Algoritmi di Ordinamento o Sorting

Il problema dell'ordinamento è il seguente:

Dato un insieme S di n oggetti presi da un dominio totalmente ordinato, ordinare S.

Quindi avremo in:

- **input**: Una sequenza di n numeri: $< a_1, a_2, \ldots . a_n >$.
- **output**: Un riarrangiamento della sequenza ordinata in modo crescente/decrescente.

Ordinare in tempo quadratico

Sono algoritmi molto intuitivi, ma inefficienti:

- **Selection Sort:** Leggiamo il primo numero che abbiamo nella sequenza, poi controlliamo tutti gli altri k numeri in modo tale da trovare il più piccolo da inserire nella posizione attuale.

```
#Selection Sort: Complessità O(n^2)

def SelectionSort(A):
    for i in range(len(A)): #i è l'indice dell'elemento da sostituire
        min = i
        for j in range(i+1,len(A)): #j è l'indice del minimo elemento
            if A[j] < A[min]:
                min = j
        A[i],A[min] = A[min],A[i] #Scambio
    return A

A = [5,4,3,2,1]
print(SelectionSort(A))

## Come funziona:
# 1. Si cerca il minimo elemento nella lista A
# 2. Si scambia il minimo con l'elemento in posizione i
# 3. Si incrementa i e si ripete il processo fino a quando i è minore della
lunghezza della lista
```

Note

- Al generico passo k, $A[1], \ldots\ldots A[k]$ sono già ordinati.
- Le linee 2-6 servono per cercare l'elemento minimo.
- min è l'indice dell'elemento più piccolo.

Analisi della Correttezza:

E' facile capire che l'algoritmo è corretti per mantiene le seguenti invarianti al generico passo k, cioè:

- I primi $k+1$ elementi sono ordinati;
- Sono i $k+1$ elementi più piccoli dell'array;

Invarianti:

Proprietà immutate dell'algoritmo.

Analisi della Complessità:

$T(n)$ = Numero di operazioni elementari sul modello RAM a costi uniformi dell'algoritmo nel caso peggiore su istanza di dimensione n.

Diamo come prima cosa un upper bound all'algoritmo:

- Il ciclo più esterno verrà eseguito al più n volte;
- Il ciclo interno verrà eseguito al più n volte per ogni ciclo esterno;

Sapendo che ogni linea di codice costa $O(1)$ possiamo dire che:

$$T(n) \leq cn^2 * O(1) = \Theta(n^2) \rightarrow T(n) = O(n^2)$$

L'analisi è stretta? Cioè, $T(n) = \Theta(n)$, per capirlo cerchiamo un lower bound e per trovarlo conto solo i confronti elementari: $n-k-1$ confronti:

$$T(n) \geq \sum_{k=0}^{n-2} n-k-1 = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2)$$

Il che significa che: $T(n) = \Omega(n^2) \rightarrow T(n) = \Theta(n^2)$

Altri algoritmi di sorting quadratici sono:

- **Insertion Sort**: Andiamo a posizionare l'elemento in $(k+1) - esima$ posizione nella posizione corretta rispetto agli altri k elementi.

- **Bubble Sort**: Esegue $n-1$ scansioni. Ad ogni scansione controlla elementi di coppie adiacenti e li scambia se non sono corretti.

Ordinare in tempo meno che quadratico

Sono algoritmi meno intuitivi rispetto a quelli precedenti, ma più veloci ed utilizzano principalmente la tecnica del **divide et impera:**

Merge Sort

Usa la tecnica del divide et impera, seguendo questi passi:

1. Divide l'array a metà (**DIVIDE**);
2. Risolve i due sottoproblemi ricorsivamente;
3. Fondi le sottosequenze ordinate (**IMPERA**);

Avremo bisogno di due funzioni, la prima per divide la lista a metà, la seconda invece per unire le sottosequenze ordinate:

La prima funzione è abbastanza semplice:

```
def mergesort(arr): #funzione che divide la lista in due sottosequenze
ordinate

    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2

    left = mergesort(arr[:mid])
    right = mergesort(arr[mid:])

    return merge(left, right)
```

Per la procedura di unione dobbiamo semplicemente continuare ad estrarre il minimo dell'array A e B e copiarlo nell'array di output, finché uno dei due non diventa vuoto, a quel punto copiato il rimanente alla fine:

```
def merge(left, right):#funzione che unisce le sottoliste
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i]) ## Se l'elemento nella lista di sinistra è
```

```
minore di quello nella lista di destra, lo aggiungo alla lista risultato

            i += 1
        else
            result.append(right[j]) ## Altrimenti aggiungo l'elemento della
lista di destra

            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
```

Lemma:

La procedura merge fonde due sequenza ordinate di lunghezza n_1 e n_2 in tempo $\Theta(n_1 + n_2)$

Dimostrazione:

Ogni confronto "consuma" un elemento di una delle due sequenze. Ogni posizione di X è riempita in tempo costante. Il numero totale di elementi è $n_1 + n_2$. Anche la copia nel vettore ausiliario.

Analisi:
E' senza dubbio corretto, la complessità si può rappresentare tramite un'equazione di ricorrenza:

$$T(n) = 2T(n/2) + O(n)$$

Ed usando il teorema Master otteniamo:

$$T(n) = O(nlog(n))$$

Uno dei problemi del Merge Sort è che non ordina in loco, la procedura Merge usa memoria ausiliaria pari alla dimensione di porzione da fondere e non sono mai attive due procedure di Merge contemporaneamente, infatti la complessità spaziale è di: $\Theta(n)$.

Quick Sort

Un altro algoritmo che usa la tecnica del divide et impera:

1. Scegli un elemento x della sequenza (**perno**) e partiziona la sequenza in elementi minori di x e maggiori x. (DIVIDE);

2. Risolvi i sottoproblemi ricorsivamente;
3. Restituisci la concatenazione delle sottosequenze ordinate (IMPERA);

Il primo passaggio è quello della partizione, scegliamo un perno della lista e cominciamo a scorrere la lista da entrambi i lati:

- Da sinistra verso destra ci fermiamo su elementi maggiori del perno;
- Da destra verso sinistra ci fermiamo su elementi minori del perno;

Quando gli indici sono entrambi fermi allora scambiamo gli elementi e ci fermiamo solo quando gli indici si incontrano:

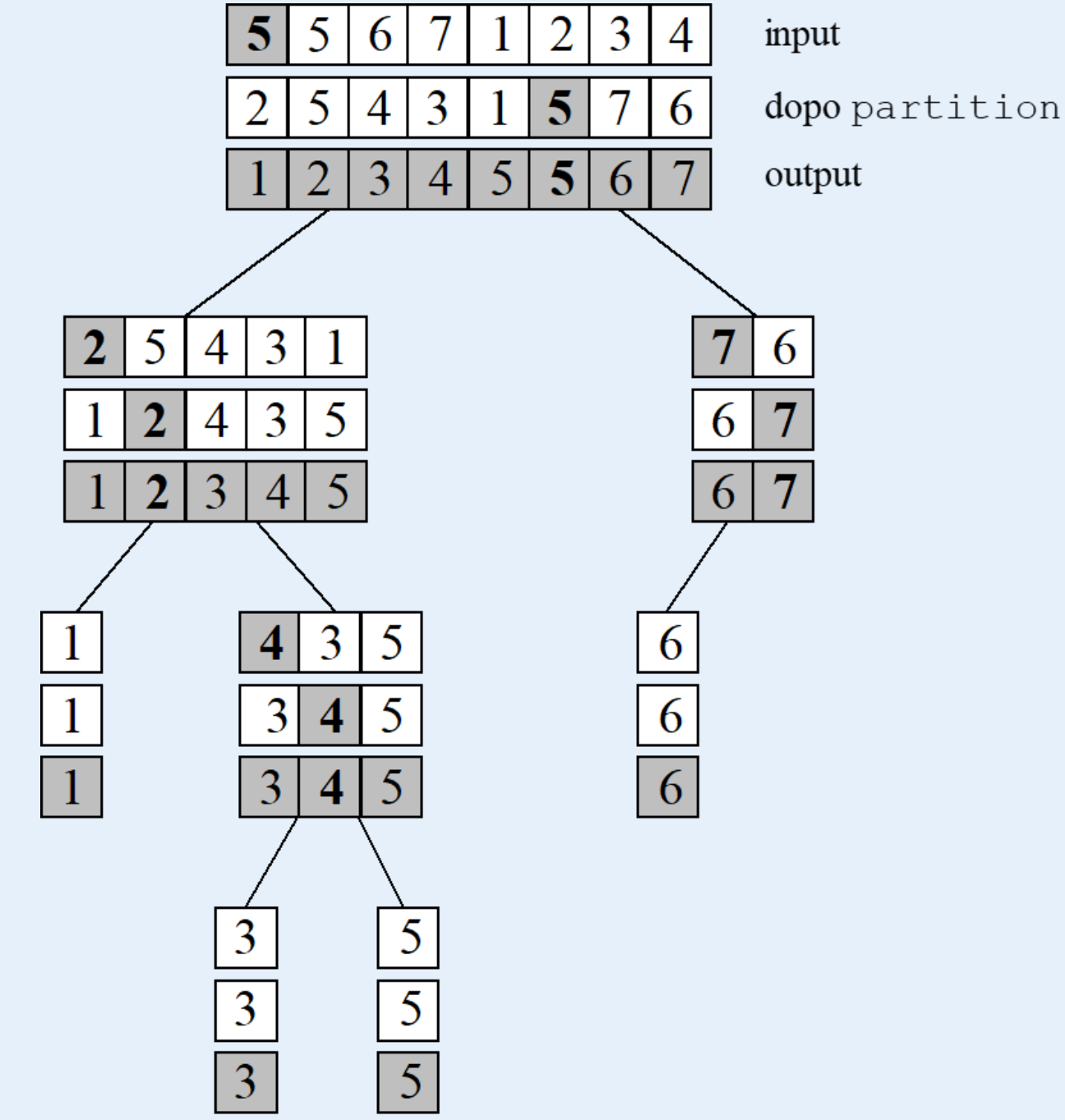
```
def partition(A,i,f): #O(n)

    perno = A[i]#perno è il primo elemento della lista
    inf = i+1
    sup = f

    while True:
        while inf <= sup and A[inf] <= perno: #Scorre la lista finchè non
trova un elemento maggiore del perno

            inf += 1

        while inf <= sup and A[sup] >= perno: #Scorre la lista finchè non
trova un elemento minore del perno

            sup -= 1
        if inf <= sup:
            A[inf],A[sup] = A[sup],A[inf] #Scambio
        else:
            break
    A[i],A[sup] = A[sup],A[i] #Scambio del perno con l'elemento in posizione
sup

    return sup
```

L'altra funzione è abbastanza semplice:

```
def quicksort(A,i,f): #O(nlogn)
    if i < f:
        p = partition(A,i,f)
        quicksort(A,i,p-1)
```

```
        quicksort(A,p+1,f)
    return A
```

Esempio di esecuzione:



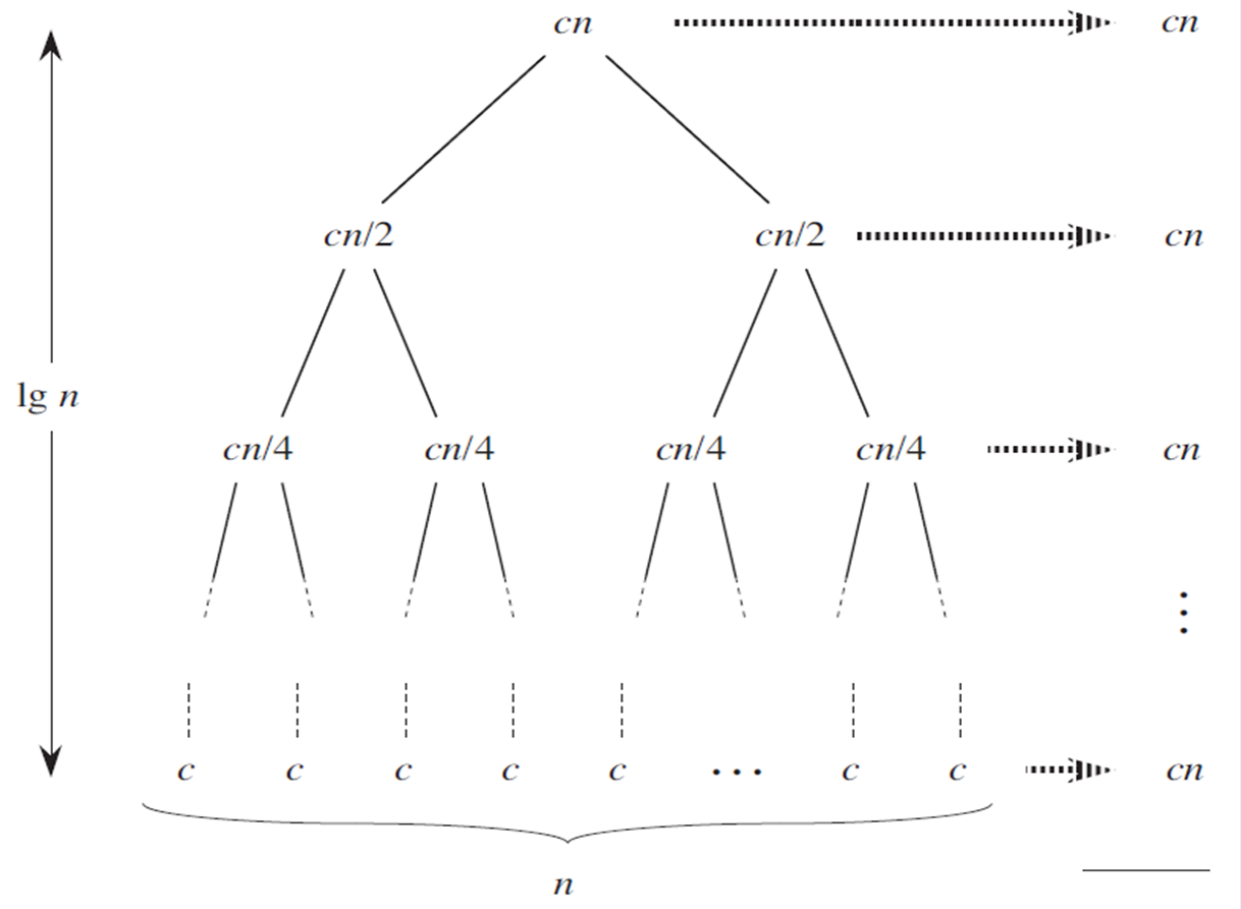
Analisi:
L'algoritmo è corretto perché:

- Sappiamo che i gli elementi che vanno da $[i : m-1]$ sono tutti minori del perno, mentre gli elementi $[m+1 : f]$ sono tutti maggiori.
- Sappiamo che le chiamate ricorsive ordinano queste sequenze.

Adesso andando ad analizzare il caso peggiore otteniamo che dopo n invocazioni di partition, ognuna di costo $O(n)$ ho il vettore ordinato. Il costo è quindi $O(n^2)$.
Questa situazione si verifica quando scelgo come perno il minimo o il massimo dell'array:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + O(n) \\ &= T(n-1) + O(1) + O(n) \\ &= T(n-1) + O(n) = O(n^2) \end{aligned}$$

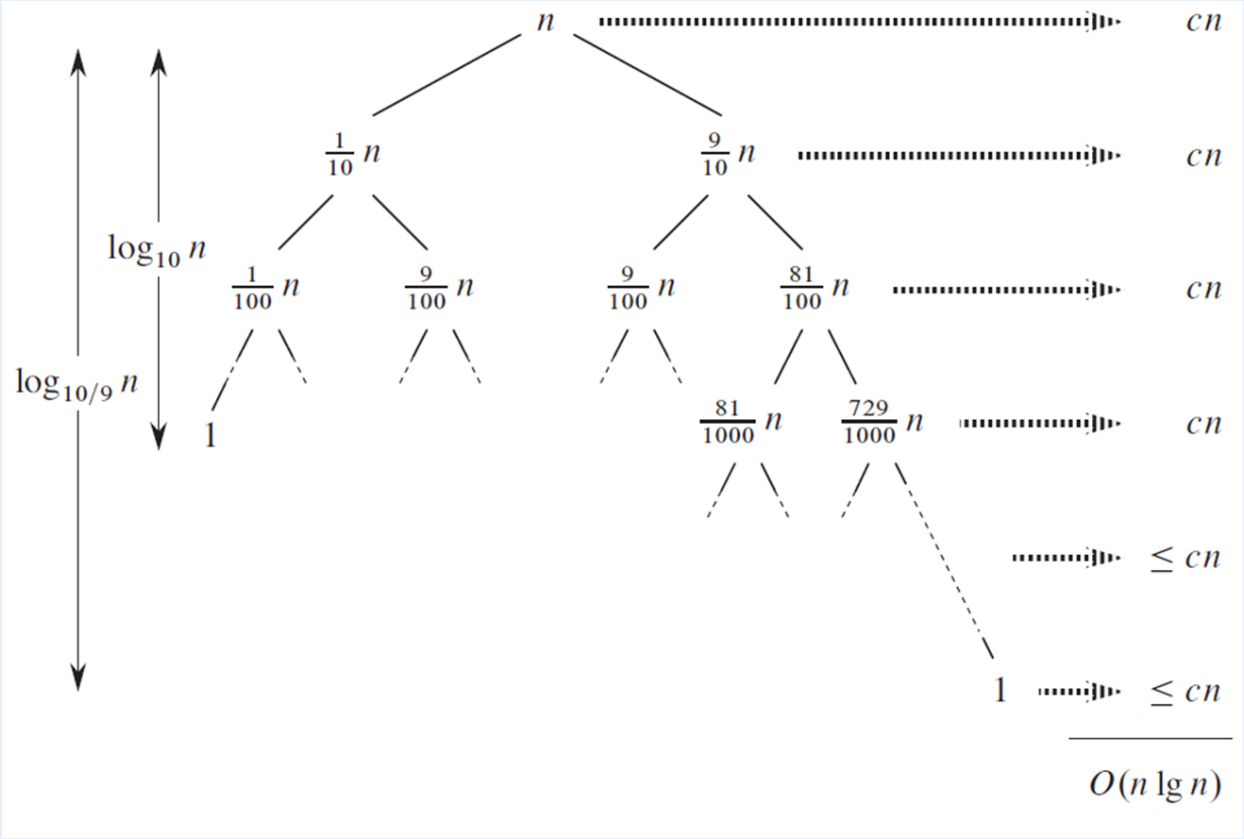
Nel caso migliore abbiamo $O(nlog(n))$, cioè quando abbiamo un partizionamento bilanciato :



Siccome il caso peggiore e il caso minore sono così discordanti mi conviene pensare all'algoritmi in termini di caso medio, ipotizzando al caso di istanze equiprobabili ottengo:

- Il problema principale si verifica quando le partizioni sono sbilanciate.
- Ma la probabilità che si verifichi la partizione peggiore è bassa.
- Infatti per partizioni non troppo sbilanciate l'algoritmo è veloce.

Partizione molto sbilanciata 9-a-1:



la complessità è di $O(nlog(n))$, quindi basta che la partizione non sia completamente sbilanciata, cioè basta non prendere il massimo o il minimo.

Versione randomizzata

Fino ad adesso abbiamo assunto che le istanze fossero equiprobabili, ma se non lo fossero?

Ecco che utilizziamo la versione **randomizzata del quicksort** nella quale il perno x è scelto a caso fra gli elementi da ordinare, fino ad ora abbiamo sempre scelto il primo elemento.

Teorema:

L'algoritmo quicksort randomizzato ordina in loco un array di lunghezza n in tempo $O(n^2)$ nel caso peggiore e $O(nlog(n))$ tempo atteso, cioè con alta probabilità, ovvero con probabilità almeno $1 - \frac{1}{n}$.

Randomizzazione $\neq$ Caso Medio:

Nella randomizzazione non facciamo nessuna assunzione sulla distribuzione di probabilità e non ci sta nessun input specifico per il quale si verifica il caso peggiore che è

determinato solo dal generatore casuale di numeri.