04-630
Data Structures and Algorithms for Engineers

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Lecture 5

Searching and Sorting Algorithms

- Linear Search & Binary Search

- In-place sorts
  - Bubble Sort
  - Selection Sort
  - Insertion Sort

- Not-in-place sort
  - Quicksort
  - Mergesort

- Characteristics of a good sort

Linear (Sequential) Search

Linear (Sequential) Search

- Begin at the beginning of the list
- Proceed through the list, sequentially and element by element,
- Until the key is encountered
  or
  Until the end of the list is reached

Linear (Sequential) Search

- Note: we treat a list as a general concept, decoupled from its implementation
- The order of complexity is $O(n)$
- The list does not have to be in sorted order

Implementation of linear search in C

```
int linear_search(item_type s[], item_type key, int low, int high) {

  int i;

   i = low;

   while ((s[i] != key) && (i < high)) {
        i = i+1;
   }

   if (s[i] == key) {
        return (i);
   }
   else {
        return(-1);
   }
}
```

Binary Search

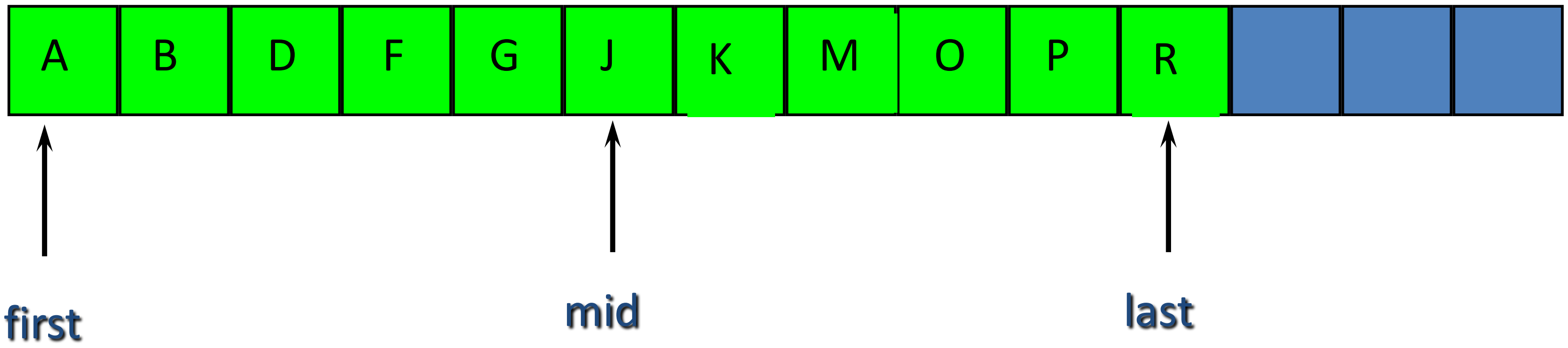
- If the list is sorted, we can use a more efficient $\mathrm{O}(\log_2(n))$ search strategy

- Check to see whether the key is

  – equal to
  – less than
  – greater than

  the middle element

Binary Search



Binary Search

- If key is equal to the middle element,
  then terminate (found)

- If key is less than the middle element,
  then search the left half

- If key is greater than the middle element, then search the right half

- Continue until either
  - the key is found or
  - there are no more elements to search

Implementation of Binary_Search

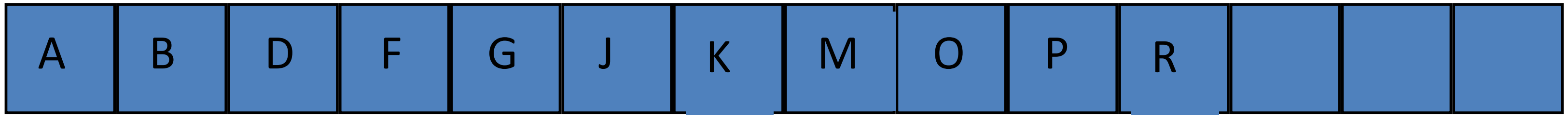
Pseudo-code

```
binary_search(list, key, lower_bound, upper_bound)

identify sublist to be searched by setting bounds on search

REPEAT
   get middle element of list
   if middle element < key
      then reset bounds to make the right sublist
           the list to be searched
      else reset bounds to make the left sublist
           the list to be searched
UNTIL list is empty or key is found
```

Implementation of binary search in C (iterative approach)

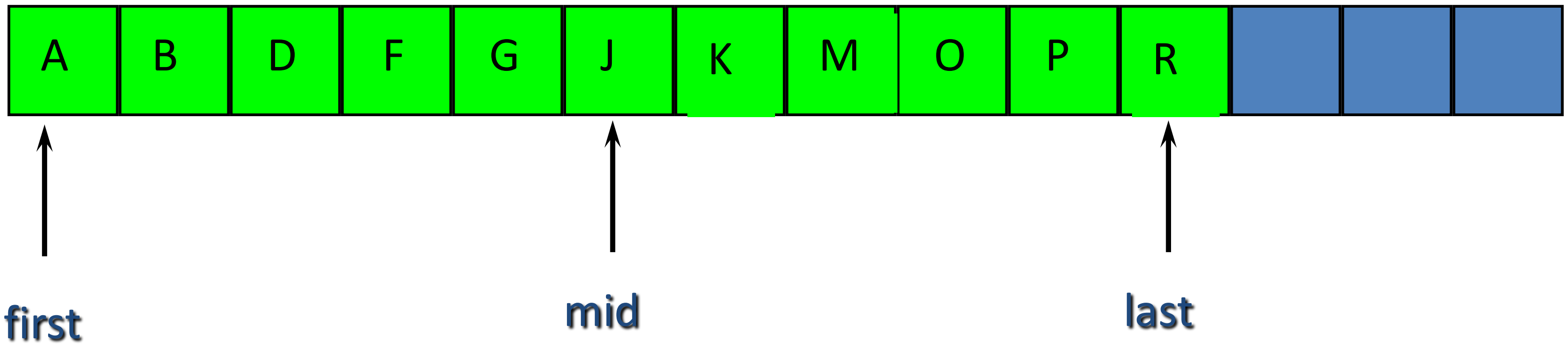
```
typedef char item_type;

int binary_search(item_type s[], item_type key, int low, int high) {

   int first, last, mid;

    first = low;
    last  = high;

    do {
        mid = (first + last) / 2;
        if (s[mid] < key) {
            first = mid + 1;
        }
        else {
            last = mid - 1;
        }
    } while ( (first <= last) && (s[mid] != key) );

    if (s[mid] == key)
        return (mid);
    else
        return (-1);
}
```

Binary Search



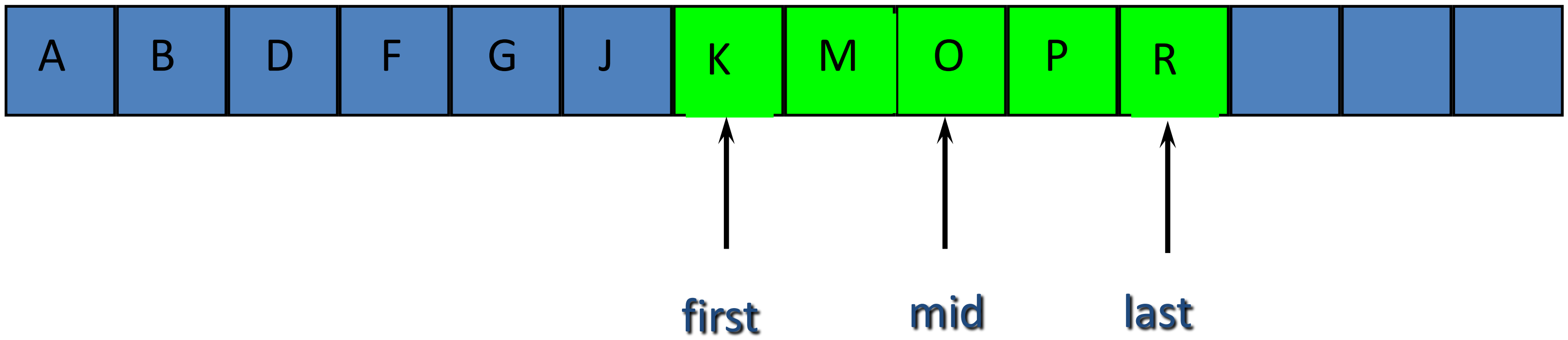

```
first:
last:
mid:
list[mid]:
key:         P
```

Binary Search



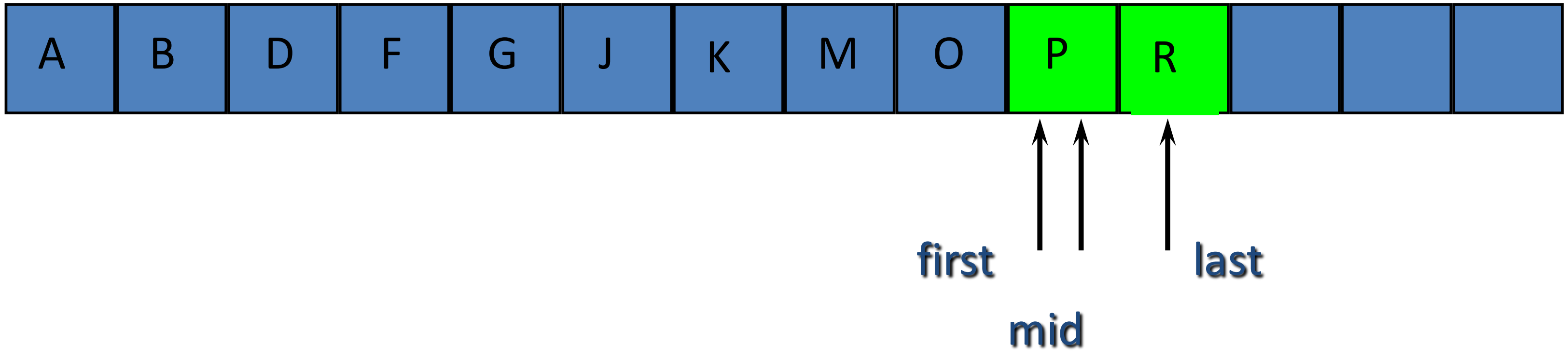

```
first:      1
last:      11
mid:        6
list[mid]:  J
key:        P
```

Binary Search



first:	1	7
last:	11	11
mid:	6	9
list[mid]:	J	O
key:	P	P

Binary Search

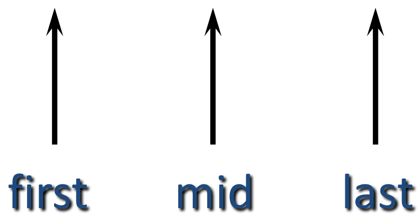



```
first:       1   7   10
last:       11  11   11
mid:         6   9   10
list[mid]:  J   O    P
key:         P   P    P
```

← FOUND!

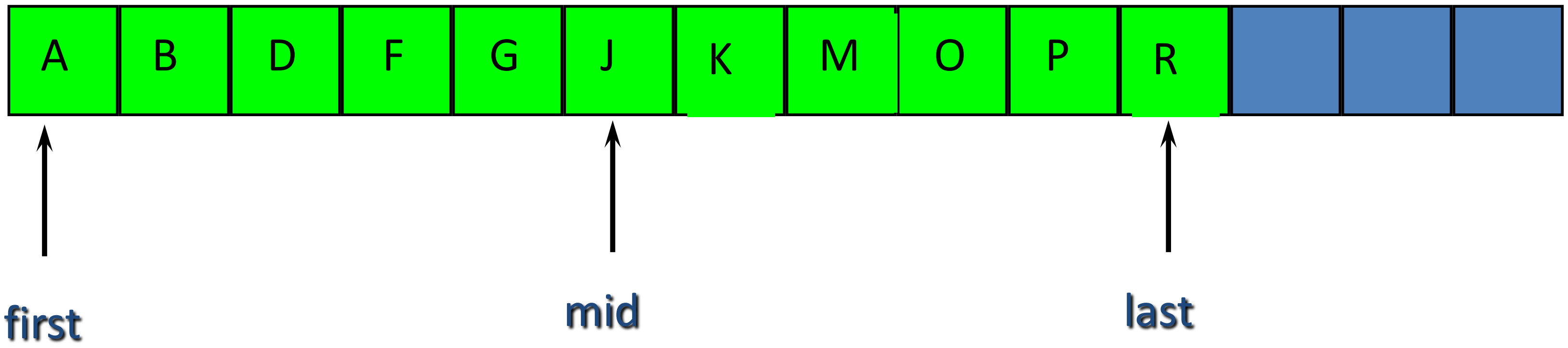
Binary Search

A	B	D	F	G	J	K	M	O	P	R			



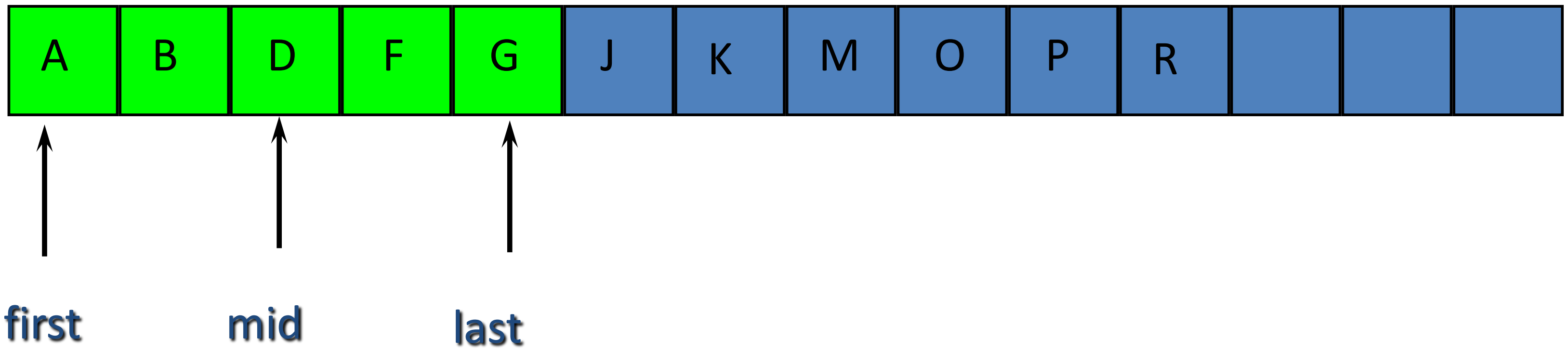

```
first:
last:
mid:
list[mid]:
key:        E
```

Binary Search



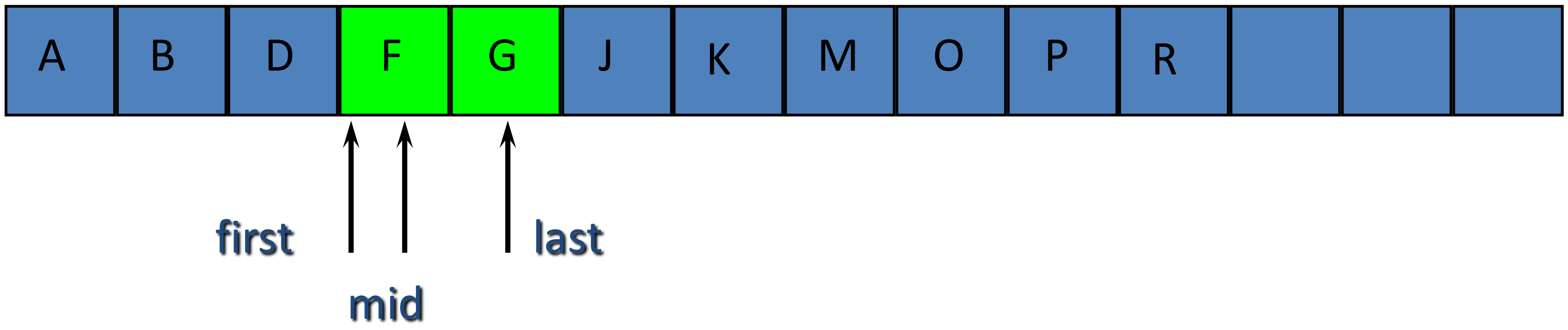

```
first:      1
last:      11
mid:        6
list[mid]: J
key:        E
```

Binary Search



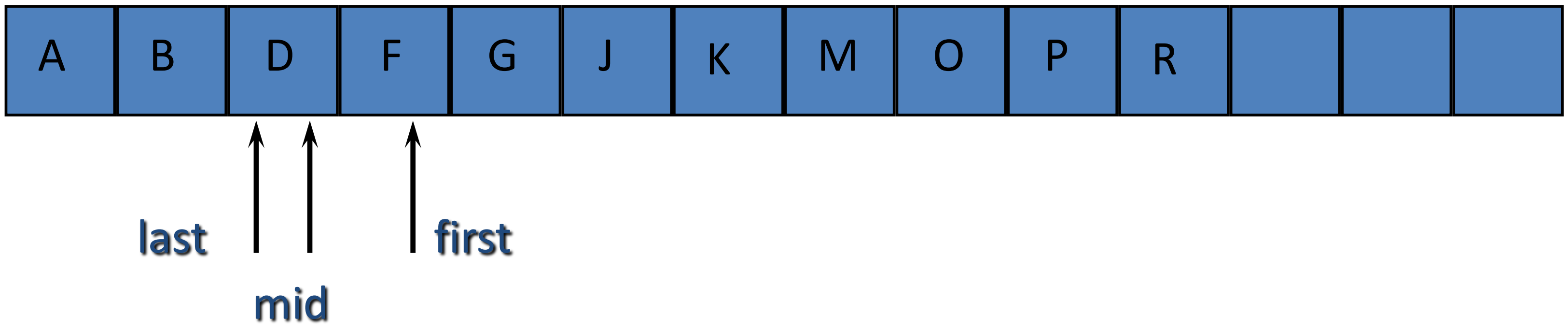
first:	1	1
last:	11	5
mid:	6	3
list[mid]:	J	D
key:	E	E

Binary Search




```
first:      1    1    4
last:      11    5    5
mid:        6    3    4
list[mid]: J    D    F
key:        E    E    E
```

Binary Search



first:	1	1	4	4
last:	11	5	5	3
mid:	6	3	4	3
list[mid]:	J	D	F	D
key:	E	E	E	E

← first > last: NOT FOUND!

Implementation of binary search in C (recursive approach)

```
typedef char item_type;

int binary_search(item_type s[], item_type key, int low, int high) {

   int mid;

    if (low > high)  return (-1); /* key not found */

    mid = (low + high) / 2;

    if (s[mid] == key) return(mid);

    if (s[mid] > key) {
        return(binary_search(s, key, low, mid-1));
    }
    else {
       return(binary_search(s, key, mid+1, high));
    }
}
```

Sorting Algorithms

The Sorting Problem

Input: A sequence of *n* numbers $< a_1, a_2, \ldots a_n >$

Output: the permutation (reordering) of the input sequence such that $a_1 \leq a_2 \leq \ldots \leq a_n$

Sorting Algorithms

- In-place sorts
  - Small number of elements stored outside the input data structure
  - Additional space requirements $O(1)$
  - Tradeoff: more computationally-complex algorithms (slower sorts)
    - Bubble Sort
    - Selection Sort
    - Insertion Sort

Sorting Algorithms

- Not-in-place sort
  - Additional space requirements not $O(1)$
  - Tradeoff: less computationally-complex algorithms but greater memory requirements (possibly unpredictable)
    - Quick Sort
    - Merge Sort
- Characteristics of a good sort

Bubble Sort

- Assume we are sorting a list represented by an array A of n integer elements

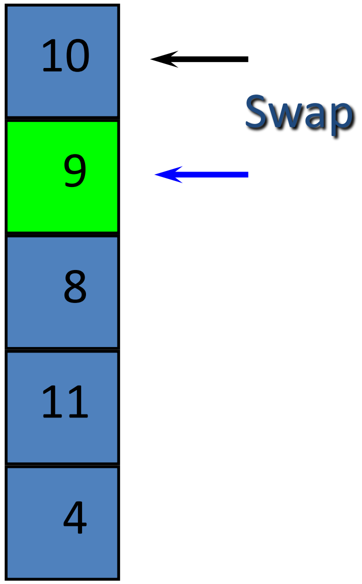
- Bubble sort algorithm in pseudo-code

```
FOR every element in the list,
    proceeding from the first to the last

   WHILE list element > previous list element
      bubble element back (up) the list
      by successive swapping with
      the element just above/prior it
```

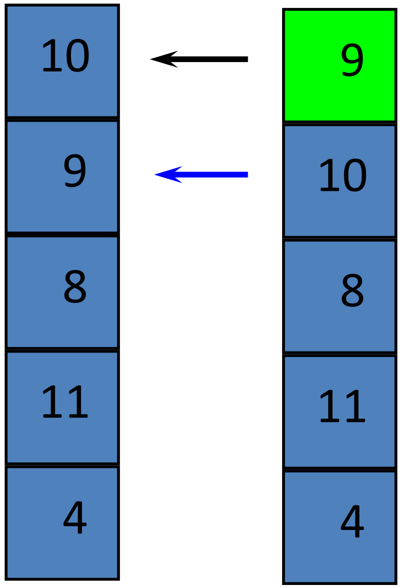
Bubble Sort

10	9	8	11	4

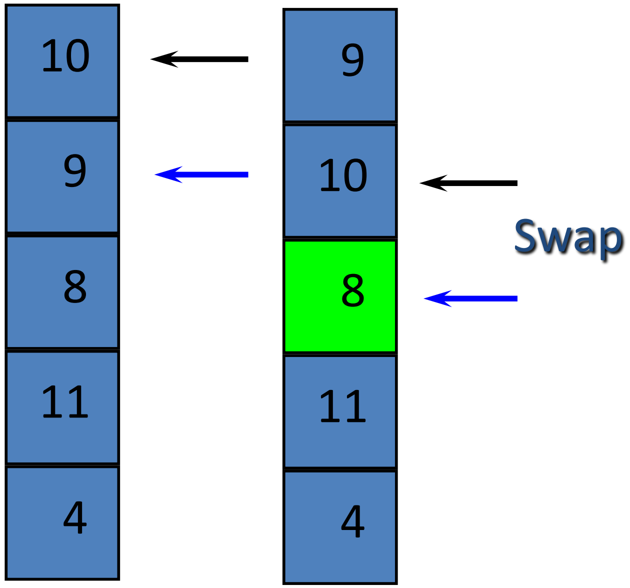
Bubble Sort



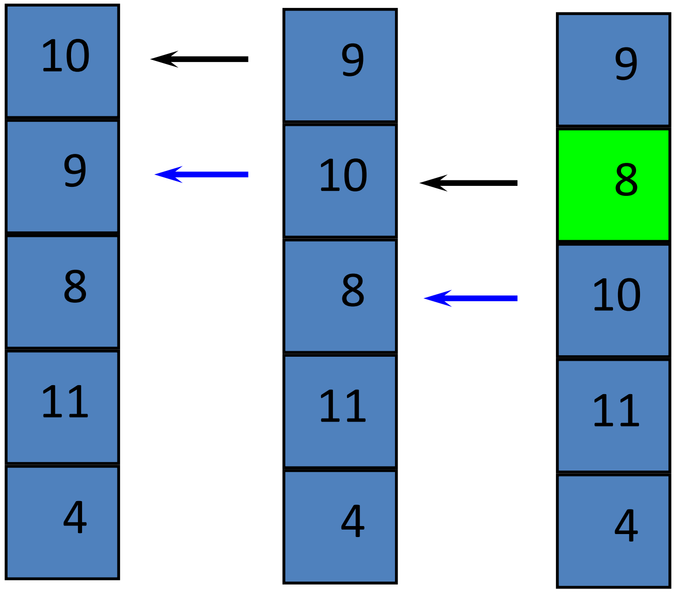
Bubble Sort



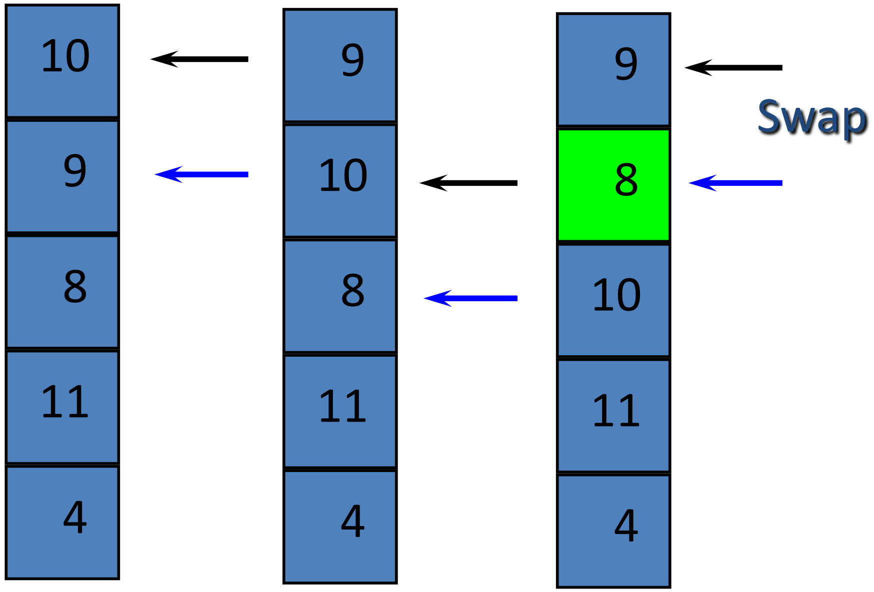
Bubble Sort



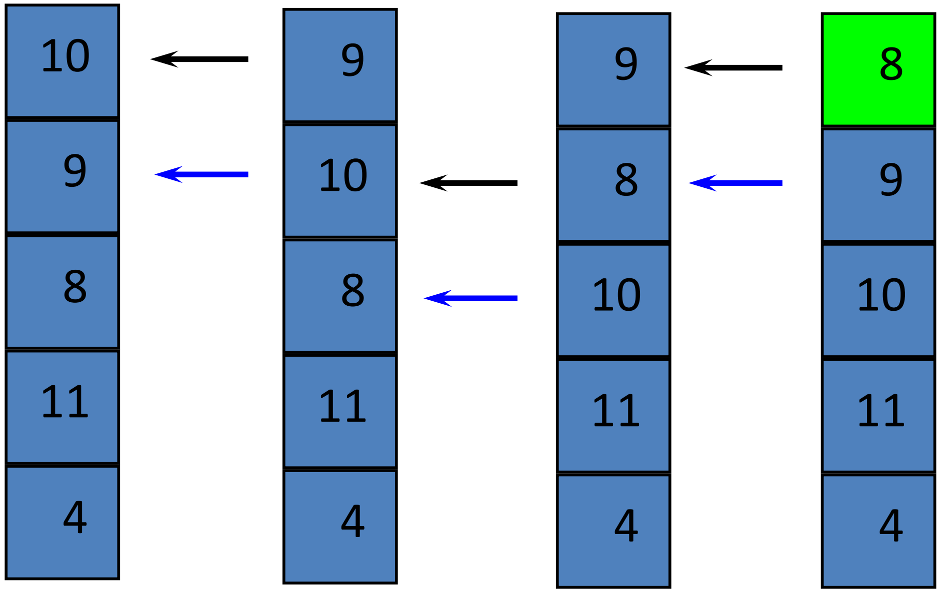
Bubble Sort



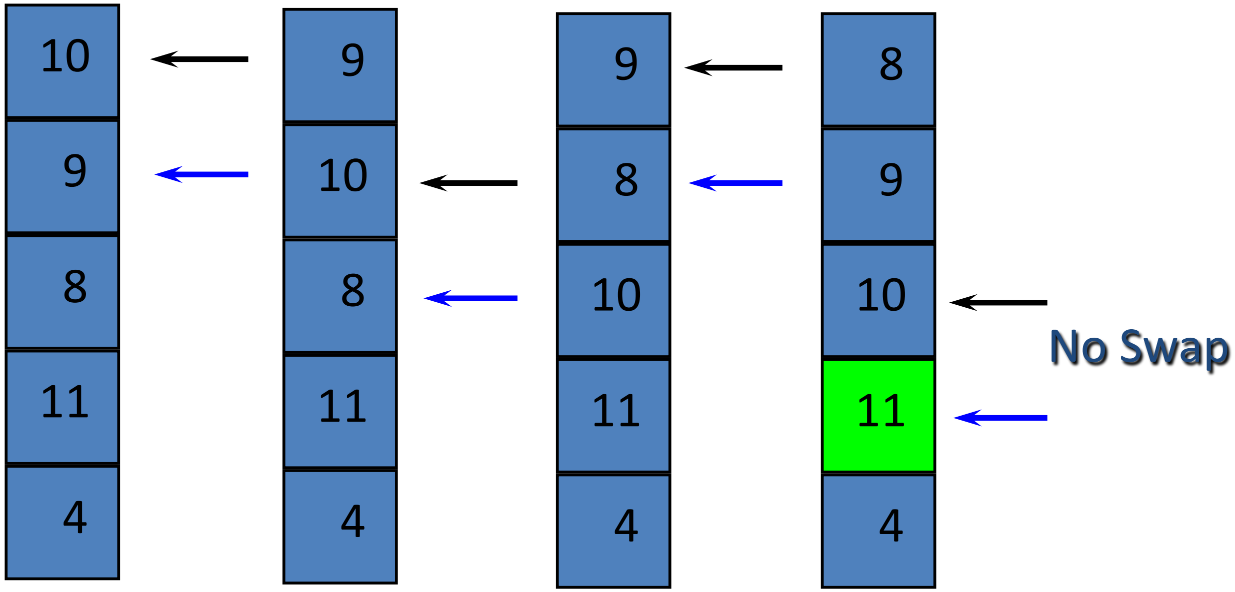
Bubble Sort



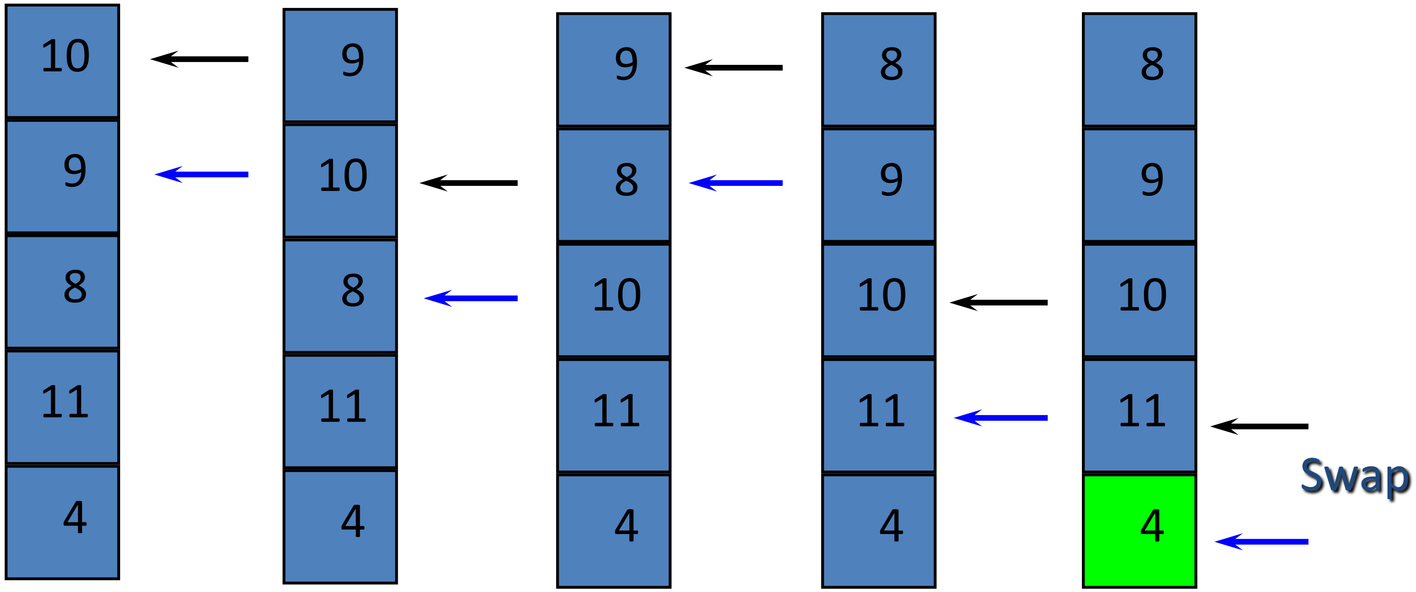
Bubble Sort



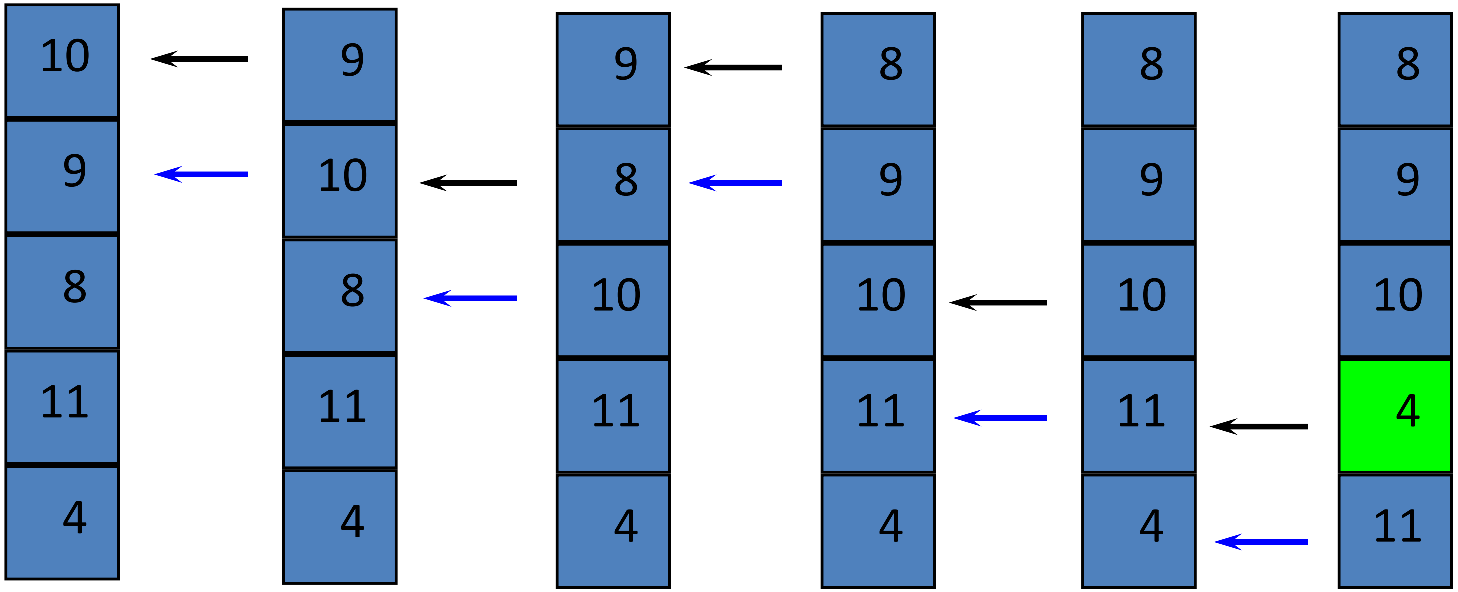
Bubble Sort



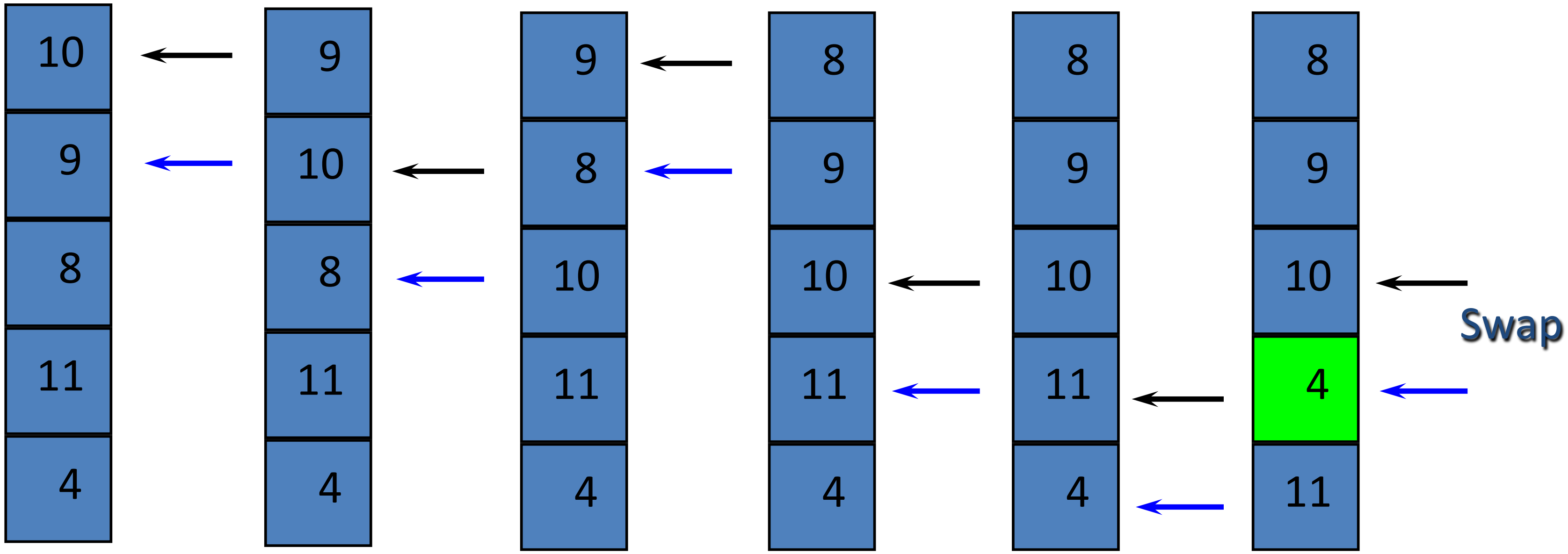
Bubble Sort



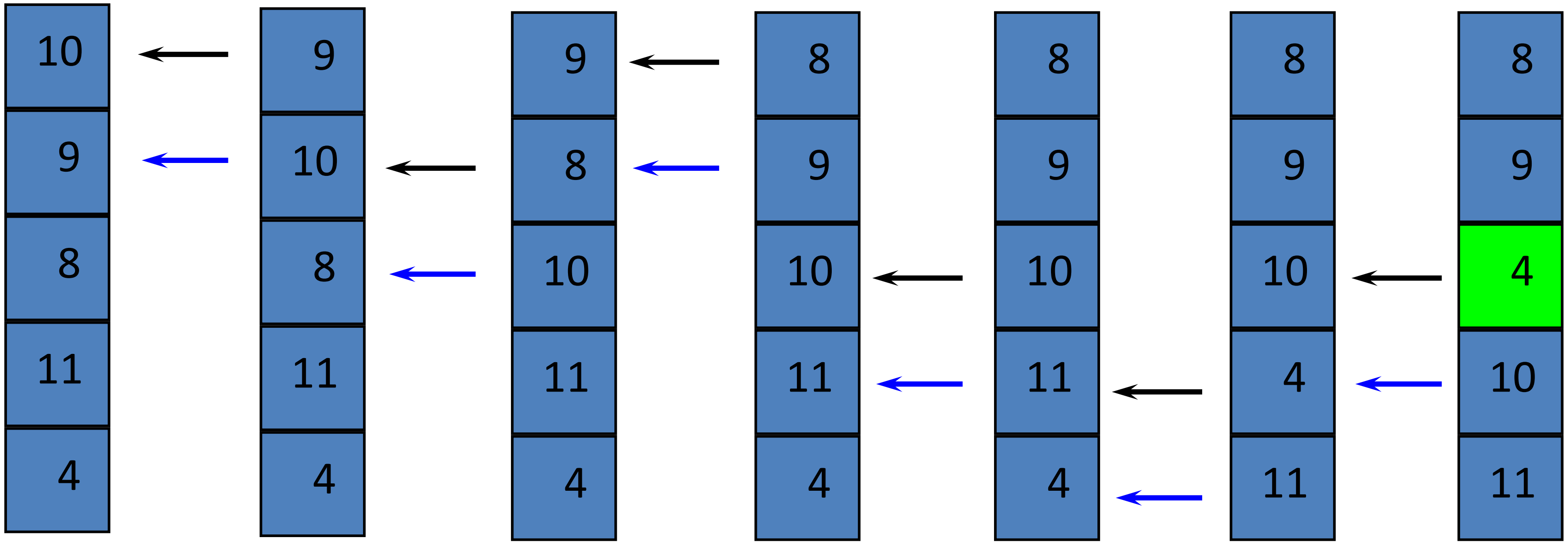
Bubble Sort



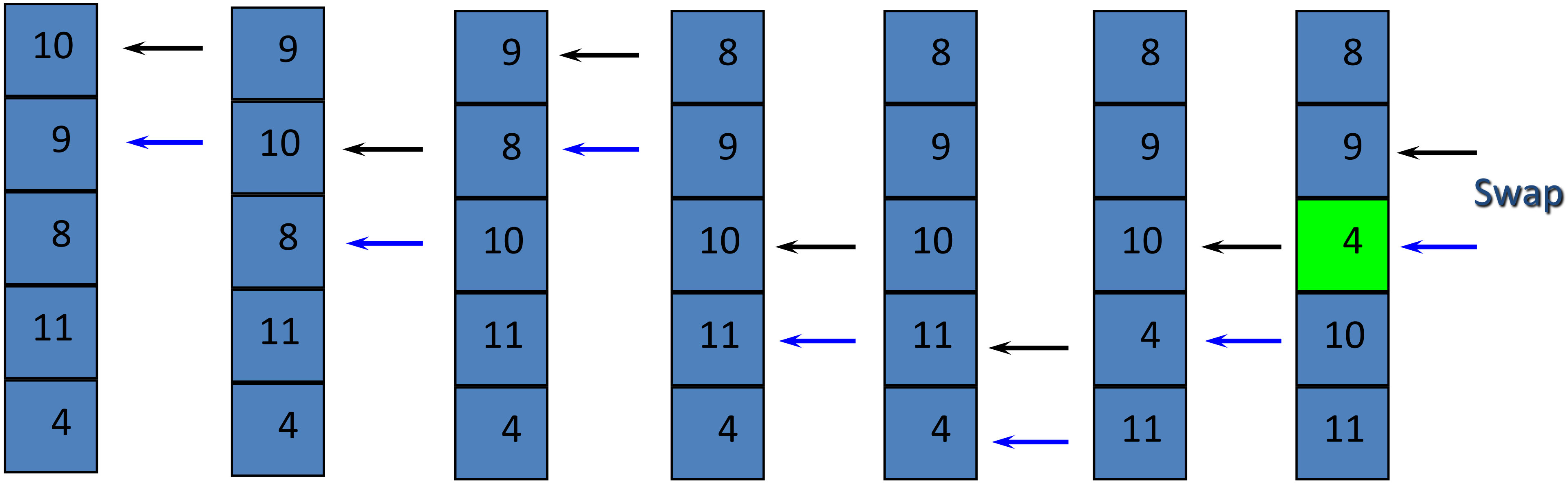
Bubble Sort



Bubble Sort

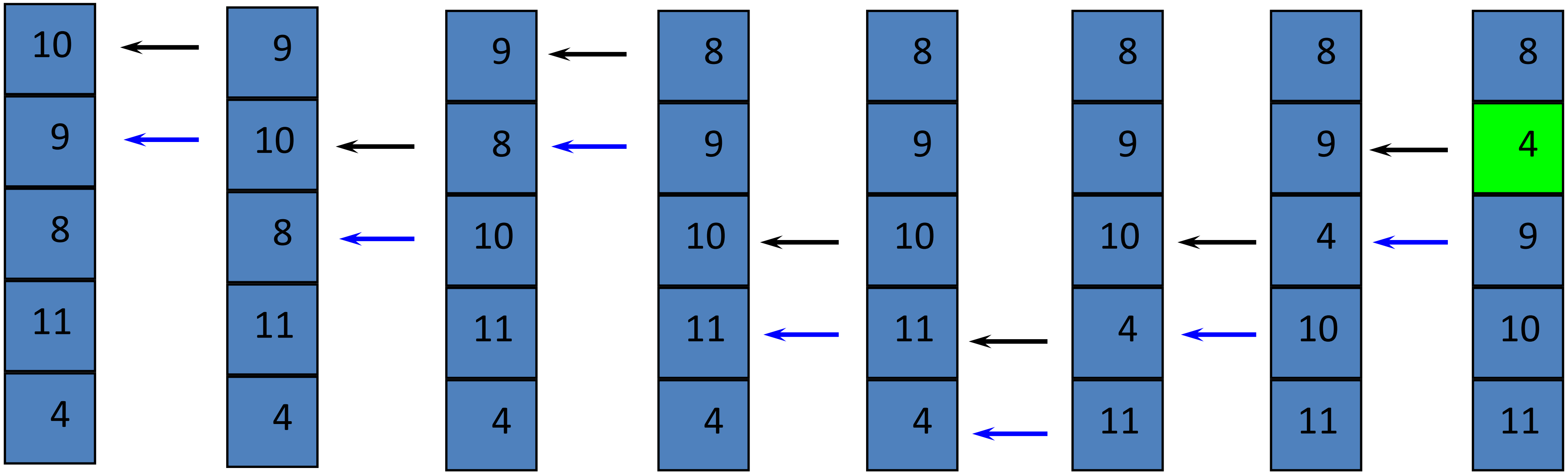


Bubble Sort



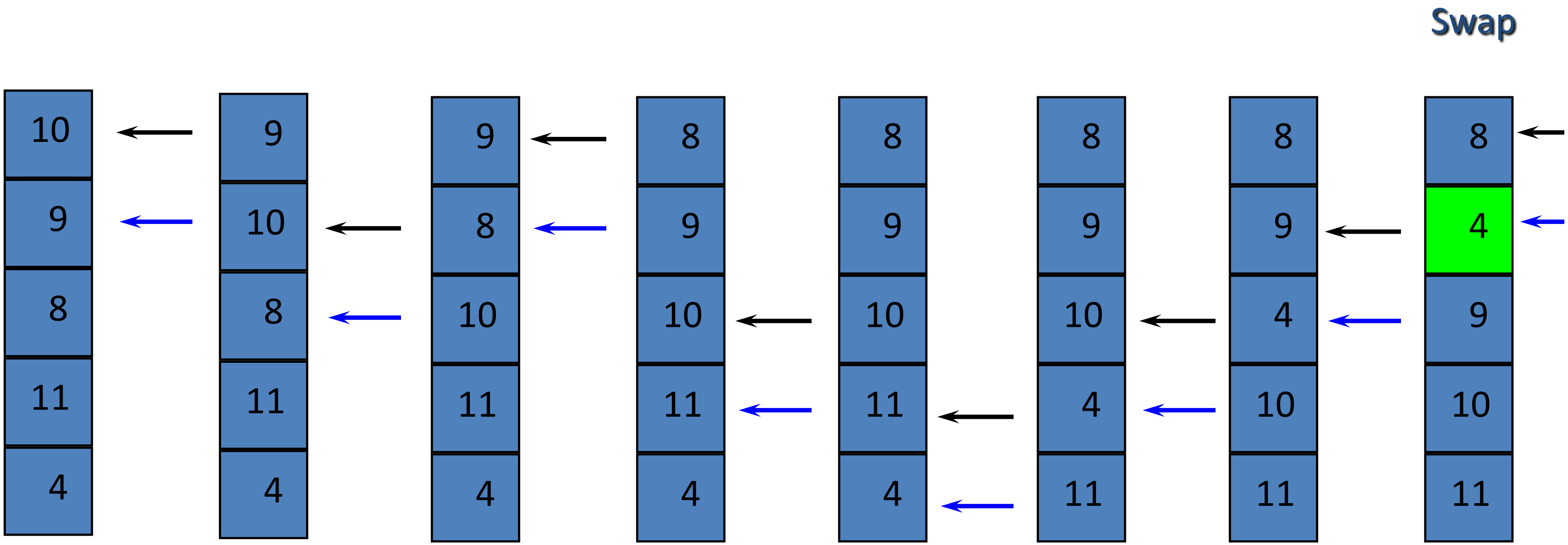
10
9
8
11
4
9
10
8
11
4
9
8
10
11
4
8
9
10
11
4
8
9
10
11
4
8
9
10
4
11
8
9
4
10
11
Swap

Bubble Sort

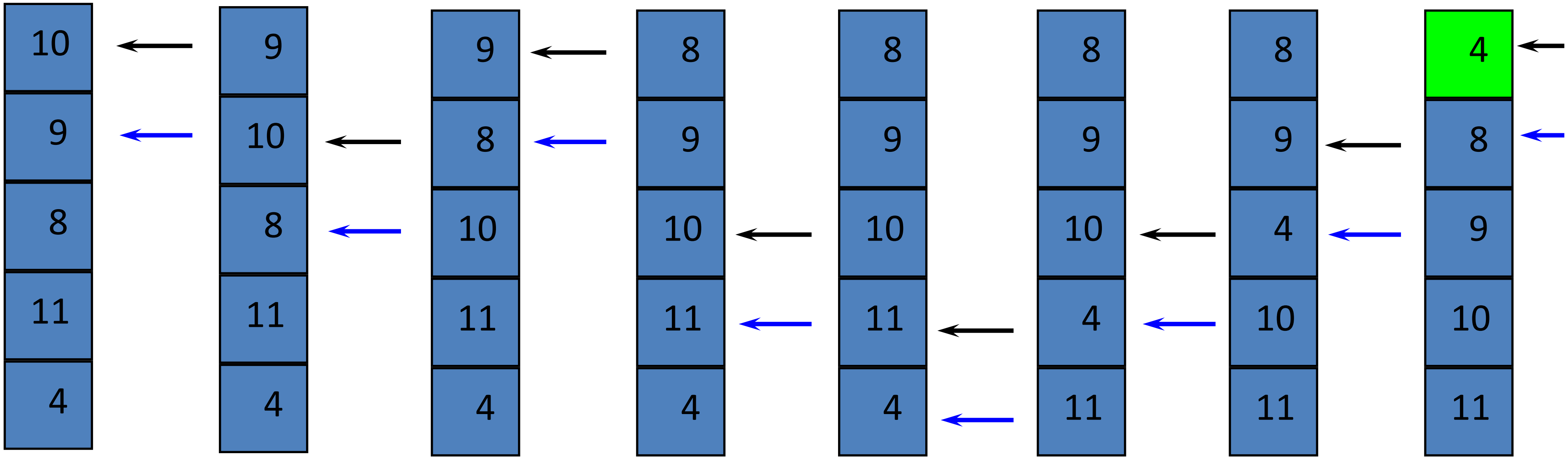


10
9
8
11
4
9
10
8
11
4
9
8
10
11
4
8
9
10
11
4
8
9
10
11
4
8
9
10
4
11
8
9
4
10
11
8
4
9
10
11

Bubble Sort


Swap
10
9
8
11
4
9
10
8
11
4
9
8
10
11
4
8
9
10
11
4
8
9
10
11
4
8
9
10
4
11
8
9
4
10
11
8
4
9
10
11

Bubble Sort


10
9
8
11
4
9
10
8
11
4
9
8
10
11
4
8
9
10
11
4
8
9
10
11
4
8
9
10
4
11
8
9
4
10
11
4
8
9
10
11

Implementation of Bubble_Sort()

```
int bubble_sort(int *a, int size) {  // int a[]
   int i,j, temp;

   for (i=0; i < size-1; i++) { // why?
      for (j=i; j >= 0; j--) {
         if (a[j] > a[j+1]) {

            /* swap */
            temp = a[j+1];
            a[j+1] = a[j];
            a[j] = temp;
         }
      }
   }
}
```

Bubble Sort

A few observations:

- we don't usually sort numbers; we usually sort records with keys
  - the key can be a number
  - or the key could be a string
  - the record would be represented with a **`struct`**
- The swap should be done with a function (so that a record can be swapped)
- We can make the preceding algorithm more efficient. How? (hint: do we always have to bubble back to the top?)

Bubble Sort

Exercise: implement these changes and write a driver program to test:

- the original bubble sort
- the more efficient bubble sort
- the bubble sort with a swap function
- the bubble sort with structures
- compute the order of time complexity of the bubble sort

Selection Sort

Example:

- Shaded elements are selected
- Boldface elements are in order

Initial Array	29	10	14	37	13
After 1st swap	29	10	14	13	**37**
After 2nd swap	13	10	14	**29**	**37**
After 3rd swap	13	10	**14**	**29**	**37**
After 4th swap	**10**	**13**	**14**	**29**	**37**

Selection Sort

- Assume we are sorting a list represented by an array A of n integer elements

- Selection sort algorithm in pseudo-code

```
last = n-1
Do
    Select largest element from a[0..last]
    Swap it with a[last]
    last = last–1
While (last >= 1)
```

Selection Sort

```
typedef  int  DataType;

void selectionSort(DataType a[] , int n) {

   DataType temp;
   int index_of_largest, index, last;

   for(last= n-1; last >= 1;  last--) {

       // select largest item in a[0..last]
      index_of_largest = 0;
      for(index=1; index <= last; index++) {
         if (a[index] > a[index_of_largest])
            index_of_largest = index;
      }

      // swap largest item with last element
      temp = a[index_of_largest];
      a[index_of_largest] = a[last]);
      a[last]) = temp;
   }
}
```

Insertion Sort

```
I N S E R T I O N S O R T
I N S E R T I O N S O R T
I N S E R T I O N S O R T
E I N S R T I O N S O R T
E I N R S T I O N S O R T
E I N R S T I O N S O R T
E I I N R S T O N S O R T
E I I N O R S T N S O R T
E I I N N O R S T S O R T
E I I N N O R S S T O R T
E I I N N O O R S S T R T
E I I N N O O R R S S T T
E I I N N O O R R S S T T
```

Insertion Sort

```
typedef  int  DataType;

insertion_sort(DataType a[], int n) {

   int i,j;
   int temp;

   for (i=1; i<n; i++) {
      j=i;
      while ((j>0) && (a[j] < a[j-1])) {
         temp = a[j-1]; // swap
         a[j-1] = a[j];
         a[j] = temp;

         j = j-1;
      }
   }
}
```