Efficienza algoritmi

- indipendente dal tipo di computer e dati
- stima del numero di operazioni necessarie
  - es. stampa di una lista linkata
  - es. un doppio loop
- stima caso peggiore – caso medio

Esempi

- stampa lista linkata

```
lp = head;
while(lp){
   cout << “Contenuto lista :”<<lp->data;
   lp = lp->Next;
}
```

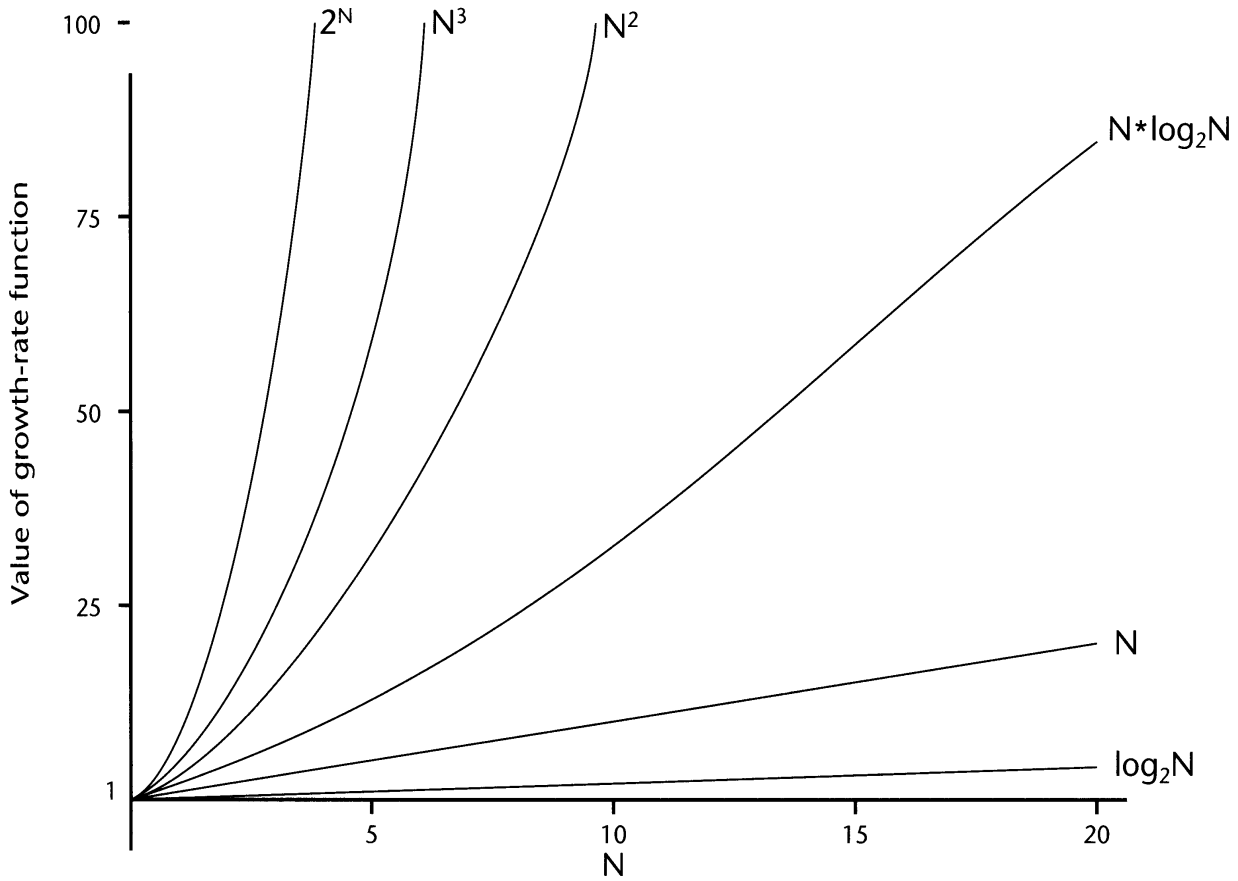
(N+1)a + Nc + Nw ≈ o(N)

- doppio loop

```
for(i=0,i<N,i++) {
  for (j=i+1,j<N;j++)
    {do something}
}
```

N(N+1)/2 ≈ o(N^2)

Ordine di grandezza



Valutazione caso peggiore/medio

- ricerca in una lista linkata

```
key = dato_da_cercare;
lp = head;
while(lp->data != key){
    lp = lp->Next;
}
```

  - caso peggiore -> **(N+1)a + Nc ≈** o(N)
  - caso medio -> **(N+1)a/2 + Nc/2 ≈** o(N)

Altri parametri da considerare

- velocità dipende dalla grandezza del problema.
  - se N=25 o(N) oppure o(logN) sono simili
- occupazione memoria
  - l'estrazione dell'elemento I-esimo in un vettore è sempre di ordine o(1), mentre in una lista linkata è di ordine o(N)
  - l'allocazione della memoria è migliore per le liste linkate

Efficienza di algoritmi di ricerca

- ricerca sequenziale di un vettore
  - caso peggiore (N) e caso medio(N/2) ≈ o(N)
- ricerca binaria (di un vettore ordinato)
    - se $N=2^k$
    - confronta l'elemento mediano di un vettore N,
    - confronta l'elemento mediano di un vettore N/2,
    - confronto l'elemento mediano di un vettore $N/2^k$
    - …
  - caso peggiore: tutte k le divisioni -> $o(\log_2 N)$
  - se N= 1.000.000 -> $\log_2(1.000.000) \approx 19$
    MA il vettore deve essere ordinato

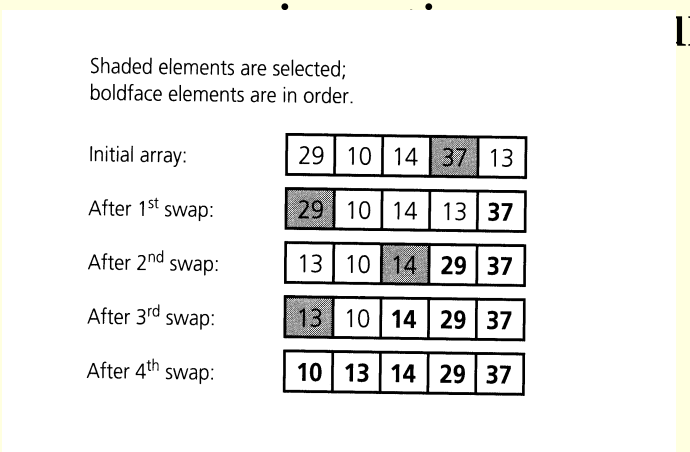
Algoritmi di ordinamento

- ordinamento interno (in main memory) o ordinamento esterno (su file)

- chiave di ordinamento (sort key)
  - nelle strutture posso avere diversi campi di dati
  - posso ordinare a seconda di una scelta su uno di questi campi

Selection Sort

- selection sort (esempio dati in un vettore)
  - si cerca l'elemento maggiore (minore) e lo si posizione nell'ultimo (primo) posto del vettore scambiando di posizione

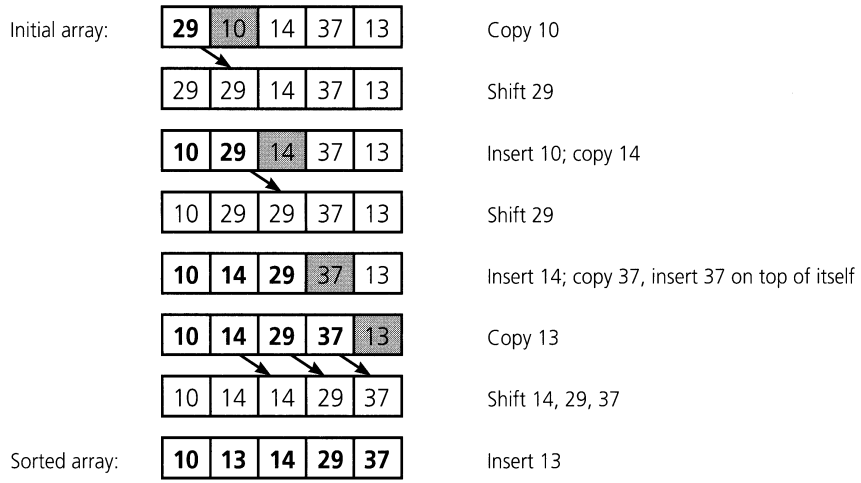


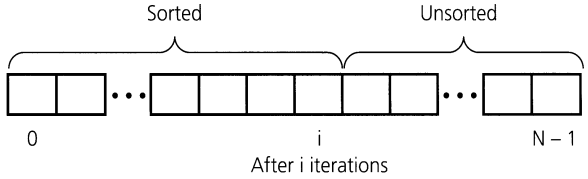
in vettore di ordine N-1

N(N-1)/2 + 3(N-1) ≈ o(N^2)

Insertion Sort

- insertion sort (inserimento ordinato)
  - meglio in una lista linkata, ma anche in un vettore (overhead degli spostamenti dei valori nel vettore)

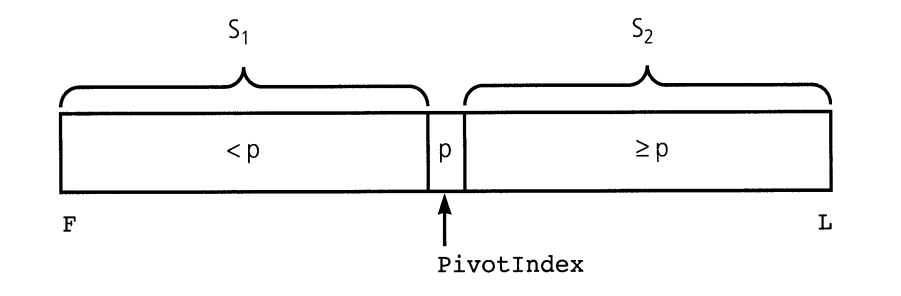




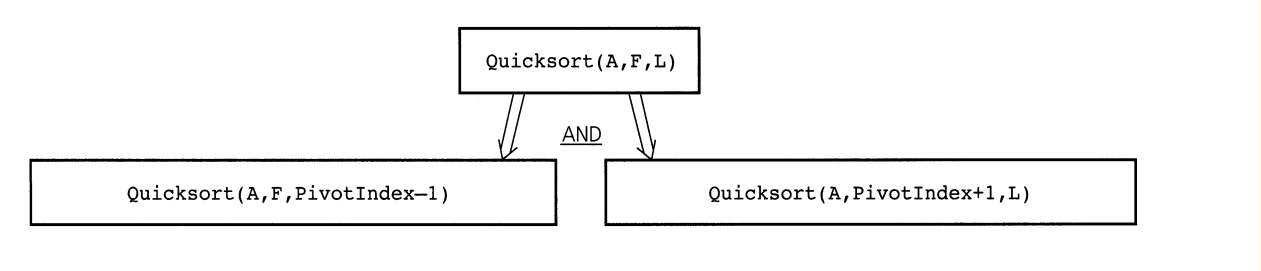
$o(N^2)$ nel caso peggiore

Quicksort

- algoritmo ricorsivo -> *divide et impera*
  - partizionare il vettore rispetto ad un *pivot*



  - ripetere per i due vettori creati fino a quando si arriva al vettore di dimensione 1



Valutazione quicksort

- caso peggiore -> vettore già ordinato

$o(N^2)$ nel caso peggiore

se N= 1.000.000 -> $N^2 \approx 1 * 10^{12}$

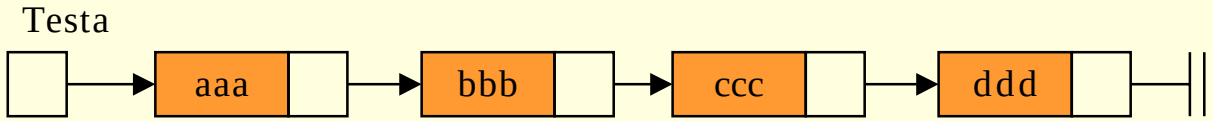
- caso medio ->

$o(N \log_2 N)$ nel caso medio

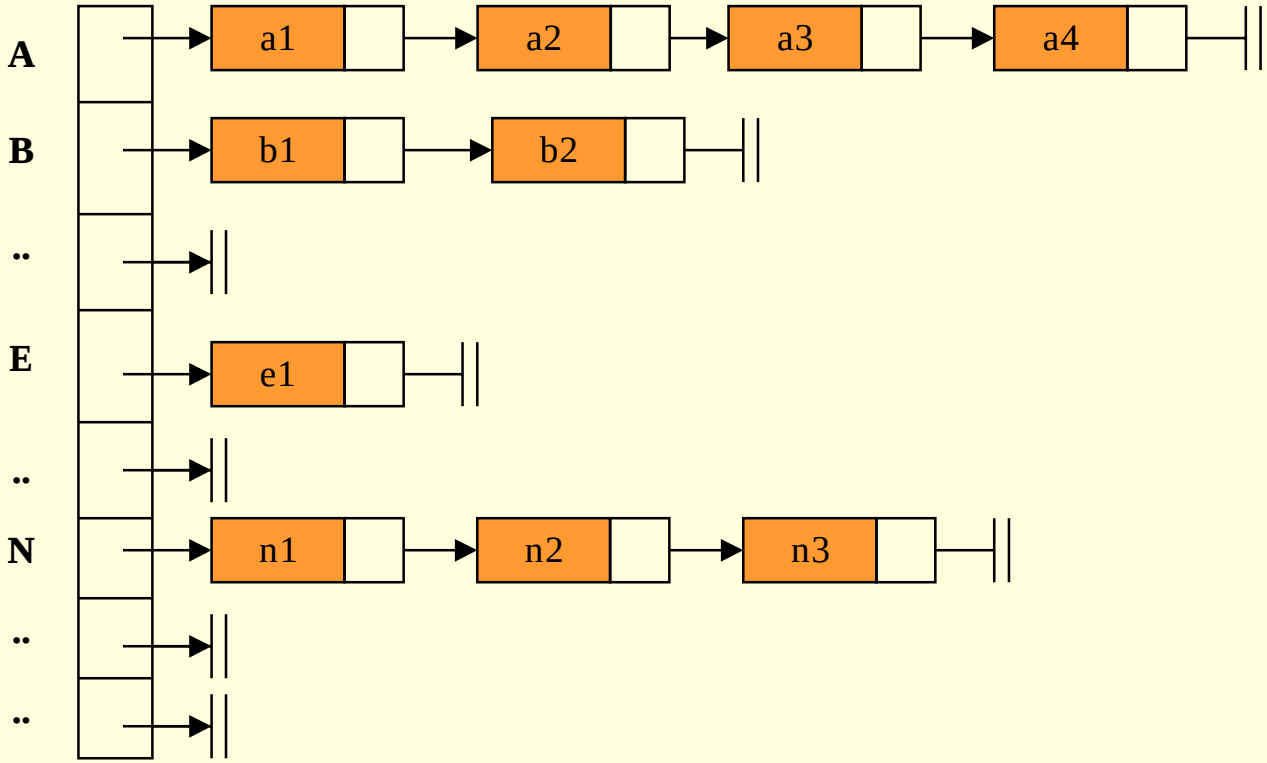
se N= 1.000.000 -> $N\log_2(1.000.000) \approx 19 * 10^6$

Dizionari

- “dizionari”: strutture dati che supportano i metodi di *inserimento, ricerca e cancellazione*
  - *Es. dizionario con lista linkata ordinata*



Un “dizionario” migliore..



Un “dizionario” migliore..

- Generalizzazione: Hash Table: tabelle “tritatutto” per ogni tipo di dato (alfanumerico, numerico…)
- Indice_tabella = funzione (key)
- Diversi metodi, a secondo del tipo di key

  Division Method: un indice compreso tra 0 e SIZE viene creato dividendo la key(numerica) per SIZE e prendendo il Resto Aritmetico (modulo-N)

```
int indice = key % SIZE
```

Con le stringhe ..

```
#define SIZE 256
typedef unsigned short int indexType;

indexType f(char * str) {
  indexType h = 0;
  while (*str) {
   h = h + *str;
   str++;
   }
  return h%SIZE;
}
```

```
h += *str++;
```

Hash table + Liste Linkate

- Hash Table di grandezza (SIZE) = 1
  -> Lista Linkata con N nodi
- Hash Table di grandezza = 2
  -> 2 liste linkate con N/2 nodi

…

- Hash Table di grandezza = 100
  -> 100 liste linkate con N/100 nodi