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 4 settembre 2019.

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

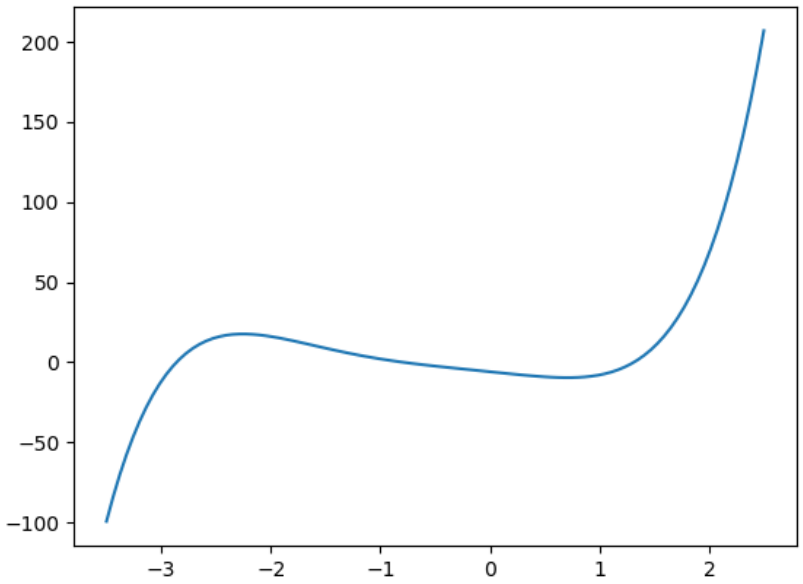
La ragione per cui usiamo i calcolatori elettronici, invece di fare i conti a mano, è che possiamo sfruttare la loro velocità. Tuttavia quando le quantità di dati da elaborare si fanno molto grandi, anche un calcolatore elettronico può diventare lento. Se il numero di operazioni da fare diventa molto 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 un polinomio

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 uno zero approssimato (i.e. un punto x per il quale $|P(x)| \leq \epsilon$ per un ϵ 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? Una possibilità è scorrere tutti i punti tra a e b (o almeno un insieme molto fitto di essi).

```
def trova_zero_A(f,a,b):
  Delta = (b - a) / 10000000
  x = a
  steps=1
  while x <= b:
    if  -0.00001 < f(x) < 0.00001:
      print("Trovato in {1} passi lo zero {0}.".format(x,steps))
      print(" - f({}) = {}".format(x,f(x)))
      break
    steps +=1
    x += Delta

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

```
Trovato in 3996048 passi lo zero -0.8011859001873525.
 - f(-0.8011859001873525) = 7.449223347499867e-06
```

Questo programma trova uno "zero" di P e ci ha messo 3996048 passi. Possiamo fare di meglio? Serve un'idea che renda questo calcolo molto più efficiente. Supponiamo che $P(a) < 0$ e $P(b) > 0$, allora possiamo provare a calcolate P sul punto $c = (a + b)/2$, a metà tra a e b.

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

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

```
def trova_zero_B(f,a,b):
    passi = 1
    start,end = a,b
    mid   = (a + b) / 2
    while not (-0.00001 < f(mid) < 0.00001):
        if f(start)*f(mid) < 0:
            end   = mid
        else:
            start = mid

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

```

```
    print("Trovato in {1} passi lo zero {0}.".format(mid,passi))
    print(" - f({}) = {}".format(mid,f(mid)))

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

```
Trovato in 17 passi lo zero -0.8011856079101562.
 - f(-0.8011856079101562) = 4.764512533839138e-06
```

Il secondo programma è molto più veloce perché invece di scorrere tutti i punti da -2.0 a 1.0, 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 un polinomio è una funzione continua. 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.

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                                                    5
    print("Non trovato. Eseguiti {} passi.".format(len(seq)))         6
```

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 python. 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)                                      1
```

```
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

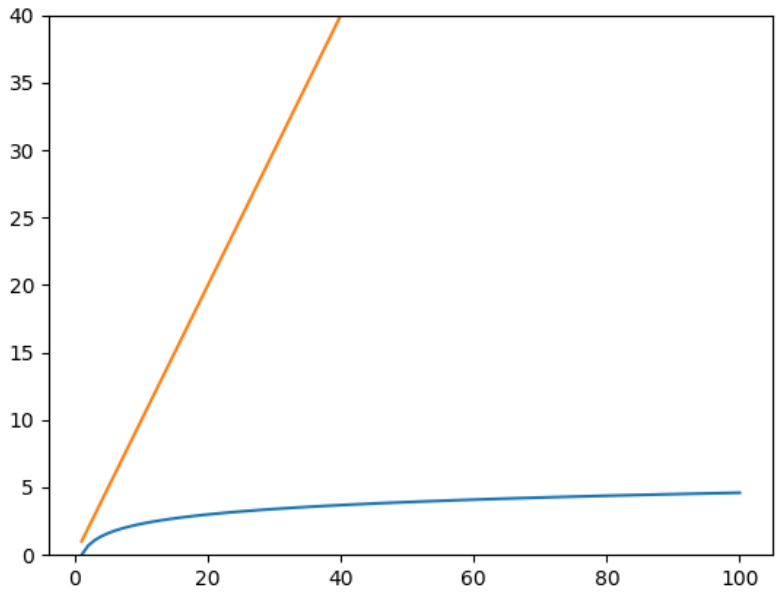
- si controlla il valore h a metà dell'intervallo;
- se $h > x$ allora x può solo trovarsi prima di v;
- se $h < x$ allora x può solo trovarsi dopo v;
- altrimenti h è uguale al valore cercato.

```
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: $R \log n + \text{Ord}(n)$

operazioni, dove $\text{Ord}(n)$ 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

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 determinate, 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 ordinate 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 il passi i-esimo 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:
        if L[pos-1] > x:
            L[pos] = L[pos-1]   #sposto a destra L[pos-1]
            pos = pos -1
        else:
            break
    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 più piccolo di tutti quelli precedenti, allora è necessario scorrere la sequenza all'indietro fino alla posizione 0. Se invece l'elemento e 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 ricerche binaria?

Il costo nel caso peggiore è circa $n^2 + t \log n$, che è comparabile con nR solo se si effettuano almeno $R \approx n$ ricerche.