Discrete Mathematics 1

Chapter 3: The Fundamentals: Algorithms, the Integers

Department of Mathematics
The FPT university

Chapter 3: Introduction

Chapter 3: Introduction

Topics covered:

Topics covered:

Topics covered:

3.1 Algorithms

3.2 The Growth of Functions

Chapter 3: Introduction

Topics covered:

3.1 Algorithms

3.2 The Growth of Functions

3.3 Complexity of Algorithms

Topics covered:

Chapter 3: Introduction

Topics covered:

Topics covered:

3.1 Algorithms

3.1 Algorithms

3.1 Algorithms

An algorithm is a finite set of precise instructions for performing a computation or for solving a problem.

3.1 Algorithms

An algorithm is a finite set of precise instructions for performing a computation or for solving a problem.

Example. Describe an algorithm to solve quadratic equations.

3.1 Algorithms

An algorithm is a finite set of precise instructions for performing a computation or for solving a problem.

Example. Describe an algorithm to solve quadratic equations.

Input. a, b, c : integers (coefficients)
Output. Solutions if they exist.

Step 1. If $a = 0$ then Print (This is not a quadratic equation).
Step 2. Compute $\Delta = b^2 - 4ac$
Step 3. If $\Delta < 0$ then Print (No solution).
Step 4. If $\Delta = 0$, compute $x = -b/2a$
Step 5. If $\Delta > 0$, compute

$$x_1 = (-b + \sqrt{\Delta})/(2a), \;\; x_2 = (-b - \sqrt{\Delta})/(2a)$$

Properties of Algorithms:

Properties of Algorithms:

- **Input:**

Properties of Algorithms:

- **Input:** An algorithm has input values from a specified set.

Properties of Algorithms:

- **Input:** An algorithm has input values from a specified set.
- **Output:**

Properties of Algorithms:

- **Input:** An algorithm has input values from a specified set.
- **Output:** From each set of input values an algorithm produces output values from a specified set, and they are solutions to the problem.

Properties of Algorithms:

- **Input:** An algorithm has input values from a specified set.
- **Output:** From each set of input values an algorithm produces output values from a specified set, and they are solutions to the problem.
- **Definiteness:**

Properties of Algorithms:

- **Input:** An algorithm has input values from a specified set.
- **Output:** From each set of input values an algorithm produces output values from a specified set, and they are solutions to the problem.
- **Definiteness:** The steps of an algorithm must be defined precisely.

Properties of Algorithms:

- **Input:** An algorithm has input values from a specified set.
- **Output:** From each set of input values an algorithm produces output values from a specified set, and they are solutions to the problem.
- **Definiteness:** The steps of an algorithm must be defined precisely.
- **Correctness:**

Properties of Algorithms:

- **Input:** An algorithm has input values from a specified set.
- **Output:** From each set of input values an algorithm produces output values from a specified set, and they are solutions to the problem.
- **Definiteness:** The steps of an algorithm must be defined precisely.
- **Correctness:** An algorithm should produce the correct output values for each set of input values.

Properties of Algorithms:

- **Input:** An algorithm has input values from a specified set.
- **Output:** From each set of input values an algorithm produces output values from a specified set, and they are solutions to the problem.
- **Definiteness:** The steps of an algorithm must be defined precisely.
- **Correctness:** An algorithm should produce the correct output values for each set of input values.
- **Finiteness:**

Properties of Algorithms:

- **Input:** An algorithm has input values from a specified set.
- **Output:** From each set of input values an algorithm produces output values from a specified set, and they are solutions to the problem.
- **Definiteness:** The steps of an algorithm must be defined precisely.
- **Correctness:** An algorithm should produce the correct output values for each set of input values.
- **Finiteness:** An algorithm should produce the desired output after a finite number of steps.

Properties of Algorithms:

- **Input:** An algorithm has input values from a specified set.
- **Output:** From each set of input values an algorithm produces output values from a specified set, and they are solutions to the problem.
- **Definiteness:** The steps of an algorithm must be defined precisely.
- **Correctness:** An algorithm should produce the correct output values for each set of input values.
- **Finiteness:** An algorithm should produce the desired output after a finite number of steps.
- **Effectiveness:**:

Properties of Algorithms:

- **Input:** An algorithm has input values from a specified set.
- **Output:** From each set of input values an algorithm produces output values from a specified set, and they are solutions to the problem.
- **Definiteness:** The steps of an algorithm must be defined precisely.
- **Correctness:** An algorithm should produce the correct output values for each set of input values.
- **Finiteness:** An algorithm should produce the desired output after a finite number of steps.
- **Effectiveness:**: It must be possible to perform each step of an algorithm exactly and in a finite amount of time.

Properties of Algorithms:

- **Input:** An algorithm has input values from a specified set.
- **Output:** From each set of input values an algorithm produces output values from a specified set, and they are solutions to the problem.
- **Definiteness:** The steps of an algorithm must be defined precisely.
- **Correctness:** An algorithm should produce the correct output values for each set of input values.
- **Finiteness:** An algorithm should produce the desired output after a finite number of steps.
- **Effectiveness:**: It must be possible to perform each step of an algorithm exactly and in a finite amount of time.
- **Generality:**

Properties of Algorithms:

- **Input:** An algorithm has input values from a specified set.
- **Output:** From each set of input values an algorithm produces output values from a specified set, and they are solutions to the problem.
- **Definiteness:** The steps of an algorithm must be defined precisely.
- **Correctness:** An algorithm should produce the correct output values for each set of input values.
- **Finiteness:** An algorithm should produce the desired output after a finite number of steps.
- **Effectiveness:**: It must be possible to perform each step of an algorithm exactly and in a finite amount of time.
- **Generality:** Algorithm should be applicable for all problems of the desired form, not just a particular set of input values.

Some Algorithms

Some Algorithms

- Find maximum element of a finite sequence

Some Algorithms

- Find maximum element of a finite sequence
- Searching algorithms:

Some Algorithms

- Find maximum element of a finite sequence
- Searching algorithms:

    Linear search algorithm

Some Algorithms

- Find maximum element of a finite sequence
- Searching algorithms:

  Linear search algorithm
  Binary search algorithm

Some Algorithms

- Find maximum element of a finite sequence
- Searching algorithms:

  Linear search algorithm
  Binary search algorithm
- Sorting algorithms:

Some Algorithms

- Find maximum element of a finite sequence
- Searching algorithms:

  Linear search algorithm
  Binary search algorithm
- Sorting algorithms:

  Bubble sort

Some Algorithms

- Find maximum element of a finite sequence
- Searching algorithms:

  Linear search algorithm
  Binary search algorithm
- Sorting algorithms:

  Bubble sort
  Insertion sort

Some Algorithms

- Find maximum element of a finite sequence
- Searching algorithms:

  Linear search algorithm
  Binary search algorithm
- Sorting algorithms:

  Bubble sort
  Insertion sort

Some Algorithms

- Find maximum element of a finite sequence
- Searching algorithms:
  - Linear search algorithm
  - Binary search algorithm
- Sorting algorithms:
  - Bubble sort
  - Insertion sort
- Greedy change-making algorithm.

Some Algorithms

- Find maximum element of a finite sequence
- Searching algorithms:

  Linear search algorithm
  Binary search algorithm
- Sorting algorithms:

  Bubble sort
  Insertion sort
- Greedy change-making algorithm.

Finding Maximum Element

Finding Maximum Element

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Finding Maximum Element

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Output: The maximum number of the sequence

Finding Maximum Element

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Output: The maximum number of the sequence

Finding Maximum Element

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Output: The maximum number of the sequence

Algorithm:

Finding Maximum Element

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Output: The maximum number of the sequence

Algorithm:

- *Step 1.* Set the temporary maximum be the first element.

Finding Maximum Element

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Output: The maximum number of the sequence

Algorithm:

- *Step 1.* Set the temporary maximum be the first element.
- *Step 2.* Compare the temporary maximum to the next element, if this element is larger then set the temporary maximum to be this integer.

Finding Maximum Element

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Output: The maximum number of the sequence

Algorithm:

- *Step 1.* Set the temporary maximum be the first element.
- *Step 2.* Compare the temporary maximum to the next element, if this element is larger then set the temporary maximum to be this integer.
- *Step 3.* Repeat *Step 2* if there are more integers in the sequence.

Finding Maximum Element

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Output: The maximum number of the sequence

Algorithm:

- *Step 1.* Set the temporary maximum be the first element.
- *Step 2.* Compare the temporary maximum to the next element, if this element is larger then set the temporary maximum to be this integer.
- *Step 3.* Repeat *Step 2* if there are more integers in the sequence.
- *Step 4.* Stop the algorithm when there are no integers left. The temporary maximum at this point is the maximum of the sequence.

Finding Maximum Element

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Output: The maximum number of the sequence

Algorithm:

- *Step 1.* Set the temporary maximum be the first element.
- *Step 2.* Compare the temporary maximum to the next element, if this element is larger then set the temporary maximum to be this integer.
- *Step 3.* Repeat *Step 2* if there are more integers in the sequence.
- *Step 4.* Stop the algorithm when there are no integers left. The temporary maximum at this point is the maximum of the sequence.

Procedure Max($a_1, a_2, \ldots, a_n$: integers)

Procedure Max($a_1, a_2, \ldots, a_n$: integers)

$max := a_1$

Procedure Max($a_1, a_2, \ldots, a_n$: integers)
$max := a_1$
for $i := 2$ **to** n

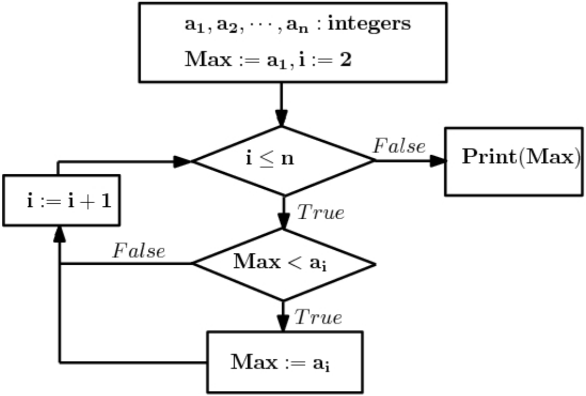
Procedure Max($a_1, a_2, \ldots, a_n$: integers)

$max := a_1$

for $i := 2$ **to** n

  if $max < a_i$ **then** $max := a_i$

Procedure Max($a_1, a_2, \ldots, a_n$: integers)
$max := a_1$
for $i := 2$ **to** n
  if $max < a_i$ **then** $max := a_i$



Linear Search

Linear Search

Input: A sequence of distinct integers $a_1, a_2, \ldots, a_n$, and an integer x

Linear Search

Input: A sequence of distinct integers $a_1, a_2, \ldots, a_n$, and an integer x

Output: The location of x in the sequence (is 0 if x is not in the sequence)

Linear Search

Input: A sequence of distinct integers $a_1, a_2, \ldots, a_n$, and an integer x

Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm:

Linear Search

Input: A sequence of distinct integers $a_1, a_2, \ldots, a_n$, and an integer x
Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm: Compare x successively to each term of the sequence until a match is found.

Linear Search

Input: A sequence of distinct integers $a_1, a_2, \ldots, a_n$, and an integer x
Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm: Compare x successively to each term of the sequence until a match is found.

Procedure LinearSearch ($a_1, a_2, \ldots, a_n$: distinct integers, x: integer)

Input: A sequence of distinct integers $a_1, a_2, \ldots, a_n$, and an integer x

Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm: Compare x successively to each term of the sequence until a match is found.

Procedure LinearSearch ($a_1, a_2, \ldots, a_n$: distinct integers, x: integer)
$i := 1$

Linear Search

Input: A sequence of distinct integers $a_1, a_2, \ldots, a_n$, and an integer x
Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm: Compare x successively to each term of the sequence until a match is found.

Procedure LinearSearch ($a_1, a_2, \ldots, a_n$: distinct integers, x: integer)
$i := 1$
while $(i \leq n)$ and $(x \neq a_i)$

Linear Search

Input: A sequence of distinct integers $a_1, a_2, \ldots, a_n$, and an integer x

Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm: Compare x successively to each term of the sequence until a match is found.

Procedure LinearSearch ($a_1, a_2, \ldots, a_n$: distinct integers, x: integer)
$i := 1$
while $(i \leq n)$ and $(x \neq a_i)$
  $i := i + 1$

Linear Search

Input: A sequence of distinct integers $a_1, a_2, \ldots, a_n$, and an integer x
Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm: Compare x successively to each term of the sequence until a match is found.

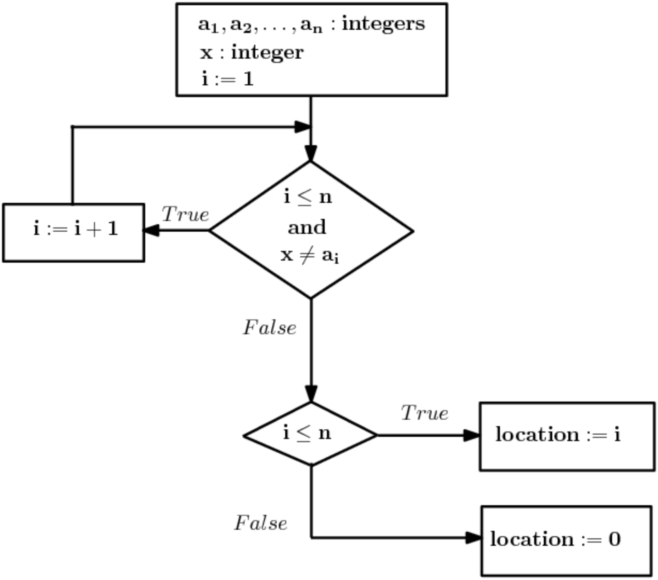
Procedure LinearSearch ($a_1, a_2, \ldots, a_n$: distinct integers, x: integer)
$i := 1$
while $(i \leq n)$ and $(x \neq a_i)$
    $i := i + 1$
if $i \leq n$ **then** $location := i$

Linear Search

Input: A sequence of distinct integers $a_1, a_2, \ldots, a_n$, and an integer x
Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm: Compare x successively to each term of the sequence until a match is found.

Procedure LinearSearch ($a_1, a_2, \ldots, a_n$: distinct integers, x: integer)
$i := 1$
while $(i \leq n)$ and $(x \neq a_i)$
    $i := i + 1$
if $i \leq n$ **then** $location := i$
else $location := 0$


$a_1, a_2, \ldots, a_n$: integers
x : integer
$i := 1$
$i \leq n$
and
$x \neq a_i$
True
$i := i + 1$
False
$i \leq n$
True
location := i
False
location := 0

Binary Search

Binary Search

Input: An increasing sequence of integers $a_1 < a_2 < \cdots < a_n$ and an integer x

Binary Search

Input: An increasing sequence of integers $a_1 < a_2 < \cdots < a_n$ and an integer x

Output: The location of x in the sequence (is 0 if x is not in the sequence)

Binary Search

Input: An increasing sequence of integers $a_1 < a_2 < \cdots < a_n$ and an integer x

Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm:

Binary Search

Input: An increasing sequence of integers $a_1 < a_2 < \cdots < a_n$ and an integer x
Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm: Compare x to the element at the middle of the list, then restrict the search to either the sublist on the left or the sublist on the right.

Binary Search

Input: An increasing sequence of integers $a_1 < a_2 < \cdots < a_n$ and an integer x
Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm: Compare x to the element at the middle of the list, then restrict the search to either the sublist on the left or the sublist on the right.

Procedure BinarySearch($a_1 < a_2 < \ldots < a_n, x$: integers)

Binary Search

Input: An increasing sequence of integers $a_1 < a_2 < \cdots < a_n$ and an integer x
Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm: Compare x to the element at the middle of the list, then restrict the search to either the sublist on the left or the sublist on the right.

Procedure BinarySearch($a_1 < a_2 < \ldots < a_n, x$: integers)
$i := 1$, $j := n$

Binary Search

Input: An increasing sequence of integers $a_1 < a_2 < \cdots < a_n$ and an integer x
Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm: Compare x to the element at the middle of the list, then restrict the search to either the sublist on the left or the sublist on the right.

Procedure BinarySearch($a_1 < a_2 < \ldots < a_n, x$: integers)
$i := 1, j := n$
while $(i < j)$

Binary Search

Input: An increasing sequence of integers $a_1 < a_2 < \cdots < a_n$ and an integer x
Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm: Compare x to the element at the middle of the list, then restrict the search to either the sublist on the left or the sublist on the right.

Procedure BinarySearch($a_1 < a_2 < \ldots < a_n, x$: integers)
$i := 1$, $j := n$
while $(i < j)$
    $m := \lfloor (i + j)/2 \rfloor$

Binary Search

Input: An increasing sequence of integers $a_1 < a_2 < \cdots < a_n$ and an integer x
Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm: Compare x to the element at the middle of the list, then restrict the search to either the sublist on the left or the sublist on the right.

Procedure BinarySearch($a_1 < a_2 < \ldots < a_n, x$: integers)
$i := 1, j := n$
while $(i < j)$
  $m := \lfloor (i+j)/2 \rfloor$
  if $x > a_m$ **then** $i := m+1$

Input: An increasing sequence of integers $a_1 < a_2 < \cdots < a_n$ and an integer x
Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm: Compare x to the element at the middle of the list, then restrict the search to either the sublist on the left or the sublist on the right.

Procedure BinarySearch($a_1 < a_2 < \ldots < a_n, x$: integers)
$i := 1$, $j := n$
while $(i < j)$
  $m := \lfloor (i + j)/2 \rfloor$
  if $x > a_m$ **then** $i := m + 1$
  else $j := m$

Binary Search

Input: An increasing sequence of integers $a_1 < a_2 < \cdots < a_n$ and an integer x
Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm: Compare x to the element at the middle of the list, then restrict the search to either the sublist on the left or the sublist on the right.

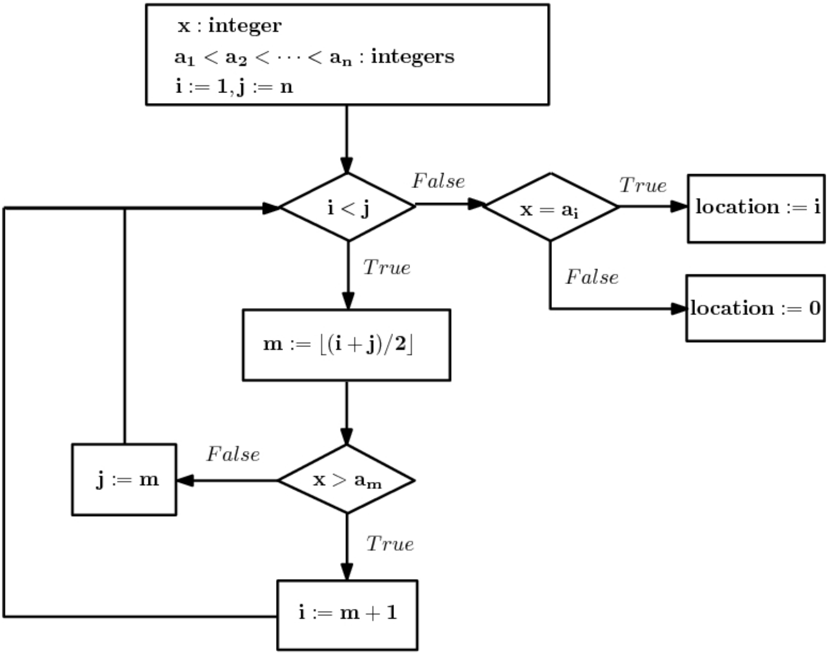
Procedure BinarySearch($a_1 < a_2 < \ldots < a_n, x$: integers)
$i := 1$, $j := n$
while $(i < j)$
  $m := \lfloor (i+j)/2 \rfloor$
  if $x > a_m$ **then** $i := m + 1$
  else $j := m$
if $x = a_i$ **then** $location := i$

Binary Search

Input: An increasing sequence of integers $a_1 < a_2 < \cdots < a_n$ and an integer x
Output: The location of x in the sequence (is 0 if x is not in the sequence)

Algorithm: Compare x to the element at the middle of the list, then restrict the search to either the sublist on the left or the sublist on the right.

Procedure BinarySearch($a_1 < a_2 < \ldots < a_n, x$: integers)
$i := 1$, $j := n$
while $(i < j)$
  $m := \lfloor (i + j)/2 \rfloor$
  if $x > a_m$ **then** $i := m + 1$
  else $j := m$
if $x = a_i$ **then** $location := i$
else $location := 0$


x : integer
a1 < a2 < · · · < an : integers
i := 1, j := n
i < j
False
x = ai
True
location := i
False
location := 0
True
m := ⌊(i + j)/2⌋
x > am
False
j := m
True
i := m + 1

Bubble Sort

Bubble Sort

Input: A sequence of integers $a_1, a_2, \ldots, a_n$

Bubble Sort

Input: A sequence of integers $a_1, a_2, \ldots, a_n$

Output: The sequence in the increasing order

Bubble Sort

Input: A sequence of integers $a_1, a_2, \ldots, a_n$

Output: The sequence in the increasing order

Algorithm:

Bubble Sort

Input: A sequence of integers $a_1, a_2, \ldots, a_n$

Output: The sequence in the increasing order

Algorithm:

1. Successively comparing two consecutive elements of the list to push the largest element to the bottom of the list.

Bubble Sort

Input: A sequence of integers $a_1, a_2, \ldots, a_n$

Output: The sequence in the increasing order

Algorithm:

1. Successively comparing two consecutive elements of the list to push the largest element to the bottom of the list.
2. Repeat the above step for the first $n-1$ elements of the list.

Bubble Sort

Input: A sequence of integers $a_1, a_2, \ldots, a_n$

Output: The sequence in the increasing order

Algorithm:

1. Successively comparing two consecutive elements of the list to push the largest element to the bottom of the list.
2. Repeat the above step for the first $n-1$ elements of the list.

Bubble Sort

Input: A sequence of integers $a_1, a_2, \ldots, a_n$

Output: The sequence in the increasing order

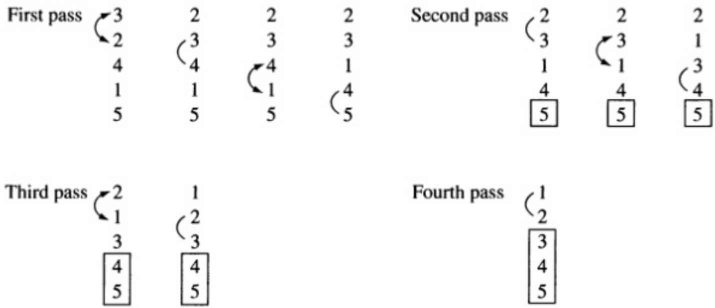
Algorithm:

1. Successively comparing two consecutive elements of the list to push the largest element to the bottom of the list.
2. Repeat the above step for the first $n-1$ elements of the list.

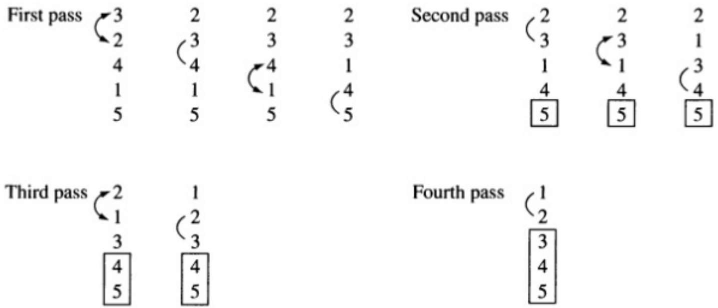
Example.

Example. Run the Bubble sort algorithm for the list $3, 2, 4, 1, 5$.

Example. Run the Bubble sort algorithm for the list $3, 2, 4, 1, 5$.

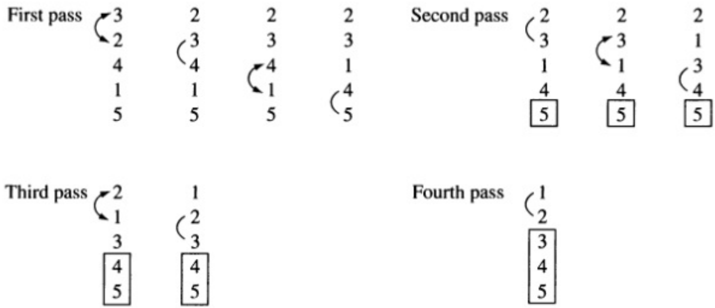


Example. Run the Bubble sort algorithm for the list $3, 2, 4, 1, 5$.



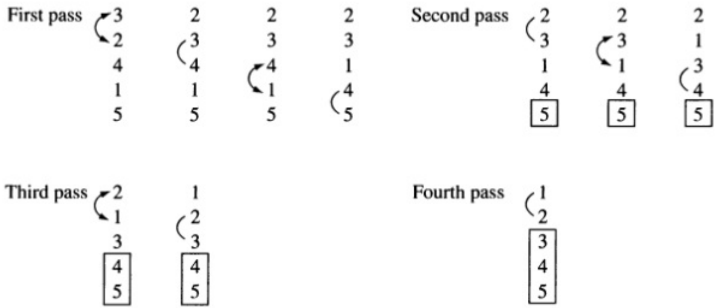
Procedure BubbleSort($a_1, a_2, \ldots, a_n$: integers)

Example. Run the Bubble sort algorithm for the list $3, 2, 4, 1, 5$.



Procedure BubbleSort($a_1, a_2, \ldots, a_n$: integers)
for $i := 1$ **to** $n - 1$

Example. Run the Bubble sort algorithm for the list $3, 2, 4, 1, 5$.

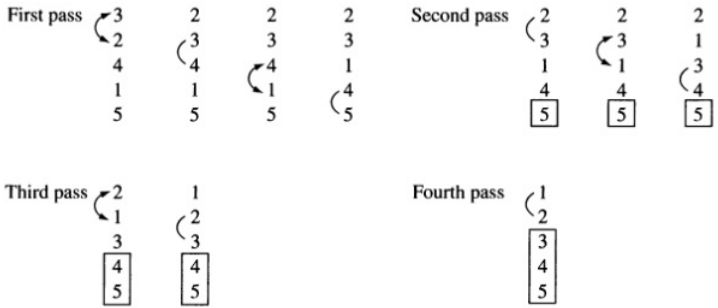


Procedure BubbleSort($a_1, a_2, \ldots, a_n$: integers)

for $i := 1$ **to** $n - 1$

  for $j := 1$ **to** $n - i$

Example. Run the Bubble sort algorithm for the list $3, 2, 4, 1, 5$.

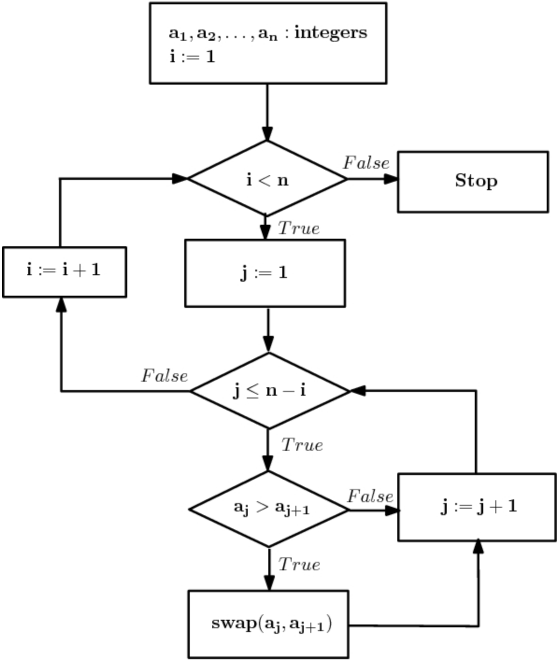


Procedure BubbleSort($a_1, a_2, \ldots, a_n$: integers)

for $i := 1$ **to** $n - 1$

  for $j := 1$ **to** $n - i$

    if $a_j > a_{j+1}$ **then** swap(a_j, a_{j+1})


$a_1, a_2, \ldots, a_n$: integers
$i := 1$
$i < n$
False
Stop
True
$i := i + 1$
$j := 1$
False
$j \leq n - i$
True
$a_j > a_{j+1}$
False
$j := j + 1$
True
$swap(a_j, a_{j+1})$

Insertion Sort

Insertion Sort

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Insertion Sort

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Output: The sequence in the increasing order

Insertion Sort

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Output: The sequence in the increasing order

Algorithm:

Insertion Sort

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Output: The sequence in the increasing order

Algorithm:

1. Sort the first two elements of the list

Insertion Sort

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Output: The sequence in the increasing order

Algorithm:

1. Sort the first two elements of the list
2. Insert the third element to the list of the first two elements to get a list of 3 elements of increasing order.

Insertion Sort

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Output: The sequence in the increasing order

Algorithm:

1. Sort the first two elements of the list
2. Insert the third element to the list of the first two elements to get a list of 3 elements of increasing order.
3. Insert the fourth element to the list of the first three elements to get a list of 4 elements of increasing order.

Insertion Sort

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Output: The sequence in the increasing order

Algorithm:

1. Sort the first two elements of the list
2. Insert the third element to the list of the first two elements to get a list of 3 elements of increasing order.
3. Insert the fourth element to the list of the first three elements to get a list of 4 elements of increasing order.

Insertion Sort

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Output: The sequence in the increasing order

Algorithm:

1 Sort the first two elements of the list

2 Insert the third element to the list of the first two elements to get a list of 3 elements of increasing order.

3 Insert the fourth element to the list of the first three elements to get a list of 4 elements of increasing order.

...

n Insert the nth element to the list of the first $n-1$ elements to get a list of increasing order.

Insertion Sort

Input: Sequence of integers $a_1, a_2, \ldots, a_n$

Output: The sequence in the increasing order

Algorithm:

1 Sort the first two elements of the list

2 Insert the third element to the list of the first two elements to get a list of 3 elements of increasing order.

3 Insert the fourth element to the list of the first three elements to get a list of 4 elements of increasing order.

...

n Insert the nth element to the list of the first $n - 1$ elements to get a list of increasing order.

Procedure InsertionSort($a_1, a_2, \ldots, a_n$: integers)

Procedure InsertionSort($a_1, a_2, \ldots, a_n$: integers)
for $j := 2$ **to** n

Procedure InsertionSort($a_1, a_2, \ldots, a_n$: integers)
for $j := 2$ **to** n
begin

Procedure InsertionSort($a_1, a_2, \ldots, a_n$: integers)
for $j := 2$ **to** n
begin
  $i := 1$

Procedure InsertionSort($a_1, a_2, \ldots, a_n$: integers)
for $j := 2$ **to** n
begin
  $i := 1$
  while $a_j > a_i$

Procedure InsertionSort($a_1, a_2, \ldots, a_n$: integers)
for $j := 2$ **to** n
begin
  $i := 1$
  while $a_j > a_i$
    $i := i + 1$

Procedure InsertionSort($a_1, a_2, \ldots, a_n$: integers)
for $j := 2$ **to** n
begin
  $i := 1$
  while $a_j > a_i$
    $i := i + 1$
  $m := a_j$

Procedure InsertionSort($a_1, a_2, \ldots, a_n$: integers)
for $j := 2$ **to** n
begin
  $i := 1$
  while $a_j > a_i$
    $i := i + 1$
  $m := a_j$
  $k := j$

Procedure InsertionSort($a_1, a_2, \ldots, a_n$: integers)
for $j := 2$ **to** n
begin
  $i := 1$
  while $a_j > a_i$
    $i := i + 1$
  $m := a_j$
  $k := j$
  while $k > i$

```
Procedure InsertionSort(a_1, a_2, ..., a_n: integers)
for j := 2 to n
begin
  i := 1
  while a_j > a_i
    i := i + 1
  m := a_j
  k := j
  while k > i
    a_k := a_{k-1}
```

Procedure InsertionSort($a_1, a_2, \ldots, a_n$: integers)

for $j := 2$ **to** n

begin

  $i := 1$

  while $a_j > a_i$

    $i := i + 1$

  $m := a_j$

  $k := j$

  while $k > i$

    $a_k := a_{k-1}$

    $k := k - 1$

Procedure InsertionSort($a_1, a_2, \ldots, a_n$: integers)
for $j := 2$ **to** n
begin
  $i := 1$
  while $a_j > a_i$
    $i := i + 1$
  $m := a_j$
  $k := j$
  while $k > i$
    $a_k := a_{k-1}$
    $k := k - 1$
  $a_i := m$

```
Procedure InsertionSort(a_1, a_2, ..., a_n: integers)
for j := 2 to n
begin
  i := 1
  while a_j > a_i
    i := i + 1
  m := a_j
  k := j
  while k > i
    a_k := a_{k-1}
    k := k - 1
  a_i := m
end
```

Greedy Change-Making Algorithm

Greedy Change-Making Algorithm

Input: n cents

Greedy Change-Making Algorithm

Input: n cents

Output: The least number of coins using quarters ($= 25$ cents), dimes ($= 10$ cents), nickles ($= 5$ cents) and pennies ($= 1$ cent).

Greedy Change-Making Algorithm

Input: n cents

Output: The least number of coins using quarters ($= 25$ cents), dimes ($= 10$ cents), nickles ($= 5$ cents) and pennies ($= 1$ cent).

Greedy Change-Making Algorithm

Input: n cents

Output: The least number of coins using quarters ($= 25$ cents), dimes ($= 10$ cents), nickles ($= 5$ cents) and pennies ($= 1$ cent).

Algorithm: Read textbook!

3.2 The Growth of Functions

In calculus, we learned following basic functions, listed in the increasing order of their complexity:

3.2 The Growth of Functions

In calculus, we learned following basic functions, listed in the increasing order of their complexity:

$$\log n, \;\; n^k, \;\; a^n, \;\; n!,$$

3.2 The Growth of Functions

In calculus, we learned following basic functions, listed in the increasing order of their complexity:

$$\log n, \ n^k, \ \ a^n, \ n!,$$

where $k > 0$ and $a > 1$.

3.2 The Growth of Functions

In calculus, we learned following basic functions, listed in the increasing order of their complexity:

$$\log n, \; n^k, \;\; a^n, \; n!,$$

where $k > 0$ and $a > 1$.

That means:

In calculus, we learned following basic functions, listed in the increasing order of their complexity:

$$\log n, \;\; n^k, \;\;\; a^n, \;\; n!,$$

where $k > 0$ and $a > 1$.

That means:

$$\lim_{n\to\infty} \frac{n^k}{\log n} = \infty$$

In calculus, we learned following basic functions, listed in the increasing order of their complexity:

$$\log n, \ n^k, \ \ a^n, \ n!,$$

where $k > 0$ and $a > 1$.

That means:

$$\lim_{n\to\infty} \frac{n^k}{\log n} = \infty \quad \lim_{n\to\infty} \frac{a^n}{n^k} = \infty$$

In calculus, we learned following basic functions, listed in the increasing order of their complexity:

$$\log n,\ \ n^k,\ \ \ a^n,\ \ n!,$$

where $k > 0$ and $a > 1$.

That means:

$$\lim_{n\to\infty} \frac{n^k}{\log n} = \infty \quad \lim_{n\to\infty} \frac{a^n}{n^k} = \infty \quad \lim_{n\to\infty} \frac{n!}{a^n} = \infty$$

In calculus, we learned following basic functions, listed in the increasing order of their complexity:

$$\log n, \;\; n^k, \;\;\; a^n, \;\; n!,$$

where $k > 0$ and $a > 1$.

That means:

$$\lim_{n\to\infty} \frac{n^k}{\log n} = \infty \quad \lim_{n\to\infty} \frac{a^n}{n^k} = \infty \quad \lim_{n\to\infty} \frac{n!}{a^n} = \infty$$

Question.

3.2 The Growth of Functions

In calculus, we learned following basic functions, listed in the increasing order of their complexity:

$$\log n, \ n^k, \ \ a^n, \ n!,$$

where $k > 0$ and $a > 1$.

That means:

$$\lim_{n\to\infty} \frac{n^k}{\log n} = \infty \quad \lim_{n\to\infty} \frac{a^n}{n^k} = \infty \quad \lim_{n\to\infty} \frac{n!}{a^n} = \infty$$

Question. Estimate the complexity (the growth) of functions like:

3.2 The Growth of Functions

In calculus, we learned following basic functions, listed in the increasing order of their complexity:

$$\log n, \; n^k, \; a^n, \; n!,$$

where $k > 0$ and $a > 1$.

That means:

$$\lim_{n\to\infty} \frac{n^k}{\log n} = \infty \quad \lim_{n\to\infty} \frac{a^n}{n^k} = \infty \quad \lim_{n\to\infty} \frac{n!}{a^n} = \infty$$

Question. Estimate the complexity (the growth) of functions like:

$$f(n) = \frac{(n+2)(n\log n + n!)}{3^n + (\log n)^2}$$

3.2 The Growth of Functions

In calculus, we learned following basic functions, listed in the increasing order of their complexity:

$$\log n, \;\; n^k, \;\; a^n, \;\; n!,$$

where $k > 0$ and $a > 1$.

That means:

$$\lim_{n\to\infty} \frac{n^k}{\log n} = \infty \quad \lim_{n\to\infty} \frac{a^n}{n^k} = \infty \quad \lim_{n\to\infty} \frac{n!}{a^n} = \infty$$

Question. Estimate the complexity (the growth) of functions like:

$$f(n) = \frac{(n+2)(n\log n + n!)}{3^n + (\log n)^2}$$

via simpler functions.

3.2 The Growth of Functions

In calculus, we learned following basic functions, listed in the increasing order of their complexity:

$$\log n, \;\; n^k, \;\;\; a^n, \;\; n!,$$

where $k > 0$ and $a > 1$.

That means:

$$\lim_{n\to\infty} \frac{n^k}{\log n} = \infty \quad \lim_{n\to\infty} \frac{a^n}{n^k} = \infty \quad \lim_{n\to\infty} \frac{n!}{a^n} = \infty$$

Question. Estimate the complexity (the growth) of functions like:

$$f(n) = \frac{(n+2)(n\log n + n!)}{3^n + (\log n)^2}$$

via simpler functions.

Big-O

Big-O

Big-O

The function $f(x)$ is called big-O of $g(x)$, write $f(x)$ is $O(g(x))$, if there is a constant C such that

Big-O

The function $f(x)$ is called big-O of $g(x)$, write $f(x)$ is $O(g(x))$, if there is a constant C such that

$$|f(x)| \leq C|g(x)|$$

Big-O

The function $f(x)$ is called big-O of $g(x)$, write $f(x)$ is $O(g(x))$, if there is a constant C such that

$$|f(x)| \leq C|g(x)|$$

for all x large enough, (meaning, it is true for all $x > k$ for some k).

Big-O

The function $f(x)$ is called big-O of $g(x)$, write $f(x)$ is $O(g(x))$, if there is a constant C such that

$$|f(x)| \leq C|g(x)|$$

for all x large enough, (meaning, it is true for all $x > k$ for some k).

Example.

Big-O

The function $f(x)$ is called big-O of $g(x)$, write $f(x)$ is $O(g(x))$, if there is a constant C such that

$$|f(x)| \leq C|g(x)|$$

for all x large enough, (meaning, it is true for all $x > k$ for some k).

Example.

(a) Show that $x^5 - 2x^2 + 7$ is $O(x^5)$.

Big-O

The function $f(x)$ is called big-O of $g(x)$, write $f(x)$ is $O(g(x))$, if there is a constant C such that

$$|f(x)| \leq C|g(x)|$$

for all x large enough, (meaning, it is true for all $x > k$ for some k).

Example.

(a) Show that $x^5 - 2x^2 + 7$ is $O(x^5)$.

(b) Show that $x^5 - 2x^2 + 7$ is not $O(x^4)$.

Theorem 1

Theorem 1

Let $f(x)$ be a polynomial of degree n. Then $f(x)$ is $O(x^n)$.

Theorem 1

Let $f(x)$ be a polynomial of degree n. Then $f(x)$ is $O(x^n)$.
This estimate is the best possible, meaning $f(x)$ is not big-O of any power of x that is less than n.

Theorem 1

Let $f(x)$ be a polynomial of degree n. Then $f(x)$ is $O(x^n)$.
This estimate is the best possible, meaning $f(x)$ is not big-O of any power of x that is less than n.

Theorem 2

Theorem 1

Let $f(x)$ be a polynomial of degree n. Then $f(x)$ is $O(x^n)$.
This estimate is the best possible, meaning $f(x)$ is not big-O of any power of x that is less than n.

Theorem 2

Assume that $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$. Then:

Theorem 1

Let $f(x)$ be a polynomial of degree n. Then $f(x)$ is $O(x^n)$.
This estimate is the best possible, meaning $f(x)$ is not big-O of any power of x that is less than n.

Theorem 2

Assume that $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$. Then:

- $f_1(x) + f_2(x)$ is big-O of the maximum of $g_1(x)$ and $g_2(x)$.

Theorem 1

Let $f(x)$ be a polynomial of degree n. Then $f(x)$ is $O(x^n)$.
This estimate is the best possible, meaning $f(x)$ is not big-O of any power of x that is less than n.

Theorem 2

Assume that $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$. Then:

- $f_1(x) + f_2(x)$ is big-O of the maximum of $g_1(x)$ and $g_2(x)$.
- $f_1(x)f_2(x)$ is big-O of $g_1(x)g_2(x)$.

Theorem 1

Let $f(x)$ be a polynomial of degree n. Then $f(x)$ is $O(x^n)$.
This estimate is the best possible, meaning $f(x)$ is not big-O of any power of x that is less than n.

Theorem 2

Assume that $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$. Then:

- $f_1(x) + f_2(x)$ is big-O of the maximum of $g_1(x)$ and $g_2(x)$.
- $f_1(x)f_2(x)$ is big-O of $g_1(x)g_2(x)$.

Example 1. Give a big-O estimate that is simple and effective for each of the functions:

Example 1. Give a big-O estimate that is simple and effective for each of the functions:

(a) $n^2 \log n + 3n^2 + 5$

Example 1. Give a big-O estimate that is simple and effective for each of the functions:

(a) $n^2 \log n + 3n^2 + 5$

(b) $(n^3 + 2^n)(n! + n^2 5^n)$

Example 1. Give a big-O estimate that is simple and effective for each of the functions:

(a) $n^2 \log n + 3n^2 + 5$

(b) $(n^3 + 2^n)(n! + n^2 5^n)$

Example 2. Find the smallest integer n such that:

Example 1. Give a big-O estimate that is simple and effective for each of the functions:

(a) $n^2 \log n + 3n^2 + 5$

(b) $(n^3 + 2^n)(n! + n^2 5^n)$

Example 2. Find the smallest integer n such that:

(a) $x^3 + x^5 \log x$ is $O(x^n)$

Example 1. Give a big-O estimate that is simple and effective for each of the functions:

(a) $n^2 \log n + 3n^2 + 5$

(b) $(n^3 + 2^n)(n! + n^2 5^n)$

Example 2. Find the smallest integer n such that:

(a) $x^3 + x^5 \log x$ is $O(x^n)$

(b) $x^5 + x^3 (\log x)^4$ is $O(x^n)$

Big-theta

Big-theta

Big-O estimates only give upper-bounds; to give sharper bounds we use big-theta notation.

Big-theta

Big-O estimates only give upper-bounds; to give sharper bounds we use big-theta notation.

The function $f(x)$ is called big-theta of $g(x)$, write: $f(x)$ is $\Theta(g(x))$, if $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.

Big-theta

Big-O estimates only give upper-bounds; to give sharper bounds we use big-theta notation.

The function $f(x)$ is called big-theta of $g(x)$, write: $f(x)$ is $\Theta(g(x))$, if $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.
In other words,

Big-theta

Big-O estimates only give upper-bounds; to give sharper bounds we use big-theta notation.

The function $f(x)$ is called big-theta of $g(x)$, write: $f(x)$ is $\Theta(g(x))$, if $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.
In other words, $f(x)$ is $\Theta(g(x))$ if there are constants $C_1, C_2 > 0$ such that

Big-theta

Big-O estimates only give upper-bounds; to give sharper bounds we use big-theta notation.

The function $f(x)$ is called big-theta of $g(x)$, write: $f(x)$ is $\Theta(g(x))$, if $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.
In other words, $f(x)$ is $\Theta(g(x))$ if there are constants $C_1, C_2 > 0$ such that

$$C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$$

Big-theta

Big-O estimates only give upper-bounds; to give sharper bounds we use big-theta notation.

The function $f(x)$ is called big-theta of $g(x)$, write: $f(x)$ is $\Theta(g(x))$, if $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.
In other words, $f(x)$ is $\Theta(g(x))$ if there are constants $C_1, C_2 > 0$ such that

$$C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$$

for all x large enough.

Big-theta

Big-O estimates only give upper-bounds; to give sharper bounds we use big-theta notation.

The function $f(x)$ is called big-theta of $g(x)$, write: $f(x)$ is $\Theta(g(x))$, if $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.
In other words, $f(x)$ is $\Theta(g(x))$ if there are constants $C_1, C_2 > 0$ such that

$$C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$$

for all x large enough.

Big-theta

Big-O estimates only give upper-bounds; to give sharper bounds we use big-theta notation.

The function $f(x)$ is called big-theta of $g(x)$, write: $f(x)$ is $\Theta(g(x))$, if $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.
In other words, $f(x)$ is $\Theta(g(x))$ if there are constants $C_1, C_2 > 0$ such that

$$C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$$

for all x large enough.

Note: is $f(x)$ is $\Theta(g(x))$ and $g(x)$ is also $\Theta(f(x))$.

Big-theta

Big-O estimates only give upper-bounds; to give sharper bounds we use big-theta notation.

The function $f(x)$ is called big-theta of $g(x)$, write: $f(x)$ is $\Theta(g(x))$, if $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.
In other words, $f(x)$ is $\Theta(g(x))$ if there are constants $C_1, C_2 > 0$ such that

$$C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$$

for all x large enough.

Note: is $f(x)$ is $\Theta(g(x))$ and $g(x)$ is also $\Theta(f(x))$. Therefore, if $f(x)$ is $\Theta(g(x))$ we say $f(x)$ is of order $g(x)$.

Example.

Example.

(a) Show that $f(x) = 2x^3 + x^2 + 3$ is $\Theta(x^3)$.

Example.

(a) Show that $f(x) = 2x^3 + x^2 + 3$ is $\Theta(x^3)$.

(b) Is the function $f(x) = x^2 \log x + 3x + 1$ big-theta of x^3?

Example.

(a) Show that $f(x) = 2x^3 + x^2 + 3$ is $\Theta(x^3)$.

(b) Is the function $f(x) = x^2 \log x + 3x + 1$ big-theta of x^3?

(c) Show that $f(x) = \lfloor x/2 \rfloor$ is $\Theta(x)$

- Space complexity:

3.3 Complexity of Algorithms

- Space complexity: Computer memory required to run the algorithm

3.3 Complexity of Algorithms

- Space complexity: Computer memory required to run the algorithm
- Time complexity:

3.3 Complexity of Algorithms

- Space complexity: Computer memory required to run the algorithm
- Time complexity: Time required to run the algorithm.

3.3 Complexity of Algorithms

- Space complexity: Computer memory required to run the algorithm
- Time complexity: Time required to run the algorithm. Time complexity can be expressed in terms of the number of operations used by the algorithm.

3.3 Complexity of Algorithms

- Space complexity: Computer memory required to run the algorithm
- Time complexity: Time required to run the algorithm. Time complexity can be expressed in terms of the number of operations used by the algorithm. Those operations can be comparisons or basic arithmetic operations.

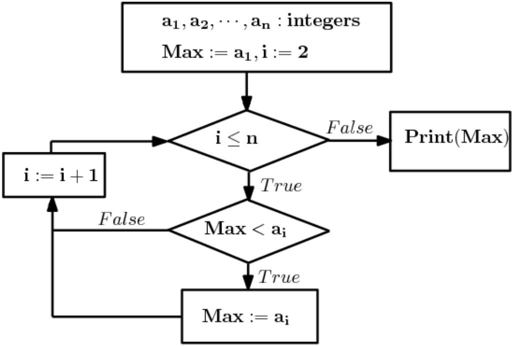
3.3 Complexity of Algorithms

- Space complexity: Computer memory required to run the algorithm
- Time complexity: Time required to run the algorithm. Time complexity can be expressed in terms of the number of operations used by the algorithm. Those operations can be comparisons or basic arithmetic operations.

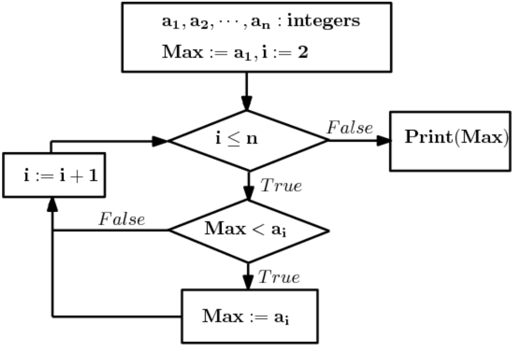
In this lecture we analyze time complexity of some algorithms studied in previous sections.

Complexity of Algorithm of Finding Maximum Element

Complexity of Algorithm of Finding Maximum Element

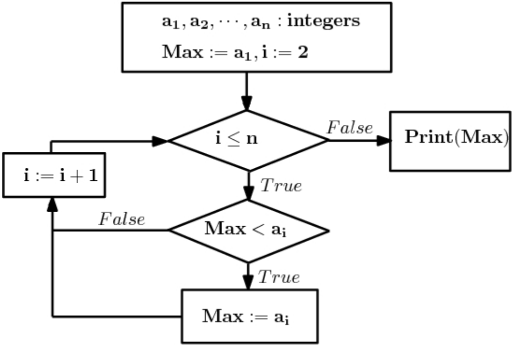


Complexity of Algorithm of Finding Maximum Element



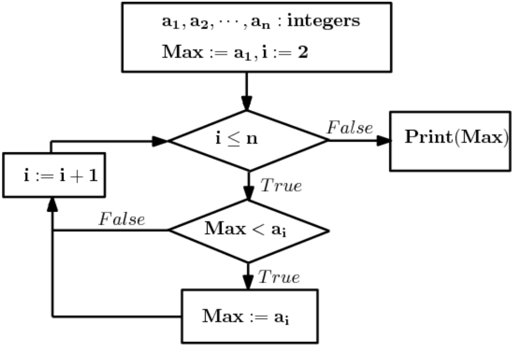
Number of loops: $n - 1$

Complexity of Algorithm of Finding Maximum Element



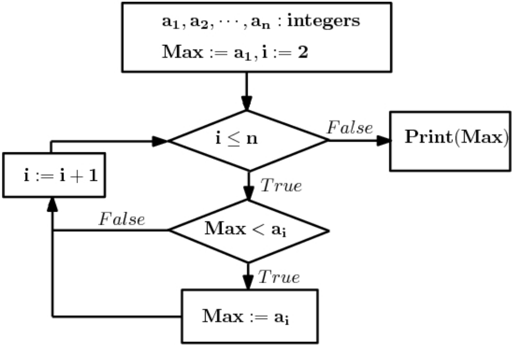
Number of loops: $n - 1$
Number of comparisons in each loop: 2

Complexity of Algorithm of Finding Maximum Element



Number of loops: $n - 1$
Number of comparisons in each loop: 2
Number of comparisons to exit the loop: 1

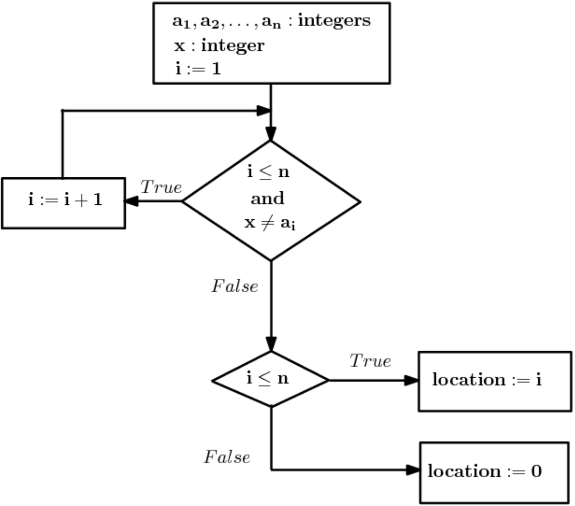
Complexity of Algorithm of Finding Maximum Element

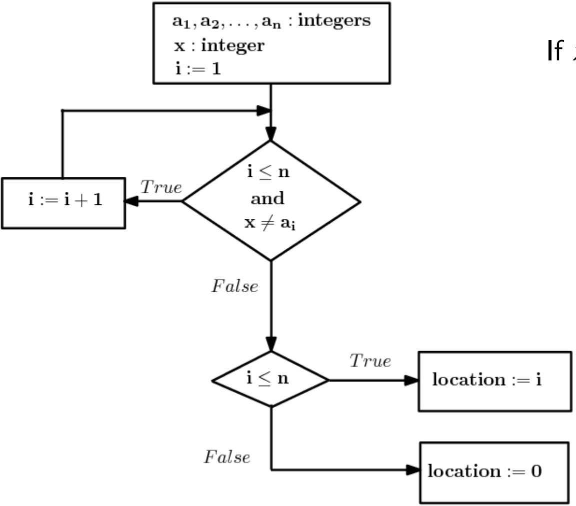


Number of loops: $n - 1$
Number of comparisons in each loop: 2
Number of comparisons to exit the loop: 1
Total number of comparisons: $2(n-1) + 1 = 2n - 1 = O(n)$

Complexity of Linear Search Algorithm

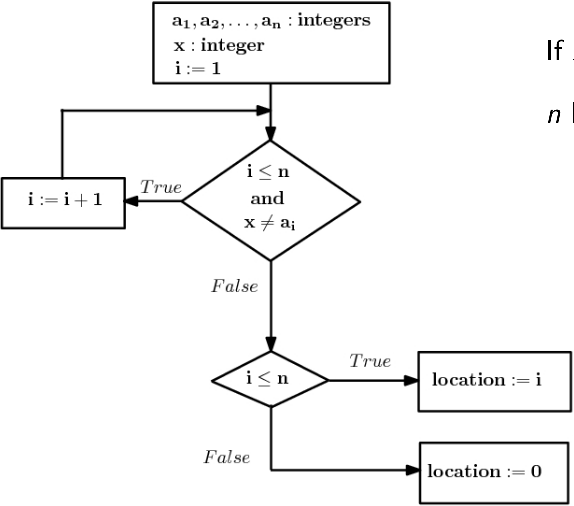
Complexity of Linear Search Algorithm





If x do not appear in the list:

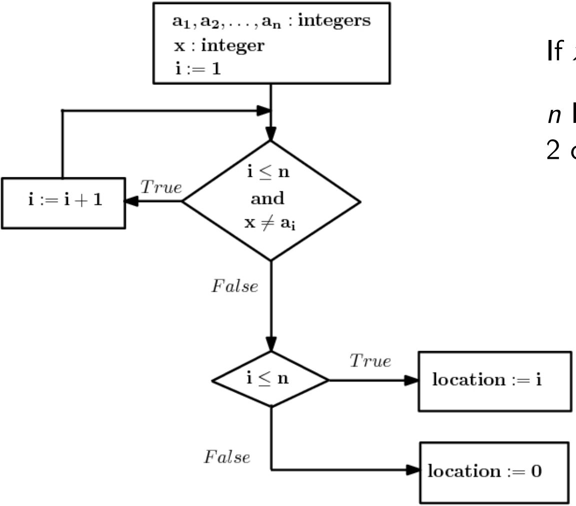
Complexity of Linear Search Algorithm



If x do not appear in the list:

n loops

Complexity of Linear Search Algorithm

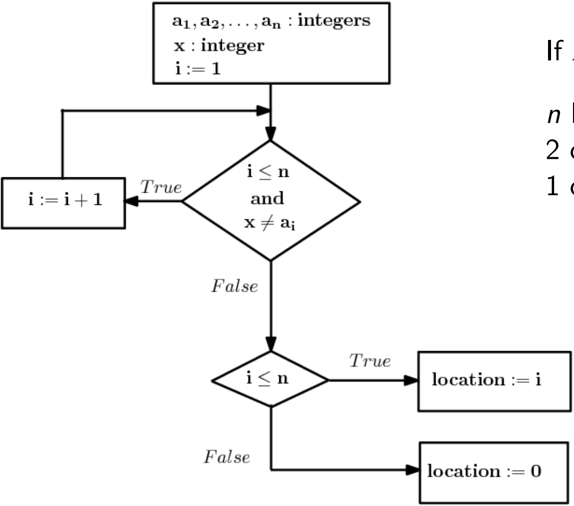


If x do not appear in the list:

n loops

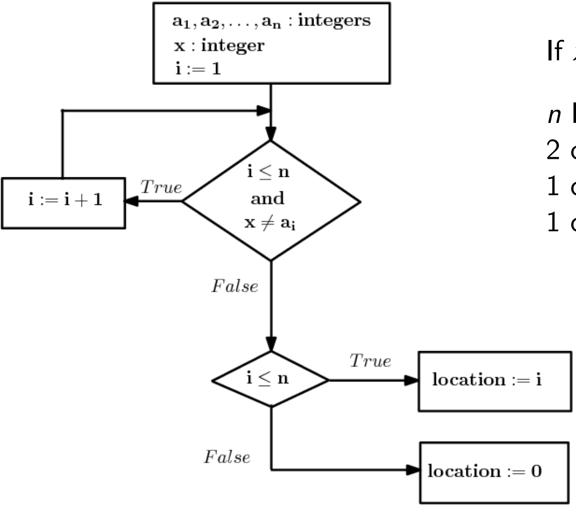
2 comparisons in each loop

Complexity of Linear Search Algorithm



If x do not appear in the list:

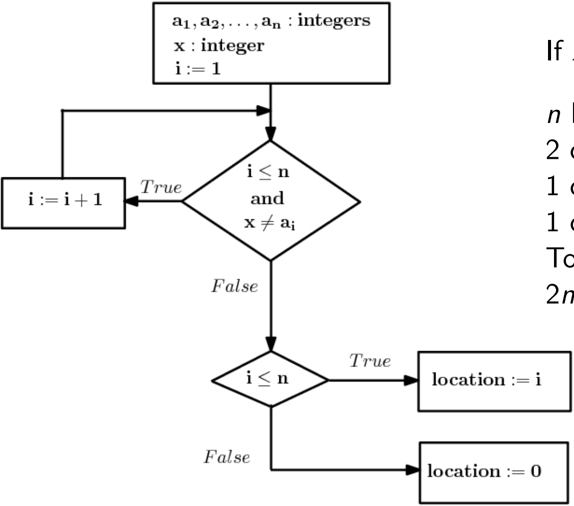
n loops
2 comparisons in each loop
1 comparison to exit the loop



If x do not appear in the list:

n loops
2 comparisons in each loop
1 comparison to exit the loop
1 comparison outside the loop

Complexity of Linear Search Algorithm

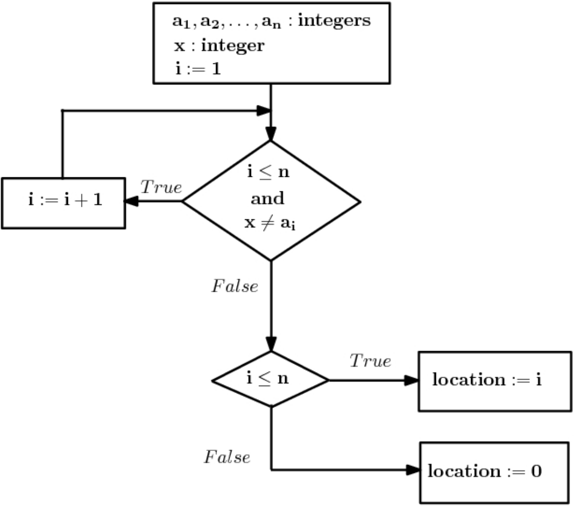


If x do not appear in the list:

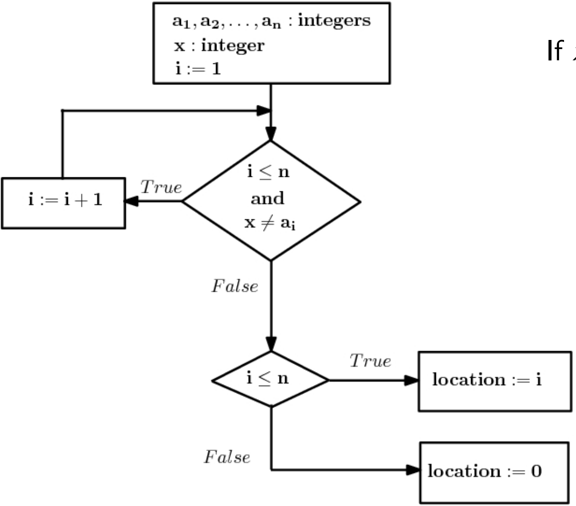
n loops
2 comparisons in each loop
1 comparison to exit the loop
1 comparison outside the loop
Total number of comparisons:
$2n + 1 + 1 = 2n + 2 = O(n)$

Complexity of Linear Search Algorithm

Complexity of Linear Search Algorithm

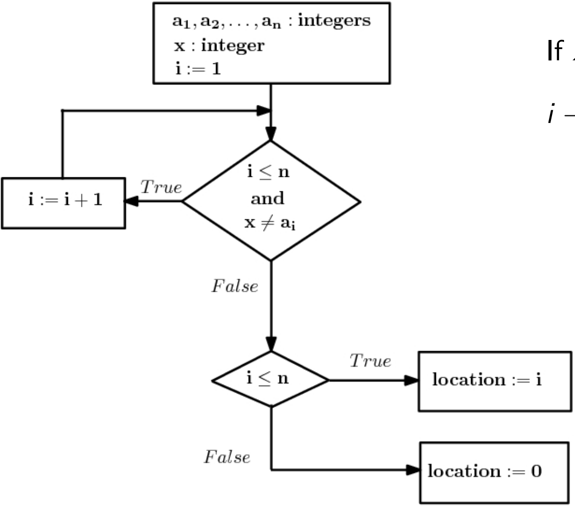


Complexity of Linear Search Algorithm



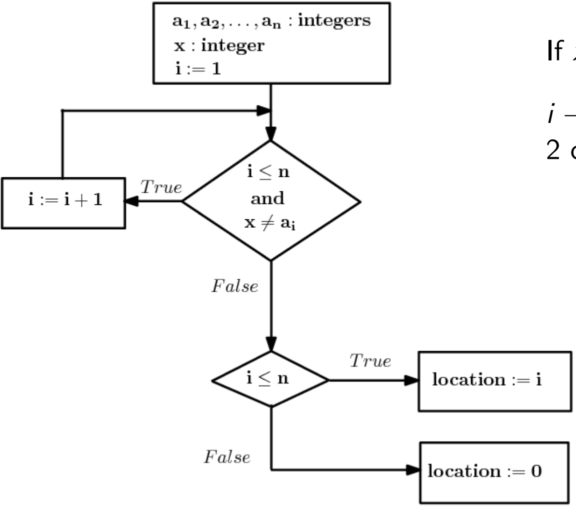
If x is at the ith location:

Complexity of Linear Search Algorithm



If x is at the ith location:

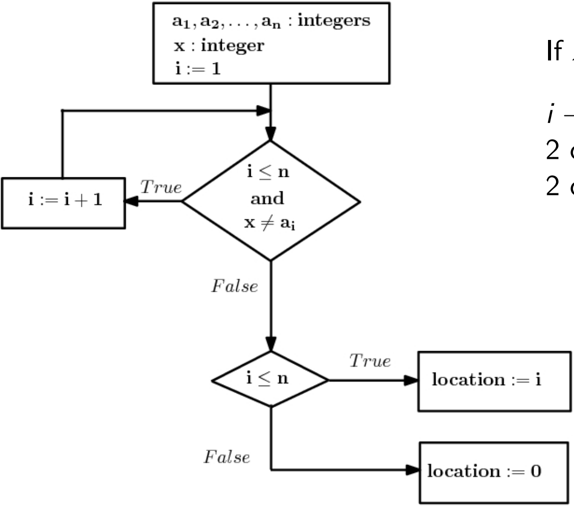
$i - 1$ loops



If x is at the ith location:

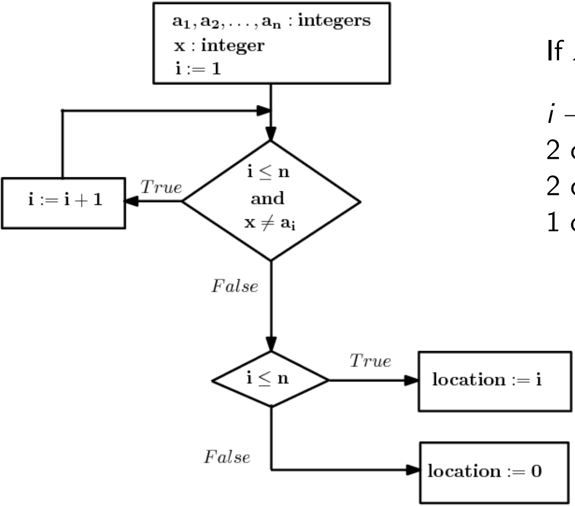
$i - 1$ loops
2 comparisons in each loop

Complexity of Linear Search Algorithm



If x is at the ith location:

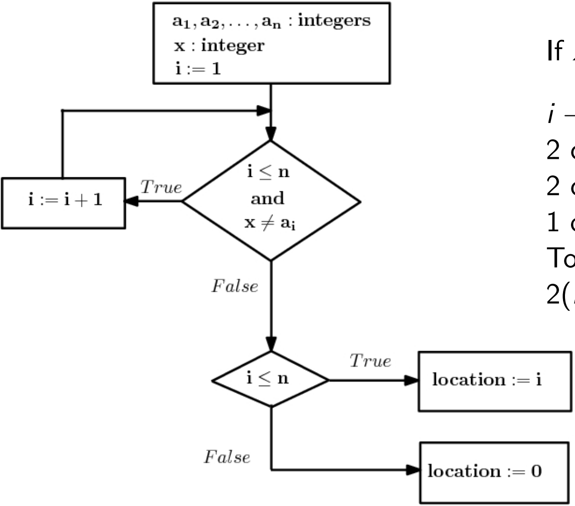
$i - 1$ loops
2 comparisons in each loop
2 comparisons to exit the loop



If x is at the ith location:

$i - 1$ loops
2 comparisons in each loop
2 comparisons to exit the loop
1 outside the loop

Complexity of Linear Search Algorithm

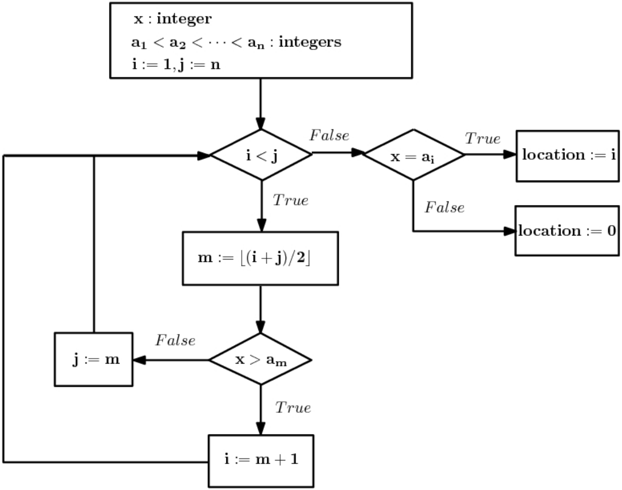


If x is at the ith location:

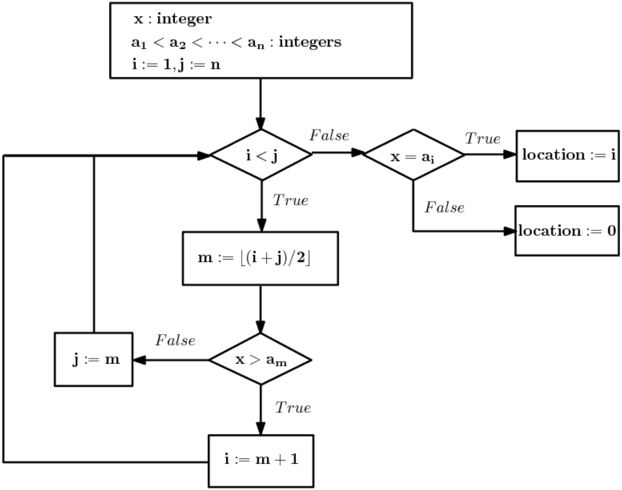
$i - 1$ loops
2 comparisons in each loop
2 comparisons to exit the loop
1 outside the loop
Total number of comparisons:
$2(i-1) + 2 + 1 = 2i + 1$

Complexity of Binary Search Algorithm

Complexity of Binary Search Algorithm

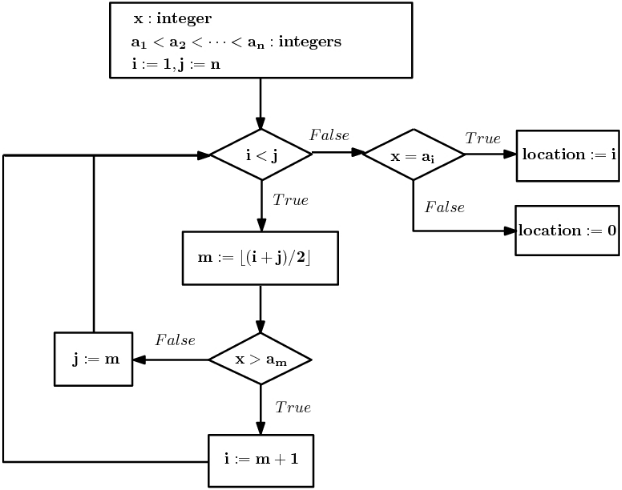


Complexity of Binary Search Algorithm



Assume $n = 2^k$

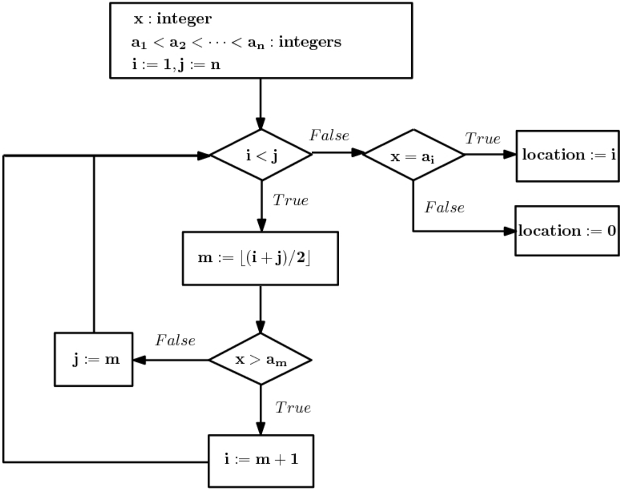
Complexity of Binary Search Algorithm



Assume $n = 2^k$

k loops

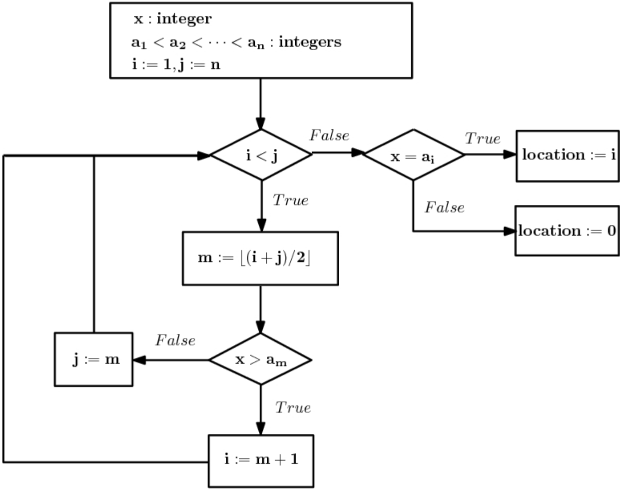
Complexity of Binary Search Algorithm



Assume $n = 2^k$

k loops
2 comparisons in each loop

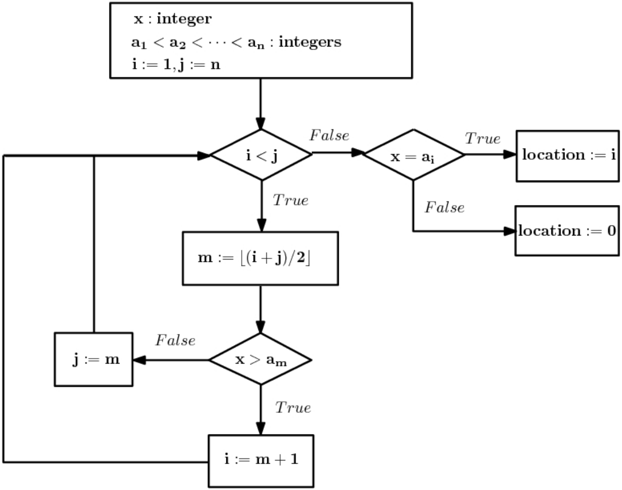
Complexity of Binary Search Algorithm



Assume $n = 2^k$

k loops
2 comparisons in each loop
1 comparison to exit the loop

Complexity of Binary Search Algorithm



Assume $n = 2^k$

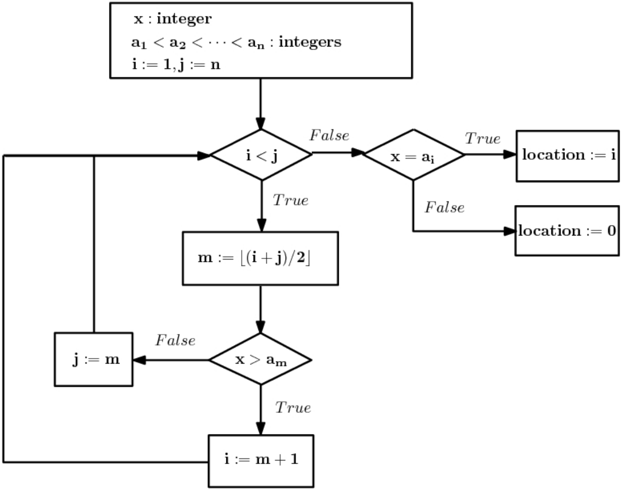
k loops

2 comparisons in each loop

1 comparison to exit the loop

1 comparison outside the loop

Complexity of Binary Search Algorithm



Assume $n = 2^k$

k loops
2 comparisons in each loop
1 comparison to exit the loop
1 comparison outside the loop
Total number of comparisons:
$2k + 1 + 1 = 2k + 2$
$= 2\lceil \log n \rceil + 2$
$= O(\log n)$

Exercise.

Exercise.

Analyze the complexity of Bubble sort and Insertion sort algorithms.

Commonly used terminologies for the complexity of algorithms	
Complexity	Terminology
$O(1)$	Constant complexity
$O(\log n)$	Logarithmic complexity
$O(n)$	Linear complexity
$O(n \log n)$	$n \log n$ complexity
$O(n^k)$	Polynomial complexity
$O(b^n), b > 1$	Exponential complexity
$O(n!)$	Factorial complexity

Let a, b be integers with $a \neq 0$. We say the integer a divides b if there is an integer m such that $b = am$.

3.4 The Integers and Division

Let a, b be integers with $a \neq 0$. We say the integer a divides b if there is an integer m such that $b = am$.

3.4 The Integers and Division

Let a, b be integers with $a \neq 0$. We say the integer a divides b if there is an integer m such that $b = am$.

If a divides b we also write:

Let a, b be integers with $a \neq 0$. We say the integer a divides b if there is an integer m such that $b = am$.

If a divides b we also write:

- b is divisible by a

3.4 The Integers and Division

Let a, b be integers with $a \neq 0$. We say the integer a divides b if there is an integer m such that $b = am$.

If a divides b we also write:

- b is divisible by a
- b is a multiple of a

Let a, b be integers with $a \neq 0$. We say the integer a divides b if there is an integer m such that $b = am$.

If a divides b we also write:

- b is divisible by a
- b is a multiple of a
- a is a factor of b

3.4 The Integers and Division

Let a, b be integers with $a \neq 0$. We say the integer a divides b if there is an integer m such that $b = am$.

If a divides b we also write:

- b is divisible by a
- b is a multiple of a
- a is a factor of b
- $a \mid b$

Let a, b be integers with $a \neq 0$. We say the integer a divides b if there is an integer m such that $b = am$.

If a divides b we also write:

- b is divisible by a
- b is a multiple of a
- a is a factor of b
- $a \mid b$

Theorem

3.4 The Integers and Division

Let a, b be integers with $a \neq 0$. We say the integer a divides b if there is an integer m such that $b = am$.

If a divides b we also write:

- b is divisible by a
- b is a multiple of a
- a is a factor of b
- $a \mid b$

Theorem

Let a, b, c be integers. Then:

3.4 The Integers and Division

Let a, b be integers with $a \neq 0$. We say the integer a divides b if there is an integer m such that $b = am$.

If a divides b we also write:

- b is divisible by a
- b is a multiple of a
- a is a factor of b
- $a \mid b$

Theorem

Let a, b, c be integers. Then:

- If $a \mid b$ and $a \mid c$ then $a \mid (b + c)$

3.4 The Integers and Division

Let a, b be integers with $a \neq 0$. We say the integer a divides b if there is an integer m such that $b = am$.

If a divides b we also write:

- b is divisible by a
- b is a multiple of a
- a is a factor of b
- $a \mid b$

Theorem

Let a, b, c be integers. Then:

- If $a \mid b$ and $a \mid c$ then $a \mid (b + c)$
- If $a \mid b$ then $a \mid bc$ for all c

3.4 The Integers and Division

Let a, b be integers with $a \neq 0$. We say the integer a divides b if there is an integer m such that $b = am$.

If a divides b we also write:

- b is divisible by a
- b is a multiple of a
- a is a factor of b
- $a \mid b$

Theorem

Let a, b, c be integers. Then:

- If $a \mid b$ and $a \mid c$ then $a \mid (b + c)$
- If $a \mid b$ then $a \mid bc$ for all c
- If $a \mid b$ and $b \mid c$ then $a \mid c$

The Division Algorithm

The Division Algorithm

- Let a be an integer and d a positive integer.

The Division Algorithm

- Let a be an integer and d a positive integer. Then there are unique integers q and r with $0 \leq r < d$ such that $a = dq + r$.

The Division Algorithm

- Let a be an integer and d a positive integer. Then there are unique integers q and r with $0 \leq r < d$ such that $a = dq + r$.
- In this division algorithm, a is called the dividend, d is the divisor, q is the quotient and r is the remainder.

The Division Algorithm

- Let a be an integer and d a positive integer. Then there are unique integers q and r with $0 \leq r < d$ such that $a = dq + r$.
- In this division algorithm, a is called the dividend, d is the divisor, q is the quotient and r is the remainder. We write

$$q = a \text{ div } d, \quad r = a \text{ mod } d$$

The Division Algorithm

- Let a be an integer and d a positive integer. Then there are unique integers q and r with $0 \leq r < d$ such that $a = dq + r$.
- In this division algorithm, a is called the dividend, d is the divisor, q is the quotient and r is the remainder. We write

$$q = a \text{ div } d, \quad r = a \bmod d$$

Example. Find the remainder and the quotient of the division:

The Division Algorithm

- Let a be an integer and d a positive integer. Then there are unique integers q and r with $0 \leq r < d$ such that $a = dq + r$.
- In this division algorithm, a is called the dividend, d is the divisor, q is the quotient and r is the remainder. We write

$$q = a \text{ div } d, \quad r = a \text{ mod } d$$

Example. Find the remainder and the quotient of the division:

(a) -23 is divided by 7

The Division Algorithm

- Let a be an integer and d a positive integer. Then there are unique integers q and r with $0 \leq r < d$ such that $a = dq + r$.
- In this division algorithm, a is called the dividend, d is the divisor, q is the quotient and r is the remainder. We write

$$q = a \text{ div } d, \quad r = a \text{ mod } d$$

Example. Find the remainder and the quotient of the division:

(a) -23 is divided by 7

(b) -125 is divided by 11

Modular Arithmetic

Let a, b be integers and m a positive integer. We say a is congruent to b modulo m is they have the same remainders when being divided by d. We use notation $a \equiv b \mod m$. If they are not congruent we write $a \not\equiv b \mod m$.

Let a, b be integers and m a positive integer. We say a is congruent to b modulo m is they have the same remainders when being divided by d. We use notation $a \equiv b \mod m$. If they are not congruent we write $a \not\equiv b \mod m$.

Some properties

Modular Arithmetic

Let a, b be integers and m a positive integer. We say a is congruent to b modulo m is they have the same remainders when being divided by d. We use notation $a \equiv b \mod m$. If they are not congruent we write $a \not\equiv b \mod m$.

Some properties

- $a \equiv b \mod m \iff a - b \equiv 0 \mod m \iff a = b + km$ for some integer k.

Modular Arithmetic

Let a, b be integers and m a positive integer. We say a is congruent to b modulo m is they have the same remainders when being divided by d. We use notation $a \equiv b \mod m$. If they are not congruent we write $a \not\equiv b \mod m$.

Some properties

- $a \equiv b \mod m \iff a - b \equiv 0 \mod m \iff a = b + km$ for some integer k.
- If $a \equiv b \mod m$ and $c \equiv d \mod m$ then $a + c \equiv b + d \mod m$ and $ac \equiv bd \mod m$

Pseudorandom numbers

Pseudorandom numbers

Pseudorandom numbers can be generated using **Linear congruential method**:

Pseudorandom numbers

Pseudorandom numbers can be generated using **Linear congruential method**:

x_0 is given, and $x_n = ax_{n-1} + c \mod m, \quad n = 2, 3, 4, \ldots$

Pseudorandom numbers

Pseudorandom numbers can be generated using **Linear congruential method**:

x_0 is given, and $x_n = ax_{n-1} + c \mod m, \quad n = 2, 3, 4, \ldots$

m is called the modulus, a is the multiplier, c is the increment and x_0 is the seed.

Cryptography.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Caesar's cipher
$f(p) = p + 3 \mod 26$

3.5 Primes and Greatest Common Divisors

A positive integer p greater than 1 is called a prime number if the only prime factors of p are 1 and p.

A positive integer p greater than 1 is called a prime number if the only prime factors of p are 1 and p. An integer greater than 1 that is not prime is called composite number.

A positive integer p greater than 1 is called a prime number if the only prime factors of p are 1 and p. An integer greater than 1 that is not prime is called composite number.

The Fundamental Theorem of Arithmetic

A positive integer p greater than 1 is called a prime number if the only prime factors of p are 1 and p. An integer greater than 1 that is not prime is called composite number.

The Fundamental Theorem of Arithmetic

Any integer greater than 1 can be written uniquely as a product of powers of distinct primes.

A positive integer p greater than 1 is called a prime number if the only prime factors of p are 1 and p. An integer greater than 1 that is not prime is called composite number.

The Fundamental Theorem of Arithmetic

Any integer greater than 1 can be written uniquely as a product of powers of distinct primes.

Theorem

3.5 Primes and Greatest Common Divisors

A positive integer p greater than 1 is called a prime number if the only prime factors of p are 1 and p. An integer greater than 1 that is not prime is called composite number.

The Fundamental Theorem of Arithmetic

Any integer greater than 1 can be written uniquely as a product of powers of distinct primes.

Theorem

There are infinitely many primes.

3.5 Primes and Greatest Common Divisors

A positive integer p greater than 1 is called a prime number if the only prime factors of p are 1 and p. An integer greater than 1 that is not prime is called composite number.

The Fundamental Theorem of Arithmetic

Any integer greater than 1 can be written uniquely as a product of powers of distinct primes.

Theorem

There are infinitely many primes.

Greatest Common Divisors, Least Common Multiples

Greatest Common Divisors, Least Common Multiples

- Let a and b be two integers, not both 0. The greatest integer d that is a divisor of both a and b is called greatest common divisor of a and b, denoted by $\gcd(a, b)$.

Greatest Common Divisors, Least Common Multiples

- Let a and b be two integers, not both 0. The greatest integer d that is a divisor of both a and b is called greatest common divisor of a and b, denoted by $\text{gcd}(a, b)$.
- Let a and b be two positive integers. The smallest positive integer d that is divisible by both a and b is called the least common multiple of a and b, denoted by $\text{lcm}(a, b)$.

Greatest Common Divisors, Least Common Multiples

- Let a and b be two integers, not both 0. The greatest integer d that is a divisor of both a and b is called greatest common divisor of a and b, denoted by $\gcd(a, b)$.
- Let a and b be two positive integers. The smallest positive integer d that is divisible by both a and b is called the least common multiple of a and b, denoted by $\text{lcm}(a, b)$.

Greatest Common Divisors, Least Common Multiples

- Let a and b be two integers, not both 0. The greatest integer d that is a divisor of both a and b is called greatest common divisor of a and b, denoted by $\gcd(a, b)$.
- Let a and b be two positive integers. The smallest positive integer d that is divisible by both a and b is called the least common multiple of a and b, denoted by $\text{lcm}(a, b)$.

To find gcd and lcd of a and b, write a, b as products of powers of distinct primes.

Greatest Common Divisors, Least Common Multiples

- Let a and b be two integers, not both 0. The greatest integer d that is a divisor of both a and b is called greatest common divisor of a and b, denoted by $\gcd(a, b)$.
- Let a and b be two positive integers. The smallest positive integer d that is divisible by both a and b is called the least common multiple of a and b, denoted by $\operatorname{lcm}(a, b)$.

To find gcd and lcd of a and b, write a, b as products of powers of distinct primes.

$$a = p_1^{a_1} p_2^{a_2} \cdots p_n^{a_n}$$
$$b = p_1^{b_1} p_2^{b_2} \cdots p_n^{b_n}$$

Greatest Common Divisors, Least Common Multiples

- Let a and b be two integers, not both 0. The greatest integer d that is a divisor of both a and b is called greatest common divisor of a and b, denoted by $\text{gcd}(a, b)$.
- Let a and b be two positive integers. The smallest positive integer d that is divisible by both a and b is called the least common multiple of a and b, denoted by $\text{lcm}(a, b)$.

To find gcd and lcd of a and b, write a, b as products of powers of distinct primes.

$$a = p_1^{a_1} p_2^{a_2} \cdots p_n^{a_n}$$
$$b = p_1^{b_1} p_2^{b_2} \cdots p_n^{b_n}$$

Then:

- $gcd(a, b) = p_1^{\min(a_1, b_1)} p_2^{\min(a_2, b_2)} \cdots p_n^{\min(a_n, b_n)}$

- Let a and b be two integers, not both 0. The greatest integer d that is a divisor of both a and b is called greatest common divisor of a and b, denoted by $\gcd(a, b)$.
- Let a and b be two positive integers. The smallest positive integer d that is divisible by both a and b is called the least common multiple of a and b, denoted by $\mathrm{lcm}(a, b)$.

To find gcd and lcd of a and b, write a, b as products of powers of distinct primes.

$$a = p_1^{a_1} p_2^{a_2} \cdots p_n^{a_n}$$
$$b = p_1^{b_1} p_2^{b_2} \cdots p_n^{b_n}$$

Then:

- $gcd(a, b) = p_1^{\min(a_1, b_1)} p_2^{\min(a_2, b_2)} \cdots p_n^{\min(a_n, b_n)}$
- $lcm(a, b) = p_1^{\max(a_1, b_1)} p_2^{\max(a_2, b_2)} \cdots p_n^{\max(a_n, b_n)}$

Greatest Common Divisors, Least Common Multiples

- Let a and b be two integers, not both 0. The greatest integer d that is a divisor of both a and b is called greatest common divisor of a and b, denoted by $\text{gcd}(a, b)$.
- Let a and b be two positive integers. The smallest positive integer d that is divisible by both a and b is called the least common multiple of a and b, denoted by $\text{lcm}(a, b)$.

To find gcd and lcd of a and b, write a, b as products of powers of distinct primes.

$$a = p_1^{a_1} p_2^{a_2} \cdots p_n^{a_n}$$
$$b = p_1^{b_1} p_2^{b_2} \cdots p_n^{b_n}$$

Then:

- $gcd(a, b) = p_1^{\min(a_1,b_1)} p_2^{\min(a_2,b_2)} \cdots p_n^{\min(a_n,b_n)}$
- $lcm(a, b) = p_1^{\max(a_1,b_1)} p_2^{\max(a_2,b_2)} \cdots p_n^{\max(a_n,b_n)}$

Theorem

Theorem

Let a, b be positive integers. Then

Theorem

Let a, b be positive integers. Then

$$ab = gcd(a, b) \cdot lcm(a, b)$$

Theorem

Let a, b be positive integers. Then

$$ab = gcd(a, b) \cdot lcm(a, b)$$

Relatively prime

- Two integers a, b are called relatively prime if $\gcd(a, b) = 1$.

Theorem

Let a, b be positive integers. Then

$$ab = gcd(a, b) \cdot lcm(a, b)$$

Relatively prime

- Two integers a, b are called relatively prime if $\gcd(a, b) = 1$.
- A set of integers are called pairwise relatively prime if any two integers in the set are relatively prime.

Theorem

Let a, b be positive integers. Then

$$ab = gcd(a, b) \cdot lcm(a, b)$$

Relatively prime

- Two integers a, b are called relatively prime if $\gcd(a, b) = 1$.
- A set of integers are called pairwise relatively prime if any two integers in the set are relatively prime.

Example. Which sets are pairwise relatively prime?

Theorem

Let a, b be positive integers. Then

$$ab = gcd(a, b) \cdot lcm(a, b)$$

Relatively prime

- Two integers a, b are called relatively prime if $\gcd(a, b) = 1$.
- A set of integers are called pairwise relatively prime if any two integers in the set are relatively prime.

Example. Which sets are pairwise relatively prime?

(a) $(13, 24, 49)$

Theorem

Let a, b be positive integers. Then

$$ab = gcd(a, b) \cdot lcm(a, b)$$

Relatively prime

- Two integers a, b are called relatively prime if $\gcd(a, b) = 1$.
- A set of integers are called pairwise relatively prime if any two integers in the set are relatively prime.

Example. Which sets are pairwise relatively prime?

(a) $(13, 24, 49)$

(b) $(14, 23, 35, 61)$

3.6 Integers and Algorithms

Representations of Integers

Representations of Integers

Let b be an integer greater than 1. Let n be a positive integer. Then n can be expressed uniquely in the form

$$n = a_k b^k + a_{k-1} b^{k-1} + \cdots + a_0,$$

where $a_0, a_1, \ldots, a_k$ are nonnegative integers and less than b.

Representations of Integers

Let b be an integer greater than 1. Let n be a positive integer. Then n can be expressed uniquely in the form

$$n = a_k b^k + a_{k-1} b^{k-1} + \cdots + a_0,$$

where $a_0, a_1, \ldots, a_k$ are nonnegative integers and less than b. This representation is called base b expansion of n, and is denoted by $n = (a_k a_{k-1} \cdots a_0)_b$.

Representations of Integers

Let b be an integer greater than 1. Let n be a positive integer. Then n can be expressed uniquely in the form

$$n = a_k b^k + a_{k-1} b^{k-1} + \cdots + a_0,$$

where $a_0, a_1, \ldots, a_k$ are nonnegative integers and less than b. This representation is called base b expansion of n, and is denoted by $n = (a_k a_{k-1} \cdots a_0)_b$.

Some important representations.

Some important representations.

- Binary expansion: 0, 1

Some important representations.

- Binary expansion: 0, 1
- Octal expansion: 0, 1, 2,..., 7

Some important representations.

- Binary expansion: 0, 1
- Octal expansion: 0, 1, 2,..., 7
- Hexadecimal expansion: 0, 1, 2,..., 9, A, B, C, D, E, F

Some important representations.

- Binary expansion: 0, 1
- Octal expansion: 0, 1, 2,..., 7
- Hexadecimal expansion: 0, 1, 2,..., 9, A, B, C, D, E, F

Example.

Some important representations.

- Binary expansion: 0, 1
- Octal expansion: 0, 1, 2,..., 7
- Hexadecimal expansion: 0, 1, 2,..., 9, A, B, C, D, E, F

Example.

(a) Find the binary expansion of 35

Some important representations.

- Binary expansion: 0, 1
- Octal expansion: 0, 1, 2,..., 7
- Hexadecimal expansion: 0, 1, 2,..., 9, A, B, C, D, E, F

Example.

(a) Find the binary expansion of 35

(b) Find the hexadecimal expansion of $(132)_5$

Some important representations.

- Binary expansion: 0, 1
- Octal expansion: 0, 1, 2,..., 7
- Hexadecimal expansion: 0, 1, 2,..., 9, A, B, C, D, E, F

Example.

(a) Find the binary expansion of 35

(b) Find the hexadecimal expansion of $(132)_5$

Hexadecimal, Octal and binary expansions of integers from 0 though 15

Hexadecimal, Octal and binary expansions of integers from 0 though 15

Decimal	0	1	2	3	4	5	6	7
Hexadecimal	0	1	2	3	4	5	6	7
Octal	0	1	2	3	4	5	6	7
Binary	0	1	10	11	100	101	110	111
Decimal	8	9	10	11	12	13	14	15
Hexadecimal	8	9	A	B	C	D	E	F
Octal	10	11	12	13	14	15	16	17
Binary	1000	1001	1010	1011	1100	1101	1110	1111

Algorithms for Integer Operations

Algorithms for Integer Operations

Let a and b be in the binary expansions.

Let a and b be in the binary expansions.

$$a = (a_{n-1}a_{n-2}\dots a_0)_2,\ b = (b_{n-1}b_{n-2}\dots b_0)_2$$

Addition algorithm.

Algorithms for Integer Operations

Let a and b be in the binary expansions.

$$a = (a_{n-1}a_{n-2}\ldots a_0)_2,\ \ b = (b_{n-1}b_{n-2}\ldots b_0)_2$$

Addition algorithm.

Procedure Addition (a, b)
$c := 0$
for $j := 0$ **to** $n-1$
  $d := \lfloor (a_j + b_j + c)/2 \rfloor$
  $s_j := a_j + b_j + c - 2d$
  $c := d$
$s_n := c$

Multiplication algorithm.

Multiplication algorithm.

$$a = (a_{n-1}a_{n-2}\dots a_0)_2,\ \ b = (b_{n-1}b_{n-2}\dots b_0)_2$$

Multiplication algorithm.

$$a = (a_{n-1}a_{n-2}\dots a_0)_2,\ \ b = (b_{n-1}b_{n-2}\dots b_0)_2$$

Procedure Multiplication (a, b)
for $j := 0$ **to** $n - 1$
    if $b_j = 1$ **then** $c_j := a$ shifted j places
    else $c_j := 0$
$p := 0$
for $j := 0$ **to** $n - 1$
    $p := p + c_j$

Euclidean Algorithm

Euclidean Algorithm

Theorem

Euclidean Algorithm

Theorem

Let $a > b$ be positive integers. Put $r = a \mod b$. Then

$$gcd(a, b) = gcd(b, r)$$

Euclidean Algorithm

Theorem

Let $a > b$ be positive integers. Put $r = a \mod b$. Then

$$gcd(a, b) = gcd(b, r)$$

Procedure GCD(a, b: positive integers)

$x := a$

$y := b$

while $y \neq 0$

$\quad r := x \mod y$

$\quad x := y$

$\quad y := r$

Print(x)

Modular Exponentiation

Modular Exponentiation

Problem:

Modular Exponentiation

Problem: Let b and m be positive integers. Compute $b^n \mod m$.

Modular Exponentiation

Problem: Let b and m be positive integers. Compute $b^n \mod m$.

Example. Compute $3^{71} \mod 13$.

Modular Exponentiation

Problem: Let b and m be positive integers. Compute $b^n \mod m$.

Example. Compute $3^{71} \mod 13$.

Procedure ModExp(b, m: positive integers, $n = (a_k \dots a_0)_2$)
$x := 1$
$power := b \mod m$
for $i := 0$ **to** k

  if $a_i = 1$ **then** $x := (x * power) \mod m$

  $power := (power * power) \mod m$

Print(x)