Sapienza — Università di Roma
Dip. Scienze Statistiche — Informatica 2019/2020
II canale

Appunti su algoritmi e complessità

Massimo Lauria
<massimo.lauria@uniroma1.it>
https://massimolauria.net/courses/informatica2019

Versione degli appunti aggiornata al 1 novembre 2019.

Avvertenza: queste appunti possono essere soggetti a cambiamenti durante il corso, dovuti a esigenze didattiche oppure alla correzione o al miglioramento dei contenuti. Gli studenti sono pregati di farmi notare tempestivamente qualunque errore o problema che dovessero riscontrare.

Nota bibliografica: Molto del contenuto di questi appunti può essere approfondito nei primi capitoli del libro *Introduzione agli Algoritmi e strutture dati* di Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein.

Indice

Capitolo 1

La ricerca in una sequenza

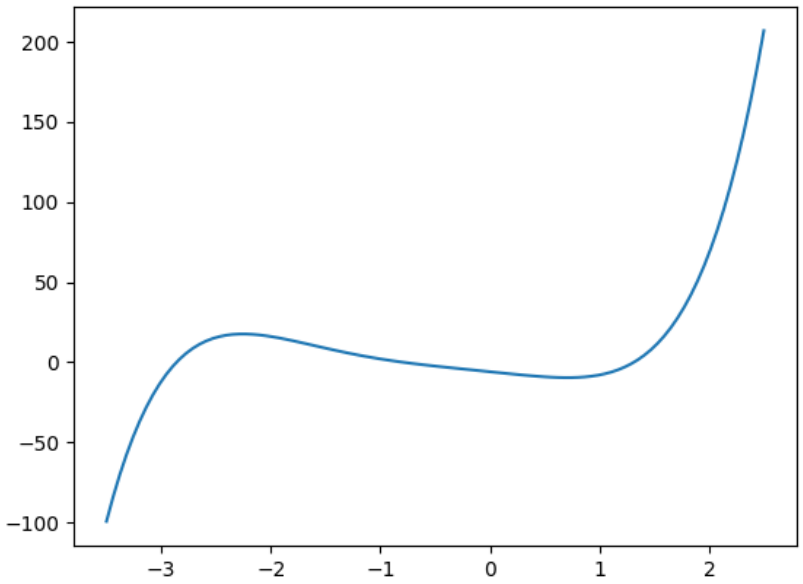
La ragione per cui usiamo i calcolatori elettronici, invece di fare i conti a mano, è che possiamo sfruttare la loro velocità. Tuttavia quando le quantità di dati da elaborare si fanno molto grandi, anche un calcolatore elettronico può diventare lento. Se il numero di operazioni da fare diventa molto elevato, ci sono due opzioni: (1) utilizzare un calcolatore più veloce; (2) scrivere un programma più efficiente (i.e. meno operazioni).

Mentre la velocità dei calcolatori migliora con il tempo, questi miglioramenti sono sempre limitati. Il modo migliore per ridurre il tempo di calcolo è scoprire nuove idee e nuovi algoritmi per ottenere il risultato in modo più efficiente possibile.

In questa parte degli appunti vediamo un problema concreto: la ricerca di un dato. Poi vediamo degli algoritmi per risolverlo.

1.1 Trovare uno zero di una funzione continua

Considerate il problema di trovare un punto della retta $\mathbb{R}$ nel quale un dato polinomio P sia 0, ad esempio con $P(x) = x^5 - 3x^4 + x^3 + 7x - 6$.



che in python si calcola

```
def P(x):                                                    1
    return x**5 + 3*x**4 + x**3 - 7*x - 6                    2
```

Non è sempre possibile trovare uno zero di una funzione matematica. Tuttavia in certe condizioni è possibile per lo meno trovare un punto molto vicino allo zero. (i.e. un punto $x \in [x_0 - \epsilon, x_0 + \epsilon]$ dove $P(x_0) = 0$ e $\epsilon > 0$ è piccolo a piacere).

Il polinomio P ha grado dispari e coefficiente positivo nel termine di grado più alto. Questo vuol dire che

$$\lim_{x \to -\infty} P(x) \qquad \lim_{x \to \infty} P(x)$$

divergono rispettivamente verso $-\infty$ e $+\infty$. In particolare un polinomio è una funzione continua ed in questo caso ci saranno due punti a, b con $a < b$ e $P(a)$ negativo e $P(b)$ positivo. Possiamo usare il teorema seguente per scoprire che P ha per forza uno zero.

Teorema 1 (Teorema degli zeri)**.** *Si consideri una funzione continua $f : [a, b] \to \mathbb{R}$. Se $f(a)$ è negativo e $f(b)$ è positivo (o viceversa), allora deve esistere un x_0 con $a < x_0 < b$ per cui $f(x_0) = 0$.*

Partiamo da due punti $a < b$ per cui P cambi di segno.

```
print(P(-2.0),P(1.0))
```

```
16.0 -8.0
```

Quindi il teorema precedente vale e ci garantisce che esiste uno zero del polinomio tra $a = -2$ e $b = 1$. Ma come troviamo questo punto x_0 tale che $P(x_0)$ sia uguale a 0?

Naturalmente il punto x_0 potrebbe non avere una rappresentazione finita precisa, tuttavia possiamo cercare di trovare un numero abbastanza vicino: diciamo che nelle condizioni del teorema precedente possiamo accontentarci di un risultato che sia compreso tra $x_0 - \epsilon$ e $x_0 + \epsilon$, per qualche valore piccolo $\epsilon > 0$.

Una possibilità è scorrere tutti i punti tra a e b (o almeno un insieme molto fitto di essi). Se troviamo un x tale che $f(x)$ e $f(x + \epsilon)$ abbiano segno diverso, dalla continuità di f sappiamo che uno zero di f è contenuto nell'intervallo $[x, x + \epsilon]$. Quindi possiamo restituire x

```
def trova_zero_A(f,a,b):
  eps = 1 / 1000000
  x = a
  steps=1
  while x <= b:
    if f(x) == 0.0 or f(x)*f(x+eps)<0:
      print("Trovato in {1} passi lo zero {0}.".format(x,steps))
      print(" f({}) = {}".format(x,f(x)))
      print(" f({}) = {}".format(x+eps,f(x+eps)))
      break
    steps +=1
    x += eps
```

```
print( P(-2.0), P(1.0) )
trova_zero_A(P,-2.0,1.0)
```

```
16.0 -8.0
Trovato in 1198815 passi lo zero -0.8011860000765496.
 f(-0.8011860000765496) = 8.36675525572872e-06
 f(-0.8011850000765496) = -8.18738014274345e-07
```

E naturalmente possiamo farlo con qualunque funzione continua

```
import math
def T(x):
    return x + math.cos(x)
```

```
# verifichiamo che ci siano due punti di segno diverso
print( T(-4.0), T(4.0) )
trova_zero_A(T,-4.0,4.0)
```

```
-4.653643620863612 3.346356379136388
Trovato in 3260915 passi lo zero -0.7390859997952081.
 f(-0.7390859997952081) = -1.450319069173922e-06
 f(-0.739084999795208) = 2.232932310164415e-07
```

Questo programma trova uno "zero" di P in 1198815 passi, e uno "zero" di $x + \cos(x)$ in 3260915. Possiamo fare di meglio? Serve un'idea che renda questo calcolo molto più efficiente. Supponiamo che per una funzione continua f abbiamo $f(a) < 0$ e $f(b) > 0$, allora possiamo provare a calcolare f sul punto $c = (a + b)/2$, a metà tra a e b.

- Se $f(c) > 0$ allora esiste $a < x_0 < c$ per cui $f(x_0) = 0$;
- se $f(c) < 0$ allora esiste $c < x_0 < b$ per cui $f(x_0) = 0$;
- se $f(c) = 0$ allora $x_0 = c$.

In questo modo ad ogni passo di ricerca **dimezziamo l'intervallo**.

```
def trova_zero_B(f,a,b):
    eps=1/1000000
    passi = 0
    start,end = a,b
    while end - start > eps :

        mid   = (start + end)/2
        passi = passi + 1

        if f(mid) == 0.0:
            start = mid
            end   = mid
        elif f(start)*f(mid) < 0:
            end   = mid
        else:
            start = mid


    print("Trovato in {1} passi lo zero {0}.".format(start,passi))
    print(" f({}) = {}".format(start,f(start)))
    print(" f({}) = {}".format(end,f(end)))

trova_zero_B(P,-2.0,1.0)
trova_zero_B(T,-4.0,4.0)
```

```
Trovato in 22 passi lo zero -0.8011856079101562.
 f(-0.8011856079101562) = 4.764512533839138e-06
 f(-0.8011848926544419) = -1.8054627952679425e-06
Trovato in 23 passi lo zero -0.7390851974487305.
 f(-0.7390851974487305) = -1.0750207668497325e-07
 f(-0.7390842437744141) = 1.488578440400623e-06
```

Il secondo programma è molto più veloce perché invece di scorrere tutti i punti, o comunque una sequenza abbastanza fitta di punti, in quell'intervallo esegue un dimezzamento dello spazio di ricerca. Per utilizzare

questa tecnica abbiamo usato il fatto che le funzioni analizzate fossero continue. Questo è un elemento **essenziale**. Il teorema degli zeri non vale per funzioni discontinue e `trova_zero_B` si basa su di esso.

> *Se i dati in input hanno della struttura addizionale, questa può essere sfruttata per scrivere programmi più veloci.*

Prima di concludere questa parte della lezione voglio sottolineare che i metodi visti sopra possono essere adattati in modo da avere precisione maggiore. Potete verificare da soli che una precisione maggiore (un fattore dieci, per esempio) costa molto nel caso di `trova_zero_A`, ma quasi nulla nel caso di `trova_zeri_B`.

1.2 Ricerca di un elemento in una lista

Come abbiamo visto nella parte precedente della lezione, è estremamente utile conoscere la struttura o le proprietà dei dati su cui si opera. Nel caso in cui in dati abbiano delle caratteristiche particolari, è possibile sfruttarle per utilizzare un algoritmo più efficiente, che però **non è corretto** in mancanza di quelle caratteristiche.

Uno dei casi più eclatanti è la ricerca di un elemento in una sequenza. Per esempio

- cercare un nome nella lista degli studenti iscritti al corso;
- cercare un libro in uno scaffale di una libreria.

Per trovare un elemento `x` in una sequenza `seq` la cosa più semplice è di scandire `seq` e verificare elemento per elemento che questo sia o meno uguale a `x`. Di fatto l'espressione `x in seq` fa esattamente questo.

```
def ricercasequenziale(x,seq):
    for i,y in enumerate(seq):
        if x==y:
            print("Trovato l'elemento dopo {} passi.".format(i+1))
            return
    print("Non trovato. Eseguiti {} passi.".format(len(seq)))
```

Per testare `ricercasequenziale` ho scritto una funzione

```
test_ricerca(S,sorted=False)
```

che produce una sequenza di `1000000` di numeri compresi tra `-2000000` e `2000000`, generati a caso utilizzando il generatore **pseudocasuale** di py-

thon. Poi cerca il valore 0 utilizzando la funzione `S`. Se il parametro opzionale `sorted` è vero allora la sequenza viene ordinata prima che il test cominci.

```
test_ricerca(ricercasequenziale)
```

```
Trovato l'elemento dopo 421608 passi.
```

Per cercare all'interno di una sequenza di n elementi la funzione di ricerca deve scorrere **tutti** gli elementi. Questo è **inevitabile** in quanto se anche una sola posizione non venisse controllata, si potrebbe costruire un input sul quale la funzione di ricerca non è corretta.

1.3 Ricerca binaria

Nella vita di tutti i giorni le sequenze di informazioni nelle quali andiamo a cercare degli elementi (e.g. un elenco telefonico, lo scaffale di una libreria) sono ordinate e questo ci permette di cercare più velocemente. Pensate ad esempio la ricerca di una pagina in un libro. Le pagine sono numerate e posizionate nell'ordine di enumerazione. È possibile trovare la pagina cercata con poche mosse.

Se una lista `seq` di n elementi è ordinata, e noi cerchiamo il numero 10, possiamo già escludere metà della lista guardando il valore alla posizione $n/2$.

- Se il valore è maggiore di 10, allora 10 non può apparire nelle posizioni successive, e quindi è sufficiente cercarlo in quelle precedenti;
- analogamente se il valore è maggiore di 10, allora 10 non può apparire nelle posizioni precedenti, e quindi è sufficiente cercarlo in quelle successive.

La ricerca binaria è una tecnica per trovare dati all'interno di una sequenza `seq` **ordinata**. L'idea è quella di dimezzare ad ogni passo lo spazio di ricerca. All'inizio lo spazio di ricerca è l'intervallo della sequenza che va da `0` a `len(seq)-1` ed x è l'elemento da cercare.

Ad ogni passo

- si controlla il valore h a metà dell'intervallo;
- se $h > x$ allora x può solo trovarsi prima di v;

- se $h < x$ allora x può solo trovarsi dopo v;

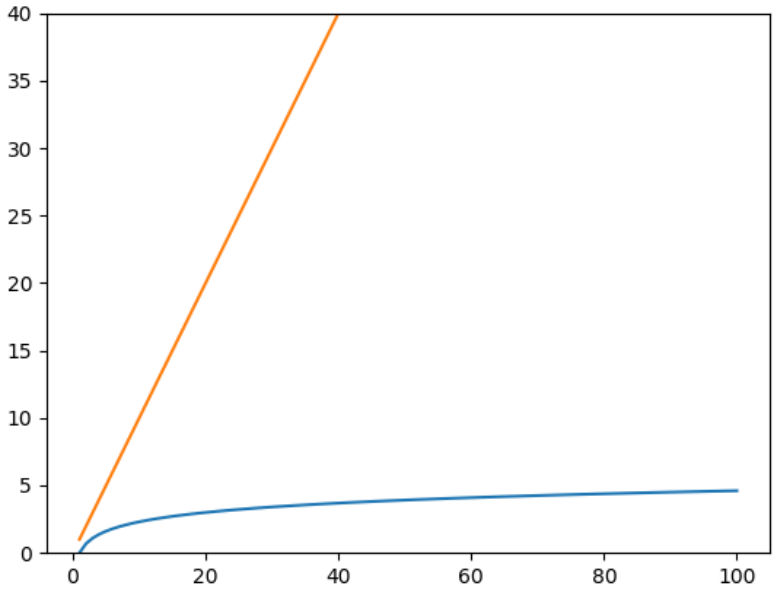
- altrimenti h è uguale al valore cercato.

```
def ricercabinaria(x,seq):
    start=0
    end=len(seq)-1
    step = 0
    while start<=end:
        step += 1
        mid = (end + start) // 2
        half = seq[mid]
        if half == x:
            print("Trovato l'elemento dopo {} passi.".format(step))
            return
        elif half < x:
            start = mid + 1
        else:
            end = mid-1
    print("Non trovato. Eseguiti {} passi.".format(step))
```

```
test_ricerca(ricerca_binaria,sorted=True )
```

```
Trovato l'elemento dopo 19 passi.
```

La nuova funzione di ricerca è estremamente più veloce. Il numero di passi eseguiti è circa uguale al numero di divisioni per due che rendono la lunghezza di `seq` uguale a 1, ovvero una sequenza di lunghezza n richiede $\lceil \log_2 n \rceil$ iterazioni. Mentre la funzione di ricerca che abbiamo visto prima ne richiede n.



Naturalmente ordinare una serie di valori ha un costo, in termini di tempo. Se La sequenza ha n elementi e vogliamo fare R ricerche, ci aspettiamo all'incirca

- Ricerca sequenziale: nR
- Ricerca binaria: $R \log n + \mathrm{Ord}(n)$

operazioni, dove $\mathrm{Ord}(n)$ è il numero di passi richiesti per ordinare n numeri. Quindi se i dati non sono ordinati fin dall'inizio, si può pensare di ordinarli, a seconda del valore di R e di quanto costi l'ordinamento. In generale vale la pena ordinare i dati se il numero di ricerche è abbastanza consistente, anche se non elevatissimo.

Non stiamo neppure parlando di quanto costi mantenere i dati ordinati in caso di inserimenti e cancellazioni.

Ricapitolando: se vogliamo cercare un elemento in una sequenza di lunghezza n, possiamo usare

- ricerca sequenziale in circa n passi
- ricerca binaria in circa $\log n$ passi (se la sequenza è ordinata).

Domanda: la ricerca (binaria o sequenziale) può impiegare più o meno tempo a seconda della posizione dell'elemento trovato, se presente, ed il

numero di passi specificato sopra è **il caso peggiore**. Sapete dire quali sono gli input per cui questi algoritmi di ricerca impiegano il numero massimo di passi, e quali sono quelli per cui l'algoritmo impiega il numero minimo?

1.4 La misura della complessità computazionale

Quando consideriamo le prestazioni di un algoritmo non vale la pena considerare l'effettivo tempo di esecuzione o la quantità di RAM occupate. Queste cose dipendono da caratteristiche tecnologiche che niente hanno a che vedere con la qualità dell'algoritmo. Al contrario è utile considerare il numero di **operazioni elementari** che l'algoritmo esegue.

- accessi in memoria
- operazioni aritmetiche
- ecc. . .

Che cos'è un'operazione elementare? Per essere definiti, questi concetti vanno espressi su modelli computazionali **formali e astratti**, sui quali definire gli algoritmi che vogliamo misurare. Per noi tutto questo non è necessario, ci basta capire a livello intuitivo il numero di operazioni elementari che i nostri algoritmi fanno per ogni istruzione.

Esempi

- una somma di due numeri `float` : 1 op.
- verificare se due numeri `float` sono uguali: 1 op.
- confronto lessicografico tra due stringhe di lunghezza n: fino a n op.
- ricerca lineare in una lista Python di n elementi: fino a n op.
- ricerca binaria in una lista ordinata di n elementi: fino a $\log n$ op.
- inserimento nella 4a posizione in una lista di n elementi: n op.
- `seq[a:b]` : $b - a$ operazioni
- `seq[:]` : `len(seq)` operazioni
- `seq1 + seq2` : `len(seq1) + len(seq2)` operazioni

Capitolo 2

Ordinare dati con Insertion sort

Nota: il contenuto di questa parte degli appunti può essere approfondito nei Paragrafi 2.1 e 2.2 del libro di testo *Introduzione agli Algoritmi e strutture dati* di Cormen, Leiserson, Rivest e Stein.

2.1 Il problema dell'ordinamento

Nella lezione precedente abbiamo visto due tipi di ricerca. Quella *sequenziale* e quella *binaria*. Per utilizzare la ricerca binaria è necessario avere la sequenza di dati ordinata. Ma come fare se la sequenza non è ordinata? Esistono degli algoritmi per ordinare sequenze di dati omogenei (ovvero dati per cui abbia senso dire che un elemento viene prima di un altro).

Facciamo degli esempi.

- `[3, 7, 6, 9, -3, 1, 6, 8]` non è ordinata.
- `[-3, 1, 3, 6, 6, 7, 8, 9]` è la corrispondente sequenza ordinata.
- `['casa', 'cane', 'letto', 'gatto', 'lettore']` non è ordinata.
- `['cane', 'casa', 'gatto', 'letto', 'lettore']` è in ordine lessicografico.
- `[ 7, 'casa', 2.5, 'lettore']` non è ordinabile.

Perché una sequenza possa essere ordinata, è necessario che a due a due tutte le coppie di elementi nella sequenza siano **confrontabili**, ovvero che sia possibile determinare, dati due elementi qualunque a e b nella sequenza, se $a = b$, $a < b$ oppure $a > b$.

2.2 Insertion sort

Insertion sort è uno degli algoritmi di ordinamento più semplici, anche se non è molto efficiente. L'idea dell'insertion sort è simile a quella di una persona che gioca a carte e che vuole ordinare la propria mano.

- Lascia la prima carta all'inizio;
- poi prende la seconda carta e la mette in modo tale che la prima e seconda posizione siano ordinate;
- poi prende la terza carta carta e la mette in modo tale che la prima e seconda e la terza posizione siano ordinate.
- ...

Al passo i-esimo si guarda l'i-esima carta nella mano e la si **inserisce** tra le carte **già ordinate** nelle posizioni da 0 a $i - 1$.

In sostanza se una sequenza $L = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ ha n elementi, l'insertion sort fa iterazioni per i da 1 a $n - 1$. In ognuna:

1. garantisce che gli elementi nelle posizioni $0, \ldots, i - 1$ siano ordinati;
2. trova la posizione $j \leq i$ tale che `L[j-1] <= L[i] < L[j]`;
3. inserisce `L[i]` il minimo nella posizione j, traslando gli elementi nella sequenza per fagli spazio.

Assumendo la proprietà (1) è possibile fare i passi (2) e (3) contemporaneamente. Infatti l'elemento da inserire può essere confrontato con gli elementi `L[i]`, `L[i-1]`, ... `L[0]` (notate che gli elementi vengono scorsi da destra a sinistra). Qualora l'elemento da inserire dovesse essere più piccolo di quello già nella lista, quest'ultimo dovrà essere spostato a destra di una posizione.

Prima di vedere come funziona l'intero algoritmo, vediamo un esempio di questa dinamica di inserimento. Consideriamo il caso $i = 4$, nella sequenza già parzialmente ordinata

$$\langle 2.1 \quad 5.5 \quad 6.3 \quad 9.7 \quad 3.2 \quad 5.2 \quad 7.8 \rangle \, .$$

Poiché stiamo osservando il passo $i = 4$, gli elementi dalla posizione 0 alla posizione 3 sono ordinati e l'elemento alla posizione $i = 3$ è quello che deve essere inserito nella posizione corretta. La dinamica è la seguente: l'elemento 3.2 viene "portato fuori" dalla sequenza, così che quella posizione possa essere sovrascritta.

$$\langle 2.1 \quad 5.5 \quad 6.3 \quad 9.7 \quad \sqcup \quad 5.2 \quad 7.8 \rangle \qquad \boxed{3.2}$$

Poi 3.2 viene confrontato con 9.7, che viene spostato a destra in quanto più grande.

$$\langle 2.1 \quad 5.5 \quad 6.3 \quad \sqcup \quad \boxed{9.7} \quad 5.2 \quad 7.8 \rangle \qquad \boxed{3.2}$$

Al passo seguente viene confrontato con 6.3 anche questo più grande di 3.2.

$$\langle 2.1 \quad 5.5 \quad \sqcup \quad \boxed{6.3} \quad 9.7 \quad 5.2 \quad 7.8 \rangle \qquad \boxed{3.2}$$

Di nuovo il confronto è con 5.5 più grande di 3.2.

$$\langle 2.1 \quad \sqcup \quad \boxed{5.5} \quad 6.3 \quad 9.7 \quad 5.2 \quad 7.8 \rangle \qquad \boxed{3.2}$$

A questo punto, visto che $3.2 \geq 2.1$, l'elemento 3.2 può essere posizionato nella cella libera.

$$\langle 2.1 \quad \boxed{3.2} \quad 5.5 \quad 6.3 \quad 9.7 \quad 5.2 \quad 7.8 \rangle \, .$$

Vediamo subito il codice che esegue questa operazione: una funzione prende come input la sequenza, e il valore di i. Naturalmente la funzione *assume* che la sequenza sia correttamente ordinata dalla posizione 0 alla posizione $i - 1$.

```
def insertion_step(L, i):
    """Esegue l'i-esimo passo di insertion sort

    Assume che L[0], L[1], ... L[i-1] siano ordinati e che i>0
    """
    x   = L[i]    # salvo il valore da inserire
    pos = i       # posizione di inserimento

    while pos > 0 and L[pos-1] > x:
        L[pos] = L[pos-1]   #sposto a destra L[pos-1]
        pos = pos -1

    L[pos] = x
```

Vediamo alcuni esempi di esecuzione del passo di inserimento. Quest'esecuzione è ottenuta utilizzando una versione modificata di `insertion_step` che contiene delle stampe aggiuntive per far vedere l'esecuzione.

```
sequenza=[2.1, 5.5,6.3, 9.7, 3.2, 5.2, 7.8]
insertion_step(sequenza,4)
```

```
Passo 0     i=4, pos=4 [2.1, 5.5, 6.3, 9.7, 3.2, 5.2, 7.8]
Passo 1     i=4, pos=3 [2.1, 5.5, 6.3, 9.7, 9.7, 5.2, 7.8]
Passo 2     i=4, pos=2 [2.1, 5.5, 6.3, 6.3, 9.7, 5.2, 7.8]
Passo 3     i=4, pos=1 [2.1, 5.5, 5.5, 6.3, 9.7, 5.2, 7.8]
Inserimento i=4, pos=1 [2.1, 3.2, 5.5, 6.3, 9.7, 5.2, 7.8]
```

```
sequenza=["cane", "gatto", "orso", "aquila"]
insertion_step(sequenza,3)
```

```
Passo 0     i=3, pos=3 ['cane', 'gatto', 'orso', 'aquila']
Passo 1     i=3, pos=2 ['cane', 'gatto', 'orso', 'orso']
Passo 2     i=3, pos=1 ['cane', 'gatto', 'gatto', 'orso']
Passo 3     i=3, pos=0 ['cane', 'cane', 'gatto', 'orso']
Inserimento i=3, pos=0 ['aquila', 'cane', 'gatto', 'orso']
```

Osserviamo che ci sono casi in cui l'algoritmo di inserimento esegue più passi, ed altri in cui ne esegue di meno. Ad esempio se l'elemento da inserire è più piccolo di tutti quelli precedenti, allora è necessario scorrere la sequenza all'indietro fino alla posizione 0. Se invece l'elemento è più grande di tutti i precedenti, non sarà necessario fare nessuno spostamento.

```
insertion_step([2,3,4,5,1],4)
```

```
Passo 0     i=4, pos=4 [2, 3, 4, 5, 1]
Passo 1     i=4, pos=3 [2, 3, 4, 5, 5]
Passo 2     i=4, pos=2 [2, 3, 4, 4, 5]
Passo 3     i=4, pos=1 [2, 3, 3, 4, 5]
Passo 4     i=4, pos=0 [2, 2, 3, 4, 5]
Inserimento i=4, pos=0 [1, 2, 3, 4, 5]
```

```
insertion_step([1,2,3,4,5],4)
```

```
Passo 0     i=4, pos=4 [1, 2, 3, 4, 5]
Inserimento i=4, pos=4 [1, 2, 3, 4, 5]
```

Ora vediamo l'algoritmo completo, che risulta molto semplice: all'inizio la sotto-sequenza costituita dal solo elemento alla posizione 0 è ordinata. Per i che va da 1 a $n-1$ ad ogni passo inseriamo l'elemento alla posizione i, ottenendo che la sotto-sequenza di elementi dalla posizione 0 alla posizione i sono ordinati.

```
def insertion_sort(L):
    """Ordina la sequenza seq utilizzando insertion sort"""
```

```
    for i in range(1,len(L)):
        insertion_step(L,i)
```

Vediamo un esempio di come si comporta l'algoritmo completo. Stavolta non mostriamo tutti i passaggi di ogni inserimento, ma solo la sequenza risultante dopo ognuno di essi.

```
dati = [21,-4,3,6,1,-5,19]
insertion_sort(dati)
```

```
Inizio   : [21, -4, 3, 6, 1, -5, 19]
Passo i=1: [-4, 21, 3, 6, 1, -5, 19]
Passo i=2: [-4, 3, 21, 6, 1, -5, 19]
Passo i=3: [-4, 3, 6, 21, 1, -5, 19]
Passo i=4: [-4, 1, 3, 6, 21, -5, 19]
Passo i=5: [-5, -4, 1, 3, 6, 21, 19]
Passo i=6: [-5, -4, 1, 3, 6, 19, 21]
```

Vediamo un esempio con una sequenza invertita

```
dati = [10,9,8,7,6,5,4,3,2,1]
insertion_sort(dati)
```

```
Inizio   : [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Passo i=1: [9, 10, 8, 7, 6, 5, 4, 3, 2, 1]
Passo i=2: [8, 9, 10, 7, 6, 5, 4, 3, 2, 1]
Passo i=3: [7, 8, 9, 10, 6, 5, 4, 3, 2, 1]
Passo i=4: [6, 7, 8, 9, 10, 5, 4, 3, 2, 1]
Passo i=5: [5, 6, 7, 8, 9, 10, 4, 3, 2, 1]
Passo i=6: [4, 5, 6, 7, 8, 9, 10, 3, 2, 1]
Passo i=7: [3, 4, 5, 6, 7, 8, 9, 10, 2, 1]
Passo i=8: [2, 3, 4, 5, 6, 7, 8, 9, 10, 1]
Passo i=9: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

2.3 La complessità di insertion sort.

Per ogni ciclo i, con i da 1 a $n-1$, l'algoritmo calcola la posizione `pos` dove fare l'inserimento e poi lo esegue. Per farlo la funzione `insertion_point` esegue circa `i-pos` spostamenti, più un inserimento. Alla peggio `pos` è uguale a 0, mentre alla meglio è uguale a i.

Quando misuriamo la complessità di un algoritmo siamo interessati al **caso peggiore**, ovvero a quegli input per cui il numero di operazioni eseguite sia il massimo possibile. Qui il caso peggiore per `insertion_point` è che si debbano fare i spostamenti, al passo i-esimo. Ma è possibile che esistano degli input per cui *tutte* le chiamate a `insertion_step` risultino così costose?

Abbiamo visto , ad esempio nel caso di una sequenza ordinata in maniera totalmente opposta a quella desiderata, che il caso peggiore può avverarsi ad ogni passaggio. Ad ogni iterazione l'elemento nuovo va inserito all'inizio della sequenza. Quindi in questo caso al passo i-esimo si fanno sempre circa i operazioni. Il totale quindi è

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \approx n^2 \tag{2.1}$$

Esercizio: abbiamo discusso il caso di input peggiore. Per quali input il numero di operazioni fatte da insertion sort è minimo?

2.4 Vale la pena ordinare prima di fare ricerche?

Abbiamo visto che R ricerche costano nR se si utilizza la ricerca sequenziale. E se vogliamo utilizzare insertion sort per ordinare la sequenza e usare in seguito la ricerca binaria?

Il costo nel caso peggiore è circa $n^2 + R \log n$, che è comparabile con nR solo se si effettuano almeno $R \approx n$ ricerche. Fortunatamente esistono algoritmi di ordinamento più veloci, alcuni dei quali verranno discussi nel corso. Usando uno di questi algoritmi ordinare i dati potrebbe essere vantaggioso anche per un numero di ricerche più basso.

Capitolo 3

Notazione asintotica

Nota: il contenuto di questa parte degli appunti può essere approfondito nel Capitolo 3 del libro di testo *Introduzione agli Algoritmi e strutture dati* di Cormen, Leiserson, Rivest e Stein.

Quando si contano il numero di operazioni in un algoritmo, non è necessario essere estremamente precisi. A che serve distinguere tra $10n$ operazioni e $2n$ operazioni se in un linguaggio di programmazione le $10n$ operazioni sono più veloci delle $2n$ operazioni implementate in un altro linguaggio? Al contrario é molto importante distinguere, ad esempio, $\frac{1}{100}n^2$ operazioni da $20\sqrt{n}$ operazioni. Anche una macchina più veloce paga un prezzo alto se esegue un algoritmo da $\frac{1}{100}n^2$ operazioni invece che uno da $20\sqrt{n}$.

Tuttavia un algoritmo asintoticamente più veloce spesso paga un prezzo più alto in fase di inizializzazione (per esempio perché deve sistemare i dati in una certa maniera). Dunque ha senso fare un confronto solo per lunghezze di input grandi abbastanza.

Per discutere in questi termini utilizziamo la notazione asintotica che è utile per indicare quanto cresce il numero di operazioni di un algoritmo, al crescere della lunghezza dell'input.

Definizione: date $f : \mathbb{N} \to \mathbb{R}$ e $g : \mathbb{N} \to \mathbb{R}$ diciamo che

$$f \in O(g)$$

se esistono $c > 0$ e n_0 tali che $0 \leq f(n) \leq cg(n)$ per ogni $n > n_0$.

- la costante c serve a ignorare il costo contingente delle singole operazioni elementari;

- n_0 serve a ignorare i valori piccoli di n.

Ad esempio $\frac{n^2}{100} + 3n - 10$ è in $O(n^2)$, non è in $O(n^\ell)$ per nessun $\ell < 2$ ed è in $O(n^\ell)$ per $\ell > 2$.

Complessità di un algoritmo: dato un algoritmo A consideriamo il numero di operazioni $t_A(x)$ eseguite da A sull'input $x \in \{0,1\}^*$. Per ogni taglia di input n possiamo definire il **costo nel caso peggiore** come

$$c_A(n) := \max_{x \in \{0,1\}^n} t_A(x) \ .$$

Diciamo che la complessità di un algoritmo A, ovvero il suo *tempo di esecuzione nel caso peggiore*, è $O(g)$ quando $c_A \in O(g)$. Per esempio abbiamo visto due algoritmi di ricerca (ricerca lineare e binaria) che hanno entrambi complessità $O(n)$, ed il secondo ha anche complessità $O(\log n)$.

La notazione $O(g)$ quindi serve a dare un limite approssimativo superiore al numero di operazioni che l'algoritmo esegue.

Esercizio: osservare che per ogni $1 < a < b$,

- $\log_a(n) \in O(\log_b(n))$; e
- $\log_b(n) \in O(\log_a(n))$.

Quindi specificare la base non serve.

Definizione (Ω e Θ): date $f : \mathbb{N} \to \mathbb{R}$ e $g : \mathbb{N} \to \mathbb{R}$ diciamo che

$$f \in \Omega(g)$$

se esistono $c > 0$ e n_0 tali che $f(n) \geq cg(n) \geq 0$ per ogni $n > n_0$. Diciamo anche che

$$f \in \Theta(g)$$

se $f \in \Omega(g)$ e $f \in O(g)$ simultaneamente.

Complessità di un algoritmo (II): dato un algoritmo A consideriamo ancora il **costo nel caso peggiore** di A su input di taglia n come

$$c_A(n) := \max_{x \in \{0,1\}^n} t_A(x) \ .$$

Diciamo che la complessità di un algoritmo A, ovvero il suo *tempo di esecuzione nel caso peggiore*, è $\Omega(g)$ quando $c_A \in \Omega(g)$, e che ha complessità $\Theta(g)$ quando $c_A \in \Theta(g)$.

Per esempio abbiamo visto due algoritmi di ricerca (ricerca lineare e binaria) che hanno entrambi complessità $\Omega(\log n)$, ed il primo anche complessità $\Omega(n)$. Volendo essere più specifici possiamo dire che la ricerca lineare ha complessità $\Theta(n)$ e quella binaria ha complessità $\Theta(\log n)$.

Fate attenzione a questa differenza:

- se A ha complessità $O(g)$ allora A gestisce tutti gli input con al massimo g operazioni circa, asintoticamente;
- se A ha complessità $\Omega(g)$ allora per ogni n abbastanza grande, esiste un input di dimensione n per cui A richiede non meno di $g(n)$ operazioni circa.

3.1 Note su come misuriamo la lunghezza dell'input

Gli ordini di crescita sono il linguaggio che utilizziamo quando vogliamo parlare dell'uso di risorse computazionali, indicando come le risorse crescono in base alla dimensione dell'input.

La lunghezza dell'input n consiste nel numero di bit necessari a codificarlo. Tuttavia per problemi più specifici è utile, e a volte più comodo, indicare con n la dimensione "logica" dell'input. Ad esempio:

- trovare il massimo in una sequenza. n sarà la quantità di numeri nella sequenza;
- trovare una comunità in una rete sociale con x nodi, e y connessioni. La lunghezza dell'input può essere sia x che y a seconda dei casi;
- moltiplicare due matrici quadrate $n \times n$. La dimensione dell'input è il numero di righe (o colonne) delle due matrici.

Naturalmente misurare così la complessità ha senso solo se consideriamo, ad esempio, dati numerici di grandezza limitata. Ovvero numeri di grandezza tale che le operazioni su di essi possono essere considerate elementari senza che l'analisi dell'algoritmo ne sia compromessa.

3.2 Esercitarsi sulla notazione asintotica

Ci sono alcuni fatti ovvi riguardo la notazione asintotica, che seguono immediatamente le definizioni. Dimostrate che

1. per ogni $0 < a < b$, $n^a \in O(n^b)$, $n^b \notin O(n^a)$;
2. per ogni $0 < a < b$, $n^b \in \Omega(n^a)$, $n^a \notin \Omega(n^b)$;
3. se $f \in \Omega(h)$ e g una funzione positiva, allora $f \cdot g \in \Omega(h \cdot g)$;
4. se $f \in O(h)$ e g una funzione positiva, allora $f \cdot g \in O(h \cdot g)$;
5. che $(n + a)^b \in \Theta(n^b)$.
6. preso un qualunque polinomio $p = \sum_{i=0}^{d} \alpha_i x^i$ con $\alpha_d > 0$, si ha che $p(n) \in \Theta(n^d)$;
7. che $n! \in \Omega(2^n)$ ma che $n! \notin O(2^n)$;
8. che $\binom{n}{d} \in O(n^d)$;
9. che per ogni $a, b > 0$, $(\log(n))^a \in O(n^b)$;
10. che per ogni $a, b > 0$, $n^b \notin O((\log(n))^a)$;

Capitolo 4

Ordinare i dati con Bubblesort

Nota: il contenuto di questa parte degli appunti può essere approfondito nel Problema 2-2 a pagina 34 del libro di testo *Introduzione agli Algoritmi e strutture dati* di Cormen, Leiserson, Rivest e Stein.

Il Bubblesort è un altro algoritmo di ordinamento. Il suo comportamento è abbastanza differente da quello dell'insertion sort. Vedete dunque che per risolvere lo stesso problema, ovvero ordinare una lista, esistono più algoritmi anche profondamente diversi.

Per introdurre il Bubblesort partiamo da una sua versione molto semplificata, descritta nella prossima sezione.

4.1 Bubble sort semplificato

La versione semplificata del bubblesort ha uno schema che a prima vista è simile a quello dell'insertion sort, infatti la lista ordinata viene costruita passo passo, mettendo al suo posto definitivo un elemento alla volta. Partendo da una lista di n elementi facciamo i seguenti $n - 1$ passi

1. Effettuando degli scambi di elementi nella lista secondo uno schema, che vedremo successivamente, mettiamo il più grande elemento della lista nella posizione $n - 1$. Questo elemento non verrà più toccato o spostato.
2. Mettiamo il secondo più grande elemento nella posizione $n - 2$. Poiché al passo precedente abbiamo messo l'elemento più grande alla

posizione $n-1$, vuol dire che il secondo elemento più grande non è altri che l'elemento più grande tra quelli nelle posizioni $0, 1, \ldots, n-2$.

3. Mettiamo il terzo elemento più grande nella posizione $n-3$. Poiché alle posizioni $n-2$ e $n-1$ ci sono i due elementi più grandi, il terzo elemento più grande non è altri che l'elemento più grande tra quelli nelle posizioni $0, 1, \ldots, n-3$.

Continuando su questa falsariga, al passo i-esimo mettiamo l'i-esimo elemento più grande alla posizione $n-i$. Arrivati al passo $n-1$, ci troviamo con gli elementi nelle posizioni 0 e 1, e dobbiamo semplicemente mettere il più grande in posizione 1 e lasciare il più piccolo in posizione 0.

Da questa descrizione del bubblesort "semplificato" mancano molti dettagli: come sono effettuati gli scambi che portano l'elemento più grande alla fine della lista?

Da questa descrizione parziale osserviamo subito che immediatamente prima di eseguire il passo i gli elementi nelle posizioni $n-i+1, \ldots, n-1$ sono ordinati e nella loro posizione **definitiva**.

Una cosa da chiarire è come venga trovato l'elemento più grande della lista e messo all'ultima posizione. E più in generale, per il passo i-esimo, come venga trovato l'elemento più grande tra le posizioni $0, \ldots, n-i$ e messo in posizione $n-i$.

Vediamo che il seguente algoritmo, applicato alla lista `seq`, la scorre e scambia ogni coppia di elementi adiacenti, se non sono tra di loro in ordine crescente.

```
for j in range(0,len(seq)-1):
    if seq[j] > seq[j+1]:
        seq[j], seq[j+1] = seq[j+1], seq[j]
```

Vediamo un esempio di esecuzione di questa procedura sulla sequenza `[2,4,1,3,6,5,2]`

```
Initial: [2, 4, 1, 3, 6, 5, 2]
0 vs 1 : [2, 4, 1, 3, 6, 5, 2]
1 vs 2 : [2, 1, 4, 3, 6, 5, 2]
2 vs 3 : [2, 1, 3, 4, 6, 5, 2]
3 vs 4 : [2, 1, 3, 4, 6, 5, 2]
4 vs 5 : [2, 1, 3, 4, 5, 6, 2]
5 vs 6 : [2, 1, 3, 4, 5, 2, 6]
```

Notate come il massimo sia messo sul fondo della lista, perché ad un certo punto l'elemento massimo risulterà sempre maggiore di quelli con cui viene confrontato e la sua posizione verrà sempre fatta avanzare. Notate

anche come gli altri elementi della lista si trovano nelle posizioni $0, \ldots, n-2$ e quindi la procedura può essere ripetuta su questa parte di lista.

Al passo i-esimo la procedura sarà la stessa, ma limitata alle posizioni da 0 a $n - i$.

```
for j in range(0,len(seq)-i):
    if seq[j] > seq[j+1]:
        seq[j], seq[j+1] = seq[j+1], seq[j]
```

L'algoritmo finale sarà quindi ottenuto ripetendo la procedura: al passo i si opera sulla sottolista dalla posizione 0 alla posizione $n - i$. L'algoritmo completo è il seguente.

```
def stupid_bubblesort(seq):
    for i in range(1,len(seq)):
        for j in range(0,len(seq)-i):
            if seq[j] > seq[j+1]:
                seq[j], seq[j+1] = seq[j+1], seq[j]
```

Vediamo un esempio di esecuzione dell'algoritmo, mostrata passo passo.

```
stupid_bubblesort([5,-4,3,6,19,1,-5])
```

```
Start  : [5, -4, 3, 6, 19, 1, -5] |]
Step 1 : [-4, 3, 5, 6, 1, -5] | [19]
Step 2 : [-4, 3, 5, 1, -5] | [6, 19]
Step 3 : [-4, 3, 1, -5] | [5, 6, 19]
Step 4 : [-4, 1, -5] | [3, 5, 6, 19]
Step 5 : [-4, -5] | [1, 3, 5, 6, 19]
Step 6 : [-5] | [-4, 1, 3, 5, 6, 19]
```

4.2 Miglioriamo l'algoritmo

Se eseguiamo la sequenza di scambi iniziali sulla lista `[3, 2, 7, 1, 8, 9]` ,vediamo che l'ultimo scambio effettuato è tra la posizione 2 e 3, ovvero quando la lista passa da `[2, 3, 7, 1, 8, 9]` a `[2, 3, 1, 7, 8, 9]`

```
Initial: [3, 2, 7, 1, 8, 9]
0 vs 1 : [2, 3, 7, 1, 8, 9]
1 vs 2 : [2, 3, 7, 1, 8, 9]
2 vs 3 : [2, 3, 1, 7, 8, 9]
3 vs 4 : [2, 3, 1, 7, 8, 9]
4 vs 5 : [2, 3, 1, 7, 8, 9]
```

Nessuno scambio viene effettuato dalla posizione 3 in poi, e questo ci garantisce che da quella posizione gli elementi siano già ordinati. In effetti in

ognuna delle fasi in cui viene effettuata una sequenza di scambi possiamo verificare che valgono alcune proprietà.

- successivamente al momento in cui viene considerata la possibilità di fare lo scambio tra posizione i e $i+1$, abbiamo che l'elemento alla posizione $i+1$ è maggiore di tutti i precedenti, indipendentemente dal fatto che lo scambio sia avvenuto.
- nessuna inversione dopo la posizione i vuol dire che tutti gli elementi dalla posizione $i+1$ in poi sono ordinati.

Nella versione semplificata del bubblesort stiamo usando solo la prima delle due proprietà, perché usiamo la serie di scambi per portare in fondo alla sequenza l'elemento più grande.

Per usare la seconda proprietà possiamo memorizzare la posizione dell'ultimo scambio effettuato nel ciclo interno. Se l'ultimo scambio avviene tra le posizioni j e $j+1$ gli elementi in posizione maggiore di j sono ordinati tra loro, e sono tutti maggiori o uguali degli elementi in posizione minore o uguale a j. Pertanto gli elementi in posizione maggiore di j sono nella loro posizione definitiva.

```
last_swap=0
for j in range(0,end):     # 'end' è la lunghezza della parte da ordinare
    if seq[j] > seq[j+1]:
        last_swap = j
        seq[j], seq[j+1] = seq[j+1],seq[j]
```

La variabile `last_swap` ci indica che alla fine del ciclo interno la parte di sequenza che va ancora ordinata va dalla posizione 0 alla posizione `last_swap` inclusa. Dalla descrizione del bubblesort "stupido" sappiamo che all fine di una esecuzione del ciclo interno il valore di `last_swap` è almeno di un'unità più piccolo del valore ottenuto alla fine dell'esecuzione precedente. Pertanto il seguente algoritmo, che è la versione finale di bubblesort, termina sempre.

```
def bubblesort(seq):
    end=len(seq)-1   # devo ordinare tutto
    while end>0:
        last_swap = 0
        for j in range(0,end):
            if seq[j] > seq[j+1]:
                last_swap = j
                seq[j], seq[j+1] = seq[j+1],seq[j]
        end = last_swap    # ordina solo fino allo scambio
```

```
bubblesort([3,2,7,1,8,9])
```

```
Start  : [3, 2, 7, 1, 8, 9] |
Step 1 : [2, 3, 1] | [7, 8, 9]
Step 2 : [2, 1] | [3, 7, 8, 9]
Step 3 : [1] | [2, 3, 7, 8, 9]
```

Questa versione fa meno passi: deve ordinare solo la parte di sequenza che precede l'ultimo scambio.

Bubblesort su sequenze ordinate Il vantaggio enorme che ha bubblesort rispetto alla sua versione semplificata è quello che se la lista è molto vicina ad essere ordinanta, o addirittura ordinata, il numero di passi nel ciclo esterno si riduce moltissimo.

```
stupid_bubblesort([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])                    1
```

```
Start  : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] |]
Step 1 : [1, 2, 3, 4, 5, 6, 7, 8, 9] | [10]
Step 2 : [1, 2, 3, 4, 5, 6, 7, 8] | [9, 10]
Step 3 : [1, 2, 3, 4, 5, 6, 7] | [8, 9, 10]
Step 4 : [1, 2, 3, 4, 5, 6] | [7, 8, 9, 10]
Step 5 : [1, 2, 3, 4, 5] | [6, 7, 8, 9, 10]
Step 6 : [1, 2, 3, 4] | [5, 6, 7, 8, 9, 10]
Step 7 : [1, 2, 3] | [4, 5, 6, 7, 8, 9, 10]
Step 8 : [1, 2] | [3, 4, 5, 6, 7, 8, 9, 10]
Step 9 : [1] | [2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
bubblesort([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])                           1
```

```
Start  : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] |
Step 1 : [1] | [2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Bubblesort su sequenze invertite Su sequenza totalmente invertite vediamo che invece il bubblesort si comporta esattamente come la sua versione semplificata, poiché nel ciclo interno viene sempre effettuato uno scambio nell'ultima posizione.

```
stupid_bubblesort([8, 7, 6, 5, 4, 3, 2, 1])                           1
```

```
Start  : [8, 7, 6, 5, 4, 3, 2, 1] |]
Step 1 : [7, 6, 5, 4, 3, 2, 1] | [8]
Step 2 : [6, 5, 4, 3, 2, 1] | [7, 8]
Step 3 : [5, 4, 3, 2, 1] | [6, 7, 8]
Step 4 : [4, 3, 2, 1] | [5, 6, 7, 8]
Step 5 : [3, 2, 1] | [4, 5, 6, 7, 8]
Step 6 : [2, 1] | [3, 4, 5, 6, 7, 8]
Step 7 : [1] | [2, 3, 4, 5, 6, 7, 8]
```

```
bubblesort([8, 7, 6, 5, 4, 3, 2, 1])                                  1
```

```
Start  : [8, 7, 6, 5, 4, 3, 2, 1] |
Step 1 : [7, 6, 5, 4, 3, 2, 1] | [8]
Step 2 : [6, 5, 4, 3, 2, 1] | [7, 8]
Step 3 : [5, 4, 3, 2, 1] | [6, 7, 8]
Step 4 : [4, 3, 2, 1] | [5, 6, 7, 8]
Step 5 : [3, 2, 1] | [4, 5, 6, 7, 8]
Step 6 : [2, 1] | [3, 4, 5, 6, 7, 8]
Step 7 : [1] | [2, 3, 4, 5, 6, 7, 8]
```

And then, you can use it this way:
Esercizio 2. Come abbiamo visto ci sono casi in cui il Bubblesort impiega $O(n)$ operazioni ma anche casi in cui non si comporta meglio della versione semplificata. Provate ad eseguire i tre algoritmi

- Insertion sort
- Bubblesort semplificato
- Bubblesort

su delle liste di elementi generate casualmente.

Capitolo 5

Ordinamenti per confronti

Nota: il contenuto di questa parte degli appunti può essere approfondito nel Paragrafo 8.1 del libro di testo *Introduzione agli Algoritmi e strutture dati* di Cormen, Leiserson, Rivest e Stein.

L'Insertion sort e il Bubblesort sono due algoritmi di ordinamento cosiddetti *per confronti*. Se osservate bene il codice vedrete che entrambi gli algoritmi di fatto non leggono gli elementi contenuti nella lista da ordinare, ma testano solamente espressioni di confronto tra elementi memorizzati. Per essere più concreti possiamo dire che tutte le decisioni prese dall'algoritmo di basano sul confronto $\leq$ tra due elementi della sequenza. Nel senso che

- le entrate e uscite dai cicli `while` e `for`
- la strada presa negli `if/else`
- gli spostamenti di valori nella lista
- ...

non dipendono dai valori stessi, ma solo dall'esito dei confronti. Gli algoritmi di ordinamento per confronto hanno un enorme vantaggio: possono essere usati su qualunque tipo di dati, ammesso che questi dati abbiamo una nozione appropriata di confrontabilità.

```
bubblesort(["cassetto","armadio","scrivania","poltrona"])                    1
```

```
['armadio', 'cassetto', 'poltrona', 'scrivania']
```

Il principale risultato di questo capitolo è la seguente limitazione degli algoritmi di ordinamento per confronti.

Teorema 3. *Un algoritmo di ordinamento per confronti* ***necessita*** *di* $\Omega(n \log n)$ *operazioni per ordinare una lista di n elementi.*

Per molti di voi questo è il primo *risultato di impossibilità* che riguarda la complessità computazionale di un problema. Che vuol dire che *nessun* algoritmo può fare meglio di così? Ci sono infiniti algoritmi possibili. Come si può dire che nessuno di essi superi questa limitazione inferiore? Per dimostrare che esista un algoritmo con determinate prestazioni è sufficiente esibirne uno che abbia le caratteristiche richieste (naturalmente può essere difficile scoprire/inventare questo algoritmo). Ma come dimostrare l'opposto?

Prima di dimostrare il teorema, vediamo a scopo di esempio un risultato analogo ma con una dimostrazione più semplice. Sappiamo che in una sequenza non ordinata possiamo effettuare la ricerca in $O(n)$ operazioni, ma non sappiamo come fare meglio di così. In effeti possiamo dimostrare come questo limite sia insuperabile.

> La ricerca in una sequenza non ordinata richiede $\Omega(n)$ operazioni.

Dimostrazione. Qualunque sia l'algoritmo, sappiamo che non deve saltare nessuna posizione nella sequenza, poiché l'elemento cercato potrebbe essere lì. Più in particolare consideriamo il comportamento di un algoritmo di ricerca dell'elemento 1 nella sequenza di lunghezza n contentente solo 0.

0	0	0	0	0	0	0	0

L'algoritmo riporterà il fatto che l'elemento 1 non è nella sequenza. Se questo algoritmo ha eseguito meno di $o(n)$ operazioni allora esiste una posizione $0 \leq i < n$ della sequenza che non è stata visitata. Consideriamo una nuova sequenza identica alla precedente dove alla posizione i viene piazzato

0	0	0	**1**	0	0	0	0

L'algoritmo si comporterà esattamente nella stessa maniera di prima, perché le due sequenze sono identiche in tutte le posizioni visitate. Ma in questo caso l'algoritmo non dà la risposta corretta.

Concludendo: dal fatto che questo algoritmo utilizza impiega $o(n)$ operazioni, abbiamo dedotto che non è un algoritmo di ricerca corretto. □

Esercizio 4. La dimostrazione precedente vale anche per algoritmi di ricerca su sequenze ordinate? Se sì, perché la ricerca binaria impiega $O(\log n)$ passi? Se no, perché la dimostrazione precedente non vale in quel caso?

5.1 Ordinamenti e Permutazioni

Come si comporta un **qualunque** algoritmo di ordinamento per confronti su una di queste sequenza di tre elementi $\langle a_0, a_1, a_2 \rangle$? Confronta coppie di elementi e in qualche modo decide come disporre gli elementi in output. Un possibile esempio (che non è unico!) è il seguente.

- Se $a_0 \leq a_1$
  - se $a_1 \leq a_2$ **output** $\langle a_0, a_1, a_2 \rangle$
  - altrimenti
    - se $a_0 \leq a_2$ **output** $\langle a_0, a_2, a_1 \rangle$
    - altrimenti **output** $\langle a_2, a_0, a_1 \rangle$
- altrimenti
  - se $a_0 \leq a_2$ **output** $\langle a_1, a_0, a_2 \rangle$
  - altrimenti
    - se $a_1 \leq a_2$ **output** $\langle a_1, a_2, a_0 \rangle$
    - altrimenti **output** $\langle a_2, a_1, a_0 \rangle$

Ci sono 6 modi possibili di disporre tre elementi, e per ognuno di essi possiamo scegliere dei valori a_0, a_1, a_2 per cui quel modo è l'unico corretto. Qundi è **necessario** che la sequenza di confronti distingua tutti e sei i modi, l'uno dall'altro.

Per proseguire questa discussione è necessario approfondire il concetto di *permutazione*. Una *permutazione di n elementi* è una funzione

$$\pi : \{0, 1, \ldots n-1\} \to \{0, 1, \ldots n-1\}$$

tale che $\pi(i) = \pi(j)$ se e solo se $i = j$. In pratica una permutazione "mischia" (i.e. assegna un ordine non necessariamente uguale a quello naturale) a tutti gli elementi da 0 a $n-1$. [1] Pertanto una permutazione è completamente descritta dalla sequenza

$$(\pi(0), \pi(1), \ldots, \pi(n-1))$$

Esempi di permutazioni per $n = 6$ sono

- $(1, 4, 3, 2, 0, 5)$
- $(5, 4, 3, 2, 1, 0)$
- $(0, 1, 2, 3, 4, 5)$

Le permutazioni su $\{0, \ldots n-1\}$ sono dotate di una naturale operazione di *composizione*. Date due permutazioni π e ρ, la permutazione ottenuta dalla loro composizione, denotata come $\pi\rho$, è la permutazione che ad ogni i tra 0 e $n-1$ associa $\rho(\pi(i))$. Essenzialmente è l'applicazione delle due permutazioni in sequenza.

Il numero di permutazioni di n elementi è uguale a $n!$ (che si legge "n fattoriale"), ovvero

$$n \times n-1 \times n-2 \times \cdots \times 2 \times 1$$

Ad esempio le permutazioni su 2 elementi sono 4, quelle su 3 elementi sono 6, quelle su 4 elementi sono 24, e così via.
Esercizio 5. Dimostrare la proposizione precedente.

Le permutazioni di n elementi hanno la struttura di un *gruppo algebrico*.

- **Identità**: esiste π per cui $\pi(i) = i$ per ogni i;
- **Associatività**: $\pi_1(\pi_2\pi_3) = (\pi_1\pi_2)\pi_3$
- **Inversa**: per ogni π esiste **un'unica** permutazione inversa, che denotiamo come π^{-1} per cui $\pi\pi^{-1}$ è la permutazione identità.

$$n = 6 \qquad \pi = (1, 4, 3, 5, 2, 0) \qquad \pi^{-1} = (5, 0, 4, 2, 1, 3)$$

[1] Naturalmente le permutazioni possono essere definite su qualunque insieme di elementi distinti che abbia un ordine ben definito, ad esempio le strighe. Tuttavia se consideriamo un insieme finito $X = \{x_0, x_1, x_2, \ldots, x_t\}$ dove $x_0 < x_1 < x_2 < \ldots x_t$ allora le permutazioni su questo insieme sono essenzialmente identiche alle permutazioni si $\{0, \ldots, t\}$.

Esercizio 6. Dimostrare le seguenti affermazioni sulle permutazioni:

- la permutazione identità e l'inversa di sé stessa;
- per qualunque π, l'inversa di π^{-1} è π.

Gli algoritmi di ordinamento sono intimamente connessi alle permutazioni. Quando l'input è una sequenza

$$\langle a_0, a_1, a_2, a_3, \ldots, a_{n-1} \rangle \tag{5.1}$$

di valori distinti, un algoritmo di ordinamento fatto calcola una permutazione π sugli indici $\{0, 1, \ldots, n-1\}$ tale che

$$\langle a_{\pi(0)}, a_{\pi(1)}, a_{\pi(2)}, a_{\pi(3)}, \ldots, a_{\pi(n-1)} \rangle \tag{5.2}$$

sia una sequenza crescente. Ad esempio da un input

$$\langle a_0, a_1, a_2, a_3, a_4 \rangle = \langle 32, -5, 7, 3, 12 \rangle$$

un algoritmo di ordinamento calcola la permutazione

$$(1, 3, 2, 4, 0)$$

che corrisponde all'output

$$\langle a_1, a_3, a_2, a_4, a_0 \rangle = \langle -5, 3, 7, 12, 32 \rangle$$

Esercizio 7. Modificate il codice python di uno degli algoritmi di ordinameno visti e fare in modo che oltre a ordinare la sequenza in input, restituisca in output la permutazione che, applicata all'input originale, la riordinerebbe. La modifica non deve cambiare la complessità dell'algoritmo né fagli perdele la proprietà di essere un algoritmo di ordinamento per confronti. (*Suggerimento*: create una lista `[0, 1, 2, ..., ]` ed effettuate su di essa gli stessi scambi che state effettuando nella sequenza in input)

Tornando all'esempio dell'ordinamento di tre elementi visto in precedenza, vediamo che la natura degli elementi stessi è inessenziale. Ad esempio l'algoritmo si comporterebbe in maniera completamente identica su ognuno di questi input:

- `[3, 2, 7]`
- `[ "casa", "abaco", "finestra" ]`

- `[-4, -10, 20 ]`

Infatti per tutti questi input l'esito di confronti è lo stesso. Da qui è evidente che per quanto riguarda un algoritmo di ordinamento per confronti, questi input sono essenzialmente identici all'input `[1,0,2]`. Questo naturalmente può essere generalizzato a sequenze di lunghezza arbitraria. Da qui vale la seguente osservazione:

> Se per due sequenze in input tutti i confronti danno lo stesso esito, un algoritmo di ordinamento per confronti produce la stessa permutazione π per entrambe.

5.2 Dimostrazione del limite $\Omega(n \log n)$

Ripetiamo il teorema da dimostrare, per comodità.

Teorema 8. *Per qualunque algoritmo di ordinamento per confronti, esiste una sequenza di n elementi per cui l'algoritmo esegue* $\Omega(n \log n)$ *confronti.*

Dimostrazione. Concentriamoci su input costituiti dagli elementi $\{0, 1, \ldots, n-1\}$ in ordine arbitrario. Osserviamo che per ogni permutazione π di $\{0, 1, \ldots, n-1\}$, esiste un modo di "mischiare" in valori in input così che l'output corretto dell'ordinamento non sia altri che permutazione π.

Ovviamente questo input non può che essere la sequenza

$$\pi^{-1}(0), \pi^{-1}(1), \pi^{-1}(2), \ldots, \pi^{-1}(n-1)$$

che può essere ordinata **solo** dalla permutazione π, per definizione. Ne concludiamo che l'algoritmo di ordinamento deve essere in grado di produrre $n!$ output differenti.

Sia h il massimo numero di confronti effettuati dall'algoritmo, qualunque sia il suo input, osserviamo che confronto ha al massimo due esiti, perciò la sequenza di confronti effettuati ha al massimo 2^h esiti distinti.

Essendo un algoritmo di ordinamento per confronti, la permutazione in output può dipendere **solo** dall'esito dei confronti pertanto

$$2^h \geq n!$$

e peralto $n! \geq (n/2)^{n/2}$. Questo vuol dire che

$$h \geq \frac{n}{2}(\log n - 1)$$

che è $\Omega(n \log n)$. □

5.3 Conclusione

Abbiamo visto che un ordinamento per confronti richiede $\Omega(n \log n)$ passi. Tuttavia sappiamo che sia Insertion sort che Bubblesort richiedono $\Theta(n^2)$ operazioni. È possibile raggiungere il limite?

Capitolo 6

Struttura a pila (stack)

La **pila** è una delle **strutture dati** più semplici. Una struttura dati è un modo ragionato di disporre le dati in memoria, aggiungendo riferimenti incrociati e informazioni di supporto, che rendano efficienti alcuni operazioni.

Quale disposizione dei dati va adottata? Questo dipende dal tipo di operazioni che si vogliono fare velocemente:

- inserimenti, cancellazioni, aggiornamenti;
- ricerca di un elemento (e.g. lista ordinata o dizionario);
- inserimenti in coda o nel mezzo dei dati.

Non ci si dovrebbe sorprendere scoprendo che per rendere veloci certe operazioni si debba sacrificare l'efficienza di altre. E che le strutture dati con le caratteristiche migliori sono anche più complesse da organizzare o richiedono molte informazioni ausiliare.

La pila è una struttura dati che permette

- inserimento di dati (**push**)
- estrazione del dato più recente inserito (**pop**)

ovvero una struttura dati LAST-IN FIRST-OUT (**LIFO**).

Prima di discutere perché è interessante utilizzare una struttura dati di questo tipo è meglio chiarire con esempi il suo funzionamento.

In generale la pila è inclusa nella libreria del linguaggio oppure va implementata. In `Python` è possibile utilizzare una lista come pila.

```
pila = []    # Una pila vuota

pila.append(10)        # operazione PUSH
print(pila)

pila.append("gatto")   # operazione PUSH
print(pila)

pila.pop()             # operazione POP
print(pila)

pila.append([1,2,3])   # operazione PUSH
print(pila)

pila.pop()             # operazione POP
print(pila)

pila.pop()             # operazione POP
print(pila)

pila.pop()             # POP da pila vuota = un errore
```

```
[10]
[10, 'gatto']
[10]
[10, [1, 2, 3]]
[10]
[]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/tmp/babel-ivgFEf/python-YIQXoB", line 21, in <module>
    pila.pop()             # POP da pila vuota = un errore
IndexError: pop from empty list
```

Il nome pila viene dal fatto che potete immaginare i dati come se fossero impilati uno sull'altro. Ogni dato nuovo va in cima alla pila. L'unico dato accessibile è quello in cima alla pila. In `Python` è possibile usare una lista, pensando che la cima della pila sia l'ultima posizione della lista stessa.

> Naturalmente se utilizzate altre operazioni di accesso ai dati diverse da `append` e `pop` allora non state più utilizzando la lista come se fosse una pila.

6.1 Usi della pila: contesti annidati

Fate conto che stiate leggendo un articolo di Economia, dove ad un certo punto viene utilizzato come argomento un'analisi statistica che utilizza un metodo che non conoscete. Allora potete andare a guardare un libro di statistica per chiarirvi le idee. Ma il formalismo matematico vi confonde e andate a guardare una pagina wikipedia di matematica. Una volta chiarito

il dubbio tornate al libro di statistica, e successivamente tornate all'articolo di economia.

Ogni volta che aprite un contesto nuovo in un certo senso **congelate** la situazione in cui eravate nel contesto precedente, e quando avete finito la riprendere da dove eravate rimasti.

6.2 Esempio: chiamate di funzioni

Quando richiamate una funzione da un vostro programma lo stato del programma **chiamante**, ovvero delle variabili, del punto del programma a cui siete arrivati, ecc. . . viene **salvato** e la funzione **chiamata** inizia la sua esecuzione. Al termine di questa funzione (che a sua volta può richiamare altre funzioni) lo stato del programma chiamante viene ripristinato ed il chiamante può procedere.

```
def primo_livello():
    i = 5
    print("Primo Livello 1: i={}".format(i))
    secondo_livello()
    print("Primo Livello 2: i={}".format(i))
    i = 4
    print("Primo Livello 3: i={}".format(i))
    secondo_livello()
    print("Primo Livello 4: i={}".format(i))

def secondo_livello():
    i=-3
    print("Secondo Livello: i={}".format(i))


i=10
primo_livello()
```

```
Primo Livello 1: i=5
Secondo Livello: i=-3
Primo Livello 2: i=5
Primo Livello 3: i=4
Secondo Livello: i=-3
Primo Livello 4: i=4
```

Per sommi capi questa procedura di salvataggio viene eseguita memorizzando le informazioni necessarie ad eseguire una funzione sullo su una struttura a pila (detta in inglese *stack*). Nel momento in cui una certa funzione viene chiamata, sulla pila viene costruito un *record di attivazione* (detto in inglese *frame*) che corrisponde a quella chiamata.

Supponiamo di avere a disposizione una funzione `func(p1,p2,p3)` e che alla riga 15 del file Python `main.py` venga eseguita l'istruzione

```
func(4,'gatto',[1,2])
```

Al momento dell'esecuzione di questa chiamata viene costruito un record di attivazione contenente, fra le altre cose

- `p1=4`
- `p2='gatto'`
- `p3=[1,2]`

e contenente anche l'informazione che la chiamata è stata effettuata alla riga 15 del file `main.py`. In questo modo al termine dell'esecuzione della chiamata `func(4,'gatto',[1,2])` il programma può riprendere l'esecuzione dall'istruzione successiva. Essenzialmente

- la chiamata di una funzione corrisponde ad un **push** di un record di attivazione nello stack;
- l'uscita da una funzione corrisponde ad un **pop** di un record di attivazione dallo stack.

6.3 Esempio: espressioni matematiche

Pensate ad un espressione matematica con le parentesi. All'apertura della parentesi l'espressione che si stava valutando in precedenza viene sospesa, fino a quando non si incontra una parentesi contraria che la bilanci. A quel punto la valutazione dell'espressione esterna viene ripresa.

Esempio: un piccolo programma che usa una pila per controllare che una serie di parentesi sia bilanciata.

```
def bilanciata(espressione):
    pila=[]
    for c in espressione:
        if c=='(':
            pila.append(c)
        elif c==')':
            if len(pila)==0:
                return False
            pila.pop()
    return len(pila)==0

print( bilanciata(" (()()) ") )
print( bilanciata(" )") )
```

```
print( bilanciata(" (") )
print( bilanciata(" (((()))(()()) ") )
print( bilanciata(" ( ) )(()()) ") )
```

```
True
False
False
True
False
```

6.4 Lo stack e le funzioni ricorsive

Una funzione può essere descritta in modo auto-referenziale, se la cosa è fatta con criterio. Naturalmente è necessario evitare che la descrizione sia circolare. Un esempio tipico è la funzione fattoriale sui numeri interi. Dato un intero $n \geq 0$ il *fattoriale* di n viene denotato come $n!$ e il suo valore è

$$n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdots 2 \cdot 1$$

e convenzionalmente $0!$ è definito uguale a 1. Una definizione più formale è la seguente.

Definizione 9. *Dato $n \geq 0$, la funzione* $\text{fact}(n)$ *è*

- 1, *se* $n = 0$
- $n \cdot \text{fact}(n-1)$ *se* $n > 0$.

La definizione non è circolare perché il fattoriale su 0 è ben definito, e se il fattoriale è ben definito per un numero allora è ben definito anche sul suo successore. Pertanto è definito su tutti i numeri interi non negativi.

Possiamo scrivere una funzione Python detta *ricorsiva* perché la funzione richiama sé stessa.

```
def fact(n):
    if n<0:
        raise ValueError("Fattoriale definito su non negativi")
    elif n==0:
        return 1
    else:
        return n * fact(n-1)
```

```
print( fact(10) )
print( fact(40) )
```

```
3628800
815915283247897734345611269596115894272000000000
```

Va osservato che la funzione ricorsiva `fact` utilizza molta memoria sullo stack in quanto, ad esempio, `fact(10)` richiama `fact(9)`, che richiama `fact(8)` e così via. Sostanzialmente la chiamata `fact(n)` induce l'inserimento sullo stack di n record di attivazione.

Questo ha un costo in termini di tempo e di memoria. In particolare è possibile che una ricorsione molto profonda (i.e., con tanti livelli) possa richiedere troppa memoria. Python non permette allo stack di crescere arbitrariamente, e se un certo limite viene superato allora solleva un `RecursionError`.

```
print( fact(1000) )
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/tmp/babel-vjEiik/python-CcC2d2", line 7, in fact
    return n * fact(n-1)
  File "/tmp/babel-vjEiik/python-CcC2d2", line 7, in fact
    return n * fact(n-1)
  File "/tmp/babel-vjEiik/python-CcC2d2", line 7, in fact
    return n * fact(n-1)
  [Previous line repeated 995 more times]
  File "/tmp/babel-vjEiik/python-CcC2d2", line 2, in fact
    if n<0:
RecursionError: maximum recursion depth exceeded in comparison
```

Spesso una funzione ricorsiva può essere riscritta in maniera *iterativa*, ovvero usando dei cicli invece della ricorsione. A volte è meno elegante, ma è sicuramente più efficiente.

```
def ifact(n):
    if n<0:
        raise ValueError("Fattoriale definito su non negativi")

    res = 1
    for i in range(1,n+1):
        res *= i
    return res
```

```
print(ifact(1000))  # Numero di 2568 cifre
```

```
402387260077093773543702433923003985719374864210714632543799910429938512398629020592044208
486969404800479988610197196058631666872994808558901323829669944590997424504087073759918823
                                  [...23 righe nascoste...]
560855903700624271243416909004153690105933983835777939410970027753472000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
```

Capitolo 7

Mergesort

Nota: il contenuto di questa parte degli appunti può essere approfondito nei Paragrafi 2.3 del libro di testo *Introduzione agli Algoritmi e strutture dati* di Cormen, Leiserson, Rivest e Stein.

La comprensione della struttura dati pila ci permette di capire più agevolmente algoritmi ricorsivi. Ora vediamo il **mergesort** un algoritmo ricorsivo di ordinamento per confronto e che opera in tempo $O(n \log n)$ e quindi è ottimale rispetto agli algoritmi di ordinamento per confronto.

7.1 Un approccio divide-et-impera

Un algoritmo può cercare di risolvere un problema

- dividendo l'input in parti
- risolvendo il problema su ogni parte
- combinando le soluzioni parziali

Naturalmente per risolvere le parti più piccole si riutilizza lo stesso metodo, e quindi si genera una gerarchia di applicazioni del metodo, annidate le une dentro le altre, su parti di input sempre più piccole, fino ad arrivare a parti così piccole che possono essere elaborate direttamente.

Lo schema divide-et-impera viene utilizzato spesso nella progettazione di algoritmi. Questo schema si presta molto ad una implementazione ricorsiva.

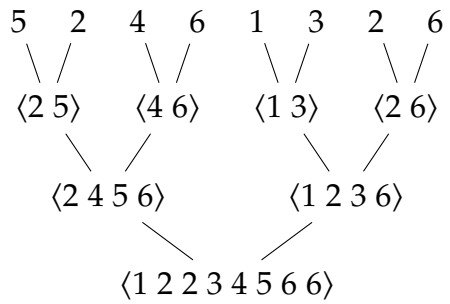
7.2 Schema principale del mergesort

1. dividere in due l'input
2. ordinare le due metà
3. fondere le due sequenze ordinate

Vediamo ad esempio come si comporta il mergesort sull'input

$$\langle 5\ 2\ 4\ 6\ 1\ 3\ 2\ 6\rangle$$

La sequenza ordinata viene ottenuta attraverso questa serie di fusioni.



7.3 Implementazione

Lo scheletro principale del mergesort è abbastanza semplice, e non è altro che la trasposizione in codice dello schema descritto in linguaggio naturale.

```
def mergesort(S,start=0,end=None):
    if end is None:
        end=len(S)-1
    if start>=end:
        return
    mid=(end+start)//2
    mergesort(S,start,mid)
    mergesort(S,mid+1,end)
    merge(S,start,mid,end)
```

7.4 Fusione dei segmenti ordinati

Dobbiamo fondere due sequenze ordinate, poste peraltro in due segmenti adiacenti della stessa lista. L'osservazione principale è che il minimo della sequenza fusa è il più piccolo tra i minimi delle due sequenze. Quindi si mantengono due indici che tengono conto degli elementi ancora da fondere e si fa progredire quello che indicizza l'elemento più piccolo. Quando una delle due sottosequenze è esaurita, allora si mette in coda la parte rimanente dell'altra. `merge` usa una **lista aggiuntiva temporanea** per fare la fusione. I dati sulla lista temporanea devono essere copiati sulla lista iniziale.

```
def merge(S,low,mid,high):
    a=low
    b=mid+1
    temp=[]
    # Parte 1 - Inserisci in testa il più piccolo
    while a<=mid and b<=high:
        if S[a]<=S[b]:
            temp.append(S[a])
            a=a+1
        else:
            temp.append(S[b])
            b=b+1
    # Parte 2 -  Esattamente UNA sequenza è esaurita. Va aggiunta l'altra
    if a<=mid:
        residuo = range(a,mid+1)
    else:
        residuo = range(b,high+1)
    for i in residuo:
        temp.append(S[i])
    # Parte 3 - Va tutto copiato su S[start:end+1]
    for i,value in enumerate(temp,start=low):
        S[i] = value
```

Questo conclude l'algoritmo

```
dati=[5,2,4,6,1,3,2,6]
mergesort(dati)
print(dati)
```

```
[1, 2, 2, 3, 4, 5, 6, 6]
```

7.5 Running time

Per cominciare osserviamo che nelle prime due parti di `merge` un elemento viene inserito nella lista temporanea ad ogni passo, e poi questo elemento non viene più considerato. La terza parte ricopia tutti gli elementi passando solo una volta su ognuno di essi. Pertanto è chiaro che `merge` di due segmenti adiacenti di lunghezza n_1 e n_2 impiega $\Theta(n_1 + n_2)$ operazioni.

Definiamo come $T(n)$ il numero di operazioni necessarie per ordinare una lista di n elementi con `mergesort`. Allora

$$T(n) = 2T(n/2) + \Theta(n) \tag{7.1}$$

quando $n > 1$, altrimenti $T(1) = \Theta(1)$ e dobbiamo risolvere **l'equazione di ricorrenza** rispetto a T. Prima di tutto per farlo fissiamo una costante $c > 0$ abbastanza grande per cui

$$T(n) \leq 2T(n/2) + cn \qquad T(1) \leq c\ . \tag{7.2}$$

Espandendo otteniamo

$$T(n) \leq 2T(n/2) + cn \leq 4T(n/4) + 2c(n/2) + cn = 4T(n/4) + 2cn \tag{7.3}$$

Si vede facilmente, ripetendo l'espansione, che

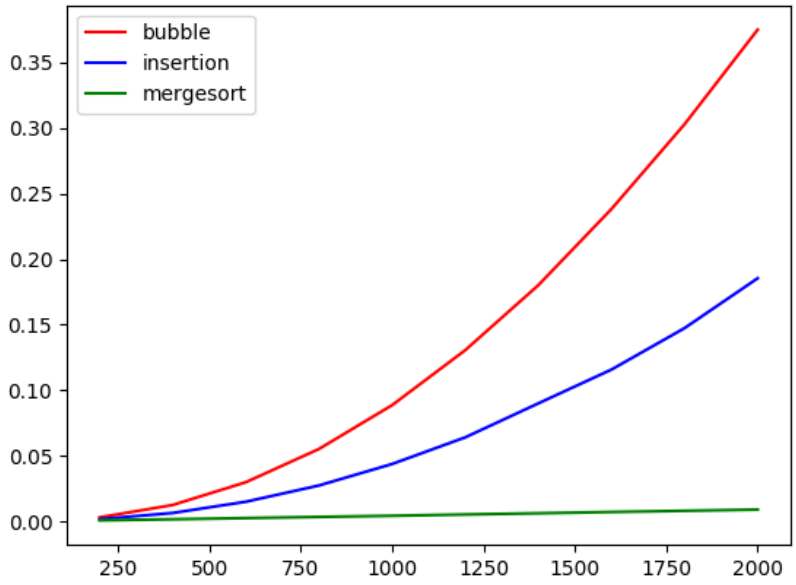
$$T(n) \leq 2^t T(n/2^t) + tcn \tag{7.4}$$

fino a che si arriva al passo t^* per cui $n/2^{t^*} \leq 1$, nel qual caso si ottiene $T(n) = c2^{t^*} + t^*cn \leq c(t^* + 1)n$.

Il più piccolo valore di t^* per cui $n/2^{t^*} \leq 1$ è $O(\log n)$, e quindi il running time totale è $O(n \log n)$.

Poichè il mergesort è un ordinamento per confronto il running time è $\Omega(n \log n)$, ed in ogni caso questo si può vedere anche direttamente dall'equazione di ricorrenza. Quindi il running time è in effetti $\Theta(n \log n)$.

7.6 Confronto sperimentale con insertion sort e bubblesort



7.7 Una piccola osservazione sulla memoria utilizzata

Mentre `bubblesort` e `insertionsort` non utilizzano molta memoria aggiuntiva oltre all'input stesso, `mergesort` produce una lista temporanea di dimensioni pari alla somma di quelle da fondere. E oltretutto deve ricopiarne il contenuto nella lista iniziale.

Con piccole modifiche al codice, che non vedremo, è possibile controllare meglio la gestione di queste liste temporanee e rendere il codice ancora più efficiente, dimezzando il tempo per le copie e riducendo quello per l'allocazione della memoria. In generale se nessuna di queste liste viene liberata prima della fine dell'algoritmo, la quantità di memoria aggiuntiva è $\Theta(n \log n)$, tuttavia se la memoria viene liberata in maniera più aggressiva allora quella aggiuntiva è $\Theta(n)$.

Capitolo 8

Ricorsione ed equazioni di ricorrenza

Analizzando il Mergesort abbiamo visto che il tempo di esecuzione di un algoritmo ricorsivo può essere espresso come **un'equazione di ricorrenza**, ovvero un'equazione del tipo

$$T(n) = \begin{cases} C & \text{when } n \leq c_0 \\ \sum_i a_i T(g_i(n)) + f(n) & \text{when } n > c_0 \end{cases} \tag{8.1}$$

dove a_i, C e c_0 costanti positive intere e g_i e f sono funzioni da $\mathbb{N}$ a $\mathbb{N}$, e vale sempre che $g_i(n) < n$.

Ad esempio il running time di Mergesort è $T(n) = 2T(n/2)+\Theta(n)$.[1] Mentre il running time della ricerca binaria è:

$$T(n) = T(n/2) + \Theta(1)$$

Ci sono diversi metodi per risolvere le equazioni di ricorrenza, o comunque per determinare se $T(n)$ è $O(g)$ oppure $\Theta(g)$ per qualche funzione g. Spesso ci interessa solo l'asintotica e per di più a volte ci interessa solo una limitazione superiore dell'ordine di crescita.

[1] In molti casi non è necessario essere precisi nel quantificare $T(1)$, oppure quali sono i valori esatti di C e c_0. Nella maggior parte quei valori condizionano $T(n)$ solo di un fattore costante, che viene comunque ignorato dalla notazione asintotica. Lo stesso vale per la funzione $f(n)$: riscalarla incide sulla soluzione della ricorrenza per un fattore costante.

8.1 Metodo di sostituzione

Nota: il contenuto di questa parte degli appunti può essere approfondito nel Paragrafo 4.3 del libro di testo *Introduzione agli Algoritmi e strutture dati* di Cormen, Leiserson, Rivest e Stein.

Si tratta di indovinare la soluzione della ricorrenza e verificarla dimostrandone la correttezza via induzione matematica. Ad esempio risolviamo la ricorrenza del Mergesort utilizzando come tentativo di soluzione $T(n) \leq cn \log n$ per c "grande abbastanza". Il caso base è verificato scegliendo $c > T(1)$. Assumiamo poi che `merge` utilizzi dn operazioni e che $c > d$. E utilizziamo l'ipotesi induttiva per sostituire nella ricorrenza.

$$T(n) = 2T(n/2)+dn \leq 2c(n/2)\log(n/2)+dn \leq cn \log n - cn + dn \leq cn \log n$$

L'uso dell'induzione per risolvere la ricorrenza può portare ad errori legati alla notazione asintotica. Facciamo conto che vogliamo dimostrare che $T(n) = O(n)$, ovvero $T(n) \leq cn$ per qualche c.

$$T(n) = 2T(n/2) + dn \leq 2cn/2 + dn \leq cn + dn$$

Si sarebbe tentati di dire che $(c + d)n = O(n)$ e che quindi ci siamo riusciti. Tuttavia la dimostrazione usa l'ipotesi induttiva che $T(n') \leq cn'$ per $n' < n$ e quindi se da questa ipotesi non deduciamo la stessa forma $T(n) \leq cn$ l'induzione non procede correttamente.

8.2 Metodo iterativo e alberi di ricorsione

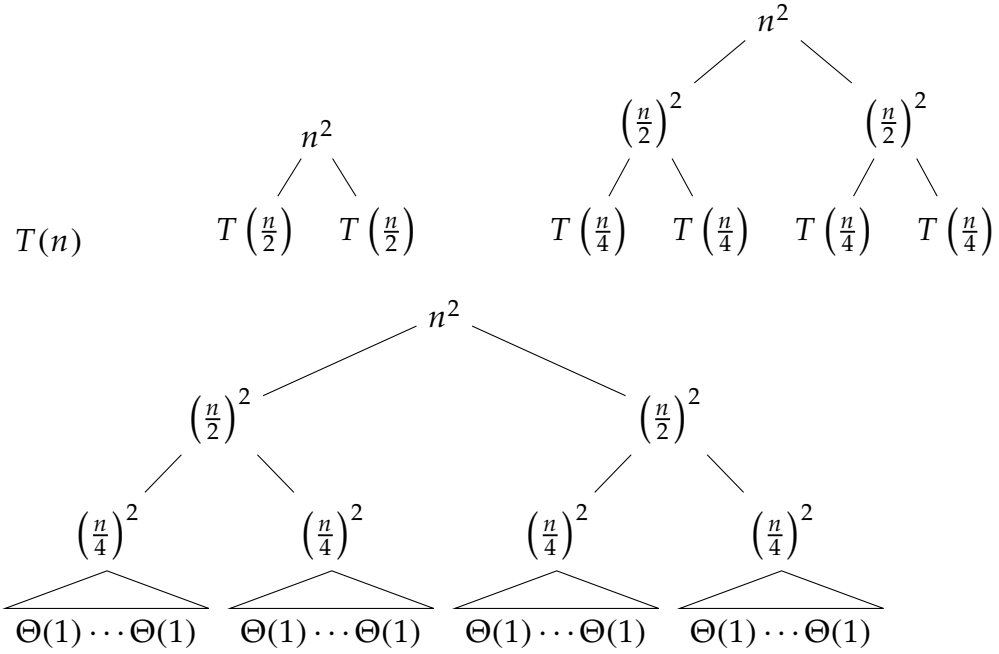
Nota: il contenuto di questa parte degli appunti può essere approfondito nel Paragrafo 4.4 del libro di testo *Introduzione agli Algoritmi e strutture dati* di Cormen, Leiserson, Rivest e Stein.

É il metodo che abbiamo utilizzato durante l'analisi delle performance di Mergesort. L'idea è quella di iterare l'applicazione della ricorrenza fino al caso base, sviluppando la formula risultante e utilizzando manipolazioni algebriche per determinarne il tasso di crescita.

Esempio: Analizziamo la ricorrenza $T(n) = 3T(\lfloor n/4 \rfloor) + n$

$$
\begin{aligned}
T(n) = \quad & n + 3T(\lfloor n/4 \rfloor) \\
= \quad & n + 3\lfloor n/4 \rfloor + 9T(\lfloor n/16 \rfloor) \\
= \quad & n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor) \\
= \quad & \dots \\
\leq \quad & n + 3n/4 + 9n/16 + 27n/64 + \ldots + 3^{\log_4(n)}\Theta(1) \\
\leq \quad & n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + \Theta(n^{\log_4(3)}) \\
= \quad & 4n + O(n) = O(n)
\end{aligned}
$$

Per visualizzare questa manipolazione è utile usare un **albero di ricorsione**. Ovvero una struttura ad albero che descrive l'evoluzione dei termini della somma. Vediamo ad esempio $T(n) = 2T(n/2) + n^2$



- L'albero ha $\log n$ livelli
- Il primo livello ha n^2 operazioni, il secondo ne ha $n^2/2$, il terzo ne ha $n^2/4, \ldots$
- L'ultimo ha $\Theta(n)$ operazioni.

il numero totale di operazioni è $\Theta(n) + n^2\left(1 + \frac{1}{2} + \frac{1}{4} + \ldots\right) = \Theta(n^2)$

8.3 Master Theorem

Nota: il contenuto di questa parte degli appunti può essere approfondito nei Paragrafi 4.5 (e opzionalmente 4.6) del libro di testo *Introduzione agli Algoritmi e strutture dati* di Cormen, Leiserson, Rivest e Stein.

Questi due metodi richiedono un po' di abilità e soprattutto un po' di improvvisazione, per sfruttare le caratteristiche di ogni esempio. Esiste un teorema che raccoglie i casi più comuni e fornisce la soluzione della ricorrenza direttamente.

Teorema 10. *Siano $a \geq 1$ e $b > 1$ costanti e $f(n)$ una funzione, e $T(n)$ definito sugli interi non negativi dalla ricorrenza:*

$$T(n) = aT(n/b) + f(n) \ ,$$

dove n/b rappresenta $\lceil n/b \rceil$ o $\lfloor n/b \rfloor$. Allora $T(n)$ può essere asintoticamente limitato come segue

1. *Se $f(n) = O(n^{\log_b(a)-\epsilon})$ per qualche costante $\epsilon > 0$, allora $T(n) = \Theta(n^{\log_b(a)})$;*
2. *Se $f(n) = \Theta(n^{\log_b(a)})$ allora $T(n) = \Theta(n^{\log_b(a)} \log n)$;*
3. *Se $f(n) = \Omega(n^{\log_b(a)+\epsilon})$, per qualche costante $\epsilon > 0$, e se $af(n/b) < cf(n)$ per qualche $c < 1$ e per ogni n sufficientemente grande, allora $T(n) = \Theta(f(n))$.*

Notate che il teorema non copre tutti i casi. Esistono versioni molto più sofisticate che coprono molti più casi, ma questa versione è più che sufficiente per i nostri algoritmi.

- Mergesort è il caso 2, con $a = b = 2$ e $f(n) = \Theta(n)$.
- Ricerca binaria è il caso , con $a = 1, b = 2$ e $f(n) = \Theta(1)$.
- $T(n) = 2T(n/2) + n^2$ è il caso 3.

Non vedremo la dimostrazione ma è sufficiente fare uno sketch dell'abero di ricorsione per vedere che questo ha

- altezza $\log_b n$;
- ogni nodo ha a figli;
- al livello più basso ci sono $a^{\log_b(n)} = n^{\log_b(a)}$ nodi che costano $\Theta(1)$ ciascuno;

- i nodi a distanza i da quello iniziale costano, complessivamente, $a^i f(n/b^i)$.

Dunque il costo totale è: $\Theta(n^{\log_b(a)}) + \sum_{j=0}^{\log_b(n)-1} a^j f(n/b^j)$. In ognuno dei tre casi enunciati dal teorema, l'asintotica è quella indicata.

Dimostrazione del Master Theorem *la dimostrazione non fa parte del programma di esame ed è inclusa solo per completezza.*

Abbiamo già osservato che il costo totale dell'algoritmo è

$$T(n) = \Theta(n^{\log_b(a)}) + \sum_{j=0}^{\log_b(n)-1} a^j f(n/b^j) \tag{8.2}$$

e ora stimiamo quanto vale questa sommatoria nei tre casi discussi nel teorema. Ci servirà anche osservare che per ogni $i \geq 0$ abbiamo la catena di equivalenze

$$a^i \cdot \left(\frac{n}{b^i}\right)^{\log_b(a)} = a^i \cdot \left(\frac{1}{b}\right)^{i\log_b(a)} \cdot n^{\log_b(a)} = a^i \cdot \left(\frac{1}{a}\right)^i \cdot n^{\log_b(a)} = n^{\log_b(a)} \tag{8.3}$$

Procediamo col dimostrare i tre casi

Caso 1: chiaramente l'equazione (8.2) implica che $T(n) = \Omega(n^{\log_b(a)})$, quindi per concludere il caso 1 è sufficiente dimostrare che la sommatoria in (8.2) sia $O(n^{\log_b(a)})$.

Poiché $f(n) = O(n^{\log_b(a)-\epsilon})$, la sommatoria diventa

$$\sum_{j=0}^{\log_b(n)-1} O\left(a^j \cdot \left(\frac{n}{b^j}\right)^{\log_b(a)-\epsilon}\right) . \tag{8.4}$$

Portiamo la sommatoria dentro l'operatore O,

$$O\left(\sum_{j=0}^{\log_b(n)-1} a^j \cdot \left(\frac{n}{b^j}\right)^{\log_b(a)-\epsilon}\right) \tag{8.5}$$

ovvero

$$O\left(n^{\log_b(a)-\epsilon} \cdot \sum_{j=0}^{\log_b(n)-1} a^j \cdot \left(\frac{1}{b^j}\right)^{\log_b(a)-\epsilon}\right) \tag{8.6}$$

ovvero

$$O\left(n^{\log_b(a)-\epsilon} \cdot \sum_{j=0}^{\log_b(n)-1} \left(\frac{ab^\epsilon}{b^{\log_b(a)}}\right)^j\right) . \tag{8.7}$$

Ora usiamo che $b^{\log_b(a)}$ è uguale ad a e otteniamo

$$O\left(n^{\log_b(a)-\epsilon} \cdot \sum_{j=0}^{\log_b(n)-1} (b^\epsilon)^j\right) \tag{8.8}$$

ovvero[2]

$$O\left(n^{\log_b(a)-\epsilon} \cdot \frac{b^{\epsilon \log_b(n)} - 1}{b^\epsilon - 1}\right) . \tag{8.9}$$

Visto che b ed ϵ sono costanti, e che $b^{\epsilon \log_b(n)} = n^\epsilon$, abbiamo che $\frac{b^{\epsilon \log_b(n)}-1}{b^\epsilon-1}$ vale $O(n^\epsilon)$, quindi l'ultima espressione vale

$$O\left(n^{\log_b(a)}\right) . \tag{8.10}$$

Questo conclude la dimostrazione del caso 1.

Caso 2: usiamo l'ipotesi $f(n) = \Theta(n^{\log_b(a)})$ valida in questo caso, ovvero esistono $n_0 > 0$ e $0 < C < D$ per cui se $n \geq n_0$ allora

$$Cn^{\log_b(a)} \leq f(n) \leq Dn^{\log_b(a)} . \tag{8.11}$$

Applichiamo le equazioni (8.3) alle disuguaglianze in (8.11) per ottenere che per ogni $i \geq 0$ e ogni $n > n_0$ vale

$$Cn^{\log_b(a)} \leq a^i f\left(\frac{n}{b^i}\right) \leq Dn^{\log_b(a)} \tag{8.12}$$

Dunque applicando queste limitazioni inferiori e superiori a tutti i termini della sommatoria in (8.2) otteniamo

$$T(n) = \Theta(n^{\log_b(a)}) + \sum_{j=0}^{\log_b(n)-1} \Theta(n^{\log_b(a)}) = \Theta(n^{\log_b(a)} \log n) . \tag{8.13}$$

Va notato che nell'ultimo passaggio abbiamo usato il fatto che $\log_b(n) = \Theta(\log n)$.

[2]Stiamo usando la formula della sommatoria geometrica, ovvero $\sum_{j=0}^{m} A^i = \frac{A^{m+1}-1}{A-1}$.

Caso 3: chiaramente $T(n) = \Omega(f(n))$, quindi per dimostrare il terzo caso è sufficiente dimostrare che $T(n) = O(f(n))$. Partiamo dall'equazione (8.2) e fissiamo n_0 e $c < 1$ tali che la disuguaglianza $af(n/b) < cf(n)$ valga. Questo è possibile per le ipotesi del terzo caso.

Applicando la disuguaglianza ripetutamente abbiamo che

$$a^i f(n/b^i) < c^i f(n) \ , \tag{8.14}$$

ma solo per i tale che n/b^i sia ancora maggiore di n_0, visto che l'ipotesi del terzo caso ci garantisce solo questo. Visto che n_0 è una costante (ovvero non dipende da n) la disuguaglianza vale per $n \geq n_0$ e ogni i che va da 0 a $\log_b(n) - t$ dove t è una costante. In particolare possiamo fissare t come il più piccolo numero intero tale che $b^t > n_0$. Sostituendo in (8.2) otteniamo

$$T(n) \leq \Theta(n^{\log_b(a)}) + \sum_{j=0}^{\log_b(n)-t} c^j f(n) + \sum_{j=\log_b(n)-t+1}^{\log_b(n)-1} a^j f(n/b^j) \ . \tag{8.15}$$

La prima sommatoria può essere estesa all'infinito e la seconda ha un numero costante di termini, per i quali vale sempre che $n/b^j < n_0$ e quindi $f(n/b^j) = \Theta(1)$. Dunque la seconda sommatoria vale al massimo $O(ta^{\log_b(n)}) = O(tn^{\log_b(a)}) = O(tn^{\log_b(a)})$. La stima (8.15) alla fina vale al massimo

$$T(n) \leq \Theta(n^{\log_b(a)}) + \sum_{j=0}^{\infty} c^j f(n) \leq \Theta(n^{\log_b(a)}) + \frac{1}{1-c} f(n) = O(f(n)) \ . \tag{8.16}$$

L'ultima diseguaglianza è vera perché $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ per ipotesi, e quindi domina sui termini $O(n^{\log_b(a)})$.

Capitolo 9

Ordinamenti in tempo lineare

Nota: il contenuto di questa parte degli appunti può essere approfondito nel Paragrafo 8.2 del libro di testo *Introduzione agli Algoritmi e strutture dati* di Cormen, Leiserson, Rivest e Stein.

Esistono modi di ordinare che impiegano solo $\Theta(n)$, ma questi metodi non sono, ovviamente, ordinamenti per confronto. Sfruttano invece il fatto che gli elementi da ordinare appartengano ad un dominio limitato.

9.1 Esempio: Counting Sort

Il countig sort si basa su un idea molto semplice: se ad esempio dobbiamo ordinare una sequenza di n elementi, dove ognuno dei quali è un numero da 1 a 10, possiamo farlo facilmente in tempo lineare:

1. tenendo 10 contatori $n_1, \ldots, n_{10}$;
2. fare una scansione della lista aggiornando i contatori;
3. riporre nella lista n_1 copie di 1, n_2 copie di 2, ecc. . .

```
def countingsort1(seq):
    if len(seq)==0:
        return
    # n operazioni
    a = min(seq)
    b = max(seq)
    # creazione dei contatori
    counter=[0]*(b-a+1)
```

```
    for x in seq:
        counter[x-a] = counter[x-a] + 1
    # costruzione dell'output
    output=[]
    for v,nv in enumerate(counter,start=a):
        output.extend( [v]*nv  )
    return output

print(countingsort1([7,6,3,5,7,9,3,2,3,5,6,7,8,8,9,9,5,4,3,2,2,3,4,6,8,8]))
```

```
[2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 5, 5, 5, 6, 6, 6, 7, 7, 7, 8, 8, 8, 8, 9, 9, 9]
```

Ovviamente qualunque tipo di dato ha un minimo e un massimo, in una lista finita. Tuttavia se la lista contiene elementi in un dominio molto grande (e.g. numeri tra 0 e n^{10} dove n è la lunghezza dell'input) allora questo algoritmo è meno efficiente degli algoritmi per confronti.

9.2 Dati contestuali

Negli algoritmi di ordinamento per confronto ci i dati originali vengono spostati o copiati all'interno della sequenza, e tra la sequenza ed eventuali liste temporanee.

Si immagini ad esempio il caso in cui ogni elemento nella lista di input sia una tupla `(i,dati)` dove `i` è la **chiave** rispetto a cui ordinare, ma `dati` invece è informazione contestuale arbitraria. Negli ordinamenti per confronto l'informazione contestuale viene spostata insieme alla chiave. La nostra implementazione del countingsort non gestisce questo caso, e va modificata.

1. Dati i contatori n_i, calcoliamo in quale intervallo della sequenza di output vadano inseriti gli elementi in input con chiave i. L'intervallo è tra le due quantità $\sum_{j=0}^{i-1} n_j$ (incluso) e $\sum_{j=0}^{i} n_j - 1$ (escluso).

2. Scorriamo l'input nuovamente e copiamo gli elementi in input nella lista di output.

```
def countingsort2(seq):
    if len(seq)==0:
        return
    # n operazioni
    a = min(k for k,_ in seq)
    b = max(k for k,_ in seq)

    # creazione dei contatori
    counter=[0]*(b-a+1)
    for k,_ in seq:
```

```
        counter[k-a] += 1
    # posizioni finali di memorizzazione
    posizioni=[0]*(b-a+1)
    for i in range(1,len(counter)):
        posizioni[i]=posizioni[i-1]+counter[i-1]

    # costruzione dell'output
    for k,data in seq[:]:
        seq[posizioni[k-a]]=(k,data)
        posizioni[k-a] = posizioni[k-a] + 1

sequenza=[(3,"paul"),(4,"ringo"),(1,"george"),(1,"pete"),(3,"stuart"),(4,"john")]
countingsort2(sequenza)
print(sequenza)
```

```
[(1, 'george'), (1, 'pete'), (3, 'paul'), (3, 'stuart'), (4, 'ringo'), (4, 'john')]
```

Capitolo 10

Ordinamento stabile

Nell'esempio precedente abbiamo visto che ci sono elementi diversi che hanno la stessa chiave di ordinamento. In generale una lista da ordinare può contenere elementi "uguali" nel senso che seppure distinti, per quanto riguarda l'ordinamento possono essere scambiati di posizione senza problemi, ad esempio `(1,'george')` e `(1,"pete")` possono essere invertiti nella ordinata, senza che l'ordinamento sia invalidato.

Si dice che un ordinamento è **stabile** se non modifica l'ordine relativo degli elementi che hanno la stessa chiave. Un'inversione nell'ordinamento di una sequenza `S` è una coppia di posizioni `i`, `j` nella lista, $0 \leq$ `i` $<$ `j` $<$ `len(S)` , tali che il valori in `S[i]` si trova dopo il valore in `S[j]` una volta finito l'ordinamento. Un ordinamento stabile minimizza il numero di inversioni.

Tutti gli ordinamenti che abbiamo visto fino ad ora sono stabili. Per esempio nel caso di ordinamenti per confronto è capitato di dover fare operazioni del tipo

```
if S[i] <= S[j]:
   operazioni che non causano un'inversione tra S[i] e S[j]
else:
   operazioni che causano un'inversione tra S[i] e S[j]
```

dove `i` è minore di `j`. Se invece dell'operatore <= avessimo utilizzato l'operatore < allora il comportamento dell'algoritmo sarebbe cambiato solo nel caso in cui `S[i]` fosse stato uguale a `S[j]`. L'ordinamento sarebbe stato comunque valido ma non sarebbe più stato un ordinamento stabile.

10.1 Ordinamenti multipli a cascata

Se avete una lista di brani nel vostro lettore musicale tipicamente avrete i vostri brani ordinati, semplificando, per

1. Artista
2. Album
3. Traccia

nel senso che i brani sono ordinati per Artista, quelli dello stesso artista sono ordinati per Album, e quelli nello stesso album sono ordinati

Un modo per ottenere questo risultato è ordinare prima per Traccia, poi per Album, e poi per Artista. Questo avviene perché gli ordinamenti usati sono stabili. Quando si ordina per Album, gli elementi con lo stesso Album verrano mantenuti nelle loro posizioni relative, che erano ordinate per Traccia. Successivamente una volta ordinati per Artista, i brani dello stesso Artista mantengono il loro ordine relativo, ovvero per Album e Traccia.

In generale è possibile ordinare rispetto a una serie di chiavi differenti, `key1`, `key2`, ... `keyN`, in maniera gerarchica, ordinando prima rispetto `keyN` e poi andando su fino a `key1`. Modifichiamo `countinsort` per farlo lavorare su una chiave di ordinamento arbitraria.

```
def default_key(x):
    return x

def countingsort(seq,key=default_key):
    if len(seq)==0:
        return
    # n operazioni
    a = min(key(x) for x in seq)
    b = max(key(x) for x in seq)
    # creazione dei contatori
    counter=[0]*(b-a+1)
    for x in seq:
        counter[key(x)-a] = counter[key(x)-a] + 1
    # posizioni finali di memorizzazione
    posizioni=[0]*(b-a+1)
    for i in range(1,len(counter)):
        posizioni[i]=posizioni[i-1]+counter[i-1]
    # costruzione dell'output
    for x in seq[:]:
        seq[posizioni[key(x)-a]]=x
        posizioni[key(x)-a] = posizioni[key(x)-a] + 1
```

Capitolo 11

Ordinare sequenze di interi grandi Radixsort

Nota: il contenuto di questa parte degli appunti può essere approfondito nel Paragrafo 8.3 del libro di testo *Introduzione agli Algoritmi e strutture dati* di Cormen, Leiserson, Rivest e Stein.

Abbiamo già detto che il `countingsort` è un ordinamento in tempo lineare, adatto a ordinare elementi le cui chiavi di ordinamento hanno un range molto limitato. Ma se i numeri sono molto grandi che possiamo fare?

Non possiamo ordinare una lista di numeri positivi da 32 bit con il counting sort, perché la lista dei contatori sarebbe enorme (e piena di zeri). Però possiamo considerare un numero di 32 come una tupla di 32 elementi $b_{31} \dots b_0$ in $\{0, 1\}$ ed utilizzare un ordinamento stabile per

- ordinare rispetto a b_0
- ordinare rispetto a b_1
- ...
- ordinare rispetto a b_{31}

Oppure, invece di lavorare bit per bit, possiamo considerare un numero di 32 come una tupla di 4 elementi $b_3 b_2 b_1 b_0$ in $\{0, \dots, 255\}$ ed utilizzare un ordinamento stabile per

- ordinare rispetto a b_0

- ordinare rispetto a b_1
- ordinare rispetto a b_2
- ordinare rispetto a b_3

Naturalmente usare una decomposizione più fitta richiede più chiamate ad ordinamento, ma ognuno su un dominio più piccolo. Il giusto compromesso dipende dalle applicazioni. Ora calcoliamo le chiavi b_i utilizzando quattro funzioni.

```
def key0(x):
    return x & 255

def key1(x):
    return (x//256) & 255

def key2(x):
    return (x//(256*256)) & 255

def key3(x):
    return (x//(256*256*256)) & 255

x =2**31 + 2**18 + 2**12 -1
print(key0(x), key1(x), key2(x), key3(x))
```

```
255 15 4 128
```

Dunque possiamo implementare `radixsort` (che ricordiamo, per come è stato realizzato funziona solo su numeri positivi di 32 bit).

```
def radixsort(seq):
    for my_key in [key0,key1,key2,key3]:
        countingsort(seq,key=my_key)


sequenza=[7,6,873823,5,7,9,3,2,12333,5,6132,7,8,1328,9,9,5,463432,4,3426,8,8]
radixsort(sequenza)
print(sequenza)
```

```
[2, 3, 4, 5, 5, 5, 6, 7, 7, 7, 8, 8, 8, 9, 9, 9, 1328, 3426, 6132, 12333, 463432, 873823]
```

11.1 Plot di esempio

In questo plot vediamo il tempo impiegato da questi algoritmi per ordinare una lista di numeri tra 0 e 1000000. Questi algoritmi sono molto più veloci di `bubblesort` e `insertionsort` e questo si vede anche in pratica. Le liste di numeri non sono particolarmente lunghe (solo 100000 elementi),

ma impossibili da ordinare utilizzando ordinamenti $\Theta(n^2)$. Vediamo tre algoritmi:

- Mergesort
- Countingsort con intervallo [0,10000] e [0,1000000]
- Radixsort con 4 chiavi da 8 bit e con 2 chiavi da 16 bit

Il running time di `countingsort` è molto più condizionato dall'intervallo di valori che dalla lunghezza della sequenza da ordinare (almeno per soli 100000 elementi da ordinare).

Radixsort utilizza più chiamate a `countingsort` ma su un dominio più piccolo. Due chiavi a 16 bit sono più efficienti di 4 chiavi a 8 bit, e 16 bit producono uno spazio delle chiavi di 65536 elementi. Uno spazio più semplice da gestire per `countingsort` rispetto ad un domi-nio di `1000000` elementi.