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)

5	$\hat{2}$	$\hat{4}$	$\hat{6}$	$\hat{1}$	$\hat{3}$

2	5	$\hat{4}$	$\hat{6}$	$\hat{1}$	$\hat{3}$

2	4	5	$\hat{6}$	$\hat{1}$	$\hat{3}$

2	4	5	6	$\hat{1}$	$\hat{3}$

1	2	4	5	6	$\hat{3}$

1	2	3	4	5	6

Insertion Sort Algorithm

```
INSERTION-SORT(A)
    for j:=2 to length of A do
        key := A[j]
        /* put A[j] into A[1..j-1] */
        i := j -1
        while ( i > 0 AND A[i] > key)
              A[i+1] := A[i]
              i:=i - 1
        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

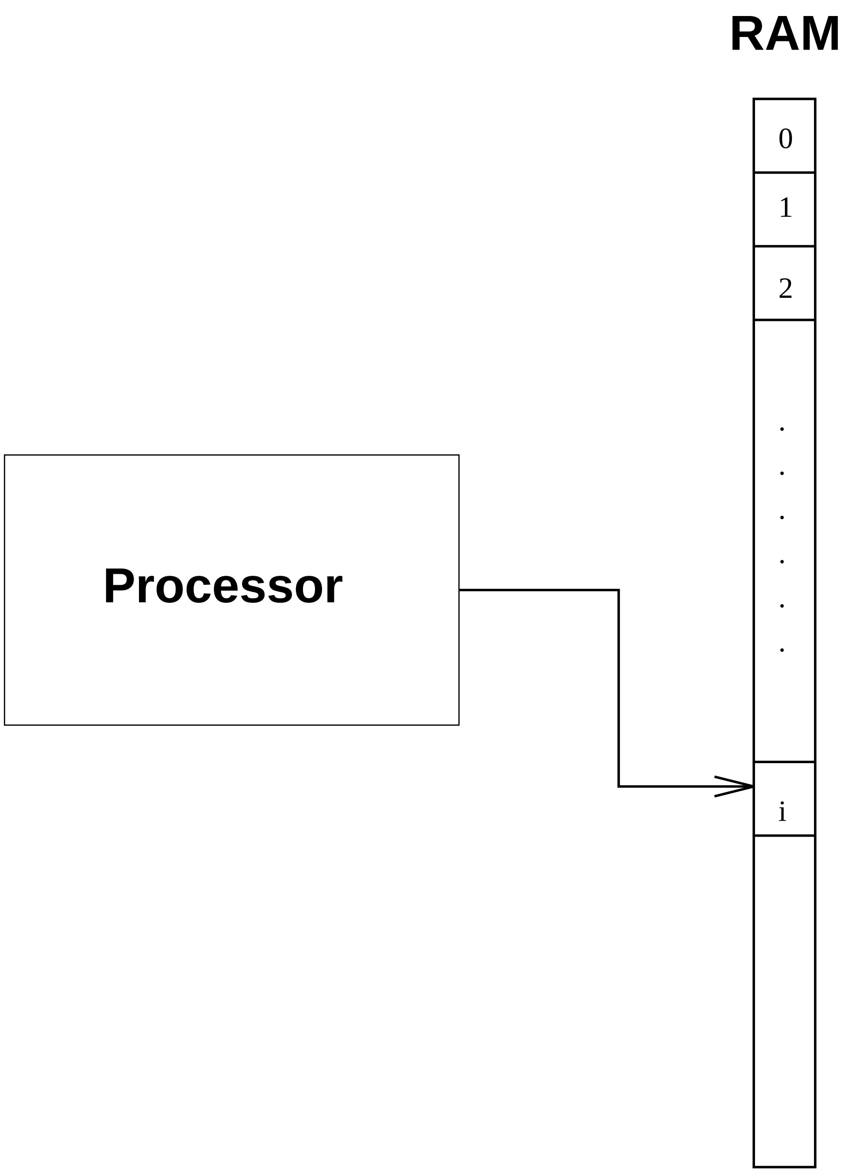
- Why don't we just use a super computer?
- What if the computer is infinitely fast and the memory is free?

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?

A RAM Machine



Memory is unbounded

Accessing the ith memory address takes constant time,

RAM (Random Access Machine) Model

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.

	InsertionSort(A)	cost	times
1	for j:=2 to length of A do	c_1	n
2	key := A[j]	c_2	$n-1$
3	/* put A[j] into A[1..j-1] */	$c_3 = 0$	
4	i := j -1	c_4	$n-1$
5	while ($i > 0$ AND A[i] $>$ key)	c_5	$\sum_{j=2}^{n} t_j$
6	A[i+1] := A[i]	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7	i:=i - 1	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8	A[i+1] := key	c_8	$n-1$

Exact Analysis of Insertion Sort

- t_j = # of times the **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.
- **# of step 5** $= t_2 + t_3 + ... + t_n$.
- **# of step 6** $= (t_2 - 1) + (t_3 - 1) + ... + (t_n - 1)$.
- **# of step 7** $= (t_2 - 1) + (t_3 - 1) + ... + (t_n - 1)$.

General Case and Best Case

General Case:

$$T(n) = \quad c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1);$$

Best Case:

If the input array is already sorted, all t_js are 1. Hence, the best case time complexity is:

$$T(n)_{best} = c_1 n + (c_2 + c_4 + c_5 + c_8)(n-1)$$

which is a **linear function** of n.

Worst Case

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 $j = 2, 3, ...n$.

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

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

$$+c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1);$$

which is a **quadratic function** of n.