Algoritmi e Strutture Dati[1]

Corso di Laurea in Ingegneria dell'Informazione
Sapienza Università di Roma – sede di Latina

Fabio Patrizi

Dipartimento di Ingegneria Informatica, Automatica e Gestionale (DIAG)
Sapienza Università di Roma – Italy
www.dis.uniroma1.it/~patrizi
patrizi@dis.uniroma1.it

[1] Slides prodotte a partire dal materiale didattico fornito con il testo *Demetrescu, Finocchi, Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione*.

Algoritmi di Ordinamento

SAPIENZA UNIVERSITÀ DI ROMA

Problema dell'Ordinamento

Il *Problema dell'Ordinamento* è un problema fondamentale:

- Si presenta in innumerevoli situazioni
- Una soluzione efficiente ha notevole impatto su altri problemi, tra cui la *ricerca* (v. ricerca binaria)

Problema dell'Ordinamento

Input: Insieme finito $S = \{x_0, \ldots, x_{n-1}\}$ di elementi provenienti da un insieme totalmente ordinato

Output: Sequenza $L = l_0, \ldots, l_{n-1}$ t.c.:

- $\{l_0, \ldots, l_{n-1}\} = S$
- $l_i < l_{i+1}$, $i \in [0, n-1)$

SAPIENZA UNIVERSITÀ DI ROMA

Algoritmi Basati su Approccio Incrementale

Tre algoritmi:

- $SelectionSort$
- $InsertionSort$
- $BubbleSort$

IDEA: se i primi k elementi sono ordinati, estendo l'ordinamento al $(k+1)$-esimo elemento (o al contrario, dall'ultimo al primo)

SAPIENZA UNIVERSITÀ DI ROMA

SelectionSort

Seleziona l'elemento minimo tra quelli rimanenti e lo scambia con quello in posizione $k+1$

SelectionSort

Algoritmo $SelectionSort(Lista\ l)$

for $(k = 0, \ldots, |l| - 2)$ **do**
    $min = k + 1;$
    for $(j = k + 2, \ldots, |l|)$ **do**
        if $(l[j] < l[min])$ **then**
            $min = j;$
    $scambia(l[k + 1], l[min]);$

Ordinamento *in loco*: usa una quantità costante di memoria aggiuntiva rispetto a quella usata per l'input

$SelectionSort$

Theorem

$SelectionSort$ ordina in loco n elementi in tempo $T(n) = \Theta(n^2)$.

Proof.

L'algoritmo esegue

$$T(n) = \Theta(\sum_{k=0}^{n-2} n - k - 1) = \Theta(\sum_{i=1}^{n-1} i) = \Theta((n-1)n/2) = \Theta(n^2)$$

operazioni. □

InsertionSort

Inserisce l'elemento in posizione $k+1$ nella posizione corretta tra i primi k (già ordinati)

InsertionSort

Algoritmo InsertionSort(Lista l)

for $(k = 1, \ldots, |l| - 1)$ **do**
    $x \leftarrow l[k+1]$;
    $j \leftarrow 1$;
    while $(l[j] \leq x$ and $j \leq k)$ **do**
        $j++$;
    if $(j < k+1)$ **then**
        for $(t = k, \ldots, j)$ **do**
            $l[t+1] = l[t]$;
        $l[j] = x$;

$InsertionSort$

Theorem

$InsertionSort$ ordina in loco n elementi in tempo $T(n) = \Theta(n^2)$.

Proof.

L'algoritmo esegue

$$T(n) = \Theta(\sum_{i=1}^{n-1} k) = \Theta(n(n-1)/2) = \Theta(n^2)$$

operazioni. □

BubbleSort

Esegue n scansioni della lista, scambiando gli elementi adiacenti che non sono in ordine crescente

BubbleSort

Algoritmo BubbleSort(Lista l)

$scambiato \leftarrow true;$
$k \leftarrow 0;$
while ($scambiato$ and $k < |l| - 1$) **do**
    $scambiato \leftarrow false;$
    for ($j = 1, \ldots, |l| - k - 1$) **do**
        if ($l[j] > l[j+1]$) **then**
            $scambia(l[j], l[j+1]);$
            $scambiato \leftarrow true;$
    $k++;$

BubbleSort

Theorem

$BubbleSort$ ordina in loco n elementi in tempo $T(n) = \Theta(n^2)$.

Proof.

L'algoritmo esegue

$$T(n) = \Theta(\sum_{k=0}^{n-2} n - k - 1) = \Theta(n^2)$$

operazioni. □

$HeapSort$

- Variante di $SelectionSort$
- Ricerca efficiente del minimo (massimo) tra i restanti $n - k$ elementi
- Usa una nuova struttura dati: *Heap*

SAPIENZA UNIVERSITÀ DI ROMA

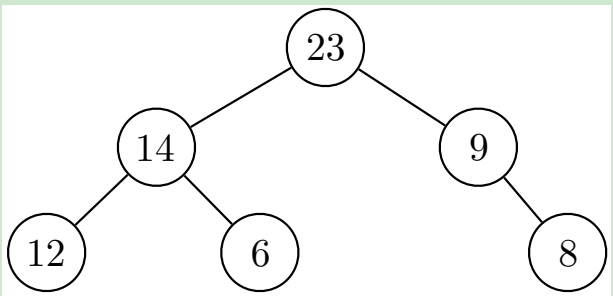
Heap

Definition (Struttura dati $Heap$)

Un *Heap* H associato ad un insieme finito L di elementi (su cui è definita una relazione di ordine totale $<$) è un albero binario tale che:

- H è *completo* almeno fino al penultimo livello
- Ciascun nodo v contiene un elemento di L, indicato con $chiave(v)$, ed ogni elemento è presente in un solo nodo
- Per ogni nodo v e per ogni suo figlio $w \in figli(v)$, si ha $chiave(v) > chiave(w)$

Example



$HeapSort$

$HeapSort$ procede essenzialmente come $SelectionSort$, ordinando dall'ultimo al primo elemento, usando un heap per individuare facilmente il valore massimo tra quelli non ancora considerati

$HeapSort$: panoramica

crea un heap H con gli elementi da ordinare come chiavi;
$X \leftarrow$ coda vuota;
while (H non vuoto) **do**
    $m =$ massimo di H;
    elimina m da H;
    rendi H nuovamente un heap;
    $X.enqueue(m)$;
return X;

Tutte le operazioni possono essere implementate efficientemente (v. seguito)

SAPIENZA UNIVERSITÀ DI ROMA

Heap: proprietà notevoli

Theorem

Un heap H contenente n nodi ha altezza $\Theta(\log n)$:

Proof.

- Un albero binario completo di altezza h contiene $2^{h-1} - 1$ nodi interni
- Essendo H completo fino al penultimo livello: $2^{h-1} - 1 \leq n \leq 2^h - 1$
- Ovviamente (per $h \geq 2$): $2^{h-2} \leq 2^{h-1} - 1 \leq n \leq 2^h - 1 \leq 2^h$
- Applicando $\log_2$ a tutti i membri: $h - 2 \leq \log_2 n \leq h$
- Pertanto: $h = \mathcal{O}(\log n)$ e $h = \Omega(\log n)$, ovvero: $h = \Theta(\log n)$

□

Heap: proprietà notevoli

Theorem

Il massimo valore tra le chiavi di un heap H è la chiave della radice di H

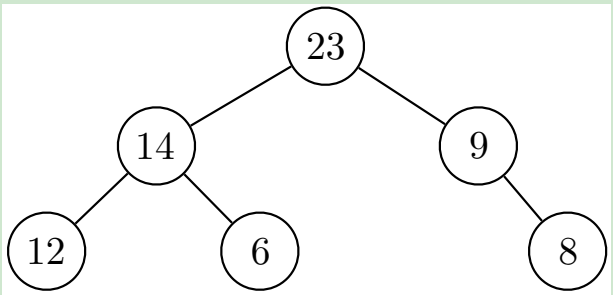
Proof.

Conseguenza immediata della definizione di heap □

Proprietà notevoli

Consideriamo la rappresentazione con vettore posizionale di un heap

Example



1	2	3	4	5	6	7
23	14	9	12	6	–	8

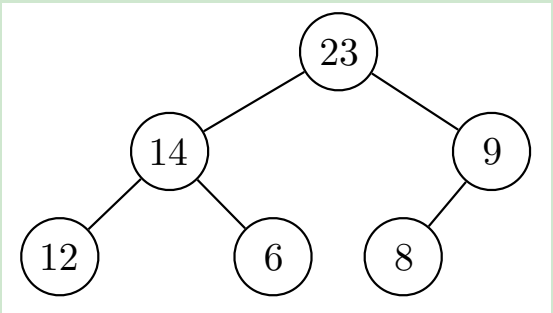
Il vettore contiene più componenti rispetto al numero di nodi, in quanto il nodo con chiave 9 non ha il figlio sx, mentre ha quello dx.

Proprietà notevoli

Definition

Un heap H è detto *con struttura rafforzata* se tutte le foglie dell'ultimo livello sono *compattate* a sinistra

Example



23	14	9	12	6	8
1	2	3	4	5	6

Il vettore contiene tante componenti quante foglie

SAPIENZA UNIVERSITÀ DI ROMA

Proprietà notevoli

Theorem

Il vettore posizionale associato ad un heap H con struttura rafforzata contiene un numero di componenti pari al numero di nodi di H

Proof.

Conseguenza del fatto che, nel vettore posizionale, le foglie dell'ultimo livello occupano posizioni contigue da sx verso dx in fondo al vettore □

SAPIENZA UNIVERSITÀ DI ROMA

Proprietà notevoli

Riassumendo:

- Un heap con n nodi ha altezza $\Theta(\log n)$
- La chiave massima di un heap è contenuta nella radice
- Un heap con struttura rafforzata contenente n nodi può essere rappresentato con un vettore di dimensione n

Queste proprietà permetteranno di implementare efficientemente l'algoritmo e di effettuare l'ordinamento *in loco*

SAPIENZA UNIVERSITÀ DI ROMA

$HeapSort$: rappresentazione dei dati

- Rappresentiamo l'input mediante un vettore di $Elem$
- (Assumiamo che il vettore di input contenga elementi distinti)
- Per gli alberi (e per l'heap) usiamo una rappresentazione mediante vettore posizionale:
  - radice in posizione 0
  - figlio sinistro del nodo i in posizione $2i + 1$
  - figlio destro del nodo i in posizione $2i + 2$
- Per gli heap garantiamo sempre la struttura rafforzata

SAPIENZA UNIVERSITÀ DI ROMA

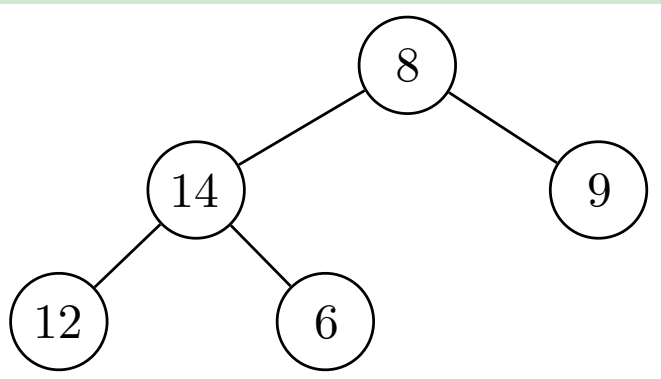
La procedura $fixHeap$

Definition

Chiamiamo *pre-heap* un albero che soddisfa tutte le proprietà di un heap eccetto per la chiave della sua radice, che potrebbe non contenere il valore massimo rispetto alle chiavi dei nodi figlio

In particolare, in un pre-heap, entrambi i sottoalberi della radice sono heap.

Example (pre heap)



La procedura $fixHeap$

Trasformazione di un pre-heap, sottoalbero di H con radice in v, in un heap

$fixHeap$

$Procedura\ fixHeap(Nodo\ v, preHeap\ H)$

if (H vuoto o v foglia) **then**
    return ;
$w \leftarrow$ figlio di v con chiave massima;
if ($chiave(v) < chiave(w)$) **then**
    scambia le chiavi di v e w;
    $fixHeap(w, H)$;

Nota: Poiché $fixHeap$ non modifica la struttura di H (ma interviene solo sulle chiavi), se H ha una struttura rafforzata prima dell'esecuzione di $fixHeap$, la avrà anche dopo

SAPIENZA UNIVERSITÀ DI ROMA

La procedura $fixHeap$

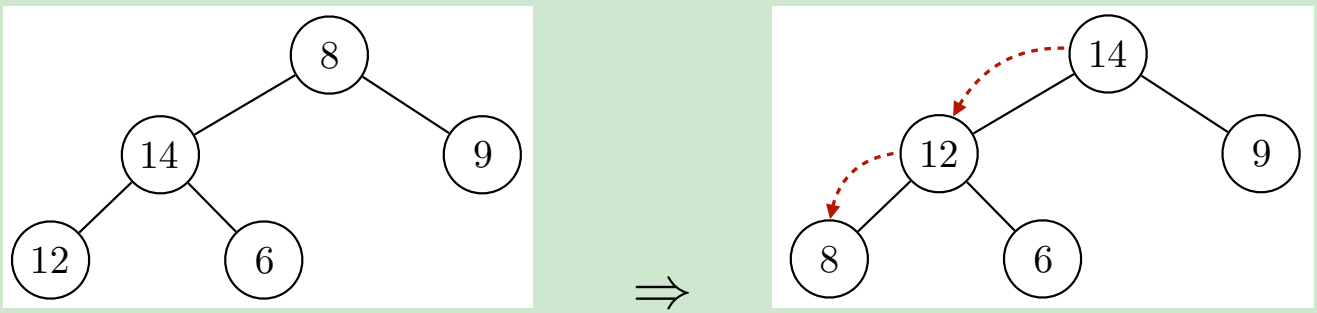
Theorem

La procedura $fixHeap$ esegue $\mathcal{O}(\log n)$ operazioni

Proof.

La procedura scambia, se necessario, la chiave di v con quella di uno dei suoi figli, procedendo ricorsivamente sul sottoalbero avente per radice w. Pertanto, il numero di operazioni è al più pari alla profondità dell'albero, che è $\Theta(\log n)$ (v. proprietà notevoli). □

Example (pre heap)



La procedura $fixHeap$

Si può sfruttare $fixHeap$ per:

- costruire un heap a partire da un albero
- estrarre il massimo dall'heap H e riorganizzare H rendendolo nuovamente un heap

SAPIENZA UNIVERSITÀ DI ROMA

Costruzione dell'heap

$heapify$

$Procedura\ heapify(Albero\ T)$

if (T è vuoto) **then return** ;

$heapify$(sottoalbero sx di T);
$heapify$(sottoalbero dx di T);
$fixHeap(radice\ di\ T, T)$;

Theorem

La procedura $heapify$ esegue $\mathcal{O}(n)$ operazioni

Proof.

Poiché $fixHeap$ ha costo temporale $\mathcal{O}(\log n)$, per $heapify$ abbiamo:

$$T(n) = 2T(n/2) + \mathcal{O}(\log n)$$

Dal teorema Master (caso 1): $T(n) = \mathcal{O}(n)$ □

Estrazione del massimo e riorganizzazione dell'albero

$getMax$

$Algoritmo\ getMax(Heap\ H) \rightarrow Elem$

$r \leftarrow$ radice di H;
$max \leftarrow chiave(r)$;
$v \leftarrow$ foglia di H più profonda e più a destra;
Sovrascrivi $chiave(r)$ con $chiave(v)$;
Elimina v;
$fixHeap(H)$;
return max;

Nota: Poiché $getMax$ rimuove sempre la foglia più a destra, la sua invocazione preserva la struttura rafforzata di H

Estrazione del massimo e riorganizzazione dell'albero

Theorem

L'algoritmo $getMax$ esegue $\mathcal{O}(\log n)$ operazioni

Proof.

Poiché H è rappresentato con vettore posizionale, la foglia più a destra è l'ultimo elemento del vettore. Pertanto la sua estrazione ha costo costante. Anche tutte le altre operazioni hanno costo costante, eccetto $fixHeap$ che ha costo $\mathcal{O}(\log n)$. □

SAPIENZA UNIVERSITÀ DI ROMA

$HeapSort$

$HeapSort$

$Algoritmo\ HeapSort(Array[Elem]\ A) \rightarrow Array[Elem]$

costruisci un albero H con gli elementi di A;
$heapify$(radice di H,H);
$X \leftarrow$ coda vuota; //Coda rappresentata come array di n elementi
while (H non vuoto) **do**
    $m \leftarrow getMax(H)$;
    $X.enqueue(m)$;
return X;

$HeapSort$

Theorem

L'algoritmo $heapSort$ ha costo temporale $\mathcal{O}(n \log n)$

Proof.

Conseguenza delle seguenti osservazioni:

- La costruzione di H ha costo $\mathcal{O}(n)$
- $heapify$ ha costo $\mathcal{O}(n)$
- La costruzione di X ha costo $\mathcal{O}(n)$
- Il ciclo while esegue n iterazioni
- $getMax$ ha costo $\mathcal{O}(\log n)$
- $enqueue$ ha costo costante

□

SAPIENZA UNIVERSITÀ DI ROMA

$HeapSort$

Notiamo che

- $HeapSort$ ha costo temporale $\mathcal{O}(n \log n)$, chiaramente preferibile ad $\mathcal{O}(n^2)$ (algoritmi con approccio incrementale)
- Tuttavia, $HeapSort$ utilizza spazio $\mathcal{O}(n)$, dovuto alle strutture ausiliarie H ed X, di dimensione lineare rispetto all'input, mentre gli altri algoritmi ordinano in loco
- È però possible ordinare in loco anche con $HeapSort$, operando direttamente sull'array di input ed adottando particolari accorgimenti

SAPIENZA UNIVERSITÀ DI ROMA

$MergeSort$

Approccio *Divide et Impera*:

- (Divide) Dividi l'array in due sottoarray di dimensione bilanciata ed ordinali separatamente
- (Impera) Fondi i due array ordinati in un nuovo array ordinato

$MergeSort$

$Algoritmo\ MergeSort(Array\ A) \rightarrow\ Array$

$n = |A|;$
if $(n <= 1)$ **then return** $A;$
$A_1 \leftarrow MergeSort(A[1, n/2]);$
$A_2 \leftarrow MergeSort(A[n/2 + 1, n]);$
return $merge(A_1, A_2);$

$MergeSort$

merge

$Algoritmo\ merge(ArrayA, B) \rightarrow ArrayOrdinato$

X : array di dimensione $|A| + |B|$;
$n \leftarrow |A|$,$m \leftarrow |B|$;
$i \leftarrow 1$, $j \leftarrow 1$, $k \leftarrow 1$;
while $(i \leq n$ and $j \leq m)$ **do**
    if $(A[i] < B[j])$ **then**
        $X[k] \leftarrow A[i]$;
        $i++$;
    else
        $X[k] \leftarrow B[j]$;
        $j++$;
    $k++$;
if $(i > n)$ **then**
    appendi $B[j, m]$ in fondo ad X;
else
    appendi $A[i, n]$ in fondo ad X;
return X;

$MergeSort$

Theorem

$MergeSort$ ordina n elementi in tempo $T(n) = \Theta(n \log n)$.

Proof.

$merge$ esegue la fusione in tempo $\Theta(|A| + |B|)$ (si ricordi che A e B sono ordinati)
Pertanto $MergeSort$ ha costo temporale: $T(n) = 2T(n/2) + \Theta(n)$
Dal teorema Master (caso 2), abbiamo: $T(n) = \Theta(n \log n)$ □

QuickSort

Approccio *Divide et Impera*:

- (Divide) Scegli un elemento p (pivot) nell'array di input A e crea due array A_1 e A_2 contenenti, rispettivamente, i valori minori o uguali di p e maggiori di p
- (Impera) Ordina A_1 e A_2 ricorsivamente e inserisci in A l'array $A_1 p A_2$

QuickSort

Algoritmo $QuickSort(Array\ A)$

scegli un elemento p di A;
$A_1 = \{y \in A : y < p\}$;
$A_2 = \{y \in A : y > p\}$;
if $(|A_1| > 1)$ **then** $quickSort(A_1)$;
if $(|A_2| > 1)$ **then** $quickSort(A_2)$;
copia la concatenazione $< A_1, p, A_2 >$ in A;

$QuickSort$

Caso peggiore: ad ogni invocazione, il pivot è il minimo o il massimo tra i valori dell'array da ordinare.

Nota: consideriamo solo il numero di confronti effettuati dall'algoritmo.

Theorem (Complessità nel caso peggiore)

$QuickSort$ ordina, nel caso peggiore, n elementi in tempo $T(n) = \mathcal{O}(n^2)$.

Proof.

Abbiamo: $T(n) = n - 1 + T(n-1)$, ovvero, risolvendo per iterazione: $T(n) = \mathcal{O}(n^2)$. □

Pertanto, analizzando la complessità nel caso peggiore, $QuickSort$ non è migliore di $HeapSort$ o di $MergeSort$.

SAPIENZA UNIVERSITÀ DI ROMA

QuickSort

Scegliendo randomicamente il pivot ad ogni chiamata, riduciamo la probabilità di essere nel caso peggiore.

Theorem (Complessità nel caso medio)

$QuickSort$ ordina n elementi in un numero atteso di operazioni $T(n) = \mathcal{O}(n \log n)$.

Proof.

Assumendo che il perno sia scelto con probabilità uniforme, abbiamo che ogni partizione ha probabilità $1/n$ di verificarsi. Il valore atteso del numero di confronti è quindi: $T(n) = 1/n \sum_{i=1}^{n} n - 1 + T(n-i) + T(i-1)$, da cui: $T(n) = n - 1 + 1/n \sum_{i=1}^{n} T(n-i) + T(i-1) =$
$n - 1 + \sum_{i=1}^{n} 2/n T(i-1) = n - 1 + 2/n \sum_{i=0}^{n-1} T(i)$

L'equazione $T(n) = n - 1 + 2/n \sum_{i=0}^{n-1} T(i)$ ha soluzione $T(n) = \mathcal{O}(n \log n)$ (dimostrazione omessa).

□

$QuickSort$

Nella pratica, $QuickSort$ mostra prestazioni migliori di $MergeSort$ e $HeapSort$ in quanto evita scambi non necessari (tra elementi già ordinati), con significativa riduzione del tempo d'esecuzione

SAPIENZA UNIVERSITÀ DI ROMA

Delimitazione inferiore al costo dell'ordinamento

È possibile migliorare la complessità degli algoritmi visti finora?

In particolare, è possibile ordinare n oggetti in meno di $\mathcal{O}(n \log n)$?

SAPIENZA UNIVERSITÀ DI ROMA

Modello basato su confronti

- Gli algoritmi visti finora non sfruttano nessuna informazione sul dominio di provenienza degli elementi
- L'unica operazione disponibile è il confronto tra due elementi
- Pertanto, tali algoritmi effettuano l'ordinamento eseguendo il confronto come unica operazione di dominio
- Stabiliremo un limite inferiore al costo per gli algoritmi *basati su confronti*

SAPIENZA UNIVERSITÀ DI ROMA

Alberi di decisione

Definition (Albero di decisione)

Un albero di decisione per un insieme V di variabili (a valori da un insieme L totalmente ordinato) è un albero binario in cui:

- Ogni nodo interno contiene una coppia $\langle v, w \rangle$ di variabili di V da confrontare
- Il figlio sx di un nodo n contiene i prossimi elementi da confrontare nel caso in cui $v \leq w$
- Il figlio dx di un nodo n contiene i prossimi elementi da confrontare nel caso in cui $v > w$
- i nodi foglia contengono l'ordinamento indotto dall'esito dei confronti radice-foglia

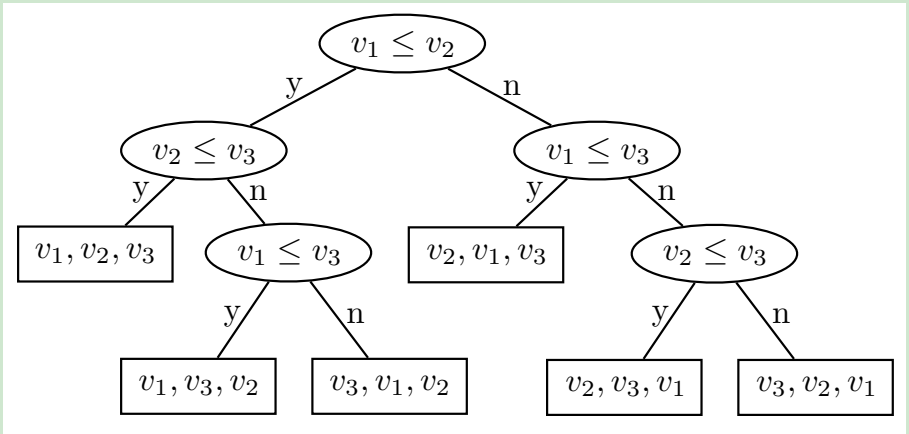
La strategia seguita da un algoritmo di ordinamento (basato su confronti) può essere catturata da un albero di decisione

SAPIENZA UNIVERSITÀ DI ROMA

Alberi di decisione

Albero di decisione

Il seguente albero di decisione rappresenta un possibile algoritmo di ordinamento per tre elementi v_1, v_2, v_3



- Ogni cammino radice-foglia rappresenta un'esecuzione dell'algoritmo per una particolare configurazione dell'input
- Ogni foglia rappresenta l'ordinamento ottenuto

Alberi di decisione

- Il cammino radice-foglia più lungo rappresenta l'esecuzione nel caso peggiore
- Pertanto, la profondità di un albero di decisione corrisponde al costo dell'algoritmo nel caso peggiore
- Determiniamo tale profondità

SAPIENZA UNIVERSITÀ DI ROMA

Profondità di un albero di decisione

Theorem

Un albero di decisione associato ad un algoritmo di ordinamento basato su confronti ha profondità $\Omega(n \log n)$.

Proof.

Consideriamo un generico algoritmo di ordinamento A basato su confronti

- Per input di n elementi esistono $n!$ possibili ordinamenti
- Quindi l'albero di decisione associato ad A dovrà contenere almeno $n!$ foglie
- Un albero di altezza h contiene al più 2^{h-1} foglie, quindi deve essere: $2^{h-1} \geq n!$
- Poiché $n! \approx (2\pi n)^{\frac{1}{2}}(n/e)^n$ (approssimazione di Stirling), abbiamo: $h \geq \log_2((2\pi n)^{\frac{1}{2}}(n/e)^n)+1 = \frac{1}{2}\log_2(2\pi)+\frac{1}{2}\log_2 n - n\log_2 e + n\log_2 n + 1$, da cui:

$$h = \Omega(n \log n)$$

□

Delimitazione inferiore al costo dell'ordinamento

Il teorema precedente ci dice che

> *Ogni algoritmo di ordinamento basato su confronti esegue, nel caso peggiore,* $\Omega(n \log n)$ *confronti*

Concludiamo pertanto che

Il costo di un algoritmo di ordinamento è $\Omega(n \log n)$

(Nota: risultato valido *solo nel caso del modello basato su confronti*)

Come importante conseguenza, abbiamo che gli algoritmi $HeapSort$ e $MergeSort$ sono *ottimali dal punto di vista della complessità nel caso peggiore*

SAPIENZA UNIVERSITÀ DI ROMA

Algoritmi di ordinamento che sfruttano operazioni di dominio

Example

- Supponiamo di dover ordinare n interi distinti nell'intervallo $[1, n]$
- Poiché conosciamo il dominio degli interi, sappiamo che l'unica sequenza possible è $1, 2, \ldots, n$
- Possiamo quindi realizzare un algoritmo che determina l'ordinamento *in tempo costante* (escludendo il tempo per costruire l'output), senza nemmeno considerare l'ordine degli elementi in input
- Questo algoritmo non è basato su confronti (sfrutta proprietà degli interi e non esegue confronti), per cui la delimitazione inferiore al costo stabilita precedentemente non è applicabile

SAPIENZA UNIVERSITÀ DI ROMA

$IntegerSort$

Ordinamento di n interi nell'intervallo $[1, k]$

$Algoritmo\ IntegerSort(Intero\ k, Array[Intero]\ X)$

Y : array di $Intero$ di dimensione k;
for $(i = 1, \ldots, k)$ **do** $Y[i] \leftarrow 0$;
for $(i = 1, \ldots, n)$ **do** $Y[X[i]] \leftarrow Y[X[i]] + 1$;
$j \leftarrow 1$;
for $(i = 1, \ldots, k)$ **do**
    while $(Y[i] > 0)$ **do**
        $X[j] \leftarrow i$;
        $j = j + 1$; $Y[i] = Y[i] - 1$;

- $Y[i]$ rappresenta il numero di occorrenze di i nell'input X

$IntegerSort$

Theorem

L'algoritmo $IntegerSort$ *ordina* n *interi nell'intervallo* $[1, k]$ *in tempo* $T(n) = \mathcal{O}(n + k)$

Proof.

- $\mathcal{O}(k)$ per creare ed inizializzare Y
- $\mathcal{O}(n)$ per scandire X e popolare Y
- $\mathcal{O}(n + k)$ per scandire Y e popolare X con gli elementi in ordine

□

- Se $k = \mathcal{O}(n)$, l'algoritmo è lineare
- Se k cresce più rapidamente di n, ad es., $k = \Theta(2^n)$, il costo segue k
- (In generale, non è possibile dire nulla a priori)

SAPIENZA UNIVERSITÀ DI ROMA

$BucketSort$

- $IntegerSort$ può essere facilmente adattato all'ordinamento di n record con chiavi intere nell'intervallo $[1, k]$
- È sufficiente usare, al posto di un array di k interi, un array Y di k liste di record
- $Y[i]$ conterrà la lista dei record che hanno chiave i

SAPIENZA UNIVERSITÀ DI ROMA

$BucketSort$

$Algoritmo\ BucketSort(Array[Record]\ X, Intero\ k)$

Y : array di dimensione k di liste di $Record$;
for $(i = 0, \ldots, k)$ **do** $Y[i] \leftarrow$ lista vuota;
for $(i = 0, \ldots, n)$ **do** appendi il record $X[i]$ alla lista $Y[chiave(X[i])]$;
for $(i = 1, \ldots, k)$ **do**
    copia gli elementi di $Y[i]$ in X, sequenzialmente e mantenendone l'ordine

Dall'analisi di costo di $IntegerSort$ segue immediatamente il seguente risultato

Theorem

L'algoritmo $BucketSort$ ordina n record con chiavi intere nell'intervallo $[1, k]$ in tempo $T(n) = \mathcal{O}(n + k)$

Proprietà di *Stabilità*

Definition (Stabilità di un algoritmo di ordinamento)

Un algoritmo di ordinamento è detto *stabile* se preserva l'ordine iniziale tra elementi con lo stesso valore (o chiave)

- La stabilità di $BucketSort$ può essere garantita gestendo le liste in Y come code

Molti algoritmi di ordinamento posso essere resi stabili con opportuni accorgimenti

Sfrutteremo questa proprietà a breve

SAPIENZA UNIVERSITÀ DI ROMA

$RadixSort$

- $IntegerSort$ (e $BucketSort$) hanno costo temporale $\mathcal{O}(n+k)$
- Se k è molto più grande di n, il vettore Y sarà *sparso*, ovvero conterrà numerose componenti con il valore 0 (o vuote)
- In tal caso, la scansione di Y richiederà un grande numero di visite a componenti non significative
- Possiamo ridurre il numero di componenti non significative?

Example

- Vogliamo ordinare il seguente array: $\langle 4368, 2397, 5924 \rangle$
- Ordiniamo ripetutamente con $BucketSort$ dalla cifra meno significativa verso quella più significativa
- $\langle 4368, 2397, 5924 \rangle \Rightarrow \langle 5924, 2397, 4368 \rangle \Rightarrow \langle 5924, 4368, 2397 \rangle \Rightarrow \langle 4368, 2397, 5924 \rangle \Rightarrow \langle 2397, 4368, 5924 \rangle$

SAPIENZA UNIVERSITÀ DI ROMA

$RadixSort$: correttezza

Theorem

$RadixSort$ ordina correttamente n valori interi

Proof.

Alla i-esima passata:

- Se due elementi x ed y hanno cifre diverse in posizione i, allora vengono ordinati secondo la i-esima cifra
- Se due elementi x ed y hanno stessa cifra in posizione i, il loro ordine viene mantenuto, grazie alla stabilità di $BucketSort$

Pertanto, dopo la i-esima passata, i numeri sono correttamente ordinati secondo le i cifre meno significative $\square$

$RadixSort$: tempo di esecuzione

- Il tempo di esecuzione di $RadixSort$ dipende dal numero ℓ di cifre contenute nell'elemento massimo k tra quelli da ordinare
- Se n è il numero di elementi da ordinare, allora $RadixSort$ esegue ℓ volte $BucketSort$, ciascuna volta su n elementi nell'intervallo $[0, 9]$, ottenendo quindi $T(n) = \mathcal{O}(\ell(n + 10))$
- È facile vedere che $\ell \leq \log_{10} k + 1$, quindi $T(n) = \mathcal{O}((\log_{10} k + 1)(n + 10)) = \mathcal{O}(n \log k)$
- Ma allora, quando $k \geq n$, conviene usare un algoritmo ottimale basato su confronti ($\mathcal{O}(n \log n)$)
- Possiamo migliorare?

SAPIENZA UNIVERSITÀ DI ROMA

$RadixSort$: tempo di esecuzione

- La dimensione ℓ della rappresentazione di un numero dipende dalla base adottata:
$$\ell \leq \log_b k + 1$$
- Con una rappresentazione in base b le cifre variano nell'intervallo $[0, b-1]$, quindi ogni esecuzione di $BucketSort$ impiega tempo $O(n+b)$
- In totale, abbiamo:
$T(n) = \mathcal{O}((\log_b k + 1)(n+b)) = \mathcal{O}((\log_b k)(n+b))$
- Scegliendo $b = \Theta(n)$, e rimpiazzando dunque b con n:
$T(n) = \mathcal{O}((\log_b k)(n+b)) = \mathcal{O}(2n \log_n k) = \mathcal{O}(n \log_n k) =$
$$\mathcal{O}\left(n \frac{\log k}{\log n}\right)$$

RadixSort: tempo di esecuzione

Theorem

RadixSort ordina n numeri nell'intervallo $[0, k]$ in tempo $\mathcal{O}(n(1 + \frac{\log k}{\log n}))$

Proof.

Per il caso $k \geq n$, dall'analisi precedente, abbiamo $T(n) = \mathcal{O}\left(n\frac{\log k}{\log n}\right)$.

Per il caso $k < n$, osserviamo che si devono eseguire almeno n passi per la lettura dell'input. Aggiungendo pertanto n al risultato precedente, otteniamo

$$T(n) = \mathcal{O}\left(n\left(1 + \frac{\log k}{\log n}\right)\right)$$

□