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

$o(g(n))=\{f(n)$: for any positive constant $c>0$, there exists a constant $n_0>0$ such that $0 \le f(n)<cg(n)$ for all $n \ge n_0\}$

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,

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

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 $\omega(g(n))=\{f(n)$: for any positive constant $c>0$, there exists a constant $n_0>0$ such that $0 \leq cg(n)<f(n)$ for all $n \geq n_0\}$
- For example, $n^2/2=\omega(n)$, but $n^2/2 \neq \omega(n^2)$. The relation $f(n)= \omega(g(n))$ implies that

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

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

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

```
jnsertjonSort(A, n) {
   for j = 2 to n {
        key = A[j]
        i = j - 1;
        whjle (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

```
jnsertjonSort(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

```
jnsertjonSort(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

```
jnsertjonSort(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] = ∅   A[i+1] = 30

```
jnsertjonSort(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

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

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

```
jnsertjonSort(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

1	2	3	4
10	*30*	*40*	*20*

j = 2   i = 0   key = 10
A[i] = ∅   A[i+1] = 10

```
jnsertjonSort(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

1	2	3	4
10	*30*	*40*	*20*

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

```
jnsertjonSort(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 key = 40
A[i] = ∅ A[i+1] = 10

```
jnsertjonSort(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

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

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

```
jnsertjonSort(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

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

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

```
jnsertjonSort(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 key = 40
A[i] = 30 A[i+1] = 40

```
jnsertjonSort(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   key = 40
A[i] = 30   A[i+1] = 40

```
jnsertjonSort(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 = 40
A[i] = 30   A[i+1] = 40

```
jnsertjonSort(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

```
jnsertjonSort(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

```
jnsertjonSort(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

```
jnsertjonSort(A, n) {
   for j = 2 to n {
        key = A[j]
        i = j - 1;
        whjle (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

```
jnsertjonSort(A, n) {
   for j = 2 to n {
        key = A[j]
        i = j - 1;
        whjle (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

```
jnsertjonSort(A, n) {
   for j = 2 to n {
        key = A[j]
        i = j - 1;
        whjle (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

```
jnsertjonSort(A, n) {
   for j = 2 to n {
        key = A[j]
        i = j - 1;
        whjle (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

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

An Example: insertion Sort

1	2	3	4
10	*30*	*40*	*40*

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

```
jnsertjonSort(A, n) {
   for j = 2 to n {
        key = A[j]
        i = j - 1;
        whjle (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

```
jnsertjonSort(A, n) {
   for j = 2 to n {
        key = A[j]
        i = j - 1;
        whjle (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

```
jnsertjonSort(A, n) {
  for j = 2 to n {
      key = A[j]
      i = j - 1;
      whjle (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

```
jnsertjonSort(A, n) {
   for j = 2 to n {
        key = A[j]
        i = j - 1;
        whjle (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

```
jnsertjonSort(A, n) {
   for j = 2 to n {
        key = A[j]
        i = j - 1;
        whjle (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

```
jnsertjonSort(A, n) {
   for j = 2 to n {
        key = A[j]
        i = j - 1;
        whjle (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

```
jnsertjonSort(A, n) {
  for j = 2 to n {
        key = A[j]
        i = j - 1;
        whjle (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

```
jnsertjonSort(A, n) {
   for j = 2 to n {
        key = A[j]
        i = j - 1;
        whjle (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…… 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

Statement	cost	times
`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

$$A = \sum_{j=2}^{n} tj$$

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

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

tj 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) = c1n + c2(n-1) + c4(n-1) +

$$c5\sum_{j=2}^{n} tj + c6\sum_{j=2}^{n}(tj-1) + c7\sum_{j=2}^{n}(tj-1) + c8(n-1)$$

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

$$T(n) = c1n + c2(n-1) + c4(n-1) + c5(n-1) + c8(n-1)$$
$$= (c1+c2+c4+c5+c8)n - (c2+c4+c5+c8)$$

- 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) = c1n + c2(n-1) + c4(n-1) + $c5(n\frac{n+1}{2}-1)$ +

$c6(n\frac{n-1}{2})$ + $c7(n\frac{n-1}{2})$ + c8(n-1)

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

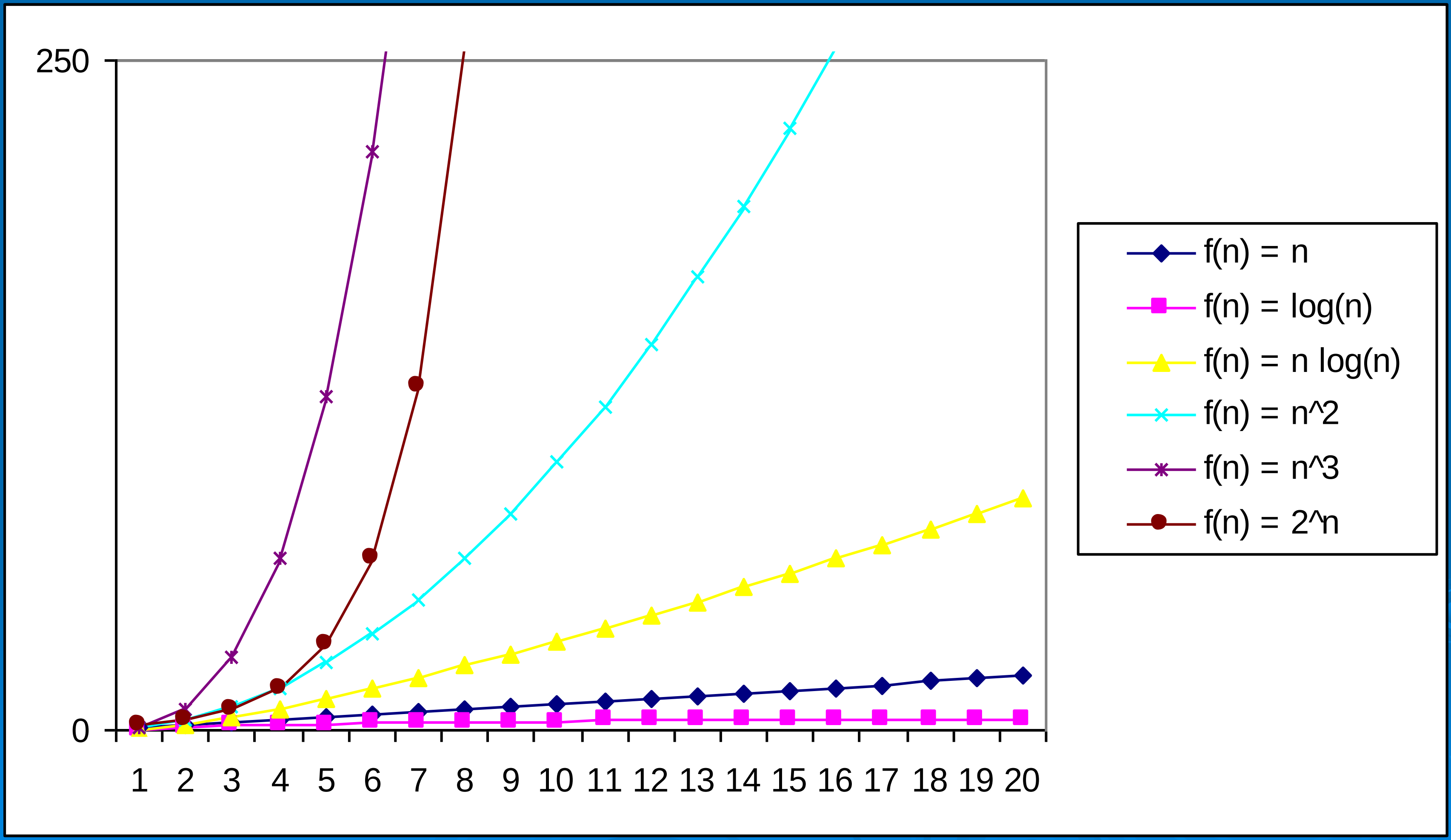
- We say Insertion Sort's run time is $O(n^2)$
  - Properly we should say run time is *in* $O(n^2)$

- Questions
  - Is InsertionSort $O(n)$?
  - Is InsertionSort $O(n^3)$?
  - Is InsertionSort $\Omega(n)$?

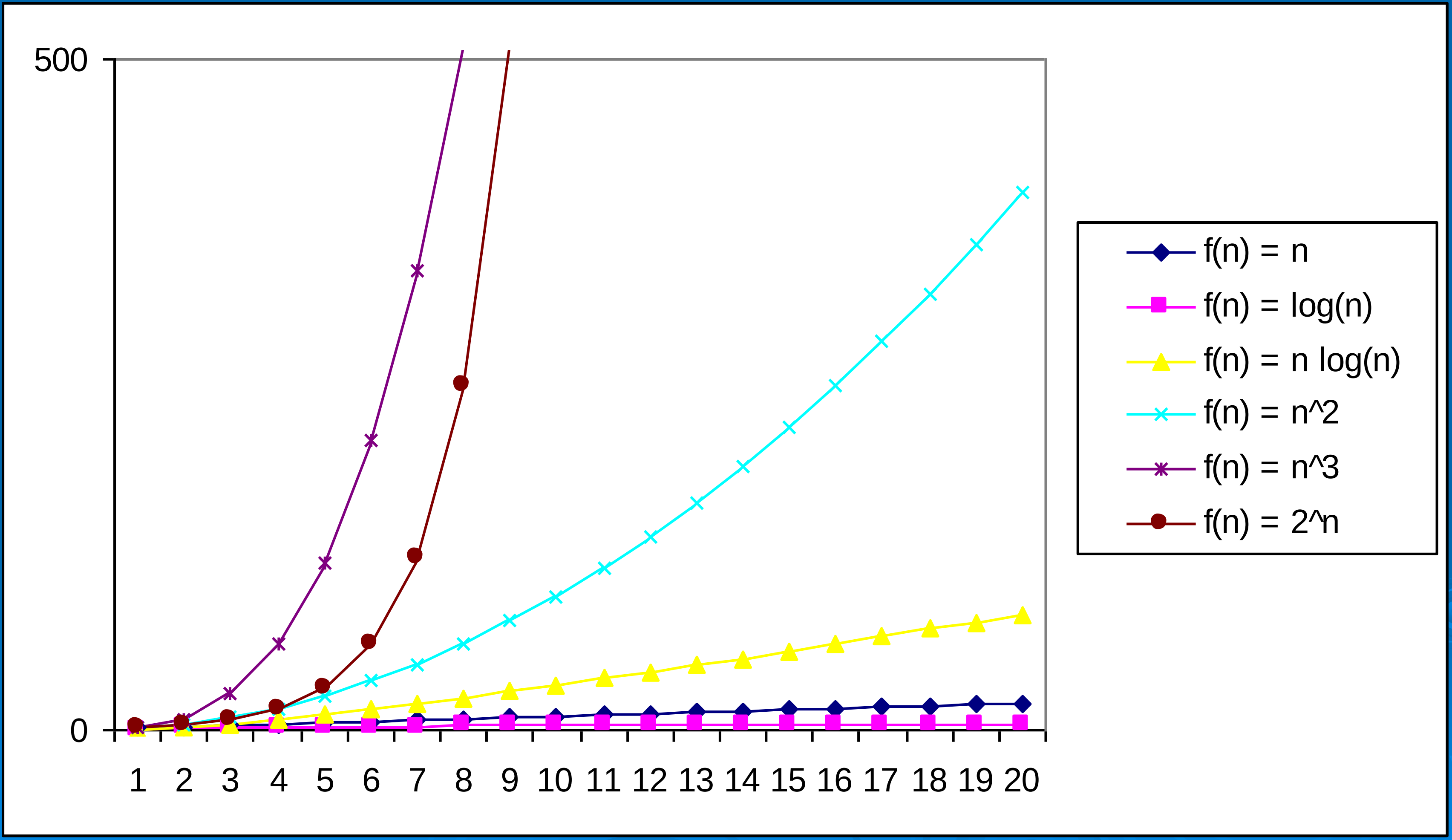
- We have studied insertion sort. The best case running time of this algorithm is $\Omega(n)$ which implies that the running time of insertion sort is $\Omega(n)$.
- So it means running time of insertion sort falls *between* $\Omega(n)$ and $O(n^2)$.
- It does not means that **running time** of insertion 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.

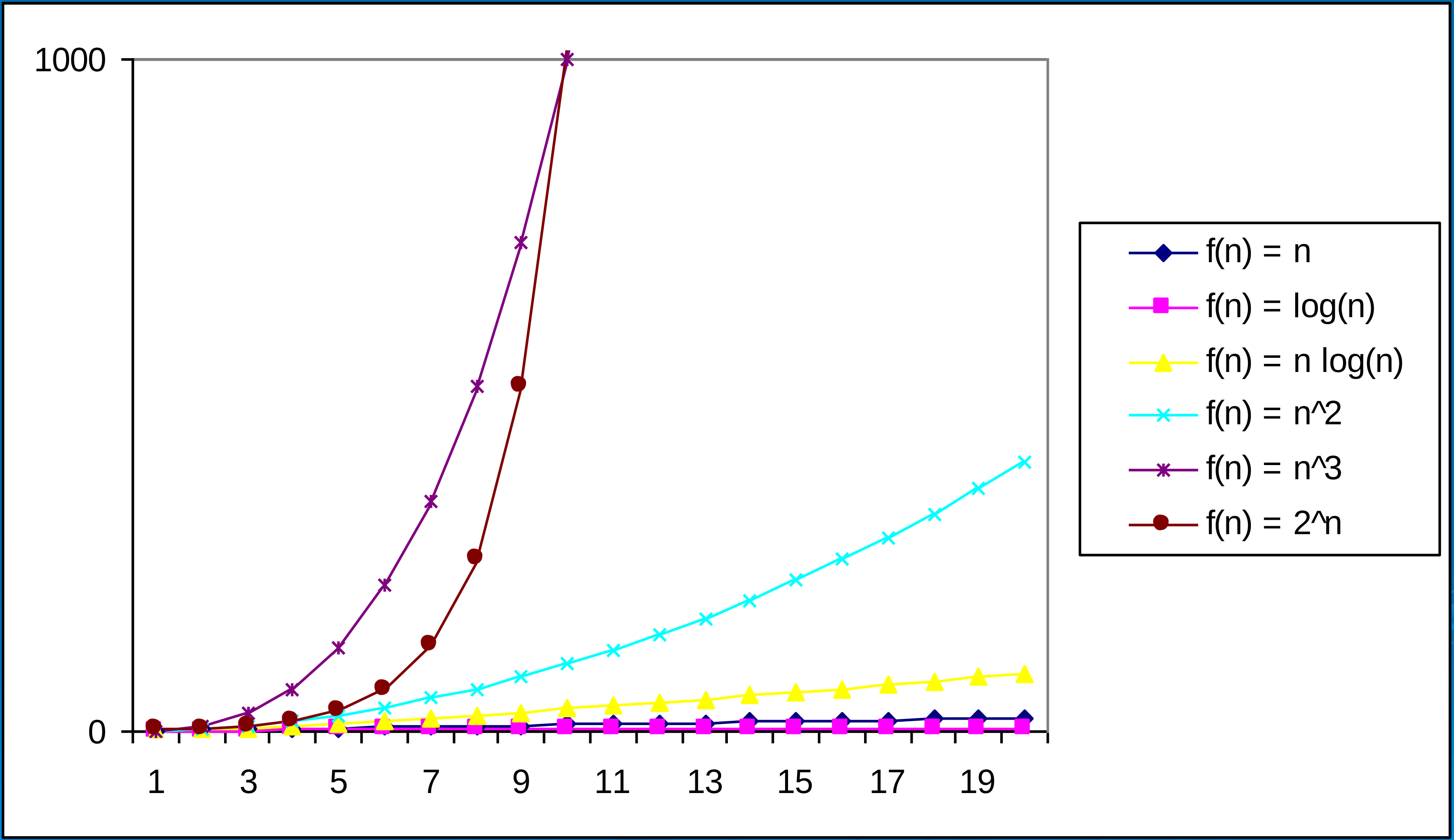
Practical Complexity



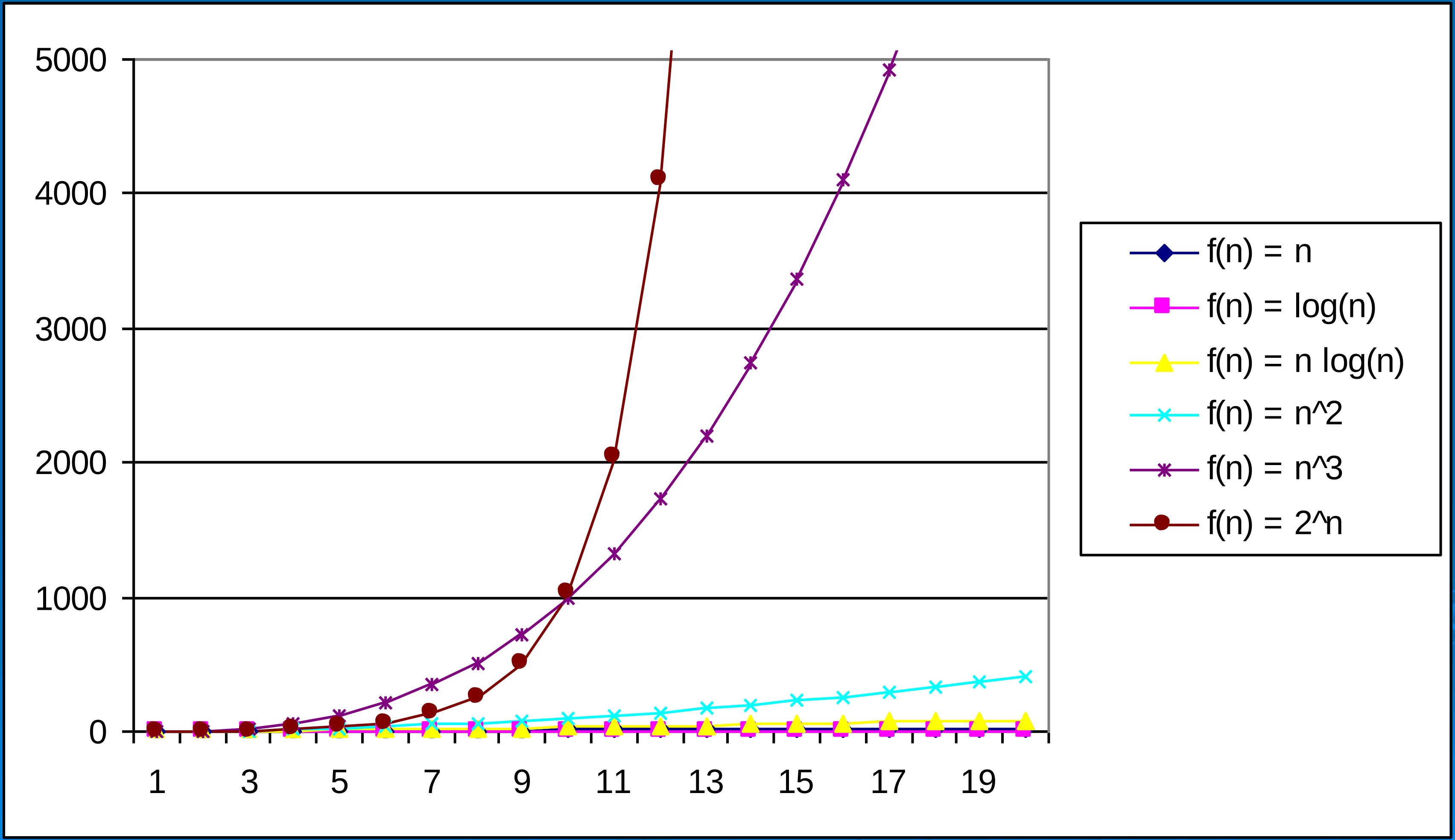
Practical Complexity



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

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

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

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

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

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

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

$\lfloor x \rfloor$

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)