Sorting && Time complexity

Basato su Competitive Programmer's Handbook, Dispense del prof. Bugatti e Halim's Competitive Programming 3

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 costi 1

Complessità computazionale

Esempio: supponiamo che

- inserire valore da input
- istruzione di confronto
- operazioni matematiche

abbiano tutte costo 1

```python
N = 100                     # costo 1
a = 5                       # costo 1
for i in range(1, N):       # costo ?
    a = a + a               # costo 1
return a
```

Soluzione:

- Assegnazioni: 2 operazioni = 2
- Ciclo for eseguito (N-1) volte, ogni iterazione costa 1
- Totale: 2 + (N-1) = N + 1
- Per N grande: **O(N)**

Esempio con cicli annidati

```python
a = int(input())            # costo 1
b = 0                       # costo 1
for i in range(N):          # eseguito N volte
    if a > 5:               # costo 1
        b = b + i           # costo 1
        a -= 1              # costo 1
    else:
        for j in range(N):   # eseguito N volte
            b = b + j        # costo 1
```

Soluzione (caso peggiore):

- Se a ≤ 5 inizialmente, entriamo sempre nell'else
- Ciclo esterno: N iterazioni

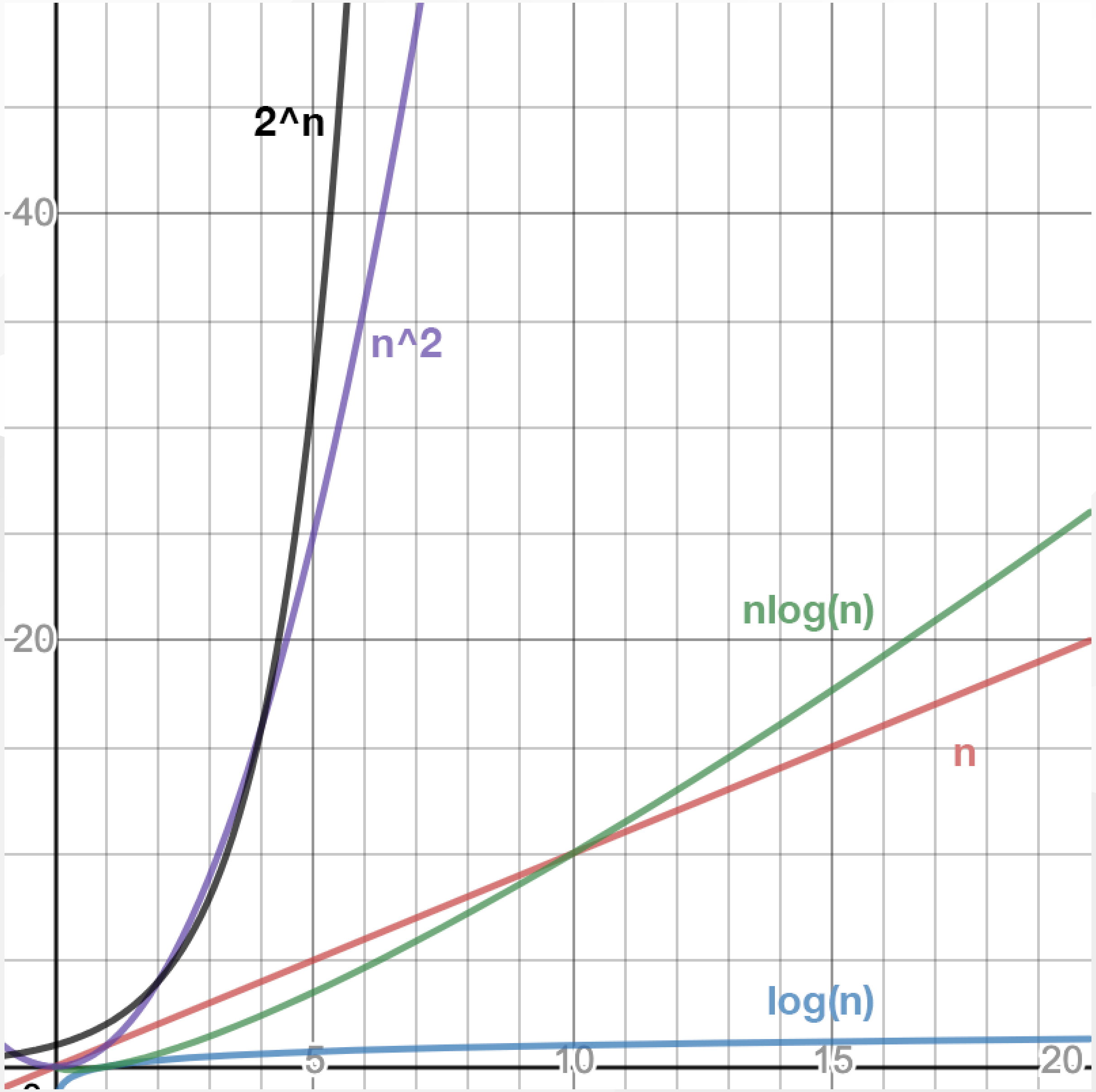
Notazione O(n)

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 può essere approssimata a delle "famiglie" di funzioni. Indichiamo queste con la notazione di "O grande":

O(1) - O(logN) - O(N) - O(NlogN) - O(N^2) - O(2^N)


2^n
n^2
nlog(n)
n
log(n)
40
20
5
10
15
20

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 "particolare"

- **Generiamo tutte le possibili permutazioni fino a quando non ne troviamo 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.
Appare quasi impossibile trovare l'ordine giusto senza generare ripetutamente permutazioni sbagliate.

Random sort && Pseudo-codice

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

Random sort && Codice completo

```
import random

def is_sorted(arr):
    """Verifica se l'array è ordinato"""
    for i in range(len(arr) - 1):
        if arr[i] > arr[i + 1]:
            return False
    return True

def random_sort(arr):
    """
    Bogo sort: mescola l'array finché non è ordinato
    ATTENZIONE: estremamente inefficiente!
    """
    while not is_sorted(arr):
        random.shuffle(arr)
```

Random sort && Complessità?

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

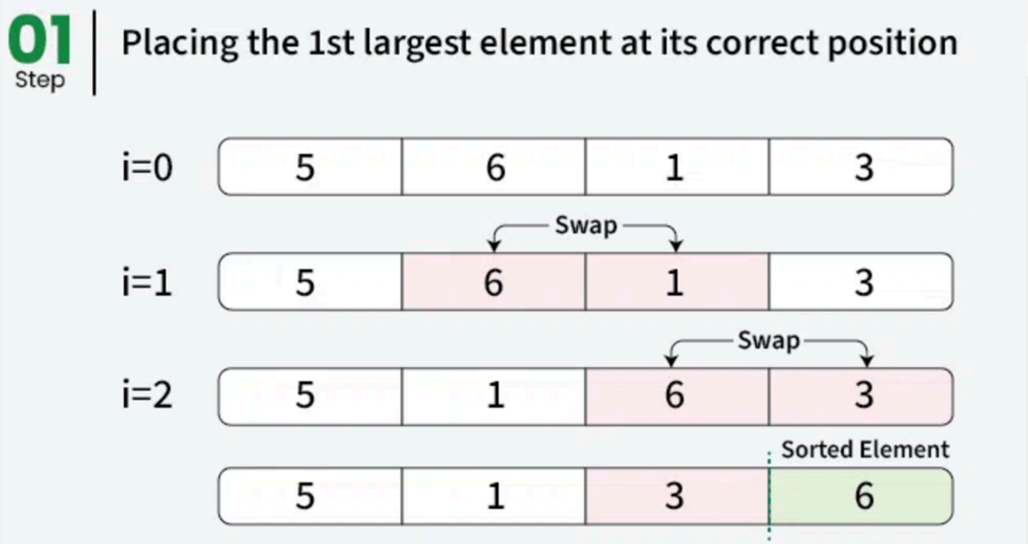
Formalmente, nel caso peggiore, la **complessità è $O(n \cdot n!)$**.

Questo perché, nel caso peggiore, dobbiamo **controllare tutte le possibilità** con *shuffle()* (che sono $n!$) e per ognuna **verificare se è ordinata** con *is_sorted()* (con costo n).

Bubble Sort && Idea

Strategia: "Fai emergere il più grande come una bolla"

- Confronta ogni coppia di elementi adiacenti
- Se sono nell'ordine sbagliato, scambiali
- Ripeti finché non ci sono più scambi da fare



Bubble Sort && Pseudo-Codice

Regola base: Se elemento[x] > elemento[x+1], scambiali ("swap")

```
def bubble_sort(arr):  # O(n)*O(n) -> O(n^2)
    n = len(arr)
    for i in range(n):  # O(n)
        for j in range(n-1):  # O(n-1)
            if arr[j] > arr[j+1]:  # confronta elementi adiacenti
                arr[j], arr[j+1] = arr[j+1], arr[j]  # O(1)
```

Questo algoritmo ha complessità ***O(n^2)***.

Bubble Sort && Codice Ottimizzato

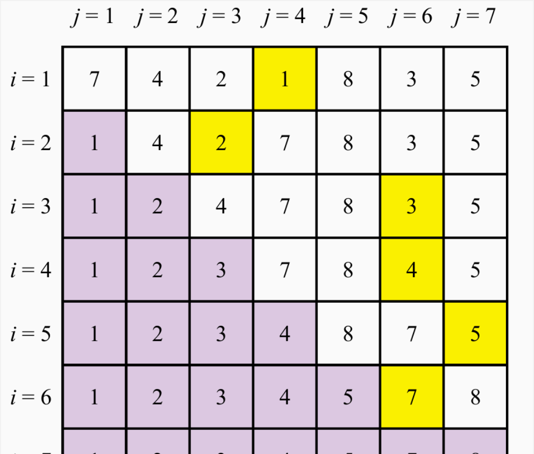
```
def bubble_sort_ottimizzato(arr):  # O(n^2) caso peggiore, O(n) caso migliore
    n = len(arr)
    for i in range(n):  # O(n)
        swapped = False
        for j in range(n-i-1):  # O(n-i)
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:  # se non ci sono stati scambi, l'array è ordinato
            break
```

Questo algoritmo ha **la stessa complessità *O(n^2)*** nel caso peggiore, **ma O(n) nel caso migliore** (array già ordinato).

Selection Sort && Idea

Strategia: "Trova il minimo e mettilo al suo posto"

- Per ogni posizione `i`, cerca il minimo tra gli elementi rimanenti `[i, fine]`
- Scambia l'elemento in posizione `i` con il minimo trovato
- Ripeti per tutte le posizioni



	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
$i=1$	7	4	2	1	8	3	5
$i=2$	1	4	2	7	8	3	5
$i=3$	1	2	4	7	8	3	5
$i=4$	1	2	3	7	8	4	5
$i=5$	1	2	3	4	8	7	5
$i=6$	1	2	3	4	5	7	8

Selection Sort && Codice

```
def selection_sort(array):  # O(n)*O(n) -> O(n^2)
    for i in range(len(array)):  # O(n)
        # trova minimo tra [i, fine], O(n)
        min_idx = i
        for j in range(i+1, len(array)):
            if array[j] < array[min_idx]:
                min_idx = j

        # scambia (i <-> min), O(1)
        array[i], array[min_idx] = array[min_idx], array[i]
```

Selection Sort && Complessità

La complessità è **sempre *O(n^2)*** in ogni caso (migliore, medio, peggiore).

Perché?

- Eseguiamo **sempre** i cicli for completi
- Cerchiamo **sempre** il minimo tra tutti gli elementi rimanenti
- **Non ci sono ottimizzazioni possibili**, anche se l'array è già ordinato

Numero di confronti: $n(n-1)/2 \approx n^2/2 \rightarrow O(n^2)$

Insertion Sort && Idea

Strategia: "Inserisci ogni elemento nella posizione corretta"

Come ordinare una mano di carte (scala quaranta):

- Manteniamo una parte "ordinata" dell'array (inizialmente solo il primo elemento)
- Per ogni nuovo elemento, lo inseriamo nella posizione corretta nella parte ordinata
- Spostiamo gli elementi maggiori verso destra per fare spazio

Insertion Sort && Codice

```python
def insertion_sort(A):  # pessimo: O(n^2), se già ordinato O(n)
    n = len(A)
    for i in range(1, n):  # O(n)
        tmp = A[i]
        j = i
        while j > 0 and A[j-1] > tmp:  # AL PIU' O(n), PUO' non eseguirsi
            A[j] = A[j-1]
            j -= 1
        A[j] = tmp
```

Insertion Sort && suo esempio

	1	2	3	4	5	6	7
	7	4	2	1	8	3	5
$i = 2, j = 2$	7	7	2	1	8	3	5
$i = 2, j = 1$	4	7	2	1	8	3	5
$i = 3, j = 3$	4	7	7	1	8	3	5
$i = 3, j = 2$	4	4	7	1	8	3	5
$i = 3, j = 1$	2	4	7	1	8	3	5

Insertion Sort && Complessità

Caso peggiore: $O(n^2)$ - array ordinato al contrario
Caso migliore: $O(n)$ - array già ordinato!

Perché è più efficiente di Selection Sort?

- Il ciclo `while` interno **può non eseguirsi** se l'elemento è già nella posizione corretta
- Se l'array è già ordinato, il while non viene mai eseguito → $O(n)$
- Molto efficiente per array quasi ordinati

Ricorsione

Per migliorare l'efficienza dei nostri algoritmi ci viene in aiuto un importante strumento: la ricorsione!
Ma attenzione a usarla correttamente: bisogna stare attenti a non creare un loop infinito e verificare sempre il caso base.

```
def function(a):
    if caso_base:
        ...
    else:
        ...
        function(b)
```

Esempio di ricorsione: fattoriale

```python
def fattoriale(n):
    if n == 0:
        return 1
    else:
        return n * fattoriale(n-1)
```

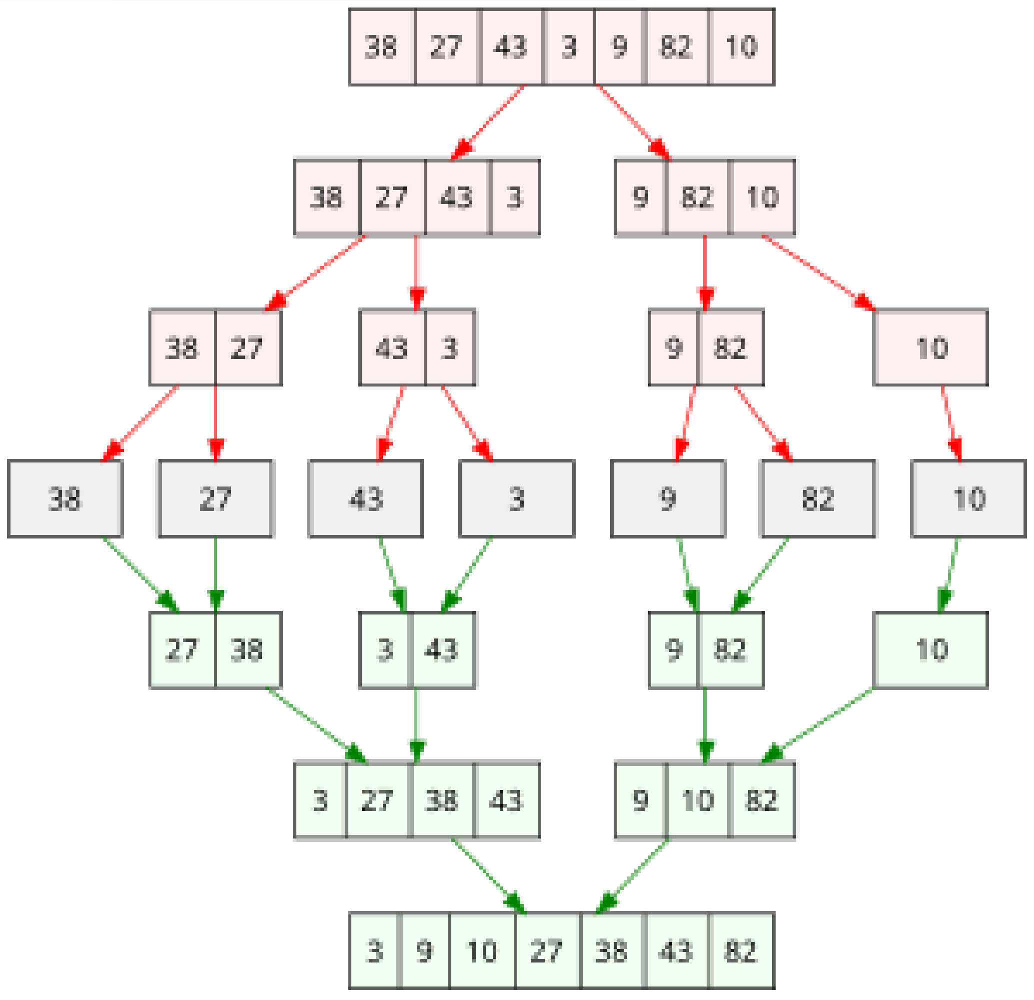
Merge Sort && Idea

Strategia: "Dividi e Conquista"

1. **Dividi:** Spezza l'array in due metà
2. **Conquista:** Ordina ricorsivamente ciascuna metà
3. **Combina:** Unisci le due metà ordinate in un unico array ordinato

Caso base: Un array di 1 elemento è già ordinato!

Merge Sort && Visualizzazione



Merge sort && Codice

```python
def merge(arr, left, mid, right):
    # Crea due array temporanei
    left_array = arr[left:mid+1]
    right_array = arr[mid+1:right+1]

    i = j = 0  # indici per left_array e right_array
    k = left   # indice per array originale

    # Confronta e unisci
    while i < len(left_array) and j < len(right_array):
        if left_array[i] <= right_array[j]:
            arr[k] = left_array[i]
            i += 1
        else:
            arr[k] = right_array[j]
            j += 1
        k += 1

    # Copia gli elementi rimanenti di left_array
    while i < len(left_array):
        arr[k] = left_array[i]
        i += 1
        k += 1

    # Copia gli elementi rimanenti di right_array
    while j < len(right_array):
        arr[k] = right_array[j]
        j += 1
        k += 1

def merge_sort(arr, left, right):
    if left < right:
        mid = (left + right) // 2

        # Ordina ricorsivamente le due metà
        merge_sort(arr, left, mid)
        merge_sort(arr, mid + 1, right)

        # Unisci le due metà ordinate
        merge(arr, left, mid, right)

# Utilizzo: merge_sort(array, 0, len(array)-1)
```

Merge Sort && Complessità

Analisi:

1. **Costo di merge():** O(n) - dobbiamo scorrere tutti gli elementi
2. **Numero di livelli:** log n - ogni volta dividiamo per 2
3. **Costo per livello:** O(n) - merge di tutti i sotto-array

Formula ricorsiva:

- T(1) = O(1) → array di 1 elemento
- T(n) = 2T(n/2) + O(n) → dividi in 2 + merge

Risultato (Master Theorem): O(n log n) ✅

Merge Sort

Versione iterativa (versione bottom-up):

Merge Sort iterativo

Idea:

Partiamo dal fondo di un array con dimensione unitaria (quindi per definizione è già ordinato) e poi usiamo gli indici per dividere l'array e specificare quale parte dell'array riordinare.

```
def merge_sort_iterativo(arr):
    n = len(arr)
    # Inizia con blocchi di dimensione 1, poi 2, 4, 8, ...
    current_size = 1

    while current_size < n:
        # Seleziona il punto di inizio del sottovettore sinistro da unire
        left_start = 0

        while left_start < n:
            # Trova il punto finale del sottovettore sinistro
            left_end = min(left_start + current_size - 1, n - 1)

            # Trova il punto finale del sottovettore destro
            right_end = min(left_start + current_size * 2 - 1, n - 1)

            # Unisci i sottovettori arr[left_start...left_end] e
```