



Università
di Catania

CORSO DI LAUREA IN INFORMATICA

La Bibbia di Algoritmi

Intersezione tra appunti presi a lezione e libro

Autore:

Andrea Girlando

Materia:

Algoritmi e Strutture Dati

A.A. 2025/2026

Ricorsione

Introduzione alla ricorsione

La ricorsione

La **ricorsione** rappresenta uno dei concetti più eleganti e profondi della programmazione. Essa si fonda sull'idea che un problema complesso possa essere affrontato scomponendolo in sotto-problemi più semplici, ciascuno dei quali conserva la stessa natura e struttura del problema originario, ma su scala ridotta.

Definizione

Un **programma ricorsivo** è una funzione o una procedura che durante la propria esecuzione richiama se stessa per risolvere versioni sempre più piccole dello stesso problema

Questo meccanismo sembra apparentemente circolare, ma è in realtà estremamente potente a condizione che esista un punto di arresto ben definito: ovvero **la condizione base**

Un esempio classico di questo meccanismo è il calcolo del fattoriale di un numero intero non negativo n

$$n! = \begin{cases} 1 & \text{se } n = 0, \\ n \cdot (n - 1)! & \text{se } n > 0, \end{cases}$$

In breve si afferma che per calcolare $n!$ non è necessario conoscere direttamente il risultato ma basta sapere come calcolare il fattoriale di un numero più piccolo $n - 1$. Questa definizione *auto-riferita* è perfettamente lecita e diventa potente perché il caso base (quello sopra) interrompe la catena di richiami e garantisce la terminazione (cosa molto importante in un programma)

Fasi della ricorsione

Le fasi della ricorsione sono:

- **Fase di suddivisione:** del problema in sotto problemi di minore complessità
- **Fase di ricombinazione:** dei risultati parziali nella soluzione complessiva
Questa duplice operazione è ciò che rende la ricorsione tanto elegante quanto potente infatti permette di passare da una visione monolitica del calcolo a una visione *gerarchica e modulare*

Definizioni ricorsive delle strutture dati

La ricorsione non riguarda solo gli algoritmi o le funzioni ma anche le **strutture dati**, di seguito degli esempi:

- **Lista:** non è altro che un *elemento iniziale* seguito da un'altra lista dello stesso tipo

$$\text{Lista} = \begin{cases} \text{Lista vuota,} & \text{oppure} \\ (\text{Elemento, Lista}) & \text{se contiene almeno un elemento.} \end{cases}$$

- **Array:** un array di lunghezza n può essere definito come un elemento iniziale seguito da un array di lunghezza $n - 1$:

$$A_n = \{a_0\} \cup A_{n-1}$$

- **Alberi:** ogni nodo dell'albero può essere considerato come la radice di un nuovo albero, composto dai suoi sottoalberi

$$\text{Albero} = \begin{cases} \text{nodo vuoto,} & (\text{albero nullo}) \\ (\text{valore, sottoalbero sinistro, sottoalbero destro}) & (\text{albero non vuoto}). \end{cases}$$

Struttura generale di un programma ricorsivo

Ogni programma ricorsivo si fonda su una struttura concettuale ben precisa, che ne garantisce la correttezza logica e la terminazione formata da:

1. **Il caso base:** il caso base è una condizione che ne interrompe la prosecuzione infinita di una ricorsione, matematicamente corrisponde alla condizione di terminazione dell'equazione definita ricorsivamente
2. **Il passo ricorsivo:** In questo punto la funzione richiama sé stessa per risolvere uno o più sotto problemi di dimensione minore, se indichiamo con P il problema principale e con P_1, P_2, \dots, P_K i sotto problemi derivati, il passo ricorsivo può essere espresso formalmente come:

$$S = \text{Combina}(F(P_1), F(P_2), \dots, F(P_K))$$

3. **La combinazione dei risultati:** Una volta risolti i sotto problemi occorre un meccanismo per ricomporre le soluzioni parziali e ottenere la soluzione complessiva del problema principale. Questo processo detto fase di combinazione è spesso ciò che distingue una ricorsione semplice da una più sofisticata

Una funzione ricorsiva $F(n)$ può essere definita in modo generale nel seguente modo:

$$F(n) = \begin{cases} S_0, & \text{se } n \text{ soddisfa la condizione base;} \\ \text{Combina}(F(n_1), F(n_2), \dots, F(n_K)), & \text{altrimenti.} \end{cases}$$

L'esecuzione di questo tipo di funzioni viene gestito automaticamente dallo [stack](#) che conserva lo stato di ciascuna chiamata, nel caso in cui la profondità di ricorsione non fosse finita si potrebbe incappare in problemi di [stack overflow](#)

Dal punto di vista operativo l'esecuzione di un programma ricorsivo può essere immaginata come una serie di scatole

Dimensione del problema

Un aspetto cruciale nella progettazione di un algoritmo ricorsivo è la definizione della **dimensione del problema**, spesso indicata con la variabile n , in alcuni casi questa grandezza è intuitiva.

Esempio: nel calcolo del fattoriale o della potenza, la dimensione del problema n coincide con un numero intero che viene progressivamente decrementato fino a raggiungere lo zero.

La scelta del parametro che rappresenti la dimensione del problema non è una semplice formalità: essa influenza profondamente sul comportamento dell'algoritmo e sulla sua **complessità computazionale**

L'equazione di ricorrenza

Ogni algoritmo ricorsivo possiede una propria **struttura quantitativa** che descrive come il costo del calcolo cresce in funzione delle dimensione del problema, questa struttura prende il nome di **equazione di ricorrenza**. Un'equazione di ricorrenza esprime il tempo di esecuzione di un algoritmo ricorsivo come funzione del tempo necessario per risolvere uno o più sotto problemi più piccoli più un termine che rappresenta il costo delle operazioni non ricorsive, la funzione del tempo per risolvere un problema di dimensione n viene indicato con $T(n)$ ovvero:

$$T(n) = \sum_{i=0}^K T(n_i) + f(n)$$

dove $T(n_i)$ rappresenta il tempo necessario per risolvere il sotto problema i -esimo (di dimensione $n_i < n$) e $f(n)$ rappresenta il tempo impiegato per la fase di divisione e riunificazione (le fasi che non comportano chiamate ricorsive)

Costruzione di una equazione di ricorrenza

Per costruire l'equazione di ricorrenza corrispondente a un algoritmo occorre osservare come esso si comporta rispetto alla dimensione del problema. Ogni chiamata ricorsiva può essere vista come un nodo dell'albero di ricorsione e il numero di nodi totali determina il tempo totale.

Esempi

- **Fattoriale:** in questo algoritmo la ricorsione riduce la dimensione del problema di 1 ad ogni chiamata (per ogni chiamata ricorsiva viene fatta una sola moltiplicazione) quindi la sua equazione di ricorrenza è:
 $T(n) = T(n - 1) + O(1)$ risolvendo questa ricorrenza otteniamo che $T(n) = O(n)$
- **Somma di un array:** in un algoritmo che calcola la somma di una array dividendolo in due metà il tempo di esecuzione ha equazione:

$$2T\left(\frac{n}{2}\right) + O(i)$$

poiché ogni chiamata genera due sotto problemi di metà dimensione, è la fase di combinazione richiede solo un tempo costante per sommare i risultati. Risolvendola otteniamo sempre $T(n) = O(n)$ ma con una struttura di chiamate ricorsive estremamente diversa.

- **Merge sort:** in questo algoritmo ogni livello di ricorsione comporta una divisione in due sottoarray ma la fase di riunificazione (il merging) richiede un tempo lineare $O(n)$. L'equazione di ricorrenza corrispondente diventa quindi:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \rightarrow O(n \log n)$$

Questo risultato è uno dei più noti nell'analisi algoritmica e mostra come la ricorsione possa amplificare o ridurre l'efficienza di un algoritmo.

Applicazioni pratiche della ricorsione

Per comprendere a fondo il funzionamento della ricorsione, è utile analizzare alcuni esempi elementari ma emblematici, mostrano come un problema ricorsivo si risolve sempre creando le due fasi:

- **Fase di divisione:** il problema viene suddiviso in versioni più semplici
- **Fase di riunificazione:** le soluzioni vengono combinate per ottenere la soluzione finale

Il fattoriale

Il calcolo del **fattoriale** di un numero intero non negativo n è forse l'esempio più iconico di funzione ricorsiva

⚠ Che cosa è un fattoriale

Il fattoriale indicato con $n!$ rappresenta il prodotto di tutti i numeri interi positivi minori o uguali ad n

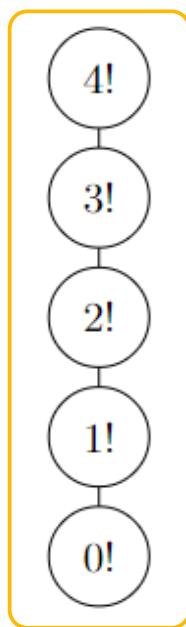
La sua definizione sottoforma di funzione ricorsiva è la seguente:

$$n! = \begin{cases} 1 & \text{se } n = 0, \\ n \cdot (n - 1)! & \text{se } n > 0. \end{cases}$$

- **Fase di divisione:** consiste nel ridurre il problema $F(n)$ a un sotto problema di dimensione minore ovvero $F(n - 1)$
- **Fase di riunificazione:** avviene durante il ritorno delle chiamate: ciascuna moltiplica il valore ottenuto dalla chiamata successiva per il proprio parametro n

```
int fattoriale ( int n ) {
    if ( n == 0) return 1;
    else return n*fattoriale(n-1)
}
```

Questo è l'albero di ricorsione con $n = 4$



Moltiplicazione come somma ripetuta

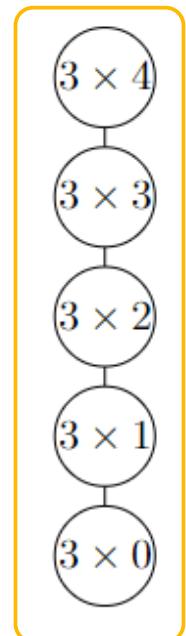
Per due numeri interi non negativi possiamo esprimere il prodotto come:

$$a \times b = \begin{cases} 0 & \text{se } b = 0, \\ a + (a \times (b - 1)) & \text{se } b > 0. \end{cases}$$

- **Fase divisione:** riduce il secondo argomento b di una unità ad ogni chiamata, il problema $P(a, b)$ viene trasformato in un sotto problema $P(a, b - 1)$ di dimensione più piccola
- **Fase di riunificazione:** consiste nel sommare a al risultato ottenuto dalla chiamata successiva

```
int moltiplica ( int a , int b ) {  
    if ( b == 0) 3 return 0;  
    else return a + moltiplica (a , b - 1);  
}
```

Questo è l'albero di ricorsione di 3×4



Potenza con moltiplicazione ripetuta

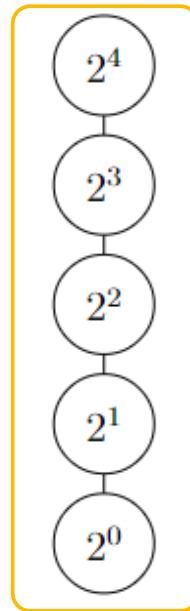
La potenza dati due numeri viene definita così

$$a^n = \begin{cases} 1 & \text{se } n = 0, \\ a \times a^{n-1} & \text{se } n > 0. \end{cases}$$

- **Fase di divisione:** consiste nel ridurre progressivamente l'esponente n fino a raggiungere lo zero
- **Fase di riunificazione:** avviene moltiplicando ogni valore intermedio per a nel momento in cui le chiamate ritornano

```
int potenza ( int a , int n ) {
    if (n == 0) return 1;
    else return a * potenza (a , n - 1)
}
```

L'albero di ricorsione per il calcolo di 2^4 ha la seguente forma:



Problemi di somma e soglia su array

Premessa

Gli array sono un ottimo terreno per mettere alla prova la ricorsione, soprattutto in contrapposizione ad una approccio iterativo.

I problemi sugli array possono essere risolti in modo ricorsivo seguendo due schemi generali:

- **Ricorsione in coda:** in cui l'array viene ridotto progressivamente di un elemento alla volta
- **Doppia ricorsione:** in cui l'array viene suddiviso in due sotto array di dimensioni più piccole che vengono risolti e poi ricombinati

Somma dei valori di un array

Supponiamo di avere un array di interi, il nostro obiettivo è quello di scrivere una procedura in grado di restituire per qualsiasi array A di dimensione n , il valore:

$$S(A, n) = \sum_{i=0}^{n-1} A[i] \text{ somma di } n \text{ numeri di un array}$$

Si tratta di un problema apparentemente semplice ma utile per comprendere la differenza tra un approccio **iterativo** e **ricorsivo**:

- **Approccio iterativo:** supponendo di avere un array di n elementi ci basta scorrere l'array da destra verso sinistra e accumulando progressivamente il valore in una variabile che chiameremo **somma**

```
int sommaIterativa(int A[], int n) {
    int somma = 0;
    for (int i = 0; i < n; i++) {
```

```

    somma += A[i]; // accumula il valore corrente
}
return somma;
}

```

- **Approccio ricorsivo:** per usare un approccio ricorsivo dobbiamo scomporre il problema in problemi più piccoli, ciò significa nel caso della somma dobbiamo immaginare l'azione di scorrimento iterativo dell'array come una funzione che richiama ste stessa su versioni ridotte dello stesso array, questa cosa si può fare in due modi:

1. *Ricorsione di coda*: usando al ricorsione in coda definiamo S come:

$$S(A, n) = \begin{cases} A[0] & \text{se } n = 1, \\ A[n - 1] + S(A, n - 1) & \text{se } n > 1. \end{cases}$$

Fase di divisione: consiste nel passare da un array di n elementi a uno di $n - 1$

Fase di riunificazione: consiste nel sommare il valore corrente $A[n - 1]$ al risultato ottenuto ricorsivamente

```

int somma(int A[], int n) {
    if (n == 1)
        return A[0]; // caso base
    else
        return A[n-1] + somma(A, n-1); // divisione e riunificazione
}

```

Albero di ricorsione: la complessità dell'albero di ricorsione è n e ogni chiamata ricorsiva genera una sola nuova chiamata quindi la complessità temporale è $O(n)$.

2. *Doppia ricorsione*: usando la doppia ricorsione definiamo S come:

$$S(A, n) = \begin{cases} A[0] & \text{se } n = 1, \\ S(A_{\text{sx}}, n/2) + S(A_{\text{dx}}, n/2) & \text{se } n > 1. \end{cases}$$

- *Fase di divisione*: suddivide il problema in due sotto problemi di dimensione $n/2$
- *Fase di riunificazione*: consiste nel sommare i due risultati parziali

```

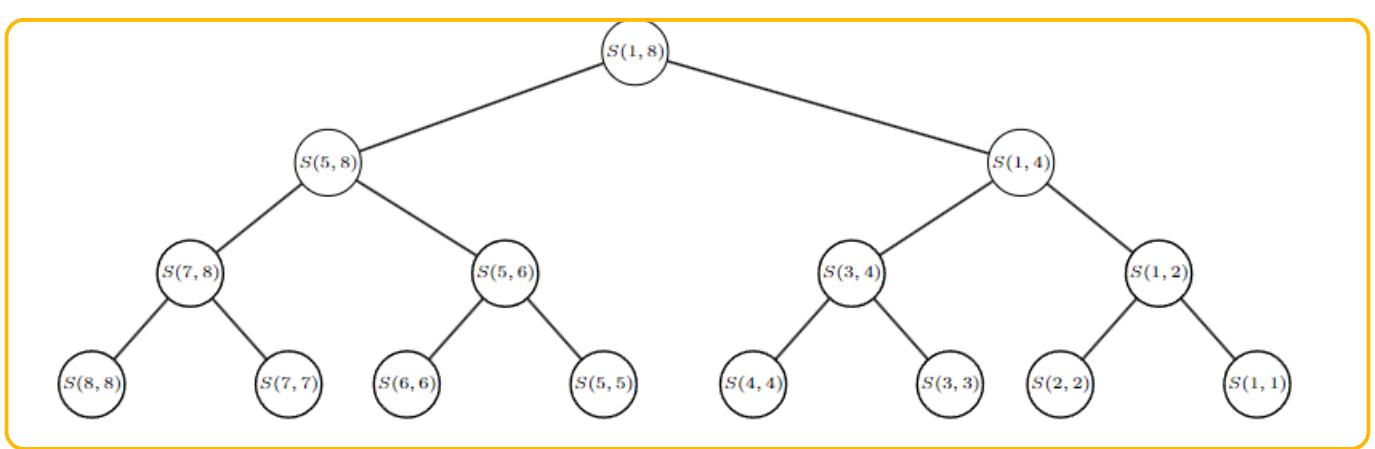
int sommaDivisa(int A[], int inizio, int fine) {
    if (inizio == fine)
        return A[inizio]; // caso base

    int medio = (inizio + fine) / 2;
    int sinistra = sommaDivisa(A, inizio, medio); // divisione 1
    int destra = sommaDivisa(A, medio + 1, fine); // divisione 2

    return sinistra + destra; // riunificazione
}

```

Albero di ricorsione: L'albero di seguito è bilanciato e completo con profondità $\log_2 n$ questo ci indica una complessità temporale complessiva è: $T(n) = 2T(\frac{n}{2}) + O(1)$



Verificare se la somma supera una soglia

Vogliamo stabilire se la somma degli elementi di un array A supera una certa soglia T e vogliamo come risultato un esito logico true\false, quello che facciamo è facciamo scorrere la soglia T verso il basso man mano che consumiamo l'array: a ogni passo sottraiamo l'ultimo elemento e chiediamo se la somma rimanente supera la nuova soglia

```
bool supera(int A[], int n, int T) {
    if (T < 0)
        return true; // soglia già superata
    if (n == 0)
        return false; // array esaurito, soglia non superata

    return supera(A, n - 1, T - A[n - 1]);
}
```

- *Fase di divisione*: il problema viene suddiviso in problemi più piccoli con una soglia sempre più bassa
- *Fase di riunificazione*: il valore dell'ultima chiamata ricorsiva risale

È naturale chiedersi se un approccio a doppia ricorsione possa offrire vantaggi, ma in questo caso la risposta è no, infatti se si suddividesse l'array in due sotto array ciascuna chiamata fornirebbe la risposta "la sinistra da sola non supera T " che non ci dice nulla sull'esito finale che dipende da sinistra+destra, ne consegue che la ricorsione binaria in forma puramente booleana è insufficiente e quindi si dovrebbe complicare il tutto per renderla funzionante

Quando l'obiettivo è un esito booleano con possibile arresto anticipato, la ricorsione lineare con soglia residua è la forma più naturale chiara ed efficace

Approccio ricorsivo e iterativo

Analisi

Dopo aver analizzato vari esempi di funzioni ricorsive e la loro costruzione passo dopo passo, è naturale chiedersi quale sia la differenza sostanziale tra un approccio ricorsivo e uno iterativo:

- **Ricorsione**:
 - *Concettualmente*: rappresenta un modo di ragionare *dall'altro verso il basso*, infatti parte dalla formulazione del problema generale e lo scomponete in sotto problemi più piccoli
 - *Computazionalmente*: richiede una quantità di memoria proporzionale alla profondità della ricorsione
- **Iterazione**:
 - *Concettualmente*: rappresenta il modo di ragionare *dal basso verso l'alto*, la soluzione viene costruita passo passo mantenendo esplicitamente lo stato intermedio della computazione

- *Computazionalmente*: usa una quantità di memoria costante poiché le variabili utilizzate vengono riutilizzate a ogni passo.
- Le funzioni ricorsive hanno molto spesso una maggiore *chiarezza espressiva* e questo permette una ottima *modularità*, naturalmente presenta anche dei *limiti pratici*, come quando la profondità di una chiamata è molto grande o anche nella gestione dei bug durante la fase di debug.

Il problema dello zaino

Il **problema dello zaino** che consiste in:

dato un insieme di oggetti, ciascuno con un valore e un peso, si vuole scegliere un sottoinsieme che massimizza il valore totale senza superare una capacità massima K .

Affronteremo questo problema in due versioni

Versione 0.1: pesi unitari, valori diversi

In questo caso abbiamo oggetto tutti con lo stesso peso ma valori differenti, lo zaino può contenere solo un numero limitato di pezzi, pari alla capacità K , inoltre gli oggetti sono già disposti in ordine crescente di valore, dal meno prezioso al più prezioso, questa rappresenta la forma più semplice di questo problema. Infatti la scelta ottimale e abbastanza logica consiste nel prendere i K oggetti che hanno il valore più alto

Formalizzazione:

- Abbiamo n oggetti ordinati per valore crescente:

$$v_1 \leq v_2 \leq \dots \leq v_n \text{ tutti di peso unitario.}$$

- Vogliamo massimizzare il valore complessivo:

$$\max\{\sum_{i \in S} v_i : S \subseteq \{1, \dots, n\}, |S| \leq K\}$$

poiché i pesi sono unitari e i valori ordinati, la soluzione ottima è semplicemente:

$$S^* = \{n - K + 1, n - K + 2, \dots, n\}, V^* = \sum_{i=n-K+1}^n v_i$$

- Il problema si risolve in maniera lineare in un tempo di esecuzione $O(K)$ poiché non sono necessarie né scelte condizionali né chiamate ricorsive

```
int zainoLineare ( int v [] , int n , int K ) {
    int somma = 0;
    for ( int i = n - K ; i < n ; i ++ ) {
        if ( i >= 0) somma += v [ i ];
    }
    return somma;
}
```

Anche se non prettamente necessario per fini didattici esprimiamo la stessa soluzione anche in modo ricorsivo

$$F(i, k) = \begin{cases} 0 & \text{se } k = 0 \text{ o } i = 0, \\ v_i + F(i - 1, k - 1) & \text{se } k > 0. \end{cases}$$

Ricordiamo che: poiché gli oggetti sono già ordinati non è necessario scegliere se includere o meno l'oggetto i : se c'è ancora spazio nello zaino ($k > 0$) lo si prende automaticamente

```
int F ( int v [] , int i , int k ) {
    if ( i == 0 || k == 0) return 0;
    return v [i -1] + F (v , i -1 , k -1);
}
```

La funzione viene richiamata inizialmente con $F(v, n, K)$ e termina dopo K chiamate ricorsive

Conclusioni: questo esempio ci permette di capire che un algoritmo iterativo può essere riscritto anche in forma ricorsiva anche quando la ricorsione non offre dei vantaggi in termini di efficienza

Versione generale 0/1: pesi e valori variabili

Il problema si evolve, gli oggetti non sono più simili tra loro: ogni oggetto ha dunque il proprio peso w_i e un valore v_i differente. La strategia di risoluzione del problema precedente non funziona più infatti: *un oggetto di grande valore ma troppo pesante potrebbe impedire di portare con se più oggetti leggeri complessivamente più vantaggiosi*, devono essere valutate tutte le possibili combinazioni e in questo la ricorsione diventa uno strumento fondamentale

Formalizzazione: Per ogni oggetto i abbiamo peso $w_i \in N_{>0}$ e valore $v_i \geq 0$ la capacità massima dello zaino è $K \in N$. Vogliamo massimizzare $\max\{\sum_{i \in S} v_i : S \subseteq \{1, \dots, n\}, \sum w_i \leq K\}$

- Occorre confrontare sistematicamente le scelte *includo/escludo*
- Definiamo $F(i, k)$ come prima, i suoi **casi base** sono:
 - $F(i, k) = 0$ se $i = 0$ o $k = 0$
 - $F(i, k) = -\infty$ se $k < 0$
 - visto che andremo a controllare tutte le possibili combinazioni il $-\infty$ ci serve per squalificare tutte le ricorsioni che superano la capacità massima.
- il suo caso **caso ricorsivo** è: $F(i, k) = \max\{F(i - 1, k), v_i + F(i - 1, k - w_i)\}$

Si osservi che se una soluzione ottima non include i allora è ottima su $(i - 1, k)$. Se include i allora la soluzione residua è ottima su $(i - 1, k - w_i)$. Massimizzando i due casi otteniamo il valore ottimo.

```
int F ( int i , int k , int v [] , int w [] ) {  
    if ( k < 0 ) return INT_MIN /4; // rappresenta - infinito  
    if ( i == 0 || k == 0 ) return 0;  
    int senza = F (i -1 , k , v , w ) ;  
    int con = v [ i ] + F (i -1 , k - w [ i ] , v , w ) ;  
    return ( senza > con ) ? senza : con ;  
}
```

La ricorsione da sola restituisce il valore ottimo, per ottenere una soluzione concreta facciamo un confronto locale per capire se i è preso o escluso. Questa semplice soluzione ha complessità $O(n)$

Esercizi sulla ricorsione

La ricorsione è uno strumento potente per affrontare i problemi di ottimizzazione, gli esercizi dove ci viene chiesto di trova una soluzione ad un problema di ottimizzazione usando la ricorsione devono essere sviluppati seguendo questi passi:

- **1.** Individuare i casi base che rendono la ricorsione terminante
- **2.** Determinare la scomposizione del problema in sottoproblemi più piccoli
- **3.** Stabilire una regola per combinare le soluzioni

Le Notazioni Asintotiche e l'Analisi della Complessità degli Algoritmi

Concetti preliminari di analisi asintotica

Confronto tra le principali funzioni di crescita

Quando si analizza l'efficienza di un algoritmo, non ci si limita a misurare il tempo effettivo di esecuzione sul computer, poiché tale misura dipende da fattori cangianti da macchina a macchina, quello che si punta a fare è descrivere in modo astratto e generale il comportamento dell'algoritmo, indipendentemente dal contesto. In particolare si studia come il **tempo di esecuzione** cresce al crescere della dimensione dell'input n .

- $T(n)$ rappresenta la **funzione di costo dell'algoritmo**.

Un algoritmo è fatto di vari passi, un **passo** può essere considerato come un'operazione di base che richiede un tempo costante.

Diremo che due algoritmi sono più o meno efficienti a seconda di come cresce la loro funzione $T(n)$: *un algoritmo la cui funzione di costo cresce più lentamente sarà, per input sufficientemente grandi, più efficiente di uno con crescita più rapida.*

Dalla necessità di classificare i vari algoritmi nascono le **notazioni asintotiche**, che ci permettono per l'appunto di rappresentare questa idea di crescita trascurando i dettagli che non influiscono sul comportamento generale dell'algoritmo.

Confronto tra le principali funzioni di crescita

Alcune funzioni crescono lentamente, altre molto rapidamente. Immaginiamo di avere diversi algoritmi che risolvono lo stesso problema, ma con funzioni di costo differente:

- uno è proporzionale a $a \log n$
- un altro a n
- un altro ancora a n^2

Per valori piccoli di n le differenze possono sembrare trascurabili, ma per input abbastanza grandi la situazione cambia, ad esempio con $n = 1000$:

$$\log_2 n \approx 10, \quad n = 1000, \quad n^2 = 10^6, \quad 2^n \approx 10^{301}$$

In generale abbiamo diversi tipi di crescita:

1. **Funzioni costanti** $T(n) = 1$ rappresentano algoritmi che impiegano sempre lo stesso numero di operazioni indipendentemente da n
 - Accesso ad un elemento di un array
2. **Funzioni logaritmiche** $T(n) = \log n$ descrivono algoritmi che riducono il problema di un fattore costante a ogni passo
 - Ricerca binaria
3. **Funzioni lineari** $T(n) = n$ corrispondono ad algoritmi che analizzano tutti gli elementi dell'input una sola volta
 - Scansione di un array
4. **Funzioni linearitmiche** $T(n) = n \log n$ caratterizzano algoritmi efficienti di ordinamento
 - Merge sort e Heap sort
5. **Funzioni polinomiali** $T(n) = n^k$ includono molti algoritmi praticabili ma diventano rapidamente onerose per $k > 2$
 - Bubble sort
6. **Funzioni esponenziali e fattoriali**: descrivono problemi di natura combinatoria per i quali il numero di soluzioni cresce esplosivamente con n
 - Calcolo di tutte le permutazioni

Sono elencate in ordine di crescita crescente: una funzione situata più in basso cresce più rapidamente di tutte quelle sopra di essa

Termini e fattori dominanti e trascurabili

Quando si analizza il tempo di esecuzione di un algoritmo la funzione di costo $T(n)$ può contenere più termini, si chiama *termine dominante* quello che determina l'andamento complessivo della funzione per valori grandi di n , quando abbiamo più termini quello che dobbiamo fare è:

- Eliminare i termini di ordine inferiore, cioè quelli che crescono più lentamente
 - Si ignorano i fattori costanti che moltiplicano i termini
 - Si conserva solo il termine che domina
- Di seguito degli esempi:

1. $T(n) = 3n^2 + 5n + 10$ in questa funzione domina il $3n^2$ ed essendo 3 un coefficiente moltiplicativo questo è trascurabile quindi possiamo dire che questo algoritmo viene descritto da n^2
2. $T(n) = n \log n + 100n$ qui il termine che cresce più velocemente è $n \log n$

Le notazioni asintotiche

Con il termine **asintotico** indichiamo il comportamento di una funzione quando la variabile indipendente cresce senza limiti, possiamo quindi affermare che:

Una notazione asintotica è un modo sintetico per descrivere il modo in cui una funzione cresce rispetto a un'altra. Essa non ci dice quanto vale esattamente una funzione, ma quanto rapidamente aumenta rispetto ad altre funzioni al crescere di n

Esempio: dire che $T(n)$ è "dell'ordine n^2 " significa che per input molto grandi il numero di operazioni richieste dall'algoritmo cresce in modo proporzionale a n^2

In generale le notazioni asintotiche forniscono un linguaggio per esprimere relazioni del tipo:

- $f(n)$ cresce non più rapidamente / alla stessa velocità / più rapidamente di $g(n)$
Esse si basano quindi sul confronto tra le due funzione nel limite per $n \rightarrow \infty$ cioè osservando il loro comportamento quando la dimensione diventa molto grande

Esempio: date queste due funzioni:

$$f(n) = 3n^2 + 5n + 7, g(n) = n^2$$

la differenza può sembrare significativa, ma man mano che n cresce il termine $3n^2$ diventa dominante, quindi possiamo affermare che $f(n)$ e $g(n)$ hanno la stessa crescita asintotica

La notazione Θ (theta)

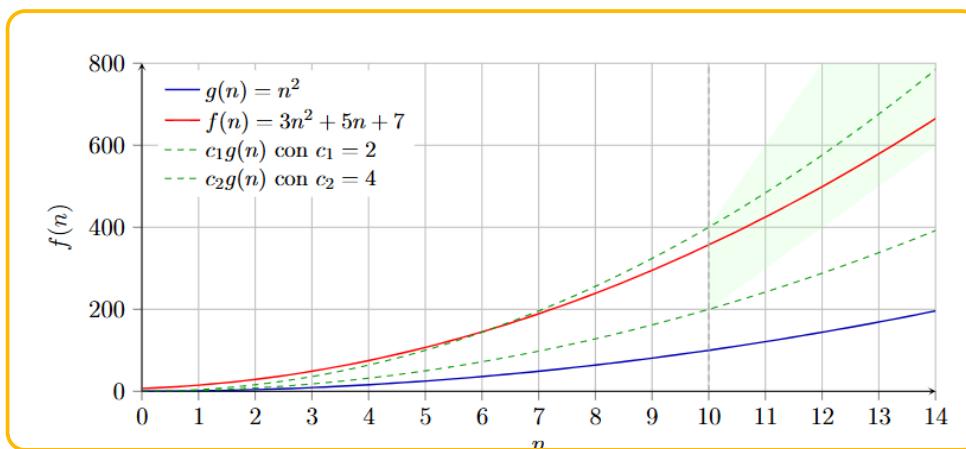
La notazione Θ fornisce un **vincolo asintotico stretto** ovvero *rappresenta l'insieme di tutte le funzioni che crescono alla stessa velocità di una funzione di riferimento $g(n)$* . Formalmente scriviamo che:

$$\Theta(g(n)) = \{f(n) | \exists c_1, c_2, n_0 > 0 \text{ tali che } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$$

In altre parole, a partire da una certa soglia n_0 la funzione $f(n)$ è sempre compresa tra due multipli costanti della funzione di riferimento $g(n)$.

Praticamente la notazione Θ descrive una crescita bilanciata: $f(n)$ cresce in modo simile a $g(n)$ differendo al più per un fattore costante. Questo significa che per input di n molto grandi possiamo dire che:

$$\frac{f(n)}{g(n)} \rightarrow k \text{ con } 0 < k < \infty$$



da questa immagine capiamo subito il significato della definizione formale infatti dopo un certo n (in questo caso $n = 7$) la nostra funzione resta confinata nella fascia delimitata dalla linea tratteggiata

Esempi:

- $5n + 20 = \theta(n)$ perché la costante additiva non influisce sul comportamento
- $\frac{1}{2}n^3 + 100 = \theta(n^3)$

La notazione θ è la più informativa poiché fornisce sia un limite superiore sia uno inferiore, se $f(n) = \Theta(g(n))$ allora $f(n)$ è simultaneamente:

- $O(g(n))$
 - $\Omega(g(n))$
- Capiremo successivamente il perché

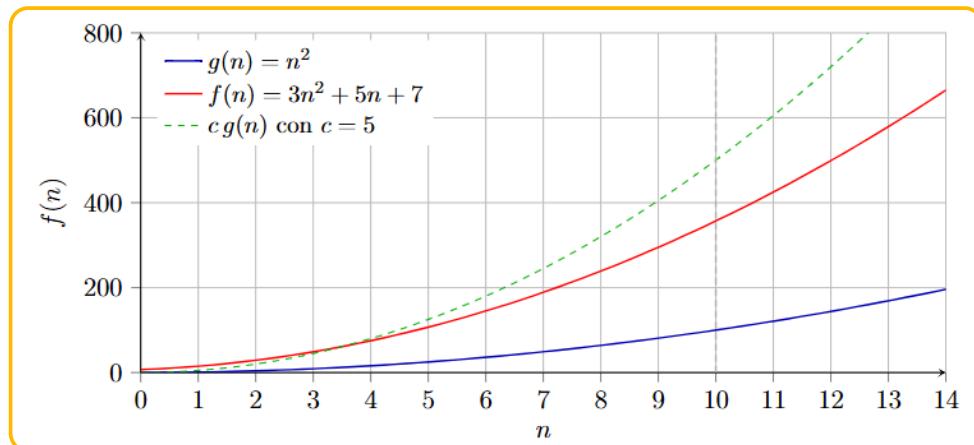
La notazione O (Big-Oh)

La notazione O rappresenta un **limite superiore asintotico** ovvero descrive *funzioni che per n sufficientemente grandi non crescono più rapidamente di una funzione di riferimento $g(n)$* , formalmente si definisce come:

$$O(g(n)) = \{f(n) | \exists c > 0, n_0 > 0 \text{ tali che } 0 \leq f(n) \leq cg(n), \forall n \geq n_0\}$$

direi che $f(n) = O(g(n))$ significa che, a partire da una certa soglia n_0 il valore di $f(n)$ non supera mai quello di $cg(n)$

Praticamente la notazione O serve per stabilire una stima superiore del comportamento asintotico infatti garantisce che $f(n)$ non crescerà mai più velocemente di $g(n)$ per n grandi



Attraverso questa immagine riusciamo subito a capire la definizione formale, infatti dopo una n la funzione $f(n)$ non cresce più velocemente di $g(n)$

Esempi:

- $7n + 200 = O(n)$
- $\log n = O(n)$
- $n = O(n^2)$

la notazione O indica solo un **limite superiore** se diciamo che $T(n) = O(n^2)$ stiamo affermando che l'algoritmo non richiede mai più di un tempo proporzionale a n^2 ma potrebbe essere anche più efficiente. Di solito questa notazione viene usata per descrivere il **caso peggiore**

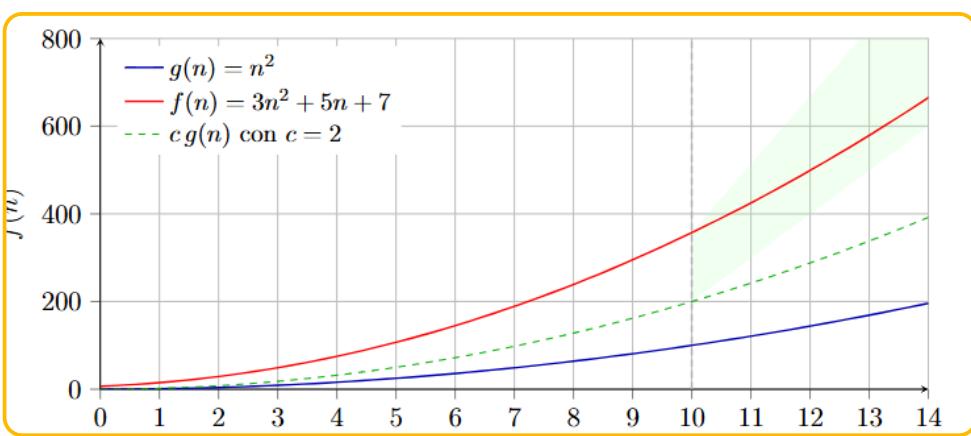
La notazione Ω (Big-Omega)

La notazione Ω rappresenta un **limite inferiore asintotico** ovvero descrive funzioni *che per n sufficientemente grande crescono almeno tanto rapidamente quanto una funzione di riferimento $g(n)$* formalmente si definisce come:

$$\Omega(g(n)) = \{f(n) | \exists c > 0, n_0 > 0 \text{ tali che } 0 \leq cg(n) \leq f(n), \forall n \geq n_0\}$$

Ciò significa che a partire da un certo punto n_0 il valore di $f(n)$ è sempre maggiore o uguale a una costante moltiplicativa di $g(n)$

Praticamente la notazione Ω descrive una crescita minima garantita, infatti oltre una certa soglia la funzione $f(n)$ rimane sempre al di sopra della funzione di riferimento $g(n)$



Attraverso questa immagine è abbastanza intuitiva la definizione formale, oltre una certa n la nostra funzione $f(n)$ rimane sempre al di sopra della curva $cg(n)$

La notazione Ω indica un **limite inferiore** e appunto per questo viene utilizzata per descrivere il **caso migliore**

Esempi:

- $5n + 200 = \Omega(n)$
- $n^3 = \Omega(n^2)$

Le notazioni o e ω (piccole)

Accanto a O e Ω che rappresentano limiti asintotici ampi, esistono le loro controparti minuscole o e ω che esprimono limiti stretti

- La prima $o(g(n))$ esprime che $f(n)$ cresce **più lentamente** di $g(n)$:

$$f(n) = o(g(n)) \Leftrightarrow \forall c > 0, \exists n_o > 0 \text{ tale che } f(n) < cg(n), \forall n \geq n_o$$

Significa che il rapporto $\frac{f(n)}{g(n)}$ tende a zero quindi $f(n)$ cresce in modo più lento di $g(n)$ che per n grandi il suo contributo diventa trascurabile

Praticamente possiamo dire che un tempo di esecuzione $o(n^2)$ significa che esso cresce più lentamente di qualunque funzione proporzionale a n^2

- La seconda $\omega(g(n))$ ha un significato opposto descrive infatti funzioni che crescono **più rapidamente** di $g(n)$:

$$f(n) = \omega(g(n)) \Leftrightarrow \forall c > 0, \exists n_0 > 0 \text{ tale che } f(n) > cg(n), \forall n \geq n_0$$

Significa che il rapporto $\frac{f(n)}{g(n)}$ tende a infinito

Praticamente possiamo dire che un tempo di esecuzione $\omega(n^2)$ significa che la nostra funzione cresce più velocemente di qualunque funzione proporzionale a n^2

Possiamo dunque pensare alle notazioni o e ω come versioni esclusive di O e Ω che indicano una crescita più lenta o più rapida che non raggiunge mai quella della funzione di riferimento

Conclusioni sulle notazioni asintotiche

La scala delle notazioni è la seguente

$$\omega(g(n)) \supseteq \Omega(g(n)) \supseteq \Theta(g(n)) \subseteq O(g(n)) \subseteq o(g(n))$$

Oltre i casi limite

Cosa accade davvero in pratica

Le notazioni asintotiche sono strumenti potenti per classificare gli algoritmi, ma è importante ricordare che esse rappresentano un modello astratto del comportamento di un programma. Dire che un algoritmo ha complessità $O(g(n))$ non significa che esegua esattamente $g(n)$ operazioni ma solo che per n grandi il suo tempo di

esecuzione non cresce più rapidamente di una funzione $g(n)$, di solito in questo tipo di analisi si distinguono tre casi:

- $O(g(n))$ che descrive il **caso peggiore** ovvero il limite superiore del tempo di esecuzione
- $\Omega(g(n))$ che descrive il **caso migliore** ovvero il limite inferiore del tempo di esecuzione
- $\Theta(g(n))$ indica che il tempo di esecuzione è compreso tra i due limiti precedenti, fornisce quindi una descrizione asintoticamente esatta di quello che è il tempo di esecuzione

Esempio 1: La ricerca lineare

Nel caso peggiore, una ricerca lineare in un array di n elementi richiede di esaminare tutti gli elementi:

$$T_{\text{worst}}(n) = O(n)$$

Tuttavia, se l'elemento cercato si trova spesso nelle prime posizioni, il numero medio di confronti può essere molto più basso, ad esempio circa $n/2$. In tal caso, il comportamento medio si avvicina a:

$$T_{\text{avg}}(n) = \Theta(n/2) = \Theta(n)$$

ma il tempo effettivo percepito sarà spesso inferiore rispetto al limite teorico.

Esempio 2: L'ordinamento tramite QuickSort

Il QuickSort ha una complessità nel caso peggiore di $O(n^2)$, che si verifica quando il pivot scelto in ogni passo è sempre il minimo o il massimo elemento (ad esempio, se l'array è già ordinato). Tuttavia, nella pratica, grazie alla scelta casuale del pivot o a tecniche di bilanciamento, l'algoritmo si comporta quasi sempre come un algoritmo $O(n \log n)$. Il suo caso medio è quindi molto più rappresentativo delle prestazioni reali:

$$T_{\text{avg}}(n) = \Theta(n \log n)$$

Esempio 3: La ricerca binaria

La ricerca binaria ha caso migliore $\Omega(1)$ (quando l'elemento cercato è esattamente quello centrale) e caso peggiore $O(\log n)$. Nella pratica, tuttavia, ogni ricerca richiede un numero di passi che dipende in modo logaritmico da n , per cui il comportamento effettivo si avvicina costantemente al caso peggiore.

Questi esempi ci mostrano che le notazioni non devono essere interpretate come previsioni puntuali, ma come **descrizioni qualitative del comportamento**, le notazioni vanno interpretate in questo modo:

- **Caso peggiore** ci garantisce la soglia di sicurezza, utile in contesti critici
- **Caso migliore**: indica il limite di ottimalità teorica, ma non sempre raggiungibile
- **Caso medio**: è spesso quello più significativo nella valutazione pratica

Quando le costanti contano

Nell'analisi asintotica degli algoritmi, è prassi trascurare i **fattori costanti** e i **termini di ordine inferiore**, questo approccio consente di concentrarsi sulla crescita dominante della funzione per n molto grandi, tuttavia questa cosa nella pratica quotidiana per input di piccole o medie dimensioni dovrebbe essere attenzionata

Esempio:

$$T_1(n) = 100n, \quad T_2(n) = 5n \log_2 n$$

dal punto di vista asintotico T_1 è più efficiente, tuttavia il fattore costante 100 nel primo algoritmo può renderlo più lento del secondo per una vasta gamma di valori realistici di n , vediamolo con $n = 100$

$$T_1(100) = 10000, \quad T_2(100) = 5 \times 100 \times \log_2 100 \approx 5 \times 100 \times 6.64 = 3320$$

In questo caso, l'algoritmo con complessità peggiore risulta più veloce. Questo esempio mette in evidenza un punto importante: le notazioni asintotiche descrivono il comportamento per n grandi, ma non sempre riflettono le prestazioni reali su input di dimensioni moderate

Risoluzione delle equazioni di ricorrenza

Introduzione alle equazioni di ricorrenza

Per analizzare il tempo di esecuzione di tutti gli algoritmi (e in particolare dei ricorsivi) si fa uso delle *equazioni di ricorrenza* che esprimono il costo totale $T(n)$ in funzione del costo dei sotto-problemi e del lavoro aggiuntivo

Definizione

Un'equazione di ricorrenza è una relazione del tipo:

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

con:

- $T(n)$ costo per il risolvere il problema di dimensione n
- a è il numero di sotto-problemi in cui il problema viene suddiviso
- ciascun sotto-problema ha dimensione $\frac{n}{b}$
- $f(n)$ rappresenta il lavoro non ricorsivo (cioè il costo per dividere, combinare o gestire i sotto-problemi)

Risoluzione dell'equazione di ricorrenza

La *soluzione* (anche detta *forma esplicita*) indica un'espressione chiusa per $T(n)$, utile per comprendere come cresce il tempo di esecuzione al crescere di n . Per fare questa cosa usiamo tre metodi:

1. **Metodo dell'albero di ricorsione:** si rappresenta la ricorsione come un albero, si calcola il costo per livello poi sommati per conoscere il costo totale, questo metodo viene usato principalmente per capire dove si concentra il lavoro (livelli iniziali o finali).
2. **Metodo della sostituzione:** si formula un'ipotesi sul comportamento asintotico di $T(n)$ e la si dimostra per induzione
3. **Metodo Master:** fornisce un risultato generale che consente di determinare l'ordine di grandezza di $T(n)$ confrontando $f(n)$ con $n^{\log_b a}$

Identificare l'equazione di ricorrenza

Fasi della ricorsione

Ogni algoritmo ricorsivo può essere idealmente scomposto in tre momenti:

1. *fase di suddivisione*: il problema viene suddiviso in più sotto-problemi di dimensione minore
2. *fase di risoluzione*: ciascun sotto-problema viene affrontato a sua volta in modo ricorsivo
3. *fasi di composizione*: i risultati parziali vengono combinati

Detto ciò possiamo aggiungere alla spiegazione dell'equazione di ricorrenza le seguenti cose:

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

- a è il numero di sotto-problemi prodotti
- b il fattore di riduzione
- $f(n)$ il costo complessivo delle operazioni non ricorsive

Esempio - Ricerca binaria

Nel caso della ricerca binaria, l'algoritmo esami un intervallo ordinato, ne calcola il punto medio e confronta il valore cercato con l'elemento corrispondente. Se il valore coincide la ricerca termina, altrimenti viene richiamata ricorsivamente sull'altra metà

```
int binarySearch(int A[], int low, int high, int x) {
    if (low > high) return -1;

    int mid = (low + high) / 2;
    if (A[mid] == x)
        return mid;
```

```

    else if (A[mid] > x)
        return binarySearch(A, low, mid - 1, x);
    else
        return binarySearch(A, mid + 1, high, x);
}

```

In ogni chiamata si effettua al massimo una sola chiamata ricorsiva su metà dell'intervallo, mentre il lavoro non ricorsivo è costante. L'equazione di ricorrenza quindi è:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Ricorrenze non uniformi

Non tutti gli algoritmi ricorsivi possono essere descritti mediante un'equazione del tipo:

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

in cui il problema viene suddiviso in un numero costante di sotto-problemi. In molti casi reali, la dimensione dei sotto-problemi può variare a seconda dei dati.

Esempio - QuickSort

- **Caso ideale:** l'elemento *pivot* scelto divide l'array esattamente in due parti uguali, ciascuna di dimensione $\frac{n}{2}$
- **Caso reale:** il *pivot* scelto non divide il nostro array in due array di dimensioni diverse

Nel caso medio possiamo dire che:

$$T(n) = T(\alpha n) + T((1 - \alpha)n) + n$$

dove il parametro α rappresenta la frazione di elementi che ricadono nella prima metà dopo la partizione ($0 < \alpha < 1$).

Quando $\alpha = \frac{1}{2}$ siamo nel caso bilanciato (il caso ideale), quando α si avvicina a 0 o a 1 il costo peggiora fino al caso limite.

Analisi

L'esempio appena fatto ci mostra come occorre analizzare la ricorrenza caso per caso, e non sempre possibili applicare le formule standard.

Il metodo dell'albero di ricorsione

Il metodo dell'albero di ricorsione nasce dall'idea di visualizzare come vengono effettuate le chiamate ricorsive. In questo modo diventa possibili analizzare passo dopo passo come si distribuisce il lavoro complesso nei diversi livelli di ricorsione, tutto ciò viene fatto con la seguente notazione

Notazione

ricordiamo sempre l'equazione di ricorrenza generale: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

- La *radice* corrisponde al problema iniziale di dimensione n
- i *nodi interni* rappresentano le chiamate ricorsive generate a ciascun livello
- ogni *nodo* è etichettato con il costo del lavoro locale $f(n_i)$ dove n_i è la dimensione del sotto-problema corrispondente
- i *figli* di un nodo rappresentano le chiamate generate da quel nodo, ognuna di dimensione ridotta di un fattore b

Analisi

La costruzione di questo albero permette di scomporre la ricorsione in livelli. A ogni livello k il numero di nodi è a^k di conseguenza la dimensione dei sotto-problemi è $\frac{n}{b^k}$. Il costo totale associato al livello k viene descritto come:

$$C_k = a^k f\left(\frac{n}{b^k}\right)$$

cioè il numero di nodi di quel livello moltiplicato per il costo del lavoro svolto in ciascun nodo. Quindi il costo totale diventa:

$$T(n) = \sum_{k=0}^L C_k \text{ ovvero } \sum_{k=0}^L a^k f\left(\frac{n}{b^k}\right),$$

con L che rappresenta la profondità dell'albero ossia il numero di livello fino a quando la dimensione del problema non si riduce a una costante

Questo approccio fornisce un modo intuitivo per stimare l'andamento del lavoro, ma è anche il modo migliore per averne una rappresentazione visiva. Il comportamento della somma può essere descritto come una serie geometrica, in cui ogni livello contribuisce con un costo proporzionale al precedente. Da cui distinguiamo tre casi:

1. **Primi livelli:** somma dominata dai primi livelli quindi il costo è determinato dal lavoro iniziale
2. **Livelli intermedi:** tutti i livelli hanno lo stesso ordine di grandezza quindi il costo aumenta con l'aumentare dei livelli
3. **Livelli finali:** Somma dominata dai livelli inferiori quindi il costo dominante si sposta alla fine dell'albero

Esempi

Esempio 1 - Ricerca binaria: consideriamo l'equazione della ricerca binaria trovata prima:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Possiamo immaginare l'albero della ricorsione come una catena di chiamate successive, in cui ogni nodo produce un unico figlio di dimensione dimezzata. Ad ogni livello il lavoro è costante (ovvero 1). Il costo totale si ottiene sommando il contributi di tutti i livelli:

$$T(n) = 1 + 1 + 1 + 1 + \dots + 1$$

dove il numero dei termini è pari a $\log_2 n + 1$. Da qui risulta immediatamente che: $T(n) = O(\log n)$

Il metodo della sostituzione

Un secondo approccio è il metodo della sostituzione, dove al posto di avere un'analisi attraverso una rappresentazione visiva abbiamo un'analisi più analitica

Definizione

Questo approccio consiste nel: formulare un'ipotesi sulla forma asintotica della soluzione e nel dimostrare che tale ipotesi è corretta attraverso un ragionamento induttivo.

Praticamente: Si parte dall'equazione di ricorrenza e, osservando la struttura del problema, si tenta di indovinare la crescita di $T(n)$ ad esempio $O(n)$, $O(n \log n)$ o $O(n^2)$. Una volta formulata questa ipotesi la si sostituisce nell'equazione e si verifica se l'uguaglianza risulta soddisfatta per valori sufficientemente grandi.

Step da seguire

1. Si formula un'ipotesi sul comportamento asintotico di $T(n)$ (spesso lo si fa attraverso metodi più intuitivi come l'albero di ricorsione)
2. Si sostituisce questa ipotesi all'interno dell'equazione di ricorrenza e si verifica se l'uguaglianza risulta soddisfatta
3. Se necessario si aggiusta l'ipotesi e si riparte dallo step 1

Ricordiamo che: questo metodo non si limita a fornire una soluzione numerica: esso insegnà a riconoscere e controllare la correttezza delle ipotesi che emergono in modo intuitivo

Esempi

Esempio 1 - Ricerca binaria: consideriamo l'equazione della ricerca binaria trovata prima:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

L'intuizione (l'albero di ricorsione) suggerisce una crescita logaritmica. Supponiamo quindi:

$$T(n) \leq c \log_2 n$$

sostituendo otteniamo che:

$$\begin{aligned} T(n) &\leq c \log_2 \frac{n}{2} + 1 = \\ &= c(\log_2 n - 1) + 1 \\ &= c \log_2 n - c + 1 \end{aligned}$$

Affinché la diseguaglianza $T(n) \leq c \log_2 n$ sia rispettata, è sufficiente che:

$$-c + 1 \leq 0 \text{ cioè } c \geq 1$$

Anche in questo caso la nostra ipotesi è coerente ed è $T(n) = O(\log n)$

Metodo basato sul teorema Master

Il teorema Master fornisce una regola generale per determinare in modo sistematico l'ordine di grandezza.

Definizione

Le equazioni che questo teorema risolve sono del tipo:

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

dove:

- $a \geq 1$ è il numero di sotto-problemi in cui viene suddiviso il problema
- $b > 1$ è il fattore di riduzione della dimensione
- $f(n)$ è il costo del lavoro non ricorsivo

L'idea del teorema master è confrontare la funzione $f(n)$ con la quantità $n^{\log_b a}$, a seconda di quale dei due termini cresce più rapidamente, si individuano tre comportamenti distinti.

Enunciato del teorema Master

Sia

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ con } a \geq 1, b > 1 \text{ e } f(n) \text{ una funzione positiva}$$

Caso 1: Il lavoro ricorsivo domina

Se esiste una costante $\varepsilon > 0$ tale che

$$f(n) = O(n^{\log_b a - \varepsilon}),$$

cioè il lavoro non ricorsivo $f(n)$ è asintoticamente più piccolo del lavoro interno alla ricorsione $n^{\log_b a}$ (polynomialmente), allora la complessità è:

$$T(n) = \Theta(n^{\log_b a}).$$

In questo caso domina il costo generato dalla parte ricorsiva dell'algoritmo (le chiamate interne). Esempio tipico: Moltiplicazione di Strassen, dove $a = 7, b = 2, f(n) = n^2$.

Caso 2: I lavori sono equivalenti (a meno di un fattore logaritmico)

Se

$$f(n) = \Theta(n^{\log_b a} \log^k n),$$

ossia il lavoro non ricorsivo ha lo stesso ordine di grandezza del lavoro ricorsivo $n^{\log_b a}$ (a meno di un fattore logaritmico $\log^k n$, con $k \geq 0$), allora la complessità è:

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n).$$

In questo caso, tutti i livelli dell'albero di ricorsione contribuiscono in modo equivalente al costo totale, e la moltiplicazione per un fattore $\log n$ nel termine $T(n)$ si traduce in un incremento di un ordine logaritmico nel costo complessivo.

(Generalizzato per $k = 0$): Se $f(n) = \Theta(n^{\log_b a})$, allora $T(n) = \Theta(n^{\log_b a} \log n)$.

Caso 3: Il lavoro non ricorsivo domina

Se esiste una costante $\varepsilon > 0$ tale che

$$f(n) = \Omega(n^{\log_b a + \varepsilon}),$$

cioè il lavoro non ricorsivo $f(n)$ cresce più velocemente del lavoro interno $n^{\log_b a}$ (polynomialmente), e se inoltre è verificata una condizione di regolarità (detta condizione di dominanza):

$$af\left(\frac{n}{b}\right) \leq cf(n)$$

per una costante $c < 1$ e per n sufficientemente grande, allora la complessità è:

$$T(n) = \Theta(f(n)).$$

In questo caso, la parte ricorsiva diventa trascurabile rispetto al lavoro non ricorsivo.

Esempio tipico: Una ricorrenza come $T(n) = 2T(n/2) + n^2$, dove il termine n^2 domina.

Interpretazione intuitiva

Il teorema Master può essere interpretato come un modo sintetico per stabilire quale parte dell'albero di ricorsione domina il costo complessivo:

1. nel *primo caso* domina il livello inferiore, dove si accumula la maggior parte di lavoro ricorsivo
 2. nel *secondo caso* ogni livello contribuisce in egual misura, e il numero di livelli aggiunge un ulteriore fattore logaritmico
 3. nel *terzo caso* domina il livello superiore, dove il lavoro ricorsivo è prevalente
- Pur non essendo applicabile in tutti i casi (ad esempio quando i sotto-problemi hanno dimensioni diverse) rimare uno degli strumenti più rapidi per l'analisi asintotica

Confronto tra funzioni

Uno degli aspetti più delicati del teorema master consiste nel comprendere come confrontare la funzione non ricorsiva $f(n)$ con la quantità $n^{\log_b a}$. Ricordiamo che:

- $n^{\log_b a}$ descrive quanto grande diventa l'albero di ricorsione
- $f(n)$ misura il costo aggiuntivo sostenuto a ciascun livello

Se $f(n)$ cresce molto meno di $n^{\log_b a}$, il termine ricorsivo domina, se cresce di più, prevale il termine non ricorsivo; se le due funzioni hanno crescita simile, i contributi si equilibrano. Viene introdotto il parametro ε per formalizzare questa differenza di crescita.

Esempio 1. Consideriamo la ricorrenza $T(n) = 2T(n/2) + n$. Qui $a = 2$, $b = 2$ e quindi $n^{\log_b a} = n$. Poiché $f(n) = n$ ha la stessa crescita, non esiste un $\varepsilon > 0$ tale che $f(n)$ sia né più piccolo né più grande di un fattore polinomiale rispetto a $n^{\log_b a}$: ci troviamo dunque nel caso intermedio, e la soluzione è $T(n) = \Theta(n \log n)$.

Esempio 2. Consideriamo invece $T(n) = 2T(n/2) + n^2$. In questo caso $n^{\log_b a} = n$, ma $f(n) = n^2$ cresce più rapidamente di un intero fattore polinomiale, cioè $f(n) = \Omega(n^{1+\varepsilon})$ con $\varepsilon = 1$. Qui il termine non ricorsivo domina e la soluzione è $T(n) = \Theta(n^2)$.

In generale, il ruolo di ε non è quello di un valore da calcolare ma di un indicatore concettuale: serve a **distinguere** tra crescite *molto piccole* o *molto più grandi* rispetto a $n^{\log_b a}$

Esempi

Esempio 1 - Ricerca binaria

La ricerca binaria è descritta dalla ricorrenza

$$T(n) = T(n/2) + 1.$$

In questo caso $a = 1$, $b = 2$ e $f(n) = 1$. Calcoliamo il termine di riferimento $n^{\log_b a}$: poiché $\log_2 1 = 0$, si ottiene $n^{\log_2 1} = n^0 = 1$. Confrontiamo ora $f(n)$ con questo valore:

$$f(n) = 1 = \Theta(1) = \Theta(n^{\log_2 1}).$$

Siamo dunque nel **secondo caso** del Teorema Master, quello in cui $f(n)$ ha lo stesso ordine di grandezza del termine ricorsivo.

Applicando la formula corrispondente, otteniamo:

$$T(n) = \Theta(n^{\log_2 1} \log n) = \Theta(\log n).$$

In ogni passo della ricerca binaria, il problema viene dimezzato, ma il lavoro svolto ad ogni livello (una sola comparazione) è costante. Poiché ci sono $\log_2 n$ livelli fino a ridurre il problema a un singolo elemento, il costo totale cresce in modo logaritmico. Il Teorema Master, in questo caso, conferma in modo immediato ciò che l'intuizione suggerisce: ogni livello contribuisce in modo uniforme, e il numero di livelli determina la crescita complessiva.

Heap

Introduzione

Si usa l'heap per implementare efficientemente le **code con priorità**. Questo tipo di coda non serve gli elementi in base al momento di arrivo, ma in base a una caratteristica intrinseca chiamata **chiave** (o priorità). Spesso viene astratta come un albero (successivamente la implementiamo come array)

Caratteristiche principali

Lo heap è un albero con le seguenti caratteristiche:

- *Binario*: ogni nodo ha al massimo due figli.
- *Posizionale*: è possibile distinguere figlio destro e figlio sinistro
- *Completo*: In ogni livello i dell'albero sono presenti 2^i nodi
- *Parzialmente ordinato*: Il valore di un nodo sarà sempre maggiore o uguale a quella dei suoi figli

Esempio - il pronto soccorso

Un esempio reale di coda con priorità è il triage ospedaliero. I codici colore rappresentano le priorità (chiavi):

- Bianco/Verde (codici bassi/meno urgenti).
- Giallo/Rosso (codici alti/urgenti).

Se un paziente arriva in codice giallo, supera automaticamente tutti quelli in attesa con codice verde, indipendentemente da quanto tempo stiano aspettando. La fila viene rispettata solo tra pazienti con lo stesso codice.

Confronto con altre implementazioni delle code con priorità

Specifiche iniziali

Per capire perché nasce lo heap dobbiamo prima fare un'analisi di come le varie strutture dati si comportano (in termini di complessità algoritmica) quando si lavora con una coda con priorità. Una implementazione di una coda con priorità deve avere necessariamente i seguenti metodi:

1. **Inserimento**: Aggiungere un nuovo elemento.

2. **Estrazione (del minimo):** Rimuovere e restituire l'elemento prioritario.
3. **Decremento (Decrease Key):** Aumentare la priorità di un elemento e quindi diminuire il valore della sua chiave
4. **Minimo:** Consultare l'elemento prioritario senza estrarre.

Le analisi successive verranno fatte supponendo n nodi

Vs Array Disordinato

- **Inserimento:** $O(1)$. Basta mettere l'elemento nella prima posizione libera
- **Estrazione/Minimo:** $\Theta(n)$. Bisogna scorrere tutto l'array per trovare il valore più piccolo.
- **Decremento:** $O(1)$ se abbiamo il puntatore all'elemento, ma non aiuta l'estrazione successiva.

Vantaggio: Inserimento immediato.

Svantaggio: Costo lineare per trovare il minimo, inaccettabile per code frequenti.

Vs Array Ordinato

- **Minimo:** $O(1)$. Il minimo è sempre in prima posizione (o ultima, a seconda dell'ordinamento).
- **Inserimento:** $O(n)$. Per inserire un valore e mantenere l'ordine, bisogna "shiftare" (spostare) tutti gli elementi successivi.
- **Estrazione:** $O(n)$. Anche togliendo il primo elemento, bisogna riorganizzare l'array (shift verso sinistra).

Vantaggio: Accesso rapido al minimo.

Svantaggio: Inserimento ed estrazione costosi.

Vs Alberi Binari di Ricerca Bilanciati (BBST)

Utilizzando un BBST (*Balanced Binary Search Tree*), l'altezza dell'albero è garantita essere logaritmica ($O(\log n)$).

- **Tutte le operazioni (Inserimento, Estrazione, Ricerca, Decremento):** $O(\log n)$.

Sebbene $O(\log n)$ sia ottimo i BBST sono strutture complesse da implementare e mantenere (richiedono puntatori e ribilanciamenti).

L'obiettivo dello **Heap** è ottenere le stesse prestazioni asintotiche dei BBST ($O(\log n)$) ma con una struttura molto più semplice, gestibile tramite un array e senza l'uso esplicito di puntatori.

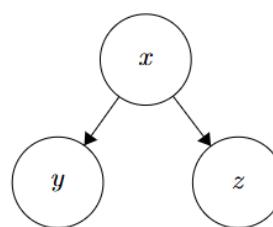
Come è fatto uno Heap

Introduzione

Un heap è un albero binario completo dove a differenza di un BST (dove tutto il sottoalbero sinistro è minore del nodo e tutto il destro è maggiore), nello Heap c'è solo una relazione verticale:

$$\text{Key}[parent] \leq \text{Key}[figlio]$$

Non c'è alcuna relazione d'ordine specifica tra fratello destro e fratello sinistro. Questo è definito **ordinamento parziale**.



Dove $P(x) \geq P(y)$ e $P(x) \geq P(z)$. Si dirà ordinamento parziale in quanto non offrirà un'ordinamento uno a tutte le chiavi.

Tipi di Heap

Esistono due varianti di Heap, simmetriche tra loro:

1. **Min Heap:** La chiave di un nodo è sempre **minore o uguale** a quella dei suoi figli. La radice contiene il minimo assoluto.
2. **Max Heap:** La chiave di un nodo è sempre **maggior o uguale** a quella dei suoi figli. La radice contiene il massimo.

Operazioni sullo Heap

Di seguito la complessità e le implementazioni delle varie funzionalità di uno heap.

Minimo

Poiché usiamo un Min Heap, il minimo si trova sempre alla radice.

Costo: $\Theta(1)$ (Accesso diretto all'indice 1 dell'array).

Nota: la stessa cosa vale per il massimo in un MaxHeap

Decrease Key (Decremento di una chiave)

Se riduciamo il valore di una chiave (es. un nodo passa da 7 a 1), potremmo violare la proprietà dello heap (il figlio diventa più piccolo del padre).

Procedura:

1. Si aggiorna il valore.
2. Si confronta il nodo con il padre.
3. Se il nodo è minore del padre, si scambiano (**swap**).
4. Si ripete il procedimento risalendo verso la radice finché la proprietà non è ripristinata o si raggiunge la radice.

Costo: Nel caso peggiore si risale tutta l'altezza dell'albero. $O(\log n)$.

```
DECREASE-KEY(H, x, k)
    key(x) = k
    p = parent(x)
    while(p != NULL and key(p)>key(x)) do
        swap(x, p)
        x = p
        p = parent(x)
```

Inserimento

L'inserimento sfrutta la logica del *Decrease Key*:

1. Inseriamo il nuovo nodo con valore ∞ nella prima posizione libera
2. Poi facciamo una *Decrease Key* al nodo inserito con il suo valore reale.
3. L'elemento "risale" (bubble-up) fino alla sua posizione corretta.

Costo: Proporzionale all'altezza dell'albero. $O(\log n)$.

```
INSERT(H, k)
    x = new node(H)
    key(x) = k
    p = parent(x)
    while(p != NULL and key(p)>key(x)) do
        swap(x, p)
```

```

x = p
p = parent(x)

```

Procedura Heapify(i):

Si applica a un nodo i assumendo che i sottoalberi sinistro e destro siano già heap validi.

1. Confronto la chiave di i con il figlio sinistro (l) e il figlio destro (r).
2. Individuo il più piccolo tra i , l e r .
3. Se il minimo non è i , scambio i con il figlio minore.
4. Chiamo ricorsivamente *Heapify* sul figlio appena scambiato.

Analisi della complessità di Heapify:

L'equazione di ricorrenza (nel caso peggiore su un albero quasi completo) è approssimabile dal Teorema Master (Caso 2):

$$T(n) \leq T(2/3n) + \Theta(1) \implies T(n) = \Theta(\log n)$$

Il fattore $2/3$ deriva dal fatto che, in un albero non perfettamente bilanciato nell'ultimo livello, il sottoalbero più grande può contenere circa i $2/3$ dei nodi totali. Tuttavia, la complessità rimane legata all'altezza: $O(\log n)$.

```

HEAPIFY(H, x)
    y = left(x)
    z = right(x)
    min = x
    IF y != NULL AND key(y)<key(x)
        min = y
    IF y \neq NULL AND key(z)<key(min)
        min = z
    IF min != x
        swap(x, min)
        heapify(H, min)

```

Estrazione del Minimo

Per estrarre il minimo (la radice), non possiamo lasciare un buco.

1. Prendiamo l'ultimo elemento dell'array (quello più in basso a destra) e lo spostiamo alla radice al posto del minimo rimosso.
 2. Ora la struttura è integra, ma l'ordinamento è violato (un elemento grande è in testa).
 3. Bisogna far "scendere" questo elemento nella posizione corretta e lo facciamo usando la funzione *heapify*.
- Costo:** richiamando *heapify* questa procedura costa $O(\log n)$.

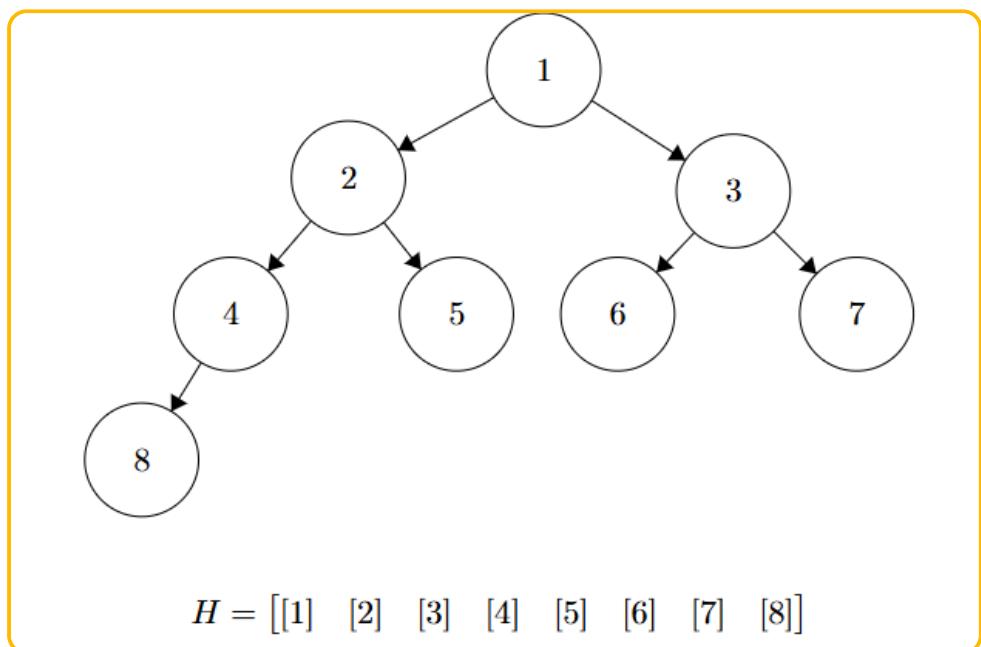
Heap usando un array

Il vero segreto dello Heap è che può essere implementato facilmente usando un **array**, eliminando la necessità di puntatori. E lo si fa usando le seguenti convenzioni:

- La radice è all'indice 1 (o 0, a seconda dell'implementazione; qui useremo l'indice 1 per semplificare le formule).
- Dato un nodo all'indice i :
 - Il **figlio sinistro** (*Left*) si trova a $2i$.
 - Il **figlio destro** (*Right*) si trova a $2i + 1$.
 - Il **genitore** (*Parent*) si trova a $\lfloor i/2 \rfloor$.

Queste operazioni possono essere eseguite in modo estremamente efficiente tramite **operazioni bitwise** (shift dei bit):

- Moltiplicare per 2 corrisponde a uno *shift a sinistra* ($i \ll 1$).
- Dividere per 2 corrisponde a uno *shift a destra* ($i \gg 1$).



Di seguito tutte le funzioni ridefinite per l'implementazione sottoforma di array del heap

Funzioni left, right, parent

```
int left(int i){
    return (2*i)+1; //Mettiamo +1 perché gli array partono da 0
}

int right(int i){
    return (2*i)+2;
}

int parent(int i){
    return (i-1)/2;
}
```

Heapify

```
void min_heapify(int i){
    int l = left(i);
    int r = right(i);
    int min = i;

    if(l < size && array[l] < array[min]) min = l;
    if(r < size && array[r] < array[min]) min = r;

    if(min != i){
        int scambio = array[i];
        array[i] = array[min];
        array[min] = scambio;
        min_heapify(min);
    }
}
```

```

void max_heapify(int i){

    int l = left(i);
    int r = right(i);
    int max = i;

    if(l < size && array[l] > array[max]) max = l;
    if(r < size && array[r] > array[max]) max = r;

    if(max != i){
        int scambio = array[i];
        array[i] = array[max];
        array[max] = scambio;
        heapify(max);
    }

}

```

Insert

```

void insert(int k){
    array[size++] = k;
    int i = size - 1;
    int p = parent(i);
    //Questa insert mantiene una struttura max-heap, per min-heap modificare in: array[p]>array[i]
    while(i>0 && array[p]<array[i]){
        int scambio = array[i];
        array[i] = array[p];
        array[p] = scambio;
        i = p;
        p = parent(i);
    }
}

```

Extract-min

```

int extractMax(){
    int max = array[0];

    int scambio = array[0];
    array[0] = array[size-1];
    array[size-1] = scambio;

    size--;
    max_heapify(0);

    return max;
}

```

Costruzione dello Heap

Come si costruisce uno heap partendo da un array disordinato di n elementi? di seguito due metodi per farlo

Metodo 1: Inserimenti successivi

Possiamo inserire gli elementi uno alla volta in uno heap vuoto (usando la funzione insert). Poiché ogni inserimento costa $O(\log n)$, per n elementi il costo totale è:

$$O(n \log n)$$

Metodo 2: Procedura Build-Min-Heap (Ottimizzata)

Esiste un metodo più efficiente che sfrutta la struttura. Prendiamo l'array così com'è e chiamiamo **Heapify** a ritroso, partendo dall'ultimo nodo interno fino alla radice.

Algoritmo:

```
FOR i = floor(n/2) DOWN TO 1:  
    Min-Heapify(array, i)
```

In pratica, sistemiamo prima i sottoalberi piccoli in basso, poi quelli medi, e infine la radice.

Analisi della complessità

A prima vista potrebbe sembrare $O(n \log n)$ perché chiamiamo Heapify $n/2$ volte. Tuttavia, l'altezza dei nodi varia:

- La maggior parte dei nodi è vicino al fondo (altezza bassa, costo Heapify basso).
- Pochissimi nodi sono in alto (altezza $\log n$).

La somma dei costi è data dalla serie:

$$\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} \cdot O(h) = n \sum_{h=0}^{\infty} \frac{h}{2^h}$$

Poiché la serie $\sum \frac{h}{2^h}$ converge a una costante, il costo totale è:

$$\Theta(n)$$

Costruire uno heap da zero quindi richiede **tempo lineare**.

Metodo 3: procedura Build-Max-Heap

Questa procedura è uguale a quella del min-heap ma richiamiamo max-heapify

```
void buildMaxHeap(){  
    for(int i = size/2-1; i >= 0; i--){  
        max_heapify(i);  
    }  
}
```

Heapsort

Analisi

Prendendo in esame il selection sort, capiamo subito che il problema principale di questo tipo di algoritmo è il tempo perso durante la ricerca del massimo, cosa che in un max-heap facciamo in tempo $O(1)$, da qui nasce l'heapsort

Algoritmo e implementazione

L'algoritmo heapsort inizia utilizzando *Build-max-heap* per costruire un max-heap nella array in input, l'elemento più grande è memorizzato nella radice, quindi ci basta scambiarlo con l'ultimo elemento nell'array per ordinarlo, poi richiamiamo max-heapify sull'array ma escludendo gli elementi ordinati, facendo questa cosa per ogni elemento otteniamo il nostro array ordinato.

```

void HeapSort(){
    buildMaxHeap();
    int len = size;
    for(int i = len-1; i > 2; i--){
        int scambio = array[i];
        array[i] = array[0];
        array[0] = scambio;
        size--;
        max_heapify(0);
    }
    size = len; //Nell'implementazione fatta serve per stampare il lo heap
}

```

Questo algoritmo ha complessità $O(n \log n)$

Esami

Nel file [6 - 29 ottobre 2025.pdf](#) ci sono alla fine degli esempi di esercizi di esame:

- Effettuare 13 estrazioni del minimo e mostrare cosa succede al nostro array
- Scrivere la procedura HeapMerge
- Scrivere la procedura ListMerge che segua delle regole specifiche
- Top-ten di punteggi più alti
- ecc...

Limiti degli ordinamenti con confronti

Albero di decisione

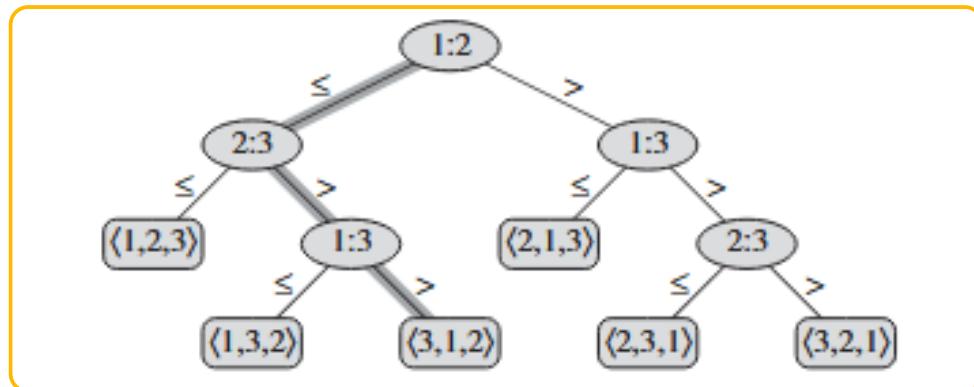
Definizione

Gli ordinamenti per confronti possono essere visti in modo astratto come degli alberi di decisione. Un albero di decisione è un albero binario pieno che rappresenta i confronti fra elementi fatti in un particolare algoritmo preso in esame.

Esempio

Attraverso un esempio è abbastanza intuitivo capire come funziona un albero di decisione:

Con $A = [1, 2, 3]$ abbiamo un albero di decisione del tipo:



Notiamo subito che il caso peggiore nell'ordinamento corrisponde all'altezza di questo albero

Teorema e dimostrazione

Teorema

Qualsiasi algoritmo di ordinamento per confronti richiede $\Omega(n \log n)$

Dimostrazione

Come abbiamo visto nell'esempio dell'albero di decisione ci basta determinare l'altezza di un albero di decisione dove possibile permutazione degli elementi compare come foglia.

Consideriamo quindi un albero di decisione:

- elementi da ordinare n
- altezza h
- numero di foglie l

ciascuna delle $n!$ permutazioni dell'input compare in una foglia, si ha quindi $n! \leq l$. Dal momento che un albero binario di altezza h non ha più di 2^h foglie si ha:

$$n! \leq l \leq 2^h$$

da questo estraiamo:

$$n! \leq 2^h \rightarrow \log_2(n!) \leq h$$

che ci indica una complessità di

$$\Omega(n \log n)$$

Conclusioni

Da questo capiamo che algoritmi come *heapSort* e *MergeSort* sono ottimi algoritmi di ordinamento usando i confronti

Algoritmi di ordinamento senza confronti

Counting sort

Premesse

Per superare la barriera imposta dall'albero di decisione abbiamo bisogno di algoritmi di ordinamento che non fanno confronti, e quindi presentiamo il counting sort. Per fare questa cosa il counting sort suppone che ciascuno degli n elementi in input sia un intero compreso tra 0 e k

Algoritmo

Per usare il counting sort abbiamo bisogno:

- A un array in input di n elementi interi che vanno da 0 – k
- C un array con k celle i cui valori indicano il numero di occorrenze di k in A
- B array di output

```
Counting-Sort(A, B, n):
    k = max(A)                      // valore massimo in A
    C = new Array(k+1)

    // inizializza C a 0
    for i = 0 to k:
        C[i] = 0

    // conta le occorrenze
    for i = 0 to n-1:
        C[A[i]] = C[A[i]] + 1

    // calcola le posizioni cumulative
    for i = 1 to k:
        C[i] = C[i] + C[i-1]
```

```

// costruisci l'array ordinato B in maniera stabile
for i = n-1 downto 0:
    B[C[A[i]] - 1] = A[i]      // -1 perché B è 0-indexed
    C[A[i]] = C[A[i]] - 1

```

Di seguito per step la spiegazione:

1. Cerchiamo il nostro k che sarebbe il massimo di A
2. Creiamo il nostro array C con $k + 1$ posizioni
3. *Primo FOR*: inizializza a 0 l'array C
4. *Secondo FOR*: controlliamo ogni elemento di A e incrementiamo di uno l'indice corrispondente in C
 - Dopo questo passo $C[i]$ contiene il numero di occorrenze di i in A
5. *Terzo FOR*: contiamo il numero di elementi in A minore o uguali i
6. *Quarto FOR*: inseriamo ogni elemento di A dentro B usando come indice il valore calcolato nel terzo passaggio, la riduzione di $C[A[i]]$ serve per fare in modo che il successivo elemento con valore uguale ad $A[j]$ venga inserito una posizione prima

Implementazione

```

void countingSort(int *A, int* B, int n){
    int k = max(A, n);
    int* C = new int[k+1];

    for(int i = 0; i <= k; i++)
        C[i] = 0;

    for(int i = 0; i < n; i++)
        C[A[i]] = C[A[i]] + 1;

    for(int i = 1; i <= k; i++)
        C[i] = C[i]+C[i-1];

    for(int i = n-1; i >= 0; i--){
        B[C[A[i]]-1] = A[i];
        C[A[i]] = C[A[i]] - 1;
    }

}

```

Questa implementazione se avviata sull'array: $A = \{2, 6, 1, 7, 8\}$ da il seguente output:

```

Array di base: 2 - 6 - 1 - 7 - 8
Primo For: 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0
Secondo For: 0 - 1 - 1 - 0 - 0 - 0 - 1 - 1
Terzo For: 0 - 1 - 2 - 2 - 2 - 2 - 3 - 4
Quarto For: 1 - 2 - 6 - 7 - 8

```

Complessità

- il primo ciclo impiega $\Theta(k)$
- il secondo $\Theta(n)$
- il terzo $\Theta(n)$

- il quarto $\Theta(k)$

Quindi il tempo totale è $\Theta(n + k)$

Inoltre è **stabile**: ovvero i numeri con lo stesso valore si presentano nell'array di output nello stesso ordine in cui si trovano nell'array di input, questa cosa solitamente è importante solo quando abbiamo dati satellite, ma nel counting è sempre importante perché molte volte viene usato come subroutine per implementare il radix sort.

Radix sort

Premesse

Il radix sort è l'algoritmo di ordinamento usato in principio per ordinare le schede perforate, ma ai giorni nostri viene utilizzato per ordinare in base a più chiavi contemporaneamente (es: anno, mese e giorno)

Algoritmo

Dati:

- n numeri
- ogni numero ha h cifre
- ogni cifra può avere fino a k valori

quello che fa questo algoritmo è ordinare rispetto considerando solo una cifra dei numeri, partendo da quella meno significativa. La sua implementazione tramite pseudocodice è molto semplice ma implica anche l'implementazione di un algoritmo di ordinamento stabile interno per ordinare rispetto ad una cifra (per noi sempre il counting sort)

```
RadixSort(A, n, h)
    for i <= 0 to h = 1 do:
        countingSort(A, n, i)
```

Implementazione

Ricordiamo che le cifre di un numero si estraggo facendo il modulo 10 nel nostro caso la formula per calcolare una cifra di un numero alle i -esima posizione diventa

$$(A[i] \setminus 10^h) \% 10$$

```
int scegliCifra(int numero, int digit){
    return (numero/(int)pow(10,digit))%10;
}

void countingSortConCifraSpecifico(int *A, int n, int cifra) {
    const int k = 9;
    int* B = new int[n];
    int* C = new int[k + 1];

    for(int j = 0; j <= k; j++)
        C[j] = 0;

    for(int j = 0; j < n; j++) {
        C[scegliCifra(A[j], cifra)] = C[scegliCifra(A[j], cifra)] + 1;
    }

    for(int j = 1; j <= k; j++)
        C[j] = C[j] + C[j-1];

    for(int j = n - 1; j >= 0; j--) {
```

```

        int d = scegliCifra(A[j], cifra);
        B[C[d] - 1] = A[j];
        C[d] = C[d] - 1;
    }

    for (int j = 0; j < n; j++) {
        A[j] = B[j];
    }

}

void radixSort(int* A, int len, int h){
    for(int i = 0; i < h; i++){
        countingSortConCifraSpecificata(A, len, i);
    }
}

```

Complessità

La sua complessità dipende dall'algoritmo di ordinamento usato al suo interno, nel nostro caso abbiamo usato il counting sort quindi abbiamo una complessità:

$$O(h(n + k)) \in O(n)$$

Tabelle Hash

Introduzione alle Hash table

Introdurremo il concetto di Tabella Hash partendo da un problema di implementazione: dobbiamo implementare un dizionario. Quale è il modo migliore di farlo?

Cos'è un dizionario?

Un dizionario è un insieme su cui possiamo effettuare operazioni di:

- inserimento
- cancellazione
- ricerca

Questa struttura dati viene usata principalmente per la ricerca. La ricerca di un elemento in un dizionario può portare a due esiti differenti:

- *Ricerca con successo*: ricerca di un elemento presente nella tabella
- *Ricerca senza successo*: ricerca di un elemento non-presente nella tabella.

Possibili implementazioni e complessità con strutture già note

- **Array non-ordinato:**
 - Inserimento: $O(1)$
 - Cancellazione: $O(1)$
 - Ricerca: $O(n)$
- **Array ordinato:**
 - Inserimento: $O(n)$
 - Cancellazione: $O(n)$
 - Ricerca: $O(\log n)$ Ricerca Binaria
- **Lista non-ordinata:**
 - Inserimento: $O(1)$
 - Cancellazione: $O(1)$
 - Ricerca: $O(n)$

- **Lista ordinata:**
 - Inserimento: $O(1)$
 - Cancellazione: $O(1)$
 - Ricerca: $O(n)$
- **BST:**
 - Inserimento: $O(\log n)$
 - Cancellazione: $O(\log n)$
 - Ricerca: $O(\log n)$
- **Tabella a Indirizzamento Diretto:**
 - Inserimento: $O(1)$
 - Cancellazione: $O(1)$
 - Ricerca: $O(1)$

Tabelle a Indirizzamento Diretto

Definizione

Una tabella a indirizzamento diretto è una soluzione molto conveniente, nei casi in cui l'insieme universo U (l'insieme di tutti i numeri registrabili) è un insieme relativamente **piccolo**. Una tabella a indirizzamento diretto avrà uno slot per ogni elemento $x \in U$. Ad ogni elemento è associata una chiave k , ovvero un indice della tabella a indirizzamento diretto.

Implementazione

Inserimento:

```
Insert(T,k)
    T[k] = 1
```

Rimozione:

```
Insert(T,k)
    T[k] = 0
```

Ricerca:

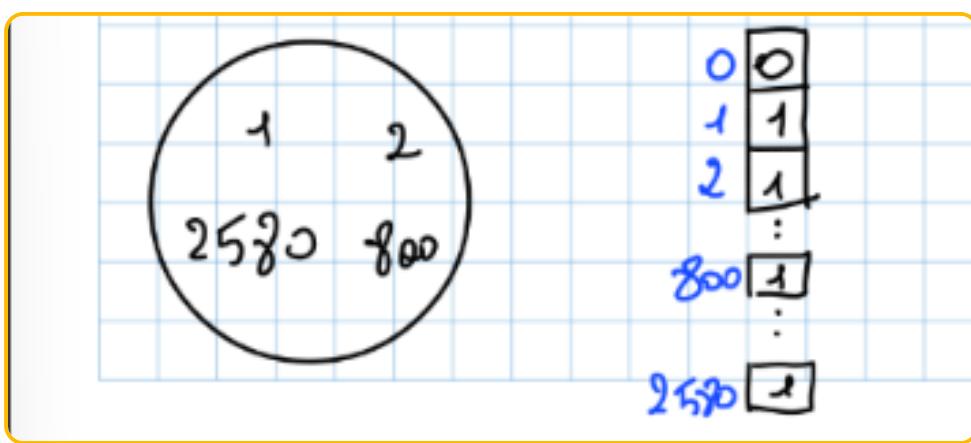
```
Search(T,k)
    IF T[k]=1
        RETURN true
    ELSE RETURN false
```

Hanno tutte complessità $O(1)$

Svantaggi di questa struttura

Apparentemente, le tabelle a indirizzamento diretto sono perfette; tuttavia, esistono dei casi in cui non sono veramente consone, e in cui anzi, non possono essere proprio utilizzate:

- **Insieme U molto grande rispetto a K :** se l'insieme universo U è molto grande, ma l'insieme K delle chiavi da memorizzare è piccolo, la memoria da allocare sarà quasi del tutto sprecata.



- **L'insieme universo è infinito.**

Un calcolatore non ha memoria infinita, e una tabella a indirizzamento diretto ha bisogno di un numero finito di indici.

Per risolvere questi problemi e poter utilizzare le tabelle ad indirizzamento diretto e quindi i dizionari nascono le funzioni di **hash**, una funzione che assegna ad un valore una specifica chiave.

Funzione hashing

Definizione

Il ruolo della funzione hashing (detta h) è quello di associare ad un elemento k dell'insieme $S \subseteq U$, un indice della tabella hash.

$$h : k \rightarrow T[h(k)]$$

In questo modo lo spazio richiesto da una **tabella Hash** (una tabella ad indirizzamento diretto ma che usa l'hashing) sarà ridotto a $\Theta(|K|)$, quindi sarà uguale al numero di chiavi da memorizzare, molto di meno rispetto a $\Theta(|U|)$, riuscendo a conservare i vantaggi di una tabella ad indirizzamento diretto (ovvero la ricerca in $O(1)$)

Collisioni

La funzione $h(k)$, avendo solitamente un codominio di cardinalità inferiore al dominio, non è una funzione biunivoca. Per il *pigeonhole principle*, se gli elementi da inserire nella Hash Table sono più degli indirizzi, almeno due chiavi finiranno allo stesso indirizzo. Quando due chiavi finiscono nella stessa cella, abbiamo un fenomeno chiamato **collisione**, dall'evitare le collisioni nascono tutti modi per implementare una tabella hash.

Implementazioni delle tabelle hash

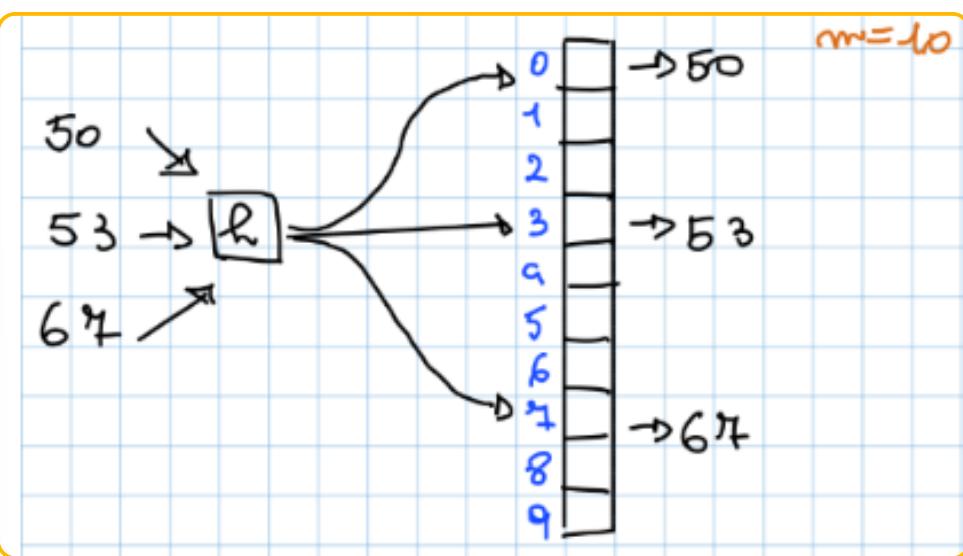
Data una chiave k sono tanti i metodi che possiamo utilizzare per indirizzare, di seguito ne vediamo alcuni.

Metodo della divisione

Dati:

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

$$h(k) = k \bmod m$$



Quando utilizziamo questo metodo evitiamo certi valori di m , per esempio non può essere una potenza di 2, perché se $m = 2^p$ allora $h(k)$ allora la creazione delle chiave si baserà solo sui p meno significativi e non su tutto il valore creando poca uniformità.

L'operazione $k \pmod{16}$ restituisce i 4 bit meno significativi di k .

1. Chiave $k_1 = 37$ (in binario: ...00100101):

$$h(37) = 37 \pmod{16} = 5$$

(In binario, il risultato è 0101)

2. Chiave $k_2 = 53$ (in binario: ...00110101):

$$h(53) = 53 \pmod{16} = 5$$

(In binario, il risultato è 0101)

Le chiavi 37 e 53 hanno lo stesso valore hash perché i loro 4 bit meno significativi sono identici (0101).

Metodo della moltiplicazione

Dati:

- $k \in U$
- $h : U \rightarrow \{0, 1, \dots, m - 1\}$
- e $0 < A < 1$

$$h(k) = \lfloor (k \cdot A \pmod{1})m \rfloor$$

Spieghiamo il principio di questo metodo:

1. $k \cdot A$ ci ritorna un valore compreso tra 0 e k .
2. Il resto della divisione per 1 con un numero reale, ritornerà il valore della parte decimale, ottenendo un valore compreso tra 0 e 1,1 escluso.
3. Moltiplicando per m , otteniamo valori compresi tra 0 e m , m escluso.
4. La funzione *floor* ci permette di prendere solo la parte intera

In questo metodo il valore di m non è critico, solitamente sceglieremo una potenza di 2 perché rende più veloce l'implementazione nei calcolatori moderni

Risolviamo le collisioni usando l'hashing con concatenazione

Definizione

Nell'hashing con concatenazione poniamo tutti gli elementi che sono associati alla stessa cella in una lista concatenata. Praticamente la cella j contiene un puntatore alla testa della lista adi tutti gli elementi memorizzati

che vengono mappati in j ; se non c'è ne sono, la cella j contiene la costante *NIL*

Implementazioni

Inserimento:

```
Insert(T, k)
    List-Insert(T[h(k)], k)
```

Costo: avrà $O(1)$ come l'inserimento in lista.

Rimozione:

```
Insert(T, k)
    List-Delete(T[h(k)], k)
```

Costo: avrà $O(1)$ come la cancellazione dalla lista.

Ricerca:

```
Search(T, k)
    RETURN List-Search(T[h(k)], k)
```

Costo: Il caso peggiore sarà quello in cui ogni elemento è inserito nella stessa posizione della tabella. $O(n)$, come la ricerca in una lista di n elementi. Andiamo a studiare tuttavia il caso medio, introducendo il concetto di uniformità.

Analisi

⚠️ Attention

Data una tavola hash T con m celle dove sono memorizzati n elementi, definiamo **fattore di carico** α della tavola T il rapporto $\frac{n}{m}$, ossia il numero medio di elementi che memorizzati in una lista, la nostra analisi sarà fatta in funzione di α che può essere minore, uguale o maggiore di 1.

Caso peggiore: il comportamento nel caso peggiore dell'hashing con concatenamento è pessimo, tutte le chiavi vengono associate alla stessa cella, quindi stiamo lavorando praticamente con una lista, con tutte le inefficienze del caso ovvero: Ricerca in $\Theta(n)$

Caso medio: il comportamento nel caso medio dipende dal modo in cui la funzione hash distribuisce mediamente l'insieme delle chiavi da memorizzare tra le m celle. Per adesso supponiamo che qualsiasi elemento abbia la stessa probabilità di essere mandato in un qualsiasi posizione. Questa ipotesi si chiama *Hashing uniforme semplice*. Se indichiamo con n_j la lunghezza della lista $T[j]$ per $j = 0, 1, \dots, m - 1$ avremo $n = n_0 + n_1 + \dots + n_{m-1}$ da questo possiamo dire che la lunghezza media delle liste è:

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

per fare un'analisi della complessità dobbiamo necessariamente distinguere due casi:

- **Ricerca senza successo:** una ricerca senza successo richiede un tempo di $\Theta(1 + \alpha)$ nel caso medio

Dimostrazione: Il tempo atteso per ricercare senza successo una chiave k è il tempo atteso per svolgere ricerche fino alla fine della lista $T[h(k)]$ che ha lunghezza attesa di α quindi il tempo totale richiesto (incluso quello per calcolare $h(k)$ che ipotizziamo sia $O(1)$) è

$$\Theta(1 + \alpha)$$

- **Ricerca con successo:** una ricerca con successo richiede un tempo di $\Theta(1 + \alpha)$ nel caso medio
Dimostrazione: il numero di elementi esaminati durante una ricerca con successo di un elemento x è uno in più del numero di elementi che si trovano prima di x nella lista di x . Gli elementi prima di x li troviamo facendo:

$$1 + \sum_{j=i+1}^n Pr(x_{ij})$$

Ricordiamo che $Pr(x_{ij}) = \frac{1}{m}$. Dunque il numero atteso di elementi esaminati con successo è:

$$\frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=n+1}^n Pr\{x_{ij}\} \right)$$

Di seguito la risoluzione:

- Distribuiamo la sommatoria dentro la parentesi e sostituiamo $Pr\{x_{ij}\}$:

$$\frac{1}{n} \left(\sum_{i=1}^n 1 + \sum_{i=1}^n \sum_{j=n+1}^n \frac{1}{m} \right)$$

- Riscrivo la prima sommatoria semplicemente come n e sposto fuori la costante $\frac{1}{m}$:

$$\frac{1}{n} \left(n + \frac{1}{m} \sum_{i=1}^n \sum_{j=n+1}^n 1 \right)$$

- Riscrivo la sommatoria con j come $n - i$:

$$\frac{1}{n} \left(n + \frac{1}{m} \sum_{i=1}^n (n - i) \right)$$

- Riscrivo le sommatorie come differenza di sommatorie

$$\frac{1}{n} \left(n + \frac{1}{m} \left(\sum_{i=1}^n n - \sum_{j=n+1}^n i \right) \right)$$

- Risolvo le sommatorie

$$\frac{1}{n} \left(n + \frac{1}{m} \left(n^2 - \frac{n(n+1)}{2} \right) \right)$$

- Risolvo i calcoli rimanenti

$$1 + \frac{1}{mn} \left(n^2 - \frac{n(n+1)}{2} \right) = 1 + \frac{n}{m} - \frac{n+1}{2m} \in O(1 + \alpha)$$

Risolviamo le collisioni usando l'hashing a indirizzamento aperto

Definizione

Un altro metodo che permette di risolvere le collisioni, è l'**indirizzamento aperto**. Supponiamo di avere una tabella con un numero m di slot. Se cercando di effettuare un inserimento nella tabella, la posizione calcolata dalla funzione di hashing risulta già occupata, verrà calcolato e ispezionato un altro slot. Ciò avverrà fino a quando non si troverà uno slot libero (contenente **NULL** o **D**, ne parleremo dopo), o fino a quando non si saranno controllati tutti gli m slot della tabella.

Sequenza di ispezione

La funzione di hashing all'interno di una tabella con indirizzamento aperto al posto di ritornare una singola posizione, sarà così definita:

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

Definiamo **sequenza di ispezione** di una chiave k , la sequenza di indici che viene generata dalla funzione di hashing calcolata su k , ovvero:

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

La sequenza di ispezione conterrà tutti gli slot della tabella e rappresenta tutte le possibili posizioni che k potrebbe prendere se non sono occupate.

Implementazioni funzioni di base

Inserimento

```
hash_insert(T, k)
    IF n = m
        RETURN // tabella piena
    i = 0
    WHILE (T[h(k, i)] != NULL AND T[h(k, i)] != D): //D spiegato in "Funzione di cancellazione"
        i = i + 1
    T[h(k, i)] = k
    n = n + 1
```

Ricerca

```
hash_search(T, k)
    i = 0
    WHILE T[h(k, i)] != NULL AND i < m:
        IF T[h(k, i)] = k
            RETURN True
        i++
    RETURN False
```

Funzione di cancellazione

La funzione di cancellazione, all'interno di una tabella hash che risolve le collisioni tramite l'indirizzamento aperto, richiederà una modifica fondamentale per il funzionamento corretto della funzione di ricerca. Cancellato un elemento dalla tabella, esso non dovrà essere sostituito da **NULL**. Esso dovrà infatti essere contrassegnato come **D** di **DELETED**. L'esempio di seguito aiuta a capire perché usiamo **D**.

Esempio:

- Prendiamo una tabella hash con $m = 4$ contiene i numeri [1, 2, 3, 5].
- Eliminiamo il numero 3 dalla tabella, ottenendo [1, 2, NULL, 5].
- Calcoliamo la sequenza di ispezione della chiave 5 che sarebbe:
 - $h(5, 0) = 1$
 - $h(5, 1) = 0$
 - $h(5, 2) = 2$
 - $h(5, 3) = 3$

Supponiamo adesso di usare la funzione di ricerca per verificare la presenza (evidente) del numero 5

La ricerca si interromperà a $h(5, 2) = 2$. Ciò non sarebbe avvenuto prima dell'eliminazione del numero 3, ed è causato proprio dal NULL in posizione 2.

Contrassegnare lo slot con **DELETED** espliciterà il fatto che lo slot è libero in caso di inserimento, ma anche che la ricerca non andrà interrotta nonostante lo slot vuoto.

```
hash_delete(T, k)
    i <- 0
    WHILE (T[h(k, i)] != NULL AND i < m):
        IF T[h(k, i)] = k:
            T[h(k, i)] = DELETED
        RETURN True
```

```

i = i + 1
RETURN False

```

Essendo l'eliminazione abbastanza macchinosa usiamo l'indirizzamento aperto solo quando non dobbiamo eliminare elementi.

Requisiti di una funzioni hash per indirizzamento aperto

Dati:

- P_m è l'insieme di tutte le permutazioni sull'insieme $\{0, \dots, m-1\}$.
 - La cardinalità di P_m è $m!$.
- Sia $p \in P_m$ una permutazione. La probabilità che $h(k) = p$ deve essere la stessa per qualsiasi permutazione.

$$Pr[h(k) = p] = \frac{1}{m!} \forall p \in P_m$$

Questa ipotesi si chiama **hashing uniforme**. Costruire funzioni hash per l'indirizzamento aperto che rispettino questa proprietà è piuttosto complesso. Successivamente vengono analizzate tre tecniche usate per creare la sequenza di ispezione

Analisi

La nostra analisi dell'indirizzamento aperto è espressa in termini del fattore di carico $\alpha = \frac{n}{m}$. Supponiamo venga applicata l'hashing uniforme quindi la sequenza

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

utilizzata per inserire o ricercare una chiave k ha la stessa probabilità di essere una qualsiasi permutazione di $\{0, 1, \dots, m-1\}$. Analizziamo il numero atteso di ispezioni dell'hashing con indirizzamento aperto, facendo differenza tra una ricerca con e senza successo:

Ricerca senza successo: data una tavola hash con un fattore di carico $\alpha = n/m < 1$ il numero atteso di ispezioni in una ricerca senza successo è al massimo $\frac{1}{1-\alpha}$

Dimostrazione:

$x = \# \text{ ispezioni potute durante un inserimento}$

$$E[x] = \sum_{i=1}^{\infty} P_x \{x \geq i\} \quad h: U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$$

$A_i = \text{tavola dopo delle i-esime isezioni}$

$$P_x \{x \geq i\} = P_x \{A_1 \cap A_2 \cap A_3 \cap \dots \cap A_{i-1}\}$$

Probabilità condizionata

$$P_x \{A_1\} \times P_x \{A_2 | A_1\} \times P_x \{A_3 | A_1 \cap A_2\} \times P_x \{A_4 | A_1 \cap A_2 \cap A_3\} \times \dots \times P_x \{A_{i-1} | A_1 \cap A_2 \cap \dots \cap A_{i-2}\}$$

$$\frac{m}{m} \geq \frac{m-1}{m-1} \geq \frac{m-2}{m-2} \geq \frac{m-3}{m-3} \geq \dots \geq \frac{m-i+2}{m-i+2}$$

$$\leq \left(\frac{m}{m}\right)^{i-1}$$

$$\leq \sum_{i=1}^{\infty} \left(\frac{m}{m}\right)^{i-1} \rightarrow \sum_{i=0}^{\infty} d^i = \frac{1}{1-d}$$

\downarrow
serie geometrica

Ricerca con successo: data una tavola hash con un fattore di carico $\alpha = n/m < 1$ il numero atteso di ispezioni in una ricerca senza successo è al massimo $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

Dimostrazione:

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

$$d_i = \frac{i}{m}$$

$$k_1, k_2, k_3, \dots, k_i, k_{i+1}, \dots, k_m$$

lavoro per inserimento

$$k_{(i+1)} \rightarrow \frac{1}{1-d_i} = \frac{1}{1-\frac{i}{m}} = \frac{1}{\frac{m}{m}-\frac{i}{m}} = \frac{m}{m-i}$$

$$\frac{1}{m} \sum_{i=0}^{m-1} \frac{m}{m-i} = \frac{m}{m} \sum_{i=0}^{m-1} \frac{1}{m-i} \rightarrow \frac{m}{m} \sum_{k=m-m+1}^m \frac{1}{k}$$

$$\frac{1}{m} + \frac{1}{m-1} + \frac{1}{m-2} + \frac{1}{m-3} + \dots + \frac{1}{m-m+1} = \sum_{k=m-m+1}^m \frac{1}{k}$$

$$\leq \frac{m}{m} \int_{m-m}^m \frac{1}{x} dx = \frac{m}{m} [\ln(m) - \ln(m-m)]$$

$$\leq \frac{m}{m} \ln\left(\frac{m}{m-m}\right) = \frac{1}{2} \ln\left(\frac{1}{1-\alpha}\right)$$

Tecniche per creare una sequenza di ispezione

Ispezione lineare

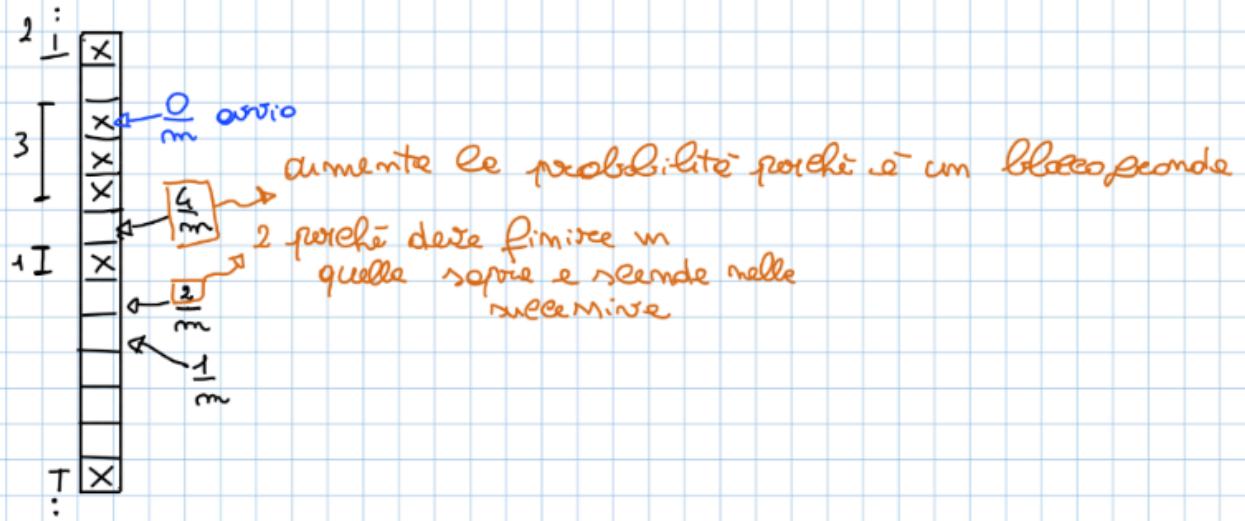
La funzione di hashing per la scansione lineare è definita in questo modo

$$h(k, i) = (h'(k) + i) \mod m$$

Con $h' : U \rightarrow \{0, 1, \dots, m-1\}$. Le sequenze della nostra funzione di hashing saranno del tipo

$$\langle h'(k), h'(k) + 1 \mod m, h'(k) + 2 \mod m, \dots, h'(k) + (m-1) \mod m \rangle$$

Questa funzione di hashing non gode della proprietà di hashing uniforme, in quanto, le permutazioni ottenute non possono non iniziare con $h'(k)$. Moltissime permutazioni avranno probabilità 0, e le restanti $\frac{1}{m}$. Un'altro problema di questo tipo di ispezione è l'**agglomerazione primaria**. In breve, la probabilità che le chiavi siano inserite in slot successivi aumenta ad ogni inserimento, favorendo agglomerati di chiavi e allungando le tempistiche relative alle operazioni.



Ispezione quadratica

Cambia solo la funzione di hashing e diventa:

$$h(k, i) = (h'(k) + c_1 + c_2 i^2) \bmod m$$

Con c_1 e c_2 costanti scelte in modo tale che l'intera tabella venga scansionata dalla funzione $h(k, i)$ ed evitare l'agglomerazione primaria.

Ispezione doppio hashing

In questa funzione vengono usate due funzioni di hashing:

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

Il numero totale di permutazioni totale è $m \times m = m^2$, in quanto la prima funzione di hashing da la prima posizione, la seconda le successive, e quindi m possibili posizioni dopo altre m .

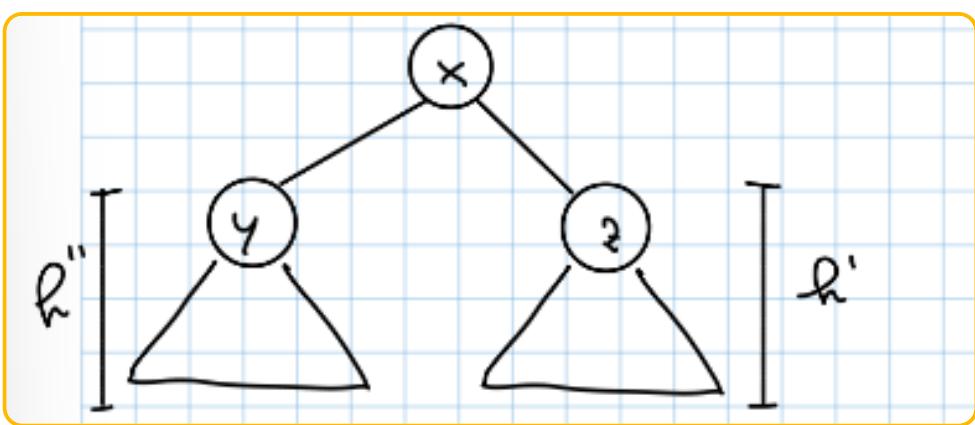
Alberi rosso-neri

Alberi bilanciati

Lavorando con gli alberi ci capita di incappare in alberi sbilanciati, nel peggior dei casi questi sono praticamente delle liste, dalla necessità di rendere l'albero autobilanciante nascono gli alberi rosso-neri

Definizione di albero bilanciato

Definiamo albero bilanciato quell'albero che ha entrambi i sottoalberi con una quantità di lavoro asintoticamente uguale. Dato questo albero:



possiamo dire che:

$$|h' - h''| \leq 2$$

Ripasso proprietà del BST

Un albero binario di ricerca è costituito esclusivamente da nodi che rispettano la seguente proprietà:

- ogni nodo è maggiore o uguale a tutti i nodi del proprio sottoalbero sinistro
- ogni nodo è minore o uguale a tutti i nodi del proprio sottoalbero destro

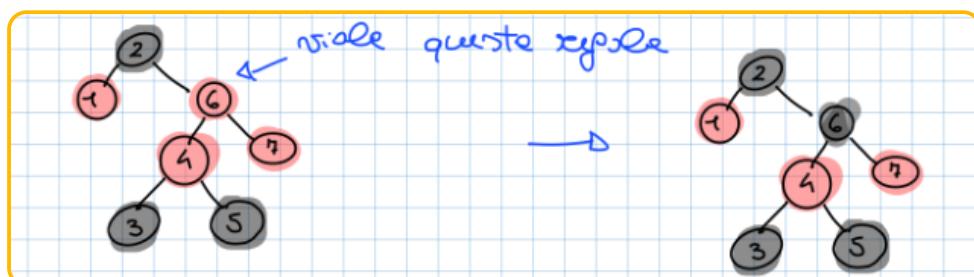
Algoritmo di inserimento: inizia confrontando il valore da inserire con la radice, vado a sinistra o a destra in base alle proprietà di prima, ripeto questo passaggio fino a quando non trovo un posto libero.

Proprietà di una albero rosso-nero

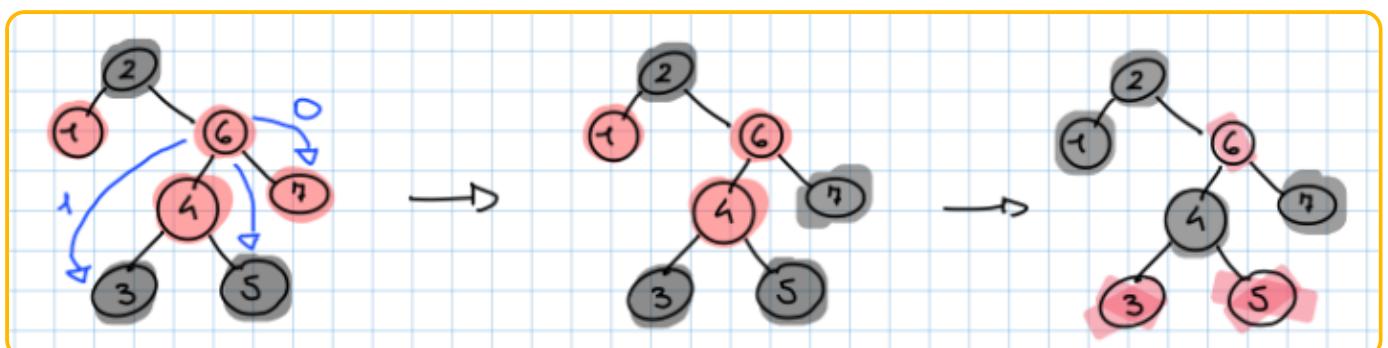
Proprietà di un albero rosso-nero

Un albero rosso-nero è *bilanciato* questo è possibile grazie a queste regole:

1. Ogni nodo è *rosso o nero*
2. La radice è *nera*
3. le foglie sono *nere*
 - Questa proprietà può anche non essere rispettata se consideriamo foglie gli ultimi nodi dell'albero quelli NIL (l'altezza aumenta di 1 e la dimensione diventa $2n$)
4. I figli di un nodo *rosso* sono *neri*



5. Per ogni nodo, tutti i cammini che vanno dal nodo alle foglie discendenti contengono lo stesso numero di nodi neri



Ogni nodo dell'albero è formato da:

- *color*: il colore del nodo
- *key*: valore del nodo
- *left*: puntatore al figlio sinistro
- *right*: puntatore al figlio destro
- *p*: puntatore al padre

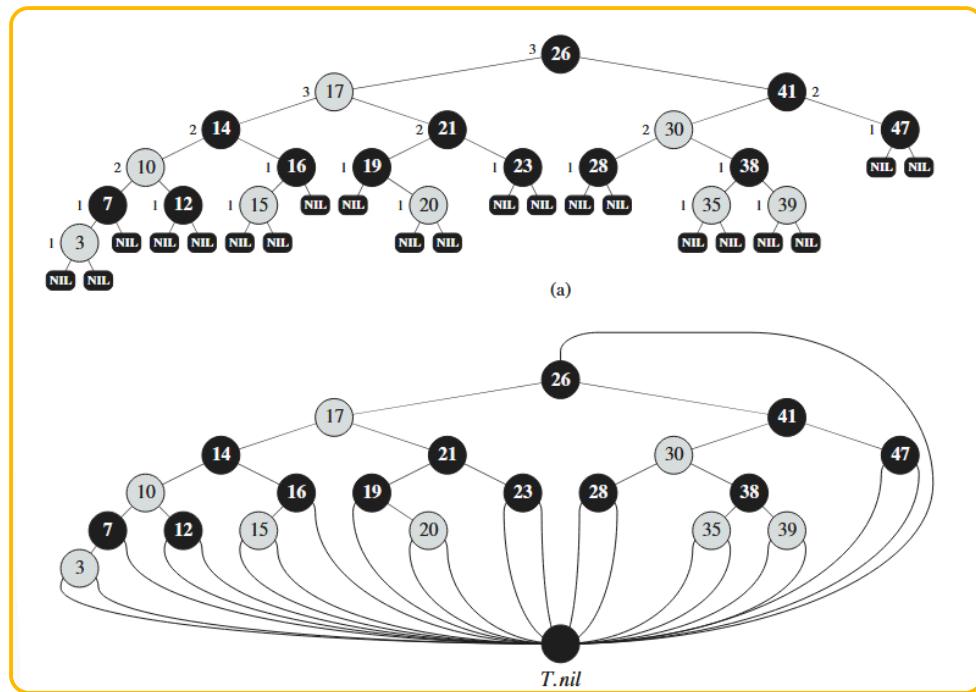
Se manca un figlio o un padre di un nodo il suo valore sarà *NIL*, i nodi NIL sono considerati come nodi neri

Esistono diversi modi per colorare un albero, basta che le proprietà vengano rispettate

Rappresentazione di un albero rosso-nero

La rappresentazione di un albero rosso nero viene fatta in questo modo:

1. Si decide di creare un nodo NIL ogni volta che è necessario
2. Tutti i NIL si fanno corrispondere ad un singolo nodo detto *sentinella NIL*



Il secondo approccio rispetto al primo risparmia della memoria ma è graficamente meno intuitivo.

Definizione altezza nera lemma altezza massima

Definizione altezza nera: definiamo *altezza nera* di un nodo x , indicata con $bh(x)$ il numero di nodi neri lungo un cammino semplice che inizia dal nodo x (ma non lo include) e finisce in un foglia. L'altezza nera di un albero è $hb(root)$.

Definizione altezza massima: l'altezza massima di un albero rosso-nero con n nodi interni è $2 \log_2(n + 1)$

Dimostrazione: Iniziamo dimostrando che il sottoalbero con radice in un nodo x qualsiasi contiene almeno $2^{bh(x)} - 1$ nodi interni, lo faremo per induzione:

- *caso base*: Se l'altezza di x è 0 allora x deve essere una foglia e il sottoalbero con radice in x contiene: $2^0 - 1 = 1 - 1 = 0$
 - *passo induttivo*: consideriamo un nodo x che ha un altezza positiva ed è quindi un nodo interno con due figli. Possiamo dire che ogni figlio ha un altezza nera pari a:
 - se rosso ha $bh(x)$
 - se nero ha $bh(x) - 1$ (perché escludo il nodo stesso dal conteggio)
- Poiché l'altezza di un figlio di x è minore dell'altezza di x possiamo applicare l'ipotesi induttiva per

concludere che ogni figlio ha almeno

$$2^{bh(x)-1} - 1 \text{ nodi interni}$$

e quindi possiamo concludere che il sottoalbero con radice in x contiene:

$$\underbrace{(2^{bh(x)-1} - 1)}_{\text{nodi interni albero sx}} + \underbrace{(2^{bh(x)-1} - 1)}_{\text{nodi interni albero dx}} + 1 = 2^{bh(x)} - 1 \text{ nodi interni}$$

il che dimostra l'asserzione iniziale.

Per completare la dimostrazione indichiamo con h l'altezza dell'albero, sappiamo che almeno metà dei nodi in qualsiasi cammino semplice della radice ad una foglia deve essere nera, di conseguenza l'altezza nera della radice è $h/2$ (perché lungo il cammino i nodi si alternano rosso/hero) quindi abbiamo che:

$$n \geq 2^{h/2} - 1$$

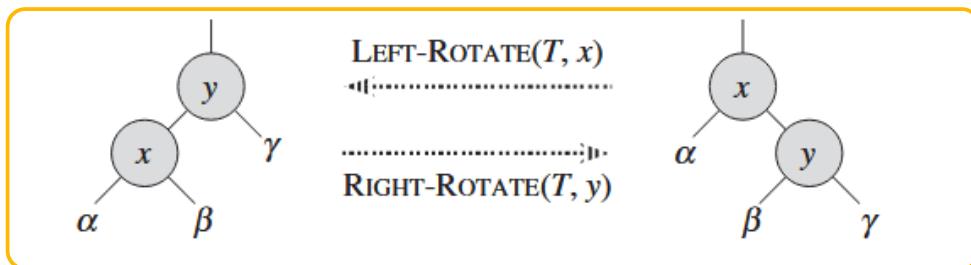
dove n è il numero di nodi interni del nostro albero. Spostando 1 nel lato sinistro e prendendo i logaritmi di entrambi i lati otteniamo:

$$\log_2(n+1) \geq h/2 \text{ ovvero } h \leq 2 \log_2(n+1)$$

Operazioni negli alberi rosso-neri: rotazioni

Definizione

Le operazioni di inserimento e eliminazione di un nodo modificano l'albero, questo di conseguenza potrebbe violare le proprietà discusse prima, per rispristinarle facciamo uso delle rotazioni che ci permettono attraverso la modifica dei colori e della struttura dei puntatori di sistemare l'albero.



Rotazione a sinistra: la rotazione a sinistra si basa sul collegamento tra x e y :

- il nodo y diventa la nuova radice
- x come figlio sinistro di y
- il figlio sinistro di y diventa il figlio destro di x

```
LEFT-ROTATE(T,x)
y = x.right // Imposta y
x.right = y.left // Sposta il sottoalbero sx di y nel sottoalbero dx di x
if y.left != T.NIL
    y.left.p = x
y.p = x.p // Collega il padre di x a y

if x.p == T.NIL
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
    y.left = x // Pone x a sinistra di y
    x.p = y
```

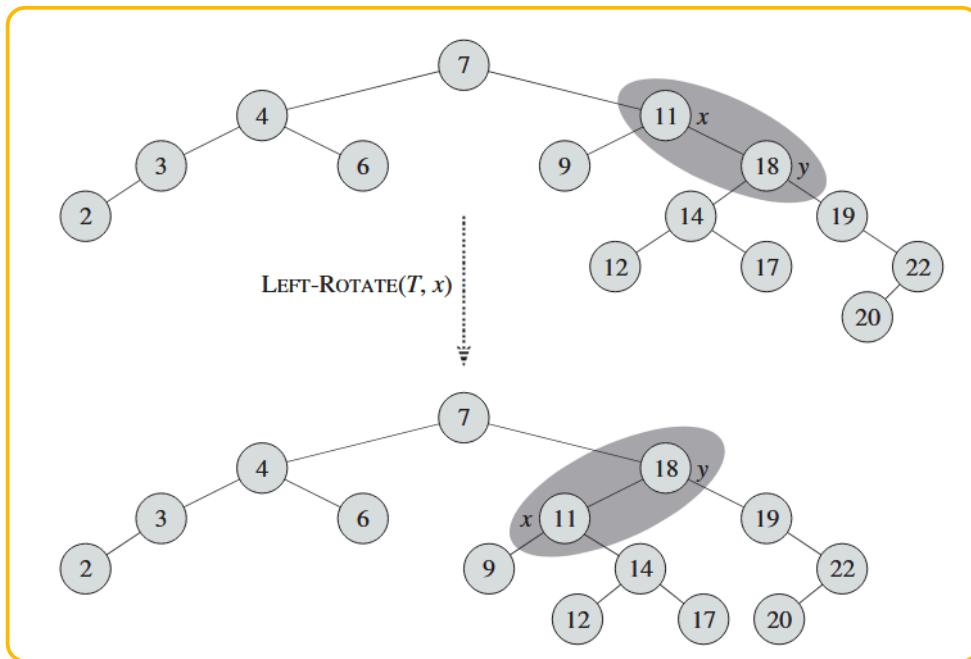
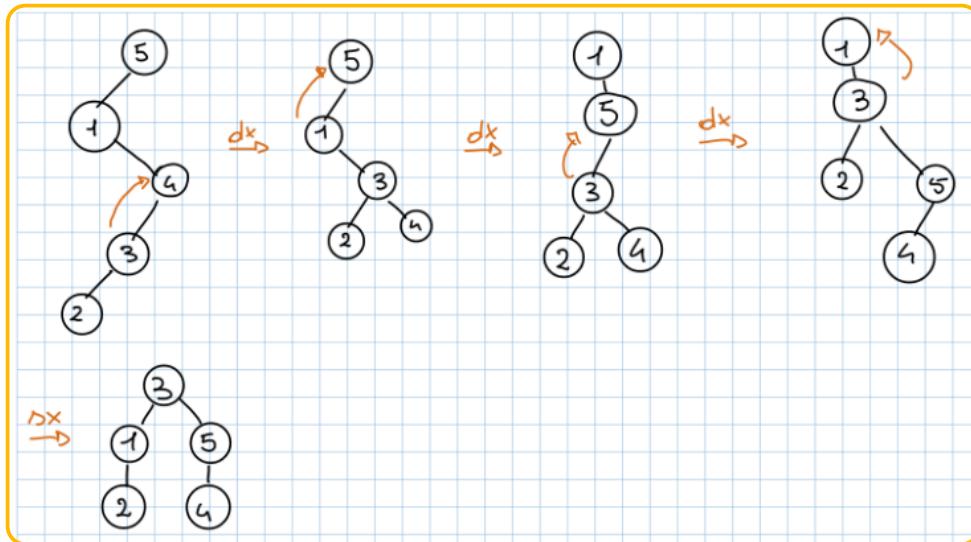
Rotazione a destra: la rotazione a destra si basa sul collegamento tra x e il suo figlio sinistro:

- x diventa la nuova radice
- y come figlio destro di x
- il figlio destro di x diventa il figlio sinistro di y

Tip

Quando applichiamo una rotazione su un albero i nodi che si invertono di posizione dovranno scambiare la propria colorazione

Esempi



Operazioni negli alberi rosso-neri: inserimento

Definizione

Non esiste un modo per implementare l'inserimento che data una qualsiasi configurazione di un albero rosso-nero riesca ad inserire un nuovo nodo senza violare le proprietà fondamentali. Avremo quindi:

1. Creazione del nuovo nodo rosso
2. Inserimento analogo ai BST
3. Chiamata a funzione RB-Insert-Fixup sul nuovo nodo

```

RB-INSERT(T, z)
    y = T.nil           // Inizializza y a NIL
    x = T.root          // Inizia la ricerca dalla radice dell'albero T

    while x ≠ T.nil    // Scendi nell'albero finché non trovi una foglia (nil)
        y = x
        if z.key < x.key // Se la chiave da inserire è minore della corrente...
            x = x.left   // ...vai a sinistra
        else
            x = x.right  // ...vai a destra

        z.p = y           // Imposta y come padre del nuovo nodo z
        if y == T.nil     // Se l'albero era vuoto...
            T.root = z   // ...il nuovo nodo diventa la radice
        elseif z.key < y.key // Altrimenti, se z è minore del padre...
            y.left = z   // ...diventa il figlio sinistro
        else
            y.right = z  // ...diventa il figlio destro

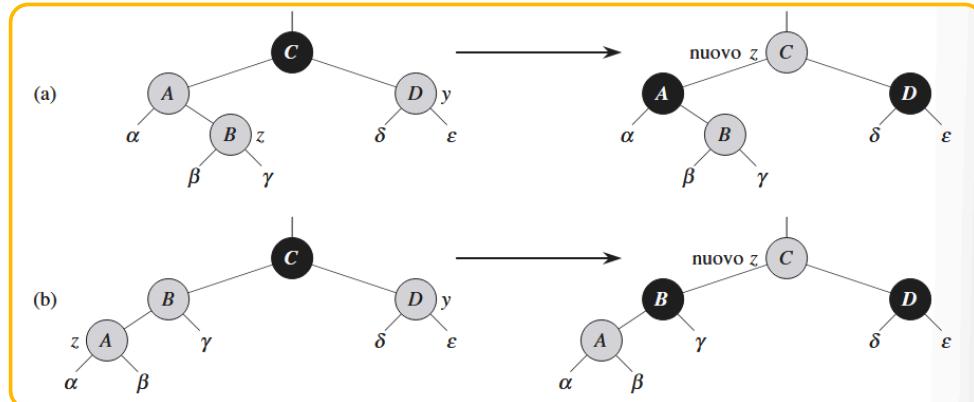
        z.left = T.nil    // Inizializza il figlio sinistro a NIL
        z.right = T.nil   // Inizializza il figlio destro a NIL
        z.color = RED    // Colora il nuovo nodo di ROSSO
    RB-INSERT-FIXUP(T, z) // Ripristina le proprietà dell'albero Red-Black

```

l'inserimento è praticamente identico a quello di un BST normale, abbiamo solo che il nodo viene colorato di rosso e che viene chiamata insert-fixup che si occupa di sistemare e rendere valido il nostro albero come rosso-nero

Definizione di insert-fixup

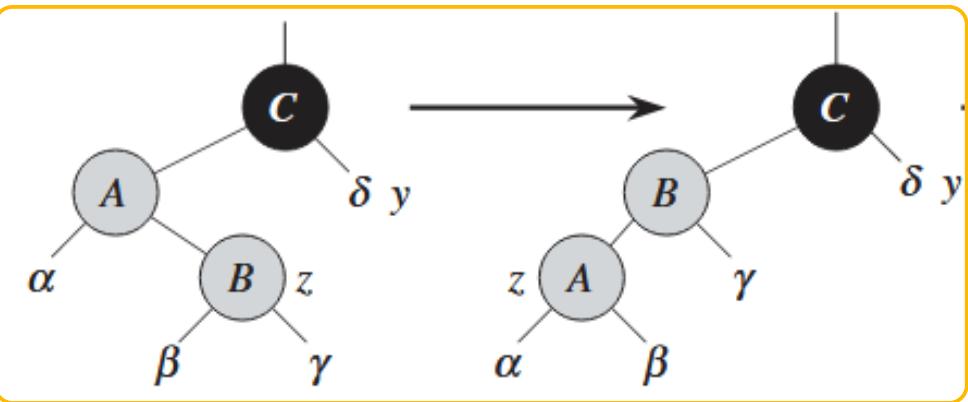
Questa funzione si basa sulla definizione di 3 casi, supponendo che **z** sia il nuovo nodo rosso inserito abbiamo:
Caso 1: lo zio *y* di *z* è rosso



La colorazione del nonno di *z* viene messa ad entrambi i figli, e richiamiamo insert-fixup sul nonno, per correggere possibili violazioni nel resto dell'albero

Caso 2: questo caso è definito su due condizioni

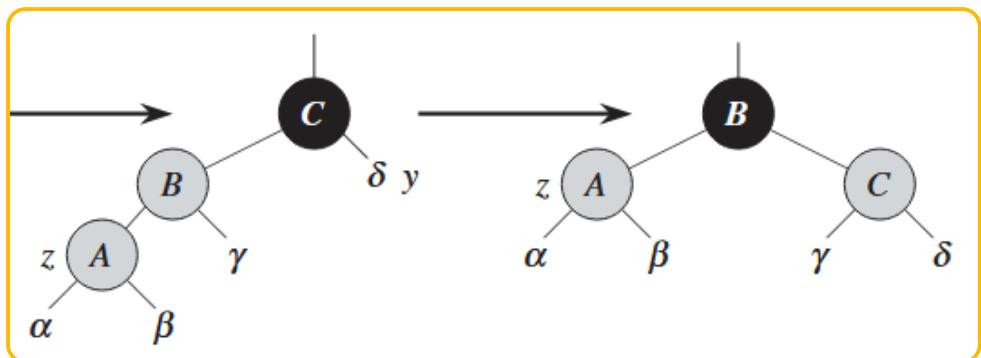
- lo zio *y* di *z* è nero
- *z* è interno



Ruotiamo il padre di z in modo tale da mettere z nella posizione del padre, ci saremo ricondotti al caso 3 e richiamiamo la procedura sulla nuova z

Caso 3: questo caso è definito su due condizioni

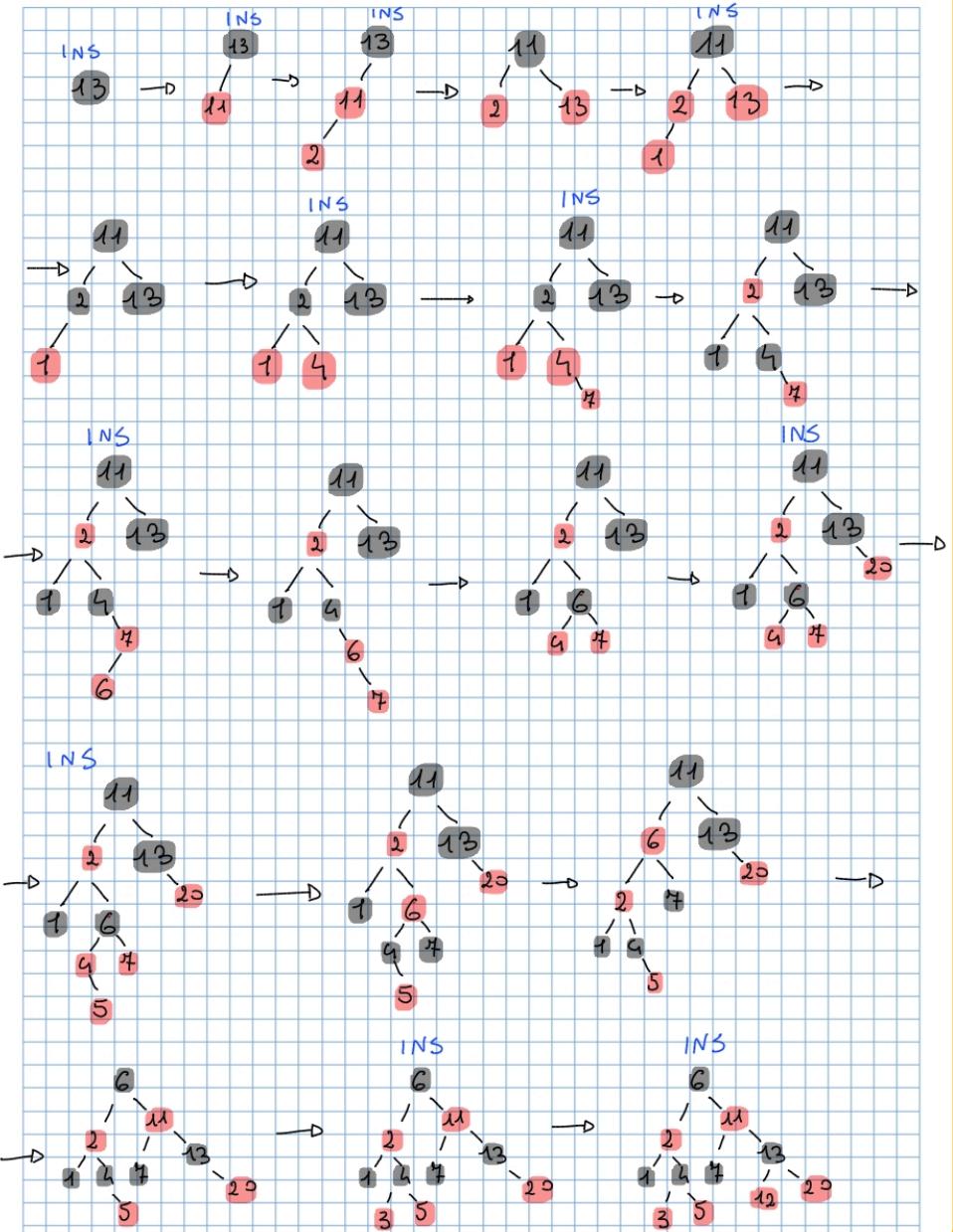
- lo zio y di z è nero
- z è esterno



Ruotiamo il padre di z con il nonno in maniera tale da metterlo al posto del nonno

Esempio

INS = promozione di inserimento



Operazioni negli alberi rosso-neri: eliminazione

Ripasso rimozione negli alberi BST

1. *cancellazione foglia*: caso banale, cancello il nodo.
2. *cancello nodo con un solo figlio*: l'unico figlio prende il posto del padre
3. *cancello nodo con due figli*: cancello la chiave del nodo che voglio eliminare, la sostituisco alla chiave del nodo più a sinistra del sottoalbero destro, elimino il nodo più a sinistra del sottoalbero destro

Definizione

Indicando con z il nodo che vogliamo rimuovere, definiamo i vari casi di rimozione:

- **Caso A:** z foglia rossa
 - elimino z
- **Caso B:** z con un solo figlio e il padre nero
 - il figlio di z diventa figlio del padre di z
 - elimino z
- **Caso C:** z nero con figlio rosso
 - il figlio prende il posto di z e diventa nero
 - elimino z
- **Caso D:** z foglia nera

- elimino z
- denoto il NIL come *doppio nero*
- **Caso E:** z nero con figlio nero
 - il figlio prende il posto di z e diventa *doppio nero*
 - elimino z

Gli due casi creano delle configurazioni anomale, quindi sarà necessario creare una funzione **delete-fixup**

Definizione delete-fixup

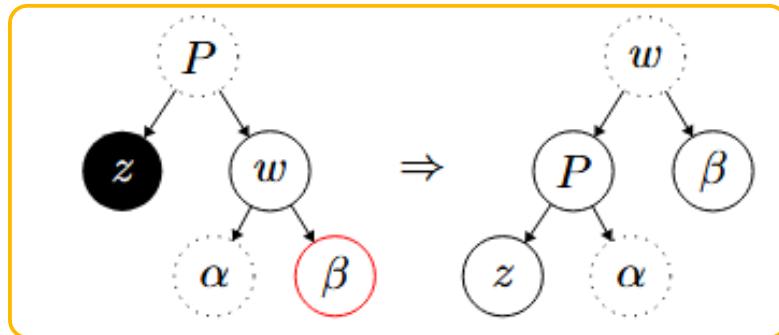
Questa funzione va a risolvere i casi problematici dell'eliminazione, definiamo 3 casi:

Premesse:

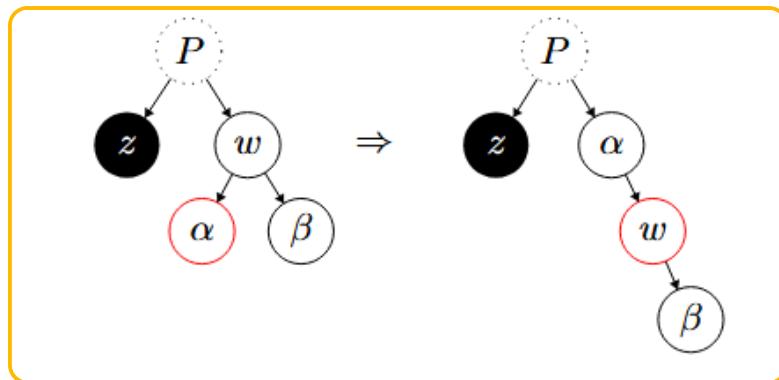
- z è nodo doppio nero
- w è il fratello di z

Caso 1: w è nero e ha almeno un figlio rosso

- *a:* figlio esterno rosso
 1. Ruoto w con il padre e lo faccio salire
 2. il doppio nero diventa nero
 3. β diventa nero

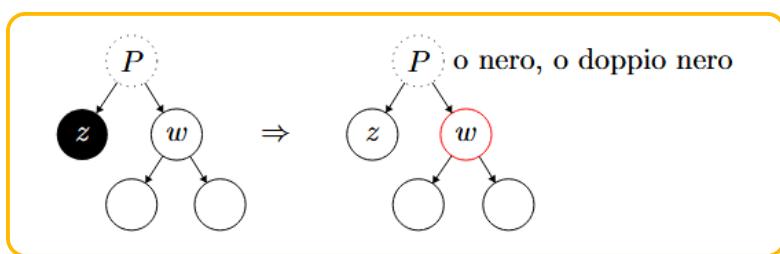


- *b:* figlio esterno nero, ma interno rosso
 1. Ruoto il figlio rosso con il padre w facendolo diventare padre
 2. w diventerà figlio esterno rosso
 3. ci siamo ricondotti al caso 1-a



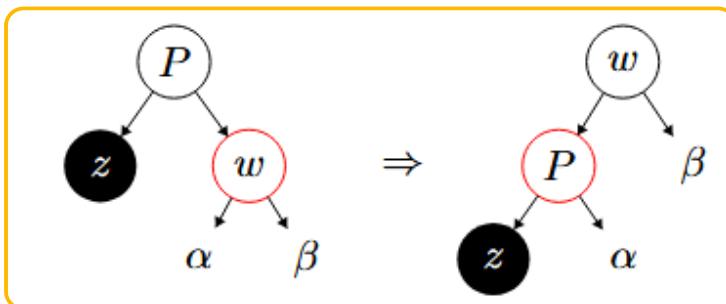
Caso 2: w nero con entrambi i figli neri

1. propaga la colorazione di w e di z al padre
 - *a:* se il padre era rosso diventa nero e ho finito
 - *b:* se il padre era nero diventa doppio nero, richiamo delete-fixup sul padre del doppio nero



Caso 3: w è rosso

2. ruoto w con il padre facendo risalire w
3. richiamo fixup sul nodo nero



TIP Generali:

- Se un nodo doppio-nero diventa la radice dell'interno albero possiamo scartare un grado di nero
- Se dobbiamo togliere un grado di nero ad un nodo NIL questo si può fare ma resterà comunque nero

Programmazione dinamica

Introduzione

Definizione

la programmazione dinamica si applica ai *problemi di ottimizzazione*, questi problemi hanno molteplici soluzioni possibili, per capire quale soluzione è la migliore definiamo la funzione *bontà* che assegna ad ogni soluzione un grado di bontà.

Sviluppare un algoritmo di programmazione dinamica

Lo sviluppo di un algoritmo di programmazione dinamica si può essere suddiviso in queste fasi:

1. Caratterizzazione di una sotto-struttura ottima
2. Definire in modo ricorsivo il valore di una soluzione ottima
3. Costruire una procedura bottom-up per il calcolo di una soluzione ottima
4. Costruire una soluzione ottima dalle informazioni calcolate

Le fasi dalla 1 alla 3 formano la base per risolvere un problema applicando la programmazione dinamica. La fase 4 può essere omessa se è richiesto soltanto il valore di una soluzione ottima. Infine ricordiamo che:

- Approcciare un problema in maniera *top-down* vuol dire risolverlo in maniera *ricorsiva*
- Approcciare un problema in maniera *bottom-up* vuol dire risolverlo in maniera *iterativa*

Fibonacci

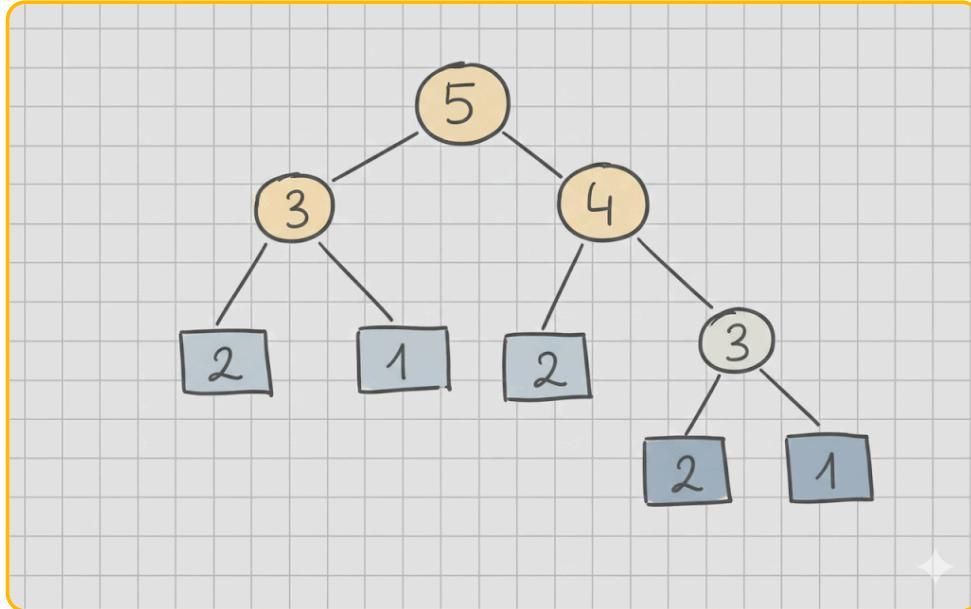
Definizione

La sequenza di Fibonacci è definita dalla seguente equazione:

$$F_n = \begin{cases} 1 & \text{se } n \leq 2 \\ F_{n-1} + F_{n-2} & \text{se } n > 2 \end{cases}$$

L'implementazione diretta che ne facciamo è:

```
F(n):
    if n <= 2 then
        return 1
    else
        return F(n-2) + F(n-1)
```



Guardando l'albero di ricorsione si nota subito che alcuni sotto-problemi vengono risolti più volte, questo è ovviamente un problema che ci fa perdere tempo. Quello che facciamo è rendere un caso base ogni sotto-problema risolto

```
// Inizializzazione
M = new Array(n) // array che contiene la soluzione
M[1] = M[2] = 1
for i = 3 to n DO M[i] <- NULL // inizializzazione

F(n)
if M[n] != null then
    return M[n]

if M[n-1] == null then
    M[n-1] <- F(n-1)

if M[n-2] == null then
    M[n-2] <- F(n-2)

return M[n-1] + M[n-2]
```

Entrambe le soluzioni sono *top-down*, ma la seconda risulta più efficiente. Di seguito una soluzione *bottom-up*:

```
F(n):
    M[1] <- M[2] <- 1
    for i <- 3 to n do
        M[i] <- M[i-2] + M[i-1]
    return M[n]
```

Analisi

- L'approccio ricorsivo puro non ha senso se un sotto-problema deve essere risolto più volte per arrivare alla soluzione, in quel caso ha più senso fare uso della *memorizzazione*.
- Se lo spazio delle soluzioni viene esplorato tutto per trovare la soluzione allora ha più senso un approccio iterativo (*bottom-up*) come nel caso della sequenza di Fibonacci

Rod-cutting

Definizione

Il problema del taglio delle aste può essere definito nel seguente modo:

- Data un'asta di lunghezza n pollici
- Una tabella di prezzi p_i con $i = 1, 2, \dots, n$
- Bisogna determinare il ricavo massimo r_n che si può ottenere tagliando l'asta e vendendone i pezzi.
- Un'asta di lunghezza n può essere tagliata in 2^{n-1} modi differenti

Denotiamo una decomposizione in pezzi utilizzando la normale *notazione additiva* ovvero $7 = 2 + 2 + 3$ indica che un asta di lunghezza 7 viene tagliata in tre pezzi.

Se una soluzione ottima prevede il taglio dell'asta in k pezzi, per $1 \leq k \leq n$ allora la decomposizione:

$$n = i_1 + i_2 + \dots + i_k$$

dell'asta in pezzi di lunghezza i_1, i_2, \dots, i_k fornisce il ricavo massimo ovvero:

$$r_n = p_{i_1} + p_{i_2} + \dots + p_{i_k}$$

Esempio: di seguito mostriamo un esempio di ricavi massimi per ogni possibile lunghezza della rod

lunghezza i	1	2	3	4	5	6	7	8	9	10
prezzo p_i	1	5	8	9	10	17	17	20	24	30

Figura 15.1 Una tabella di prezzi campione delle aste. Un'asta di lunghezza i pollici viene venduta al prezzo di p_i dollari.

- | | |
|---------------|---|
| $r_1 = 1$ | dalla soluzione $1 = 1$ (nessun taglio) |
| $r_2 = 5$ | dalla soluzione $2 = 2$ (nessun taglio) |
| $r_3 = 8$ | dalla soluzione $3 = 3$ (nessun taglio) |
| $r_4 = 10$ | dalla soluzione $4 = 2 + 2$ |
| $r_5 = 13$ | dalla soluzione $5 = 2 + 3$ |
| $r_6 = 17$ | dalla soluzione $6 = 6$ (nessun taglio) |
| $r_7 = 18$ | dalla soluzione $7 = 1 + 6$ o $7 = 2 + 2 + 3$ |
| $r_8 = 22$ | dalla soluzione $8 = 2 + 6$ |
| $r_9 = 25$ | dalla soluzione $9 = 3 + 6$ |
| $r_{10} = 30$ | dalla soluzione $10 = 10$ (nessun taglio) |

Generalizzando

Generalizzando possiamo esprimere i valori di r_n in funzione dei ricavi delle aste più corte ovvero:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

dove:

- p_n corrisponde alla vendita dell'asta di lunghezza n senza tagli
- Gli altri $n - 1$ argomenti corrispondono al ricavo massimo ottenuto facendo un taglio iniziale in due pezzi di dimensione i e $n - i$ (con $i = 1, 2, \dots, n - 1$) e poi tagliando in modo ottimale gli ulteriori pezzi ottenendo i

ricavi r_i e r_{n-i}

Poiché non sappiamo a priori quale valore di i ottimizza i ricavi *dobbiamo considerare tutti i possibili valori di i* e selezionare quello che massimizza i ricavi. Per risolvere il problema originale stiamo risolvendo i problemi più piccoli dello stesso tipo questo lo possiamo fare perché il problema del taglio delle asta presenta una *sottostruttura ottima* ovvero le soluzioni ottime di un problema incorporano le soluzioni ottime dei sotto-problemi correlati.

Se vediamo ciascuna decomposizione di un'asta come un primo pezzo seguito da un'eventuale decomposizione del pezzo restante, possiamo *riformulare r_n come*:

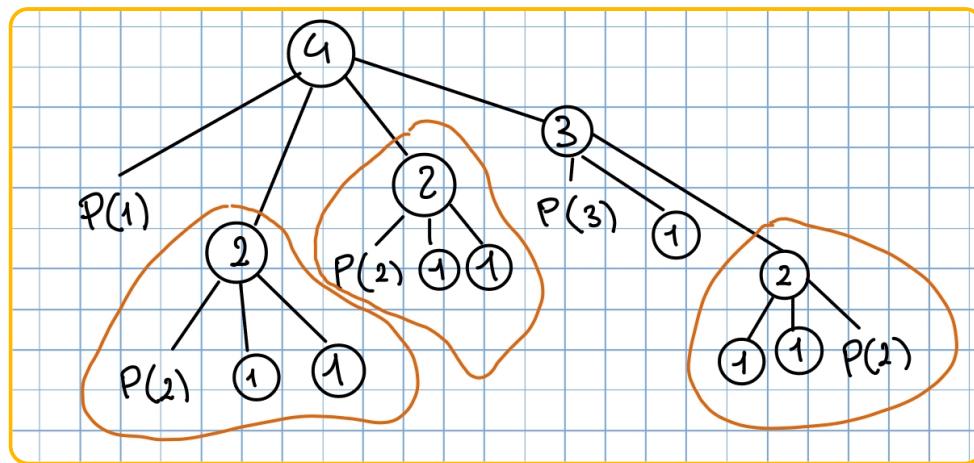
$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

quindi possiamo definire il nostro problema come:

$$r(i) = \begin{cases} 0 & \text{se } i = 0 \\ \max_{1 \leq k \leq i} \{p(k) + r(i-k)\} & \text{se } i \geq 1 \end{cases}$$

```
CUT-ROD(p, n)
    if n == 0
        return 0
    q = -infinity
    for i = 1 to n
        q = max(q, p[i] + CUT-ROD(p, n - i))
    return q
```

Di seguito l'albero di ricorsione con $i = 4$



Notiamo subito che ci sono dei sotto-problemi che si ripetono più volte, quindi dobbiamo fare uso della memorizzazione per rendere il tutto più efficiente.

Implementazione bottom-up del ROD-CUT con memorizzazione

Globalmente avremo definito:

- P array con i valori rispetto alla lunghezza
- R array con i valori dei casi già esaminati

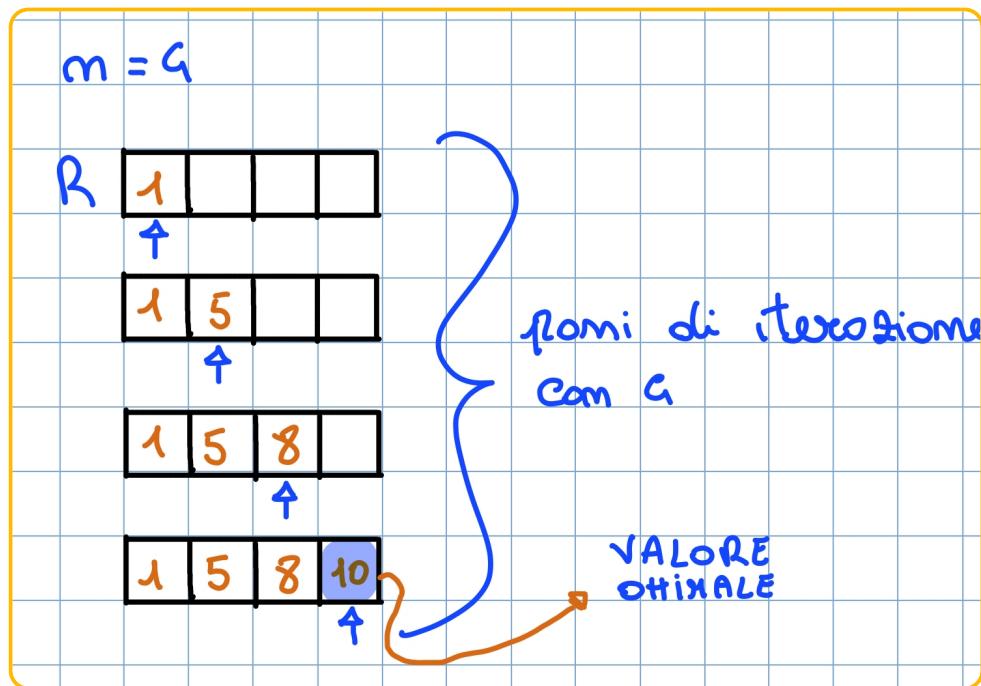
```
ROD-CUT(n)
    if n = 0 then
        return 0

    for i <- 1 to n
        m <- P(i) // salviamo il valore senza aver tagliato
        for k <- 1 to i-1 DO // proviamo tutti i possibili tagli
            if R(k) + R(i-k) > m // vediamo con quale otteniamo il valore migliore
                52/82
```

```

        then m <- R(k) + R(i-k) // salviamo il valore migliore ottenuto
R[i] <- m // lo salviamo nei casi già esaminati

```



Oltre a memorizzare i casi già risolti ora vogliamo memorizzare il modo migliore per tagliare la rod di lunghezza i e lo facciamo usando un array K

Esempio: con $K = [1, \frac{1}{2}, \frac{3}{2}, 2]$ sappiamo che una rod di lunghezza 4 deve essere tagliata nel punto 2 per ottenere il taglio migliore

```

ROD-CUT(n)
    R <- new Array(n)
    K <- new Array(n)

    if n == 0 then
        return 0

    for i <- 1 to n
        m <- P(i)
        K[i] <- i
        for k <- 1 to i-1 DO
            if R[k] + R[i-k] > m then
                m <- R[k] + R[i-k]
                K[i] <- k //Salviamo il taglio migliore

    R[i] <- m

    return K

PRINT-CUT(n, K) //Funzione che stampa il taglio migliore dato n
    if K[n] == n
        print(n)
    else
        PRINT-CUT(K[n], K)
        PRINT-CUT(n - K[n], K)

```

Ottimizzazione della moltiplicazione tra matrici

Questo è un problema di ottimizzazione basato sul prodotto riga per colonna delle matrici, visto che questa operazione gode della proprietà associativa associando i prodotti in maniera diversa è possibile ottenere lo stesso risultato ma con un numero di operazioni scalari minori

Definizione della moltiplicazione tra matrici

Date due matrici A e B per poter effettuare il prodotto tra queste due dobbiamo rispettare la seguente condizione:

- Il numero delle colonne di A deve essere uguale al numero di righe di B
L'implementazione della moltiplicazione è la seguente:

```
MATRIX-MULTIPLTY(A,B,p,q,w)
  C = NEW MATRIX[p,w]
  FOR i = 1 TO p DO
    FOR j = 1 TO w DO
      C[i,j] = 0
      FOR k = 1 TO q DO
        C[i,j] = C[i,j] + A[i,k]*B[k,j]
  RETURN c
```

Analizziamo la complessità

Supponendo di voler moltiplicare una sequenza di tre matrici

$$A_1 \cdot A_2 \cdot A_3$$

con:

- $A_1 = p \times q$
- $A_2 = q \times w$
- $A_3 = w \times q$

Il numero di operazioni scalari di $A_1 \cdot A_2$ sarà uguale a $p \cdot q \cdot w$ (ottenendo in output una matrice di dimensione $p \times w$). Quindi la complessità di $A_1 \cdot A_2 \cdot A_3$ sarà uguale alla somma della complessità tra i vari prodotti riga per colonna, quindi la complessità cambia in base a quale operazione effettuiamo prima.

Esempio

Con

- $A_1 = 10 \times 100$
- $A_2 = 100 \times 5$
- $A_3 = 5 \times 50$

La complessità della parentesizzazione $(A_1 \cdot A_2) \cdot A_3$ sarà:

$$10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 5000 + 2500 = 7500$$

La complessità della parentesizzazione $A_1 \cdot (A_2 \cdot A_3)$ sarà:

$$100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 25000 + 50000 = 75000$$

Associazioni differenti creano complessità molto differenti, la domande dalla quale nasce questo problema è:

Come troviamo la parentesizzazione migliore per minimizzare il numero di operazioni scalari?

Fasi della programmazione dinamica

Premesse comuni a tutte le fasi

- Abbiamo una sequenza di matrici A_1, A_2, \dots, A_n
- abbiamo un vettore di interi $P = P_0, P_1, P_2, P_3, \dots, P_n$
Le dimensioni della matrice A_i sono P_{i-1} e P_i

1. Ricerca della sottostruttura ottima

Presa la nostra sequenza di matrici A_1, A_2, \dots, A_n supponiamo che una parentesizzazione ottima di quest'ultima

suddivida il prodotto in questo modo

$$(A_1 \dots A_k) \cdot (A_{k+1} \dots A_n)$$

ovviamente anche entrambe le sotto sequenze devono essere parentesizzate in modo ottimo e quindi avremo che:

$$S_{1,n}^* = S_{1,k}^* + S_{k+1,n}^* + P_0 P_K P_n$$

dove $P_0 P_K P_n$ sono il numero di moltiplicazioni per $A_{1,k} \cdot A_{k+1,n}$.

Dimostrazione:

Se ci fosse un modo meno costoso di parentesizzare $A_1 \dots A_k$ sostituendo questa parentesizzazione in quella ottima otterremo un'altra parentesizzazione di A_1, A_2, \dots, A_n il cui costo sarebbe minore di quella ottima: una contraddizione.

Formalmente:

Supponiamo che la sottostruttura non sia ottima. In particolare supponiamo che per la prima parte (1 a k) esista una soluzione migliore di quella usata, esiste un $S_{1,k}^{\text{migliore}}$ tale che:

$$S_{1,k}^{\text{migliore}} < S_{1,k}^*$$

se costruiamo la soluzione ottima con questa versione più economica otteniamo

$$S_{1,n}^{\text{migliore}} = S_{1,k}^{\text{migliore}} + S_{k+1,n}^* + P_0 P_K P_n$$

Visto che abbiamo ipotizzato che $S_{1,k}^{\text{migliore}} < S_{1,k}^*$ allora necessariamente

$$S_{1,n}^{\text{migliore}} < S_{1,n}^*$$

Questo è un assurdo in quanto avevamo dichiarato che $S_{1,n}^*$ era già la soluzione ottimale

Usiamo la sottostruttura ottima per trovare una soluzione ricorsiva

2. Definizione ricorsiva di una soluzione ottima

Abbiamo la nostra sequenza di matrici: $A_i \dots A_j$ preso un k generico avremo che $(A_i \dots A_k)(A_{k+1} \dots A_j)$ e su questo definiamo:

$$S_{i,j} = \begin{cases} 0 & \text{se } i = j \\ \min_{i \leq k < j} (S_{i,k} + S_{k+1,j} + p_{i-1} p_k p_j) & \text{se } i < j \end{cases}$$

Segue l'implementazione:

```
MATRIX-CHAIN-ORDER(p, i, j):
    if i == j:
        return 0

    sm = +infinity // elemento più grande

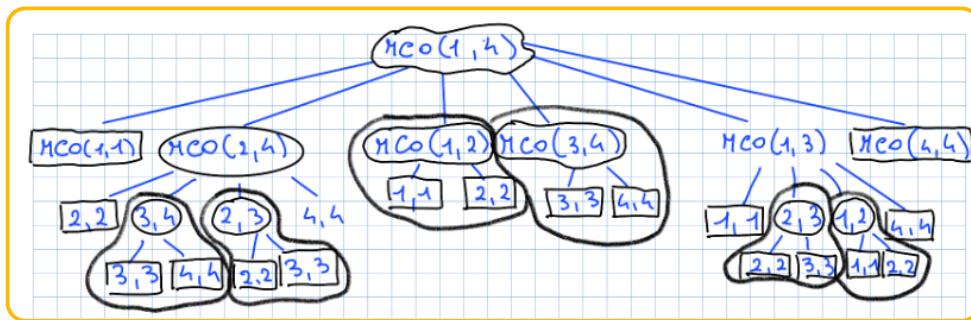
    for k from i to j-1:
        // Chiamate ricorsive per calcolare i costi dei sottoproblemi
        res_sinistro = MATRIX-CHAIN-ORDER(p, i, k)
        res_destro = MATRIX-CHAIN-ORDER(p, k+1, j)

        // Calcolo del costo attuale: costo sx + costo dx + costo moltiplicazione finale
        costo_attuale = res_sinistro + res_destro + p[i-1] * p[k] * p[j]

        if sm > costo_attuale:
            sm = costo_attuale

    return sm
```

Se eseguiamo questa funzione con $i = 1$ e $j = 4$ cerchiamo il modo ottimale di moltiplicare A_1, A_2, A_3, A_4 e otteniamo il seguente albero di ricorsione:



dalla quale notiamo subito che ci sono dei sotto problemi che si ripetono quindi dobbiamo usare una approccio bottom-up per rendere più efficiente l'algoritmo

3. Ricerca di una soluzione ottima usando un approccio bottom-up

Per risolvere il problema dei sotto problemi che si ripetono dobbiamo fare uso della memorizzazione, in questo caso non attraverso un array ma attraverso una matrice, visto che abbiamo due indici. Data la matrice S possiamo dire che:

- $S[i, j]$ è il costo minimo per moltiplicare le matrici da A_i fino ad A_j
- la diagonale principale contiene sempre 0 perché moltiplicare una matrice per stesse non richiede alcuna operazione
- Visto che l'algoritmo calcola il costo solo dove $i < j$ sarà sola il triangolo superiore alla diagonale ad essere riempito
- Il risultato finale ovvero il costo minimo per moltiplicare la sequenza di n matrici si trova in posizione $S[1, n]$

```
MATRIX-CHAIN-ORDER(P, n)
S = new matrix(n, n)

// Inizializzazione diagonale: costo 0 per catene di lunghezza 1
for i = 1 to n do
    S[i, i] = 0      // Dim 1

// l è la lunghezza della catena
for l = 2 to n do  // Dim l > 1
    for i = 1 to n - l + 1 do
        j = i + l - 1
        S[i, j] = +∞

        for k = i to j - 1 do
            // Calcolo del costo temporaneo
            costo = S[i, k] + S[k + 1, j] + P[i-1] * P[k] * P[j]

            // Se il nuovo costo è minore, aggiorna la matrice
            if S[i, j] > costo then
                S[i, j] = costo

return S[1, n]
```

ESEMPIO

$$A_1(10 \times 100) \quad A_2(100 \times 5) \quad A_3(5 \times 50)$$

$$P = [10, 100, 5, 50]$$

STEP 0: inizializzazione

	1	2	3
1	○		
2	-	○	
3	-	-	○

STEP 1: calcolare $S[1,2]$ (comune $A_1 A_2$) e $S[2,3]$ (comune $A_2 A_3$)

$S[1,2]$ ($i=1, j=2$) unico punto di scattura è $k=1$

$$\text{Formule: } S[1,1] + S[2,2] + (P_0 \cdot P_1 \cdot P_2)$$

$$\text{Calcolo: } 0 + 0 + (10 \cdot 100 \cdot 5) = 5000$$

$S[2,3]$ ($i=2, j=3$) unico punto di scattura è $k=2$

$$\text{Formule: } S[2,2] + S[3,3] + (P_1 \cdot P_2 \cdot P_3)$$

$$\text{Calcolo: } 0 + 0 + (100 \cdot 5 \cdot 50) = 25000$$

	1	2	3
1	○ 5000		
2	-	○ 25000	
3	-	-	○

STEP 2: calcolare $S[1,3]$ (comune $A_1 A_2 A_3$)

Ora abbiamo due punti di scatture possibili:

$$k=1 \quad (A_1 \times (A_2 A_3))$$

$$\text{Formule: } S[1,1] + S[2,3] + (P_0 \cdot P_1 \cdot P_3)$$

$$\text{Calcolo: } 0 + 25000 + (10 \cdot 100 \cdot 50) = 45000$$

$$k=2 \quad ((A_1 A_2) \times A_3)$$

$$\text{Formule: } S[1,2] + S[3,3] + (P_0 \cdot P_2 \cdot P_3)$$

$$\text{Calcolo: } 5000 + 0 + (10 \cdot 5 \cdot 50) = 4500$$

$$\min(45000, 4500) \rightarrow$$

	1	2	3
1	○ 5000	○ 4500	
2	-	○ 25000	
3	-	-	○

soluzione ottimale

Questo algoritmo calcola il *costo minimo della soluzione* ma non abbiamo modo di sapere quale sia l'ordine effettivo delle matrici da moltiplicare.

Molti algoritmi richiedono di trovare il valore della soluzione ottima, quindi ci potremmo fermare al terzo step, in questo caso però abbiamo bisogno anche ricostruire la parentesizzazione della soluzione ottima in modo da poter effettuare la vera e propria moltiplicazione, da questa esigenza nasce lo step successivo, solitamente opzionale.

4. Costruzione della soluzione ottimale

```

MATRIX-CHAIN-ORDER(p, n)
S = new matrix(n, n)
D = new matrix(n, n)

for i = 1 to n do:
    S[i, i] = 0

for l = 2 to n do:
    for i = 1 to n - l + 1 do:
        j = i + l - 1
        S[i, j] = +infinity
    
```

```

        for k = i to j - 1 do: # k è il punto di scissione
            q = S[i, k] + S[k+1, j] + p[i-1] * p[k] * p[j]

            if S[i, j] > q:
                S[i, j] = q
                D[i, j] = k      # Memorizza il punto di scissione ottimale

    return S[1, n]

PRINT-CHAIN(D, i, j)
    if i == j:
        print "A" + i
    else:
        k = D[i, j]
        print "("
        PRINT-CHAIN(D, i, k)
        PRINT-CHAIN(D, k+1, j)
        print ")"

```

Mentre l'algoritmo che è rimasto essenzialmente lo stesso si occupa di esplorare e trovare tutti i possibili modi di parentesizzare la catena k , la matrice $D[i, j]$ salva esattamente quale indice k ha prodotto la parentesizzazione migliore. La funzione print-chain serve per stampare la parentesizzazione migliore in base ai parametri passati.

Complessità

Avendo tre cicli annidati la complessità è $O(n^3)$, è una complessità molto alta, infatti questo algoritmo viene utilizzando quando si devono moltiplicare matrici molto grandi dove c'è un vero e proprio guadagno.

Distanza di Editing e LCS

Introduzione al Problema

Il problema fondamentale analizzato riguarda il calcolo della distanza tra due stringhe, X e Y . Esistono diverse metriche per valutare questa distanza, come la **Distanza di Hamming** (che considera solo sostituzioni su stringhe di uguale lunghezza) o la più generale **Distanza di Editing** (o distanza di Levenshtein).

La Distanza di Editing definisce il numero minimo di operazioni necessarie per trasformare la stringa X nella stringa Y . Le operazioni consentite, ciascuna con un costo unitario, sono:

1. **Inserimento** di un carattere.
2. **Cancellazione** di un carattere.
3. **Sostituzione** di un carattere.

Consideriamo ad esempio la trasformazione dalla parola "CASA" alla parola "CHIESA".

- $CASA \rightarrow CHASA$ (Inserimento 'H')
- $CHASA \rightarrow CHESA$ (Sostituzione 'A' con 'E')
- $CHESA \rightarrow CHIESA$ (Inserimento 'I')

Definizione Formale e Ricorsiva

Date due stringhe:

- X di lunghezza n , dove $x_i = X[1 \dots i]$ è il prefisso di lunghezza i .
- Y di lunghezza m , dove $y_j = Y[1 \dots j]$ è il prefisso di lunghezza j .

Definiamo $ED(i, j)$ come la distanza di editing tra i prefissi x_i e y_j . La soluzione può essere espressa ricorsivamente:

Casi Base

Se una delle due stringhe è vuota, la distanza è pari alla lunghezza dell'altra stringa (tutti inserimenti o cancellazioni):

- $ED(i, 0) = i$
- $ED(0, j) = j$

Passo Ricorsivo

Per calcolare $ED(i, j)$ con $i, j > 0$, consideriamo i caratteri x_i e y_j :

1. Se $x_i = y_j$ (Match): Non è necessaria alcuna operazione sui caratteri correnti. Il costo è ereditato dai prefissi precedenti:

$$ED(i, j) = ED(i - 1, j - 1)$$

2. Se $x_i \neq y_j$ (Mismatch): Dobbiamo scegliere l'operazione che minimizza il costo tra le tre possibili:

- *Sostituzione*: $ED(i - 1, j - 1) + 1$
- *Cancellazione (da X)*: $ED(i - 1, j) + 1$
- *Inserimento (in Y)*: $ED(i, j - 1) + 1$

La formula completa è:

$$ED(i, j) = \min(ED(i - 1, j) + 1, ED(i, j - 1) + 1, ED(i - 1, j - 1) + \text{costo_sostituzione})$$

(Nota: il costo di sostituzione è 1 se i caratteri sono diversi, 0 se uguali).

Programmazione Dinamica

L'approccio ricorsivo puro ("top-down") è inefficiente a causa della ricalcolazione dei sottoproblemi sovrapposti. La soluzione ottimale utilizza la **Programmazione Dinamica** ("bottom-up"), costruendo una tabella (matrice) di dimensione $(n + 1) \times (m + 1)$.

Algoritmo per il calcolo della Distanza (EDT)

Input: Stringhe X, Y di lunghezza n, m .

Complessità: Tempo $O(n \times m)$, Spazio $O(n \times m)$ (ottimizzabile a $O(\min(n, m))$ se si memorizzano solo le ultime due righe).

Pseudocodice:

```
EDT(X, Y, n, m)
Dichiara matrice ED[0..n, 0..m]

// Inizializzazione
FOR i = 0 TO n DO ED[i, 0] = i
FOR j = 0 TO m DO ED[0, j] = j

// Riempimento Matrice
FOR i = 1 TO n DO
    FOR j = 1 TO m DO
        IF X[i] == Y[j] THEN
            ED[i, j] = ED[i-1, j-1]
        ELSE
            ED[i, j] = min(ED[i, j-1],      // Inserimento
                           ED[i-1, j],      // Cancellazione
                           ED[i-1, j-1]) // Sostituzione
                           + 1
```

Ricostruzione della Soluzione Ottima

Una volta compilata la matrice, il valore in $ED[n, m]$ rappresenta il costo minimo. Per trovare la sequenza di operazioni, si esegue un "backtracking" dalla cella (n, m) alla cella $(0, 0)$.

Pseudocodice Ricostruzione:

```
PRINT-EDT(ED, X, Y, n, m)
  i = n, j = m
  WHILE (i > 0 OR j > 0)
    IF (i > 0 AND j > 0 AND X[i] == Y[j]) THEN
      // Match: nessun costo, ci spostiamo in diagonale
      i = i - 1
      j = j - 1
    ELSE
      // Determiniamo da quale cella proviene il minimo
      min_val = min(ED[i, j-1], ED[i-1, j], ED[i-1, j-1])

      IF (j > 0 AND min_val == ED[i, j-1]) THEN
        STAMPA("Inserimento di " + Y[j])
        j = j - 1
      ELSE IF (i > 0 AND min_val == ED[i-1, j]) THEN
        STAMPA("Cancellazione di " + X[i])
        i = i - 1
      ELSE
        STAMPA("Sostituzione di " + X[i] + " con " + Y[j])
        i = i - 1
        j = j - 1
```

Longest Common Substring

Un problema correlato è trovare la più lunga stringa di caratteri *consecutivi* comune a entrambe le stringhe.

Definizione Ricorsiva (Suffissi)

Sia $LCS_{suffix}(i, j)$ la lunghezza del suffisso comune più lungo tra $X[1 \dots i]$ e $Y[1 \dots j]$.

- Se $X[i] = Y[j]$: $LCS_{suffix}(i, j) = LCS_{suffix}(i - 1, j - 1) + 1$
- Se $X[i] \neq Y[j]$: $LCS_{suffix}(i, j) = 0$ (la continuità si interrompe).

Durante il calcolo, è necessario mantenere una variabile globale `max_len` per tracciare il valore massimo trovato nella matrice, poiché la sottostringa più lunga può terminare in qualsiasi punto, non necessariamente alla fine delle stringhe.

Ottimizzazione Spaziale:

Poiché per calcolare la riga i serve solo la riga $i - 1$, è possibile utilizzare solo due array ("current" e "previous") invece di un'intera matrice, riducendo lo spazio.

Longest Common Subsequence

A differenza della sotto-stringa, una **sotto-sequenza** non richiede che i caratteri siano consecutivi, ma deve mantenere l'ordine relativo originale.

Esempio:

$X = ACTAAA$

$Y = CCATAG$

$LCS = "ATA"$ (lunghezza 3).

Relazione di Ricorrenza

Sia $LCS(i, j)$ la lunghezza della più lunga sotto-seguenza comune tra $X[1 \dots i]$ e $Y[1 \dots j]$.

1. Se $X[i] = Y[j]$: Il carattere fa parte della LCS.

$$LCS(i, j) = LCS(i - 1, j - 1) + 1$$

2. Se $X[i] \neq Y[j]$: Il carattere corrente non può estendere una corrispondenza comune. La soluzione è il massimo tra ignorare il carattere di X o ignorare il carattere di Y .

$$LCS(i, j) = \max(LCS(i - 1, j), LCS(i, j - 1))$$

Ricostruzione e Stampa della LCS

Per stampare la sotto-seguenza, si utilizza una procedura ricorsiva partendo dalla fine della matrice calcolata (L).

```
PRINT-LCS(L, X, Y, i, j)
IF (L[i, j] == 0) RETURN

IF (X[i] == Y[j]) THEN
    // Trovato un carattere comune: stampa prima i precedenti (ordine inverso)
    PRINT-LCS(L, X, Y, i-1, j-1)
    STAMPA(X[i])
ELSE
    // Segui la direzione del massimo valore
    IF (L[i-1, j] >= L[i, j-1]) THEN
        PRINT-LCS(L, X, Y, i-1, j)
    ELSE
        PRINT-LCS(L, X, Y, i, j-1)
```

Algoritmi golosi

Introduzione

Definizione

Un'altra strategia che possiamo sfruttare per risolvere problemi di ottimizzazione è la strategia greedy, in italiano golosa. Un algoritmo goloso fa sempre la scelta che sembra ottima in un determinato momento, ovvero fa una scelta *localmente ottima*, nella speranza che tale scelta porterà a una soluzione globalmente ottima. La scelta greedy che viene fatta cambia da problema a problema.

Problema dello zaino

Definizione

Riprendendo il problema dello zaino presentato nella lezione precedente, ricordiamo che:

- k capienza dello zaino
- $A = \{a_1, a_2, \dots, a_n\}$
- $P(a_i)$ = peso dell'oggetto
- $V(a_i)$ = valore dell'oggetto
- *Obiettivo*: massimizzare il guadagno
in questo caso però ogni oggetto ha lo stesso valore ovvero:

$$v(a) = 1 \quad \forall a \in A$$

Banalmente ciò che dobbiamo massimizzare è il numero di oggetti. Un ottima strategia (greedy) è quella di scegliere l'oggetto con il peso minore.

Dimostrazione scelta greedy

$$a_w : P(a_w) = \min\{\text{Peso}(a_j) : a_j \in A\}$$

Ovvero a_w scelta greedy. $S \subseteq A$ è una soluzione ottima. Allora $a_w \in S$.

Ipotizziamo per assurdo che $a_w \notin S$.

$$a_\phi \rightarrow P(a_\phi) = \min\{P(a_j) : a_j \in S\}$$

Ovvero a_ϕ elemento meno pesante della soluzione ottima.

Allora consideriamo $S - \{a_\phi\}$ e considero

$$S' = (S - \{a_\phi\}) \cup \{a_w\}$$

Per ipotesi, S' è una soluzione ottima, per cui

$$\sum_{a \in S'} P(a) \leq k$$

Compressione di Huffman

Definizione

I codici di Huffman sono una tecnica molto diffusa ed efficace per comprimere i dati, risparmi dal 20% al 90% sono tipici, a seconda delle caratteristiche dei dati da comprimere. Consideriamo la compressione di un testo dove ogni carattere è rappresentato da una stringa binaria detta *parola in codice*, queste possono essere create in diversi modi. Spesso l'alfabeto (ovvero tutti i caratteri presenti nel testo) viene indicato con Σ

Come costruiamo le parole in codice

Lunghezza fissa: se utilizziamo un codice a lunghezza fissa occorrono 3 bit per rappresentare sei caratteri: $a = 000, b = 001, \dots, f = 101$ (posso rappresentare 2^3 caratteri)

Lunghezza variabile: questa codifica assegna ai caratteri più frequenti delle parole in codice corte e ai meno frequenti delle parole in codice lunghe

	a	b	c	d	e	f
Frequenza (in migliaia)	45	13	12	16	9	5
Parola in codice a lunghezza fissa	000	001	010	011	100	101
Parola in codice a lunghezza variabile	0	101	100	111	1101	1100

Codici prefissi

I codici considerati d'ora in avanti sono soltanto i codici in cui nessuna parola in codice è anche un prefisso di qualche altra parola in codice, questi codici sono detti *codici prefissi*.

💡 Perché usiamo i codici prefissi

Data una tabella di codifica del tipo:

$A = 0$

$B = 01$

$C = 1$

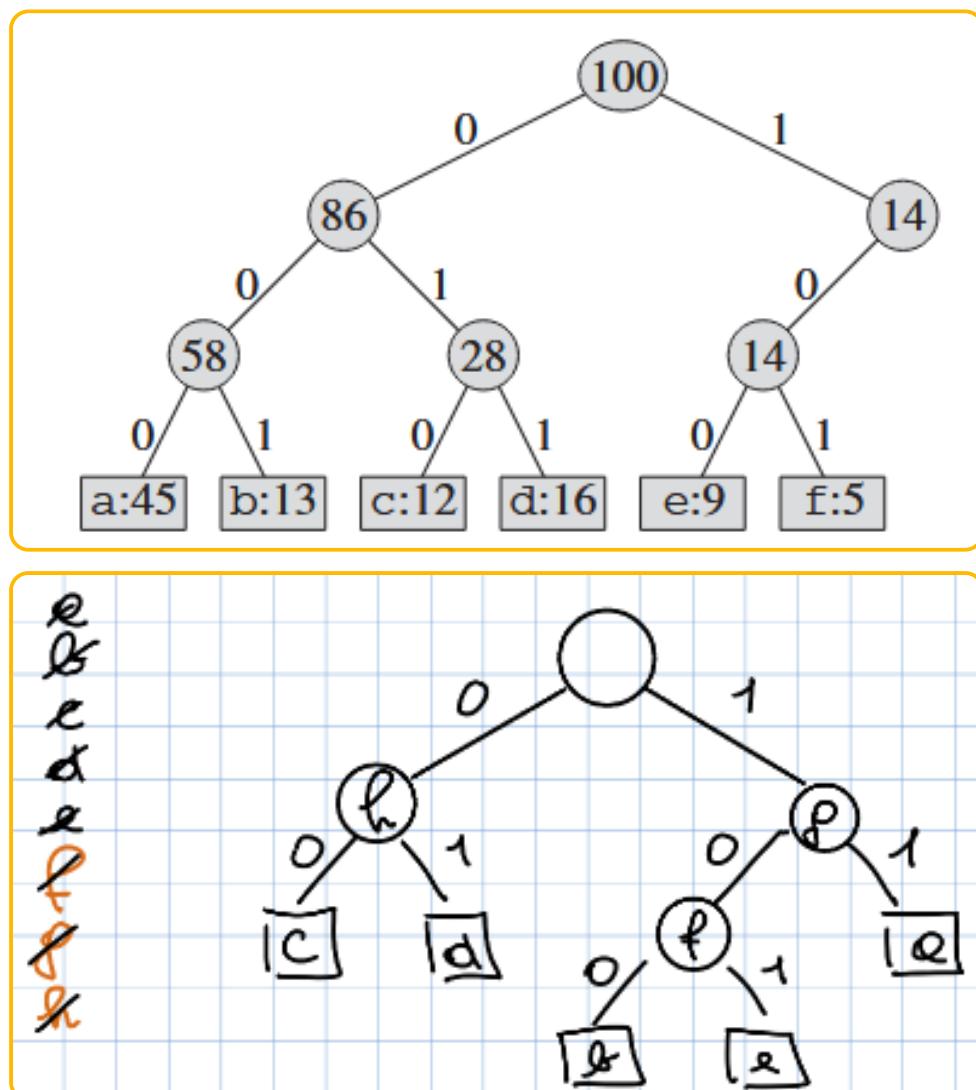
Notiamo subito che il codice di A è un prefisso del codice B .

Questo comporta che quando al decodificatore arriva un codice del tipo 01 non sa se questo equivale ad AC o solo B

Per creare questo tipo di codici usiamo l'albero di Huffman che funziona in questo modo:

- Ogni foglia è etichettata con un carattere e la sua frequenza
 - Ogni nodo interno è etichettato con la somma delle frequenze delle foglie nel suo sottoalbero
- Per creare i nodi interni prendiamo i nodi con la frequenza più bassa e li "uniamo" in un padre con la somma delle frequenze (alla quale assegniamo un'altra lettera), facendo questa cosa ricorsivamente otterremo un

albero del tipo:



Il cammino semplice che c'è tra la radice e quel carattere *definisce il suo codice prefisso*. **La profondità di una foglia corrisponde al numero di bit necessari per la codifica**

Costo di una codifica

Dato un albero T che corrisponde a un codice prefisso, è semplice calcolare il numero di bit richiesti per codificare un file. Dato un nodo foglia c definiamo:

- $f(c)$ la frequenza di quel nodo nel file
 - $d_T(c)$ la profondità della foglia nell'albero
- Il costo della codifica di un albero diventa:

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

quindi dobbiamo trovare una strategia che dato testo minimizza $B(T)$

Dimostrazione della sottostruttura ottima

Definizioni iniziali:

- Σ : Insieme dei caratteri (soluzione T)
- Consideriamo due nodi a e b (le foglie con frequenza minima) e il loro genitore z .
- $\Sigma' = \Sigma - \{a, b\} \cup \{z\}$ (soluzione T')

Relazioni tra frequenze e profondità:

1. $f(z) = f(a) + f(b)$

- La frequenza del nodo padre è la somma delle frequenze dei figli.
2. $d_T(a) = d_{T'}(z) + 1$ e $d_T(b) = d_{T'}(z) + 1$
- La profondità di a/b nell'albero originale è la profondità di z nell'albero ridotto più 1.

Costruzione di una relazione tra T e T\$:

Il costo dell'albero T , denotato come $B(T)$, è dato dalla somma delle frequenze per le profondità:

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

3. Espandendo la sommatoria per mettere in relazione T con T' :

$$B(T) = \left[\sum_{c \in \Sigma'} f(c) d_{T'}(c) \right] - f(z) \cdot d_{T'}(z) + f(a)d_T(a) + f(b)d_T(b)$$

Nota: Il termine tra parentesi quadre è $B(T')$. Sottraiamo il contributo di z (che è in T' ma non è foglia in T) e aggiungiamo i contributi di a e b .

4. Sostituendo le relazioni di profondità e frequenza ($f(z) = f(a) + f(b)$):

$$B(T) = B(T') - (f(a) + f(b))d_{T'}(z) + f(a)(d_{T'}(z) + 1) + f(b)(d_{T'}(z) + 1)$$

5. Svolgendo i calcoli, i termini con $d_{T'}(z)$ si cancellano:

$$= f(a) + f(b)$$

6. Quindi la relazione fondamentale è:

$$B(T) = B(T') + f(a) + f(b)$$

Suppongo che la sottostruttura ottima non esista

- *Hipotesi per assurdo:* Supponiamo che T **non** sia l'albero ottimo per Σ (e quindi anche che T' non sia l'albero ottimo per Σ'). Di conseguenza, deve esistere un albero T'_{opt} con costo strettamente inferiore a T :

$$B(T'_{opt}) < B(T)$$

- *Costruzione dell'albero ridotto:* Prendiamo T'_{opt} e uniamo le foglie a e b nel padre z . Otteniamo un nuovo albero T'_{opt}' valido per l'alfabeto ridotto Σ' . Il costo di questo albero ridotto è:

$$B(T'_{opt}') = B(T'_{opt}) - (f(a) + f(b))$$

- *Sviluppo algebrico:* Riprendiamo la diseguaglianza del punto 1:

$$B(T'_{opt}) < B(T)$$

Sottraiamo a entrambi i membri la quantità costante ($f(a) + f(b)$):

$$B(T'_{opt}) - (f(a) + f(b)) < B(T) - (f(a) + f(b))$$

Sostituiamo i termini con le definizioni dei costi ridotti ($B(T'_{opt})$ e $B(T')$):

$$\underbrace{B(T'_{opt}) - (f(a) + f(b))}_{B(T'_{opt}')} < \underbrace{B(T) - (f(a) + f(b))}_{B(T')}$$

Otteniamo infine:

$$B(T'_{opt}') < B(T')$$

Abbiamo dimostrato l'esistenza di un albero T'_{opt}' con costo inferiore a T' . Questo *contraddice l'ipotesi iniziale* che T' fosse l'albero ottimo per Σ' .

Implementazione

```

HUFFMAN( $\Sigma$ ,  $f$ )
Q <- new Priority Queue

// Inizializzazione: crea un nodo foglia per ogni carattere
foreach c in  $\Sigma$  do:
    
```

```

x <- new Node(c)
insert x in Q

// Costruzione dell'albero dal basso verso l'alto
for i <- 1 to |Σ| - 1 DO
    x <- extractMin(Q)      // Estraie il nodo con frequenza minima
    y <- extractMin(Q)      // Estraie il secondo nodo con frequenza minima

    z <- new Node            // Crea un nuovo nodo interno
    f(z) = f(x) + f(y)       // Somma le frequenze
    left(z) = x              // Assegna i figli
    right(z) = y

    insert z in Q           // Inserisce il nuovo nodo nella coda

return extractMin(Q)        // Ritorna la radice dell'albero

```

Il processo inizia inserendo tutti i caratteri dell'alfabeto in una coda di priorità, dove vengono trattati come nodi foglia ordinati in base alla loro frequenza di apparizione. L'algoritmo procede estraendo iterativamente i due nodi con la frequenza minore assoluta e unendoli per creare un nuovo nodo intermedio, il cui valore è dato dalla somma delle frequenze dei due figli. Questo nuovo nodo viene reinserito nella coda, e l'operazione si ripete ciclicamente fino a quando rimane un unico nodo che diventa la radice dell'albero. Il risultato è una struttura in cui i caratteri più frequenti si trovano più vicini alla radice (ottenendo codici binari più corti), mentre quelli più rari sono posizionati in profondità (con codici più lunghi).

Grafi

Proprietà dei grafi

Definizione e proprietà

Un grafo è definito come

$$G = (V, E)$$

con

- $V = \{v_1, v_2, v_3 \dots v_n\}$ insieme di vertici
- $E = \{(a_i, a_j) : i, j \in V\}$ insieme di archi

Se $(v_i, v_j) = (v_j, v_i)$ allora il grafo si dice **non orientato**

Se $(v_i, v_j) \neq (v_j, v_i)$ allora il grafo si dice **orientato**

Cammino

Dato un percorso $P = < u_1, u_2, u_3, \dots u_k >$ è un cammino se:

- $u_i \in V \quad \forall i \in 1 \leq i \leq k$ (tutti gli elementi devono essere vertici del grafo)
- $(u_i, u_{i+1}) \in E \quad \forall i \in 1 \leq i \leq k$ (ogni coppia di nodi consecutivi deve essere collegata da un arco)

Se nel cammino abbiamo che $u_1 = u_k$ allora c'è un **ciclo**

Cammino semplice

I nodi devono essere tutti diversi, quindi è un cammino senza cicli

Grafo aciclico

Un grafo è detto aciclico quando non ha cicli

Se un grafo ha un ciclo allora ne ha infiniti

Grafo pesato

Un grafo è pesato se ad ogni arco diamo un peso una la funzione peso:

$$w : E \rightarrow R$$

quindi il costo totale di un eventuale cammino è:

$$w(P) = \sum_{i=1}^{k-1} w(u_i, u_{i+1})$$

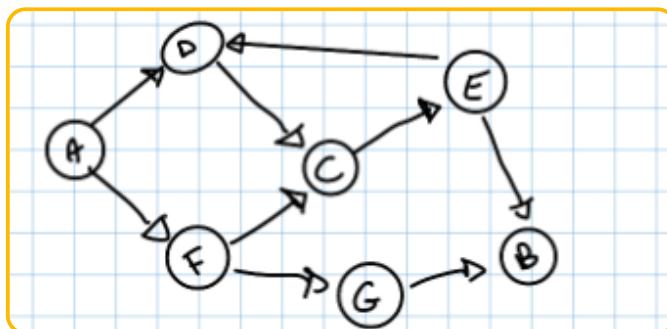
la maggior parte dei grafi pesati ha valori positivi negli archi

Ordinamento topologico

Un ordinamento topologico è un ordinamento lineare dei nodi in modo che valga la relazione:

$$\exists(u, v) \in R \Rightarrow u < v$$

Ovviamente se c'è un ciclo questa caratteristica non può essere rispettata



Un eventuale ordinamento di questo grafo è: $A - F - D - G - C - E - B$

Componente connessa e fortemente connessa

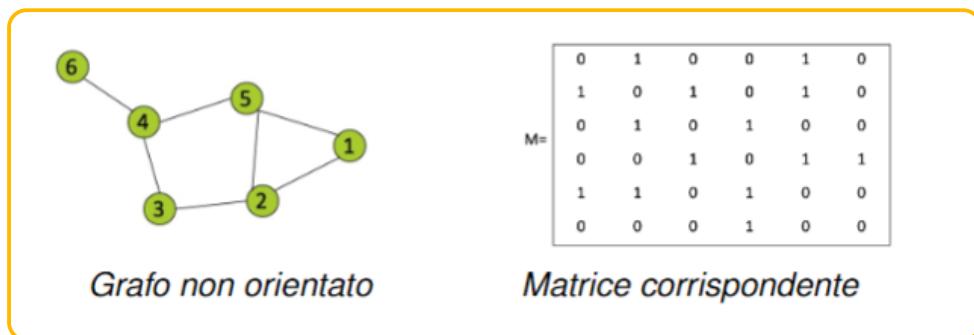
Componente connessa: dato un grafo $G = (V, E)$ diciamo che due vertici u, v sono connessi se esiste un cammino da u a v

Componente fortemente connessa: dato un grafo orientato $G = (V, E)$, diciamo che due vertici u, v sono fortemente connessi se esiste sia un cammino da u a v che un cammino da v ad u .

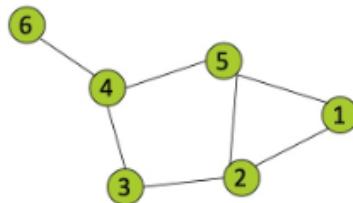
Rappresentazioni dei grafi

Tramite matrice di adiacenza: Dato un grafo $G = (V, E)$ con $|V| = n$. La matrice M detta matrice di adiacenza del grafo ha dimensione $n \times n$ dove n è il numero di vertici, ed è formata in questo modo:

- $M[i, j] = 1$ se i vertici i e j sono connessi da un arco
- $M[i, j] = 0$ se i vertici i e j non sono connessi da un arco



Tramite lista di adiacenza: un grafo può essere rappresentato pure con le liste di adiacenza, ovvero un array i cui elementi sono i nodi e per ogni nodo viene associato un altro array con la lista dei nodi collegati ad esso.



Grafo non orientato

1	2	3	4	5	6
2	1	2	3	1	4
5	3	4	5	2	
5	6	4			

Liste di adiacenza

BFS

Definizione

La Breadth-First-Search ovvero ricerca in ampiezza è una algoritmo di ricerca nei grafi. Dato un grafo $G = (V, E)$ e un vertice distinto s detto *sorgente*, la visita in ampiezza ispeziona sistematicamente gli archi di G per "scoprire" tutti i vertici che sono raggiungibili da s .

Implementazione

Per tenere traccia del lavoro svolto, la visita in ampiezza colora i vertici di:

- *bianco*: nodo non ancora visitato
- *grigio*: nodo scoperto ma non ancora visitato
- *nero*: nodo visitato

All'inizio tutti i nodi sono bianchi, si sceglie un nodo iniziale (la sorgente s) e da lì:

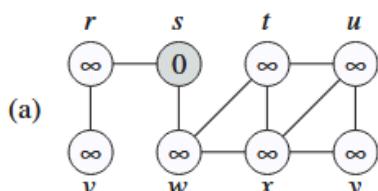
- Aggiungiamo ad una coda tutti i nodi adiacenti a s da (colorandoli di grigio)
- Per ogni nodo aggiunto alla coda calcoliamo la sua distanza da s

```
BFS(V, s)
    Foreach v in V //for di inizializzazione
        color[v] = White
        d[v] = +infinito

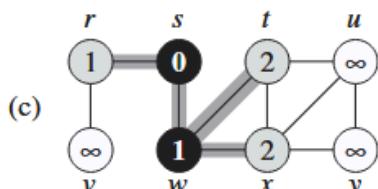
        d[s] = 0; // la distanza della sorgente da stessa è 0
        Q = {} // inizializzazione della coda come vuota
        Q.enqueue(s)

        While Q != {} DO
            v = Q.dequeue(Q)
            foreach u in ADJ(v) //scorriamo tutti i nodi adiacenti a v
                if(color[u] == W)
                    d[u] = d[v]+1
                    color[u] = Gray
                    Q.enqueue(u)
            color[v] = Black
```

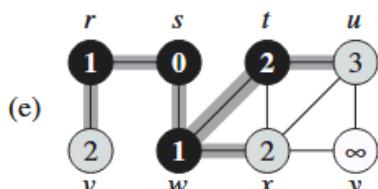
La ricerca oltre a calcolare la distanza di ogni nodo dalla sorgente s crea un albero, detto albero BFS dove il cammino che va da s ad un generico v corrisponde al *cammino minimo* da s a v



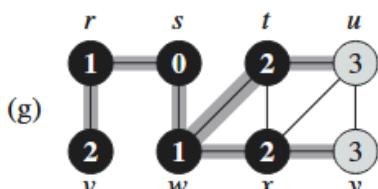
$$Q \quad \boxed{s} \\ 0$$



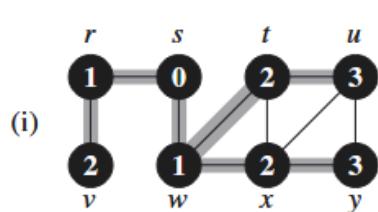
$$Q \quad \boxed{r \ t \ x} \\ 1 \ 2 \ 2$$



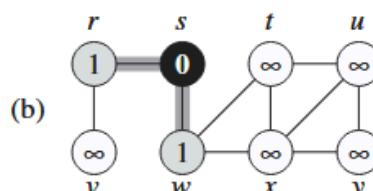
$$Q \quad \boxed{x \ v \ u} \\ 2 \ 2 \ 3$$



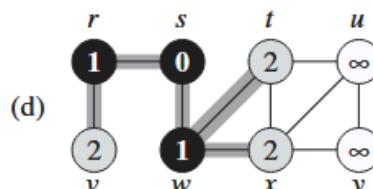
$$Q \quad \boxed{u \ y} \\ 3 \ 3$$



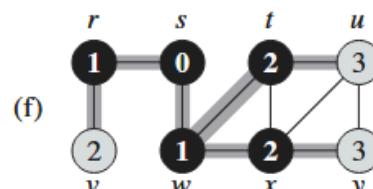
$$Q \quad \emptyset$$



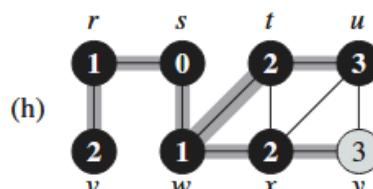
$$Q \quad \boxed{w \ r} \\ 1 \ 1$$



$$Q \quad \boxed{t \ x \ v} \\ 2 \ 2 \ 2$$



$$Q \quad \boxed{v \ u \ y} \\ 2 \ 3 \ 3$$



$$Q \quad \boxed{y} \\ 3$$

DFS

Definizione

Un altro algoritmo di ispezione dei grafi è il depth first search, a differenza del BFS dove la ricerca era in ampiezza qui abbiamo una ricerca in profondità, vedremo successivamente che questo algoritmo ci permette di trarre dal nostro grafo diverse informazioni utili come:

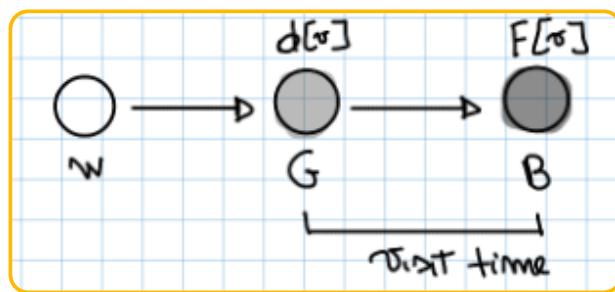
- la composizione delle componenti connesse
 - un ordinamento topologico
- tutto questo lo facciamo grazie all'albero DFS che crea durante la ricerca. Vedremo diverse implementazioni

Premesse comuni alle implementazioni fatte

I nodi sono colorati in modo diverso in base allo stato in cui si trovano:

- *bianco*: nodo non ancora visitato
- *grigio*: nodo scoperto ma non visitato
- *nero*: nodo visitato

Nella nostra implementazione teniamo traccia del tempo di inizio e fine visita di un nodo usando due array d e F , con $d[v]$ e $d[F]$ indichiamo rispettivamente il tempo di inizio e fine di v (T sarà una variabile che scandirà il



Avremo anche altri due array:

- $color[v]$: che ci indicherà il colore del nodo v
- $\Pi[v]$: che ci indicherà il padre di quel nodo nel albero DFS

Inoltre useremo una funzione $adj(v)$ che ritorna una lista dei nodi adiacenti a v

Prima implementazione

```
DFS-visit(v):
    color[v] = gray;
    foreach u in adj(v)
        if color[u] = white
            π[u] = v
            DFS-visit(u)
    color[v] = black

DFS(G,s):
    foreach v in V DO:
        color[v] = white
        π[] = null
    DFS-visit(s);
```

Questa è l'implementazione più grezza, infatti ci potrebbero essere dei nodi non visitati, ad esempio quelli non collegati a nulla. Da queste problematiche nasce la seconda implementazione.

Seconda implementazione

In questa implementazione aggiungiamo un for che scorre la lista dei nodi, in modo che qualsiasi nodo verrà sicuramente visitato, il fatto che ogni nodo può essere una ipotetica sorgente potrebbe creare più alberi DFS ovvero una *foresta DFS*

```
DFS-visit(v):
    color[v] = gray;
    foreach u in adj(v)
        if color[u] = white
            π[u] = v
            DFS-visit(u)
    color[v] = black

DFS(G,s):
    foreach v in V DO:
        color[v] = white
        π[] = null
    foreach v in V DO:
        if color[v] = white:
            DFS-visit(s);
```

Questa implementazione è perfetta, ma non raccoglie nessuna informazione sul nostro grafo, da qui nasce la necessità di introdurre d e F per scandire il tempo, usati nella terza implementazione

Terza implementazione

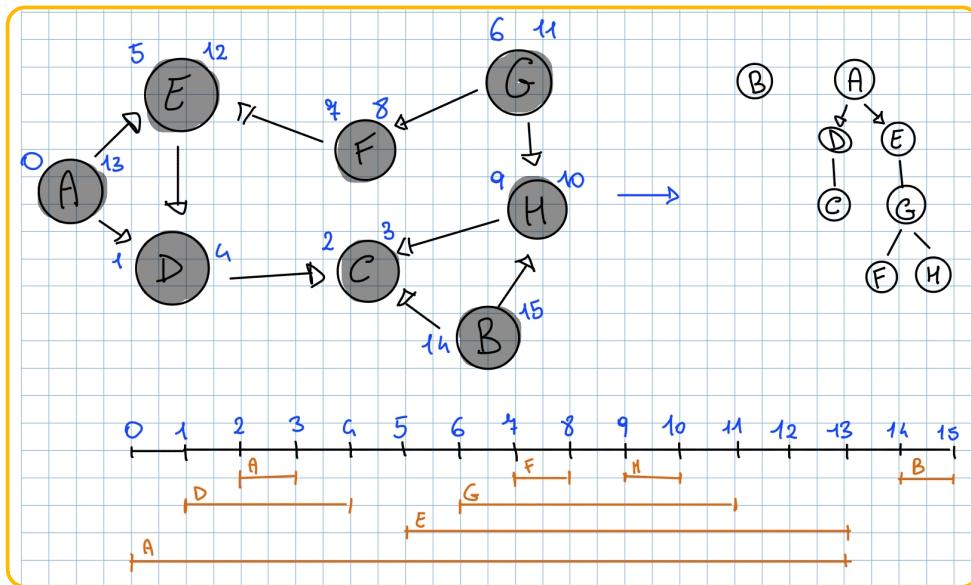
```

DFS-visit(v):
    d[v] = T
    T = T+1
    color[v] = gray;
    foreach u in adj(v)
        if color[u] = white
            π[u] = v
            DFS-visit(u)
    color[v] = black
    F[v] = t
    t = t+1

DFS(G,s):
    foreach v in V DO:
        color[v] = white
        π[] = null
    foreach v in V DO:
        if color[v] = white:
            DFS-visit(s);

```

Alla fine avremo alla fine di questo algoritmo negli array d ed F il tempo di inizio e fine, molto utile nell'utilizzo della DFS per scopi terzi. Di seguito un grafo con le informazioni ottenute dopo una DFS:



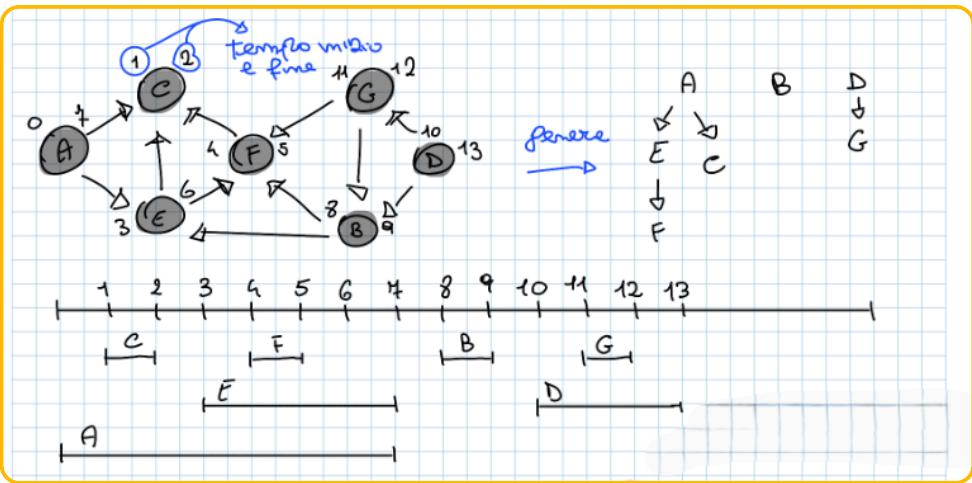
Etichette degli archi

Nel nostro grafo etichettiamo gli archi in base al ruolo che hanno durante l'esplorazione del grafo, e sono:

- T archi consecutivi: sono gli archi usati dalla DFS per scoprire nuovi nodi
- I archi all'indietro: collegano un nodo a un suo antenato nell'albero DFS
- F archi in avanti: collegano un nodo al suo discendente
- C archi trasversali: collegano nodi che non hanno relazione antenato-discendente

Per cosa usiamo la DFS

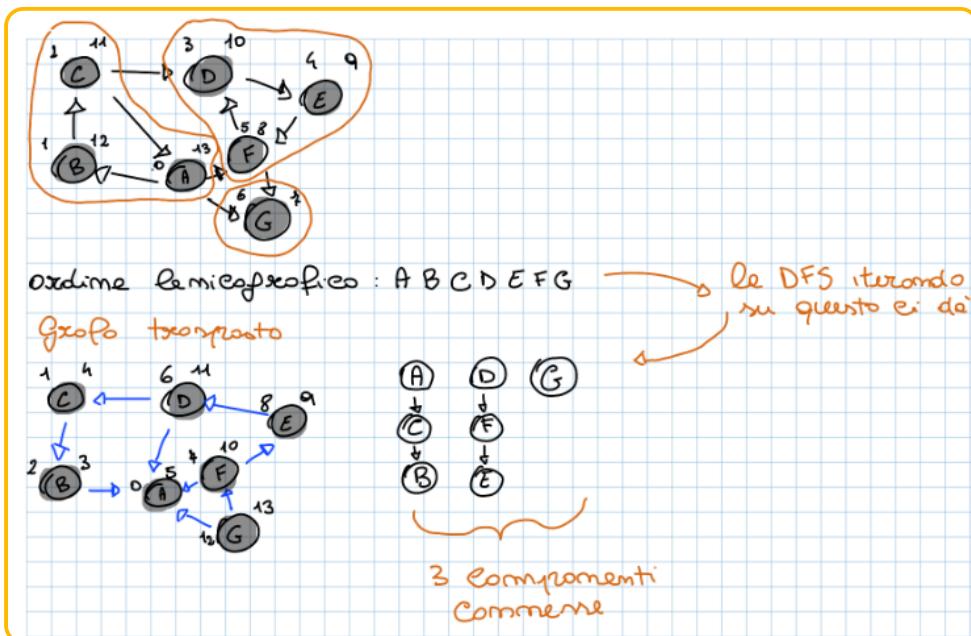
Ordinamento topologico: dato il tempo di inizio e fine visita di ogni vertice del grafo otteniamo un ordinamento topologico valido se mettiamo i vertici in ordine rispetto a di fine visita:



Dai tempi di fine visita otteniamo un ordinamento topologico: $D - G - B - A - E - F - C$

Identificazione delle componenti connesse: per identificare le componenti connesse di un grafo facciamo uso della DFS nel seguente modo:

1. Faccio una DFS e metto in ordine rispetto al tempo di inizio visita
 2. Creo il grafo trasposto di quello dato in input (inverto le direzioni di tutti gli archi)
 3. Faccio la DFS sul grafo trasposto usando nel foreach principale la lista di nodi definita nel punto 1
- Otteniamo una foresta di alberi DFS, tanti alberi quanti sono le componenti connesse. Di seguito un esempio:



Ricerca dei cammini minimi

Introduzione

Definizioni

In un *problema dei cammini minimi* sono dati:

- un grafo orientato pesato $G = (V, E)$
- funzione peso $w : E \rightarrow R$ che associa agli archi dei pesi

Il peso di un cammino $p = \langle v_0, v_1, \dots, v_k \rangle$ è la somma dei pesi degli archi che lo compongono:

$$w(p) = \sum_{i=1}^k w(v_{i-1}, v_i)$$

il peso di un cammino minimo $\delta(u, v)$ da u a v è definito in questo modo:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\} & \text{se esiste un cammino da } u \text{ a } v \\ \infty & \text{negli altri casi} \end{cases}$$

Un **cammino minimo** è definito come un cammino qualsiasi p con peso $w(p) = \delta(u, v)$

Varianti di questo problema

Esistono 4 varianti del problema dei cammini minimi:

1. *Problema dei cammini minimi da sorgente unica(single source)*: dato un grafo vogliamo trovare un cammino minimo che va da un dato vertice sorgente $s \in V$ a ciascun vertice $v \in V$
2. *Problema dei cammini minimi con destinazione unica(single destination)*: trovare un cammino minimo da ciascun vertice v a un dato vertice t destinazione. (Invertendo la direzione di ciascun arco nel grafo lo possiamo ricondurre la prima caso)
3. *Problema del cammino minimo per una coppia di vertici(single pair)*: trovare un cammino minimo da u a v , è una variante del primo problema.
4. *Problema dei cammini minimi fra tutte le coppie di vertici(all-pairs)*: trovare un cammino minimo da u a v per ogni coppia di vertici.

Di seguito affronteremo il primo e il quarto.

Proprietà dei cammini minimi

Sottostruttura ottima

Teorema: Dati

- un grafo orientato $G = (V, E)$
- la funzione peso $w : E \rightarrow R$
- sia $p = (v_0, v_1, \dots, v_k)$ un cammino minimo dal vertice v_0 al v_k
per qualsiasi i e j tali che $0 \leq i \leq j \leq k$ sia p_{ij} un sotto-cammino di p dal vertice v_i al vertice v_j allora P_{ij} è un cammino minimo da v_i a v_j

Dimostrazione: Se scomponiamo il cammino p in

$$v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$$

abbiamo $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$.

Supponiamo adesso che ci sia un cammino p'_{ij} da v_i a v_j con peso $w(p'_{ij}) < w(p_{ij})$. Allora $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p'_{ij}} v_j \xrightarrow{p_{jk}} v_k$ è un cammino da v_0 a v_k il cui peso è minore di $w(p)$ che contraddice l'ipotesi che p sia un cammino minimo da v_0 a v_k

Archi di peso negativo e cicli

All'interno del nostro grafo potrebbero essere contenuti dei cammini di peso negativo, o anche dei cicli:

- Quando *non sono presenti* dei cicli di peso complessivo negativo raggiungibili è sempre possibile definire $\delta(s, v)$
- Quando *sono presenti* dei cicli di peso complessivo negativo non sarà possibile individuare di $\delta(s, v)$ in quanto sarà sempre possibile diminuirlo effettuando un numero indefinito di passi all'interno del ciclo. In tal caso diremo che $\delta(s, v) = -\infty$

Rilassamento dei cammini minimi

Il processo di *rilassamento di un arco* consiste nel verificare se passando per u , è possibile migliorare il cammino minimo per v da s precedentemente trovato. Nelle nostre implementazioni avremo un array d dove $d[v]$ è peso del cammino da v ad s , con ogni passo di rilassamento possiamo ridurre questo valore.

RELAX(u, v, w):

if($d[v] > d[u] + w(u, v)$)

$$d[v] = d[u] + w(u, v)$$

Alla creazione del nostro grafo tutti i valori del nostro array d vengono inizializzati a $+\infty$.

Proprietà dei cammini minimi

1. Disuguaglianza triangolare

Per qualsiasi arco $(u, v) \in E$, si ha $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

2. Proprietà del limite superiore

Per tutti i vertici $v \in V$, si ha sempre $v.d \geq \delta(s, v)$ e, una volta che il limite superiore $v.d$ assume il valore $\delta(s, v)$, esso non cambia più.

3. Proprietà dell'assenza di un cammino

Se non c'è un cammino da s a v , allora si ha sempre $v.d = \delta(s, v) = \infty$.

4. Proprietà della convergenza

Se $s \leadsto u \rightarrow v$ è un cammino minimo in G per qualche $u, v \in V$ e se $u.d = \delta(s, u)$ in un istante qualsiasi prima del rilassamento dell'arco (u, v) , allora $v.d = \delta(s, v)$ in tutti gli istanti successivi.

5. Proprietà del rilassamento del cammino

Se $p = \langle v_0, v_1, \dots, v_k \rangle$ è un cammino minimo da $s = v_0$ a v_k e gli archi di p vengono rilassati nell'ordine $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$, allora $v_k.d = \delta(s, v_k)$. Questa proprietà è soddisfatta indipendentemente da altri passi di rilassamento effettuati.

6. Proprietà del sotto grafo dei predecessori

Una volta che $v.d = \delta(s, v)$ per ogni $v \in V$, il sotto grafo dei predecessori è un albero di cammini minimi radicato in s .

Single-Source Shortest Path in un DAG

Generic SSSP

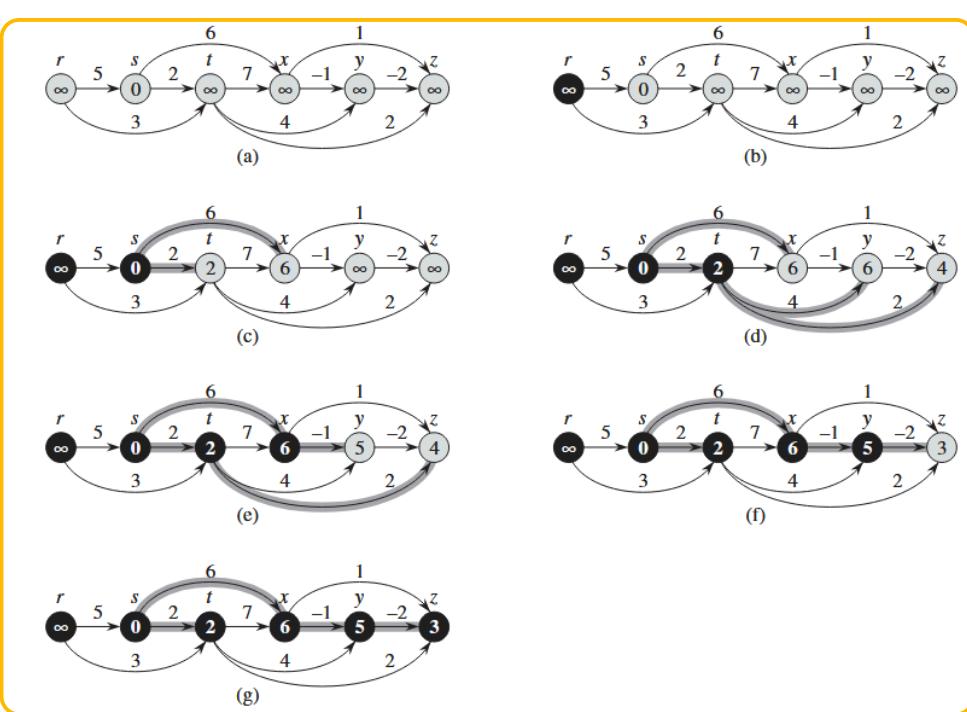
Di seguito vediamo il primo algoritmo per la ricerca di un cammino minimo ovvero il *Single-Source Shortest Path* che risolve il primo problema dei cammini minimi.

```
Generic-SSSP(G, s)
    d = new array(len(V))
    foreach v in V do
        d[v] = +infinito
    d[s] = 0
    while esiste (u, v) in E tale che d[u] + w(u, v) < d[v]:
        RELAX(u, v)
    return d
```

SSSP in un grafo orientato aciclico (DAG)

Rilassando gli archi di un dag (Directed Acyclic Graph) pesato $G = (V, E)$ secondo un ordine topologico dei suoi vertici è possibile calcolare i cammini minimi da una sorgente unica nel tempo $\Theta(V + E)$. L'algoritmo inizia ordinando topologicamente il dag, se esiste un cammino dal vertice u al vertice v , allora u precede v nell'ordine topologico. Effettuiamo un passaggio sui vertici secondo l'ordine topologico. Durante l'elaborazione vengono rilassati tutti gli archi che escono dal vertice

```
DAG-SHORTEST-PATHS(G, w, s)
    U = getTopologicalOrder(G);
    foreach v in U:
        foreach u in G.Adj[v]
            RELAX(u, v, w)
```



Algoritmo di Bellman-Ford

Definizione

Questo algoritmo si basa su un cavillo logico molto semplice, visto che non so quali archi rilassare li rilasso sempre tutti. Li rilasso $n - 1$ volte dove n è la lunghezza del cammino più lungo (ovvero il numero di nodi), dopo aver fatto $n - 1$ rilassamenti degli archi sono sicuro che anche il cammino più lungo sarà rilassato. Questo algoritmo a differenza di tutti gli altri è in grado di capire se l'output che sta ritornando è valido, aggiungendo infatti un rilassamento ulteriore dopo le $n - 1$ passate verifichiamo se il nostro grafo contiene dei cicli negativi.

Implementazione

```
Bellman-Ford(G, s, w)
    // 1. Inizializzazione
    d = new Array(len(V))
    for each v ∈ V do
        d[v] = +∞
        π[v] = NIL
    d[s] = 0

    // 2. Rilassamento iterativo degli archi
    for i ← 1 to |V|-1 do
        for each (u, v) ∈ E do
            relax(u, v, w)

    // 3. Controllo dei cicli di peso negativo
    for each (u, v) ∈ E do
        if (d[v] > d[u] + w(u, v)) then
            return false

    return d
```

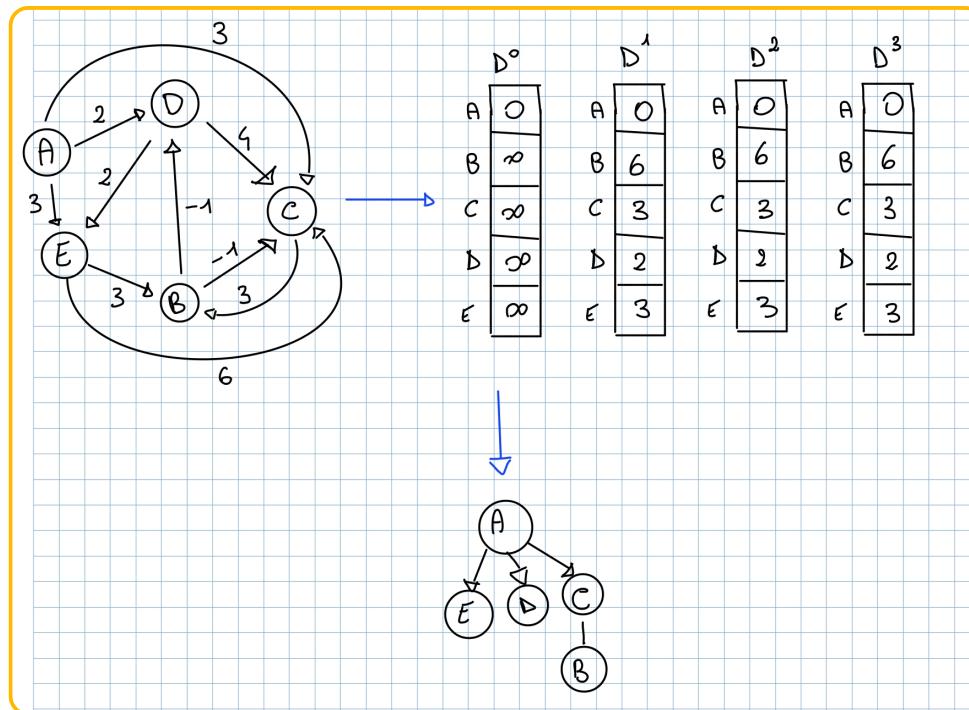
La parte iniziale viene utilizzata per inizializzare gli array d e π che rappresentano:

- $d[v]$ rappresenta la distanza di v da s
- $\pi[v]$ è il nodo padre di v nell'albero dei cammini minimi

Il punto tre verifica la validità del risultato, stiamo controllando se si possono effettuare altri relax, se questo succede vuol dire che il nostro grafo contiene dei cicli negativi, e quindi la ricerca di un cammino minimo non può esistere

Esempio

Di seguito un esempio di come dovremmo implementare questo algoritmo in sede di esame



Complessità

La complessità di questo algoritmo cambia in base al tipo di implementazione utilizzata per rappresentare i grafi:

- *Lista di adiacenza*: $O(VE)$
- *Matrice di adiacenza*: $O(V^3)$

Algoritmo di Dijkstra

Definizione

Per poter funzionare questo algoritmo ha bisogno che tutti i pesi nel grafo siano positivi. Questo algoritmo si basa sull'idea che se troviamo un nodo convergente (quindi completamente rilassato) possiamo rilassare i tutti i suoi archi uscenti. Dati due insiemi:

- *insieme dei nodi in convergenza*
- *insieme dei nodi non in convergenza*

Un nodo convergente sarà sicuramente il nodo con il valore più piccolo nell'insieme dei nodi non in convergenza questo poiché tutti i pesi degli archi sono positivi (o nulli), non esiste alcun cammino alternativo che, passando per altri nodi non ancora visitati, possa "tornare indietro" e ridurre ulteriormente la distanza di quel nodo specifico.

Implementazione

```

Dijkstra(G, s, w)
  d = new Array(len(V))
  for each v ∈ V do
    d[v] = +∞
    π[v] = NIL
  d[s] = 0
  
```

```

 $Q \leftarrow \text{build-min-heap}(V)$ 

while  $Q \neq \emptyset$  do
     $v \leftarrow \text{extract-min}(Q)$ 
    foreach  $u \in \text{ADJ}(v)$ 
        if  $d[u] > d[v] + w(v, u)$  then
             $\text{decrease-key}(Q, u, d[v] + w(v, u))$ 

```

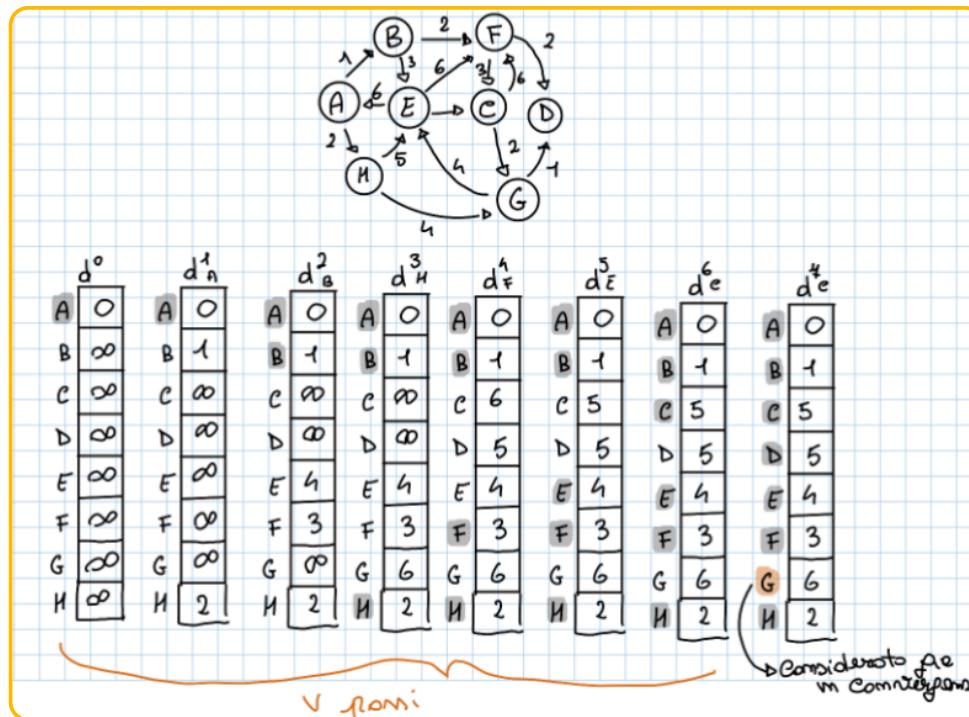
La parte iniziale viene utilizzata per inizializzare gli array d e π che rappresentano:

- $d[v]$ rappresenta la distanza di v da s
- $\pi[v]$ è il nodo padre di v nell'albero dei cammini minimi

Quello che fa il resto del algoritmo consiste nel creare un min-heap rispetto al valore che hanno i nodi in d e poi rilassare tutti gli archi uscenti dai minimi della coda fino a quando la coda non è vuota

Esempio

Di seguito un esempio di come dovremmo implementare questo algoritmo in sede di esame



Complessità

Questa analisi deve essere divisa in varie sezioni:

- *Inizializzazione*: scorre tutti i nodi quindi ha un costo pari a: $O(V)$
- *Costruzione min-heap*: la costruzione del min-heap è lineare al numero di elementi quindi nel nostro caso avremo un costo pari a: $O(V)$
- *While principale*: la funzione extractMin di un min-heap come è noto impiega tempo $O(\log n)$ nel nostro caso viene eseguita V volte, quindi si avrà un costo pari a: $O(V \times \log V)$
- *Foreach interno*: la funzione decreaseKey di un min-heap come è noto impiega tempo $O(\log n)$, viene richiamata ogni volta che troviamo un cammino più breve, quindi nel caso peggiore avviene per tutti gli archi quindi si avrà un costo pari a: $O(E \times \log V)$

Quindi il costo complessivo di questo algoritmo sarà:

$$O((V \times \log V) + (E \times \log V)) \rightarrow O((V + E) \times \log V)$$

Conclusioni sul problema dei cammini minimi da una sorgente

Per la risoluzione di questo problema abbiamo presentato tre algoritmi:

- Single-Source Shortest Path
- Bellam-Ford
- Dijkstra

Sia SSSP che Dijkstra hanno delle condizioni sul grafo per poter funzionare, questo li porta ad avere una complessità migliore rispetto a Bellam-Ford che resta però il più generalista.

Utilizzi molteplici

Questi algoritmi studiati principalmente per la risoluzione del problema single source possono essere usati anche per risolvere il problema di trovare il cammino minimo tra tutte le coppie però la complessità aumenta:

- Bellam-Ford: $O(V^4)$
- Dijkstra: $O(V^3 \log V)$
- DAG: $O(V^3)$

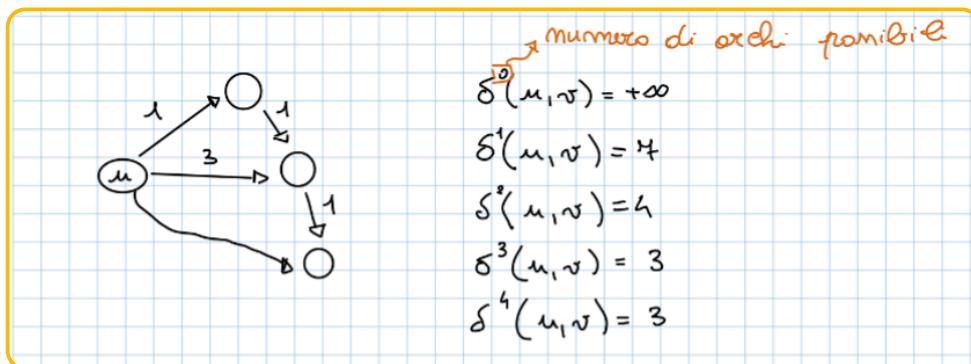
Programmazione dinamica per la risoluzione dei cammini minimi tra tutte le coppie

Definizione

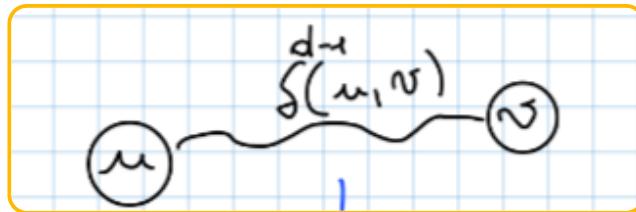
In questo algoritmo la dimensione del problema viene identificata come: *il numero di archi che possono essere presenti in un cammino minimo* ovvero la *lunghezza massima di un cammino minimo* che sarebbe pari a $V - 1$, la dimensione del problema verrà di seguito identificata con d . Indicheremo con

$$\delta^d(u, v)$$

il peso del cammino minimo da u a v usando d archi.

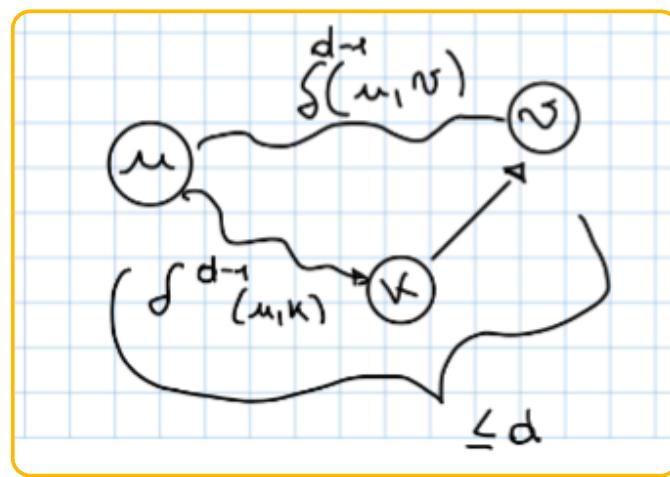


Ipotizzando di avere un cammino minimo da due generici vertici u e v al passo $d - 1$ avremo questa cosa:



se al passo d (quindi con più archi disponibili) scopriamo di avere un altro percorso che passa dal nodo k , quindi

siamo in una situazione del tipo:



come facciamo a capire quale dei due percorsi ha il peso minore, lo facciamo in questo modo:

```
if ( $\delta^{d-1}(u, k) + w(k, v) < \delta^{d-1}(u, v)$ )
    then  $\delta^d(u, v) = \delta^{d-1}(u, k) + w(k, v)$ 
else  $\delta^d(u, v) = \delta^{d-1}(u, v)$ 
```

Come individuiamo il nostro K ? Non lo individuo, provo tutti i k ad ogni passaggio. Per ogni d andiamo a creare quindi una matrice l definita così:

$$l_{i,j}^d = \begin{cases} w_{i,j} & \text{se } d = 1 \\ \min_{1 \leq k \leq n} (l_{i,j}^{d-1}, l_{i,k}^{d-1} + w_{k,j}) & \text{se } d > 1 \end{cases}$$

il primo termine del \min è contenuto nel secondo quindi si può elidere

$$l_{i,j}^d = \begin{cases} w_{i,j} & \text{se } d = 1 \\ \min_{1 \leq k \leq n} (l_{i,k}^{d-1} + w_{k,j}) & \text{se } d > 1 \end{cases}$$

Implementazione

Dalla funzione ricorsiva definita nel punto precedente nasce il seguente algoritmo

```
Extend-APSP(L, W)
n = L.rows
L_new = new Matrix(n, n)
for i = 1 to n do
    for j = 1 to n do
        L_new[i, j] = infinity
        for k = 1 to n do
            if (L[i, k] + W[k, j] < L_new[i, j])
                L_new[i, j] = L[i, k] + W[k, j]
return L_new
```

che viene richiamato dalla seguente funzione:

```
APSP(W)
n = m
L_1 = W
for d = 2 to n - 1 do
    L_d = EXTEND-APSP(L_{d-1}, W)
return L_{n-1}
```

La funzione $APSP$ ha complessità $O(V^4)$ quindi non siamo riusciti a fare meglio di Bellam-Ford.

Miglioramento

Prendiamo in esame un algoritmo che moltiplica le matrici, se moltiplichiamo una matrice per stessa non è necessario fare la moltiplicazione n volte se nel passo precedente è stato fatto trovato il risultato per la moltiplicazione $n/2$ volte.

$$\begin{aligned} A^1 &= A \\ A^2 &= A \times A \\ A^3 &= A^2 \cdot A \\ A^4 &= A^2 \cdot A^2 \\ A^8 &= A^4 \times A^4 \\ A^{16} &= A^8 \times A^8 \\ \dots \end{aligned}$$

Questa cosa è fattibile anche nel nostro algoritmo, quindi la ora la matrice l viene definita come:

$$l_{i,j}^d = \begin{cases} w_{i,j} & \text{se } d = 1 \\ \min_{1 \leq k \leq n} (l_{i,k}^{d/2} + l_{k,j}^{d/2}) & \text{se } d > 1 \end{cases}$$

Da questo nasce la una nuova implementazione della funzione *APSP*:

```
Fast-APSP(W)
    n = m // numero di vertici
    L^1 = W
    d = 1
    while d < n - 1 do
        L^{2d} = EXTEND-APSP(L^d, L^d)
        d = 2d
    return L^d
```

Se inseriamo un iterazione in più otteniamo lo stesso risultato del ciclo finale di bellam-ford (la verifica di cicli negativi)

Complessità

Il Fast-APSP ha complessità $O(V^3 \log V)$ che è un miglioramento lieve rispetto a quello di bellam-ford

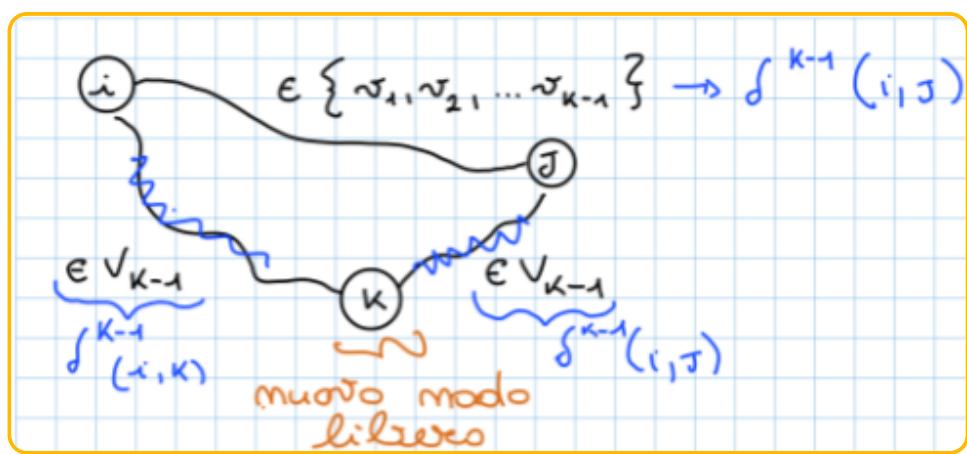
Floyd-Warshall

Definizione

D'ora in avanti indicheremo con V_K il sottoinsieme di V formato dai primi K nodi

$$\begin{aligned} V_k &= \{v_1, v_2, v_3, \dots, v_k\} \\ V_0 &= \{\} \\ V_1 &= \{v_1\} \\ V_2 &= \{v_1, v_2\} \\ &\vdots \\ V_n &= V \end{aligned}$$

La dimensione del problema al passo k è definita *dalla possibilità di includere il k -esimo nodo come potenziale punto di passaggio per tutte le coppie (u,v)* .



considerando i due cammini evidenziati in blu ottengo un cammino che passa da k , ora bisogna capire quale dei due cammini è il migliore. E lo facciamo usando una matrice

$$D^K[i, j] = \delta^K(i, j) = \begin{cases} w[i, j] & \text{se } k = 0 \\ \min(D^{k-1}[i, j], D^{k-1}[i, k] + D^{k-1}[k, j]) & \text{se } k > 0 \end{cases}$$

è molto simile al caso di programmazione dinamica che abbiamo fatto precedentemente, però in questo caso non dobbiamo provare tutte le k ad ogni passaggio perché k è già definita

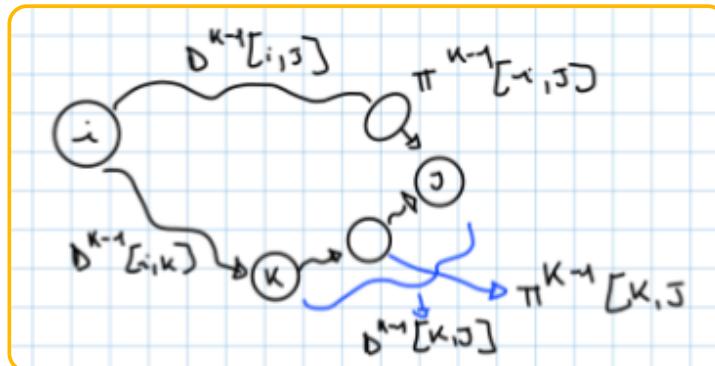
Implementazione

Dalla definizione di matrice precedentemente presentata otteniamo il seguente algoritmo:

```
FLOYD-WARSHALL(W)
    n = m // numero di vertici
    D^0 = W
    for k = 1 to n do
        D^k = new Matrix(n, n)
        for i = 1 to n do
            for j = 1 to n do
                // Se il cammino attraverso k è più breve, aggiorna
                if (D^{k-1}[i, j] > D^{k-1}[i, k] + D^{k-1}[k, j])
                    D^k[i, j] = D^{k-1}[i, k] + D^{k-1}[k, j]
                else
                    D^k[i, j] = D^{k-1}[i, j]
    return D^n
```

Inoltre vogliamo creare un albero dei cammini minimi, per fare ciò dobbiamo tenere traccia dei predecessori dei nostri nodi, e lo facciamo usando una seconda matrice definita in questo modo:

$$\pi^K[i, j] = \begin{cases} i & \text{se } [i, j] \in E \\ \text{null} & \text{altrimenti} \end{cases}$$



Di conseguenza il nostro algoritmo diventa:

```

Floyd-Warshall(W)
n = m // numero di vertici
D^0 = W

// Inizializzazione della matrice dei predecessori Pi
Pi^0 = new Matrix(n, n)
for i = 1 to n do
    for j = 1 to n do
        if (i != j and W[i, j] < infinity)
            Pi^0[i, j] = i
        else
            Pi^0[i, j] = null

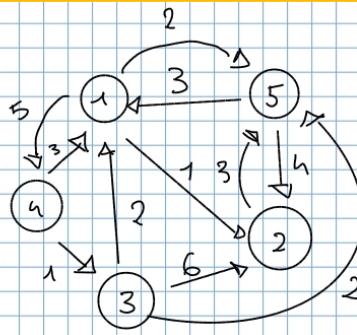
// Ciclo principale dell'algoritmo
for k = 1 to n do
    D^k = new Matrix(n, n)
    Pi^k = new Matrix(n, n)
    for i = 1 to n do
        for j = 1 to n do
            // Manteniamo i valori precedenti come base
            D^k[i, j] = D^{k-1}[i, j]
            Pi^k[i, j] = Pi^{k-1}[i, j]

            // Controllo se il passaggio per il nodo k migliora il cammino
            if (D^{k-1}[i, j] > D^{k-1}[i, k] + D^{k-1}[k, j])
                D^k[i, j] = D^{k-1}[i, k] + D^{k-1}[k, j]
                Pi^k[i, j] = Pi^{k-1}[k, j] // Aggiorna il predecessore

return D^n, Pi^n

```

Esempio



$W = D^0$

	1	2	3	4	5
1	0	1	∞	5	2
2	∞	0	∞	∞	3
3	2	6	0	∞	2
4	3	∞	1	0	∞
5	3	4	∞	∞	0

$W = D^1$

	1	2	3	4	5
1	0	1	∞	5	2
2	∞	0	∞	∞	3
3	2	3	0	4	2
4	3	4	1	0	5
5	3	4	∞	8	0

$W = D^2$

	1	2	3	4	5
1	0	1	∞	5	2
2	∞	0	∞	∞	3
3	2	3	0	4	2
4	3	4	1	0	5
5	3	4	∞	8	0

$W = D^3$

	1	2	3	4	5
1	0	1	∞	5	2
2	∞	0	∞	∞	3
3	2	3	0	4	2
4	3	4	1	0	3
5	3	4	∞	8	0

$W = D^4$

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

$W = D^5$

	1	2	3	4	5
1	0	1	6	5	2
2	6	0	12	11	3
3	2	3	0	4	2
4	3	4	1	0	3
5	3	4	5	8	0

matrice con i cammini minimi

Ci sono delle euristiche da seguire per ridurre il carico di lavoro:

in generale devo fare $\text{len}(V) + 1$ matrici (la prima è quella di adiacenza)

1. Riscrivo la riga e la colonna k perché sicuramente non cambia
2. Riscrivo la diagonale principale perché è sicuramente sempre 0 (non capita mai di avere dei cappi)
3. Prendo la colonna k e se in una delle celle che la compongono ci trovo un ∞ riscrivo tutta la riga
4. Prendo la riga k e se in una delle celle che la compongono ci trovo un ∞ riscrivo tutta la colonna

Complessità

La complessità di questo algoritmo è $O(V^3)$ ottima rispetto a tutto quello che abbiamo trovato fino ad adesso. In generale si preferisce questo algoritmo anche quando servono i cammini minimi da una singola sorgente, perché ha la stessa complessità di bellam-ford.