Analisi e Progettazione Algoritmi

Adrian Castro, Ilaria Battiston *

Ottobre 2018

*Slides di Mauri

Contents

1 Introduzione

1.1 Chiarimenti sul corso

1. Ricevimento U14 1010 1° piano;
2. zandron@disco.unimib.it;
3. Modalità d'esame:
   (a) Scritto: esercizi + teoria;
   (b) Verbalizzazione normale;
   (c) Compitini:
      i. 1o: esercizi;
      ii. 2o: esercizi + teoria.

1.2 A grandi linee

1.2.1 Programmazione dinamica

La programmazione dinamica è un metodo di risoluzione dei problemi che consiste nel trovare sotto-informazioni che poi verranno riutilizzate con il tempo. Il problema viene spezzato in parti, e si cerca di trovare una sotto-informazione che può essere sfruttata quando necessario. Efficiente in termini di spazio e tempo.

Alcuni algoritmi hanno una soluzione intuitiva, ma che richiedono di ricalcolare tante volte la stessa cosa. Una soluzione può essere usare una **cache**, memoria dove vengono salvati i valori calcolati precedentemente. A questo punto, però, l'algoritmo è meno efficiente dal punto di vista dello **spazio**. Bisogna quindi capire in quali casi è opportuno attuare una tecnica del genere.

La programmazione dinamica viene usata nei problemi di ottimizzazione e di decisione. Gli algoritmi di decisione devono saper rispondere a ogni tipo di input (es. esiste un cammino che collega due nodi?).

Un altro problema è la ricerca: dato un input x, determinare una soluzione S che rispetti le caratteristiche del problema. Deve esistere S tale che esista un collegamento $(x,\ S)$.

L'ottimizzazione associa a ogni soluzione il relativo costo e cerca la miglior soluzione possibile in base al costo minimo: $S^* \mid C(S^*) \leq C(S)$.
Bisogna considerare anche che alcune soluzioni non sono ammissibili.

1.2.2 Programmazione reedy

Il tipo di programmazione **greedy** (o **goloso**), viene usata per determinare la soluzione migliore tramite una serie di calcoli effettuati localmente, per esempio previsioni in base ai dati correnti. Non si conosce l'andamento dei dati nel futuro.

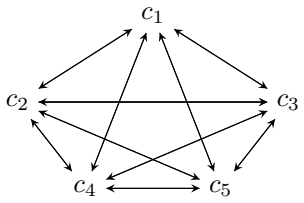
Una tecnica di programmazione **greedy** si basa sulla situazione attuale per capire cosa fare successivamente.

Esempio: problema del *travelling salesman*. La tecnica greedy continua a ridurre le possibilità di scelta, più avanti si va nell'algoritmo. Ciò significa che le scelte attuali di viaggio possono sembrare poco costose, ma poi magari ci si ritrova con le ultime scelte che hanno spese insostenibili.

In casi di questo tipo, un algoritmo **greedy** non è la scelta più adatta. Possiamo quindi fare la seguente osservazione:

Definizione 1.1: Algoritmo Greedy

Gli algoritmi di tipo greedy funzionano al meglio se tutti i pesi (o valori) delle scelte sono uguali a 1.



Alcuni algoritmi sui grafi implementabili con uno dei due metodi di programmazione sono BFS, DFS, cammini minimi, problemi di flusso. Quando nessun algoritmo è in grado di risolvere un problema in tempo accettabile (non esponenziale) si parla di NP-completezza.

2 Ripasso di algoritmi

2.1 Risoluzione di algoritmi e tempi di esecuzione

Un problema risoluzione di algoritmi rientra sempre in uno di tre casi:

1. Problema risolvibile in modo efficiente;
2. Problema risolvibile in modo non efficiente;
3. Problema non risolvibile.

Il metodo può essere iterativo o ricorsivo. Con il modo iterativo, una serie di operazioni sono ripetute più volte all'interno della stessa funzione, e lo sviluppo dell'algoritmo è intuitivo. Per ogni possibile input, la risposta ottenuta alla fine della sequenza di istruzioni dev'essere corretta. Alcuni esempi di algoritmi iterativi sono l'ordinamento tramite selection sort, insertion sort e bubble sort; altri sono la **ricerca dicotomica**, **ricerca del massimo e del minimo**.

Il tempo di esecuzione viene misurato in base al numero di istruzioni, utilizzando limiti asintotici. Valutare il tempo di esecuzione significa capire (asintoticamente) quante istruzioni vengono eseguite

con un input di dimensione n, cioè

$$T(n), n = |x|, x = input$$

Ogni algoritmo ha tre modi di rappresentarne il tempo, che indicano l'ordine di complessità di $f(n)$:

- Caso migliore (numero minimo di istruzioni), $t(n)$ oppure $\Omega(f(n))$, che significa che la funzione è limitata inferiormente;
- Caso peggiore (numero massimo di istruzioni, garanzia sul tempo massimo), $T(n)$ oppure $O(f(n))$, che significa che la funzione è limitata superiormente;
- Caso medio (tempo medio di esecuzione in base al caso possibile più frequente), $T_m(n)$ oppure $\Theta(f(n))$, che significa che la funzione è compresa tra due limiti.

2.2 Algoritmi iterativi

Selection sort è l'algoritmo di sorting più semplice, che consiste nella ricerca lineare del minimo in un ciclo for.

Algorithm 1 Selection sort iterativo

```
function SELECTION_SORT(A)
    for j ← 1 to (LENGTH(A) - 1) step 1 do
        for i ← j to LENGTH(A) step 1 do
            min ~ A[i]
        end for
        SCAMBIA(A, j, min)
    end for
end function
```

Il tempo di esecuzione sarebbe

$$\sum_{i=1}^{n} i + \sum_{i=1}^{n} i + 3(n-1)$$

approssimabile a $\Theta(n^2)$.

Insertion sort è un algoritmo meglio funzionante rispetto a selection sort, rappresentabile come l'ordinamento di un mazzo di carte di cui k ordinate, scoprendo una carta per volta. Ogni carta viene controllata e riposizionata in modo da avere un array di carte ordinate.

Il tempo di esecuzione dipende dal numero di volte in cui $while$ è vero, cioè

$$4n + 3\sum_{i=2}^{n} t_w i$$

(con $t_w i$ numero di volte che viene eseguito $while$).

1. Caso Migliore: array già ordinato, $\Theta(n)$.
2. Caso Peggiore: array ordinato al contrario, $O(n^2)$
3. Caso Medio: l'ordinamento richiede metà iterazioni di $while$, approssimato a $\Theta(n^2)$

Algorithm 2 Insertion sort iterativo

```
function INSERTION_SORT(A)
    for j ← 2 to LENGTH(A) step 1 do
        carta ← A[i]
        while A[j] > carta ∧ j > 0 do
            A[j + 1] ← A[j]
            j ← j − 1
            A[j + 1] ← carta
        end while
    end for
end function
```

2.3 Algoritmi ricorsivi

Nonostante selection sort e bubble sort abbiano tempi computazionali generamente minori rispetto a selection sort, essi non sono ancora ottimizzati per input lunghi.

Gli algoritmi ricorsivi contengono funzioni che chiamano loro stesse con un input di dimensione minore rispetto all'originale, continuando a diminuirne le dimensioni fino ad arrivare al caso base (semplice).

Algorithm 3 Ricerca del massimo ricorsivo

```
function MAX_RIC(A, i, n)
    if i == n then
        return i
    else
        MAX_RIC(A[], i+1, n)
        if A[i] > A[Rp] then
            return i
        else
            return Rp
        end if
    end if
end function
```

Il tempo di esecuzione può essere descritto in questo modo:

$$T(n) = \begin{cases} 2 & se\ n = 1 \\ 3 + T(n-1) & se\ n > 1 \end{cases}$$

L'equazione di ricorrenza viene risolta con tre possibili metodi:

1. Metodo iterativo (dell'albero);
2. Metodo di induzione;
3. Metodo dell'esperto.

Utilizzando questo metodo è possibile arrivare a un tempo asintotico di $\Theta(n)$.

Un altro algoritmo ricorsivo è la ricerca del k-esimo elemento della sequenza di Fibonacci. L'implementazione standard richiama la funzione su $(n-1)$ e $(n-2)$. Il problema di questo algoritmo è l'elevata complessità computazionale, la quale cresce esponenzialmente (al contrario di altri come il fattoriale).
Il metodo più efficiente per Fibonacci è tramite la programmazione dinamica, salvando i risultati intermedi in memoria.

Merge sort è un popolare algoritmo di algoritmi ricorsivo che sfrutta la tecnica del divide-et-impera: divide il problema in uno o più sotto-problemi, usa la ricorsione per risolvere i sotto-problemi e ne combina le soluzioni.

Algorithm 4 Merge sort ricorsivo

```
function MS(A[], I, F)
    m ← (I + F)/2
    MS(A[], I, m)
    MS(A[], m + 1, F)
    MERGE(A[], I, m, F)
end function
```

Questo algoritmo ha una parte iterativa (divide e combina) e una parte ricorsiva (impera) le quali hanno un tempo computazionale approssimato al tempo della ricorsione, essendo esso maggiore.

$$T(n) = \Theta(nlogn)$$

2.4 Riassumendo

La soluzione a un problema può essere sotto forma di elemento, insieme di elementi o operazioni su un insieme di elementi. Un algoritmo iterativo usa la strategia bottom-up, mentre uno ricorsivo usa top-down.

Un algoritmo iterativo è efficiente per risolvere tutti i problemi per ottenere la soluzione, perché in questo modo si evitano tutte le chiamate sullo stack.
Un algoritmo ricorsivo è più efficiente se può terminare subito non appena la soluzione viene trovata, evitando di risolvere tutti i sottoproblemi.

3 Programmazione dinamica e grafi

La programmazione dinamica viene introdotta da Bellman come processo per trovare le soluzioni migliori, una dopo l'altra (problemi di natura ricorsiva). Programmazione prende il significato di pianificazione per l'azione da produrre.

Il problema degli algoritmi ricorsivi è la complessità esponenziale (lo stack va a puttane). Lo scopo della programmazione dinamica è aggirare questo problema salvando le soluzioni dei sotto-problemi con una strategia bottom-up (invece che top-down).

La struttura dati più comunemente usata per il salvataggio dei risultati è l'array, talvolta n-dimensionale.
La parte più importante dell'algoritmo è l'equazione di ricorrenza, che deve permettere di selezionare le soluzioni intermedie necessarie per calcolare quella finale.

3.1 Sequenza di Fibonacci

La sequenza di Fibonacci è un classico algoritmo la cui classica implementazione ricorsiva ha un tempo computazionale esponenziale (ogni chiamata di funzione genera altre due chiamate).

Algorithm 5 Fibonacci ricorsivo

```
function FIB_RIC(n)
    if (n == 1) ∨ (n == 2) then
        return 1
    else
        return FIB_RIC(n − 1) + FIB_RIC(n − 2)
    end if
end function
```

3.1.1 Alternative alla ricorsione

Per evitare il numero esponenziale di chiamate di funzione, si ricorre a un algoritmo iterativo il quale salva in un array ogni risultato della sequenza e usa i numeri nelle due posizioni precedenti per calcolare un nuovo valore.

Algorithm 6 Fibonacci iterativo

```
function FIB_IT(n)
    F[1] ← 1
    F[2] ← 2
    for i ← 3 to n step 1 do
        F[i] ← F[i − 1] + F[i − 2]
    end for
end function
```

Per risolvere il problema dell'n-esimo numero della sequenza di Fibonacci è possibile utilizzare teoremi di algebra lineare. Si ha che

$$F_n = F_{n-1} + F_{n-2}$$

Similmente,

$$F_{n-1} = Fn - 1 + 0 \cdot F_{n-2}$$

Il sistema ottenuto risulta

$$\begin{cases} F_n = F_{n-1} + F_{n-2} \\ F_{n-1} = Fn - 1 + 0 \end{cases}$$

Sotto forma di matrice,

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix}$$

In altre parole, il valore F_n può essere ottenuto moltiplicando il vettore che ha come componenti F_{n-1} e F_{n-2} per la matrice numerica. Ogni numero quindi può essere usato per calcolare quello successivo, il quale verrà salvato a sua volta.

Questo metodo di risoluzione è basato sugli autovalori e gli autovettori (teorema di Perron-Frobenius). Risolvendo l'equazione $Ax = \lambda x$ sotto forma di matrice, si ha che gli autovalori di $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ sono $\lambda_1 = \frac{1+\sqrt{5}}{2}$ ($\lambda_\heartsuit$, cuoricino perché è il più grande tra i due autovalori in valore assoluto) e $\lambda_2 = \frac{1-\sqrt{5}}{2}$
Allora,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = P \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} p^{-1}$$

dove $P = \begin{pmatrix} \lambda_1 & \lambda_2 \\ 1 & 1 \end{pmatrix}$ con $\begin{pmatrix} \lambda_1 \\ 1 \end{pmatrix}$ autovettori corrispondenti a λ_i.

Allora,

$$\begin{aligned} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k-2} &= P \begin{pmatrix} \lambda_1^{k-2} & 0 \\ 0 & \lambda_2^{k-2} \end{pmatrix} p^{-1} \\ &= \begin{pmatrix} \lambda_1 & \lambda_2 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} \lambda_1^{k-2} & 0 \\ 0 & \lambda_2^{k-2} \end{pmatrix} \frac{1}{\sqrt{5}} \begin{pmatrix} 1 & -\lambda_2 \\ -1 & \lambda_2 \end{pmatrix} \end{aligned}$$

La formula finale è

$$y_k = \frac{1}{\sqrt{5}}(\lambda_1^k \lambda_2^k)$$

Complessità finale:

$$T(n) = \frac{1}{\sqrt{5}}(\lambda_{1\heartsuit}^n - \lambda_2^n) \; circa$$

$$\Theta(\lambda_1^n) = \Theta((\frac{1+\sqrt{5}}{2})^n)$$

Riscriviamo il problema: calcolare l'n-esimo numero della sequenza di Fibonacci. Possiamo spezzare il problema in sottoproblemi:

1. Calcolare il numero di Fibonacci con $n = 1$;
2. Calcolare il numero di Fibonacci con $n = 2$;
3. ...
4. Calcolare il numero di Fibonacci con $n = n$.

4 Ripasso strutture dati

4.1 Array

Gli array sono strutture statiche con memoria allocata a priori. Sono veloci perché tutta la memoria viene allocata nello stesso blocco.

4.2 Liste dinamiche

Le liste dinamiche hanno con memoria dinamica che varia durante l'esecuzione (la memoria non è allocata nella stessa locazione spaziale, quindi sono più lente).

Esempi di liste dinamiche sono le linked list: hanno una *head* che punta a una casella contenente *key* e *next*, il quale a sua volta punta all'elemento successivo. Seguendo i puntatori si può visitare la lista.
Il tempo di accesso ai singoli dati non è costante, ma dipende dalla lunghezza della lista (inefficiente).
La gestione dei puntatori è a carico del programmatore, quindi possono esserci errori anche gravi.

Può essere utile immagazzinare anche il puntatore al valore precedente *previous*, per operazioni come la cancellazione. In questo caso la struttura dati si definisce lista doppia.

Un altro puntatore statico è *tail*, all'ultima casella. Serve per aggiungere degli elementi in ordine senza dover scorrere tutta la lista e va aggiornato dopo ogni inserimento.

Se l'ultima casella punta alla prima e la prima ha come *previous* l'ultima la lista è doppia circolare, e non è necessario avere *tail*. A volte si definisce un valore sentinella, una casella "finta" per gestire alcuni sotto-casi particolari (lista vuota, ultimo elemento, ...).

4.3 Alberi radicati

Gli alberi radicati (generalmente binari) hanno puntatori a $left$ e $right$, che indicano i rami del nodo. Il campo *key* contiene il valore.

4.4 Alberi binari di ricerca

Gli alberi binari di ricerca (BST) sono particolari alberi che rispettano la seguente proprietà: nel sottoalbero sinistro a ogni nodo ci sono solo valori inferiori, e nel sottoalbero destro ci sono solo valori superiori. Il tempo computazionale delle operazioni sui BST è di solito logaritmico.

4.5 Heap

Un heap è un array rappresentabile come albero. Tutti i livelli sono riempiti tranne l'ultimo, che viene riempito da sinistra verso destra. L'ordinamento di un heap si effettua con la visita in-order. Eventualmente viene aggiunto un puntatore *parent* al padre di ogni foglia.

Il max heap è un tipo particolare di heap dove la radice corrisponde al massimo.

4.6 Pile

Le pile (stack) possono essere implementate con array e liste. Sono gestite con politica LIFO, l'ultimo elemento è il primo a essere rimosso. Le operazioni su pile sono *push* e *pop*, e *stackempty* per evitare errori di overflow/underflow. Per leggere l'elemento in cima si usa *top*.

4.7 Code

Le code (queue) possono essere implementate con array e liste. Sono gestite con politica FIFO, il primo elemento è il primo a essere rimosso. Le operazioni su code sono *enqueue* e *dequeue*, *emptyqueue* per capire se la lista è vuota.

5 Sottosequenze

5.1 Sottosequenza singola crescente (LGS)

Una sottosequenza di una stringa $x = < x_1, x_2, \ldots, x_n >$ è un insieme di indici $i_1, i_2, \ldots i_k$ con $k \leq n$ con indici k strettamente crescenti. Gli indici non devono necessariamente essere consecutivi.

Esempio di sottosequenza con una stringa:

$$\begin{array}{l} \quad 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 8 \\ X = < \mathrm{C}, \mathrm{C}, \mathrm{A}, \mathrm{B}, \mathrm{B}, \mathrm{C}, \mathrm{A}, \mathrm{B}, \mathrm{D} > \end{array}$$

$Z = < C, B, B, D >$

Z è sottosequenza di X alle posizioni:

$$(1\ 4\ 5\ 9), (1\ 4\ 8\ 9), (1\ 5\ 8\ 9), (2\ 4\ 5\ 9), (2\ 4\ 8\ 9), (2\ 5\ 8\ 9)$$

L'ordine non è casuale: inizia sempre dalla prima lettera disponibile, poi cerca tutte le alternative a partire dall'ultima posizione. Una volta cambiata la prima lettera, la sequenza viene ricontrollata.

Variabili:

- $C[1 \ldots n]$, numeri interi che indicano le posizioni;
- $C[i]$, numero di elementi della più lunga sottosequenza crescente da 1 a i che termina in x_i.

Caso base: $c[1] = 1,\ max = 0$;

Passo ricorsivo: $c[i] = max\{c[j] \mid 1 < j < i \vee x_j < x_i\} + 1$ (*se stesso*);

Soluzione: $maxtot = max\{c[i] \mid 1 \leq i \leq n\}$.

L'algoritmo iterativo illustrato calcola la lunghezza massima delle sottostringhe in ordine crescente. Il valore $c[i]$ contiene l'ultima posizione della più lunga sottosequenza.
Per trovare la sequenza è sufficiente salvare l'elemento compatibile precedente a $c[i]$ per poi fare backtracking. In questo modo la sequenza può essere generata utilizzando $maxtot$.

Algorithm 7 Ricerca iterativa della sottostringa

```
function LONGEST_SUBSTRING(x)
    maxtot ← 1
    c[1] ← 1
    for i ← 2 to n step 1 do
        max ← 0
        for j ← 1 to (i − 1) do
            if x[j] < x[i] ∧ c[j] > max then
                max ← c[j]
            end if
            c[i] ← max + 1
            if c[i] > maxtot then
                maxtot ← c[i]
            end if
        end for
    end for
    return maxtot
end function
```

Tempo: $\Theta(n^2)$

Spazio: $\Theta(n)$ (riducendo lo spazio, il tempo aumenta)

5.2 Longest Common Substring

Longest **C**ommon **S**ubstring è il problema della sottostringa comune più lunga. Si ha:

$X = < x_1, x_2, \ldots, x_n >$,

$Y = < y_1, y_2, \ldots, y_n >$.

Z è sottosequenza comune a X, Y se è sottosequenza di X e contemporaneamente sottosequenza di Y. Il problema viene esteso al caso con due stringhe, si ricerca la più lunga sottosequenza con caratteri in ordine (come definito precedentemente).

NB: la funzione non è $1 : 1$, ci possono essere "salti" da una lettera all'altra (altrimenti sarebbe un semplice algoritmo che utilizza un ciclo for).

Le soluzioni iterative più banali hanno un tempo di 2^n, quindi si deve ricorrere alla programmazione dinamica.

Le istanze relative ai sottoproblemi sono X_i, Y_j, le quali sono $n+1$ e $m+1$ prefissi del problema dato. Si deve definire che cosa rappresentano gli indici. Un sottoproblema generico è identificato da una coppia di indici.

A ogni soluzione è associata una lunghezza: in questo caso la lunghezza della massima sottosequenza comune K è la maggiore tra le lunghezze di tutte le sottosequenze comuni W.

$Z_k = LCS(X_i,\ Y_j)$.
Se $x_i = y_j$, $LCS(x_{i-1}, y_{j-1}) \mid x_i$ (ultimo simbolo).

Altrimenti, se $z_k \neq x_i,\ Z_k = LCS(X_i,\ Y_j) = LCS(X_{i-1},\ Y_j)$ e $C_{i,j} = C_{i-1,\ j}$. Se $z_k \neq y_j,\ Z_k = LCS(X_i,\ Y_j) = LCS(X_i,\ Y_{j-1})$ e $C_{i,j} = C_{i,\ j-1}$.

Si cerca un algoritmo tale che

$$LCS(x,\ y) \rightarrow LCS(x - \{A\},\ y - \{A\})$$

La struttura dati per risolvere il problema in modo dinamico è una **matrice**, che indica come sono allineate le sequenze restringendo il numero di caratteri. L'algoritmo controlla i primi i elementi di X ed i primi j caratteri di Y.

Esempio 5.1

$$\begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ X = < & \mathrm{A}, & \mathrm{B}, & \mathrm{C}, & \mathrm{B}, & \mathrm{D}, & \mathrm{A}, & \mathrm{B} > \end{matrix}$$

$$\begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 \\ Y = < & \mathrm{B}, & \mathrm{D}, & \mathrm{C}, & \mathrm{A}, & \mathrm{B}, & \mathrm{A} > \end{matrix}$$

X/Y	0	B	D	C	A	B	A
0	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

La prima riga e la prima colonna sono fissi 0, e il controllo inizia dalla prima riga di 0. Non appena l'algoritmo trova un match, copia nella casella della matrice corrispondente il valore contenuto nella casella precedente sulla diagonale a sinistra, incrementandolo di 1.

Se i due caratteri confrontati sono diversi, viene copiato nella casella il valore massimo tra quello a sinistra e quello sopra, e l'algoritmo prosegue.

La casella alla posizione (n, m) rappresenta la massima lunghezza.

Variabile: $c[n, m]$ è una matrice il cui elemento $C[i, j]$ contiene la lunghezza della stringa più lunga fra gli i caratteri di X ed i j caratteri di Y.

Caso base:

$$C[i, j] = 0\ se\ i = 0\ \vee j = 0$$

Caso generico:

$$c[i, j] = \begin{cases} C[i-1, j-1] + 1 & se\ x_i = y_j \\ max\{C[i-1, j], C[i, j-1]\} & se\ x_i \neq y_j \end{cases}$$

Il problema può essere esteso con la richiesta della sottostringa più lunga crescente. In questo caso è necessario controllare la compatibilità di ogni lettera con le lettere precedenti. Per ottenere la

Algorithm 8 LCS

```
function LCS(X, Y)
    for i ← 0 to n step 1 do
        C[i, 0] ← 0
    end for
    for j ← 0 to m step 1 do
        C[0, j] ← 0
    end for
    for i ← 1 to n step 1 do
        for j ← 1 to m step 1 do
            if X[i] == Y[j] then
                C[i, j] = C[i − 1, j − 1] + 1
            else
                C[i, j] = MAX(C[i − 1, j], C[i, j − 1] + 1)
            end if
        end for
    end for
    return C[n, m]
end function
```

sottostringa, oltre alla lunghezza, bisogna salvare in $S[i, j]$ (una matrice aggiuntiva) i movimenti eseguiti da $S[n, m]$ a ritroso per ricomporre la stringa.

Il tempo di computazione è limitato superiormente dai due cicli for dell'algoritmo, quindi $\theta(nm)$ che è approssimabile a $\theta(n^2)$

Lo spazio richiesto è $\theta(nm)$ (la matrice), e può essere ridotto tenendo solo l'ultima riga oppure sostituendo le righe e colonne di 0 con degli if.

5.3 LCS crescente

Date due sequenze X, Y, si vuole trovare la sottostringa comune in ordine crescente più lunga. I sottoproblemi vengono formulati in questo modo: qual'è la più lunga sottostringa che termina con $x_i,\ y_i$?

Bisogna controllare ogni volta se la sottosequenza comune precedente è compatibile (crescente) con l'ultimo carattere aggiunto, per mantenere l'ordine.

Si ha che se $X_i = Y_i$ viene calcolata la LCS dei caratteri precedenti + 1; altrimenti l'algoritmo restituisce $\emptyset$.

Variabili:

- X_n con $|X_n| = n$, Y_m con $|Y_m| = m$;
- $C[n, m]$, una matrice;
- $C[i, j]$, il numero di caratteri che compongono la più lunga sottosequenza comune a X, Y crescente che termina con $(x_i = y_i)$.

L'equazione di ricorrenza del problema è

$$\begin{cases} C[i,j] = 0 & se\ x_i \neq y_i \\ C[i,j] = max\{C[h,k] \mid 1 \leq h < i,\ 1 \leq k < j,\ x_h < x_i,\ y_k < y_j\} + 1 & x_i = y_j \end{cases}$$

Questo può essere espresso semplicemente come

$$max\{C[i,j] \mid 1 \leq i \leq n,\ 1 \leq j \leq m\}$$

L'algoritmo funziona tramite una matrice similmente a LCS: le caselle corrispondenti a caratteri diversi vengono settate a 0.

Quando un carattere è uguale viene inserito il valore della più lunga sottosequenza comune compatibile precedente: in altre parole si cerca la casella con il valore più alto tale che l'elemento sia in ordine crescente.
Alla fine della ricerca del massimo, il numero viene incrementato di 1. Per eseguire queste operazioni sono necessari 4 cicli for.

Al completamento della scansione della matrice e dell'inserimento dei valori, bisogna trovare il massimo assoluto tra di essi: perciò è necessaria un'ulteriore ricerca tramite cicli for.

Questo rende l'algoritmo abbastanza pesante computazionalmente: il tempo è di circa $\theta(n^2m^2)$, approssimabile a $\theta(n^4)$.

5.3.1 Algoritmo

Algorithm 9 LCS_crescente

```
function LCS_CRESCENTE(X, Y)
    for i ← 1 to n step 1 do
        for j ← 1 to m step 1 do
            if X[i] ≠ Y[j] then
                C[i, j] ← 0
            else
                max ← 0
                for h ← 1 to i − 1 step 1 do
                    for k ← 1 to j − 1 step 1 do
                        if C[h, k] > max ∧ x[k] < x[i] then
                            max ← C[h, k]
                        end if
                    end for
                end for
                C[i, j] ← max + 1
            end if
        end for
    end for
    maxtot ← 0
    for i ← 1 to n step 1 do
        for j ← 1 to m step 1 do
            if C[i, j] > maxtot then
                maxtot ← C[i, j]
            end if
        end for
    end for
    return maxtot
end function
```

5.3.2 Implementazione

```
import sys
import numpy as np

X = [3,4,9,1]
Y = [5,3,8,9,10,2,1]

C = np.zeros(shape=(len(X), len(Y)))

# Risposta: [3,9], length = 2

for i in range(len(X)):
    for j in range(len(Y)):
```

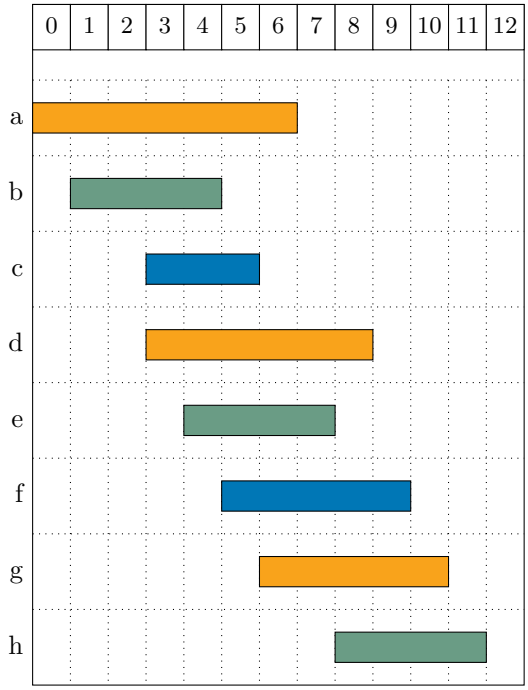
```
        if(X[i] != Y[j]):
            C[i][j] = 0
        else:
            max = 0
            for h in range(i - 1):
                for k in range(j - 1):
                    if (C[h][k] > max and X[k] < X[i]):
                        max = C[h][k]
            C[i][j] = max + 1

maxtot = 0
for i in range(len(X)):
    for j in range(len(Y)):
        if (C[i][j] > maxtot):
            maxtot = C[i][j]

print(maxtot)
```

5.4 Heaviest Common Subsequence

Questa è una variante di LCS, che associa a ogni elemento X_i o Y_j un peso. L'obiettivo è trovare la sequenza comune di peso maggiore (non necessariamente la più lunga).

A ogni aggiornamento della sequenza comune, va aggiunto il peso del nuovo elemento; il resto dell'algoritmo è concettualmente uguale. La programmazione dinamica aiuta a ottenere la soluzione ottimale.

6 Weighted Interval Scheduling



6.1 Il problema

Ci sono n attività, con $n \in \mathbb{N},\ n \geq 1$. Ogni attività (job) $i \in \{1,\ \ldots,\ n\}$ ha le seguenti proprietà:

1. Un job i inizia ad s_i, finisce a f_i, ed ha un peso (o valore) v_j;
2. Due jobs sono **compatibili** se non si sovrappongono: $[s_i, f_i) \cap [s_j, f_j) = \emptyset$.

Scopo: trovare il massimo subset di **peso** di jobs mutualmente compatibili, $A \subseteq \{1,\ \ldots,\ n\}$ per cui $\forall i,\ j \in A\ con\ i \neq j\ [s_i, f_i) \cap [s_j, f_j) = \emptyset$.

Ciò significa semplicemente: trovare la sequenza di jobs mutualmente compatibili (cioè una lista di jobs che non si sovrappongono), la cui **somma combinata** del peso (o valore) è la massima possibile. La compatibilità viene espressa tramite la seguente formula:

$$comp : \mathcal{P}(\{1,\ \ldots,\ n\}) \to \{true,\ false\}\ \forall\ A \subseteq \{1,\ \ldots,\ n\}$$

In altre parole:

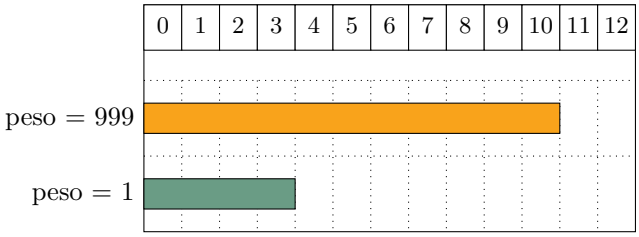
$$comp(A) = \begin{cases} true & se\ A\ contiene\ attività\ mutualmente\ compatibili \\ false & altrimenti \end{cases}$$

Definendo la funzione in base al peso v, si ha

$$v : \mathcal{P}(\{1,\ \ldots,\ n\}) \to \mathbb{R}\ \forall\ A\ \in\ \mathcal{P}(\{1,\ \ldots,\ n\}),\ A \subseteq \{1,\ \ldots,\ n\}$$

$$v(A) = \begin{cases} \sum_{i \in A} & se\ A \neq \emptyset \\ 0 & se\ A = \emptyset \end{cases}$$

Si ricorda che usare un algoritmo di tipo **greedy** non è la scelta più saggia, poichè i jobs hanno tutti peso (o valore) differenti.
Esempio:



6.2 Approccio

Il problema consiste nel trovare un sottoinsieme di $\{1,\ \ldots,\ n\}$ composto di attività mutualmente compatibili e di valore massimo.
Quindi, un'istanza $n \in \mathbb{N}$ con $n \geq 1$ tale che $\forall i \in \{1,\ \ldots,\ n\}$ si ha s_i, f_i, v_i massimi.

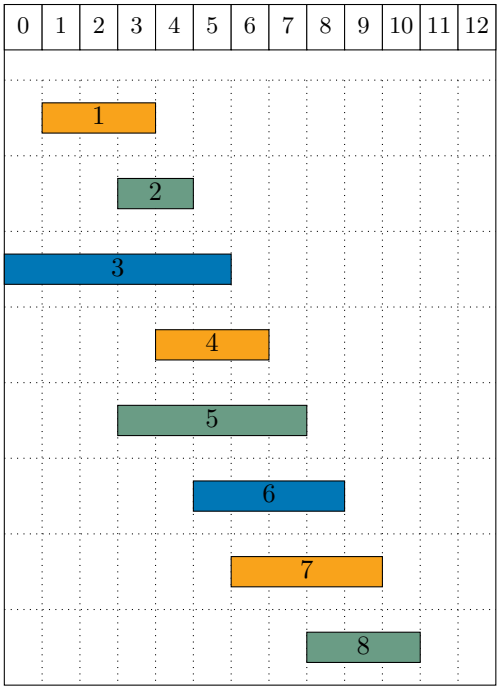
La soluzione più ovvia sarebbe il brute-force, che ha una complessità di 2^n. La programmazione dinamica risolve il problema dividendolo in sottoproblemi, e applicando *comp* ai sottoinsiemi fino ad arrivare al caso base (insieme vuoto). A ogni step si riduce l'insieme delle posizioni di 1.

Si ha che, se l'insieme ha una sola attività, $s_1 = \{1\}$ (caso base).
Se l'insieme è vuoto, $s_0 = \{\emptyset\}$ (caso base).
Si suppone di aver già risolto i problemi $\{0,\ 1,\ \ldots,\ i-1\}$ per poi risolvere il sottoproblema i (ci sono $n+1$ sottoproblemi).

La soluzione dell'algoritmo è $S \subseteq \{1,\ \ldots,\ n\}$ tale che:

1. $comp(S) = true$;
2. $v(S) = max\{v(A)\}$ con $\subseteq \{1,\ \ldots,\ n\},\ comp(A) = true$.

Si potrebbe ordinare i jobs secondo il loro tempo di fine:
$f_1 \leq f_2 \leq \cdots \leq f_n$.
$\forall i \in (\{1,\ \ldots,\ n\}),\ p(i) = max\{j | j < i, j\ è\ compatibile\ con\ i\}$.



j	$p(j)$
0	-
1	0
2	0
3	0
4	1
5	0
6	2
7	3
8	5

Ci sono due possibilità:

1. $i \in S_i$, allora $S_i = S_{p(i)} \cup \{i\}$ e $v(S_i) = v(S_{p(i)}) + v_i$;
2. $i \notin S_i$, allora la soluzione coincide con il sottoproblema precedente e $v(S_i) = v(S_{i-1})$.

La condizione del controllo è

$$v(S_i) = max\{v(S_{i-1}),\ v(S_{p(i)}) + 1\}$$

La tabella rappresenta con $p(i)$ il primo indice a ritroso di job compatibile con l'attività i.

6.3 Algoritmo

Algorithm 10 Weighted Interval Scheduling

```
function WIS-PD(n)
    M[0] ← 0
    M[1] ← v_1
    for i ← 2 to n step 1 do
        if M[p_i] + v_1 > M[i − 1] then
            M[i] ← M[p(i)] + v_i
        else
            M[i] ← M[i − 1]
        end if
    end for
end function
```

Le soluzioni dei sottoproblemi vegono memorizzate in un array M per evitare il tempo esponenziale di un algoritmo ricorsivo.

7 Scatole

Problema: $B_i = (a_i, b_i, c_i)$ in cui le variabili sono le 3 dimensioni (lunghezza, larghezza e altezza) di una scatola. Ogni elemento B_i è quindi una scatola. Le scatole non possono essere ruotate. Si chiede la lunghezza più lunga di scatole che possono essere contenute una dentro l'altra.

Si vuole descrivere un algoritmo efficiente per determinare il massimo valore di k tale che esista una sequenza $x \to B_1, B_2, \ldots, B_i$ tale che

$$B_{i1} \subset B_{i2} \subset \cdots \subset Bik$$

$$i_1 < i_2 < \ldots i_k$$

Problema risolto similmente alla sottosequenza crescente e al Weighted Interval Scheduling, ma la verifica viene effettuata su tutte e 3 le dimensioni.

Viene introdotto un vettore z con n componenti, tale che $z[i]$ sia la lunghezza massima di una sottosequenza crescente di elementi $B_1, B_2, \ldots, B_i$. La soluzione del problema sarà il valore massimo del vettore.

$$z[i] = \begin{cases} 1 + max\{z[j] \mid 1 \leq j < i,\ a_j < a_i,\ b_j < b_i,\ c_j < c_i\} & se\ i > 1 \\ 1 & se\ i = 1 \end{cases}$$

Supponiamo che $max\{\emptyset\} = 0$.

7.1 Algoritmo

Algorithm 11 Max Boxes Chain

```
function MAX-BOXES(B)
    z[1] ← 1
    for i ← 2 to n step do
        max ← 0
        for j ← 1 to i − 1 step do
            if (a_j < a_i) ∧ (b_i < b_j) ∧ (c_j < c_i) ∧ (z[j] < max) then
                max ← z[j]
            end if
        end for
        z[i] ←  +max
    end for
    return z
end function
```

La complessità è $O(n^2)$, mentre lo spazio richiesto è quello per memorizzare il vettore, cioè $\Theta(n)$.

8 Longest Zig-Zag Subsequence

Anche conosciuta come **Longest Alternating Subsequence**, consiste nel trovare la più lunga sottosequenza "zig-zag".

Una sequenza y è definita una sequenza alternata se i suoi elementi soddisfano una delle seguenti relazioni:

$$y_1 < y_2 > y_3 < y_4 > y_5 < \dots y_n$$

$$y_1 > y_2 < y_3 > y_4 < y_5 > \dots y_n$$

In questo caso si ricerca specificatamente una sequenza tale che
dispari $<$ *pari* $>$ *dispari* $<$ *pari* e così via.

Questo problema viene risolto passando da un problema ausiliario, cioè la ricerca delle sottosequenze pari e dispari che finiscano con l'elemento i incluso. Si sfrutta il metodo della programmazione dinamica.

Viene costruito un array bidimensionale Z:

- $Z[i, 0]$ conterrà la lunghezza della **LAS** che termina in posizione i, e dove l'ultimo elemento dell'array è più **grande** dell'elemento che lo precede;
- $Z[i, 1]$ conterrà la lungheza della **LAS** che termina in posizione i, e dove l'ultimo elemento dell'array è più **piccolo** dell'elemento che lo precede.

Equazione di ricorrenza:

$$Z[i,0] = \begin{cases} 1 & se\ i\ = 1 \\ max\{Z[k,1] \mid 1 \le k < i,\ y_k < y_i\} + 1 & se\ i > 1 \\ vale\ 0\ quando\ k\ non\ si\ trova \end{cases}$$

$$Z[i,1] = \begin{cases} 1 & se\ i\ = 1 \\ max\{Z[k,0] \mid 1 \le k < i,\ y_k > y_i\} + 1 & se\ i > 1 \\ vale\ max\ quando\ k\ non\ si\ trova \end{cases}$$

Il caso "pari" può facilmente non avere nessuna sottosequenza che soddisfi i vincoli. Può capitare che la maggior sottosequenza abbia lunghezza 0 per qualche i, il che non era mai stato considerato negli algoritmi precedenti. Pertanto, le due z non corrispondono a casi simmetrici.

L'elemento in posizione i dispari può essere aggiunto se è quello più piccolo del precedente: ma quest'ultimo dev'essere scelto cercando di ottenere la lunghezza maggiore e rispettando tutti gli altri vincoli.

8.1 Algoritmo

Algorithm 12 Longest Zig-Zag Subsequence

```
function ZIG-ZAG(Y)
    for i ← 1 to n do
        Z[i][0] ← Z[i][1] ← 1
    end for
    ris ← 1
    for i ← 2 to n do
        for j ← 1 to i do
            if Y[j] < Y[i] ∧ Z[i][0] < Z[j][1] + 1 then
                Z[i][0] ← Z[j][1] + 1
            end if
            if Y[j] > Y[i] ∧ Z[i][1] < Z[j][0] + 1 then
                Z[i][1] ← Z[j][0] + 1
            end if
        end for
        if ris < max(Z[i][0], Z[i][1]) then
            ris < max(Z[i][0], Z[i][1])
        end if
    end for
    return ris
end function
```

Esempio 8.1

$Y = \begin{pmatrix} 3 & 4 & 8 & 5 & 6 & 2 \end{pmatrix}$
E questo sarà:
$Z = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$
Alla prima iterazione avremo:
$i = 2, j = 1, Y = \begin{pmatrix} \textcircled{3} & \textcircled{4} & 8 & 5 & 6 & 2 \end{pmatrix}$
$Z = \begin{bmatrix} \textcircled{1} & 1 & 1 & 1 & 1 & 1 \\ 1 & \textcircled{1} & 1 & 1 & 1 & 1 \end{bmatrix} \Rightarrow Z = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \textcircled{2} & 1 & 1 & 1 & 1 \end{bmatrix}$
2a:
$i = 3, j = 1, Y = \begin{pmatrix} \textcircled{3} & 4 & \textcircled{8} & 5 & 6 & 2 \end{pmatrix}$
$Z = \begin{bmatrix} \textcircled{1} & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & \textcircled{1} & 1 & 1 & 1 \end{bmatrix} \Rightarrow Z = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & \textcircled{2} & 1 & 1 & 1 \end{bmatrix}$
$i = 3, j = 2, Y = \begin{pmatrix} 3 & \textcircled{4} & \textcircled{8} & 5 & 6 & 2 \end{pmatrix}$
$Z = \begin{bmatrix} 1 & \textcircled{1} & 1 & 1 & 1 & 1 \\ 1 & 2 & \textcircled{2} & 1 & 1 & 1 \end{bmatrix} \Rightarrow Z = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 1 & 1 & 1 \end{bmatrix}$
3a:

$i = 4, j = 1, Y = \begin{pmatrix} \textcircled{3} & 4 & 8 & \textcircled{5} & 6 & 2 \end{pmatrix}$

$Z = \begin{bmatrix} \textcircled{1} & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & \textcircled{1} & 1 & 1 \end{bmatrix} \Rightarrow Z = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & \textcircled{2} & 1 & 1 \end{bmatrix}$

$i = 4, j = 2, Y = \begin{pmatrix} 3 & \textcircled{4} & 8 & \textcircled{5} & 6 & 2 \end{pmatrix}$

$Z = \begin{bmatrix} 1 & \textcircled{1} & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & \textcircled{2} & 1 & 1 \end{bmatrix} \Rightarrow Z = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 & 1 & 1 \end{bmatrix}$

$i = 4, j = 3, Y = \begin{pmatrix} 3 & 4 & \textcircled{8} & \textcircled{5} & 6 & 2 \end{pmatrix}$

$Z = \begin{bmatrix} 1 & 1 & 1 & \textcircled{1} & 1 & 1 \\ 1 & 2 & \textcircled{2} & 2 & 1 & 1 \end{bmatrix} \Rightarrow Z = \begin{bmatrix} 1 & 1 & 1 & \textcircled{3} & 1 & 1 \\ 1 & 2 & 2 & 2 & 1 & 1 \end{bmatrix}$

$i = 5, j = 1, Y = \begin{pmatrix} \textcircled{3} & 4 & 8 & 5 & \textcircled{6} & 2 \end{pmatrix}$

$Z = \begin{bmatrix} \textcircled{1} & 1 & 1 & 3 & 1 & 1 \\ 1 & 2 & 2 & 2 & \textcircled{1} & 1 \end{bmatrix} \Rightarrow Z = \begin{bmatrix} 1 & 1 & 1 & 3 & 1 & 1 \\ 1 & 2 & 2 & 2 & \textcircled{2} & 1 \end{bmatrix}$

$\vdots$

$i = 6, j = 5, Y = \begin{pmatrix} 3 & 4 & 8 & 5 & \textcircled{6} & \textcircled{2} \end{pmatrix}$

$Z = \begin{bmatrix} 1 & 1 & 1 & 3 & 3 & \textcircled{3} \\ 1 & 2 & 2 & 2 & \textcircled{4} & 1 \end{bmatrix} \Rightarrow Z = \begin{bmatrix} 1 & 1 & 1 & 3 & 3 & \textcircled{5} \\ 1 & 2 & 2 & 2 & 4 & 1 \end{bmatrix}$

E infatti $LAS = 5 : 3 \not> 4 < 8 > 5 < 6 > 2$

9 Knapsack

9.1 Il problema

Algoritmo funzionante similmente a WIS. Data una serie di oggetti $X_n = \{1, \ldots, n\}$, $\forall i \in \{1, \ldots, n\}$ si hanno i valori v_i che corrisponde al valore e w_i che corrisponde al peso.
Uno zaino di capacità L dev'essere riempito con la combinazione ottimale di oggetti (sottoinsieme $S \subseteq X$) che massimizzi il valore.

In altre parole:

1. $\sum_{i \in S_n} w_i \leq C$;
2. $v(S_n) = max\{A \subseteq X_n,\ \sum_{i \in S_n} w_i \leq C\}$.

Questo problema deve tenere conto del peso e del valore di ogni oggetto, cercando di trovare la soluzione ottima in base a entrambi. La risoluzione, pertanto, utilizzerà una matrice.

Le variabili utilizzate sono:

- Le informazioni relative a v e w per ogni oggetto appartenente all'insieme di oggetti X;
- $M[n, L]$, matrice;
- $M[i, j]$, massimo valore ottenibile scegliendo tra i primi i oggetti per avere peso S.

$$\begin{cases} M[0, j] = 0 \\ M[i, 0] = 0 \\ M[i, j] = max\{M[i-1, j],\ M[i-1, j-w_i] + v_i\} \end{cases}$$

La soluzione corrisponde a $max\{M[n, j]\}$.

Esempio 9.1

$X = \{1, 2, 3, 4, 5\}$
$V = \{1, 6, 18, 22, 28\}$
$W = \{1, 2, 5, 6, 7\}$
$L = 11$
Si procede con la costruzione della matrice:

M	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0
2	0	1	6	7	0	0	0	0	0	0	0	0
3	0	1	6	7	0	18	19	24	25	0	0	0
4	0	1	6	7	0	18	22	24	29	29	0	0
5	0	1	6	7	0	18	22	28	29	34	35	40

La soluzione migliore è $S = \{3, 4\}$ con $P = 11$ e $V = 40$.
La prima riga e la prima colonna sono inizializzate a 0. Le caselle della matrice rappresentano i valori, ma esiste anche un'alternativa con i pesi. Se il peso dell'oggetto da aggiungere è più grande della colonna j da considerare, è sufficiente cercare $max\{M[i-1, j],\ M[i-1, j-w_i] +$

$v_i\}$.

9.2 L'algoritmo

Il tempo dell'algoritmo è $\Theta(nL)$, e lo spazio è $\Theta(nL)$ approssimabile a $\Theta(L)$.
Knapsack è considerato un problema NP-completo: come mai questo algoritmo lo risolve in tempo polinomiale?

Se i numeri dei pesi non sono piccoli rispetto alla quantità di oggetti da scegliere, il tempo è esponenziale: perciò l'algoritmo è pseudo-polinomiale. Se L è grande, il problema non è facilmente risolvibile.

Algorithm 13 Knapsack

```
function KNAPSACK(X, v, w, L)
    for j ← 1 to L do
        M[0, j] ← 0
    end for
    for i ← 1 to n do
        M[i, 0] ← 0
    end for
    for i ← 2 to n do
        for j ← 2 to L do
            if w_i > j then
                M[i, j] ← M[i − 1, j]
            else
                M[i, j] = max{M[i − 1, j], M[i − 1, j − w_i] + v_i}
            end if
        end for
    end for
    return M[n, L]
end function
```

Esiste una versione di Knapsack (frazionario, mentre questo è 0/1) che permette di scegliere un sottoinsieme di oggetti senza dover prendere un oggetto completamente. L'approccio è greedy.

10 Partition

Marco e Alessandro devono suddividersi un'eredità, composta da n oggetti con un valore intero associato. Trovare un algoritmo per dividerla equamente.

Si ha che, per avere una divisione equa, la somma dei valori dev'essere pari. Sia $m = \sum_{i=1}^{n} v_i$, ognuno dovrebbe ricevere $m/2$.

Esempio: $V = \{12, 4, 7, 5, 10, 5, 5\}$
Numeri in ordine decrescente: $\{12, 10, 7, 5, 5, 5, 4\}$

Una soluzione greedy sarebbe il calcolo della media usando per primi i valori grandi e poi distribuendo i valori più piccoli, assumendo di aver trovato una soluzione ottima.

La programmazione dinamica, invece, utilizza una variante di Knapsack con l'obiettivo di raggiungere $m/2$. Si costruisce una matrice booleana, il cui significato è:

- $M[i, j] = 1$ se esiste un sottoinsieme di $\{1, \ldots, i\}$ tale che la somma dei loro valori è uguale a j;
- $M[i, j] = 0$ altrimenti.

Le righe sono gli oggetti, le colonne sono i valori.
Il problema è come calcolare $M[i, j]$ senza dover computare 2^n possibilità.

$$M[i,j] = \begin{cases} M[i-1,j] \vee M[i-1,j-v_i] \\ M[0,j] = 0 \; con \; 1 \leq j \leq m/2 \\ M[i,0] = 1 \; con \; 0 \leq j \leq n \end{cases}$$

La soluzione è contenuta nella casella $M[n, m/2]$: se è 0 non c'è possibilità di riempire la matrice. Per fermare l'algoritmo è sufficiente fermarsi al primo 1 nell'ultima colonna.

11 Subset sum

Due insiemi $O = \{1, 2, \ldots, n\}$ e $V = \{v_1, v_2, \ldots, v_n\}$, trovare se esiste un sottoinsieme di O tale che $\sum_{i \in O} v_i = T$.

Se T è minore del minimo, il sottoinsieme (vuoto) viene trovato in $\Theta(n)$; la stessa cosa vale per il caso in cui T è maggiore della somma degli elementi.
Altrimenti, si procede analogamente al problema Partition, considerando che la matrice dovrà essere grande da 1 a T.

Il tempo è $\Theta(nT)$: il tempo di tutte le altre operazioni non viene superato dal tempo polinomiale. L'applicazione di Partition per risolvere il problema rende più facile risolvere Subset sum replicando le caratteristiche ed evitando il tempo esponenziale.

12 Programmazione greedy

La programmazione greedy, o golosa, riguarda problemi che hanno l'obiettivo di arrivare al risultato ottimo utilizzando sottostrutture ottime locali. Il problema, in questo caso, è la comprensione della possibilità di applicazione per tutti i possibili input.

La principale difficoltà di un algoritmo greedy è la dimostrazione della non esistenza di un algoritmo migliore.

12.1 Knapsack frazionario

Questo algoritmo è il classico esempio di problema applicabile alla programmazione greedy. In questo caso viene considerato il rapporto tra il peso e il valore, v_i/w_i, di cui il valore totale dev'essere il massimo possibile (fino a 1).

Knapsack frazionario presume che si possa mettere nello zaino anche solo una parte di oggetto.

Il primo step è il calcolo dei valori da utilizzare per ogni elemento, poi viene calcolato il massimo di essi. L'algoritmo cerca quali inserire in ordine di grandezza finché gli oggetti ci stanno completamente, poi prende la parte rimanente in termini di frazione.

Il caso migliore si verifica quando tutti gli oggetti ci stanno nello zaino. Il tempo computazionale è $O(n^2)$ nel caso di elementi non ordinati: quindi conviene ordinarli in tempo $O(n \log n)$ per poi prenderli con $O(n)$. L'ordinamento è il fattore più pesante computazionalmente.

Riassumendo, gli step sono:

1. Calcolo dei valori $R[i]$;
2. Ordinamento dei valori;
3. Considerazione del prossimo valore (se possibile), $S = S \cup R[i]$ che significa che il peso attuale è la somma dei pesi che comprendono il nuovo peso aggiunto;
4. Ripetizione del terzo step fin quanto possibile;
5. La soluzione viene trovata alla fine delle iterazioni.

Algorithm 14 Knapsack Frazionario

```
function KnapFraz(R, W, n, L)
    s ← 0
    peso ← 0
    for i ← 1 to n do
        R[i] ← V[i]/W[i]
    end for
    MergeSort(R)
    i ← 1
    while peso < L ∧ i ≤ n do
        if W[i]+ peso ≤ L then
            s ← s + 1
        else
            s ← s + (L − p)/W[i]
        end if
        peso = peso +W[i]
        i ← i + 1
    end while
    return s
end function
```

Il tempo totale è quindi approssimabile a $\Theta(n \log n)$. Esso non dipende dai valori, ma solo dal

numero di oggetti. Per memorizzare l'input il tempo è identico, quindi sicuramente l'algoritmo è polinomiale.

12.2 Benzinaio

Una macchina ha autonomia di r chilometri e ci sono n benzinai, ordinati per distanza in km. Lo scopo dell'algoritmo è minimizzare le soste percorrendo N km.

Calcolare tutte le possibili alternative costerebbe un tempo esponenziali. Una strategia possibile sarebbe calcolare il massimo della distanza che sia minore di k, per poi fare benzina (e ripetere fin quando non si arriva a destinazione).

Si ha $S = \emptyset$ che è l'insieme dei benzinai, inizialmente vuoto. I valori da utilizzare sono le differenze tra l'autonomia e la distanza di ogni benzinaio; a ogni pieno le distanze devono essere riaggiornate.

Per capire dove fermarsi (fare il pieno) c'è bisogno di una variabile che indica la posizione; poi viene cercato il massimo valore subito prima di arrivare a r. Ciò viene effettuato inizialmente con una ricerca del massimo, poi ripetendo la ricerca con i valori $p + r$.

La soluzione è ottimale: ogni stazione alternativa non consente di saltare due stazioni di S, quindi il numero di elementi nell'alternativa è sempre maggiore o uguale rispetto a quelli della scelta migliore.

13 LCS con 3 stringhe

Date 3 stringhe X_i, Y_j, W_l, di lunghezza rispettivamente n, m e l, determinare una più lunga sottosequenza comune utilizzando la programmazione dinamica.

Si considera il problema in versione ridotta: trovare la lunghezza di una delle più lunghe sottosequenze comuni a 3 stringhe.

Il generico sottoproblema (prefisso) è individuato da X_i, Y_j, W_h, con $i \in \{0, \ldots, n\}$, $j \in \{0, \ldots, m\}$, $h \in \{0, \ldots, l\}$, e consiste nel determinare la lunghezza di una delle più lunghe sottosequenze comuni a X_i, Y_j, W_k.

Per ogni sottoproblema, si ha una variabile $c_{i,j,h}$ (con $i \in \{0, \ldots, n\}$, ...) rappresentata da un numero intero che contiene la soluzione del sottoproblema $(i,\ j,\ h)\ \forall\ i,\ j,\ h$.

I casi limite sono gli unici conosciuti e per i quali non serve la soluzione del sottoproblema precedente, e corrispondono alla sottosequenza vuota (sottoproblemi individuati da triplette $(i,\ j,\ h)$ con $i = 0 \vee j = 0 \vee h = 0$).
Ci sono $(n+1)(l+1) + (n(l+1)) + (mn)$ sottoproblemi che corrispondono al caso base.

Il passo ricorsivo consiste in ogni tripletta $(i,\ j,\ h)$ con $i > 0$, $j > 0$, $h > 0$. Assumendo di aver già risolto tutti i sottoproblemi più piccoli, cioè tutti i sottoproblemi $(\bar{i},\ \bar{j},\ \bar{h})$ con $\bar{i} \leq i$, $\bar{j} \leq j$, $\bar{h} \leq h$.

Se le lettere sono uguali, similmente a LCS con due stringhe, si procede in diagonale nella matrice tridimensionale incrementando la lunghezza di 1:

$$c_{i-1,j-1,l-1} + 1$$

Se le lettere sono diverse tra di loro, invece, la soluzione sarà LA SIBILLA CUMANO:

$$max \begin{cases} LCS(X_{i-1}, Y_j, W_k) & \text{così} \\ LCS(X_i, Y_{j-1}, W_k) & \text{colà} \\ LCS(X_i, Y_j, W_{k-1}) & \text{pomì} \end{cases}$$

Merdaccia! Fila come una cartolina.

Algorithm 15 LCS 3 Sequences

```
function LCS(X, Y, Z)
    for j ← 0 to n do
        for h ← 0 to l do
            C_{0,j,h} ← 0
        end for
    end for
    for i ← 0 to n do
        for h ← 1 to l do
            C_{i,0,h} ← 0
        end for
    end for
    for i ← 1 to n do
        for h ← 1 to l do
            C_{i,j,0} ← 0
        end for
    end for
    for i ← 1 to n do
        for j ← 1 to l do
            for h ← 1 do
                if x_i = y_j = w_h then
                    C_{i,j,h} ← c_{i−1,j−1,h−1} + 1
                else
                    C_{i,j,h} ← Max(c_{i−1,j,h}, c_{i,j−1,h}, c_{i,j,h−1})
                end if
            end for
        end for
    end for
end function
```

14 Programmazione greedy

La programmazione greedy (o golosa, ingorda, miope) è una tecnica per la progettazione di algoritmi di ottimizzazione. Essa consiste nella costruzione di una soluzione mediante una successione di passi durante ciascuno dei quali viene scelto un elemento localmente migliore.

In altre parole, a ciascun passo la scelta migliore viene compiuta in ambito limitato, senza controllare che il procedimento porti effettivamente a una soluzione ottima.

14.1 Esempi di algoritmi greedy

Gli algoritmi greedy sono semplici e facilmente implementabili.

Il primo passo è la comprensione del problema e del metodo per risolverlo: non sempre la programmazione greedy riesce a trovarne la soluzione ottima. Un modo per sapere se questo è possibile è utilizzare il **Teorema di Rado**, ma è anche necessario individuare la NP-completezza.

In seguito, va dimostrato che l'algoritmo trovato è il migliore.

14.2 Altezza minima

In una sala da ballo ci sono n uomini di altezza $\{u_1,\ u_2,\ \ldots,\ u_n\}$ e n donne di altezza $\{d_1,\ d_2,\ \ldots,\ d_n\}$. Accoppiare ogni uomo con una donna in modo da minimizzare l'altezza, quindi $\sum {i_1}^n$ utilizzando un algoritmo greedy.

Una soluzione consiste nell'ordinare le due serie e accoppiarle in colonna, in modo da avere un'associazione $1:1$. In questo caso $c = c + |u'[i] - d'[i]|$ in un ciclo for.

La dimostrazione che questa sia la soluzione ottima è per assurdo, non esistono combinazioni migliori (proprietà del valore assoluto).

14.3 Travelling salesman

Si ha un insieme di vertici $\{v_1,\ v_2,\ \ldots,\ v_n\}$ con un peso $w : E \to \mathbb{R}^+$ associato a ogni coppia di vertici, con $(v_i,\ v_j) \in E\ \forall\ i,\ j$.

Si richiede un algoritmo per minimizzare il peso complessivo visitando tutti i vertici, quindi minimizzare $\sum_i w(v'_i,\ v'_{i+1})$. Ogni vertice dev'essere visitato solo una volta.

Un approccio semplice sarebbe la ricerca del minimo peso e la visita del vertice associato, e così via. Il problema è che se l'ultimo arco ha un peso elevato, non ci sono altre scelte e il valore complessivo aumenta di molto.

14.4 Intervalli chiusi

Si ha una serie di punti $\{x_1,\ x_2,\ \ldots,\ x_n\} \in \mathbb{R}$ (es. 1.1, 2.4). Trovare un insieme di intervalli tale che la distanza data sia coperta ma il numero di intervalli sia il minimo.

I punti devono essere ordinati, e il primo intervallo deve coprire l'estremità inferiore arrivando il più lontano possibile. Poi, tramite un ciclo, si iterano gli altri intervalli cercando quello con l'estremo sinistro che più si avvicina.

Qualsiasi altra soluzione alternativa coincide con quella ottima, oppure lascia spazi vuoti.

14.5 Banconota

Si suppone di avere una banconota da n euro, che si vuole cambiare in 20, 10, 5 o 1 euro. L'obiettivo è minimizzare la quantità di banconote totali restituite.

La programmazione greedy risolve questo problema assegnando il maggior numero di banconote da 20 finché il resto non è meno di 20; poi passando al taglio subito minore e così via.

Questa soluzione è la migliore perché, non usando banconote da 20, la quantità aumenterebbe. Il problema della dimostrazione più intuitiva è che in questo caso le banconote devono essere una multipla dell'altra: altrimenti, essa fallisce.

14.6 Bin packing

Si ha un insieme di oggetti $O = \{1,\ 2,\ \ldots,\ n\}$ con a ciascuno associato un peso w, e una serie di container $C = \{c_1,\ c_2,\ \ldots,\ c_n\}$. Si vuole trovare la combinazione ottimale di oggetti nei rispettivi container.

Gli oggetti vengono ordinati in base al peso, e ogni $O[i]$ viene inserito nel primo bagaglio che lo può contenere. Esiste anche il caso in cui la distribuzione ottimale abbia più oggetti nello stesso contenitore, quindi l'algoritmo greedy non sempre restituisce l'ottimo.

I singoli oggetti sono svincolati e possono essere aggiunti in base al loro costo, quindi una singola scelta non è influenzata dall'ambiente globale.

Questo è un problema NP-completo: nemmeno la programmazione dinamica riesce a risolverlo in tempo polinomiale.

15 Problemi di ottimizzazione greedy

Siano $< E,\ F >$ due insiemi, di cui E è il sovrainsieme e F una famiglia dell'insieme delle parti E (tutte le possibili combinazioni). Si ricorda che $|\mathcal{P}(E)| = 2^n$ con n elementi.

Una struttura di questo tipo si definisce **insieme di indipendenza** se vale la seguente proprietà: $\forall\ A \in F,\ B \subseteq A \rightarrow B \subseteq F$ (ideale d'ordine rispetto alla relazione di inclusione).

Sia $\mathbb{R}^+$ l'insieme dei numeri reali non negativi, una funzione peso è una arbitraria funzione $w : E \rightarrow \mathbb{R}^+$, per un elemento E.

Volendo estendere la funzione peso w da un elemento a un sottoinsieme, si può semplicemente porre $w'(A) = \sum_{i \in F} w_i$ per ottenere il peso complessivo di A.

Il problema di ottimizzazione sarà composto da istanza e soluzione. L'istanza è un sistema di indipendenza $< E, F >$ e una funzione peso $w : E \rightarrow \mathbb{R}^+$.

Tra tutte le soluzioni, si cercano quelli che rispettano due condizioni:

1. Ammissibilità, il peso totale non deve eccedere quello del contenitore;
2. Massimo, tra le soluzioni bisogna cercare la migliore.

In altre parole, si cerca un algoritmo greedy in grado di trovare il massimo della funzione peso estesa alle famiglie di sottoinsiemi.

Si parte dal sottoinsieme vuoto; dato un sottoinsieme $i \in E$, se esso è compatibile è possibile aggiungere il suo peso al totale. Un approccio greedy cerca il massimo valore possibile in un insieme delle scelte Q, ed eventualmente lo somma alla soluzione.

La variante con ordinamento impiega circa $O(n \log n)$ per il sorting, un tempo migliore rispetto alla scelta casuale. La verifica dell'appartenenza ha tempo lineare $C(n)$, quindi il tempo totale è di al più $O(n \log n + nC(n))$.

16 Matroidi

I matroidi (Birkhoff) servono per capire se un problema può essere risolto utilizzando un algoritmo greedy, qualunque sia la funzione peso considerata. Viene dimostrato che un sistema di indipendenza ha una soluzione ottima greedy se e solo se esso è un matroide.

$< E,\ F >$ è matroide se $\forall\ A,\ B \in F$, con $|B| = A + 1$, si ha che $\exists\ b \in B - A$ tale che $A \cup \{b\} \in F$.

Esempio: sia E un insieme finito di vettori generici in un sottospazio vettoriale, e F un insieme di vettori linearmente indipendenti. $< E,\ F >$ è un matroide?
Togliendo degli elementi a F l'insieme appartiene comunque ai vettori, ed esiste un b (che non sia già in A) che sia comunque linearmente indipendente rispetto agli altri. Di conseguenza, questo è un matroide.

Knapsack non ha un matroide: non è possibile effettuare operazioni di rimozione di elementi a piacere, perché ogni inserimento è vincolato dagli oggetti precedenti. A ogni passaggio si dovrebbero rimettere in discussione tutte le altre scelte.

16.1 Teorema di Rado

$< E,\ F >$ (sistema di indipendenza) è matroide se e solo se $\forall\ w : E \to \mathbb{R}^+$, l'algoritmo greedy per ogni w restituisce l'ottimo.

17 Algoritmi su grafi

Gli algoritmi su grafi sono rilevanti perché possono risolvere numerose tipologie di problemi, riducendoli appunto a un grafo.

Si ricorda che un grafo è un insieme $G = (V, E)$ dove V sono i vertici, E sono gli archi (edges, lati), ed esiste una funzione peso $w : E \to \mathcal{R}$. Un cappio viene rappresentato come $e(i,\ i)$.

Ci sono due modi principali per rappresentare un grafo:

1. Liste di adiacenza, array statico con liste dinamiche, utilizzate per grafi sparsi (con pochi lati);
2. Matrice di adiacenza, statica, con ogni vertice rappresentato in una rispettiva riga e colonna (per grafi densi).

Per decidere quale delle due usare bisogna confrontare $|E|$ con $|V|^2$: se E è molto minore si procede con la lista, altrimenti la matrice.

17.1 Liste di adiacenza

Si ha un array statico *Adj* di nodi, con un valore numerico univoco assegnato a essi. Ogni vertice è collegato agli altri tramite puntatori: se il grafo non è orientato, può esserci o non esserci il collegamento dall'altro lato (tranne nel caso dei cappi).

Le liste di adiacenza permettono di conoscere quali nodi sono collegati al nodo di partenza, ma non necessariamente ci sono cammini tra più di due di essi. I puntatori *next*, però, sono uno dopo l'altro e ogni singolo vertice ha un solo puntatore (che successivamente punta agli altri).

Tempo peggiore: $O(|V|)$, tempo medio: $\Theta(|V|/2)$, tempo migliore: $\Omega(1)$.
Spazio: $O(|V| + |E|)$ se il grafo è orientato, $O(|V| + 2\,|E|)$ altrimenti.

Se il grafo è pesato, si aggiunge a ogni valore un'altra casella (due campi, quindi), per rendere veloce la ricerca rimettendoci in termine di spazio.

17.2 Matrici di adiacenza

Si ha che la matrice di adiacenza, in caso di grafo non orientato, è simmetrica: quindi non è necessario tenere tutta la matrice in memoria. La rappresentazione e l'accesso in questo caso sono più complicate, quindi per convenzione non si divide.

Per indicare il peso è sufficiente mettere nell'incrocio riga-colonna corrispondente il valore, anziché 1; viceversa possono esserci valori NIL oppure (meglio) 0 o -1.

I tempi sono costanti: per accedere alla matrice è sufficiente trovare la casella desiderata, quindi $\Theta(1)$ nonostante lo spazio sia maggiore, $\Theta(|V|^2)$. Questa soluzione è comunque vantaggiosa se il grafo è denso, perché non è necessario scorrere l'array.

18 Visite di un grafo

Ci sono due visite principali: BFS e DFS, rispettivamente in ampiezza e in profondità.

La scelta cambia a seconda di quello che si vuole ottenere: BFS analizza una sola componente del grafo che è connessa alla sorgente, dando informazioni sulle distanze da tutti i vertici (per trovare quella minima), mentre la DFS analizza tutte le componenti dando informazioni su quali vertici sono collegati e quali no.

18.1 BFS

La BFS si concentra sui vertici adiacenti, mentre vede come infinitamente distanti quelli non direttamente raggiungibili. Per ogni vertice, si ha il valore della distanza minima (intero o infinito). BFS permette di trovare il **cammino minimo**.

Un vertice a distanza $d+1$ viene visitato solo dopo tutti quelli a distanza d. Un'altra informazione che di solito viene memorizzata è $p(v)$, cioè il nodo precedente nella visita.

Per capire se un vertice è già stato visistato si usa un colore: bianco se non è mai stato toccato, grigio se è stato incontrato ma non visitato in profondità e nero altrimenti.

L'algoritmo inizia segnando tutte le distanze tra ogni coppia di vertice come infinita, ogni parent è NIL e tutti i colori sono bianchi. Partendo dalla sorgente, li visita tutti da sinistra.

Per capire come trovare i vertici, l'algoritmo deve memorizzare gli n nodi da analizzare successivamente: la struttura dati più comoda è la coda, che garantisce la visita a partire dal primo livello.

Se ci sono due o più grafi scollegati, i nodi vengono inizializzati, ma al termine dell'algoritmo essi restano con distanza infinita: BISF può anche indicare se un grafo è connesso. Se il grafo non è orientato, ogni nodo viene visitato due volte.

Algorithm 16 BFS

```
function BFS(G, S)
    for i ← 1 to n do
        d[v_i] ← ∞
        c[v_i] ← White
        p[v_i] ← nil
    end for
    d[S] ← 0
    C[S] ← Grey
    ENQUEUE(Q, S)
    while !(EMPTYQUEUE(Q)) do
        u ← DEQUEUE(Q)
        for each w ∈ Adj[u] do
            if c[w] == White then
                d[w] ← d[u] + 1
                p[w] ← u
                c[w] ← Grey
                ENQUEUE(Q, W)
            end if
        end for
        c[u] = Black
    end while
end function
```

Esempio 18.1

Dato un grafo $G = (V, E)$, con $x \in V$, stampare i vertici y che hanno la seguente propretà: il numero di archi su un cammino minimo da x a y è pari.

Algorithm 17 Print-Even-Distance

```
function PRINT-EVEN-DISTANCE(G, x)
    BFS(G, x)
    for n ∈ V do
        if d[n] ≠ ∞ and d[n] mod 2 == 0 then
            PRINT(n)
        end if
    end for
end function
```

Esempio 18.2

Modificare l'algoritmo di visita BFS in modo che visiti fino a scoprire i vertici che distano k da s, con $k > 0$ (dentro la padella).

Algorithm 18 BFS

```
function BFS(G, S)
    for i ← 1 to n do
        d[v_i] ← ∞
        c[v_i] ← White
        p[v_i] ← nil
    end for
    d[S] ← 0
    C[S] ← Grey
    Enqueue(Q, S)
    while !(EmptyQueue(Q)) do
        u ← Dequeue(Q)
        for each w ∈ Adj[u] do
            if c[w] == White then
                d[w] ← d[u] + 1
                p[w] ← u
                if d[w] < k then
                    c[w] ← Grey
                    Enqueue(Q, w)
                else
                    c[w] ← Black
                end if
            end if
        end for
        c[u] = Black
    end while
end function
```

Esempio 18.3

Dato un grafo non orientato $G = (V, E)$, stabilire (true, false) se G è connesso (per ogni coppia di vertici esiste un cammino, relazione di raggiungibilità simmetrica).

Algorithm 19 Is-Connected

```
function Is-Connected(G)
    S ← Random(V)
    BFS(G, S)
    for n ∈ V do
        if c[n] == White then
            return false
        else
            return true
        end if
    end for
end function
```

Esempio 18.4

Dato un grafo non orientato $G = (V, E)$, stabilire se G è un albero (grafo connesso senza cicli, oppure $|E| = |V| - 1$).
Per risolvere il problema è possibile contare il numero di vertici e quello degli archi con variabili globali, e verificare se rispetta la relazione.

Esempio 18.5

Dato un grafo non orientato $G = (V, E)$ e un nodo $s \in V$, stabilire se G è un albero con radice in s.
Il colore del padre del u è nero. I figli non possono mai essere nella coda insieme al padre, quindi se un vertice è destinato a diventare nero, il padre è necessariamente nero (eheh).

19 DFS

DFS è un algoritmo che visita un grafo in profondità, al contrario di BFS che visita subito tutti i vertici adiacenti. La struttura dati utilizzata è la pila: ogni elemento viene subito rimosso.

Un'altra differenza è che DFS permette di ottenere informazioni sui nodi di un grafo come insieme, per esempio il numero di componenti connesse e le loro caratteristiche, mentre BFS è più utile per il cammino minimo o per i singoli vertici.

Similmente, invece, DFS ricorre all'utilizzo dei colori: bianco, grigio e nero, con gli stessi significati di BFS. Se il grafo non è orientato, non è detto che tutti i nodi vengano scoperti; per questo motivo la visita viene lanciata ogni volta che viene trovato un elemento bianco.

Viene settato un clock d per ogni vertice, che indica a quale unità di tempo esso è stato scoperto. Analogamente, f è il tempo relativo alla fine della visita di quel vertice.

Per capire se gli archi sono in avanti oppure semplicemente di attraversamento (portano a nodi neri), è sufficiente guardare il tempo di fine. Se i segmenti dei tempi sono disgiunti, è un arco di cross: se uno è sottoinsieme dell'altro, sono disgiunti.

La ricorsione è utile per simulare una pila senza doverla gestire direttamente: le chiamate ricorsive vengono eseguite dallo stack. Se necessario, l'algoritmo ricorsivo viene lanciato su ogni nodo.

19.1 Algoritmo

DFS prende in ingresso un grafo G e per ogni vertice inizializza il colore, il predecessore, il tempo di scoperta e di fine visita (implicitamente).

Se il vertice è bianco, chiama l'algoritmo *DFS_visit* che esegue la visita in profondità vera e propria.

Una volta scoperto il vertice, il tempo di esecuzione viene incrementato e salvato, e il colore diventa grigio. La procedura deve analizzare tutti i nodi sottostanti e, una volta finito, segnare il tempo di fine.

Tra l'inizio e la fine della scoperta, per ogni vertice della lista di adiacenza viene ricontrollato il colore. Se esso è bianco, viene visitato ricorsivamente il sottografo richiamando *DFS_visit*.

Algorithm 20 DFS

```
function DFS(G)
    for v ∈ G do
        c[v] ← White
        p[v] ← nil
    end for
    time ← 0
    for v ∈ G do
        if c[v] == White then
            DFS_VISIT(G,v)
        end if
    end for
end function
function DFS_VISIT(G,v)
    time ← time + 1
    d[v] ← time
    c[v] ← Gray
    for a ∈ Adj[v] do
        if c[a] == White then
            DFS_VISIT(G,a)
        end if
    end for
    time ← time + 1
    f[v] ← time
    c[v] = Black
end function
```

Il caso migliore per l'algoritmo è chiaramente il tempo costante. Il caso peggiore si verifica quando tutti i nodi vengono visitati con *DFS_visit*.
Tempo medio: $\Theta(|V| + |E|)$.

19.2 Proprietà

Teorema delle parentesi: dati due intervalli $i_1 = (d[u], f[u])$ e $i_2 = (d[v], f[v])$, confrontandoli si possono avere tre casi:

1. $i_1 \cap i_2 = \emptyset$, nodi disgiunti;
2. $i_1 \subseteq i_2$, figlio;
3. $i_1 \supseteq i_2$, padre.

Non esistono casi in cui gli intervalli siano accavallati.

Un lato si può etichettare in questi modi:

- Tree-edge, albero di scoperta (senza cicli);
- Back-edge (lato all'indietro), arco che porta a un vertice grigio;
- Forward-edge (lato in avanti), porta a un vertice nero, $d[u] < f[v]$;
- Cross-edge, lato di attraversamento da un grafo all'altro, $d[u] > f[v]$.

In un grafo non orientato, è possibile avere tree-edge e back-edge, ma non sono mai presenti forward-edge perché ogni arco viene prima esplorato dal lato opposto.

19.2.1 Ordinamento topologico

L'ordinamento topologico è utile per stabilire un ordine logico su grafi particolari, che rientrano nella categoria dei DAG (grafi diretti senza cicli).

Si fissa un nodo che deve per forza essere il primo, e poi ci sono delle possibili scelte (che rispettino i vincoli).

Viene lanciato DFS sul grafo che permette di visitare i nodi nell'ordine richiesto, mettendoli in ordine man mano.

Esempio 19.1

Modificare DFS affinché ad ogni lato del grafo venga messa l'etichetta corretta e stampata a video

Algorithm 21 DFS

```
function DFS_VISIT(G,v)
    time ← time + 1
    d[v] ← time
    c[v] ← Gray
    for a ∈ Adj[v]\P[v] do          ▷ Si rimuove P[v] solo se il grafo non è orientato
        if c[a] == White then
            PRINT("(v,a), Tree")
            DFS_VISIT(G,a)
        else if c[a] == Gray then
            PRINT("(v,a), Back")
        else if c[a] == Black then          ▷ Solo se il grafo è orientato
            if d[a] < f[a] then
                PRINT("Forward")
            else
                PRINT("Cross")
            end if
        end if
    end for
end function
```

20 Algoritmo di Floyd-Warshall

Dato un grafo pesato $G = (V, E, w)$, l'algoritmo determina il *cammino minimo tra due nodi*. Esiste anche l'alternativa per ogni vertice a partire da una data sorgente.

Un algoritmo greedy in questo caso non funziona: il numero di possibilità è troppo elevato, e il cammino con il minor numero di archi potrebbe non essere il meno pesante. Un'alternativa è il calcolo della distanza senza vertici intermedi, e il miglioramento dei valori conoscendo tutte le distanze.

Se due nodi non sono direttamente collegati da un arco, inizialmente la distanza è infinita; in seguito, l'algoritmo prova a trovare percorsi con il minimo peso attraversando altri nodi.

Il calcolo iniziale delle distanze viene effettuato usando una matrice quadrata di dimensione $n \times n$, che indica quanto pesa il cammino minimo che collega ogni coppia di vertici con un **collegamento diretto**. Sulla diagonale ci sono solo 0 (la distanza minima tra un nodo e se stesso è ovviamente 0); se una coppia non è collegata, viene inserito ∞.

$$D_{i,j}[0] = \begin{cases} 0 & se\ i = j \\ w_{ij} & se\ i \neq j \wedge (i,j) \in E \\ \infty & altrimenti \end{cases}$$

Questa procedura viene estesa con la possibilità di attraversare un vertice arbitrario, per esempio 1. Se ciò è conveniente, le entrate corrispondenti nella matrice vengono modificate con il nuovo peso minimo ottenuto.

Alcuni percorsi intermedi ottimali possono essere riutilizzati per le caselle successive: se è già stato calcolato un valore minimo per un percorso, quest'ultimo può servire come percorso intermedio tra altri due nodi.

La domanda è: come costruire la matrice $D(k)$ a partire dalla matrice precedente $D(k-1)$? Si ha:

$$D_{i,j}[k] = min \begin{cases} d_{i,j}[K-1] \\ d_{i,k}[K-1] + d_{k,j}[K+1] \end{cases}$$

La costruzione di tutte le matrici avviene in $\Theta(n^3)$, cioè il tempo di costruzione di una singola matrice moltiplicato per il tempo del confronto, che viene effettuato $(n+1)$ volte.

Lo spazio utilizzato è $\Theta(n^3)$, che può essere ottimizzato a $\Theta(n^2)$ considerando che una matrice una volta usata viene eliminata. Se il grafo non è orientato, tutte le matrici sono simmetriche, il che permette di ridurre ulteriormente lo spazio.

Esiste anche la versione esistenziale di questo algoritmo, i cui valori sono rappresentati da 0 e 1 (*true* e *false*). Al posto del minimo viene usato un *or* logico, dato che è sufficiente garantire l'esistenza del cammino minimo.

20.1 Algoritmo

Algorithm 22 Floyd-Warshall

```
function FL-WA
    for i ← 1 to n do
        for j ← 1 to n do
            if (i, j) ∉ E then
                d_{i,j} ← ∞
            else
                if i == j then
                    d_{0,i,j} ← 0
                else
                    d_{0,i,j} ← ω_{i,j}
                end if
            end if
        end for
    end for
    for k ← 1 to n do
        for i ← 1 to n do
            for k ← 1 to n do
                d_{k,i,j} = MIN(d_{k−1,i,j}, d_{k−1,i,k} + d_{k−1,k,j})
            end for
        end for
    end for
    return D^{(n)}
end function
```

21 Applicazioni di Floyd-Warshall

21.1 Soluzione ricorsiva del cammino minimo

Questo algoritmo funziona vincolando ogni sottoproblema a tenere in considerazione solo un determinato sottoinsieme di vertici. L'equazione di ricorrenza è simile a quella di Floyd-Warshall:

$$d_{ij}^{k} = \begin{cases} w_{ij} & se\ k = 0 \\ min(d_{ij}^{k-1}, d_{ik}^{k-1}, d_{kj}^{k-1}) & altrimenti \end{cases}$$

21.2 Esempio 1

Sia G un grafo con archi $E = E_1 \cup E_2$ con E_1, E_2 disgiunti. Si determino le equazioni di ricorrenza di un algoritmo di programmazione dinamica per il calcolo del costo minimo di un cammino dal

vertice i al vertice j con al più z archi di E_1.

$$w_{ij} = \begin{cases} 0 & se\ i = j \\ \infty & se\ i \neq j \wedge (i,j) \notin E \\ c_{ij} & se\ i \neq j \wedge (i,j) \in E \end{cases}$$

21.3 Esempio 2

Dato un grafo senza cappi $G = (V, E, W)$ con $n = |V|$ e una funzione $col : V \to C$, calcolare per ogni (i, j) con $i, j \in V$, il peso di un cammino minimo da i a j che contenga al massimo $M = 2$ vertici consecutivi di ugual colore.

Il problema viene diviso in sottoproblemi: il sottoproblema k rappresenta il peso di un cammino minimo da i a j che contiene al massimo m coppie di vertici consecutivi dello stesso colore con vertici intermedi $\in \{1 \dots k\}$.

Ogni variabile è rappresentata da una matrice con n^2 elementi $D^{(k,m)}$, di cui ogni casella indica $\forall\ (i, j) \in V^2$ il peso di un cammino minimo da i a j, cioè $d_{ij}^{(k,m)}$.

Il caso base è il sottoproblema (k, m) dove $k = 0$ e $n = 0$, quindi il sottoinsieme di vertici intermedi è vuoto e dev'esserci necessariamente un arco. Al massimo devono esserci 0 coppie di vertici consecutivi dello stesso colore (esattamente 0): la distanza non è infinito solo nel caso in cui ci sia un arco e che il colore di i sia diverso da quello di j.

22 Strutture dati per elementi disgiunti

Si ha un insieme $X = \{x_1,\ x_2,\ \dots,\ x_n\}$, tale che $\bar{X}_i \cap \bar{X}_j = \emptyset$. Ogni insieme ha un elemento x come rappresentante, che lo identifica.

Le operazioni fondamentali su questa struttura sono:

- *make_set(x)* che crea un insieme che ha un solo elemento: x, il suo rappresentante. Questo consente di inizializzare la struttura dati e da un singolo elemento creare un sottoinsieme a sé stante;
- *union(x, y)* che unisce due partizioni a partire da due elementi generici (non necessariamente i rappresentanti). Il risultato sarà l'unione delle partizioni a cui appartengono x, y. Il nuovo rappresentante non viene scelto secondo una regola precisa: qualsiasi va bene, anche se comunemente è uno dei due rappresentanti.
  $R = part(x) \cup part(y)$;
- *find_set(x)*, che restituisce il rappresentante dell'insieme a cui appartiene l'elemento x.

In un grafo, per capire quali componenti sono disgiunte, è sufficiente classificare ogni singolo elemento come parte di una componente connessa.

Dato $G = (V, E)$, $\forall\ v \in V$ viene chiamata *make_set(v)*; $\forall\ (u, v) \in E$, viene chiamata *union(u, v)*. Per trovare le diverse coppie (in cui $u \neq v$) si usa *find_set*. Se esse hanno lo stesso rappresentante, sono già nello stesso insieme.

In pratica, c'è una matrice con come righe i vertici e come colonne gli archi (non orientati): se due elementi non sono parte della stessa partizione, ne viene creata una nuova, altrimenti le due vengono unite.

Il tempo richiesto dalle operazioni dipende da come le informazioni sono memorizzate.

22.1 Liste di adiacenza

Se l'insieme non è un grafo, si utilizzano le liste di adiacenza che per ogni elemento indicano quelli vicini. Non è possibile unire direttamente una lista all'altra, quindi viene creata una lista di liste in tempo totale $\Theta(n)$.

Si suppone che il rappresentante sia identificato dalla testa della lista. Se è doppia, è sufficiente scorrere all'indietro con tempo peggiore n, ma per ottimizzare basta aggiungere un puntatore a ogni elemento. *find_set* ha tempo costante.

Per le altre operazioni, è utile avere puntatori *head*, *next* e *tail*. In questo modo, mettere insieme due liste ha tempo costante per trovare l'ultimo elemento e quelli precedenti, che vengono collegati all'altra lista e poi devono essere ordinati.

Ogni volta che *union* viene chiamata, il numero di elementi aumenta: le opzioni sono farlo tante volte con liste corte, o poche con liste lunghe.

Se le liste sono il più lunghe possibile, la quantità di aggiornamenti del puntatore al rappresentante è circa 2^i con i aggiornamenti, mai maggiore degli elementi della lista. Il totale dei passaggi considerando le liste in ordine di lunghezza è quindi di $O(n \log n)$.

Riassumendo:

1. *make_set* ha tempo $O(1)$;
2. *find_set* ha tempo $O(n)$;
3. *union* ha tempo $O(n)$, e ci sono fino a $\log n$ union: per accelerare si utilizzano liste dinamiche con campi *elemento*, *head*, *next*, *tail*.

22.2 Alberi

Si possono utilizzare gli alberi come strutture dati di appoggio. In questo caso, *make_set* crea un albero con un nodo solo, e il rappresentante nell'unione è la radice. La *union* deve collegare due radici, imponendone una come parent.

Il costo delle operazioni diventa:

1. *make_set* ha tempo $O(1)$;
2. *find_set* ha tempo ammortizzato $O(n)$, utilizzando la compessione dei cammini;
3. *union* ha tempo $O(1)$ (si uniscono le radici);

Unione per rango: serve per velocizzare *find_set*. Quando due alberi vengono uniti, si tiene traccia del rango, ovvero dell'altezza dell'albero. L'albero con rango minore viene unito con quello di rango maggiore, per tenere il calcolo del rappresentante in $O(h)$, con h altezza dell'albero.

Per ottimizzare la ricerca del rappresetante si possono salvare quelli intermedi per poterli riutilizzare in futuro. La meccanica è sicura, perché comunque andrebbe trovato un nuovo rappresentante con l'unione, e si evita di formare catene di parents.

Per migliorare ulteriormente, si possono comprimere i cammini. Gli alberi hanno performance migliori se l'unione è l'operazione più utilizzata, altrimenti le liste.

23 Minimum Spanning Tree

Minimum Spanning Tree è un problema di copertura di grafi non orientati.

Dato un grafo $G = (V, E)$ in cui a ogni lato è associato un peso, un Minimum Spanning Tree è un cammino senza cicli che colleghi tutti i nodi (connesso) con peso totale minimo.

Questo algoritmo si basa sulle seguenti definizioni:

- Un *taglio* in un grafo non orientato è una partizione di G, in modo che ogni vertice appartenga o meno a essa;
- Sia $G = (V, E)$ un grafo connesso, non orientato e pesato, con $A \subseteq E$ un sottoinsieme di archi di MST. Sia inoltre S un taglio tale che non ci sono archi in A che collegano due nodi di cui solo uno appartiene a S:
  - Un arco leggero (u, v) è un arco di peso minimo che attraversa il taglio (cioè collega due nodi di diverse partizioni);
  - (u, v) è anche un arco sicuro, ovvero se lo aggiungiamo ad A questo contnua a rappresentare un sottoinsieme di archi appartenenti a MST.

NB: un arco è leggero se è quello di peso minimo fra un sottoinsieme di archi che rispetti una determinata proprietà.

23.1 Algoritmo di Kruskal

Questo algoritmo serve per risolvere **Minimum Spanning Tree**, cercando archi sicuri di peso minimo da aggiungere alla foresta in costruzione.

L'algoritmo greedy funziona in questo modo:

- I lati vengono ordinati in base al peso non decrescente;
- Si parte dall'insieme A vuoto;
- Per ogni coppia di nodi (u, v) che formano un arco, se la find_set di u è diversa da quella di v (sicuro), l'arco viene aggiunto ad A;
- L'arco dev'essere leggero, cioé il costo dev'essere minimo;
- Il processo viene ripetuto fino a trovare il MST.

Vengono sfruttate le strutture dati per insiemi disgiunti. Il tempo totale è $O(ElogV)$ per il merge sort, e si ha che $|E| \geq |V - 1|$ quindi il tempo dell'ordinamento resta il limite superiore.

24 Algoritmi NP-completi

Lo studio della NP-completezza formalizza il concetto di problemi risolvibili in tempo polinomiale, e le proprietà di chiusura del loro insieme.

Ci sono due questioni che vanno affrontate: l'esistenza di una soluzione in termini di algoritmo (problemi indecidibili), e i tempi di calcolo accettabili (problemi intrattabili, non risolvibili rapidamente).

In base alla complessità, i problemi vengono divisi in classe: una classe è quindi l'insieme di problemi che, se esiste una soluzione, possono essere risolti da una macchina M usando $O(f(n))$ della risorsa R, con dimensione dell'input n.

Esempio di problema indecidibile: halting problem.
Esempio di problema intrattabile: qualsiasi algoritmo che richieda 2^n.

Un problema di decisione ha solo due possibili risposte: sì o no (al contrario dei problemi computazionali). Alcuni problemi di decisione sono NP-completi.

La classe P ha come istanze linguaggi, e si può descrivere come i problemi di decisioni prese in tempo polinomiale da una macchina di Turing. P rappresenta la classe problemi risolti da T_M in tempo $P(n)$, cioè $T_M(n) = O(P(n))$ dove $P(n)$ è un polinomio in n.

Le categorie di algoritmi, quindi, si dividono in:

- Risolvibili in tempo polinomiale, rappresentati da P;
  - Esempio: trovare un arco di peso minimo in un grafo;
- NP, non deterministici, per le quali le istanze che rispondono in modo affermativo al problema di decisione possono essere verificate in tempo polinomiale;
  - Esempio 1: esistenza di un cammino provando tutte le combinazioni. Questo non implica che non esista un algoritmo migliore, ma se non esiste va dimostrato (da qui $P \subseteq NP$);
  - Esempio 2: i test di primalità, fino a qualche decennio fa era NP, ma ora ci sono modi per risolverlo in tempo polinomiale;
  - Esempio 3: decifratura della crittografia;
- NP-completi, sottoinsieme di NP con problemi che hanno tutte le stesse caratteristiche, in cui ogni problema è riducibile a tutti gli altri in tempo polinomiale:
  - Se si risolve un singolo algoritmo NP-completo, si ha il modo per risolvere tutti gli altri;
  - Se anche solo un NP non ha soluzione in tempo polinomiale, sicuramente nessun NP-completo avrà soluzione;
  - Ogni problema NP-completo appartiene a NP, e può essere ridotto a P;
    - Esempio: Traveling Salesman Problem;
- NP-hard, che rappresenta i problemi NP-completi riducibili uno all'altro in tempo polinomiale;
  - Sono i problemi che sono almeno difficili quanto gli NP-completi, non necessariamente di decisione;
  - Tutti i problemi NP possono essere ridotti a NP-hard in tempo polinomiale.

$P \subseteq NP$ è uno dei principali quesiti in ambito informatico, e consiste nel capire se per ogni problema la cui soluzione è verificata in tempo polinomiale, esiste anche un modo per risolverlo in tempo polinomiale.

Un problema in P ha come limite superiore un tempo esponenziale, così come gli NP, ma esistono problemi al di fuori da questi insiemi che comunque hanno la stessa complessità computazionale: quest'ultima classe è definita **exp**.

24.1 Macchine di Turing

Per stabilire il tempo di computazione si utilizza una **funzione di complessità**:
$T_M(n) = max\{t_M(x) \mid |x| = n\}$, dove T_M è una macchina di Turing.

NB: è importante ricordare che una T_M può entrare in loop infinito. In generale, una macchina di Turing computa funzioni su stringhe oppure decide e accetta linguaggi, cioé risponde all'accettazione di un determinato input.

Alcuni algoritmi non hanno una collocazione precisa negli insiemi: non esiste un modo per risolverli in tempo polinomiale, ma non è stato dimostrato che serve un tempo esponenziale.

I problemi in P vengono rappresentati con una macchina di Turing che può essere deterministica o non deterministica: la classe di linguaggi accettata e la potenza è la stessa.

Esempio di linguaggio con T_M deterministica: $\{a^n b^n\}$, con il relativo problema di decisione.
Esempio di linguaggio con T_M non deterministica: $\{a^n b^n\} \cup \{a^n b^{2n}\}$, in cui all'inizio c'è la scelta tra quale dei due insiemi contiene la stringa.

Una T_M non deterministica è più veloce, perché il suo tempo di calcolo è:
$t_n(x) = \{|B| \mid$ B è ramo più breve accettante se $X \in L \vee$ B è ramo più lungo rifiutante se $X \notin L\}$.