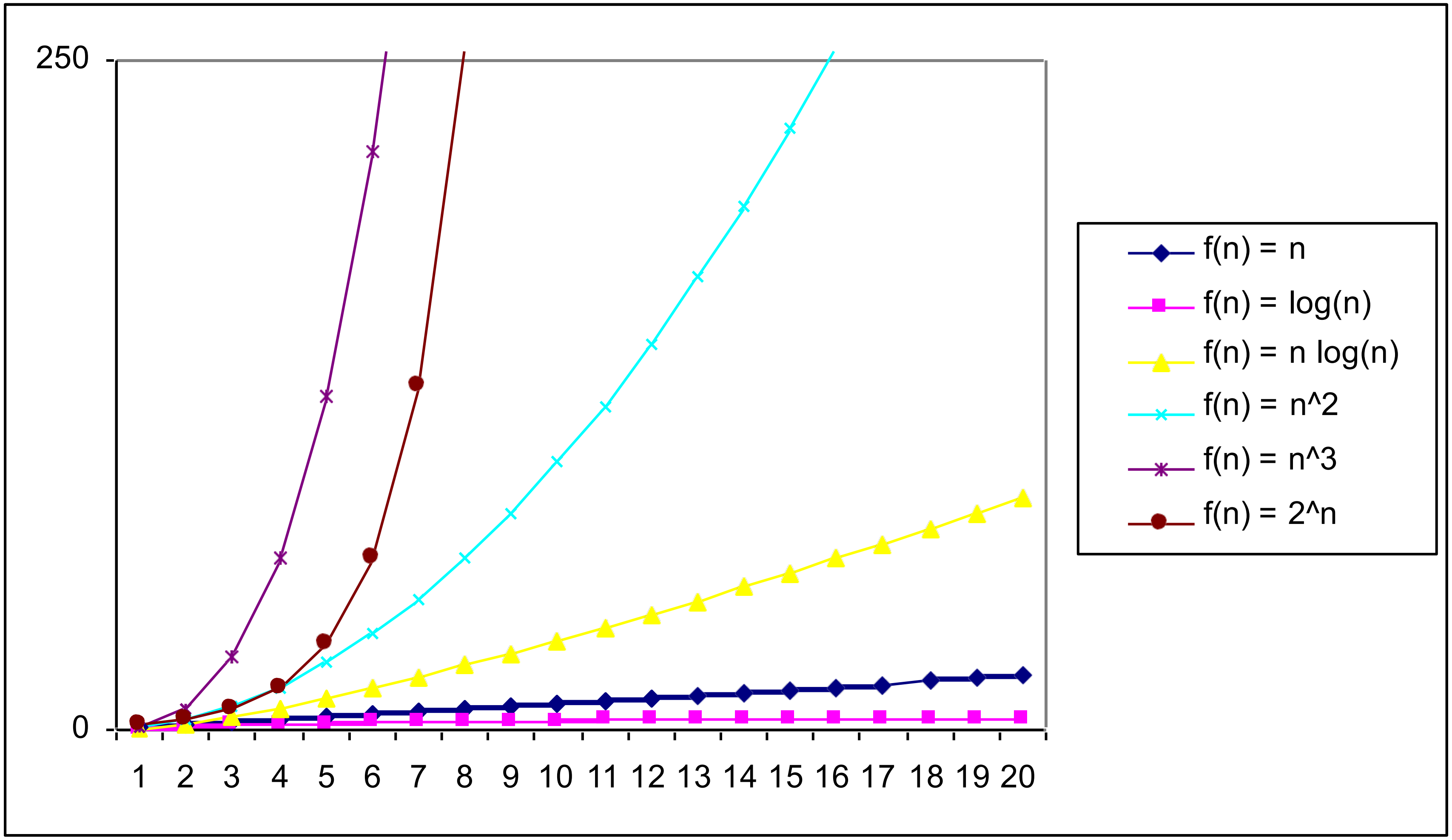
$$\lim_{n->\infty}\frac{t(n)}{g(n)}=\begin{cases}0 & \text{T(n) has a smaller order of growth than g(n)}\\ c>0 & \text{T(n) has the same order of growth than g(n)}\\ \infty & \text{T(n) has a larger order of growth than g(n)}\end{cases}$$

Example: compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2

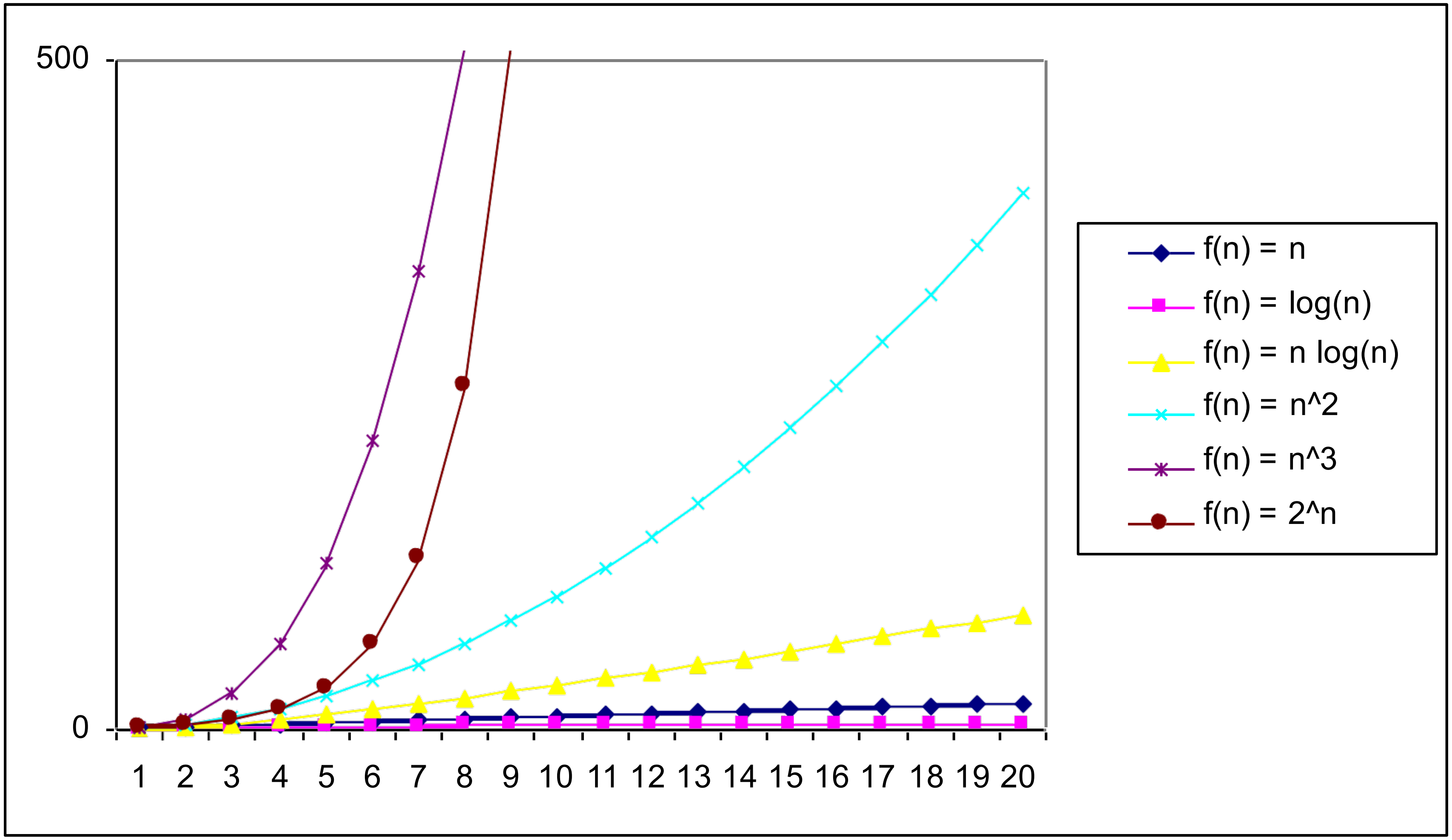
$$\lim_{n->\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{T(n) has a smaller order of growth than g(n)} \\ c > 0 & \text{T(n) has the same order of growth than g(n)} \\ \infty & \text{T(n) has a larger order of growth than g(n)} \end{cases}$$

$\frac{1}{2}n(n-1)$ and n^2

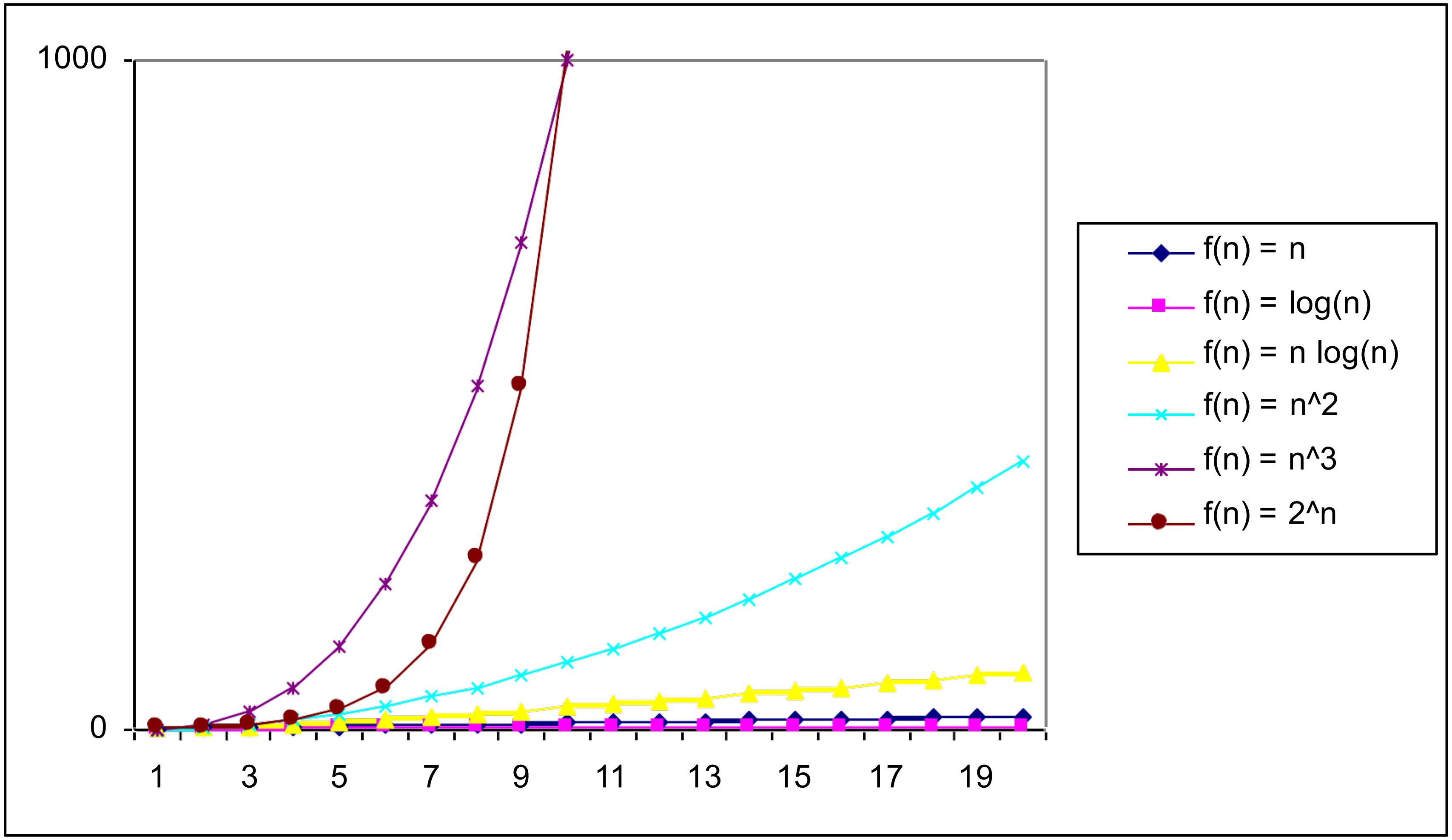
Practical Complexity



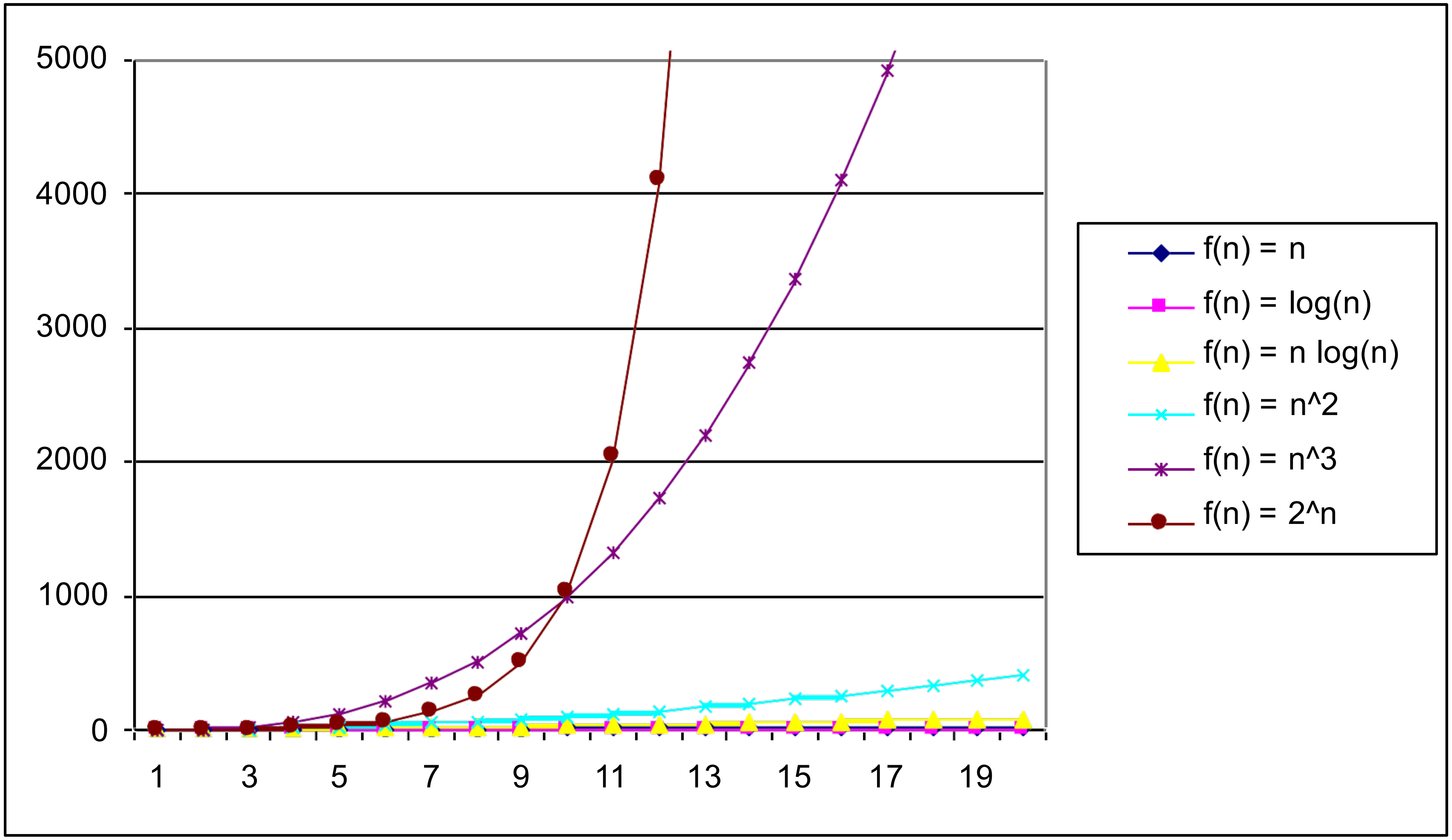
Practical Complexity



Practical Complexity



Practical Complexity



Basic asymptotic efficiency classes

1	constant
$\log n$	logarithmic
n	linear
$n \log n$	n-log-n
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

Analysis for Algorithms

- Non-recursive algorithms
- Recursive algorithms
- Often, we focus on worst case

Analysis for Algorithms

- Non-recursive algorithms
- Recursive algorithms
- ~~Often,~~ we focus on worst case

T(n) for nonrecursive algorithms

General Plan for Analysis

- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst*, *average*, and *best* cases for input of size *n*
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules

Useful summation formulas and rules

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le n} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n}\, i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n}\, i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n}\, a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

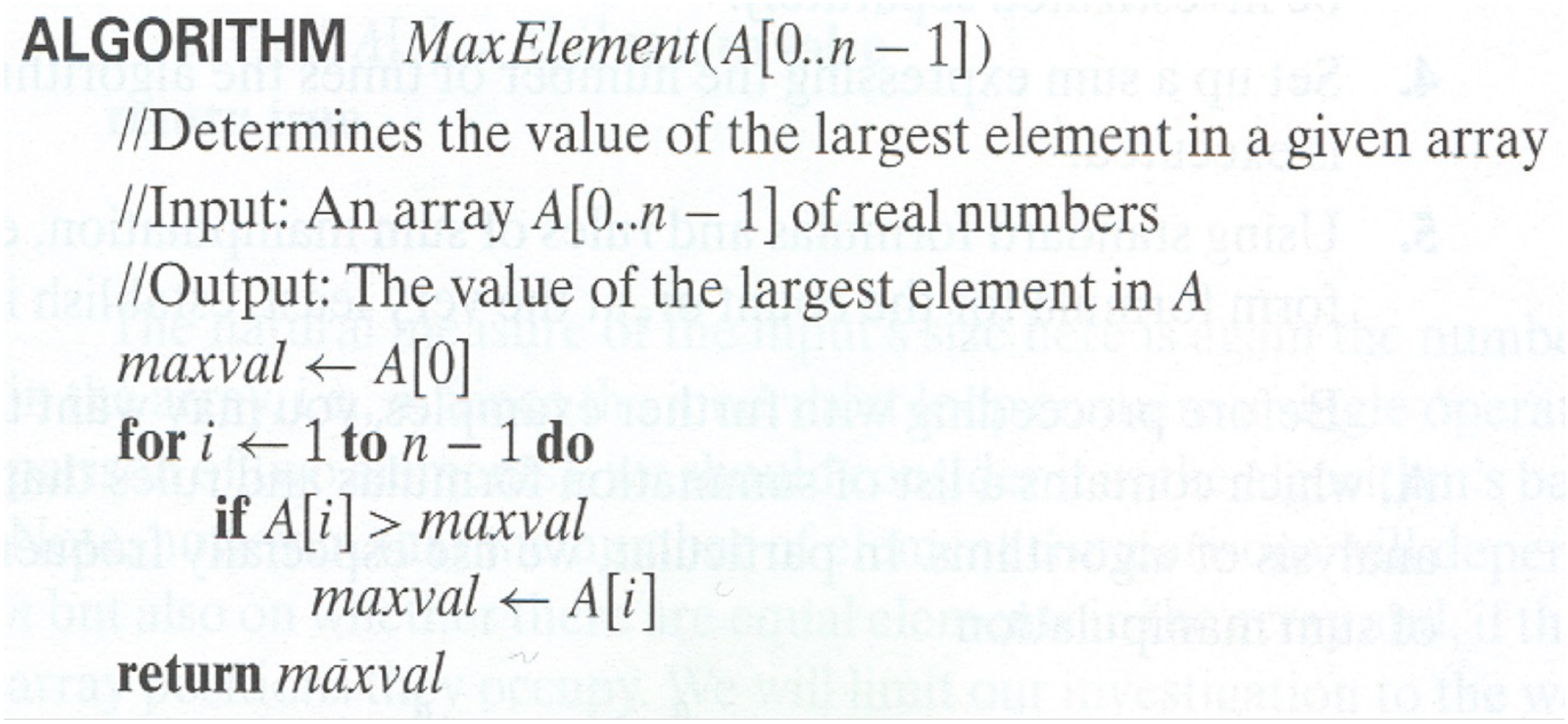
In particular, $\Sigma_{0 \le i \le n}\, 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i$

$\Sigma c a_i = c \Sigma a_i$

$\Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

Example 1: Maximum element

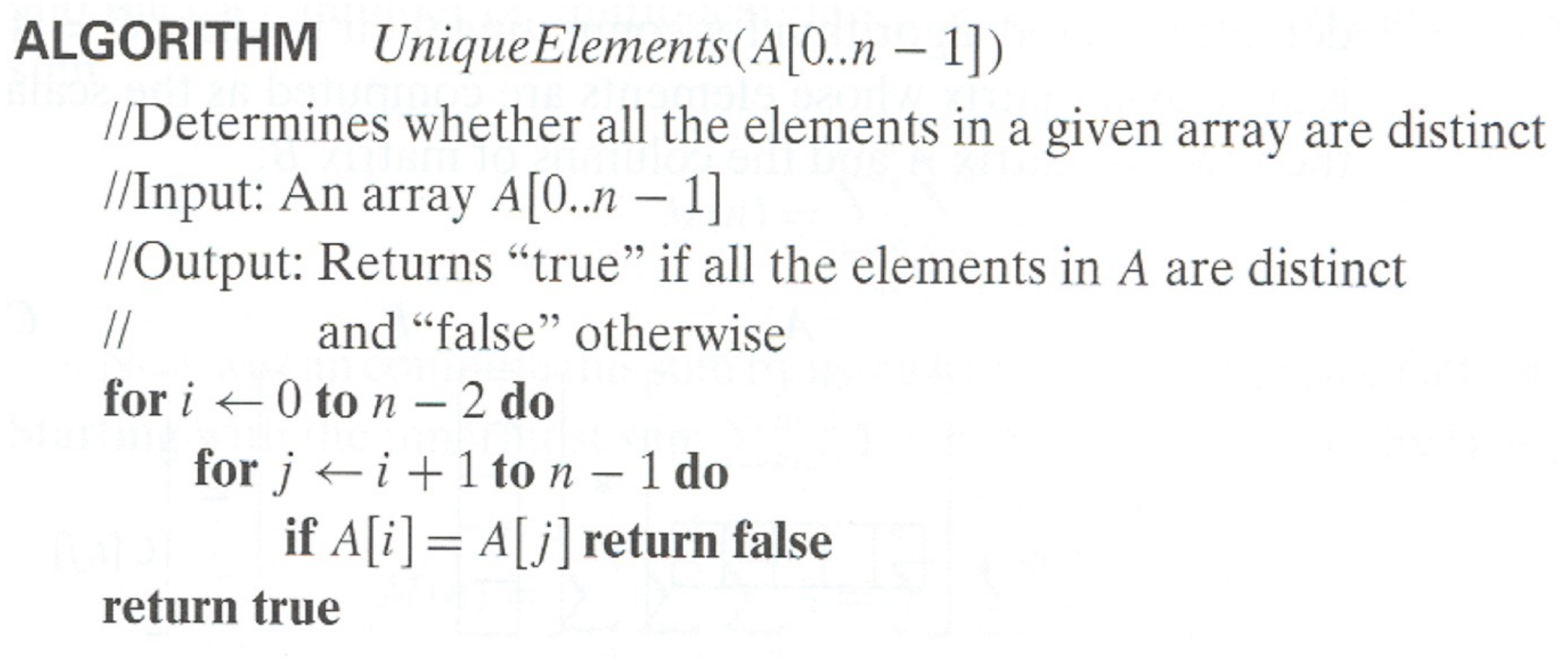


ALGORITHM *MaxElement*$(A[0..n-1])$

```
//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in $A$
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ to $n-1$ do
    if $A[i] > maxval$
        $maxval \leftarrow A[i]$
return $maxval$
```

```
for $i \leftarrow 1$ to $n - 1$ do
    if $A[i] > maxval$
        $maxval \leftarrow A[i]$
return $maxval$
```

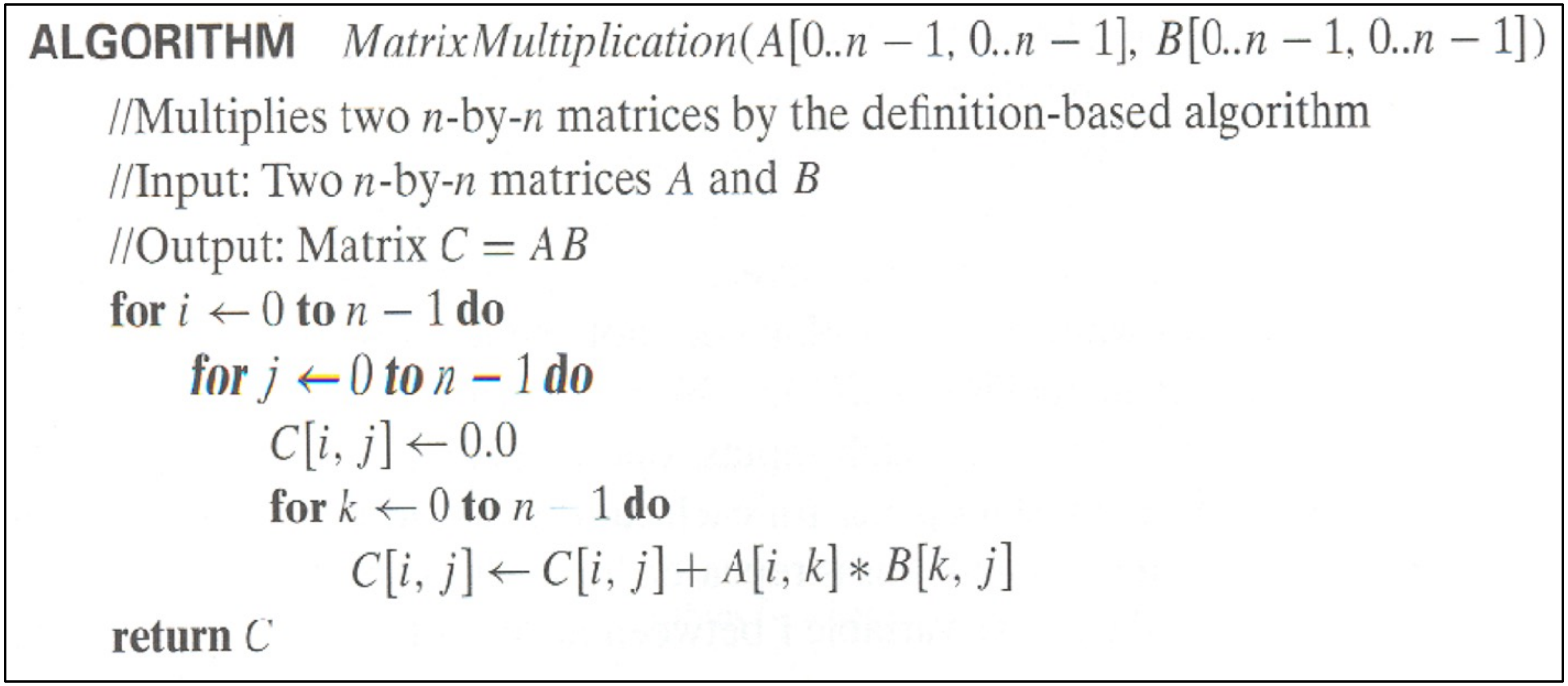
Example 2: Element uniqueness problem



```
ALGORITHM UniqueElements($A[0..n-1]$)
    //Determines whether all the elements in a given array are distinct
    //Input: An array $A[0..n-1]$
    //Output: Returns “true” if all the elements in $A$ are distinct
    //         and “false” otherwise
    for $i \leftarrow 0$ to $n-2$ do
        for $j \leftarrow i+1$ to $n-1$ do
            if $A[i] = A[j]$ return false
    return true
```

```
for $i \leftarrow 0$ to $n - 2$ do
    for $j \leftarrow i + 1$ to $n - 1$ do
        if $A[i] = A[j]$ return false
return true
```

Example 3: Matrix multiplication



```
ALGORITHM  $MatrixMultiplication(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$
    //Multiplies two $n$-by-$n$ matrices by the definition-based algorithm
    //Input: Two $n$-by-$n$ matrices $A$ and $B$
    //Output: Matrix $C = AB$
    for $i \leftarrow 0$ to $n - 1$ do
        for $j \leftarrow 0$ to $n - 1$ do
            $C[i, j] \leftarrow 0.0$
            for $k \leftarrow 0$ to $n - 1$ do
                $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
    return $C$
```

```
for $i \leftarrow 0$ to $n-1$ do
    for $j \leftarrow 0$ to $n-1$ do
        $C[i, j] \leftarrow 0.0$
        for $k \leftarrow 0$ to $n-1$ do
            $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return $C$
```

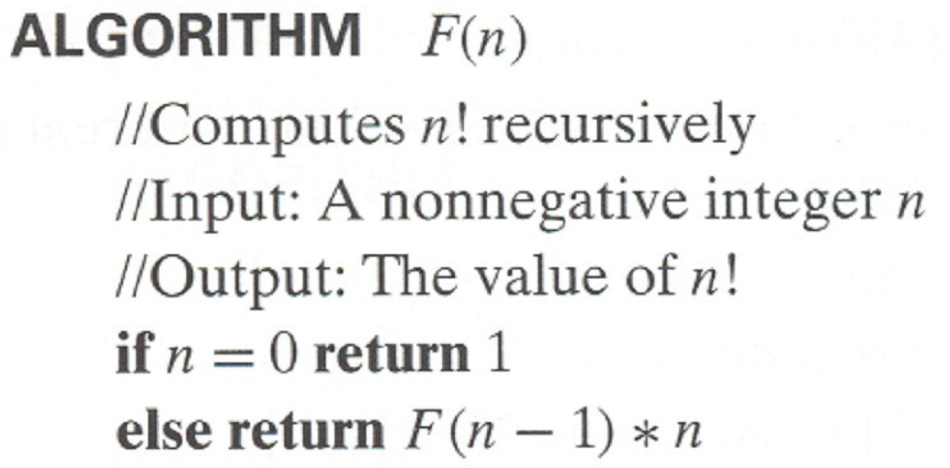
Plan for Analysis of Recursive Algorithms

- Decide on a parameter indicating an input’s size.
- Identify the algorithm’s basic operation.
- Check whether the number of times the basic operation is executed may vary on different inputs of the same size.   (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic operation is executed.
- Solve the recurrence (or, at the very least, establish its solution’s order of growth) by backward substitutions or another method.

Example 1: Recursive evaluation of *n*!

Definition: $n ! = 1 * 2 * \ldots *(n\text{-}1) * n$ for $n \geq 1$ and $0! = 1$

Recursive definition of *n*!: $F(n) = F(n\text{-}1) * n$ for $n \geq 1$ and
$F(0) = 1$



```
ALGORITHM  $F(n)$
    //Computes $n!$ recursively
    //Input: A nonnegative integer $n$
    //Output: The value of $n!$
    if $n = 0$ return 1
    else return $F(n - 1) * n$
```

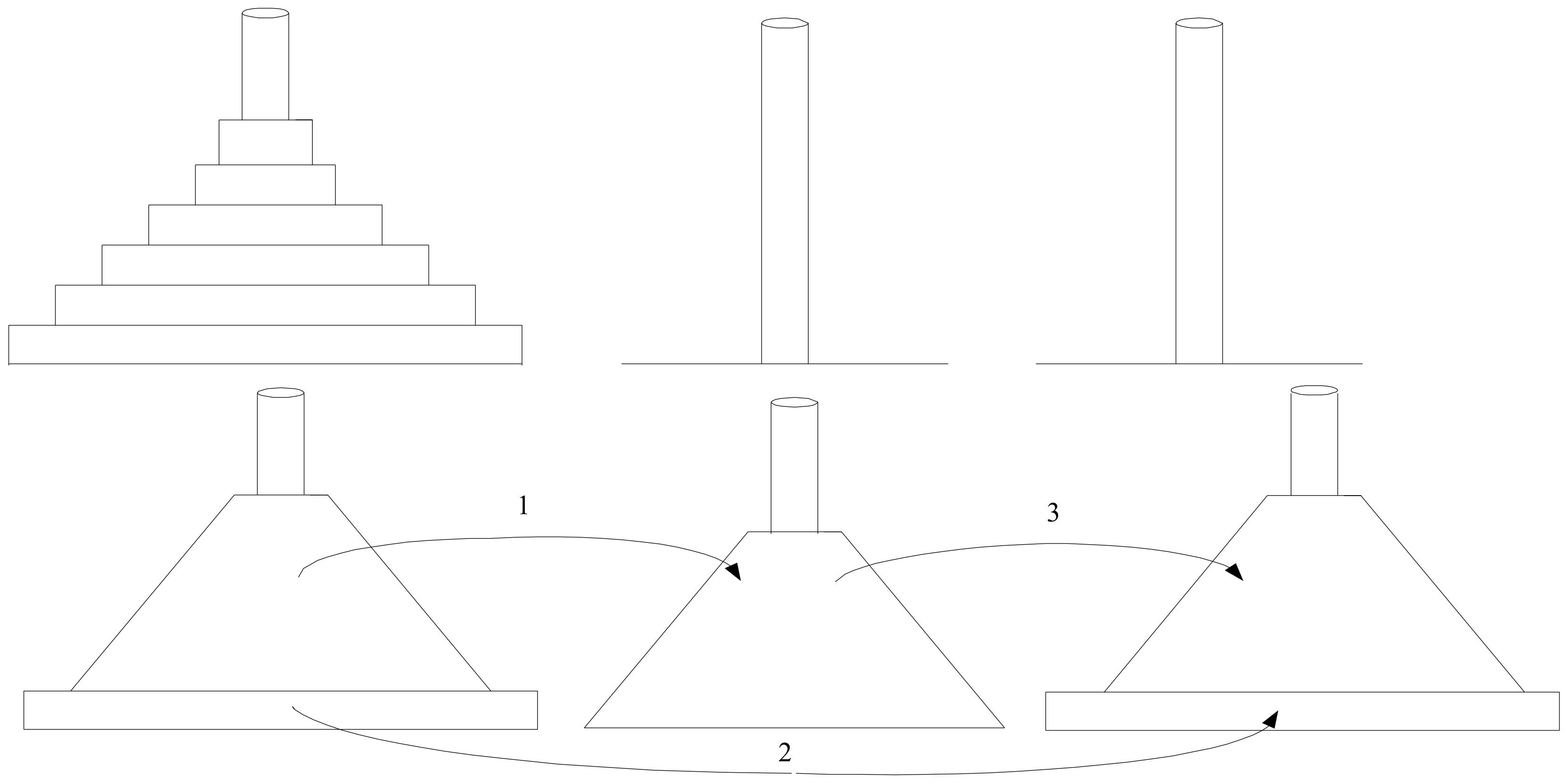
Size:

Basic operation:

Recurrence relation:

```
**if** $n = 0$ **return** $1$
**else return** $F(n-1) * n$
```

Example 2: The Tower of Hanoi Puzzle



Recurrence for number of moves:

n disks of different sizes and 3 pegs; the goal is to move all the disks to the third peg using the 2nd one as an auxiliary if necessary. Move on disk at a time and do not place a larger disk on top of a smaller one!

The Tower of Hanoi Puzzle

```
TOH(n, x, y, z)
{
   if (n >= 1)
   {
      // put (n-1) disk to z by using y
      TOH((n-1), x, z, y)

       // move larger disk to right place
       move:x-->y

      // put (n-1) disk to right place
      TOH((n-1), z, y, x)
   }
}
```

The Tower of Hanoi Puzzle

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

$$M(n) = 2M(n-1) + 1 \qquad \text{sub. } M(n-1) = 2M(n-2) + 1$$
$$= 2[2M(n-2) + 1] + 1 = 2^2M(n-2) + 2 + 1 \qquad \text{sub. } M(n-2) = 2M(n-3) + 1$$
$$= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3M(n-3) + 2^2 + 2 + 1.$$

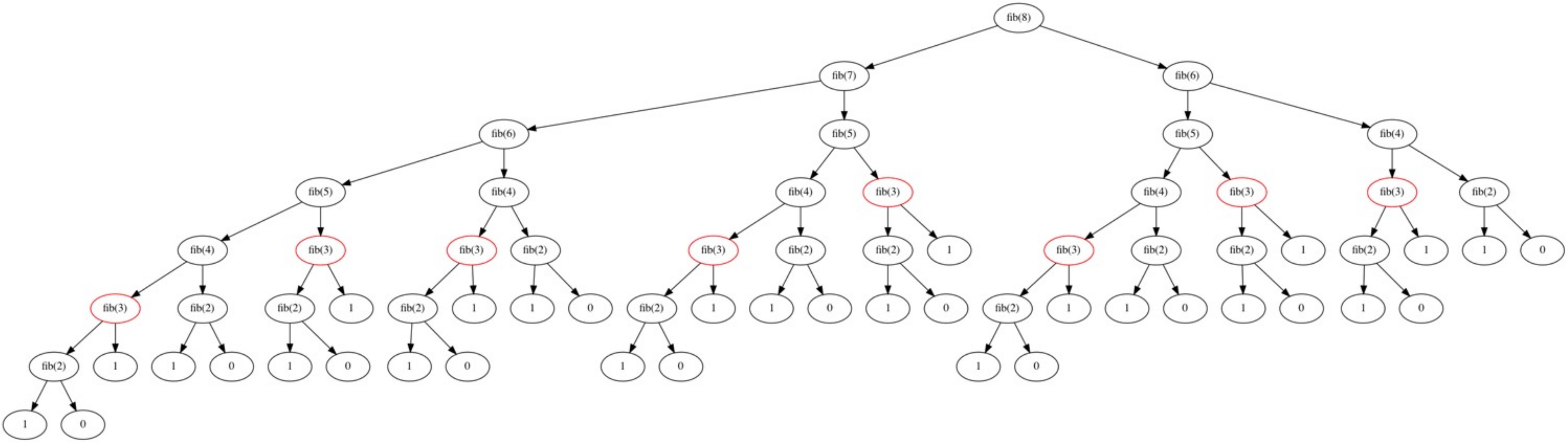
$$M(n) = 2^{n-1}M(n-(n-1)) + 2^{n-1} - 1$$
$$= 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.$$

…so what is Big Oh?

Fibonacci – why O(2^n)?

- What does this look like?

- The number of ops *approximates* a full binary tree, so…



Fibonacci’s hidden shame

“One should be careful with recursive algorithms because their succinctness may mask their inefficiency.”

Next week…

Brute force!

Chapter 3.1-3.4