

Sorting & Time complexity

Basato su [Competitive Programmer's Handbook](#), [Dispense del prof. Bugatti](#) e [Halim's Competitive Programming 3](#)

Index

- Teoria su complessità computazionale (20 min)
- Algoritmi di sorting (55 min)
 - Random & bucket sort (5 min)
 - Bubble sort (20 min)
 - Selection o insertion sort (30 min)
- Teoria recursion (15 min)
- Merge sort (30 min)

Complessità computazionale

Quando parliamo di efficienza come facciamo a capire quando un algoritmo è migliore rispetto a un altro?

Complessità computazionale

Quando parliamo di efficienza come facciamo a capire quando un algoritmo è migliore rispetto a un altro?

Si calcola una stima di costo dell'algoritmo.

“ costo = tempo di esecuzione ”

Esempio

“ Supponiamo che la dichiarazione, assegnazione e qualsiasi operazione su una variabile costa 1 ”

```
int a = 5;           //costo 1
for(int i=1; i<N; i++){ // costo ?
    a = a * a;       //costo 1
}
return a;
```

Complessità computazionale

Esempio

“ Supponiamo che

- inserire valore da tastiera
- istruzione di confronto
- operazioni matematiche
hanno tutte costo 1

”

Calcolate la complessità

```
int a, b = 0;           // dichiarazione
cin >> a;               // inserimento
for(int i=0; i<N; i++){ // ?
    if(a > 5) {          // confronto
        b = b + i;       // operazione
        a--;             // continue voi
    }else{
        for(int j=0; j<N; j++){
            b = b + j
        }
    }
}
return a + b;
```

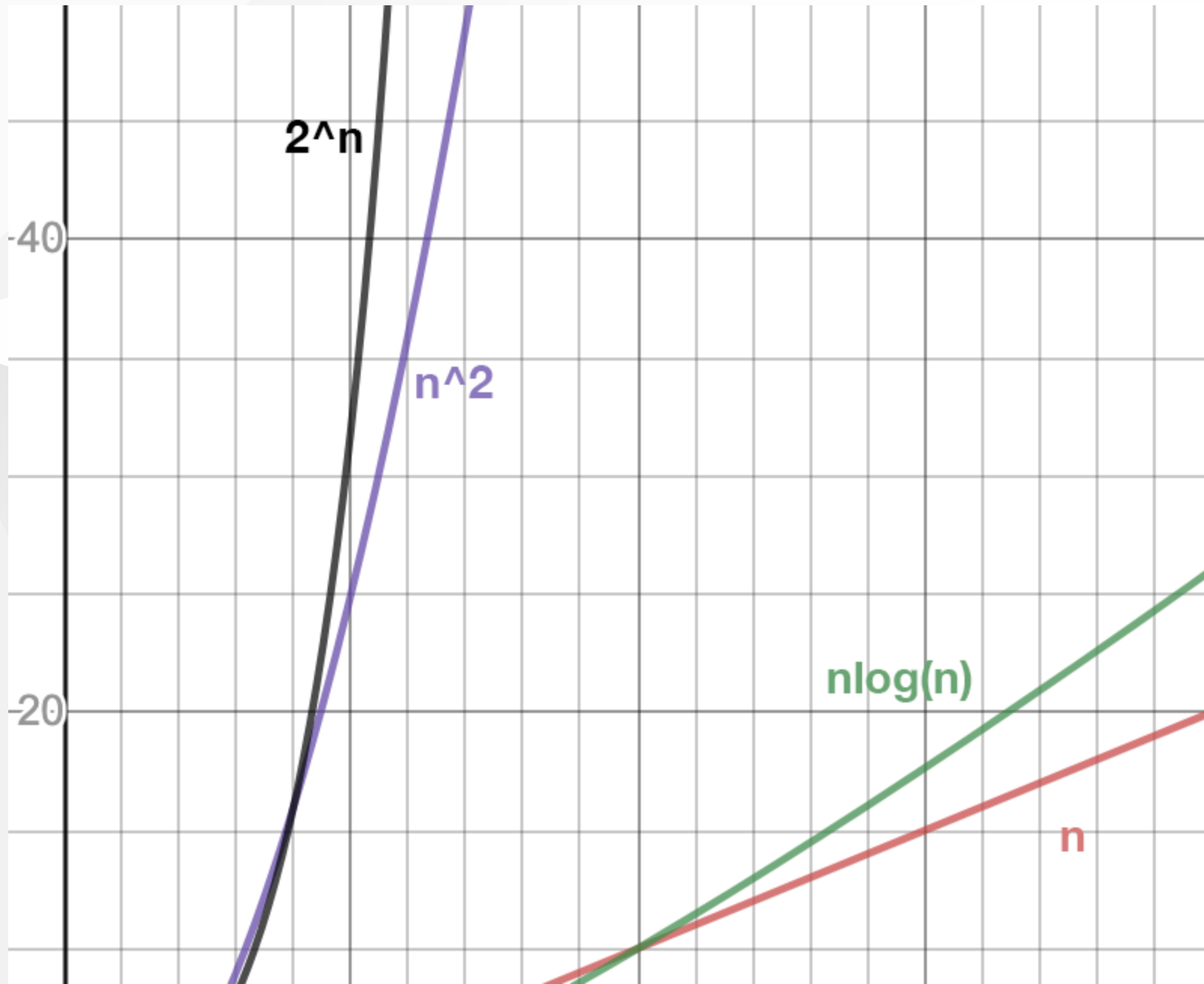
Notazione $O(n)$

Per calcolare la complessità dobbiamo:

- analizzare il caso peggiore
- valutare il costo per un N molto grande

Quando N è molto grande possiamo "dimenticarci" degli altri termini. A questo punto la complessità possiamo approssimarla a delle "famiglie" di funzioni. Indichiamo queste con la notazione di $O(_)$ ("o" grande):

$O(1)$ - $O(\log N)$ - $O(N)$ - $O(N \log N)$ - $O(N^2)$ - $O(2^N)$



Algoritmi di sorting

Affronteremo ora il problema di riordinamento di un array. Ci eserciteremo a scrivere questi algoritmi e a cercare l'algoritmo più efficiente

Random sort

Approccio "demente" (ergo MEGA non efficiente):

“ Genero tutte le possibili permutazioni fino a quando non ne trovo una già ordinata. ”

Un'analogia è quella di ordinare un mazzo di carte lanciandolo in aria, raccogliendo le carte a caso e ripetendo il processo fino a quando il mazzo non è ordinato.

Ergo Sembra quasi-impossibile di trovare l'ordine giusto e non prendere sempre la versione sbagliata.

Random sort && Psuedo-codice

Ma ... e se per sbaglio scegliessi sempre lo stesso ordine sbagliato e ripetessi sempre quello? Buona notte perché in tale caso sarebbe infinito tempo.

Pseudo-codice:

```
while not sorted(deck):  
    shuffle(deck)
```

Source: <https://en.wikipedia.org/wiki/Bogosort>

Random sort && Codice

```
bool is_in_order() {  
    for (int i = 0; i < input.size() - 1; ++i) {  
        if (input[i] > input[i + 1])  
            return false;  
    }  
    return true;  
}  
  
void bogo_sort() {  
    random_device rd;           // Generatore di numeri casuali basato su hardware  
    mt19937 g(rd());            // Generatore casuale a 32 bit  
    while (!is_in_order()) {    // Continua a mescolare finché non è ordinato  
        shuffle(input.begin(), input.end(), g);  
    }  
}
```

Random sort && Complessità?

Caso peggiore: Non c'è limite superiore pratico perché l'algoritmo potrebbe generare sempre le stesse permutazioni sbagliate.

Formalmente, nel caso peggiore, la **complessità** è $O(n \cdot n!)$.

Perché se mi va male devo **controllare tutte le possibilità** con *shuffle()* (che sono $n!$) e per ognuna di essere **vedere se è ordinata** con *is_in_order()* (con costo n)

Bucket Sort

Ipotesi sull'input

Valori reali **uniformemente distribuiti** nell'intervallo $[0,1)$.

Qualunque insieme di valori distribuiti uniformemente può essere normalizzato nell'intervallo $[0,1)$ in tempo lineare.

Idea

Dividere l'intervallo in n sottointervalli di dimensione $1/n$, detti bucket, e poi **distribuire gli n numeri nei bucket**.

Per l'ipotesi di uniformità, il numero atteso di valori nei bucket è 1. Possono essere ordinati con Insertion Sort o altri (stabili).

Bucket Sort & Idea Generale

Bucket Sort Example

- Imposta un array di "bucket" inizialmente vuoti.
- Esamina l'array originale, inserendo ogni oggetto nel suo bucket (tipo $array[i] = bucket[n * array[i]]$).
- Ordina ogni bucket non vuoto (con, ad esempio, Selection Sort).
- Visita i bucket in ordine e rimetti tutti gli elementi nell'array originale tramite concatenazione.

Bucket Sort & Idea Dettagliata

1. Crea un *array-bucket* con 10 celle (k), per avere la uniformità;
2. Inserisci ogni elemento del array iniziale nella cella più ***simile*** in *array-bucket* (es: $0.23 \cdot 10 \rightarrow 2$);
3. Applico un algoritmo efficiente di sorting in ogni *sub_bucket*;
4. Concateno ogni *sub_bucket* nell'array originale

Bucket Sort & Pseudo-Codice

```
function bucketSort(array, k) is
    buckets = new array of k empty lists
    M = 1 + the maximum key value in the array
    for i = 0 to length(array) do
        insert array[i] into buckets[floor(k × array[i] / M)]
    for i = 0 to k do
        sort(buckets[i])
    return the concatenation of buckets[0], ..., buckets[k]
```

Source: <https://www.geeksforgeeks.org/bucket-sort-2/>
https://en.wikipedia.org/wiki/Bucket_sort

Bucket Sort & Codice

```
void bucketSort() {  
    int num_in_bucket = 10; //il "k", per forza per garantire uniformità  
    vector<vector<float>> bucket(num_in_bucket); //matrice "lunga k", O(k)  
    float M = 1 + *max_element(input.begin(), input.end()); //O(n)  
  
    for (float elem : input) { //O(1 + 1)*n -> O(n)  
        int index = floor(num_in_bucket * elem / M); //hashing func, O(1)  
        bucket[index].push_back(elem); //putting elem in the similar bucket, O(1)  
    }  
    for (int i = 0; i < num_in_bucket; i++) //sorting each sub-vector, O(k)  
        selection_sort(bucket[i]); //meglio insertion_sort per pochi elementi, se no sicuramente O(W^2)  
        //con "W" = n/k grazie alla distribuzione uniforme  
        //ergo O(W^2) -> O((n/k)^2)  
        //quindi il tutto è O(k)*O((n/k)^2) -> O(n^2 / k), se k è const allora ho O(n^2)  
  
    input.clear(); //delete all element  
    for (int i = 0; i < num_in_bucket; ++i) { //concatenate, O(k)  
        for (float num : bucket[i]) //O(n/k)  
            input.push_back(num); //O(1)  
        }  
        //ergo il tutto è: O(k)*O(n/k) -> O(n)  
    } //in conclusione: O(n) + O(n^2 / k) + O(n) -> = O(n)  
}
```

Bucket Sort & Complessità

Vedendo (come sempre di deve fare) il **caso pessimo** $O(n^2)$.

Si verifica quando un bucket ottiene tutti gli elementi.

In questo caso, eseguiremo l'ordinamento per inserimento su tutti gli elementi, il che renderà la complessità temporale pari a **$O(n^2)$ SE usiamo *selection_sort()***.

Possiamo ridurre la complessità temporale del caso peggiore a $O(n \log n)$ utilizzando un algoritmo $O(n \log n)$ come Merge Sort

DI SOLITO il caso medio è $O(n + k)$ se uso *insertion_sort()* e ogni bucket ha lo stesso numero di elementi.

Bubble Sort & Idea

f



Bubble Sort Example

Source: <https://www.geeksforgeeks.org/bubble-sort-algorithm/>
[https://en.wikipedia.org/wiki/Bubble sort](https://en.wikipedia.org/wiki/Bubble_sort)

Bubble Sort & Psuedo-Codice

Idea:

“ Se elemento[x] è MAGGIORE di elemento[x+1], allora "swappa". ”

```
bubbleSort(ITEM[] A)
  n = length(A)
  repeat until not swapped
    swapped = false
    for i = 1 to n-1 do
      if A[i-1] > A[i] then //to be swapped
        //swap them and remember something changed
        swap(A[i-1], A[i])
        swapped = true
```

Bubble Sort && Codice && Complessità

```
void bubbleSort() { //  $O(n) * O(n-1) \rightarrow O(n^2)$   
    // NON OTTIMIZZATO !!! in caso usare un booleano di controllo  
    for (int i = 0; i < input.size(); i++) { //  $O(n)$   
        for (int j = i+1; j < input.size(); j++) { //  $O(n-1)$   
            if (input[i] > input[j])  
                iter_swap(input.begin()+i, input.begin()+j); //  $O(1)$   
        }  
    }  
}
```

Codesto ci costa **SEMPRE** $O(n^2)$.

Bubble Sort && Codice Ottimizzato && Complessità

```
void bubbleSort() { //  $O(n) * O(n-1) \rightarrow O(n^2)$ 
    for (int i = 0; i < input.size(); i++) { //  $O(n)$ 
        bool swapped=false;
        for (int j = 1; j < input.size()-i-1; j++) { //  $O(n-1)$ 
            if(input[j-1]>input[j])
                iter_swap(input.begin()+j, input.begin()+(j-1)); //  $O(1)$ 
            swapped=true;
        }
        if (!swapped)
            break;
    }
}
```

Codesto ci costa **LO STESSO $O(n^2)$** in teoria, **ma nella pratica è più "veloce".**

Selection Sort & Idea

Idea:

“ Trovo il minimo e lo metto "apposto" dove "dovrebbe" essere.
Ovvero cerco il minimo e lo metto in posizione corretta,
riducendo il problema agli $n-1$ restanti valori. ”



Selection Sort Example

Selection Sort & Pseudo-Codice

```
void selectionSort(ITEM[] A, int n) //  $O(n) * O(n) \rightarrow O(n^2)$   
    for i=1 to n-1 do //  $O(n-1) \rightarrow O(n)$   
        int min = min(A, i, n) // finding minimum between [i, n],  $O(n)$   
        A[i] <-> A[min] // swap them
```

```
int min(ITEM[] A, int i, int n) { //  $O(n)$   
    // posizione del minimo parziale  
    int min = i  
    for j=i+1 to n do //  $O(n)$   
        if A[j] < A[min] then  
            min = j // nuovo minimo parziale  
    return min  
}
```

Selection Sort & Codice

```
void selection_sort(vector<float> array) {//O(n)*O(n) -> O(n^2)  
    for (int i = 0; i < array.size(); i++)//O(n)  
        //find Min between [i, end], O(n)  
        auto min_iter = min_element(input.begin() + i, input.end());  
  
        //swap between (i <-> Min), O(1)  
        iter_swap(input.begin()+i, min_iter);  
}
```

Selection Sort & Complessità

Spiace dirlo ma **SEMPRE $O(n^2)$** , ovvero in caso ottimo, pessimo e medio $O(n^2)$.

Perché, in codesto algoritmo, **faremo sempre il *for loop* e sempre chiameremo *min_element()***, ergo entrambe costano sempre $O(n)$ ciascuna.

Mentre in *insertion_sort()* un *for loop* lo facciamo per forza, ma un *while loop* potremmo anche non farlo se già ordinato.

Accenni Insertion Sort

- Algoritmo efficiente per ordinare "piccoli" insiemi di elementi
- Si basa sul principio di ordinamento di una "mano" di carte da gioco (e.g. scala quaranta)

```
void insertionSort(ITEM[] A, int n){//pessimo:  $O(n^2)$ , se già ordinato  $O(n)$   
    for i = 2 to n do // $O(n)$   
        ITEM tmp = A[i]  
        int j=i  
        while j>1 and A[j-1]>tmp do //AL PIU'  $O(n)$   
            A[j] = A[j-1]  
            --j  
        A[j]=tmp  
}
```

Insertion Sort && suo esempio

 Insertion Sort example

Ricorsione

Per migliorare l'efficienza dei nostri algoritmi ci viene in aiuto un'importante strumento: la ricorsione!

Ma attenzione a usarla bene

Pattern della ricorsione:

```
function(a){  
  if caso base  
    ...  
  else  
    ...  
    function(b)  
}
```

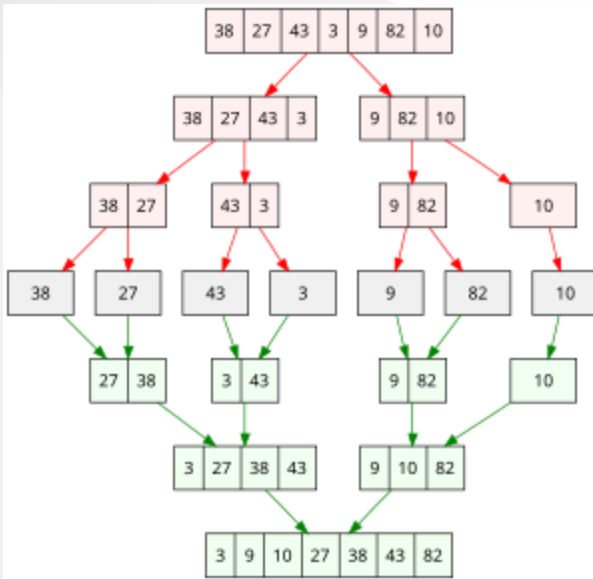
Esempio di ricorsione: fattoriale

```
fattoriale(int n){  
    if (n == 0) return 1  
    else return n*fattoriale(n-1)  
}
```


Merge sort & Idea

Idea: Divido il mio array in due sottoarray e li riordino, poi alla fine li rimetto insieme in modo ordinato

Esempio:



Merge sort & Pseudo Codice

```
merge(ITEM[] A, ITEM[] B, ITEM[] C, int fine){
    int i, j, z = 0
    while(z <= fine){
        if B[i] < C[j] {
            A[z] = B[i]           //scelgo il più piccolo
            i++, z++
        }else{
            A[z] = C[j]
            j++, z++
        }
    }
}

mergeSort(ITEM[] A, int inizio, int fine){
    if inizio >= fine
        return
    else{
        //divido A in due array B e C
        mergeSort(B, inizio, metà)           //chiamo mergeSort su B e C
        mergeSort(C, metà+1, fine)
        merge(A, B, C, fine)                 //unisco in modo ordinato B e C
        return
    }
}
```

Merge Sort & Complessità

Qual è il costo della funzione merge? $O(N)$

Come facciamo a calcolare la complessità di mergeSort?

“ In mergeSort facciamo due chiamate ricorsive con dimensione pari a $n/2$ ”

Si definisce una formula per calcolare la complessità:

$T(1)$ -> se abbiamo un solo elemento nell'array

$2T(n/2) + O(N)$ -> altrimenti

Con un po' di calcoli (Master Theorem [che hai detto?]) otteniamo che la complessità è quindi $O(N \log N)$.

Merge Sort

Versione iterativa (versione bottom-up):

 merge

Merge Sort iterativo

Ora scriviamo noi il codice!

Idea:

Parto dal fondo di un array con dimensione unitaria (ergo per forza è già ordinata) e poi uso gli indici per dividere l'array e specificare quale parte dell'array riordinare