



Iterative sorting algorithms



Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino



Insertion sort

- Data: integers in array A
- Array partitioned in 2 sub-arrays
 - Left: sorted
 - Right: unsorted
- An array of just one element is sorted
- Incremental approach
 - At each step we expand the sorted sub-array by inserting one more element (invariance of the sorting property)
- Termination
 - All elements inserted in proper order



i-th step: sorted insertion

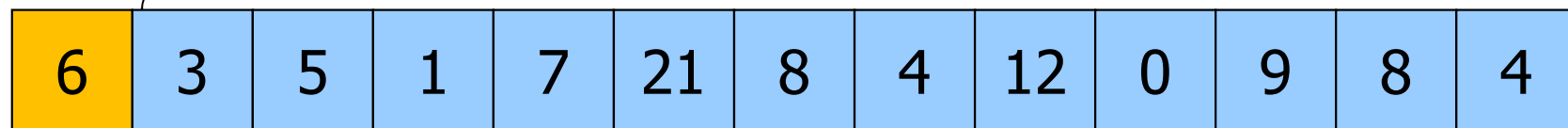
I-th step: put in the proper position $x = A_i$

- Scan the sorted subarray (from A_{i-1} to A_0) until we find $A_k > A_i$
- Right shift by one position of the elements from A_k to A_{i-1}
- Insert A_i in k-th position

Example

Already sorted

Not yet considered

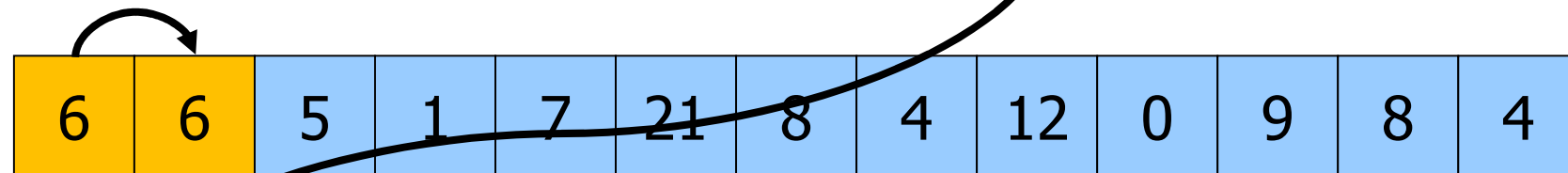


$i=1$

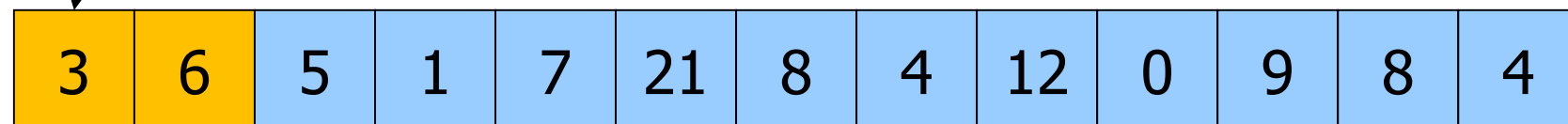
Element to insert x

3

$3 < 6$



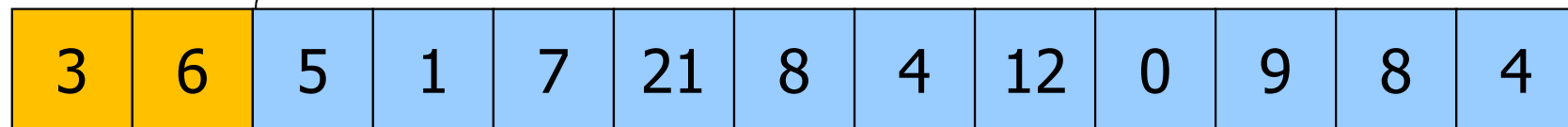
left boundary reached, insert 3



Example

Already sorted

Not yet considered

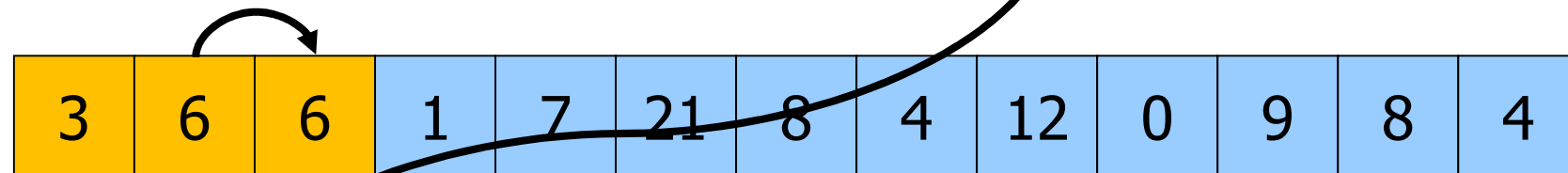


$i=2$

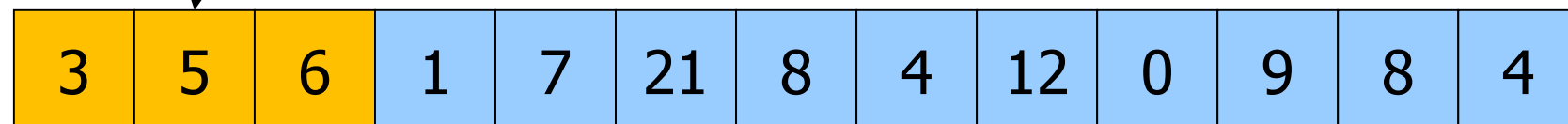
Element to insert x

5

$5 < 6$



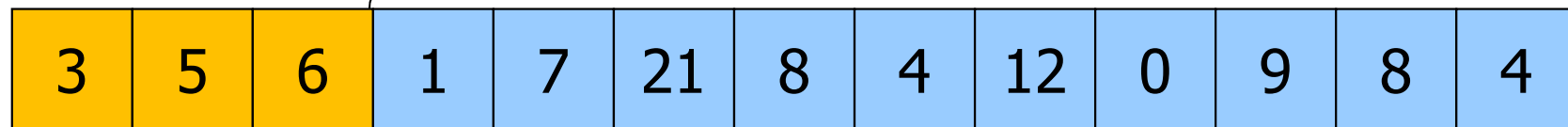
$5 > 3$, insert 5



Example

Already sorted

Not yet considered

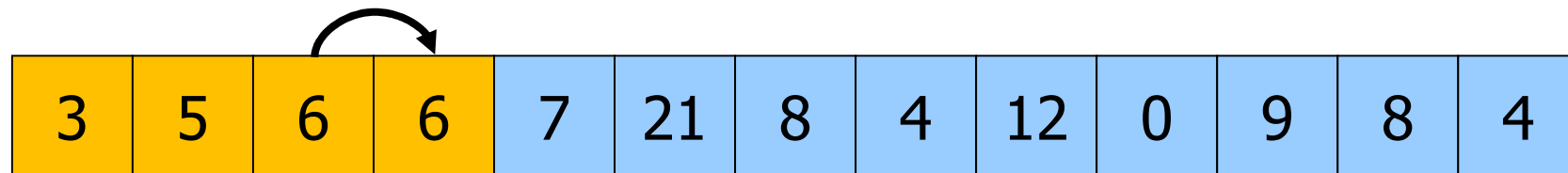


Element to insert x

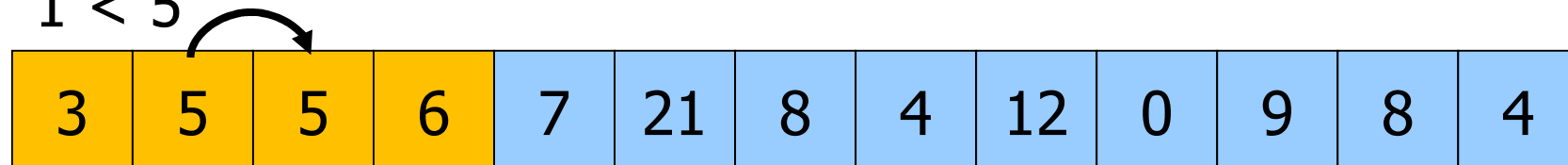
1

i=3

$1 < 6$



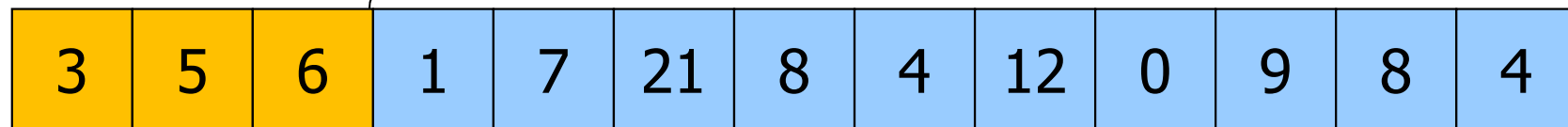
$1 < 5$



Example

Already sorted

Not yet considered



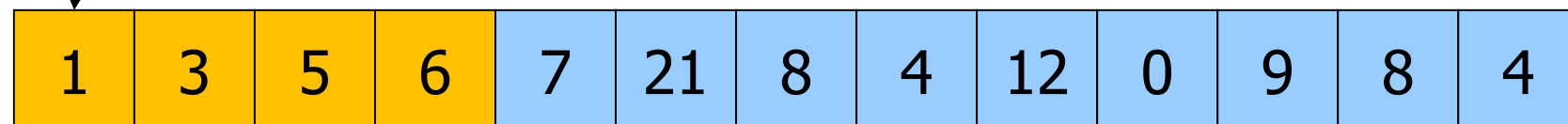
Element to insert x

1

$1 < 3$



left boundary reached, insert 1





C Code

```
void InsertionSort (int A[], int n) {  
    int i, j, x;  
  
    for (i=1; i<n; i++) {  
        x = A[i];  
        j = i - 1;  
        while (j>=0 && x<A[j]) {  
            A[j+1] = A[j];  
            j--;  
        }  
        A[j+1] = x;  
    }  
  
    return;  
}
```

It is also possible to reason
in terms of
l = index of the leftmost element
r = index of the rightmost element

Complexity Analysis

- Analytic analysis
 - Worst case
 - We assume unit cost for all operations

```
for (i=1; i<n; i++) {  
    x = A[i];  
    j = i - 1;  
    while (j>=0 && x<A[j]){  
        A[j+1] = A[j];  
        j--;  
    }  
    A[j+1] = x;  
}
```

#checks

#iterations

$$1 + (n-1+1) + (n-1)$$
$$n - 1$$
$$n - 1$$
$$\sum_{i=1}^{n-1} (i + 1)$$
$$\sum_{i=1}^{n-1} i$$
$$\sum_{i=1}^{n-1} i$$
$$n - 1$$

Complexity Analysis

$$\begin{array}{c}
 1 + (n-1+1) + (n-1) \\
 n - 1 \\
 n - 1 \\
 \sum_{i=1}^{n-1} (i + 1) \\
 \sum_{i=1}^{n-1} i \\
 \sum_{i=1}^{n-1} i \\
 n - 1
 \end{array}$$

Recalling that

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

$$\sum_{i=1}^{n-1} (i + 1) = \frac{n(n+1)}{2} - 1$$

$$\begin{aligned}
 T(n) &= \\
 &= 2n + (n-1) + (n-1) + \sum_{i=1}^{n-1} (i + 1) + \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} i + (n-1) \\
 &= 2n + 3(n-1) + \frac{n(n+1)}{2} - 1 + 2 \frac{(n-1)n}{2} \\
 &= 2n + 3n - 3 + \frac{1}{2}n^2 + \frac{1}{2}n - 1 + n^2 - n \\
 &= \frac{3}{2}n^2 + \frac{9}{2}n - 4 = \Theta(n^2)
 \end{aligned}$$

$T(n)$ grows quadratically

Worst case
 $O(n^2)$ overall



Complexity Analysis

- More intuitive analysis

- Two nested cycles
- Outer loop: $n-1$ executions
- Inner loop: the worst-case $\rightarrow i$ executions at the i -th iteration of the outer loop

- Complexity


- $T(N) = 1 + 2 + 3 + 4 + \dots + (n - 2) + (n - 1)$
- $T(N) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

$T(n)$ grows quadratically with n



Complexity Analysis

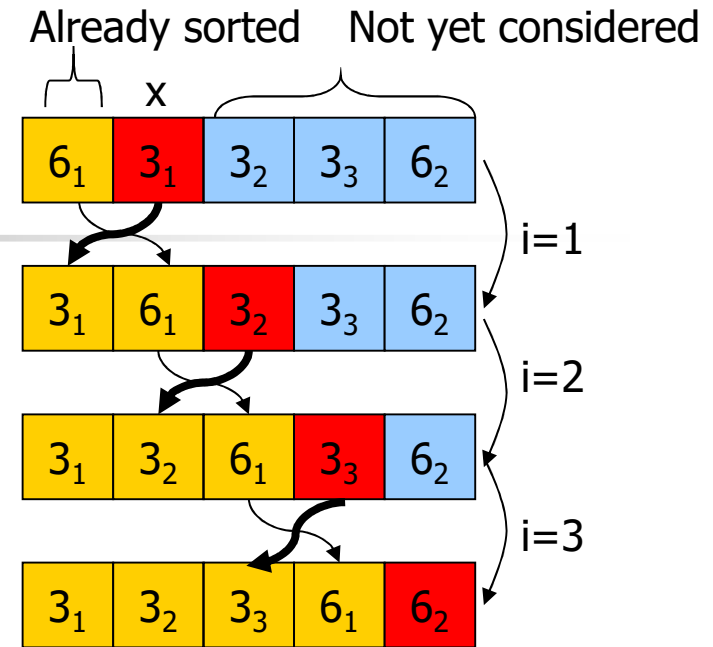
■ Best case scenario

- Inner loop: 1 execution at the i-th iteration of the outer loop
- Complexity
- $T(n) = 1 + 1 + 1 + \dots + 1 = n-1$


n-1 times
- T(n) grows linearly with n

Features

- In place
- Number of exchanges in worst-case
 - $O(n^2)$
- Number of comparisons in worst-case
 - $O(n^2)$
- Stable
 - If the element to insert is a duplicate key, it can't pass over to the left a preceding occurrence of the same key

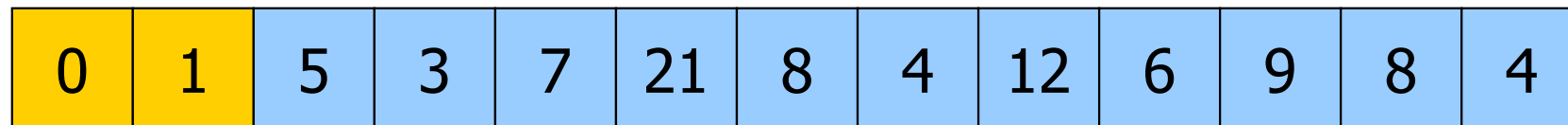
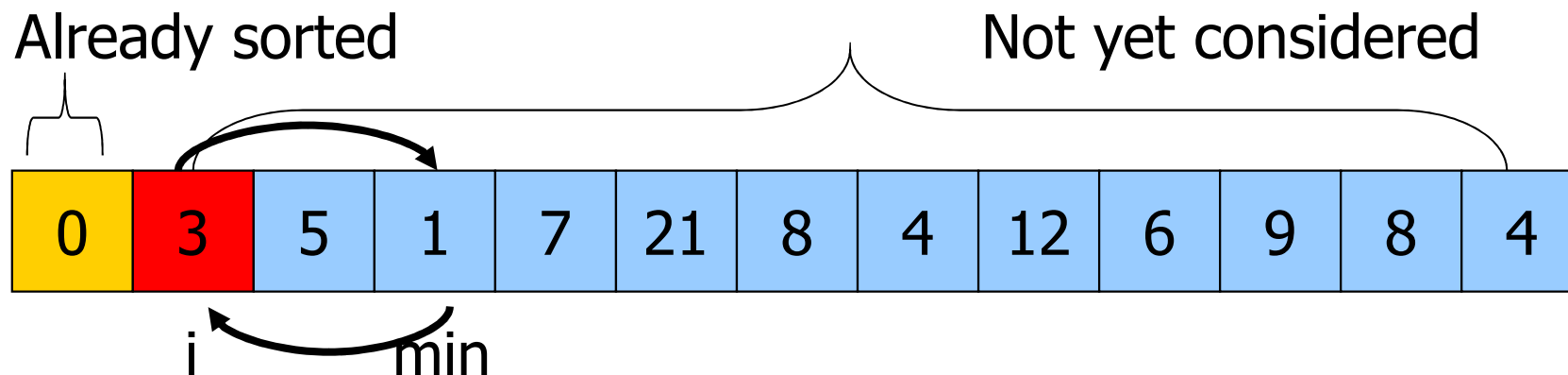
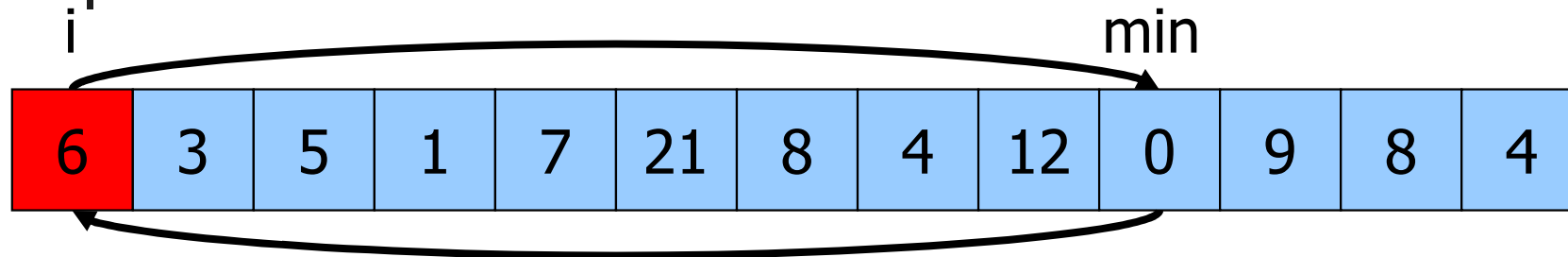




Selection sort

- Sort n integers in array A
- Array divided into two sub-arrays
 - Left: sorted, initially empty
 - Right: unsorted, initially it coincides with A
- Incremental approach
 - Iteration i : the minimum of the right sub-array ($A_i \dots A_r$) is assigned to $A[i]$; increment i
- Termination: all elements are inserted in the correct location
- Searching for the minimum in the right sub-array entails scanning the sub-array

Example





C Code

```
void selectionSort (int A[], int n) {
    int i, j, min, temp;

    for (i=0; i<n-1; i++) {
        min = i;
        for (j=i+1; j<n; j++) {
            if (A[j] < A[min]) {
                min = j;
            }
        }
        temp = A[i];
        A[i] = A[min];
        A[min] = temp;
    }

    return;
}
```


Complexity Analysis

■ Analytic analysis

- Worst case
- We assume unit cost for all operations

Number of operations

#checks

#iterations

```
for (i=0; i<n-1; i++) {  
    min = i;  
    for (j=i+1; j<n; j++) {  
        if (A[j] < A[min]) {  
            min = j;  
        }  
    }  
    temp = A[i];  
    A[i] = A[min];  
    A[min] = temp;  
}
```

$$1 + \frac{(n-1+1) + (n-1)}{n-1}$$

$$\sum_{i=0}^{n-2} (1 + (n - (i + 1) + 1) + (n - (i + 1)))$$

$$\sum_{i=0}^{n-2} (n - (i + 1))$$

$$\sum_{i=0}^{n-2} (n - (i + 1))$$

$$n-1$$

$$n-1$$

$$n-1$$

Complexity Analysis

Recalling that

$$\sum_{i=0}^{n-2} n = n(n-1)$$

$$\sum_{i=0}^{n-2} i = \frac{(n-2)(n-1)}{2}$$

$$\sum_{i=0}^{n-2} 1 = n-1$$

$$1 + \frac{(n-1+1) + (n-1)}{n-1}$$

$$\sum_{i=0}^{n-2} (1 + (n - (i+1) + 1) + (n - (i+1)))$$

$$\sum_{i=0}^{n-2} (n - (i+1))$$

$$\sum_{i=0}^{n-2} (n - (i+1))$$

$$n-1$$

$$n-1$$

$$n-1$$

$$T(n) =$$

$$= 2n + 4(n-1) + 2 \sum_{i=0}^{n-2} (n - i) + 2 \sum_{i=0}^{n-2} (n - i - 1)$$

$$= 6n - 4 + 4 \sum_{i=0}^{n-2} n - 4 \sum_{i=0}^{n-2} i - 2 \sum_{i=0}^{n-2} 1$$

$$= 6n - 4 + 4n(n-1) - 4 \frac{(n-2)(n-1)}{2} - 2(n-1)$$

$$= 6n - 4 + 4n^2 - 4n - 2n^2 + 6n - 4 - 2n + 2$$

$$= 2n^2 + 6n - 6 = \Theta(n^2)$$

$T(n)$ grows quadratically

Worst case
 $O(n^2)$ overall

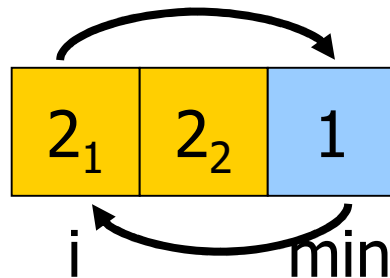


Worst-case asymptotic analysis

- More intuitive analysis
 - Two nested loops
 - Outer loop: executed $n-1$ times
 - Inner loop: at the i -th iteration executed $n-i-1$ times
 - $T(n) = (n-1) + (n-2) + \dots 2 + 1 = O(n^2)$
- Number of exchanges in worst-case $O(n)$
- Number of comparisons in worst-case $O(n^2)$

Features

- In place
- Not stable
 - A swap of "far away" elements may result in a duplicate key passing over to the left of a preceding instance of the same key





Exchange (Bubble) Sort

- Data: integers in array A delimited by left and right indices l and r
- Array divided in 2 sub-arrays
 - Right : sorted, initially empty
 - Left: unsorted, initially it coincides with A
- Elementary operation
 - Compare successive elements of the array $A[j]$ and $A[j+1]$, swap if $A[j] > A[j+1]$

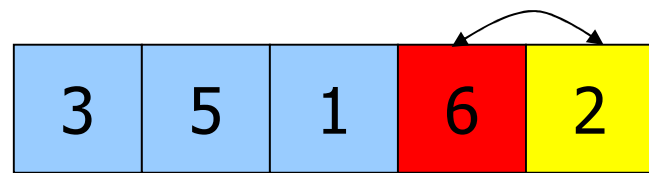
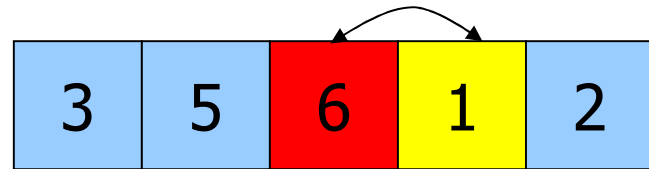
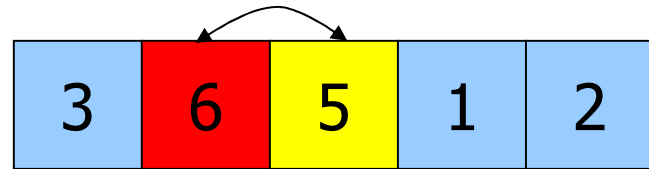


Exchange (Bubble) Sort

- Incremental approach
 - At iteration i the maximum of the left sub-array ($A_i \dots A_{r-i+1}$) is assigned to $A[r-i+1]$; increment i
 - The sorted right sub-array increases in size by 1 to the left, dually the left sub-array decreases in size by 1
- Termination: all elements are inserted in the correct location
- Possible optimization: flag to record that there have been swaps, early loop exit

Example

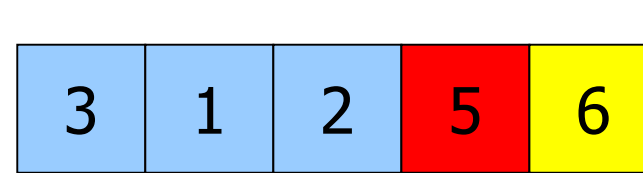
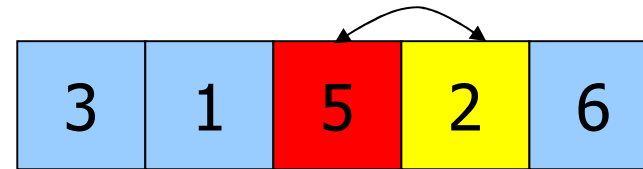
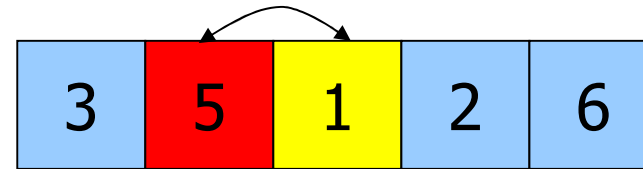
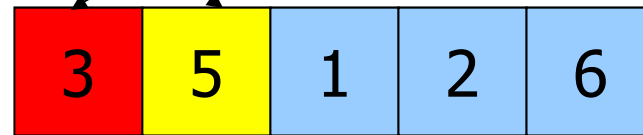
$i = 0$



unsorted sorted



$i = 1$



unsorted sorted

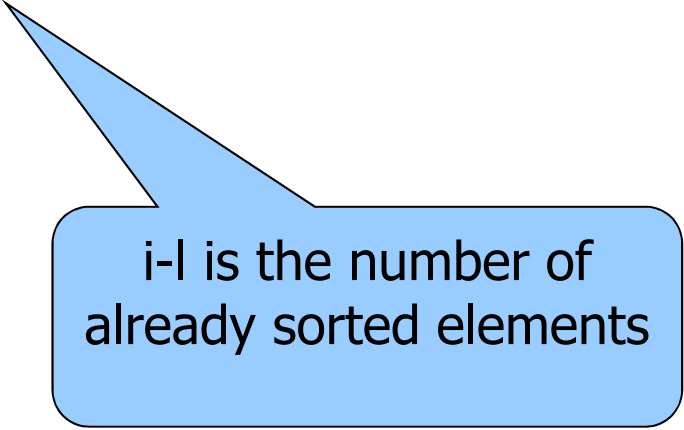




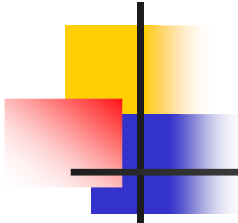
C Code

```
void BubbleSort (int A[], int n){
    int i, j, temp;

    for (i=0; i<n-1; i++) {
        for (j=0; j<n-i-1; j++) {
            if (A[j] > A[j+1])) {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
    return;
}
```



i-1 is the number of
already sorted elements



Complexity

- Analytic analysis
 - Worst case
 - We assume unit cost for all operations

```
for (i=0; i<n-1; i++) {  
    for (j=0; j<n-i-1; j++) {  
        if (A[j] > A[j+1]) {  
            temp = A[j];  
            A[j] = A[j+1];  
            A[j+1] = temp;  
        }  
    }  
}
```

$1 + (n-1+1) + (n-1)$

$\sum_{i=0}^{n-2} (1 + (n-i-1+1) + (n-i-1))$

$\sum_{i=0}^{n-2} (n-i-1)$

$\sum_{i=0}^{n-2} (n-i-1)$

$\sum_{i=0}^{n-2} (n-i-1)$

$\sum_{i=0}^{n-2} (n-i-1)$



Complexity

$$\begin{aligned} & 1 + (n-1+1) + (n-1) \\ & \sum_{i=0}^{n-2} (1 + (n-i-1+1) + (n-i-1)) \\ & \sum_{i=0}^{n-2} (n-i-1) \\ & \sum_{i=0}^{n-2} (n-i-1) \\ & \sum_{i=0}^{n-2} (n-i-1) \\ & \sum_{i=0}^{n-2} (n-i-1) \end{aligned}$$

$$\begin{aligned} T(n) &= \\ &= 2n + \sum_{i=0}^{n-2} 1 + \sum_{i=0}^{n-2} (n-i) + 5\sum_{i=0}^{n-2} (n-i-1) \\ &= 2n + \sum_{i=0}^{n-2} 1 + \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i + 5\sum_{i=0}^{n-2} n - 5\sum_{i=0}^{n-2} i - 5\sum_{i=0}^{n-2} 1 \\ &= 2n + 6\sum_{i=0}^{n-2} n - 6\sum_{i=0}^{n-2} i - 4\sum_{i=0}^{n-2} 1 \\ &= 2n + 6n(n-1) - 6\frac{(n-2)(n-1)}{2} - 4(n-1) \\ &= 3n^2 + n - 2 \end{aligned}$$

$T(n)$ grows quadratically

Worst case
 $O(n^2)$ overall



Complexity

- More intuitive analysis
 - Two nested loops
 - Outer loop
 - Executed $n-1$ times
 - Inner loop
 - At the i -th iteration executed $n-1-i$ times
 - $T(n) = (n-1) + (n-2) + \dots + 2 + 1$
 $= O(n^2)$



Optimized C Code

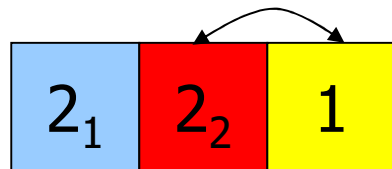
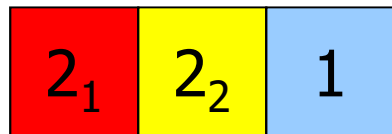
```
void optBubbleSort (int A[], int n) {
    int i, j, flag, temp;

    flag = 1;
    for(i=0; i<n-1 && flag==1; i++) {
        flag = 0;
        for (j=0; j<n-i; j++)
            if (A[j] > A[j+1]) {
                flag = 1;
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
    return;
}
```

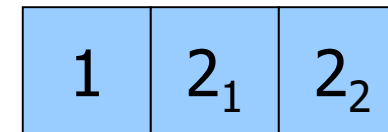
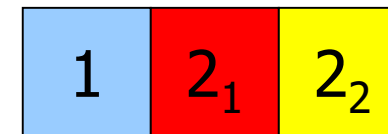
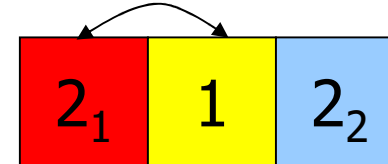
Features

- In place
- Stable
 - Among several duplicate keys, the rightmost one takes the rightmost position and no other identical key ever moves past it to the right

$i = 0$



$i = 1$



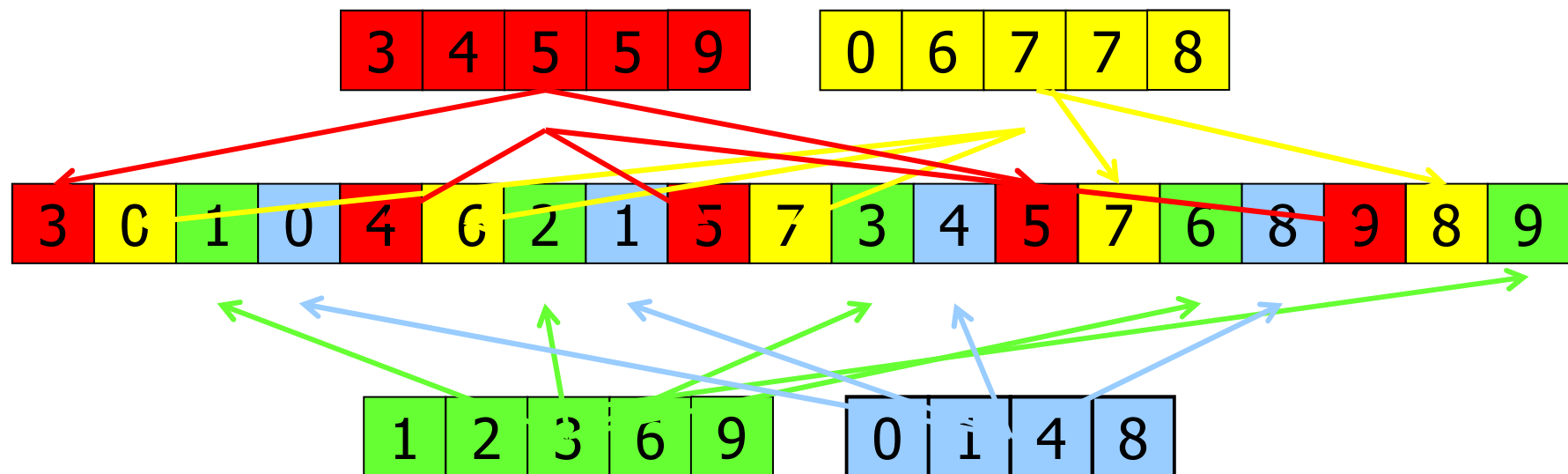


Shellsort (Shell, 1959)

- Limit of insertion sort
 - Comparison, thus exchange takes place only between adjacent elements
- Rationale of Shellsort
 - Compare, thus possibly exchange, elements at distance h
 - Defining a decreasing sequence of integers ending with 1

Shellsort (Shell, 1959)

- An array formed by non contiguous sequences composed by elements whose distance is h is h -sorted
- Example
 - Sorted non contiguous subsequences with $h=4$





Shellsort (Shell, 1959)

- For each of the subsequences we apply insertion sort
- The elements of the subsequence are those at distance h from the current one

```
for (i=h; i<n; i++) {  
    int j = i, v = A[i];  
    while (j>=h && v<A[j-h])) {  
        A[j] = A[j-h];  
        j -=h;  
    }  
    A[j] = v;  
}
```




Example

5	8	9	0	4	6	3	1	3	7	6	4	5	7	1	8	9	0	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Sequence h: 13, 4, 1

Step1: h=13

5	1	8	0	0	2	3	1	3	7	6	4	5	7	8	9	9	4	6
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Step 2: h=4

0	1	3	0	3	2	6	1	5	4	6	4	5	4	8	9	9	7	8
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Step 3: h=1

0	0	1	1	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



Choosing the sequence

- Has an impact on performance
- Knuth's sequence
 - $h = 3 \cdot h + 1 = 1 \ 4 \ 13 \ 40 \ 121 \dots$
- Sequence
 - $h = 1$ then $4^{i+1} + 3 \cdot 2^i + 1 = 1 \ 8 \ 23 \ 77 \ 281 \ 1073$
...
- Sedgewick's sequence
 - $h = 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, \dots$



C Code

```
void shellsort(int A[], int n) {
    int i, j, temp, h;
    h=1;
    while (h < n/3)
        h = 3*h+1;
    while (h >= 1) {
        for (i=h; i<n; i++) {
            j = i;
            temp = A[i];
            while (j>=h && temp<A[j-h]) {
                A[j] = A[j-h];
                j -=h;
            }
            A[j] = temp;
        }
        h = h/3;
    }
}
```



Features

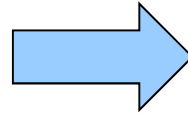
- In place
- Not stable
 - An exchange between "far away" elements may result in a duplicate key that passes over to the left a preceding occurrence of the same key



Example

2_1	2_2	2_3	2_4	2_5	0
-------	-------	-------	-------	-------	---

2_1	2_2	2_3	2_4	2_5	0
-------	-------	-------	-------	-------	---



2_1	0	2_3	2_4	2_5	2_2
-------	---	-------	-------	-------	-------

- Step 1: $h=4$

0	2_1	2_3	2_4	2_5	2_2
---	-------	-------	-------	-------	-------

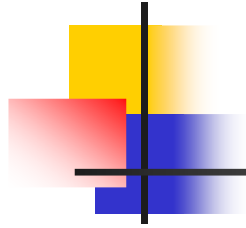
- Step 2: $h=1$



Worst-case asymptotic analysis

Shellsort

- With Knuth's sequence:
 - 1 4 13 40 121 ...
 - It executes less than $O(n^{3/2})$ comparisons
- With the sequence
 - 1 8 23 77 281 1073 ...
 - It executes less than $O(n^{4/3})$ comparisons
- With Shell's original sequence
 - 1 2 4 8 16 ...
 - It may degenerate to $O(n^2)$



Counting sort

- Sorting based on computation (not on comparison)
 - Find, for each element to sort x , how many elements are less than or equal to x
 - Assign x directly to its final location
- Features
 - Stable
 - Not in place



Data Structures

- 3 arrays
 - Starting array
 - $A[0..n-1]$ of n integers
 - Resulting array
 - $B[0..n-1]$ of n integers
 - Occurrence array
 - C of k integers **if** data belong to the range $[0..k-1]$



Algorithm

- Step 1: simple occurrences
 - $C[i]$ = number of elements of A equal to i
- Step 2: multiple occurrences
 - $C[i]$ = number of elements of $A \leq i$
- Step3: $\forall j$
 - $C[A[j]]$ = number of elements $\leq A[j]$
- Thus final location of $A[j]$ in B
$$B[C[A[j]]] = A[j]$$

(beware of indices in C , see code!)



Example ($n=8$, $k=6$)

	0	1	2	3	4	5	6	7
A	2	5	3	0	2	3	0	3

Array to sort

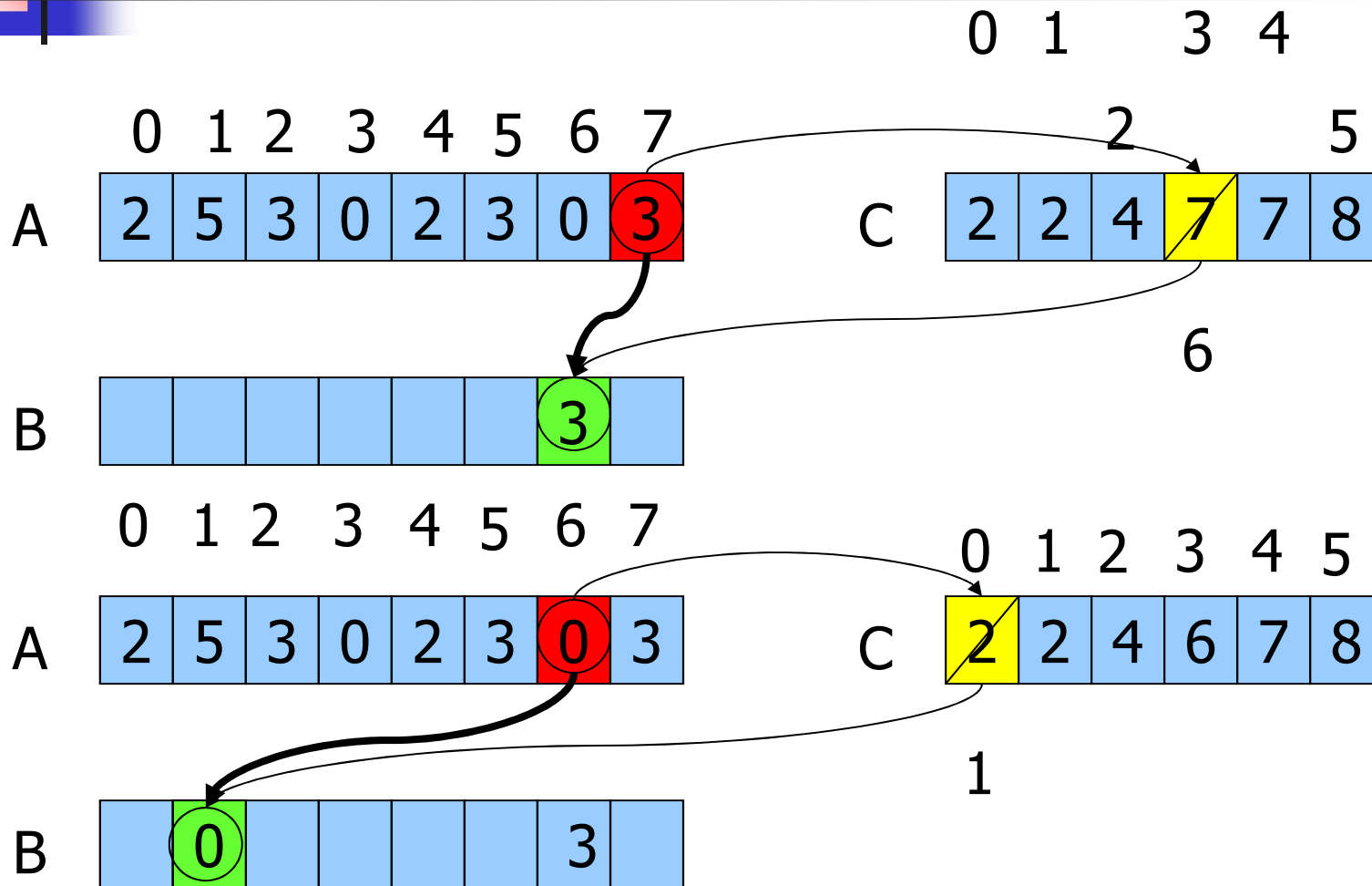
	0	1	2	3	4	5
C	2	0	2	3	0	1

Simple occurrences

	0	1	2	3	4	5
C	2	2	4	7	7	8

Multiple occurrences

Example (n=8, k=6)





Example (n=8, k=6)

```
#define MAX 100
```

```
void CountingSort(int A[], int n, int k) {  
    int i, C[MAX], B[MAX];  
  
    for (i=0; i<k; i++)  
        C[i] = 0;  
    for (i=0; i<n; i++)  
        C[A[i]]++;  
    for (i=1; i<k; i++)  
        C[i] += C[i-1];  
    for (i=n-1; i>=0; i--) {  
        B[C[A[i]]-1] = A[i];  
        C[A[i]]--;  
    }  
    for (i=0; i<n; i++)  
        A[i] = B[i];  
}
```



Worst-case asymptotic analysis

- Initialization loop for C: $O(k)$
- Loop to compute simple occurrences: $O(n)$
- Loop to compute multiple occurrences: $O(k)$
- Loop to copy result in B: $O(n)$
- Loop to copy in A: $O(n)$

$$T(n) = O(n+k)$$

Applicability

- $k=O(n)$, thus $T(n) = O(n)$