

## Ricorrenza

### Teorema Master

Ricorrenza della forma  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

**Teorema:**

Siano  $a \geq 1$  e  $b > 1$  costanti e sia  $f(n)$  una funzione assegnata.

Sia inoltre  $T(n)$  tale che  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ :

1- se  $f(n) = O\left(n^{\log_b a - \epsilon}\right)$  per qualche  $\epsilon > 0$ , allora  $T(n) = \Theta\left(n^{\log_b a}\right)$

2- se  $f(n) = \Theta\left(n^{\log_b a}\right)$ , allora  $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$

3- se  $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$  per qualche  $\epsilon > 0$  e se  $a f\left(\frac{n}{b}\right) \leq c f(n)$  per qualche  $c < 1$  e per valori di  $n$  sufficientemente grandi, allora  $T(n) = \Theta(f(n))$

Oss:

Il 2° caso puo' essere generalizzato:

2- se  $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$ , con  $k \geq 0$ , allora  $T(n) = \Theta\left(n^{\log_b a} f(n)\right)$

## Heapsort

L'Heapsort è un algoritmo che lavora sul posto e con complessità  $\Theta(n \log n)$ . È basato sulla struttura dati Heap, per la gestione efficiente di code di priorità.

### Heap:

È una struttura dati basata su alberi binari quasi completi soddisfacenti una proprietà dell'Heap:

#### Proprietà Del Max-Heap:

Il valore di un nodo è minore o uguale al valore del padre.

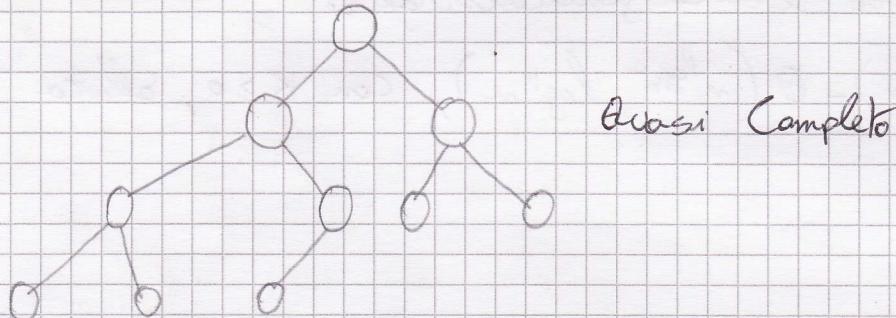
#### Proprietà Del Min-Heap:

Il valore di un nodo è maggiore o uguale al valore del padre.

#### Alberi Binari Quasi Completi:

Sono alberi binari posizionali tali che:

- Tutti i livelli, salvo eccezione possibilmente dell'ultimo, sono completi.
- Nell'ultimo livello tutte le foglie sono addossate a sinistra.



Questi alberi binari vengono rappresentati in maniera efficiente mediante un array

- A.length (Lunghezza Array)
- A.heap-size (Dimensioni Dell'Heap)

$A[1]$  è la radice dell'Heap

Egli del nodo  $i$ :

- $\text{Left}(i) = 2i$

- $\text{Right}(i) = 2i + 1$

Padre del nodo  $i$ :

- $\text{Parent}(i) = \lfloor \frac{i}{2} \rfloor \leftarrow \text{FLOOR}$

Proprietà Max-Heap:

$$1 < i \leq A[\text{heap-size}] \Rightarrow A[\text{Parent}(i)] \geq A[i]$$

Proprietà:

- Il più grande elemento in un Max-Heap è nella radice
- Il più piccolo elemento in un Min-Heap è nella radice
- L'altezza di un nodo è il numero di vertici nel cammino semplice più lungo dal nodo fino ad una foglia.
- L'altezza di un heap è l'altezza della radice.

Proprietà:

L'altezza di un Heap con  $n$  elementi è  $\lceil \log n \rceil$   
(Dunque  $\Theta(\log n)$ )

Dim:

Sia  $h$  l'altezza di un Heap con  $n$  elementi. Si ha:

$$n = \sum_{i=0}^h n_i = \sum_{i=0}^{h-1} 2^i + n_h = 2^h - 1 + n_h$$

$$2^h \leq 2^h - 1 + n_h \leq 2^h - 1 + 2^h = 2^{h+1} - 1$$

Cioè:  $2^h \leq n < 2^{h+1}$

$$h \leq \log n < h+1$$

$$h = \lceil \log n \rceil$$

**Lemma:**

Se  $2^{\lfloor \log_2 n \rfloor} \leq i \leq n$ , il nodo  $i$  è una foglia.

**Dim:**

Sia  $h$  l'altezza di un Heap con  $n$  elementi.

Si osservi che i nodi a livello  $h = \lfloor \log_2 n \rfloor$  sono foglie. Questi hanno un indice  $i$  tale che:

$$n \geq i \geq \sum_{i=0}^{h-1} 2^i + 1 = 2^h - 1 + 1 = 2^h = 2^{\lfloor \log_2 n \rfloor}$$

**Procedure:** Max-Heapify

**Input:**

Un array  $A$  e un indice  $1 \leq i \leq A.length$  tali che gli alberi con radici  $\text{Left}(i)$  e  $\text{Right}(i)$  siano Max-Heap.

**Output:**

Una permutazione dell'array  $A$  tale che l'albero con radice  $i$  sia un Max-Heap

**Costruzione Di Un Heap**

- Le foglie godono della proprietà Max-Heap
- Se  $K$  è il massimo degli indici dei nodi interni, le chiamate Max-Heapify ( $A, K$ )

Max-Heapify ( $A, K-1$ )

⋮

Max-Heapify ( $A, 1$ )

Consentono di propagare la proprietà Max-Heap anche ai nodi  $K, K-1, \dots, 2, 1$ .

## Algoritmo

Heapsort(A)

Build-Max-Heap(A)

for  $i = A.length$  down to 2

exchange  $A[l]$  with  $A[i]$

$A.heap-size = A.heap-size - 1$

Max-Heapify(A, l)

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

Build-Max-Heap(A)

$A.heap-size = A.length$

for  $i = \lfloor A.length/2 \rfloor$  down to 1

Max-Heapify(A, i)

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

Max-Heapify(A, i)

$l = \text{Left}(i)$

$r = \text{Right}(i)$

if  $l \leq A.heap-size$  and  $A[l] > A[i]$

    largest = l

else largest = i

if  $r \leq A.heap-size$  and  $A[r] > A[\text{largest}]$

    largest = r

if largest  $\neq i$

    exchange  $A[i]$  with  $A[\text{largest}]$

    Max-Heapify(A, largest)

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

## Quicksort

E' basato sul paradigma Divide-Et-Impera.

Per ordinare un sottovettore  $A[p \dots r]$ , l'algoritmo Quicksort esegue i seguenti passaggi:

### Divide:

Partitiona  $A[p \dots r]$  in due sottovettori  $A[p \dots q-1]$  e  $A[q+1 \dots r]$  tali che  $A[i] \leq A[q] \leq A[s]$ , per ogni  $p \leq i \leq q-1$ ,  $q+1 \leq s \leq r$  calcolando  $q$  ( $A[q]$  è detto PIVOT)

### Impera:

Ordina ricorsivamente  $A[p \dots q-1]$  e  $A[q+1 \dots r]$

### Combina:

Poiché Quicksort opera sul posto, alla fine  $A[p \dots r]$  risulta già ordinato.

### Algoritmo

Quicksort( $A, p, r$ )

if  $p < r$

$q = \text{Partition}(A, p, r)$

    Quicksort( $A, p, q-1$ )

    Quicksort( $A, q+1, r$ )

### Complessità

Caso peggiore:  $\Theta(n^2)$

Caso migliore:  $\Theta(n \lg n)$

Partition( $A, p, r$ )

$x = A[c]$

$i = p - 1$

    for  $s = p$  to  $r - 1$

        if  $A[s] \leq x$

$i = i + 1$

            exchange  $A[i]$  with  $A[s]$

    exchange  $A[i+1]$  with  $A[r]$

    return  $i + 1$

## Counting Sort

Dato un array  $A[1 \dots n]$ , presuppone che ciascuno degli elementi  $A[i]$  da ordinare sia un numero intero compreso nell'intervallo  $[0, K]$  per qualche intero  $K$ . Lo complessità è  $\Theta(n + k)$  (dunque se  $K = O(n)$  la complessità sarà  $O(n)$ )

Per ogni  $0 \leq i \leq K$ , conta il numero di elementi in  $A$  minori o uguali ad  $i$ , determinando così il range degli elementi di  $A$

## Algoritmo

Counting-Sort( $A, B, K$ )

let  $C[0 \dots K]$  be a new array

for  $i = 0$  to  $K$

$C[i] = 0$

$\Theta(K)$

for  $s = 1$  to  $A.length$

$C[A[s]] = C[A[s]] + 1$

$\Theta(n)$

for  $i = 1$  to  $K$

$C[i] = C[i] + C[i-1]$

$\Theta(K)$

for  $s = A.length$  down to 1

$B[C[A[s]]] = A[s]$

$\Theta(n)$

$C[A[s]] = C[A[s]] - 1$

Totale  $\Theta(n+k)$

se  $K = O(n)$ , allora  $\Theta(n+k) = \Theta(n)$

Oss:

E' un algoritmo stabile, cioè i numeri con lo stesso valore si presentano nell'array di output nello stesso ordine in cui si trovavano nell'array di input

## Radix Sort

Il Radix Sort ordina rispetto a sequenze di chiavi

Vi sono due approcci all'ordinamento rispetto a più chiavi:

- Ordinamento delle chiavi più significative a quelle meno significative (Ricorsivo)

**PROBLEMA:** Alto numero di pile intermedie da gestire.

- Ordinamento delle chiavi meno significative a quelle più significative (Con Algoritmo Di Ordinamento Stabile)  $\Rightarrow$  Radix-Sort

**Algoritmo:**

Radix-Sort( $A, d$ )

for  $i = 1$  to  $d$

Use a stable sort to sort array  $A$  on digit  $i$

Dati  $n$  numeri di  $d$  cifre, dove ogni cifra può avere sino a  $K$  valori, la procedura Radix-Sort ordina correttamente i numeri nel tempo  $\Theta(d(n+K))$  se l'ordinamento stabile utilizzato dalla procedura ha complessità  $\Theta(n+K)$

Se  $d = \Theta(1)$  e  $K = \Theta(n)$ , la complessità di RadixSort è  $\Theta(n)$

## Bucket Sort

Il Bucket Sort assume che i valori da ordinare siano numeri reali random nell'intervallo  $[0, 1]$  con distribuzione uniforme.

In tal caso il Bucket Sort ha un tempo medio di esecuzione  $\Theta(n)$ .

Per ordinare un array  $A[1..n]$ , Bucket Sort divide l'intervallo  $[0, 1]$  in  $n$  sottointervalli uguali  $(0, \frac{1}{n}), (\frac{1}{n}, \frac{2}{n}), \dots, (\frac{n-1}{n}, 1)$  e poi distribuisce gli  $n$  input nei Bucket.

I Bucket, mantenuti come liste, sono ordinati e concatenati.

### Algoritmo:

Bucket-Sort( $A$ )

```
case  
    Cattivo:  
        let  $B[0..n-1]$  be a new array  
         $n = A.length$   
        for  $i = 0$  to  $n-1$   
            make  $B[i]$  an empty list  
            for  $j = i$  to  $n$   
                insert  $A[j]$  into list  $B[\lfloor nA[j] \rfloor]$   
        for  $i = 0$  to  $n-1$   
            sort list  $B[i]$  with insertion sort  
        concatenate the list  $B[0], B[1], \dots, B[n-1]$  together in order  
    Caso Peggior:  $\Theta(n)$   
    Caso Miglior:  $\Theta(n^2)$ 
```

Caso Peggior: Tutti gli elementi confluiscono in un unico Bucket

Caso Miglior: Ciascun Bucket riceve esattamente un elemento

## Hashing

Rappresentazione Redondante Tavole Ad Indirizzi Diretto

$$U = \{0, 1, \dots, m-1\} \quad (\text{Universo Delle Chiavi})$$

Rappresentiamo un insieme  $A$  di elementi le cui sono in  $U$  mediante un array  $T_A[0, \dots, m-1]$  tale che

$$x \in A \Rightarrow T_A[x.\text{Key}] = x$$

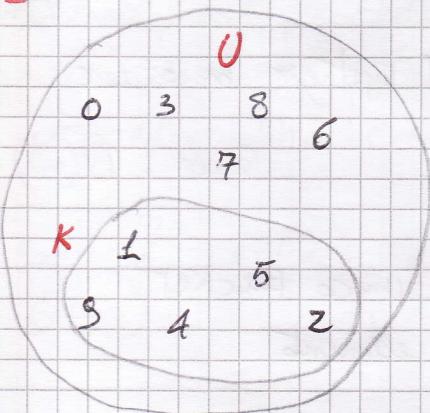
$$x \notin A \Rightarrow T_A[x.\text{Key}] = \text{Nil}$$

Le chiavi debbono essere a due a due distinte

### Vettori Di Bit

In caso speciale si ha nella rappresentazione di insiemi di chiavi (senza dati satelliti). In tal caso si possono utilizzare array  $T[0, \dots, m-1]$  di Bit

Es.



0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	0	0	1

Tale rappresentazione supporta in maniera abbastanza efficiente anche le operazioni di:

- Unione
- Intersezione
- Differenza Insiemistica

## Tabelle Hash

Le tabelle hash risolvono il problema della rappresentazione di insiemi dinamici quando:

- L'universo  $U$  delle possibili chiavi è grande (Anche infinito) e quindi risulta probitivo allocare un array  $T$  di  $|U|$  componenti.
- La dimensione dell'insieme da rappresentare è piccola

Viene allocato una Tabella Hash di dimensione  $m$  confrontabile con quella dell'insieme che si intende rappresentare.

Si stabilisce una opportuna funzione Hash

$$h: U \rightarrow \{0, 1, \dots, m-1\}$$

La tabella hash viene utilizzata come una tabella ad indirettamente diretto, filtrando le chiavi mediante la funzione hash

Se  $|U| > |T|$  è inevitabile che si possa verificare il problema delle collisioni, cioè che vi siano chiavi  $K_1 \neq K_2$  tali che  $h(K_1) = h(K_2)$

Vedremo due soluzioni al problema delle collisioni:

- Tabelle Hash con Concatenamento
- Tabelle Hash ad indirettamente aperto

## Tavole Hash Con Concatenamento

Si pongono tutti gli elementi associati ad una stessa cella in una lista concatenata.

## Analisi Probabilistica Dell'Hashing Con Concatenamento

L'analisi sarà effettuata sotto la seguente ipotesi sulla funzione hash  $h$ :

Ipotesi Di Hashing Uniforme Semplice:

$$\forall i \in \{0, 1, \dots, m-1\}, \Pr\{h(x) = i\} = \frac{1}{m}$$

Per  $S = 0, 1, \dots, m-1$ , sia  $n_S = \text{lunghezza } T[S]$

Si ha:  $n = n_0 + n_1 + \dots + n_{m-1}$

$$\text{Quindi } n = E[n] = E[n_0] + E[n_1] + \dots + E[n_{m-1}]$$

E poiché  $E[n_0] = E[n_1] = \dots = E[n_{m-1}]$ , si ottiene

$$E[n_3] = \frac{n}{m}$$

Chiamiamo fattore di carico della tavola T il rapporto

$$\lambda = \frac{n}{m}$$

Faremo l'ipotesi che  $h(K)$  si calcoli in tempo  $O(1)$

### Teorema 1:

In una tavola hash con concatenamento, una ricerca senza successo richiede un tempo  $O(1+d)$  nel caso medio, nell'ipotesi di Hashing Uniforme Semplice

### Teorema 2:

In una tavola hash con concatenamento, una ricerca con successo richiede un tempo  $O(1+d)$  nel caso medio, nell'ipotesi di Hashing Uniforme Semplice

### Interpretazione Dell'Analisi Di Complessità

Se  $n = O(m)$ , si ha  $d = \frac{n}{m} = \frac{O(m)}{m} = O(1)$ . Quindi se si sceglie  $m$  proporzionale ad  $n$ , la ricerca con o senza successo richiede in media tempo  $O(1)$

### Funzioni Hash

Una buona funzione hash deve soddisfare approssimativamente l'ipotesi di Hashing Uniforme Semplice. Per verificare tale ipotesi sarebbe necessario conoscere la distribuzione di probabilità delle chiavi. Inoltre le estrazioni delle chiavi dovrebbero essere indipendenti l'una dall'altra.

### Interpretazione Delle Chiavi Come Numeri Naturali

Nella maggior parte delle funzioni hash, viene assunto che  $V = \mathbb{N} = \{0, 1, 2, \dots\}$ . Quindi, per utilizzare tali funzioni hash occorre mappare  $V$  in  $\mathbb{N}$  qualcosa,  $V \not\subseteq \mathbb{N}$ .

## Funzioni Hash Con Il Metodo Della Divisione

Sia  $m$  la dimensione della Tabella Hash. Si pone  $h(K) = K \text{ mod } m$ .  
Tale metodo è molto efficiente, ma occorre avere cura di scegliere un  
valore  $m$  che approssimi bene l'ipotesi di Hashing Uniforme Semplice.

## Funzioni Hash Con Il Metodo Della Moltiplicazione

Sia  $0 < A < 1$  una costante fissata. Si pone  $h(K) = \lfloor m(KA \text{ mod } 1) \rfloor$   
(Dove  $KA \text{ mod } 1 = KA - \lfloor KA \rfloor$ ).

La scelta di  $m$  non è critica. Conviene utilizzare il valore

$$A = (\sqrt{5} - 1)/2 = 0,6180339887$$

Tipicamente si sceglie  $m = 2^p$ , ciò rende il calcolo di  $h(K)$  particolarmente efficiente in situazioni in cui, posto  $w$  la dimensione di una parola,  
si ha

- $0 \leq K < 2^w$
- $A = \frac{s}{2^w}$ , con  $0 \leq s < 2^w$
- $p \leq w$



## Hashing Universale

Al fine di evitare che possano esistere insiemi di chiavi che causino sempre  
un alto numero di collisioni, è stato proposto lo schema dell'Hashing Universale,  
che prevede che la funzione hash sia selezionata in maniera random da  
una certa famiglia di funzioni hash, che gode di opportune proprietà.

Sia  $H$  una famiglia di funzioni hash  $h: U \rightarrow \{0, \dots, m-1\}$

Definizione:

$H$  si dice universale se

$$(\forall x, y \in U)(x \neq y \rightarrow |\{h \in H : h(x) = h(y)\}| \leq \frac{|H|}{m})$$

Proprietà:

Sia  $H$  universale, e siano  $x, y \in U$  tali che  $x \neq y$ , allora

$$\Pr \{ h \in H : h(x) = h(y) \} = \frac{|\{h \in H : h(x) = h(y)\}|}{|H|} \leq \frac{1}{m}$$

Cioè, la probabilità di avere una collisione su due elementi  $x$  e  $y$  selezionando  $h$  da  $H$  è minore o uguale alla probabilità di

ottenere una collisione selezionando in maniera casuali i due valori hash su  $x$  e  $y$ .

### Teorema:

Supponiamo che una funzione hash  $h$  sia scelta a caso da una famiglia universale  $H$  di funzioni hash e che sia utilizzata per inserire  $n$  elementi in una tabella hash  $T$  di dimensione  $m$  ove le collisioni sono risolte mediante concatenamento.

Sia  $n_i = |T[i]|$ , per  $i = 0, 1, \dots, m-1$   
allora:

$$E[n_{h(K)}] \leq d = \frac{m}{m} , \text{ se } K \notin T$$

$$E[n_{h(K)}] \geq 1 + d , \text{ se } K \in T$$

### Corollario:

Utilizzando Hashing Universale con concatenamento per eseguire una sequenza di  $N$  operazioni insert, search e delete contenente  $O(m)$  operazioni insert, dove  $m$  è la dimensione di una tabella inizialmente vuota, occorre un tempo medio  $\Theta(N)$ .

### COSTRUZIONE DI UNA CLASSE UNIVERSALE DI FUNZIONI HASH

Sia  $p$  primo tale che  $U \subseteq \mathbb{Z}_p$ , con

$$\mathbb{Z}_p = \{0, 1, \dots, p-1\}$$

Poniamo  $\mathbb{Z}_p^* = \{1, 2, \dots, p-1\}$

Avremo anche  $p > m$ , dove  $m$  è la dimensione della tabella  $T$ .

$\forall a \in \mathbb{Z}_p^*$  e  $b \in \mathbb{Z}_p$ , poniamo:

$$h_{ab}(K) = ((aK + b) \bmod p) \bmod m$$

**Teorema:**

La classe  $H_{pm}$ , con  $m < p$ , è universale

### Tabelle Hash Ad Indirizzamento Aperto

Nelle tabelle hash con concatenamento parte della memoria è impegnata con puntatori.

Si può evitare gli utilizzatori i puntatori mantenendo tutti i dati all'interno della stessa tabella, utilizzando lo schema dell'indirizzamento aperto. In tal caso si avrà  $d \leq L$ .

L'inserimento di un nuovo elemento utilizzerà una sequenza di scansione dipendente dal valore della chiave  $K$ .

La medesima sequenza di scansione dovrà essere utilizzata nella ricerca.

Per generare le sequenze di scansione, saranno utilizzate funzioni hash del tipo:

$$h: U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$$

( $m$  è la dimensione della tabella)

Dato la chiave  $K$ , sarà utilizzata la seguente sequenza:

$$\langle h(K, 0), h(K, 1), \dots, h(K, m-1) \rangle$$

Per aumentare l'efficienza delle tabelle hash ad indirizzo aperto è importante che le sequenze di scansione siano permutazioni di  $\langle 0, 1, \dots, m-1 \rangle$

Hash - Insert ( $T, K$ )

$i = 0$

repeat

$s = h(K, i)$

if  $T[s] == \text{Nil}$

$T[s] = K$

return  $s$

else  $i = i + 1$

Until  $i = m$

error "hash table overflow"

Hash - Search ( $T, K$ )

$i = 0$

repeat

$s = h(K, i)$

if  $T[s] == K$

return  $s$

$i = i + 1$

until  $T[s] == \text{Nil}$  or  $i == m$

return  $\text{Nil}$

Tale schema non supporta la cancellazione in maniera immediata

### Indirizzamento Aperto E Cancellazione

Per non interrompere le sequenze di scansione in caso di cancellazione, si possono marcare come deleted le celle da cui sono state cancellate le chiavi.

L'analisi di complessità dovrà tenere conto delle celle marcate delete per il calcolo corretto del fattore di carico  $\alpha$ .

Hash-Insert' ( $T, K$ )

$i = 0$

repeat

$s = h(K, i)$

if  $T[s] == NIL$  or  $T[s] == 'DELETED'$

$T[s] = K$

return  $s$

else  $i = i + 1$

until  $i = m$

error "hash table overflow"

Nella nostra analisi di complessità faremo l'ipotesi di Hashing Uniforme:

$\forall$  permutazione  $\pi$  di  $\{0, 1, \dots, m-1\}$  si ha

$$P_{\pi} \{ \langle h(K, 0), h(K, 1), \dots, h(K, m-1) \rangle = \pi \} = \frac{1}{m!}$$

### Ispezione Lineare

Sia  $h': U \rightarrow \{0, 1, \dots, m-1\}$  una funzione Hash ausiliaria

Poniamo  $h(K, i) = (h'(K) + i) \bmod m$

Variante:

$$h(K, i) = (h'(K) + c \cdot i) \bmod m$$

con  $c \leq m$  costante prima con  $m$

# sequenze di scansione =  $m \ll m!$

La scansione lineare soffre del problema dell'Agglomerazione Primaria: si tendono a formare lunghe sequenze di celle occupate che rallentano le operazioni di ricerca e di inserimento.

Infatti, la probabilità che venga occupata una cella preceduta da  $i$  celle occupate è  $\frac{i+1}{m}$

Ricerca Senza Successo:  $\frac{1}{2} \left( 1 + \frac{1}{(1-\alpha)^2} \right)^2$

Ricerca Con Successo:  $\frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right)$

### Scansione Quadratica

Sia  $h': V \rightarrow \{0, 1, \dots, m-1\}$  una funzione Hash ausiliaria

Poniamo  $h(K, i) = (h'(K) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$ , dove  $c_1$  e  $c_2$  sono costanti con  $c_2 \neq 0$

Perché le sequenze di scansione risultino permutazioni di  $\{0, 1, \dots, m-1\}$ , i valori di  $c_1$  e  $c_2$  ed  $m$  debbono essere scelti con accuratezza.

# sequenze di scansione =  $m \ll m!$

La scansione quadratica soffre del problema dell'Agglomerazione Secondaria, una forma più lieve dell'Agglomerazione Primaria.

Come scegliere  $m$ ,  $c_1$  e  $c_2$ :

Poniamo  $c_1 = c_2 = \frac{1}{2}$ ,  $m = 2^r$

$$h(K, i) = (h'(K) + \frac{1}{2}i + \frac{1}{2}i^2) \bmod 2^r$$

Verifichiamo che se  $i \neq s$ , con  $0 \leq i, s < 2^r$ , allora

$$h(K, i) \neq h(K, s)$$

Se per assurdo  $h(K, i) = h(K, s)$ , allora

$$(h'(K) + \frac{1}{2}i + \frac{1}{2}i^2) \equiv (h'(K) + \frac{1}{2}s + \frac{1}{2}s^2) \pmod{2^r}$$

$$\frac{1}{2}(i - s) + \frac{1}{2}(i^2 - s^2) \equiv 0 \pmod{2^r}$$

$$2^r \mid \frac{1}{2}(i - s)(i + s + 1) \Rightarrow 2^{r+1} \mid (i - s)(i + s + 1)$$

Poiché  $2^{r+s} \nmid i+s+1$  (in quanto  $L \leq i+s+1 < 2^{r+s}$ )  
e  $i+s+1, i-s$  hanno parità diverse, si ha:

$$2^{r+s} \mid i-s$$

$$\text{Per } 0 \leq i \leq 2^r-1 \quad 0 \leq s \leq 2^r-1 \Rightarrow$$

$$\Rightarrow -(2^r-1) \leq i-s \leq 2^r-1 \Rightarrow |i-s| \leq 2^r-1 < 2^{r+1} \Rightarrow$$

$$\Rightarrow |i-s|=0 \Rightarrow i=s, \text{ Assurdo!}$$

$$\text{Quindi: } i \neq s \Rightarrow h(K, i) \neq h(K, s) \quad \forall K \Rightarrow$$

$\langle h(K, 0), h(K, 1), \dots, h(K, 2^r-1) \rangle$  è una permutazione di  $\langle 0, 1, \dots, 2^r-1 \rangle$

### Doppio Hashing

Siamo  $h_1, h_2$  due funzioni Hash auxiliarie

$$\text{Poniamo } h(K, i) = (h_1(K) + i \cdot h_2(K)) \bmod m$$

Poiché le sequenze di scansione sono permutazioni di  $\langle 0, 1, \dots, m-1 \rangle$  è necessario e sufficiente che  $h_2(K)$  sia primo con  $m, \forall K \in U$ .

Una possibile scelta è:

$$m = 2^r$$

$$h_2: U \rightarrow \{1, 3, 5, \dots, m-1\}$$

In questo caso, il numero di sequenze di scansione è  $m \cdot \frac{m}{2} = \Theta(m^2)$

Un'altra possibile scelta è:

$$m \text{ primo}$$

$$h_2: U \rightarrow \{1, 2, \dots, m-1\}$$

Anche in questo secondo caso il numero di sequenze di scansione è  $\Theta(m^2)$

L'ipotesi di Hashing Uniforme è meglio approssimata con il doppio hashing.

## Analisi Dell' Hashing A Indirizzamento Aperto

Effettueremo un' analisi probabilistica assumendo l' ipotesi di Hashing Uniforme:

A permutazione  $\pi$  di  $\{0, 1, \dots, m-1\}$  si ha:

$$P_{\pi} \{ h(K, 0), h(K, 1), \dots, h(K, m-1) = \pi \} = \frac{1}{m!}$$

L' analisi sarà espressa in termini del fattore di carico  $\alpha = \frac{n}{m}$

### Teorema

Nell' ipotesi di Hashing Uniforme, il numero atteso di ispezioni in una ricerca su un successo in una tabella Hash a indirizzamento aperto con fattore di carico  $\alpha = \frac{n}{m} \leq 1$  è al più  $\frac{1}{1-\alpha}$

### Corollario

L' inserimento di un elemento in una tabella Hash a indirizzamento aperto con fattore di carico  $\alpha$  richiede in media non più di  $\frac{1}{1-\alpha}$  ispezioni, nell' ipotesi di Hashing Uniforme.

### Teorema

Nell' ipotesi di Hashing Uniforme, il numero atteso di ispezioni in una ricerca con successo in una tabella Hash a indirizzamento aperto con fattore di carico  $\alpha = \frac{n}{m} \leq 1$  è al più  $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ , supponendo che ogni chiave nella tabella abbia la stessa probabilità di essere cercata.

## Alberi Rosso - Neri

Gli alberi rosso-neri rappresentano una variante "bilanciata" degli alberi binari di ricerca.

In tali alberi, i nodi contengono i seguenti attributi:

- Left
- Right
- P
- Key
- Color

più eventuali campi satellite.

Se manca un figlio o il padre di un nodo, il corrispondente puntatore conterrà il valore NIL.

Tratteremo tali valori NIL come Foglie Esterne. I rimanenti nodi saranno trattati come Nodi Interni.

Un albero rosso-nero è un albero binario di ricerca che soddisfa le seguenti proprietà:

- Ogni nodo è rosso o nero
- La radice è nera
- Ogni foglia (NIL) è nera
- Se un nodo è rosso, allora entrambi i suoi figli sono neri
- A nodo, tutti i cammini semplici che vanno dal nodo a ciascuna sua foglia (NIL) discendente, contengono lo stesso numero di nodi neri (Proprietà Di Bilanciamento).

### Definizione Di Altezza Di Un Nodo $x$ ( $bh(x)$ )

È il numero di nodi neri in un qualunque cammino da  $x$  (escluso  $x$ ) sino ad una sua foglia (NIL) discendente.

### Lemme

Il sottoalbero radicato in un nodo  $x$  di un albero rosso-nero contiene almeno  $2^{bh(x)} - 1$  nodi interni.

### Lemme

L'altezza di un albero rosso-nero con  $n$  nodi interni è al più  $2 \lg(n+1)$

### Corollario

In un albero rosso-nero con  $n$  nodi interni, le operazioni:

- Search
- Minimum
- Maximum
- Predecessor
- Successor

Potranno essere implementate nel tempo  $O(\lg n)$

### Rotazione

Per ripristinare le proprietà degli Alberi Rosso-Neri a seguito di inserimenti e cancellazioni, si utilizzano operazioni rotazioni che preservano le proprietà degli alberi di ricerca.

Left-Rotate ( $T, x$ )

$$y = x.\text{right}$$

$$x.\text{right} = y.\text{left}$$

$$\text{if } y.\text{left} \neq T.\text{nil}$$

$$y.\text{left}.p = x$$

$$y.p = x.p$$

$$\text{if } x.p == T.\text{nil}$$

$$T.\text{root} = y$$

$$\text{else if } x == x.p.\text{left}$$

$$x.p.\text{left} = y$$

$$\text{else } x.p.\text{right} = y$$

~~$$y.\text{left} = x$$~~

$$x.p = y$$

## Insertments

BB-Insert( $T, z$ )

$y = T.\text{nil}$

$x = T.\text{root}$

While  $x \neq T.\text{nil}$

$y = x$

if  $z.\text{key} < x.\text{key}$

$x = x.\text{left}$

else  $x = x.\text{right}$

$z.p = y$

if  $y == T.\text{nil}$

$T.\text{root} = z$

else if  $z.\text{key} < y.\text{key}$

$y.\text{left} = z$

else  $y.\text{right} = z$

$z.\text{left} = T.\text{nil}$

$z.\text{right} = T.\text{nil}$

$z.\text{color} = \text{RED}$

RB-Insert-Fixup( $T, z$ )

RB-Insert-Fixup( $T, z$ )

While  $z.p.\text{color} == \text{RED}$

if  $z.p == z.p.p.\text{left}$

$y = z.p.p.\text{right}$

if  $y.\text{color} == \text{RED}$

$z.p.\text{color} = \text{BLACK}$

$y.\text{color} = \text{BLACK}$

$z.p.p.\text{color} = \text{RED}$

$z = z.p.p$

else if  $z == z.p.\text{right}$

$z = z.p$

Left-Rotate( $T, z$ )

$z.p.\text{color} = \text{BLACK}$

$z.p.p.\text{color} = \text{RED}$

Right-Rotate( $T, z$ ). $p.p$ )

else (same as them clause with "right" and "left" exchanged)

T. root. color = BLACK

Complessità RB-Insert-Fixup =  $O(\lg n)$

Complessità RB-Insert =  $O(\lg n)$

## Programmazione Dinamica

Caratterizzazione della struttura di una soluzione ottima.

- 1- Si dimostra che una soluzione consiste nel fare una scelta, che lascia uno o più sottoproblemi da risolvere.
- 2- Si suppone temporaneamente di conoscere tale scelta.
- 3- Fatta la scelta, si determinano i sottoproblemi da considerare e lo spazio più piccolo di sottoproblemi risultante.
- 4- Si dimostra che le soluzioni dei sottoproblemi all'interno di una soluzione ottima sono ottime (Tecnica Cut-Paste)

La sottostruttura ottima varia in funzione del tipo di problema in due modi:

- Numero  $n_1$  di sottoproblemi utilizzati in una soluzione ottima
- Numero  $n_2$  di scelte per determinare quali sottoproblemi utilizzare.

Generalmente la complessità di un algoritmo di programmazione dinamica dipende da due fattori:

- Dimensione dello spazio dei sottoproblemi
- Numero di scelte da considerare per ogni sottoproblema.

La programmazione dinamica usa la sottostruttura ottima secondo uno schema

Bottom-Up, cioè:

- Prima: Vengono trovate le soluzioni ottime dei sottoproblemi
- Dopo: Viene trovata una soluzione ottima del problema.

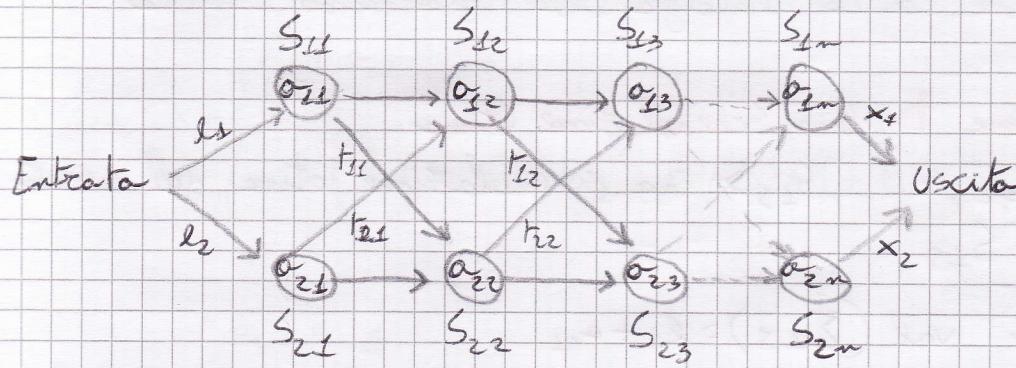
Di solito il costo della soluzione del problema è pari ai costi per risolvere i sottoproblemi più un costo direttamente imputabile alla scelta stessa.

### Ripetizione Dei Sottoproblemi

Lo spazio dei sottoproblemi deve essere "piccolo", nel senso che un algoritmo ricorsivo risolve ripetutamente gli stessi problemi.

Gli algoritmi di programmazione dinamica sfruttano i sottoproblemi ripetuti risolvendo ciascun sottoproblema una sola volta, memorizzando la soluzione in una tabella.

Supponiamo di avere un'industria automobilistica con due diverse linee d'assemblaggio così caratterizzata:



Ciascuna linea è suddivisa in  $n$  stazioni, tali che la  $j$ -esima stazione  $S_{1j}$  della prima linea esegue esattamente le stesse funzioni della  $j$ -esima stazione  $S_{2j}$  della seconda linea,  $1 \leq j \leq n$ .

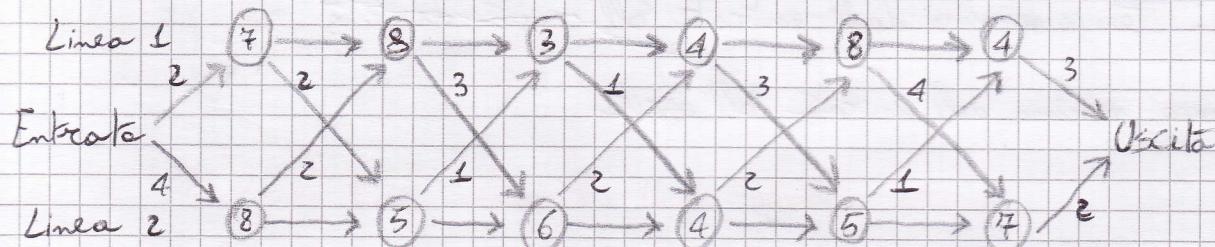
La stazione  $S_{1j}$  completa il suo task in tempo  $\alpha_{1j}$  ( $1 \leq j \leq n$ ,  $1 \leq i \leq n$ )  
Per passare dalla stazione  $S_{1j}$  alla stazione  $S_{1j+1}$  si impiega un tempo trascurabile.

Per passare dalla stazione  $S_{1j}$  a quella  $S_{2j-i}$  si impiega un tempo  $t_{1j}$ .

Per entrare nella linea  $i$  si impiega un tempo  $t_i$ , per uscire dalla linea  $i$  si impiega un tempo  $x_i$ .

### Problema

Il problema è di trovare la schedulazione  $S_{11}, S_{12}, \dots, S_{1n}$ , con  $\alpha_1, \alpha_2, \dots, \alpha_n \in \{1, 2\}$  che impiega minor tempo.



### Soluzione Mediante Forza Bruta

Determinare tutte le possibili ~~selezioni~~ schedulazioni e per ciascuna di esse calcolare il tempo impiegato, per poi selezionare la schedulazione caratterizzata dal tempo minimo.

Le possibili schedulazioni sono  $2^n$ , cioè tante quante le sequenze  $(\alpha_1, \alpha_2, \dots, \alpha_n)$  con  $\alpha_j \in \{1, 2\}$

Il tempo richiesto per calcolare il costo di una schedulazione è  $\Theta(n)$ , quindi la complessità della soluzione mediante forze brute è  $\Theta(n^2)$

### Soluzione Con La Metodologia Della Programmazione Dinamica

#### Basso 1: Caratterizzazione Di Una Schedulazione Ottima

Sia  $\Sigma_{LS} = \langle S_{0,1}, S_{0,2}, \dots, S_{L,S} \rangle$  una schedulazione ottima dell'entrata sull'uscita della stazione  $S_{LS}$

- se  $S=1$  allora  $\text{val}(\Sigma_{LS}) = l_1 + o_{11}$
- se  $S \geq 2$  distinguiamo i casi:
  - $a_{S-1} = 1$
  - $a_{S-1} = 2$

#### Caso $a_{S-1} = 1$

La schedulazione  $\langle S_{0,1}, \dots, S_{1,S-1} \rangle$  è la più veloce del punto di entrata sull'uscita  $S_{1,S-1}$

#### Caso $a_{S-1} = 2$

La schedulazione  $\langle S_{0,1}, \dots, S_{2,S-1} \rangle$  è la più veloce del punto di entrata sull'uscita  $S_{2,S-1}$

In altre parole, una soluzione ottima contiene soluzioni ottime a sottoproblemi, vale cioè la proprietà della sottostruttura ottima.

#### Basso 2: Definizione Ricorsiva Del Valore Di Una Soluzione Ottima

$f_1[S]$ : tempo minimo dall'entrata sull'uscita di  $S_{1,S}$

$f^*$ : tempo minimo dell'entrata sull'uscita

Si ha:

$$f^* = \min (f_1[n] + x_1, f_2[n] + x_2)$$

$$f_1[1] = l_1 + o_{11}$$

$$f_2[1] = l_2 + o_{21}$$

per  $S = 2, 3, \dots, n$  si ha:

$$f_1[S] = \min (f_1[S-1] + o_{1S}, f_2[S-1] + f_{2,S-1} + o_{2S})$$

$$f_2[S] = \min(f_2[S-1] + c_{2S}, f_2[S-1] + t_{2S-1} + c_{2S})$$

E quindi si hanno le seguenti ricorsioni:

$$f_1[S] = \begin{cases} l_1 + c_{1S} & \text{se } S=1 \\ \min(f_1[S-1] + c_{1S}, f_1[S-1] + t_{1S-1} + c_{1S}) & \text{se } 2 \leq S \leq n \end{cases}$$

$$f_2[S] = \begin{cases} l_2 + c_{2S} & \text{se } S=1 \\ \min(f_2[S-1] + c_{2S}, f_2[S-1] + t_{2S-1} + c_{2S}) & \text{se } 2 \leq S \leq n \end{cases}$$

E' anche utile definire i seguenti valori  $l_i[S]$  ed  $l^*$  che consentiranno di costruire schedulazioni ottime ( $i=1, 2, 2 \leq S \leq n$ )

$$l_1[S] = \begin{cases} 1 & \text{se } f_1[S] = f_1[S-1] + c_{1S} \\ 2 & \text{se } f_1[S] = f_1[S-1] + t_{1S-1} + c_{1S} \end{cases}$$

$$l_2[S] = \begin{cases} 2 & \text{se } f_2[S] = f_2[S-1] + c_{2S} \\ 1 & \text{se } f_2[S] = f_2[S-1] + t_{2S-1} + c_{2S} \end{cases}$$

$$l^* = \begin{cases} 1 & \text{se } f^* = f_1[n] + x_1 \\ 2 & \text{se } f^* = f_2[n] + x_2 \end{cases}$$

### Passo 3: Calcolo Del Valore Di Una Soluzione Ottima

Soluzione Ricorsiva Con Memorizzazione

M-OPT( $i, S, \vec{c}, \vec{T}, \vec{e}$ )

if  $\pi[i, S]$  is defined then

return  $\pi[i, S]$

if  $S=1$  then

return  $\pi[i, S] := l_i + c_{i1}$

else

return  $\pi[i, S] := \min(\pi\text{-OPT}(i, S-1, \vec{c}, \vec{T}, \vec{e}) + c_{iS}, \pi\text{-OPT}(3-i, S-1, \vec{c}, \vec{T}, \vec{e}) + t_{iS-1} + c_{iS})$

## Soluzione Bottom-UP

Fastest-Way ( $\vec{c}, \vec{f}, \vec{l}, \vec{x}, n$ )

$$f_1[S] := l_1 + c_{1S}$$

$$f_2[L] := l_2 + c_{2L}$$

for  $S \in \Sigma$  to  $n$  do

if  $f_1[S-1] \leq f_2[S-1] + f_1[S-1]$  then

$$f_1[S] = f_1[S-1] + c_{1S}$$

$$l_1[S] = 1$$

else

$$f_2[S] = f_2[S-1] + f_2[S-1] + c_{2S}$$

$$l_2[S] = 2$$

if  $f_2[S-1] \leq f_1[S-1] + f_2[S-1]$  then

~~$$f_2[S] = f_2[S-1] + c_{2S}$$~~

$$l_2[S] = 2$$

else

$$f_2[S] = f_1[S-1] + f_2[S-1] + c_{2S}$$

$$l_2[S] = 1$$

if  $f_1[x] + x_1 \leq f_2[n] + x_2$  then

$$f^* = f_1[n] + x_1$$

$$l^* = 1$$

else

$$f^* = f_2[n] + x_2$$

$$l^* = 2$$

La complessità della procedura Fastest-Way è  $O(n)$

## Passo 4: Costruzione Di Una Soluzione Ottima

Print - Stations ( $\vec{l}$ ,  $l^*$ , n)

$i = l^*$

print("Linea di assemblaggio", i, "Stazione", n);

for  $s = n$  DownTo 2 do

$i = l_s[s]$

print("Linea di assemblaggio", i, "Stazione", s);

## Il Problema Della Moltiplicazione Di Seguenti Di Matrici

- A matrice  $p \times q$

- B matrice  $q \times r$

(e quindi A e B sono Compatibili)

Prodotto Di Matrici "Righe Per Colonne"

Matrix-Multiply(A, B)

$p = \text{rows}[A]$

$q = \text{columns}[A]$

$r = \text{columns}[B]$

for  $i = 1$  to  $p$  do

    for  $s = 1$  to  $r$  do

$C[i, s] = 0$

        for  $K = 1$  to  $q$  do

$C[i, s] = C[i, s] + A[i, K] \cdot B[K, s]$

return C

Complessità  $O(p \cdot q \cdot r)$

Moltiplicazione Di Seguenti Di Matrici

$A_1 P_0 \times P_1$

$A_2 P_1 \times P_2$

$\vdots$

$A_n P_{n-1} \times P_n$

} Seguenti Di Matrici Compatibili

- A noi interessa calcolare  $A_1 \cdot A_2 \cdot \dots \cdot A_n$

- Il prodotto di matrici è associativo, cioè  $A \cdot (B \cdot C) = (A \cdot B) \cdot C$

## Definizione: Parentesiizzazione Completa Di Una Sequenza Di Matrici

Si dice che un'espressione  $E$  è completamente parentesiizzata se vale una delle seguenti condizioni:

- $E$  è una singola matrice
- $E$  ha la forma  $(E_1 \cdot E_2)$ , dove  $E_1$  ed  $E_2$  sono espressioni completamente parentesiizzate.

## Metodo Esauritivo

La complessità del metodo esauritivo è dominata dal numero di diverse parentesiizzazioni.

$P(n) = \#$  Diverse parentesiizzazioni di una sequenza di  $n$  matrici

$$\begin{cases} P(1) = 1 \\ P(n) = \sum_{i=1}^{n-1} P(i) \cdot (P(n-i)) \end{cases}$$

$n \geq 3$

$$P(n) = \sum_{i=1}^{n-1} P(i) \cdot P(n-i) = 2P(1) \cdot P(n-2) + \sum_{i=2}^{n-2} P(i) \cdot P(n-i) \geq 2P(n-2)$$

$$P(n) \geq 2P(n-2) \geq 2 \cdot 2P(n-2) = 2^2 P(n-2) \geq 2^{n-2} P(2) = 2^{n-2}$$

$$P(n) = \Omega(2^n)$$

## Caratterizzazione Di Una Soluzione Ottima

Sia  $E$  una parentesiizzazione ottima per la sequenza di matrici  $(A_1, A_2, \dots, A_n)$  di dimensioni  $(P_0, P_1, P_2, \dots, P_n)$

Supponiamo  $n \geq 2$

$$E = (E_1 \cdot E_2)$$

con  $E_1$  parentesiizzazione di  $(A_1, \dots, A_K)$

$E_2$  parentesiizzazione di  $(A_{K+1}, \dots, A_n)$

$$1 \leq K \leq n-1$$

Perché

$$\#(E) = \#(E_1) + \#(E_2) + P_0 P_K P_n$$

ne segue che:

- $E_1$  parentesiizzazione ottima di  $(A_1, \dots, A_K)$
- $E_2$  parentesiizzazione ottima di  $(A_{K+1}, \dots, A_n)$

Pertanto la classe dei sottoproblemi da risolvere è data da:

$$\{ (A_i, \dots, A_s) : 1 \leq i \leq s \leq n \}$$

$m[i, s]$  = costo di una soluzione ottima di  $(A_i, \dots, A_s)$

### Definizione Ricorsiva Del Costo Di Una Parentesiizzazione Ottima

$$m[i, s] = \begin{cases} 0 & i = s \\ \min(m[i, k] + m[k+1, s] + p_{i-1} p_k p_s) & i < s \end{cases}$$

### Matrix - Chain - Order (p)

for  $i = 1$  to  $n$  do

$$m[i, i] = 0$$

for  $\Delta = 1$  to  $n-1$  do

for  $i = 1$  to  $n-\Delta$  do

$$s = \Delta + i$$

$$m[i, s] = +\infty$$

for  $K = i$  to  $s-1$  do

$$q = m[i, K] + m[K+1, s] + p_{i-1} p_k p_s$$

if  $q < m[i, s]$  then

$$m[i, s] = q$$

$$s[i, s] = K$$

return  $m, s$

### Matrix - Chain - Multiply (A, s, i, s)

if  $i = s$  then

return  $A$ ;

else

$$X = \text{Matrix - Chain - Multiply}(A, s, i, s[i, s])$$

$$Y = \text{Matrix - Chain - Multiply}(A, s, s[i, s] + 1, s)$$

return  $\text{Matrix - Multiply}(X, Y)$

## La Più Lunga Sottosequenza Comune (LCS)

### Definizione

Una sottosequenza di una sequenza dato  $X$  è una sequenza ottenuta cancellando da  $X$  zero o più elementi.

Più precisamente, se  $X = \langle x_1, x_2, \dots, x_m \rangle$ , una ~~sottosequenza~~  $Z = \langle z_1, z_2, \dots, z_k \rangle$  è una sottosequenza di  $X$  se  $\exists$  una sequenza  $\langle i_1, i_2, \dots, i_k \rangle$  di indici tale che :

- $1 \leq i_1 \leq i_2 \leq \dots \leq i_k \leq m$
- $z_s = x_{i_s}, \forall s=1, 2, \dots, k$

### Definizione

Date due sequenze  $X$  e  $Y$ , diciamo che  $Z$  è una sottosequenza comune di  $X$  e  $Y$  se  $Z$  è una sottosequenza di entrambe le sequenze  $X$  e  $Y$ .

### Problema Della Più Lunga Sottosequenza Comune

DATE DUE SEQUENZE  $X$  E  $Y$  DETERMINARE UNA SOTTOSEQUENZA DI LUNGHEZZA MASSIMA (LCS) CHE SIA COMUNE A  $X$  E  $Y$ .

### Soluzione Mediante Ricerca Esauritiva

È esponenziale in  $\min(|X|, |Y|)$ , in quanto una sequenza di lunghezza  $m$  ha esattamente  $2^m$  sottosequenze.

Il problema della LCS può essere risolto in modo efficiente utilizzando la programmazione dinamica.

### Fase 1: Caratterizzazione Della Più Lunga Sottosequenza Comune

#### Notazione:

Data una sequenza  $X = \langle x_1, x_2, \dots, x_m \rangle$ . Definiamo  $X_i = \langle x_1, x_2, \dots, x_i \rangle$ , per  $i = 0, 1, 2, \dots, m$ . Inoltre  $\forall$  simbolo  $a$  dell'alfabeto poniamo  $X_a = \langle x_1, x_2, \dots, x_m, a \rangle$

#### Teorema: Sottostruzione Ottima Di Uno LCS

Siamo date due sequenze  $X = \langle x_1, x_2, \dots, x_m \rangle$  e  $Y = \langle y_1, y_2, \dots, y_n \rangle$ , tali che  $m \geq 1$  e  $n \geq 1$ , e sia  $Z = \langle z_1, z_2, \dots, z_k \rangle$  una LCS di  $X$  e  $Y$ .

1 - Se  $x_m = y_n$  e

- $z_k = x_m = y_n$  e

•  $Z_{k-1}$  è una LCS di  $X_{m-1}$  e  $Y_{n-1}$

2 - Se  $x_m \neq y_n$ , allora

$Z_k \neq Z_{k-1} \Rightarrow Z$  è una LCS di  $X_{m-1}$  e  $Y$

3 - Se  $x_m \neq y_n$ , allora

$Z_k \neq Z_{k-1} \Rightarrow Z$  è una LCS di  $X$  e  $Y_{n-1}$

### Spatio Dei Sottoproblemi

Dal precedente teorema segue che lo spazio dei sottoproblemi è

$\{(x_i, y_j) : 0 \leq i \leq m, 0 \leq j \leq n\}$ , la cui cardinalità è  $O(mn)$

### Fase 2: Soluzione Ricorsiva

Definiamo  $c[i, s]$ , per  $0 \leq i \leq m$  e  $0 \leq s \leq m$ , come la lunghezza di una LCS di  $x_i$  e  $x_s$  in virtù della sottostante relazione si ha:

$$c[i, s] = \begin{cases} 0 & \text{se } i=0 \text{ o } s=0 \\ c[i-1, s-1] + 1 & \text{se } i, s > 0 \text{ e } x_i = y_s \\ \max(c[i, s-1], c[i-1, s]) & \text{se } i, s > 0 \text{ e } x_i \neq y_s \end{cases}$$

### Fase 3: Calcolo Della Lunghezza di una LCS

LCS-length ( $X, Y$ )

$m = \text{length}[X]$

$n = \text{length}[Y]$

for  $i=1$  to  $m$  do

$c[i, 0] = 0$

for  $s=0$  to  $n$  do

$c[0, s] = 0$

for  $i=1$  to  $m$  do

    for  $s=1$  to  $n$  do

        if  $x_i = y_s$  then

$c[i, s] = c[i-1, s-1] + 1$ ;    $b[i, s] = "R"$

        else if  $c[i-1, s] \geq c[i, s-1]$  then

$c[i, s] = c[i-1, s]$ ;    $b[i, s] = "T"$

        else

$c[i, s] = c[i, s-1]$ ;    $b[i, s] = "L"$

. return  $c, b$

Complessità di LCS- Length :  $\Theta(mn)$

### Fase 4: Costruzione Di Una LCS

Print-LCS(b, X, i, s)

if  $i=0$  o  $s=0$  then  
return

if  $b[i, s] = " \star "$  then

Print-LCS(b, X, i-1, s-1)

else if  $b[i, s] = " \uparrow "$  then

Print-LCS(b, X, i-1, s)

else

Print-LCS(b, X, i, s-1)

Complessità :  $O(m+n)$

## Algoritmi Greedy

Si tratta di algoritmi per problemi di ottimizzazione in grado di costruire una soluzione attraverso una sequenza "Top-Down" di scelte localmente ottime.

Non sempre una strategia greedy porta ad una soluzione ottima.

### Un Problema Di Selezione Di Attività

Sistema Di Attività  $(S, s, f)$

-  $S = \{a_1, a_2, \dots, a_n\}$

Insieme di attività in competizione per l'uso esclusivo di una risorsa

-  $s : \{1, \dots, n\} \rightarrow \mathbb{R}^+$

( $s_i = s(i)$ ) inizio attività

-  $f : \{1, \dots, n\} \rightarrow \mathbb{R}^+$

( $f_i = f(i)$ ) fine attività

Tali che  $s_i < f_i$ , per  $i = 1, \dots, n$

-  $[s_i, f_i]$

Intervallo temporale per cui l'attività  $a_i$  richiede l'uso esclusivo della risorsa.

Due attività  $a_i$  e  $a_j$  sono compatibili se  $[s_i, f_i] \cap [s_j, f_j] = \emptyset$ , cioè:



oppure



$$f_i \leq s_j$$

$$f_j \leq s_i$$

Un sottoinsieme  $A \subseteq S$  è un insieme di attività mutuamente compatibili se ogni coppia di attività distinte  $a_i, a_j$  in  $A$  è costituita da attività compatibili.

Dato un sistema di attività  $(S, s, f)$ , il problema della selezione delle attività consiste nel determinare un sottoinsieme di cardinalità massima  $A \subseteq S$  di attività mutuamente compatibili.

## Soluzione Mediante Ricerca Esauriente

- Si generano tutti i possibili sottoinsiemi  $A \subseteq S$

$$\Omega(2^n)$$

- Si verifichi per ciascuno di essi se si tratta di un insieme di attività mutuamente compatibili

$$O(n^2)$$

- Si determini il sottoinsieme  $A \subseteq S$  di attività mutuamente compatibili di cardinalità massima.

Complessità:  $\Omega(2^n)$

## Studio Di Una Soluzione Ottima

Sia  $A \subseteq S$  una soluzione ottima, sia  $a_i \in A$  (spesso scriveremo  $i \in A$ ),  
 $a_i$  induce i due sottoproblemi:

- $S_i^- = \{k \in S : g_k \leq s_i\}$
- $S_i^+ = \{k \in S : s_k \geq g_i\}$

E' immediato verificare che:

- $A \cap S_i^-$  è una soluzione ottima per  $S_i^-$
- $A \cap S_i^+$  è una soluzione ottima per  $S_i^+$

Cioè vale la proprietà della sottostruzione ottima.

## Studio Dello Spazio Dei Sottoproblemi

$$\begin{aligned} & (S_i^-)_S^- = S_S^- \\ & \exists s \in S_i^- \quad (S_i^-)_S^+ = S_{iS} = \{k \in S : g_S \leq s_k \leq g_K \leq s_i\} \end{aligned}$$

$$\begin{aligned} & (S_i^+)_S^+ = S_S^+ \\ & \exists s \in S_i^+ \quad (S_i^+)_S^- = S_{iS} = \{k \in S : g_i \leq s_k \leq g_K \leq g_S\} \end{aligned}$$

Introducendo due nuove attività di comodo  $a_0, a_{n+1}$ , caratterizzate da  $f_0 = 0$  e  $s_{n+1} > \max f_k$ , possiamo scrivere:  $s_i^+ = s_{0i}$ ,  $s_i^- = s_{i n+1}$ .

Allora è facile verificare che lo spazio dei sottoproblemi rilevati per il problema delle attività è:

$$\{S_{is} : 0 \leq i, s \leq n+1\}$$

Indichiamo con  $c[i, s]$  la cardinalità di una soluzione ottima del problema  $S_{is}$ , si ha:

$$c[i, s] = \begin{cases} 0 & \text{se } S_{is} = \emptyset \\ \max(c[i, k] + c[k, s+1]) & \text{se } S_{is} \neq \emptyset \quad k \in S_{is} \end{cases}$$

Supponiamo che le attività siano ordinate in modo tale che:

$$f_1 \leq f_2 \leq \dots \leq f_n$$

Allora:

$$k \in S_{is} \rightarrow i < k < s \quad s - i - 1 \leq 0$$

da cui:

$$|S_{is}| \leq \max(s - i - 1, 0)$$

In particolare,  $s \leq i+1 \rightarrow S_{is} = \emptyset$

Nell'ipotesi  $f_1 \leq f_2 \leq \dots \leq f_n$ , le cardinalità  $c[i, s]$  possono essere calcolate a partire dalle coppie  $(i, i+1)$  procedendo per valori di  $s - i$  non discreti.

Complessità:  $O(n^3)$

E' possibile convertire la precedente soluzione in una soluzione "Greedy"?

Si riconsidera lo precedente ricorrenza:

$$C[i, S] = \begin{cases} 0 & \text{se } S_{iS} = \emptyset \\ \max(C[i, K] + C[K, S-i]) & \text{se } S_{iS} \neq \emptyset \quad K \in S_{iS} \end{cases}$$

Per applicarlo ad ogni passo occorre "individuare" la scelta giusta di  $K$  fra gli più  $S-i-1$  valori e quindi risolvere due sottoproblemi.

E' possibile "individuare"  $K$  direttamente e ridurre i sottoproblemi ad uno solo? (Scelta Greedy)

Dato  $S_{iS} \neq \emptyset$ , si ponga  $\bar{K} = \min S_{iS}$ . Chiaramente  $S_{i\bar{K}} = \emptyset$ , da cui  $C[i, \bar{K}] = 0$ .

Quindi in corrispondenza di  $\bar{K}$  c'è da risolvere un solo sottoproblema.

Sia  $A'_{iS}$  una soluzione ottima per  $S_{iS}$ , allora  $A'_{iS} = (A_{iS} \setminus \{\min A_{iS}\}) \cup \{\bar{K}\}$  è una soluzione ottima per  $S_{iS}$ .

Infatti, se  $\min A_{iS} = \bar{K}$ , allora  $A'_{iS} = A_{iS}$  è ottima.

Se  $\min A_{iS} > \bar{K}$ , poiché  $|A'_{iS}| = |A_{iS}|$  è sufficiente verificare che tutte le elementi in  $A_{iS} \setminus \{\min A_{iS}\}$  sono compatibili con  $\bar{K}$ .

Sia  $l \in A_{iS} \setminus \{\min A_{iS}\}$ , poniamo  $m = \min A_{iS}$ , allora  $f_m \leq f_l$ .

Da  $f_{\bar{K}} \leq f_m$ , quindi  $f_{\bar{K}} \leq f_l$ , cioè  $\bar{K}$  ed  $l$  sono compatibili.

Pertanto:

- $A'_{iS} \setminus \{\bar{K}\}$  è una soluzione ottima per  $S_{i\bar{K}}$ , quindi  $|A'_{iS} \setminus \{\bar{K}\}| = C[\bar{K}, S]$ .

Poiché  $A'_{iS}$  è una soluzione ottima per  $S_{iS}$  si ha  $|A'_{iS}| = C[i, S]$ , da cui:

$$C[i, S] = C[\bar{K}, S] + 1$$

La scelta  $\bar{K} = \min S_{iS}$  è Greedy relativamente alla seguente intuizione:

Scegliendo tra tutte le attività compatibili quella che termina prima, si massimizza la disponibilità della risorsa per le rimanenti attività.

Si osservi che per selezionare  $\bar{k} = \min S_i$  non è necessario aver risolto precedentemente il problema  $S_{\bar{k}} s$ .

Quindi una soluzione ottima può essere costituita in maniera Top-Down.

Recursive - Activity - Selector ( $s, f, i, s$ )

for  $k = i+1$  to  $s-1$  do

if  $f_i \leq s_k < f_k \leq s_s$  then

return  $\{k\} \cup$  Recursive - Activity - Selector ( $s, f, k, s$ )

return  $\emptyset$

Poiché tutte le chiamate a R-A-S sono del tipo R-A-S ( $s, s, f, i, n+1$ ), essa può essere semplificata così:

Recursive - Activity - Selector ( $s, f, i$ )

$n = |s|$

for  $k = i+1$  to  $n$  do

if  $f_i \leq s_k$  then

return  $\{k\} \cup$  Recursive - Activity - Selector ( $s, f, k$ )

return  $\emptyset$

Ogni attività viene considerata esattamente una volta, quindi R-A-S è lineare.

La quasi ricorsione di coda di R-A-S può essere facilmente eliminata, dando luogo al seguente algoritmo iterativo:

Greedy - Activity - Selector ( $s, f$ )

$n = |s|$

$A = \{\}$

$i = 1$

for  $n=2$  to  $n$  do

if  $s_n \geq f_i$  then

$A = A \cup \{m\}$

$i = m$

return  $A$

Complessità:  $O(n)$

Riassumendo, la strategia del Greedy consiste nei seguenti passi:

- 1 - Verificare la proprietà della sottostruttura ottima;
- 2 - Verificare che esiste sempre una soluzione ottima che include la scelta Greedy;
- 3 - Verificare che dopo la scelta Greedy il problema iniziale è ricondotto ad un sotto problema dello stesso tipo la cui soluzione ottima può essere combinata con la scelta Greedy per dare luogo ad una soluzione ottima del problema iniziale.

### Programmazione Dinamica E Strategia Greedy

Tali metodologie utilizzano entrambe la proprietà della sottostruttura ottima. È quindi possibile che:

- Si utilizzi la programmazione dinamica quando è sufficiente una più efficiente strategia Greedy.
- Per esempio si dia una soluzione basata sulla strategia Greedy quando invece è necessario utilizzare la programmazione dinamica.

## Codici Di Huffman

Consentono fattori di compressione tra il 20% e il 30%.

**Problema:** Trovare una codifica di un file di caratteri in modo da minimizzare la dimensione.

**Codici Prefissi:** Sono codici in cui nessuna codifica è prefisso di un'altra codifica.

**Problema:** Tra tutti gli alberi di decodifica relativi ad un sistema  $(C, f)$  (Dove  $f: C \rightarrow \mathbb{N}$ ) determinare quello di costo minimo, cioè l'albero binario di decodifica  $T$  tale che:

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

Sia minimo.

**Osservazione:** Possiamo limitare la nostra ricerca agli alberi binari pieni, quelli cioè privi di nodi interni con un solo figlio.

**Osservazione:** Il numero di nodi interni in un albero binario pieno con  $n$  figli è  $n - 1$ .

Per costruire un albero binario pieno con  $n$  nodi si possono effettuare  $(n-1)$  operazioni di Merging.

Una possibile strategia Greedy per costruire un albero di costo minimo consiste nell'effettuare le operazioni di Merging di costo minimo.

Huffman ( $C, f$ )

$$n = |C|$$

$Q = \text{make-queue}(C, f)$

for  $i = 1$  to  $n-1$  do

si allochi un nuovo nodo interno  $z$

$\text{left}[z] = x = \text{Extract-Min}(Q)$

$\text{right}[z] = y = \text{Extract-Min}(Q)$

$$f[z] = f[x] + f[y]$$

$\text{insert}(Q, z, f)$

return  $\text{Extract-Min}(Q)$

Complessità:  $(2(n-1)) \text{ Extract-Min } O(n \lg n)$

$(n-1) \text{ Insert } O(n \lg n)$

Build-Heap  $\frac{O(n)}{O(n \lg n)}$

## Correttezza Dell'Algoritmo Di Huffman

### Lemma

Sia  $C$  un alfabeto e  $f: C \rightarrow \mathbb{N}$  una funzione frequenza. Siano  $x$  ed  $y$  i due caratteri in  $C$  di frequenza minima.

Allora  $\exists$  un codice attimo prefisso per  $C$  in cui le codifiche di  $x$  ed  $y$  differiscono solo per l'ultimo bit.

### Lemma

Sia  $C$  un alfabeto e  $f: C \rightarrow \mathbb{N}$  una funzione frequenza. Siano  $x$  ed  $y$  i due caratteri in  $C$  di frequenza minima.

Sia  $C' = \{C \setminus \{x, y\} \cup \{z\}\}$ , con  $z \notin C$ .

Sia  $f': C' \rightarrow \mathbb{N}$  tale che:

$$f'(c) = \begin{cases} f(c) & \text{se } c \neq z \\ f(x) + f(y) & \text{se } c = z \end{cases}$$

Sia  $T'$  un albero ottimo per  $(C', f')$

Sia  $T$  l'albero ottenuto da  $T'$  sostituendo la foglia  $Z$  con un nodo interno avente come figli due foglie etichettate con  $X$  ed  $Y$ , rispettivamente. Allora  $T$  è ottimo per  $(C, f)$

due

## Grafi

Un grafo  $G = (V, E)$  è costituito da un insieme di vertici  $V$  ed un insieme di archi  $E$  ciascuno dei quali connette due vertici in  $V$  detti estremi dell'arco.

Un grafo è orientato quando vi è un ordine fra i due estremi degli archi.

Un cappio è un arco i cui estremi coincidono.

Un grafo non orientato è semplice se:

- non ha cappi
- non ci sono due archi con gli stessi estremi

In caso contrario si parla di multografo.

Salvo indicazioni contrarie noi assumiamo sempre che un grafo sia semplice.

### Archi E Orientamento

Se un grafo è semplice possiamo identificare un arco con la coppia dei suoi estremi:  $e = (v, v) \in E$

Quando  $e = (v, v) \in E$  diciamo che l'arco è incidente in  $v$  e in  $v$ .

Se il grafo è orientato, la coppia  $(v, v)$  è ordinata. In questo caso diciamo che l'arco esce da  $v$  ed entra in  $v$ .

### Grado Di Un Vertice

Il grado  $\delta(v)$  del vertice  $v$  è il numero di archi incidenti in  $v$ .

Se il grafo è orientato  $\delta(v)$  si suddivide in:

- grado entrante  $\delta^-(v)$ : numero di archi entranti in  $v$  (~~in~~<sup>in</sup>-degree)
- grado uscente  $\delta^+(v)$ : numero di archi uscenti da  $v$  (out-degree)

Se  $e = (v, v) \in E$  diciamo che il vertice  $v$  è adiacente al vertice  $v$ .

Se il grafo non è orientato la relazione di adiacenza è simmetrica.

## Cammini E Cicli

Un cammino  $\pi$  di lunghezza  $K$  dal vertice  $u$  al vertice  $v$  in un grafo  $G = (V, E)$  è una sequenza di  $K+1$  vertici.

$$\pi = x_0, x_1, \dots, x_K$$

Tali che  $x_0 = u$ ,  $x_K = v$  e  $(x_{i-1}, x_i) \in E$  per  $i = 1, \dots, K$ . Il cammino  $\pi = x_0$  ha lunghezza  $K = 0$ .

Se  $K > 0$  e  $x_0 = x_K$  diciamo che il cammino è chiuso.

Un cammino semplice è un cammino i cui vertici  $x_0, x_1, \dots, x_K$  ( $K > 0$ ) sono tutti distinti con la possibile eccezione di  $x_0 = x_K$ , nel qual caso è un ciclo.

Un ciclo di lunghezza  $K = 1$  è un cappio. Un grafo aciclico è un grafo che non contiene cicli.

## Raggiungibilità E Connessione

Quando esiste almeno un cammino dal vertice  $u$  al vertice  $v$  diciamo che il vertice  $v$  è accessibile o raggiungibile da  $u$ .

Un grafo non orientato si dice连通的 se esiste almeno un cammino tra ogni coppia di vertici.

Le componenti connesse di un grafo sono le classi di equivalenza dei suoi vertici rispetto alla relazione di accessibilità.

## Connessione Forte

Un grafo orientato si dice fortemente连通的 se esiste almeno un cammino da ogni vertice  $u$  ad ogni altro vertice  $v$ .

Le componenti fortemente connesse di un grafo orientato sono le classi di equivalenza dei suoi vertici rispetto alla relazione di totale accessibilità.

## Completezza

Un grafo si dice completo se esiste un arco per ogni coppia di vertici. Il grafo completo con  $n$  vertici è denotato  $K_n$ .

Un grafo  $G = (V, E)$  è bipartito se esiste una partizione  $(V_1, V_2)$  dell'insieme dei vertici  $V$ , tale che tutti gli archi hanno come estremi un vertice di  $V_1$  ed un vertice di  $V_2$ .

Un grafo bipartito è completo  $(K_{n,m})$  se esiste un arco per ogni coppia (della bipartizione) di vertici.

## Sottografi

Un sottografo del grafo  $G = (V, E)$  è un grafo  $G' = (V', E')$  tale che  $V' \subseteq V$  e  $E' \subseteq E$ .

Il sottografo di  $G = (V, E)$  indotto da  $V' \subseteq V$  è il grafo  $G' = (V', E')$  tale che:

$$E' = \{(u, v) : (u, v) \in E \text{ e } u, v \in V'\}$$

## Rappresentazione Dei Grafi

Vi sono due modi standard per rappresentare un grafo  $G = (V, E)$ : con le liste di adiacenza o con la matrice delle adiacenze.

La rappresentazione di  $G = (V, E)$  mediante liste delle adiacenze è costituita da una lista  $AdS[v]$  per ogni vertice  $v \in V$  in cui vengono memorizzati i vertici adiacenti al vertice  $v$ .

Nella rappresentazione di  $G = (V, E)$  mediante matrice delle adiacenze assumiamo che i vertici siano numerati  $1, 2, \dots, |V|$  in qualche modo arbitrario. La rappresentazione è quindi costituita da una matrice booleana  $A = (a_{ij})$  tale che:

$$a_{ij} = \begin{cases} 1 & \text{se } (i, j) \in E \\ 0 & \text{se } (i, j) \notin E \end{cases}$$

## Visita Di Un Grafo: BFS

Dato un grafo  $G = (V, E)$  ed un vertice particolare  $s \in V$ , detto sorgente, la visita in ampietta partendo dalla sorgente  $s$  visita sistematicamente il grafo per scoprire tutti i vertici che sono raggiungibili da  $s$ . Nel contempo calcola la distanza di ogni vertice del grafo dalla sorgente  $s$  (lunghezza minima di un cammino dalla sorgente allo stesso vertice).

Essa produce anche un albero di ricerca in ampietta, detto albero BF, i cui ramo sono cammini di lunghezza minima.

La visita viene detta in ampietta perché l'algoritmo espande uniformemente la frontiera fra i vertici scoperti e quelli non ancora scoperti. Anzio scopre tutti i vertici a distanza  $K$  da  $s$  prima di scoprire quelli a distanza  $K+1$ .

Per mantenere traccia del punto a cui si è arrivati nell'esecuzione dell'algoritmo i vertici sono colorati:

- Bianco (vertici non ancora raggiunti)
- Grigi (vertici raggiunti che stanno sulla frontiera)
- Nero (vertici raggiunti che non stanno sulla frontiera)

I vertici adiacenti ad un vertice nero possono essere soltanto neri o grigi, mentre quelli adiacenti ad un vertice grigio possono essere anche bianchi.

L'algoritmo costruisce un albero, detto albero BF, che all'inizio contiene soltanto la radice  $s$ . Quando viene scoperto un vertice bianco  $v$  mediante un arco  $(u, v)$  che lo connette ad un vertice  $u$  scoperto precedentemente, il vertice  $v$  e l'arco  $(u, v)$  vengono aggiunti all'albero. Il vertice  $u$  viene detto padre di  $v$ .

### L'Algoritmo : BFS

BFS ( $G, s$ )

for  $v \in V[G]$  do

color [ $\forall v$ ]  $\leftarrow$  bianco,  $d[v] \leftarrow \infty$ ,  $p[v] \leftarrow \text{nil}$

color [ $s$ ]  $\leftarrow$  grigio,  $d[s] \leftarrow 0$

Enqueue ( $\emptyset, s$ )

while not Empty ( $Q$ ) do

$u \leftarrow \text{First} (Q)$

for  $v \in \text{Adjs}[u]$  do

if color [ $v$ ]  $\leftarrow$  bianco then

color [ $v$ ]  $\leftarrow$  grigio,  $d[v] \leftarrow d[u] + 1$ ,  $p[v] \leftarrow u$

Enqueue ( $Q, v$ )

~~Dequeue ( $Q$ )~~

Dequeue ( $Q$ )

color [ $v$ ]  $\leftarrow$  nero

## BFS: Complessità

Vediamo la complessità di BFS in funzione del numero di vertici  $|V|$  e degli  $|E|$  del grafo  $G = (V, E)$ .

L'initializzazione richiede tempo  $O(|V|)$  dovendo percorrere tutti i vertici del grafo.

Dopo l'initializzazione nessun vertice viene colorato bianco e questo ci assicura che ogni vertice verrà inserito nella coda al più una sola volta. Quindi il ciclo while viene eseguito al più  $n$  volte.

Il tempo richiesto per il ciclo while è dunque  $O(|V|)$  più il tempo richiesto per eseguire i cicli for interni.

I cicli for interni percorrono una sola volta le liste delle adiacenze di ciascun vertice visitato. Siccome la somma delle lunghezze di tutte le liste è  $O(|E|)$  possiamo concludere che la complessità è :

$$O(|V| + |E|)$$

Con abuso di notazione, scrivremo anche :

$$O(V+E)$$

## Visita Di Un Grafo: DFS

Nella visita in profondità a partire da un dato vertice, si esplorano sempre gli archi uscenti dell'ultimo vertice  $v$  raggiunto. Se viene scoperto un nuovo vertice  $v$ , ci si sposta su tale vertice. Se tutti gli archi uscenti da un vertice  $v$  portano a vertici già scoperti si torna indietro e si riprende esplorando gli altri archi uscenti dal vertice da cui  $v$  è stato scoperto.

Si continua finché tutti i vertici raggiungibili dal vertice iniziale scelto sono stati scoperti. Se non tutti i vertici del grafo sono stati raggiunti, il procedimento viene ripetuto partendo da un qualunque vertice non ancora raggiunto.

Quando viene scoperto un nuovo vertice  $v$  visitando la lista delle adiacenze di un vertice  $u$  si memorizza un puntatore da  $u$  a  $v$  ed  $u$ . Si costruisce così, in generale, non un albero, ma una foresta (Insieme di alberi disgiunti).

Come nella BFS, i vertici sono colorati di:

- Bianco (vertici non ancora raggiunti)
- Grigio (vertici scoperti)
- Nero (vertici finiti, la cui lista delle adiacenze è stata completamente esplorata)

### DFS: Marcatempi

La DFS utilizza due marcatempi su ogni vertice  $v$ :

- $d[v]$ : tempo di scoperto, quando il vertice è colorato di grigio
- $f[v]$ : tempo di fine visita, quando il vertice è colorato di nero

### L'Algoritmo: DFS

DFS( $G$ )

```
for  $v \in V[G]$  do  
    color [ $v$ ]  $\leftarrow$  bianco,  $p[v] \leftarrow \text{nil}$ 
```

$T \leftarrow 1$

```
for  $v \in V[G]$  do  
    if color [ $v$ ] = bianco then  
        DFSric ( $v, T$ )
```

DFSric ( $v, T$ )

```
color [ $v$ ]  $\leftarrow$  grigio,  $d[v] \leftarrow T$ ,  $T \leftarrow T+1$ 
```

```
for  $u \in \text{AdS}[v]$  do  
    if color [ $u$ ] = bianco then
```

$p[u] \leftarrow v$

DFSric ( $u, T$ )

```
color [ $v$ ]  $\leftarrow$  nero,  $f[v] \leftarrow T$ ,  $T \leftarrow T+1$ 
```

## DFS: Complessità

I due cicli for di DFS richiedono tempo  $\Theta(|V|)$ .

La funzione DFS ric viene chiamata solo su vertici bianchi che vengono subito colorati di grigio. Essa viene quindi richiamata al più una sola volta per ogni vertice. Il ciclo interno percorre la lista delle adiacenze dei vertici.

Gioccome la somma delle lunghezze di tutte le liste delle adiacenze è  $\Theta(|E|)$  l'intero algoritmo ha complessità  $\Theta(|V| + |E|)$

## DFS: Proprietà Delle Parentesi

Se si rappresenta la scoperta di ogni vertice  $v$  con una parentesi aperta ( $v$ ) e la fine con una parentesi chiusa ( $v$ ) si ottiene una sequenza bilanciata di parentesi. Ossia,  $\forall$  coppia di vertici  $u$  e  $v$  ci sono quattro possibilità:

1  $(v \dots u) \dots (v \dots v)$

2  $(v \dots v) \dots (v \dots v)$

3  $(v \dots (v \dots v) \dots u)$

4  $(v \dots (v \dots v) \dots v)$

## Altre Proprietà

Discendenti: Il vertice  $v$  è discendente del vertice  $u$  in un albero della foresta di ricerca in profondità se e solo se:

$$(u \dots (v \dots v) \dots v)$$

Cammino Bianco: Il vertice  $v$  è discendente del vertice  $u$  in un albero della foresta di ricerca in profondità se e solo se nell'istante in cui  $v$  viene scoperto  $\exists$  un cammino da  $u$  a  $v$  i cui vertici sono tutti bianchi (cammino bianco)

## Classificazione Degli Archi

Archi D'Albero: archi  $(u, v)$  con  $v$  scoperto visitando le adiacenze di  $u$  (da grigio a bianco).

Archi All'Indietro: archi  $(u, v)$  con  $u = v$  oppure  $v$  discendente di  $u$  in un albero della foresta di ricerca in profondità (da grigio a grigio)

Archi In Avanti: archi  $(u, v)$  con  $v$  discendente di  $u$  in un albero della foresta (da grigio a nero)

Archi Trasversali: archi  $(u, v)$  in cui  $v$  ed  $u$  appartengono a rami o alberi distinti dalla foresta (da grigio a nero)

### Oordinamento Topologico

La ricerca in profondità si può usare per ordinare topologicamente un grafo orientato aciclico (detto anche DAG).

In un ordinamento topologico di un grafo orientato aciclico  $G = (V, E)$  è un ordinamento (Permutazione) dei suoi vertici tale che  $\forall$  arco  $(u, v) \in E$  il vertice  $u$  precede il vertice  $v$

### L'Algoritmo: TopSort

TP\\_DFS( $G$ )

for  $v \in V[G]$  do

color[v]  $\leftarrow$  bianco, p[v]  $\leftarrow$  nil

t  $\leftarrow$  1

TP  $\leftarrow$  nil // Lista Concatenata Vuota

for  $v \in V[G]$  do

if color[v] = bianco then TP\\_DFSric(v, t)

return TP;

TP\\_DFSric( $u, t$ )

color[u]  $\leftarrow$  grigio, d[u]  $\leftarrow$  t, t  $\leftarrow$  t + 1

for  $v \in \text{AdS}[u]$  do

if color[v] = bianco then p[v]  $\leftarrow u$ , TP\\_DFSric(v, t)

color[u]  $\leftarrow$  nero, f[u]  $\leftarrow t$ , t  $\leftarrow$  t + 1

TP  $\leftarrow u + TP$  // Inserimento In Testa Alla Lista Concatenata.

## L'Algoritmo TopSort: Complessità E Correttezza

Complessità: La stessa di DFS, ossia:

$$O(|V| + |E|)$$

Correttezza: Dal momento che un grafo è un DAG se e solo se non ci sono orchi d'indietro, allora, quando un arco  $(v, v) \in E$  viene ispezionato da DFS il vertice  $v$  non può essere grigio, quindi  $v$  è nero oppure è bianco.

## Componenti Fortemente Connesse

La ricerca in profondità si può usare anche per calcolare le componenti fortemente connesse di un grafo orientato.

Una componente fortemente connessa (cfc) di un grafo orientato  $G = (V, E)$  è un insieme massimale di vertici  $U \subseteq V$  tale che  $\forall u, v \in U \exists$  un cammino da  $u$  a  $v$  ed un cammino da  $v$  ad  $u$ . L'algoritmo per il calcolo delle cfc si compone di 3 fasi:

- 1 - usa la ricerca in profondità in  $G$  per ordinare i vertici in ordine di tempo di fine  $f$  decrescente:  $O(|V| + |E|)$
- 2 - calcola il grafo trasposto  $G^T$  del grafo  $G$ :  $O(|V| + |E|)$
- 3 - esegue una ricerca in profondità in  $G^T$  usando l'ordine dei vertici calcolato nella 1° fase nel ciclo principale:  $O(|V| + |E|)$

Alla fine gli alberi della ricerca in profondità in  $G^T$  rappresentano le cfc.

## Proprietà Del Cammini

Se due vertici  $u$  e  $v$  sono in una stessa cfc allora tale cfc contiene anche i vertici di tutti i cammini da  $u$  a  $v$ .

## Proprietà Degli Alberi

Una ricerca in profondità mette nello stesso albero tutti i vertici di una cfc.

## Grafo delle Componenti

Dato un grafo orientato  $G$ , il grafo delle componenti fortemente connesse di  $G$  è il grafo orientato  $H$  avente come vertici le cfc di  $G$  e un arco da una cfc  $C$  ad una cfc  $C'$  se e solo se in  $G$  vi è un arco che connette un vertice di  $C$  ad un vertice di  $C'$ .

### Proprietà

Il grafo  $H$  delle componenti fortemente connesse di  $G$  è acchico.

## Mediana

Select(A, i)

- Si divide A in  $\lfloor \frac{n}{5} \rfloor$  gruppi di 5 elementi, più un eventuale gruppo con i restanti  $n \bmod 5$  elementi.
- Si determinano le mediane degli  $\lceil \frac{n}{5} \rceil$  gruppi e sia M lo sequenza di tali mediane.
- Si effettui la chiamata ricorsiva  $m = \text{Select}(M, \lfloor \frac{|M|}{2} \rfloor)$
- Si partizioni A in  $A_1, A_2, A_3$  dove:
  - $A_1 = [x \in A : x < m]$
  - $A_2 = [x \in A : x = m]$
  - $A_3 = [x \in A : x > m]$
- if  $|A_1| \geq i$  then  
    return Select( $A_1, i$ )
- else if  $(|A_1| + |A_2| \geq i)$  then  
    return m
- else return Select( $A_3, i - |A_1| - |A_2|$ )

## Metodologia Greedy

La metodologia greedy consiste in un insieme di algoritmi di ottimizzazione che ci permettono di ottenere una soluzione ottimale attraverso un'intuizione iniziale, senza dover utilizzare necessariamente la programmazione dinamica.

Per il problema della selezione delle attività: "se si sceglie tra le attività compatibili quella che termina prima, essa sarà considerata la più efficiente e sarà quest'ultima a liberare prima le risorse per le rimanenti attività".

Nel nostro problema abbiamo un insieme di attività  $S = \{a_1, \dots, a_n\}$ , il problema di ottimizzazione relativo alla selezione di attività, consiste nel trovare un sottoinsieme di  $S$  di cardinalità massima, di attività mutuamente compatibili, cioè che i tempi di esecuzione delle due attività non siano in conflitto.

In particolare date due attività  $a_i$  e  $a_j$ , sono compatibili se il tempo di fine dell'attività che inizia prima, sia minore o uguale del ~~tempo~~ tempo di inizio della seconda attività.

### Algoritmo

Supponiamo che le attività siano ordinate in modo crescente rispetto al tempo di fine.

Siano  $S$  e  $F$  array contenente i tempi di inizio e fine.

Greedy - Activity - Selector ( $S, F$ )

$$A = \{\}\}$$

$$j = 1$$

for  $i = 2$  to  $S.length$

if  $S[i] \geq F[j]$

$$A = A \cup \{i\}$$

$$j = i$$

return  $A$

$O(n)$  escludendo  
l'ordinamento iniziale

## Programmazione Dinamica

Oggi

$$\rightarrow a \oplus b =_{\text{def}} 3a + 4b \quad \text{Associativa?}$$

$$(a \oplus b) \oplus c = a \oplus (b \oplus c)$$

$\overbrace{a} \quad \overbrace{b} \quad \overbrace{c} \quad \overbrace{b}$

$$3(3a + 4b) + 4c = 3a + 4(3b + 4c)$$

$$9a + 12b + 4c \neq 3a + 12b + 16c$$

Non è associativa

A noi interessa calcolare  $a_1 \oplus a_2 \oplus \dots \oplus a_n$ L'operazione  $a \oplus b = 3a + 4b$  non è associativaDiciamo che un'espressione  $E$  è completamente parentesizzata se vale una delle seguenti condizioni:

- $E$  è un singolo elemento
- $E$  ha la forma  $(E_1 \oplus E_2)$ , dove  $E_1$  ed  $E_2$  sono espressioni completamente parentesizzate

Sia  $E$  una parentesizzazione ottima per la sequenza di elementi  $(a_1, a_2, \dots, a_n)$ Supponiamo  $n \geq 2$ 

$$E = (E_1 \oplus E_2)$$

con  $E_1$  parentesizzazione di  $(a_1, a_2, \dots, a_k)$  $E_2$  parentesizzazione di  $(a_{k+1}, \dots, a_n)$ 

$$1 \leq k \leq n-1$$

Poiché:

$$\#(E) = \#(E_1) + \#(E_2)$$

ne segue che:

-  $E_1$  parentesizzazione ottima di  $(a_1, a_2, \dots, a_k)$ -  $E_2$  parentesizzazione ottima di  $(a_{k+1}, \dots, a_n)$ 

Pertanto la classe dei sottoproblemi da risolvere è data da:

$$\{(a_i, \dots, a_s) : 1 \leq i \leq s \leq n\}$$

 $m[i, s] = \text{costo di una soluzione ottima di } (a_i, \dots, a_s)$   
 valore

$$m[i, s] = \begin{cases} m_{ii} & i=s \\ \max_{i \leq k < s} (m[i, k] + m[k+1, s]) & i < s \end{cases}$$

Number-Value-Order (A)

for  $i=1$  to  $n$  do

$$m[i, i] = A_i$$

for  $\Delta=1$  to  $n-1$  do

for  $i=1$  to  $n-\Delta$  do

$$s = \Delta + i$$

$$m[i, s] = -\infty$$

for  $k=i$  to  $s-1$  do

$$q = 3m[i, k] + 4m[k+1, s]$$

if  $q \geq m[i, s]$  then

$$m[i, s] = q$$

$$s[i, s] = k$$

return  $m, s$

Number-Value-Extract ( $A, s, i, s$ )

if  $i=s$  then

return  $A_i$

else

$$X = \text{Number-Value-Extract}(A, s, i, s[i, s])$$

$$Y = \text{Number-Value-Extract}(A, s, s[i, s]+1, s)$$

$$\text{return } 3*X + 4*Y$$