Il problema dell'ordinamento

Salvatore Filippone
salvatore.filippone@uniroma2.it

Problema dell'ordinamento

Problema dell'ordinamento: Abbiamo una collezione di oggetti (*records*) R_i ciascuno con una *chiave* K_i, $i = 1, \dots, n$. Le chiavi ammettono una

Relazione di ordinamento

1. Ogni coppia di chiavi K_i, K_j soddisfa esattamente una delle tre relazioni (tricotomia): $K_i < K_j$ o $K_i = K_j$ o $K_j < K_i$;
2. Se $K_i < K_q$ e $K_q < K_j$ allora $K_i < K_j$ (transitività).

Insieme ordinato

Vogliamo modificare l'insieme dei record in modo tale che

$$i < j \Rightarrow K_i <= K_j.$$

Ricordiamo che la ricerca in un insieme ordinato ha un costo $O(\log n)$, quindi avere una sequenza in ordine può far risparmiare tempo.
Stiamo assumendo di saper *soltanto confrontare chiavi tra loro*.

Problema dell'ordinamento

Considerando quindi (per ora) un esempio contenente solo le chiavi, vogliamo passare da

$$V = [1, 5, 3, 9, 4, 8, -1, 12]$$

a

$$VS = [-1, 1, 3, 4, 5, 8, 9, 12]$$

con un qualche metodo da identificare.

Permutazioni

Una *permutazione* è una sequenza di n oggetti *distinguibili* tra loro *in uno specifico ordine*.

$$(a, c, b, d)$$

Se l'insieme degli oggetti è l'insieme $\{0, 1, 2, 3, \dots\}$ possiamo usarli come *indici* per accedere un altro insieme di oggetti. per esempio la permutazione

$$P = [6, 0, 2, 4, 1, 5, 3, 7]$$

consente di accedere al vettore V come

$$\begin{aligned} V[P[0]] &== -1 \\ V[P[1]] &== 1 \\ V[P[2]] &== 3 \end{aligned}$$

e quindi di accedere a tutti gli elementi *in ordine* (per poi verificare che sia effettivamente così).

In altre parole,

Ordinare una sequenza

può quindi essere considerato (formalmente) equivalente a trovare una permutazione che accede ai suoi elementi in ordine

Un primo metodo per risolvere il problema dell'ordinamento di n oggetti è

Simple sort

Input: Array contenente le chiavi $V[i]$, $i = 0, \ldots, n-1$;
for $p \leftarrow \ldots$ **do**
  Generare la prossima permutazione possibile p;
  Verificare se effettivamente p riordina l'input;

Generare tutte le permutazioni (D. Knuth, Vol. 4A, pt 1)

```
Input: Elementi della permutazione a_k = [0, 1, 2, 3, ..., n − 1];
while true do
    Stampa la permutazione attuale;
    j ← n − 1;
    while a_j ≥ a_{j+1} do
        j ← j − 1;
        if j == 0 then
            termina;
        l ← n;
        while a_j ≥ a_l do
            l ← l − 1;
            Scambia a_j ↔ a_l;
        k ← j + 1;
        l ← n;
        while k < l do
            Scambia a_k ↔ a_l;
            k ← k + 1;
            l ← l − 1;
```

Problema dell'ordinamento

Il metodo sicuramente funziona, in quanto almeno una delle NP permutazioni mette in ordine l'input.

Problema dell'ordinamento

Il metodo sicuramente funziona, in quanto almeno una delle NP permutazioni mette in ordine l'input.

Ma quanto costa?

- Caso migliore: la prima permutazione funziona; costo: $O(n)$ (solo il costo della verifica);
- Caso peggiore: solo l'ultima funziona; costo $O(NP)$;
- Caso medio: ci si fermerà in una posizione intermedia, p.es. $O(NP/2) == O(NP)$.

Ok, quante sono le permutazioni?

Il metodo sicuramente funziona, in quanto almeno una delle NP permutazioni mette in ordine l'input.

Ma quanto costa?

- Caso migliore: la prima permutazione funziona; costo: $O(n)$ (solo il costo della verifica);
- Caso peggiore: solo l'ultima funziona; costo $O(NP)$;
- Caso medio: ci si fermerà in una posizione intermedia, p.es. $O(NP/2) == O(NP)$.

Ok, quante sono le permutazioni?

$$NP = n! \approx \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \frac{1}{12n} + O\left(\frac{1}{n^2}\right)\right)$$

Problema dell'ordinamento

Il metodo sicuramente funziona, in quanto almeno una delle NP permutazioni mette in ordine l'input.

Ma quanto costa?

- Caso migliore: la prima permutazione funziona; costo: $O(n)$ (solo il costo della verifica);
- Caso peggiore: solo l'ultima funziona; costo $O(NP)$;
- Caso medio: ci si fermerà in una posizione intermedia, p.es. $O(NP/2) == O(NP)$.

Ok, quante sono le permutazioni?

$$NP = n! \approx \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \frac{1}{12n} + O\left(\frac{1}{n^2}\right)\right)$$

Il metodo è chiaramente improponibile!

Un primo metodo "avvicinabile":

Algorithm 1: Bubble sort

Input: Elementi dell'insieme $V = [V[0], V[1], \ldots, V[n-1]]$;
change $\leftarrow$ *true* ;
while *change* **do**
    change $\leftarrow$ *false*;
    for $i \leftarrow 0$ **to** $n-2$ **do**
        if $V[i] > V[i+1]$ **then**
            Scambia $V[i] \leftrightarrow V[i+1]$;
            change $\leftarrow$ *true*;

L'algoritmo funziona perché a forza di scambiare di posto, non ci saranno più elementi fuori ordine.

E ora fate del vostro meglio per dimenticarvene l'esistenza, non c'è alcun motivo di usarlo in pratica.

Un primo metodo *ragionevole* è l'ordinamento per *inserzione*

Algorithm 2: Insertion Sorting

Input: Array contenente le chiavi $V[i]$, $i = 0, \ldots, n-1$;
for $i \leftarrow 1$ **to** $n-1$ **do**
  $temp \leftarrow V[i]$;
  $j \leftarrow i$;
  while $j > 0$ **and** $V[j-1] > temp$ **do**
    $V[j] \leftarrow V[j-1]$;
    $j \leftarrow j - 1$;
  $V[j] \leftarrow temp$;

Esempio: Insertion Sorting

Abbiamo ora due problemi:

1. Dimostrare che l'algoritmo effettivamente produce in uscita una sequenza ordinata;
2. Valutare la sua complessità.

Correttezza

1. Prima della esecuzione del ciclo `For`, la sottosequenza $V[0]$ è sicuramente ordinata in quanto contiene un solo elemento;
2. L'esecuzione del ciclo `While` all'interno della iterazione i-esima del ciclo `For` trasferisce un elemento nella sua posizione corretta;
3. All'ultimo passo della iterazione i-esima del ciclo `For`, la sottosequenza $V[0], \ldots, V[i]$ è quindi ordinata;
4. Quando il ciclo `For` termina definitivamente, questa proprietà è valida per $V[0], \ldots, V[n-1]$, quindi il vettore è ordinato.

Utile esercizio: rivedere/completare la dimostrazione.

Quale è il costo del nostro algoritmo?

Algorithm 3: Insertion Sorting

Input: Array contenente le chiavi $V[i]$, $i = 0, \dots, n-1$;

for $i \leftarrow 1$ **to** $n-1$ **do**
  $temp \leftarrow V[i]$;
  $j \leftarrow i$;
  while $j > 0$ **and** $V[j-1] > temp$ **do**
    $V[j] \leftarrow V[j-1]$;
    $j \leftarrow j-1$;
  $V[j] \leftarrow temp$;

Insertion sort

Quale è il costo?

- Ad ogni iterazione del ciclo `For` $i = 0 \dots n-1$, abbiamo un ciclo interno, ed un paio di assegnazioni;
- Il ciclo interno viene eseguito per valori di j che possono andare (nel caso peggiore) i a 0, mentre nel caso migliore non verrà eseguita alcuna operazione.

Ogni iterazione del ciclo esterno ha un costo potenzialmente diverso. Quindi, nel caso peggiore :

$$\text{opcnt} = (\sum_{i=0}^{n-1} 1 + 1) + (\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} 1) \leq 2n + \sum_{i=0}^{n-1}(i+1) = 3n + \frac{(n-1)n}{2} = O(n^2)$$

mentre nel caso migliore sopravvive solo il ciclo esterno e quindi il costo è $\Omega(n)$.

Espressioni utili:

- $$\sum_{i=n_1}^{n_2} 1 = n_2 - n_1 + 1$$
- $$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \approx \frac{n^2}{2} = O(n^2)$$
- $$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{n^3}{3} = O(n^3)$$
- Il termine dominante può essere calcolato con un integrale

$$\sum_{i=0}^{n} i^2 \approx \frac{n^3}{3} = \int_0^n x^2 dx$$

Insertion sort

Per quanto visto, *insertion sort* è

$$\Omega(n)$$
$$O(n^2)$$

Si può fare di meglio?

Limiti dell'ordinamento

Quale è un limite alla complessità del problema dell'ordinamento?

Limiti dell'ordinamento

Quale è un limite alla complessità del problema dell'ordinamento?

- Dobbiamo distinguere tra $n!$ configurazioni possibili;

Limiti dell'ordinamento

Quale è un limite alla complessità del problema dell'ordinamento?

- Dobbiamo distinguere tra $n!$ configurazioni possibili;
- Stiamo effettuando delle scelte binarie (ossia, ad ogni passo ci chiediamo se $V[k] \gtrless V[j]$ per qualche coppia k, j);

Limiti dell'ordinamento

Quale è un limite alla complessità del problema dell'ordinamento?

- Dobbiamo distinguere tra $n!$ configurazioni possibili;
- Stiamo effettuando delle scelte binarie (ossia, ad ogni passo ci chiediamo se $V[k] \gtrless V[j]$ per qualche coppia k, j);
- Possiamo quindi assumere di stare esaminando un albero binario; ma bisogna che il numero delle foglie delll'albero sia almeno $n!$;

Limiti dell'ordinamento

Quale è un limite alla complessità del problema dell'ordinamento?

- Dobbiamo distinguere tra $n!$ configurazioni possibili;
- Stiamo effettuando delle scelte binarie (ossia, ad ogni passo ci chiediamo se $V[k] \gtrless V[j]$ per qualche coppia k, j);
- Possiamo quindi assumere di stare esaminando un albero binario; ma bisogna che il numero delle foglie delll'albero sia almeno $n!$;
- Per arrivare dalla radice alle foglie si percorrono k livelli;

Limiti dell'ordinamento

Quale è un limite alla complessità del problema dell'ordinamento?

- Dobbiamo distinguere tra $n!$ configurazioni possibili;
- Stiamo effettuando delle scelte binarie (ossia, ad ogni passo ci chiediamo se $V[k] \gtrless V[j]$ per qualche coppia k, j);
- Possiamo quindi assumere di stare esaminando un albero binario; ma bisogna che il numero delle foglie delll'albero sia almeno $n!$;
- Per arrivare dalla radice alle foglie si percorrono k livelli;
- Ogni livello i contiene 2^i nodi, quindi il numero di livelli k è eguale al logaritmo (in base 2) del numero delle foglie;

Limiti dell'ordinamento

Quale è un limite alla complessità del problema dell'ordinamento?

- Dobbiamo distinguere tra $n!$ configurazioni possibili;
- Stiamo effettuando delle scelte binarie (ossia, ad ogni passo ci chiediamo se $V[k] \gtrless V[j]$ per qualche coppia k, j);
- Possiamo quindi assumere di stare esaminando un albero binario; ma bisogna che il numero delle foglie delll'albero sia almeno $n!$;
- Per arrivare dalla radice alle foglie si percorrono k livelli;
- Ogni livello i contiene 2^i nodi, quindi il numero di livelli k è eguale al logaritmo (in base 2) del numero delle foglie;
- Quanti livelli ci vogliono per distinguere $n!$ foglie? $\log(n!) \approx \log(n^n) = n \log n$.

Limiti dell'ordinamento

Quale è un limite alla complessità del problema dell'ordinamento?

- Dobbiamo distinguere tra $n!$ configurazioni possibili;
- Stiamo effettuando delle scelte binarie (ossia, ad ogni passo ci chiediamo se $V[k] \gtrless V[j]$ per qualche coppia k, j);
- Possiamo quindi assumere di stare esaminando un albero binario; ma bisogna che il numero delle foglie delll'albero sia almeno $n!$;
- Per arrivare dalla radice alle foglie si percorrono k livelli;
- Ogni livello i contiene 2^i nodi, quindi il numero di livelli k è eguale al logaritmo (in base 2) del numero delle foglie;
- Quanti livelli ci vogliono per distinguere $n!$ foglie? $\log(n!) \approx \log(n^n) = n \log n$.

Limiti dell'ordinamento

Quale è un limite alla complessità del problema dell'ordinamento?

- Dobbiamo distinguere tra $n!$ configurazioni possibili;
- Stiamo effettuando delle scelte binarie (ossia, ad ogni passo ci chiediamo se $V[k] \gtrless V[j]$ per qualche coppia k, j);
- Possiamo quindi assumere di stare esaminando un albero binario; ma bisogna che il numero delle foglie delll'albero sia almeno $n!$;
- Per arrivare dalla radice alle foglie si percorrono k livelli;
- Ogni livello i contiene 2^i nodi, quindi il numero di livelli k è eguale al logaritmo (in base 2) del numero delle foglie;
- Quanti livelli ci vogliono per distinguere $n!$ foglie? $\log(n!) \approx \log(n^n) = n \log n$.

Stiamo sempre assumendo di saper solo fare confronti tra chiavi!

Limiti dell'ordinamento

Quale è un limite alla complessità del problema dell'ordinamento?

- Dobbiamo distinguere tra $n!$ configurazioni possibili;
- Stiamo effettuando delle scelte binarie (ossia, ad ogni passo ci chiediamo se $V[k] \gtrless V[j]$ per qualche coppia k, j);
- Possiamo quindi assumere di stare esaminando un albero binario; ma bisogna che il numero delle foglie delll'albero sia almeno $n!$;
- Per arrivare dalla radice alle foglie si percorrono k livelli;
- Ogni livello i contiene 2^i nodi, quindi il numero di livelli k è eguale al logaritmo (in base 2) del numero delle foglie;
- Quanti livelli ci vogliono per distinguere $n!$ foglie? $\log(n!) \approx \log(n^n) = n \log n$.

Stiamo sempre assumendo di saper solo fare confronti tra chiavi!

Se sappiamo solo fare confronti

Non possiamo aspettarci una complessità migliore di $O(n \log(n))$

Quicksort

Vediamo ora un primo algoritmo *ricorsivo* che *raggiunge* una complessità $O(n \log(n))$.

- Sia dato $(V[0], \ldots, V[n-1])$;
- Identifichiamo una posizione $0 \leq k \leq n-1$;
- Se possiamo trasformare V in modo che valga sempre

  $$V[i] \leq V[j]$$

  per ogni valore $0 \leq i < k$ e $k < j \leq n-1$, e inoltre $V[i] \leq V[k]$ e $V[k] \leq V[j]$; si noti che le due sezioni del vettore *non sono necessariamente* ordinate al loro interno;
- Allora, possiamo richiamare la stessa procedura due volte, una su $(V[0], \ldots, V[k-1])$ e poi su $(V[k+1], \ldots, V[n-1])$, ricorsivamente, per ordinare ciascuna sezione;
- Il risultato finale sarà una sequenza ordinata.

Algorithm 4: QuickSort

Input: Elementi dell'insieme $V = (V[0], V[1], \dots, V[n-1])$, *primo*, *ultimo* ;

$k \leftarrow$ perno $(V, primo, ultimo)$;

Quicksort($V, primo, k-1$);

Quicksort($V, k+1, ultimo$);

Algorithm 5: Perno

Input: Elementi dell'insieme $V = (V[0], V[1], \ldots, V[n-1])$, $primo$, $ultimo$;

$x \leftarrow V[primo]$;

$k \leftarrow primo$;

for $i \leftarrow primo$ **to** $ultimo$ **do**

  if $V[i] < x$ **then**

    $k \leftarrow k + 1$;

    Scambia $V[i] \leftrightarrow V[k]$;

$V[primo] \leftarrow V[k]$;

$V[k] \leftarrow x$;

Risultato k

Un esempio (vedi libro):

$$[9, 12, 8, 18, 6, 13, 11, 3, 5, 10]$$

Un esempio (vedi libro):

$$[9, 12, 8, 18, 6, 13, 11, 3, 5, 10]$$

$$[5, 8, 6, 3][9][13, 11, 18, 12, 10]$$

Un esempio (vedi libro):

$$[9, 12, 8, 18, 6, 13, 11, 3, 5, 10]$$

$$[5, 8, 6, 3][9][13, 11, 18, 12, 10]$$

$$[3][5][6, 8][9][10, 11, 12][13][18]$$

Quicksort

Un esempio (vedi libro):

$$[9, 12, 8, 18, 6, 13, 11, 3, 5, 10]$$

$$[5, 8, 6, 3][9][13, 11, 18, 12, 10]$$

$$[3][5][6, 8][9][10, 11, 12][13][18]$$

$$[3][5][6][8][9][10][11, 12][13][18]$$

Un esempio (vedi libro):

$$[9, 12, 8, 18, 6, 13, 11, 3, 5, 10]$$

$$[5, 8, 6, 3][9][13, 11, 18, 12, 10]$$

$$[3][5][6, 8][9][10, 11, 12][13][18]$$

$$[3][5][6][8][9][10][11, 12][13][18]$$

$$[3][5][6][8][9][10][11][12][13][18]$$

Un esempio (vedi libro):

$$[9, 12, 8, 18, 6, 13, 11, 3, 5, 10]$$

$$[5, 8, 6, 3][9][13, 11, 18, 12, 10]$$

$$[3][5][6, 8][9][10, 11, 12][13][18]$$

$$[3][5][6][8][9][10][11, 12][13][18]$$

$$[3][5][6][8][9][10][11][12][13][18]$$

Si noti che (in questo caso al primo passo), i sottovettori a sinistra e destra del perno *non* sono già ordinati.

Usiamo il teorema sulle ricorrenze lineari con partizione bilanciata.

Nella partizione, ci aspettiamo che la posizione del perno k cada a metà del vettore, ossia che si stia partizionando il vettore iniziale in due metà eguali:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(\frac{n}{2}) + n & n > 1 \end{cases}$$

e allora avremo un tempo di esecuzione

$$T(n) = O(n \log(n)).$$

Mergesort

Vediamo ora un secondo algoritmo ricorsivo che raggiunge una complessità $O(n \log(n))$.

- Sia dato $(V[0], \ldots, V[n-1])$;
- Supponiamo che esista una posizione $0 \leq k \leq n-1$ tale che:
  1. La sottosequenza $(V[0], \ldots, V[k])$ è ordinata;
  2. La sottosequenza $(V[k+1], \ldots, V[n-1])$ è ordinata.
- Allora, possiamo ottenere una sequenza ordinata con una semplice procedura in cui andiamo a fondere tra loro le due sottosequenze ordinate;
- Per completare il metodo è sufficiente spezzare ricorsivamente la sequenza iniziale fino ad arrivare a sottosequenze di dimensione unitaria, che sono necessariamente ordinate.

Algorithm 6: MergeSort

Input: Elementi dell'insieme $V = (V[0], V[1], \ldots, V[n-1])$, *primo*, *ultimo* ;

if *primo* $<$ *ultimo* **then**

  $mezzo \leftarrow \lfloor (primo + ultimo)/2 \rfloor$;
  Mergesort(V, *primo*, *mezzo*);
  Mergesort(V, *mezzo* $+ 1$, *ultimo*);
  Merge(V, *primo*, *ultimo*, *mezzo*);

Utile esercizio: rieseguire l'algoritmo sullo stesso esempio del QuickSort.

Algorithm 7: Merge

Input: Elementi dell'insieme $V = (V[0], V[1], \ldots, V[n-1])$, *primo*, *ultimo*, *mezzo* ;

$i \leftarrow primo$;
$j \leftarrow mezzo + 1$;
$k \leftarrow primo$;
while $i \leq mezzo$ **and** $j \leq ultimo$ **do**
    if $V[i] \leq V[j]$ **then**
        $T[k] = V[i]$;
        $i \leftarrow i + 1$;
    else
        $T[k] = V[j]$;
        $j \leftarrow j + 1$;
    $k \leftarrow k + 1$;
for $h \leftarrow mezzo$ **downto** i **do**
    $V[j] \leftarrow V[h]$;
    $j \leftarrow j - 1$;
for $j \leftarrow primo$ **to** $k - 1$ **do**
    $V[j] \leftarrow T[j]$;

Complessità Computazionale del Mergesort

La complessità computazionale del Mergesort è analoga a quella del Quicksort, dato che ad ogni passo ricorsivo sto spezzando la sequenza iniziale in due sottosequenze eguali,

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(\frac{n}{2}) + n & n > 1 \end{cases}$$

per cui

$$T(n) = O(n \log(n)).$$

Ancora sul problema dell'ordinamento

Abbiamo cominciato la discussione del problema dell'ordinamento con:

Abbiamo una collezione di oggetti (*records*) R_i ciascuno con una *chiave* K_i, $i = 1, \ldots, n$.

Negli ultimi esempi, abbiamo considerato che il record R_i e la chiave K_i coincidano semplicemente in un numero intero; ma questo in generale non è vero, in quanto il record R_i può contenere ulteriori informazioni. Si può quindi dare il caso che $K_i == K_j$, ma $R_i \neq R_j$.

Ordinamento stabile

Un ordinamento è *stabile* se ogni volta che due chiavi sono eguali $K_i == K_j$,i record R_i ed R_j vengono mantenuti nello stesso ordine tra loro con cui comparivano nella sequenza iniziale.

Mergesort è un algoritmo stabile, Quicksort no, e questo è un problema per alcune applicazioni.

Ancora qualche osservazione

- Quicksort e Mergesort sono in qualche senso speculari, in quanto il lavoro di "sistemazione" viene effettuato da Quicksort *prima* delle chiamate ricorsive, mentre invece Mergesort lo effettua *dopo*
- Nel descrivere il comportamento di Quicksort abbiamo detto che ci aspettiamo una partizione bilanciata a metà; tuttavia questo non è garantito (cosa succede con una sequenza che fin da principio è ordinata?)
- Il comportamento di Mergesort è invece garantito essere bilanciato;
- Mergesort richiede uno spazio aggiuntivo di memoria $O(n)$, Quicksort $O(1)$.

Ancora sul problema dell'ordinamento

Complessità nel caso peggiore

- Quicksort: $O(n^2)$
- Mergesort: $O(n \log(n))$.

Caso "medio"

E tuttavia, nel caso "medio" (per una qualche definizione di "medio"), Quicksort è più veloce

Esistono molti metodi, bisogna scegliere quello più appropriato per la applicazione che si sta affrontando.

Code con priorità

Poniamoci ora il seguente problema:

Coda con priorità

Abbiamo un insieme di elementi ciascuno con assegnata una *priorità* (un valore intero, p.es. priorità massima per valori piccoli). Vogliamo poter eseguire le operazioni:

- `min` restituisce l'elemento con priorità di valore numerico più basso;
- `DeleteMin` come `min` ma elimina l'elemento dall'insieme;
- `insert(t,p)` Aggiunge un oggetto in una coda con priorità p;
- `decrease(t,p)` Diminuisce la prorità di un oggetto portandola a p;

Se realizziamo la coda con una semplice lista (p.es. con puntatori), che tempi otteniamo per le operazioni suddette?

Code con priorità

Liste non ordinate

- `insert` $O(1)$;
- `min` $O(N)$;
- `DeleteMin` $O(N)$.

Liste ordinate

- `insert` $O(N)$;
- `min` $O(1)$;
- `DeleteMin` $O(1)$.

Si noti che una complessità $O(N)$ sulla singola operazione comporta una complessità totale $O(N^2)$ se l'operazione viene iterata N volte.

Una particolare struttura dati adatta per la rappresentazione delle code con priorità ci consente di definire un metodo di ordinamento con complessità $O(n\log(n))$.
Il metodo usa la struttura dati di supporto:

Heap

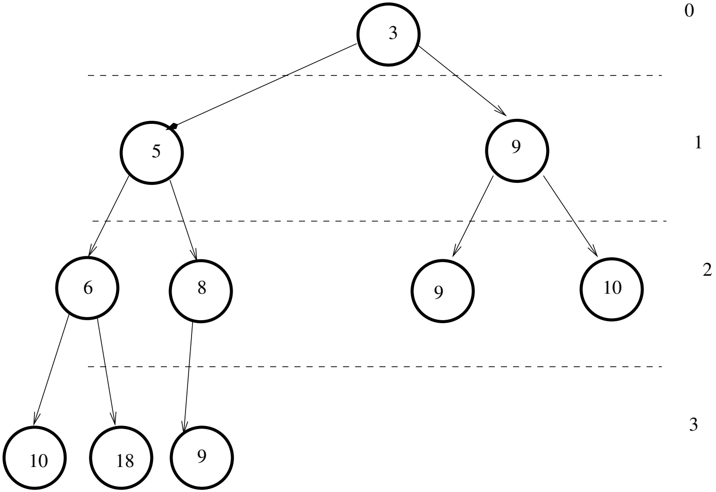
Una *heap* è una struttura dati che consente di realizzare in modo efficiente (in tempo $O(\log(n))$) le operazioni:

1. Selezionare l'elemento con chiave più piccola;
2. Aggiungere un nuovo elemento alla struttura dati.

Si noti che nella realizzazione più comune, la selezione del minimo in sé è banale, ma il vero problema è di lasciare la struttura dati in uno stato tale che anche la *prossima* selezione possa essere effettuata con la stessa complessità.

Una struttura dati *heap* può essere descritta facilmente come:

Un albero binario in cui la chiave associata con ciascun nodo è minore della chiave associata con ciascuno dei due figli



Un esempio di una *heap* memorizzata in un vettore:

$$[3, 5, 9, 6, 8, 9, 10, 10, 18, 9]$$

In sostanza, si tratta di un albero binario memorizzato in un array lineare. La descrizione migliora se assumiamo un indirizzamento dell'array a partire dall'indice 1:

1. L'elemento in posizione i è minore degli elementi in posizione $2i$ e $2i + 1$;
2. L'elemento in posizione 1 è il più piccolo (radice dell'albero)
3. Vale sempre

$$K_{\lfloor j/2 \rfloor} \leq K_j \quad \text{per } 1 \leq \lfloor j/2 \rfloor < j \leq N.$$

ossia l'elemento di posizione $\lfloor j/2 \rfloor$ è il padre dei due elementi in posizione j e $j + 1$

Per una descrizione con indice di base 0 bisogna modificare le espressioni precedenti in:

1. L'elemento più piccolo è in posizione 0;
2. L'elemento in posizione i è minore degli elementi in posizione $2i + 1$ e $2i + 2$;
3. L'elemento padre è nella posizione $\lfloor (j - 1)/2 \rfloor$.

Vediamo ora come realizzare le operazioni fondamentali (sempre con indice di base 1):

Algorithm 8: DeleteMin

Input: Una *heap* $K = (K[1], K[2], \ldots, K[N])$, un array V con i risultati precedenti

$K_{\text{min}} \leftarrow K[1]$;
$K[1] \leftarrow R[N]$;
$N \leftarrow N - 1$;
$i \leftarrow 1$;
while $i \leq N/2$ **do**
    if $2i = N$ *o* $K[2i] < K[2i+1]$ **then**
        $j \leftarrow 2i$;
    else
        $j \leftarrow 2i + 1$;
    if $K[i] > K[j]$ **then**
        Scambia $K[i]$ con $K[j]$;
    $i \leftarrow j$;
Risultato $V \leftarrow [K_{\text{min}}, V]$

Un esempio di una *heap* memorizzata in un vettore:

$$K = [3, 5, 9, 6, 8, 9, 10, 10, 18, 9] \quad V = []$$

Eseguiamo l'algoritmo `DeleteMin` "a mano":

$$\begin{array}{rcl}
K_{min} & gets & 3 \\
K & \leftarrow & [9, 5, 9, 6, 8, 9, 10, 10, 18] \\
i & \leftarrow & 1,\ j \leftarrow 2 \\
K[i] & > & K[j]?\ \text{Si.} \\
K & \leftarrow & [5, 9, 9, 6, 8, 9, 10, 10, 18] \\
i & \leftarrow & 2,\ j \leftarrow 4 \\
K[i] & > & K[j]?\ \text{Si.} \\
K & \leftarrow & [5, 6, 9, 9, 8, 9, 10, 10, 18] \\
i & \leftarrow & 4,\ j \leftarrow 8 \\
K[i] & > & K[j]?\ \text{No.} \\
\text{Salva} & & K_{min}\ \text{in}\ V \\
K & \leftarrow & [5, 6, 9, 9, 8, 9, 10, 10, 18] \\
V & \leftarrow & [3]
\end{array}$$

L'algoritmo funziona correttamente e con la complessità promessa perché

DeleteMin

- L'ultimo elemento della heap viene spostato in prima posizione, ed è l'unico che potrebbe essere in una posizione "sbagliata";
- Ad ogni passo (a cominciare da $i \leftarrow 1$), l'elemento K_i viene confrontato con *il più piccolo* K_j tra i suoi due figli ($2i$ oppure $2i + 1$);
- Se $K_i < K_j$, allora la proprietà fondamentale della heap è già soddisfatta, perché tutti gli elementi "figli" di K_j sono già a posto;
- Altrimenti, possiamo scambiare $K_i \leftrightarrow K_j$, e K_j sarà a posto visto che diventa "padre" sia di K_i che del suo "fratello", di cui era minore per come j è stato scelto

Siccome ad ogni iterazione del ciclo `while` il valore di i viene (almeno) raddoppiato, la procedura termina in tempo $O(\log(n))$.

La seconda operazione da realizzare è l'inserimento nella *heap* di un nuovo elemento:

Algorithm 9: HeapInsert

Input: La nuova chiave K_n e una *heap* $K = (K[1], K[2], \ldots, K[N])$

$N \leftarrow N + 1$;
$K[N] \leftarrow K_n$;
$i \leftarrow N$;
while $i > 1$ *e* $K_i < K_{\lfloor i/2 \rfloor}$ **do**
  Scambia $K_i \leftrightarrow K_{\lfloor i/2 \rfloor}$;
  $i \leftarrow i/2$;

Di nuovo, ad ogni iterazione del ciclo `while` il valore di i viene dimezzato e quindi la procedura termina in tempo $O(\log(n))$.

Possiamo quindi definire

Algorithm 10: HeapSort

Input: Un vettore di chiavi $K_{in} = (K[1], K[2], \ldots, K[N])$

Crea una heap vuota V;

for $j \leftarrow 1$ **to** N **do**

  HeapInsert($K[j], V$);

for $j \leftarrow 1$ **to** N **do**

  DeleteMin($V[j : N]$);

Risultato V

L'algoritmo

- Come vanno gestite eventuali eguaglianze tra chiavi?
- Come lo si può correggere?
- Come convertire gli indici dei vettori da base 1 a base 0?
- Quanta memoria serve?

External sorting

L'ordinamento presenta particolari problemi quando si debbano ordinare grosse quantità di dati, tali da NON poter essere memorizzate nella RAM.
Si tratta di una argomento specializzato che non tratteremo oltre, ma che è estremamente rilevante per applicazioni con database di grandi dimensioni.

Altri algoritmi

straight selection: Ricerca sequenziale del minimo corrente ad ogni passo $O(N^2)$;

tree selection: Ricerca usando alberi (binari) "normali"; dipende dalla struttura dell'albero, e nel caso peggiore $O(N^2)$ (vedi lezioni sugli alberi);

shellsort: Simile ad insertion sort, scelta della distanza tra i termini da confrontare; è possibile una implementazione $O(N^{1.5})$;

Shellsort

Uno dei problemi di insertion sort è che gli scambi tra chiavi avvengono tra posizioni adiacenti, quindi una chiave "viaggia" a velocità bassa. Se invece scegliessimo di confrontare tra loro elementi "lontani" allora si potrebbe migliorare:

Shellsort

- Generiamo una sequenza di incrementi h_k, $k = 1, \ldots, t$ (p.es. 29,13,5,1) ;
- Per ogni elemento della sequenza, andiamo ad applicare un passo di sort per inserzione sui dati con confronti a distanza h_k;
- Se la sequenza contiene (come ultimo incremento) il valore $h_t = 1$, allora il risultato finale è necessariamente ordinato (in quanto l'ultimo passo si riduce ad insertion sort normale);
- Se la sequenza è scelta opportunamente, il tempo di esecuzione totale diminuisce.

Algorithm 11: Shellsort

Input: Array contenente le chiavi $V[i]$, $i = 0, \dots, n-1$;

```
h ← 1;
while h ≤ n do
    h ← 3h + 1;
h ← ⌊h/3⌋;
while h ≥ 1 do
    for i ← h + 1 to n do
        temp ← V[i];
        j ← i;
        while j > h and V[j − h] > temp do
            V[j] ← V[j − h];
            j ← j − h ;
        V[j] ← temp;
    h ← ⌊h/3⌋;
```

Esistono algoritmi $O(N)$?

Secondo la discussione che abbiamo fatto in precedenza, sembrerebbe di no.

Esistono algoritmi $O(N)$?

Secondo la discussione che abbiamo fatto in precedenza, sembrerebbe di no.

E invece *si*.

Esistono algoritmi $O(N)$?

Secondo la discussione che abbiamo fatto in precedenza, sembrerebbe di no.

E invece *si*.

Algoritmi $O(N)$

Sono possibili algoritmi lineari *SE* abbiamo a disposizione *PIÙ* informazioni, in particolare se siamo capaci di fare qualcosa di più che semplicemente confrontare due chiavi tra loro.

Dobbiamo sapere esattamente come sono fatte le chiavi.

Esistono algoritmi $O(N)$?

binsort

Supponiamo che le chiavi siano numeri compresi tra 0 e 7; allora possiamo:

binsort

Supponiamo che le chiavi siano numeri compresi tra 0 e 7; allora possiamo:

- Preparare 8 array ausiliary (da 0 a 7);

Esistono algoritmi $O(N)$?

binsort

Supponiamo che le chiavi siano numeri compresi tra 0 e 7; allora possiamo:

- Preparare 8 array ausiliary (da 0 a 7);
  - Scorrere il vettore di input V;
  - Per ogni chiave, accodarla all'array ausiliario che abbia come indice la chiave stessa;

Esistono algoritmi $O(N)$?

binsort

Supponiamo che le chiavi siano numeri compresi tra 0 e 7; allora possiamo:

- Preparare 8 array ausiliary (da 0 a 7);
  - Scorrere il vettore di input V;
  - Per ogni chiave, accodarla all'array ausiliario che abbia come indice la chiave stessa;
- Mettere "in fila" il contenuto dell'array 0, seguito dal contenuto dell'array 1, etc...

Esistono algoritmi $O(N)$?

binsort

Supponiamo che le chiavi siano numeri compresi tra 0 e 7; allora possiamo:

- Preparare 8 array ausiliary (da 0 a 7);
    - Scorrere il vettore di input V;
    - Per ogni chiave, accodarla all'array ausiliario che abbia come indice la chiave stessa;
- Mettere "in fila" il contenuto dell'array 0, seguito dal contenuto dell'array 1, etc...

Avremo un ordinamento in tempo lineare

Questa idea si generalizza nell'algoritmo di *radix sort*, in cui tipicamente si procede esaminando prima il primo byte di tutte le chiavi, poi il secondo etc.