

Lecture 3

Asymptotic Notations – Insertion Sort



o-notation

- The asymptotic upper bound provided by big O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not.
- We use o-notation to denote an upper bound that *is not asymptotically tight*.
- We formally define $o(g(n))$ as the set
 $o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$
For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.

- The definition of O -notation and o -notation are similar. The main difference is that in $f(n)=O(g(n))$, the bound $0 \leq f(n) \leq cg(n)$ holds for **some constant** $c>0$, but in $f(n)=o(g(n))$, the bound $0 \leq f(n) < cg(n)$ holds for **all constants** $c>0$.
- Intuitively, in the o -notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity; that is,

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

ω -notation

- We use ω -notation to denote a lower bound that is not asymptotically tight. One way to define it is by $f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$.
- Formally, however we define $\omega(g(n))$ as the set $\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$
- For example, $n^2/2 \in \omega(n)$, but $n^2/2 \notin \omega(n^2)$. The relation $f(n) \in \omega(g(n))$ implies that

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

if the limit exists that is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity.

- Some readers may find it strange that we should write, for example, $n = O(n^2)$.
- In literature, O – notation is sometimes used informally to describe ***asymptotic tight bound***, i.e. what we have done using Θ –notation.
- However, in this course, when we write $f(n) = O(g(n))$, we are merely (purely) claiming that some **constant multiple** of $g(n)$ is an ***asymptotic upper bound*** on $f(n)$, with no claim about how tight an upper bound it is.

➤ 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

➤ Best case

- Provides lower bound on running time

Input Size

- In general, time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of the program as a function of *size of its input*.
- **How we characterize input (for e.g.):**
 - **Sorting**: number of input items
 - **Multiplication**: total number of bits
 - If the input to the algorithm is **Graph**, the *input size* can be described by the : number of nodes & edges
 - Etc

Running Time

- The running time of an algorithm on a particular input is the number of primitive steps or operations executed.
- For the next coming lectures we will assume a constant amount of time is required to execute each line of our pseudo code.
- One line may take different amount of time than another line, but we shall assume that each execution of the i^{th} line takes time c_i , where c_i is the constant.

Example: Insertion Sort (pseudo code)

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

30	10	40	20
1	2	3	4

An Example: insertion Sort

30	10	40	20
1	2	3	4

$j = \emptyset$	$i = \emptyset$	$key = \emptyset$
$A[i] = \emptyset$	$A[i+1] = \emptyset$	



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

An Example: insertion Sort

30	10	40	20
1	2	3	4

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

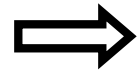


```
insertionSort(A, n) {  
    for j = 2 to n {  
        key = A[j]  
        i = j - 1;  
        while (i > 0) and (A[i] > key) {  
            A[i+1] = A[i]  
            i = i - 1  
        }  
        A[i+1] = key  
    }  
}
```

An Example: insertion Sort

30	30	40	20
1	2	3	4

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



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

An Example: insertion Sort

30	30	40	20
1	2	3	4

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

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



An Example: insertion Sort

30	30	40	20
1	2	3	4

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

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


⇒

An Example: insertion Sort

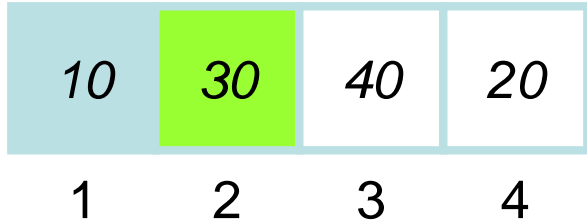
30	30	40	20
1	2	3	4

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

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



An Example: insertion Sort

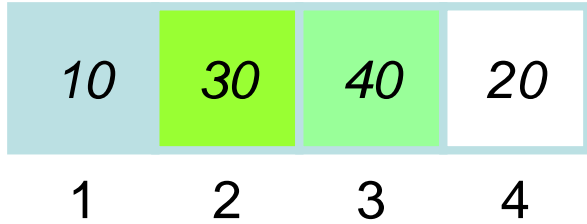


$j = 2$	$i = 0$	$key = 10$
$A[i] = \emptyset$	$A[i+1] = 10$	

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

➡

An Example: insertion Sort

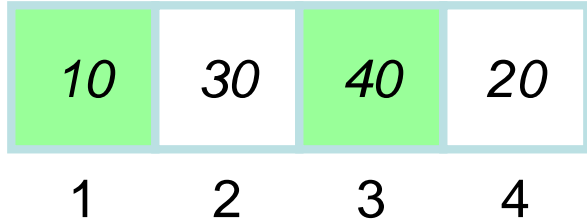


$j = 3$	$i = 0$	$key = 10$
$A[i] = \emptyset$	$A[i+1] = 10$	



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

An Example: insertion Sort



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



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

An Example: insertion Sort

10	30	40	20
1	2	3	4

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



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

An Example: insertion Sort

10	30	40	20
1	2	3	4

$j = 3$	$i = 2$	$\text{key} = 40$
$A[i] = 30$	$A[i+1] = 40$	




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

An Example: insertion Sort

10	30	40	20
1	2	3	4

$j = 3$	$i = 2$	$\text{key} = 40$
$A[i] = 30$	$A[i+1] = 40$	

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



An Example: insertion Sort

10	30	40	20
1	2	3	4

$j = 3$	$i = 2$	$\text{key} = 40$
$A[i] = 30$	$A[i+1] = 40$	

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

➡

An Example: insertion Sort

10	30	40	20
1	2	3	4

$j = 4$	$i = 2$	$\text{key} = 40$
$A[i] = 30$	$A[i+1] = 40$	



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

An Example: insertion Sort

10	30	40	20
1	2	3	4

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



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

An Example: insertion Sort

10	30	40	20
1	2	3	4

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



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

An Example: insertion Sort

10	30	40	20
1	2	3	4

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



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

An Example: insertion Sort

10	30	40	20
1	2	3	4

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



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

An Example: insertion Sort

10	30	40	40
1	2	3	4

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



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


An Example: insertion Sort

10	30	40	40
1	2	3	4

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




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

An Example: insertion Sort

10	30	40	40
1	2	3	4

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

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



An Example: insertion Sort

10	30	40	40
1	2	3	4

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

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

⇒

An Example: insertion Sort

10	30	40	40
1	2	3	4

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



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

An Example: insertion Sort

10	30	30	40
1	2	3	4

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




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

An Example: insertion Sort

10	30	30	40
1	2	3	4

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

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



An Example: insertion Sort

10	30	30	40
1	2	3	4

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

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


⇒

An Example: insertion Sort

10	30	30	40
1	2	3	4

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

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



An Example: insertion Sort

10	20	30	40
1	2	3	4

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

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

⇒

An Example: insertion Sort

10	20	30	40
1	2	3	4

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

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

Done!

- We start by presenting the insertion sort with the time “cost” of each statement and the number of times each statement is executed.
- For each value of $j = 2, 3, 4, \dots, n$, where $n =$ length of array (input), we say t_j *be the number of times the while loop test in above program is executed for that value of j .*

Insertion Sort

<u>Statement</u>	<u>cost</u>	<u>times</u>
InsertionSort(A, n) {		
for j = 2 to n {	C_1	n
key = A[j]	C_2	(n-1)
i = j - 1;	C_4	(n-1)
while (i > 0) and (A[i] > key) {	C_5	A
A[i+1] = A[i]	C_6	B
i = i - 1	C_7	C
}	0	
A[i+1] = key	C_8	(n-1)
}	0	
}		

Where

$$\mathbf{A} = \sum_{j=2}^n t_j$$

$$\mathbf{B} = \sum_{j=2}^n t_j - 1$$

$$\mathbf{C} = \sum_{j=2}^n t_j - 1$$

t_j means t_j

- So The running time of the algorithm is the sum of each statement executed. To compute $T(n)$, the total running time of INSERTION SORT, we sum the products of the cost and times columns, obtaining

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

- The best case occurs if array is already sorted. Thus $t_j = 1$ for all $j = 2, 3, 4, \dots, n$, and the best case running time would be

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1+c_2+c_4+c_5+c_8)n - (c_2+c_4+c_5+c_8) \end{aligned}$$

- This running time can be expressed as $an + b$ for constants a & b ; depends on the statement costs: thus it is a **linear function** of n .

- If array is in reverse sorted order, then worst case running time would be resulting.
- We have the mathematical formula:

$$\sum_{j=1}^n j = n \frac{n+1}{2}$$

- So we will have

$$\sum_{j=2}^n j = n \frac{n+1}{2} - 1$$

- Similarly we have

$$\sum_{j=2}^n j-1 = n \frac{n-1}{2}$$

- We find that worst case running time of Insertion sort is:

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5\left(n \frac{n+1}{2} - 1\right) + \\ c_6\left(n \frac{n-1}{2}\right) + c_7\left(n \frac{n-1}{2}\right) + c_8(n-1)$$

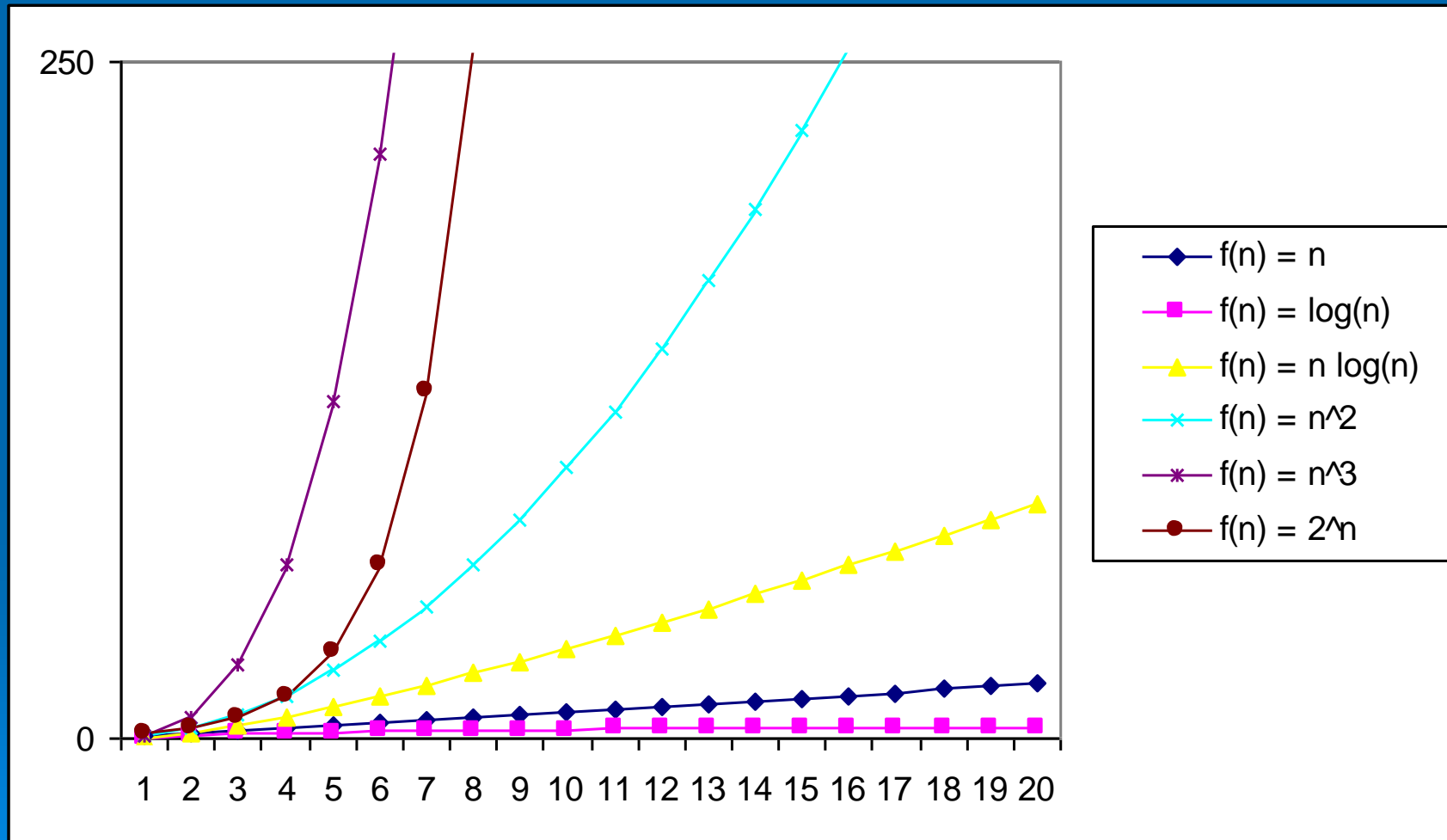
The worst case running time can be expressed as $an^2 + bn + c$; thus a **quadratic function** of n .

- We say Insertion Sort's run time is $O(n^2)$
 - Properly we should say run time is *in* $O(n^2)$
- Questions
 - Is InsertionSort $O(n)$?
 - Is InsertionSort $O(n^3)$?
 - Is InsertionSort $\Omega(n)$?

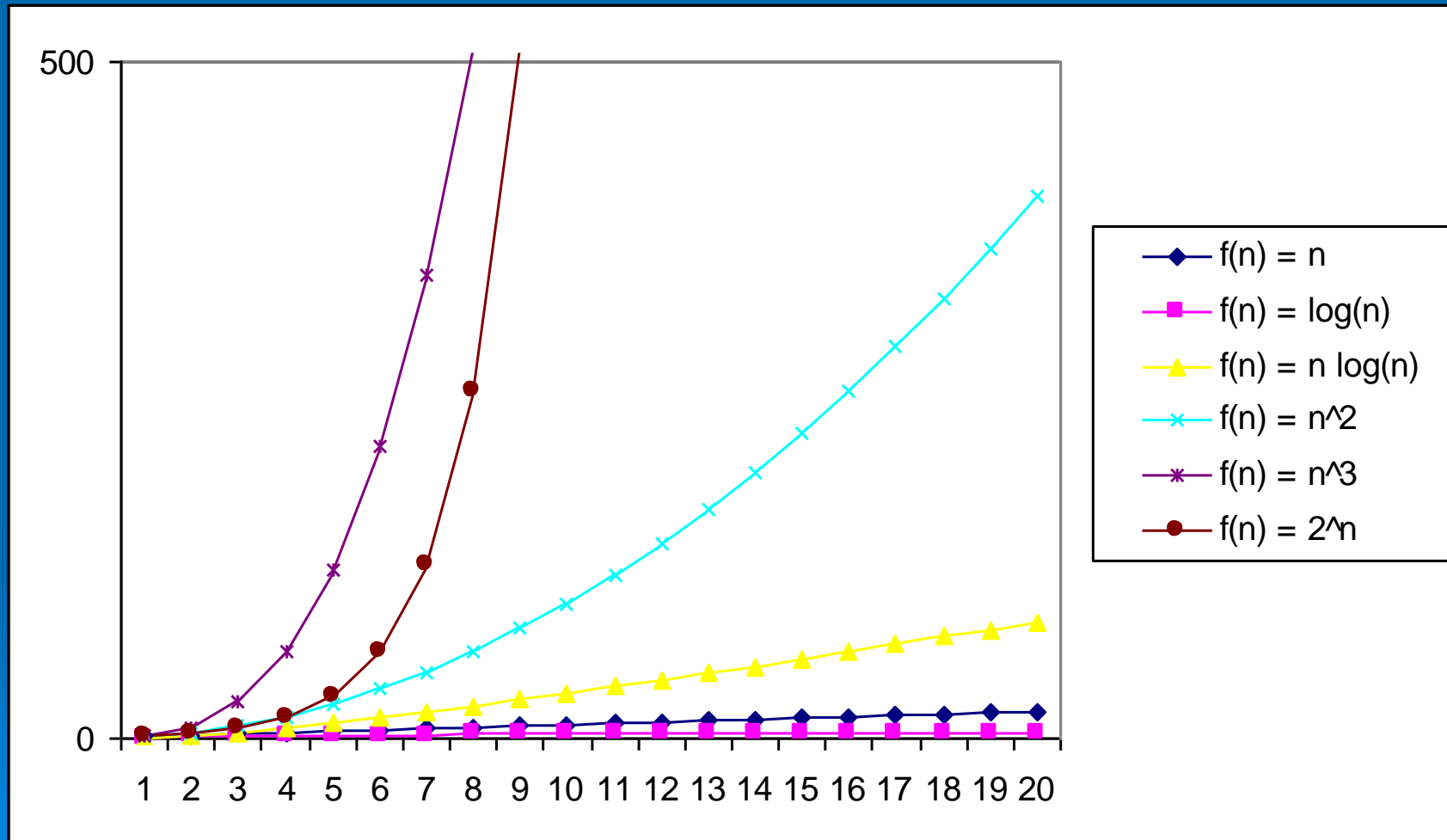
- We have studied insertion sort. The best case running time of this algorithm is $\Omega(n)$ which implies that the **running time** of insertion sort is $\Omega(n)$.
- So it means running time of insertion sort falls *between* $\Omega(n)$ and $O(n^2)$.
- It does not means that **running time** of insertion sort is $\Omega(n^2)$.
- It is not contradictory, however to say that **worst case running time** of insertion sort is $\Omega(n^2)$, since there is an input which causes the algorithm to take $\Omega(n^2)$ time.

- When we say that the **running time** (no modifier) of an algorithm is $\Omega(g(n))$, we mean that no matter what particular input of size n is chosen , the running time on that input is **at least constant times $g(n)$** for *sufficiently large n* .

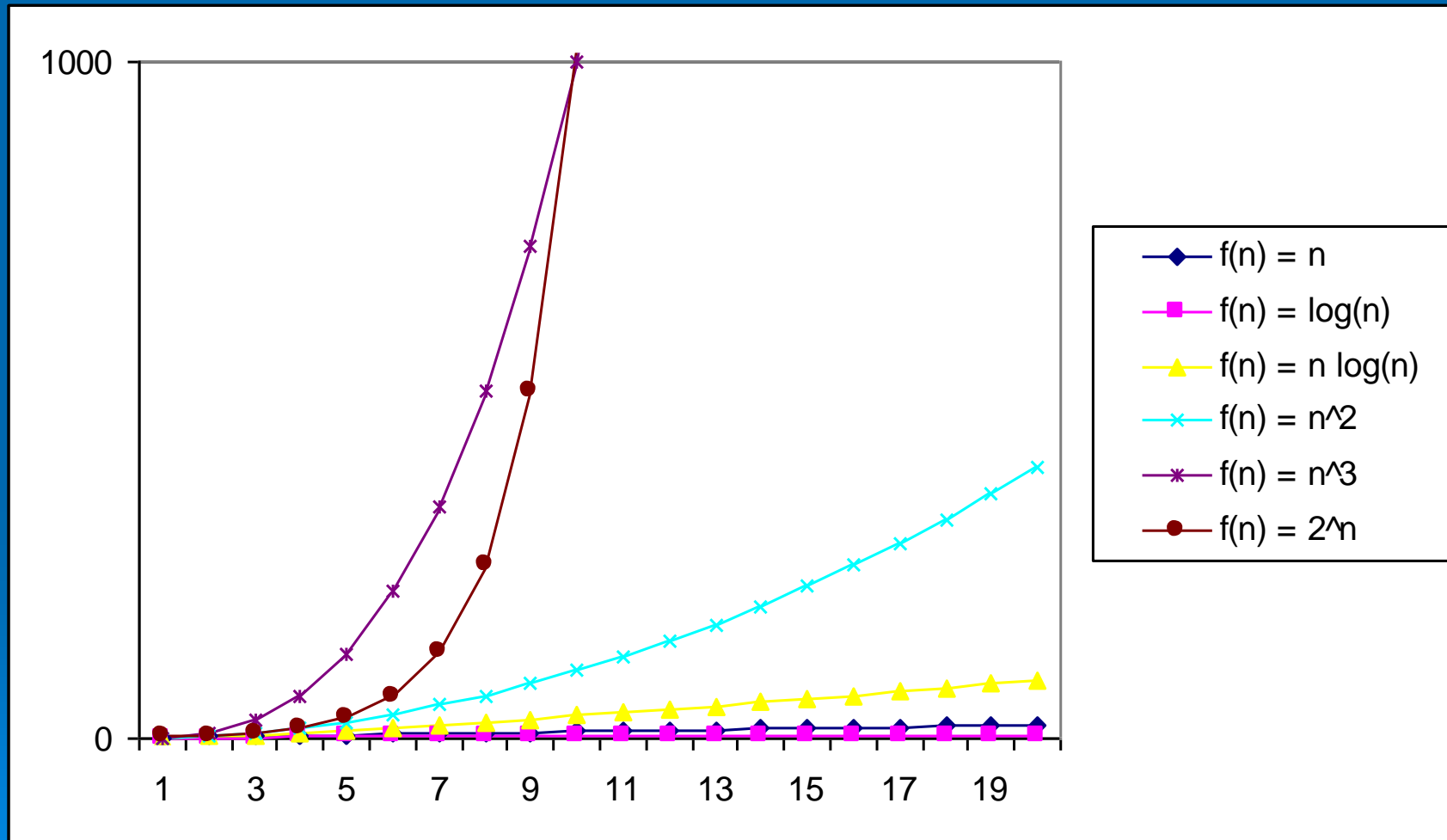
Practical Complexity



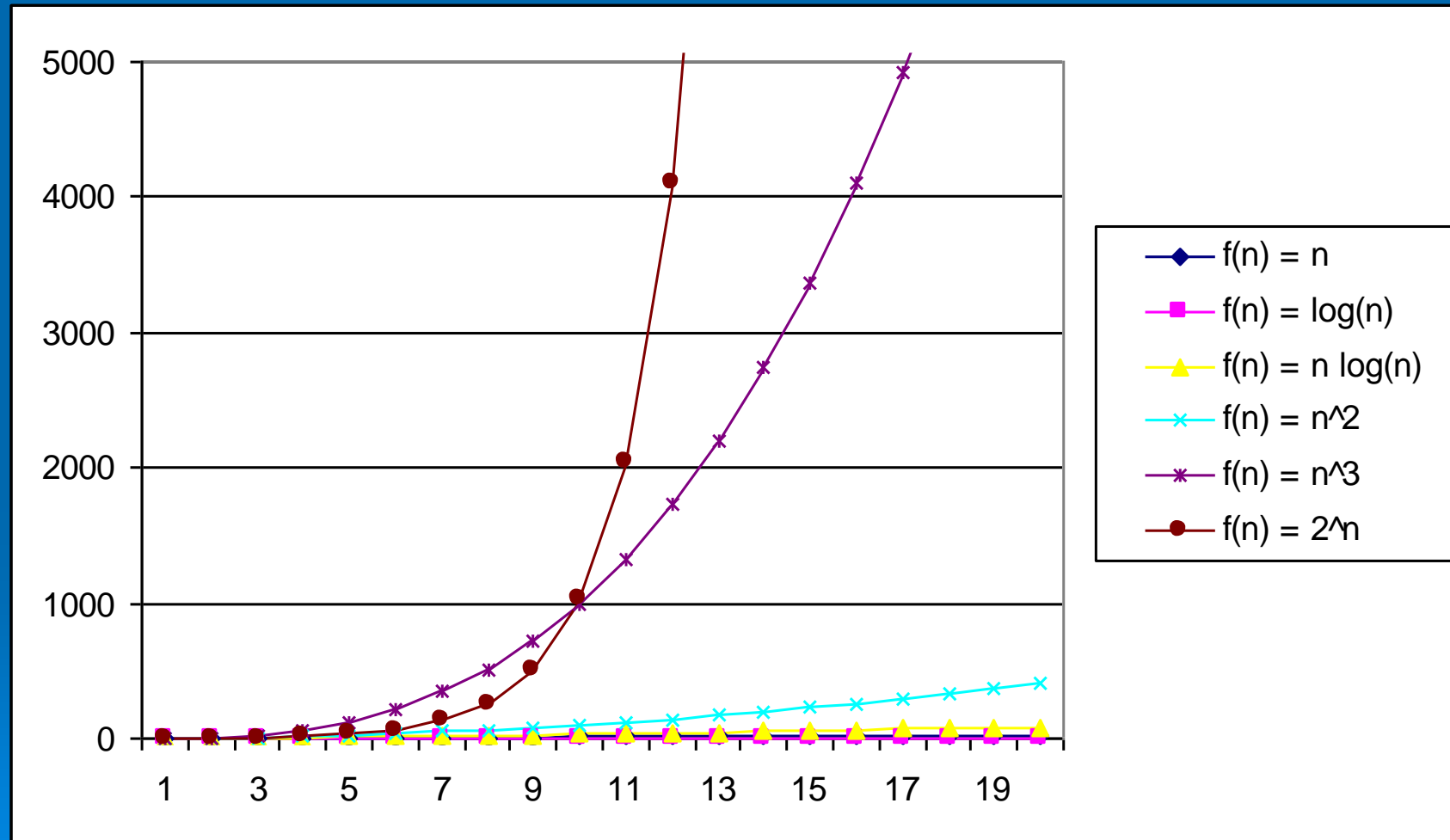
Practical Complexity



Practical Complexity



Practical Complexity



Useful Properties

➤ Transitivity

$$f(n) = \Theta(g(n)) \ \& \ g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$

$$f(n) = O(g(n)) \ \& \ g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$

$$f(n) = \Omega(g(n)) \ \& \ g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$

➤ Reflexivity

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

$$f(n) = O(f(n))$$

$$f(n) = \Omega(f(n))$$

➤ Symmetry

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n))$$

Floors and ceilings

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

$$\lceil n / 2 \rceil + \lfloor n / 2 \rfloor = n$$

$$\lceil \lceil n / a \rceil / b \rceil = \lceil n / ab \rceil$$

$$\lfloor \lfloor n / a \rfloor / b \rfloor = \lfloor n / ab \rfloor$$

$$\lceil a / b \rceil \leq (a + (b - 1)) / b$$

$$\lfloor a / b \rfloor \geq (a - (b - 1)) / b$$

$$\lfloor x \rfloor$$

Stands for “floor of x”
(greatest integer less than
or equal to x)

$$\lceil x \rceil$$

Stands for “ceiling of x”
(least integer greater than
or equal to x)