How to prove an algorithm is correct?

• To prove the incorrectness of an algorithm, one counter-example is enough.

• Proving the correctness of an algorithm is similar to proving a mathematical theorem; fundamentally, it’s algorithm-dependent.

• But there are still some general guidelines we can follow.

• An example: **Proof by Loop Invariant**.
Insertion Sort

- Input: an array of numbers with length n.
- Output: a non-decreasing reordering of the array.
- Intuition: sorting a hand of playing cards.
- Formal description: start from an empty list (empty left hand); successively insert new elements in the proper positions.
Insertion Sort (an input instance)

```
<table>
<thead>
<tr>
<th>5</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
```
**Insertion Sort Algorithm**

**INSERTION-SORT**(A)

1. for j:=2 to length of A do  
2.     key := A[j]  
4.     i := j - 1  
5.     while (i > 0 AND A[i] > key)  
7.         i := i - 1  
8.     A[i+1] := key
Proof by Mathematical Induction

• The aim is to prove a statement $P(n)$ is true for all positive integers, starting with $n = 1$.

• Using mathematical induction, two steps are sufficient for this purpose:
  
  1. Prove that $P(1)$ is true (the base case).
  
  2. Assume that $P(k)$ is true for some $k$. Derive from here that $P(k+1)$ is also true (the inductive step).
Correctness Proof by Loop Invariant

**Step 0:** find a P first, which is called **loop invariant** in insertion sort.

At the start of each iteration of the for loop of line 1-8, the sub-array \( A[1..j-1] \) consists of the elements originally in \( A[1..j-1] \) but in sorted order.

**Step 1:** Initialization (the base case)
when \( j = 2 \), the sub-array \( A[1..j-1] \), consists of \( A[1] \), which is obviously sorted.

**Step 2:** Maintenance (the inductive step)

**Step 3:** Termination The algorithms terminates when \( j \) exceeds \( n \), namely \( j = n+1 \). So based on the loop invariant, \( A[1..j-1] = A[1..n] \) is sorted.
Efficiency

- Why don’t we just use a super computer?
- What if the computer is infinitely fast and the memory is free?

Measure of efficiency: space complexity and time complexity

**Space complexity:** the amount of storage needed to solve the problem. Typically expressed as a function of the input size (number of bits to represent input).

**Time complexity:** the amount of time needed to solve the problem?
Memory is unbounded

Accessing the ith memory address takes constant time,
1) **Each simple operation takes constant time.** What are simple operations? arithmetic (add, subtract, multiply, divide, remainder, floor, ceiling) data movement (load, store, copy) and control (conditional and unconditional branch, subroutine call and return).

2) **Things that do not take constant time** are loops and subroutine calls like sort.

3) **Each memory access takes the same amount of time.**
Time Complexity:

Time Complexity: the total number of basic operations performed, expressed as a function of the input size.

Input Size:

- the number of elements in the input (e.g. sorting), or
- the number of bits needed to represent the input (e.g. integer multiplication).
Exact Analysis of Insertion Sort

Note: In *for* loops and *while* statements, the loop header will be executed one more time than the body.

<table>
<thead>
<tr>
<th>InsertionSort(A)</th>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 for j:=2 to length of A do</td>
<td>$c_1$</td>
<td>$n$</td>
</tr>
<tr>
<td>2 key := A[j]</td>
<td>$c_2$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>3 /* put A[j] into A[1..j-1] */</td>
<td>$c_3 = 0$</td>
<td></td>
</tr>
<tr>
<td>4 i := j - 1</td>
<td>$c_4$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>5 while ( i &gt; 0 AND A[i] &gt; key)</td>
<td>$c_5$</td>
<td>$\sum_{j=2}^{n} t_j$</td>
</tr>
<tr>
<td>6 A[i+1] := A[i]</td>
<td>$c_6$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
</tr>
<tr>
<td>7 i:=i - 1</td>
<td>$c_7$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
</tr>
<tr>
<td>8 A[i+1] := key</td>
<td>$c_8$</td>
<td>$n - 1$</td>
</tr>
</tbody>
</table>
Exact Analysis of Insertion Sort

- $t_j = \#$ of times the \textbf{while} loop runs for the value $j$.
- $t_j = 1 + \#$ of elements that have to be shifted to the right to insert the $j^{th}$ item.

- \textbf{# of step 5} = $t_2 + t_3 + \ldots + t_n$.
- \textbf{# of step 6} = $(t_2 - 1) + (t_3 - 1) + \ldots + (t_n - 1)$.
- \textbf{# of step 7} = $(t_2 - 1) + (t_3 - 1) + \ldots + (t_n - 1)$.
General Case:

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

Best Case:

If the input array is already sorted, all \( t_j \)'s are 1. Hence, the best case time complexity is:

\[ T(n)_{\text{best}} = c_1n + (c_2 + c_4 + c_5 + c_8)(n - 1) \]

which is a **linear function** of \( n \).
Worst Case:

If the input is sorted in descending order, we will have to shift all of the already-sorted elements, so \( t_j = j \) for \( j = 2, 3, \ldots n \).

Note that:

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

\[T(n) = c_1 n + c_2(n - 1) + c_4(n - 1) + c_5\left(\frac{n(n+1)}{2} - 1\right)\]
\[+ c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n - 1) ;
\]

which is a \textbf{quadratic function} of \( n \).