ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Сортирање

АЛГОРИТМИ И ПОДАТОЧНИ СТРУКТУРИ

- предавања -

Сортирање

- Пребарување
- Споредување на колекции од записи
- Сортирање на колекции од записи
- Алгоритми за сортирање со споредување
  - Максимален елемент
  - Bubble sort
  - Сортирање со вметнување
  - Брзо сортирање – Quick sort
  - Сортирање со спојување
  - Анализа на претходните алгоритми
- Сортирање во линеарно време
  - Сортирање со броење
  - Bucket и radix сортирање

АЛГОРИТМИ ЗА ПРЕБАРУВАЊЕ НИЗ КОЛЕКЦИИ

Пребарување низ колекција

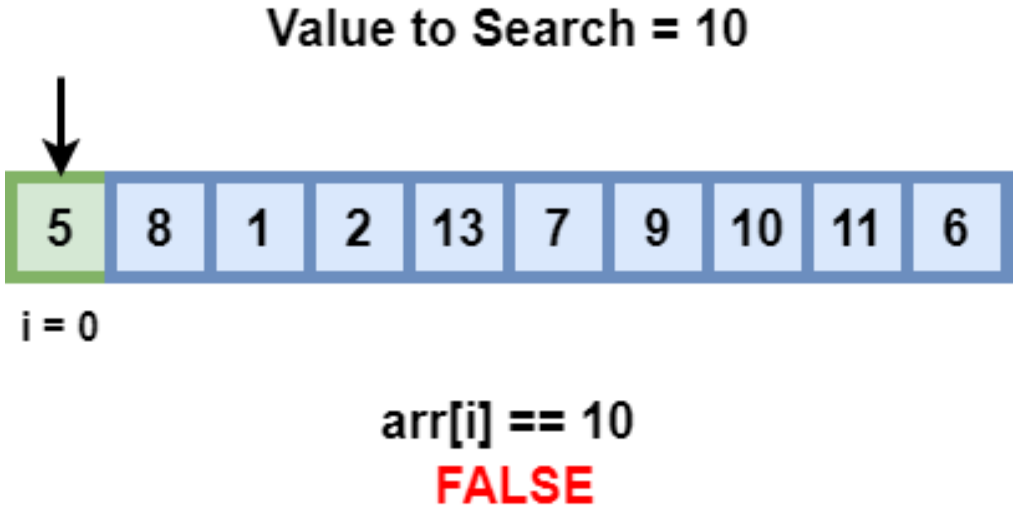
- Колекција на записи со клучни полиња или клучеви
- Начин на чување
  - Подредено
  - Неподредено
- Пребарување на вредност на клуч во подредена колекција
  - Во терминилогија на низи: наоѓање на елемент со дадена вредност во подредена низа

Секвенцијално пребарување

- Пребарување кое започнува од почетокот
- Се испитува секој следен елемент
- Се додека не се најде соодветна вредност или додека не се дојде до крајот

Секвенцијално пребарување - пример

```
def sequential_serach(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
```



- Сложеност: O(n)

А може ли подобро од ова?

Бинарно пребарување

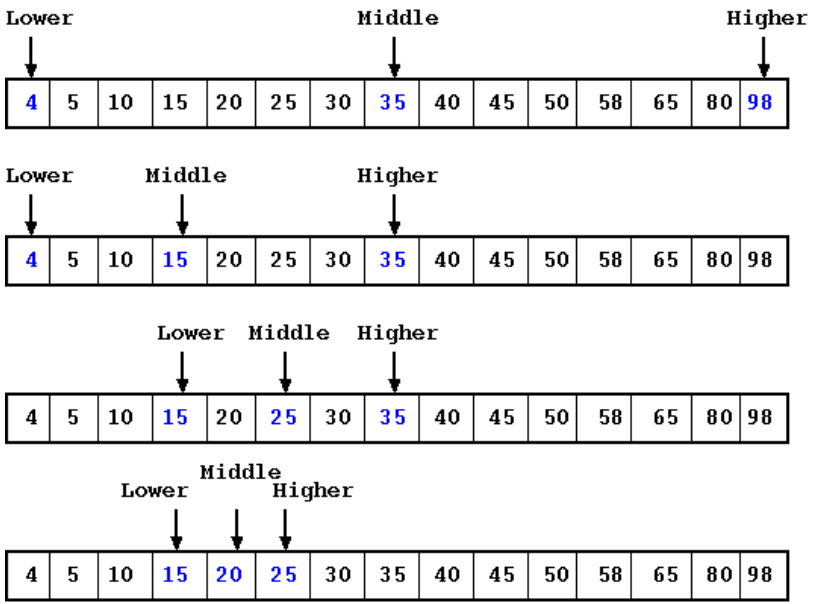
- Претходниот алгоритам работи и на подредени и на неподредени колекции
- Дали подреденоста може да помогне во побрзо пронаоѓање?
- Бинарно пребарување (логаритамско пребарување) – овозможува пребарување на сортирани низи со многу подобри време на извршување:
  - O(Log n) - средно и најлошо
  - O(1) - најдобро

Бинарно пребарување - алгоритам

- Клучот кој го бараме К, го споредуваме со клучот кој се наоѓа на средината на низата K_m , каде m= n/2. Можни се три случаи:
  - (i) Ако $K < K_m$, тогаш доколку клучот постои во низата тоа е делот од низата со помали индекси;
  - (ii) Ако $K = K_m$, тогаш клучот е пронајден;
  - (iii) Ако $K > K_m$, тогаш доколку клучот постои во низата тоа е делот од низата со поголеми индекси;
- Предходните чекори се повторуваат се додека се најде клучот или не постои дел од низата за проверка.

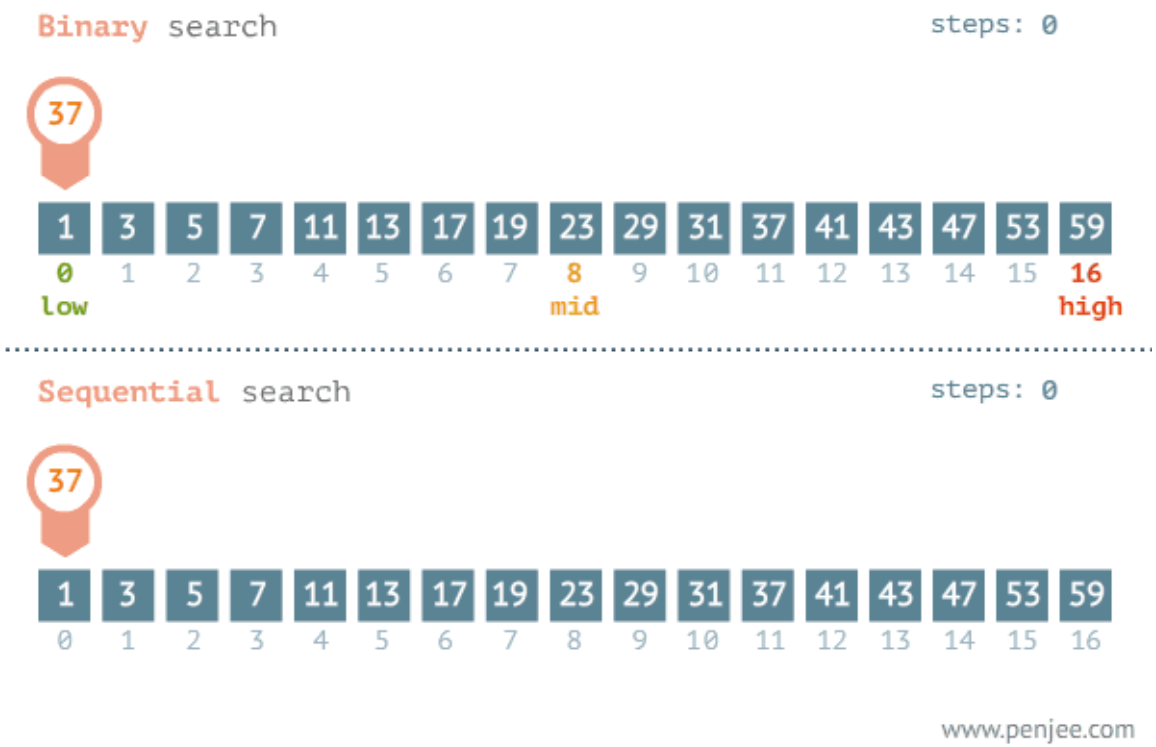
Бинарно пребарување

```
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
```



- Сложеност O(log n)

Споредба на секвенцијално и бинарно пребарување



СПОРЕДУВАЊЕ НА КОЛЕКЦИИ

Споредување на колекции

- Проблем – дали две низи ги содржат истите елементи?
- Споредување на неподредени колекции
  - За секој елемент од првата се проверува секвенцијално дали постои во втората
  - Сложеност $O(n^2)$
- Споредување на подредени колекции
  - Се споредуваат елемент по елемент дали ако се на иста позиција, имаат иста вреднот
  - Сложеност $O(n)$

Споредување на колекции - дискуција

- Препорака: Ако колекциите се неподредени, подобро е:
  - да се подредат (2 подредувања),
  - па да се споредуваат како подредени

- Сложеност

  $O(n\log_2 n) + O(n\log_2 n) + O(n) = O(2n\log_2 n+n) = O(n\log_2 n) < O(n^2)$

- Како да се подредат?

СОРТИРАЊЕ НА КОЛЕКЦИИ

Сортирање на колекции

- Зошто сортирање
- Од претходните примери
  - Сортираните колекции се пребаруваат побрзо
  - Сортираните колекции се споредуваат побрзо
- Алгоритми за сортирање
  - Внатрешно сортирање
  - Надворешно сортирање
- Сортирање со помош на споредба
- Сортирање во линеарно време

Илустрација на алгоритми

- Алгоритми за сортирање: https://visualgo.net/en/sorting

Сортирање со помош на споредби

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

Стабилно сортирање

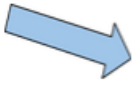
- При стабилно сортирање се задржува релативниот редослед на елементите со еднакви сортирачки клучеви.
- Се сортира по оцена (Grade)

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

Стабилно

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

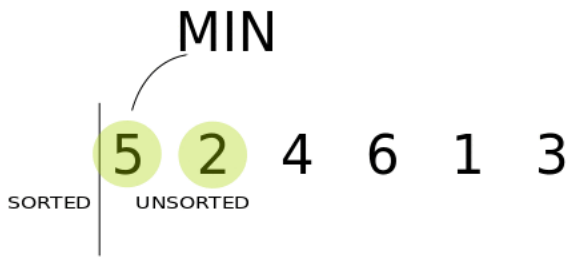


AFTER	
Name	**Grade**
Greg	A
Harry	A
Gill	B
Fabian	B
Earl	B
Dave	C

Нестабилно

MAXIMUM ENTRY (SELECTION) SORT

- Наједноставен пристап
  - Најди го најголемиот (најмалиот) елемент и стави го на последно (прво) место во новата низа
  - За заштеда на простор, наместо нова низа, замени го елементот кој е на последно (прво) место со наголемиот (најмалиот) елемент во низата
  - Продолжи ја постапката со поднизата составена останатите елементи без првиот (последниот) елемент



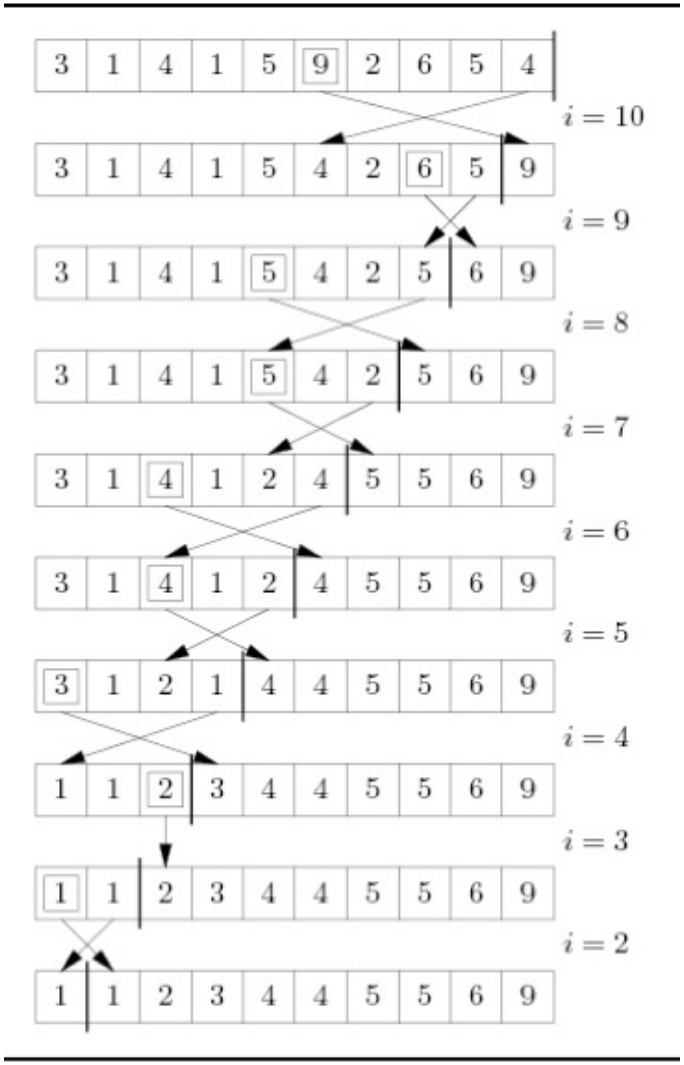
MAXIMUM ENTRY (SELECTION) SORT

```
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

- Сложеност
  - n-1 замена
  - (n-1)+(n-2)+…+2+1 споредби
  - $O(n^2)$
  - O(1) екстра простор
  - $\Theta(n^2)$ споредби
  - Θ(n) замени
  - Не е стабилен за различни типови на влезни низи
    - за случајни
    - речиси сортирани
    - обратно сортирани
    - мал број на уникатни вредности

MAXIMUM ENTRY (SELECTION) SORT



INSERTION SORT

- Идејата на алгоритмот: вметнување на запис со даден клуч во претходно сортирана подниза
- Обично е побрз и поедноставен од Bubble и Selection сорт

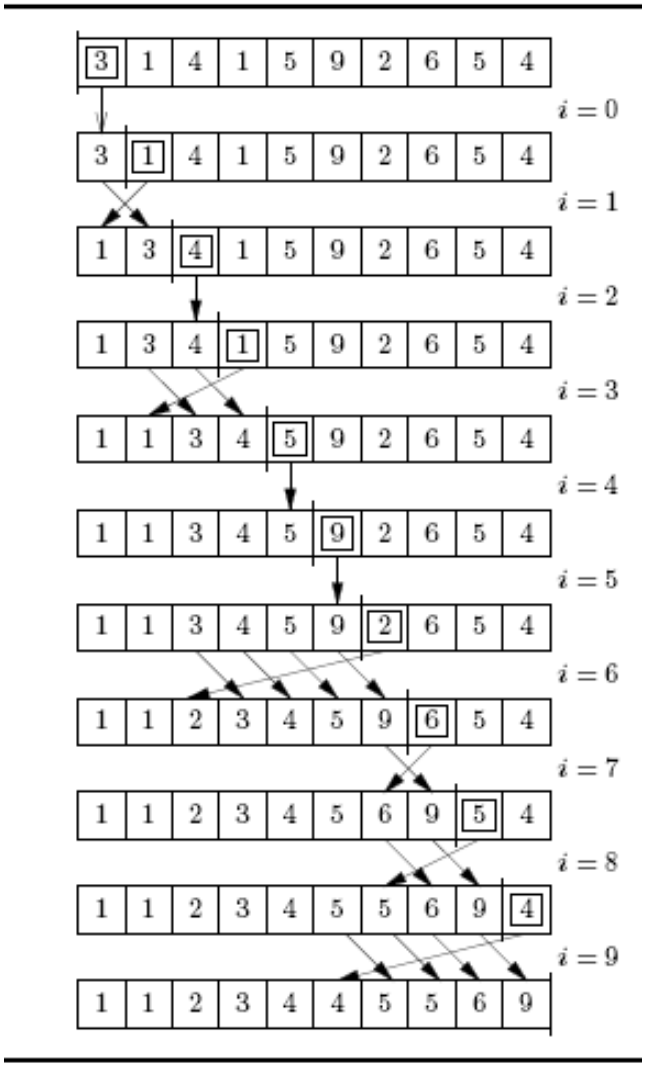
INSERTION SORT

```
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

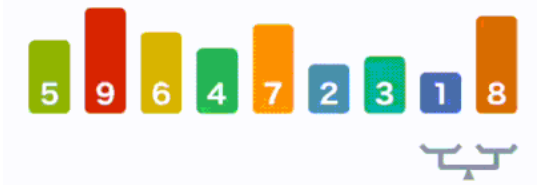
- Сложеност:
  - Стабилен
  - O(1) екстра простор
  - $O(n^2)$ споредби и замени
  - Прилагодлив: O(n) време кога има речиси сортиран влез

INSERTION SORT



BUBBLE SORT

- Работи на принцип на испливување на најлесните на површината, односно тонење на најтешките на дното
- Едноставен за реализација
- Повеќекратно поминување низ низата
- Во секое поминување, се заменуваат соседни елементи ако не се подредени (помал пред поголем)

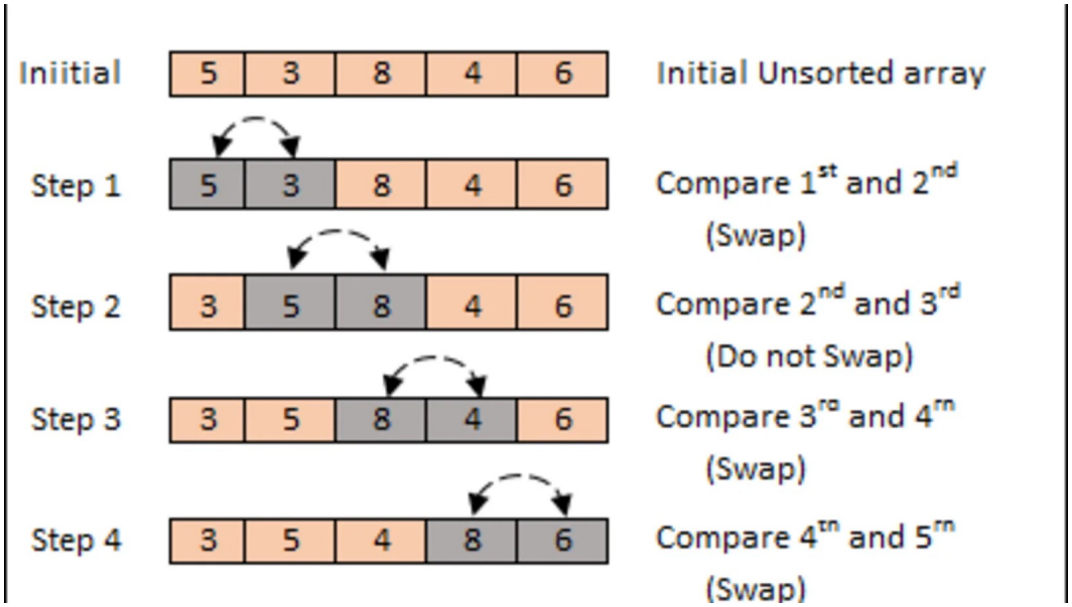


BUBBLE SORT

```
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
```

- Тривијален случај: подредена низа
- Лесен за имплементација, и мемориски ефикасен
- Стабилен
- Сложеност: $O(n^2)$

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 3ra and 4rn
(Swap)
Step 4
3 5 4 8 6
Compare 4tn and 5rn
(Swap)

QUICKSORT

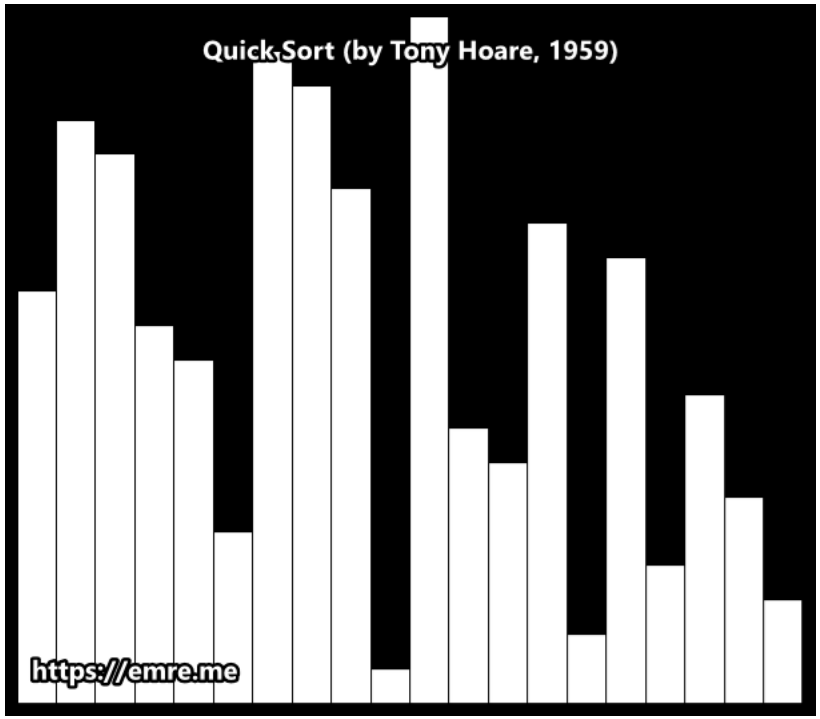
- Има најдобро време на извршување во просечен случај
- Идејата е да се најде запис кој ќе се позиционира на вистинското место во низата
- Потоа, сите помали од него треба да се преместат пред него, а сите поголеми, после него
- Користи раздели-па-владеј техника

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

- Сложеност: $O(n^2)$ најлош случај
- Сложеност: ако добро се избере контролниот клуч: O(n log n)
- O(lg(n)) екстра простор
- Не прилагодлив, не стабилен

- Избор на пивот:
  - Првиот
  - Последниот
  - Случаен
  - Средниот



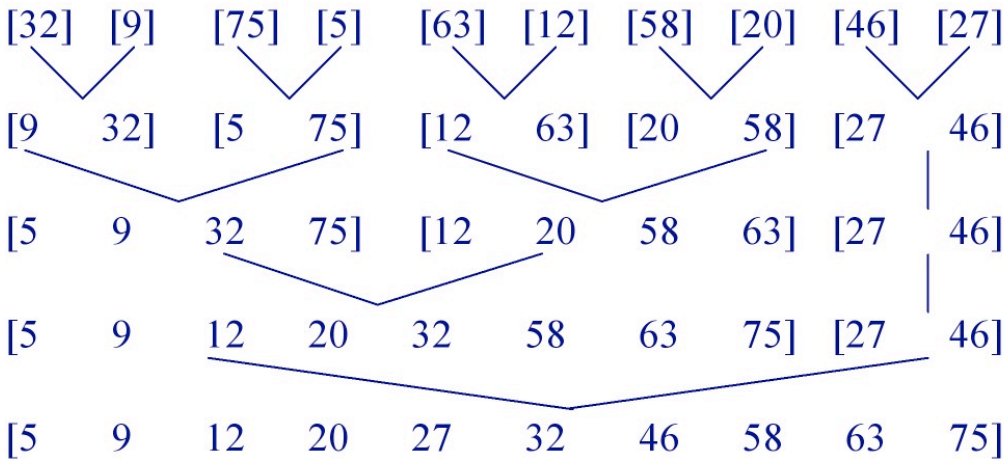
QUICKSORT

```
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

- Базиран на идејата за спојување на две веќе сортирани низи
  - Се започнува со споредба на првите елементи од двете низи. При тоа, во резултантната се става помалиот од нив, а во низата од која е изваден тој елемент сега се разгледува вториот елемент
  - Кога со првиот чекор ќе се исцрпат елементите од едната низа, елементите од другата се додаваат еден по еден по резултантната низа

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 – спојување на сортирани листи

```
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

```
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

Споредба на алгоритмите

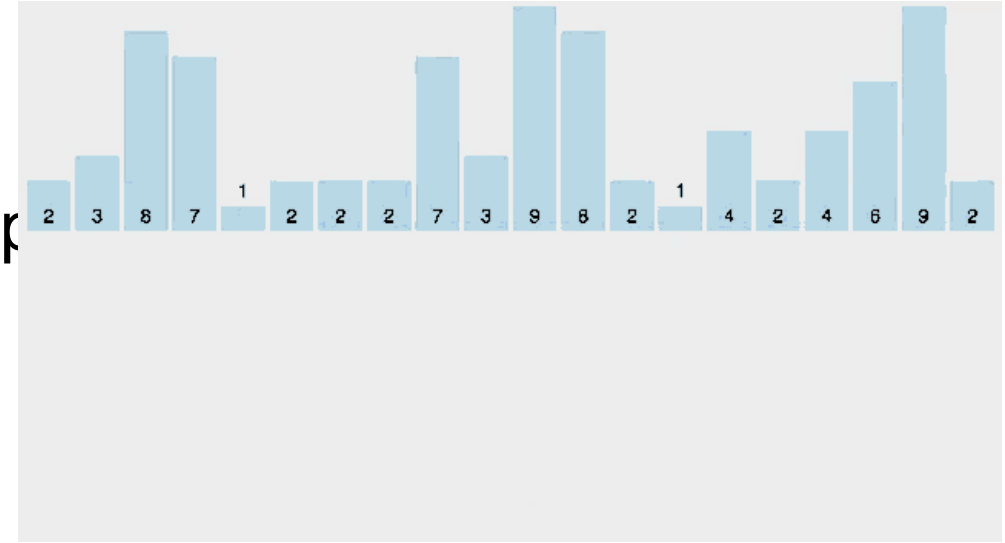
Growth Rate to Srot N Items					
Alorithm	Stability	Inplace	Running Time	Extra Space	Note
Insertion	Yes	Yes	Between n and n^2	1	Depends on the order of the input key
Selection	No	Yes	n^2	1	
Bubble	Yes	Yes	n^2	1	
Merge	Yes	No	nlogn	n	
Heap	No	Yes	nlogn	n	
Quick	No	Yes	nlogn	logn	Probabilistic

Сортирање во линеарно време

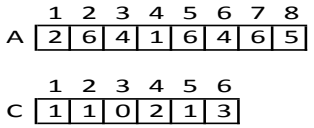
- Под одредени услови, можно е да се направи сортирање во линеарно време
  - Клучевите кои се сортираат се во интервал 1...к
- Сортирање со броење
- Bucket sort
- Radix sort

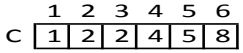
Сортирање со броење

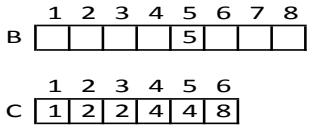
- За секоја различна вредност на клучот, се бројат неговите појавувања во колекцијата
- Сложеност: O(k+n), пр
  k<<n, O(n)

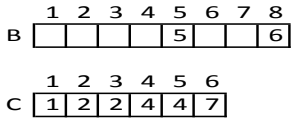


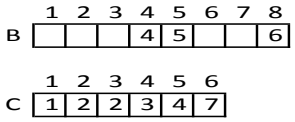
Сортирање со броење

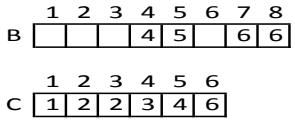


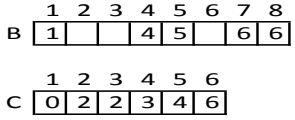


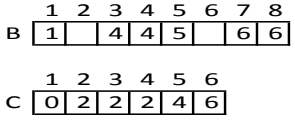


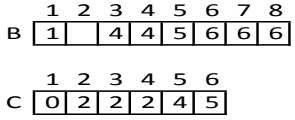


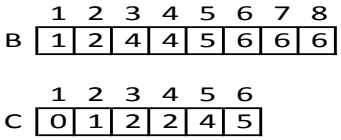












A – почетна низа
B – конечна низа
C – помошна низа за броење

Сортирање со броење

```
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
```

Други алгоритми во линеарно време

- Bucket sort – секој елемент се поистоветува со одредена “кофичка”, потоа се бројат колку има во секоја кофичка
  - Варијација на алгоритмот со броење
- Radix sort – користење на карактиртиките на претставувањето на бројот во броен систем за сортирање
  - Броење по стотки, па десетки, па единици
  - Броење според дел од бројот во бинарна репрезентација