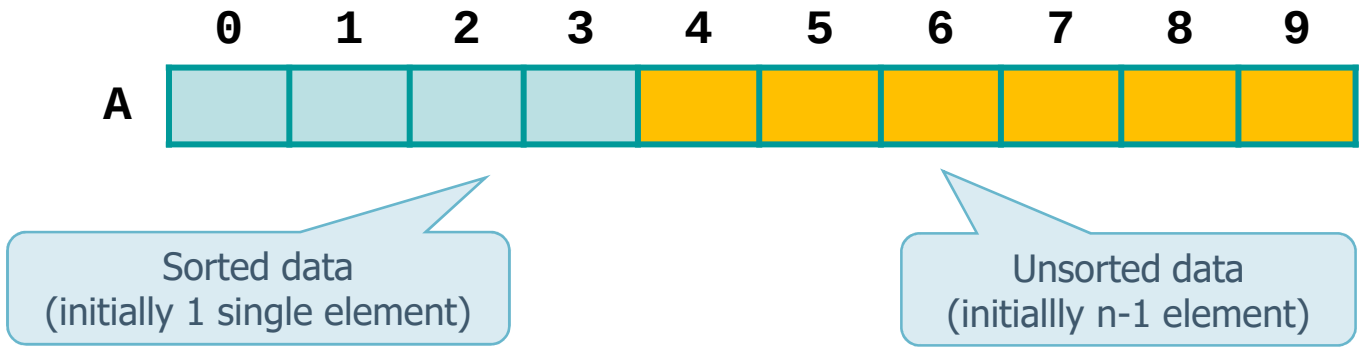
Sorting algorithms

Iterative algorithms

Paolo Camurati and Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

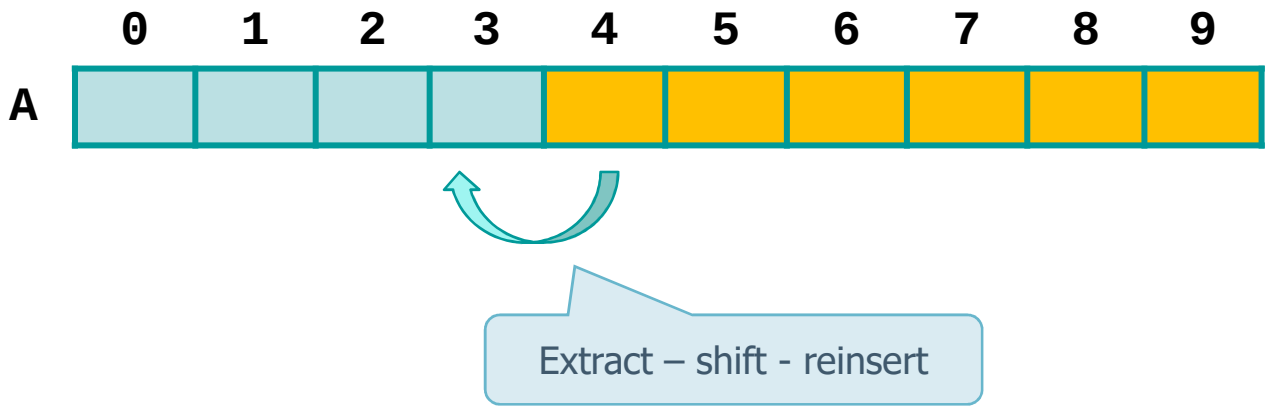
Insertion sort

- Data
  - Integers in array A on n elements
- Array partitioned in 2 sub-arrays
  - Left subarray: sorted
  - Right subarray: unsorted
- An array of just one element is sorted



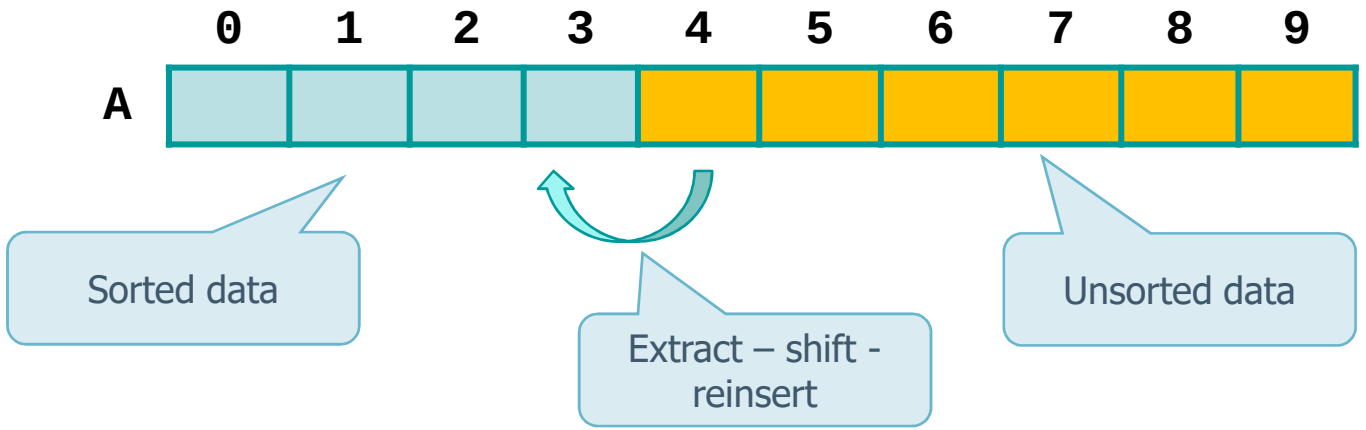
Insertion sort

- 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

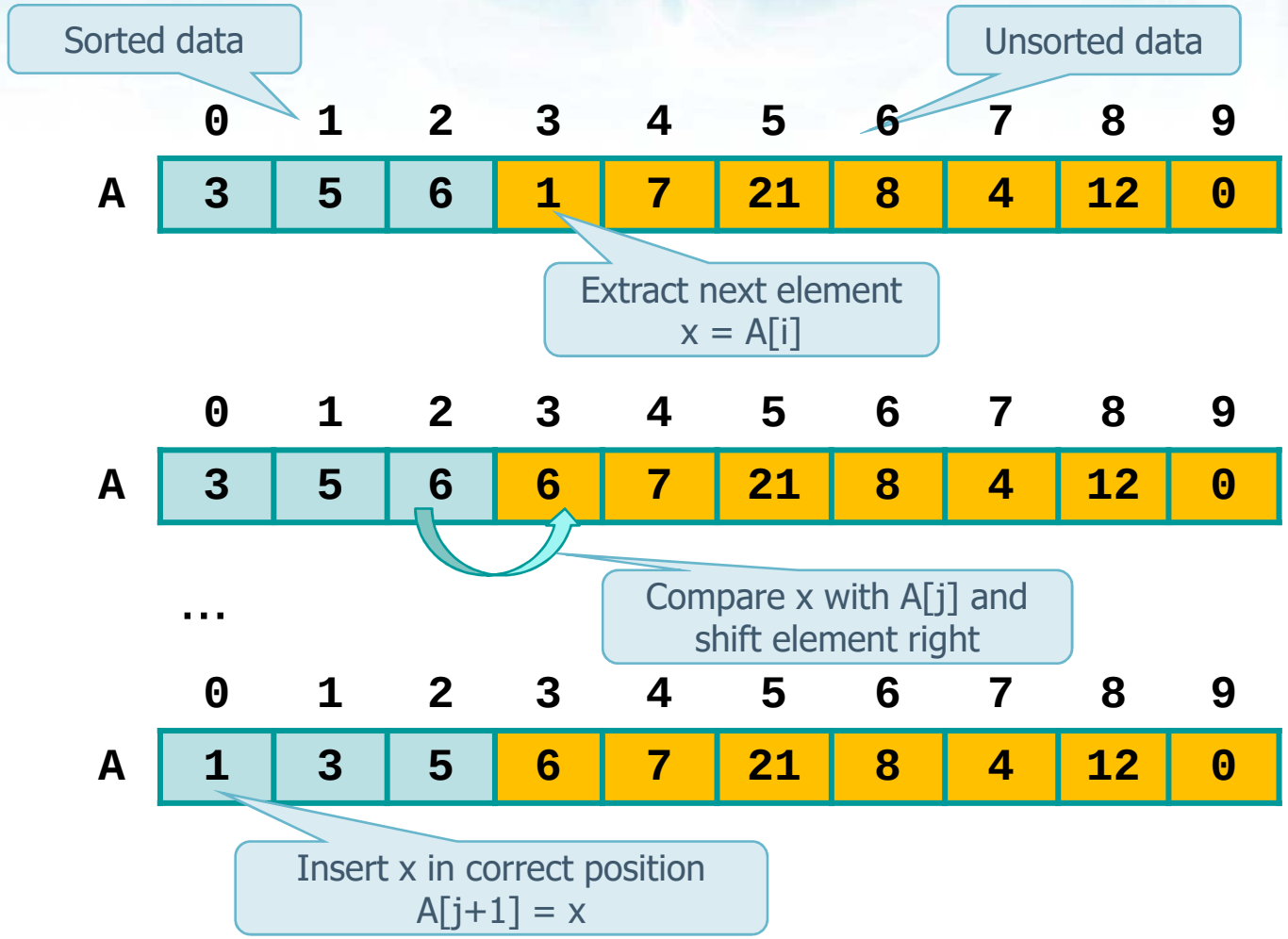


i-th step: Sorted insertion

- 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


Sorted data
Unsorted data
0 1 2 3 4 5 6 7 8 9
A 3 5 6 1 7 21 8 4 12 0
Extract next element
x = A[i]
0 1 2 3 4 5 6 7 8 9
A 3 5 6 6 7 21 8 4 12 0
Compare x with A[j] and
shift element right
...
0 1 2 3 4 5 6 7 8 9
A 1 3 5 6 7 21 8 4 12 0
Insert x in correct position
A[j+1] = x

Example

Sorted data

Unsorted data

0	1	2	3	4	5	6	7	8	9
6	3	5	1	7	21	8	4	12	0
3	6	5	1	7	21	8	4	12	0
3	5	6	1	7	21	8	4	12	0
1	3	5	6	7	21	8	4	12	0
1	3	5	6	7	21	8	4	12	0
1	3	5	6	7	21	8	4	12	0
1	3	5	6	7	8	21	4	12	0
1	3	4	5	6	7	8	21	12	0
1	3	4	5	6	7	8	12	21	0
0	1	3	4	5	6	7	8	12	21

Implementation

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

```
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;
}
```

Select a new element

Extract it

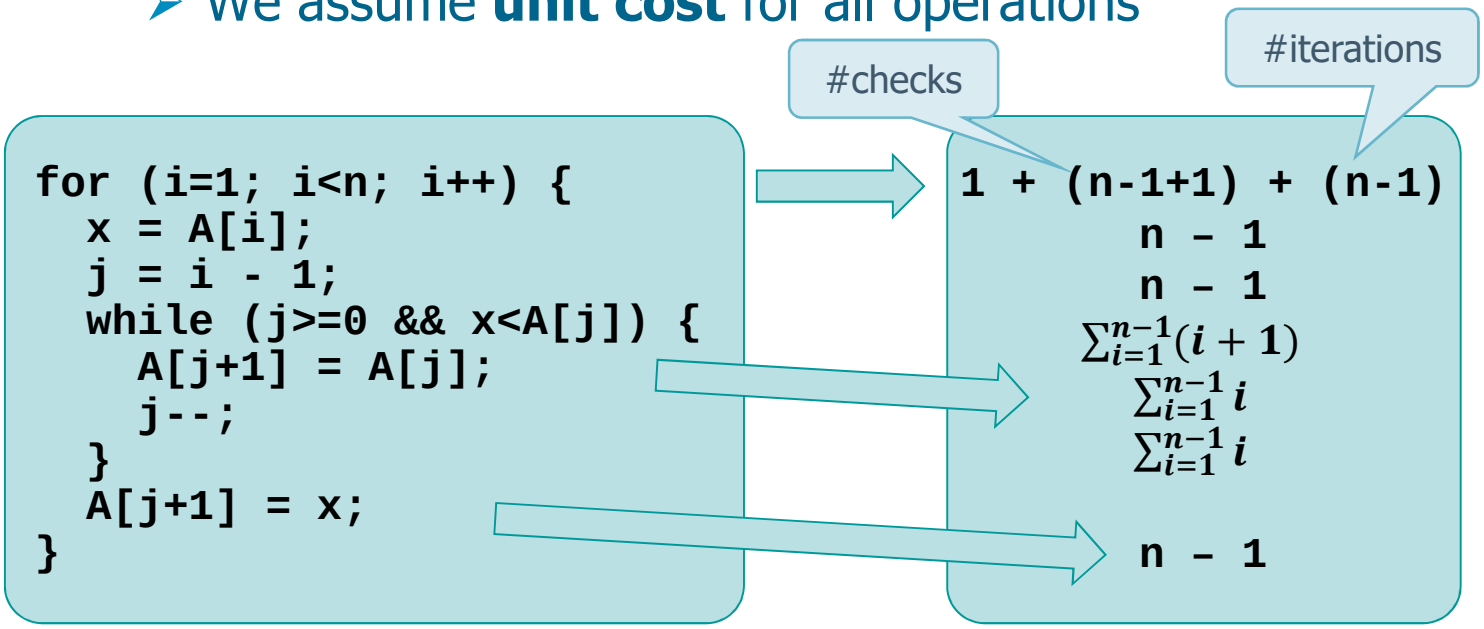
Find its correct position

Right shift

Insert

Complexity analysis

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



Complexity analysis

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

```
1 + (n-1+1) + (n-1)
      n - 1
      n - 1
  Σ_{i=1}^{n-1} (i + 1)
  Σ_{i=1}^{n-1} i
  Σ_{i=1}^{n-1} i
      n - 1
```

Recall that

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

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

$$= 2n + 3(n-1) + \frac{n(n+1)}{2} - 1 + 2\frac{(n-1)n}{2}$$

$$= 2n + 3n - 3 + \frac{1}{2}n^2 + \frac{1}{2}n - 1 + n^2 - n$$

$$= \frac{3}{2}n^2 + \frac{9}{2}n - 4 = \Theta(n^2)$$

Worst case

- T(n) grows quadratically

$O(n^2)$ overall

Complexity analysis

- Intuitive analysis for the **worst** case
  - Two nested cycles
  - Outer loop
    - Always n-1 executions
  - Inner loop
    - The worst-case → i executions at the i-th iteration of the outer loop
- Complexity
    - $T(n) = 1 + 2 + 3 + 4 + \cdots + (n-2) + (n-1)$
    - $T(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
  - T(n) grows quadratically with n

Complexity analysis

- Intuitive analysis for the **best** case
  - Inner loop
    - Only 1 execution at the i-the iteration of the outer loop
  - Complexity
    - T(n) = 1 + 1 + 1 + ... + 1 = n-1
  - T(n) grows linearly with n

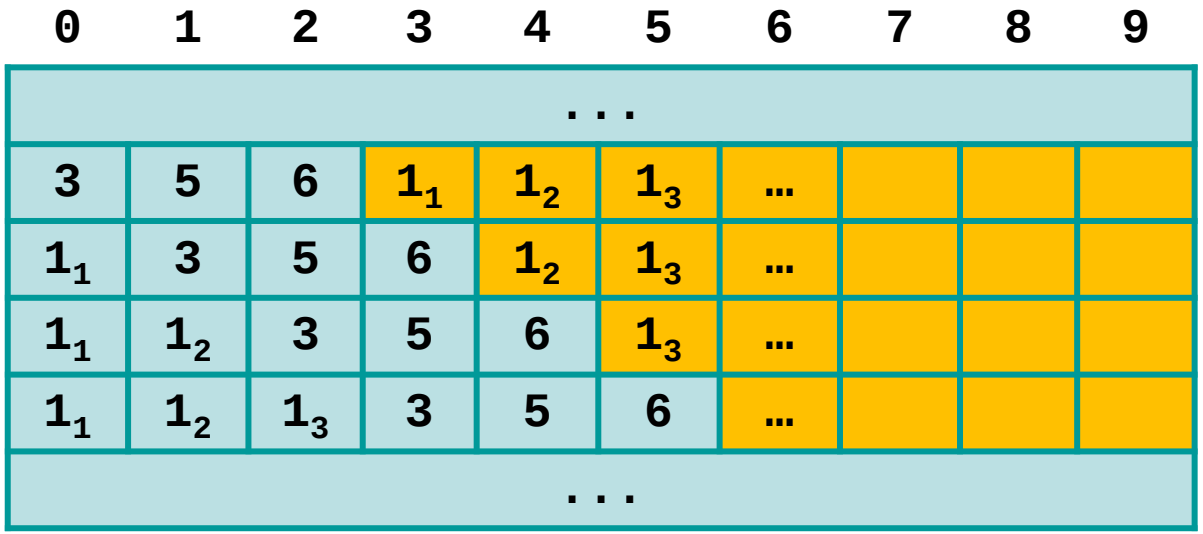
1 + … n-1 times

Features

- In place
- Number of exchanges in worst-case
  - $O(n^2)$
- Number of comparisons in worst-case
  - $O(n^2)$

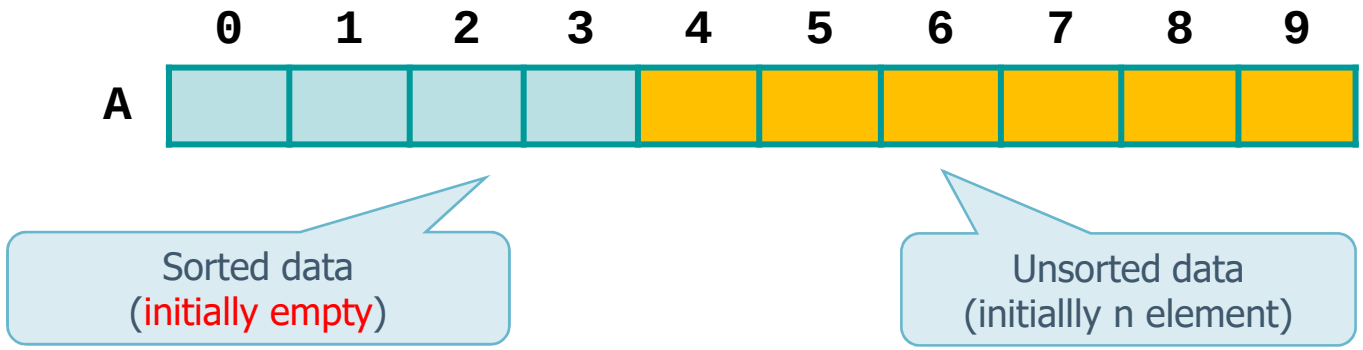
Features

- Stable
  - If the element to insert is a duplicate key, it can't pass over to the left a preceeding occurrence of the same key



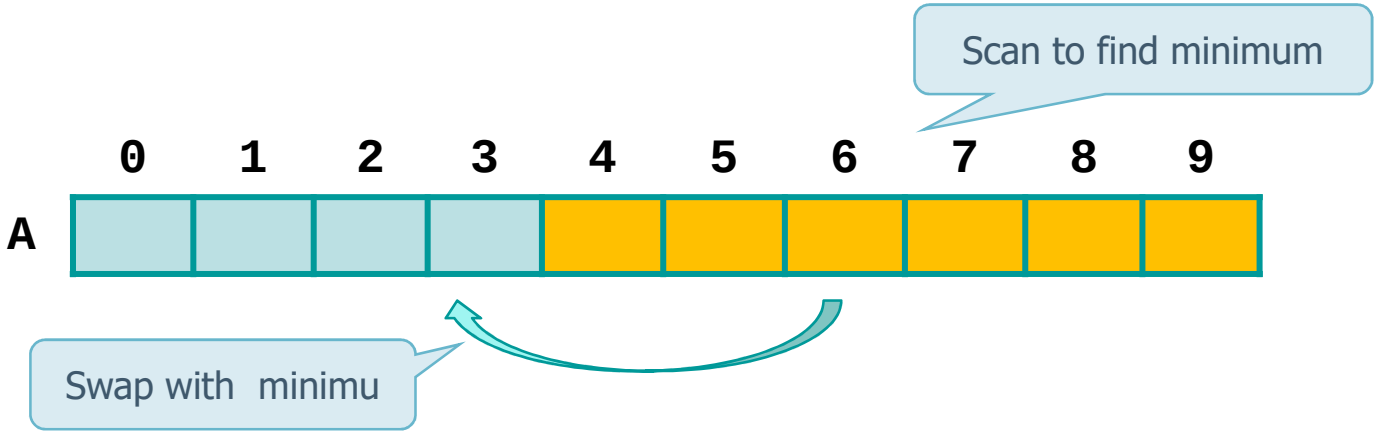
Selection sort

- Data
  - Integers in array A on n elements
- Array partitioned in 2 sub-arrays
  - Left subarray: sorted
  - Right subarray: unsorted
- An array of just one element is sorted

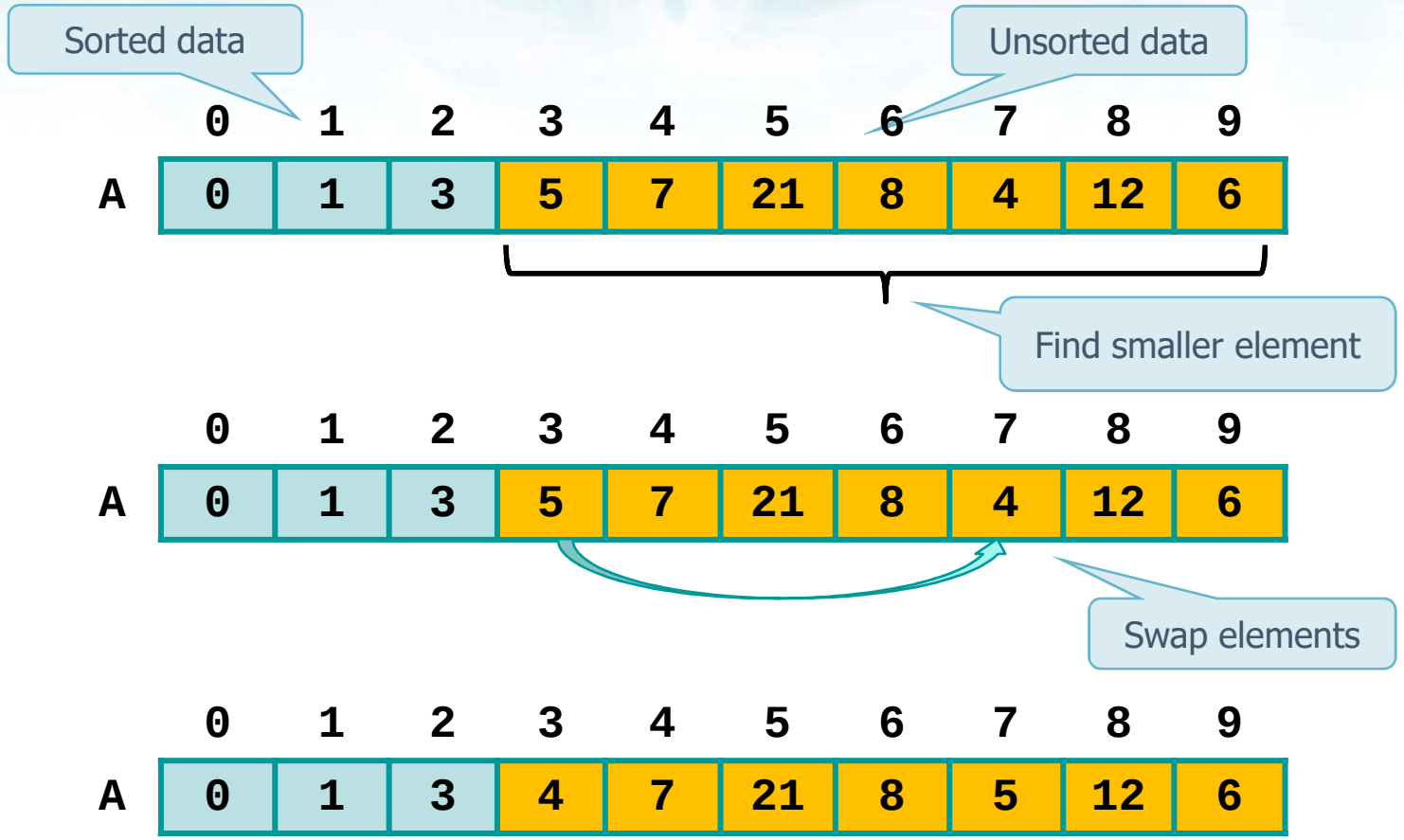


Selection sort

- Incremental approach
  - Scan the unsorted subarray (from A[i] to A[n-1]) to find the minimum value
  - Swap the minimum value with A[i]
- Termination
  - All elements inserted in proper order



Example


Sorted data
Unsorted data
0 1 2 3 4 5 6 7 8 9
A 0 1 3 5 7 21 8 4 12 6
Find smaller element
0 1 2 3 4 5 6 7 8 9
A 0 1 3 5 7 21 8 4 12 6
Swap elements
0 1 2 3 4 5 6 7 8 9
A 0 1 3 4 7 21 8 5 12 6

Example

0	1	2	3	4	5	6	7	8	9
6	3	5	1	7	21	8	4	12	0
0	3	5	1	7	21	8	4	12	6
0	1	5	3	7	21	8	4	12	6
0	1	3	5	7	21	8	4	12	6
0	1	3	4	7	21	8	5	12	6
0	1	3	4	5	21	8	7	12	6
0	1	3	4	5	6	8	7	12	21
0	1	3	4	5	6	7	8	12	21
0	1	3	4	5	6	7	8	12	21
0	1	3	4	5	6	7	8	12	21

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;
}
```

Select a new element

Find the minimum in unsorterd subarray

Swap

Complexity Analysis

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

Number of operations

```
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 + (n-1+1) + (n-1)
n-1
∑_{i=0}^{n-2}(1 + (n − (i + 1) + 1) + (n − (i + 1))
∑_{i=0}^{n-2}(n − (i + 1))
∑_{i=0}^{n-2}(n − (i + 1))

n-1
n-1
n-1
```

Complexity Analysis

1 + (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

Recalling that

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

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

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

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

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

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

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

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

T(n) grows quadratically

Worst case $O(n^2)$ overall

Complexity Analysis

- Intuitive analysis for the **worst** case
  - Two nested loops
  - Outer loop: executed n-1 times
  - Inner loop: at the i-th iteration executed n-i-1 times
    - $T(n) = (n-1) + (n-2) + \dots 2 + 1 = O(n^2)$

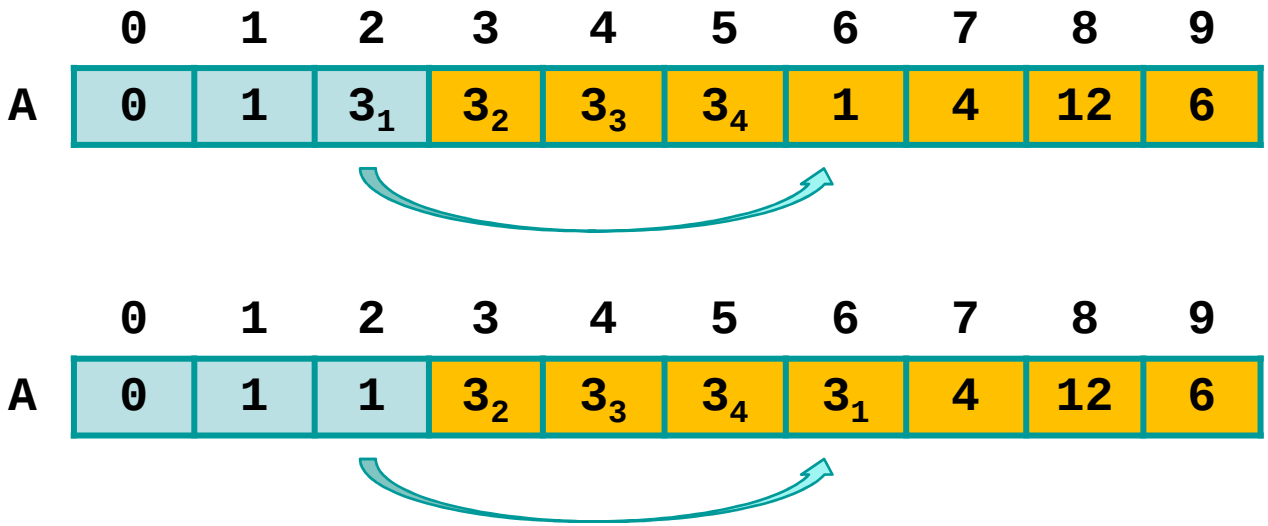
Recalling that $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

Features

- In place
- Number of exchanges in worst-case
  - O(n)
- Number of comparisons in worst-case
  - $O(n^2)$

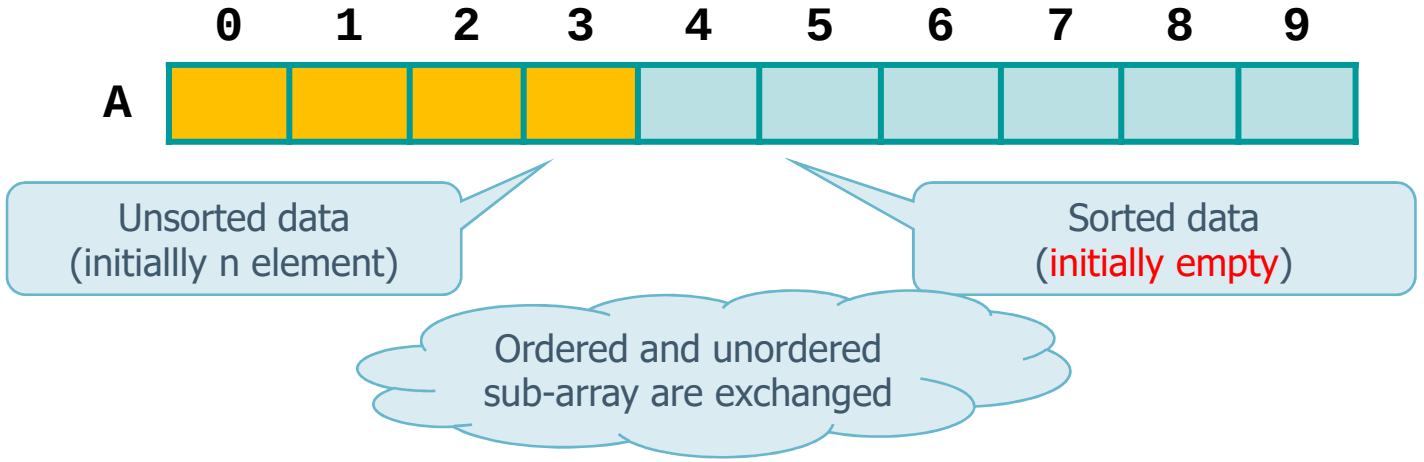
Features

- 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



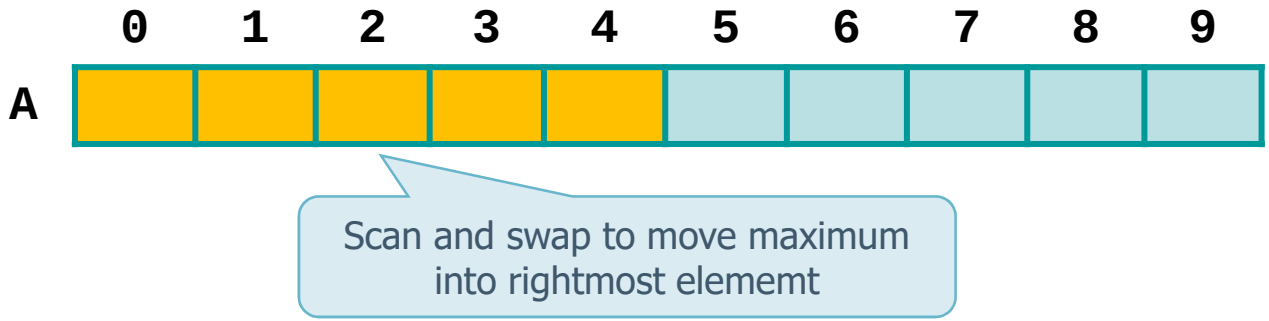
Exchange (bubble) sort

- Data
  - Integers in array A on n elements
- Array partitioned in 2 sub-arrays
  - Left subarray: sorted
  - Right subarray: unsorted
- An array of just one element is sorted

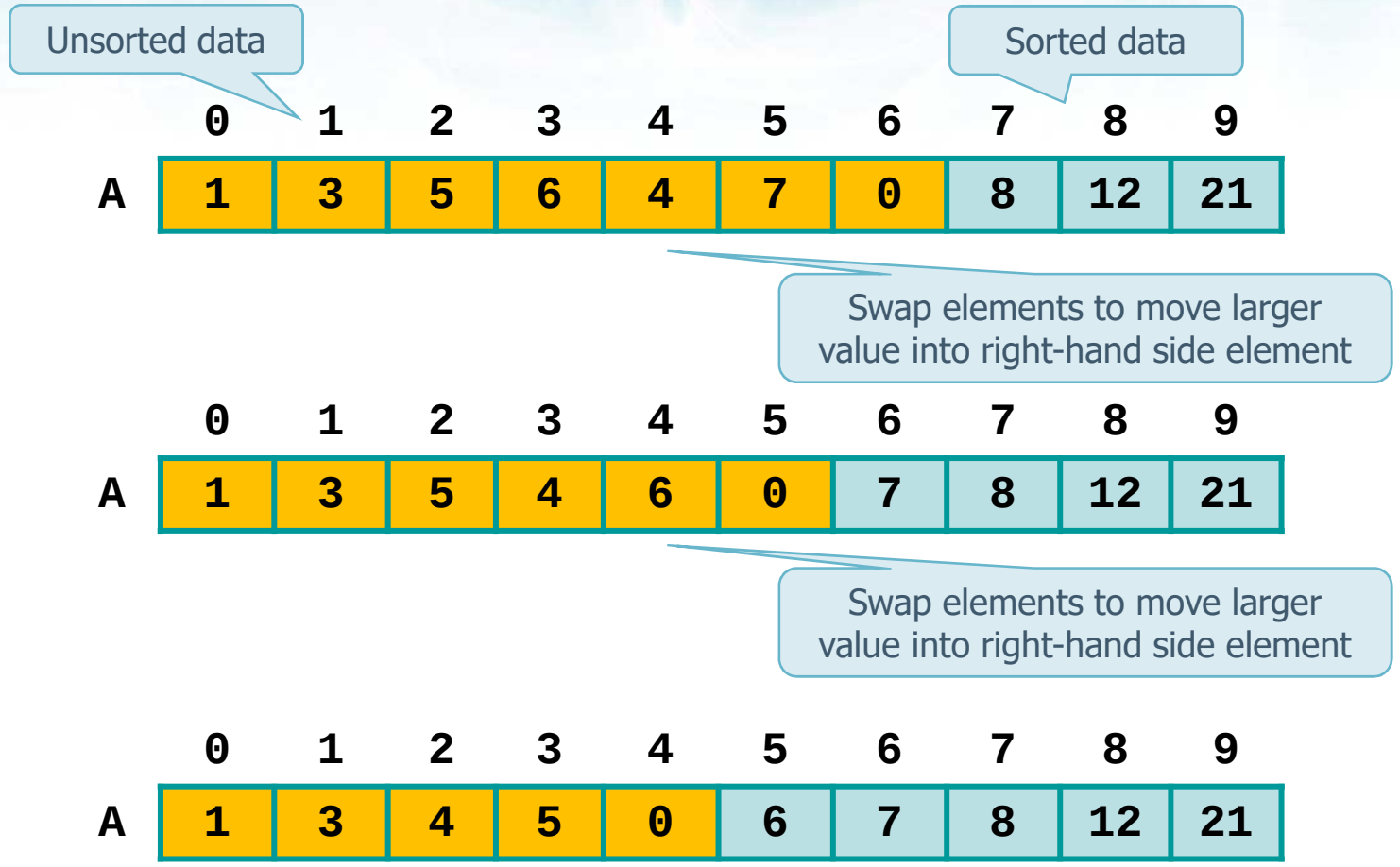


Exchange (bubble) sort

- Incremental approach
  - Scan the unsorted subarray (from A[0] to A[n-i-1])
  - Compare successive elements of the array A[j] and A[j+1], swap if A[j] > A[j+1]
- Termination
  - All elements inserted in proper order



Example


Unsorted data
Sorted data
0 1 2 3 4 5 6 7 8 9
A 1 3 5 6 4 7 0 8 12 21
Swap elements to move larger value into right-hand side element
0 1 2 3 4 5 6 7 8 9
A 1 3 5 4 6 0 7 8 12 21
Swap elements to move larger value into right-hand side element
0 1 2 3 4 5 6 7 8 9
A 1 3 4 5 0 6 7 8 12 21

Example

0	1	2	3	4	5	6	7	8	9
6	3	5	1	7	21	8	4	12	0
3	5	1	6	7	8	4	12	0	21
3	1	5	6	7	4	8	0	12	21
1	3	5	6	4	7	0	8	12	21
1	3	5	4	6	0	7	8	12	21
1	3	4	5	0	6	7	8	12	21
1	3	4	0	5	6	7	8	12	21
1	3	0	4	5	6	7	8	12	21
1	0	3	4	5	6	7	8	12	21
0	1	3	4	5	6	7	8	12	21

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 is the number of already sorted elements

For each i we scan the array but last i elements

Swap elements in case they are in wrong position

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

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

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

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

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

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

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

T(n) =

$= 2n + \sum_{i=0}^{n-2} 1 + \sum_{i=0}^{n-2}(n-i) + 5\sum_{i=0}^{n-2}(n-i-1)$

$= 2n + \sum_{i=0}^{n-2} 1 + \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i + 5\sum_{i=0}^{n-2} n - 5\sum_{i=0}^{n-2} i - 5\sum_{i=0}^{n-2} 1$

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

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

$= 3n^2 + n - 2 = \Theta(n^2)$

T(n) grows quadratically

Worst case $O(n^2)$ overall

Complexity

- Intuitive analysis for the **worst** case
  - Two nested loops
  - Outer loop
    - Executed n-1 times
  - Inner loop
    - At the i-th iteration executed n-1-i times
  - Overall
    - $T(n) = (n-1) + (n-2) + \ldots + 2 + 1 = O(n^2)$

Optimization 1

- A possible optimization for the previous algorithms consist of terminating the entire process as soon as the array is already sorted
  - If the array is already sorted the are no swap in the **inner** loop
  - We can
    - Use a flag to record that there have been no swaps
    - If there have been no swaps, early exit the **outer** loop

```
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;
    }
  }
}
```

C Code

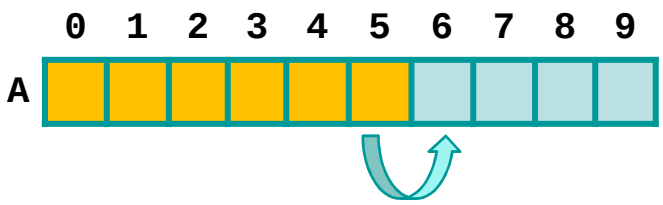
```
void OptBubbleSort (int A[], int n) {
  int i, j, flag, temp;

  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;
}
```

Check "there was a swap in the previous inner iteration?"

Store if there is a swap

Optimization 2

- The previous solution can be further optimized by registering the position of the last swap
  - If the last swap has been done in position temp (and temp+1) all values beyond temp+1 are in the correct position



Use index insted of flag

```
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;
  }
}
```

Features

- In place
- Number of exchanges in worst-case
  - $O(n^2)$
- Number of comparisons in worst-case
  - $O(n^2)$

Features

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

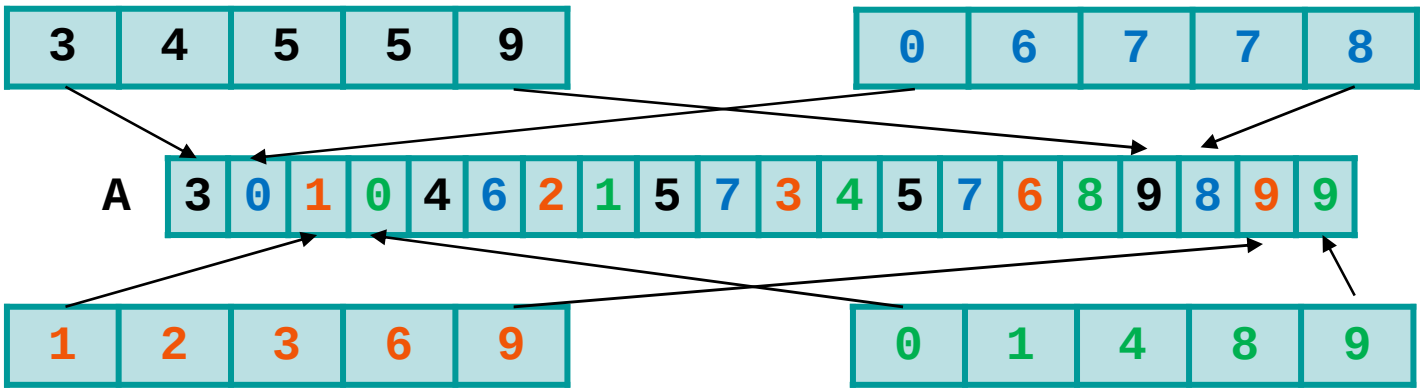
0	1	2	3	4	5	6	7	8	9
...									
3_1	3_2	3_3	1	5	6	7	8	9	10
3_1	3_2	1	3_3	5	6	7	8	9	10
3_1	1	3_2	3_3	5	6	7	8	9	10
1	3_1	3_2	3_3	5	6	7	8	9	10
...									

Shellsort

- Limit of insertion sort
  - Comparison, thus exchange takes place only between adjacent elements
  - Shell (1959) propose shellsort
- Rationale of Shellsort
  - Compare, thus possibly exchange, elements at distance h
  - Defining a decreasing sequence of integers h ending with 1

Shellsort

- 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

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

(original) insertion sort

Example

- Input array

0	1	2	3	4	5	6	7	8	9
3	4	0	17	11	8	3	14	1	5

- Sequence
  - h = 4, 1
- Process and output array

0	1	2	3	4	5	6	7	8	9
3	4	0	17	11	8	3	14	1	5
1	4	0	14	3	5	3	17	11	8
0	1	3	3	4	5	8	11	14	17

h=4

h=4

Example

- Input array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
7	3	2	4	1	0	4	17	16	11	9	8	1	3	5	3	12	17	14	5

- Sequence
  - h = 13, 4, 1
- Process and output array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
7	3	2	4	1	0	4	17	16	11	9	8	1	3	5	3	12	17	14	5
3	3	2	4	1	0	4	17	16	11	9	8	1	7	5	3	12	17	14	5
1	0	2	3	1	3	4	4	3	7	5	5	12	11	9	8	16	17	14	17
0	1	1	2	3	3	3	4	4	5	5	7	8	9	11	12	14	16	17	17

Example

- Input array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	5	0	6	7	0	2	7	6	4	3	2	2	4	6	7	9	8	1	0

- Sequence
  - h = 13, 4, 1
- Process and output array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	5	0	6	7	0	2	7	6	4	3	2	2	4	6	7	9	8	1	5
1	5	0	6	7	0	2	7	6	4	3	2	2	4	6	7	9	8	1	5
1	0	0	2	2	4	1	5	6	4	2	6	7	5	3	7	9	8	6	7
0	0	1	1	2	2	2	3	4	4	5	5	6	6	6	7	7	7	8	9

Choosing the sequence

- Shell sort starts with large vaue of h and move to small values
  - The first value is the largest one smallr than n
- The sequence adopted has an impact on performance
  - Up to 25%
- The original Shell's sequence
  - $h_i = 2^i$

  has very bad performances

Choosing the sequence

- Knuth's sequence (1969)
  - Starts from
    - $h_0 = 1$
  - Computes
    - $h_i = 3 \cdot h_{i-1} + 1$, for i = 1, 2, …
  - Generates
    - h = 1, 4, 13, 40, 121, …

Choosing the sequence

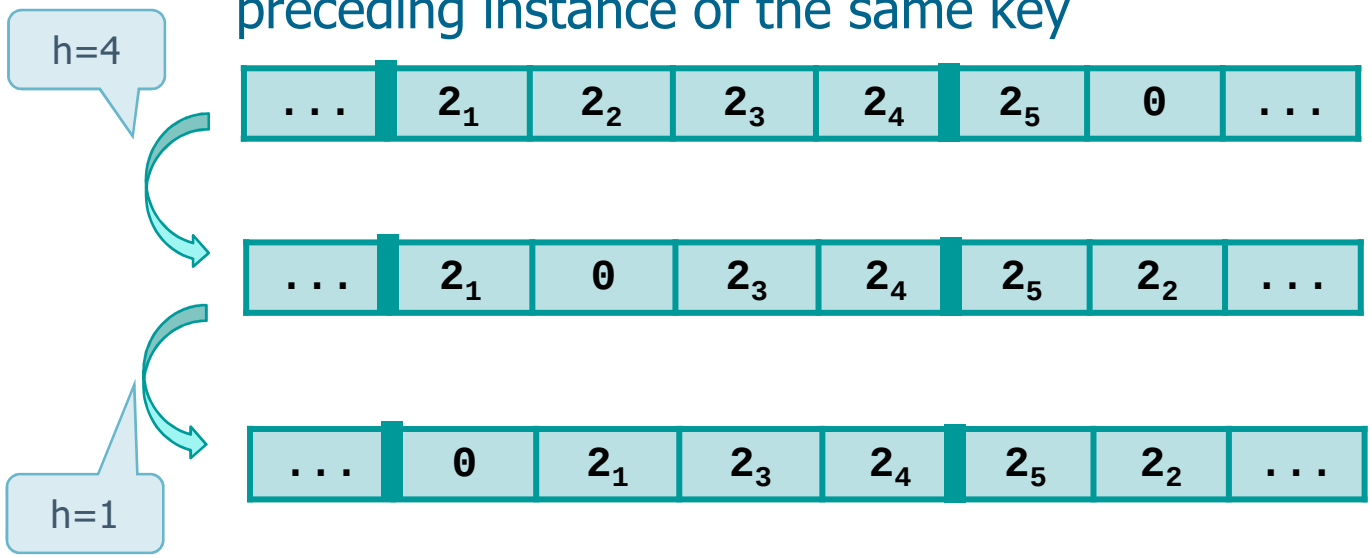
- Sequence (provably faster worst case behaviour)
  - Starts from
    - $h_0 = 1$
  - Computes
    - $h_{i+1} = 4^{i+1} + 3 \cdot 2^i + 1$, for i = 0, 1, …
  - Generates
    - h = 1, 8, 23, 77, 281, 1073, …
- Sedgewick's sequence
  - h = 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, …

C Code

```
void ShellSort (int A[], int n) {
  int i, j, x, h;
  h=1;
  while (h < n/3)
    h = 3*h+1;
  while (h >= 1) {
    for (i=h; i<n; i++) {
      x = A[i];
      j = i - h;
      while (j>=0  && x<A[j]) {
        A[j+h] = A[j];
        j -= h;
      }
      A[j+h] = x;
    }
    h = h/3;
  }
}
```

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



Complexity

- Intuitive analysis for the **worst** case
  - With Shell's original sequence
    - h = 1, 2, 4, 8, 16, …
    - It may degenerate to $O(n^2)$
  - Knuth's sequence
    - h = 1, 4, 13, 40, 121, …
    - It executes less than $O(n^{3/2})$ comparisons
  - Sequence
    - h = 1, 8, 23, 77, 281, 1073, …
    - It executes less than $O(n^{4/3})$ comparisons

Counting sort

- Counting sorting is based on computation, it is not based on comparison
  - Data keys must be countable values
    - Integer, characters
  - Find, for each element to sort x, how many elements are less than or equal to x
  - Assign x directly to its final location

Counting sort

- Data structure
  - 3 arrays
    - Starting array
      - A[0]…A[n-1] of n integers
      - Each integer must be in a specifi range, A[i]∈[0,k]
    - Resulting array
      - B [0]...B[n-1] of n integers
    - Occurrence array
      - C of k integers if data belong to the range [0, k-1]

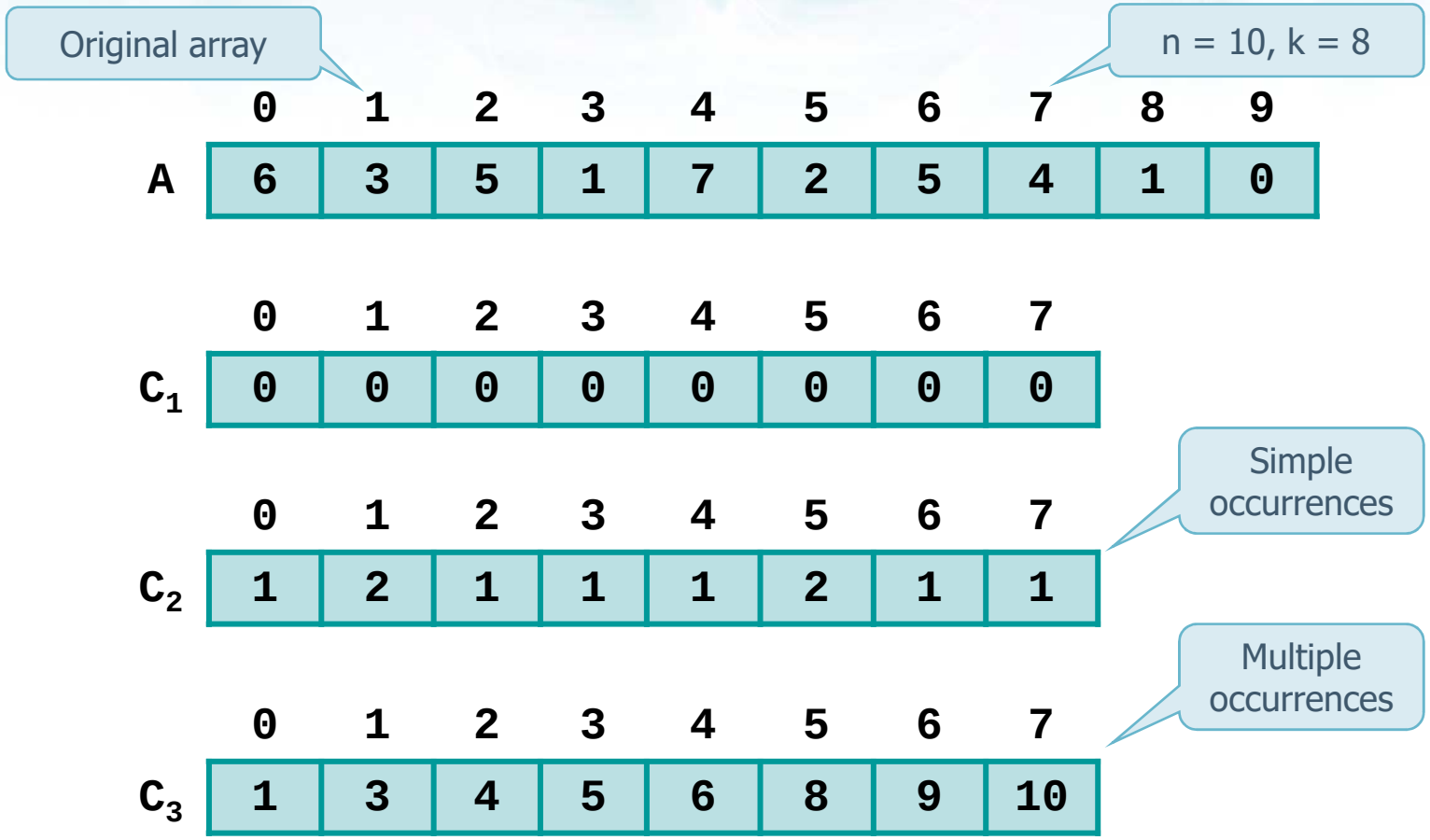
Algorithm

- Logic
  - Step 1
    - Initialize array C
    - C[i] = 0 for all i
  - Step 2
    - Generate simple occurrences
    - C[i] = number of elements of A equal to i
  - Step 3
    - Generate multiple occurrences
    - C[i]-1 = number of elements of A <= i

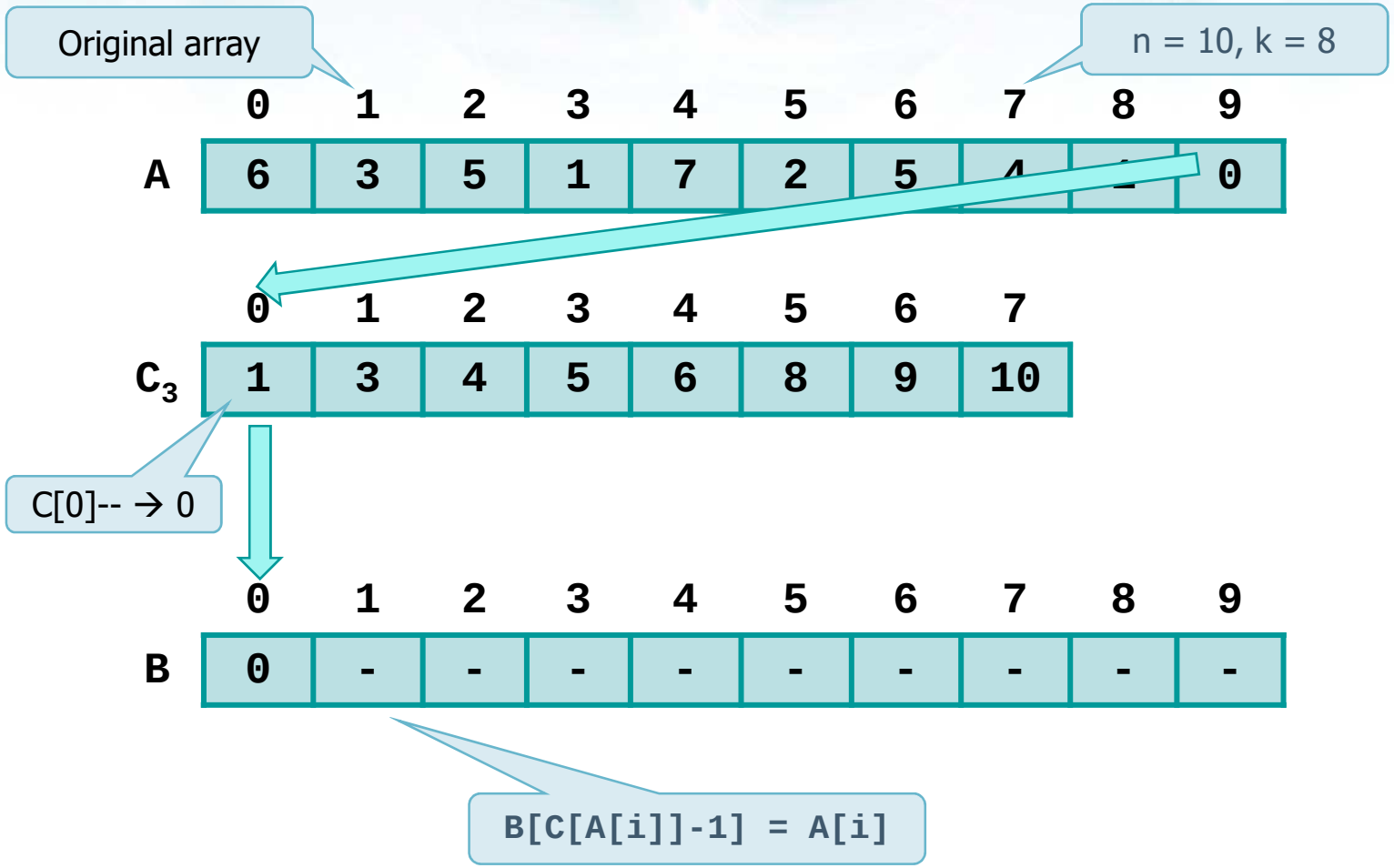
Algorithm

- Step4
  - Copy elements from A to B
  - ∀ j, C[A[j]-1] = number of elements <= A[j]
  - Thus final location of A[j] in B is
    - B[C[A[j]]-1] = A[j]
    - Decrement C[A[j]] after each copy
  - Beware of indices in C (see code)
- Step 5
  - Copy B back into A

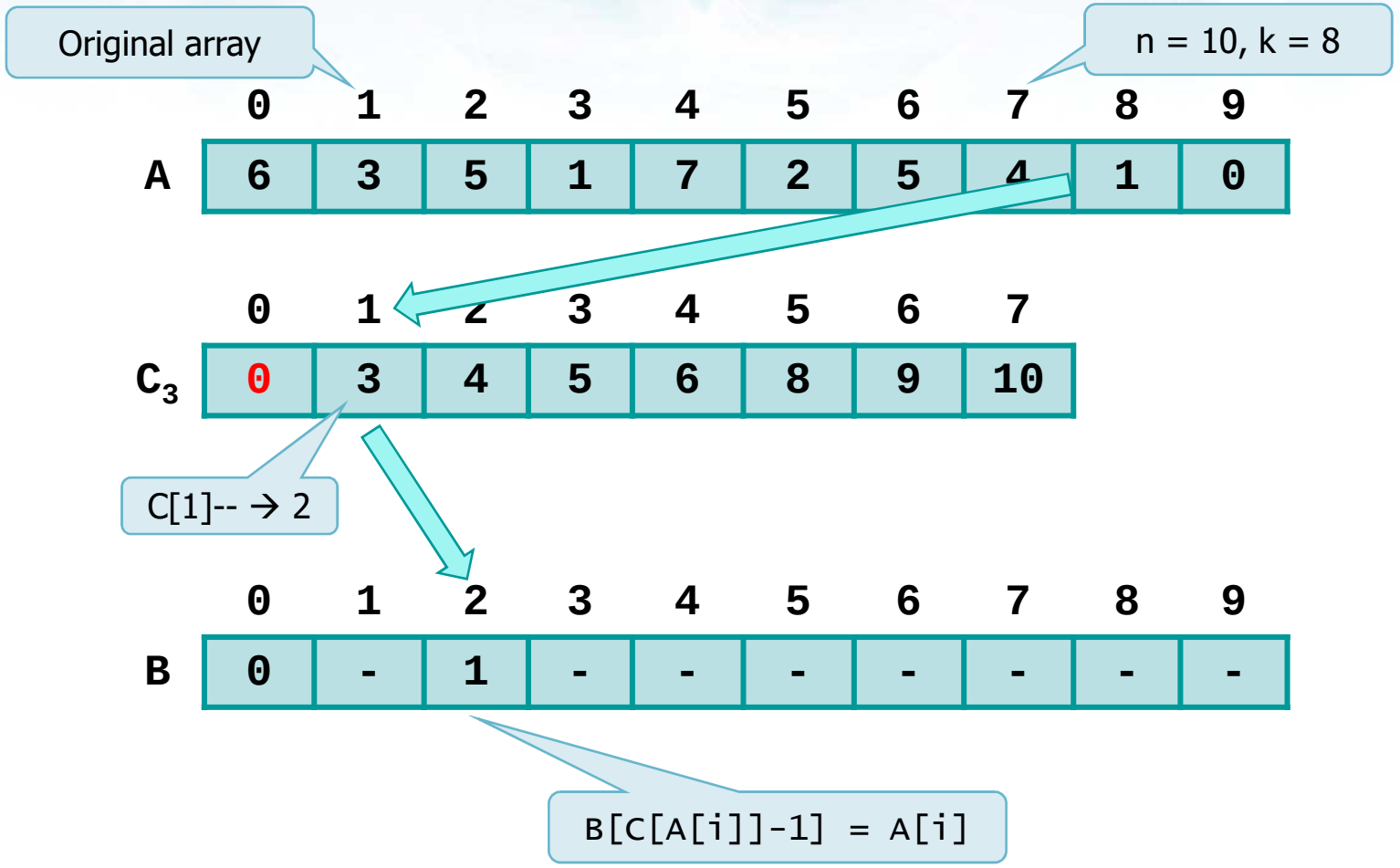
Example


Original array
n = 10, k = 8
0 1 2 3 4 5 6 7 8 9
A
6 3 5 1 7 2 5 4 1 0
0 1 2 3 4 5 6 7
C1
0 0 0 0 0 0 0 0
Simple occurrences
0 1 2 3 4 5 6 7
C2
1 2 1 1 1 2 1 1
Multiple occurrences
0 1 2 3 4 5 6 7
C3
1 3 4 5 6 8 9 10

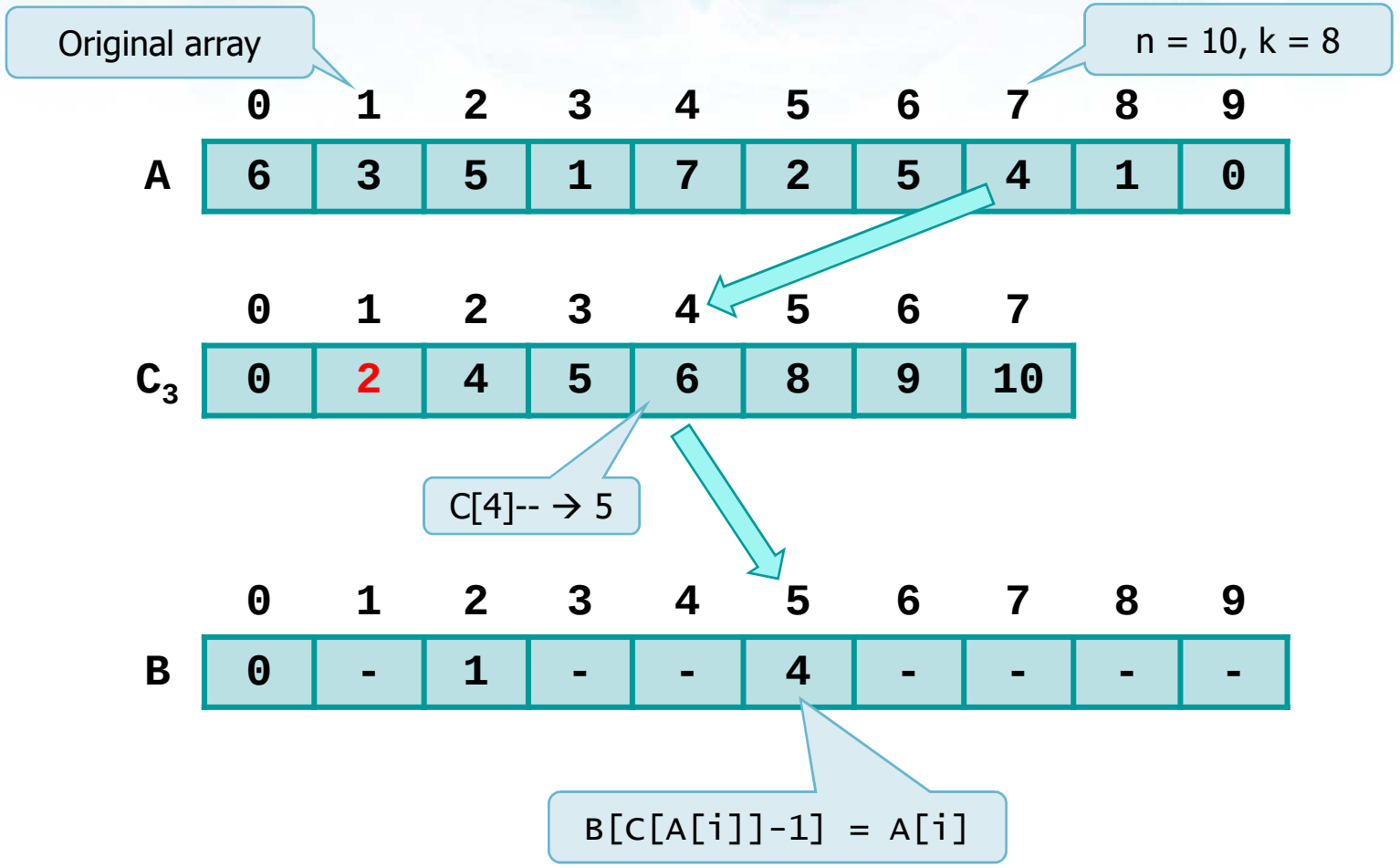
Example


Original array
n = 10, k = 8
0 1 2 3 4 5 6 7 8 9
A
6 3 5 1 7 2 5 4 1 0
0 1 2 3 4 5 6 7
C3
1 3 4 5 6 8 9 10
C[0]-- → 0
0 1 2 3 4 5 6 7 8 9
B
0 - - - - - - - - -
B[C[A[i]]-1] = A[i]

Example


Original array
n = 10, k = 8
0 1 2 3 4 5 6 7 8 9
A 6 3 5 1 7 2 5 4 1 0
0 1 2 3 4 5 6 7
C3 0 3 4 5 6 8 9 10
C[1]-- → 2
0 1 2 3 4 5 6 7 8 9
B 0 - 1 - - - - - - -
B[C[A[i]]-1] = A[i]

Example


Original array
n = 10, k = 8
0 1 2 3 4 5 6 7 8 9
A
6 3 5 1 7 2 5 4 1 0
0 1 2 3 4 5 6 7
C3
0 2 4 5 6 8 9 10
C[4]-- → 5
0 1 2 3 4 5 6 7 8 9
B
0 - 1 - - 4 - - - -
B[C[A[i]]-1] = A[i]

C Code

```
#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];
}
```

Complexity

- Intuitive 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)
  - Overall
    - T(n) = O(n+k)
  - Applicability
    - k=O(n)
    - Thus T(n) = O(n)

```
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];
```

Features

- Not in place
- Stable
  - As it copies element from A to B sgtarting from the last one it is guaranteed that the relative order with elements with the same key is maintained