Corso di laurea in Informatica
Progettazione d'algoritmi
Didattica blended

Tecnica Divide et Impera 3

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Il problema della selezione

Data una lista A, il rango di un elemento x in A è il numero di elementi che sono minori o uguali ad x in A.

Esempi:

- il minimo in A è l'elemento di rango 1.
- il mediano in A è l'elemento di rango $\lceil \frac{n}{2} \rceil$
- il massimo in A è l'elemento di rango n.

Problema: Abbiamo una lista A di n numeri distinti ed un intero k, $1 \leq k \leq n$. Vogliamo l'elemento di rango k in A.

Un semplice algoritmo di complessità $O(n \log n)$ è il seguente:

```
def selezione1(lista, k):
    A=sorted(lista )
    return A[k-1]

>>> lista[3,2,5,6,1,4,9,8,7]
>>> selezione1(lista,4)
4
```

- Se $k = 1$ o $k = n$ il problema si riduce alla ricerca del *minimo* o del *massimo* e questi casi sono risolvibili in tempo $O(n)$.
- Utilizzando la tecnica del *divide_et_impera*, vedremo che, anche nel caso generale, il problema può risolversi in $O(n)$.
- In altre parole dimostreremo che il problema della selezione è computazionalmente più semplice di quello dell'ordinamento (che richiede tempo $\Omega(n \log n)$).

Approccio basato sul Divide et impera:

- scegli nella lista A un elemento x (che chiameremo perno).
- a partire da A costruisci due liste A_1 ed A_2, la prima contenente gli elementi di A minori di x e la seconda gli elementi di A maggiori di x.
- dove si trova l'elemento di rango k?
  - se $|A_1| \geq k$ allora l'elemento di rango k è nel vettore A_1
  - se $|A_1| = k - 1$ allora l'elemento di rango k è proprio x.
  - se $|A_1| < k - 1$ allora l'elemento di rango k è in A_2 è l'elemento di rango $k - |A_1| - 1$ in A_2

Nota che: dopo aver costruito dalla lista A di n elementi le due liste $A1$ e $A2$, grazie al test sulla cardinalità di $A1$ il problema della selezione dell'elemento di rango k o risulta risolto o viene ricondotto alla selezione di un elemento in una lista con meno di n elementi.

```
def selezione2(lista, k):
    if (len(lista) == 1): return (lista[0])
    p = lista[0]
    lista1,lista2=[],[]
    for x in lista:
        if x<p: lista1.append(x)
        elif x>p:lista2.append(x)
    if len(lista1) == k-1: return p
    if len(lista1) >= k: return selezione2(lista1, k)
    return selezione2(lista2, k-(len(lista1)+1))
```

```python
def selezione2(lista, k):
    if (len(lista) == 1): return (lista[0])
    p = lista[0]
    lista1,lista2=[],[]
    for x in lista:
        if x<p: lista1.append(x)
        elif x>p:lista2.append(x)
    if len(lista1) == k-1: return p
    if len(lista1) >= k: return selezione2(lista1, k)
    return selezione2(lista2, k-(len(lista1)+1))
```

Complessità: se indichiamo con m la cardinalità massima tra quella delle due sottoliste $lista1$ e $lista2$ risultanti dalla scelta del perno possiamo limitare la complessità $T(n)$ dell'algoritmo con la seguente ricorrenza

$$T(n) \leq T(m) + \Theta(n)$$

Complessità di $selezione2$ **nel caso peggiore:**
La procedura che tripartisce la lista in $lista1$, p e $lista2$ può restituire una partizione massimamente sbilanciata in cui si ha ad esempio

$$|lista1| = 0 \quad \text{e} \quad |lista2| = n - 1.$$

questo accade quando il perno p scelto risulta l'elemento minimo nella lista. Qualora questo evento sfortunato si ripesse sistematicamente nel corso delle varie partizioni eseguite dall'algoritmo (questo può accadere quando cerco il massimo in una lista ordinata), allora la complessità dell'algoritmo al caso pessimo è catturata dalla seguente equazione:

$$T(n) = T(n-1) + \Theta(n)$$

che risolta dà

$$T(n) = \Theta(n^2)$$

Se avessimo una regola di scelta del perno p in grado di garantire una partizione perfettamente bilanciata (vale a dire $m = \max\{|lista1|, |lista2|\} = n/2$) allora per la complessità $T(n)$ dell'algoritmo avremmo

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n)$$

che produce

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

Chiedere che la partizione sia perfettamente bilanciata forse è chiedere troppo, potremmo allora accontentarci di chiedere che p garantisca partizioni non troppo sbilanciate come ad esempio quelle per cui $m \leq 3n/4$. In questo caso si ha

$$T(n) \leq T\left(\frac{3}{4}n\right) + \Theta(n)$$

che risolta dà ancora $T(n) = O(n)$.
In generale finché m è una frazione di n (anche piuttosto vicina ad n come ad esempio $\frac{99}{100}n$) la ricorrenza dà sempre $T(n) = O(n)$.

IDEA:
scegliamo il perno p a caso in modo equiprobabile tra gli elementi della lista.

- Anche se la scelta "casuale" non produce necessariamente una partizione bilanciata, quanto visto ci fa intuire che la complessità rimane lineare in n.

```
from random import randint

def selezione2_prob(lista, k):
    if (len(lista) == 1): return (lista[0])
    p = lista[randint(0,len(lista)-1)]
    lista1,lista2=[],[]
    for x in lista:
        if x<p: lista1.append(x)
        elif x>p:lista2.append(x)
    if len(lista1) == k-1: return p
    if len(lista1) >= k: return selezione2_prob(lista1, k)
    return selezione2_prob(lista2, k-(len(lista1)+1))
```

Algoritmo probabilistico per la selezione:

```
from random import randint

def selezione2_prob(lista, k):
    if (len(lista) == 1): return (lista[0])
    p = lista[randint(0,len(lista)-1)]
    lista1,lista2=[],[]
    for x in lista:
        if x<p: lista1.append(x)
        elif x>p:lista2.append(x)
    if len(lista1) == k-1: return p
    if len(lista1) >= k: return selezione2_prob(lista1, k)
    return selezione2_prob(lista2, k-(len(lista1)+1))
```

- Un'analisi rigorosa dimostra che, se la scelta del perno avviene in modo equiprobabile a caso tra i vari elementi della lista, il tempo di calcolo dell'algoritmo risulta con alta probabilità lineare in n.
- il tempo dell'algoritmo *selezione2_prob* è $O(n)$ con alta probabilità.
- Ovviamente nel caso peggiore, quando nelle varie partizioni che si succedono nell'iterazione dell'algoritmo si verifica che il perno scelto a caso risulta sempre vicino al massimo o al minimo della lista (e quindi $m \approx n-1$), la complessità dell'algoritmo rimane $O(n^2)$. Però questo accade con probabilità molto piccola.

Vedremo ora un algoritmo deterministico che garantisce complessità $O(n)$ anche al caso pessimo.

Sappiamo che riuscire a selezionare un perno in grado di garantire che nessuna delle due sottoliste $lista1$ e $lista2$ abbia più di cn elementi per una qualche costante $0 < c < 1$, avrebbe come conseguenza una complessità di calcolo $O(n)$

Descriviamo ora un metodo (noto come il mediano dei mediani) per selezionare un perno p che garantisce di produrre **sempre** due sottoliste $lista1$ ed $lista2$ ciascuna delle quali ha al più $\frac{3}{4}n$ elementi:

- **Dividi gli n elementi di $lista$ in gruppi di 5 elementi ciascuno tranne al più l'ultimo che potrebbe averne di meno e considera i primi $\left\lfloor \frac{n}{5} \right\rfloor$ gruppi ottenuti (tutti di 5 elementi ciascuno)**
- **Trova il mediano in ciascuno di questi $\left\lfloor \frac{n}{5} \right\rfloor$ gruppi.**
- **Trova il mediano p dei $\left\lfloor \frac{n}{5} \right\rfloor$ mediani.**
- **Usa p come perno di $lista$.**

Vedremo ora un algoritmo deterministico che garantisce complessità $O(n)$ anche al caso pessimo.

- **Dividi gli n elementi di *lista* in gruppi di 5 elementi ciascuno tranne al più l'ultimo che potrebbe averne di meno e considera i primi $\left\lfloor \frac{n}{5} \right\rfloor$ gruppi ottenuti (tutti di 5 elementi ciascuno)**
- **Trova il mediano in ciascuno di questi $\left\lfloor \frac{n}{5} \right\rfloor$ gruppi.**
- **Trova il mediano p dei $\left\lfloor \frac{n}{5} \right\rfloor$ mediani.**
- **Usa p come perno di *lista*.**

ESEMPIO:

lista=[15, 2, 10, 16, 21,12, 1, 9, 11, 17, 22, 3, 8,13, 14, 4, 9, 19, 5, 6, 20, 23,18, 7]

15, 2, 10, 16, 21 | 12, 1, 9, 11, 17 | 22, 3, 8, 13, 14 | 4, 19, 5, 6, 20 | 23, 18, 7

2,10, 15, 16, 21 | 1, 9, 11,12, 17 | 3, 8, 13, 14, 22 | 4, 5, 6, 19, 20 |

15, 11, 13, 6

6, 11, 13, 15

p=11

PROPRIETA': Scegliendo il perno con cui partizionare una *lista* di n elementi , con $n > 120$, in base alla regola appena descritta si è sicuri che la dimensione di ciascuna delle due sottoliste *lista*1 e *lista*2 ottenute sarà limitata da $\frac{3}{4}n$.

PROVA:

- il perno scelto p ha la proprietà di trovarsi in posizione $\lceil \frac{n}{10} \rceil$ nella lista degli $\lfloor \frac{n}{5} \rfloor$ mediani selezionati in *lista*.
  Ci sono dunque $\lceil \frac{n}{10} \rceil - 1$ mediani di valore inferiore a p e $\lfloor \frac{n}{5} \rfloor - \lceil \frac{n}{10} \rceil$ mediani di valore superiore a p.
- Prova che $|lista2| < \frac{3}{4}n$:
  - considera gli $\lceil \frac{n}{10} \rceil - 1$ mediani di valore inferiore a p.
  - Ognuno di questi mediani appartiene ad un gruppo di 5 elementi in n.
  - Ci sono dunque in n altri 2 elementi inferiori a p per ogni mediano. In totale abbiamo $3\left(\lceil \frac{n}{10} \rceil - 1\right) \geq 3\frac{n}{10} - 3$ elementi di *lista* che finiranno in *lista*1.
  - $|lista2| \leq n - (3\frac{n}{10} - 3) = \frac{7}{10}n + 3 \leq \frac{3}{4}n$ (dove l'ultima diseguaglianza segue dal fatto che $n \geq 120$)
- Prova che $|lista1| < \frac{3}{4}n$:
  - $\lfloor \frac{n}{5} \rfloor - \lceil \frac{n}{10} \rceil \geq \left(\frac{n}{5} - 1\right) - \left(\frac{n}{10} + 1\right) = \frac{n}{10} - 2$ mediani di valore superiore a p.
  - Ognuno di questi mediani appartiene ad un gruppo di 5 elementi in n.
  - Ci sono dunque in n altri 2 elementi superiori a p per ogni mediano. In totale abbiamo almeno $3\frac{n}{10} - 6$ elementi di *lista* che finiranno in *lista*2.
  - $|lista1| \leq n - (3\frac{n}{10} - 6) = \frac{7}{10}n + 6 \leq \frac{3}{4}n$ (dove l'ultima diseguaglianza segue dal fatto che $n \geq 120$)

Implementazione dell'algoritmo selezione:

```
from math import ceil
def selezione(lista, k):
    # seleziona il k-mo elemento scegliendo ogni volta
    # il perno con la regola del mediano dei mediani
    if len(lista) <= 120:
        lista.sort( )
        return lista[k-1]
    # inizializza  B con i mediani dei len(lista)//5 gruppetti
    # di lista, ciascuno di 5 elementi
    lista1=ordina(lista)
    B=[lista1[5*i+2] for i in range(len(lista1)//5)]
    # individua il perno p con la regola del mediano dei mediani
    p=selezione(B,ceil(len(lista)/10))
    lista1 = [x for x in lista if x < p]
    lista2 = [x for x in lista if x > p]
    if len(lista1) == k-1: return p
    if len(lista1) >= k: return selezione(lista1, k)
    return selezione(lista2, k-len(lista1)-1)
```

```
def ordina(lista):
    # considera i primi len(lista)//5 gruppetti
    # di 5 elementi di lista, li ordina
    # e li restituisce come lista1
    lista1=[]
    for i in range(len(lista)//5):
        a=lista[5*i:5*i+5]
        a.sort()
        lista1.extend(a)
    return lista1
```

Implementazione dell'algoritmo selezione:

```python
from math import ceil
def selezione(lista, k):
    # seleziona il k-mo elemento scegliendo ogni volta
    # il perno con la regola del mediano dei mediani
    if len(lista) <= 120:
        lista.sort( )
        return lista[k-1]
    # inizializza  B con i mediani dei len(lista)//5 gruppetti
    # di lista, ciascuno di 5 elementi
    lista1=ordina(lista)
    B=[lista1[5*i+2] for i in range(len(lista1)//5)]
    # individua il perno p con la regola del mediano dei mediani
    p=selezione(B,ceil(len(lista)/10))
    lista1 = [x for x in lista if x < p]
    lista2 = [x for x in lista if x > p]
    if len(lista1) == k-1: return p
    if len(lista1) >= k: return selezione(lista1, k)
    return selezione(lista2, k-len(lista1)-1)
```

```python
def ordina(lista):
    # considera i primi len(lista)//5 gruppetti
    # di 5 elementi di lista, li ordina
    # e li restituisce come lista1
    lista1=[]
    for i in range(len(lista)//5):
        a=lista[5*i:5*i+5]
        a.sort()
        lista1.extend(a)
    return lista1
```

Complessità:

- nota che:
  - ordinare 120 elementi richiede tempo $O(1)$.
  - ordinare una lista di n elementi in gruppetti da 5 richiede tempo $O(n)$.
  - selezionare i mediani dei mediani di gruppi da 5 da una lista in cui gli elementi sono stati ordinati in gruppetti da 5 richiede tempo $O(n)$.

- Sappiamo che per $n \geq 120$ risulta $|lista1| \leq \frac{3}{4}n$ e $|lista2| \leq \frac{3}{4}n$.
  Dunque per la complessità $T(n)$ dell'algoritmo si ha

$$T(n) \leq \begin{cases} O(1) & \text{se } n \leq 120 \\ T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right) + O(n) & \text{altrimenti} \end{cases}$$

Notiamo che la ricorrenza è del tipo

$$T(n) = T(\alpha \cdot n) + T(\beta \cdot n) + O(n) \text{ con } \alpha + \beta = 1/5 + 3/4 = 19/20 < 1.$$

mostreremo nel prossimo lucido che ricorrenze di questo tipo hanno tutte

come soluzione

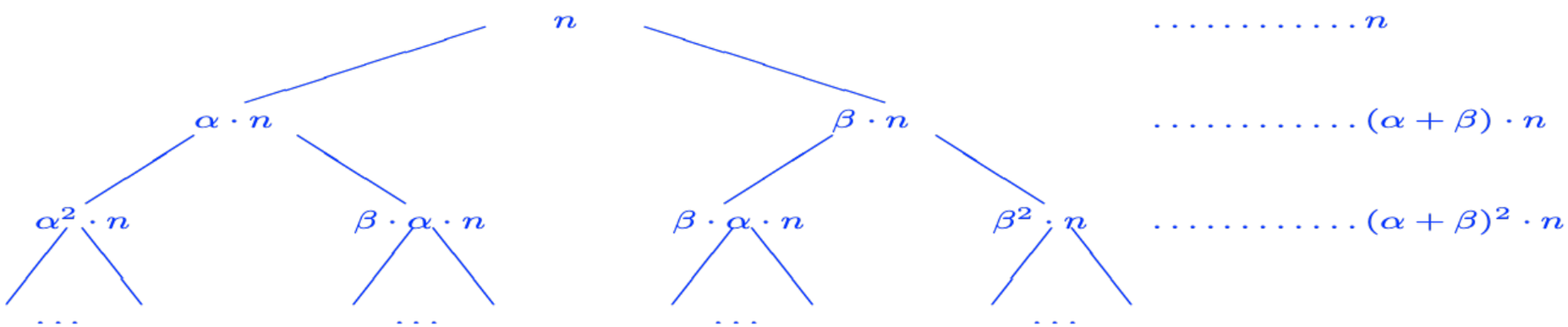
$$T(n) = O(n).$$

ESERCIZIO:

Se $T(n) = T(\alpha \cdot n) + T(\beta \cdot n) + cn$ e $\alpha + \beta < 1$, allora $T(n) = O(n)$.

PROVA:

- Il fatto che $\alpha + \beta$ sia inferiore ad 1 gioca un ruolo fondamentale nella prova.
  Consideriamo l'albero delle chiamate ricorsive generato dalla ricorrenza e analizziamone il costo per livelli.



- Al primo livello abbiamo un costo $(\alpha+\beta)\cdot n$, al secondo un costo $(\alpha+\beta)^2 \cdot n$ al terzo un costo $(\alpha+\beta)^3 \cdot n$ e così via.

- Il tempo di esecuzione totale è la somma dei contributi dei vari livelli:

$$T(n) < c{\cdot}n + c{\cdot}(\alpha+\beta){\cdot}n + c{\cdot}(\alpha+\beta)^2{\cdot}n \ldots = c{\cdot}n{\cdot}\sum_{i=0}^{\infty}(\alpha+\beta)^i = c{\cdot}n\frac{1}{1-(\alpha+\beta)} = \Theta(n)$$

dove nel calcolare la serie abbiamo sfruttato il fatto che $\alpha + \beta < 1$ e la serie geometrica $\sum_{i=0}^{\infty} x^i$, con $x < 1$, converge a $\frac{1}{1-x}$.

Abbiamo quindi dimostrato che il problema della selezione può essere risolto in tempo lineare.
Abbiamo infatti un algoritmo che risolve il problema in $O(n)$ al caso pessimo.
Tuttavia, a causa delle grandi costanti moltiplicative nascoste dall'$O(n)$, nella pratica l'algoritmo randomizzato che ha tempo $O(n)$ con alta probabilità si comporta molto meglio.

ESERCIZIO:

Data una lista di n interi A, un'inversione in A è una coppia $(A[i], A[j])$ con $0 \leq i < j < n$, per cui $A[i] > A[j]$.

Ad esempio per $A = \boxed{1 \mid 3 \mid 4 \mid 2 \mid 5 \mid 3}$ sono presenti le inversioni $(3, 2)$, $(4, 2)$, $(4, 3)$ e $(5, 3)$.

PROBLEMA: progettare un algoritmo che, data la lista A, in tempo $O(n \log n)$ determina il numero di inversioni presenti nella lista A.

L'algoritmo di ricerca esaustiva che esamina tutte le coppie di elementi della lista richiede tempo $O(n^2)$.

Un algoritmo basato sul divide et impera è il seguente:

- **separa la lista in due metà X e Y**
- **conta ricorsivamente le inversioni in ciascuna due metà**
- **conta le inversioni (x, y) con $x \in X$ e $y \in Y$**
- **restituisci la somma dei tre numeri trovati.**

$A =$

1	5	4	8	10	2	6	9	3	7

$X =$

1	5	4	8	10

$(5, 4)$

$Y =$

2	6	9	3	7

$(6, 3), (9, 3), (9, 7)$

$X =$

1	5	4	8	10

$Y =$

2	6	9	3	7

$(5, 2), (5, 3), (4, 2), (4, 3), (8, 2), (8, 6), (8, 3), (8, 7), (10, 2), (10, 6), (10, 9), (10, 3), (10, 7)$

restituisce $1 + 3 + 13 = 17$

- separa il vettore in due metà X e Y
- conta ricorsivamente le inversioni in ciascuna delle due metà
- conta le inversioni (x, y) con $x \in X$ e $y \in Y$
- restituisci la somma dei tre conteggi.

- il conteggio diretto delle inversioni (x, y) con $x \in X$ e $y \in Y$ richiede $\frac{n}{2} \times \frac{n}{2} = \Theta(n^2)$ confronti.
- in questo modo per la complessità dell'algoritmo si ha:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$$

- e quindi $T(n) = O(n^2)$
  (terzo caso del Master Theorem con $a = b = 2$ e $c = 2$)

Se vogliamo migliorare serve un modo più furbo per combinare le due soluzioni.

$X =$

1	5	4	8	10

$Y =$

2	6	9	3	7

- conta le inversioni (x, y) con x in X e y in Y

IDEA:

- ordina X
- per ogni elemento y in Y:
  - effettua una ricerca binaria in X per scoprire quanti elementi di X sono più grandi di y

$X =$

1	5	4	8	10

$Y =$

2	6	9	3	7

$X =$

1	4	5	8	10

$Y =$

2	6	9	3	7
4,	2,	1,	4,	2

Le inversioni (x, y) con x in X e y in Y sono $4 + 2 + 1 + 4 + 2 = 13$

IDEA:

- Per ogni elemento $y \in Y$
  - effettua una ricerca binaria in X per scoprire quanti elementi di X sono più grandi di y:
    - cerca in X la posizione i del primo elemento a valore $> y$ ($i = |X| + 1$ se questo elemento non c'è)
    - il numero di inversioni che coinvolgono y è quindi: $|X| + 1 - i$.

$X =$

1	4	5	8	10

$Y =$

2	6	9	3	7

2

- **conta ricorsivamente quante inversioni ci sono nelle due sottoliste**
  - **ordina la sottolista X**
  - **per ogni elemento della sottolinea Y calcola con ricerca binaria quante inversioni forma con X**
- **restituisci la somma totale delle inversioni calcolate**

- ordinare il vettore X richiede $O(n \log n)$.
- il conteggio delle inversioni (x, y) con $x \in X$ e $y \in Y$ tramite ricerca binaria nel vettore ordinato X richiede ora tempo $\frac{n}{2} \times O(\log n) = O(n \log n)$.
- in questo modo per la complessità dell'algoritmo si ha:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$$

- e quindi $T(n) = O(n \log^2 n)$
  (secondo caso del Master Theorem con $a = b = 2$ e $k = 1$)

Se vogliamo migliorare il tempo serve un modo più furbo per combinare le due soluzioni.

IDEA2:

- manteniamo ordinati entrambi gli insieme X e Y
- scandiamo X e Y da sinistra a destra:
- quando confrontiamo x_i con y_j:
  - se $x_i < y_j$ allora x_i non è invertito con alcun elemento alla sinistra in Y (incrementiamo l'indice i)
  - se $x_i > y_j$ allora y_j è invertito con ogni elemento alla sinistra di X (abbiamo trovato $|X| + 1 - i$ inversioni, incrementiamo l'indice j)

$X =$	1	4	5	8	10
	i				
		i			
		i			
		i			
			i		
				i	
				i	
					i

$Y =$	2	3	6	7	9	
	j					0 inversioni
	j					4 inversioni
		j				4 inversioni
			j			0 inversioni
			j			0 inversioni
			j			2 inversioni
				j		2 inversioni
					j	0 inversioni
					j	1 inversioni

IDEA2:

- manteniamo ordinati entrambi gli insieme X e Y
- scandiamo X e Y da sinistra a destra:
- quando confrontiamo x_i con y_j:
  - se $x_i < y_j$ allora x_i non è invertito con alcun elemento alla sinistra in Y (incrementiamo l'indice i)
  - se $x_i > y_j$ allora y_j è invertito con ogni elemento alla sinistra di X (abbiamo trovato $|X| + 1 - i$ inversioni, incrementiamo l'indice j)

- modifichiamo la funzione ricorsiva di modo che oltre a restituire il numero di inversioni ordina anche gli elementi della lista in questo modo al momento della ricerca delle inversioni con un estremo in X e l'altro in Y, le due parti X e Y risultano ordinate.
- contare le inversioni con x in X e y in Y utilizzando l'idea 2 richiede $O(n)$
- prima di restituire il numero di inversioni totali, con una funzione $fondi$ di tempo $O(n)$ dalle due parti ordinate X e Y otteniamo la lista ordinata.

-il tempo richiesto dalla procedura per ricomporre le due soluzioni è dunque $O(n)$.

-la ricorsione che cattura la complessità dell'algoritmo è $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

- la complessità è quindi $O(n \log n)$
(terzo caso del Master Theorem con $a = b = 2$ e $c = 1$)

Implementazione:

```
def inversioni(lista):
    return inversioniR(lista, 0, len(lista)-1)


def inversioniR(lista,i,j):
    if i==j: return 0
    k=(i+j)//2
    inv1=inversioniR(lista, i,k)
    inv2=inversioniR(lista, k+1,j)
    inv3=conta(lista,i,k,j)
    fondi(lista, i,k,j)
    return inv1+inv2+inv3


>>>lista=[1,5,4,8,10,2,6,9,3,7]
>>> inversioni(lista)
17
```

```
def conta(lista,i,k,j):
    t=0
    a,b=i,k+1
    while a<=k and b<=j:
        if lista[a]<lista[b]:
            a+=1
        else:
            t+=k-a+1
            b+=1
    return t
```

```
def fondi(lista, i,k,j):
    lista1=[]
    a,b=i,k+1
    while a<=k and b<=j:
        if lista[a]<lista[b]:
            lista1.append(lista[a])
            a+=1
        else:
            lista1.append(lista[b])
            b+=1
    while a<=k:
        lista1.append(lista[a])
        a+=1
    while b<=j:
        lista1.append(lista[b])
        b+=1
    for a in range(len(lista1)):
        lista[i+a]=lista1[a]
```

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

che risolta dà $O(n \log n)$