Una lezione (quasi) universitaria di Informatica.

Liceo Scientifico Statale "Galileo Ferraris".

Torino, 27 aprile 2007

Parte III

Elio Giovannetti
Dipartimento di Informatica
Università di Torino

Il problema dell'ordinamento

Data una sequenza di elementi, ordinarla rispetto ad una relazione d'ordine definita sugli elementi (di solito su un particolare componente di tali elementi);

ad es. ordinare una lista di studenti rispetto al numero di matricola (costituente la chiave identificativa dello studente), oppure rispetto al cognome, ecc.

Consideriamo per semplicità solo sequenze di numeri interi, rappresentate in memoria come *arrays* (cioè sequenze di celle contigue, ognuna delle quali è direttamente accessibile con una operazione elementare.

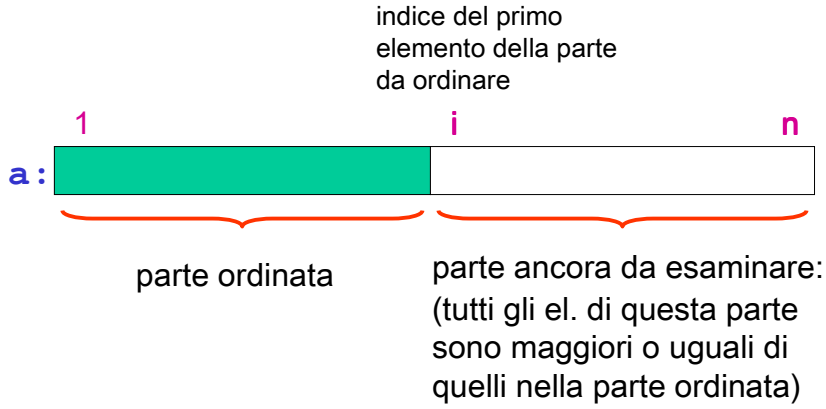
Algoritmo di ordinamento per selezione o per estrazione successiva del minimo

Si cerca il minimo dell'array e lo si mette al primo posto (mettendo il primo al posto del minimo); poi si cerca il minimo nella parte di array dal secondo elemento alla fine, e lo si scambia con il secondo, e così via.

Algoritmo di ordinamento per selezione.

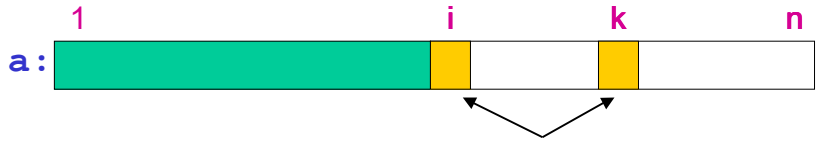
Situazione ad un generico passo intermedio:

array **a**, con **n** = **lunghezza dell'array**



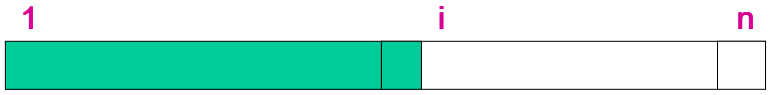
Ordinamento per selezione

a[k] sia il minimo valore in a[i .. n]



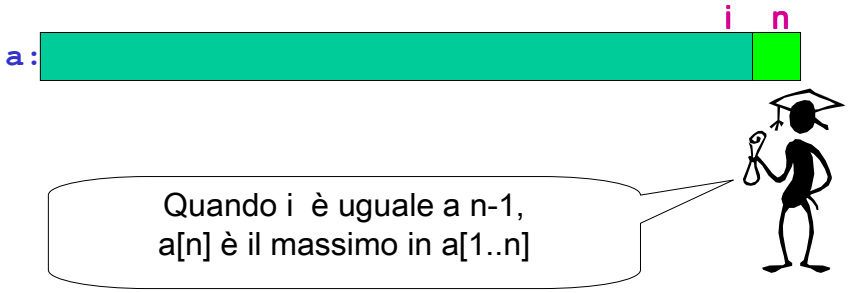
scambia `a[i]` con `a[k]`

così un nuovo elemento viene spostato nella sua posizione corretta definitiva i:
la lunghezza della porzione ordinata aumenta di 1,
la lunghezza di quella ancora da ordinare diminuisce di 1;



Ordinamento per selezione

Condizione di controllo



Quindi il passo deve essere ripetuto per tutto il tempo in cui è **i <= n-1**:

```
for i:= 1 to n-1
```

Cioè il ciclo non controlla l'ultimo elemento, di indice n, poiché esso risulta automaticamente al posto giusto.

La procedura Pascal

```
procedure ssort(var a: arraytype; n: integer);
var i, k: integer;
begin
  for i := 1 to n-1 do begin
    k := indiceMinDa(a, n, i);
    scambia(a[i], a[k]);
  end
end;
```

dove:
`indiceMinDa(a,n,i)` è l'invocazione di una funzione che fornisce come risultato l'indice del minimo elemento della porzione di array `a[i .. n]`;
`scambia` è una procedura che effettua lo scambio dei contenuti di due variabili.

La procedura scambia in Pascal.

```
procedure scambia(var x:int...; var y:int...);
var temp: integer;
begin
  temp:= x;
  x:= y;
  y:= temp;
end;
```

Nota Bene: la qualifica `var` indica che il parametro è passato per riferimento e non per valore.

Se la si omette, la procedura esegue solo uno scambio delle proprie copie locali e non ha quindi alcun effetto sulle variabili passate come argomenti.

Complessità temporale dell'algoritmo di selection-sort.

La funzione **indiceMinDa**, quando viene invocata la prima volta, effettua n passi per trovare il minimo dell'array;
la seconda invocazione effettua n-1 passi, per trovare il minimo di n-1 elementi;
la terza volta effettua n-2 passi, e così via.

Il numero totale di passi effettuati dall'algoritmo è quindi

$$n + (n - 1) + (n - 2) + (n - 3) + \ldots + 3 + 2$$

cioè (formula di "Gauss bambino"):

$$n(n+1)/2 - 1 \text{ cioè } (n^2 + n)/2 - 1$$

Il tempo di calcolo cresce come il quadrato della lunghezza dell'array: quando la lunghezza dell'array si raddoppia, il tempo si qadruplica, quando la lunghezza si triplica, il tempo si moltiplica per nove, ecc.

Un algoritmo di ordinamento quadratico non è un buon algoritmo di ordinamento !

Algoritmo di ordinamento per inserimento o insertion sort

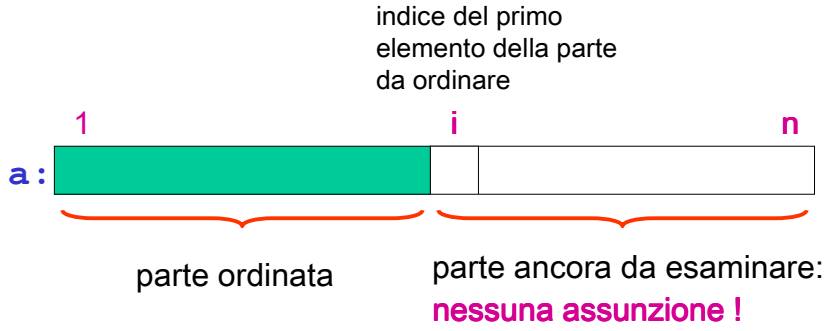
Si prende via via l'elemento successivo dell'array e lo si inserisce al posto giusto nella parte già ordinata.

Attenzione: "ordinamento per inserimento" è il nome di un *algoritmo di ordinamento*, cioè di un algoritmo che prende come argomento una sequenza e la ordina; NON è una procedura che inserisce in modo ordinato degli elementi in una sequenza vuota, pur basandosi sullo stesso principio di essa.

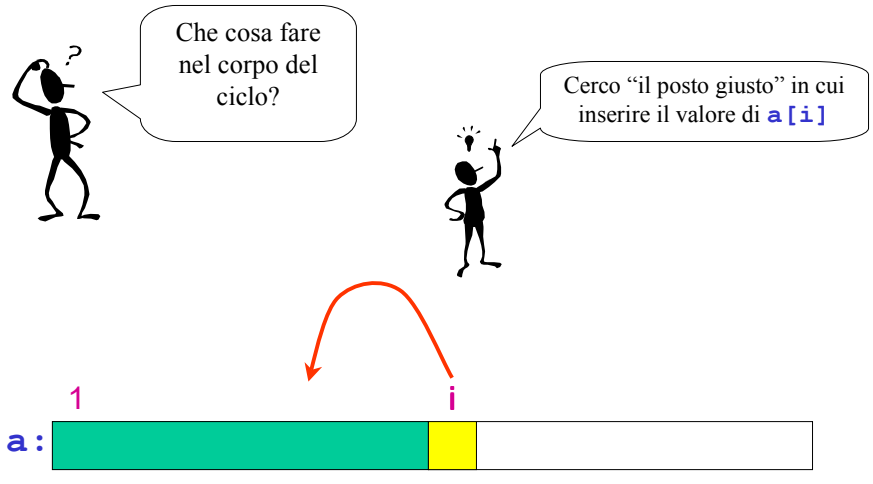
Ordinamento per inserimento.

array **a**, con **n** = **lunghezza dell'array**

Situazione ad un generico passo intermedio:



Ordinamento per inserimento

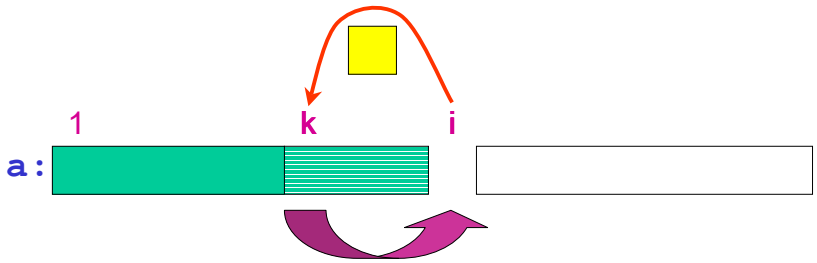


Ordinamento per inserimento

Usiamo una procedura ausiliaria che, percorrendo l'array all'indietro a partire da **i-1**, cerchi il posto in cui inserire **a[i]** facendo "slittare" **a[k .. i-1]** su **a[k+1 .. i]**, e poi effettui l'inserimento stesso:

inserisci(a[i], a, i)

dove **inserisci** è una procedura che definirò opportunamente.

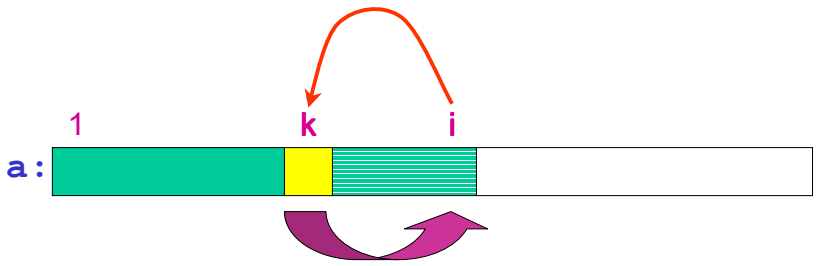


Ordinamento per inserimento

Usiamo una procedura ausiliaria che, percorrendo l'array all'indietro a partire da **i-1**, cerchi il posto in cui inserire **a[i]** facendo "slittare" **a[k .. i-1]** su **a[k+1 .. i]**, e poi effettui l'inserimento stesso:

inserisci(a[i], a, i)

dove **inserisci** è una procedura che definirò opportunamente.



Ordinamento per inserimento

Test di controllo

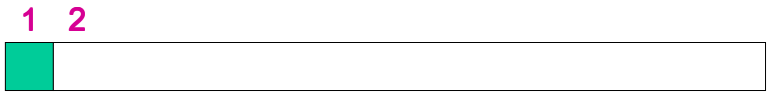
A differenza del selection sort, l'algoritmo non modifica via via la porzione ancora da esaminare, su cui non si ha quindi alcuna condizione; ogni elemento, compreso l'ultimo, deve essere inserito al posto giusto.


```
for i:= ... to n do inserisci(a[i], a, i);
```

ma ...

Ordinamento per inserimento

Inizializzazione



- Il sotto-array a[1], di un solo elemento, è banalmente ordinato
- i è l'indice del primo elemento della parte da ordinare: dunque i = 2

```
for i:= 2 to n do inserisci(a[i], a, i);
```

Numero di passi della procedura *inserisci*

caso migliore: il valore da inserire è maggiore o uguale all'ultimo elemento della parte ordinata (e quindi a tutti); il calcolo richiede **un solo passo**;

caso peggiore: il valore da inserire è minore di tutti gli elementi della parte ordinata, quindi esso verrà confrontato con tutti i suddetti elementi, che dovranno essere tutti spostati di uno;
il numero di passi di calcolo è **proporzionale a i**
(ossia **lineare in i**)

in media il valore da inserire sarà a metà della parte ordinata, quindi il numero di passi sarà circa proporzionale a i/2, cioè ancora **proporzionale a i** (o **lineare in i**)

Numero di passi dell'algoritmo di ordinamento per inserimento

L'algoritmo è costituito da un ciclo for di n passi.

Tuttavia all'interno del ciclo viene eseguito l'algoritmo *inserisci* il cui tempo di calcolo non è costante, ma in generale sarà dipendente dal valore di i.

In media, *inserisci* farà un numero di passi circa uguale a i/2, cioè 1/2 la prima volta, 2/2 la seconda, 3/2 la terza, ...; approssimativamente, il numero totale di passi sarà allora:

$1/2\ (1 + 2 + 3 + \ldots + n) = 1/2\ (n(n+1)/2) = (n^2 + n)/4$

Il tempo di calcolo cresce in modo **quadratico** rispetto alla lunghezza dell'array da ordinare.

Numero di passi dell'algoritmo

caso peggiore: l'array di partenza è ordinato inversamente; allora ogni chiamata della procedura inserisci esegue esattamente k passi, con k successiv. uguale a 1, 2, ...,n (verificare simulando a mano l'esecuzione);

il numero totale di passi è quindi:

$1+2+3+ \ldots + n = n(n+1)/2 = (n^2 + n)/2$

quadratico

caso migliore: l'array di partenza è già ordinato; allora ogni inserimento, inserendo un elemento che è maggiore o uguale dei precedenti, fa un solo passo; il numero totale di passi è quindi

1+1+1+ ... (n volte)

cioè **proporzionale a n** (o **lineare in n**)

Complessità temporale dell'algoritmo

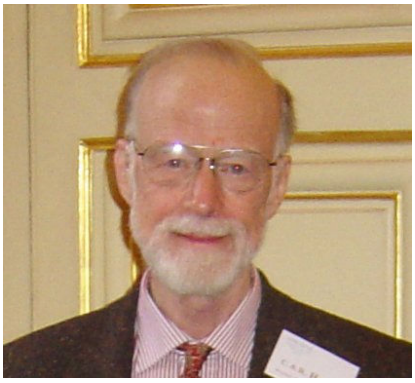
Riassumendo:

- caso peggiore: quadratica (rispetto alla lunghezza)
- caso medio: quadratica
- caso migliore: lineare

Nota: Se l'array è quasi ordinato, cioè ha pochi elementi fuori posto, il numero di passi sarà di poco superiore a n, quindi si è di fatto ancora nel caso migliore.

L'insertion sort è quindi l'algoritmo da usare quando ci si aspetta che le sequenze da ordinare siano di solito già quasi ordinate.

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



Tony Hoare: inventore del Quicksort.

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.

...

Il quicksort

L'idea.

Una possibile definizione <u>induttiva</u> di sequenza ordinata:

- una sequenza vuota o di un solo elemento è ordinata;
- 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 gli $v_1, v_2, \ldots, v_h$ sono maggiori o uguali a p;
  - $u_1, u_2, \ldots, u_k$ è ordinata;
  - $v_1, v_2, \ldots, v_k$ è ordinata;

  è una sequenza ordinata.

Da questa definizione di che cosa è una sequenza ordinata si ricava immediatamente un algoritmo per trasformare una sequenza qualunque in una sequenza ordinata, cioè un algoritmo di ordinamento.

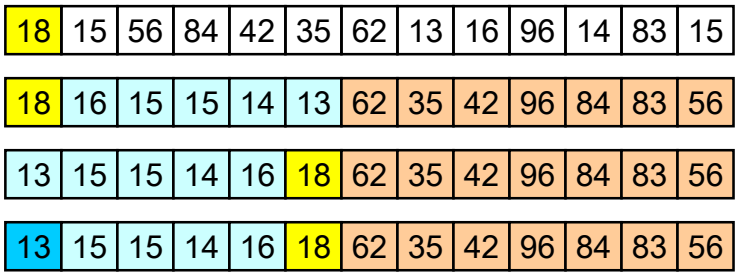
Quicksort: schema astratto dell'algoritmo

```
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.

Esempio di esecuzione su un array

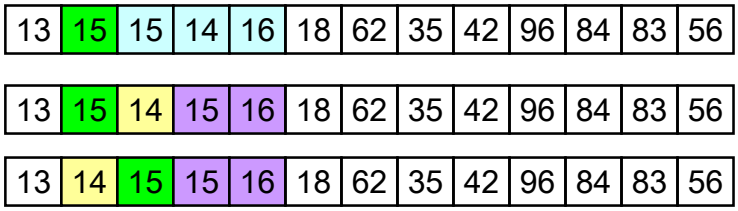
Scelgo ogni volta come pivot il primo elemento della sezione



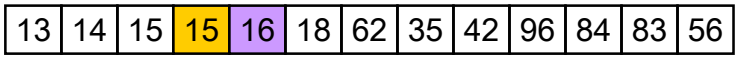
rispetto al pivot 13 tutti gli elementi della sezione rimangono dalla stessa parte (a destra); a sinistra del pivot 13 non c' è quindi nulla, passo a ordinare la sotto-sezione di destra.

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 come, 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.

Nota Bene

Poiché viene "tolto" un elemento dalla (sotto-)sequenza S prima di farne la partizione, la lunghezza di S1 e la lunghezza di S2 sono certamente entrambe minori della lunghezza di S, anche nel caso in cui una delle due parti risulti vuota:

$$\text{lunghezza}(S1) < \text{lunghezza}(S)$$

$$\text{lunghezza}(S2) < \text{lunghezza}(S)$$

Se così non fosse, si potrebbe avere una chiamata ricorsiva su una sottosequenza S1 (o S2) della stessa lunghezza di S e in tal caso la procedura in generale non terminerebbe.

Analisi della complessità: caso medio.

- La partizione di una sequenza in due parti può essere effettuata in tempo lineare (rispetto alla lunghezza n della sequenza).
- Per dividere in 2 parti l'intera sequenza occorrono quindi n passi.
- Mediamente, la sequenza risulterà divisa in due parti di dimensioni circa n/2 ciascuna.
- Per dividere in due parti ciascuna di esse occorreranno quindi circa n/2 passi, quindi per dividere entrambe le parti occorreranno 2(n/2) = n passi.
- Per dividere in due ognuna delle quattro porzioni così ottenute occorreranno mediamente 4(n/4) = n passi.
- ...
- Ad ogni "livello di suddivisione" si fanno in media n passi.
- Quanti sono i "livelli di suddivisione" ?

Analisi della complessità: caso medio.

- Assumiamo per semplicità che la lunghezza n della sequenza sia una potenza di 2:

$$n = 2^k$$

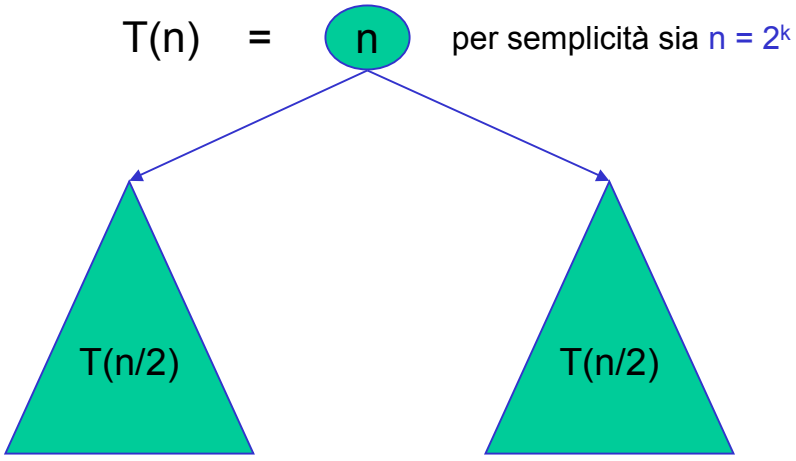
- Allora, poiché a ogni livello si divide per 2, al k-esimo livello si ottengono porzioni di lunghezza 1, su cui non vi è da fare nulla. I livelli sono quindi k.
- Il numero totale di passi è quindi nk.
- Ma per definizione è $k = \log_2 n$.
- Quindi il numero totale di passi, in funzione di n, è

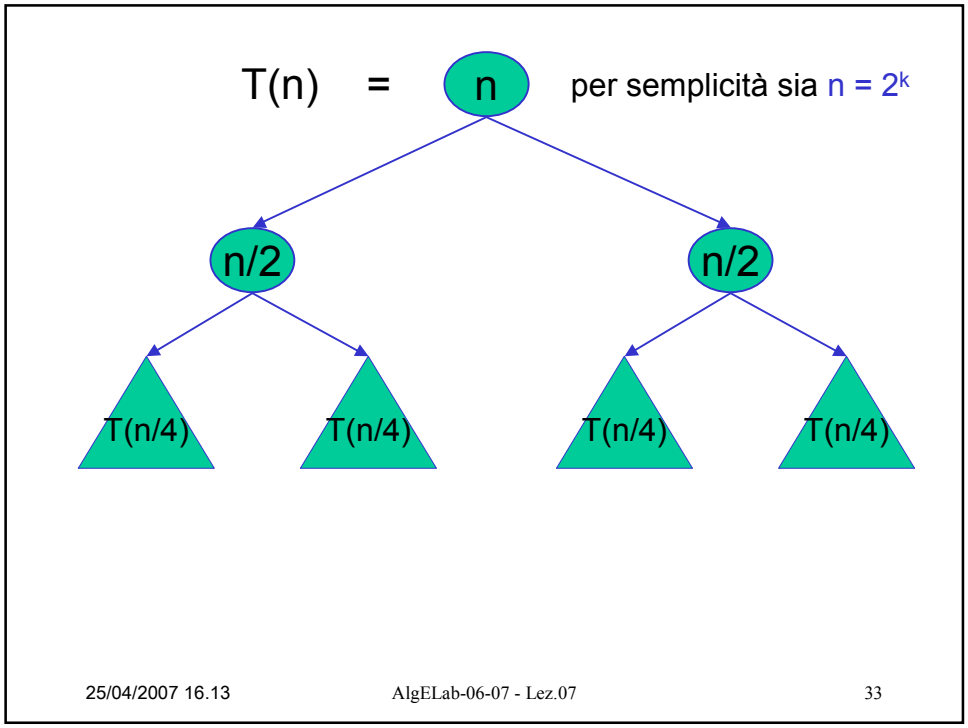
$$n \log_2 n$$

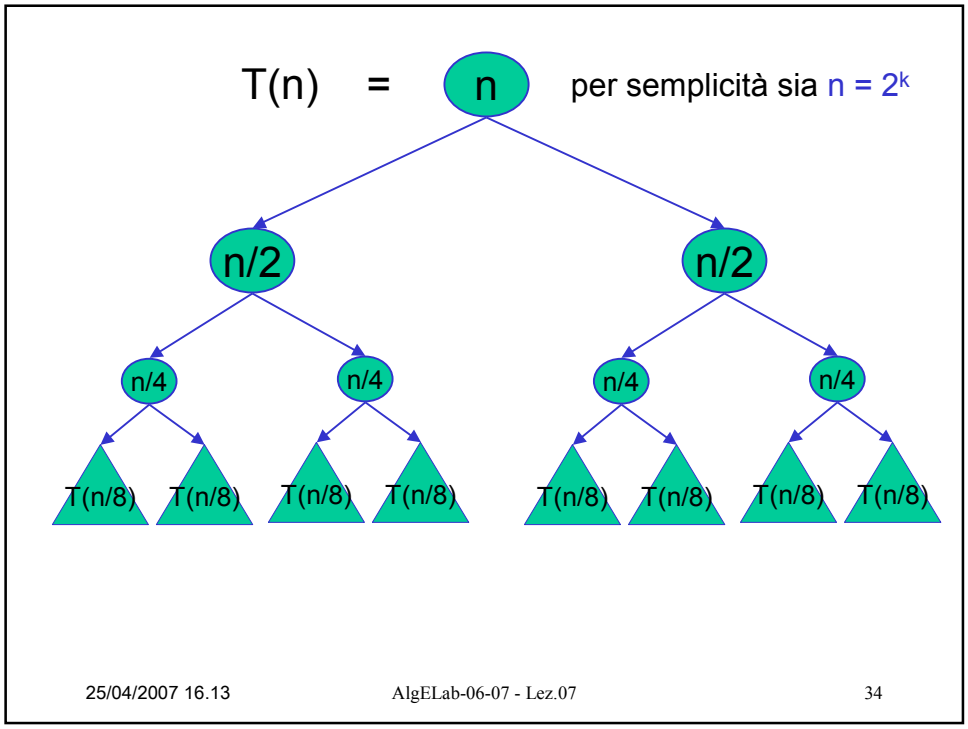
- La funzione $n \log_2 n$ cresce molto più lentamente di n^2 !
- Al crescere della lunghezza della sequenza, il tempo di calcolo del quicksort cresce quindi molto più lentamente (in media) rispetto agli algoritmi precedenti !

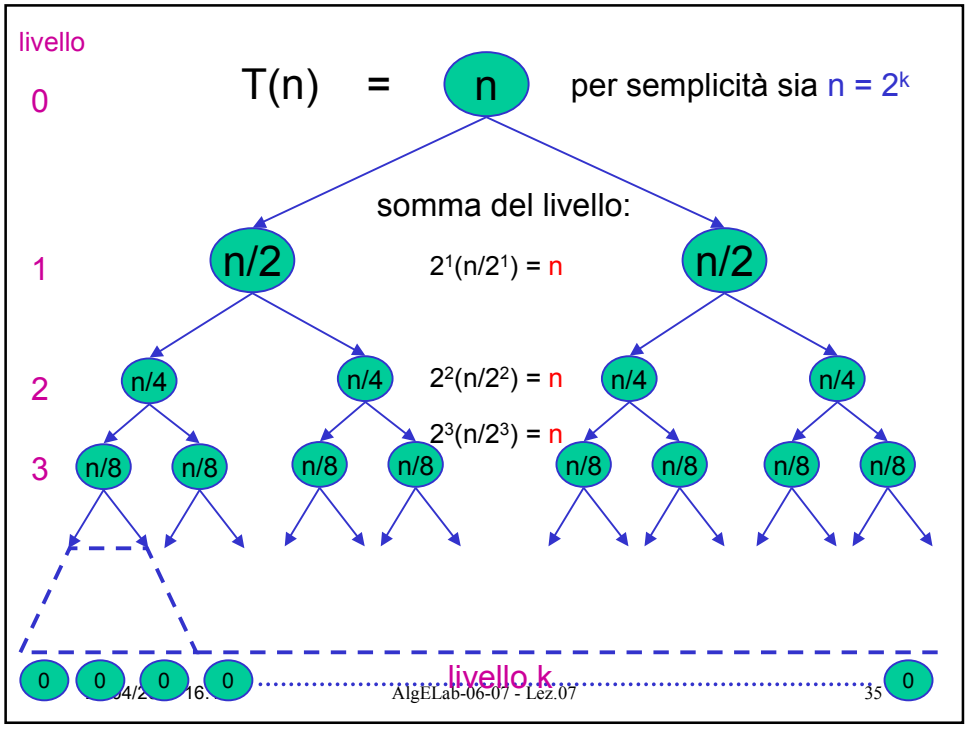
Confronto fra quicksort e insertion-sort

- Il quicksort è più complicato dell'insertion-sort o del selection-sort; per sequenze di un piccolo numero di elementi (al più qualche decina) è di solito più lento di quelli.
- Tuttavia, poiché n log n cresce molto più lentamente di n^2, per sequenze di lunghezza maggiore il quicksort è in media molto più veloce !
- Tale differenza di velocità è importante in pratica: l'esecuzione di un algoritmo di ordinamento quadratico, se la sequenza è abbastanza grande, richiede un tempo inaccettabile: provare per credere !




T(n) =
n
per semplicità sia n = 2k
n/2
n/2
T(n/4)
T(n/4)
T(n/4)
T(n/4)


T(n) =
n
per semplicità sia n = 2k
n/2
n/2
n/4
n/4
n/4
n/4
T(n/8)
T(n/8)
T(n/8)
T(n/8)
T(n/8)
T(n/8)
T(n/8)
T(n/8)



Abbiamo posto $n = 2^k$, quindi $k = \log_2 n$.
L'albero ha k livelli;
per ciascun livello il tempo complessivo è n.
Quindi:

$$T(n) = k \cdot n = n \log_2 n$$

L'albero disegnato nelle slides precedente è l'albero delle chiamate ricorsive, o albero di ricorsione.

In esso si vede bene che i due fattori dell'espressione della complessità temporale hanno i seguenti significati:

- $\log_2 n$ è il numero di livelli di ricorsione, cioè il massimo numero di chiamate ricorsive "annidate";
- n, numero degli elementi, è il tempo impiegato nel partizionamento complessivamente da tutte le chiamate di uno stesso livello di ricorsione.

Nota Bene: Nel quicksort tutto il lavoro dell'algoritmo viene effettuato dal partizionamento, così come nel mergesort veniva effettuato dalla fusione.

Analisi della complessità: caso peggiore.

- Si può presentare un caso molto sfortunato ...
- Il pivot scelto ogni volta risulta essere sempre il più piccolo o il più grande degli elementi della porzione da dividere in due.
- Allora, dopo la prima partizione (n passi), bisogna dividere in due una porzione di $n-1$ elementi, per la quale occorrono ovviamente $n-1$ passi.
- Ma se anche questa volta il pivot è il minimo o il massimo degli $n-1$ elementi, ottengo una porzione di $n-2$ elementi, per dividere la quale occorrono $n-2$ passi.
- ... e se, come abbiamo supposto, si è così sfortunati che questa situazione si ripete per ogni pivot, il numero totale di passi dell'algoritmo sarà:

  $$n + (n-1) + (n-2) + \ldots + 3 + 2 + 1 = n(n+1)/2 = (n^2 + n)/2$$

- Nel caso peggiore il quicksort è quadratico !