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 $n$
    
      o 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 \((q)\)

\[ a_1, a_2, a_3, \ldots, a_n; \quad q \]

\[ 2 \quad 5 \quad 4 \quad 10 \quad 7; \quad 5 \]

\[ 2 \quad 5 \quad 4 \quad 10 \quad 7; \quad 9 \]

**Output**
- index or NIL

\[ j \]

\[ 2 \quad NIL \]
Linear Search

\[
j=1
\]
\[\text{while } j \leq \text{length}(A) \text{ and } A[j] \neq q \text{ do } j++\]
\[\text{if } j \leq \text{length}(A) \text{ then return } j\]
\[\text{else return NIL}\]

- 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 \(a_1 \leq a_2, \ldots, a_{n-1} \leq a_n\)
• search for a specific number \(q\)

Output
• index or NIL

\[ a_1, a_2, a_3, \ldots, a_n; \quad q \]

\[ \begin{align*}
2 & \quad 4 & \quad 5 & \quad 7 & \quad 10; & \quad 10 \\
2 & \quad 4 & \quad 5 & \quad 7 & \quad 10; & \quad 8
\end{align*} \]

Did the sorted sequence help in the search?
Binary Search

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

```python
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?
    - \( \lg_2 n \)

\[
\begin{align*}
\text{lg}_2 n &= x \iff n = 2^x \\
\text{lg}_2 8 &= 3
\end{align*}
\]
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
    - $f = (g + h) - (i + j)$
    - $y = m \times x + b$
    - $c = \frac{5}{9} \times (t - 32)$
    - $z = f(x) + g(y)$

- We can be more exact if needed

```plaintext
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”?  
    o Random (equally likely) inputs
    o Real-life inputs
An Example: Insertion Sort

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

<table>
<thead>
<tr>
<th>30</th>
<th>10</th>
<th>40</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

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

Initial array:

<table>
<thead>
<tr>
<th>30</th>
<th>10</th>
<th>40</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

At i = 2, j = 1, key = 10:

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

Updated array:

<table>
<thead>
<tr>
<th>10</th>
<th>30</th>
<th>40</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
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) {
      j = j - 1
    }
    A[j+1] = key
  }
}

<table>
<thead>
<tr>
<th>30</th>
<th>30</th>
<th>40</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

i = 2    j = 1    key = 10

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

```
An Example: Insertion Sort

30  30  40  20
1   2   3   4
```

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

$$\begin{array}{|c|c|c|c|}
\hline
30 & 30 & 40 & 20 \\
1 & 2 & 3 & 4 \\
\hline
\end{array}$$

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

$$\begin{array}{|c|c|c|}
\hline
\hline
\end{array}$$

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

i = 2  j = 0  key = 10

<table>
<thead>
<tr>
<th>30</th>
<th>30</th>
<th>40</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
An Example: Insertion Sort

<table>
<thead>
<tr>
<th>10</th>
<th>30</th>
<th>40</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

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

\[ \begin{array}{ccc}
\text{i} = 2 & \text{j} = 0 & \text{key} = 10 \\
\end{array} \]
An Example: Insertion Sort

\[
\begin{array}{cccc}
10 & 30 & 40 & 20 \\
1 & 2 & 3 & 4 \\
\end{array}
\]

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

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

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>30</th>
<th>40</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

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

i = 3  j = 0  key = 40
An Example: Insertion Sort

\[
\begin{array}{c|c|c|c}
10 & 30 & 40 & 20 \\
1 & 2 & 3 & 4 \\
\end{array}
\]

\[
i = 3 \quad j = 2 \quad \text{key} = 40 \\
\]

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

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>30</th>
<th>40</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

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

i = 3   j = 2   key = 40
An Example: Insertion Sort

<table>
<thead>
<tr>
<th>10</th>
<th>30</th>
<th>40</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

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

\begin{align*}
i &= 4 \\
j &= 2 \\
key &= 20 \\
A[j] &= 30 \\
A[j+1] &= 40
\end{align*}
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) {
            j = j - 1
        }
        A[j+1] = key
    }
}

i = 4    j = 3    key = 20

10 30 40 20
1 2 3 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) {
            j = j - 1
        }
        A[j+1] = key
    }
}

10 30 40 20
1 2 3 4

i = 4  j = 3  key = 20
An Example: Insertion Sort

<table>
<thead>
<tr>
<th>10</th>
<th>30</th>
<th>40</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

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

i = 4  j = 3  key = 20
An Example: Insertion Sort

<table>
<thead>
<tr>
<th>10</th>
<th>30</th>
<th>40</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

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

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

An Example: Insertion Sort

\[
\begin{array}{cccc}
10 & 30 & 40 & 40 \\
1 & 2 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{ccc}
i = 4 & j = 2 & key = 20 \\
\end{array}
\]

\[
\text{InsertionSort}(A, n) \{
\text{for } i = 2 \text{ to } n \{ \\
\text{key} = A[i] \\
\text{j} = i - 1; \\
\text{while } (j > 0) \text{ and } (A[j] > key) \{ \\
\text{A[j+1]} = A[j] \\
\text{j} = j - 1 \\
\} \\
\text{A[j+1]} = \text{key} \\
\}\}
\]
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) {
            j = j - 1
        }
        A[j+1] = key
    }
}
An Example: Insertion Sort

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>30</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

\[
i = 4 \quad j = 2 \quad \text{key} = 20
\]
\[
\]

\[
\text{InsertionSort}(A, n) \{
\begin{align*}
& \text{for } i = 2 \text{ to } n \{ \\
& \quad \text{key} = A[i] \\
& \quad j = i - 1; \\
& \quad \text{while } (j > 0) \text{ and } (A[j] > \text{key}) \{ \\
& \quad \quad A[j+1] = A[j] \\
& \quad \quad j = j - 1 \\
& \quad \}\}
\end{align*}
\]
\[
A[j+1] = \text{key}
\]
An Example: Insertion Sort

<table>
<thead>
<tr>
<th>10</th>
<th>30</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

i = 4  j = 2  key = 20

Here is the InsertionSort function with the example:

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

<table>
<thead>
<tr>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

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

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

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

\[
\]}

\[
A[j+1] = \text{key}
\]

\[
\}
\]

\[
T = t_2 + t_3 + \ldots + t_n \text{ where } t_i \text{ is number of while expression evaluations for the } i^{th} \text{ for loop iteration}
\]
Analysing Insertion Sort

- $T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4T + c_5(T - (n-1)) + c_6(T - (n-1)) + c_7(n-1)$

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

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

  $T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$

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

  - $T(n) = an - b$
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:

\[
\sum_{j=2}^{n} j = 2 + 3 + \cdots + n = \frac{n(n+1)}{2} - 1
\]

Similarly, we obtain:

\[
\sum_{j=2}^{n} (j-1) = \cdots = \frac{n(n+1)}{2} - n = \frac{n(n+1)-2n}{2} = \frac{n(n-1)}{2}
\]
Analysing Insertion Sort

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

\[
\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \quad \sum_{j=2}^{n} (j - 1) = \frac{n(n-1)}{2}
\]

\[
T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right)
\]

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

- **Average case**

  - ???

  - \( T(n) = an^2 + bn - c \)
Simplifications

- Abstract statement costs
- Order of growth

![Graph showing the relationship between inputs size and running time for best, average, and worst cases. The graph plots running time on the y-axis and inputs size on the x-axis. The best case is represented by the black line, the average case by the blue line, and the worst case by the red line. The graph shows how the running time increases with the inputs size.]
"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

<table>
<thead>
<tr>
<th>( \log n )</th>
<th>( n^{1/2} )</th>
<th>( n )</th>
<th>( n^2 )</th>
<th>( n^3 )</th>
<th>( 2^n )</th>
<th>( n! )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 second</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>( \log n )</th>
<th>1 Second</th>
<th>1 Minute</th>
<th>1 Hour</th>
<th>1 Day</th>
<th>1 Month</th>
<th>1 Year</th>
<th>1 Century</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sqrt{n} )</td>
<td>1 \times 10^{12}</td>
<td>3.6 \times 10^{15}</td>
<td>1.29 \times 10^{19}</td>
<td>7.46 \times 10^{21}</td>
<td>6.72 \times 10^{24}</td>
<td>9.95 \times 10^{26}</td>
<td>9.96 \times 10^{30}</td>
</tr>
<tr>
<td>( n )</td>
<td>1 \times 10^{6}</td>
<td>6 \times 10^{7}</td>
<td>3.6 \times 10^{9}</td>
<td>8.64 \times 10^{10}</td>
<td>2.59 \times 10^{12}</td>
<td>3.15 \times 10^{13}</td>
<td>3.16 \times 10^{15}</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>62746</td>
<td>2801417</td>
<td>133378058</td>
<td>2755147513</td>
<td>71870856404</td>
<td>797633893349</td>
<td>6.86 \times 10^{13}</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>1000</td>
<td>7745</td>
<td>60000</td>
<td>293938</td>
<td>1609968</td>
<td>5615692</td>
<td>56176151</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>100</td>
<td>391</td>
<td>1532</td>
<td>4420</td>
<td>13736</td>
<td>31593</td>
<td>146679</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>19</td>
<td>25</td>
<td>31</td>
<td>36</td>
<td>41</td>
<td>44</td>
<td>51</td>
</tr>
<tr>
<td>( n! )</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
</tbody>
</table>

Assume a 30 day month and 365 day year
Upper Bound Notation

- InsertionSort’s runtime is $O(n^2)$
  - 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 $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
Insertion Sort Is $O(n^2)$

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

\[
O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } \]
\[
0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}.
\]

  ▪ If any of $a$, $b$, and $c$ are less than 0 replace the constant with its absolute value
    o $0 \leq f(n) \leq k \cdot g(n)$ for all $n \geq n_0$ ($k$ and $n_0$ must be positive)
    o $0 \leq an^2 + bn + c \leq kn^2$
    o $0 \leq a + b/n + c/n^2 \leq k$

• 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 $an + b$
    - $0 \leq cn \leq an + b$
    - $0 \leq c \leq a + b/n$
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 $\Theta$

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

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

\(^1\text{Within set notation, a colon means “such that”}\)
Exercise: Asymptotic Notation

- Use the formal definition of $\Theta$

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

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

- A function $f(n)$ is $o(g(n))$ if $\exists$ positive constants $c$ and $n_0$ such that
  \[ f(n) < c \cdot g(n) \quad \forall \quad n \geq n_0 \]

- A function $f(n)$ is $\omega(g(n))$ if $\exists$ positive constants $c$ and $n_0$ such that
  \[ c \cdot g(n) < f(n) \quad \forall \quad n \geq n_0 \]

- Intuitively,
  - $o()$ is like $<$
  - $\omega()$ is like $>$
  - $\Theta()$ is like $=$
  - $O()$ is like $\leq$
  - $\Omega()$ is like $\geq$
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 ($O$, $\Omega$, $\Theta$, $o$, $\omega$) notations
  - use a generic uniprocessor random-access machine
  - Time and space complexity (input size)
  - Best, average and worst-case