Sorting Techniques

Criteria for Analysis

1. Number of comparisons - decides time complexity
2. Number of swaps
3. Adaptive - if a method is taking minimum time over a sorted list
4. Stable
5. Extra Memory

Explanation of stable:

Name:	A	B	C	D	E	F	G
Mark:	5	8	6	4	6	7	10

The list is already sorted according to the names

If a sorting algorithm is preserving the order of duplicate elements in the sorted list, then that algorithm is stable

Name:	D	A	C	E	F	B	G
Mark:	4	5	6	6	7	8	10

} arranged in a stable way according to marks

If there are duplicate marks at least names must be in sorted order.

1. Bubble
2. Insertion
3. Selection

} $O(n^2)$ comparison based sorts

4. Heap Sort
5. Merge Sort
6. Quick Sort
7. Tree Sort

} $\rightarrow n\log)$

8. Shell Sort $\rightarrow O(n^{3/2})$

Index based sorts

9. Count Sort
10. Bucket / Bin Sort
11. Radix Sort

} O(n

Bubble Soft

A | 8 | 5 | 7 | 3 | 2 | , n = 5

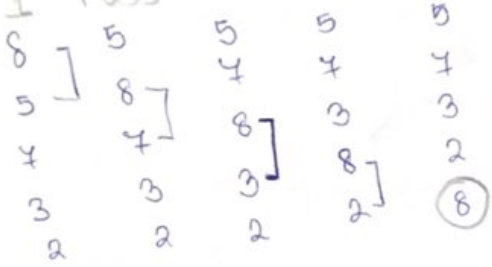
0 1 2 3 4

pass - when all elements are compared once (we're going through the list)

total number of el - 5

we have compared 4 possible pairs

4 - swaps

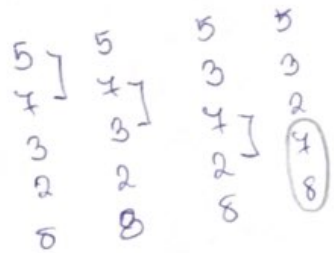


→ largest el. is sorted

2nd Pass

3 comparisons

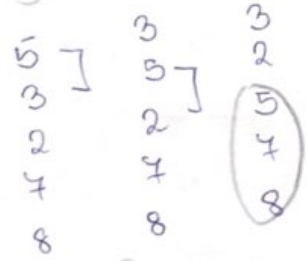
2 swaps are performed, but the max. nº 3



3rd Pass

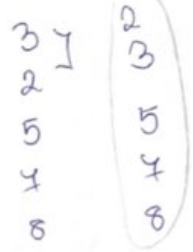
2 comparisons

max Nº of swaps = 2



4th Pass

1 comp., 1 swap



Analysis: Nº of passes: 4. For n-elem.: (n-1) passes

Nº comparisons: $1+2+3+\ldots \rightarrow 1+2+3\ldots+(n-1) = \frac{n(n-1)}{2} = \Theta(n^2)$ - time taken

Nº swaps (max:) $\rightarrow 1+2+3+\ldots+(n-1) = \frac{n(n-1)}{2} = \Theta(n^2)$

```
void BubbleSort(int A[], int n)
{
    int isSorted;

    for(i=0 ; i<n-1; i++) // for Nº of passes
    {
      isSorted = false;
      for( j=0 ; j< n-1-i ; j++)
      {
        if (A[j] > A[j+1])
        {
          swap(A[j], A[j+1])
          isSorted = true // there's a swap done
        }
      }
      if isSorted == false
      {
        return; // break;
      }
}
```

Checking for adaptiveness: If the array is already sorted:

Nº comp: n-1

swap = 0

$\Theta(n)$

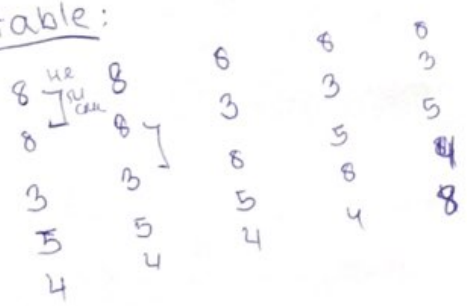
Bubble Sort Time:

min $\Omega(n)$ - if the list is already sorted

max $\Omega(n^2)$

Bubble Sort is adaptive

Stable:



→ so it is stable

Comparison between Bubble & Insertion Sort

	Bubble Sort	Insersion sort	
min comp	$\Theta(n)$	$\Theta(n)$	Already Asc
max comp	$\Theta(n^2)$	$\Theta(n^2)$	Descendi
min swap	$O(1)$	$O(1)$	ascendi
max swap	$\Theta(n^2)$	$\Theta(n^2)$	descendi
Adaptive	✓	✓	
Stable	✓	✓	
Inversion dist	NO	yes	
K passes	yes	NO	

Selection Sort (Сортиране чрез пряк избор)

A | 8 | 6 | 3 | 2 | 5 | 4 |

0 1 2 3 4 5

1st pass

index	A	sorted (smallest)
0 (k →, j →, ← i)	8	(2)
1 (j →)	6	6
2	3	3
3 (k ↘)	2	8
4	5	5
5	4	4

5 comp.
swap 1

IInd pass

index	A	result
0	2	2
1 (← i, j, k)	6	3
2	3	2
3	8	6
4	5	8
5	4	5
		4

4 comp.
1

Взимаме 2 pointer-а, местим j на позиция 1 и проверяваме дали елементът, към който j сочи е по-малък от този на k, да, значи караме и k при j. J отива до 2, по-малък ли е елементът на позиция 2 от този, към който k сочи, да, слагаме k до j.

Още един път същото k=3.

:

j=4 по-малък ли е от k, не, не нека си стои

IIIrd

2
3
i, j, k → 6
8
5
4

→

2
3
4
8
5
6

3 com
1

IVth

2
3
4
8
5
6

2 co
1 и

Vth pass

1 comp.
1

No of comp: $1+2+3 \ldots +(n-1) = \frac{n(n-1)}{2} = \Theta(n^2)$ -time

No swaps → $n-1 \rightarrow \Theta(n)$

```
void selectionSort(int A[], int n)
{
    for (i=0; i<n-1; i++) //no. of passes
    {
        for(j=k=i; j<n; j++)
        {
            if(A[j] < A[k])
                k=j;
        }
        swap(A[i], A[k])
    }
```

n=10

	A
i → 0	8
1	6
2	3
3	10
4	9
5	4
6	12
7	5
8	2
9	7

Analysis: Adaptive - not, Binary $\Theta(n^2)$ -time

Stable - not

Merging

1. Merging two arrays - process of combining 2 sorted lists into a single sorted list

m — A

i	A
i → 0	2
1	10
2	18
3	20
4	23

n — B

j	B
j → 0	4
1	9
2	19
3	25

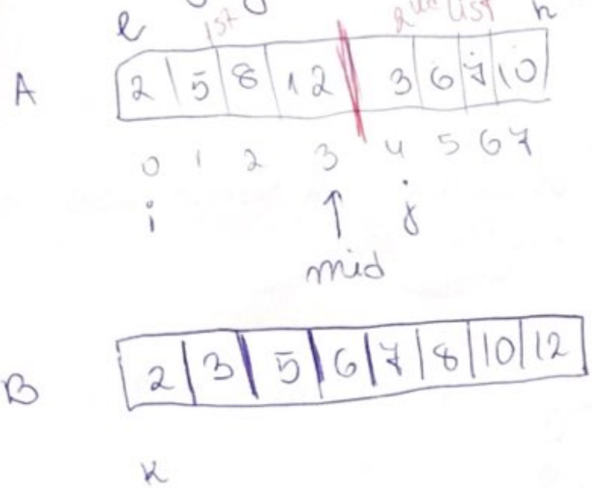
```
void merge(int A[], int B[], int m, int n)
{   int i:0
    int j=0
    int k:0

while (i<m && j<n)
{       if(A[i]<B[j])
           C[k++] = A[i++];
        else
           C[k++] = B[j++]

}
 for (i; i<m; i++) // i започва от там, където си спре
     C[k++] = A[i];
 for( j; j<n; j++)
     C[k++] = B[j];

}
```

C

k	C
k → 0	2
1	4
2	9
3	10
4	18
5	19
6	20
7	23
8	25

Time: $\Theta(m+n)$

Merging 2 list in a single array




```
void Merge (int A[], int l, int mid, h)
{   int i=l
    int j=mid+1
    int =l (загубмо можем [illegible] в масив)
    int B[h+1];
while (i<=mid && j<=h)
{   if(A[i] <A [j])
      B [k++]=A [i++]
    else
      B[k++]=A[j++]
for ( ; i<=mid; i++)
      B[k++]= A [i];
for ( ; i<=h ,j++)
      B[k++] = A[j]
```

Merging

-> 2-way merging

Iterative MergeSort

A | 8 | 3 | 7 | 4 | 9 | 2 | 6 | 5 |
0 1 2 3 4 5 6 7

- we can think it as every element is a list, so there's an array containing 8 lists, where each list contains 1 element

1st pass: 3 8   4 7   2 9   (5 6) -> we look them as lists   n-merged
p=2   l m h   l m h

2nd pass: 3 4 7 8   2 5 6 9 - n merged
p=4   l m h

3rd pass: 2 3 4 5 6 7 8
p=8   n elem. are merged

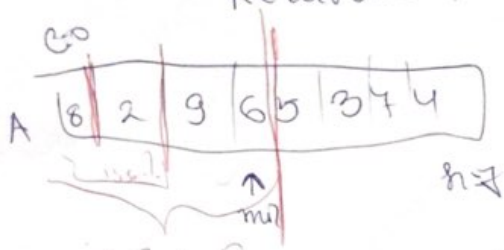
Analysis: log n - times (time)

$\log_2 8 = 3$ | time comp $\Theta(n \log n)$

It looks as a tree if we look it from upside down.

```
void IMergeSort(int A[], int n)
{ int p,i,l,mid,h
  for (p=2; p<=n, p=p*2) //for passes
     for (i=0; i+p-1<n, i=i+p)
     { l=i
       h=i+p-1
       mid = ⌊(l+h)/2⌋
       merge(A, l, mid, h);
     }
}  if(p/2<n) -> Merge(A, 0, p/2, n-1)
```

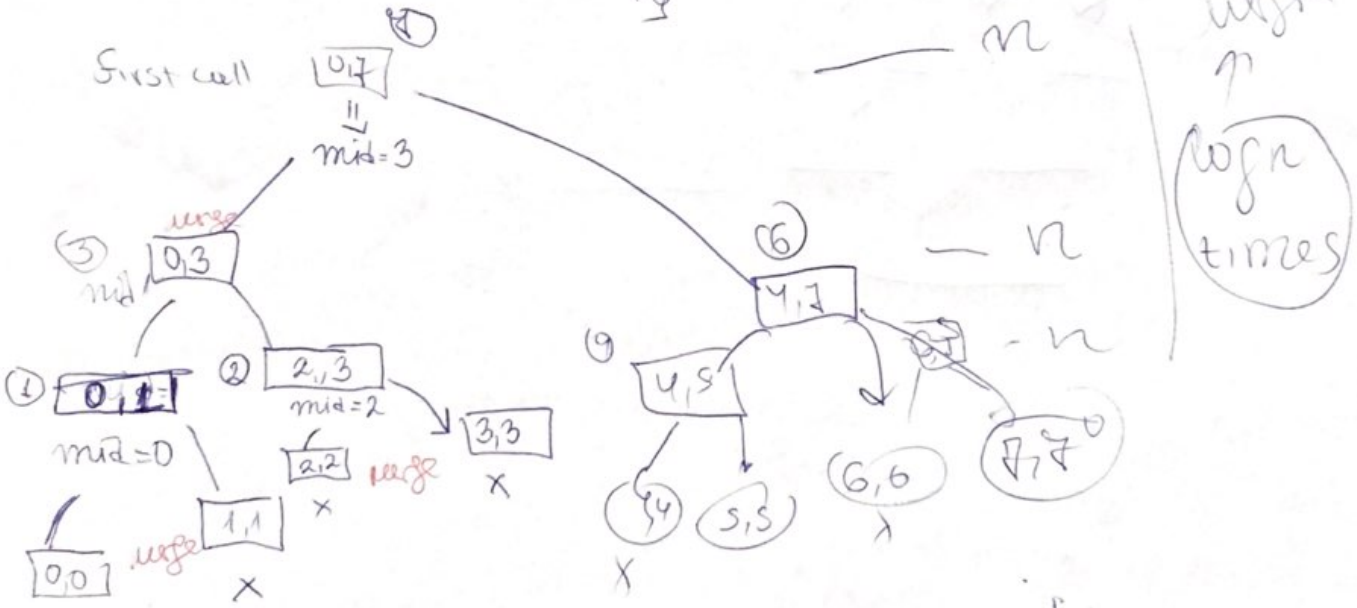
Recursive MergeSort



2,8 6,9 3,5 4,7

2,6,8,9 3,4,5,7

```
void MergeSort(int A[], int l, int h)
{
  if(l<h)
  {
    1. mid=[(l+h)/2]
    2. MergeSort(A, l, mid)
    3. MergeSort(A, mid+1, h)
    4. Merge(A, l, mid, h)
  }
}
```



Analysis: O(nlogn) - time complexity

space complexity: A-n
B-n

requires extra space

size of stack - log n

2n + log n → n + log n