Chapter 5. Searching and sorting algorithms

5.3 Sorting: Introduction
5.4 Selection sort
5.5 Insertion sort

Ordering the elements of a list is a problem that occurs in many contexts.

sorting is putting elements into a list in which the elements are in increasing (or decreasing) order.

Ordering the elements of a list is a problem that occurs in many contexts.

sorting is putting elements into a list in which the elements are in increasing (or decreasing) order.

Example 1:
Given a list {1, 5, 2, 7, 3, 4}, the sorted list will be {1, 2, 3, 4, 5, 7}

Given a list {a, g, s, d, f, p} the sorted list will be {a, d, f, g, p, s}

5.3 Sorting

Ordering the elements of a list is a problem that occurs in many contexts.

sorting is putting elements into a list in which the elements are in increasing (or decreasing) order.

Example 1:
Given a list {1, 5, 2, 7, 3, 4}, the sorted list will be {1, 2, 3, 4, 5, 7}

Given a list {a, g, s, d, f, p} the sorted list will be {a, d, f, g, p, s}

There are many sorting algorithms. Some algorithms are easy to implement, some a more efficient, some take advantage of particular computer architecture, and so on.

Some of the names:
Bubble sort
Insertion sort
Merge sort
Selection sort
Quicksort

Bubble sort

Let's consider Bubble sort.
It is a simplest one, but not an efficient algorithm

idea: compares adjacent elements and interchanges them if necessary

Bubble sort

Let's consider Bubble sort.
It is a simplest one, but not an efficient algorithm

idea: compares adjacent elements and interchanges them if necessary

Input: real numbers with $n \geq 2$
Output: $a_1, a_2, ..., a_n$ is in increasing order

procedure *bubblesort*($a_1,...,a_n$)
For *i := 1* **to** *n-1*
    F**or** *j := 1* **to** *n-i*
        I**f** ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
    End-for
End-for

Bubble sort

Let's consider Bubble sort.
It is a simplest one, but not an efficient algorithm

idea: compares adjacent elements and interchanges them if necessary

procedure *bubblesort*($a_1,...,a_n$)
For *i := 1* **to** *n-1*
    F**or** *j := 1* **to** *n-i*
        I**f** ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
    End-for
End-for

summary: the bubble sort is done in n-1 passes.
During *each pass* we start at the beginning of the list and compare first and second elements: if the first element is larger than the second – we interchange them, and do nothing otherwise. Then we compare the second and the third elements (and interchange them if the second element is larger than the third one). And so on – till we reach the end of the list.

Bubble sort

Example 2: Let's see the work of the Bubble sort on the list {3, 1, 7, 5, 0}

First pass (i=1):

a_5 0
a_4 5
a_3 7
a_2 1
a_1 3

j=1

$a_1 > a_2$?

3 > 1 ?

For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

Example 2: Let's see the work of the Bubble sort on the list {3, 1, 7, 5, 0}

First pass (i=1):

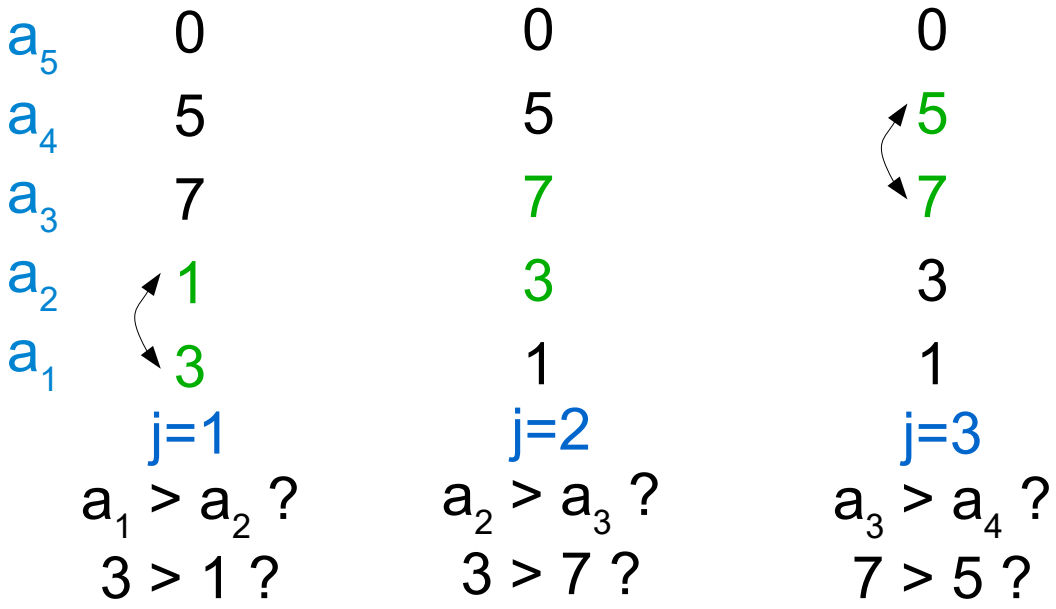
a_5	0	0
a_4	5	5
a_3	7	7
a_2	1	3
a_1	3	1
	j=1	j=2
	$a_1 > a_2$?	$a_2 > a_3$?
	3 > 1 ?	3 > 7 ?

For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

Example 2: Let's see the work of the Bubble sort on the list {3, 1, 7, 5, 0}

First pass (i=1):

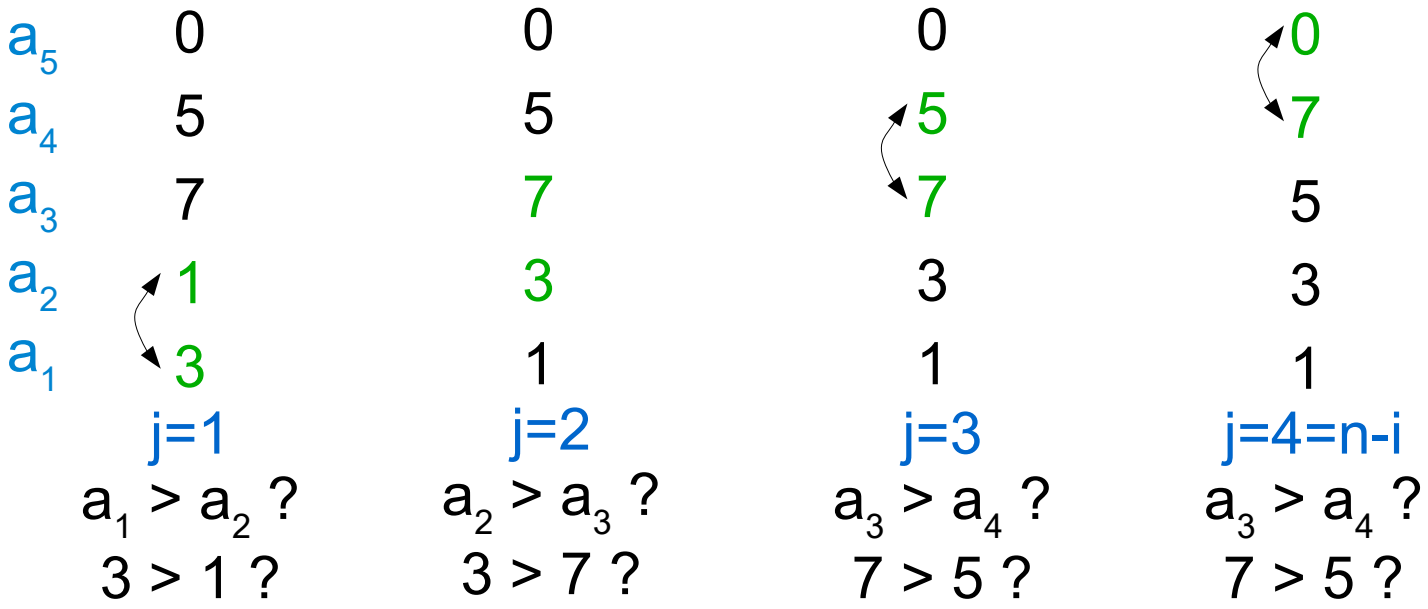


For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

Example 2: Let's see the work of the Bubble sort on the list {3, 1, 7, 5, 0}

First pass (i=1):

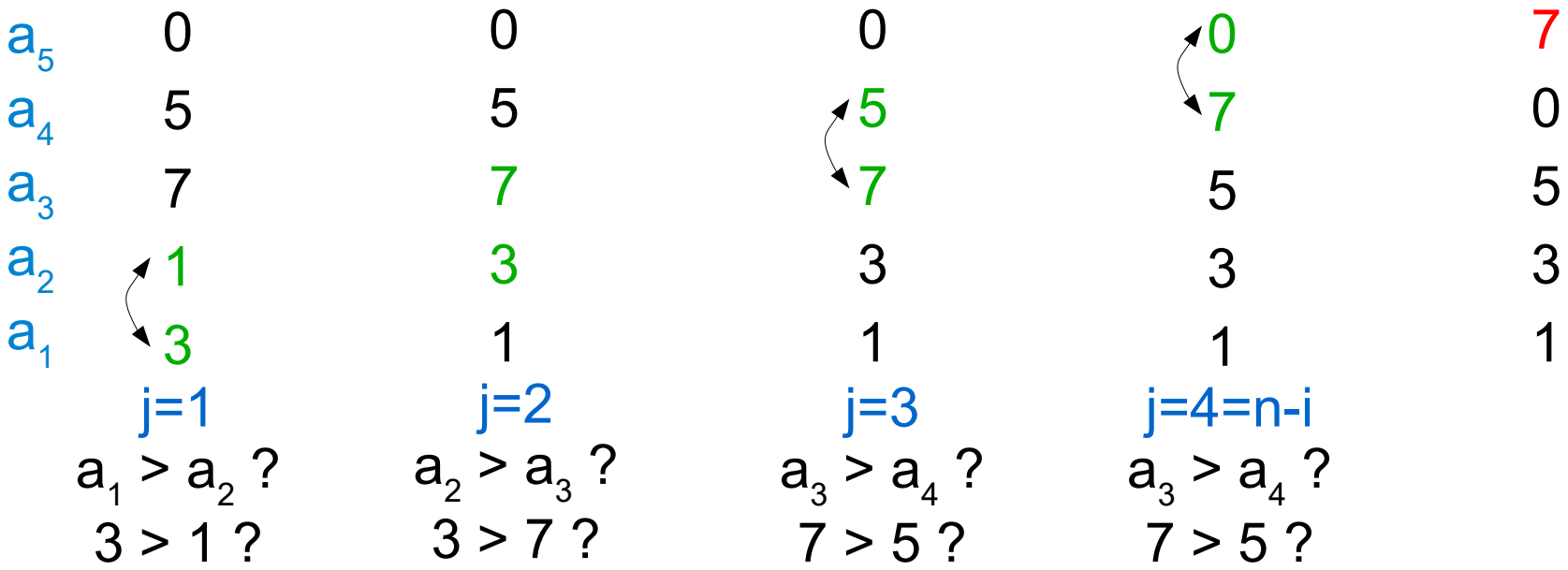


For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

Example 2: Let's see the work of the Bubble sort on the list {3, 1, 7, 5, 0}

First pass (i=1):



For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

Example 2: Let's see the work of the Bubble sort on the list {3, 1, 7, 5, 0}

Second pass (i=2):

a_5 7
a_4 0
a_3 5
a_2 3
a_1 1

j=1

$a_1 > a_2$?

1 > 3 ?

For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

Example 2: Let's see the work of the Bubble sort on the list {3, 1, 7, 5, 0}

Second pass (i=2):

a_5	7	7
a_4	0	0
a_3	5	5
a_2	3	3
a_1	1	1
	j=1	j=2
	$a_1 > a_2$?	$a_2 > a_3$?
	1 > 3 ?	3 > 5 ?

For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

Example 2: Let's see the work of the Bubble sort on the list {3, 1, 7, 5, 0}

Second pass (i=2):

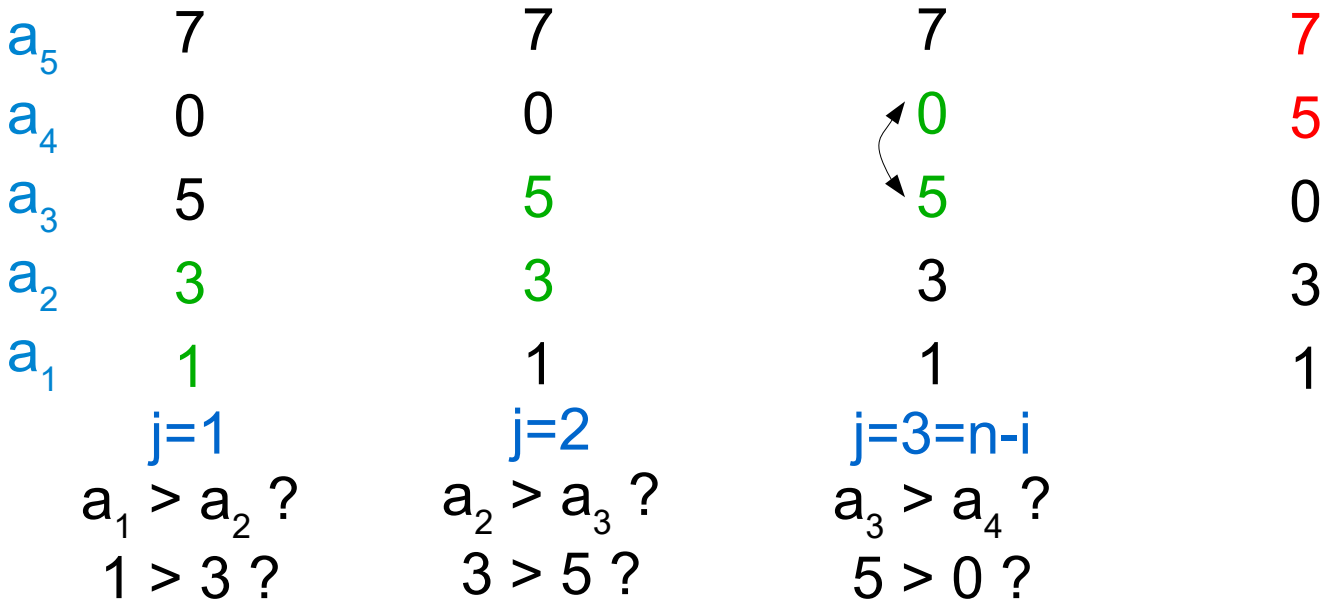
a_5	7	7	7
a_4	0	0	0
a_3	5	5	5
a_2	3	3	3
a_1	1	1	1
	j=1	j=2	j=3=n-i
	$a_1 > a_2$?	$a_2 > a_3$?	$a_3 > a_4$?
	1 > 3 ?	3 > 5 ?	5 > 0 ?

For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

Example 2: Let's see the work of the Bubble sort on the list {3, 1, 7, 5, 0}

Second pass (i=2):



For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

Example 2: Let's see the work of the Bubble sort on the list {3, 1, 7, 5, 0}

Third pass (i=3):

a_5 7
a_4 5
a_3 0
a_2 3
a_1 1

j=1

$a_1 > a_2$?

1 > 3 ?

For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

Example 2: Let's see the work of the Bubble sort on the list {3, 1, 7, 5, 0}

Third pass (i=3):

a_5	7	7
a_4	5	5
a_3	0	0
a_2	3	3
a_1	1	1
	j=1	j=2=n-i
	$a_1 > a_2$?	$a_2 > a_3$?
	1 > 3 ?	3 > 0 ?

For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

Example 2: Let's see the work of the Bubble sort on the list {3, 1, 7, 5, 0}

Third pass (i=3):

a_5	7	7	7
a_4	5	5	5
a_3	0	0	3
a_2	3	3	0
a_1	1	1	1
	j=1	j=2=n-i	
	$a_1 > a_2$?	$a_2 > a_3$?	
	1 > 3 ?	3 > 0 ?	

For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

Example 2: Let's see the work of the Bubble sort on the list {3, 1, 7, 5, 0}

Fourth pass (i=4):

a_5 7
a_4 5
a_3 3
a_2 0
a_1 1

j=1=n-i

$a_1 > a_2$?

1 > 0 ?

For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

Example 2: Let's see the work of the Bubble sort on the list {3, 1, 7, 5, 0}

Fourth pass (i=4):

a_5	7	7
a_4	5	5
a_3	3	3
a_2	0	1
a_1	1	0

j=1=n-i

$a_1 > a_2$?

1 > 0 ?

For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

Example 2: Let's see the work of the Bubble sort on the list {3, 1, 7, 5, 0}

Fourth pass (i=4):

a_5	7	7	7
a_4	5	5	5
a_3	3	3	3
a_2	0	1	1
a_1	1	0	0

j=1=n-i

$a_1 > a_2$?

1 > 0 ?

Stop

For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

procedure *bubblesort*(a_1,...,a_n)
For *i := 1* **to** *n-1*
    F**or** *j := 1* **to** *n-i*
        I**f** (a_j > a_{j+1}) **,** interchange a_j and a_{j+1}
    End-for
End-for

How many iterations (comparisons) are performed on an n-element list?

Bubble sort

procedure *bubblesort*(a_1,...,a_n)
For *i := 1* **to** *n-1*
    F**or** *j := 1* **to** *n-i*
        I**f** ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
    End-for
End-for

How many iterations (comparisons) are performed on an n-element list?
for i=1 n-1
for i=2 n-2
for i=3 n-3
....
for i=n-1 n-(n-1)

Bubble sort

procedure *bubblesort*(a_1,...,a_n)

For *i := 1* **to** *n-1*

  For *j := 1* **to** *n-i*

    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}

  End-for

End-for

How many iterations (comparisons) are performed on an n-element list?

for i=1 n-1

for i=2 n-2

for i=3 n-3

....

for i=n-1 n-(n-1)

Therefore we have the following sum:

(n-1) + (n-2) + (n-3) + (n-4) + ... + (n-(n-1)) =

Bubble sort

procedure *bubblesort*(a_1,...,a_n)

For *i := 1* **to** *n-1*

  For *j := 1* **to** *n-i*

    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}

  End-for

End-for

How many iterations (comparisons) are performed on an n-element list?

for i=1 n-1

for i=2 n-2

for i=3 n-3

....

for i=n-1 n-(n-1)

Therefore we have the following sum:

(n-1) + (n-2) + (n-3) + (n-4) + ... + (n-(n-1)) = (n-1)*n – (1+2+3+4+...(n-1))=

i=1 i=2 i=3 i=4 i=n-1

Bubble sort

procedure *bubblesort*($a_1,...,a_n$)

For *i := 1* **to** *n-1*

  For *j := 1* **to** *n-i*

    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}

  End-for

End-for

How many iterations (comparisons) are performed on an n-element list?

for i=1 n-1
for i=2 n-2
for i=3 n-3
....
for i=n-1 n-(n-1)

arithmetic progression

Therefore we have the following sum:

(n-1) + (n-2) + (n-3) + (n-4) + ... + (n-(n-1)) = (n-1)*n – (1+2+3+4+...(n-1))=

i=1 i=2 i=3 i=4 i=n-1

$n^2 – n – (1+(n-1)) \cdot \frac{n-1}{2} =$

Bubble sort

procedure *bubblesort*(a_1,...,a_n)

For *i := 1* **to** *n-1*

  For *j := 1* **to** *n-i*

    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}

  End-for

End-for

How many iterations (comparisons) are performed on an n-element list?

for i=1 n-1
for i=2 n-2
for i=3 n-3
....
for i=n-1 n-(n-1)

arithmetic progression ↓

Therefore we have the following sum:

(n-1) + (n-2) + (n-3) + (n-4) + ... + (n-(n-1)) = (n-1)*n – (1+2+3+4+...(n-1))=

i=1 i=2 i=3 i=4 i=n-1

$n^2 - n - (1+(n-1)) \cdot \frac{n-1}{2}$ = $n^2/2 - n/2$ - quadratic

5.4 Selection sort

Selection sort is another sorting algorithm. It is also a slow algorithm.

Input: $a_1,...,a_n$: real numbers with $n \geq 2$
Output: $a_1, a_2, ..., a_n$ is in increasing order

procedure *insertion sort*($a_1,...,a_n$)

```
For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_min), min := j
    End-for
    temp := a_i
    a_i := a_min
    a_min := temp
End-for
```

Here is a link to a nice video demonstrating this algorithm's work:
https://www.youtube.com/watch?v=f8hXR_Hvybo

5.4 Selection sort

Selection sort is another sorting algorithm. It is also a slow algorithm.

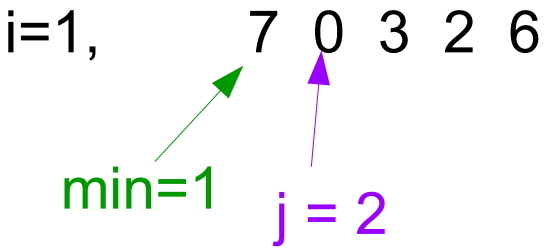
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

Summary: the algorithm starts by finding the smallest value in the sequence and swapping it with the value in the first position. Hence our first position is sorted.
It proceeds then by searching for the next smallest element (starting from position 2), and swaps it with the value in position 2. Therefore, two first positions are occupied by the values in increasing order.
And so forth,

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

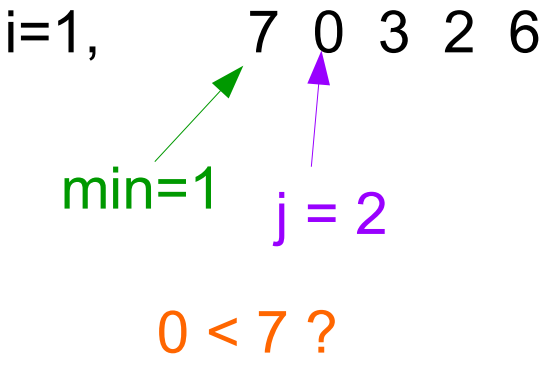


procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



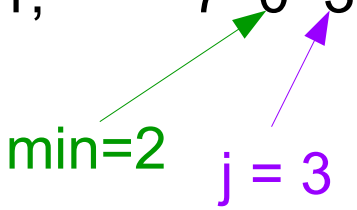
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=1, 7 0 3 2 6



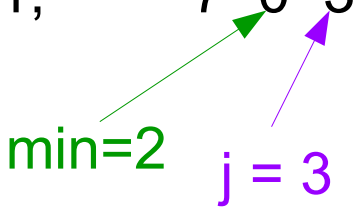
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=1, 7 0 3 2 6



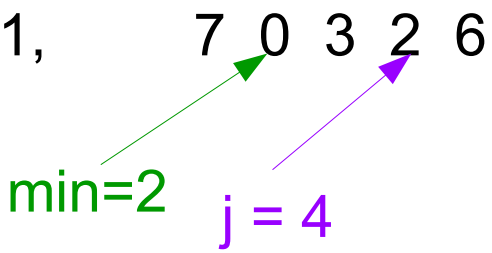
3 < 0 ?

procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

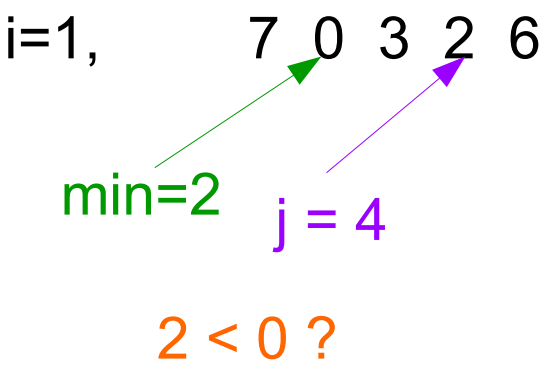


procedure *insertion sort*($a_1,...,a_n$)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If ($a_j < a_{min}$), min := j
    End-for
    temp := a_i
    $a_i := a_{min}$
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

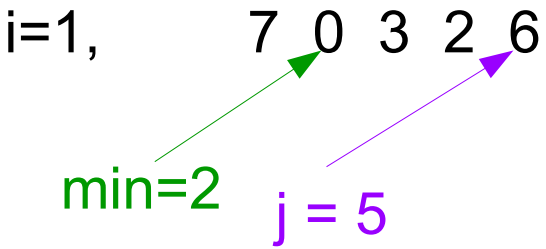


procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

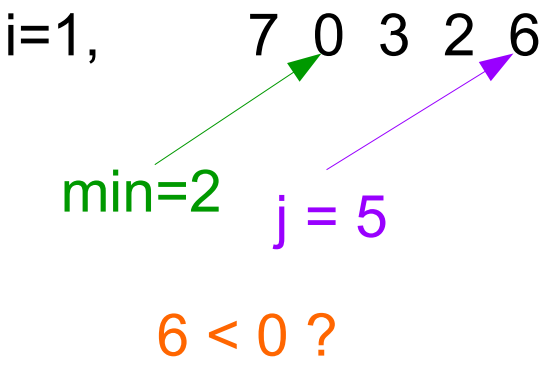


procedure *insertion sort*($a_1,...,a_n$)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If ($a_j < a_{min}$), min := j
    End-for
    temp := a_i
    $a_i := a_{min}$
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

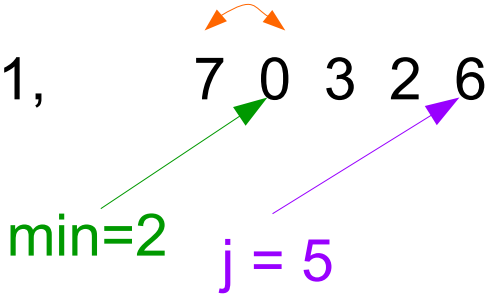


procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

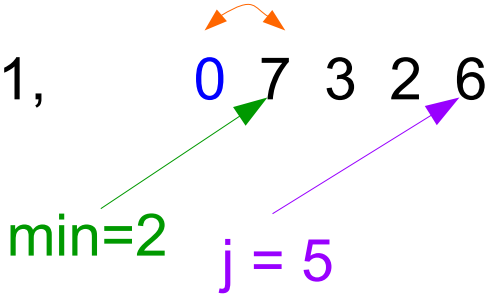


procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



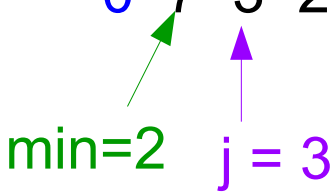
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=2, 0 7 3 2 6



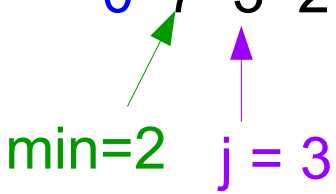
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=2, 0 7 3 2 6



3 < 7 ?

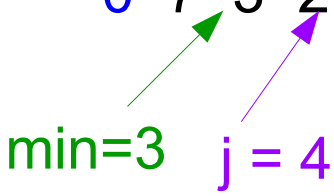
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=2, 0 7 3 2 6



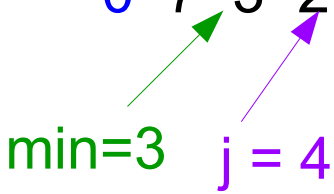
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=2, 0 7 3 2 6



2 < 3 ?

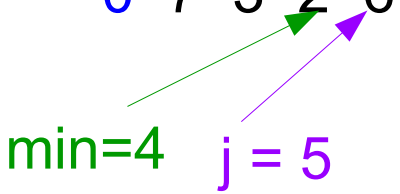
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=2, 0 7 3 2 6



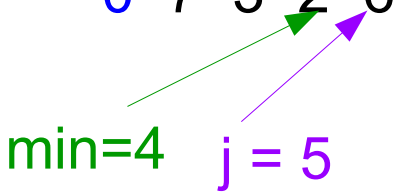
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=2, 0 7 3 2 6



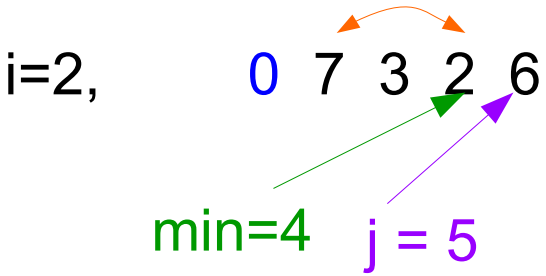
6 < 2 ?

procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



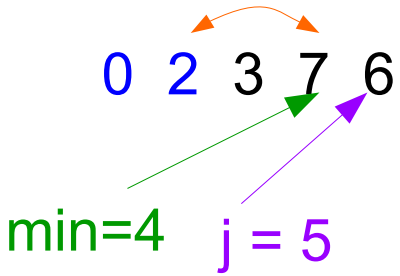
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=2,



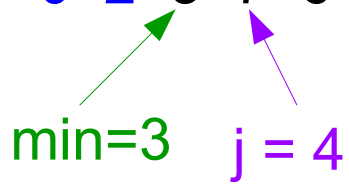
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=3, 0 2 3 7 6



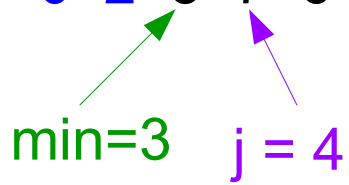
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=3, 0 2 3 7 6



7 < 3 ?

procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=3, 0 2 3 7 6

6 < 3 ?

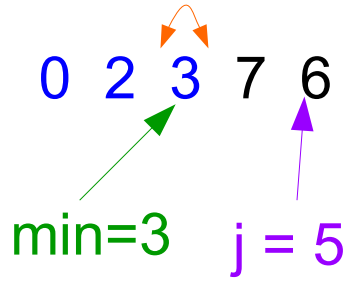
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=3,



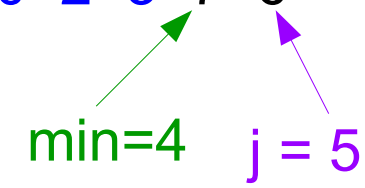
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=4,



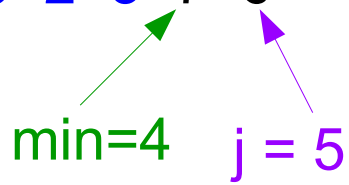
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=4, 0 2 3 7 6



6 < 7 ?

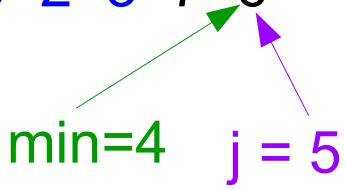
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=4, 0 2 3 7 6



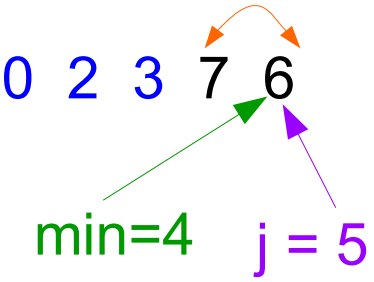
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=4,



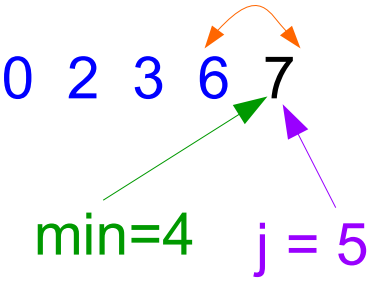
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=4,



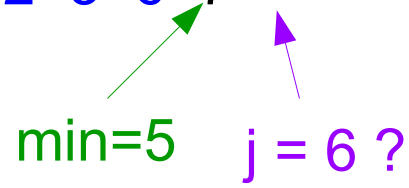
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=5, 0 2 3 6 7



procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If (a_j < a_{min}), min := j
    End-for
    temp := a_i
    a_i := a_{min}
    a_{min} := temp
End-for

5.4 Selection sort

Example:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

i=5, 0 2 3 6 7

procedure *insertion sort*($a_1,...,a_n$)

For i := 1 to n
    min := i
    For j: = i+1 to n
        If ($a_j < a_{min}$), min := j
    End-for
    temp := a_i
    $a_i := a_{min}$
    a_{min} := temp
End-for

5.5 Insertion sort

Now, let's consider Insertion sort.
It is a simple algorithm, but still not an efficient one usually

Input: $a_1,...,a_n$: real numbers with $n \geq 2$

Output: $a_1, a_2, ..., a_n$ is in increasing order

```
procedure insertion sort(a1,...,an)
For j := 2 to n
    i : = 1
    While (aj > ai)
        i := i+1
    End-while
    m := aj
    For k := 0 to j-i-1
        aj-k := aj-k-1
    End-for
    ai := m
End-for
```

Here is a link to a nice video demonstrating work of this algorithm:
https://www.youtube.com/watch?v=DFG-XuyPYUQ

5.5 Insertion sort

procedure *insertion sort*(a_1,...,a_n)

For *j := 2* **to** *n*

i : = 1

While ($a_j > a_i$)

i := i+1

End-while

$m := a_j$

For *k := 0* **to** *j-i-1*

$a_{j-k} := a_{j-k-1}$

End-for

$a_i := m$

End-for

summary: insertion sort starts with the second element.
It compares this element to the first one. If it is smaller than the first one it shifts the first element one place to the right, and places this "ex-second element into the first position. If not, does nothing.

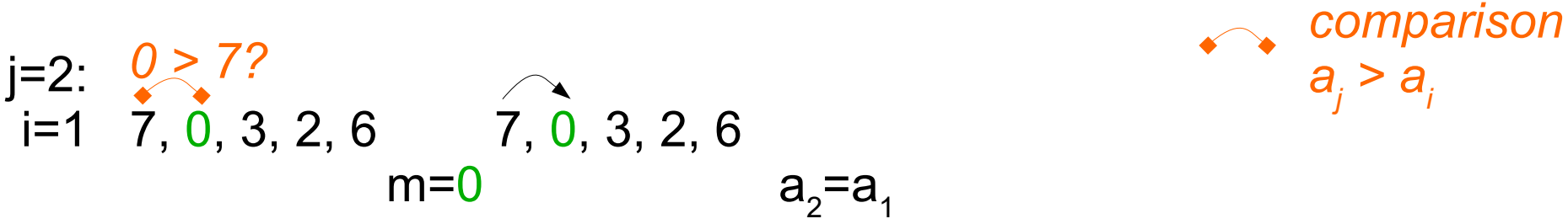
Then it takes the third element and compares it with the first element. If it is smaller than the first one, the first and the second elements are shifted one place to the right, then the "ex-third" element is placed into the first position. If not, the third element is compared to the second one and procedure is repeated. If the third element is not less than the first and the second, nothing is done.
And so on.

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

j=2:
i=1 7, 0, 3, 2, 6

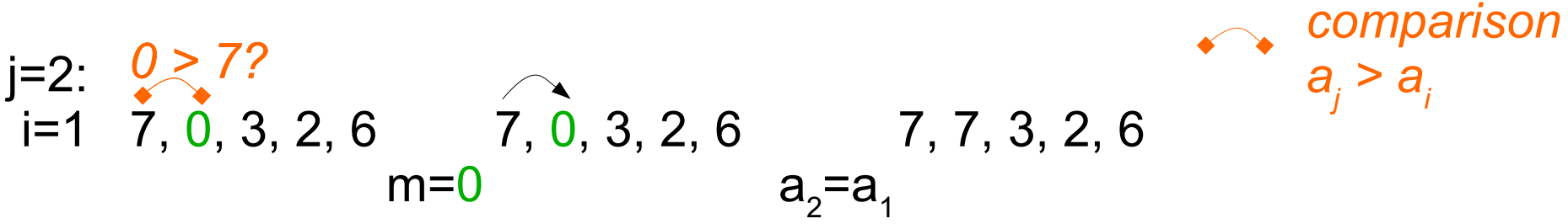
```
For j := 2 to n
    i : = 1
    While (a_j > a_i)
        i := i+1
    End-while
    m := a_j
    For k := 0 to j-i-1
        a_j-k := a_j-k-1
    End-for
    a_i := m
End-for
```

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



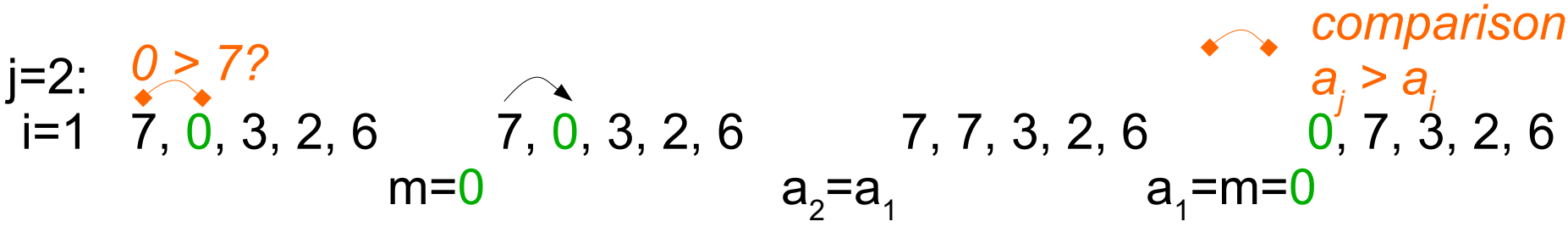

```
For j := 2 to n
    i : = 1
    While (a_j > a_i)
        i := i+1
    End-while
    m := a_j
    For k := 0 to j-i-1
        a_(j-k) := a_(j-k-1)
    End-for
    a_i := m
End-for
```

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



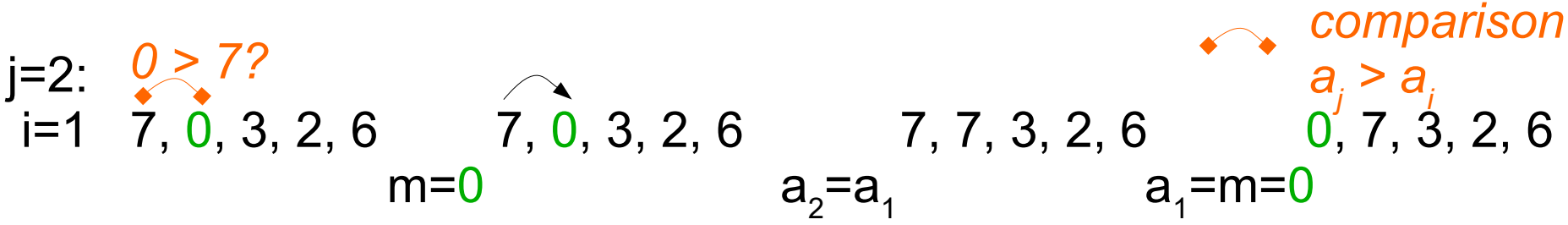
For *j := 2* **to** *n*
    i : = 1
    While ($a_j > a_i$)
        i := i+1
    End-while
    m := a_j
    For *k := 0* **to** *j-i-1*
        a_{j-k} := a_{j-k-1}
    End-for
    a_i := m
End-for

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



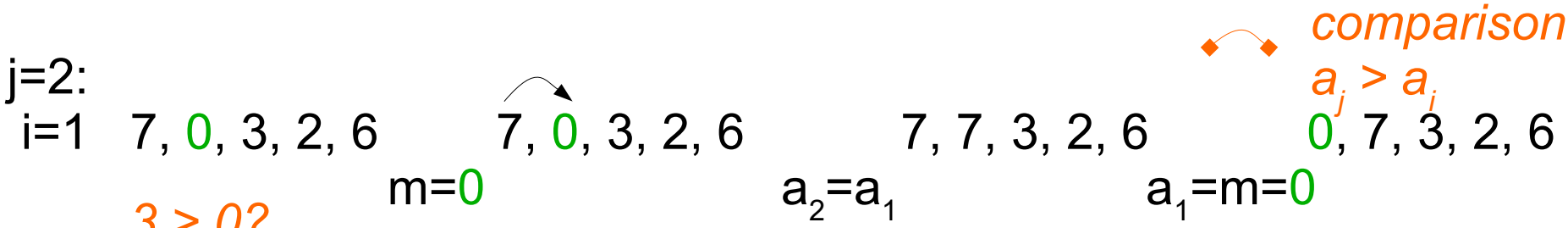
For *j := 2* **to** *n*
    i : = 1
    While ($a_j > a_i$)
        i := i+1
    End-while
    m := a_j
    For *k := 0* **to** *j-i-1*
        a_{j-k} := a_{j-k-1}
    End-for
    a_i := m
End-for

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



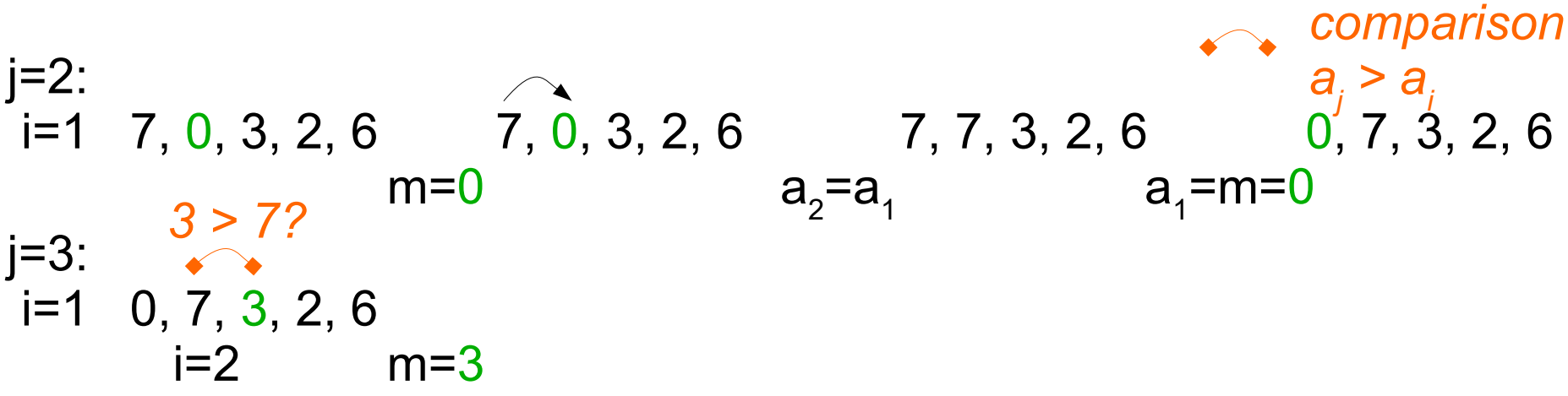
For *j := 2* **to** *n*
    i : = 1
    While ($a_j > a_i$)
        i := i+1
    End-while
    m := a_j
    For *k := 0* **to** *j-i-1*
        $a_{j-k} := a_{j-k-1}$
    End-for
    $a_i := m$
End-for

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



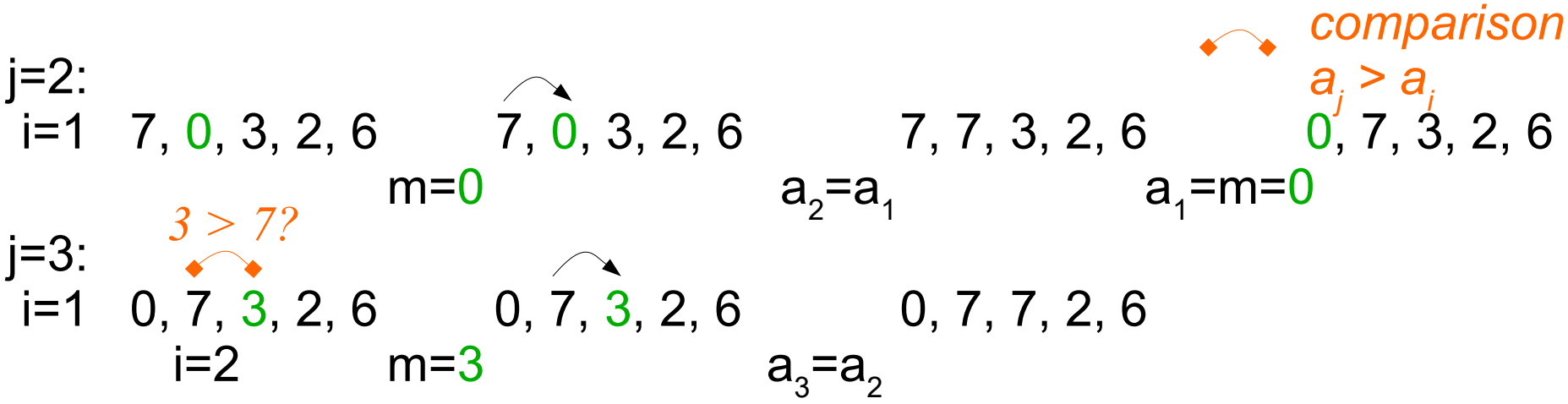
For *j := 2* **to** *n*
    i : = 1
    While ($a_j > a_i$)
        i := i+1
    End-while
    m := a_j
    For *k := 0* **to** *j-i-1*
        $a_{j-k} := a_{j-k-1}$
    End-for
    $a_i := m$
End-for

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



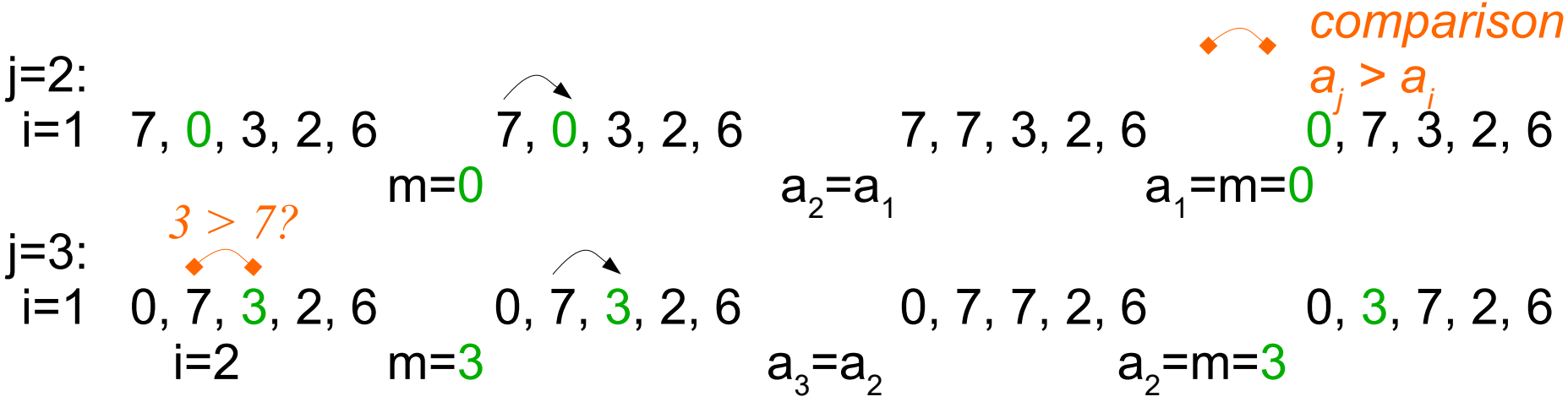
For *j := 2* **to** *n*
    i : = 1
    While ($a_j > a_i$)
        i := i+1
    End-while
    m := a_j
    For *k := 0* **to** *j-i-1*
        a_{j-k} := a_{j-k-1}
    End-for
    a_i := m
End-for

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



For *j := 2* **to** *n*
    i : = 1
    While ($a_j > a_i$)
        i := i+1
    End-while
    m := a_j
    For *k := 0* **to** *j-i-1*
        a_{j-k} := a_{j-k-1}
    End-for
    a_i := m
End-for

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



For *j := 2* **to** *n*
    i : = 1
    While ($a_j > a_i$)
        i := i+1
    End-while
    m := a_j
    For *k := 0* **to** *j-i-1*
        a_{j-k} := a_{j-k-1}
    End-for
    a_i := m
End-for

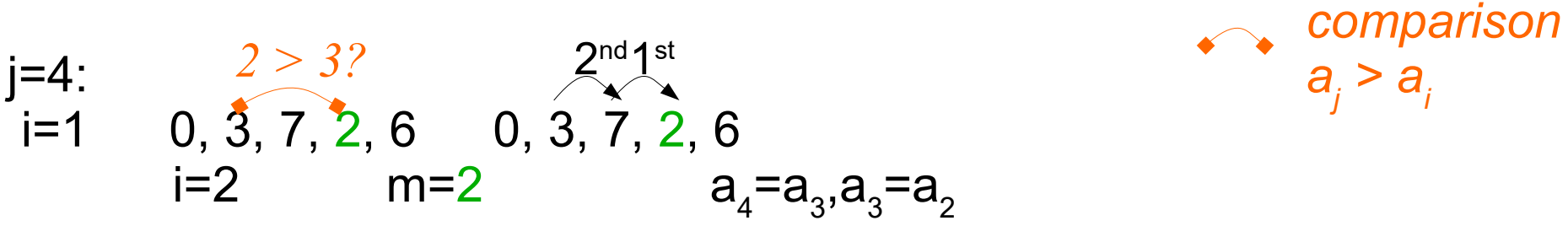
Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}


```
For j := 2 to n
    i : = 1
    While (a_j > a_i)
        i := i+1
    End-while
    m := a_j
    For k := 0 to j-i-1
        a_(j-k) := a_(j-k-1)
    End-for
    a_i := m
End-for
```

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}

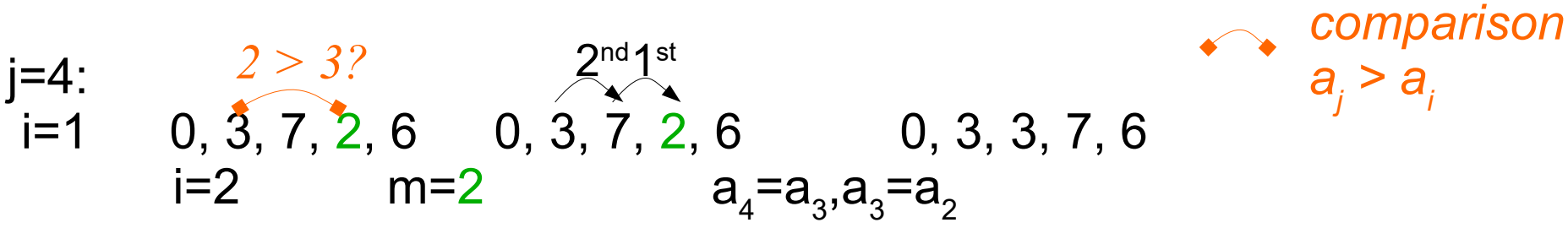

```
For j := 2 to n
    i : = 1
    While (a_j > a_i)
        i := i+1
    End-while
    m := a_j
    For k := 0 to j-i-1
        a_(j-k) := a_(j-k-1)
    End-for
    a_i := m
End-for
```

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



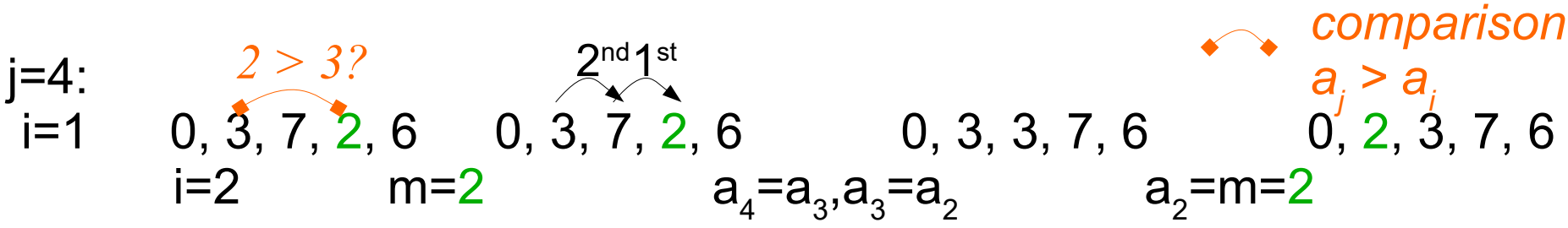

```
For j := 2 to n
    i : = 1
    While (a_j > a_i)
        i := i+1
    End-while
    m := a_j
    For k := 0 to j-i-1
        a_{j-k} := a_{j-k-1}
    End-for
    a_i := m
End-for
```

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



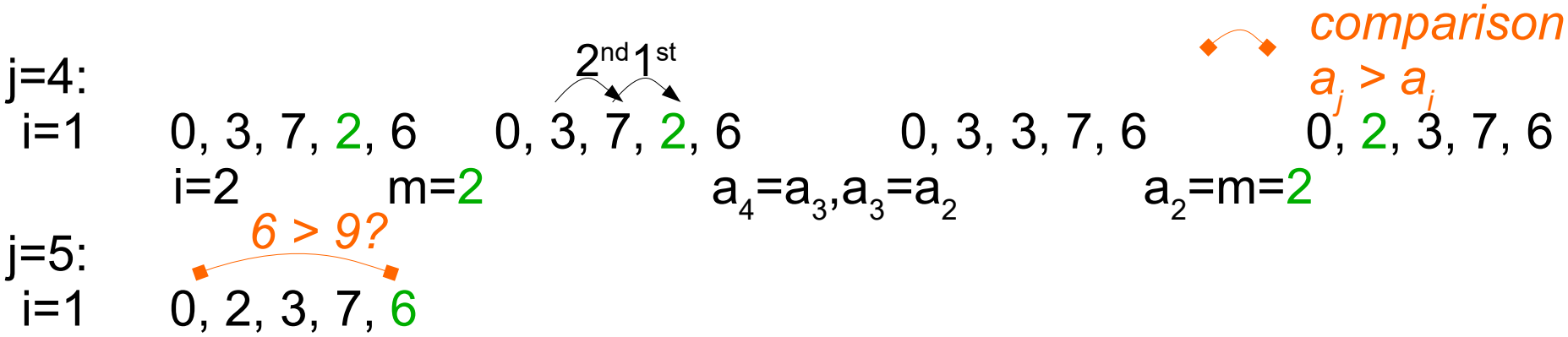
For *j := 2* **to** *n*
    i : = 1
    While ($a_j > a_i$)
        i := i+1
    End-while
    m := a_j
    For *k := 0* **to** *j-i-1*
        a_{j-k} := a_{j-k-1}
    End-for
    a_i := m
End-for

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



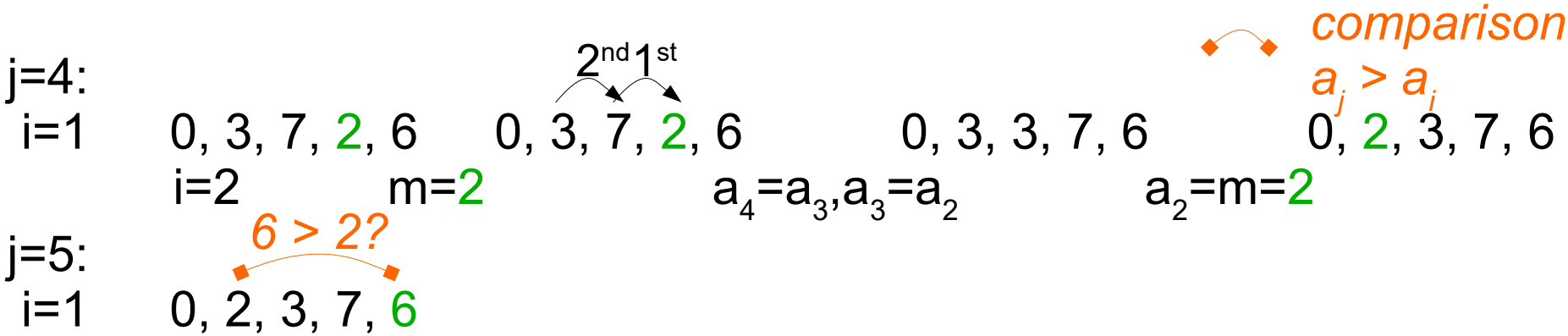
For *j := 2* **to** *n*
  i : = 1
  While ($a_j > a_i$)
    i := i+1
  End-while
  m := a_j
  For *k := 0* **to** *j-i-1*
    a_{j-k} := a_{j-k-1}
  End-for
  a_i := m
End-for

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



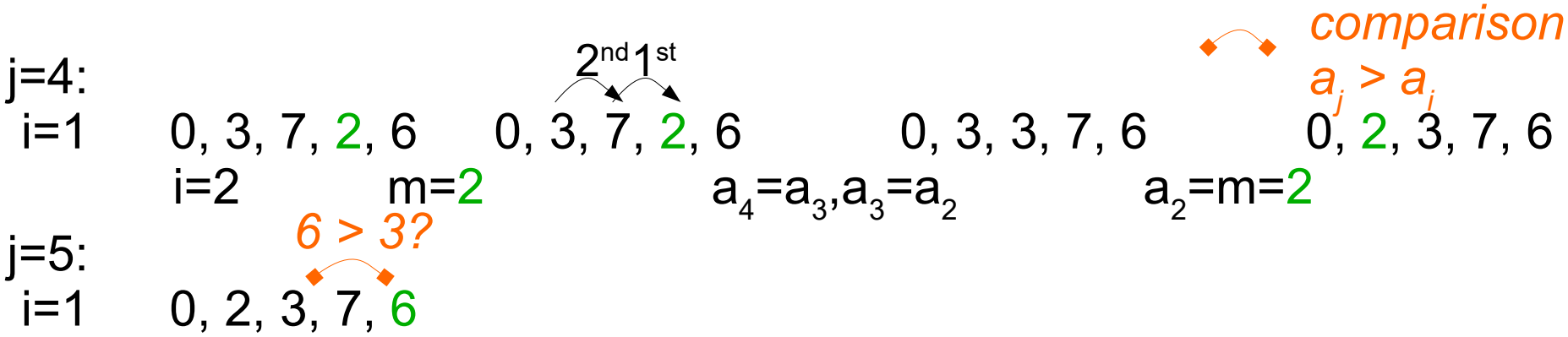
For *j := 2* **to** *n*
    i : = 1
    While ($a_j > a_i$)
        i := i+1
    End-while
    m := a_j
    For *k := 0* **to** *j-i-1*
        a_{j-k} := a_{j-k-1}
    End-for
    a_i := m
End-for

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



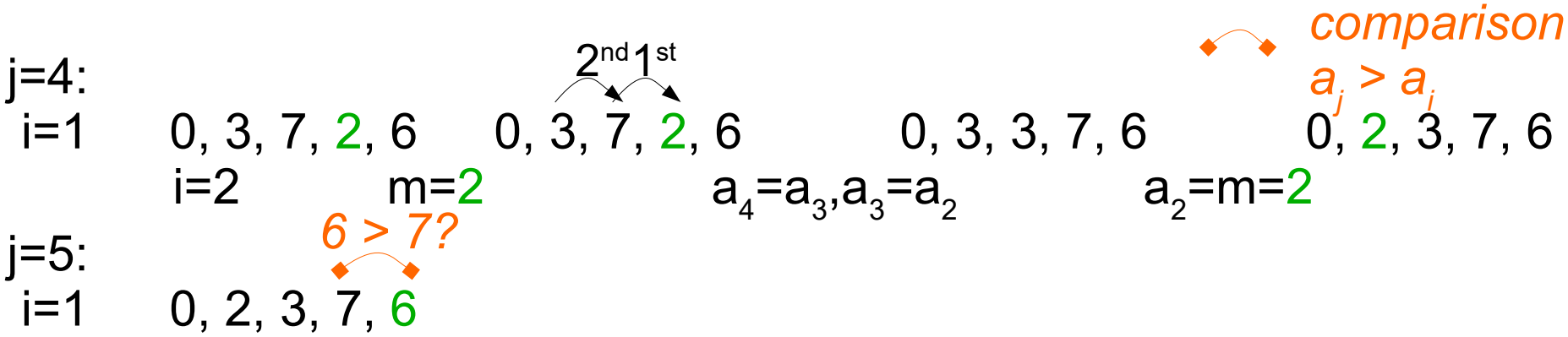
For *j := 2* **to** *n*
    i : = 1
    While ($a_j > a_i$)
        i := i+1
    End-while
    m := a_j
    For *k := 0* **to** *j-i-1*
        a_{j-k} := a_{j-k-1}
    End-for
    a_i := m
End-for

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



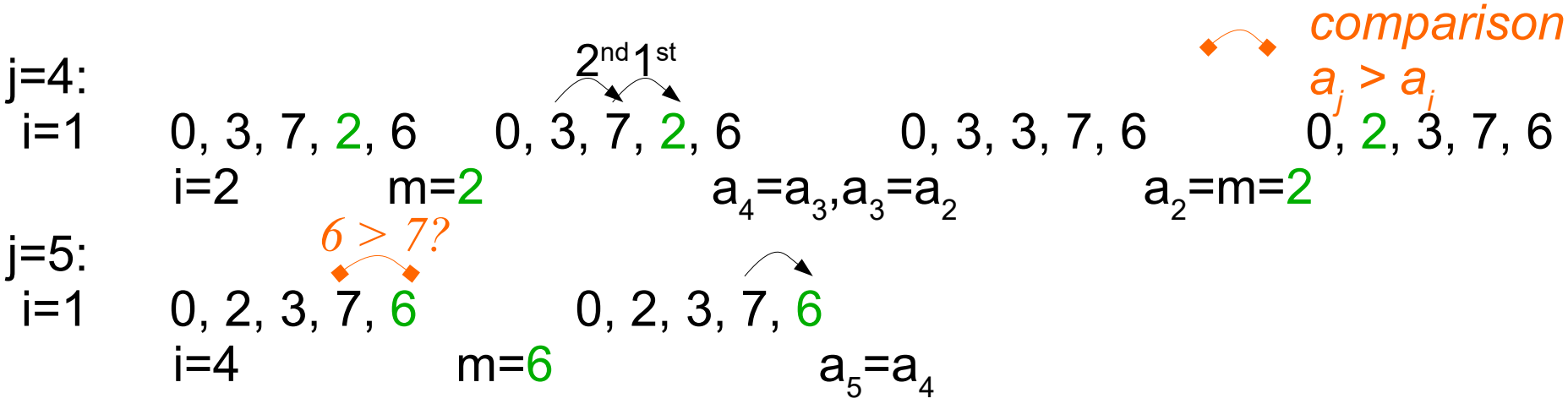
For *j := 2* **to** *n*
    i : = 1
    While ($a_j > a_i$)
        i := i+1
    End-while
    m := a_j
    For *k := 0* **to** *j-i-1*
        a_{j-k} := a_{j-k-1}
    End-for
    a_i := m
End-for

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



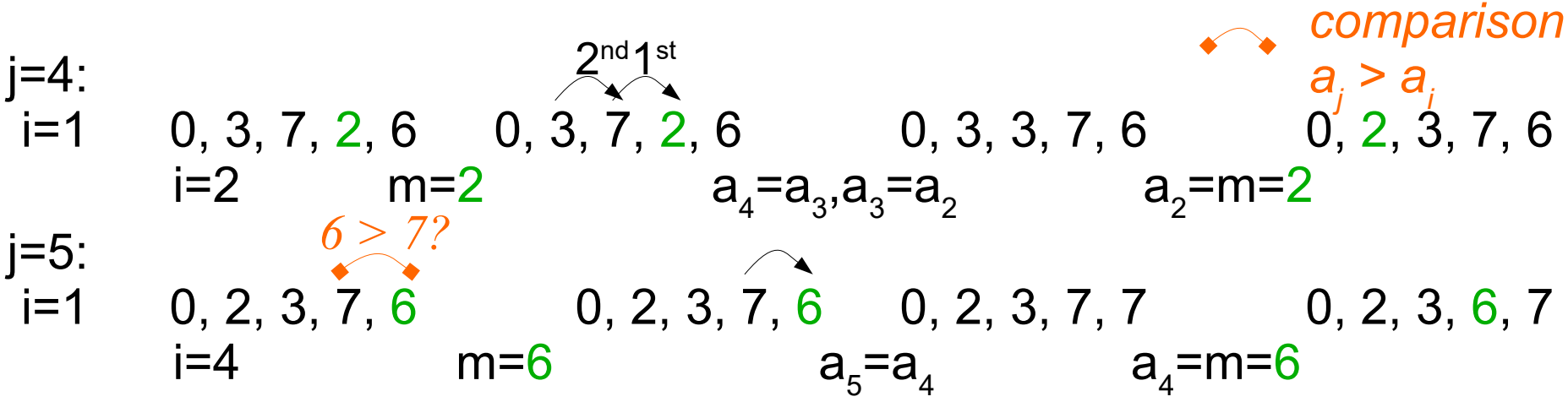
For *j := 2* **to** *n*
    i : = 1
    While ($a_j > a_i$)
        i := i+1
    End-while
    m := a_j
    For *k := 0* **to** *j-i-1*
        a_{j-k} := a_{j-k-1}
    End-for
    a_i := m
End-for

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



For *j := 2* **to** *n*
    i : = 1
    While ($a_j > a_i$)
        i := i+1
    End-while
    m := a_j
    For *k := 0* **to** *j-i-1*
        a_{j-k} := a_{j-k-1}
    End-for
    a_i := m
End-for

Example 3:
Let's see how insertion sort works on the list {7, 0, 3, 2, 6}



For *j := 2* **to** *n*
    i : = 1
    While ($a_j > a_i$)
        i := i+1
    End-while
    m := a_j
    For *k := 0* **to** *j-i-1*
        a_{j-k} := a_{j-k-1}
    End-for
    a_i := m
End-for

optimization problem – is a computational problem in which the goal is to find the "best" of all possible solutions.

"best" is different from problem to problem, for example:

- find a shortest route from city A to city B
- find a fastest route from city A to city B

optimization problem – is a computational problem in which the goal is to find the "best" of all possible solutions.

"best" is different from problem to problem, for example:
- find a shortest route from city A to city B
- find a fastest route from city A to city B

A simplest approach: *select the "best" choice at each step*

Greedy Algorithms

optimization problem – is a computational problem in which the goal is to find the "best" of all possible solutions.

"best" is different from problem to problem, for example:
- find a shortest route from city A to city B
- find a fastest route from city A to city B

A simplest approach: *select the "best" choice at each step*

Algorithms that make what seems to be the "best" choice at each step are called greedy algorithms.
- they often lead to a solution of optimization problem.

Greedy Algorithms

optimization problem – is a computational problem in which the goal is to find the "best" of all possible solutions.

"best" is different from problem to problem, for example:
- find a shortest route from city A to city B
- find a fastest route from city A to city B

A simplest approach: *select the "best" choice at each step*

Algorithms that make what seems to be the "best" choice at each step are called greedy algorithms.

- they often lead to a solution of optimization problem.

Once we know that a greedy algorithm finds a feasible solution, we need to determine whether it has found an *optimal solution*. To do this we:

- prove that the solution is optimal, or
- show that there is a counterexample where the algorithm yields a non-optimal solution.

Example 4:
Make *n* cents change with quarters (*q*), nickels (*c*), dimes (*d*), and pennies (*p*), using the least total number of coins.

Example 4:
Make *n* cents change with quarters (*q*), nickels (*c*), dimes (*d*), and pennies (*p*), using the least total number of coins.

A greedy algorithm:
Let's try to make the a *locally optimal choice* at each step: at each step we choose the coin of largest denomination possible to add to the pile of change without exceeding *n* cents.

Example 4:
Make *n* cents change with quarters (*q*), nickels (*c*), dimes (*d*), and pennies (*p*), using the least total number of coins.

A greedy algorithm:
Let's try to make the a *locally optimal choice* at each step: at each step we choose the coin of largest denomination possible to add to the pile of change without exceeding *n* cents.

procedure *change*(*n*: positive integer; c_1, c_2, c_3, ..., c_r : values of denominations of coins, where $c_1 > c_2 > c_3 > \ldots > c_r$

for *i* := *1* **to** r
    while $n \geq c_i$
        add a coin with value c_i to the change
        $n := n - c_i$
{the pile has change of *n* cents}

Example 4:
Make *n* cents change with quarters (*q*), nickels (*c*), dimes (*d*), and pennies (*p*), using the least total number of coins.

A greedy algorithm:
Let's try to make the a *locally optimal choice* at each step: at each step we choose the coin of largest denomination possible to add to the pile of change without exceeding *n* cents.

procedure *change*(*n*: positive integer; c_1, c_2, c_3, ..., c_r : values of denominations of coins, where $c_1 > c_2 > c_3 > \ldots > c_r$

for *i := 1* **to** r
    while $n \geq c_i$
        add a coin with value c_i to the change
        $n := n - c_i$
{the pile has change of *n* cents}

- presented algorithm *leads to an optimal solution* (*solves optimization problem*) in the sense that it uses the least number of coins.

Let's see how the presented algorithm works for *n=85*

procedure *change*(*n*: positive integer; c_1, c_2, c_3, ..., c_r : values of denominations of coins, where $c_1 > c_2 > c_3 > ... > c_r$

for *i := 1* **to** r

    while $n \geq c_i$

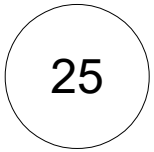
        add a coin with value c_i to the change

        $n := n-c_i$

{the pile has change of *n* cents}

Greedy Algorithms

Let's see how the presented algorithm works for *n=85*



Total: 25

procedure *change*(*n*: positive integer; c_1, c_2, c_3, ..., c_r : values of denominations of coins, where $c_1 > c_2 > c_3 > ... > c_r$

for *i := 1* **to** r

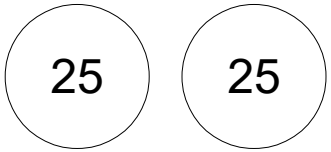
  while $n \geq c_i$

    add a coin with value c_i to the change

    $n := n-c_i$

{the pile has change of *n* cents}

Let's see how the presented algorithm works for *n=85*



Total: 25 50

procedure *change*(*n*: positive integer; c_1, c_2, c_3, ..., c_r : values of denominations of coins, where $c_1 > c_2 > c_3 > ... > c_r$

for *i := 1* **to** r

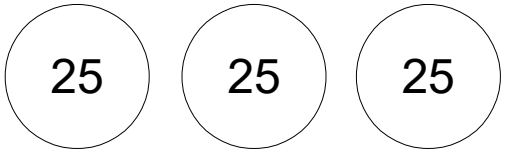
  while $n \geq c_i$

    add a coin with value c_i to the change

    $n := n-c_i$

{the pile has change of *n* cents}

Let's see how the presented algorithm works for *n=85*



Total: 25 50 75

procedure *change*(*n*: positive integer; c_1, c_2, c_3, ..., c_r : values of denominations of coins, where $c_1 > c_2 > c_3 > ... > c_r$

for *i := 1* **to** r

  while $n \geq c_i$

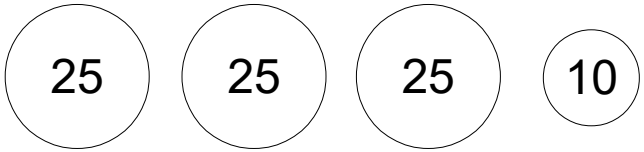
    add a coin with value c_i to the change

    $n := n-c_i$

{the pile has change of *n* cents}

Greedy Algorithms

Let's see how the presented algorithm works for *n=85*



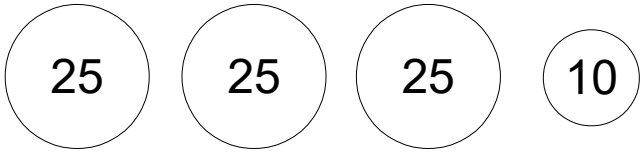
Total: 25 50 75 85

procedure *change*(*n*: positive integer; c_1, c_2, c_3, ..., c_r : values of denominations of coins, where $c_1 > c_2 > c_3 > ... > c_r$

for *i := 1* **to** r
    while $n \geq c_i$
        add a coin with value c_i to the change
        $n := n - c_i$
{the pile has change of *n* cents}

Greedy Algorithms

Let's see how the presented algorithm works for *n=85*



Total: 25 50 75 85

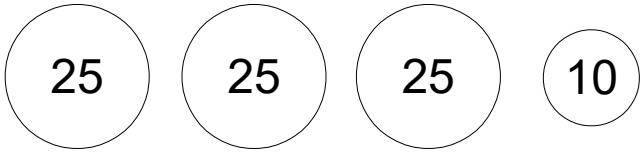
Let's see how the presented algorithm works for *n=98*

procedure *change*(*n*: positive integer; c_1, c_2, c_3, ..., c_r : values of denominations of coins, where $c_1 > c_2 > c_3 > ... > c_r$

for *i := 1* **to** r
    while $n \geq c_i$
        add a coin with value c_i to the change
        $n := n - c_i$
{the pile has change of *n* cents}

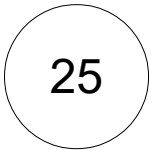
Greedy Algorithms

Let's see how the presented algorithm works for *n=85*



Total: 25 50 75 85

Let's see how the presented algorithm works for *n=98*



Total: 25

procedure *change*(*n*: positive integer; c_1, c_2, c_3, ..., c_r : values of denominations of coins, where $c_1 > c_2 > c_3 > ... > c_r$

for *i := 1* **to** r

  while $n \geq c_i$

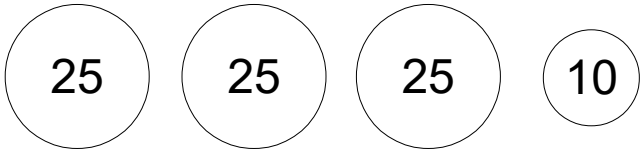
    add a coin with value c_i to the change

    $n := n - c_i$

{the pile has change of *n* cents}

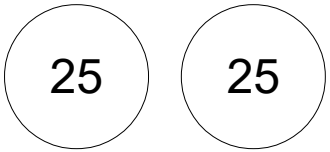
Greedy Algorithms

Let's see how the presented algorithm works for *n=85*



Total: 25 50 75 85

Let's see how the presented algorithm works for *n=98*



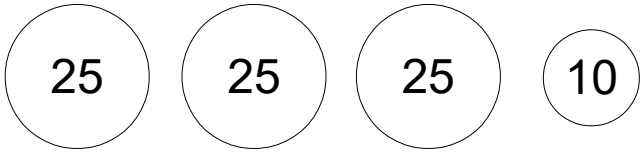
Total: 25 50

procedure *change*(*n*: positive integer; c_1, c_2, c_3, ..., c_r : values of denominations of coins, where $c_1 > c_2 > c_3 > \ldots > c_r$

for *i := 1* **to** r
    while $n \geq c_i$
        add a coin with value c_i to the change
        $n := n - c_i$
{the pile has change of *n* cents}

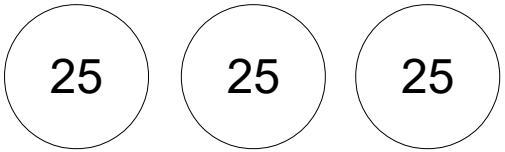
Greedy Algorithms

Let's see how the presented algorithm works for *n=85*



Total: 25 50 75 85

Let's see how the presented algorithm works for *n=98*



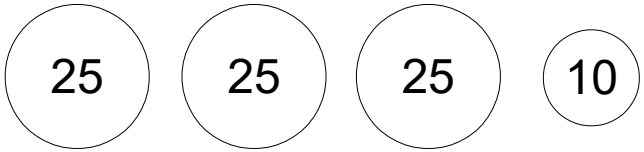
Total: 25 50 75

procedure *change*(*n*: positive integer; c_1, c_2, c_3, ..., c_r : values of denominations of coins, where $c_1 > c_2 > c_3 > \ldots > c_r$

for *i := 1* **to** r
  while $n \geq c_i$
    add a coin with value c_i to the change
    $n := n - c_i$
{the pile has change of *n* cents}

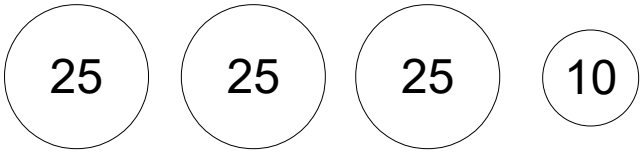
Greedy Algorithms

Let's see how the presented algorithm works for *n=85*



Total: 25 50 75 85

Let's see how the presented algorithm works for *n=98*



Total: 25 50 75 85

procedure *change*(*n*: positive integer; c_1, c_2, c_3, ..., c_r : values of denominations of coins, where $c_1 > c_2 > c_3 > \ldots > c_r$

for *i := 1* **to** r

  while $n \geq c_i$

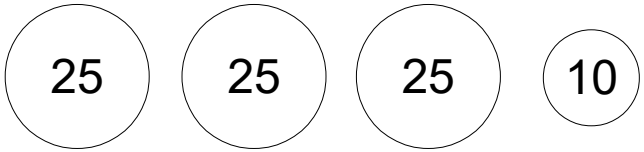
    add a coin with value c_i to the change

    $n := n - c_i$

{the pile has change of *n* cents}

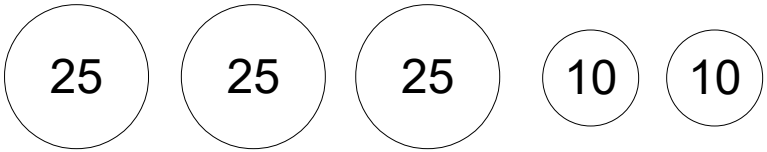
Greedy Algorithms

Let's see how the presented algorithm works for *n=85*



Total: 25 50 75 85

Let's see how the presented algorithm works for *n=98*



Total: 25 50 75 85 95

procedure *change*(*n*: positive integer; c_1, c_2, c_3, ..., c_r : values of denominations of coins, where $c_1 > c_2 > c_3 > \ldots > c_r$

for *i := 1* **to** r

  while $n \geq c_i$

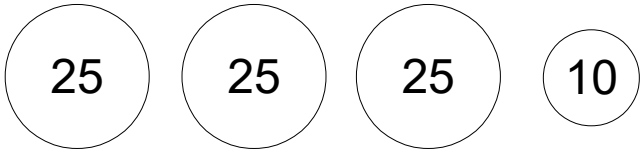
    add a coin with value c_i to the change

    $n := n - c_i$

{the pile has change of *n* cents}

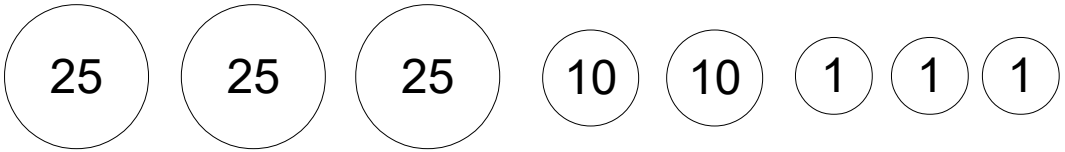
Greedy Algorithms

Let's see how the presented algorithm works for *n=85*



Total: 25 50 75 85

Let's see how the presented algorithm works for *n=98*



Total: 25 50 75 85 95 96 97 98

procedure *change*(*n*: positive integer; c_1, c_2, c_3, ..., c_r : values of denominations of coins, where $c_1 > c_2 > c_3 > \ldots > c_r$

for *i := 1* **to** r

  while $n \geq c_i$

    add a coin with value c_i to the change

    $n := n - c_i$

{the pile has change of *n* cents}

Greedy Algorithms

procedure *change*(*n*: positive integer; c_1, c_2, c_3, ..., c_r : values of denominations of coins, where $c_1 > c_2 > c_3 > ... > c_r$

for *i := 1* **to** r

  while $n \geq c_i$

    add a coin with value c_i to the change

    $n := n - c_i$

{the pile has change of *n* cents}

- presented algorithm *leads to an optimal solution* (*solves optimization problem*) in the sense that it uses the least number of coins.

It is not enough to present few examples to show that the algorithm leads to an optimal solution. We should present a *proof* (which we won't see in here, but if you are curious – see the book, pages 175-176) .

Greedy Algorithms

procedure *change*(*n*: positive integer; c_1, c_2, c_3, ..., c_r : values of denominations of coins, where $c_1 > c_2 > c_3 > \ldots > c_r$

for *i := 1* **to** r

  while $n \geq c_i$

    add a coin with value c_i to the change

    $n := n - c_i$

{the pile has change of *n* cents}

- presented algorithm *leads to an optimal solution* (*solves optimization problem*) in the sense that it uses the least number of coins.

It is not enough to present few examples to show that the algorithm leads to an optimal solution. We should present a *proof* (which we won't see in here, but if you are curious – see the book, pages 175-176) .

! There are sets of coins (for example, quarters, dimes and pennies) for which the presented greedy algorithm doesn't produce change using the fewest coins possible.