anno 2014-15

Introduzione all'Algoritmica per i Licei.

8 – Il quicksort.

Elio Giovannetti
Dipartimento di Informatica
Università di Torino

versione 01/03/2015

Il quicksort: l'algoritmo di ordinamento più usato nel mondo.

Tony Hoare: inventore del Quicksort.

(fonte: Wikipedia)

Tony Hoare's interest in computing was awakened in the early fifties, when he studied philosophy (together with Latin and Greek) at Oxford University, under the tutelage of John Lucas. He was fascinated by the power of mathematical logic as an explanation of the apparent certainty of mathematical truth.

During his National Service (1956-1958), he studied Russian in the Royal Navy. Then he took a qualification in statistics (and incidentally) a course in programming given by Leslie Fox.

In 1959, as a graduate student at Moscow State University, he studied the machine translation of languages (together with probability theory, in the school of Kolmogorov).

To assist in efficient look-up of words in a dictionary, he discovered the well-known sorting algorithm Quicksort.

...

L'algoritmo

Per ordinare (in ordine crescente) una sequenza S (ad es. di interi) si può procedere così:

- si prende un elemento a caso p della sequenza S (ad esempio il primo), e lo si toglie dalla sequenza;
- si percorre tutta la sequenza S' così ottenuta, mettendo da una parte tutti gli elementi minori di p, e dall'altra tutti quelli maggiori di p o uguali a p, ottenendo in tal modo due sottosequenze $u_1, u_2, \ldots, u_k$ e $v_1, v_2, \ldots, v_h$;
- si ri-inserisce p fra le due sottosequenze, ottenendo così una sequenza: $u_1, u_2, \ldots, u_k, p, v_1, v_2, \ldots, v_h$ tale che:
  - tutti gli $u_1, u_2, \ldots, u_k$ sono minori di p;
  - tutti i $v_1, v_2, \ldots, v_h$ sono maggiori o uguali a p;
- se ora si ordinano separatamente $u_1, u_2, \ldots, u_k$ e $v_1, v_2, \ldots, v_h$, l'intera sequenza risulterà ordinata.

Il quicksort (continuazione)

- Come si fa a ordinare $u_1, u_2, \ldots, u_k$? e $v_1, v_2, \ldots, v_h$?

  Semplicemente riapplicando l'algoritmo prima a $u_1, u_2, \ldots, u_k$ e poi a $v_1, v_2, \ldots, v_h$!
- Si riapplica quindi l'algoritmo a sottosequenze via via più corte: quando ci si ferma ?
- Quando si arriva a una sottosequenza di lunghezza 1, cioè costituita da un solo elemento, oppure di lunghezza 0, cioè vuota: in tal caso, per risolvere il problema, non c'è da fare nulla, perché il problema è già risolto ! una sequenza vuota o di un solo elemento è già banalmente ordinata !

Quicksort: schema astratto dell'algoritmo

```
void qsort(sequenza S) {
  if (lunghezza di S  >  1) {
    togli un elemento p da S (ad esempio il primo);
    ripartisci tutti gli altri elementi di S in due parti:
    una sequenza S1 contenente tutti gli elem. < p;
    una sequenza S2 contenente tutti gli elem. ≥ p;
    forma la sequenza S1 p S2;
    qsort(S1);
    qsort(S2);
  }
}
```

l'elemento p è detto pivot o perno della partizione.

Quicksort: una versione forse più facile da ricordare.

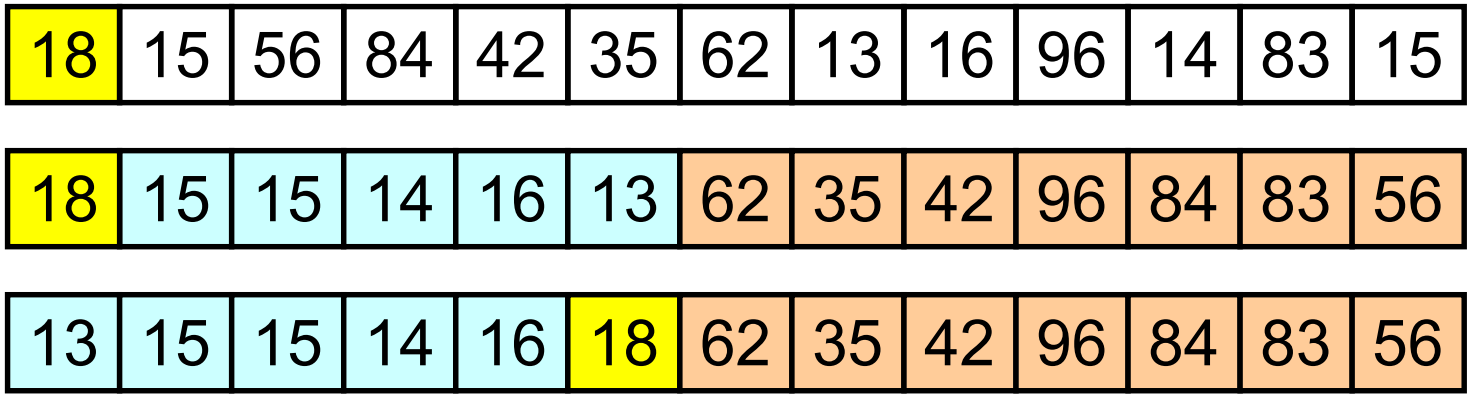
```
void qsort(sequenza S) {
  if (lunghezza di S  <=  1) return;
  togli un elemento p da S (ad esempio il primo);
  ripartisci tutti gli altri elementi di S in due parti:
  una sequenza S1 contenente tutti gli elem. < p;
  una sequenza S2 contenente tutti gli elem. ≥ p;
  forma la sequenza S1 p S2;
  qsort(S1);
  qsort(S2);
}
```

l'elemento p è detto pivot o perno della partizione.

Abbiamo scritto esplicitamente nella prima riga la condizione di terminazione e l'uscita nel caso in cui la condizione è verificata.

Esempio di esecuzione

Scelgo come pivot il primo elemento della sezione (18), ed effettuo il partizionamento (qui non interessa come viene fatto):



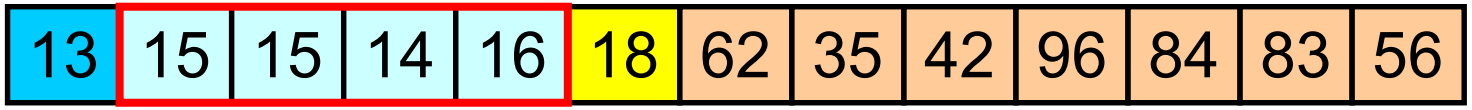
Ora passo a ordinare la prima sottosequenza: scelgo come pivot di nuovo il primo elemento (13), ed effettuo il partizionamento di tale sottosequenza:

ma rispetto al pivot 13 tutti gli elementi della sezione rimangono dalla stessa parte (a destra).

Esempio di esecuzione su un array

Passo quindi a ordinare la sottosequenza a sinistra di 13: ma è vuota, non devo fare nulla !

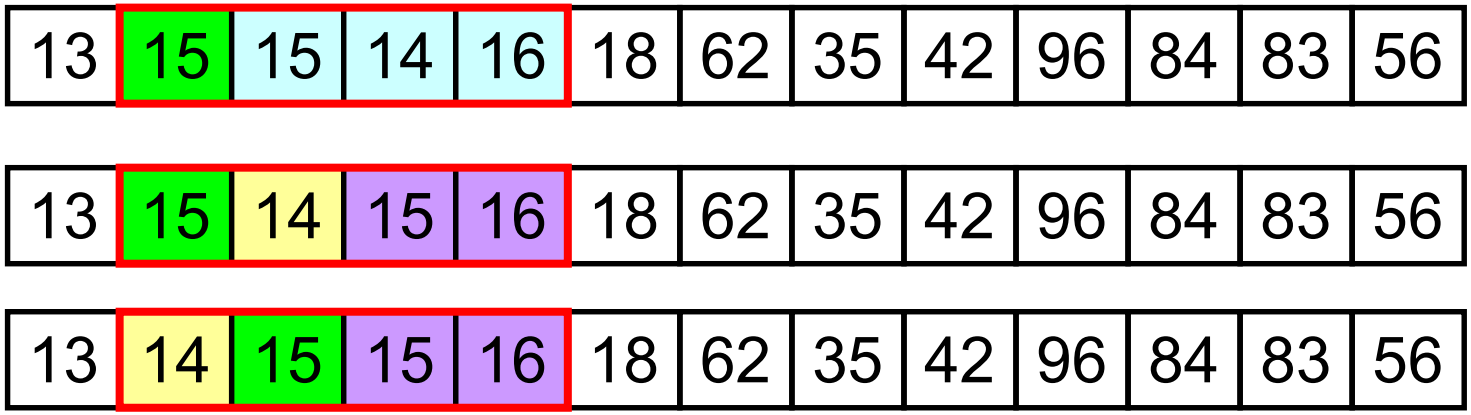
Devo allora ordinare la sottosequenza destra (della sezione di sinistra della sequenza iniziale):



Riapplico lo stesso metodo (vedi slide successiva).

Esempio di esecuzione (continua)

Scelgo come pivot il primo elemento della sottosezione, 15, e ripartisco rispetto ad esso



La parte a sinistra del pivot consiste di un solo elemento, 14, quindi non devo fare nulla. Ordino allora la parte a destra del pivot, costituita da due elementi. Scelgo il primo, 15, come pivot, ecc.

(continuazione)

Così, dopo pochi altri passaggi, la prima metà dell'array è finalmente ordinata. Passo allora alla seconda metà.
Prendo come pivot il primo elemento di tale parte, 62, ed effettuo la partizione rispetto ad esso.
Poi considero la prima delle due parti così ottenute, ecc.

13	14	15	15	16	18	62	35	42	96	84	83	56
13	14	15	15	16	18	62	35	42	56	84	83	96
13	14	15	15	16	18	56	35	42	62	84	83	96
13	14	15	15	16	18	56	35	42	62	84	83	96

e così via fino ad ottenere l'ordinamento di tutto l'array.

Realizzazione dell'algoritmo in Python

Usando la comprensione di lista l'implementazione è immediata, scrivendo una versione che senza modificare la lista di partenza dà come risultato la lista ordinata:

```
def qsort(lis):
  if lis == []: return []
  else:
    p = lis[0]
    minori = qsort([x for x in lis[1:] if x < p])
    maggiori = qsort([x for x in lis[1:] if x >= p])
    return minori + [p] + maggiori
```

Ricorda che l'operatore + fra liste ne fa la concatenazione.

Realizzazione dell'algoritmo in Python

Usando la comprensione di lista l'implementazione è immediata, scrivendo una versione che senza modificare la lista di partenza dà come risultato la lista ordinata; oppure:

```
def qsort(lis):
  n = len(lis)
  if n == 0 or n == 1: return lis
  else:
    p = lis[0]
    minori = qsort([x for x in lis[1:] if x < p])
    maggiori = qsort([x for x in lis[1:] if x >= p])
    return minori + [p] + maggiori
```

Ricorda che l'operatore + fra liste ne fa la concatenazione.

Realizzazione tradizionale "sul posto" dell'algoritmo

Vogliamo realizzare una procedura che non crei una nuova lista, ma modifichi la lista di partenza (ordinandola).

La procedura deve essere richiamata su porzioni di array, occorre quindi passare come argomenti gli indici di inizio e di fine della porzione; chiamiamo **qs** la procedura ricorsiva.

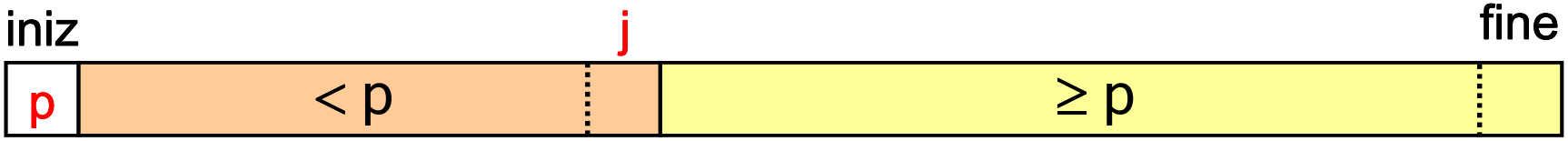
```
def qs(list, iniz, fine):
  ...
```

Per uniformità con gli altri algoritmi di ordinamento, si definirà poi come procedura che deve essere invocata dall'utente una procedura qsort (con gli stessi parametri di isort e ssort) che si limita ad eseguire l'invocazione iniziale della procedura "vera":

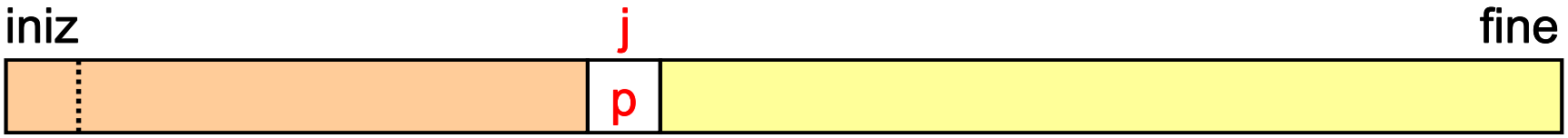
```
def qsort(list):
  qs(list, 0, len(list)-1)
```

Realizzazione concreta dell'algoritmo (array di double)

se la porzione da ordinare ha lunghezza ≥ 2:
scegli come pivot il primo elemento: double p = a[iniz];
partiziona la porzione di array a[iniz+1 .. fine] in due parti:
a destra gli elementi $< p$, a sinistra gli elementi $\geq p$;
sia j l'indice dell'ultimo elemento $< p$;



inserisci il pivot fra le due parti, "spostando indietro di uno" la parte $< p$, cioè: scambia a[ini] con a[j]



ordina con qsort la porzione a[iniz .. j-1];
ordina con qsort la porzione a[j+1 .. fine]

Traduzione in Python dello schema

```
def qs(list, iniz, fine):
  if(fine >= iniz): return
  p = list[iniz]
```

partiziona `list[iniz+1 … fin]` in due parti: < p e >= p

sia j l'indice dell'ultimo elemento < p

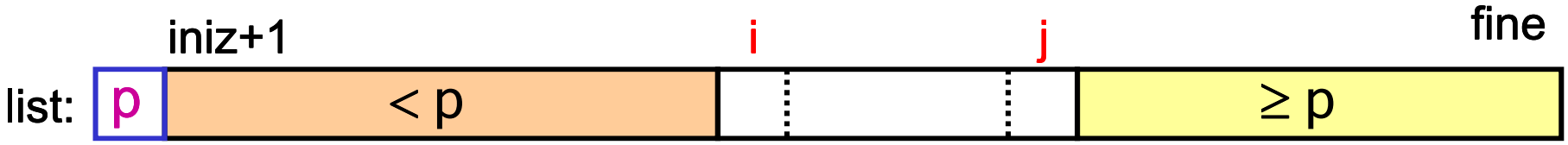
scambia `list[iniz]` con `list[j]`

```
  qs(list, iniz, j-1)     // ordina la prima parte
  qs(list, j+1, fine)     // ordina la seconda parte
```

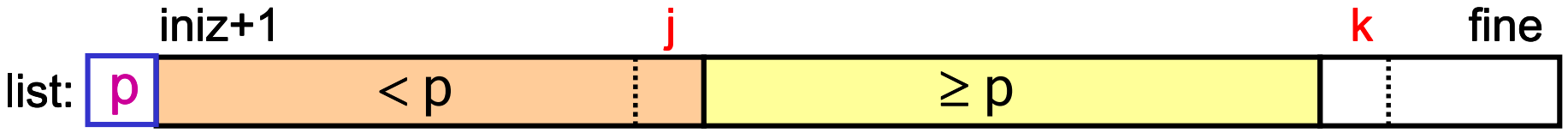
La partizione

Si deve percorrere la porzione tramite un ciclo, esaminando gli elementi uno per uno e mettendo a sinistra quelli < p e a destra quelli ≥ p. Ciò può essere fatto in diversi modi.

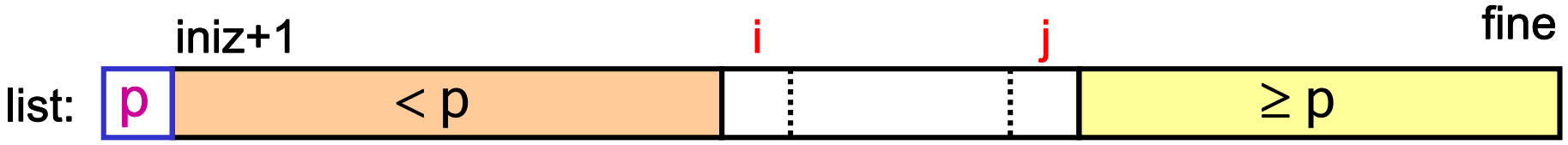
Modo 1).
Al passo generico, la parte ancora da esaminare è in mezzo:



Modo 2).
Al passo generico, la parte ancora da esaminare è in fondo:



Modo 1: al passo generico.



Corpo del ciclo:

```
if(a[i] < p): i = i+1
else:
```

scambia `list[i]` con `list[j]`

```
j = j-1
```

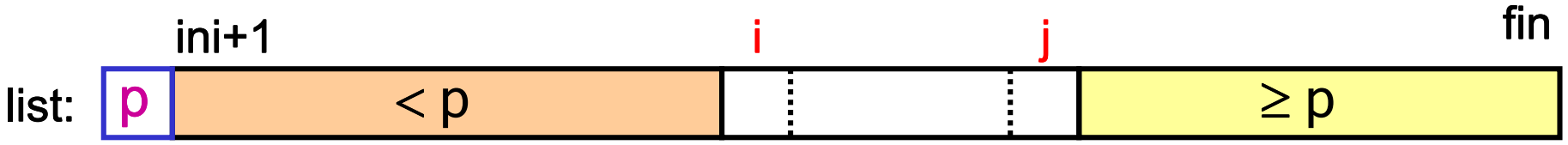
Inizializzazione: la parte da esaminare è l'intera porzione

```
i = iniz+1
j = fine
```

Test: se la parte ancora da esaminare è non vuota, si continua

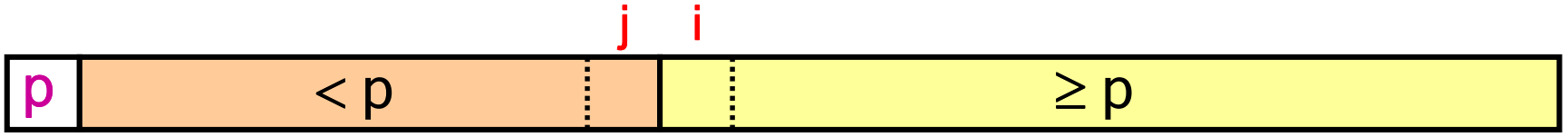
```
while i <= j:
```

Modo 1



Qual è la situazione all'uscita dal ciclo `while i <= j:` ?

Ovviamente `i <= j` è falso; quindi è `i > j`;
anzi è `i == j+1` (perché?): gli indici `i` e `j` si sono incrociati,
e `j` è proprio l'indice dell'ultimo dei minori!



Dove va inserito il pivot?

Come abbiamo detto, va inserito al posto dell'ultimo dei minori, quindi in a[j].

La procedura completa

```
def qs(list, iniz, int fine):
  if fine <= iniz: return
  pivot = list[iniz]
  i = iniz+1
  j = fine
  while i <= j:
    if list[i] <= pivot: i = i+1
    else:
      list[i], list[j] = list[j], list[i]
      j = j-1
  list[iniz] = list[j]
  list[j] = pivot
  qs(list, iniz, j-1)
  qs(list, j+1, fine)
```

scambia col pivot

rimettendo il pivot in mezzo

La procedura "esterna".

```
def qsort(list):
  qs(list, 0, len(list)-1)
```

Modo 2

Esercizio: effettuare la partizione nel modo 2.

Tempo di calcolo

Per effettuare la partizione di una porzione di lista ci vuole un numero di passi proporzionale alla lunghezza di quella porzione.

Se gli elementi da ordinare hanno valori disposti a caso, mediamente ad ogni partizione ci saranno all'incirca tanti elementi <= al pivot quanti elementi > del pivot.

Quindi ogni volta l'algoritmo viene richiamato su due porzioni di lunghezza circa metà.

Immaginiamo che ogni volta le due chiamate ricorsive vengano effettuate in parallelo da due nuovi esecutori e contiamo i passi fatti globalmente, dove n sia la lunghezza della lista.

1ª volta: partizione di tutta la lista: n passi

2ª volta: 2 partizioni di circa ½ lista: n/2 + n/2 = n passi

3ª volta: 4 partizioni di circa ¼ lista: 4 (n/4) = n passi

ecc.

Tempo di calcolo

Si hanno quindi, in tutto, $n \cdot k$ passi, dove k è il numero di volte che occorre dividere per due le porzioni di array per arrivare a porzioni di lunghezza 1 o 0.

Cioè, come nella ricerca binaria, $k = \log_2 n$.

Il numero totale di passi è quindi

$$n \cdot \log_2 n$$

La funzione $n \cdot \log_2 n$, al crescere di n, cresce pochissimo di più della funzione lineare, cioè in modo poco più che proporzionale a n.

Gli algoritmi il cui tempo di calcolo, al crescere della dimensione n dell'input, cresce come $n \cdot \log n$ si dicono

pseudo-lineari o quasi-lineari.

Tempo di calcolo: esempio

Consideriamo un array di lunghezza 1024.

1ª volta: partizione di tutto l'array.
2ª volta: 2 partizioni ciascuna su una porzione di ~512 elementi
3ª volta: 4 partizioni ciascuna su una porzione di ~256 elementi
4ª volta: 8 partizioni ciascuna su una porzione di ~128 elementi
…
10ª volta: 512 partizioni ciascuna su una porzione di ~2 elem.

Quindi k è circa = 10.

Il numero totale di passi è quindi circa $1024 \cdot 10$, che è molto minore del numero di passi 1024^2 richiesto dai due algoritmi di ordinamento visti in precedenza!

Ma si può essere sfortunati ...

Che cosa succede se si esegue l'algoritmo su un array che è già ordinato o quasi ordinato?

Ad ogni partizione le due parti risultano molto sbilanciate in lunghezza: nel caso peggiore si ha:
1ª volta: partizione dell'array in una parte di lunghezza n-1 ed in una parte vuota (lunghezza 0);
2ª volta: partizione della parte di lunghezza n-1 in una parte di lunghezza n-2 ed in una parte vuota;
3ª volta: partizione della parte di lunghezza n-2 in una parte di lunghezza n-3 ed in una parte vuota;
ecc.

Quanti sono in tutto i passi in un caso sfortunato come questo?

$n + (n-1) + (n-2) + \ldots + 2 = n(n+1)/2 - 1 = (n^2 + n)/2 - 1$

L'algoritmo è quadratico!

Tuttavia ...

... si possono rendere tali casi sfortunati altamente improbabili:

ad esempio, invece di scegliere ogni volta come pivot il primo elemento della porzione, se ne sceglie uno a caso (utilizzando un generatore di numeri pseudo-casuali per sceglierne l'indice).

In questo modo la probabilità che in tutte le partizioni si abbia sempre uno sbilanciamento è molto bassa.

Osserva infatti che anche se ogni tanto si ha una partizione che risulta sbilanciata, ciò ha poca influenza sul numero di passi totale: perché l'algoritmo diventi quadratico occorre che quasi tutte le partizioni siano sbilanciate (ciò può essere dimostrato rigorosamente, con una dimostrazione matematica non troppo facile).

Sono poi possibili molti miglioramenti nel modo di eseguire la partizione, ecc.

Quicksort con pivot scelto a caso

```
import random
```

nella funzione esterna si inizializza il generatore di numeri pseudocasuali:

```
def qsort(list):
  random.seed()
  qs(list, 0, len(list)-1)
```

Quicksort con pivot scelto a caso

```
def qs(list, iniz, int fine):
  if fine <= iniz: return
  iPivot = random.randint(iniz, fine)
  pivot = list[iPivot]
  list[iPivot] = list[iniz]
  i = iniz+1; j = fine
  while i <= j:
    if list[i] <= pivot: i = i+1
    else:
      list[i], list[j] = list[j], list[i]
      j = j-1
  list[iniz] = list[j]
  list[j] = pivot
  qs(list, iniz, j-1)
  qs(list, j+1, fine)
```

scambia col pivot

rimettendo il pivot in mezzo

Conclusione

Il quicksort, se implementato in modo opportuno, risulta essere in quasi tutte le situazioni l'algoritmo più veloce di tutti!

Ricorda: è un algoritmo inventato nel 1959 da un allora studente britannico di dottorato all'Università di Mosca, che è poi diventato uno dei grandi dell'informatica del XX secolo, nominato infine baronetto per i suoi meriti scientifici:

sir Charles Antony Richard Hoare,

più noto come

Tony Hoare.

Uno degli ultimi raffinamenti dell'algoritmo è quello realizzato nel 2009 dallo sviluppatore software Vladimir Yaroslavskiy della Sun (poi acquisita dalla Oracle) in California.

Yaroslavskiy si è laureato all' Università di San Pietroburgo e verosimilmente è ancora cittadino russo.

Appendice matematica: i logaritmi.

Abbiamo visto, sia nella ricerca binaria che nel quicksort, l'importanza della funzione inversa dell'esponenziale.

Ad es. nel caso della ricerca binaria in una sequenza di n elementi, il numero di passi è all'incirca quel numero k tale che

$$2^k = n$$

Tale numero si chiama logaritmo in base 2 di n e si scrive:

$$\log_2 n$$

Esempio:

$2^2 = 4$ quindi $\log_2 4 = 2$

$2^3 = 8$ quindi $\log_2 8 = 3$

$2^4 = 16$ quindi $\log_2 16 = 4$

$2^5 = 32$ quindi $\log_2 32 = 5$ ….

I logaritmi

Omettendo, per brevità, di indicare la base che in questi esempi è sempre 2, possiamo continuare:

$2^{10} = 1024$ quindi $\log 1024 = 10$

$2^{20} \cong$ un milione quindi $\log$ (un milione) $\cong 20$

$2^{30} \cong$ un miliardo quindi $\log$ (un miliardo) $\cong 30$

Come si vede, al crescere di n la funzione log n cresce in modo estremamente lento!

Per questo un algoritmo il cui tempo di calcolo, al crescere della dimensione n dell'input, cresce come il logaritmo di n, è un algoritmo molto efficiente!

Gli algoritmi di ordinamento elementari (ssort e isort) hanno un tempo medio di calcolo che cresce come n^2;

il qsort ha un tempo medio di calcolo che cresce come $n \cdot \log n$:

molto più efficiente!!

Algoritmi e logaritmi

I logaritmi sono molto importanti nello studio degli algoritmi!

"logaritmo" è anagramma di "algoritmo" !

Entrambe le parole vengono dal nome di

al-Khvarismi الخوارزمي