

Algoritmi di ORDINAMENTO

1) INSERTION-SORT (A)

ASD-1 L3

for $j \leftarrow 2$ to $A.length$

$key \leftarrow A[j]$

$i \leftarrow j - 1$

 while ($i > 0$ AND $A[i] > key$)

$A[i + 1] \leftarrow A[i]$

$i \leftarrow i - 1$

$A[i + 1] \leftarrow key$

2) MERGE-SORT (A, p, q)

ASD-1 L4

if $p < q$

then $r \leftarrow [(p+q)/2]$

 MERGE-SORT (A, p, r)

 MERGE-SORT (A, r+1, q)

 MERGE (A, p, q, r)

MERGE SUL POSTO CON SENTINELLE DI CORMEN

MERGE (A, p, q, r)

$n_1 \leftarrow q - p + 1$

$n_2 \leftarrow r - q$

Crea due nuovi array $L[1..n_1+1]$ e $R[1..n_2+1]$

for $i \leftarrow 1$ to n_1

$L[i] \leftarrow A[p + i - 1]$

for $j \leftarrow 1$ to n_2

$R[j] \leftarrow A[q + j]$

$L[n_1 + 1] \leftarrow +\infty$

$R[n_2 + 1] \leftarrow +\infty$

$i \leftarrow 1$

$j \leftarrow 1$

for $k \leftarrow p$ to r

if $L[i] \leq R[j]$

$A[k] \leftarrow L[i]$

$i \leftarrow i + 1$

else $A[k] \leftarrow R[j]$

$j \leftarrow j + 1$

3) Quick Sort (A, p, r)

ASD-1 L7

```
if p < r  
    q ← Partition(A, p, r)  
    Quicksort(A, p, q)  
    Quicksort(A, q+1, r)
```

PARTITION (A, p, r) VERSIONE HOARE

```
x ← A[p]  
i ← p - 1  
j ← r + 1  
while TRUE  
do REPEAT j ← j - 1  
    UNTIL A[j] ≤ x  
do REPEAT i ← i + 1  
    UNTIL A[i] ≥ x  
if i < j then SCAMBIA A[i] con A[j]  
else return j
```

PARTITION (A, p, r)

```
x ← A[r]  
i ← p - 1  
for j ← p to r - 1  
do if A[j] ≤ x  
    then i ← i + 1  
        scambia A[i] <=> A[j]  
scambia A[i+1] <=> A[r]  
return i + 1
```

4) RANDOMIZED_Quicksort (A, p, r) ASD-1 L7

if $p < r$

$q \leftarrow \text{RANDOMIZED-PARTITION}(A, p, r)$

$\text{RANDOMIZED-Quicksort}(A, p, q)$

$\text{RANDOMIZED-Quicksort}(A, q+1, r)$

RANDOMIZED_PARTITION (A, p, r)

$i \leftarrow \text{Random}(p, r)$

Scambia $A[i]$ con $A[r]$

PARTITION (A, p, r)

5) HEAP SORT

1) MAX-HEAPIFY (A, i)

ASD-1 L8

$\ell \leftarrow \text{Left}[i]$

$r \leftarrow \text{Right}[i]$

if $A[i] < A[\ell]$ AND $\ell \leq A.\text{heapsize}$

 then $\max \leftarrow \ell$

 else $\max \leftarrow i$

if $A[\max] < A[r]$ AND $r \leq A.\text{heapsize}$

 then $\max \leftarrow r$

if $\max \neq i$

 then scambia $A[i] \leftrightarrow A[\max]$

MAX-HEAPIFY (A, \max)

2) BUILD-MAX-HEAP (A)

ASD-1 LEZ9

$A.\text{heapsize} \leftarrow A.\text{length}$

for $i \leftarrow \left\lfloor \frac{A.\text{length}}{2} \right\rfloor$ downto 1

 MAX-HEAPIFY (A, i)

3) HEAP-SORT (A)

BUILD-MAX-HEAP (A)

for $j \leftarrow A.\text{length}$ downto 2

 SCAMBIA $A[1] \leftrightarrow A[j]$

$A.\text{heapsize} \leftarrow A.\text{heapsize} - 1$

 MAX-HEAPIFY (A, 1)

4) HEAP_MAXIMUM (A)

return A[1]

5) HEAP_EXTRACT_MAX (A)

if A.heapsize < 1

error UNDERFLOW dell' HEAP

max \leftarrow A[1]

A[1] = A.heapsize

A.heapsize \leftarrow A.heapsize - 1

MAX-HEAPIFY (A, 1)

return max

6) HEAP_INCREASE_KEY (A, i, key)

if key < A[i]

error "INVALID DATA"

A[i] \leftarrow key

while i > 1 AND A[parent(i)] < A[i]

SCAMBIA A[parent(i)] \leftrightarrow A[i]

i \leftarrow parent(i)

7) MAX-HEAP-INSERT (A , key)

$A.\text{heapsize} \leftarrow A.\text{heapsize} + 1$

$A[A.\text{heapsize}] \leftarrow -\infty$

HEAP-INCREASE-KEY (A , $A.\text{heapsize}$, key)

6) BUCKET-SORT (A)

ASD-1 L9

Sia $B[0 \dots n-1]$ un nuovo array

$n \leftarrow A.length$

for $i \leftarrow 0$ to $n-1$

crea la lista vuota $B[i]$

for $i \leftarrow 1$ to n

inserisci $A[i]$ nella lista $B[i]$

for $j \leftarrow 0$ to $n-1$

ORDINA la lista $B[i]$ con INSERTION SORT

CONCATENA le liste $B[0], B[1], \dots, B[n-1]$ in ORDINE

ALGORITMI DI ORDINAMENTO ASD-2

1) BUBBLE-SORT

ASD-2 L6

```
void bubble(int a[], int l, int r)
Σ for (int i = l; i < r; i++)
    for (int j = r; j > i; j--)
        compswap(a[j-1], a[j]);
```

Σ

```
void compswap(int &A, int &B)
```

```
Σ if (B < A) swap(A, B);
```

Σ

```
void swap(int &A, int &B)
```

```
Σ int t = A;
```

```
    A = B;   B = t;
```

Σ

2) INSERTION-SORT , come ASD-1

3) IL PROBLEMA DEL MASSIMO SOTTOARRAY

FIND-MAXIMUM-SUBARRAY ($A, low, high$)

if $high == low$

 return ($low, high, A[low]$)

else $mid = \lfloor (low + high) / 2 \rfloor$

$(left_low, left_high, left_sum) = \text{FIND-MAXIMUM-SUBARRAY}(A, low, mid)$

$(right_low, right_high, right_sum) = \text{FIND-MAXIMUM-SUBARRAY}(A, mid + 1, high)$

$(cross_low, cross_high, cross_sum) = \text{FIND-MAX-CROSSING-SUBARRAY}(A, low, mid, high)$

if $left_sum \geq right_sum$ AND $left_sum \geq cross_sum$

 return ($left_low, left_high, left_sum$)

else if $right_sum \geq left_sum$ AND $right_sum \geq cross_sum$

 return ($right_low, right_high, right_sum$)

else return ($cross_low, cross_high, cross_sum$)

Il problema del massimo sottoarray

FIND-MAX-CROSSING-SUBARRAY ($A, low, mid, high$)

// Find a maximum subarray of the form $A[i..mid]$.

$left_sum = -\infty$

$sum = 0$

for $i = mid$ downto low

$sum = sum + A[i]$

 if $sum > left_sum$

$left_sum = sum$

$max-left = i$

// Find a maximum subarray of the form $A[mid + 1..j]$.

$right_sum = -\infty$

$sum = 0$

for $j = mid + 1$ to $high$

$sum = sum + A[j]$

 if $sum > right_sum$

$right_sum = sum$

$max-right = j$

// Return the indices and the sum of the two subarrays.

return ($max-left, max-right, left_sum + right_sum$)

4) MERGE-SORT come ASD-1
+ MERGE A DUE VIE

Merge a due vie (pseudocodice)

```
A=a1, ... an
B=b1,...bp
C=c1,...cn+p
While A!= Insieme-Vuoto AND B!= Insieme-Vuoto do
    if a1 > b1
        APPENDI(b1 in C)
        CANCELLA (b1 in B)
    else
        APPENDI(a1 in C)
        CANCELLA (a1 in A)
    if A = Insieme-Vuoto
        APPENDI (B in C)
    else
        APPENDI (A in C)
```

Merge astratto sul posto

- La procedura di merge in questo caso memorizza il risultato della combinazione di $a[1], \dots, a[m]$ con $a[m+1], \dots, a[r]$ in un singolo insieme ordinato, mantenendo il risultato in $a[1] \dots a[r]$



```
typedef int Item;
void merge(Item a[], int left, int center, int right) {
    const int n=8;
    static Item aux[n];
    int i,j;
    for (i = center+1; i > left; i--)
        aux[i-1] = a[i-1];
    for (j = center; j < right; j++)
        aux[right+center-j] = a[j+1];
    for (int k = left; k <= right; k++)
        if (aux[j] < aux[i])
            a[k] = aux[j--];
        else
            a[k] = aux[i++];
}
```



```
void mergesort(Item a[ ], int left, int right) {
    if (left<right) {
        int center = (left+right)/2;
        mergesort(a, left, center);
        mergesort(a, center+1, right);
        merge(a, left, center, right);
    }
}

int main() {
    const int n=8;
    int a[n] = {10, 3, 15, 2, 1, 4, 9, 0};
    mergesort(a,0,n-1);
    return 0;
}
```

5) Quicksort come ASD-1

6) HYBRID-SORT

Combiniamo i miglioramenti:

- INSERTION-SORT per trattare gli array piccoli;
- METODO DELLA MEDIANA per evitare le partizioni sfavorevoli.

Hybridsort

```
static const int M=10;
void quicksort(int a[], int l, int r)
{
    if (r - l > M)
    {
        swap(a[(l+r)/2], a[r-1]);
        compswap(a[r-1], a[l]);
        compswap(a[r], a[l]);
        compswap(a[r], a[r-1]);
        int i = partition(a, l+1, r-1);
        quicksort(a, l, i-1);
        quicksort(a, i+1, r);
    }
    insertionsort(a, l, r)
}
```



Hybridsort

```
void swap( int &A, int &B)
{
    int t=A;
    A=B;
    B=t;
}

void compswap ( int &A, int &B)
{
    if (B > A) swap(A,B);
}
```

7) HEAPSORT come ASD-1

8) ORDINAMENTI IN TEMPO LINEARE

1) COUNTING SORT

L11

Counting Sort

```
min ← max ← A[1] //Calcolo degli elementi max e min
for i ← 2 to length[A] do
    if (A[i] > max) then max ← A[i]
    else if(A[i] < min) then min ← A[i]
//Costruzione dell'array C * crea un array C di dimensione
max - min + 1
for i ← 1 to length[C] do
    C[i] ← 0 //inizializza a zero gli elementi di C
for i ← 1 to length[A] do
    //aumenta il numero di occorrenze del valore
    C[A[i] - min+1] ← C[A[i] - min+1] + 1
```

Counting Sort

```
//Ordinamento in base al contenuto dell'array delle frequenze
C
k ← 1 //indice per l'array A
for i ← 1 to length[C] do
    while C[i] > 0 do
//scrive C[i] volte il valore (i + min-1) nell'array A
    A[k] ← i + min-1
    k ← k + 1
    C[i] ← C[i] - 1
```

Counting Sort (Cormen)

```
C[0..k]
for i=0 to k
    C[i]=0
for j=1 to A.length
    C[A[j]]=C[A[j]]+1
for i=1 to k
    C[i]=C[i]+C[i-1]
for j=A.length downto 1
    B[C[A[j]]]=A[j]
    C[A[j]]=C[A[j]]-1
```

2) RADIX SORT

Radix Sort

- Il Radix Sort è un algoritmo di ordinamento usato per ordinare record con chiavi multiple
- Un esempio di record con chiavi multiple è dato dalla data gg/mm/aaaa
 - Per ordinare per data si deve ordinare l'anno
 - a parità di anno si deve ordinare per mese
 - a parità di mese per giorno
- Un altro esempio di record a chiave multipla è dato dal considerare le cifre di un intero come chiavi separate



Radix Sort

- Il Radix Sort opera in modo contro intuitivo ordinando prima le cifre meno significative e poi quelle via via più significative
- Supponiamo di dover ordinare una sequenza di numeri a 3 cifre
- Utilizzando un ordinamento di tipo stabile possiamo procedere ordinando prima per le unità, poi le decine e in ultimo le centinaia
- Ad ogni passo la stabilità ci garantisce che le cifre precedenti sono già ordinate

Radix Sort

```
Radix-Sort(A, d)
for i = 1 to d do
    metodo di ordinamento stabile su cifra i
```

Radix Sort: prestazioni

- Se vogliamo ordinare 10^6 numeri a 3 cifre
 - con il radix sort si effettua per ogni dato 3 chiamate al counting sort
 - con algoritmi $O(n \lg n)$ per ogni dato si effettuano $\lg_2 10^6 = 20$ operazioni
- Andando a estrarre le costanti numeriche nascoste nella notazione asintotica si vede che il radix sort può essere conveniente
- Lo svantaggio sta nel fatto che il metodo non è un ordinamento in loco e ha bisogno di più del doppio della memoria dei metodi in loco

3) BUCKET SORT

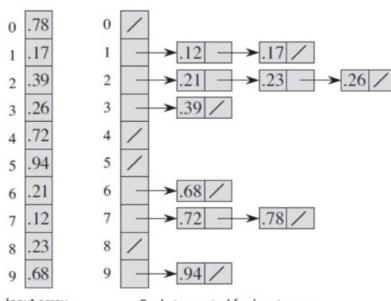
Bucket Sort

```
BUCKET_SORT (A)
n ← length [A]
for i = 1 to n do
    Inserisci A[i] nella lista B[ j = f(A[i]) ]
for j = 0 to m-1 do
    Ordina la lista B con insertion-sort
Concatena le liste B[0], B[1], . . . B[m-1] in ordine.
```

Bucket Sort (Cormen)

- Bucket sort assume che l'input sia estratto da una distribuzione uniforme
- Gli elementi dell'array da ordinare sono distribuiti uniformemente nell'intervallo [0,1)
- Si divide l'intervallo [0,1) in n sottointervalli (bucket)
- Gli input si distribuiranno più o meno egualmente nei vari bucket
- Ordinando i vari bucket e concatenandoli si ottiene l'ordinamento dell'array di input

Bucket Sort (Cormen)



Bucket Sort (Cormen)

BUCKET-SORT(A)

```
1 let  $B[0..n - 1]$  be a new array
2  $n = A.length$ 
3 for  $i = 0$  to  $n - 1$ 
4     make  $B[i]$  an empty list
5 for  $i = 1$  to  $n$ 
6     insert  $A[i]$  into list  $B[nA[i]]$ 
7 for  $i = 0$  to  $n - 1$ 
8     sort list  $B[i]$  with insertion sort
9 concatenate the lists  $B[0], B[1], \dots, B[n - 1]$  together in order
```

- Tutte le linee tranne la 8 costano $O(n)$ nel caso peggiore
- Dobbiamo calcolare il costo delle n chiamate ad insertion sort

