

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>



Copyright © 2019, 2020 di Massimo Lauria.
Opera distribuita con Licenza Creative Commons
Attribuzione — Condividi allo stesso modo 4.0
(Internazionale).

Versione aggiornata al 5 febbraio 2020: questi 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: Gran parte del contenuto di questi appunti può essere approfondito nel libro *Introduzione agli Algoritmi e strutture dati (terza edizione)* di Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. In particolare ci riferiamo ai seguenti contenuti del libro:

- Paragrafi 2.1, 2.2, e 2.3;
- Problema 2-2 a pagina 34;
- Capitolo 3;
- Paragrafi 4.3, 4.4, 4.5, e opzionalmente 4.6;
- Capitolo 7;
- Paragrafi 8.1, 8.2, e 8.3;
- Paragrafi 31.1 e 31.2.

In ogni capitolo potete trovare le indicazioni bibliografiche specifiche.

Indice

1	Introduzione	5
1.1	Algoritmi e problemi reali	6
1.2	Efficienza computazionale	7
2	La ricerca in una sequenza	9
2.1	Trovare uno zero di una funzione continua	9
2.2	Ricerca di un elemento in una lista	13
2.3	Ricerca binaria	14
2.4	La misura della complessità computazionale	16
3	Ordinare dati con Insertion sort	19
3.1	Il problema dell'ordinamento	19
3.2	Insertion sort	20
3.3	La complessità di insertion sort.	23
3.4	Vale la pena ordinare prima di fare ricerche?	24
4	Notazione asintotica	25
4.1	Note su come misuriamo la lunghezza dell'input	28
4.2	Esercitarsi sulla notazione asintotica	28
5	Ordinare i dati con Bubblesort	31
5.1	Bubble sort semplificato	31
5.2	Miglioriamo l'algoritmo	33
6	Ordinamenti per confronti	37
6.1	Ordinamenti e Permutazioni	39
6.2	Dimostrazione del limite $\Omega(n \log n)$	42
6.3	Conclusione	43
7	Struttura a pila (stack)	45
7.1	Usi della pila: contesti annidati	46

7.2	Esempio: chiamate di funzioni	47
7.3	Esempio: espressioni matematiche	48
7.4	Lo stack e le funzioni ricorsive	49
7.5	Numeri di Fibonacci	51
8	Massimo Comun Divisore: algoritmo di Euclide	55
8.1	Implementazione dell'algoritmo MCD di Euclide	56
9	Mergesort	59
9.1	Un approccio divide-et-impera	59
9.2	Schema principale del mergesort	60
9.3	Implementazione	61
9.4	Fusione dei segmenti ordinati	61
9.5	Running time	62
9.6	Una piccola osservazione sulla memoria utilizzata	63
10	Quicksort	65
10.1	Schema principale del Quicksort	66
10.2	Implementazione della funzione di partizione	67
10.3	Running time	69
10.4	Considerazioni pratiche sul Quicksort	70
10.5	Confronto sperimentale con Mergesort	71
11	Ricorsione ed equazioni di ricorrenza	73
11.1	Metodo di sostituzione	74
11.2	Metodo iterativo e alberi di ricorsione	74
11.3	Master Theorem	76
12	Ordinamenti in tempo lineare	81
12.1	Esempio: Counting Sort	81
12.2	Dati contestuali	82
13	Ordinamento stabile	85
13.1	Ordinamenti multipli a cascata	86
14	Ordinare sequenze di interi grandi con Radixsort	89
14.1	Plot di esempio	90

Capitolo 1

Introduzione

“L’informatica non riguarda i computer più di quanto l’astronomia riguardi i telescopi.”

– E. Dijkstra

Questi appunti riguardano la parte teorica di un corso partito in modo molto pratico con la programmazione in Python. Una prima parte del corso così pratica non deve ingannare: l’informatica è una disciplina che ha una componente fortemente teorica, derivata dalla logica matematica. Anche se si è capaci di usare un linguaggio di programmazione, non è assolutamente detto che si sia in grado di decidere quale tecnica o ragionamento risolutivo esprimere in quel linguaggio.

Il pensiero computazionale/algorithmico è quella capacità, davanti ad un problema, di effettuare i seguenti quattro passi.

1. formulazione **non ambigua** del problema (astrazione)
2. determinare i **passi logici** che portino alla soluzione
3. codificare questi passi in un programma (programmazione)
4. eseguire il programma e analizzare la qualità dello stesso.

I computer sono efficaci per risolvere problemi che possono essere formulati in maniera non ambigua. Per esempio “calcolare la media di questi 100 numeri” può essere descritto in maniera precisa, mentre “trova il mio più caro amico tra i contatti di Facebook” è un’espressione vaga che non vuol dire nulla per un computer. La ragione è che siamo noi umani i primi a non

sapere cosa voglia dire in modo preciso questa frase, anche se il concetto ci è chiaro a livello intuitivo.

Una volta che un problema viene formulato in maniera precisa e non ambigua, è possibile determinare quali passi logici possono portare alla soluzione. In altri termini è possibile trovare un *algoritmo* che risolva il problema.

Ma che cosa vuol dire “trovare un algoritmo”? E che cos’è? Un algoritmo descrive una computazione che dato un input produce un output, e specifica quale operazione elementare effettuare in un dato momento del calcolo.

Anche se l’input può avere dimensione potenzialmente illimitata, la descrizione di un algoritmo è *finita e limitata*. La dimensione dell’algoritmo quindi non dipende dall’input, la cui lunghezza non è necessariamente limitata. L’algoritmo *termina sempre* e produce un risultato ben definito. L’algoritmo è una computazione anche molto complessa, ma è costituita da soli *passi elementari*. Tutto viene riassunto nella definizione di algoritmo fornita da Wikipedia.

"una sequenza ordinata e finita di passi (operazioni o istruzioni) elementari che conduce a un ben determinato risultato in un tempo finito".
— Wikipedia

Importante: un programma Python può descrivere efficacemente un algoritmo, tuttavia ci sono programmi Python che descrivono computazioni che non sono algoritmi (e.g. quelle che non terminano).

1.1 Algoritmi e problemi reali

Naturalmente l’informatica esiste ed è usata nel modo reale dove tutto è ambiguo e definito in maniera imprecisa o informale. È un compito niente affatto semplice quello di capire come formulare un problema reale in una forma più astratta e non ambigua, così che si possa tentare di progettare un algoritmo per risolverlo.



Figura 1.1: Muḥammad ibn Mūsā al-Khwārizmī (780–850 ca.) Matematico Persiano inventore dell’algebra, e quindi della manipolazione simbolica di elementi matematici. La parola algoritmo deriva dal suo nome.

Se poi l'algoritmo risolvesse di fatto il problema modello questa non è la fine della storia. Potrebbe essere il modello a non essere adatto, e può anche darsi che questo venga scoperto solo a posteriori. Chi scrive un programma (oppure chi lo commissiona) tipicamente *decide* come modellare i problemi che gli interessano, e le sue scelte di modellazione influiscono sull'impatto che l'algoritmo ha poi sul problema reale.

Trasformare un problema reale in un modello che sia manipolabile e gestibile con degli algoritmi è un'attività scientifica, ingegneristica, a volte addirittura filosofica. Richiede infatti di individuare, tra le altre cose, le *caratteristiche essenziali* del problema. Vediamo degli esempi di problemi affrontati da chi offre servizi di tipo informatico.

- mostrare all'utente un prodotto che vorrà comprare (Amazon);
- mostrare un* potenziale fidanzat* (OkCupid);
- mostrare le attività più interessanti tra quelle degli amici (Facebook).

In questi casi non si sa bene neppure cosa voglia dire aver risolto il problema. Normalmente si cerca di osservare il comportamento degli utenti per capire se il proprio algoritmo sta funzionando, se si può migliorare, ecc. . .

1. si **decide** come analizzare i dati;
2. si **decide** quali sono i parametri significativi;
3. si **decide** come usarli per ottenere conclusioni;
4. si aggiusta il tiro.

La ragione per cui i metodi statistici sono diventati estremamente importanti in informatica è che la rete, i negozi online, i social network hanno messo un'enorme mole di dati a disposizione. Naturalmente una volta che si decide come analizzare i dati e come usare i risultati di queste analisi, allora l'esecuzione di questi calcoli e analisi torna ad essere un processo squisitamente più algoritmico.

1.2 Efficienza computazionale

Un programma complesso, se eseguito su molti dati, può risultare lento, tuttavia alcuni programmi sono più veloci di altri. A volte lo stesso calcolo può essere effettuato in più modi, e allora quale scegliere? Ci sono diversi *algoritmi* per lo stesso problema:

- la correttezza dei risultati è il criterio principale;
- ma un programma troppo lento è inutilizzabile.

L'*efficienza computazionale* di un algoritmo è una misura della quantità di tempo e memoria che il computer impiega per risolvere un problema utilizzando l'algoritmo stesso. Vedremo che per lo stesso problema esistono diversi algoritmi, con prestazioni molto differenti. In realtà la teoria che studia questi problemi non vuole essere legata a singole tecnologie: invece di misurare il tempo di esecuzione (che varia da computer a computer) si cerca di stimare il numero di operazioni elementari che l'algoritmo impiega, almeno in termini di ordini di grandezza.

Tuttavia è sempre utile avere in mente una tabella come la seguente, con i tempi di esecuzione *reali* di operazioni tipiche. Le operazioni più costose sono gli accessi a memoria, pertanto la tabella si concentra su quelli. Per confronto, si consideri che un'istruzione CPU viene eseguita in circa 1 nanosecondo (ns), ovvero un milionesimo di secondo.

Accesso ai dati	tempo (in ns)
memoria cache L1	0.5
memoria cache L2	7
RAM	100
1MB seq. da RAM	250.000
4K random disco rigido SSD	150.000
1MB seq. da disco SSD	1.000.000
disco rigido HDD	8.000.000
1MB seq. da disco HDD	20.000.000
pacchetto dati Europa -> US -> Europa	150.000.000

Fonte: Peter Norvig, <http://norvig.com/21-days.html>

Updated by: Jeff Dean <https://ai.google/research/people/jeff>

Capitolo 2

La ricerca in una sequenza

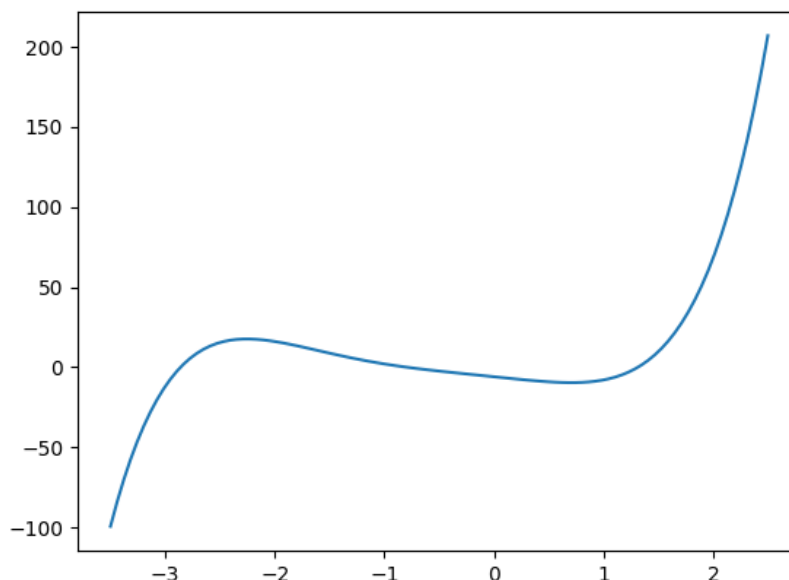
La ragione per cui usiamo i calcolatori elettronici, invece di fare i conti a mano, è che possiamo sfruttare la loro velocità. Tuttavia quando le quantità di dati da elaborare si fanno molto grandi, anche un calcolatore elettronico può diventare lento. Se il numero di operazioni da fare diventa molto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

2.1 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):
    return x**5 + 3*x**4 + x**3 - 7*x - 6
```

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

$$\lim_{x \rightarrow -\infty} P(x) \quad \lim_{x \rightarrow \infty} P(x)$$

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 2.1 (Teorema degli zeri). *Si consideri una funzione continua $f : [a, b] \rightarrow \mathbb{R}$. 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))
```

1

```
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
```

1

2

3

4

5

6

7

8

9

10

11

12

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

1

2

```
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)
```

1

2

3

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

1

2

3

```
-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(-0.801184892654419) = -1.8054627952679425e-06
Trovato in 23 passi lo zero -0.7390851974487305.
f(-0.7390851974487305) = -1.0750207657395094e-07
f(-0.7390842437744141) = 1.4885784402896007e-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`.

2.2 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):          1
    for i in range(len(seq)):          2
        if x==seq[i]:                 3
            print("Trovato l'elemento dopo {} passi.".format(i+1)) 4
            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 10000000 di numeri compresi tra -20000000 e 20000000, generati a caso utilizzando il generatore **pseudocasuale** di python. Poi cerca il valore 0 utilizzando la funzione `S`. Se il parametro `op-`

zionale `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.

2.3 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 minore 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 $\text{len}(seq)-1$ ed x è l'elemento da cercare.

Ad ogni passo

- si controlla il valore h a metà della sequenza;
- se $x < h$ allora x deve essere nella prima metà;
- se $h < x$ allora x deve essere nella seconda;
- se $h = x$ abbiamo trovato la posizione di x .

```

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 )

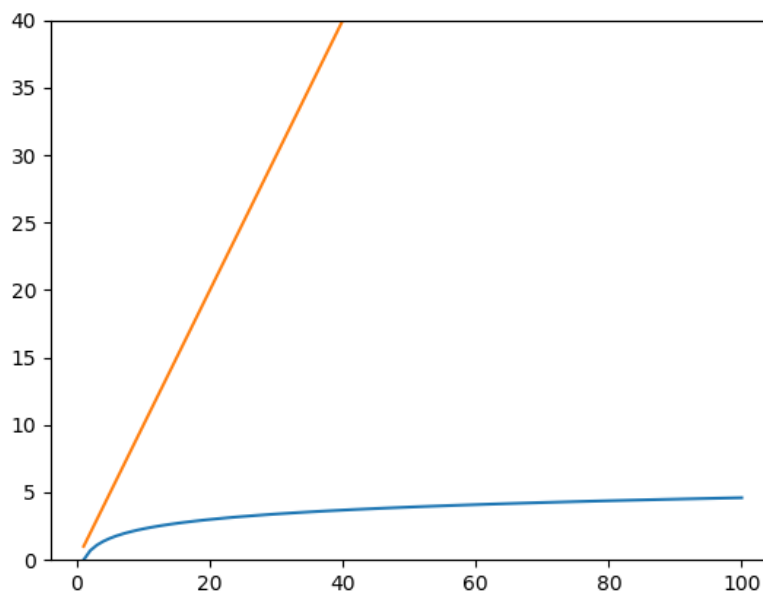
```

```

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?

2.4 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 3

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.

3.1 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 $a > b$.

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

1. garantisce che gli elementi nelle posizioni $0, \dots, i - 1$ siano ordinati;
2. trova la posizione $j \leq i$ tale che $L[j-1] \leq L[i] < L[j]$;
3. inserisce $L[i]$ 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 $L[i], L[i-1], \dots, L[0]$ (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

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

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.

⟨2.1 5.5 6.3 9.7 ◻ 5.2 7.8⟩ 3.2

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

⟨2.1 5.5 6.3 ◻ 9.7 5.2 7.8⟩ 3.2

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

⟨2.1 5.5 ◻ 6.3 9.7 5.2 7.8⟩ 3.2

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

⟨2.1 ◻ 5.5 6.3 9.7 5.2 7.8⟩ 3.2

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

⟨2.1 3.2 5.5 6.3 9.7 5.2 7.8⟩ .

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]	1
insertion_step(sequenza,4)	2

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"]	1
insertion_step(sequenza,3)	2

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)	1
-------------------------------	---

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)	1
-------------------------------	---

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):	1
"""Ordina la sequenza seq utilizzando insertion sort"""	2
for i in range(1,len(L)):	3
insertion_step(L,i)	4

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]	1
insertion_sort(dati)	2

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]	1
insertion_sort(dati)	2

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]

3.3 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'ini-

zio della sequenza. Quindi in questo caso al passo i -esimo si fanno sempre circa i operazioni. Il totale quindi è

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \approx n^2 \quad (3.1)$$

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

3.4 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 4

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 stima il numero di operazioni eseguite da 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 è 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. Informalmente la notazione asintotica serve a dire cose del tipo:

- su input di lunghezza n questo algoritmo ci mette al massimo $\approx n^2$ operazioni;

- su input di lunghezza n questo algoritmo ci mette nel caso peggiore **almeno** $\approx \log(n)$ operazioni;

e così via. Per fare questo in maniera più precisa abbiamo bisogno di una notazione matematica che ci permetta di mettere a paragone due funzioni, cioè di determinare quale domina l'altra, ignorando i dettagli di queste funzioni che sono per noi inessenziali. Introduciamo tre notazioni O , Ω , Θ il cui significato informale è il seguente:

- “ f è $O(g)$ ” vuol dire che f in un certo senso è più piccola di g , e si legge “ f è o grande di g ”;
- “ f è $\Omega(g)$ ” vuol dire che f in un certo senso è più grande di g , e si legge “ f è omega grande di g ”;
- “ f è $\Theta(g)$ ” vuol dire che f in un certo senso è simile a g , e si legge “ f è teta grande di g ”.

Ora discutiamo queste notazioni più formalmente, partendo dalla definizione di O .

Definizione: data una funzione $g : \mathbb{N} \rightarrow \mathbb{R}$ diciamo che

$$f \in O(g)$$

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

Alcune osservazioni su questa definizione:

- $O(g)$ è un insieme di funzioni;
- $f \in O(g)$ si legge anche “ f è o grande di g ”;
- 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 $O(n^2)$, non è $O(n^\ell)$ per nessun $\ell < 2$ ed è $O(n^\ell)$ per qualunque $\ell > 2$.

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 è $O(g)$. 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 $f : \mathbb{N} \rightarrow \mathbb{R}$ e $g : \mathbb{N} \rightarrow \mathbb{R}$ 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.

- $\Omega(g)$ e $\Theta(g)$ sono insiemi di funzioni;
- $f \in \Omega(g)$ si legge anche "f è omega grande di g";
- $f \in \Theta(g)$ si legge anche "f è teta grande di g".

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 è $\Omega(g)$, e che ha complessità $\Theta(g)$ quando c_A è $\Theta(g)$.

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 usando 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.

4.1 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.

4.2 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 5

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.

5.1 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 $0, 1, \dots, n - 2$.

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 $0, 1, \dots, n - 3$.

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 $n - i + 1, \dots, n - 1$ 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 $0, \dots, n - i$ 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]`

```

Python 3.8.1 (default, Jan 10 2020, 09:27:48)
[Clang 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
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]
python.el: native completion setup loaded

```


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 $0, \dots, n-2$ 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]

```

5.2 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.

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.

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

1

```
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 ordinata, 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]
```

Esercizio 5.1. 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 6

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 6.1. *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 effetti 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 contenente solo 0.

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	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 $o(n)$ operazioni, abbiamo dedotto che non è un algoritmo di ricerca corretto. \square

Esercizio 6.2. 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?

6.1 Ordinamenti e Permutazioni

Come si comporta un **qualunque** algoritmo di ordinamento per confronti su una di queste sequenze 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. Quindi 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

$$\pi : \{0, 1, \dots, n-1\} \rightarrow \{0, 1, \dots, n-1\}$$

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

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

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 $\{0, \dots, n-1\}$ 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.

Proposizione 6.3. *Il numero di permutazioni di n elementi è uguale a $n!$ (che si legge “ n fattoriale”), ovvero*

$$n \times (n-1) \times (n-2) \times \dots \times 2 \times 1$$

Ad esempio le permutazioni su 2 elementi sono 2, quelle su 3 elementi sono 6, quelle su 4 elementi sono 24, e così via.

Esercizio 6.4. 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:** $\pi_1(\pi_2\pi_3) = (\pi_1\pi_2)\pi_3$

¹Naturalmente 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 $X = \{x_0, x_1, x_2, \dots, x_t\}$ dove $x_0 < x_1 < x_2 < \dots < x_t$ allora le permutazioni su questo insieme sono essenzialmente identiche alle permutazioni su $\{0, \dots, t\}$.

- **Inversa:** per ogni π esiste **un'unica** permutazione inversa, che denotiamo come π^{-1} per cui $\pi\pi^{-1}$ è la permutazione identità.

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

Esercizio 6.5. 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

$$\langle a_0, a_1, a_2, a_3, \dots, a_{n-1} \rangle \quad (6.1)$$

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

$$\langle a_{\pi(0)}, a_{\pi(1)}, a_{\pi(2)}, a_{\pi(3)}, \dots, a_{\pi(n-1)} \rangle \quad (6.2)$$

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 6.6. Modificate il codice python di uno degli algoritmi di ordinamento 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 perdere la proprietà di essere un algoritmo di ordinamento per confronti. (*Suggerimento:* create una lista $[0, 1, 2, \dots,]$ 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.

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

Ripetiamo il teorema da dimostrare, per comodità.

Teorema 6.7. *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 $\{0, 1, \dots, n-1\}$ in ordine arbitrario. Osserviamo che per ogni permutazione π di $\{0, 1, \dots, n-1\}$, 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

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

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

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

che è $\Omega(n \log n)$. □

6.3 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 7

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                                1
                                                         2
pila.append(10)      # operazione PUSH                        3
print(pila)          4
                                                         5
pila.append("gatto") # operazione PUSH                       6
print(pila)          7
                                                         8
pila.pop()           # operazione POP                         9
print(pila)         10
                                                         11
pila.append([1,2,3]) # operazione PUSH                       12
print(pila)         13
                                                         14
pila.pop()           # operazione POP                         15
print(pila)         16
                                                         17
pila.pop()           # operazione POP                         18
print(pila)         19
                                                         20
pila.pop()           # POP da pila vuota = un errore        21

```

```

[10]
[10, 'gatto']
[10]
[10, [1, 2, 3]]
[10]
[]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/var/folders/kf/python-NqA5Eq", 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.

7.1 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.

7.2 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 primoLivello():
    i = 5
    print("Primo Livello 1: i={}".format(i))
    secondoLivello()
    print("Primo Livello 2: i={}".format(i))
    i = 4
    print("Primo Livello 3: i={}".format(i))
    secondoLivello()
    print("Primo Livello 4: i={}".format(i))

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

i=10
primoLivello()

```

```

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.

7.3 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 bilancia. 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
```

7.4 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 7.1. 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
8159152832478977343456112695961158942720000000000
```


7.5 Numeri di Fibonacci

La successione di Fibonacci è una sequenza di numeri F_n per $n \geq 0$ definita in maniera ricorsiva secondo la seguente formula:

$$F_0 = 0 \quad F_1 = 1 \quad F_n = F_{n-1} + F_{n-2} \quad \text{per } n \geq 2 \quad (7.1)$$

Ad esempio i primi valori della successione di Fibonacci sono:

0 1 1 2 3 5 8 13 21 34 55 89 ...

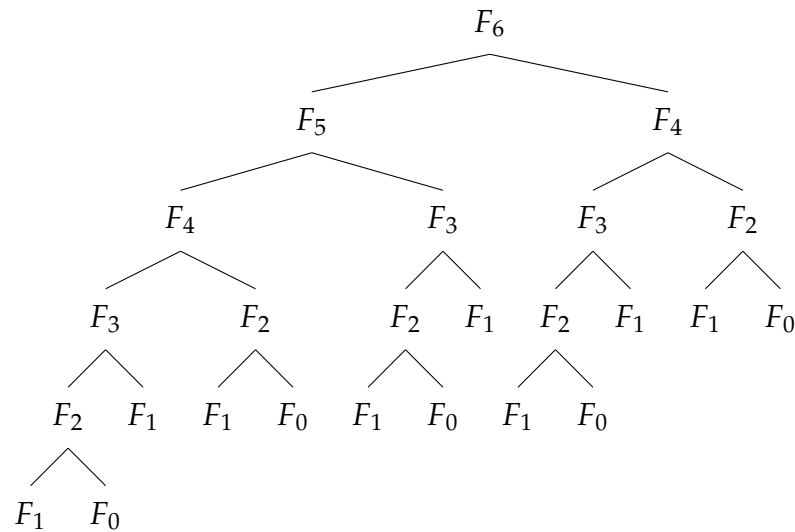
Questa successione può essere calcolata facilmente attraverso il programma ricorsivo seguente.

```
def fib1(n):
    if n < 0:
        raise ValueError("Fibonacci non definito su negativi.")
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib1(n-1) + fib1(n-2)
```

```
for n in range(16):
    print(fib1(n), end=' ')
print('')
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

Osserviamo per esempio l'albero che rappresenta le chiamate ricorsive fatte per calcolare F_6 .



È abbastanza evidente da questo albero che molte delle chiamate ricorsive calcolano lo stesso valore della sequenza di Fibonacci, e quindi duplicano del lavoro già fatto. Vediamo esattamente quante chiamate sono necessarie per calcolare i numeri di Fibonacci fino a 100.

```

Chiamate per Fibonacci 0: 1
Chiamate per Fibonacci 1: 1
Chiamate per Fibonacci 2: 3
Chiamate per Fibonacci 3: 5
Chiamate per Fibonacci 4: 9
Chiamate per Fibonacci 5: 15
Chiamate per Fibonacci 6: 25
Chiamate per Fibonacci 7: 41
Chiamate per Fibonacci 8: 67
Chiamate per Fibonacci 9: 109
...
Chiamate per Fibonacci 91: 15080227609492692857
Chiamate per Fibonacci 92: 24400320830243753475
Chiamate per Fibonacci 93: 39480548439736446333
Chiamate per Fibonacci 94: 63880869269980199809
Chiamate per Fibonacci 95: 103361417709716646143
Chiamate per Fibonacci 96: 167242286979696845953
Chiamate per Fibonacci 97: 270603704689413492097
Chiamate per Fibonacci 98: 437845991669110338051
Chiamate per Fibonacci 99: 708449696358523830149
Chiamate per Fibonacci 100: 1146295688027634168201
  
```

Il numero diventa enorme e ingestibile molto velocemente. Pochi tentativi e diventa evidente che **questa** versione ricorsiva permette di calcolare solo i primi numeri di Fibonacci. Mentre è possibile scrivere versioni ricorsive più efficienti, vediamo una semplice versione iterativa basata sulla seguente osservazione: se calcoliamo i numeri di Fibonacci da F_0 a F_n in successione, e li teniamo memorizzati, possiamo usare i valori precedenti (precalcolati)

per calcolare i valori successivi. Vediamo ad esempio che la funzione `fib2` può calcolare valori molto più avanzati nella successione.

```
def fib2(n):
    if n<0:
        raise ValueError("Fibonacci non definito su negativi.")

    if n == 0:
        return 0
    elif n==1:
        return 1

    F=[0,1]
    while len(F) <= n:
        F.append( F[-1] + F[-2] )

    return F[n]
```

```
for n in range(100,103):
    print(fib2(n), end=' ')
print('')
```

```
354224848179261915075 573147844013817084101 927372692193078999176
```

Visto che usiamo solo gli ultimi due valori della successione per calcolarne uno nuovo, forse non vale la pena tenere tutti i valori precedenti in memoria.

```
def fib3(n):
    if n<0:
        raise ValueError("Fibonacci non definito su negativi.")

    if n == 0:
        return 0
    elif n==1:
        return 1

    F=[0,1]
    for i in range(2,n+1):
        # calcola Fib(i) e Fib(i-1) da Fib(i-1) e Fib(i-2)
        F[-1], F[-2] = F[-1] + F[-2], F[-1]

    return F[-1]
```

```
print(fib3(300))
```

```
22232244629420445529739893461909967206666939096499764990979600
```


Capitolo 8

Massimo Comun Divisore: algoritmo di Euclide

Si ringrazia il Prof. Paolo Giulio Franciosa per aver contribuito agli appunti con questo capitolo.

Nota: il contenuto di questa parte degli appunti può essere approfondito nei Paragrafi 31.1 e 31.2 del libro di testo *Introduzione agli Algoritmi e strutture dati* di Cormen, Leiserson, Rivest e Stein.

Il Massimo Comun Divisore (MCD) di due numeri interi positivi¹ è il più grande intero positivo che divida entrambi. Ad esempio il MCD di 4 e 6 è 2; il MCD di 7 e 3 è 1; il MCD 12 e 6 è 6.

Un algoritmo banale per trovare il Massimo Comun Divisore (MCD) di due numeri naturali a, b consiste nel cercare un divisore comune tra a e b procedendo a ritroso a partire dal minimo tra a e b . Nel caso i due numeri non abbiano divisori in comune si arriverà fino al valore 1. Questo approccio porta ad eseguire, nel caso peggiore, un numero di tentativi pari al minimo tra a e b , ed è impraticabile se questo ha un numero di cifre anche limitato.

Infatti questo approccio, applicato a numeri di 20 cifre e utilizzando un computer attuale di media potenza, richiederebbe un tempo dell'ordine di 1000 anni.

¹Vediamo che la definizione di MCD si può estendere naturalmente ad altri casi. Poichè tutti i numeri interi positivi dividono lo zero, allora il MCD di 0 e n è n per ogni n intero positivo. La definizione di MCD si può anche estendere a numeri negativi ponendo il MCD di a e b uguale al MCD di $|a|$ e $|b|$, qualora almeno uno tra a e b non sia zero. Il MCD tra 0 e 0 è l'unico caso indefinito.

L'algoritmo di Euclide consente di abbattere drasticamente questa complessità, portando i tempi di esecuzione a frazioni di secondo anche su numeri di centinaia di cifre.

Vediamo come calcolare il MCD di a e b . Assumendo senza perdita di generalità $a \geq b$, abbiamo che $a = \ell \cdot b + r$ dove $\ell = a // b$ (l'operatore $//$ è la divisione intera) e $r = a \% b$ (l'operatore $\%$ è il resto della divisione intera). Ovviamente avremo $r < b$.

Semplici proprietà dell'aritmetica intera mostrano che:

- se d è un divisore di a e di b , allora d è anche un divisore di r .

Infatti, a/d è intero, quindi $\ell \cdot (b/d) + r/d$ è intero, e poiché b/d è intero anche r/d è intero;

- se d è un divisore di b e di r , allora d è anche un divisore di a .

Infatti d è un divisore di $a = \ell \cdot b + r$, poiché d divide sia $\ell \cdot b$ che r .

Ne deriva che i divisori comuni di a e b , con $a \geq b$, sono esattamente i divisori comuni di b e r .

L'algoritmo di Euclide è il seguente: dati due naturali a, b , con $a \geq b$:

- calcola il resto r della divisione tra a e b
- se r è uguale a 0 allora il MCD tra a e b è b
- altrimenti il MCD tra a e b è uguale al MCD tra b e r

8.1 Implementazione dell'algoritmo MCD di Euclide

Usando il ragionamento descritto sopra, l'algoritmo di Euclide si può implementare sia in modo iterativo che ricorsivo. L'idea è usare il fatto che il calcolo del MCD tra due numeri si può ridurre al calcolo del MCD tra due numeri più piccoli.

```
# MCD Euclide, versione iterativa      1
def MCDEuclideIterativo(a, b) :      2
    a, b = max(a,b), min(a,b)        3
    while b > 0 :                    4
        a, b = b, a % b              5
    return a                          6
```

```
# MCD Euclide, versione ricorsiva      1
def MCDEuclide(a,b) :                2
```



```

a, b = max(a,b), min(a,b)          3
return MCDEuclideRicorsiva(a,b)   4
def MCDEuclideRicorsiva(a,b) :    6
    if b == 0 :                   7
        return a                  8
    else :                         9
        return MCDEuclideRicorsiva(b, a % b) 10

```

Un esempio di applicazione dell'algoritmo sui numeri 178 e 46:

```

Iterazione 1: MCD(178,46) =
Iterazione 2: MCD(46,40) =
Iterazione 3: MCD(40,6) =
Iterazione 4: MCD(6,4) =
Iterazione 5: MCD(4,2) = 2

```

Un esempio su numeri di dimensioni maggiori è il seguente. Notare che 7160892618827651 e 6940754193709799 sono primi tra loro (non hanno fattori comuni), quindi il MCD è 1. L'approccio banale avrebbe eseguito circa 10^{16} tentativi di divisione, l'algoritmo di Euclide esegue 32 iterazioni.

```

Iterazione 1: MCD(7160892618827651,6940754193709799) =
Iterazione 2: MCD(6940754193709799,220138425117852) =
Iterazione 3: MCD(220138425117852,116463015056387) =
Iterazione 4: MCD(116463015056387,103675410061465) =
Iterazione 5: MCD(103675410061465,12787604994922) =
Iterazione 6: MCD(12787604994922,1374570102089) =
Iterazione 7: MCD(1374570102089,416474076121) =
Iterazione 8: MCD(416474076121,125147873726) =
Iterazione 9: MCD(125147873726,41030454943) =
Iterazione 10: MCD(41030454943,2056508897) =
Iterazione 11: MCD(2056508897,1956785900) =
Iterazione 12: MCD(1956785900,99722997) =
Iterazione 13: MCD(99722997,62048957) =
Iterazione 14: MCD(62048957,37674040) =
Iterazione 15: MCD(37674040,24374917) =
Iterazione 16: MCD(24374917,13299123) =
Iterazione 17: MCD(13299123,11075794) =
Iterazione 18: MCD(11075794,2223329) =
Iterazione 19: MCD(2223329,2182478) =
Iterazione 20: MCD(2182478,40851) =
Iterazione 21: MCD(40851,17375) =
Iterazione 22: MCD(17375,6101) =
Iterazione 23: MCD(6101,5173) =
Iterazione 24: MCD(5173,928) =
Iterazione 25: MCD(928,533) =
Iterazione 26: MCD(533,395) =
Iterazione 27: MCD(395,138) =
Iterazione 28: MCD(138,119) =
Iterazione 29: MCD(119,19) =
Iterazione 30: MCD(19,5) =
Iterazione 31: MCD(5,4) =
Iterazione 32: MCD(4,1) = 1

```


Capitolo 9

Mergesort

Nota: il contenuto di questa parte degli appunti può essere approfondito nel Paragrafo 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.

9.1 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.

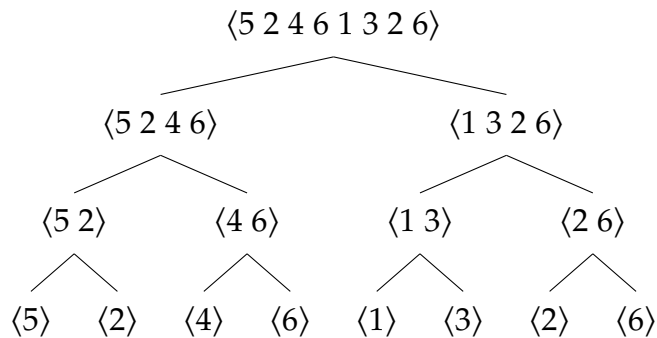
9.2 Schema principale del mergesort

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

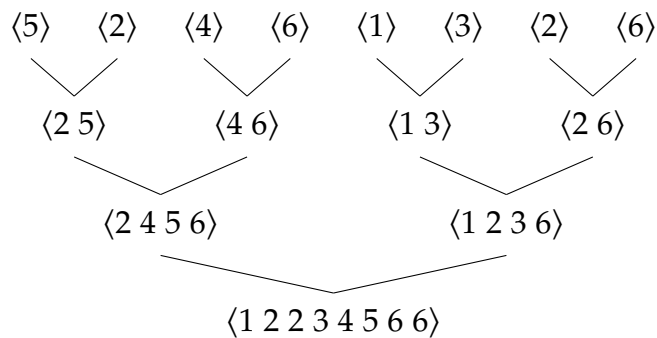
Vediamo ad esempio come si comporta il mergesort sull'input

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

La sequenza ordinata viene ottenuta mediante questa serie di suddivisioni



seguita da questa serie di fusioni



9.3 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)
```

9.4 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[low:high+1]
    for i in range(len(temp)):
        S[low+i] = temp[i]
```

Questo conclude l'algoritmo

<code>dati=[5,2,4,6,1,3,2,6]</code>	1
<code>mergesort(dati)</code>	2
<code>print(dati)</code>	3

<code>[1, 2, 2, 3, 4, 5, 6, 6]</code>

9.5 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

$$T(n) = 2T(n/2) + \Theta(n) \quad (9.1)$$

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

$$T(n) \leq 2T(n/2) + cn \quad T(1) \leq c. \quad (9.2)$$

Espandendo otteniamo

$$T(n) \leq 2T(n/2) + cn \leq 4T(n/4) + 2c(n/2) + cn = 4T(n/4) + 2cn \quad (9.3)$$

Si vede facilmente, ripetendo l'espansione, che

$$T(n) \leq 2^t T(n/2^t) + tcn \quad (9.4)$$

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

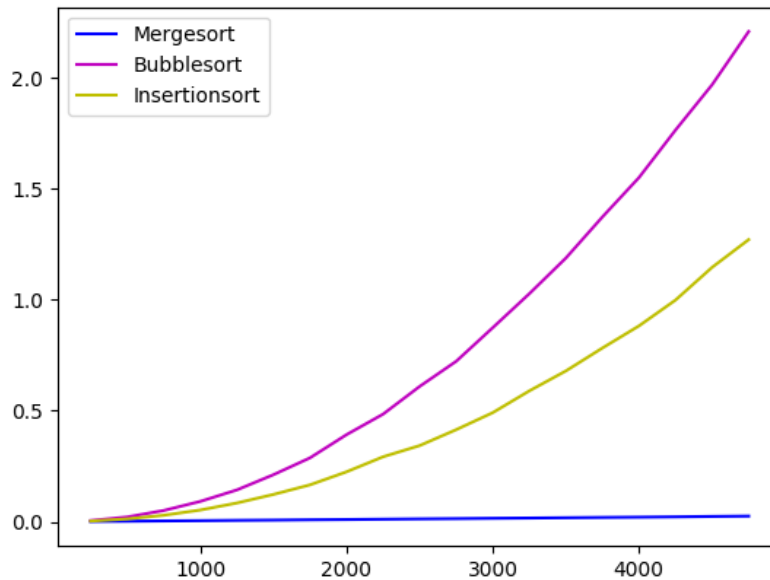
Il più piccolo valore di t^* per cui $n/2^{t^*} \leq 1$ è $O(\log n)$, e quindi il running time totale è $O(n \log n)$.

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 $\Theta(n \log n)$.

Vediamo i tempi di esecuzione dei tre algoritmi Bubblesort, Insertionsort e Mergesort su sequenze casuali di numeri. Osserviamo che il vantaggio

9.6. UNA PICCOLA OSSERVAZIONE SULLA MEMORIA UTILIZZATA⁶³

di Mergesort nel tempo di esecuzione asintotico non è soltanto teorico. Di fatto Mergesort è molto più veloce degli algoritmi a complessità quadratica.



9.6 Una piccola osservazione sulla memoria utilizzata

Mentre Bubblesort e Insertionsort non utilizzano molta memoria aggiuntiva oltre all'input stesso, la nostra implementazione di 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 10

Quicksort

Si ringrazia il Prof. Paolo Giulio Franciosa per aver contribuito agli appunti con questo capitolo.

Nota: il contenuto di questa parte degli appunti può essere approfondito nel Capitolo 7 del libro di testo *Introduzione agli Algoritmi e strutture dati* di Cormen, Leiserson, Rivest e Stein.

L'algoritmo di ordinamento per confronto Quicksort, come il Mergesort, segue l'approccio del divide et impera, e viene realizzato sfruttando la ricorsione. L'idea alla base dell'algoritmo è quella di scegliere un elemento separatore, detto **pivot**, e separare l'insieme da ordinare in due sottoinsiemi: l'insieme contenente gli elementi minori del pivot (che nel seguito chiameremo "elementi piccoli") e l'insieme contenente gli elementi maggiori o uguali al pivot (che nel seguito chiameremo "elementi grandi")

La sequenza ordinata sarà quindi formata dagli elementi piccoli ordinati, il pivot, e gli elementi grandi ordinati. L'ordinamento degli elementi piccoli e l'ordinamento degli elementi grandi vengono ottenuti mediante lo stesso metodo, applicato alla porzione di sequenza formata dagli elementi piccoli e alla porzione di sequenza formata dagli elementi grandi.

Il quick sort è in pratica e nel caso generale un algoritmo estremamente efficiente. Nonostante nel caso peggiore richieda tempo $O(n^2)$, si può dimostrare che nel caso medio richiede tempo $O(n \log n)$, e che l'eventualità che non impieghi tempo $O(n \log n)$ è decisamente remota. Per questo motivo molto spesso le procedure di ordinamento disponibili nelle librerie software si basano sul Quicksort.

10.1 Schema principale del Quicksort

L'algoritmo applica uno schema ricorsivo descritto di seguito. Va osservato che il passo 1 viene tipicamente realizzato compiendo una scelta casuale. Per questa ragione il Quicksort è, nella maggior parte delle sue implementazioni, un *algoritmo casuale*.

1. scegliere un elemento pivot. Per scongiurare che si presenti il caso peggiore è preferibile scegliere il pivot in maniera casuale tra gli elementi da ordinare;
2. permutare gli elementi della sequenza in modo tale che tutti gli elementi minori si trovino prima del pivot, e che tutti gli elementi maggiori del pivot si trovino dopo il pivot;
3. ordinare gli elementi minori del pivot;
4. ordinare gli elementi maggiori del pivot.

Come nel caso del Mergesort, per ordinare gli elementi minori e per ordinare gli elementi maggiori si riutilizza ancora il Quicksort, e quindi si genera una gerarchia di applicazioni dell'algoritmo annidate le une dentro le altre su parti di input sempre più piccole, fino ad arrivare a porzioni composte da un solo elemento, che sono ovviamente porzioni già ordinate.

L'implementazione del Quicksort, sfruttando la ricorsione, è la seguente

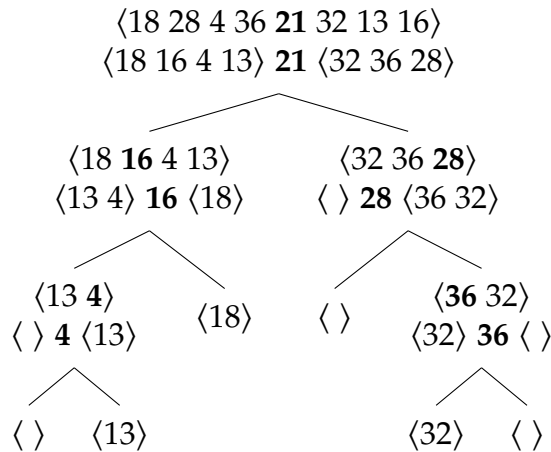
```
def quicksort(S, start=0, end=None):           1
    if end is None:                           2
        end = len(S)-1                       3
    if start >= end:                          4
        return                               5
    pivot_pos = partition(S, start, end)      6
    quicksort(S, start, pivot_pos - 1)       7
    quicksort(S, pivot_pos+1, end)           8
```

Il metodo `partition` ha il compito di permutare gli elementi della porzione di lista dalla posizione `start` alla posizione `end`, estremi compresi, scegliendo l'elemento pivot e spostando alla sinistra della lista gli elementi piccoli e alla destra gli elementi grandi. Inoltre, il metodo `partition` posiziona il pivot nella sua posizione definitiva (separando gli elementi piccoli dagli elementi grandi), e restituisce tale posizione. Nelle ultime due righe del metodo viene di nuovo richiamato il metodo `quicksort` per ordinare tra loro gli elementi piccoli e ordinare tra loro gli elementi grandi, rispettivamente.

Mostriamo nel seguito un esempio di applicazione del Quicksort partendo dall'input

$\langle 18 \ 28 \ 4 \ 36 \ 21 \ 32 \ 13 \ 16 \rangle .$

Per ciascuna porzione da ordinare verrà scelto casualmente un pivot, evidenziato in grassetto.



10.2 Implementazione della funzione di partizione

Il passo fondamentale del metodo *partition* è la scelta casuale del pivot tra gli elementi da ordinare e la successiva separazione degli elementi della lista in

- elementi minori del pivot;
- pivot;
- elementi maggiori o uguali al pivot.

Naturalmente, per ottenere la separazione si potrebbe scandire in sequenza la lista di partenza, privata del pivot, e costruire altre due liste in cui si separano gli elementi confrontandoli con il pivot. Questo metodo richiederebbe un notevole aggravio nella richiesta di memoria.


È invece possibile separare gli elementi "in-place", cioè senza utilizzare altre liste, nel seguente modo. Una volta scelto il pivot in maniera casuale, si scambia con l'elemento di testa. Si definiscono poi due indici i e j : l'indice i scandisce la lista da sinistra verso destra, a partire dall'elemento successivo al pivot, mentre l'indice j la scandisce da destra verso sinistra.

L'indice i si ferma quando si incontra un elemento maggiore o uguale al pivot (che quindi dovrebbe andare nella parte destra della lista), oppure quando incontra l'indice j . L'indice j si ferma quando si incontra un elemento minore del pivot (che quindi dovrebbe andare nella parte sinistra della lista), oppure quando incontra l'indice i . Se vengono incontrati due elementi da scambiare si esegue lo scambio. La scansione prosegue fino a quando gli indici i e j si incontrano.

Vediamo un esempio dell'esecuzione di `partition` con la lista seguente.

⟨18 28 4 36 21 32 13 16⟩

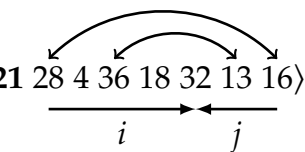
L'elemento scelto per essere il pivot è 21, qui evidenziato. Per prima cosa questo viene messo in testa alla lista scambiandolo con l'elemento in quella posizione,


 ⟨18 28 4 36 21 32 13 16⟩

ottenendo la lista

⟨21 28 4 36 18 32 13 16⟩

A questo punto vengono scambiati elementi nella lista in modo da ottenere le due sottoliste con gli elementi più piccoli e gli elementi più grandi del pivot, rispettivamente,

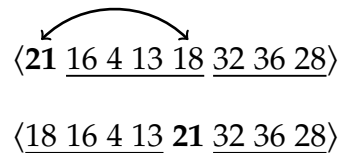

 ⟨21 28 4 36 18 32 13 16⟩

ottenendo la lista

⟨21 16 4 13 18 32 36 28⟩ .

nella quale abbiamo evidenziato le due sottoliste. Alla fine di questo procedimento avremo il pivot in testa alla lista, seguito dagli elementi minori del pivot e poi dagli elementi maggiori o uguali al pivot. Possiamo allora posizionare il pivot tra le due liste effettuando un solo scambio, che conclude la fase di separazione.

Per posizionare il pivot tra le due sequenze è ora sufficiente scambiarlo con l'ultimo elemento minore del pivot, che si troverà in posizione i oppure $i - 1$, a seconda di quale tra gli indici i e j è stato modificato per ultimo.



La funzione `partition` restituisce la posizione in cui è stato posizionato il pivot, che indica implicitamente anche il termine della porzione contenente elementi piccoli e l'inizio della porzione che contiene elementi grandi.

10.3 Running time

Il tempo di esecuzione del Quicksort dipende da come il metodo `partition` di volta in volta separa la porzione da ordinare in due parti. Diversamente da quanto accade per il Mergesort, le due porzioni possono avere dimensioni diverse. Su ciascuna porzione da ordinare viene eseguito il metodo `partition`. Tale metodo scandisce la porzione partendo dai due estremi verso il centro, eventualmente eseguendo alcuni scambi. Quindi se la porzione da ordinare contiene k elementi impiega tempo $\Theta(k)$.

Caso migliore Il caso migliore si verifica quando ogni volta che si esegue il metodo `partition` il pivot viene posizionato al centro della porzione da ordinare. In questo caso verranno eseguite due chiamate ricorsive, ciascuna su una porzione la cui dimensione è al massimo la metà della dimensione della porzione da ordinare. Poiché il metodo `partition` richiede tempo $O(n)$, dove n è la dimensione della porzione da ordinare, e vengono eseguite due chiamate ricorsive, ciascuna su al massimo $n/2$ elementi, otteniamo una equazione di ricorrenza identica a quella che descrive il tempo di esecuzione del Mergesort. Definendo come $T(n)$ il numero di operazioni necessarie per ordinare una lista di n elementi con quicksort, abbiamo

$$T(n) = 2T(n/2) + \Theta(n) \quad (10.1)$$

quando $n > 1$, altrimenti $T(1) = \Theta(1)$.

Il tempo di esecuzione del Quicksort è quindi, nel caso migliore, asintoticamente identico a quello del Mergesort, pari a $\Theta(n \log n)$.

Caso peggiore Il caso peggiore si verifica quando in ogni esecuzione del metodo `partition` viene scelto come pivot il minimo o il massimo tra gli elementi della porzione da ordinare.

Infatti in questo caso, dopo aver impiegato tempo $\Theta(n)$ per eseguire il `partition`, verrà eseguita una sola chiamata ricorsiva su una porzione di $n - 1$ elementi, poiché gli altri elementi della porzione da ordinare saranno tutti minori oppure tutti maggiori del pivot.

Quindi l'equazione di ricorrenza che descrive il tempo di esecuzione del Quicksort nel caso peggiore è

$$T(n) = T(n - 1) + \Theta(n) \quad (10.2)$$

quando $n > 1$, altrimenti $T(1) = \Theta(1)$.

Una semplice sostituzione mostra che

$$T(n) = n/2 \cdot \Theta(n) = \Theta(n^2) \quad (10.3)$$

10.4 Considerazioni pratiche sul Quicksort

Running time atteso Un'implementazione semplice di `partition` potrebbe scegliere come pivot il primo elemento della porzione da ordinare. Nel caso, non infrequente nella pratica, che la lista si presenti già ordinata o quasi ordinata, tale scelta porterebbe spesso ad utilizzare come elemento separatore proprio l'elemento minimo o massimo della porzione da ordinare. Per questo motivo è preferibile scegliere come pivot un elemento a caso della porzione da ordinare. Semplici considerazioni probabilistiche mostrano che, se ciascun elemento della porzione da ordinare ha la stessa probabilità di essere scelto come pivot, il tempo di esecuzione atteso è asintoticamente lo stesso del caso migliore, cioè $\Theta(n \log n)$.

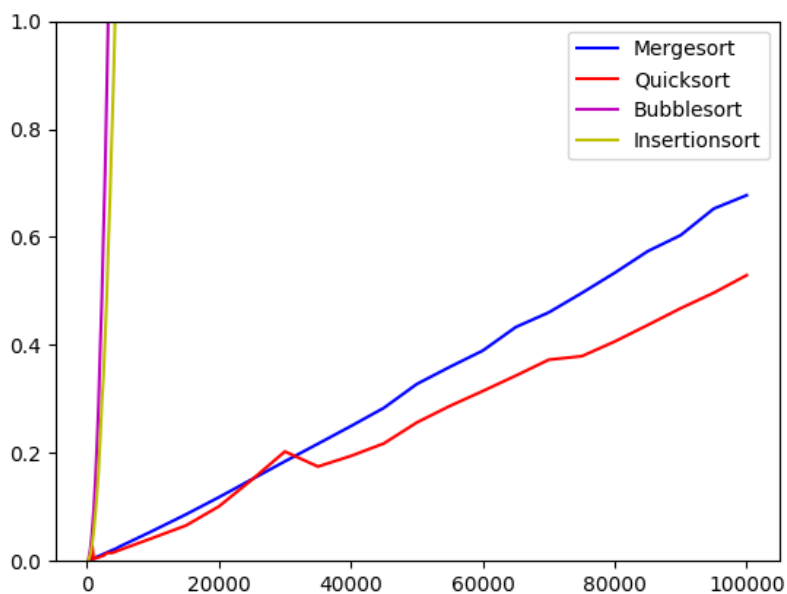
Memoria utilizzata I metodi invocati dal Quicksort non richiedono la allocazione di alcuna lista oltre alla lista da ordinare, quindi ciascuno dei metodi attivati necessita solo di una quantità costante di memoria. Il numero di chiamate ricorsive è, sia nel caso migliore che nel caso peggiore, lineare nel numero di elementi della lista. Tuttavia il numero massimo di record di attivazione sullo stack è pari alla profondità (i.e. numero di livelli) dell'albero che rappresenta lo schema di ricorsione. Nel caso peggiore, ovvero quello di albero molto sbilanciato, la profondità è $\Theta(n)$ così come lo è la quantità di memoria aggiuntiva. Nel caso tipico l'albero di ricorsione ha $\Theta(\log n)$ livelli, e quindi la memoria aggiuntiva è anch'essa $\Theta(\log n)$.

Stabilità del Quicksort L'implementazione descritta del metodo `partition` non assicura la stabilità dell'algoritmo,

È possibile realizzare una versione stabile dell'algoritmo se la separazione tra elementi piccoli e grandi viene eseguita creando due nuove sequenze. Questo non comporta un aggravio del tempo di calcolo asintotico, ma richiede globalmente l'utilizzo di memoria aggiuntiva $\Theta(n \log n)$.

10.5 Confronto sperimentale con Mergesort

Vediamo come si comportano Quicksort e Mergesort con input casuali, e come entrambi siano notevolmente più veloci di Bubblesort e Insertionsort.



Notate dal grafico che il tempo di esecuzione di Quicksort ha un andamento un po' più irregolare, probabilmente dovuto alla scelta casuale del pivot che influisce in maniera non completamente prevedibile sui tempi di esecuzione. Nel grafico si vede anche che Mergesort e Quicksort hanno tempi di esecuzione abbastanza simili, ma con un vantaggio significativo per Quicksort. Ricordiamo che l'implementazione di Mergesort in questi appunti usa delle liste aggiuntive su cui i dati vengono copiati ad ogni fase. Con un po' più di impegno è possibile scrivere una versione di Mergesort

che, rispetto alla nostra versione, esegue circa metà delle copie e che quindi può risultare più competitiva con Quicksort.

Capitolo 11

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

$$T(n) = \begin{cases} C & \text{when } n \leq c_0 \\ \sum_i a_i T(g_i(n)) + f(n) & \text{when } n > c_0 \end{cases} \quad (11.1)$$

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.

¹In 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.

11.1 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.

11.2 Metodo iterativo e alberi di ricorsione

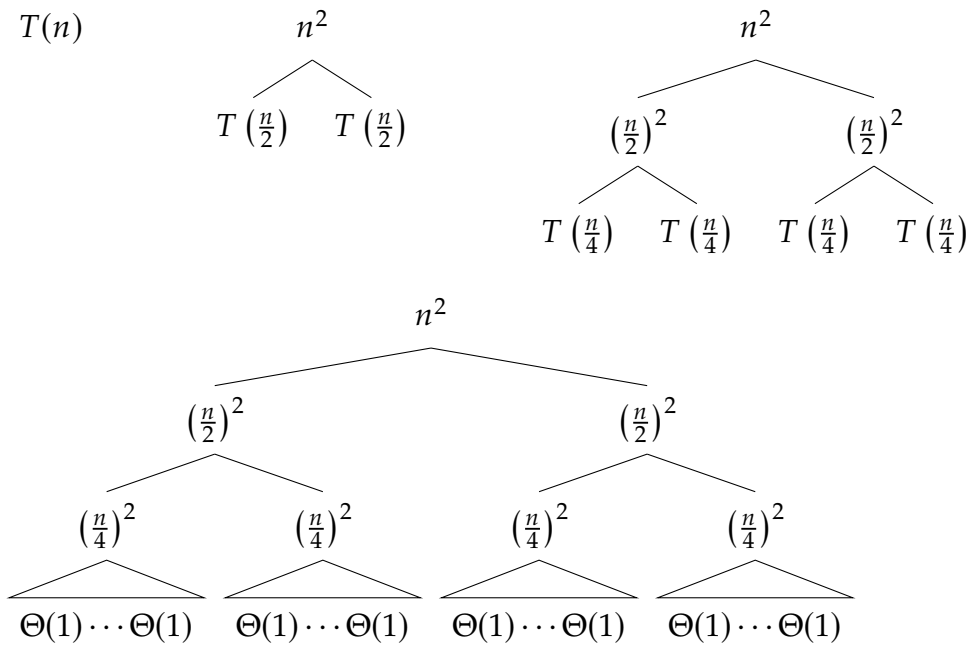
Nota: il contenuto di questa parte degli appunti può essere approfondito nel Paragrafo 4.4 del libro di testo *Introduzione agli Algoritmi e strutture dati* di Cormen, Leiserson, Rivest e Stein.

È 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 $T(n) = 3T(\lfloor n/4 \rfloor) + n$

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

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 $n^2/4, \dots$
- L'ultimo ha $\Theta(n)$ operazioni.

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

11.3 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 11.1. *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 $a f(n/b) < c f(n)$ 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 2, 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 in questi appunti solo per completezza.*

Abbiamo già osservato che il costo totale dell'algoritmo è

$$T(n) = \Theta(n^{\log_b(a)}) + \sum_{j=0}^{\log_b(n)-1} a^j f(n/b^j) \quad (11.2)$$

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

$$a^i \cdot \left(\frac{n}{b^i}\right)^{\log_b(a)} = a^i \cdot \left(\frac{1}{b}\right)^{i \log_b(a)} \cdot n^{\log_b(a)} = a^i \cdot \left(\frac{1}{a}\right)^i \cdot n^{\log_b(a)} = n^{\log_b(a)} \quad (11.3)$$

Procediamo col dimostrare i tre casi

Caso 1: chiaramente l'equazione (11.2) implica che $T(n) = \Omega(n^{\log_b(a)})$, quindi per concludere il caso 1 è sufficiente dimostrare che la sommatoria in (11.2) sia $O(n^{\log_b(a)})$.

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

$$\sum_{j=0}^{\log_b(n)-1} O\left(a^j \cdot \left(\frac{n}{b^j}\right)^{\log_b(a)-\epsilon}\right). \quad (11.4)$$

Portiamo la sommatoria dentro l'operatore O ,

$$O\left(\sum_{j=0}^{\log_b(n)-1} a^j \cdot \left(\frac{n}{b^j}\right)^{\log_b(a)-\epsilon}\right) \quad (11.5)$$

ovvero

$$O\left(n^{\log_b(a)-\epsilon} \cdot \sum_{j=0}^{\log_b(n)-1} a^j \cdot \left(\frac{1}{b^j}\right)^{\log_b(a)-\epsilon}\right) \quad (11.6)$$

ovvero

$$O\left(n^{\log_b(a)-\epsilon} \cdot \sum_{j=0}^{\log_b(n)-1} \left(\frac{ab^\epsilon}{b^{\log_b(a)}}\right)^j\right). \quad (11.7)$$

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

$$O\left(n^{\log_b(a)-\epsilon} \cdot \sum_{j=0}^{\log_b(n)-1} (b^\epsilon)^j\right) \quad (11.8)$$

ovvero²

$$O\left(n^{\log_b(a)-\epsilon} \cdot \frac{b^{\epsilon \log_b(n)} - 1}{b^\epsilon - 1}\right). \quad (11.9)$$

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

$$O\left(n^{\log_b(a)}\right). \quad (11.10)$$

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

$$Cn^{\log_b(a)} \leq f(n) \leq Dn^{\log_b(a)}. \quad (11.11)$$

Applichiamo le equazioni (11.3) alle disuguaglianze in (11.11) per ottenere che per ogni $i \geq 0$ e ogni $n > n_0$ vale

$$Cn^{\log_b(a)} \leq a^i f\left(\frac{n}{b^i}\right) \leq Dn^{\log_b(a)} \quad (11.12)$$

Dunque applicando queste limitazioni inferiori e superiori a tutti i termini della sommatoria in (11.2) otteniamo

$$T(n) = \Theta(n^{\log_b(a)}) + \sum_{j=0}^{\log_b(n)-1} \Theta(n^{\log_b(a)}) = \Theta(n^{\log_b(a)} \log n). \quad (11.13)$$

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

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

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 (11.2) 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

$$a^i f(n/b^i) < c^i f(n), \quad (11.14)$$

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 (11.2) otteniamo

$$T(n) \leq \Theta(n^{\log_b(a)}) + \sum_{j=0}^{\log_b(n)-t} c^j f(n) + \sum_{j=\log_b(n)-t+1}^{\log_b(n)-1} a^j f(n/b^j). \quad (11.15)$$

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 (11.15) alla fine vale al massimo

$$T(n) \leq \Theta(n^{\log_b(a)}) + \sum_{j=0}^{\infty} c^j f(n) \leq \Theta(n^{\log_b(a)}) + \frac{1}{1-c} f(n) = O(f(n)). \quad (11.16)$$

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

Capitolo 12

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.

12.1 Esempio: Counting Sort

Il counting 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, \dots, 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
    counters=[0]*(b-a+1)
    for x in seq:
        counters[x-a] = counters[x-a] + 1
```

```

# costruzione dell'output                                     11
output=[]                                                  12
value = a                                                  13
for value in range(a,b+1):                                  14
    repetitions = counters[value-a]                         15
    output.extend( [value]*repetitions)                     16
return output                                              17
print(countingsort1([7,6,3,5,7,9,3,2,3,5,6,7,8,8,9,5,4,3,2,2,3,4,6,8,8])) 18

```

```
[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.

12.2 Dati contestuali

Negli algoritmi di ordinamento per confronto 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):                                     1
    if len(seq)==0:                                       2
        return                                           3
    # calcolo minimo e massimo: n operazioni              4
    a = seq[0][0]                                         5
    b = seq[0][0]                                         6
    for chiave_e_dato in seq:                              7
        if chiave_e_dato[0] < a:                          8
            a = chiave_e_dato[0]                          9
        elif chiave_e_dato[0] > b:                        10
            b = chiave_e_dato[0]                          11
    # creazione dei contatori                              12

```

```
counter=[0]*(b-a+1) 13
for chiave_e_dato in seq: 14
    k = chiave_e_dato[0] 15
    counter[k-a] += 1 16
17
# posizioni finali di memorizzazione 18
posizioni=[0]*(b-a+1) 19
for i in range(1,len(counter)): 20
    posizioni[i]=posizioni[i-1]+counter[i-1] 21
22
# costruzione dell'output 23
for chiave_e_dato in seq[:]: 24
    k,data=chiave_e_dato 25
    seq[posizioni[k-a]]=(k,data) 26
    posizioni[k-a] += 1 27
28
sequenza=[(3,"paul"),(4,"ringo"),(1,"george"),(1,"pete"),(3,"stuart"),(4,"john")] 29
countingsort2(sequenza) 30
print(sequenza) 31
```

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


Capitolo 13

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, con $0 \leq i < j < \text{len}(S)$, tali che il valore in $S[i]$ si trova dopo il valore in $S[j]$ una volta finito l'ordinamento. Un ordinamento stabile minimizza il numero di inversioni.

Molti degli ordinamenti che abbiamo visto fino ad ora sono stabili. Per esempio nel caso di ordinamenti Insertionsort, Bubblesort e Mergesort è capitato di fare operazioni del tipo

```
if S[i] > S[j]:
    operazioni che causano un'inversione tra S[i] e S[j]
else:
    operazioni che non 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 del blocco if-else sarebbe cambiato solo nei casi in cui $S[i]$ fosse stato uguale a $S[j]$. L'ordinamento sarebbe stato comunque valido ma non sarebbe più stato un ordinamento stabile.

13.1 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 per Traccia.

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 verranno 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, $key_1, key_2, \dots, key_N$, in maniera gerarchica, ordinando prima rispetto key_N e poi andando su fino a key_1 . Modifichiamo `countingsort` 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
    # calcolo minimo e massimo: n operazioni
    a, b = key(seq[0]), key(seq[-1])
    for x in seq:
        if key(x) < a:
            a = key(x)
        elif key(x) > b:
            b = key(x)
    # 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
```

24

```
posizioni[key(x)-a] = posizioni[key(x)-a] + 1
```

25

Capitolo 14

Ordinare sequenze di interi grandi con 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]
```

14.1 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 di complessità $\Theta(n^2)$. Nel grafico vediamo i tempi di esecuzione degli algoritmi:

- Mergesort e Quicksort;
- Countingsort con intervallo $[0,10000]$ e $[0,1000000]$;
- Radixsort con 4 chiavi da 8 bit e con 2 chiavi da 16 bit;
- Bubblesort e Insertionsort (eseguiti solo per input piccoli).

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 dominio di 1000000 elementi.

