Searching and Sorting

Searching

- We search everyday!
  - Search for your wallet before leaving home
  - Search for the answer of an assignment from the Internet
  - Search for your friends in the canteen.

Searching

- In chapter 1,
  - We search for the textbook from the library

EXAMPLE OF SEARCHING

- Algorithm 1
  - Input: Title/Author/Keywords/Call number
  - Output: Get the book
  - Procedures:
    - 1. Go to the library
    - 2. Scan all books in the library from the leftmost bookshelf
    - 3. Compare the book information with the input
    - 4. If they are matched, get the book. Otherwise, compare with the next book

EXAMPLE OF SEARCHING

- Algorithm 2
  - Input: Title/Author/Keywords/Call number
  - Output: Get the book
  - Procedures:
    - 1. Go to the library
    - 2. Use a computer to find the call number of the book
    - 3. Directly go to the bookshelf stated in the call number
    - 4. Compare the book information with the input
    - 5. If they are matched, get the book. Otherwise, compare with the next book

Sequential Search

Sequential Search

- Consider the array of seven elements shown in below.

list						
35	12	27	18	45	16	38

Sequential Search

- Suppose that you want to determine whether 27 is in the list

27

Sequential Search

- The sequential search works as follows:
  - 1. Compare 27 with list[0]
    - That is, compare 27 with 35.
  - 2. Because list[0] != 27, then compare 27 with list[1]
    - That is, compare 27 with 12
  - 3. Because list[1] != 27, then compare 27 with list[2]
    - That is, compare 27 with 27

Sequential Search

- The sequential search works as follows:
  - 4. Because list[2] = 27, the search stops.
- The is a successful search!

Sequential Search

- Now search for 10

$\boxed{10}$

Sequential Search

- As before, the search starts with the first element in the list – list[0]
- This time the search item, which is 10, is compared with every item in the list.
- Eventually, no more data is left in the list to compare with the search item.
- This is an unsuccessful search.

Sequential Search

- Two situation to stop the search
  - As soon as you find an element in the list that is equal to the search item
  - You stop the search and report "success"
  - Usually, you also tell the location in the list where the search item was found.

Sequential Search

- Two situation to stop the search
  - Otherwise, after the search item is compared with every element in the list
  - You must stop the search and report “failure”.

Binary Search

BINARY SEARCH

- Generally, to find a value in unsorted array, we should look through elements of an array one by one, until searched value is found.
- In case of searched value is absent from array, we go through all elements.
- In average, complexity of such an algorithm is proportional to the length of the array.

BINARY SEARCH

- Situation changes significantly, when array is sorted.

- If we know it, random access capability can be utilized very efficiently to find searched value quick.

BINARY SEARCH

- Cost of searching algorithm reduces to binary logarithm of the array length.
  - For reference, $\log_2(1\,000\,000) \approx 20$.
- It means, that **in worst case**, algorithm makes 20 steps to find a value in sorted array of a million elements or to say, that it doesn't present it the array.

ALGORITHM

- Algorithm is quite simple. It can be done either recursively or iteratively:

1. get the middle element;

ALGORITHM

- Algorithm is quite simple. It can be done either recursively or iteratively:

2. if the middle element equals to the searched value, the algorithm stops;

ALGORITHM

- Algorithm is quite simple. It can be done either recursively or iteratively:

3. otherwise, two cases are possible:

    - searched value is less than the middle element.
      In this case, go to the step 1 for the part of the array, before middle element.

    - searched value is greater than the middle element.
      In this case, go to the step 1 for the part of the array, after middle element.

ALGORITHM

- Now we should define, when iterations should stop.
  - First case is when searched element is found.
  - Second one is when subarray has no elements.
    - In this case, we can conclude, that searched value doesn't present in the array.

EXAMPLE

- *Example 1.*
  Find 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

  - Step 1 (middle element is 19 > 6): -
    1 5 6 18 19 25 46 78 102 114

  - Step 2 (middle element is 5 < 6): -
    1 5 6 18 19 25 46 78 102 114

  - Step 3 (middle element is 6 == 6): -
    1 5 6 18 19 25 46 78 102 114

EXAMPLE

- *Example* 2. Find 103 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.
  - Step 1 (middle element is 103 > 19): -
    1 5 6 18 19 25 46 78 102 114
  - Step 2 (middle element is 103 > 78): -
    1 5 6 18 19 25 46 78 102 114
  - Step 3 (middle element is 103 > 102): -
    1 5 6 18 19 25 46 78 102 114
  - Step 4 (middle element is 114 > 103): -
    1 5 6 18 19 25 46 78 102 114
  - Step 5 (searched value is absent):

Sorting

SORTING

- Sorting is one of the most important operations performed by computers.

SORTING

- The efficiency of data handling can often be substantially increased if the data are sorted according to some criteria of order.

SORTING

- For example:
  - It would be practically impossible to find a name in the telephone directory if the names were not alphabetically ordered.
  - The same can be said about dictionaries, book indexes, payrolls, bank accounts, student lists, and other alphabetically organized materials.

SORTING

- The first step is to choose the criteria that will be used to order data.
  - Very often, the sorting criteria are natural, as in the case of numbers.
    - A set of numbers can be sorted in ascending or descending order.
      - Ascending: (1, 2, 5, 8, 20)
      - Descending: (20, 8, 5, 2, 1)

SORTING

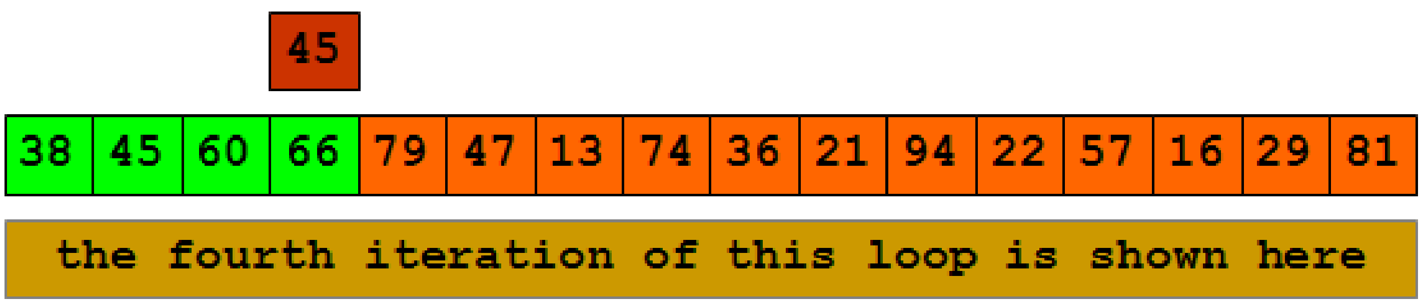
- Names in the phone book are ordered alphabetically by last name, which is the natural order.
  - For alphabetic and non-alphabetic characters, the American Standard Code for Information Interchange (ASCII) code is commonly used.

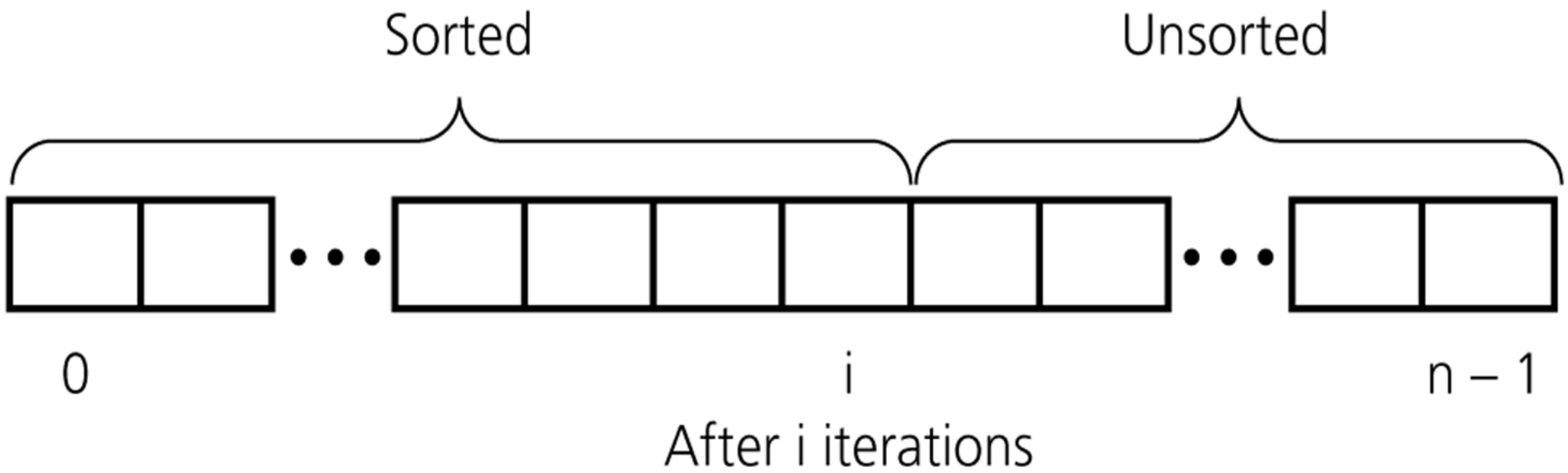
SORTING

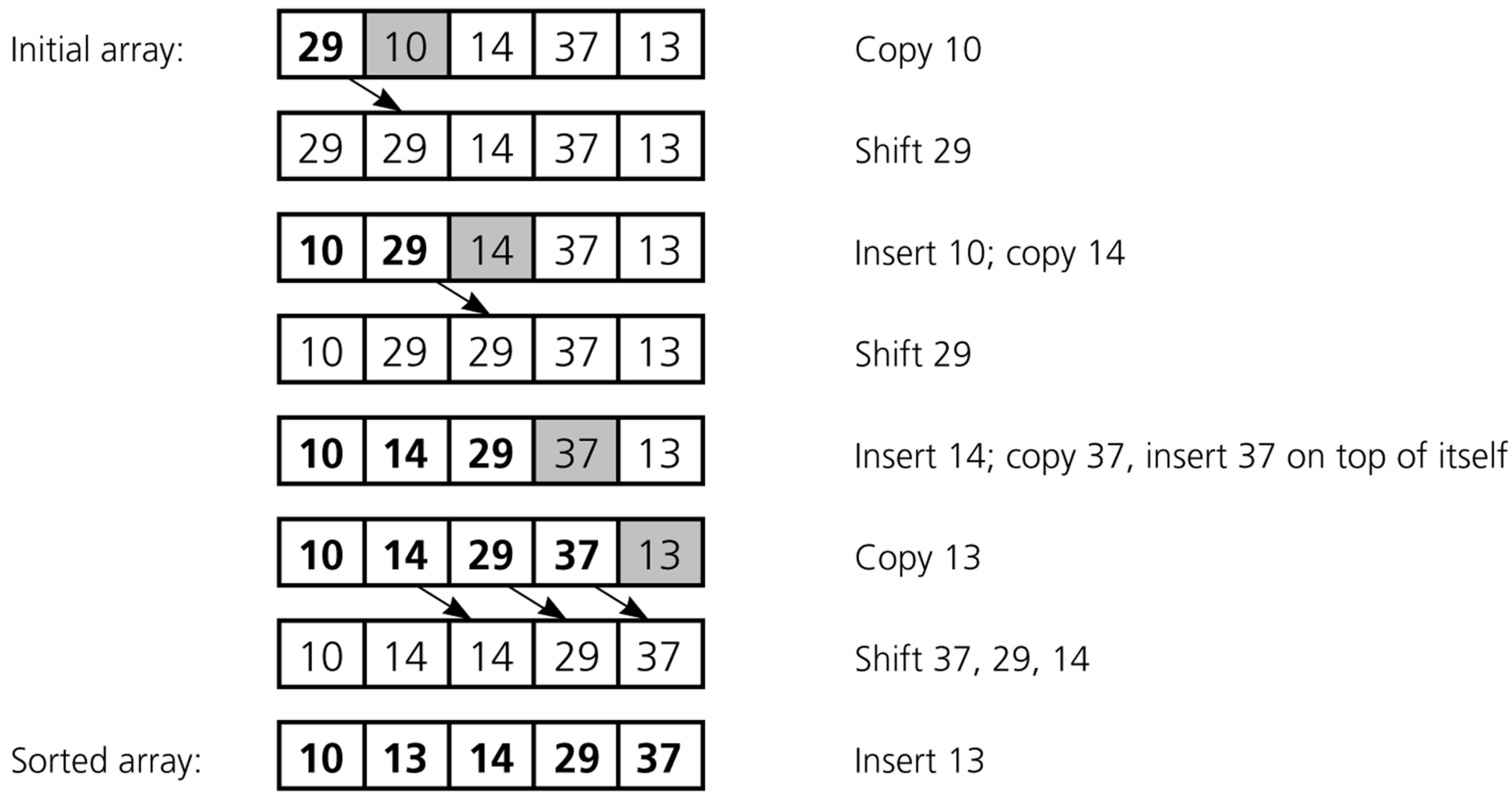
- There are many sorting algorithms and we will focus on the following:
  - Insertion sort
  - Selection sort
  - Bubble sort
  - Quicksort

Insertion Sort

- while some elements unsorted:
  - Using linear search, find the location in the sorted portion where the 1st element of the unsorted portion should be inserted
  - Move all the elements after the insertion location up one position to make space for the new element




Sorted
Unsorted
0
i
n – 1
After i iterations

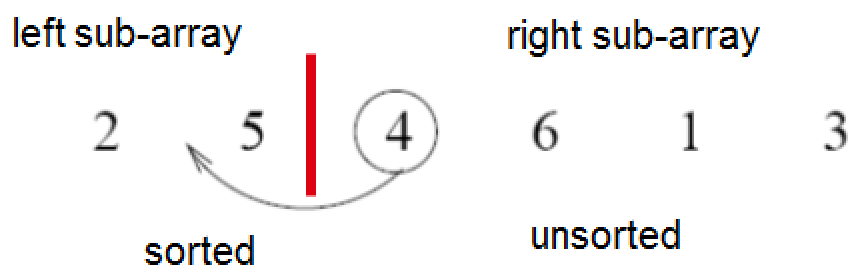

Initial array:
29 10 14 37 13
Copy 10
29 29 14 37 13
Shift 29
10 29 14 37 13
Insert 10; copy 14
10 29 29 37 13
Shift 29
10 14 29 37 13
Insert 14; copy 37, insert 37 on top of itself
10 14 29 37 13
Copy 13
10 14 14 29 37
Shift 37, 29, 14
Sorted array:
10 13 14 29 37
Insert 13

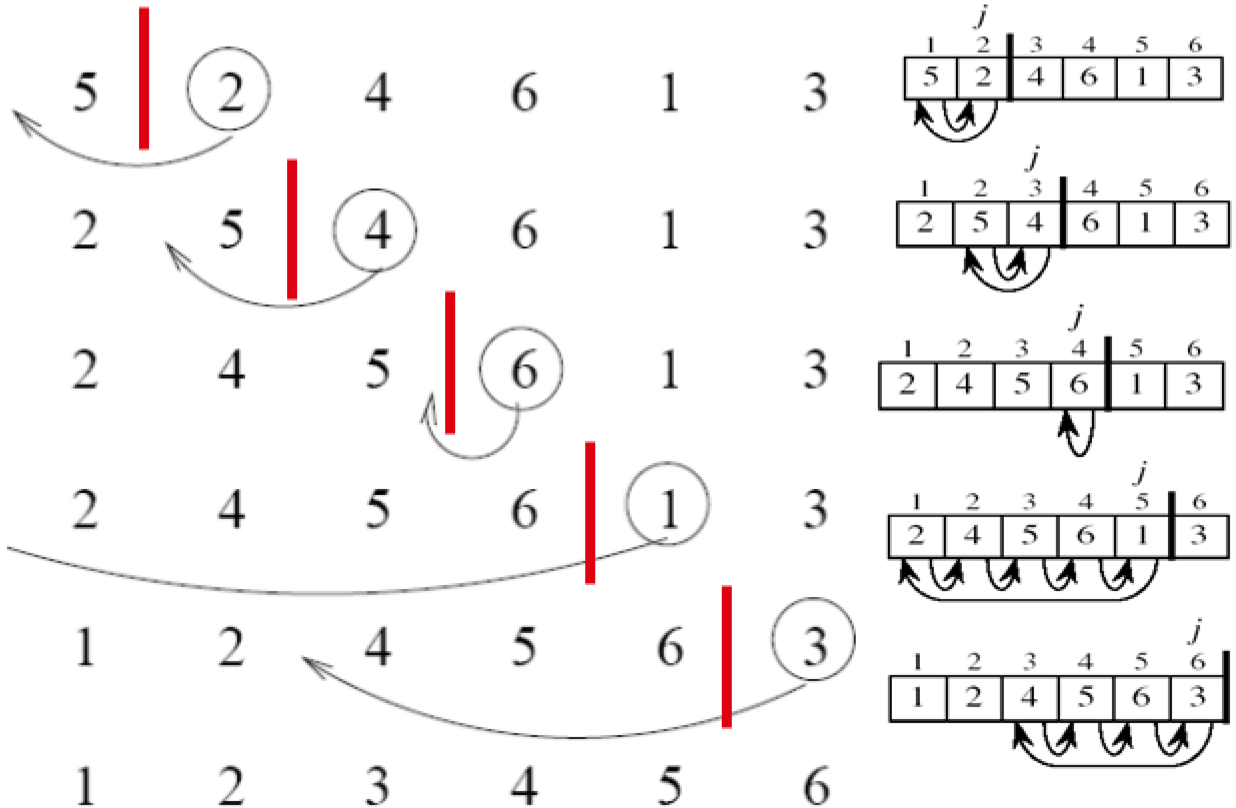
One more example

input array

5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:




5 2 4 6 1 3
2 5 4 6 1 3
2 4 5 6 1 3
2 4 5 6 1 3
1 2 4 5 6 3
1 2 3 4 5 6
j
1 2 3 4 5 6
5 2 4 6 1 3
j
1 2 3 4 5 6
2 5 4 6 1 3
j
1 2 3 4 5 6
2 4 5 6 1 3
j
1 2 3 4 5 6
2 4 5 6 1 3
j
1 2 3 4 5 6
1 2 4 5 6 3

C function

```
public void insertionSort(Comparable[] arr) {
    for (int i = 1; i < arr.length; ++i) {
        Comparable temp = arr[i];
        int pos = i;
        // Shuffle up all sorted items > arr[i]
        while (pos > 0 &&
               arr[pos-1].compareTo(temp) > 0) {
            arr[pos] = arr[pos–1];
            pos--;
        } // end while
        // Insert the current item
        arr[pos] = temp;
    }
}
```

Selection Sort

Selection Sort Algorithm

- Given an array of items, arrange the items so that they are sorted from smallest to largest.
- **Select** next item, in turn, that will be appended to the sorted part of the array:
  - Scan the array to find the smallest value, then swap this value with the value at cell 0.
  - Scan the remaining values (all but the first value), to find the next smallest, then swap this value with the value at cell 1.
  - Scan the remaining values (all but the first two) to find the next smallest, then swap this value with the value at cell 2.
  - Continue until the array is sorted.

Example: Selection Sort

[0]	[1]	[2]	[3]	[4]	
5	1	3	7	2	find min
1	5	3	7	2	swap to index 0
1	5	3	7	2	find min
1	2	3	7	5	swap to index 1
1	2	3	7	5	find min
1	2	3	7	5	swap to index 2
1	2	3	7	5	find min
1	2	3	5	7	swap to index 3

```java
public static void selectionSort(int[] data) {

  for (int numSort = 0; numSort <  data.length-1 ; numSort++){

     // find the next minimum
     int minPos =  numSort  ; // initial position of next min
     for (int pos =  numSort+1  ; pos < data.length; pos++) {
        if (data[minPos] > data[pos])
            minPos = pos; // found new min
     }

     // swap min to next position in sorted list
     int temp = data[minPos];
     data[minPos] = data[ numSort ];
     data[ numSort ] = temp;
  }
}
```

Bubble Sort

- Simplest sorting algorithm
- Idea:
  - 1. Set flag = false
  - 2. Traverse the array and compare pairs of two consecutive elements
    - 1.1 If $E1 \leq E2$ -> OK (do nothing)
    - 1.2 If $E1 > E2$ then Swap(E1, E2) and set flag = true
  - 3. repeat 1. and 2. while flag=true.

Bubble Sort

1	1	23	2	56	9	8	10	100
2	1	2	23	56	9	8	10	100
3	1	2	23	9	56	8	10	100
4	1	2	23	9	8	56	10	100
5	1	2	23	9	8	10	56	100

---- finish the first traversal ----

1	1	2	23	9	8	10	56	100
2	1	2	9	23	8	10	56	100
3	1	2	9	8	23	10	56	100
4	1	2	9	8	10	23	56	100

---- finish the second traversal ----

…

C function

```
public void bubbleSort (Comparable[] arr) {
  boolean isSorted = false;
  while (!isSorted) {
    isSorted = true;
    for (i = 0; i<arr.length-1; i++)
      if (arr[i].compareTo(arr[i+1]) > 0) {
        Comparable tmp = arr[i];
        arr[i] = arr[i+1];
        arr[i+1] = tmp;
        isSorted = false;
      }
  }
}
```

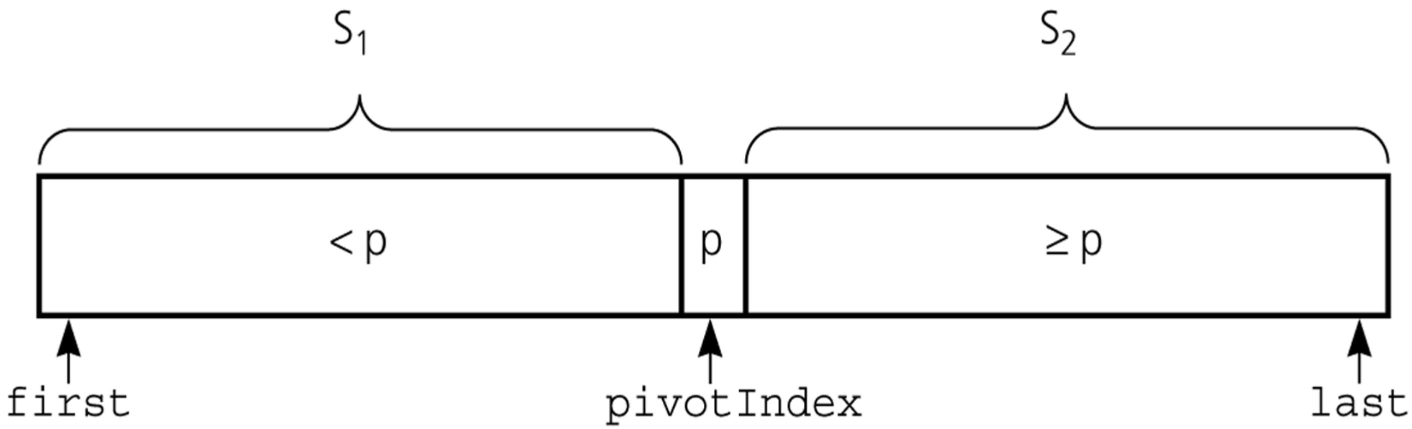
Quick Sort

- Quicksort is also based on the *divide-and-conquer* paradigm.
- It works as follows:
    1. First, it partitions an array into two parts,
    2. Then, it sorts the parts independently,
    3. Finally, it combines the sorted subsequences by a simple concatenation.

The quick-sort algorithm consists of the following three steps:

1. ***Divide***: Partition the list.
    - To partition the list, we first choose some element from the list for which we hope about half the elements will come before and half after. Call this element the ***pivot***.
    - Then we partition the elements so that all those with values less than the pivot come in one sublist and all those with greater values come in another.
2. ***Recursion***: Recursively sort the sublists separately.
3. ***Conquer***: Put the sorted sublists together.

- Partitioning places the pivot in its correct place position within the array.



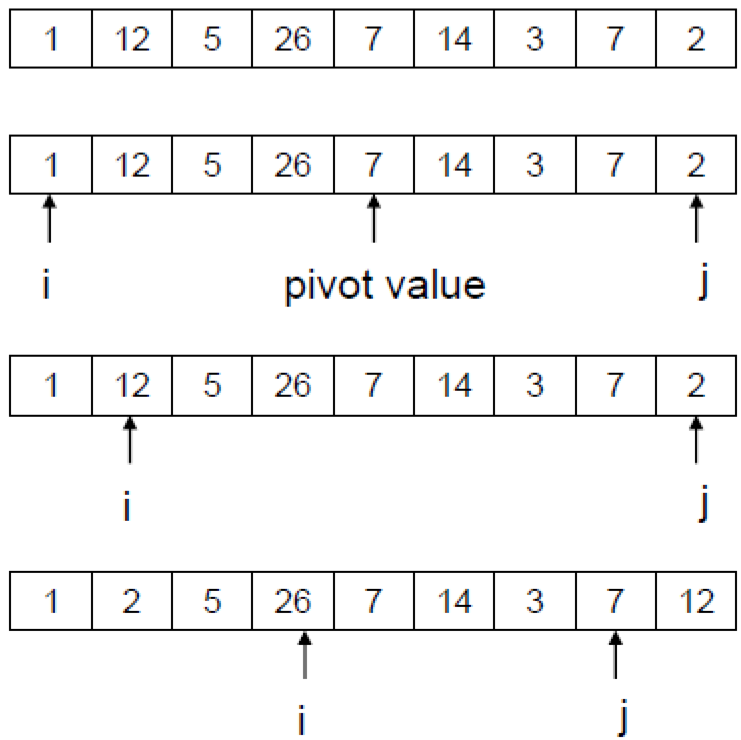
- Arranging the array elements around the pivot p generates two smaller sorting problems.
  - sort the left section of the array, and sort the right section of the array.
  - when these two smaller sorting problems are solved recursively, our bigger sorting problem is solved.

QUICKSORT

- *Example:*
  - Sort {1, 12, 5, 26, 7, 14, 3, 7, 2} using quicksort.

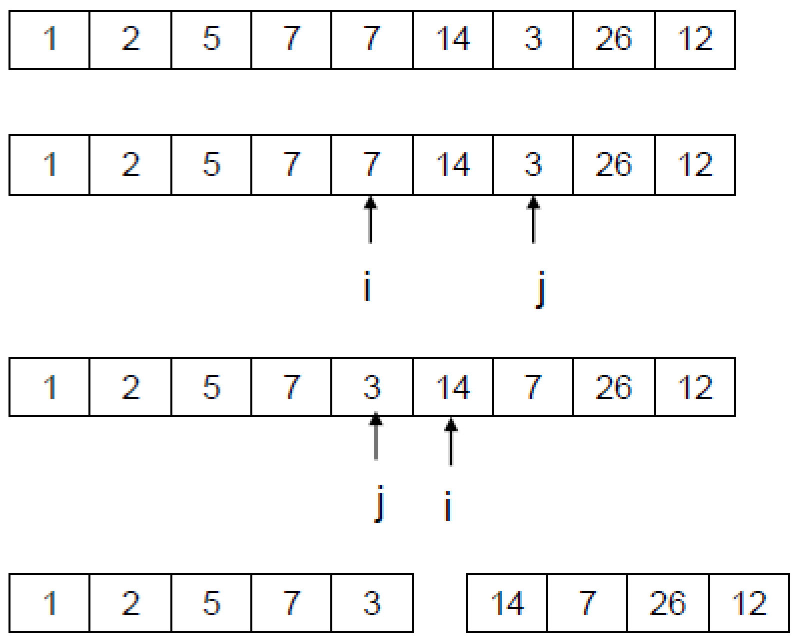
1	12	5	26	7	14	3	7	2

QUICKSORT



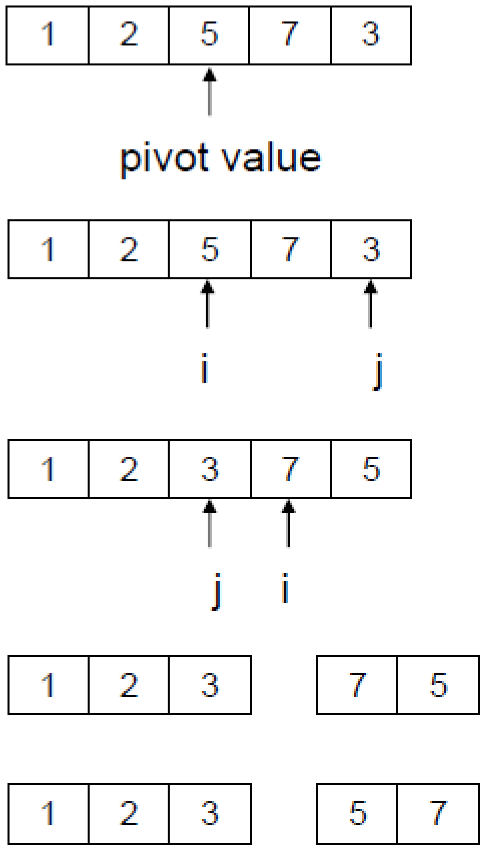
- Unsorted
- Pivot value = 7
- 12 >= 7 >= 2. swap
- 26 >= 7 >= 7, swap

QUICKSORT



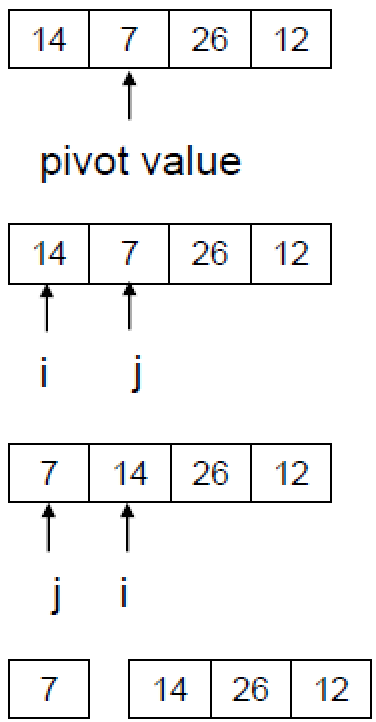
- 7 >= 7 >= 3, swap
- i > j, stop partition
- run quicksort recursively

QUICKSORT



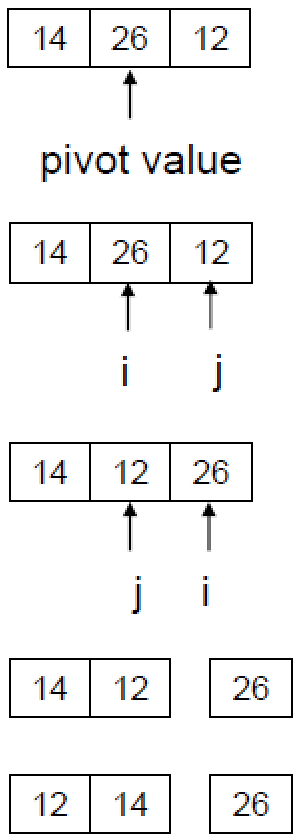
- Pivot value = 5
- 5 >= 5 >= 3. swap
- i > j, stop partition
- run quicksort recursively

QUICKSORT



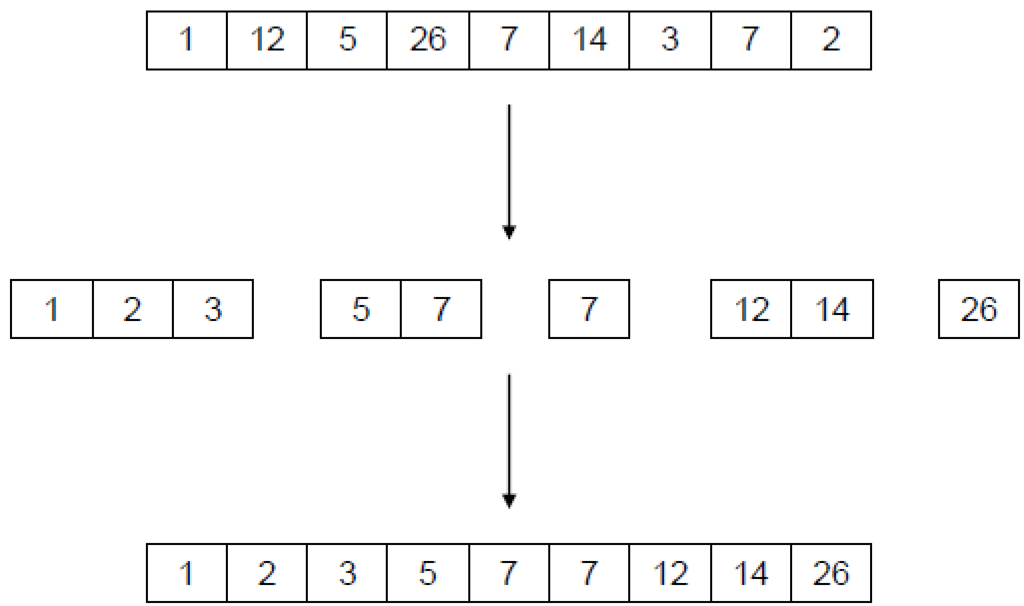
- Pivot value = 7
- 14 >= 7 >= 7. swap
- i > j, stop partition
- run quicksort recursively

QUICKSORT



- Pivot value = 26
- 26 >= 26 >= 12. swap
- i > j, stop partition
- run quicksort recursively

QUICKSORT



```
void quickSort(int arr[], int left, int right) {
    int i = left, j = right;
    int tmp;
    int pivot = arr[(left + right) / 2];

    /* partition */
    while (i <= j) {
        while (arr[i] < pivot)
            i++;
        while (arr[j] > pivot)
            j--;

        if (i <= j) {
            tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
            i++;
            j--;
        }
    };

    /* recursion */
    if (left < j)
        quickSort(arr, left, j);
    if (i < right)
        quickSort(arr, i, right);
}
```