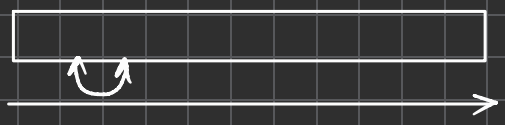
Lab 1 -> direct sorting methods

1) bubble sort

a



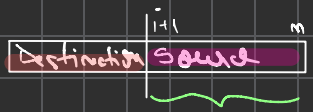
swap a[i] with a[i+1] if they are not in ascending order
=> after each iteration, the numbers after the last swap are in the correct place

=> we can store the last position to make the algorithm more efficient

best case scenario: it goes through the loop only once, checks if there are any swaps, doesn't find any and stops executing the loop => the array is already sorted => O(n)

worst case scenario: it executes the loop for $0 < poz < size$ times => $poz!$ => $O(n^2)$ => the array is in descending order
initially size-1 => (size-1)!

2) selection sort



extract should consider the smallest in S and insert it at position i+1

=> c is always size-i
=> $O(n^2)$

a is always incremented by 3 in the loop => size·3
=> $O(n)$

=> this is NO best or worst sol.

the algorithm is not stable

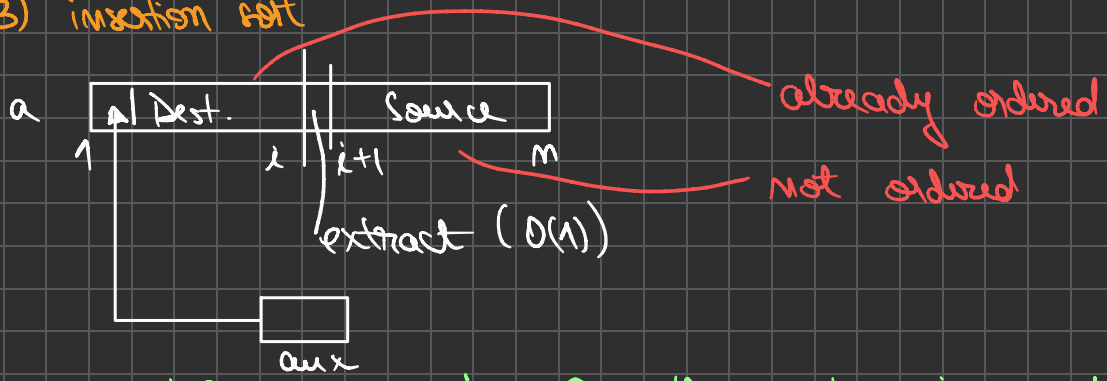
ex.: 19_1 19_2 1

=> 1 19_2 19_1

to make it more efficient, I used only the indexes

```
a = 0
c = 0
min_index = i
c++
[ if a[j] < a[min_index]
      min_index = j
swap(a[i], a[min_index])
a = a + 3
```

3) insertion sort



→ take the number from the position i+1 and insert it in the right position in the destination part

└ this means $a[i] < aux < a[i+1]$

└ do this with a buffer and shift the values as you go through the vector to find the position

if we use cond $aux < a[j]$ ⇒ stability

≡ strict inequality

we execute "extract" and "add" n(size) times

in each loop - in the worst case - c = i-1 ⇒ reverse ordered vector ⇒ $O(n^2)$

- $a = 1 + (i-1) + 1 = i+1$ ⇒ $O(n^2)$

- in the best case - c = 1 → ascending order already ⇒ O(n)

- $a = 1 + 0 + 1 = 2$ ⇒ O(n)

4) binary insertion sort

it's the same as insertion sort, just we find the position where the number should be placed through binary search

⇒ we can't use the aux and shift only one element at each step we need to move all the elements from position i (where to insert) and the position on which the element was before

the complexity of binary search ⇒ best case O(1)

└ the number is in the middle

worst case O(log n)

- item not found or found last

for inserting the number at the right position, we iterate through the vector and move them a position to the right

=> at each iteration i - indexWhere

=> best case : indexWhere = i => 1 -> O(n)
worst case : indexWhere = 0 => i => O(n²)

-> worst case O(n² log n), best case O(n)

not happening -> O(n log n)

because the element can't be both in the middle of the already sorted part and at the index i

the algorithm is not stable if there are duplicates

for making it stable, we need to do more steps (after finding the position where the number should be inserted, go to the right until the number is not equal anymore to the one at pos)