Problem Definition

Input:
- A sequence of \( n \) elements \( <a_1, a_2, ..., a_n> \)

Output:
- A permutation \( <a'_1, a'_2, ..., a'_n> \) of such elements, so that \( a'_1 \leq a'_2 \leq ... \leq a'_n \)
Types of Ordering

- Internal Ordering
  - All the elements to be ordered are in main memory
  - Direct access to all elements

- External Ordering
  - Elements cannot be loaded all in memory at the same time
  - It is necessary to act on elements stored on a file
  - Usually, sequential access

Practical observations

- Elements to be ordered are usually structures (struct) made of many variables (fields)
- The key of such structure is usually one field (or a value calculated from one or more fields)
- Remaining fields are additional data but useless for ordering
- Ordering is made for increasing values of the key
Example

```c
struct student {
    int id;
    char surname[30];
    char name[30];
    int grade;
} ;

struct student class[100] ;
```

Example

```c
struct student {
    int id;
    char surname[30];
    char name[30];
    int grade;
} ;

struct student class[100] ;
```

Ordering by id

Ordering by name and surname (key = concatenation name and surname)

Ordering by grade (repeated values)
Stability

A sorting algorithm is called *stable* whenever, even if there are elements with the same value of the key, in the resulting sequence such elements appear in the *same order* in which they appeared in the initial sequence.

Simple Assumption

During the study of sorting algorithms there are usually arrays of $n$ integer values:

```c
int A[n];
```
Algorithms

There are many sorting algorithms with different complexity:

- **O(n²)**: simple, iterative
  - Insertion sort, Selection sort, Bubble sort, ...

- **O(n)**: only applicable in particular cases
  - Counting sort, Radix sort, Bin (or Bucket) sort, ...

- **O(n log n)**: more complex, recursive
  - Merge sort, Quicksort, Heapsort

Insertion sort

Already sorted

![Insertion sort diagram](image-url)

Not considered yet

Move forward all the elements so that v[I] > v[j]
Pseudo-code

**Insertion-Sort**

1. for $j \leftarrow 2$ to length[$A$]
2. do
3.   $key \leftarrow A[j]$
4.   $i \leftarrow j - 1$
5.   while $i > 0$ AND $A[i] > key$
6.   do
7.     $A[i+1] \leftarrow A[i]$
8.     $i \leftarrow i - 1$
9.   end while
10. $A[i+1] \leftarrow key$

Implementation in C

```c
void InsertionSort(int A[], int n)
{
    int i, j, key;
    for(j=1; j<n; j++) {
        key = A[j];
        i = j - 1;
        while ( i >= 0 && A[i] > key ) {
            A[i+1] = A[i];
            i--;
        }
        A[i+1] = key;
    }
}
```
From pseudo-code to C

Note well:
- In C, array indexes are from 0 to n-1, while pseudo-code use ranges from 1 to n.
- Indentation of code is useful but remember braces to identify blocks { ... }

Complexity

Number of comparisons:
- $C_{\text{min}} = n-1$
- $C_{\text{avg}} = \frac{1}{4}(n^2+n-2)$
- $C_{\text{max}} = \frac{1}{2}(n^2+n)-1$

Number of data-copies
- $M_{\text{min}} = 2(n-1)$
- $M_{\text{avg}} = \frac{1}{4}(n^2+9n-10)$
- $M_{\text{max}} = \frac{1}{2}(n^2+3n-4)$

Best case: array already ordered
Worst case: array ordered inversely

$C = O(n^2)$, $M = O(n^2)$
$T(n) = O(n^2)$
$T(n) \text{ non è } \Theta(n^2)$
$T_{\text{worst case}}(n) = \Theta(n^2)$
Other quadratic algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Min</th>
<th>Average</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>( C = n - 1 )</td>
<td>( (n^2 + n - 2)/4 )</td>
<td>( (n^2 - n)/2 - 1 )</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>( C = (n^2 - n)/2 )</td>
<td>( (n^2 - n)/2 )</td>
<td>( (n^2 - n)/2 )</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>( C = (n^2 - n)/2 )</td>
<td>( (n^2 - n)/2 )</td>
<td>( (n^2 - n)/2 )</td>
</tr>
</tbody>
</table>

Execution Time (ms)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Ordered</th>
<th>Random</th>
<th>Inversely Ordered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Insertion</td>
<td>12</td>
<td>23</td>
<td>144</td>
</tr>
<tr>
<td>Binary Insertion</td>
<td>56</td>
<td>125</td>
<td>1327</td>
</tr>
<tr>
<td>Direct Selection</td>
<td>489</td>
<td>1907</td>
<td>695</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>540</td>
<td>2165</td>
<td>1492</td>
</tr>
<tr>
<td>Bubble sort with change notification</td>
<td>5</td>
<td>8</td>
<td>5931</td>
</tr>
<tr>
<td>Shaker sort</td>
<td>5</td>
<td>9</td>
<td>1619</td>
</tr>
<tr>
<td>Shell sort</td>
<td>58</td>
<td>116</td>
<td>157</td>
</tr>
<tr>
<td>Heap sort</td>
<td>116</td>
<td>235</td>
<td>492</td>
</tr>
<tr>
<td>Quick sort</td>
<td>31</td>
<td>69</td>
<td>226</td>
</tr>
<tr>
<td>Merge</td>
<td>99</td>
<td>234</td>
<td>99</td>
</tr>
</tbody>
</table>

\( n = 256 \) | 512 | 256 | 512 | 256 | 512
Impact of data

<table>
<thead>
<tr>
<th></th>
<th>Ordinati</th>
<th>Disordinati</th>
<th>Inversamente Ordinati</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inserimento diretto</td>
<td>12</td>
<td>46</td>
<td>366</td>
</tr>
<tr>
<td>Inserimento binario</td>
<td>56</td>
<td>76</td>
<td>373</td>
</tr>
<tr>
<td>Selezione diretta</td>
<td>489</td>
<td>547</td>
<td>509</td>
</tr>
<tr>
<td>Bubblesort</td>
<td>540</td>
<td>610</td>
<td>1026</td>
</tr>
<tr>
<td>Bubblesort con segnalatore</td>
<td>5</td>
<td>5</td>
<td>1104</td>
</tr>
<tr>
<td>di scambio</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shakersort</td>
<td>5</td>
<td>5</td>
<td>961</td>
</tr>
<tr>
<td>Shellsort</td>
<td>58</td>
<td>186</td>
<td>961</td>
</tr>
<tr>
<td>Heapsort</td>
<td>116</td>
<td>264</td>
<td>246</td>
</tr>
<tr>
<td>Quicksort</td>
<td>31</td>
<td>55</td>
<td>60</td>
</tr>
<tr>
<td>Fusione *</td>
<td>99</td>
<td>196</td>
<td>195</td>
</tr>
</tbody>
</table>

2 byte 2 byte 2 byte
16 byte 16 byte 16 byte

Counting sort

It cannot be applied in general, as it is based on this hypothesis:

- The $n$ elements to be ordered are integer numbers between 1 and $k$, with $k$ integer.

With such hypothesis, if $k = O(n)$, then the algorithm’s complexity is just $O(n)$. 
Basic Idea

Find out, for each element \( x \), how many elements of the array are less than \( x \).
Such information allows to put \( x \) directly in the final position in the array.

Data Structure

- Three arrays are needed:
  - Initial array: \( A[1..n] \)
  - Final array: \( B[1..n] \)
  - Temporary Array: \( C[1..k] \)
- Array \( C \) keeps track of number of elements of \( A \) having a certain value: \( C[i] \) is the number of elements of \( A \) equals to \( i \).
- Sum of the first \( i \) elements of \( C \) defines the number of elements of \( A \) whose values is \( \leq i \).
Pseudo-code

For each j, \( C[A[j]] \) represents the number of elements less than or equals to \( A[j] \), and then it is the final position of \( A[j] \) in \( B \):


The correction \( C[A[j]] \leftarrow C[A[j]] - 1 \) is needed to handle duplicate elements.
Example (n=8, k=6)

A
3 6 4 1 3 4 1 4
C
2 0 2 3 0 1
C
2 2 4 7 7 8

for \( j \leftarrow 1 \) to length[A]
    do \( C[A[j]] \leftarrow C[A[j]] + 1 \)

for \( i \leftarrow 2 \) to \( k \)
    do \( C[i] \leftarrow C[i] + C[i - 1] \)

for \( j \leftarrow length[A] \) downto 1
    do \( B[C[A[j]]] \leftarrow A[j] \)
        \( C[A[j]] \leftarrow C[A[j]] - 1 \)

---

Example (2)

A
3 6 4 1 3 4 1 4
B
1 1 1 1 1 1 1 1
B
1 1 1 1 1 1 1 1
B
1 1 1 1 1 1 1 1
B
1 1 1 1 1 1 1 1
B
1 1 1 1 1 1 1 1
B
1 1 1 1 1 1 1 1
B
1 1 1 1 1 1 1 1
B
1 1 1 1 1 1 1 1

for \( j \leftarrow length[A] \) downto 1
    do \( B[C[A[j]]] \leftarrow A[j] \)
        \( C[A[j]] \leftarrow C[A[j]] - 1 \)

j=8
C
2 2 4 6 7 8
j=7
C
1 2 4 6 7 8
j=6
C
1 2 4 5 7 8
j=5
C
1 2 3 5 7 8
j=4
C
0 2 3 5 7 8
j=3
C
0 2 3 4 7 8
j=2
C
0 2 2 4 7 7
j=1
Complexity

- 1-2: Initialization of C: $O(k)$
- 3-4: Calculate C: $O(n)$
- 6-7: Sum in C: $O(k)$
- 9-11: Copy in B: $O(n)$

Total complexity is $O(n+k)$.
Algorithm is useful only when $k=O(n)$, because the resulting complexity is $O(n)$.

Note

The condition of applicability of the algorithm can be extended in this way:
- The key field of $n$ elements to be ordered has a limited number of possible values $k$. 
Bubble Sort

- In each cycle compare every couple of consecutive elements and if they are not ordered, then swap (exchange) them.
- Repeat this process $N$ times and all the elements will be ordered
- Complexity is $O(n^2)$
- Optimization: if during last cycle there are no swaps, then the elements are already sorted

Bubble sort in C

```c
void BubbleSort(int A[], int n) {
    int i, j, t;
    for(i=1; i<n-1; i++) {
        for(j=1; j<n-1; j++) {
                t = A[j] ;
                A[j] = A[j+1];
                A[j+1] = t;
            }
        }
    }
}
```
Bubble sort (optimized) in C

```c
void BubbleSort2(int A[], int n) {
    int i, j, t, repeat = 1;
    while (repeat) {
        repeat = 0; /* if no swaps remains 0 -> exit while*/
        for (j = 1; j < n - 1; j++) {
            if (A[j] > A[j + 1]) {
                t = A[j]; /* swap elements*/
                A[j] = A[j + 1];
                A[j + 1] = t;
                repeat = 1;
            }
        }
    }
}
```