COMP26120: Introducing Complexity Analysis (2018/19)

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 (COMP11212)

• *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 \( (n-1) \times (n-1) \)
- The complexity is \( O(n^2) \)

Perform worst case analysis and ignore constants
Intended Learning Outcomes

• Define asymptotic notation, functions, and running times
• Analyse the running time used by an algorithm via asymptotic analysis
• Provide examples of asymptotic analysis using the insertion sorting algorithm
• Review proof by induction
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 very small inputs
- How does the algorithm behave as the problem size gets very large?
  - 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? $O(n^2)$? $O(n \lg n)$?
  - How does asymptotic running time relate to asymptotic memory usage?

• Our first task is to define this notation more formally and completely
A Search Problem (Arbitrary Sequence)

**Input**
- sequence of numbers
- search for a specific number

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

**Output**
- number index or NIL

\[ \begin{array}{cccccc}
2 & 5 & 4 & 10 & 7; & 5 \\
2 & 5 & 4 & 10 & 7; & 9 \\
\end{array} \]

\[ \begin{array}{c}
2 \\
NIL \\
\end{array} \]
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
- search for a specific number

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

2 5 4 7 10; 5
2 5 4 7 10; 9

**Output**
- number index or NIL

\[ j \]

2
NIL

Did the sorted sequence help in the search?
Binary Search

• Assume that the array is ordered and then perform successive divisions of the search space

```plaintext
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
```
Analysis of the Binary Search

• How many times the loop is executed?
  ▪ At each interaction the number of positions $n$ is cut in half
  ▪ How many times do you cut in half $n$ to reach 1?
    ○ $\lg_2 n$

\[
\begin{align*}
3 & \quad 4 & \quad 6 & \quad 8 & \quad 9 & \quad 10 & \quad 12 & \quad 15 \\
\downarrow & & \downarrow & & \downarrow & & \\
\end{align*}
\]

\[
\lg_2 n = x \iff n = 2^x
\]

\[
\lg_2 8 = 3
\]
Analysis of Algorithms

• Analysis is performed with respect to 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
    o Except, of course, function calls
  ▪ Constant word size
    o Unless we are explicitly manipulating bits
Input Size

• Time and space complexity
  ▪ This is generally a function of the input size
    o E.g., sorting, multiplication
  ▪ How we characterize input size depends:
    o Sorting: number of input items
    o Multiplication: total number of bits
    o Graph algorithms: number of nodes & edges
    o 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
    - \( f = (g + h) - (i + j) \)
    - \( y = m \times x + b \)
    - \( c = 5/9 \times (t - 32) \)
    - \( z = f(x) + g(y) \)

• We can be more exact if need be

```
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

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
    }
}

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
</tbody>
</table>

i = 2  j = 1  key = 10
An Example: Insertion Sort

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

\[
\begin{array}{c|c|c|c}
& i = 2 & j = 1 & key = 10 \\
\end{array}
\]

\[
\text{InsertionSort}(A, n) \{ \\
\quad \text{for } i = 2 \text{ to } n \{ \\
\quad \quad \text{key} = A[i] \\
\quad \quad j = i - 1; \\
\quad \quad \text{while } (j > 0) \text{ and } (A[j] > key) \{ \\
\quad \quad \quad A[j+1] = A[j] \\
\quad \quad \quad j = j - 1 \\
\quad \quad \} \\
\quad \quad A[j+1] = key \\
\quad \}\}
\]
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>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
</tbody>
</table>

\[
i = 2 \quad j = 1 \quad \text{key} = 10 \\
\text{A}[j] = 30 \quad A[j+1] = 30
\]
An Example: Insertion Sort

<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>

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
    }
}
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
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

\begin{align*}
 & i = 3 \quad j = 0 \quad \text{key} = 40 \\
 & A[j] = \emptyset \quad A[j+1] = 10 \\
\end{align*}

\begin{align*}
\text{InsertionSort}(A, n) \{ \\
\quad & \text{for } i = 2 \text{ to } n \{ \\
\quad \quad & \text{key} = A[i] \\
\quad \quad & j = i - 1; \\
\quad \quad & \text{while } (j > 0) \text{ and } (A[j] > \text{key}) \{ \\
\quad \quad \quad & A[j+1] = A[j] \\
\quad \quad \quad & j = j - 1 \\
\quad \quad & \} \\
\quad \quad & A[j+1] = \text{key} \\
\quad & \} \\
\} 
\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 = 3    j = 0    key = 40
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 = 3    j = 2    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

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

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

\[
\begin{array}{cccc}
i = 4 & j = 2 & \text{key} = 40 \\
\end{array}
\]

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

$\begin{array}{*{20}c}
i = 4 & j = 2 & \text{key} = 20 \\
\end{array}$

```
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

<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
    }
}
```

<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>

i = 4  j = 3  key = 20

1  2  3  4

An Example: Insertion Sort
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>

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

\[i = 4 \quad j = 3 \quad \text{key} = 20\]
\[A[j] = 40 \quad A[j+1] = 20\]
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
    }
}
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>30</th>
<th>40</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

i = 4   j = 3   key = 20

1
2
3
4

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

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

$$\begin{array}{cccc}
i = 4 & j = 3 & \text{key} = 20 \\
\end{array}$$

```c
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
    }
}

1 2 3 4
10 30 40 40

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

\[
\]
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>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>

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

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

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
    }
}

\[
\begin{array}{cccc}
\text{i = 4} & \text{j = 1} & \text{key = 20} \\
\text{A[j] = 10} & \text{A[j+1] = 30} \\
\end{array}
\]
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
    }
}

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
</tbody>
</table>

i = 4  j = 1  key = 20
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>20</td>
<td>30</td>
<td>40</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

\[
i = 4 \quad j = 1 \quad \text{key} = 20 \\
\]

```
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
    }
    Done!
}
```
Insertion Sort

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
    }
}

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) {
            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..j-1] consists of the elements originally in A [1..j-1], but in sorted order
## Insertion Sort

<table>
<thead>
<tr>
<th>Statement</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>InsertionSort(A, n) {</code></td>
<td></td>
</tr>
<tr>
<td><code>for i = 2 to n {</code></td>
<td><code>c_1 n</code></td>
</tr>
<tr>
<td><code>  key = A[i]</code></td>
<td><code>c_2(n-1)</code></td>
</tr>
<tr>
<td><code>  j = i - 1;</code></td>
<td><code>c_3(n-1)</code></td>
</tr>
<tr>
<td><code>  while (j &gt; 0) and (A[j] &gt; key) {</code></td>
<td><code>c_4 T</code></td>
</tr>
<tr>
<td><code>     j = j - 1</code></td>
<td><code>c_6(T-(n-1))</code></td>
</tr>
<tr>
<td><code>  }</code></td>
<td><code>0</code></td>
</tr>
<tr>
<td><code>  A[j+1] = key</code></td>
<td><code>c_7(n-1)</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>0</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
</tbody>
</table>

\[
T = t_2 + t_3 + ... + t_n \text{ where } t_i \text{ is number of while expression evaluations for the } i^{th} \text{ for loop iteration}
\]
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
  - \( T = t_2 + t_3 + \ldots + t_n \)
  \[
  \sum_{j=2}^{n} t_j = t_2 + t_3 + \ldots + t_n = 1 + 1 + \ldots + 1 = n - 1
  \]
  \[
  T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)
  \]
  \[
  T(n) = (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7)
  \]
  - \( T(n) = an - b \)
Analysing Insertion Sort

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

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

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

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

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

- Average case

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

• Ignore actual and abstract statement costs
• **Order of growth** is the interesting measure

![Graph showing time complexity](image)

- **Worst case**
- **Average case**
- **Best case**
Upper Bound Notation

• InsertionSort’s run time is $O(n^2)$
  ▪ run time 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 $n \geq n_0$

• Formally
  ▪ $O(g(n)) = \{ f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0 \}$
Insertion Sort Is $O(n^2)$

• Proof:
  § Use the formal definition of $O$ to demonstrate that $an^2 + bn + c = O(n^2)$
  § If any of $a$, $b$, and $c$ are less than 0 replace the constant with its absolute value
    - $0 \leq f(n) \leq k \cdot g(n)$ for all $n \geq n_0$ ($k$ and $n_0$ must be positive)
    - $0 \leq an^2 + bn + c \leq kn^2$
    - $0 \leq a + b/n + c/n^2 \leq k$

• Question
  § Is InsertionSort $O(n)$?
Lower Bound Notation

• InsertionSort’s run time 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 $\frac{1}{2} n^2 - 3n = \Theta(n^2)$

Solution:

\[
0 \leq c_1 n^2 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2 \text{ for all } n \geq n_0
\]

\[
\frac{1}{2} - \frac{3}{n} \leq c_2
\]

\[
\frac{1}{2} \leq c_2
\]

\[
c_1 \leq \frac{1}{2} - \frac{3}{n}
\]

\[
c_1 \leq 1/14 \text{ for } n \geq 7
\]

Note that $c_1$ and $c_2$ must be positive constants.

For sufficiently large $n$, the term $\frac{1}{2}$ is kept.

$n=7$ is the smallest value for $c_1$ to be a positive constant.

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

Use the formal definition of $\Theta$

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

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

\[c_1 n^2 \leq 6n^3 \leq c_2 n^2 \text{ for all } n \geq n_0\]

\[6n \leq c_2 \therefore n \leq \frac{c_2}{6}\]

This can not be true for sufficiently large \(n\) since \(c_2\) must be a constant
Other Asymptotic Notations

- A function $f(n)$ is $o(g(n))$ if $\exists$ positive constants $c$ and $n_0$ such that
  $$f(n) < c \cdot g(n) \forall n \geq n_0$$
- A function $f(n)$ is $\omega(g(n))$ if $\exists$ positive constants $c$ and $n_0$ such that
  $$c \cdot g(n) < f(n) \forall n \geq n_0$$
- 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) $O(n) \times O(n) = O(n)$
Exercise: Asymptotic Notation

Check whether these statements are true:

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

b) $2^{2n} = O(2^n)$ \textit{False}

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

d) $\Theta(n) + \Theta(1) = \Theta(n)$ \textit{true}

e) $O(n^2) + O(n^2) = O(n^2)$ \textit{true}

f) $O(n) \times O(n) = O(n)$ \textit{False}
Proof by Induction (COMP11120)

- Why is it important to learn proof by induction?

```plaintext
unsigned int N=*
unsigned int i = 0;
long double x=2;
while( i < N ){
    x = ((2*x) - 1);
    i++;
}
assert( i == N );
assert(x>0);
```

How do we prove that this program is correct?
Proof by Induction of Programs

Handling Unbounded Loops with ESBMC 1.20

(Competition Contribution)

Jeremy Morse¹, Lucas
¹ Electronics and Co
² Electronic and Information
³ Department of Comp

Model Checking Embedded C Software using k-Induction and Invariants

Herbert Rocha*, Hussana Ismail², Lu
*Federal University of Roraima,
E-mail: herbert.rocha@ufrr.br
loscardeiro@ufam.edu.br

Abstract—We present a proof by induction algorithm, which
combines k-induction with invariants to model check embedded
C software with unbounded and unbounded loops. The k-induction
algorithm consists of three cases: in the base case, we aim to
find a counterexample with up to k loop unwindings; in the forward
condition, we check whether loops have been fully unraveled
and that the safety property holds in all states reachable within k
unwindings; and in the inductive step, we check that whenever
x holds for k unwindings, it also holds after the next unwinding
of the system. For each step of the k-induction algorithm, we infer
invariants using affine constraints (i.e., polytope) to specify pre- and post-conditions. Experimental results show that our approach
can handle wide use safety properties in typical embedded
software applications from telecommunications, control systems,
and medical devices; we demonstrate an improvement of the
induction algorithm effectiveness if compared to other approaches.

1 Overview

ESBMC is a context-bounded single- and multi-threaded C/C
ESBMC can only be used to verify programs, unless we know,
there is generally not the c prove safety properties in bound
The details of ESBMC are dist. the differences to the version 1
on the combination of the k-in

2 Differences to ESBM

Except for the loop handling
version. The main changes co.
where we replaced CBMC’s
where we replaced the name a

Proof by Induction of Programs

Handling Unbounded Loops with ESBMC 1.20

(Competition Contribution)

Jeremy Morse¹, Lucas
¹ Electronics and Co
² Electronic and Information
³ Department of Comp

Model Checking Embedded C Software using k-Induction and Invariants

Herbert Rocha*, Hussana Ismail², Lu
*Federal University of Roraima,
E-mail: herbert.rocha@ufrr.br
loscardeiro@ufam.edu.br

Abstract—We present a proof by induction algorithm, which
combines k-induction with invariants to model check embedded
C software with unbounded and unbounded loops. The k-induction
algorithm consists of three cases: in the base case, we aim to
find a counterexample with up to k loop unwindings; in the forward
condition, we check whether loops have been fully unraveled
and that the safety property holds in all states reachable within k
unwindings; and in the inductive step, we check that whenever
x holds for k unwindings, it also holds after the next unwinding
of the system. For each step of the k-induction algorithm, we infer
invariants using affine constraints (i.e., polytope) to specify pre- and post-conditions. Experimental results show that our approach
can handle wide use safety properties in typical embedded
software applications from telecommunications, control systems,
and medical devices; we demonstrate an improvement of the
induction algorithm effectiveness if compared to other approaches.

1 Overview

ESBMC is a context-bounded single- and multi-threaded C/C
ESBMC can only be used to verify programs, unless we know,
there is generally not the c prove safety properties in bound
The details of ESBMC are dist. the differences to the version 1
on the combination of the k-in

2 Differences to ESBM

Except for the loop handling
version. The main changes co.
where we replaced CBMC’s
where we replaced the name a

Proof by Induction of Programs

Handling Unbounded Loops with ESBMC 1.20

(Competition Contribution)

Jeremy Morse¹, Lucas
¹ Electronics and Co
² Electronic and Information
³ Department of Comp

Model Checking Embedded C Software using k-Induction and Invariants

Herbert Rocha*, Hussana Ismail², Lu
*Federal University of Roraima,
E-mail: herbert.rocha@ufrr.br
loscardeiro@ufam.edu.br

Abstract—We present a proof by induction algorithm, which
combines k-induction with invariants to model check embedded
C software with unbounded and unbounded loops. The k-induction
algorithm consists of three cases: in the base case, we aim to
find a counterexample with up to k loop unwindings; in the forward
condition, we check whether loops have been fully unraveled
and that the safety property holds in all states reachable within k
unwindings; and in the inductive step, we check that whenever
x holds for k unwindings, it also holds after the next unwinding
of the system. For each step of the k-induction algorithm, we infer
invariants using affine constraints (i.e., polytope) to specify pre- and post-conditions. Experimental results show that our approach
can handle wide use safety properties in typical embedded
software applications from telecommunications, control systems,
and medical devices; we demonstrate an improvement of the
induction algorithm effectiveness if compared to other approaches.

1 Overview

ESBMC is a context-bounded single- and multi-threaded C/C
ESBMC can only be used to verify programs, unless we know,
there is generally not the c prove safety properties in bound
The details of ESBMC are dist. the differences to the version 1
on the combination of the k-in

2 Differences to ESBM

Except for the loop handling
version. The main changes co.
where we replaced CBMC’s
where we replaced the name a
Review: Proof by Induction

• Show that a property $P$ is true for $n \geq k$
  ▪ Base case
  ▪ Step case or inductive step

• Suppose that
  ▪ $P(k)$ is true for a fixed constant $k$
    ○ Often $k = 0$
  ▪ $P(n) \implies P(n+1)$ for all $n \geq k$

• Then $P(n)$ is true for all $n \geq k$
Induction Example: Our Unbounded loop program

• Suppose that one wants to prove that $P : x > 0$

• In the base case, it holds initially:

$$N = * \land i = 0 \land x = 2 \implies x > 0.$$  
\hspace{1cm} \text{initial condition}

• In the inductive step, whenever $P$ holds for $k$ loop unwindings, it also holds for $k + 1$ steps

$$x > 0 \land x = 2 \land x' = 2 \ast x - 1 \land i' = i + 1 \implies x' > 0.$$  
\hspace{1cm} \text{transition relation}
Induction Example: Gaussian Closed Form

• Prove \( 0+1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} \)

  - Basis: If \( n = 0 \), then \( 0 = 0(0+1)/2 \)
  - Inductive hypothesis: Assume that \( 1 + 2 + 3 + \ldots + k = \frac{k(k+1)}{2} \)
  - Inductive step: show that if \( P(k) \) holds, then also \( P(k+1) \) holds

\[
(0+1+2+\ldots+k)+(k+1) = \frac{(k+1)((k+1)+1)}{2}.
\]

  - Using the induction hypothesis that \( P(k) \) holds:

\[
k\frac{k+1}{2} + (k+1).
\]

\[
k\frac{k+1}{2} + (k+1) = \frac{k(k+1)+2(k+1)}{2} = \frac{k^2 +3k+2}{2} = \frac{(k+1)(k+2)}{2}
\]

\[
= \frac{(k+1)((k+1)+1)}{2} \quad \text{hereby showing that indeed} \quad P(k+1) \quad \text{holds}
Induction Example: Geometric Closed Form

• Prove $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for all $a \neq 1$

  - Basis: show that $a^0 = (a^{0+1} - 1)/(a - 1)$
    
    $a^0 = 1 = (a^1 - 1)/(a - 1)$

  - Inductive hypothesis:
    
    o Assume $a^0 + a^1 + \ldots + a^k = (a^{k+1} - 1)/(a - 1)$

  - Inductive step: show that if $P(k)$ holds, then also $P(k+1)$ holds:
    
    $a^0 + a^1 + \ldots + a^{k+1} = a^0 + a^1 + \ldots + a^k + a^{k+1}$
    
    $= (a^{k+1} - 1)/(a - 1) + a^{k+1} = (a^{k+1+1} - 1)/(a - 1)$
Summary

• Proof by induction
  ▪ Base case, induction hypothesis and inductive step

• 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