COMP20005 Workshop Week 8

Preparation:

- open `grok` for exercises
- Programming: open & use `VS/gcc` if possible

1	Discussion 1: Searching
2	Discussion 2: Sorting & loop invariant Ex 7.2, 7.6, 7.3
LAB	• Assignment: Q&A • **Work on Assignment**, AND/OR (if ass1 totally done): • implement array exercises
	Looking Ahead: • Assignment 1 due this Friday, at **6pm Melbourne time** [note: the testing machine might be over-loaded on Friday afternoon] • Quiz 2: 4:15pm on Wednesday 4 May (Melbourne time)

Searching

General task: given an array of n elements A[n], find a specific element.
Example:

1. Find element that equal x, ie. A[i]==x
2. Find maximal element
3. Find the most frequent element

Output can be a value as in case 2 and 3, but it's better and more general to be an index, ie. the above task are

1. Return index i such that A[i]==x (or return -1 if NOTFOUND)
2. Return index of the maximal element

Searching

Two different type of searching:

1. Return index i such that A[i]==x (or return -1 if NOTFOUND): *we might not need to check the whole array, and there is a chance for NOTFOUND*

```
#define NOTFOUND -1

// return an index or NOTFOUND
// if there are many i such that A[i]=x, rerturn the smallest one
int search(int A[], int n) {
  for (i=0; i<n; i++) {
    // process A[i], ie. do something with A[i]
  }
  return ???;
}
```

2. Return index of the maximal element: *solution always found, and we always need to check the whole array.*

```
// return index of a max value in A[]
//   on tie, return the smallest index
int find_imax(int A[], int i) {
  ???
  for (i=0; i<n; i++) {
    // process A[i] , ie. do somethimng with A[i]
  }
  // return ???;
}
```

Searching

Two different type of searching:

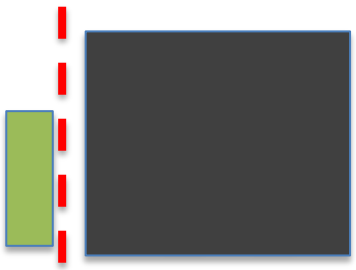
1. Return index i such that A[i]==x (or return -1 if NOTFOUND): *we might not need to check the whole array, and there is a chance for NOTFOUND*

```
int search(int A[], int n) {
  for (i=0; i<n; i++) {
    if ( A[i]==x ) {    // if A[i] is a looking-for
      return I;
    }
  }
  return NOTFOUND;
}
```

2. Return index of the maximal element: *solution always found, and we always need to check the whole array.*

```
int find_imax(int A[], int i) {
  int i, imax= 0;   // supposing A[0] is the answer
  // imax is the best solution so far
  for (i=1; i<n; i++) {
    if ( A[i]> A[imax] ) {    // if found a better solution
      imax= i;                //   then update imax
    }
  }
  return imax;
}
```

Insertion Sort: understanding



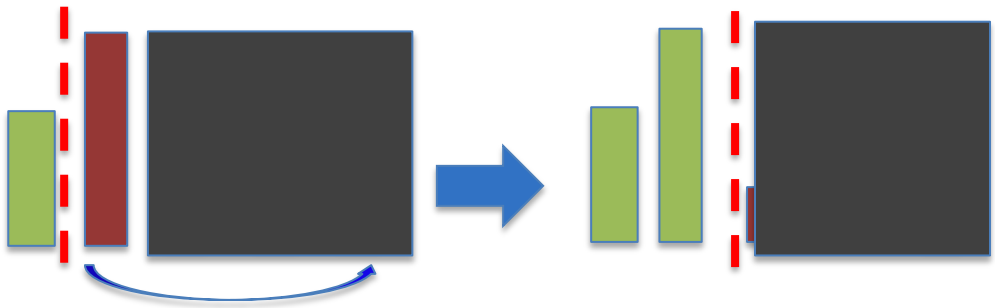
In this algorithm, we process element-by-element, *keeping* ***all processed elements*** *in sorted order*.

At first we note that:

- Nothing needs to do when processing at `A[0]`

Next step= ?

Insertion Sort: understanding



Then: when processing element `A[1]`

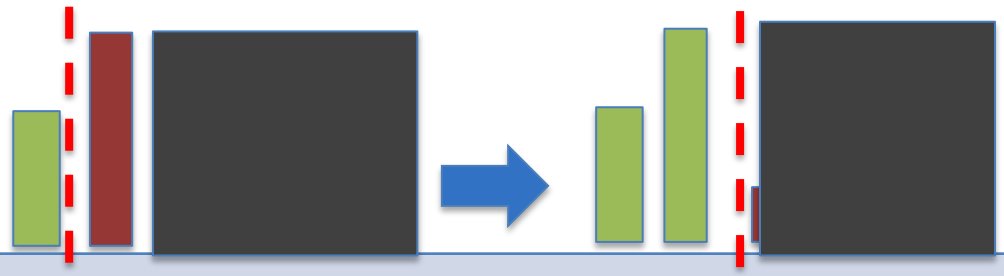
- Think of **the left** of `A[1]` as an array `A[0..0]`
- Note that that left array is sorted
- Now, insert `A[1]` to the left (and hence expand the left) so that the new left array `A[0..1]` is sorted

Next step= ?

Insertion Sort: understanding (n=5)

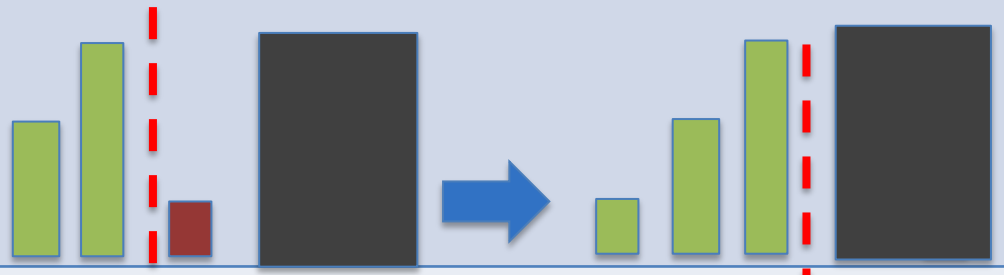
Round 1: process `A[1]`

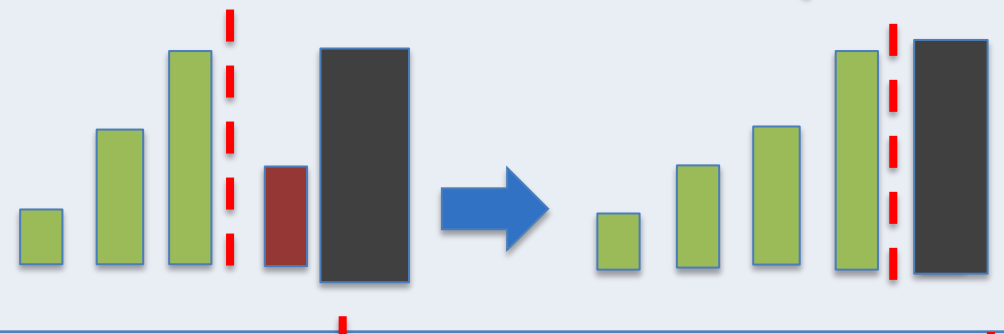
- `A[0..0]` *is sorted*
- *insert* `A[1]` *to the left so that* `A[0..1]` *is sorted*



Round 2: process `A[2]`

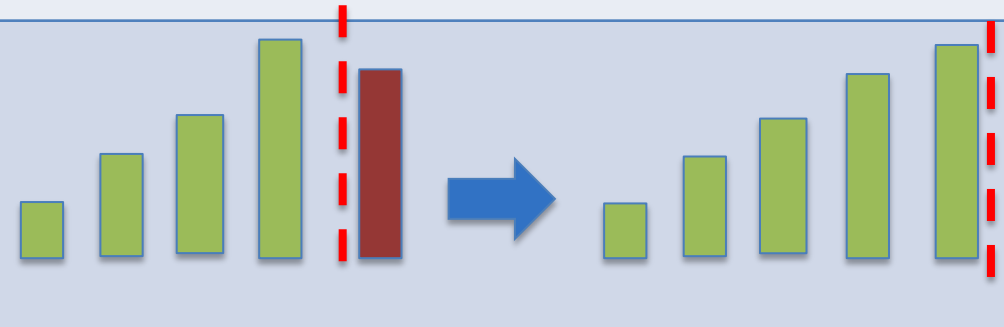
- `A[0..1]` *is sorted*
- *insert* `A[2]` *to the left so that* `A[0..2]` *is sorted*





Round n-1: process `A[n-1]`

- `A[0..4]` *is sorted*
- *insert* `A[4]` *to the left so that* `A[0..4]` *is sorted*



Insertion Sort: main loop

Round i=1: consider `A[1]`

- `A[0..0]` *is sorted*
- *insert* `A[1]` *to the left so that* `A[0..1]` *is sorted*

Round

-
-

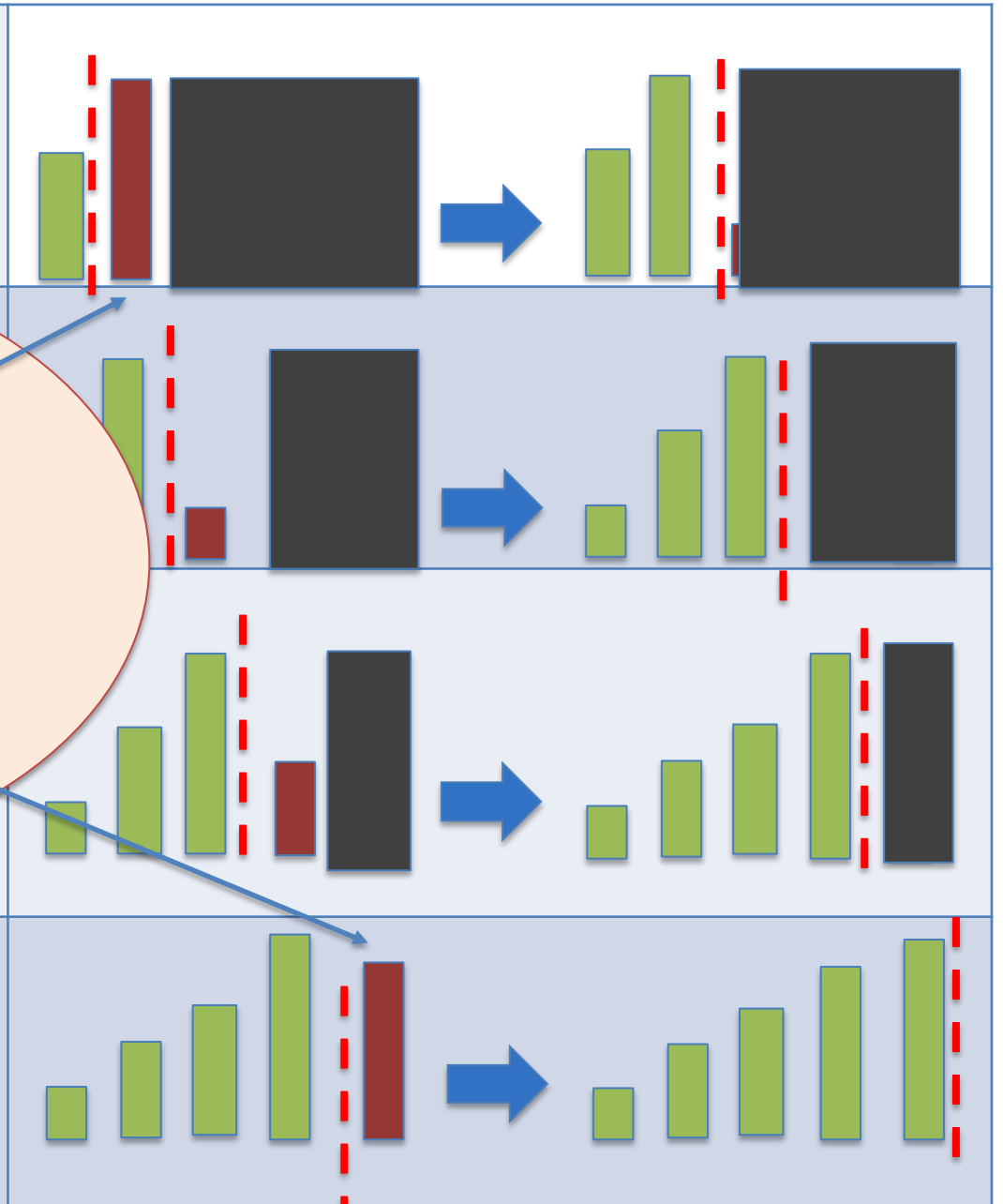
So we make a loop:

```
for (i=1;i<n; i++)
```

last value `i= n-1`

Round i=4: consider `A[4]`

- `A[0..4]` *is sorted*
- *insert* `A[i]` *to the left so that* `A[0..4]` *is sorted*



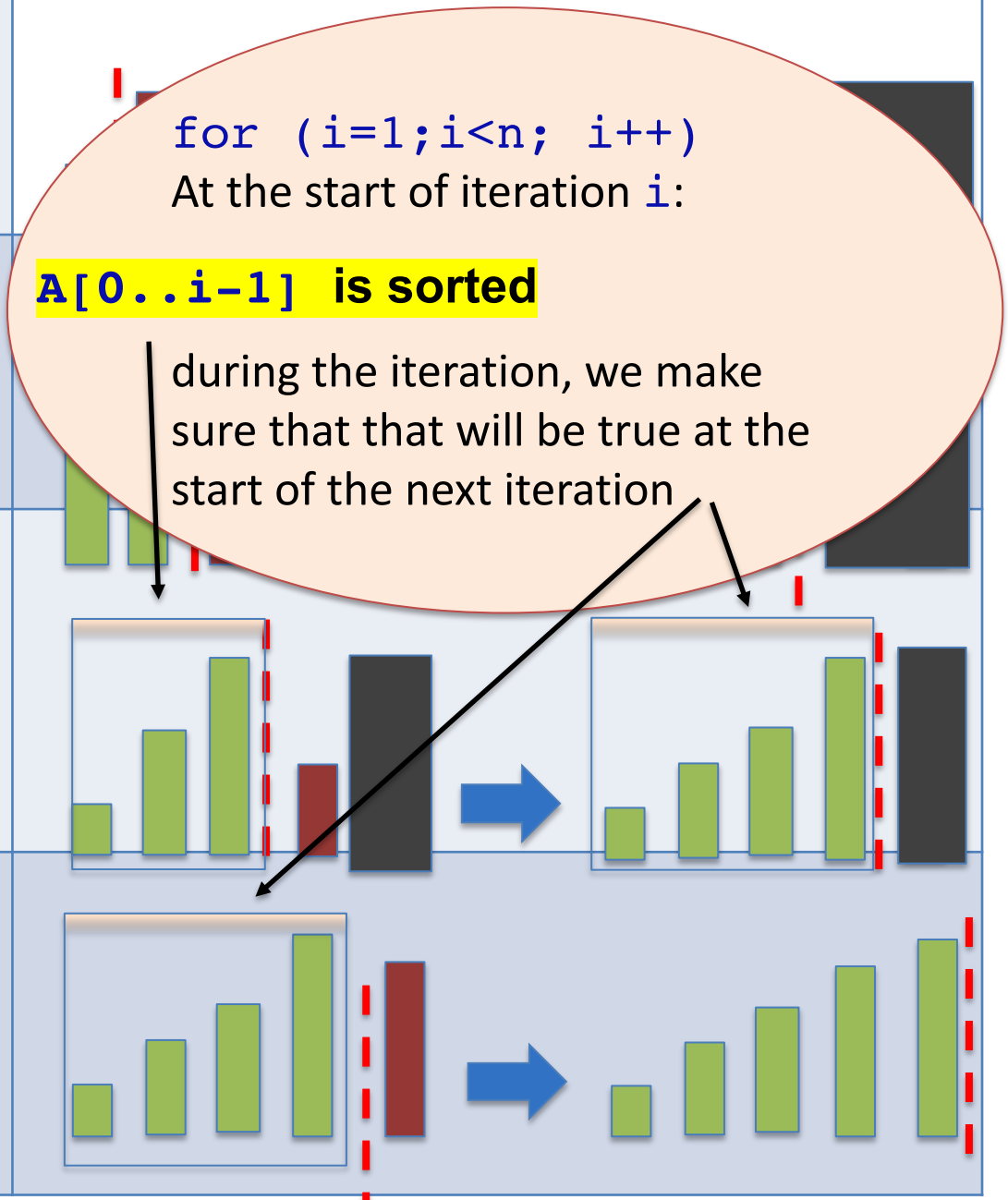
Insertion Sort & loop invariant

Round i=3: process `A[i]`

- `A[0..i-1]` *is sorted*
- *insert* `A[2]` *to the left so that* `A[0..i]` *is sorted*

Note:

- *loop invariant is also true at the start and end of the loop*
- *understanding loop invariant help us to write robust loops*

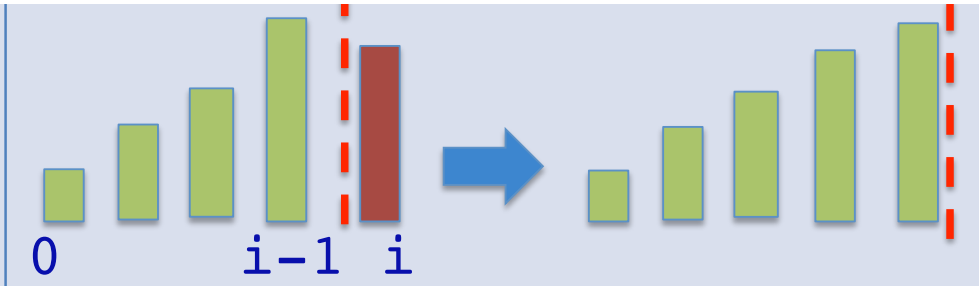


Insertion Sort: Algorithm

Process `A[1]` in round `1`, ..., `A[n-1]` in round `n-1`

Round i: consider `A[i]`

- `A[0..i-1]` *is sorted*
- *insert* `A[i]` *to the left so that* `A[0..i]` *is sorted*



How to: insert `A[i]` to the left?

```
void ins_sort(int A[], int n) {
  int i,j;
  for (i=1; i<n; i++) {
    // loop invariant: A[0..i-1] is sorted
    // now insert A[i] to A[0..i-1] to update i in the loop invariant
    ??
  }
}
```

Ex 7.2

Modify so that the array of values is sorted into decreasing order.

```
    // insertion sort:  sorting elements A[0] to A[n-1] min ascending order

1   for (i= 1; i<n ; i++) {
2     /* swap A[i] left into correct position */
3     for (j= i-1; j>=0 && A[j+1]<A[j]; j--) {
4        /* not there yet */
5        int_swap( &A[j] , &A[j+1] );
6     }
7   }
```

Selection Sort: Ex 7.6

An alternative sorting algorithm is selection sort. It goes like this: scan the array to determine the location of the largest element, and swap it into the last position. Then repeat the process, concentrating at each stage on the elements that have not yet been swapped into their final position.
Write a function to sort an integer array using selection sort.

Selection Sort: understanding

Selection sort: scan the array to determine the location of the largest element, and swap it into the last position. Then repeat the process …

So at first:

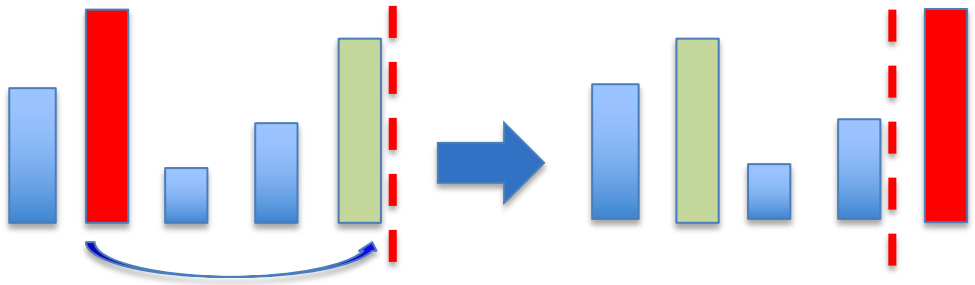
- *scan all un-sorted elements from position* `0` *to position* `n-1` *to determine the position of the largest element,*
- *swap it into the last position, ie. position* `n-1`*.*

What next?

Ex 7.6: Write A Function

?: consider `A[0..n-1]`

- *determine position of the largest*
- *swap with the last, ie.* `A[n-1]`, *then* `...`



```
// sort array A in ascending order using selection sort
void sel_sort(int A[], int n) {

}
```

Ex 7.03

7.03: Modify the insertion sort program so that after the array has been sorted, only the distinct values are retained in the array with variable n suitably reduced.

How about:
Modify the *selection* sort program so that after the array has been sorted, only the distinct values are retained in the array with variable n suitably reduced.

Approach:
???

```
// ???
? my_new_function( ? ) {
}
```

Ex 7.03

7.03: Modify the insertion sort program so that after the array has been sorted, only the distinct values are retained in the array with variable n suitably reduced.

Approach: write an additional function that can be applied to any sorted array.

```
//  removes duplicated element in a sorted array
//  returns updated number of elements
int remove_duplicate(int A[], int n) {
  // example: input  A= {1,2,2,2,3,3,9}, n=7
  //          output A= {1,2,3,9}  & return 4
}
```

What do we want to do in the loop? What's the loop invariant?

Looking Ahead, A1 Q&A, LAB

Assignment 1 :

- due at 6pm this Friday
- make sure to test your code and
- read the *verification report* carefully, make sure to see
  - NO compilation message,
  - NO sign **+** or **-** at the start of any line
- final LMS submit Thursday night, enjoy your Friday afternoon.

Quiz 2: 4:15pm on Wednesday 4 May (Melbourne time). See Practice Quiz..

Before next workshop: do sample quiz as many times as possible, take note and bring questions to the workshop.

Programming is fun: Check each of the following exercises, make sure you know how to do it, and implement if not yet done and time permitted.

- ARRAYS: *Arrays, and array manipulation, are very important! 15 exercises in C07, for examples:*
  - Searching: Exercises 7.7, 7.8, 7.15
  - Sorting: 7.5, 7.4 , 7.2, 7.3, 7.6

Assignment1 Q&A - LAB time

Any question on assignment 1?

If ass1 completely done, implement exercises from:

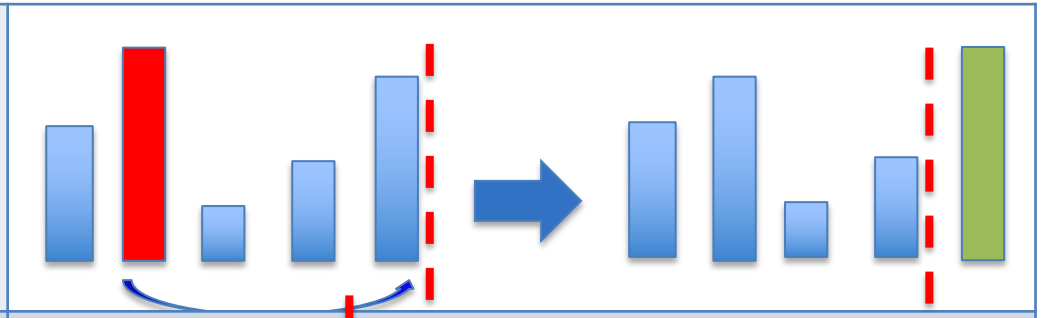
- ARRAYS:
  - Searching: Exercises 7.7, 7.8, 7.9,, 7.10, 7.5
  - Sorting: 7.4 , 7.2, 7.3, 7.6
  - Others: typedef & 2D array – ex 7.11

Additional Materials for self-study

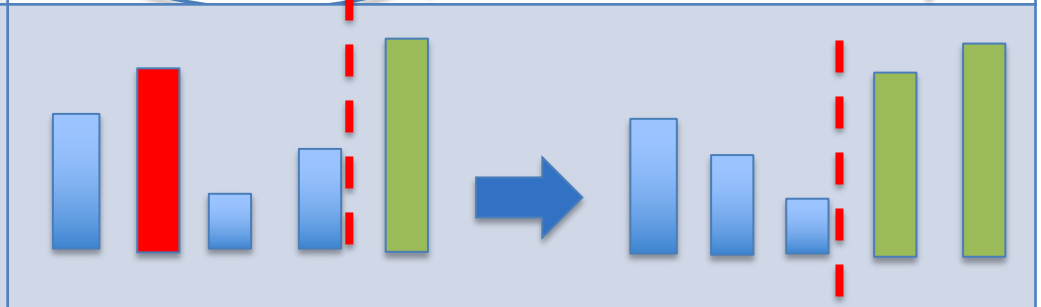
See next pages

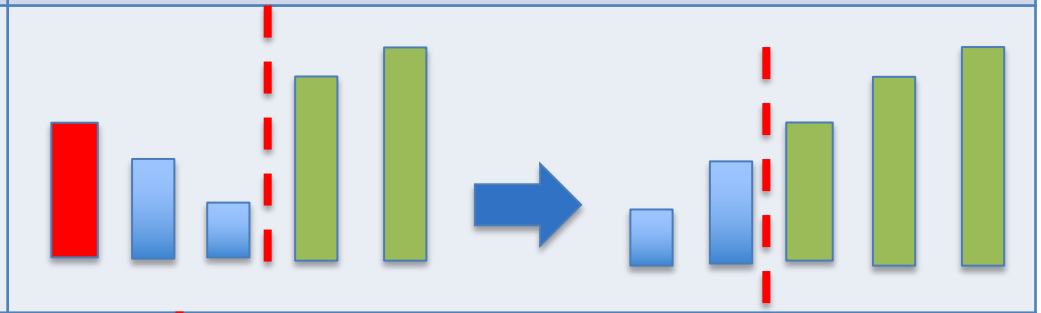
Selection Sort: understanding (n=5)

Round 1: consider `A[0..4]`
- *determine position of the largest*
- *swap with the last, ie.* `A[4]`

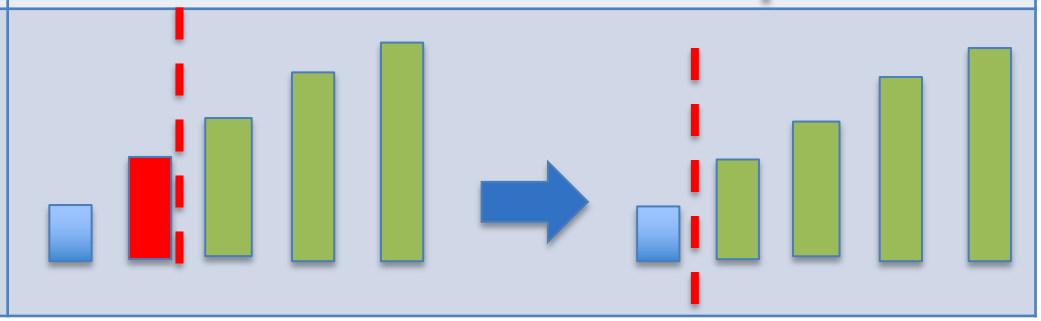


Round 2: consider `A[0..3]`
- *determine position of the largest*
- *swap with the last, ie.* `A[3]`





Round 4: consider `A[0..1]`
- *determine position of the largest*
- *swap with the last, ie.* `A[1]`

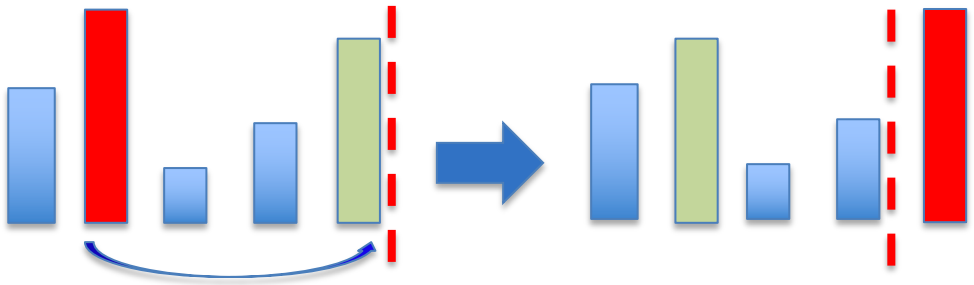


Ex 7.6: Write recursive function

First round: examining `A[0..n-1]`, last round: examining `A[0..1]`

Round ?: consider `A[0..i]`

- *determine position of the largest*
- *swap with the last, ie.* `A[i]`



```
void sel_sort(int A[], int n) {




}
```