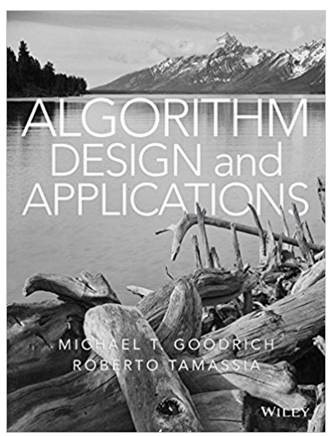
COMP26120: Introducing Complexity Analysis (2019/20)

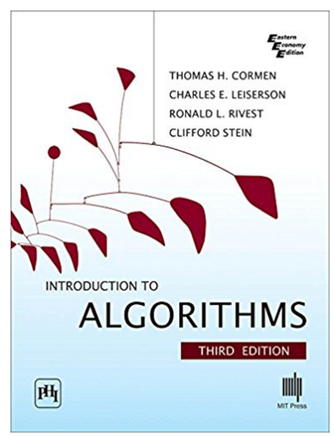
Lucas Cordeiro

lucas.cordeiro@manchester.ac.uk

Introducing Complexity Analysis

- Textbook:
  - *Algorithm Design and Applications,* Goodrich, Michael T. and Roberto Tamassia (Chapter 1)
  - *Introduction to Algorithms*, Cormen, Leiserson, Rivest, Stein (Chapters 2 and 3)





Motivating Example

- What does this code fragment represent? What is its complexity?

```
…
for(i = 0; i < N-1; i++) {
   for(j = 0; j < N-1; j++) {
     if (a[j]> a[j+1]) {
       t = a[j];
       a[j] = a[j+1];
       a[j+1] = t;
     }
   }
 }
…
```

- This is bubble sort
- There are two loops
- Both loops make *n-1* iterations so we have *(n-1)*(n-1)*
- The complexity is $O(n^2)$

Perform worst case analysis and ignore constants

Intended Learning Outcomes

- Define **asymptotic notation**, **functions**, and **running times**
- Analyze the **running time** used by an algorithm via **asymptotic analysis**
- Provide examples of asymptotic analysis using the **insertion sorting** algorithm

Asymptotic Performance

- In this course, we care most about ***asymptotic performance***
  - We focus on the **infinite set of large *n*** ignoring small values of *n*
    - The best choice for all, but minimal inputs
- How does the algorithm behave as the problem size gets huge?
  - Running time
  - Memory/storage requirements
  - Bandwidth/power requirements/logic gates/etc.

Asymptotic Notation

- By now, you should have an **intuitive feel** for asymptotic (big-O) notation:
  - *What does O(n) running time mean? $O(n^2)$? O(log n)?*
  - *How does asymptotic running time relate to asymptotic memory usage?*
- Our first task is to **define this notation more formally**

Search Problem (Arbitrary Sequence)

Input

- *sequence of numbers* $(a_1, \ldots a_n)$
- *search for a specific number* (q)

Output

- *index or NIL*

$a_1, a_2, a_3, \ldots, a_n;\ q \longrightarrow j$

2	*5*	*4*	*10*	*7;*	*5*	$\longrightarrow$	*2*
2	*5*	*4*	*10*	*7;*	*9*	$\longrightarrow$	*NIL*

Linear Search

```
j=1
while j<=length(A) and A[j]!=q
   do j++
if j<=length(A) then return j
else return NIL
```

- Worst case: ***f(n)=n***, average case: ***n/2***
- Can we do better using this approach?
  - this is a **lower bound** for the search problem in an **arbitrary sequence**

A Search Problem (Sorted Sequence)

Input

- *sequence of numbers* $(a_1 <= a_2, \ldots, a_{n-1} <= a_n)$
- *search for a specific number (q)*

Output

- *index or NIL*

$a_1, a_2, a_3, \ldots, a_n;\ q \longrightarrow j$

2	*4*	*5*	*7*	*10;*	*10*	*5*
2	*4*	*5*	*7*	*10;*	*8*	*NIL*

Did the sorted sequence help in the search?

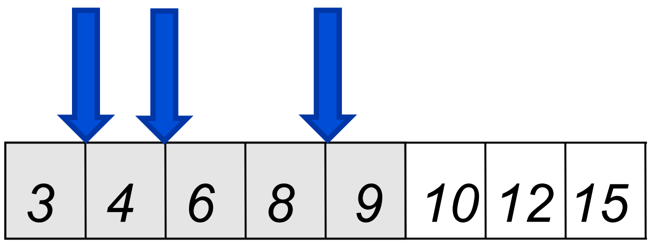
Binary Search

- Assume that the array is **sorted** and then perform **successive divisions**

```
left=1
right=length(A)
do
   j=(left+right)/2
   if A[j]==q then return j
   else if A[j]>q then right=j-1
   else left=j+1
while left<=right
return NIL
```

Binary Search Analysis

- How many times is the loop executed?
  - At each interaction, the number of positions **n** is cut in half
  - How many times do we cut in half **n** to reach 1?
    - o **$\lg_2 n$**



$$\lg_2 n = x \Leftrightarrow n = 2^x$$

$$\lg_2 8 = 3$$

Analysis of Algorithms (COMP15111)

- We perform the analysis concerning a **computational model**
- We will usually use a generic uniprocessor **random-access machine** (RAM)
  - All memory **equally expensive** to access
  - Instructions executed one after another (**no concurrent operations**)
  - All reasonable instructions take **unit time**
    - Except, of course, function calls
  - Constant word size
    - Unless we are explicitly manipulating bits

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 and 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
    - f = (g + h) – (i + j)
    - y = m * x + b
    - c = 5 / 9 * (t - 32)
    - z = f(x) + g(y)
- We can be more exact if needed

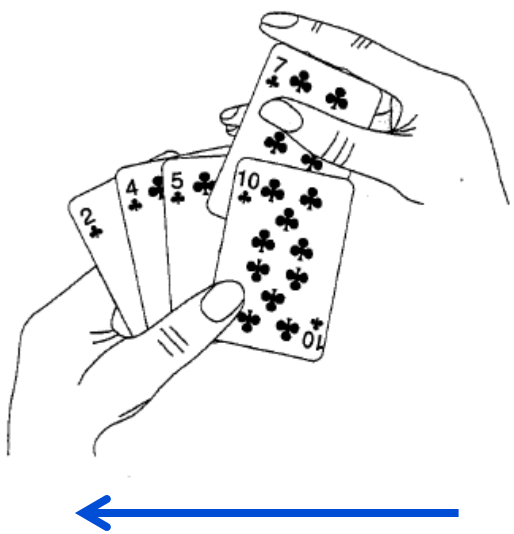
```
add t0, g, h # temp t0 = g + h
add t1, i, j # temp t1 = i + j
sub f, t0, t1 # f = t0 - t1
```

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

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

30	10	40	20
1	2	3	4

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

```
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

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

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

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

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

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

```
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

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

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

```
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

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

$i = 2$ $\quad$ $\mathbf{j = 0}$ $\quad$ $\text{key} = 10$

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

```
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

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

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

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

```
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

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

i = 2 j = 0 key = 10
A[j] = ∅ **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

10	30	40	20
1	2	3	4

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

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

```
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

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

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

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

```
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

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

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

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

```
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

10	30	40	20
1	2	3	4

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

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

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

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

```
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

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

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

```
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

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

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

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

$i = 4$ $j = 2$ **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

10	30	40	20
1	2	3	4

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

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	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

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

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

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

```
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

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

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

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

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

```
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

10	30	40	40
1	2	3	4

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

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

```
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

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

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

$A[j] = 30 \quad \mathbf{A[j+1] = 30}$

```
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

10	30	30	40
1	2	3	4

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

```
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

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

$i = 4 \quad j = 1 \quad key = 20$

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

```
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

10	30	30	40
1	2	3	4

$i = 4 \quad j = 1 \quad key = 20$

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

```
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

10	20	30	40
1	2	3	4

$i = 4 \quad j = 1 \quad key = 20$

$A[j] = 10 \quad \mathbf{A[j+1] = 20}$

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

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

```
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
  }
}
```

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

```
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 ***post-condition*** *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
  }
}
```

Invariant: A [1..i-1] consists of the elements originally in A [1..i-1], but in sorted order

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
  }
}
```

Termination: when i == n+1 we have A[1..i-1] which leads to A[1..n]

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

Analysing Insertion Sort

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

- What can *T* be?
  - **Best case** -- inner loop body never executed
  - $T = t_2 + t_3 + \ldots + t_n$

$$\sum_{j=2}^{n} t_j = t_2 + t_3 + \ldots + t_n = 1 + 1 + \ldots + 1 = n - 1$$

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

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7)$$

  - $T(n) = an - b$

Sum Review

Gaussian Closed Form can be defined as:

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

Thus, we have:

$$\sum_{j=2}^{n} j = 2 + 3 + \ldots + n = \frac{n(n+1)}{2} - 1$$

Similarly, we obtain:

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

Analysing Insertion Sort

- **Worst case --** inner loop body executed for all previous elements

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

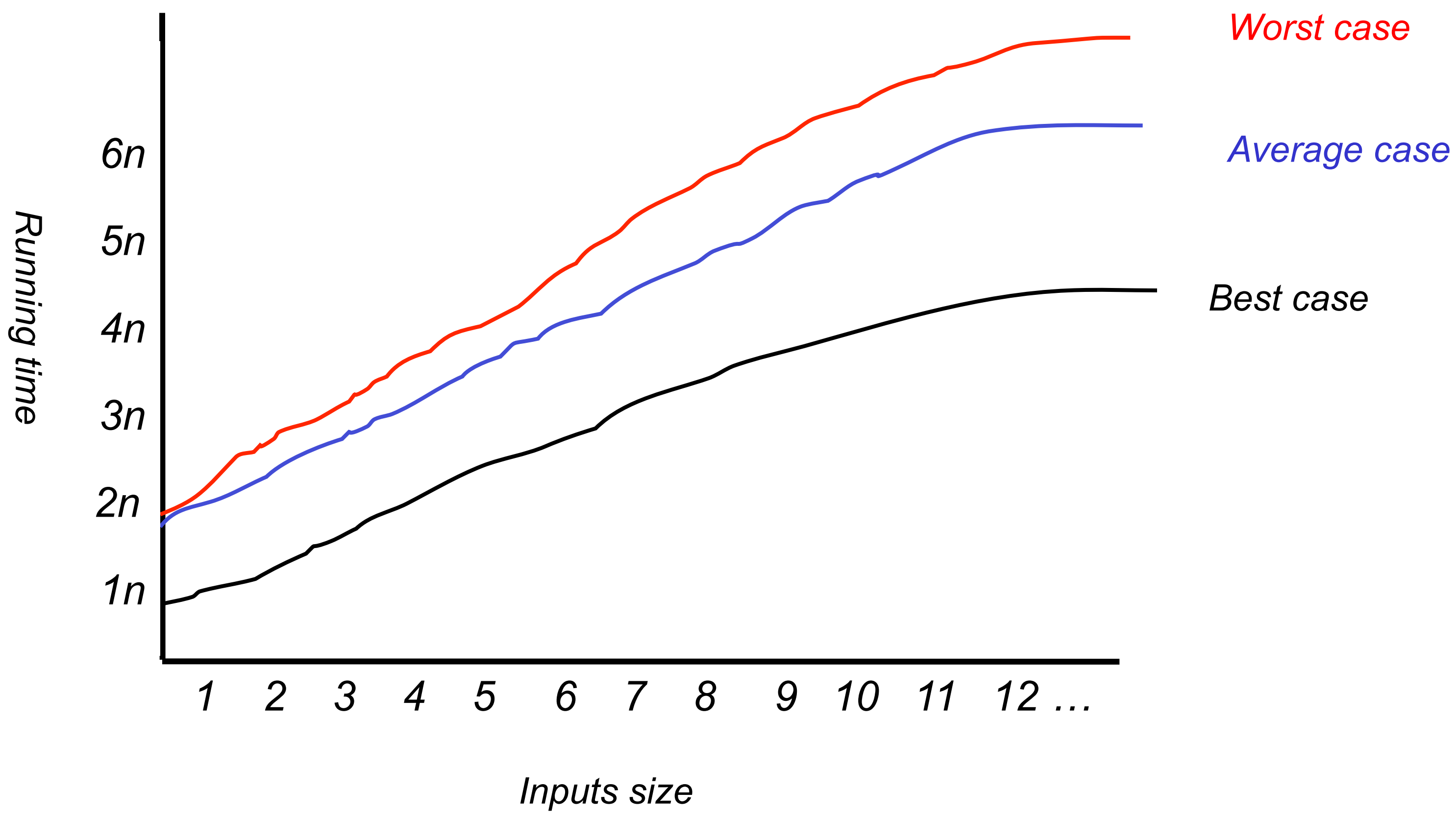
$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right)$$

$$+ c_7(n-1) = \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right)n - (c_2 + c_3 + c_4 + c_7)$$

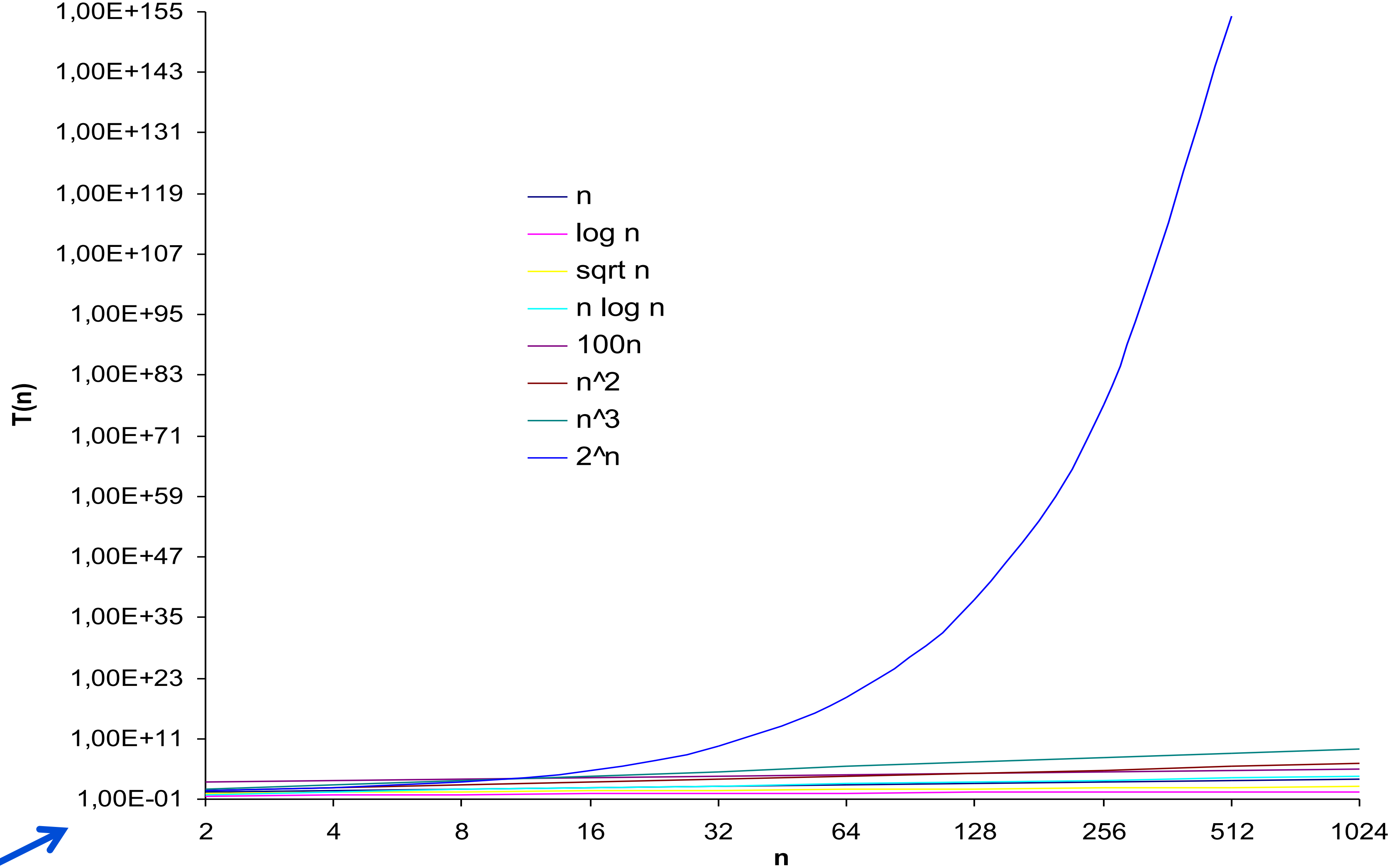
  - o *$T(n) = an^2 + bn - c$*
- **Average case**
  - o ???

Simplifications

- Abstract statement costs
- **Order of growth**



Growth Functions



"E" represents "times ten raised to the power of"

Scheduler

- the scheduler allows one thread to execute at a given time (emulate the execution on a *single core*)

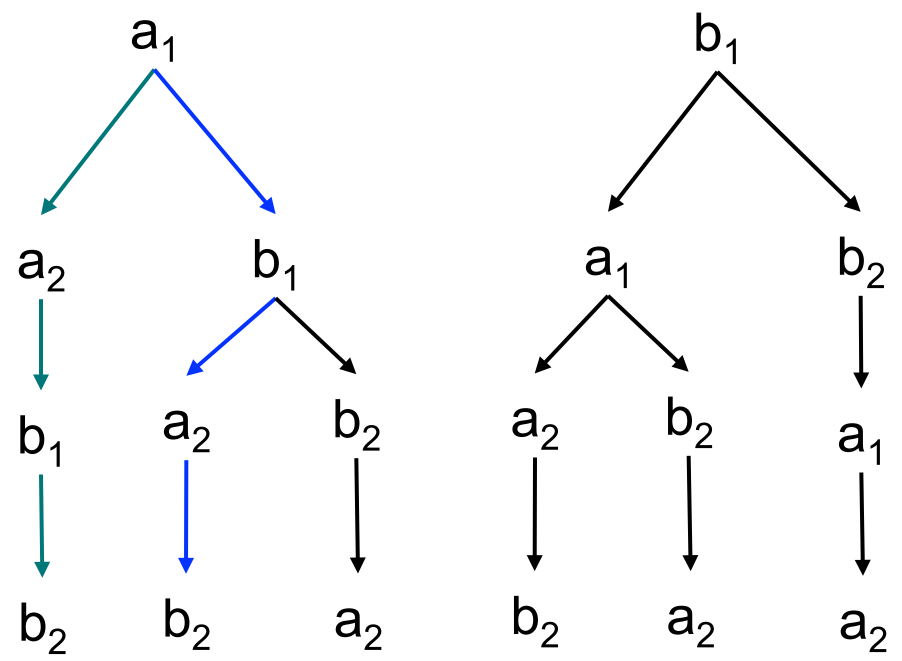
Thread T_1	Thread T_2
a_1	b_1
a_2	b_2

Thread interleavings:

$a_1; a_2; b_1; b_2$

$a_1; b_1; a_2; b_2$

…



- allow preemptions only before visible statements (global variables and synchronization points)

Exercise: Comparison of Running Times

- For each function *f(n)* and time *t*, determine the largest size *n* of a problem that can be solved in time *t*
  - the algorithm to solve the problem takes *f(n)* microseconds

	log n	$n^{1/2}$	***n***	n^2	n^3	2^n	***n!***
1 second							

Exercise: Comparison of Running Times

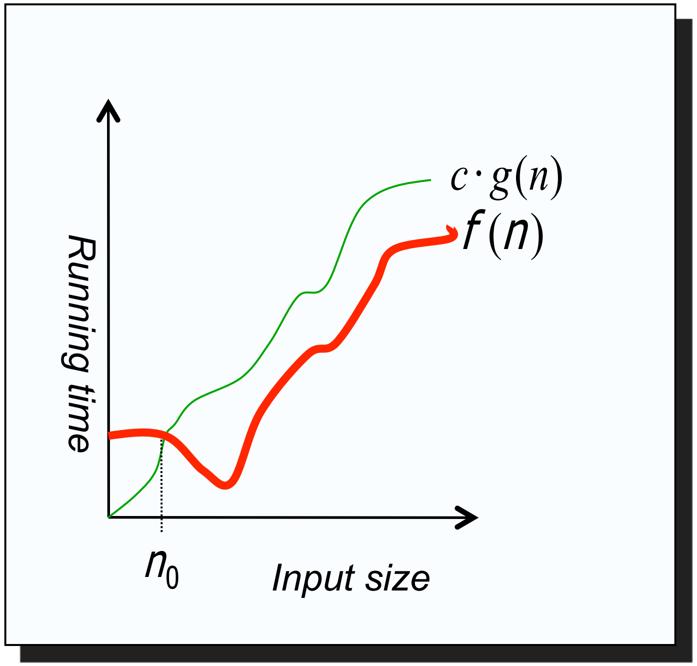
- For each function *f(n)* and time *t*, determine the largest size *n* of a problem that can be solved in time *t*
  - the algorithm to solve the problem takes *f(n)* microseconds

	1 Second	1 Minute	1 Hour	1 Day	1 Month	1 Year	1 Century
$\lg n$	$2^{1\times10^6}$	$2^{6\times10^7}$	$2^{3.6\times10^9}$	$2^{8.64\times10^{10}}$	$2^{2.592\times10^{12}}$	$2^{3.1536\times10^{13}}$	$2^{3.15576\times10^{15}}$
$\sqrt{n}$	1×10^{12}	3.6×10^{15}	1.29×10^{19}	7.46×10^{21}	6.72×10^{24}	9.95×10^{26}	9.96×10^{30}
n	1×10^6	6×10^7	3.6×10^9	8.64×10^{10}	2.59×10^{12}	3.15×10^{13}	3.16×10^{15}
$n\lg n$	62746	2801417	133378058	2755147513	71870856404	797633893349	6.86×10^{13}
n^2	1000	7745	60000	293938	1609968	5615692	56176151
n^3	100	391	1532	4420	13736	31593	146679
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

Assume a 30 day month and 365 day year

Upper Bound Notation

- InsertionSort's runtime is ***O(n²)***
  - runtime is *in* $O(n^2)$
  - Read O as "Big-O"
- In general, a function
  - $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that $f(n) \le c \cdot g(n)$ for all $n \ge n_0$



Insertion Sort Is $O(n^2)$

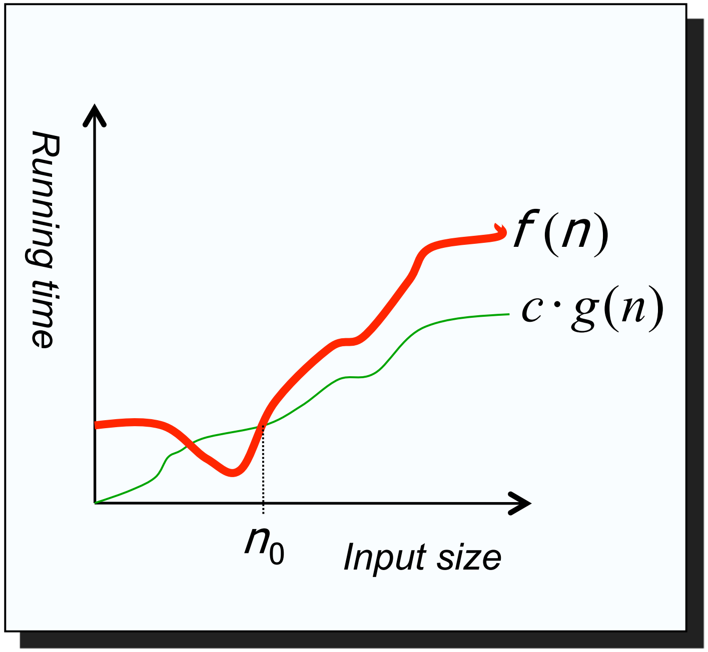
- Proof:
  - Use the formal definition of O to demonstrate that $an^2 + bn + c = O(n^2)$

$$O(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$

  - If any of a, b, and c are less than 0 replace the constant with its absolute value
    - $0 \le f(n) \le k \cdot g(n)$ for all $n \ge n_0$ *(k and n_0 must be positive)*
    - $0 \le an^2 + bn + c <= kn^2$
    - $0 \le a + b/n + c/n^2 <= k$
- Question
  - Is InsertionSort $O(n)$?

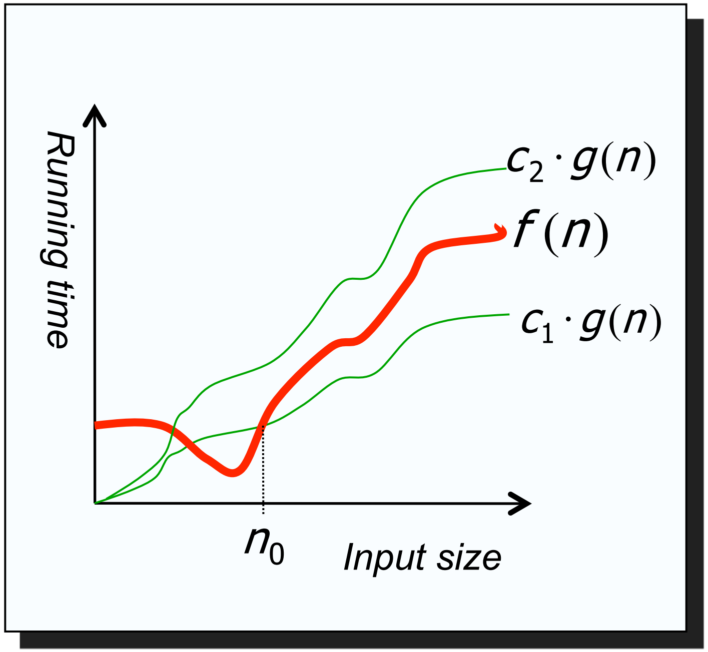
Lower Bound Notation

- InsertionSort's runtime is $\Omega(n)$
- In general, a function
  - $f(n)$ is $\Omega(g(n))$ if there exist positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n) \;\; \forall\, n \geq n_0$
- Proof:
  - Suppose runtime is an + b
    - o $0 \leq cn \leq an + b$
    - o $0 \leq c \leq a + b/n$



Asymptotic Tight Bound

- A function $f(n)$ is $\Theta(g(n))$ if there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ *for all* $n \geq n_0$
- Theorem
  - $f(n)$ is $\Theta(g(n))$ *iff* f(n) is both $O(g(n))$ and $\Omega(g(n))$



Exercise: Asymptotic Notation

- Use the formal definition of Θ

$$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \} .^1$$

to demonstrate that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

[1]Within set notation, a colon means "such that"

Exercise: Asymptotic Notation

- Use the formal definition of Θ

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}\ .^1$$

to demonstrate that $6n^3 \neq \Theta(n^2)$

Other Asymptotic Notations

- A function f(n) is o(g(n)) if ∃ positive constants *c* and n_0 such that
  $f(n) < c\, g(n)\ \forall\ n \geq n_0$
- A function f(n) is ω(g(n)) if ∃ positive constants *c* and n_0 such that
  $c\, g(n) < f(n)\ \forall\ n \geq n_0$
- Intuitively,
  - o() is like <
  - ω() is like >
  - Θ() is like =
  - O() is like ≤
  - Ω() is like ≥

Asymptotic Comparisons

- We can draw an analogy between the asymptotic comparison of **two functions f and g** and the comparison of **two real numbers a and b**
  - $f(n) = O(g(n))$ is like $a \leq g$
  - $f(n) = \Omega(g(n))$ is like $a \geq g$
  - $f(n) = \Theta(g(n))$ is like $a = g$
  - $f(n) = o(g(n))$ is like $a < g$
  - $f(n) = \omega(g(n))$ is like $a > g$
- Abuse of notation:
  - $f(n) = O(g(n))$ indica que $f(n) \in O(g(n))$

Exercise: Asymptotic Notation

Check whether these statements are true:

a) In the worst case, the insertion sort is $\Theta(n^2)$

b) $2^{2n} = O(2^n)$

c) $2^{n+1} = O(2^n)$

d) $\Theta(n) + \Theta(1) = \Theta(n)$

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

f) $O(n) \times O(n) = O(n)$

Summary

- Analyse the running time used by an algorithm via asymptotic analysis
  - asymptotic (O, Ω, Θ, o, ω) notations
  - use a generic uniprocessor random-access machine
  - Time and space complexity (input size)
  - Best, average and worst-case