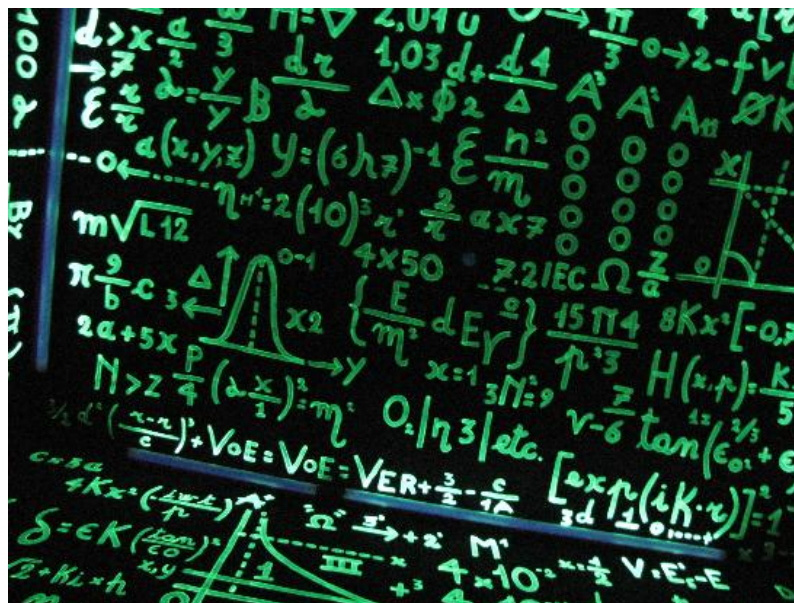


Dall'Algoritmo al Codice



Il Pensiero Computazionale

Percorso Formativo per i Docenti della Scuola Secondaria di II Grado

Lezione 1 - Parte 2 (per coder "esperti")

Stefano Forti, Davide Neri e Luca Rinaldi

Algoritmi (e Python)

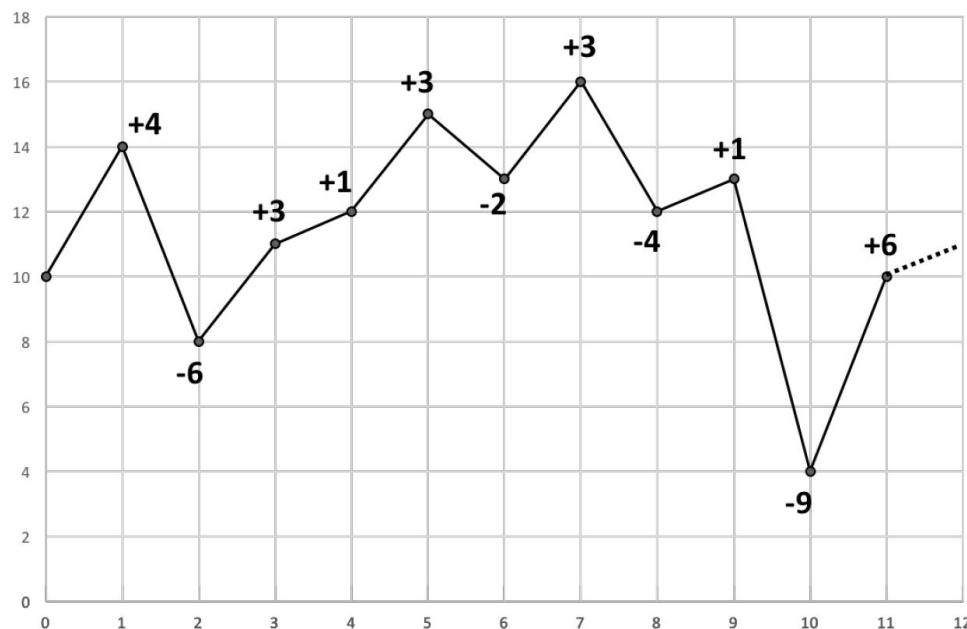
Risolveremo tre problemi con Python:

1. Un Problema Finanziario
2. Multi-Key Quick Sort
3. Il Problema della Partizione

Un Problema Finanziario

Andamento delle quotazioni di una particolare società S.

(asse x = giorni di un anno, asse y = quotazione di un'azione)



Obiettivo: determinare l'istante di acquisto a e quello di vendita v al fine di garantire che la differenza di quotazione tra la vendita e l'acquisto sia *massima*.

Sotto Sequenza Massima

Data una lista D di n interi (positivi e negativi), come si può stabilire la sottolista di somma massima, ovvero trovare due indici, a e b , tali da ottenere il massimo

$$\max_{a,b} \left\{ \sum_{i=a}^b D[i] \right\}.$$

Esistono almeno tre soluzioni rispettivamente di complessità cubica $O(n^3)$, quadratica $O(n^2)$ e lineare $O(n)$.

🧐 Scrivere lo pseudocodice di almeno due soluzioni e implementare la più efficiente.

Algoritmo Cubico (Pseudocodice)

Si provano tutti i possibili intervalli $[i, j]$ e si memorizza la soluzione migliore trovata ogni volta.

programma CUBICO (D)

// Il vettore $D[1 : n]$ consiste di n numeri positivi e negativi

1. $MaxSomma \leftarrow -\infty; a \leftarrow 1; v \leftarrow 0;$
2. **for** $i = 1$ **to** n
3. **for** $j = i$ **to** n
4. $Tmp \leftarrow 0;$ // Tmp è un valore temporaneo
5. **for** $k = i$ **to** j
6. $Tmp \leftarrow Tmp + D[k];$
7. **if** $Tmp > MaxSomma$
8. $MaxSomma \leftarrow Tmp; a \leftarrow i; v \leftarrow j;$
9. **print** “Il guadagno massimo è” $MaxSomma;$
10. **print** “Esso si realizza nell’intervallo di giorni” $[a, v];$

Algoritmo Cubico (Codice)

```
def cubico(d):  
    n = len(d)                                # n indica il numero di elementi di d  
    max_somma = -float('inf')  
    a = 1  
    v = 0  
  
    for i in range(1, n):  
        for j in range(i, n):  
            tmp = 0    # tmp e' un valore temporaneo  
            for k in range(i, j + 1):    # sommiamo gli elementi in d[i,j]  
                tmp = tmp + d[k]  
            if tmp > max_somma:  
                max_somma = tmp  
                a = i  
                v = j  
  
    print ("Il guadagno massimo e' {}".format(max_somma))  
    print ("Esso si realizza nell'intervallo di giorni [{},{}]".format(a, v))  
    print ("Porzione di d avente somma massima {}".format(d[a:v+1]))
```

Algoritmo Quadratico (Pseudocodice)

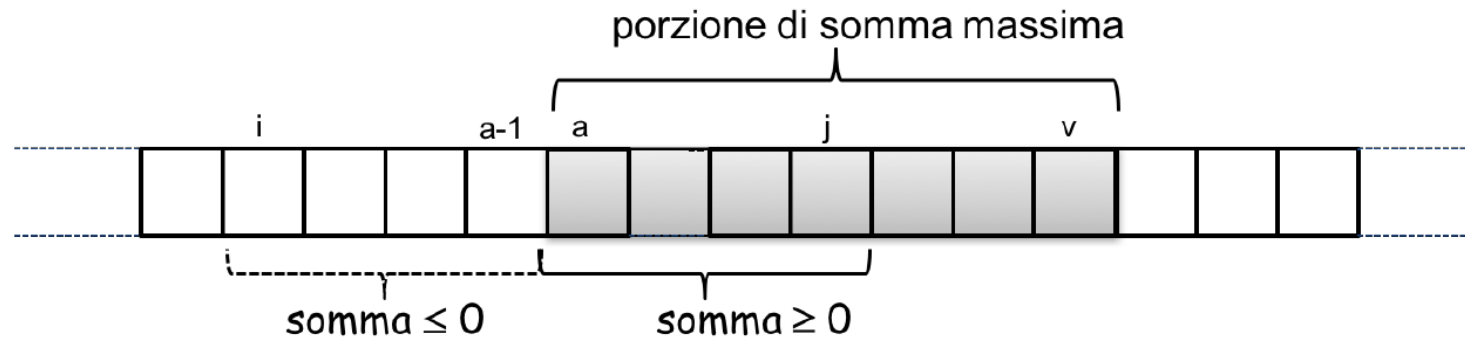
Si provano tutti i possibili intervalli in $[1, n]$ mediante il calcolo incrementale della somma di ogni porzione di vettore esaminata, memorizzando la soluzione migliore trovata.

```
programma QUADRATICO ( $D$ )  
// Il vettore  $D[1 : n]$  consiste di  $n$  numeri positivi e negativi  
1.   $MaxSomma \leftarrow -\infty; a \leftarrow 1; v \leftarrow 0;$   
2.  for  $i = 1$  to  $n$   
3.     $Tmp \leftarrow 0;$   
4.    for  $j = i$  to  $n$   
5.       $Tmp \leftarrow Tmp + D[j];$   
6.      if  $Tmp > MaxSomma$   
7.         $MaxSomma \leftarrow Tmp; a \leftarrow i - 1; v \leftarrow j;$   
8.  print “Il guadagno massimo è”  $MaxSomma;$   
9.  print “Esso si realizza nell’intervallo di giorni”  $[a, v];$ 
```

Algoritmo Quadratico (Codice)

```
def quadratico(d):  
    n = len(d)                                # n indica il numero di elementi di d  
    max_somma = -float('inf')  
    a = 1  
    v = 0  
  
    for i in range(1, n):  
        tmp = 0                                # tmp e' un valore temporaneo  
        for j in range(i, n):  
            tmp = tmp + d[j]  
            if tmp > max_somma:  
                max_somma = tmp  
                a = i  
                v = j  
  
    print ("Il guadagno massimo e' {}".format(max_somma))  
    print ("Esso si realizza nell'intervallo di giorni [ {}, {} ]".format(a, v))  
    print ("Porzione di d avente somma massima {}".format(d[a:v+1]))
```


Algoritmo Lineare (Idea)



Proprietà 1 - Ogni porzione che termina immediatamente prima della porzione di somma massima, e quindi avente la forma $D[i : a - 1]$, ha somma negativa.

Proprietà 2 - Ogni porzione che inizia ove inizia la porzione di somma massima ed è inclusa in essa, e quindi avente la forma $D[a : j]$, ha somma positiva.

Algoritmo Lineare (Pseudocodice)


Si provano n intervalli in $[1, n]$, memorizzando la soluzione migliore trovata ogni volta.

```
programma LINEARE ( $D$ )  
// Il vettore  $D$  consiste di  $n$  numeri positivi e negativi  
1.   $MaxSomma \leftarrow -\infty$ ;  $v \leftarrow 0$ ;  $Tmp \leftarrow 0$ ;  
2.   $a \leftarrow 1$ ;  $atmp \leftarrow 1$ ;  
3.  for  $i = 1$  to  $n$   
4.       $Tmp \leftarrow Tmp + D[i]$ ;  
5.      if  $Tmp > MaxSomma$   
6.           $MaxSomma \leftarrow Tmp$ ;  $v \leftarrow i$ ;  $a \leftarrow atmp$ ;  
7.      if  $Tmp < 0$   
8.           $Tmp \leftarrow 0$ ;  $atmp \leftarrow i + 1$ ;  
9.  print “Il guadagno massimo è”  $MaxSomma$ ;  
10. print “Esso si realizza nell’intervallo di giorni”  $[a, v]$ ;
```

Algoritmo Lineare (Codice)

```
def lineare(d):  
    n = len(d) # n indica il numero di elementi di d  
    max_somma = -float('inf')  
    a = 1  
    v = 0  
    atmp = a # atmp e' un indice temporaneo  
    tmp = 0 # tmp e' un valore temporaneo  
  
    for i in range(1, n):  
        tmp = tmp + d[i]  
        if tmp > max_somma:  
            max_somma = tmp  
            a = atmp  
            v = i  
        if tmp < 0:  
            tmp = 0  
            atmp = i + 1  
  
    print ("Il guadagno massimo e' {}".format(max_somma))  
    print ("Esso si realizza nell'intervallo di giorni [ {}, {} ]".format(a, v))  
    print ("Porzione di d avente somma massima {}".format(d[a:v+1]))
```

Prendere il tempo!

 In Python possiamo cronometrare la durata di esecuzione di un programma. Basta importare la libreria `time` e usare la funzione `time.time()` come nell'esempio qui sotto:

```
import time
import random

giorni = 5000
d = [random.randrange(-10,10) for num in range(giorni)]

start = time.time()
cubico(d)
stop = time.time()
print('Cubico ', stop-start, "secondi.")
```

Confrontare il tempo di esecuzione dei tre algoritmi che risolvono il problema precedente.

Risultati

Risultati dei tempi di esecuzione con `d = 5000` sulla "nostra" macchina:

```
Cubico    16.844367742538452 secondi.  
Quadratico 0.07895207405090332 secondi.  
Lineare   0.0009975433349609375 secondi.
```

Multi-key QuickSort (Idea e Complessità)

Come si ordina una collezione S di n parole di lunghezza L ?

💡 *L'algoritmo è simile al QuickSort.*

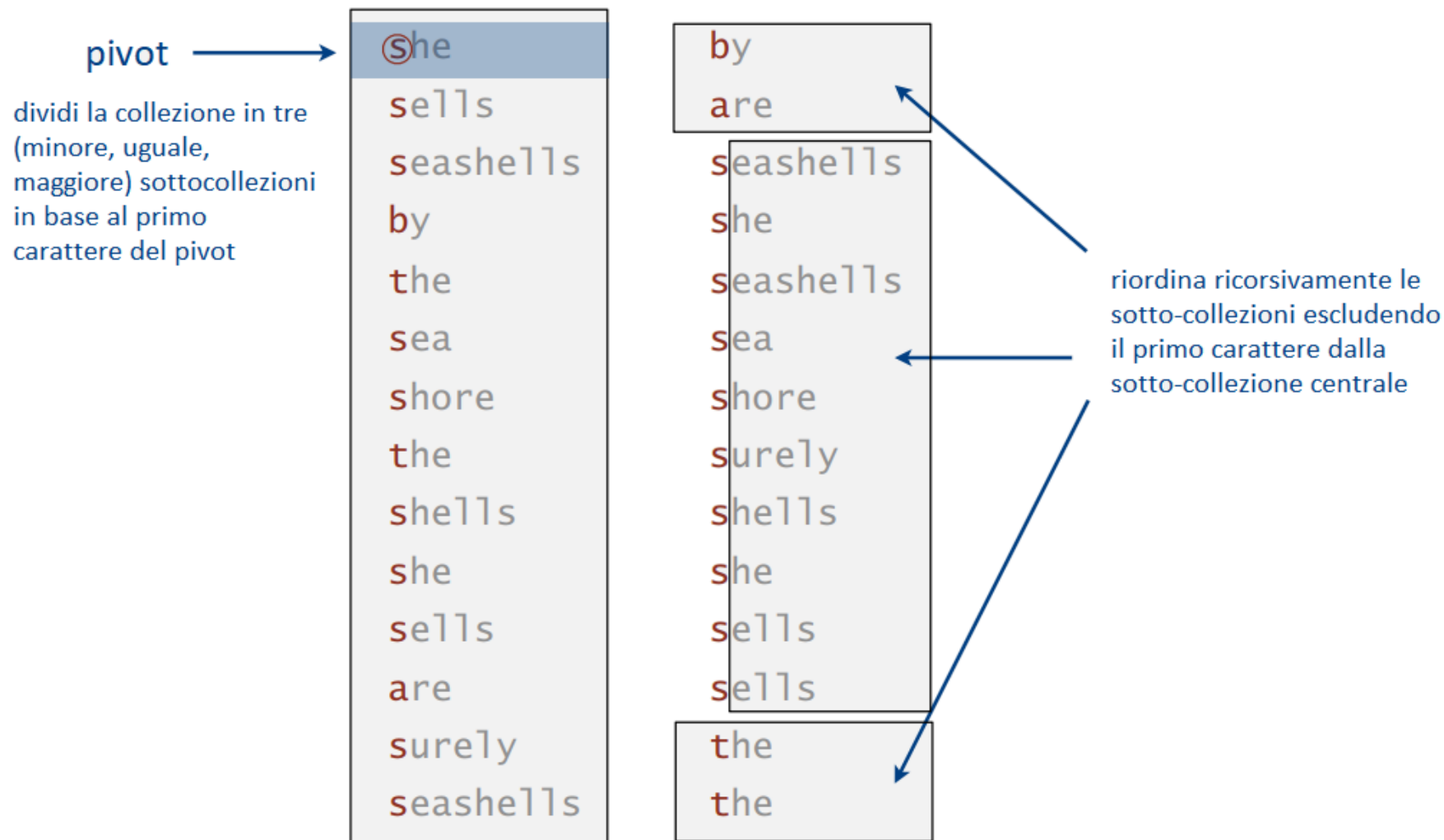
L'idea di base segue due passi che si applicano a $\langle S, r \rangle$:

1. Sceglie una stringa p (*pivot*) tra quelle da ordinare e divide la collezione S in tre sotto-collezioni, una $S_{<}$ con le stringhe s tali che $s[r] < p[r]$, una $S_{=}$ con le stringhe s tali che $s[r] = p[r]$, la terza $S_{>}$ con le stringhe s tali che $s[r] > p[r]$.
2. Ripete (1) su $\langle S_{<}, r \rangle$, $\langle S_{=}, r + 1 \rangle$ e $\langle S_{>}, r \rangle$.

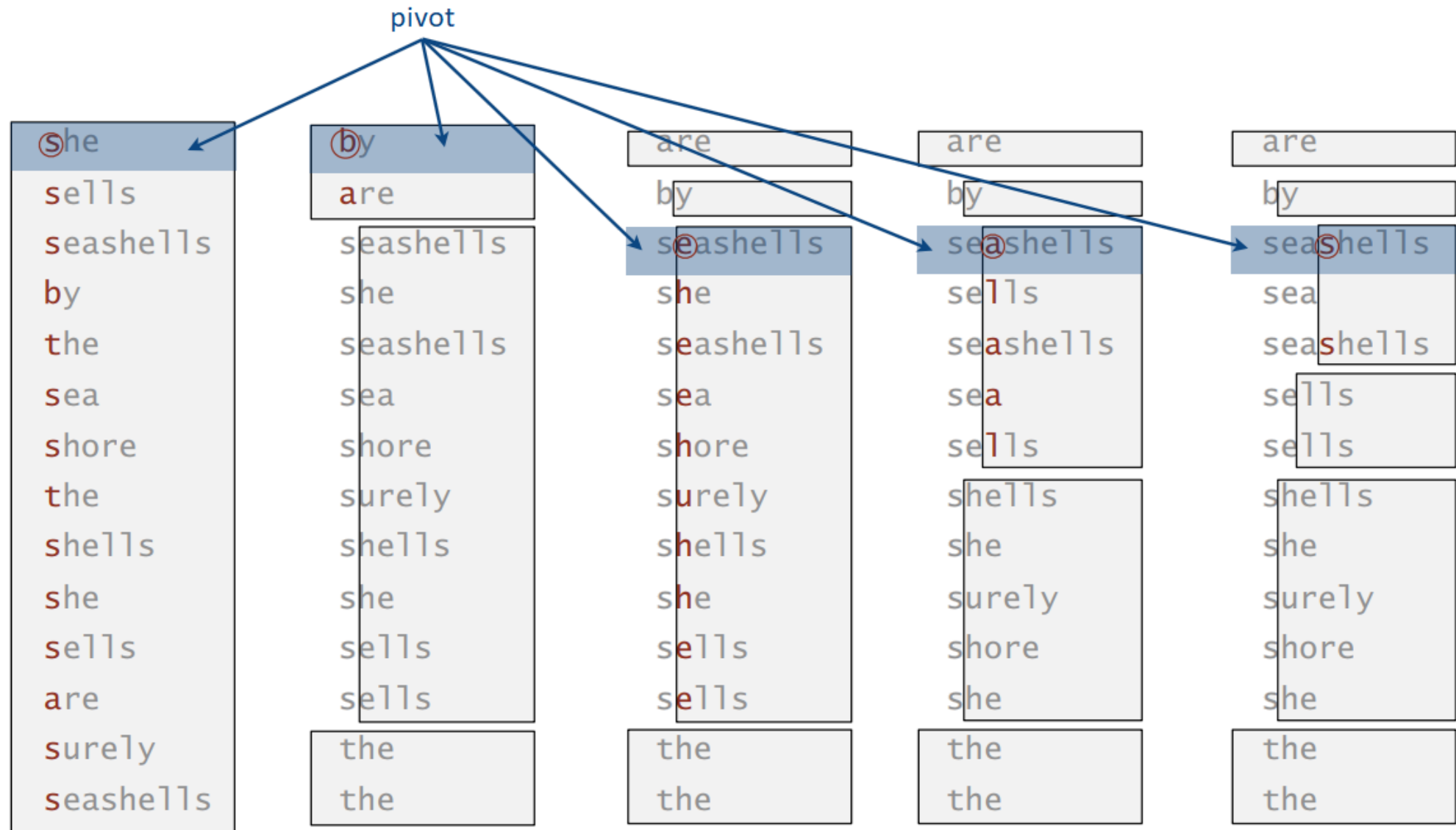
La complessità in tempo è $O(D + n \lg n)$ dove D è la lunghezza del prefisso distintivo di S (ovvero, quello che distingue una stringa da tutte le altre).

Multi-key QuickSort (Demo)

Inizialmente $r = 0$.



Multi-key QuickSort (Demo)



Multi-key QuickSort (Codice)

```
def multkeyQS(S,k):  
    if (len(S) <= 1):  
        return S  
    pivot_pos = random.randint(0, len(S)-1)  
    pivot_char = S[pivot_pos][k]  
    S1 = []  
    S2 = []  
    S3 = []  
    for s in S:  
        if (s[k] < pivot_char):  
            S1.append(s)  
        elif (s[k] == pivot_char):  
            S2.append(s)  
        else:  
            S3.append(s)  
    L1 = multkeyQS(S1,k)  
    L2 = multkeyQS(S2,k+1)  
    L3 = multkeyQS(S3,k)  
    return L1+L2+L3
```

Multi-key QuickSort (Codice)

```
B = []
with open("280000_parole_italiane.txt","r") as file:
    for parola in file.readlines():
        parola = parola.replace('\n', ' ')
        B.append(parola)

start = time.time()
B_sorted = multikeyQS(B,0)
stop = time.time()

print('MQSort',stop-start)
```

Multi-Key QuickSort In-Place (🧐 Molto Avanzato!)

```
def multikeyQS2(a, lo, hi, r):  
    if hi <= lo:  
        return  
    lt = lo  
    gt = hi  
    v = a[lo][r]  
    i = lo + 1  
  
    while i <= gt:  
        t = a[i][r]  
        if t < v:  
            swap(a, lt, i)  
            lt += 1  
            i += 1  
        elif t > v:  
            swap(a, i, gt)  
            gt -= 1  
        else:  
            i += 1  
  
    multikeyQS2(a, lo, lt-1, r)  
    if ord(v) >= 0:  
        multikeyQS2(a, lt, gt, r+1)  
    multikeyQS2(a, gt+1, hi, r)
```

Il Problema della Partizione

Se abbiamo a disposizione due dischi rigidi da s byte ciascuno e vogliamo usarli per memorizzare n file che occupano $2s$ byte in totale, è possibile dividere i file in due **partizioni** ciascuna di s byte?

Formalmente:

Dato un insieme di interi $A = \{a_0, \dots, a_{n-1}\}$ tali che $\sum_{i=0}^{n-1} a_i = 2s$, vogliamo verificare se esiste $A' \subseteq A$ tale che $\sum_{a_i \in A'} a_i = s$.

Il Problema della Partizione

Una semplice soluzione, ma inefficiente, consiste nel generare tutti i possibili sottoinsiemi di A . Essa impiega un tempo proporzionale a $O(2^n)$, quindi è esponenziale nella dimensione dell'input.

Una soluzione più "furba" prova a risolvere una serie di sottoproblemi per arrivare alla soluzione del problema originale e sfrutta la cosiddetta **programmazione dinamica**.

Determiniamo il booleano $T(i, j)$, per $0 \leq i \leq n$ e $0 \leq j \leq s$, che è **True** se e solo se esiste un sottoinsieme di $A_{i-1} = \{a_0, \dots, a_{i-1}\}$ con somma pari a j .

La soluzione del problema originale si troverà in $T(n, s)$.

Partizione (Idea e Complessità)

Banalmente: $T(0, 0) = \text{True}$ e $T(0, j) = \text{False}$ per $j \neq 0$.

Nel caso generale:

- $T(i, j) = \text{True}$ se $i = 0$ e $j = 0$
- $T(i, j) = \text{True}$ se $i > 0$ e $T(i-1, j) = \text{True}$ (la parte di somma j non contiene a_{i-1})
- $T(i, j) = \text{True}$ se $i > 0$, $j \geq a[i-1]$ e $T(i-1, j-a[i-1]) = \text{True}$ (contiene a_{i-1})
- $T(i, j) = \text{False}$ altrimenti.

Il risultato è una tabella $T(i, j)$ che si riempie con una complessità in tempo pari a $O(ns)$ (dovendo infatti risolvere $(n + 1) \times (s + 1)$ sotto-problemi).

Partizione (Codice)

```
def partizione(a):  
    if (sum(a) % 2) != 0:  
        return False  
  
    n = len(a)  
    s = sum(a)/2  
    parti = []  
  
    for i in range(n+1):  
        parti.append([])  
        for j in range(s+1):  
            parti[i].append(False)  
  
    parti[0][0] = True  
    for i in range(1, n+1):  
        for j in range(s+1):  
            if parti[i-1][j]:  
                parti[i][j] = True  
            if j >= a[i-1] and parti[i-1][j-a[i-1]]:  
                parti[i][j] = True  
    return parti[n][s]
```

Pseudopolinomialità di Partizione

L'algoritmo proposto per il problema della partizione ha un costo $O(ns)$. Tale costo in tempo è **polinomiale** in n e s ma non lo è necessariamente nella dimensione dei dati in ingresso.

Infatti, ciascuno degli n interi da partizionare richiede $k = O(\lg s)$ bit di rappresentazione.

Quindi la dimensione dei dati è $O(nk)$ e il costo dell'algoritmo $O(ns) = O(n2^k)$ che rimane polinomiale solo se si usano interi piccoli rispetto a n (es., $s = O(n^c)$ con c costante fissata).

Questa *anomalia* è dovuta al fatto che, per valori numerici sufficientemente grandi, il problema della partizione è NP-completo.

Esercizio

1. **Il problema dello zaino** Un ladro si introduce in un museo in cui sono esposti gli elementi dell'insieme $A = \{a_0, \dots, a_{n-1}\}$ ciascuno associati a un *valore*(a) e un *peso*(a) interi positivi. Lo zaino del ladro ha una capacità massima `cap`. Come può il ladro usare il paradigma della programmazione dinamica per determinare un sottoinsieme $A' \subseteq A$ tale che il peso totale di A' sia minore di `cap` e tale per cui $\sum_{a \in A'} \text{valore}(a)$ sia il massimo possibile?