Pinotti Troia

# Tabella Riassuntiva

|  |  |
| --- | --- |
| Selection Sort |  |
| Insertion Sort |  |
| Merge Sort |  |
| Heap Sort |  |
| Quick Sort |  |
| Counting Sort |  |

# MergeSort

**Complessità in tempo:** nel caso peggiore

**:Codice**

MergeSort(A, p, r){

if(p < r){ // analogo a dire n > 1

q = int( (p+r)/2 )

MergeSort(A, p, q)

MergeSort(A, q+1, r)

Merge(A, p, q, r)

}

}

Merge(A, p, q, r){

n1 = q – p +1

n2 = (r – p+1) – n1

L = [1, …, n1 +1]

for i = 1 to n1{

L[i] = A[p+i-1]

}

L[n1+1] =

R = [1, …, n2+1]

for i = 1 to n2{

R[i] = A[q+i]

}

R[n2+1] =

i = 1, j = 1

for k = p to r{

if(L[i] < R[j]){

A[k] = L[i]

i++

}

else {

A[k] = R[j]

j++

}

}

}

Dividere sempre per 2 è la scelta migliore.  
L’algoritmo sfrutta della memoria aggiuntiva (nella funzione Merge) che potrebbe anche essere evitata.

# QuickSort

**Complessità in tempo:**

dipende dalla scelta del pivot,

* Nel caso pessimo pivot = max, .
* Nel caso pessimo in cui il pivot si alterna con il massimo e il minimo, .

Generalmente non capita molto spesso di imbattersi nei casi peggiori e la complessità in tempo nel caso medio è:

Non ho idea di cosa rappresenti

**Codice:**

QuickSort(A, p, r){

if(p < r){

q = partition(A, p, r)

QuickSort (A, p, q-1)

QuickSort (A, q+1, r)

}

}

Partition(A, p, r){

pivot = A[r]

i = p-1

for j = p to (r-1){

if(A[j] <= pivot){

i++

scambia(A[i], A[j])

}

}

i++

scambia(A[r], A[i])

return i

}

# CountingSort

**Complessità in tempo:**

Richiede due array supplementari: B[1, …, n] per l’output ordinato e C[1, …, k] per la memoria di lavoro temporanea.  
Per eseguire il CountingSort si presume che l’array di partenza A sia un array di interi del tipo A[1, …, n] (indicano gli indici). Si assume anche che il contenuto di A varia tra 0 e k.

Il countingsort non essendo un algoritmo di ordinamento per confronti ha limite inferiore (se ) e non come gli algoritmi per confronti.  
È un algoritmo Stabile: elementi con lo stesso valore compaiono nell’array di output con lo stesso ordine che avevano in quello di input.

**Codice:**

CountingSort(A, k){

C = [1, …, k]

for i = 1 to k{

C[i] = 0

}

for j = 1 to length(A){

C[A[j]] = C[A[j]] + 1

}

for i = 2 to k{

C[i] = C[i]+C[i-1]

}

B = [1, …, length(A)]

for j = length(A) to 1{

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

C[A[j]] = C[A[j]] -1

}

return B

}

# RadixSort (è tutto sbagliato)

**Complessità in tempo:**

L’algoritmo viene usato per ordinare numeri interi con d cifre in base b.

Il costo è:

L’algoritmo è ottimo quando e costa

**Codice:**

RadixSort(A, d){

for i = 1 to d{

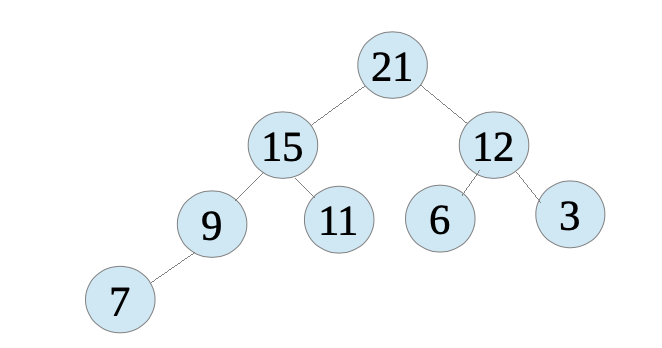
qualcosa di non ben definito

}

}

# **Heap**

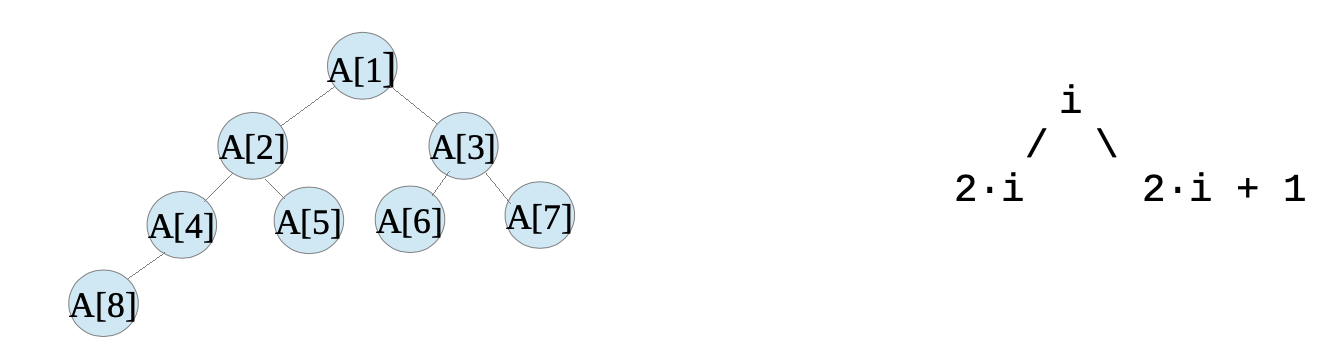
È un albero binario completo fino al penultimo livello e nell’ultimo le foglie sono addossate a sinistra.



Ha una forma ben definita e può essere:

* un max-heap: ha come chiave della radice il massimo
* un min-heap: ha come chiave della radice il minimo

Questa struttura permette di fare un algoritmo di ordinamento sul posto con costo e permette di calcolare il massimo e il minimo con costo inferiore.  
L’heap viene rappresentato tramite un array.



I vari elementi hanno i seguenti indici:

BuildMaxHeap(A, n){

for i = [n/2] to 1{

MaxHeapify(A, i, n)

}

}

MaxHeapify(A, i, n){

largest = A[i]

t = i

if(2\*i <= n) && (A[i] < A[2\*i]){

t = 2\*i

largest = A[2\*i]

}

if(2\*i +1 <= n) && (A[2\*i +1] > largest){

t = 2\*i+1

# largest = A[2\*i+1]

}

if(i != t){

scambia(A[t], A[i])

MaxHeapify(A, t, n)

}

}

Ha costo

È utile per la ricerca del massimo o minimo

# HeapSort

**Complessità in tempo:**

È un algoritmo in-place ma non stable.

**Codice:**

HeapSort(A, n){

BuildMaxHeap(A,n)

for i = n to 2{

scambia(A[1], A[i])

MaxHeapify(A, 1, i-1)

}

}

Con o senza BuildMaxHeap() il costo della funzione rimane invariato

Operazione sugli Heap

## Insertion-key

Ha costo .

Aggiunge un nuovo nodo all’Heap e ne ricalcola il posto.

**Codice:**

Insertion-key(A, n, k){

n = n + 1

A[n] = k

i = n

while(A[i/2] < A[i] && i/2 >= 1){

scambia(A[i], A[i/2])

i = i/2

}

}

## Extract-max

Ha costo .

Elimina il massimo (la radice) e ricostruisce l’heap.

**Codice:**

Extract-max(A, n){

max = A[1]

A[1] = A[n]

n = n – 1

Maxheapify(A, 1, n)

return max

}

## Extract-key

Ha costo .

Elimina l’elemento alla posizione data e ricostruisce l’heap.

**Codice:**

Extract-key(A, n, index){

tmp = A[index]

A[index] = A[n]

tmp2 = A[n]

n = n – 1

MaxHeapify(A, index, n)

if(A[index] == tmp2){

i = index

while( A[i/2] < A[i] && i/2 >= 1 ){

scambia(A[i/2], A[i])

i = i/2

}

}

}

# Albero delle Decisioni

n = numeri da ordinare

h = log2(n!)

n\_foglie = n!

Un algoritmo è ottimo quando la sua complessità nel worst-case coincide con la complessità intrinseca del problema