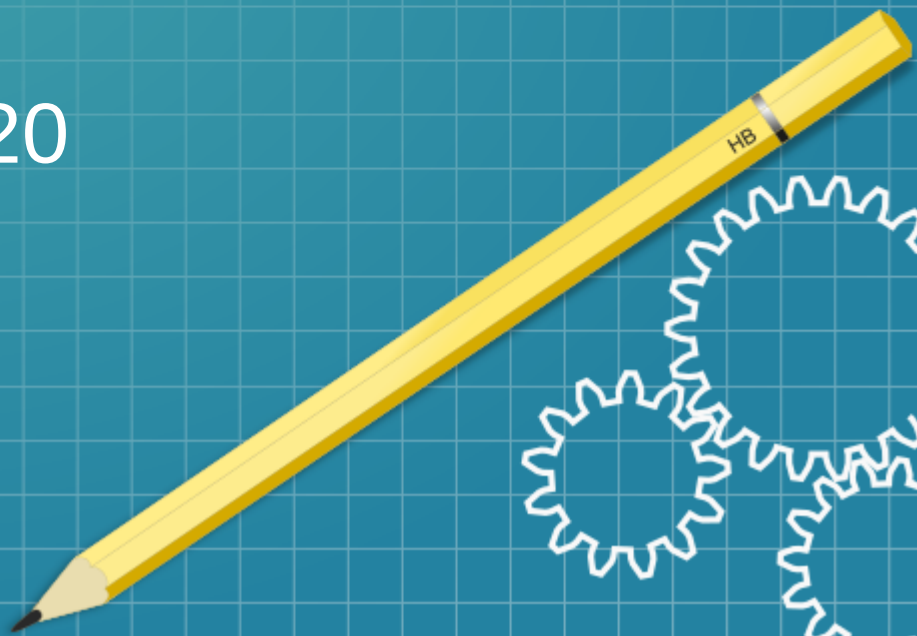
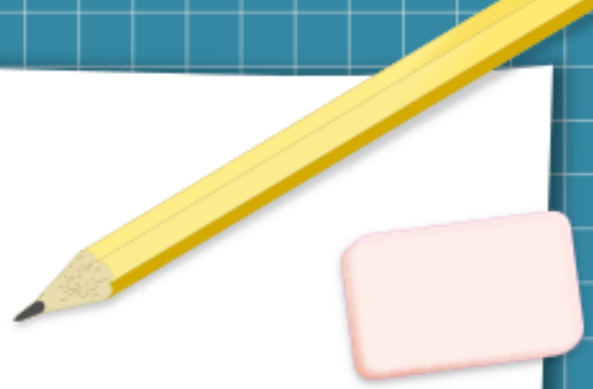
Searching and Sorting

Chapter 20



Today we will discuss



- Searching for a given value in an array:
  - using *linear search*
  - using *binary search*
- Sorting a given array:
  - using *bubble sort*
  - using *selection sort*
- Runtime complexity

Searching

Searching a collection of values involves determining whether a value, *search key*, is present in the data or not.

If a *search key* is present in the data, we can:

- return `True`, or
- return its *location*

Two popular searching algorithms are:

- *linear search*
  - slow, but does not require the values to be ordered
- *binary search*
  - faster, but all the elements must be sorted in increasing order

Linear Search

Given a sequence of elements, the algorithm starts with the very first element to check if it is the search key, then compares the second element to the search key, and so forth.

As soon as the search key is found, the algorithm terminates.

If the search key is not present in the sequence, the algorithm checks all the elements in the sequence.

Linear Search: pseudocode

```
def search(items, searchKey):
   i = 0
   while i < items.size():
      if items[i] == searchKey:
         return i
      i += 1
   return -1
```

Let's write a template function for linear search:

see linearSearch.cpp

You can also take a look at linearSearch2.cpp after the class that works with an array of strings

Linear Search: runtime complexity

In the *worst-case scenario*, if the *search key* is not present in the sequential collection, the algorithm will compare each element to the search key, i.e. we will have *n* comparisons (in an *n*-element sequential collection).

In this case, we say that the algorithm has a *linear running time*, `O(n)`.

Other pronunciations : *“order of n”*

Linear Search: runtime complexity

In the *worst-case scenario*, if the *search key* is not present in the sequential collection, the algorithm will compare each element to the search key, i.e. we will have *n* comparisons (in an *n*-element sequential collection).

In this case, we say that the algorithm has a *linear running time*, `O(n)`.

Other pronunciations : *"order of n"*

Even if we had *n-2* or *n-10* comparisons, as *n* grows larger, the *n* will "dominate" (if n = 10385, then n-10 = 10375, not a big difference). So the running time complexity will be still `O(n)`.

Linear Search: runtime complexity

In the *worst-case scenario*, if the *search key* is not present in the sequential collection, the algorithm will compare each element to the search key, i.e. we will have *n* comparisons (in an *n*-element sequential collection).

In this case, we say that the algorithm has a *linear running time*, `O(n)`.

Other pronunciations : *"order of n"*

Even if we had *n-2* or *n-10* comparisons, as *n* grows larger, the *n* will "dominate" (if n = 10385, then n-10 = 10375, not a big difference). So the running time complexity will be still `O(n)`.

`O` represents the *upper bound of the growth*.

Binary Search

Given an ordered sequence of elements, from smallest to largest, the binary search is a more efficient search algorithm.

It starts with the middle of the sequential collection, comparing it with the search key:

- If the middle element matches the search key, its *location* is returned.
- If not, the half of the sequential collection is selected, by comparing the search key to the middle element, and the process is repeated: the middle element is compared to the search key, and so forth.

If the size of a split leads to a sub-collection of size 0, the algorithm stops and -1 is returned for *location*.

Binary Search: pseudocode

```
def search(items, target):
   low = 0
   high = len(items) - 1
   while low <= high:

      middle = (low + high + 1) / 2

      if target == items[mid] :
         return middle
      else if target < item:
         high = middle - 1
      else:
         low = middle + 1
   return -1
```

Let's write a template function for the binary search:
see binarySearch.cpp

Binary Search: runtime complexity

At every iteration of the while loop, the array is “halved”.

This leads to *logarithmic time complexity*, i.e. `O(log n)`.

You will discuss it in more details in CSI 33.

Sorting

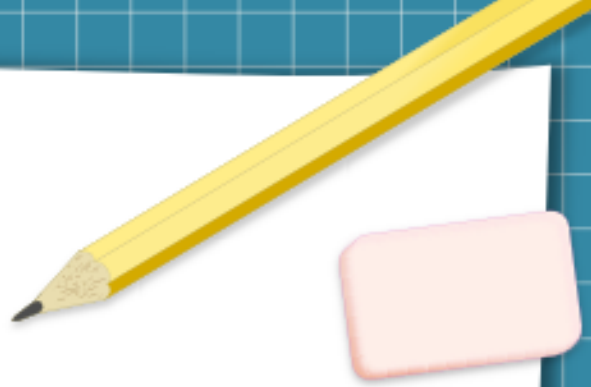
Ordering the elements of a list is a problem that occurs in many contexts.

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}

Sorting



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

procedure *bubblesort*($a_1,...,a_n$)
For *i := 1* **to** *n-1*
    F**or** *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*

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

    End-for

End-for

Description: 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,
- If not, do nothing.

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 perform n-1 passes.

Bubble sort

Example: 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

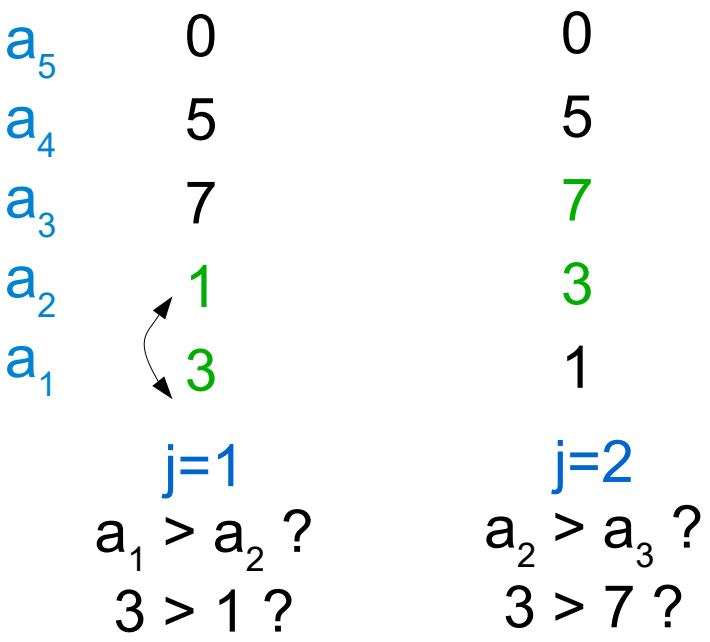
j=1
$a_1 > a_2$?
3 > 1 ?

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

First pass (i=1):

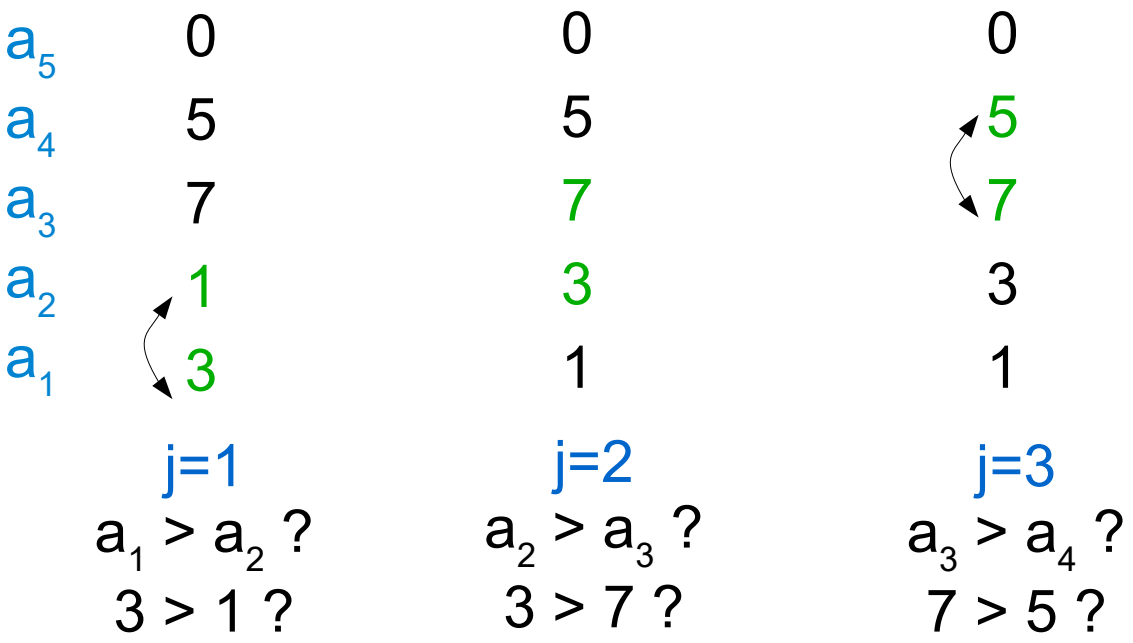


For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

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

First pass (i=1):

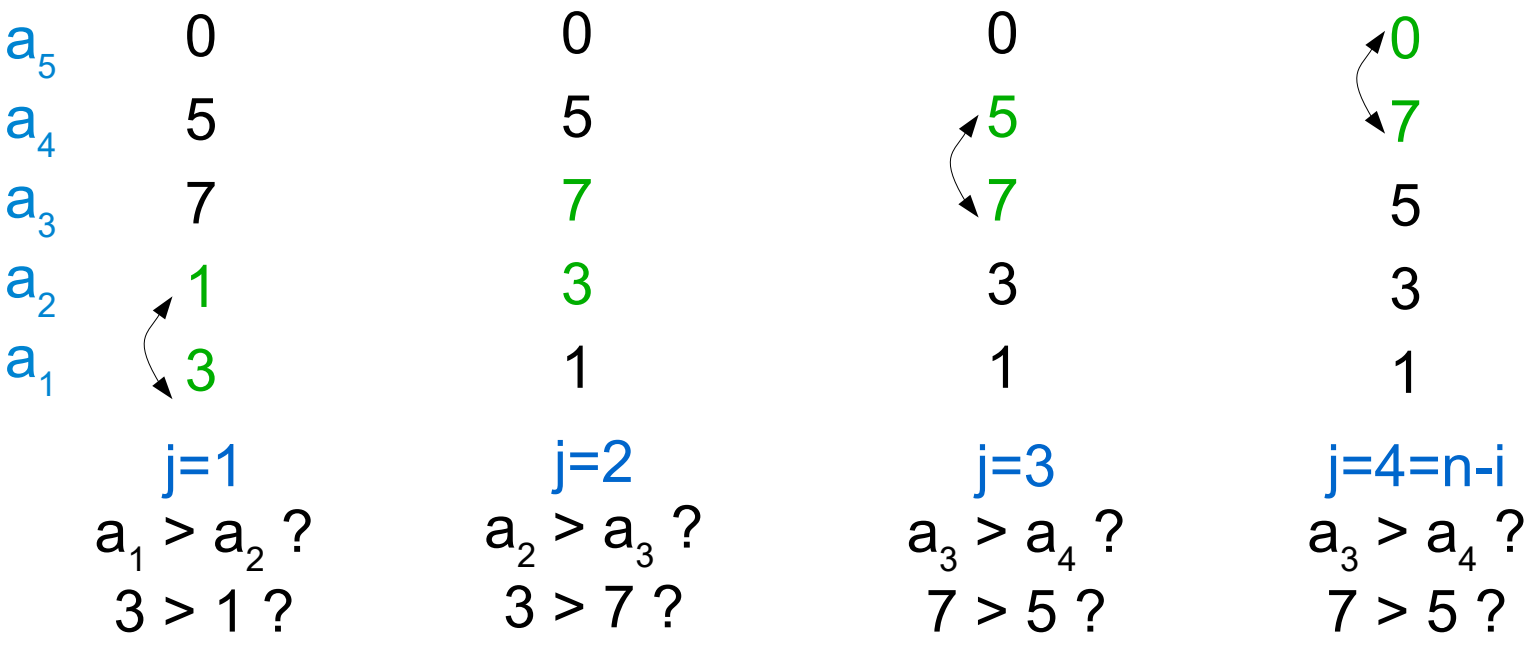


For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

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

First pass (i=1):

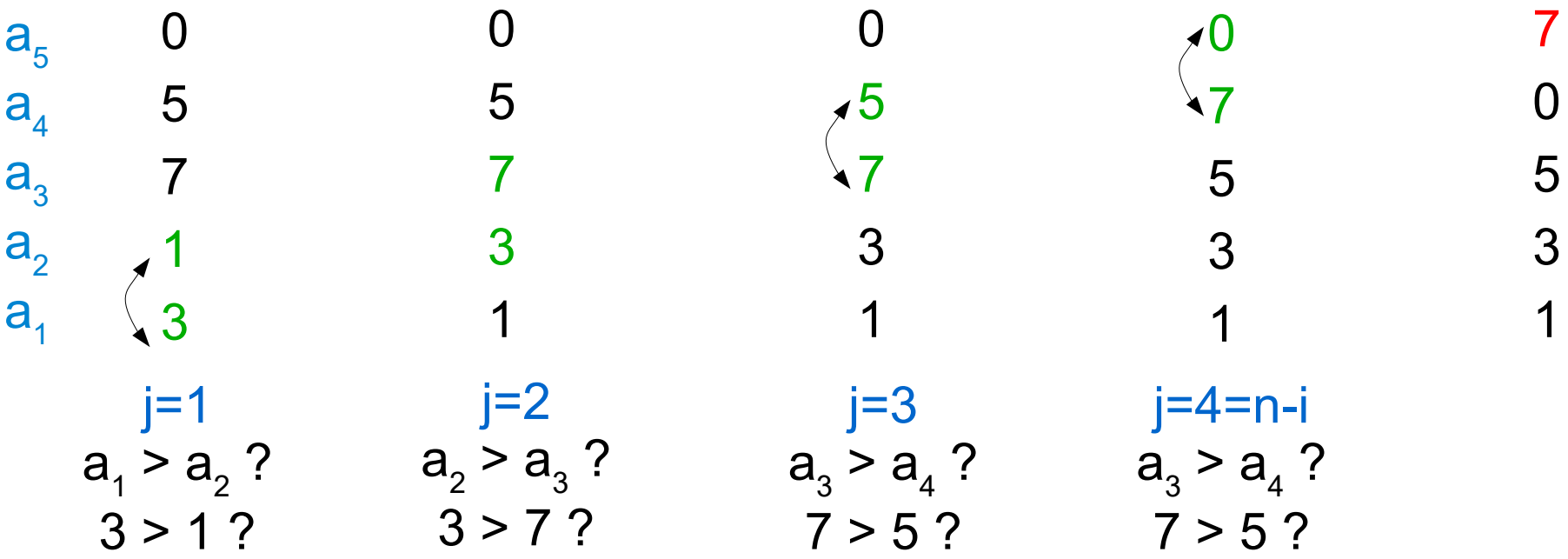


For *i := 1* **to** *n-1*
  For *j := 1* **to** *n-i*
    If ($a_j > a_{j+1}$) **,** interchange a_j and a_{j+1}
  End-for
End-for

Bubble sort

Example: 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: 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: 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: 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: 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	7
a_4	0	0	0	5
a_3	5	5	5	0
a_2	3	3	3	3
a_1	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: 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*

  F**or** *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: 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: 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: 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*
  F**or** *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: 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*

  F**or** *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: 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*

  F**or** *j := 1* **to** *n-i*

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

  End-for

End-for

Bubble Sort: runtime complexity

procedure *bubblesort*($a_1,\ldots,a_n$)

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

    F**or** *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?

Bubble Sort: runtime complexity

procedure *bubblesort*($a_1,\ldots,a_n$)

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)

Bubble Sort: runtime complexity

procedure *bubblesort*($a_1,\ldots,a_n$)

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

  F**or** *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)) =

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

Bubble Sort: runtime complexity

procedure *bubblesort*($a_1,\ldots,a_n$)

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

  F**or** *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 +

i=1 i=2 i=3 i=4 i=n-1 3 + 4 + … (n-1))=

Bubble Sort: runtime complexity

procedure *bubblesort*($a_1,\ldots,a_n$)

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

  F**or** *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:

$(n-1) + (n-2) + (n-3) + (n-4) + \ldots + (n-(n-1)) = (n-1)*n - (1 + 2 + 3 + 4 + \ldots (n-1)) =$

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

$= n^2 - n - (1+(n-1))(n-1)/2$

Bubble Sort: runtime complexity

procedure *bubblesort*($a_1,\ldots,a_n$)
For *i := 1* **to** *n-1*
    F**or** *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 +
i=1 i=2 i=3 i=4 i=n-1 3 + 4 + … (n-1))=
= n^2 – n – (1+(n-1))(n-1)/2 = … = $n^2/2$ – n/2 - quadratic

Bubble Sort: runtime complexity

procedure *bubblesort*($a_1,\ldots,a_n$)
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

$O(n^2)$

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 +
i=1 i=2 i=3 i=4 i=n-1 3 + 4 + … (n-1))=
= n^2 – n – (1+(n-1))(n-1)/2 = … = $n^2/2$ – n/2 - quadratic

Selection Sort

Selection sort is another sorting algorithm (also slow).

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 i := 1 to n-1
   min := i
   For j: = i+1 to n
      If (aj < amin), min := j
   End-for
   temp := ai
   ai := amin
   amin := temp
End-for
```

Selection Sort

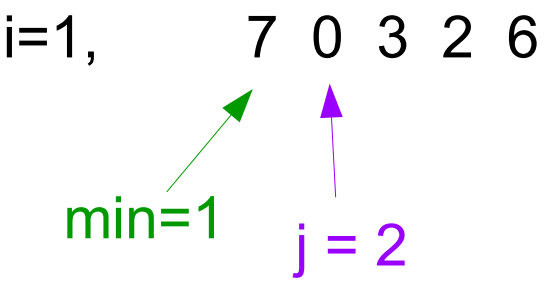
```
procedure insertion sort(a1,...,an)
For i := 1 to n-1
   min := i
   For j: = i+1 to n
      If (aj < amin), min := j
   End-for
   temp := ai
   ai := amin
   amin := temp
End-for
```

Description: 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.

Selection Sort Demonstration

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



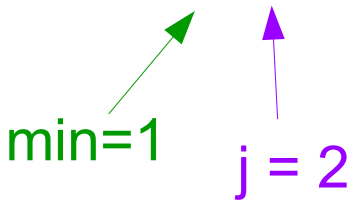
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

i=1, 7 0 3 2 6



0 < 7 ?

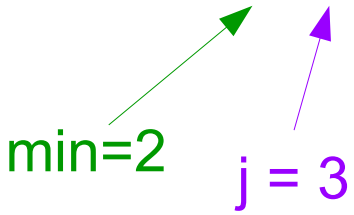
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

i=1, 7 0 3 2 6



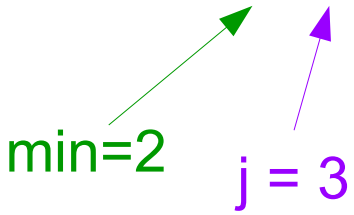
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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 ?

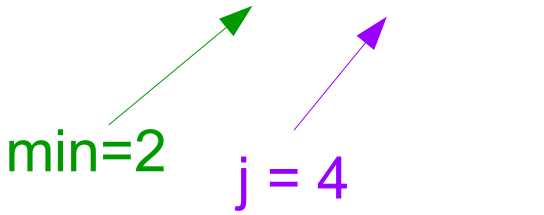
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

i=1, 7 0 3 2 6



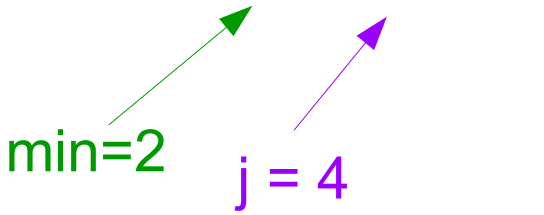
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

i=1, 7 0 3 2 6



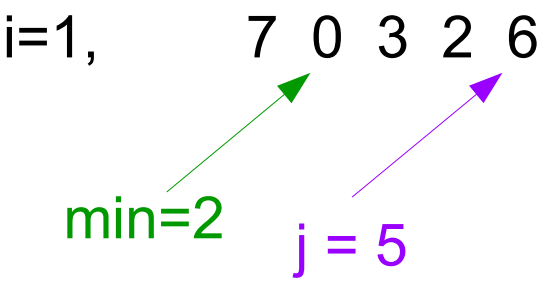
2 < 0 ?

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

For i := 1 to n-1
    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

Selection Sort Demonstration

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



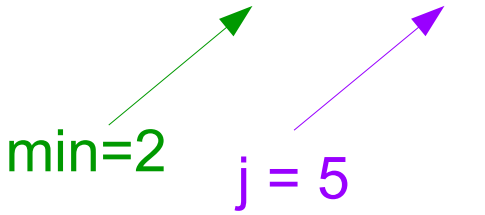
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

i=1, 7 0 3 2 6



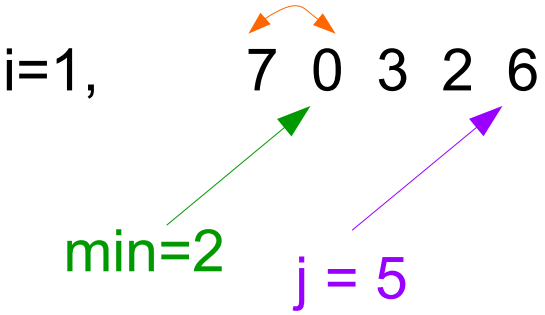
6 < 0 ?

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

For i := 1 to n-1
    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

Selection Sort Demonstration

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

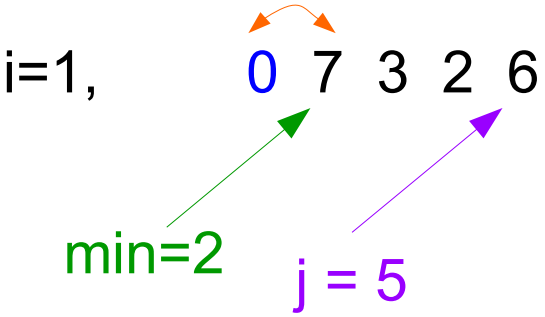


procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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



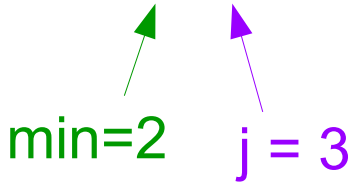
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

i=2, 0 7 3 2 6



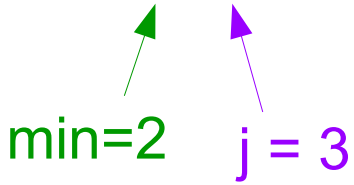
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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 ?

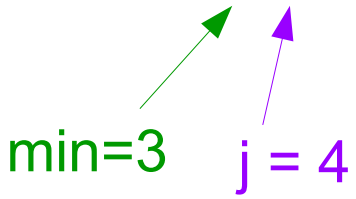
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

i=2, 0 7 3 2 6



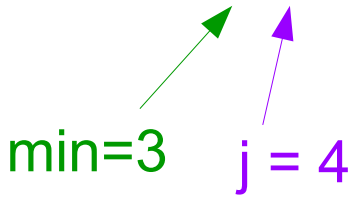
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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 ?

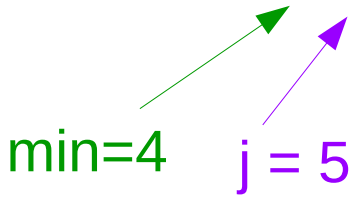
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

i=2, 0 7 3 2 6



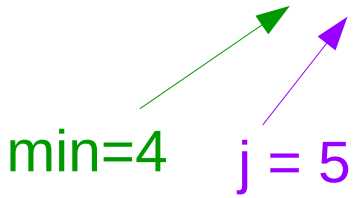
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

i=2, 0 7 3 2 6



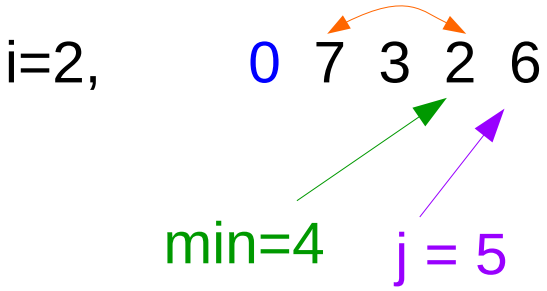
6 < 2 ?

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

For i := 1 to n-1
    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

Selection Sort Demonstration

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



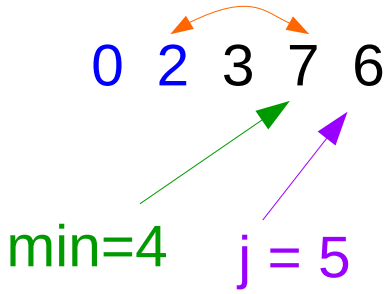
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

i=2,



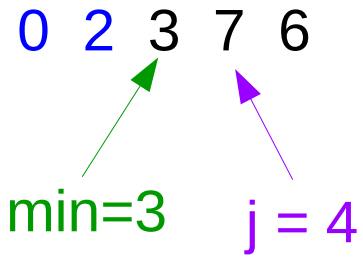
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

i=3,



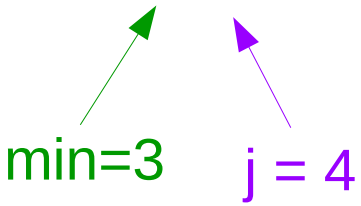
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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 ?

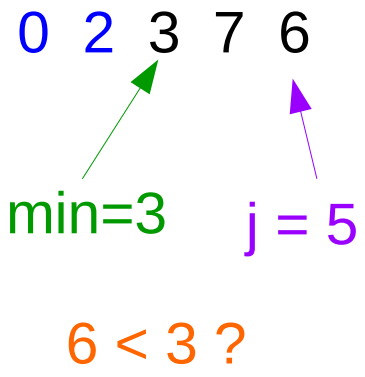
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

i=3,



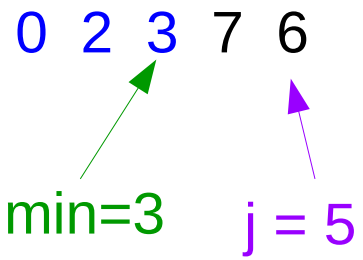
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

i=3,



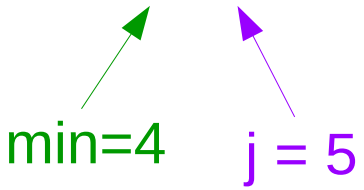
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

i=4, 0 2 3 7 6



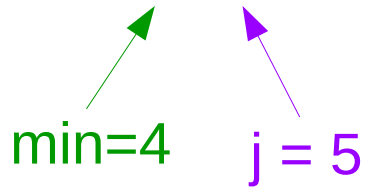
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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 ?

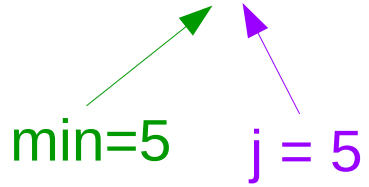
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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 ?

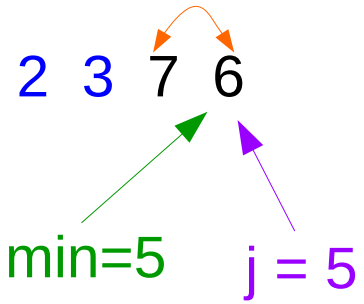
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

i=4,



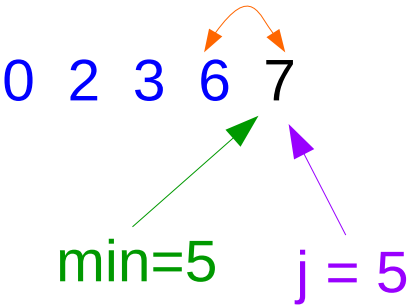
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

i=4,



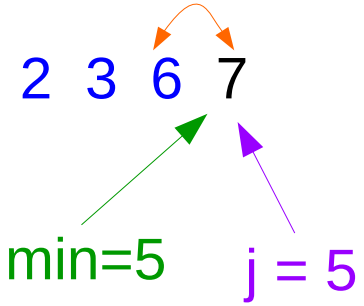
procedure *insertion sort*(a_1,...,a_n)

For i := 1 to n-1
    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

Selection Sort Demonstration

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

I=4, 0 2 3 6 7 Sorted! 0 2 3 6 7



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

For i := 1 to n-1
    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

Selection Sort: runtime complexity

```
procedure insertion sort(a1,...,an)
For i := 1 to n-1
   min := i
   For j: = i+1 to n
      If (aj < amin), min := j
   End-for
   temp := ai
   ai := amin
   amin := temp
End-for
```

{7, 0, 3, 2, 6}

Runtime complexity: $O(n^2)$

see selectionSort.cpp

Suggested Practice

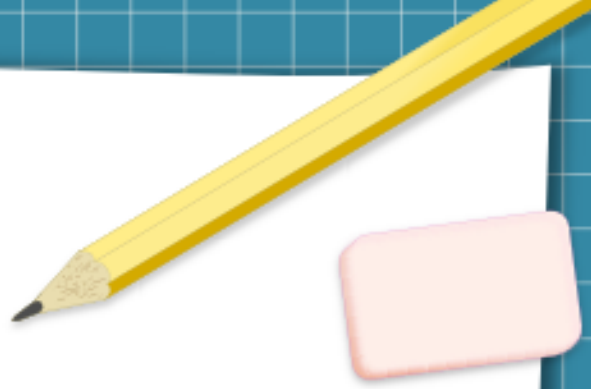
(1) Show the work of *binary search* when searching for value 27 in the following array of integers: {-23, 5, 10, 19, 23, 27, 45, 67, 129}

(2) Show the work of the *Bubble sort* for the following array of integer values: { 5, 1, 9, 3, 8, 0}

(3) Show the work of the *Selection sort* for the following array of integer values: { 5, 1, 9, 3, 8, 0}

(4) Exercise 20.6 (b): improvement in Bubble sort

(5) Exercise 20.9 : recursive Binary Search



Self-Study:
Section 20.3.1 Insertion sort

Chapter 20 Summary and Self-Review Exercises

Suggested Reading:
Section 20.3.3 Merge Sort