Introduction to Data Structures and Algorithms (ESO 207)

Lecture 1

What are Algorithms?

[We begin by discussing two important words in the course title.]

What are Algorithms?

[We begin by discussing two important words in the course title.]

- You have written many programs in ESc101.
- Programs implement algorithms.

What are Algorithms?

[We begin by discussing two important words in the course title.]

- You have written many programs in ESc101.
- Programs implement algorithms.
- Algorithm is an abstract and more general concept. Here is a possible definition.
  - A sequence of well defined steps to carry out a task mechanically.

What are Algorithms?

[We begin by discussing two important words in the course title.]

- You have written many programs in ESc101.
- Programs implement algorithms.
- Algorithm is an abstract and more general concept. Here is a possible definition.
  - A sequence of well defined steps to carry out a task mechanically.
- Essential characteristics of algorithm
  - Each step is precise and well defined
  - Terminates in finitely many steps

Examples

1. A manual of an electric appliance may provide an `algorithm` to do preliminary troubleshooting of the device if it does not work properly.

Examples

1. A manual of an electric appliance may provide an `algorithm` to do preliminary troubleshooting of the device if it does not work properly.
2. A manufacturing factory process may be an `algorithm` to produce an end product starting from raw materials.

Examples

1. A manual of an electric appliance may provide an `algorithm` to do preliminary troubleshooting of the device if it does not work properly.
2. A manufacturing factory process may be an `algorithm` to produce an end product starting from raw materials.
3. To add two numbers in their decimal representation, using sum of the digits and carry, a method all of us learnt in school, is an algorithm to add two numbers.

Examples

1. A manual of an electric appliance may provide an `algorithm` to do preliminary troubleshooting of the device if it does not work properly.
2. A manufacturing factory process may be an `algorithm` to produce an end product starting from raw materials.
3. To add two numbers in their decimal representation, using sum of the digits and carry, a method all of us learnt in school, is an algorithm to add two numbers.

- We are interested in algorithms for solving problems using computers.

Examples

1. A manual of an electric appliance may provide an `algorithm` to do preliminary troubleshooting of the device if it does not work properly.
2. A manufacturing factory process may be an `algorithm` to produce an end product starting from raw materials.
3. To add two numbers in their decimal representation, using sum of the digits and carry, a method all of us learnt in school, is an algorithm to add two numbers.

- We are interested in algorithms for solving problems using computers.

Computational Problems

- A Problem in our context is a `function`, from input domain to output domain.

Computational Problems

- A Problem in our context is a `function`, from input domain to output domain.
- Examples of problems
  - matrix multiplication.
  - sorting an array of numbers.

Computational Problems

- A Problem in our context is a `function`, from input domain to output domain.
- Examples of problems
  - matrix multiplication.
  - sorting an array of numbers.
- Algorithmic solution to such a problem is a sequence of steps, implementable on a computer, which start from an input instance of the problem and produce output specified by the problem.

What are Data Structures?

- Algorithms operate on information stored in the computer.

What are Data Structures?

- Algorithms operate on information stored in the computer.
- Information stored is the data. It could be organized/structured in many different ways. This structuring of data is called data structures.

  You have seen some basic data structures such as Arrays (sorted/unsorted), Linked Lists and Trees in Esc101.

What are Data Structures?

- Algorithms operate on information stored in the computer.
- Information stored is the data. It could be organized/structured in many different ways. This structuring of data is called data structures.

  You have seen some basic data structures such as Arrays (sorted/unsorted), Linked Lists and Trees in Esc101.
- Algorithms depend on the data structures they operate on.

What are Data Structures?

- Algorithms operate on information stored in the computer.
- Information stored is the data. It could be organized/structured in many different ways. This structuring of data is called data structures.

  You have seen some basic data structures such as Arrays (sorted/unsorted), Linked Lists and Trees in Esc101.
- Algorithms depend on the data structures they operate on.
- Efficiency (number of steps performed) for solving a particular problem depends on how data is structured.

What are Data Structures?

- Algorithms operate on information stored in the computer.
- Information stored is the data. It could be organized/structured in many different ways. This structuring of data is called data structures.

  You have seen some basic data structures such as Arrays (sorted/unsorted), Linked Lists and Trees in Esc101.
- Algorithms depend on the data structures they operate on.
- Efficiency (number of steps performed) for solving a particular problem depends on how data is structured.

Examples

1. Searching an element in easier in sorted list than in unsorted list.

Examples

1. Searching an element in easier in sorted list than in unsorted list.
2. On the other hand, insertion of a new element is easier in unsorted list than in sorted list.

Examples

1. Searching an element in easier in sorted list than in unsorted list.
2. On the other hand, insertion of a new element is easier in unsorted list than in sorted list.

In this course we will learn several commonly used data structures and algorithms design techniques used in devising efficient computational solution to problems.

An Example: Insertion Sort

Problem description

- Input: List $[a_1, a_2, \ldots, a_n]$
- Output: List $[a'_1, a'_2, \ldots, a'_n]$ s.t.

An Example: Insertion Sort

Problem description

- Input: List $[a_1, a_2, \ldots, a_n]$
- Output: List $[a'_1, a'_2, \ldots, a'_n]$ s.t.
  - $[a'_1, a'_2, \ldots, a'_n]$ is a permutation of $[a_1, a_2, \ldots, a_n]$ and

An Example: Insertion Sort

Problem description

- Input: List $[a_1, a_2, \ldots, a_n]$
- Output: List $[a'_1, a'_2, \ldots, a'_n]$ s.t.
  - $[a'_1, a'_2, \ldots, a'_n]$ is a permutation of $[a_1, a_2, \ldots, a_n]$ and
  - $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

An Example: Insertion Sort

Problem description

- Input: List $[a_1, a_2, \ldots, a_n]$
- Output: List $[a'_1, a'_2, \ldots, a'_n]$ s.t.
  - $[a'_1, a'_2, \ldots, a'_n]$ is a permutation of $[a_1, a_2, \ldots, a_n]$ and
  - $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Incremental Design: Suppose we have solved the problem for list of k elements how to extend it to a list with one extra element.

- Given $[b_1, \ldots, b_{i-1}, b_i, b_{i+1}, \ldots, b_n]$ s.t. $[b_{i+1}, \ldots, b_n]$ is sorted.
- Obtain $[b_1, \ldots, b_{i-1}, b'_i, b'_{i+1}, \ldots, b'_n]$ s.t.

- Given $[b_1, \ldots, b_{i-1}, b_i, b_{i+1}, \ldots, b_n]$ s.t. $[b_{i+1}, \ldots, b_n]$ is sorted.
- Obtain $[b_1, \ldots, b_{i-1}, b'_i, b'_{i+1}, \ldots, b'_n]$ s.t.
  - $[b'_i, b'_{i+1}, \ldots, b'_n]$ is sorted.

- Given $[b_1, \ldots, b_{i-1}, b_i, b_{i+1}, \ldots, b_n]$ s.t. $[b_{i+1}, \ldots, b_n]$ is sorted.
- Obtain $[b_1, \ldots, b_{i-1}, b'_i, b'_{i+1}, \ldots, b'_n]$ s.t.
  - $[b'_i, b'_{i+1}, \ldots, b'_n]$ is sorted.
  - $b'_i, b'_{i+1}, \ldots, b'_n$ is a permutation of $b_i, b_{i+1}, \ldots, b_n$.

- Given $[b_1, \ldots, b_{i-1}, b_i, b_{i+1}, \ldots, b_n]$ s.t. $[b_{i+1}, \ldots, b_n]$ is sorted.
- Obtain $[b_1, \ldots, b_{i-1}, b'_i, b'_{i+1}, \ldots, b'_n]$ s.t.
    - $[b'_i, b'_{i+1}, \ldots, b'_n]$ is sorted.
    - $b'_i, b'_{i+1}, \ldots, b'_n$ is a permutation of $b_i, b_{i+1}, \ldots, b_n$.
- This involves finding right position for b_i in the sorted list $[b_{i+1}, \ldots, b_n]$ and inserting b_i there.

- Given $[b_1, \ldots, b_{i-1}, b_i, b_{i+1}, \ldots, b_n]$ s.t. $[b_{i+1}, \ldots, b_n]$ is sorted.
- Obtain $[b_1, \ldots, b_{i-1}, b'_i, b'_{i+1}, \ldots, b'_n]$ s.t.
  - $[b'_i, b'_{i+1}, \ldots, b'_n]$ is sorted.
  - $b'_i, b'_{i+1}, \ldots, b'_n$ is a permutation of $b_i, b_{i+1}, \ldots, b_n$.
- This involves finding right position for b_i in the sorted list $[b_{i+1}, \ldots, b_n]$ and inserting b_i there.

  Elements of the sorted list, which are $< b_i$, need to be shifted one position to the left in the array to make room for inserting b_i

- More concretely, we start by comparing b_i with $b_{i+1}, b_{i+2}, ...$, shifting them one position to the left in the array if $< b_i$.

- More concretely, we start by comparing b_i with $b_{i+1}, b_{i+2}, ...$, shifting them one position to the left in the array if $< b_i$.
- Here is an example with $A = [18, 7, 1, 3, 6, 10, 11]$ and $i = 2$.

- More concretely, we start by comparing b_i with $b_{i+1}, b_{i+2}, ...$, shifting them one position to the left in the array if $< b_i$.
- Here is an example with $A = [18, 7, 1, 3, 6, 10, 11]$ and $i = 2$.
  - Store $A[2]$ in temporary variable s. $s = 7$.

$$
\begin{aligned}
& [18, 7, 1, 3, 6, 10, 11] \\
\longrightarrow \quad & [18, 1, -, 3, 6, 10, 11] \\
\longrightarrow \quad & [18, 1, 3, -, 6, 10, 11] \\
\longrightarrow \quad & [18, 1, 3, 6, -, 10, 11] \\
\longrightarrow \quad & [18, 1, 3, 6, 7, 10, 11]
\end{aligned}
$$

Steps of Algorithm in general case

$s = b_i$ (store b_i in a temporary variable)

$$[b_1, \ldots, b_{i-1}, b_i, b_{i+1}, \ldots, b_n]$$

$$\downarrow b_{i+1} < s$$

$$[b_1, \ldots, b_{i-1}, b_{i+1}, \curvearrowleft, b_{i+2}, \ldots, b_n]$$

$$\downarrow b_{i+2} < s$$

$$[b_1, \ldots, b_{i-1}, b_{i+1}, b_{i+2}, \curvearrowleft, b_{i+3}, \ldots, b_n]$$

$$\downarrow b_{i+3} < s$$

$$\vdots$$

$$\downarrow b_l < s$$

$$[b_1, \ldots, b_{i-1}, b_{i+1}, b_{i+2}, \ldots, b_l, \curvearrowleft, b_{l+1}, \ldots, b_n]$$

$$\downarrow b_{l+1} \geq s$$

$$[b_1, \ldots, b_{i-1}, b_{i+1}, b_{i+2}, \ldots, b_l, s, b_{l+1}, \ldots, b_n]$$

Algorithm Insert

Following is a precise description of this algorithm.

```
Insert (A,i,n)
  k=A[i]
  j=i
  while (j<n) and (A[j+1] < k) do
    A[j] = A[j+1]
    j = j+1
  A[j]=k
```

Another Example Run

Example run of Insert$(A, 1, 6)$ with $A = [18, -2, 3, 5, 10, 12]$

Another Example Run

Example run of Insert$(A, 1, 6)$ with $A = [18, -2, 3, 5, 10, 12]$

$k = 18$

Another Example Run

Example run of Insert$(A, 1, 6)$ with $A = [18, -2, 3, 5, 10, 12]$

$k = 18$

(inside while loop)

Another Example Run

Example run of Insert$(A, 1, 6)$ with $A = [18, -2, 3, 5, 10, 12]$

$k = 18$

(inside while loop)

$[-2, \_, 3, 5, 10, 12]$

Another Example Run

Example run of Insert$(A, 1, 6)$ with $A = [18, -2, 3, 5, 10, 12]$

$k = 18$

(inside while loop)

$[-2, \_, 3, 5, 10, 12] \rightarrow [-2, 3, \_, 5, 10, 12]$

Another Example Run

Example run of Insert$(A, 1, 6)$ with $A = [18, -2, 3, 5, 10, 12]$

$k = 18$

(inside while loop)

$[-2, \_, 3, 5, 10, 12] \to [-2, 3, \_, 5, 10, 12]$

$\to [-2, 3, 5, \_, 10, 12]$

Another Example Run

Example run of Insert$(A, 1, 6)$ with $A = [18, -2, 3, 5, 10, 12]$

$k = 18$

(inside while loop)

$[-2, \_, 3, 5, 10, 12] \rightarrow [-2, 3, \_, 5, 10, 12]$

$\rightarrow [-2, 3, 5, \_, 10, 12] \rightarrow [-2, 3, 5, 10, \_, 12]$

Another Example Run

Example run of Insert$(A, 1, 6)$ with $A = [18, -2, 3, 5, 10, 12]$

$k = 18$

(inside while loop)

$[-2, \_, 3, 5, 10, 12] \to [-2, 3, \_, 5, 10, 12]$

$\to [-2, 3, 5, \_, 10, 12] \to [-2, 3, 5, 10, \_, 12]$

$\to [-2, 3, 5, 10, 12, \_]$

Another Example Run

Example run of Insert$(A, 1, 6)$ with $A = [18, -2, 3, 5, 10, 12]$

$k = 18$

(inside while loop)

$[-2, \_, 3, 5, 10, 12] \rightarrow [-2, 3, \_, 5, 10, 12]$

$\rightarrow [-2, 3, 5, \_, 10, 12] \rightarrow [-2, 3, 5, 10, \_, 12]$

$\rightarrow [-2, 3, 5, 10, 12, \_]$

After completing this (5th) iteration, $j = 6$.

Another Example Run

Example run of Insert$(A, 1, 6)$ with $A = [18, -2, 3, 5, 10, 12]$

$k = 18$

(inside while loop)

$[-2, \_, 3, 5, 10, 12] \rightarrow [-2, 3, \_, 5, 10, 12]$

$\rightarrow [-2, 3, 5, \_, 10, 12] \rightarrow [-2, 3, 5, 10, \_, 12]$

$\rightarrow [-2, 3, 5, 10, 12, \_]$

After completing this (5th) iteration, $j = 6$.

Guard evaluates to false for the next iteration, as $6 \not< 6$

Another Example Run

Example run of Insert$(A, 1, 6)$ with $A = [18, -2, 3, 5, 10, 12]$

$k = 18$

(inside while loop)

$[-2, \_, 3, 5, 10, 12] \rightarrow [-2, 3, \_, 5, 10, 12]$

$\rightarrow [-2, 3, 5, \_, 10, 12] \rightarrow [-2, 3, 5, 10, \_, 12]$

$\rightarrow [-2, 3, 5, 10, 12, \_]$

After completing this (5th) iteration, $j = 6$.

Guard evaluates to false for the next iteration, as $6 \not< 6$

program exits the while loop with $j = 6$.

Another Example Run

Example run of Insert$(A, 1, 6)$ with $A = [18, -2, 3, 5, 10, 12]$

$k = 18$

(inside while loop)

$[-2, \_, 3, 5, 10, 12] \rightarrow [-2, 3, \_, 5, 10, 12]$

$\rightarrow [-2, 3, 5, \_, 10, 12] \rightarrow [-2, 3, 5, 10, \_, 12]$

$\rightarrow [-2, 3, 5, 10, 12, \_]$

After completing this (5th) iteration, $j = 6$.

Guard evaluates to false for the next iteration, as $6 \not< 6$

program exits the while loop with $j = 6$.

After executing the last instruction $(A[j] = k)$

Another Example Run

Example run of Insert$(A, 1, 6)$ with $A = [18, -2, 3, 5, 10, 12]$

$k = 18$

(inside while loop)

$[-2, \_, 3, 5, 10, 12] \rightarrow [-2, 3, \_, 5, 10, 12]$

$\rightarrow [-2, 3, 5, \_, 10, 12] \rightarrow [-2, 3, 5, 10, \_, 12]$

$\rightarrow [-2, 3, 5, 10, 12, \_]$

After completing this (5th) iteration, $j = 6$.

Guard evaluates to false for the next iteration, as $6 \not< 6$

program exits the while loop with $j = 6$.

After executing the last instruction ($A[j] = k$)

$\rightarrow [-2, 3, 5, 10, 12, 18]$

Correctness

- In convincing oneself of correctness of the above algorithm, main step is to prove correctness of the loop.

Correctness

- In convincing oneself of correctness of the above algorithm, main step is to prove correctness of the loop.
- We describe a technique for proving correctness of a loop.

Correctness

- In convincing oneself of correctness of the above algorithm, main step is to prove correctness of the loop.
- We describe a technique for proving correctness of a loop.

  A Loop invariant is a property that holds after every (zero or more) iteration of the body of the loop.

```
while c do
  P //loop body

//endWhile
```

Correctness

- In convincing oneself of correctness of the above algorithm, main step is to prove correctness of the loop.
- We describe a technique for proving correctness of a loop.

  A Loop invariant is a property that holds after every (zero or more) iteration of the body of the loop.

  ϕ `--(i)`

```
while c do
  P //loop body
```

  ϕ `--(ii)`

```
//endWhile
```

Correctness

- In convincing oneself of correctness of the above algorithm, main step is to prove correctness of the loop.
- We describe a technique for proving correctness of a loop.

  A Loop invariant is a property that holds after every (zero or more) iteration of the body of the loop.

  ϕ `--(i)`

  ```
  while c do
    P //loop body
  ```

  ϕ `--(ii)`

  ```
  //endWhile
  ```

  $\phi \wedge \neg c$ `--(iii)`
- It is easy to see that if ϕ is a loop invariant then $\phi \wedge \neg c$ holds at (iii). [Assumption: evaluating c does not alter program state]

Loop Invariant

ϕ --(i)

```
while c do

  P //loop body
```

ϕ --(ii)

```
//endWhile
```

$\phi \wedge \neg c$ --(iii)

Loop Invariant

```
ϕ --(i)
while c do
ϕ ∧ c
  P //loop body
ϕ --(ii)
//endWhile
ϕ ∧ ¬c --(iii)
```

- To show that ϕ is a loop invariant, we need to show that
  - (A) ϕ holds at (i) and
  - (B) If $\phi \wedge c$ holds before executing P then ϕ holds after execution of P.

Loop Invariant for Insert

Following is a strong enough loop invariant that is sufficient to prove correctness of program Insert.

Loop Invariant (Conjunction of I.1 to I.5)

// $k = a_i$ --(I.1)

// $i \leq j \leq n$ --(I.2)

Loop Invariant for Insert

Following is a strong enough loop invariant that is sufficient to prove correctness of program Insert.

Loop Invariant (Conjunction of I.1 to I.5)

// $k = a_i$ --(I.1)

// $i \leq j \leq n$ --(I.2)

// $a_{i+1}, ..., a_j < a_i$ --(I.3)

Following is a strong enough loop invariant that is sufficient to prove correctness of program Insert.

Loop Invariant (Conjunction of I.1 to I.5)

// $k = a_i$ --(I.1)

// $i \leq j \leq n$ --(I.2)

// $a_{i+1}, ..., a_j < a_i$ --(I.3)

// $A[i], ..., A[j-1] = a_{i+1}, ..., a_j$ --(I.4)

Loop Invariant for Insert

Following is a strong enough loop invariant that is sufficient to prove correctness of program Insert.

Loop Invariant (Conjunction of I.1 to I.5)

// $k = a_i$ --(I.1)
// $i \leq j \leq n$ --(I.2)
// $a_{i+1}, ..., a_j < a_i$ --(I.3)
// $A[i], ..., A[j-1] = a_{i+1}, ..., a_j$ --(I.4)
// $A[l] = a_l$ for l not in $[i, j)$ --(I.5)

To show that it is a loop invariant, we first show condition **A** . That is, invariant holds at point $(*)$ below in the program Insert.

Verification of Loop Invariant

```
1.Insert (A,i,n) // A[l] = a_l, for 1 ≤ l ≤ n
2.    k=A[i]
3.    j=i
//     (*)
4.    while (j<n) and (A[j+1] < k) do
```

In this case $i = j$

Verification of Loop Invariant

```
1.Insert (A,i,n) // A[l] = a_l, for 1 ≤ l ≤ n
2.    k=A[i]
3.    j=i
//     (*)
4.    while (j<n) and (A[j+1] < k) do
```

In this case $i = j$

(I.1), (I.2) clearly hold. (I.3), (I.4) hold because ranges, $[i+1, j)$, $[i, j)$ are empty. (I.5) holds because array A has not been modified.

Verification of Loop Invariant Continued ...

Verification of Loop Invariant Continued ...

- To show condition **B**, assume that $j = h$ before executing P.

Verification of Loop Invariant Continued ...

- To show condition **B**, assume that $j = h$ before executing P.

  We show below for each invariant, that it holds after the execution of P.

(I.1) $k = a_i$

It clearly holds because k is not changed in execution of P.

Verification of Loop Invariant Continued ...

- To show condition **B**, assume that $j = h$ before executing P.

  We show below for each invariant, that it holds after the execution of P.

(I.1) $k = a_i$

It clearly holds because k is not changed in execution of P.

(I.2) $i \leq j \leq n$

By I.2, $i \leq h \leq n$ and by c, $h < n$.

$\Rightarrow i \leq h < n$

$\Rightarrow i \leq h + 1 \leq n$

After executing P, $j = h + 1$.

$\Rightarrow i \leq j \leq n$

Correctness of invariants continued

(I.3) $a_{i+1}, \ldots, a_j < a_i$

$\Rightarrow a_{i+1}, \ldots, a_h < a_i$

$a_{h+1} < a_i$ (by c)

$\Rightarrow a_{i+1}, \ldots, a_{h+1} < a_i$

After executing P, $j = h + 1$

$\Rightarrow a_{i+1}, \ldots, a_j < a_i$

(I.4) $A[i] \ldots, A[j-1] = a_{i+1}, \ldots, a_j$

$\Rightarrow A[i] \ldots, A[h-1] = a_{i+1}, \ldots, a_h$

By IH (I.5), $A[h+1] = a_{h+1}$.

After executing statement 5, $A[h] = a_{h+1}$.

$\Rightarrow A[i] \ldots, A[h] = a_{i+1}, \ldots, a_{h+1}$

After executing P, $j = h + 1$

$\Rightarrow A[i] \ldots, A[j-1] = a_{i+1}, \ldots, a_j$

Correctness of invariants continued

(I.5) $A[l] = a_l$ for l $\notin [i, j)$

$\Rightarrow A[l] = a_l$ for $l \notin [i, h)$

Only $A[h]$ is modified in P.

$\Rightarrow A[l] = a_l$ for $l \notin [i, h+1)$

After executing P, $j = h + 1$

$\Rightarrow A[l] = a_l$ for $l \notin [i, j)$

Correctness of invariants continued

(I.5) $A[l] = a_l$ for l $\notin [i, j)$

$\Rightarrow A[l] = a_l$ for $l \notin [i, h)$

Only $A[h]$ is modified in P.

$\Rightarrow A[l] = a_l$ for $l \notin [i, h+1)$

After executing P, $j = h + 1$

$\Rightarrow A[l] = a_l$ for $l \notin [i, j)$

Overall correctness

When execution reaches the last instruction (Instruction 7), $I \wedge \neg c$ holds.

By $\neg c$, $j = n$ or $a_{j+1} \geq a_i$

Overall correctness

When execution reaches the last instruction (Instruction 7), $I \land \neg c$ holds.

By $\neg c$, $j = n$ or $a_{j+1} \geq a_i$

By (I.3) $a_{i+1}, ..., a_j < a_i$

Overall correctness

When execution reaches the last instruction (Instruction 7), $I \wedge \neg c$ holds.

By $\neg c$, $j = n$ or $a_{j+1} \geq a_i$

By (I.3) $a_{i+1}, ..., a_j < a_i$

- In the first case ($j = n$), using (I.4) and (I.5) program outputs $a_1, \ldots, a_{i-1}, a_{i+1} ..., a_n, a_i$ which satisfies output specification.
- In the second case ($a_{j+1} \geq a_i$), using (I.4) and (I.5) program outputs $a_1, \ldots, a_{i-1}, a_{i+1} ..., a_j, a_i, a_{j+1}, \ldots, a_n$ which satisfies output specification.

In most of our work, we will be less formal in proving correctness though we will use loop invariant etc. informally to convince ourselves of correctness.

Program for Insertion Sort

Using the intermediate step Insert(A,i,n) repeatedly, we get an algorithm to sort first n elements of array A as follows.

Program for Insertion Sort

Using the intermediate step Insert(A,i,n) repeatedly, we get an algorithm to sort first n elements of array A as follows.

Inserion Sort

```
Insertion_Sort(A,n)
  for i=n-1 downto 1 do
  Insert(A,i,n)
```

Program for Insertion Sort

Using the intermediate step Insert(A,i,n) repeatedly, we get an algorithm to sort first n elements of array A as follows.

Inserion Sort

```
Insertion_Sort(A,n)
  for i=n-1 downto 1 do
  Insert(A,i,n)
```

- Exercise: Write Invariant for the 'for loop'. Prove correctness of Insertion_Sort.

Pseudo Programming Language

- We write all our algorithms in the (pseudo) language used in the above program.

Pseudo Programming Language

- We write all our algorithms in the (pseudo) language used in the above program.

- The language has a small number of simple instructions but is capable of expressing all algorithms.

- We write all our algorithms in the (pseudo) language used in the above program.

- The language has a small number of simple instructions but is capable of expressing all algorithms.

  [A practical language, in contrast has many additional features added for pragmatic and implementation related reasons.]

Pseudo Programming Language

- We write all our algorithms in the (pseudo) language used in the above program.
- The language has a small number of simple instructions but is capable of expressing all algorithms.

  [A practical language, in contrast has many additional features added for pragmatic and implementation related reasons.]
- Simplicity of the language allows us to express algorithm in it in a transparent manner. It also allows analysis of these algorithms without getting obfuscated by unnecessary details.

- Our pseudo language has the following main instructions.
  - comparison ($a < b$), assignment ($i=j$)
  - if statement
  - while, for, repeat-until loops
  - Arrays (and Objects, to be used later).
  - For an object A, its field x is accessed as A.x.

- Our pseudo language has the following main instructions.
  - comparison ($a < b$), assignment ($i=j$)
  - if statement
  - while, for, repeat-until loops
  - Arrays (and Objects, to be used later).
  - For an object A, its field x is accessed as A.x.
- Semantics of these instructions is the same as in usual languages like C/Java.

- Our pseudo language has the following main instructions.
  - comparison ($a < b$), assignment ($i=j$)
  - if statement
  - while, for, repeat-until loops
  - Arrays (and Objects, to be used later).
  - For an object A, its field x is accessed as A.x.
- Semantics of these instructions is the same as in usual languages like C/Java.
- Objects (and Array) variables store pointers to the actual object. Call by value mechanism is used for parameter passing.

Course Textbook:

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Introduction to Algorithms, 3rd Edition. MIT Press 2009.