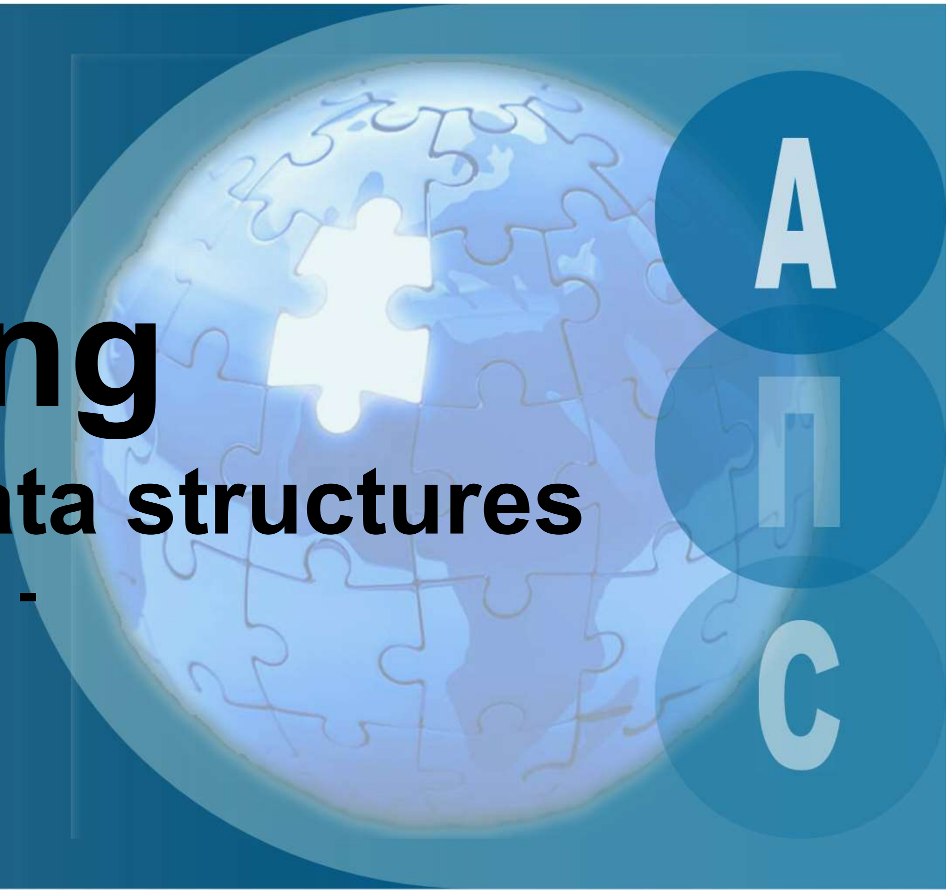
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО



Sorting

Algorithms and data structures

- lectures -

Sorting

- Search
- Comparing collections of records
- Sorting collections of records
- Comparison Sorting Algorithms
  - Maximum element
  - Bubble sort
  - Insertion sort
  - Quick sort
  - Merge sort
  - Analysis of previous algorithms
- Sorting in linear time
- Sort by counting
  - Bucket and radix sort

ALGORITHMS FOR SEARCH IN COLLECTIONS

Search in collection

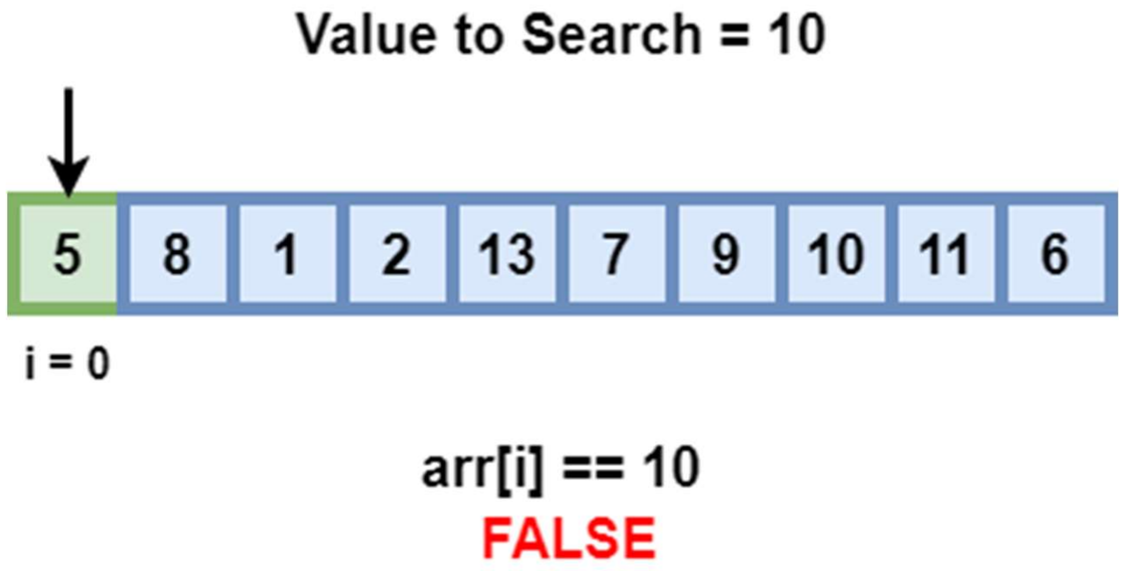
- A collection of records with key fields or keys
- Method of storage
  - Ordered
  - Unordered
- Searching for a key value in an ordered collection
- In array terminology: finding an element with a given value in a sorted array

Sequential search

- A search that starts from the beginning
- Each subsequent element is examined
- Until a suitable value is found or until the end is reached

Sequential search - example

```
def sequential_search(arr: list, n: int, value):
    i = 0
    for x in arr:  # iterate through all values in the list
        if x == value:
            return i  # returns the position when the value is found
        i += 1
    return -1  # returns that the value hasn't been found
```



- Complexity: O(n)

Can it be better than this?

Binary search

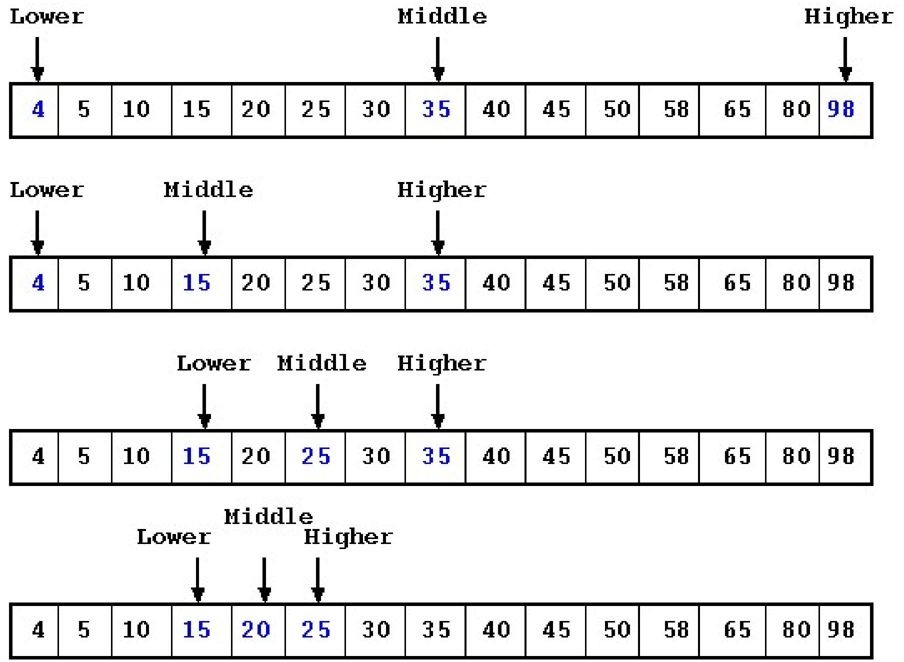
- The preceding algorithm works on both ordered and unordered collections
- Can sorting help with faster finding?
- Binary search (logarithmic search) – allows searching in sorted arrays with much better execution time:
    - O(log n) – the medium and worst
    - O(1) - the best

Binary search - algorithm

- The key that is looked for К, is compared with the key which is at the middle of the array K_m, where m= n/2. The are three possible cases:
    - (i) If $K < K_m$, then if the key exists in the array that is in the part of the array with lower indexes;
    - (ii) If $K = K_m$, then the key is found;
    - (iii) If $K > K_m$, then if the key exists in the array that is in the part of the array with higher indexes;
- The previous steps are repeated until the key is found or there is no part of the array to check.

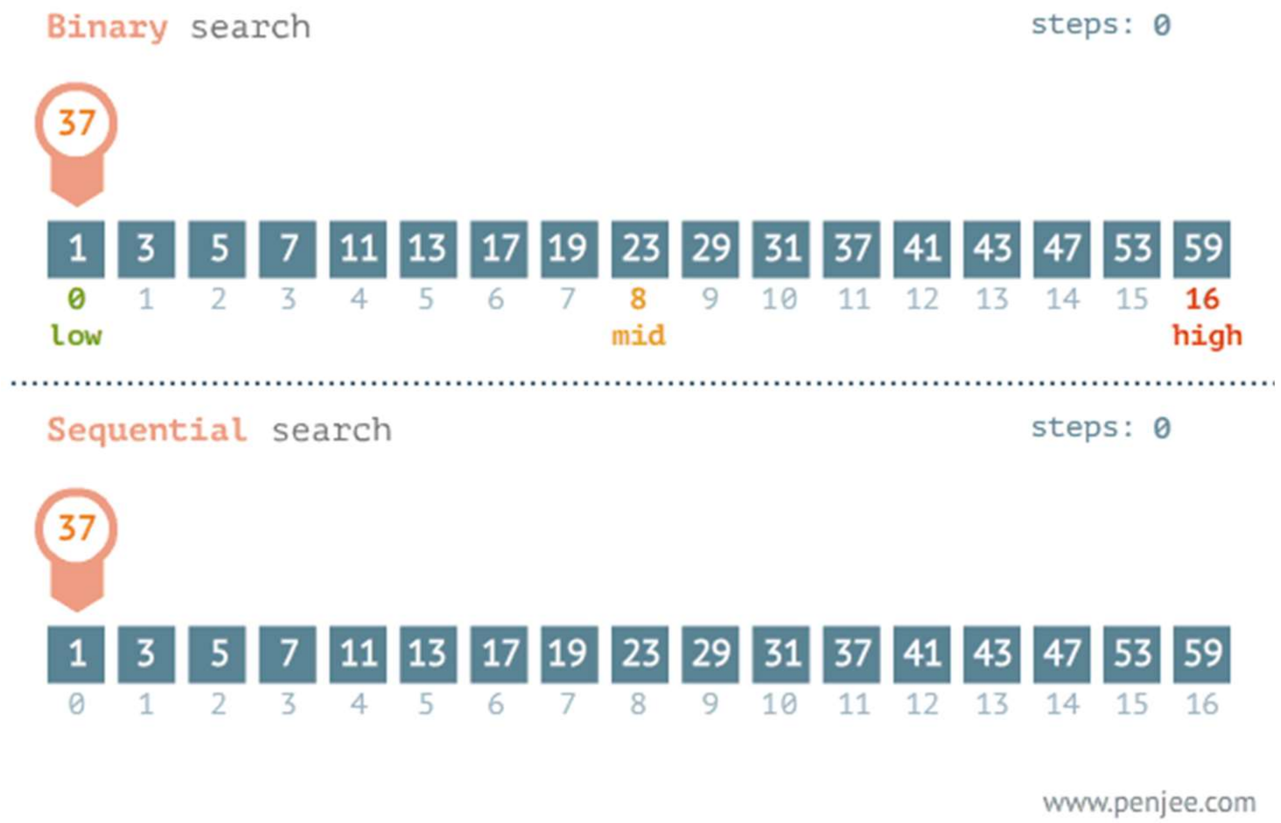
Binary search

```
def binary_search(arr: list, value):
    start = 0
    end = len(arr)
    while start < end:
        mid = (start + end) // 2
        # compare the value in the middle of the list, to the searched value
        if arr[mid] > value:
            end = mid  # next, we search in the left half
        elif arr[mid] < value:
            start = mid + 1  # next, we search in the right half
        else:
            return mid  # we found the value, so return the position
    return -1  # returns that the value hasn't been found
```



- Complexity O(log n)

Comparison of sequential and binary search



COMPARING COLLECTIONS

Comparing collections

- Problem – do two arrays contain the same elements?
- Comparing unordered collections
  - For each element of the first, it is checked sequentially whether it exists in the second
  - Complexity O(n^2)
- Comparing ordered collections
  - They are compared element by element, if they are in the same position, they have the same value
  - Complexity O(n)

Comparing collections - discussion

- Recommendation: If the collections are unordered, it is better to:
  - to order them (2 arrangements),
  - so then they are compared as ordered
- Complexity

  $O(n\log_2 n) + O(n\log_2 n) + O(n) = O(2n\log_2 n+n) = O(n\log_2 n) < O(n^2)$
- How to order them?

COLLECTIONS SORTING

Collections sorting

- Why sorting?
- From the previous examples
  - Sorted collections are faster to search
  - Sorted collections compare faster
- Sorting algorithms
  - Internal sorting
  - External sorting
- Sorting using comparison
- Sorting in linear time

Algorithms illustration

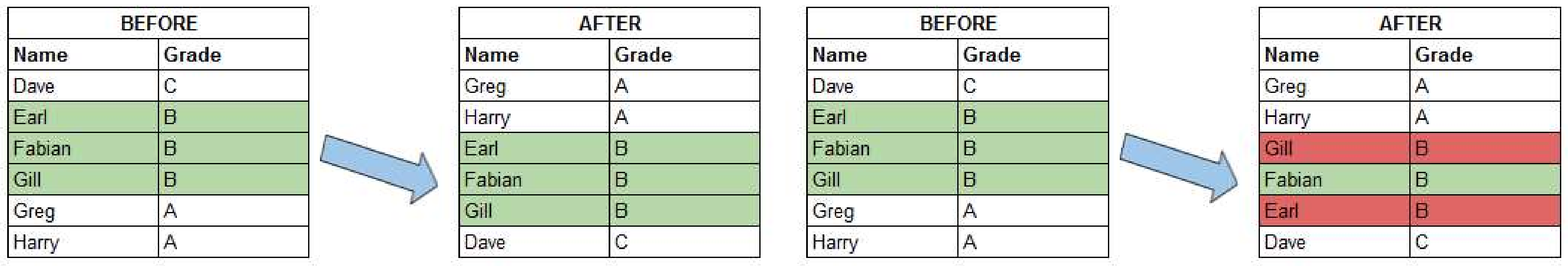
- Sorting algorithms: https://visualgo.net/en/sorting

Sorting using comparison

- ❑ MAXIMUM ENTRY SORT (selection sort)
- ❑ BUBBLE SORT
- ❑ INSERTION SORT
- ❑ QUICKSORT
- ❑ MERGE SORT

Stable sort

- In stable sorting, the relative order of elements with equal sort keys is preserved
- Sort by Grade



BEFORE	
Name	**Grade**
Dave	C
Earl 🟩	B
Fabian 🟩	B
Gill 🟩	B
Greg	A
Harry	A

AFTER	
Name	**Grade**
Greg	A
Harry	A
Earl 🟩	B
Fabian 🟩	B
Gill 🟩	B
Dave	C

Stable

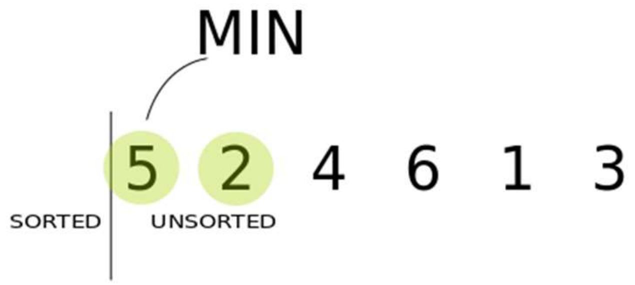
BEFORE	
Name	**Grade**
Dave	C
Earl 🟩	B
Fabian 🟩	B
Gill 🟩	B
Greg	A
Harry	A

AFTER	
Name	**Grade**
Greg	A
Harry	A
Gill 🟥	B
Fabian 🟩	B
Earl 🟥	B
Dave	C

Unstable

MAXIMUM ENTRY (SELECTION) SORT

- The simplest approach
  - Find the largest (smallest) element and put it last (first) in the new array
  - To save space, instead of a new array, replace the last (first) element with the largest (smallest) element in the array
  - Continue the procedure with the subarray composed of the rest of the elements without the first (last) element



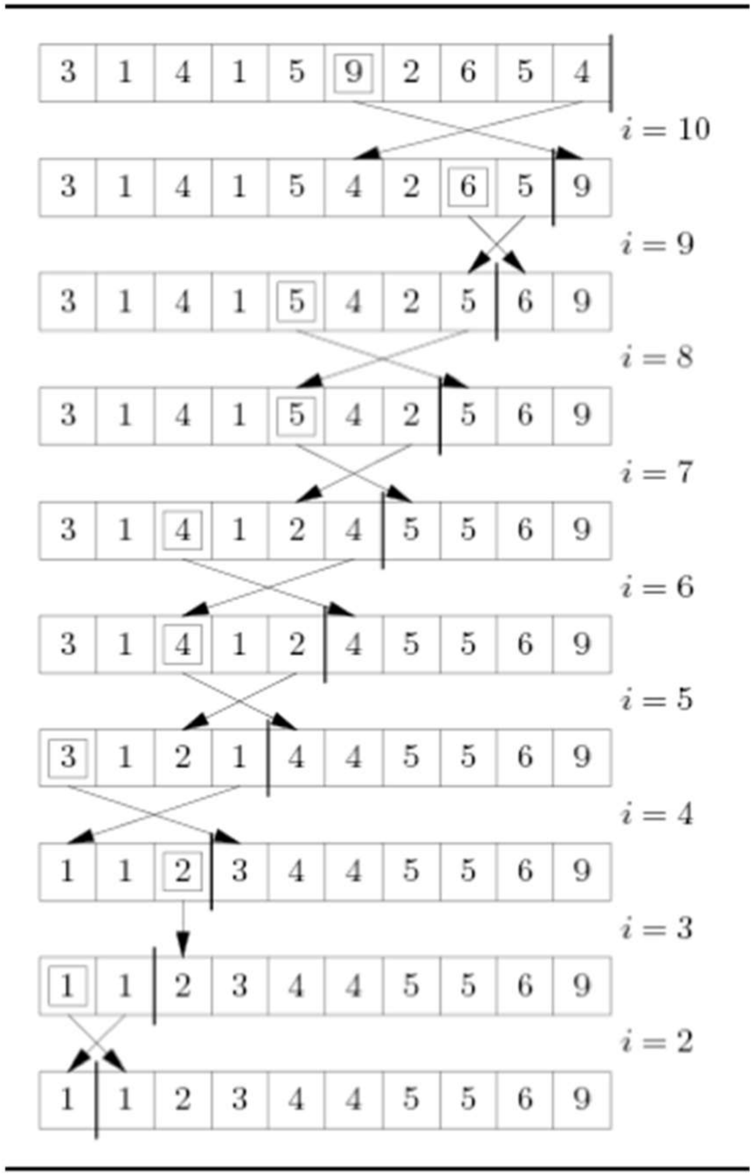
MAXIMUM ENTRY (SELECTION) SORT

```
def selection_sort(arr):
    for i in range(len(arr)):
        # We assume that the first item of
        # the unsorted segment is the smallest
        min_value = i

        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_value]:
                min_value = j

        # Swap values of the lowest unsorted element
        # with the first unsorted element
        arr[i], arr[min_value] = arr[min_value], arr[i]

    return arr
```

MAXIMUM ENTRY (SELECTION) SORT

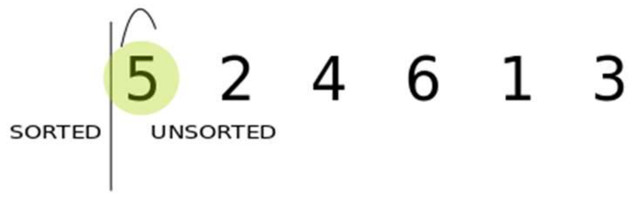
- Complexity
  - n-1 replacements
  - (n-1)+(n-2)+…+2+1 comparisons
  - $O(n^2)$
  - $O(1)$ extra memory
  - $\Theta(n^2)$ comparisons
  - $\Theta(n)$ replacements
  - It is not stable for different types of input arrays
    - for random
    - almost sorted
    - reversed sorted
    - small number of unique values

MAXIMUM ENTRY (SELECTION) SORT



INSERTION SORT

- The idea of the algorithm: insert a record with a given key into a pre-sorted subarray
- It is usually faster and simpler than the Bubble and Selection sort



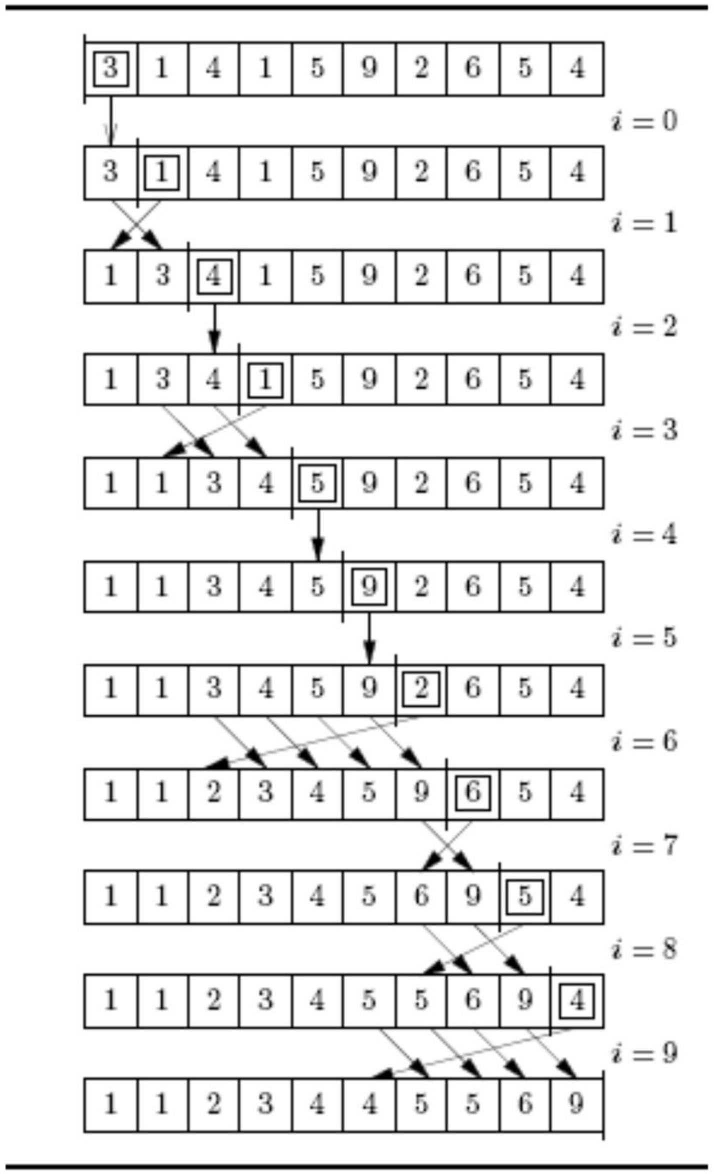
INSERTION SORT

```python
def insertion_sort(arr):
    # We assume that the first item is sorted
    for i in range(1, len(arr)):
        picked_item = arr[i]

        # Reference of the index of the previous element
        j = i - 1

        # Move all items to the right until finding the correct position
        while j >= 0 and arr[j] > picked_item:
            arr[j + 1] = arr[j]
            j -= 1

        # Insert the item
        arr[j + 1] = picked_item

    return arr
```

INSERTION SORT

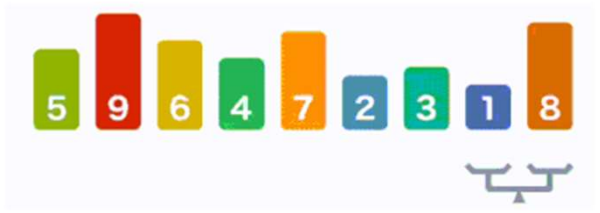
- Complexity:
  - Stable
  - O(1) extra memory
  - O(n^2) comparisons and replacements
  - Adaptable: O(n) time complexity for almost sorted input

INSERTION SORT



BUBBLE SORT

- It works on the principle of the lightest ones floating to the surface, that is, the heaviest ones sinking to the bottom
- Simple for implementation
- Multiple traversals through the array
- In each pass, adjacent elements are swapped if they are not ordered (smaller first)

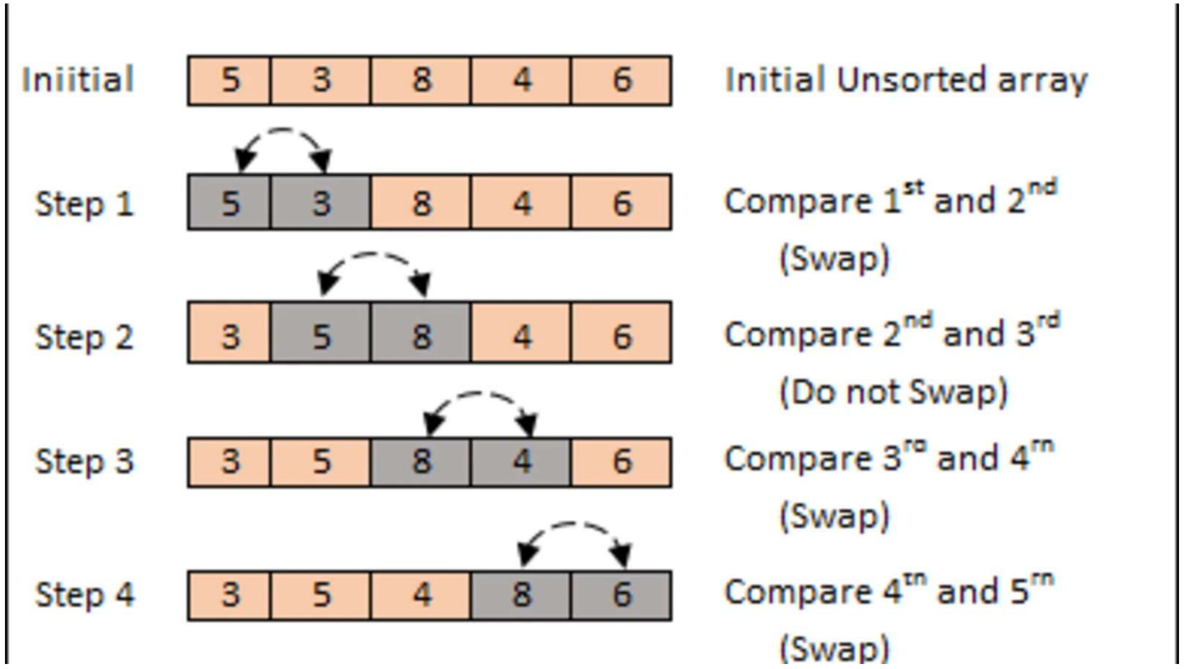


BUBBLE SORT

```python
def bubble_sort(arr):
    # We set swapped to True so the loop runs at least once
    swapped = True

    while swapped:
        swapped = False
        for i in range(len(arr) - 1):
            if arr[i] > arr[i + 1]:
                # Swap the elements
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                # Set the flag to True so we'll loop again
                swapped = True

    return arr
```

- Trivial case: sorted array
- Easy to implement, and memory efficient
- Stable
- Complexity: O(n^2)

BUBBLE SORT


Iniitial
5 3 8 4 6
Initial Unsorted array
Step 1
5 3 8 4 6
Compare 1st and 2nd (Swap)
Step 2
3 5 8 4 6
Compare 2nd and 3rd (Do not Swap)
Step 3
3 5 8 4 6
Compare 3rd and 4th (Swap)
Step 4
3 5 4 8 6
Compare 4th and 5th (Swap)

QUICKSORT

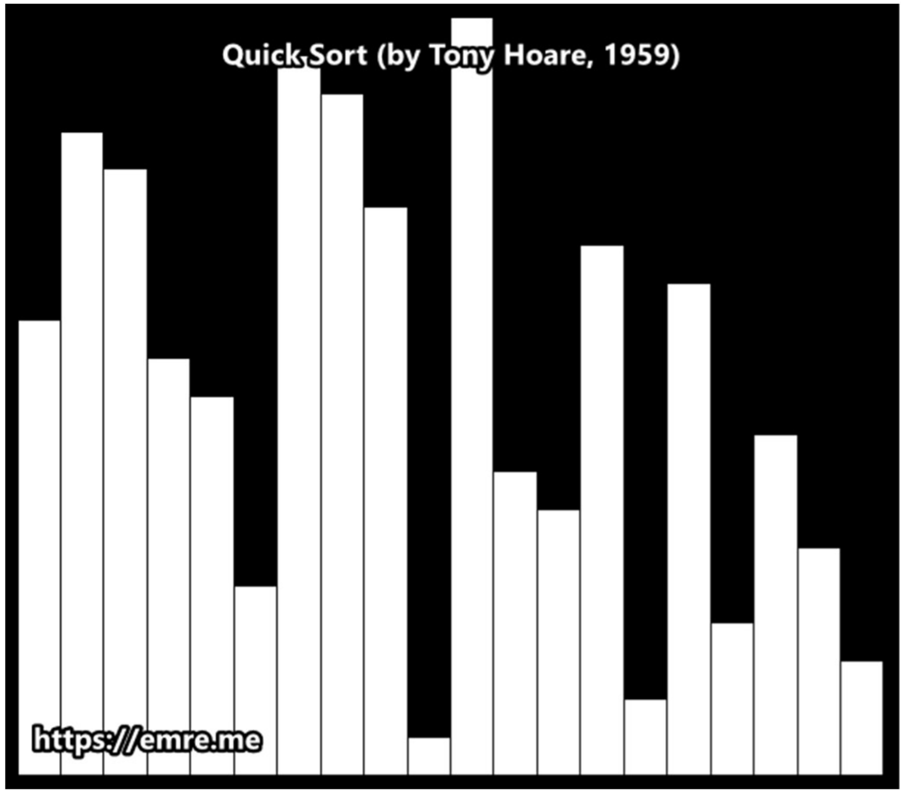
- It has the best execution time in the average case
- The idea is to find an element that will be positioned at the right place in the array
- Then, all the smaller ones should be moved in front of it, and all the bigger ones, after it
- It uses a divide-and-conquer technique

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	m	n
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37]	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

QUICKSORT

- Complexity: $O(n^2)$ worst case
- Complexity: if the pivot key is well selected: $O(n \log n)$
- $O(\lg(n))$ extra memory
- Not adaptable, unstable

- Pivot choice:
  - First
  - Last
  - Random
  - Middle



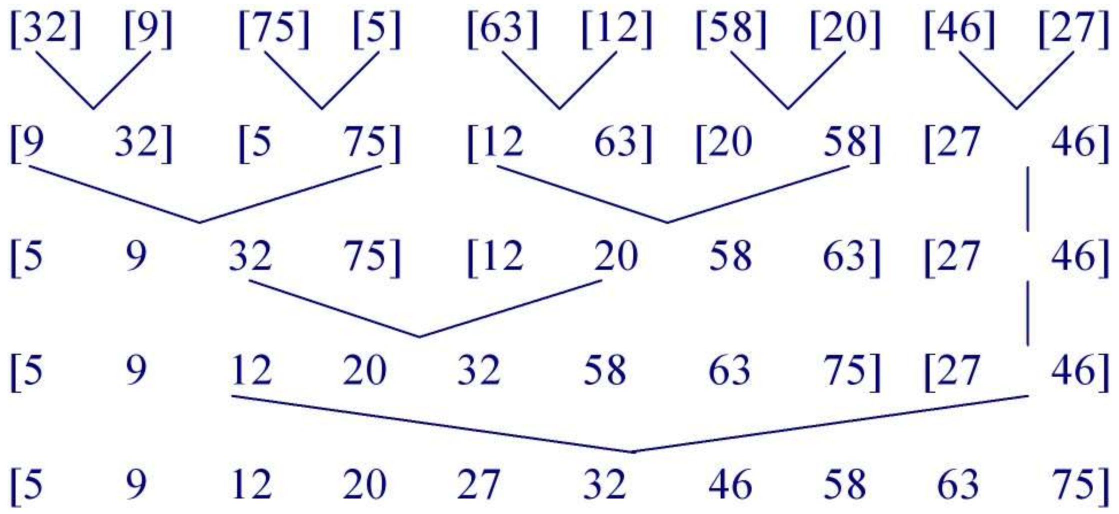
QUICKSORT

```
def partition(arr, low, high):
    # We select the middle element to be the pivot
    pivot = arr[(low + high) // 2]
    i = low - 1
    j = high + 1
    while True:
        i += 1
        while arr[i] < pivot:
            i += 1
        j -= 1
        while arr[j] > pivot:
            j -= 1
        if i >= j:
            return j
        # Swap
        arr[i], arr[j] = arr[j], arr[i]

def quick_sort_helper(arr, low, high):
    if low < high:
        split_point = partition(arr, low, high)
        quick_sort_helper(arr, low, split_point)
        quick_sort_helper(arr, split_point + 1, high)

def quick_sort(arr):
    quick_sort_helper(arr, 0, len(arr) - 1)
```

MERGE SORT

- Based on the idea of concatenating two already sorted arrays
  - It starts by comparing the first elements of the two arrays. In doing so, the smaller of them is placed in the result, and the second element is now considered in the array from which that element was extracted.
  - When the first step exhausts the elements of one array, the elements of the other are added one by one to the resulting array

MERGE SORT


[32] [9] [75] [5] [63] [12] [58] [20] [46] [27]
[9 32] [5 75] [12 63] [20 58] [27 46]
[5 9 32 75] [12 20 58 63] [27 46]
[5 9 12 20 32 58 63 75] [27 46]
[5 9 12 20 27 32 46 58 63 75]

MERGE SORT – merge of sorted lists

```python
def _merge(l_list, r_list):
    result = []
    l_index, r_index = 0, 0

    for i in range(len(l_list) + len(r_list)):
        if l_index < len(l_list) and r_index < len(r_list):

            # If the item at the beginning of the left list is smaller, add it to the sorted list
            if l_list[l_index] <= r_list[r_index]:
                result.append(l_list[l_index])
                l_index += 1

            # If the item at the beginning of the right list is smaller, add it to the sorted list
            else:
                result.append(r_list[r_index])
                r_index += 1

        # If we have reached the end of the of the left list, add the elements from the right list
        elif l_index == len(l_list):
            result.append(r_list[r_index])
            r_index += 1

        # If we have reached the end of the of the right list, add the elements from the left list
        elif r_index == len(r_list):
            result.append(l_list[l_index])
            l_index += 1

    return result
```

MERGE SORT

```python
def merge_sort(array):
    # If the list is a single element, return it
    if len(array) <= 1:
        return array

    mid = len(array) // 2

    # Recursively sort and merge each half
    l_list = merge_sort(array[:mid])
    r_list = merge_sort(array[mid:])

    # Merge the sorted lists into a new one
    return _merge(l_list, r_list)
```

IF YOU COULD JUST TELL ME THE BEST ALGORITHM
YEAH THAT'D BE GREAT
memegenerator.net

Algorithms comparison

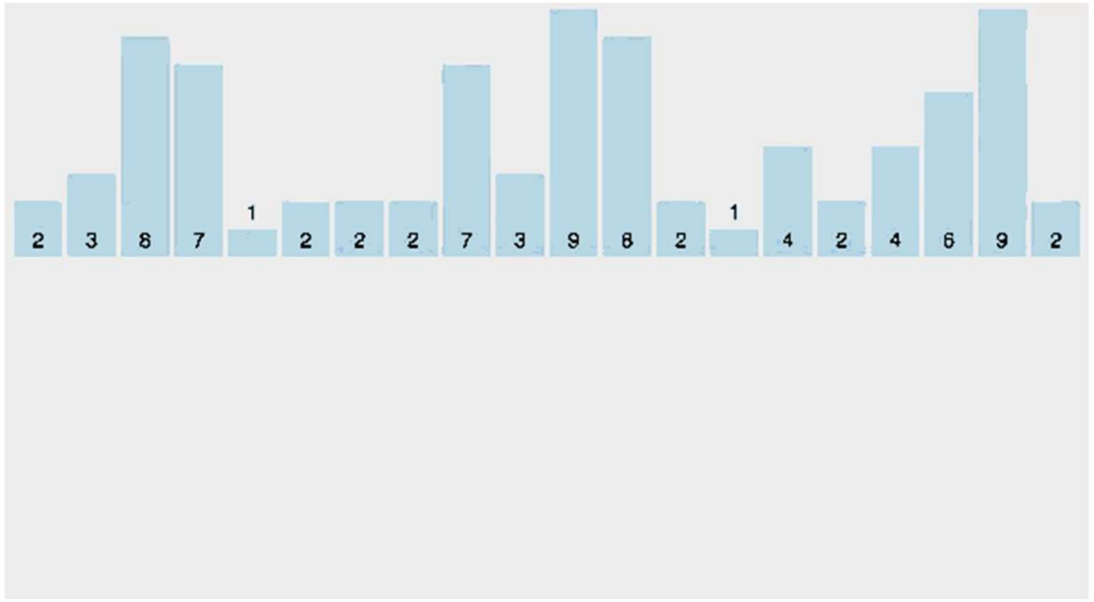
Growth Rate to sort N Items					
Algorithm	Stability	Inplace	Running Time	Extra Space	Note
Insertion	Yes	Yes	Between n and n^2	1	Depends on the order of the input key
Selection	No	Yes	n^2	1	
Bubble	Yes	Yes	n^2	1	
Merge	Yes	No	nlogn	n	
Heap	No	Yes	nlogn	n	
Quick	No	Yes	nlogn	logn	Probabilistic

Sorting in linear time

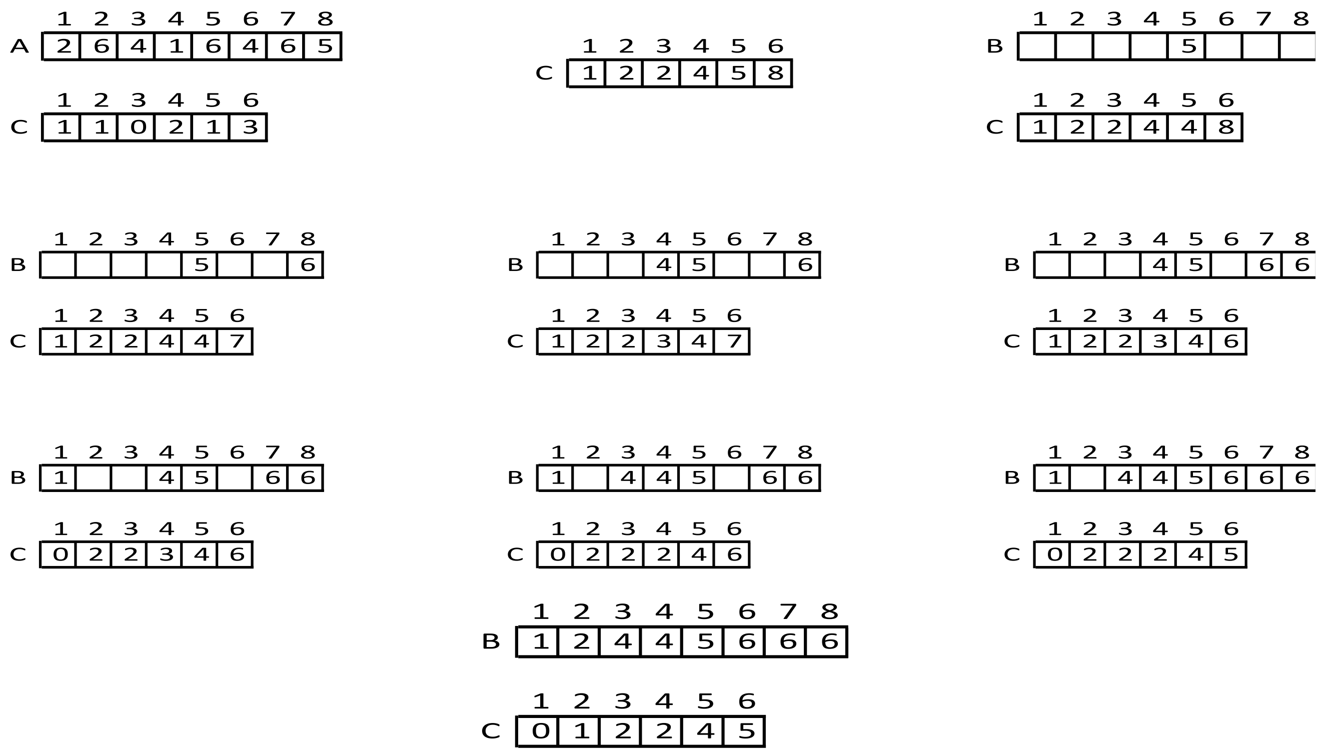
- Under certain conditions, it is possible to do sorting in linear time
  - The keys to be sorted are in the interval 1...k
- Counting sort
- Bucket sort
- Radix sort

Counting sort

- For each different key value, the number of appearances in the collection are counted
- Complexity: O(k+n), where k<<n, O(n)



Counting sort



A – input array
B – output array
C – help array for counting

Sort by counting

```
def count_sort(arr):
    # The output array that will have sorted arr
    output = [0 for i in range(len(arr))]

    # Create a count array to store count of individual values and initialize count array as 0
    count = [0 for i in range(256)]

    # For storing the resulting answer since the string is immutable
    ans = [0 for x in arr]

    # Store count of each character
    for i in arr:
        count[i] += 1

    # Change count[i] so that count[i] now contains actual position of this value in output array
    for i in range(256):
        count[i] += count[i - 1]

    # Build the output character array
    for i in range(len(arr)):
        output[count[arr[i]] - 1] = arr[i]
        count[arr[i]] -= 1

    # Copy the output array to arr, so that arr now contains sorted values
    for i in range(len(arr)):
        ans[i] = output[i]
    return ans
```

Other algorithms in linear time

- Bucket sort – each element is identified with a certain "bucket", then it is counted how many are in each bucket
  - A variation of the counting algorithm
- Radix sort – using the characteristics of the number representation in a number system for sorting
  - Counting by hundreds, then tens, then ones
  - Counting by part of a number in binary representation