What is an Algorithm?

- An ***algorithm*** is a sequence of unambiguous instructions for solving a problem for obtaining the desired output for any legitimate input in a finite amount of time.

- An ***algorithm*** is a finite procedure, written in a fixed symbolic vocabulary, governed by precise instructions, moving in discrete steps, 1, 2, 3, …, whose execution requires no insight, cleverness, intuition, intelligence, or perspicuity, and that sooner or later comes to an end.

- The nonambiguity requirement is critical
- The range of inputs has to be carefully specified
- An algorithm can be implemented in several different ways
- Algorithms for the same problem can be based on very different ideas and can solve the problem with very different speeds

Fundamental Algorithms

- Selection (Linear Median)
- Quicksort, Heapsort
- Linear search, Hash search
- Tree traversals
- Graph traversals
- Depth first search, breadth first search

Two Problems with several Algorithms

Sorting

- Bubblesort
- Insertionsort
- Selectionsort
- Quicksort
- Heapsort
- Smoothsort

Searching

- Linear search
- Binary search
- Hash search

What is a Data Structure?

- A data structure is a particular scheme for organizing and accessing related data items
- The nature of the data elements is dictated by the problem at hand

Fundamental Data Structures

- Primitive data types
  - int, char, string, Boolean, double
- Compound data types
  - Arrays, records
  - Combination thereof
- Records or structs
  - Nested records or structs
- Arrays
  - 1-D arrays or vectors
  - 2-D arrays or tables
  - n-D arrays
- Lists
  - Linked lists, singly and doubly linked lists, skip lists
  - Stack, queues
- Trees
  - Rooted trees and forests
  - Ordered trees
  - Binary trees and m-way trees
  - AVL-trees
  - Heaps
  - Tries
- Graphs
  - Directed and undirected graphs
  - Weighted graphs
  - Paths and cycles
  - Representations: adjacency list, adjacency matrix, doubly connected edge list
- Sets
- Dictionaries
- Priority queues

Pseudo Code

- An algorithm is the procedural or step-by-step solution of a problem
- The procedural solution can be at different levels of abstraction

- Machine code
- Assembly language
- C
- Pascal
- C++
- Java
- C#
- Pseudo code

Banking Example

To honor their "most economical" young client, a Canadian bank wants to find the child or student having the largest amount of money in his or her savings accounts.

- *Problem 1*: For each child/student: sum up all the amounts in his or her savings accounts
- *Problem 2*: Find the largest number over all children

Problem 2: Find the maximum value in an array of *n* numbers

…

```
public int arrayMax(int[] A, int n) {
  int currentMax = A[0];
  for (int k = 1; k<n; i++) {
   if (currentMax < A[k]) {
     currentMax = A[k];
   }
  }
  return currentMax;
}
```

…

Pseudo-code vs. Java code

```
currentMax ← A[0]
for k ← 1 to n-1 do
 if currentMax < A[k]
then
    currentMax ← A[k]
return currentMax
```

```
                …
public int arrayMax(int[] A,
   int n) {
   int currentMax = A[0];
   for (int i=1; i<n; i++) {
    if (currentMax < A[i]) {
     currentMax = A[i];
    }
   }
 return currentMax;
}
                …
```

Problem 2: Find the maximum value in an array of *n* numbers

```
Algorithm arrayMax(A,n):
  Input: An array A storing n ≥ 1
         integers
  Output: The maximum element in A

  currentMax ← A[0]
  for k ← 1 to n-1 do
     if currentMax < A[k] then
         currentMax ← A[k]
  return currentMax
```

Pseudo Code: Expressions

- Assignment operator (←)
  - `currentMax ← A[i]`
- Equality relation (=)
  - `currentMax = A[i]`

  is `true` if `currentMax` and `A[i]` have the same value, otherwise `false`.
- Smaller than (<), greater than (>), smaller or equal to (≤), greater or equal to (≥)

Pseudo Code: Method declarations

- **Algorithm** name(param1, param2,…)

```
Algorithm arrayMax(A,n)
```

Pseudo Code: Decision structures

-
  ```
  if condition then
      true-actions
  [else
      false-actions]
  end
  ```

-
  ```
  if currentMax < A[k] then
          currentMax ← A[k]
  end
  ```

Pseudo Code: While loops

-
  ```
  while condition do
      actions
  end
  ```

-
  ```
  while currentMax < A[k] do
      count ← 2*count
      i ← k+1
  end
  ```

Pseudo Code: Repeat loops

-
  ```
  repeat
      actions
  until condition
  ```

-
  ```
  repeat
    i ← k+1
    count ← 2*count
  until currentMax < A[i]
  ```

Pseudo Code: For loops

-
  ```
  for step-definition do
      actions
  end
  ```

-
  ```
  for k ← 1 to n-1 do
      if currentMax < A[k] then
          currentMax ← A[k]
      end
  end
  ```

Pseudo Code: Array indexing and record field selection

- A[*k*] represents the k^{th} cell in array A, indexed from *A*[0] to *A*[*n–1*].
- *r*.*key* represents the field *key* in record or struct *r*

Pseudo Code: Method calls and return statements

- Method call
  - object.method(args)
  - Example: `arrayMax(X,13)`

- Return statement
  - **return** value

Algorithmics

- Design and analysis process of algorithms and their underlying data structures
- Given a problem
  - ➢ Understand and specify
  - ➢ Model the problem
  - ➢ Design of data structures and algorithms
    - Investigate algorithm design techniques
    - Investigate data structures
  - ➢ Prove correctness of the algorithm
  - ➢ Analyze the efficiency of the algorithm
  - ➢ Implement solution
  - ➢ Test implementation

Analysis of Algorithm Efficiency

Not everything that can be counted counts,
and not everything that counts can be counted.
—Albert Einstein (1879-1955)

- Time efficiency
  - ➢indicates how fast an algorithm runs
- Space efficiency
  - ➢indicates how much extra space an algorithm requires

Measuring an Algorithm's Running Time

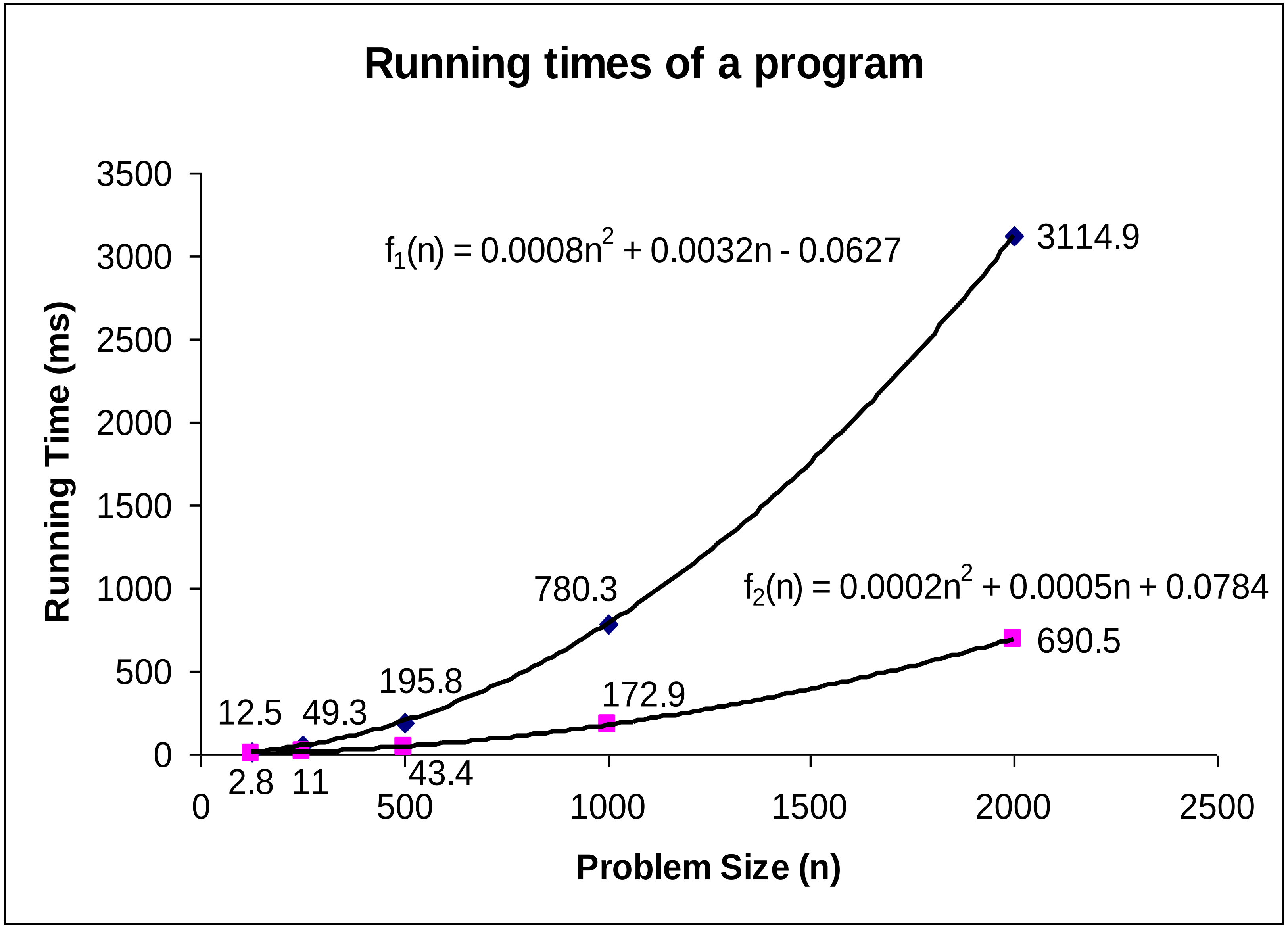
- Two approaches
  - Counting primitive operations and computing upper and lower bounds—topic in CSC 115/160, CSC 225, CSC 326, CSC 425, CSC 426, CSC 428 and many other courses
  - Instrument the code to measure computer clock cycles—topic in SENG 265

What is the running time of this algorithm?

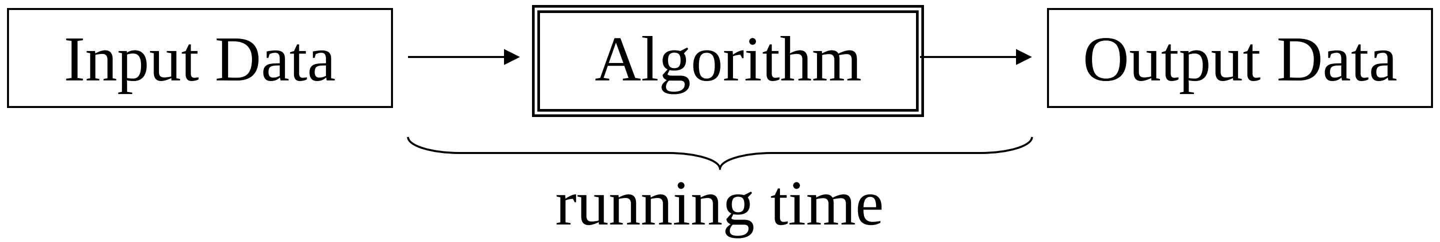
```
Algorithm arrayMax(A,n):
  Input: An array A storing n ≥ 1
     integers.
  Output: The maximum element in A.

  currentMax ← A[0]
  for k ← 1 to n-1 do
     if currentMax < A[k] then
          currentMax ← A[k]
     end
  end
  return currentMax
```

Instrumenting the Code to Measure Clock Cycles

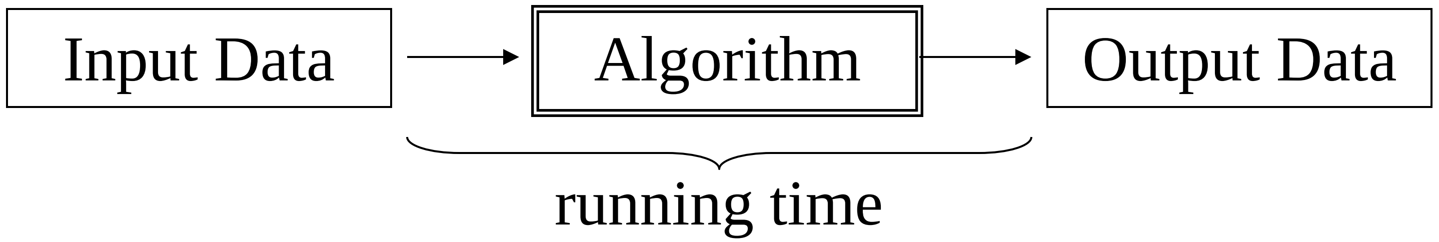


Limitations of Experiments Results



- The algorithm must be implemented
- Experiments can typically be done only on a limited set of test inputs
- When is a set of test inputs representative?
- To compare two algorithms the experiments must run on the same machine

Algorithm Analysis Wish List



- Take all possible inputs into account
- Compare the efficiency of two different algorithms independent from computer and implementation
- Analyze algorithm *before* starting implementation

Time Complexity Analysis

- Measuring the size of the algorithm's input
- Defining units for measuring running time
- Computing function for running time
- Performing worst-case, best-case, average-case analysis
- Insert function into the order of growth
- Asymptotic notations and efficiency classes
- Analysis of recursive and non-recursive algorithms

Primitive Operations

- Assignments (A)
- Comparisons (C)
- Boolean expressions (B)
- Array indexing (I)
- Record selector or object reference (R)
- Arithmetic operations
  - Add, subtract (S)
  - Multiplication, division (D)
- Trigonometric operations (e.g., sin, cos, tan) (T)
- Calling a method, function, procedure, routine (M)

Worst Case Running Time T(n)
Counting Primitive Operations

```
currentMax ← A[0]
for k ← 1 to n-1 do
    if currentMax < A[k] then
        currentMax ← A[k]
    end
end
return currentMax
```

- How has the input to be arranged to produce the best case and the worst case?

Worst case

- 1A + 1I +
- 1A + (N-1)*{
  - 1A + 1S +
  - 1C +
  - 1C + 1I +
  - 1A + 1I +
- }
- 1C (to terminate loop) +
- 1M

T(n) = 5+(n-1)*7

T(n) = 7n - 2

Worst Case Running Time T(n)
Counting Assignments & Comparisons

```
currentMax ← A[0]
for k ← 1 to n-1 do
    if currentMax < A[k] then
        currentMax ← A[k]
    end
end
return currentMax
```

Worst case

- 1A +
- 1A + (N-1)*{
  - 1A +
  - 1C +
  - 1C +
  - 1A +
- }
- 1C (to terminate loop) +

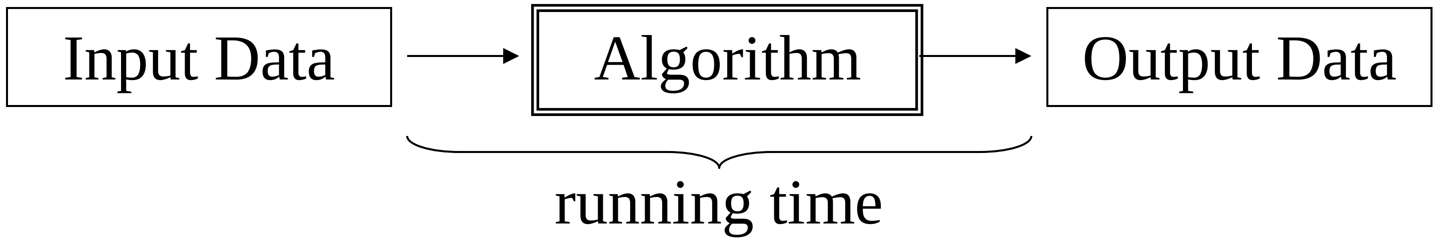
T(n) = 3+(n-1)*4

T(n) = 4n - 1

Tools for Algorithm Analysis

- Language for describing algorithms
  - ➢*pseudo-code*
- Metric for measuring algorithm running time
  - ➢*primitive operations*
- Most common approach for characterizing running times
  - ➢*Worst-case analysis*

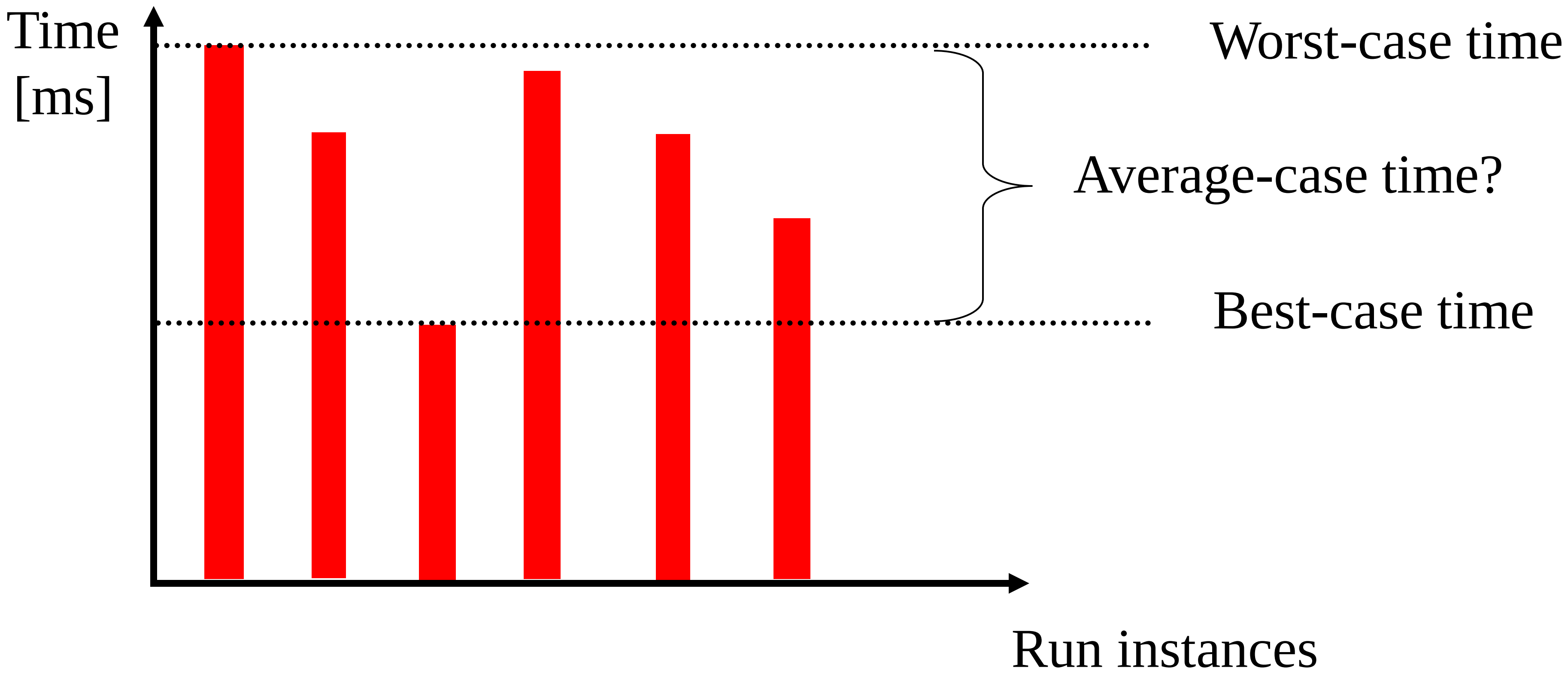
Algorithm analysis



Categories of algorithm running times

- Best case analysis $T_b(n)$
- Average-case analysis $T_a(n)$
- Worst-case analysis $T(n)$

Best-case, Average-case and Worst-case Analysis



Determining the Average-case Time

- Calculate expected running times based on a given input distribution
- Typically the analysis requires heavy math and probability theory

Determining the Best-case and Worst-case Running Time

- Worst-case T(n)
  - What is the maximum number of primitive operations (depending on *n*) executed by the algorithm, taken over all inputs of size *n*?
  - Worst-case analysis is most common and may aid in the design of the algorithm (e.g., Linear Median algorithm)
- Best-case $T_b(n)$
  - What is the minimum number of primitive operations (depending on *n*) executed by the algorithm, given the most advantageous or best input configuration of size *n*?

Examples: Worst-case, Best-case and Average-case Analysis

Basic units: A & C

```
a = 3 * n
cnt = 1
while a > n do
   a = a - 1
   cnt = cnt + 1
end
```

$T(n) = 2 + 2n(3)+1$

$T(n) = 3 + 6n$

$T_b(n) = T(n)$

Basic units: A & C

```
A: array[0..n-1]
k = 0
while k < n do
   if A[k] = key then
      return k
   end
   k = k + 1
end

return “not found”
```

$T(n) = 1 + n(3) + 2$

$T(n) = 3 + 3n$

$T_b(n) = 4$

$T_a(n) = 3 + 3n/2$

Examples: Worst-case, Best-case and Average-case Analysis

Basic units: A & C

```
A: array[0..n-1]
Swap: 3 A per swap
For: 2 A per iteration;
   ignore initial A in loops
for k=0 to n-1 do
   for j=0 to n-1 do
      if A[k]<=A[j] then
         swap(A[k],A[j])
      end
   end
end
```

$$\boxed{\begin{array}{l} T(n) = (6n+2)n \\ T(n) = 6n^2 + 2n \end{array}}$$

Basic units: A & C

```
For: 2 A per iteration;
   ignore initial A;
   ignore return A
s = 0
for k=1 to n do
   s = s + k*k
end
return s
```

$$\boxed{T(n) = 1 + 3n}$$

Structure of a Recursive Algorithm

```
Algorithm recursiveAlgorithm(n)
   if n = 1 then
      base-case
   else
      induction-step
      recursiveAlgorithm(n-1)
   end
```

- Let the worst case running time of `recursiveAlgorithm` be T(n)
- Then $$\boxed{T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ T(n-1) + c_2 & \text{otherwise} \end{cases}}$$

Structure of a Recursive Algorithm

```
Algorithm recursiveMax(A,n):
   Input: An array A storing n ≥ 1 integers.
   Output: The maximum element in A.

if n = 1 then return A[0] else
return max(recursiveMax(A,n−1),A[n−1]) end
```

- **Base case:** 3 operations (n=1, A[0], return)
- **Induction step:** T(n-1)+7 ops (n=1, n-1, n-1,A[n-1], call, max, ret)

$$\boxed{T(n) = \begin{cases} 3 & \text{if } n = 1 \\ T(n-1) + 7 & \text{otherwise} \end{cases}}$$

Recurrence Equation

Solving Recurrence Equation by Repeated Substitution

$$T(n) = T(n-1) + 7$$
$$T(n-1) = T(n-2) + 7$$
$$T(n-2) = T(n-3) + 7$$
$$\ldots$$
$$T(2) = T(1) + 7$$
$$T(1) = 3$$

$$T(1) = 3$$
$$T(2) = 3 + 7 = 10$$
$$T(3) = 3 + 7 + 7 = 17$$
$$T(4) = 3 + 7 + 7 + 7 = 24$$
$$\ldots$$
$$T(n) = 3 + 7(n-1)$$
$$T(n) = 3 + 7(n-1)$$
$$\boxed{\begin{gathered} T(n) = 7n - 4 \\ T(n) \in O(n) \end{gathered}}$$

$$\boxed{\begin{gathered} T(n) = c_1 + c_2(n-1) \\ T(n) \in O(n) \end{gathered}}$$