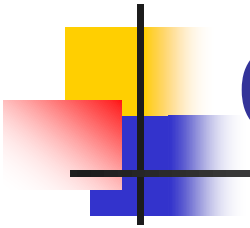
Gli algoritmi di ordinamento

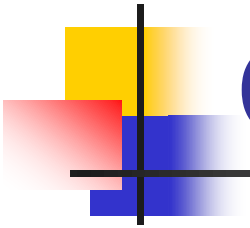
Paolo Camurati

Dip. Automatica e Informatica
Politecnico di Torino



Classificazione: interni/esterni

- Ordinamento interno
  - dati in memoria centrale
  - accesso diretto agli elementi
- Ordinamento esterno
  - dati in memoria di massa
  - accesso sequenziale agli elementi



Classificazione: in loco / stabili

- Ordinamento in loco

  vettore di n dati + locazioni di memoria ausiliarie in numero fisso

- Ordinamento stabile

  immutato l'ordinamento relativo di dati con ugual valore della chiave

Esempio

- Record con 2 chiavi: nome (la chiave è la prima lettera) e gruppo (la chiave è un intero)

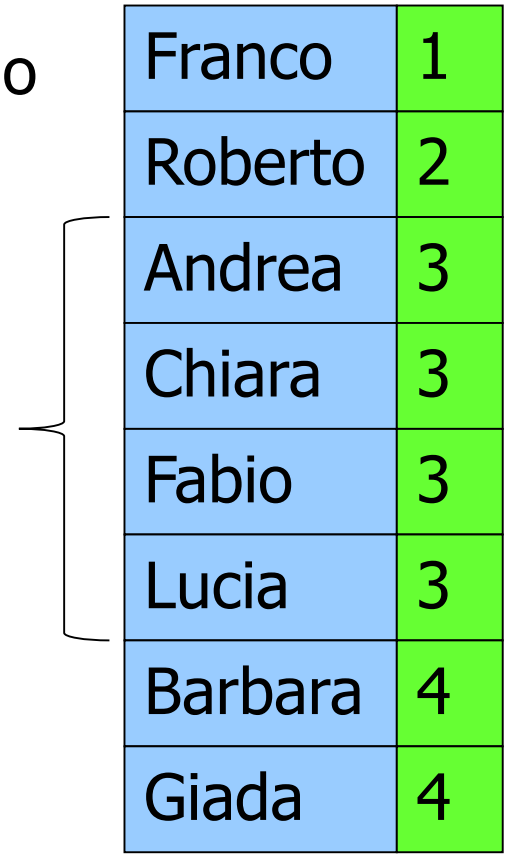
Primo ordinamento per prima lettera:

Andrea	3
Barbara	4
Chiara	3
Fabio	3
Franco	1
Giada	4
Lucia	3
Roberto	2

Secondo ordinamento per gruppo:

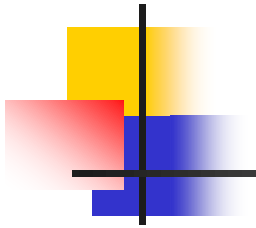
Algoritmo
NON stabile

Algoritmo
stabile



Classificazione: complessità

- $O(n^2)$:
  semplici, iterativi, basati sul confronto
  Insertion sort, Selection sort, Exchange/Bubble sort
- $O(n^{3/2})$: Shellsort (con certa scelta di sequenza)
- $O(n \log n)$:
  più complessi, ricorsivi, basati sul confronto
  Merge sort, Quicksort, Heapsort
- $O(n)$:
  applicabili solo con ipotesi restrittive sui dati, basati sul calcolo
  Counting sort, Radix sort, Bin/Bucket sort

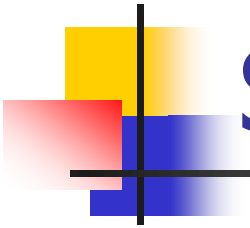


E’ possibile un’analisi più raffinata, in cui si distinguono le operazioni di:

- confronto
- scambio.

Quando infatti il dato da ordinare occupa molta memoria, lo spostamento può essere costoso.

La complessità asintotica comunque non cambia.



Struttura dei dati

- Elementi da ordinare: tipo `Item` definito come `struct`
- uno dei campi = *chiave* di ordinamento
- restanti campi = dati aggiuntivi
- funzioni di interfaccia a oggetti di tipo `Item`
  - lettura/scrittura
  - generazione di valori casuali
  - accesso alla chiave
- operatori relazionali su oggetti di tipo `Item`

Esempio

```
...
#define maxKey 100

typedef struct { int x; int y; } Item;
typedef int Key;

Key key(Item x);
int eq(Item A, Item B);
int neq(Item A, Item B);
int less(Item A, Item B);
int greater(Item A, Item B);
Item ITEMscan();
void ITEMshow(Item x);
Item ITEMrand();
```

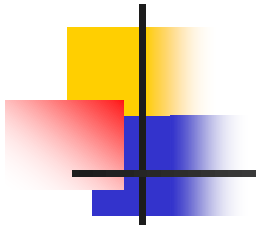
Definizione di un nuovo tipo Item

Definizione di un nuovo tipo Key

Prototipi di funzioni su dati di tipo Item

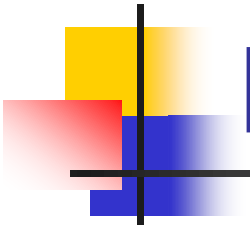
Implementazione di funzioni su dati di tipo Item

```
...
int eq(Item A, Item B) {
  return (key(A) == key(B));
}

int neq(Item A, Item B) {
  return (key(A) != key(B));
}

int less(Item A, Item B) {
  return (key(A) < key(B));
}

int neq(Item A, Item B) {
  return (key(A) > key(B));
}
```



Implementazione di funzioni su dati di tipo Item

```
Item ITEMscan(){
  Item item;
  printf("x = "); scanf("%d", &item.x);
  printf("y = "); scanf("%d", &item.y);
  return item;
}
void ITEMshow(Item item) {
  printf("a.x= %6d \t a.y= %6d \n", item.x, item.y);
}
Key key(Item item) {
  return item.x;
}
Item ITEMrand() {
  Item item={maxKey*rand()/RAND_MAX, maxKey*rand()/RAND_MAX};
  return item;
}
```



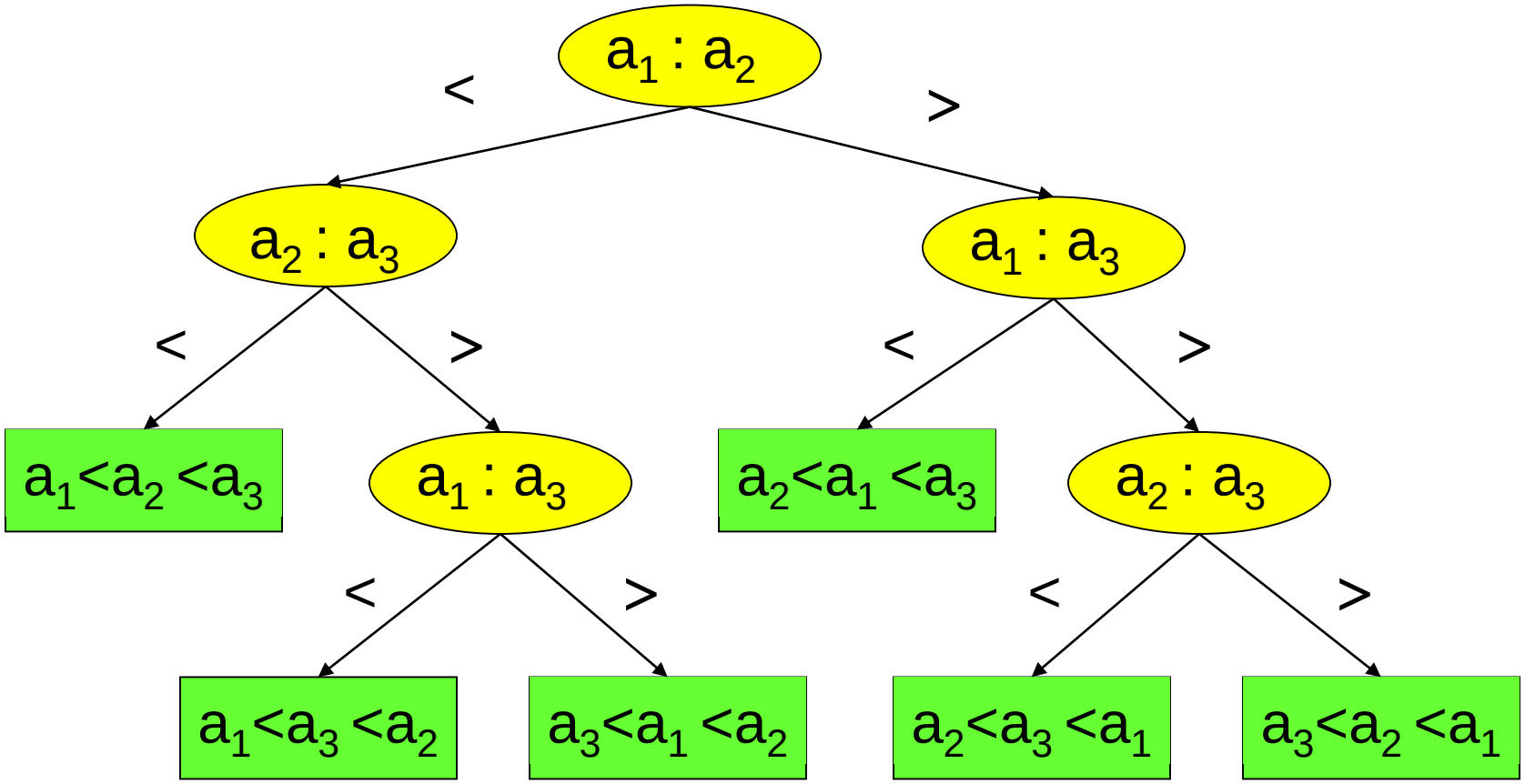
Limite inferiore

Algoritmi basati sul confronto

- operazione elementare: confronto $a_i : a_j$
- esito: decisione ($a_i > a_j$ o $a_i \leq a_j$), riportata su un albero delle decisioni.

Esempio

Ordinare il vettore a_1, a_2 , a_3 di elementi distinti



Generalizzazione

- Per n interi distinti: numero di ordinamenti = numero di permutazioni n!
- Complessità: numero h di confronti (altezza dell'albero)
- Ogni soluzione = foglia
- Numero di foglie = 2^h
- Approssimazione di Stirling: $n! > (n/e)^n$

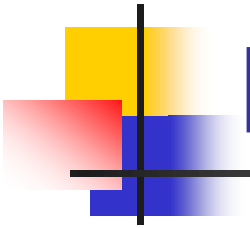
$$2^h \geq n! > (n/e)^n$$

$$h > \lg(n/e)^n = n \lg n - n \lg e = \Omega(n \lg n)$$

Gli algoritmi iterativi di ordinamento

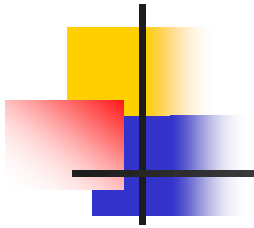
Paolo Camurati

Dip. Automatica e Informatica
Politecnico di Torino



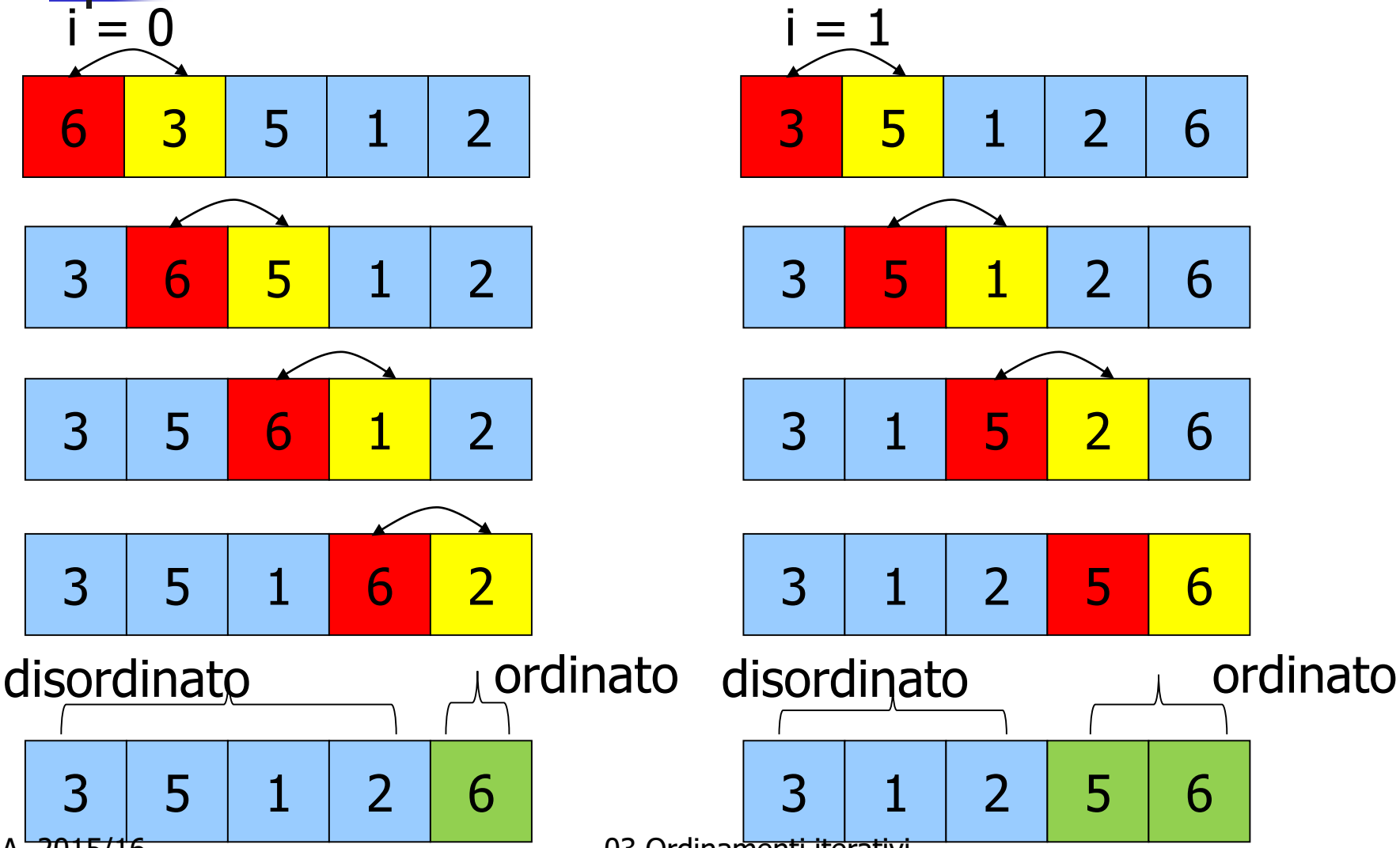
Bubble Sort

- Dati: interi in un vettore A con indici compresi tra l e r
- Vettore diviso in 2 sotto-vettori:
  - di destra: ordinato, inizialmente vuoto
  - di sinistra: disordinato, inizialmente coincide con A.
- Operazione elementare: confronto tra elementi successivi del vettore A[j] e A[j+1], scambio se A[j] > A[j+1]



- Approccio incrementale: iterazione i: il massimo del sotto-vettore SX (A_l ... A_{r-i+l}) è assegnato a A[r-i+l]; incremento di i. Il sotto-vettore DX ordinato cresce di 1 posizione verso SX, dualmente quello di SX decresce di 1 posizione
- Terminazione: tutti gli elementi inseriti ordinatamente
- Possibili ottimizzazioni: flag che indica se vi sono stati scambi, interrompendo anticipatamente il ciclo.

Esempio


i = 0
6 3 5 1 2
3 6 5 1 2
3 5 6 1 2
3 5 1 6 2
disordinato
ordinato
3 5 1 2 6
i = 1
3 5 1 2 6
3 5 1 2 6
3 1 5 2 6
3 1 2 5 6
disordinato
ordinato
3 1 2 5 6

```
void BubbleSort(Item A[], int l, int r){
  int i, j;
  Item temp;
  for( i = l; i < r; i++) {
     for (j = l; j < r - i +l; j++)
       if (greater(A[j], A[j+1])) {
         temp = A[j];
         A[j] = A[j+1];
        A[j+1] = temp;
       }
   }
  return;
}
```

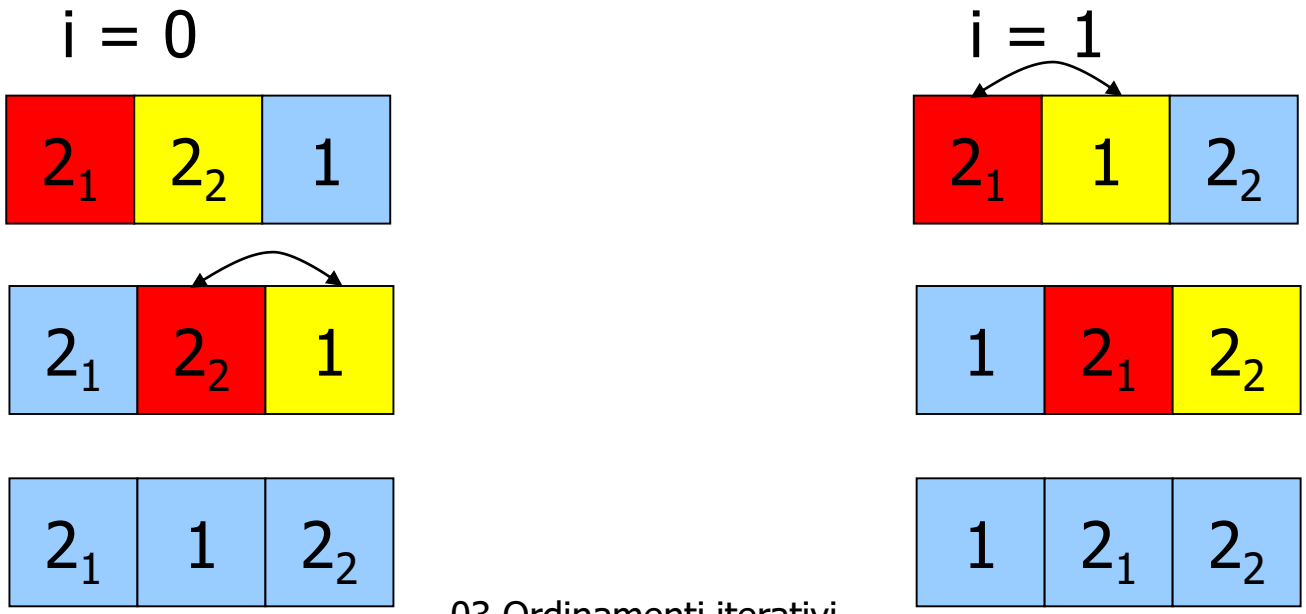
IterativeSort.c

i-l è il numero di caselle a destra già ordinato


```
void OptBubbleSort(Item A[], int l, int r) {
   int i, j, flag;
  Item temp;
   flag = 1;
  for(i = l; i < r && flag==1; i++) {
     flag = 0;
    for (j = l; j < r - i + l; j++)
      if (greater(A[j], A[j+1])) {
        flag = 1;
        temp = A[j];
        A[j] = A[j+1];
       A[j+1] = temp;
    }
   }
   return;
}
```

Caratteristiche

- In loco
- Stabile: tra più chiavi duplicate quella più a DX prende la posizione più a DX e non viene mai «scavalcata» a DX da un'altra chiave uguale:



Analisi asintotica di caso peggiore

- Due cicli annidati:
  - esterno: eseg
  - interno: all'i iene eseguito n-1-i volte

progressione aritmetica finita di ragione 1 (Gauss, fine XVII sec.)

$T(n) = (n-1) + (n-2) + \ldots 2 + 1$

$= \Sigma_{1 \leq i < n} 1 = n(n-1)/2$

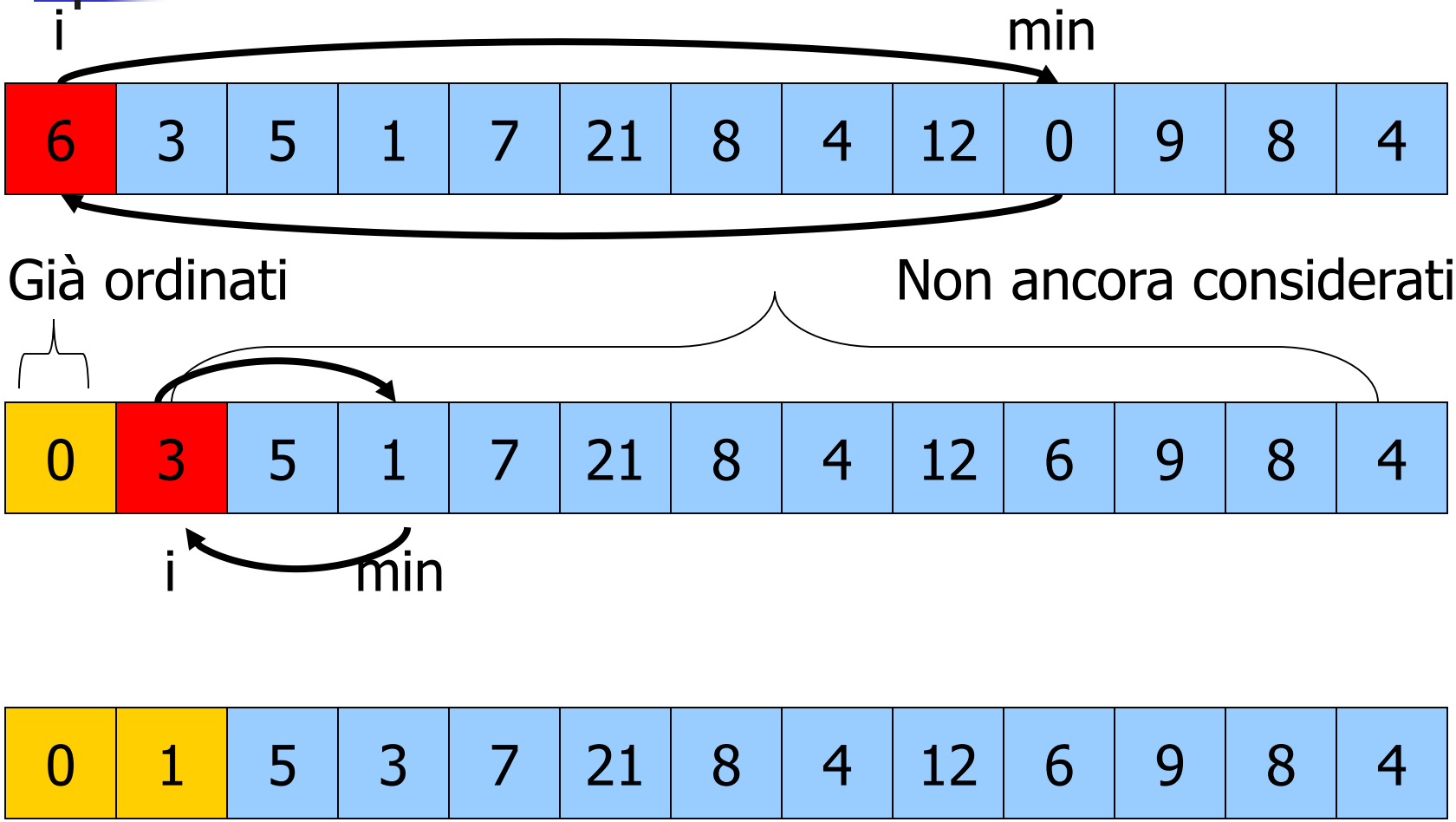
$T(n) = O(n^2)$.

- Scambi nel caso peggiore: $O(n^2)$
- Confronti nel caso peggiore: $O(n^2)$

Selection sort

- Dati: interi in un vettore A con indici compresi tra l e r
- Vettore diviso in 2 sotto-vettori:
  - di sinistra: ordinato, inizialmente vuoto
  - di destra: disordinato, inizialmente coincide con A
- Approccio incrementale: iterazione i: il minimo del sotto-vettore DX (A_i ... A_r) è assegnato a A[i]; incremento di i
- Terminazione: tutti gli elementi inseriti ordinatamente
- La ricerca del minimo nel sottovettore DX comporta la sua scansione.

Esempio

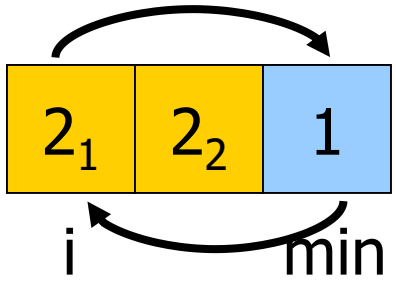

i
min
6 3 5 1 7 21 8 4 12 0 9 8 4
Già ordinati
Non ancora considerati
0 3 5 1 7 21 8 4 12 6 9 8 4
i
min
0 1 5 3 7 21 8 4 12 6 9 8 4

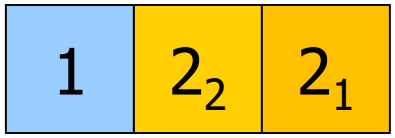
```
void SelectionSort(Item A[], int l, int r) {
   int i, j, min;
   Item temp;
   for(i = l; i < r; i++) {
      min = i;
      for (j = i+1; j <= r; j++)
         if (less(A[j], A[min]))
            min = j;
      temp = A[i];
      A[i] = A[min];
      A[min] = temp;
   }
   return;
}
```

IterativeSort.c

Caratteristiche

- In loco
- Non stabile: uno scambio tra elementi «lontani» può far sì che un'occorrenza di una chiave duplicata si sposti a SX di un'occorrenza precedente della stessa chiave «scavalcandola»:

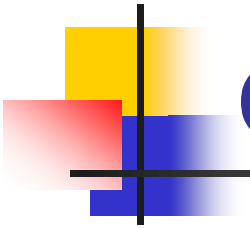




Analisi asintotica di caso peggiore

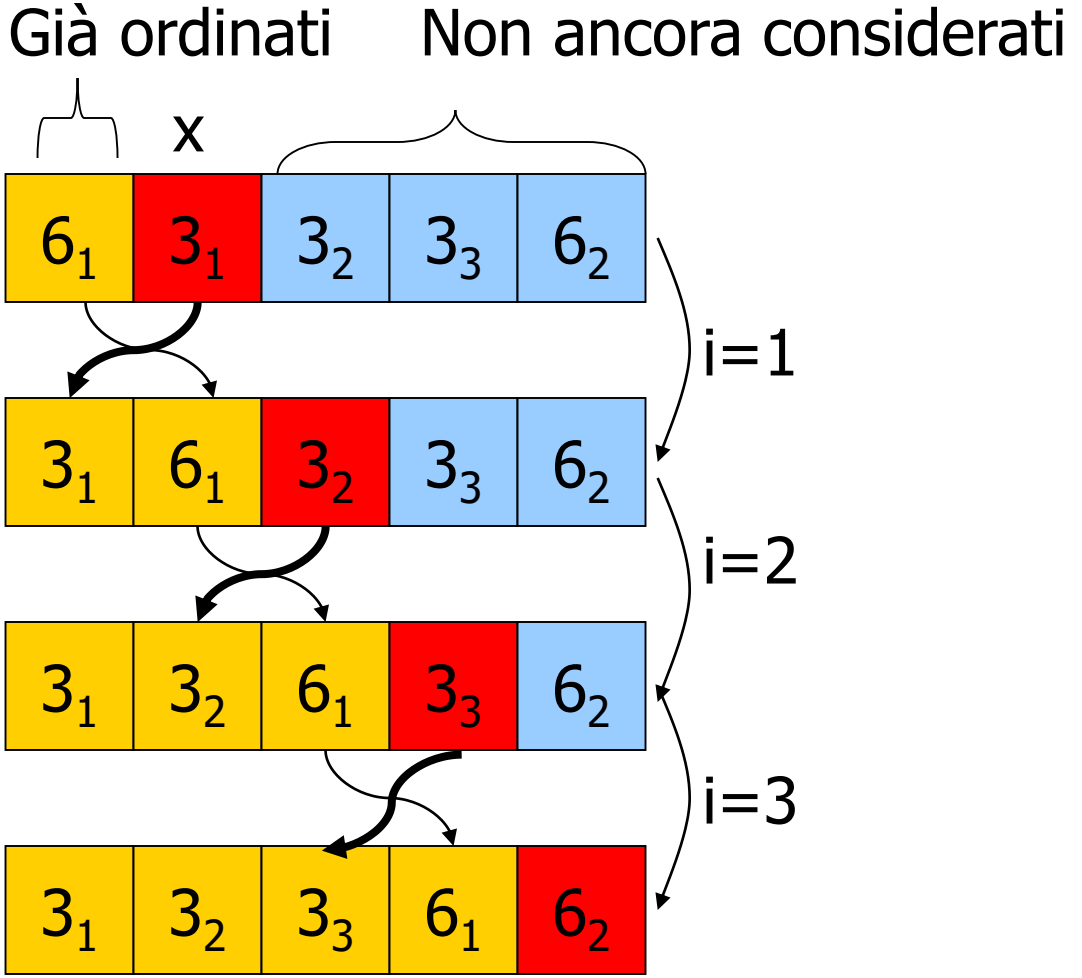
- Due cicli annidati:
  - esterno: eseguito n-1 volte
  - interno: all'i-esima iterazione viene eseguito n-i-1 volte:

    $T(n) = (n-1) + (n-2) + \ldots 2 + 1 = O(n^2)$
- Scambi nel caso peggiore: $O(n)$
- Confronti nel caso peggiore: $O(n^2)$

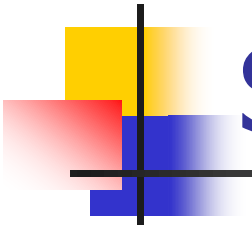


Caratteristiche dell'Insertion sort

- In loco
- Scambi nel caso peggiore: $O(n^2)$
- Confronti nel caso peggiore: $O(n^2)$
- Stabile: se l'elemento da inserire è una chiave duplicata, non può mai «scavalcare» a SX un'occorrenza precedente della stessa chiave:


Già ordinati
Non ancora considerati
x
6_1 3_1 3_2 3_3 6_2
i=1
3_1 6_1 3_2 3_3 6_2
i=2
3_1 3_2 6_1 3_3 6_2
i=3
3_1 3_2 3_3 6_1 6_2


```
void InsertionSort(Item A[], int l, int r) {
  int i, j;
  Item x;
  for(i = l+1; i <= r; i++) {
    x = A[i];
    j = i - 1;
    while (j >= l && less(x, A[j])) {
      A[j+1] = A[j];
       j--;
    }
    A[j+1] = x;
  }
}
```



Shellsort (Shell, 1959)

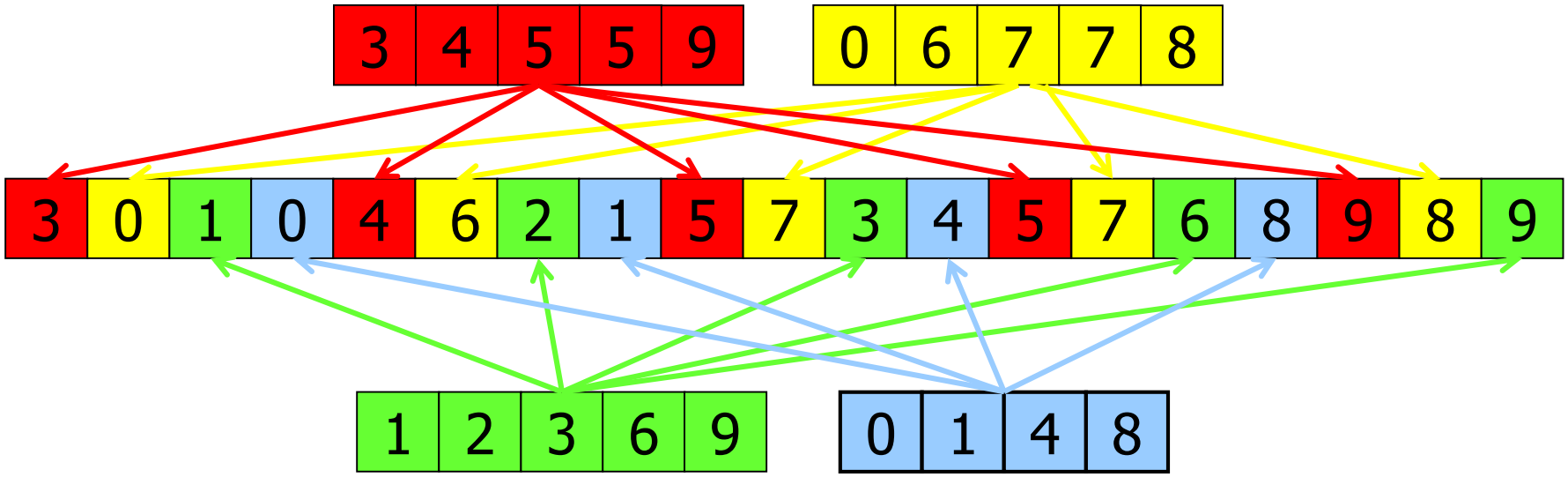
Limite dell'insertion sort: il confronto, quindi lo scambio, avviene solo tra elementi adiacenti.

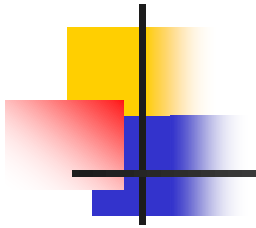
Idea dello Shellsort:

- confrontare, quindi eventualmente scambiare, elementi a distanza h tra di loro
- definendo una sequenza decrescente di interi h che termina con 1.

Si dice h-ordinato un array formato da h sottosequenze non contigue ordinate composte da elementi che distano tra di loro h.

Esempio con h=4 (sottosequenze ordinate e non contigue)





Per ognuna delle sottosequenze si applica l'insertion sort. Gli elementi della sottosequenza sono quelli a distanza h da quello corrente.

```
for i = l+h; i <= r; i++) {
  int j = i, temp = A[i];
  while (j>= l+h && less(temp, A[j-h])) {
    A[j] = A[j-h];
    j -=h;
  }
  A[j] = temp;
}
```

Esempio

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

sequenza h: 13, 4, 1

Passo 1: h=13

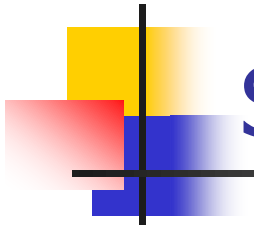
5	1	8	0	0	2	3	1	3	7	6	4	5	7	8	9	9	4	6

Passo 2: h=4

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

Passo 3: h=1

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



Scelta della sequenza

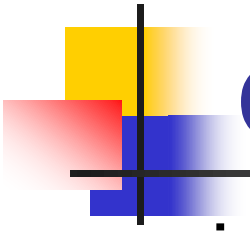
- Influenza le prestazioni
- Sequenza di Knuth:

  h = 3*h+1 = 1 4 13 40 121 …
- Sequenza

  h = 1 e poi $4^{i+1} + 3*2^{i} +1$ = 1 8 23 77 281 1073 …
- Sequenza di Sedgewick:

  h = 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, ...


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



Caratteristiche

- in loco
- non stabile: uno scambio tra elementi «lontani» può far sì che un'occorrenza di una chiave duplicata si sposti a SX di un'occorrenza precedente della stessa chiave «scavalcandola»:

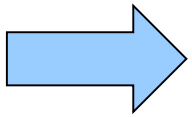
Esempio

2_1	2_2	2_3	2_4	2_5	0

sequenza h: 4, 1

- Passo 1: h=4

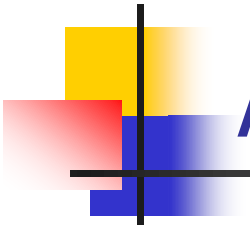
2_1	2_2	2_3	2_4	2_5	0



2_1	0	2_3	2_4	2_5	2_2

- Passo 2: h=1

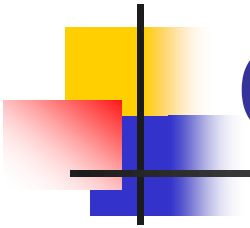
0	2_1	2_3	2_4	2_5	2_2



Analisi asintotica di caso peggiore

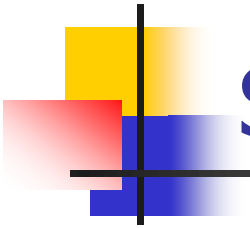
Shellsort:

- con la sequenza di Knuth: 1 4 13 40 121 … esegue meno di $n^{3/2}$ confronti, quindi $O(n^{3/2})$
- con la sequenza 1 8 23 77 281 1073 … esegue meno di $n^{4/3}$ confronti, quindi $O(n^{4/3})$
- con la sequenza originale di Shell 1 2 4 8 16 … può degenerare a n^2, quindi $O(n^2)$



Counting sort

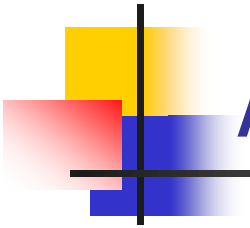
- Ordinamento per calcolo: determinare, per ciascun elemento da ordinare x, quanti elementi sono minori o uguali a x
- x direttamente assegnato alla posizione finale
- Caratteristiche: stabile, non in loco.



Strutture dati

Si usano 3 vettori:

- Vettore di partenza: A[0..n-1] di n interi
- Vettore risultato: B [0..n-1] di n interi
- Vettore delle occorrenze: C di k interi se i dati sono nell'intervallo [0..k-1]



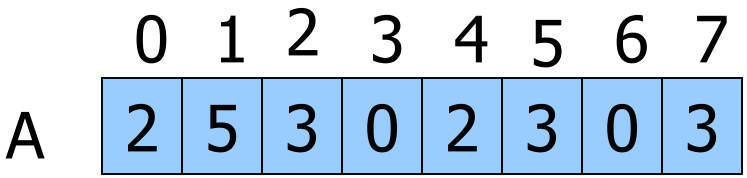
Algoritmo

- Passo 1: occorrenze semplici:
  C[i] = numero di elementi di A pari ad i
- Passo 2: occorrenze multiple:
  C[i] = numero di elementi di A <= i
- Passo 3: $\forall$ j
  C[A[j]] = numero di elementi <= A[j]
  quindi posizione finale di A[j] in B:

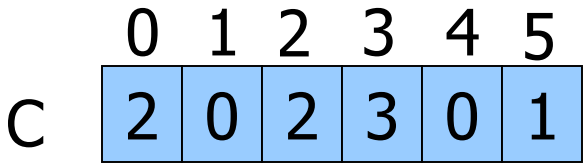
  B[C[A[j]]] = A[j]

  (attenzione agli indici in C, cfr codice!)

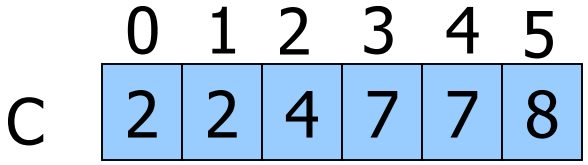
Esempio (n=8, k=6)



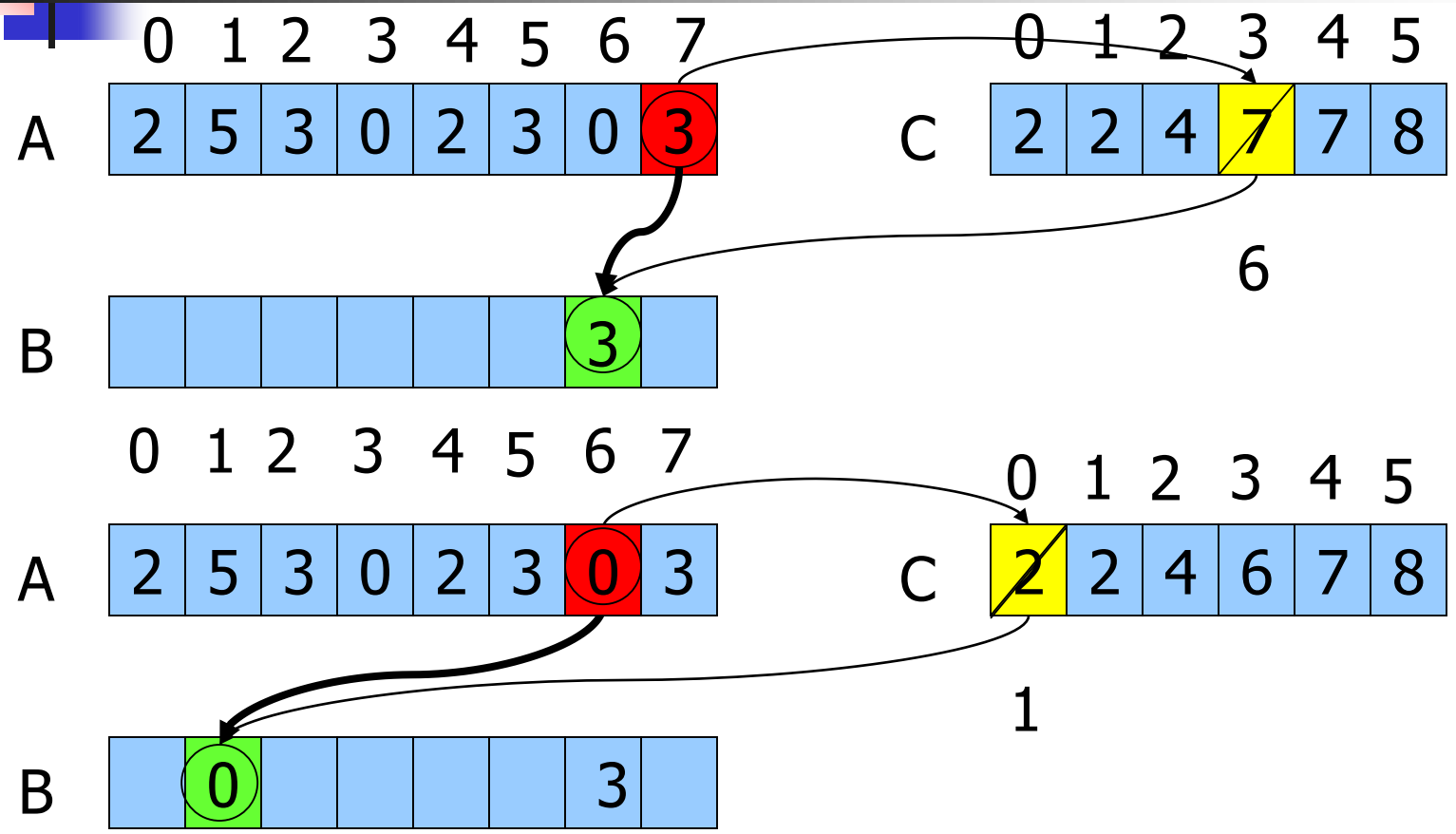
Vettore da ordinare



Occorrenze semplici



Occorrenze multiple


0 1 2 3 4 5 6 7
A
2 5 3 0 2 3 0 3
B
3
0 1 2 3 4 5
C
2 2 4 7 7 8
6
0 1 2 3 4 5 6 7
A
2 5 3 0 2 3 0 3
B
0 3
0 1 2 3 4 5
C
2 2 4 6 7 8
1

```
#define MAX 100
```

Vettori sovrallocati staticamente

IterativeSort.c

```
void CountingSort(Item A[], int l, int r, int k) {
  int i, n, C[MAX];
  Item B[MAX];
  n = r - l + 1;
.....
}

void CountingSort(Item A[], int l, int r, int k) {
  int i, n, *C;
  Item *B;
  n = r - l + 1;
  B = malloc(n*sizeof(Item)); C = malloc(k*sizeof(int));
.....
}
```

Vettori allocati dinamicamente

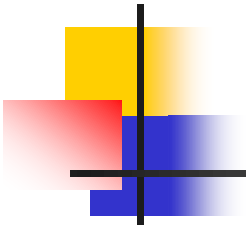
```
  for (i = 0; i < k; i++)
    C[i] = 0;
  for (i = l; i <= r; i++)
    C[A[i]]++;
  for (i = 1; i < k; i++)
    C[i] += C[i-1];
  for (i = r; i >= l; i--) {
    B[C[A[i]]-1] = A[i];
    C[A[i]]--;
  }
  for (i = l; i <= r; i++)
    A[i] = B[i];
}
```

Analisi asintotica di caso peggiore

- Ciclo di inizializzazione di C: O(k)
- Ciclo di calcolo delle occorrenze semplici: O(n)
- Ciclo di calcolo delle occorrenze multiple: O(k)
- Ciclo di ricopiatura in B: O(n)
- Ciclo di ricopiatura in A: O(n)

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

Applicabilità: k=O(n), quindi T(n) = O(n).



Riferimenti

- Limite inferiore:
  - Cormen 9.1
- Bubble sort:
  - Sedgewick 6.4
  - Deitel 6.6
- Selection sort:
  - Sedgewick 6.2
- Shell sort:
  - Sedgewick 6.6
- Counting sort:
  - Cormen 9.2