Appunti di
Analisi e Progettazione di Algoritmi

A cura di:
Francesco Refolli
Matricola 865955

Anno Accademico 2022-2023

Chapter 1

Note sul Corso

todo: segnare delle note

Part I

Teoria

Chapter 2

Problema LCS

2.1 Introduzione

Definizioni

Alfabeto Un alfabeto Σ e' un insieme finito e non vuoto di simboli.
Si usano le prime lettere dell'alfabeto minuscolo per identificare i simboli generici.

Stringa Una stringa (o parola) w e' una giustapposizione (o concatenazione) di simboli dell'Alfabeto Σ.

Si usano le ultime lettere dell'alfabeto minuscolo per identificare il nome di una stringa. Esempio: una stringa $X = a \cdot b \cdot c \cdot d$. Sempre sara' indicata in futuro senza l'operatore per semplicita'.

Sequenza Una sequenza S e' un concetto analogo alla stringa, ma e' generalmente concepita come una elencazione di simboli di un alfabeto. La principale differenza sta nel fatto che questa puo' avere elementi di altri alfabeti.

Si usano le ultime lettere dell'alfabeto maiuscolo per identificare il nome di una sequenza. Si indica con $X = < a, b, c, d >$

L'i-esimo elemento si indica con x_i. La lunghezza puo' essere indicata tramite $|X|$.

Sottostringa Una sottostringa S di una stringa (o sequenza) e' una stringa che sia la concatenazione di elementi consecutivi della stringa o sequenza di partenza. Di solito prodotta tagliando un pezzo di lunghezza m dal capo, un pezzo di lunghezza n dalla coda.

Sottosequenza Una sottosequenza S di una stringa (o sequenza) e' una sequenza di elementi della stringa o sequenza di partenza che mantenga l'ordine degli stessi. (Non per forza la consecutivita'!). In questo senso una sottostringa e' analoga ad una sottosequenza che mantenga la consecutivita'. Di solito prodotta tramite la cancellazione di k elementi.

Prefisso Un prefisso di lunghezza i e' una sottosequenza composta dai primi i elementi consecutivi di una stringa o sequenza. Detta X una stringa, si indica con X_i.

Suffisso Un suffisso di indice i e' una sottosequenza composta da tutti gli elementi consecutivi a partire dall'indice i.

LCS

Introduzione Il problema LCS (Longest Common Subsequence) consiste nel cercare in tempo ragionevole la sottosequenza piu' lunga comune a due sequenze o stringhe.

Esempi

1. $X =<$ S, C, O, I, A, T, T, O, L, O $>$
   $Y =<$ B, A, R, A, T, T, O, L, O $>$
   La LCS e' $Z =<$ A, T, T, O, L, O $>$ **nota: puo' essere usata sia la prima che la seconda 'A' di** Y.

2. $X =<$ M, A, G, I, C, O $>$
   $Y =<$ M, A, N, T, E, N, E, R, E $>$
   La LCS e' $Z =<$ M, A $>$

3. $X =<$ M, A, I, A, L, E $>$
   $Y =<$ N, A, T, A, L, E $>$
   La LCS e' $Z =<$ A, A, L, E $>$

Si mette in evidenza che l'operazione di estrazione della LCS gode delle seguenti proprieta':

$$\cdot : \Sigma^* \times \Sigma^* \to \Sigma^* \mid A \cdot B = LCS(A, B) \tag{2.1}$$
$$A \cdot B = B \cdot A \tag{2.2}$$
$$A \cdot A = A \tag{2.3}$$
$$A \cdot B = A \cdot (A \cdot B) = B \cdot (A \cdot B) \tag{2.4}$$
$$(A \cdot B) \cdot C = A \cdot (B \cdot C) = B \cdot (A \cdot C) \tag{2.5}$$

2.2 Algoritmo TOP-DOWN

Ragionamento Detti X, Y due sequenze, i, j i rispettivi indici e n, m le rispettive lunghezze. Detti $Z = LCS(X, Y)$ e k il suo indice. Si inizializzano gli indici alle lunghezze. (si parte dal fondo!)

Si puo' ragionevolmente pensare che:

1. A) Se $x_i = y_j$ allora $z_k = x_i$ e $Z_{k-1} = LCS(X_{i-1}, Y_{j-1})$
2. B) Se $x_i \neq y_j$ e $z_k = x_i$ allora $Z_k = LCS(X_i, Y_{j-1})$
3. C) Se $x_i \neq y_j$ e $z_k = y_j$ allora $Z_k = LCS(X_{i-1}, Y_j)$
4. D) Se $z_k \neq y_j$ e $z_k \neq x_i$ allora $Z_k = LCS(X_{i-1}, Y_{j-1})$

Si nota che il caso D e' direttamente traducibile in codice, ma viene comunque compreso dai casi B e C, di conseguenza non avrebbe senso ripeterlo. In particolare si osserva che $D \Rightarrow (B \cdot C \Leftrightarrow C \cdot B)$.

Possiamo quindi sviluppare uno pseudo-codice basato su questi teoremi. Non potendo sapere nei casi B) e C) quale sia tra i due il risultato, l'unico modo per verificarlo e' esplorare entrambi i casi. Gli indici delle sequenze appartengono all'insieme $[1, length]$.

2.3 Procedura TOP-DOWN

Algorithm Algoritmo Banale per LCS

```
procedure LCS(X_i, Y_j)
    if X_i = <> or Y_j = <> then
        return <>
    else
        if x_i = y_j then
            return append(LCS(X_{i-1}, Y_{j-1}), x_i)
        else
            B ← LCS(X_i, Y_{j-1})
            C ← LCS(X_{i-1}, Y_j)
            if len(B) ≥ len(C) then
                return B
            else
                return C
            end if
        end if
    end if
end procedure
```

Complessita' Abbiamo appurato che questo algoritmo (TOP-DOWN) puo' portare a problemi quando l'input e' troppo grande, tuttavia e' possibile porvi rimedio

Uno dei problemi e' per esempio la ripetizione di lavoro gia' svolto in ricorsioni "parallele". Si potrebbe immaginare di memorizzare i risultati delle chiamate e usare quelli memorizzati all'occorrenza.
Resta pero' l'alto numero di chiamate: tendenzialmente $O(2^{n+m})$. Per questo si usa la programmazione dinamica: si risolve iterativamente un procedimento ricorsivo.

2.4 Usare la Programmazione Dinamica

Ragionamento L'Approccio Bottom-Up si basa sul partire dal risultato piu' piccolo, caso base, e costruire pr>essivamente i risultati delle chiamate "precedenti". Questo significa in pratica calcolare tutte le chiamate per tutte le combinazioni possibili d'input attiente al problema. Potrebbe sembrare un procedimento dispendioso ma in alcuni casi puo' essere quello piu' efficiente.

Supponendo di avere a disposizione due matrici b, c di dimensione $(n+1)(m+1)$, queste possono essere usate per simulare l'algoritmo ricorsivo nei vari passi. Attenzione, gli indici della matrice partono con 0!

Caso base:

$$\begin{cases} c[0][j] = 0 & 0 \leq j \leq n \\ c[0][j] = 0 & 0 \leq i \leq m \end{cases}$$

Caso ricorsivo:

$$c[i][j] = \begin{cases} c[i][j] = c[i-1][j-1] + 1 & x_i = y_j \\ c[i][j] = c[i-1][j] & c[i-1][j] \geq c[i][j-1] \\ c[i][j] = c[i][j-1] & c[i-1][j] < c[i][j-1] \end{cases}$$

$$b[i][j] = \begin{cases} b[i][j] = \text{" } \nwarrow \text{ "} & x_i = y_j \\ b[i][j] = \text{" } \uparrow \text{ "} & c[i-1][j] \geq c[i][j-1] \\ b[i][j] = \text{" } \leftarrow \text{ "} & c[i-1][j] < c[i][j-1] \end{cases}$$

Questo algoritmo riempie due matrici:

- c contiene le lunghezze dei vari $LCS(X_i, Yj)$
- b contiene il percorso da seguire per ottenere l'ottimale

Per ottenere $LCS(X, Y)$ non restera' da fare che seguire il percorso indicato da b con l'accortezza di raccogliere in coda gli x_i tali che $b[i][j] = Diagonal$.

Avremo bisogno di tre algoritmi:

- una funzione che inizializza la matrice generando il caso base
- una funzione che riempie la matrice con i casi ricorsivi
- una funzione che legge il percorso della matrice b
- una funzione che stampa la LCS

2.5 Procedura BOTTOM-UP

Algorithm Inizializza Matrice

```
procedure InizializzaMatrice(m, n)
    for i ← 0 to m do
        c[i][0] ← 0
    end for
    for j ← 0 to n do
        c[0][j] ← 0
    end for
end procedure
```

Algorithm Riempi Matrice

```
procedure RiempiMatrice(X, Y)
    for i ← 1 to m do
        for j ← 1 to n do
            if x_i = y_j then
                c[i][j] ← c[i − 1][j − 1] + 1
                b[i][j] ← " ↖ "
            else
                if c[i − 1][j] ≥ c[i][j − 1] then
                    c[i][j] ← c[i − 1][j]
                    b[i][j] ← " ↑ "
                else
                    c[i][j] ← c[i][j − 1]
                    b[i][j] ← " ← "
                end if
            end if
        end for
    end for
end procedure
```

Complessita'

- La complessita' temporale di InizializzaMatrice e' $O(n + m)$.
- La complessita' temporale di RiempiMatrice e' $O(n * m)$.
- La complessita' temporale di LeggiLCS e' $O(n + m)$.

LeggiLCS La complessita' di questo algoritmo (analogo e identico a StampaLCS), e' data in termini di upper-bound rispetto al caso peggiore, ovvero quando le due sequenze hanno solo il primo simbolo in comune. In quel caso il percorso seguito dall'algoritmo seguira' una traiettoria angolata, facendo quindi tutti i "piccoli passi" (l'unico tassello obliquo e' il target $b[1][1]$), per un totale di $n + m$ iterazioni.
LeggiLCS non fa parte del programma d'esame, e' riportato qua per completezza.

Algorithm Leggi LCS

```
procedure LeggiLCS(b, X, i, j)
    if i = 0 ∨ j = 0 then
        return <>
    end if
    if b[i][j] = " ↖ " then
        return append(LeggiLCS(b, X, i − 1, j − 1), x_i)
    else
        if b[i][j] = " ↑ " then
            return LeggiLCS(b, X, i − 1, j)
        else
            return LeggiLCS(b, X, i, j − 1)
        end if
    end if
end procedure
```

Algorithm Stampa LCS

```
procedure StampaLCS(b, X, i, j)
    if i = 0 ∨ j = 0 then
        return <>
    end if
    if b[i][j] = " ↖ " then
        StampaLCS(b, X, i − 1, j − 1)
        print(x_i)
    else
        if b[i][j] = " ↑ " then
            StampaLCS(b, X, i − 1, j)
        else
            StampaLCS(b, X, i, j − 1)
        end if
    end if
end procedure
```

Conclusione Quindi la complessita' temporale totale e' $O(n * m)$, molto migliore dell'algoritmo TOP-DOWN presentato nel paragrafo precedente.
E' pero' da tenere in considerazione che questo richiede spazio in memoria con una complessita' di $O(n * m)$, visto che deve mantenere due matrici $n * m$.
Certamente superiore al $O(n + m)$ dell'algoritmo TOP-DOWN, ma resta pur sempre un costo che vale la pena pagare per usufruire della strategia Bottom-Up.

Chapter 3

Problema LIS

3.1 Introduzione

LIS

Introduzione Il problema LIS (Longest Increasing Subsequence) consiste nel cercare in tempo ragionevole la sottosequenza piu' lunga crescente all'interno di una stringa o sequenza. Si mette in evidenza che $LIS(X) = LCS(X, SORTED(X))$.
Inoltre il teorema di Erdos-Szekeres afferma:
$\forall X \mid interiDistinti(X) \wedge |X| = n^2 + 1 \Rightarrow |LIS(X)| = |LDS(X)| = n + 1$.
Dove LDS e' il problema opposto a LIS, la Longest Decreasing Subsequence.

Esempi

- $X =$ <0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15>
  $Z =$ <0, 2, 6, 9, 11, 15>

  Ma non esiste solo quella soluzione, altri candidati sono

  $Z =$ <0, 4, 6, 9, 11, 15>
  $Z =$ <0, 2, 6, 9, 13, 15>
  $Z =$ <0, 4, 6, 9, 13, 15>

3.2 Algoritmo

Ragionamento

Notazione

$$X_i =< x_1, x_2, x_3, x_4, x_5, x_6, .., x_i > \tag{3.1}$$
$$Z_k =< z_1, z_2, z_3, .., z_k > \tag{3.2}$$
$$Z = LIS(X) \tag{3.3}$$

Caso 1 Sia $x_i = z_k$.
$Z_k = Z_{k-1} \cup \{x_i\}$.
Dove $Z_{k-1} = LIS(X_p)$, con $p \in [1, i)$.
Data la definizione di LIS: $\forall j \in [1, k) \;\; z_j < x_i$.
Ricorsivamente e' sufficiente tenere a mente che $z_{k-1} < x_i$.

Definisco una funzione $V : R \to R$ tale che: $V(j) = |LIS(X_j)|$

Definisco una funzione $P : R \to R$ tale che:
$P(i) = max_{V(j)}(\{\forall j \in [1, i) \;\; z_k < x_i\})$.

Caso 2 Sia $x_i \neq z_k$.
Allora $Z_k = LIS(X_{i-1})$.

Ricorsione Da qui, $Z_i = LIS(X_i)$

Caso Base

$$i = 0 \Rightarrow Z_i = <> \tag{3.4}$$
$$i = 1 \Rightarrow Z_i = X_i \tag{3.5}$$

Caso Ricorsivo

$$Z_i = \begin{cases} Z_{P(i)} \cup \{x_i\} & |Z_{P(i)}| + 1 \geq |Z_{i-1}| \\ Z_{i-1} & |Z_{P(i)}| + 1 < |Z_{i-1}| \end{cases}$$

Caso Generale

$$Z_i = \begin{cases} X_i & i < 2 \\ Z_{P(i)} \cup \{x_i\} & |Z_{P(i)}| + 1 \geq |Z_{i-1}| \\ Z_{i-1} & |Z_{P(i)}| + 1 < |Z_{i-1}| \end{cases}$$

3.3 Procedura BOTTOM-UP

Il vettore C: $C[i] = |LIS(X_i)|$.
Il vettore L: $L[i]$ =LAST-ELEMENT($LIS(X_i)$).
Il vettore H: $H[i] = P(i)$.

---

procedure LIS(X)
  $C[0] \leftarrow 0$
  $C[1] \leftarrow 1$
  $H[1] \leftarrow 0$
  $L[1] \leftarrow 1$
  for $i = 2$ to n **do**
    $p \leftarrow 0$
    for $j = 1$ to $i - 1$ **do**
      if $x_{L[j]} < x_i \wedge C[j] > C[p]$ **then**
        $p \leftarrow j$
      end if
    end for
    $H[i] \leftarrow p$
    if $C[p] + 1 \geq C[i - 1]$ **then**
      $C[i] \leftarrow C[p] + 1$
      $L[i] \leftarrow i$
    else
      $C[i] \leftarrow C[i - 1]$
      $L[i] \leftarrow L[i - 1]$
    end if
  end for
end procedure

---

procedure LEGGI-LIS(X, n)
  $i \leftarrow n$
  $Z \leftarrow <>$
  while $i \neq 0$ **do**
    if $i = L[i]$ **then**
      $Z \leftarrow prepend(Z, x_i)$
      $i \leftarrow H[i]$
    else
      $i \leftarrow L[i]$
    end if
  end while
  return Z
end procedure

Complessita' $T(n) = O(n^2) + O(n) = O(n^2)$

3.4 Variante Rizzi

LIS_V e' un sottoproblema di LIS, ovvero la LIS che termina con l'elemento i-esimo di X_i.
Il vettore C: $C[i] = |LIS_V(X_i)|$.
Il vettore H: $H[i] = P(i)$.

```
procedure LIS(X)
    C[1] ← 1
    H[1] ← 0
    Z ← C[1]
    for i = 2 to m do
        max ← 0
        H[i] ← 0
        for h = 1 to i − 1 do
            if x_h < x_i ∧ C[h] > max then
                max ← C[h]
                H[i] ← h
            end if
        end for
        C[i] ← 1 + max
        if C[i] > Z then
            Z ← C[i]
        end if
    end for
    return Z
end procedure
```

```
procedure BUILD-LIS(X)
    if H[i] ≠ 0 then
        BUILD-LIS(H[i])
    end if
    print(x_i)
end procedure
```

Complessita' $T(n) = O(n^2) + O(n) = O(n^2)$

Chapter 4

Problema LICS

4.1 Introduzione

LICS

Introduzione Il problema LICS (Longest Increasing Common Subsequence) consiste nel cercare in tempo ragionevole la sottosequenza comune piu' lunga crescente all'interno di due stringhe o sequenze.

Esempi

- X = <3,4,9,1>
  Y = <5, 3, 8, 9, 10, 2, 1>

  La soluzione sarebbe Z = <3,9>

4.2 Algoritmo

Ragionamento

Notazione

$$X_i =< x_1, x_2, x_3, x_4, x_5, x_6, .., x_i > \tag{4.1}$$
$$Y_i =< y_1, y_2, y_3, y_4, y_5, y_6, .., y_j > \tag{4.2}$$
$$Z_k =< z_1, z_2, z_3, .., z_k > \tag{4.3}$$
$$Z = LIS(X, Y) \tag{4.4}$$

Definisco una funzione $V : R \times R \to R$ tale che: $V(i, j) = |LICS(X_i, Y_j)|$

E una funzione analoga alla $P(i)$ di LIS:

Definisco una funzione $P : R \times R \to R \times R$ tale che:
$P(i, j) = max_{V(p,q)}(\{\forall p, q \mid p \in [1, i) \wedge q \in [1, j) \mid z_k < x_i \wedge z_k < y_i\})$.

Caso 1 $x_i = y_i \wedge x_i = z_k$

$Z_k = Z_{k-1} \cup \{x_i\}$
$Z_{k-1} = LICS(X_p, Y_q)$ con $p \in [1, i) \wedge q \in [1, j)$

Data la definizione di LICS:
$\forall w \in [1, k)\ \ z_w < x_i$.
$\forall w \in [1, k)\ \ z_w < y_j$.

In particolare:
$z_{k-1} < x_i$
$z_{k-1} < y_j$

Ora, visto che $x_i = y_i$, la $P(i, j)$ si semplifica:
$P(i, j) = max_{V(p,q)}(\{\forall p, q \mid p \in [1, i) \wedge q \in [1, j) \mid z_k < x_i\})$.

Quindi scrivo:
$\exists p, q \mid P(i, j) = (p, q) \Rightarrow Z_{k-1} = LICS(X_p, Y_q)$

Caso 2 $x_i = y_i \wedge x_i \neq z_k$

Logicamente, devo escludere questi valori:
$Z_k = LICS(X_{i-1}, Y_{j-1})$

Caso 3 $x_i \neq y_i \wedge x_i = z_k \Rightarrow y_j \neq z_k$
$Z_k = LICS(X_i, Y_{j-1})$

Caso 4 $x_i \neq y_i \wedge y_j = z_k \Rightarrow x_i \neq z_k$
$Z_k = LICS(X_{i-1}, Y_j)$

Caso 5 $x_i \neq y_i \wedge x_i \neq z_k \wedge y_j \neq z_k$
$Z_k = LICS(X_{i-1}, Y_{j-1})$

Ricorrenza Non conoscendo a priori il valore di Z, l'esplorazione binaria fara' si che si esplori il Caso 5 come sottocaso dei casi 3 e 4. Da qui, $Z_{i,j} = LIS(X_i, Y_j)$

Caso Base

$$i = 0 \Rightarrow Z_{i,j} =<> \tag{4.5}$$
$$j = 0 \Rightarrow Z_{i,j} =<> \tag{4.6}$$

Caso Ricorsivo

$$Z_i = \begin{cases} Z_{P(i,j)} & (x_i = y_j) \wedge (|Z_{P(i,j)}| + 1 \geq |Z_{i-1,j-1}|) \\ Z_{i-1,j-1} & (x_i = y_j) \wedge (|Z_{P(i,j)}| + 1 < |Z_{i-1,j-1}|) \\ Z_{i,j-1} & (x_i \neq y_j) \wedge (|Z_{i,j-1}| \geq |Z_{i-1,j}|) \\ Z_{i-1,j} & (x_i \neq y_j) \wedge (|Z_{i,j-1}| < |Z_{i-1,j}|) \end{cases}$$

Caso Generale

$$Z_i = \begin{cases} <> & i = 0 \vee j = 0 \\ Z_{P(i,j)} & (x_i = y_j) \wedge (|Z_{P(i,j)}| + 1 \geq |Z_{i-1,j-1}|) \\ Z_{i-1,j-1} & (x_i = y_j) \wedge (|Z_{P(i,j)}| + 1 < |Z_{i-1,j-1}|) \\ Z_{i,j-1} & (x_i \neq y_j) \wedge (|Z_{i,j-1}| \geq |Z_{i-1,j}|) \\ Z_{i-1,j} & (x_i \neq y_j) \wedge (|Z_{i,j-1}| < |Z_{i-1,j}|) \end{cases}$$

4.3 Procedura BOTTOM-UP

La matrice C: $C[i][j] = |LICS(X_i, Y_j)|$.
La matrice L: $L[i][j]$ =LAST-ELEMENT($LICS(X_i, Y_j)$).
La matrice H: $H[i][j] = P(i, j)$.

---

procedure LICS(X, Y)
  $INIZIALIZZA(|X|, |Y|)$
  for $i = 1$ to n **do**
    for $j = 1$ to m **do**
      if $x_i = y_i$ **then**
        $(p, q) \leftarrow P(i, j)$
        $H[i][j] \leftarrow (p, q)$
        if $C[p][q] + 1 \geq C[i-1][j-1]$ **then**
          $C[i][j] \leftarrow C[p][q] + 1$
          $L[i][j] \leftarrow (i, j)$
        else
          $C[i][j] \leftarrow C[i-1][j-1]$
          $L[i][j] \leftarrow L[i-1][j-1]$
        end if
      else
        if $C[i][j-1] \geq C[i-1][j]$ **then**
          $C[i][j] \leftarrow C[i][j-1]$
          $L[i][j] \leftarrow L[i][j-1]$
        else
          $C[i][j] \leftarrow C[i-1][j]$
          $L[i][j] \leftarrow L[i-1][j]$
        end if
      end if
    end for
  end for
end procedure

Complessita' $T(n) = \Theta(n \cdot m) * O(n \cdot m) + O(n)$
$T(n) = O(n^2 \cdot m^2) + O(n) = O(n^2 \cdot m^2)$

```
procedure INIZIALIZZA(n, m)
    for i = 0 to n do
        C[i][0] ← 0
        H[i][0] ← (0, 0)
    end for
    for j = 0 to m do
        C[0][j] ← 0
        H[0][j] ← (0, 0)
    end for
end procedure
```

```
procedure P(i, j)
    p ← 0
    q ← 0
    for w = 1 to i − 1 do
        for z = 1 to j − 1 do
            if L[w][z][0] = 0 ∧ C[w][z] = 0 then
                p ← w
                q ← z
            else if x_{L[w][z][0]} < x_i ∧ C[w][z] > C[p][q] then
                p ← w
                q ← z
            end if
        end for
    end for
    return (p, q)
end procedure
```

```
procedure LEGGI-LICS(X, n, m)
    i ← n
    j ← m
    Z ←<>
    while (i, j) ≠ (0, 0) do
        if (i, j) = L[i][j] then
            Z ← prepend(Z, x_i)
            (i, j) ← H[i][j]
        else
            (i, j) ← L[i][j]
        end if
    end while
    return Z
end procedure
```

4.4 Variante Rizzi

$LICS_V$ e' un sottoproblema di LICS, ovvero la LICS che termina con l'elemento i-esimo di X_i.
La matrice C: $C[i][j] = |LICS_V(X_i, Y_j)|$.
La matrice H: $H[i][j] = P(i, j)$.

```
procedure LICS(X, Y)
    Z ← 0
    for i = 1 to m do
        for j = 1 to n do
            if x_i ≠ y_j then
                C[i][j] ← 0
            else
                max ← 0
                H[i][j] ← (0, 0)
                for h = 1 to i − 1 do
                    for k = 1 to j − 1 do
                        if C[h][k] > max ∧ x_h < x_i then
                            max ← C[h][k]
                            H[i][j] ← (h, k)
                        end if
                    end for
                end for
                C[i][j] ← 1 + max
                if C[i][j] > Z then
                    Z ← C[i][j]
                end if
            end if
        end for
    end for
    return Z
end procedure
```

```
procedure BUILD-LICS(X)
    if H[i][j] ≠ (0, 0) then
        BUILD-LICS(H[i][j])
    end if
    print(x_i)
end procedure
```

Complessita' $T(n) = \Theta(n \cdot m) * O(n \cdot m) + O(n)$
$T(n) = O(n^2 \cdot m^2) + O(n) = O(n^2 \cdot m^2)$

Chapter 5

Problema Knapsack

5.1 Introduzione

Il problema dello zaino o Knapsack, e' un problema di ottimizzazione combinatoria che prevede di trovare un'opportuna composizione di multiinsieme con peso inferiore o uguale alla capacita' fissata affinche' il valore (o premio) totale sia ottimale (il massimo possibile). Si hanno:

- N oggetti con un peso w_i e un valore c_i.
- il peso massimo W sopportabile dallo zaino.
- la quantita' di oggetto i-esimo x_i inserito nello zaino.

Di questo problema esistono varianti che fissano ulteriori vincoli sul multiinsieme.

Knapsack Senza Limiti In questa versione lo zaino puo' contenere un oggetto in una quantita' $x_i \geq 0$ a piacere e illimitata superiormente.

Knapsack Con Limiti Ad ogni elemento i-esimo e' associata una disponibilita' massima $d_i \geq 0$ tale che $x_i \leq d_i$.

Knapsack 0-1 I limiti di disponibilita' degli oggetti sono binari, ovvero $d_i = 1$.

Chapter 6

Knapsack Senza Limiti

6.1 Ragionamento

Intuizione Si riduce il problema al cercare il massimo valore della soluzione.
Quindi ragiono in termini di capacita' $k \leq W$ dello zaino con $A(k)$ la soluzione del problema ridotto con massimo peso k.
Si puo' quindi ragionevolmente pensare che se k = 0 allora A(k) = 0.
E' naturale pensare che $\forall j \leq k \mid A(k) \geq A(j)$.
D'altra parte se $A(k) > 0$ allora $\exists j \in [1, N]$ tale che $w_j \leq k \wedge A(k) = A(k - w_j) + c_j$.
Questo implica ovviamente anche che se $A(k) = A(k - w_j) + c_j$ allora $\forall i \in [1, N] \mid w_i \leq k \Rightarrow A(k - w_j) + c_j \geq A(k - w_i) + c_i$.
Di conseguenza posso impostare la soluzione $A(k)$ ricorsivamente rispetto a tutti i pesi $w_j \leq k$ e adoperare la sottosoluzione migliore.

Caso Base

$$\begin{cases} 0 & k = 0 \end{cases}$$

Caso Ricorsivo

$$\begin{cases} max(\{\forall j \in [1, N] \wedge w_j \leq k \mid A(k - w_j) + c_j\}) & k > 0 \end{cases}$$

Caso Generale

$$\begin{cases} 0 & k = 0 \\ max(\{\forall j \in [1, N] \wedge w_j \leq k \mid A(k - w_j) + c_j\}) & k > 0 \end{cases}$$

6.2 Procedura PD

Detti X i pesi, Y i valori, W la capacita' massima.

```
procedure KS(X, Y, W)
    A[0] ← 0
    for k = 1 to W do
        max ← 0
        for j = 0 to N do
            if x_j ≤ k ∧ A[k − x_j] + y_j > max then
                max ← A[k − x_j] + y_j
            end if
        end for
        A[k] ← max
    end for
end procedure
```

Ricostruzione Per ricostruire il Knapsack e' sufficiente mantenere uno storico dei passi e seguire il percorso a partire da $A[W]$. In alternativa si puo' simulare la ricorsione top-down con $A[i]$ al posto della chiamata ricorsiva.

Complessita' La complessita' temporale e' $O(N \cdot W)$. Questa complessita' temporale e' pseudo-polinomiale perche' W e' proporzionale alla sua lunghezza in bit.

Chapter 7

Knapsack 0-1

7.1 Ragionamento

Oltre a iterare sul peso a disposizione e' utile iterare sulla varieta' degli oggetti a disposizione.

Detti quindi $i \in [0, N]$, $j \in [0, k]$, $A(i, j)$ e' la soluzione con i oggetti e capacita' massima j.
Ovviamente $j = 0 \vee i = 0 \Rightarrow A(i, j) = 0$.
Naturalmente se $w_i > j$ allora $A(i, j) = A(i - 1, j)$.
Se invece $w_i \leq j$ le uniche opzioni possibili sono di prendere o non prendere l'oggetto.
Se $i \in Knapsack$ allora $A(i, j) = A(i - 1, j - w_i) + c_i$, altrimenti $A(i, j) = A(i - 1, j)$.
Per stabilirlo e' sufficiente prelevare la sottosoluzione migliore, ovvero il valore totale piu' grande.

Caso Base

$$\begin{cases} 0 & i = 0 \vee j = 0 \end{cases}$$

Caso Ricorsivo

$$\begin{cases} A(i-1,j) & w_i > j \\ A(i-1,j-w_i) + c_i & w_i \leq j \wedge A(i-1,j-w_i) + c_i \geq A(i-1,j) \\ A(i-1,j) & w_i \leq j \wedge A(i-1,j-w_i) + c_i < A(i-1,j) \end{cases}$$

Caso Generale

$$\begin{cases} 0 & i = 0 \vee j = 0 \\ A(i-1,j) & w_i > j \\ A(i-1,j-w_i) + c_i & w_i \leq j \wedge A(i-1,j-w_i) + c_i \geq A(i-1,j) \\ A(i-1,j) & w_i \leq j \wedge A(i-1,j-w_i) + c_i < A(i-1,j) \end{cases}$$

7.2 Procedura PD

Detti X i pesi, Y i valori, W la capacita' massima.

```
procedure INIZIALIZZA(N, W)
    for i = 0 to N do
        M[i][0] ← 0
    end for
    for j = 0 to W do
        M[0][j] ← 0
    end for
end procedure
```

```
procedure KS(X, Y, W)
    INIZIALIZZA(N, W)
    for i = 0 to N do
        for j = 0 to W do
            if x_i ≤ j then
                if M[i − 1][j − x_i] + y_i ≥ M[i − 1][j] then
                    M[i][j] ← M[i − 1][j − x_i] + y_i
                else
                    M[i][j] ← M[i − 1][j]
                end if
            else
                M[i][j] ← M[i − 1][j]
            end if
        end for
    end for
end procedure
```

Ricostruzione Per ricostruire il Knapsack e' sufficiente mantenere uno storico degli svincoli e seguire il percorso a partire da $M[N][W]$. In alternativa si puo' simulare la ricorsione top-down con $M[i][j]$ al posto della chiamata ricorsiva.

Complessita' La complessita' temporale e' $O(N \cdot W)$. Questa complessita' temporale e' pseudo-polinomiale perche' W e' proporzionale alla sua lunghezza in bit.

Chapter 8

Knapsack Con Limiti

8.1 Ragionamento

TODO

8.2 Procedura PD

TODO

Chapter 9

Cenni Fondamentali

9.1 Teoria Spicciola

Definizione Un grafo e' un insieme di elementi detti nodi o vertici che possono essere collegati fra loro da linee chiamate archi o lati o spigoli. Si difinisce grafo una coppia ordinata G = (V , E) di insiemi, con V insieme dei nodi ed E insieme degli archi, tali che gli elementi di E siano coppie di elementi di V (da $E \subseteq (V \times V)$ segue in particolare che $|E| \leq |V \times V|$).

Un cammino da v_i a v_j e' una sequenza finita di vertici $s \subseteq V$, con $s_1 = v_i$ e $s_{|s|} = v_j$, tali che $\forall i \in [1, |s|) \mid \exists e \in E \ \ con \ \ e = (s_i, s_{i+1})$.
Un vertice v_j si dice ragigungibile da un vertice v_i se esiste un cammino da v_i a v_j.
La distanza di un vertice v_i a un vertice v_j e' il numero di archi su un cammino minimo da v_i a v_j.

Approfondimento Un grafo e' una tripla (V, E, f), dove V e' detto insieme dei nodi, E e' detto insieme degli archi e f e' una funzione che associa ad ogni arco $e \in E$ due vertici $u, v \in V$ (in tal caso il grafo verra detto ben specificato). Se invece E e' un multiinsieme, allora si parla di multigrafo.
Due vertici u, v connessi da un arco e prendono il nome di estremi dell'arco; l'arco e viene anche identificato con la coppia formata dai suoi estremi (u, v). Se E e' una relazione simmetrica allora si dice che il grafo e' non orientato (o indiretto), altrimenti si dice che e' orientato (o diretto).
Un arco che ha due estremi coincidenti, ovvero un arco che connette un nodo con se stesso, si dice cappio. Un grafo non orientato, sprovvisto di cappi si dice grafo semplice.
Lo scheletro sk(G) di G e' il grafo che si ottiene da G eliminandone tutti i cappi e sostituendone ogni multiarco con un solo arco avente gli stessi estremi.
I vertici appartenenti a un arco o spigolo sono chiamati estremi, punti estremi o vertici estremi dell'arco. Un vertice può esistere in un grafo e non appartenere a un arco.

Rappresentazione di Grafi Su un calcolatore il grafo non viene rappresentato in forma di coppia ordinata $G = (V, E)$, ma utilizzando due strutture principali:

- Liste di Adiacenza
- Matrici di Adiacenza

Entrambe sono sufficienti a rappresentare sia grafi orientati che non orientati, semplicemente per quelli orientati necessiteranno di piccole modifiche per supportare l'indicazione di orientamento dell'arco.

Algoritmi di Visita Gli algoritmi che si approfondiscono in questo corso sono:

- Breadth-First Search
- Depth-First Search

9.2 Liste di Adiacenza

In questa forma abbiamo per un grafo $G = (V, E)$, con $n = |V|$ e $m = |E|$, liste che contengono i vertici collegati all'i-esimo vertice:

$$L_i = \{v_j \in V \mid \exists e \in E \ \ con \ \ e = (v_i, v_j)\}$$

Uso di Memoria La dimensione del sistema di liste e' impattato principalmente dal numero di archi, perche' e' proprio questo dato che viene rappresentato direttamente in memoria. Nel caso di grafi orientati l'uso di memoria e' $S(n) = \Theta(m)$. Nel caso di grafi non-orientati l'uso di memoria e' $S(n) = \Theta(2m)$, perche' devo rappresentare un arco non orientato sia dal punto di vista di v_i che dal punto di vista di v_j. Le liste di adiacenza sono utili quindi per rappresentare in modo compatto i grafi sparsi (grafi in cui ci sono pochissimi archi rispetto al numero di nodi).

Tempi di Esecuzione Se voglio sapere se due vertici sono collegati da un arco devo scorrere tutta la lista di adiacenza del nodo i-esimo, quindi $T(n) = O(n)$.

9.3 Matrici di Adiacenza

Detto un grafo $G = (V, E)$, con $n = |V|$ e $m = |E|$, la sua matrice di adiacenza e' una Matrice $n \times n$ con funzione caratteristica $f : V \to \{0, 1\}$, definita come:

$$f(i,j) = \begin{cases} 1 & \exists e \in E \wedge e = (v_i, v_j) \\ 0 & altrimenti \end{cases}$$

Se gli archi sono pesati, si sceglie un valore k che rappresenti l'inesistenza dell'arco e si usano k al posto degli zeri, i pesi al posto degli uni nella funzione caratteristica.

Uso di Memoria L'uso di memoria e' immediato: $S(n) = \Theta(n^2)$. La matrice di adiacenza e' ottimale per rappresentare un grafo non-sparso, perche' ottimizza l'uso della memoria e i tempi di accesso (simil-hashmap). Se il grafico e' non-orientato la matrice e' simmetrica, quindi si potrebbe rappresentare anche solo i valori sopra la diagonale principale.

Tempi di Esecuzione Per scoprire se due nodi sono connessi e' sufficiente visitare la matrice nella cella (i, j). Quindi $T(n) = \Theta(1)$

Chapter 10

Breadth-First Search

10.1 l'Algoritmo

Questo algoritmo esplora un grafo andando in ampiezza, ovvero considerando prima tutti i "figli" e solo dopo i sottoalberi dei figli.
l'algoritmo BFS scopre tutti i vertici raggiungibili dal vertice sorgente. Ovvero genera un albero radicato nel nodo input contentente tutti i nodi raggiungibili da esso.
Si supponga di avere a disposizione le liste di adiacenza del grafo G. Inoltre attribuisco ad ogni nodo un colore:

$$color[i] = \begin{cases} bianco & \text{vertice non visitato} \\ grigio & \text{non visitato completamente} \\ nero & \text{gia' visitato} \end{cases}$$

Ho inoltre bisogno di mantere $parent[i]$, il predecessore del nodo v_i.

10.2 Procedura BFS

```
procedure BFS(G, i)
    for j ∈ V \ v_i do
        color[j] ← bianco
        distance[j] ← ∞
        parent[j] ← NIL
    end for
    color[i] ← grigio
    distance[i] ← 0
    parent[i] ← NIL
    Queue ← {i}
    while Queue! = ∅ do
        j = dequeue(Queue)
        for k ∈ L_j do
            if color[k] = bianco then
                color[k] = grigio
                parent[k] = j
                distance[k] = distance[j]
                enqueue(Queue, k)
            end if
        end for
        color[j] ← nero
    end while
    color[i] ← nero
end procedure
```

Tempi di Calcolo $T(n) = (n + m)$

Chapter 11

Depth-First Search

11.1 l'Algoritmo

Questo algoritmo esplora un grafo "andando il piu' possibile in profondita'".

Si supponga di avere a disposizione le liste di adiacenza. Assegno ad ogni vertice un colore:

$$color[i] = \begin{cases} bianco & \text{vertice non visitato} \\ grigio & \text{non visitato completamente} \\ nero & \text{gia' visitato} \end{cases}$$

Ho inoltre bisogno di mantere $parent[i]$, il predecessore del nodo v_i. Mantengo anche $discover[i]$, il tempo di scoperta del vertice v_i. Mantengo anche $finish[i]$, il tempo di fine visita del vertice v_i. Queste informazioni restituiscono informazioni sulla relazione trai vari vertici del grafo. Sono utili per analisi post-esecuzione dell'algoritmo.

11.2 Procedura DFS

```
procedure DFS(G)
    for i = 1 to n do
        color[i] ← bianco
        parent[i] ← NIL
    end for
    time ← 0
    for i = 1 to n do
        if color[i] = bianco then
            DFS-visit(G, i)
        end if
    end for
end procedure
```

```
procedure DFS-VISIT(G, i)
    time ← Time + 1
    disc[i] ← time
    color[i] ← grigio
    for j ∈ L_i do
        if color[j] = bianco then
            parent[j] = i
            DFS-visit(G, j)
        end if
    end for
    color[i] ← nero
    time ← Time + 1
    finish[i] ← time
end procedure
```

Tempi di Calcolo L'algoritmo DFS usa l'algoritmo DFS-visit su tutti i nodi che non sono stati scoperti. $T(n) = \Theta(n + m)$

Chapter 12

All Pairs Shortest Path

l'**All Pairs Shortest Path** consiste nel trovare lo **Shortest Path** per ogni coppia di nodi v_1, v_2. Per grafi diretti e indiretti esistono algoritmi differenti che consentono di trovare una soluzione. Nel caso di grafi non pesati, si usa un'approssimazione di tutti i pesi degli archi al minimo strettamente positivo del range di pesi dell'algoritmo. Studieremo 3 algoritmi per trovare il cammino minimo tra due nodi: Floyd-Warshall, Dijkstra e Bellman-Ford.

Chapter 13

Floyd-Warshall APSP

13.1 l'Algoritmo

Questo algoritmo risolve il All Pairs Shortest Path. Un cammino da v_i a v_j e' una sequenza finita di vertici $s \subseteq V$, con $s_1 = v_i$ e $s_{|s|} = v_j$, tali che $\forall i \in [1, |s|) \mid \exists e \in E \ \ con \ \ e = (s_i, s_{i+1})$. Il peso del cammino e' la somma dei pesi degli archi che lo compongono.

Questo algoritmo prende in input i pesi sottoforma di Matrice di Adiancenza, le cui celle:

$$\begin{aligned} i = j &\Rightarrow w_{i,j} = 0 \\ i \neq j \wedge (i,j) \in E &\Rightarrow w_{i,j} = p_{i,j} \\ i \neq j \wedge (i,j) \notin E &\Rightarrow w_{i,j} = \infty \end{aligned}$$

L'algoritmo restituisce due Matrici di Adiacenza tali che:

$$\begin{aligned} i = j &\Rightarrow d_{i,j} = 0 \\ i \neq j \wedge \exists cammino(i,j) &\Rightarrow d_{i,j} = \text{peso del cammino} \\ i \neq j \wedge \nexists cammino(i,j) &\Rightarrow d_{i,j} = \infty \end{aligned}$$

$$\begin{aligned} i = j &\Rightarrow \pi_{i,j} = NIL \\ i \neq j \wedge \exists cammino(i,j) &\Rightarrow \pi_{i,j} = u \wedge (u,j) \in E \\ i \neq j \wedge \nexists cammino(i,j) &\Rightarrow \pi_{i,j} = NIL \end{aligned}$$

13.2 Riempimento di $d_{i,j,k}$

Ora si ipotizzi di disporre per il cammino minimo intermedio a (v_i, v_j) di tutti e soli i primi k vertici. La matrice $d_{i,j,k}$ quindi contiene il peso del cammino minimo che dispone dei primi k vertici.

Caso Base

$$d_{i,j,k} = \begin{cases} 0 & k = 0 \wedge i = j \\ w_{i,j} & k = 0 \wedge (i,j) \in E \\ \infty & k = 0 \wedge (i,j) \notin E \end{cases}$$

Questo corrisponde a dire che $d_{i,j,0} = w_{i,j}$. Ad ogni modo la soluzione sara' contenuta in $d_{i,j,n}$.

Caso Ricorsivo Ora si presentano due scenari.

$$k \in cammino \Rightarrow d_{i,j,k} = d_{i,k,k-1} + d_{k,j,k-1}$$
$$k \notin cammino \Rightarrow d_{i,j,k} = d_{i,j,k-1}$$

Visto che si tratta del cammino minimo, allora $k \in cammino$ sse $d_{i,k,k-1} + d_{k,j,k-1} < d_{i,j,k-1}$. Quindi:

$$d_{i,j,k} = \begin{cases} d_{i,k,k-1} + d_{k,j,k-1} & d_{i,k,k-1} + d_{k,j,k-1} \leq d_{i,j,k-1} \\ d_{i,j,k-1} & d_{i,k,k-1} + d_{k,j,k-1} > d_{i,j,k-1} \end{cases}$$

13.3 Riempimento di $\pi_{i,j,k}$

Caso Base

$$\pi_{i,j,k} = \begin{cases} NIL & k = 0 \wedge i = j \\ i & k = 0 \wedge (i,j) \in E \\ NIL & k = 0 \wedge (i,j) \notin E \end{cases}$$

Caso Ricorsivo Usiamo ora il caso ricorsivo di $d_{i,j,k}$:

$$\pi_{i,j,k} = \begin{cases} \pi_{i,j,k-1} & d_{i,k,k-1} + d_{k,j,k-1} < d_{i,j,k-1} \\ \pi_{k,j,k-1} & d_{i,k,k-1} + d_{k,j,k-1} \geq d_{i,j,k-1} \end{cases}$$

Come prima la soluzione $\pi_{i,j} = \pi_{i,j,n}$.

13.4 Procedura FW

Tempi di calcolo Il riempimento delle matrici D, K e' $T(n) = \Theta(n^3)$.

```
procedure FLOYD-WARSHALL(V, E, W)
    D^0 ← W
    for i = 1 to n do
        for j = 1 to n do
            if i ≠ j ∧ W[i, j] ≠ ∞ then
                Π^0[i, j] ← i
            else
                Π^0[i, j] ← NIL
            end if
        end for
    end for
    for k = 1 to n do
        for i = 1 to n do
            for j = 1 to n do
                a ← D^{k-1}[i, k] + D^{k-1}[k, j]
                b ← D^{k-1}[i, j]
                if a < b then
                    D^k[i, j] ← a
                    Π^k[i, j] ← Π^{k-1}[k, j]
                else
                    D^k[i, j] ← b
                    Π^k[i, j] ← Π^{k-1}[i, j]
                end if
            end for
        end for
    end for
end procedure
```

```
procedure STAMPA-CAMMINO(Π, i, j)
    if Π[i, j] ≠ NIL then
        STAMPA-CAMMINO(Π, Π[i, j], j)
        PRINT(j)
    end if
end procedure
```

Chapter 14

Bellman-Ford

Questo algoritmo trova tutti i minimi cammini tra due generici nodi x e y.

14.1 Algoritmo

Come in *Dijkstra* si pre-impostano distanze e predecessori. Quindi si processano tutti gli archi del grafo per un numero di volte pari a $|V|$. Per ogni arco (u, v, d) si controlla la distanza (x, v) vincolata a passare in u e si aggiornano le informazioni su distanza minima e predecessore se questo passaggio risulta fin'ora vantaggioso.

14.2 Procedura

Sia $G = (V, E)$ con $E = \{(v_a, v_b, peso)\}$.

Si mantiene un array D delle minime distanze da x al generico nodo $v \in V \setminus x$.

Si mantiene un array H dove ogni cella rappresenta il predecessore del generico nodo $v \in V \setminus x$ nel percorso minimo tra x e v.

---

procedure BELLMAN-FORD(V, E, x, y)
  for $i = 1$ to n **do**
    $D[v_i] \leftarrow \infty$
    $H[v_i] \leftarrow NIL$
  end for
  $D[v_x] \leftarrow 0$
  for $i = 1$ to n **do**
    for $(u, v, d) \in E$ **do**
      $length \leftarrow D[u] + d$
      if $D[v] > length$ **then**
        $D[v] \leftarrow length$
        $H[v] \leftarrow u$
      end if
    end for
  end for
  return $D[v_y]$
end procedure

---

procedure STAMPA-RICORSIVA(H, c)
  if $H[v_c] \neq NIL$ **then** STAMPA-RICORSIVA($H, H[v_c]$)
  end ifprint(v_c)
end procedure

Complessita' Sia $n = |V|$ e $m = |E|$.

La complessita' di BELLMAN-FORD e' $O(n \times m)$, la complessita' di STAMPA-RICORSIVA e' O(n).

Chapter 15

Dijkstra

Questo algoritmo trova tutti i percorsi minimi tra un nodo $v_x \in V$ e tutti gli altri nodi $v_y \in V \setminus x$.

Si puo' modificare questo algoritmo per cercare un certo percorso tra due nodi definiti in partenza $v_x \in V$ e $v_y \in V \setminus x$, semplicemente fermandosi quando il percorso minimo viene trovato dall'algoritmo.

15.1 Algoritmo

Si marcano tutti i nodi come non-visitati. Quindi si impostano le distanze minime a ∞, tranne $d(x, x) = 0$.

Quindi si preleva ad ogni turno il nodo non-visitato c con distanza minima piu' bassa e si notano le lunghezze totali dei percorsi verso i nodi v adiacenti a c, (x, v), vincolato a passare per c. Se questo percorso e' migliore di quello fin'ora trovato per (x, v), si aggiorna questa informazione e si segna il passaggio per c.

Quando tutti gli archi verso nodi adiacenti sono stati processati, si segna come visitato il nodo corrente c e si preleva il prossimo.

Quando y e' stato visitato o non rimangono nodi da visitare, l'algoritmo termina.

15.2 Procedura

Si mantiene un array D delle minime distanze da x al generico nodo $v \in V \setminus x$.

Si mantiene un array H dove ogni cella rappresenta il predecessore del generico nodo $v \in V \setminus x$ nel percorso minimo tra x e v.

Si mantiene una priority queue dei nodi non visitati ordinati in base alla minima distanza in D.

Utilizzo le Liste di Adiacenza L_n del generico nodo $n \in V$.

```
procedure DIJKSTRA(V, L, x, y)
    for i = 1 to n do
        D[v_i] ← ∞
        H[v_i] ← NIL
    end for
    D[v_x] ← 0
    Q ← PriorityQueue(V)
    c ← x
    while v_y ∈ Q do
        for (v, d) ∈ L_c do
            length ← D[v_c] + d
            if length < D[v] then
                D[v] ← length
                H[v] ← c
            end if
        end for
        c ← dequeue(Q)
    end while
    return D[v_y]
end procedure
```

```
procedure STAMPA-RICORSIVA(H, c)
    if H[v_c] ≠ NIL then STAMPA-RICORSIVA(H, H[v_c])
    end ifprint(v_c)
end procedure
```

Complessita' Definisco h come la media del numero di nodi adiacenti per un generico nodo v. Definisco $n = |V|$.

La complessita' di DIJKSTRA e' $O(n \times h)$, la complessita' di STAMPA-RICORSIVA e' O(n).

Chapter 16

Insiemi Disgiunti

16.1 Operazioni

makeset(x) Crea un insieme con rappresentante l'elemento x.

union(x, y) Unisce due set che hanno come rappresentanti rispettivamente gli elementi x e y.

findset(x) Trova il set che ha come rappresentante l'esemento x.

16.2 Componenti Connesse di Grafi

---

procedure COMP-CONN(V, E)
  for $v \in V$ **do**
    $makeset(v)$
  end for
  for $(v_i, v_j) \in E$ **do**
    if $findset(v_i) \neq findset(v_j)$ **then**
      $union(v_i, v_j)$
    end if
  end for
end procedure

Chapter 17

Albero di Copertura

Dato un grafo connesso e non orientato G = (V, E), un Albero di Copertura (o di Supporto) per esso e' un albero il cui insieme dei vertici e' coincidente con l'insieme dei nodi del gravo V, e dove gli archi che collegano i vertici padre con i vertici figlio siano un sottoinsieme dell'insieme degli archi del grafo E.

Dato un grafo pesato, connesso e non orientato G = (V, E), sia A l'insieme degli Alberi di Copertura di G.

Il peso dell'Albero di Copertura $a \in A$ si valuta $p(a) = \Sigma_{(i,j)\in a.E} w(v_i, v_j)$.

L'Albero di Copertura (abbr. AC o AS) s, se vale $\forall a \in A \mid p(s) \leq p(a)$ allora si dice Albero di Copertura Minimo (abbr. ACM).

Studieremo due algoritmi ottimi per trovare gli ACM: Kruskal e Prim.

Chapter 18

Kruskal

18.1 l'Algoritmo

L'algoritmo di Kruskal e' una rielaborazione dell'algoritmo per le Componenti Connesse di un grafo. Sia un grafo G = (E, V).
Creo una foresta di $|V|$ alberi, ognuno contiene solo un vertice del grafo, tramite la struttura dati degli Insiemi Disgiunti.
L'Albero di Copertura Minimo F e' rappresentato tramite una lista di archi.
Si itera sugli archi ordinati in senso crescente rispetto al peso. Ad ogni iterazione si cerca di collegare due alberi disgiunti della foresta. Visto che si visitano per primi gli archi piu' leggeri, e' triviale che l'Albero di Copertura cosi' generato sara' minimo.

18.2 Procedura K

---

procedure K(V, E)
  $F \leftarrow \emptyset$
  for $v \in V$ **do**
    $makeset(v)$
  end for
  for $(v_i, v_j) \in SORT_{weight}(E)$ **do**
    if $findset(v_i) \neq findset(v_j)$ **then**
      $union(v_i, v_j)$
      $F \leftarrow F \cup \{(v_i, v_j)\} \cup \{(v_j, v_i)\}$
    end if
  end for
end procedure

Chapter 19

Prim

19.1 Algoritmo

Sia un grafo, connesso, non orientato e pesato G = (V, E) su cui si voglia calcolare l'ACM. Per astrazione si consideri l'albero A = ($A_V \subset V$, $A_E \subset E$) che sia garantito essere un albero di copertura di parte del grafo. E' possibile costruire la soluzione espandendo ad ogni iterazione l'estensione di A_V.

Sia $A_n = ACM(G = (V_n, E_n))$, con $V_n = V[1 \ : \ n]$ e $E_n = \{(v_i, v_j) \mid (v_i, v_j) \in E \wedge v_i \in V_n \wedge v_i \in V_n\}$.

Ora, $V_n = V_{n-1} \cup v_n \Rightarrow v_n \notin V_{n-1}$ e quindi $\forall (v_n, v_i) \in E \mid (v_n, v_i) \notin A_{n-1}$.

Quindi supponendo di voler collegare l'albero A_{n-1} a v_n si usera' l'arco di peso minimo $(v_n, v_i) \in E$.

Si supponga per assurdo che l'arco corretto sia invece $(v_n, v_j) \neq (v_n, v_i)$. Allora $(v_n, v_i) < (v_n, v_j) \Rightarrow p(A_{n-1}) + w((v_n, v_i)) < p(A_{n-1}) + w(v_n, v_j)$. Quindi necessariamente $p(A_{n-1}) + w((v_n, v_i)) < p(A_{n-1}) + w(v_n, v_j) \Rightarrow p(A_{ni}) < p(A_{nj})$, ma cio' contravviene al concetto stesso di ACM, quindi l'arco corretto e' sempre quello di peso minimo.
Ora, sfortunatamente non e' sempre vero $\forall i, j \in V \mid (v_i, v_j) \in E$, quindi esite un ordine non garantito dalla consequenzialita' all'interno di V. E' quindi necessario valutare tutti gli archi nel grafo agganciati a nodi non connessi all'ACM parziale per verificare quale sia piu' vantaggioso tra quelli raggiungibili esclusivamente tramite i nodi gia' nell'ACM.

Visto che il grafo e' connesso, e' ininfluente da quale nodo si parte, puo' essere scelto in modo arbitrario. Se il grafo non fosse connesso, la reiterazione di questo algoritmo su tutti i nodi non ancora identificati in un ACM porta a trovare tutte le componenti connesse di un grafo.

A differenza del Kruskal, l'ACM e' rappresentato tramite A_V e A_E.

19.2 Procedura P

```
procedure P(V, E)
    A_V ← {v_0}
    A_E ← ∅
    while A_V ≠ V do
        best_a ← NIL
        best_b ← NIL
        best_w ← NIL
        for (a, b) ∈ E do
            if a ∈ A_V ∧ b ∉ A_V then
                if best_a = NIL ∨ best_w > w(a, b) then
                    best_w ← w(a, b)
                    best_a ← a
                    best_b ← b
                end if
            end if
        end for
        A_V ← A_V ∪ {b}
        A_E ← A_E ∪ {(a, b)}
    end while
    return (A_V, A_E)
end procedure
```

19.3 Considerazioni

Tempi di Calcolo L'algoritmo e' costutuito da un ciclo *while* e un ciclo for innestati. Ad ogni passo di *while* si esplorano tutti gli archi in E e si verificano per essi dei requisiti. Visto che tendenzialmente il contenuto del for e' $\Theta(1)$, si conclude che il contenuto del *while* e' $\Theta(|E|)$. Inoltre si deve considerare che i nodi vengono aggiunti uno ad uno all'interno di A_V, quindi i passi del while saranno $\Theta(|V|)$, per un totale di $T(n) = \Theta(|E| \times |V|)$.

Ottimizzazioni Ovviamente bisogna considerare che questo tempo e' ottimizzabile. Per esempio e' possibile iterare sui soli archi che appartengono ai nodi del ACM, in modo da risparmiare soprattutto nelle prime iterazioni notevole tempo. Questo porterebbe il tempo ad essere $T(n) = O(|E| \times |V|)$.

Implementazioni Addirittura utilizzando diverse implementazioni per memorizzare le informazioni si puo' ottimizzare accessi e ricerche. Fonte: Wikipedia

- Con la Matrice di Adiacenza per esempio diventa $T(n) = O(|V|^2)$.

- Con le Liste di Aidacenza e il Binary Heap diventa $T(n) = O(|E|log|V|)$.
- Con le Liste di Aidacenza e il Fibonacci Heap diventa $T(n) = O(|E| + |V|log|V|)$.

Part II

Esercitazione

Chapter 20

Applicazione di PD

20.1 Distanza di Edit

Introduzione E' definita come numero minimo di **cancellazioni**, **sostituzioni**, **inserimenti** che trasformano una stringa X in una seconda stringa Y. E' un problema simmetrico. $D_{edit}(X, Y) = D_{edit}(Y, X)$.

Esempio $X =< 2, 4, 10, 3, 1 > Y =< 2, 4, 2, 1 >$

- Cancellazione di 10.
- Sostituisco 3 con 2.

Quindi $D_{edit}(X, Y) = 2$.

- Inserimento di 10.
- Sostituisco 2 con 3.

Quindi $D_{edit}(Y, X) = 2$.

- Cancellazione di 3.
- Sostituisco 10 con 2.

Possono non esistere soluzioni uniche.

Problemi

PR Date due sequenze X, Y, trovare la distanza di edit di X in Y.

P Date due sequenze X, Y, trovare il minimo insieme di operazioni di cancellazione, inserimento, sostituzione che trasformano X in Y.

Soluzione

Sottoproblema di PR Trovare la distanza di edit dei prefissi X_i e Y_j. Il numero di sottoproblemi e' $(m+1) \cdot (n+1)$. $d_{i,j} = distanzadieditdeiprefissiX_iY_j$. Soluzione 'e $d_{m,n}$

Casi base di PR

- $i = 0 \wedge j = 0 \Rightarrow d_{0,0} = 0$
- $i > 0 \wedge j = 0 \Rightarrow d_{i,0} = i$
- $i = 0 \wedge j > 0 \Rightarrow d_{0,j} = j$

Caso ricorsivo di PR Con $i > 0 \wedge j > 0$:

$$d_{i,j} = \begin{cases} d_{i-1,j-1} & x_i = y_j \\ min(d_{i-1,j-1}, d_{i,j-1}, d_{i-1,j}) + 1 & altrimenti \end{cases}$$

Il caso "altrimenti" si spiega in questo modo:

- $Sostituzione(x_i \rightarrow y_j) + d_{i-1,j-1}$
- $Inserimento(y_j) + d_{i,j-1}$
- $Cancellazione(x_i) + d_{i-1,j}$

20.2 Algoritmo TOP-DOWN

---

procedure ED-RIC(X, Y)
  if $j = 0$ **then**
    return i
  else if $i = 0$ **then**
    return j
  else
    $A \leftarrow ED - RIC(X_{i-1}, Y_j) + 1$
    $B \leftarrow ED - RIC(X_i, Y_{j-1}) + 1$
    $C \leftarrow ED - RIC(X_{i-1}, Y_{j-1}) + 1$
    if $A \leq B \wedge A \leq C$ **then**
      return A
    else if $B \leq A \wedge B \leq C$ **then**
      return B
    else
      return C
    end if
  end if
end procedure

Complessita' La complessita' dell'algoritmo ricorsivo e' (3^{n+m}).

20.3 Algoritmo BOTTOM-UP

Preparo una Matrice $B[m][n]$ che contiene la ricostruzione del percorso iterativo. Ogni cella $B[i][j] =< Direction, Operation >$

---

procedure ED-ITER-RM(m, n)
  for $i = 0$ to m **do**
    $M[i][0] \leftarrow i$
    $B[i][0] \leftarrow <$ " $\uparrow$ ", $Null >$
  end for
  for $j = 0$ to n **do**
    $M[0][j] \leftarrow j$
    $B[0][j] \leftarrow <$ " $\leftarrow$ ", $Null >$
  end for
end procedure

---

procedure ED-ITER(X, Y)
  $ED - ITER - RM(m, n)$
  for $i = 1$ to m **do**
    for $j = 0$ to n **do**
      if $x_i = y_j$ **then**
        $M[i][j] \leftarrow M[i-1][j-1]$
        $B[i][j] \leftarrow <$ "", $Null >$
      else if $M[i-1][j] \leq M[i][j-1] \wedge M[i-1][j] \leq M[i-1][j-1]$ **then**
        $M[i][j] \leftarrow M[i-1][j] + 1$
        $B[i][j] \leftarrow <$ " $\uparrow$ ", $Delete >$
      else if $M[i][j-1] \leq M[i-1][j] \wedge M[i][j-1] \leq M[i-1][j-1]$ **then**
        $M[i][j] \leftarrow M[i][j-1] + 1$
        $B[i][j] \leftarrow <$ " $\leftarrow$ ", $Insert >$
      else
        $M[i][j] \leftarrow M[i-1][j-1] + 1$
        $B[i][j] \leftarrow <$ " $\nwarrow$ ", $Change >$
      end if
    end for
  end for
  return $M[m][n]$
end procedure

Complessita' di $ED - ITER(X, Y)$ L'algoritmo e' formato da due cicli innestati, quindi $T(n) = \Theta(n \cdot m)$.

Chapter 21

Problema LZS

21.1 Introduzione

Il problema Longest Zig-Zag Subsequence consiste nel trovare una sottosequenza piu' lunga tale che

$$\forall i \in [0, |LZS(X)|) \mid P : \begin{cases} z_i < z_{i+1} & dispari(i) \\ z_i > z_{i+1} & pari(i) \end{cases}$$

21.2 Ragionamento

Ragiono per prefissi. Sia X_i il prefisso i-esimo della sequenza di input. Sia $Z_k = LZS(X)$ e quindi $S_i = LZS(X_i)$.

Se $0 \leq i < 2$ allora $Z_k = X_i$ e $S_i = X_i = Z_k$.
Questo perche' banalmente LZS(<>) = <> e LZS(<α>) = <α>.

Mi rifaccio ai due sotto problemi $LZS_{p:i}$, $LZS_{d:i}$ che risolvono rispettivamente la LZS pari vincolata a terminare con x_i e la LZS dispari vincolata a terminare con x_i.

Assegno $S_i = max\{LZS_{p:i}, LZS_{d:i}\}$. Trovare $LZS_{p:i}$ e $LZS_{d:i}$ a questo punto coincide con il cercare il S_i compatibile piu' lungo.
Quindi, $Z_k = max_{length}\{S_i \mid \forall i \in [1, n]\}$.

Caso Base

$$\begin{cases} <> & i = 0 \\ < x_1 > & i = 1 \end{cases}$$

Caso Ricorsivo

$$\begin{cases} LZS_{p:i} & i > 1 \wedge z_k < x_i \wedge |LZS_{p:i}| > |LZS_{d:i}| \\ LZS_{d:i} & i > 1 \wedge z_k > x_i \wedge |LZS_{p:i}| \leq |LZS_{d:i}| \end{cases}$$

21.3 Procedura

```
procedure LZS(X)
    C[1] ← 1
    H[1] ← 0
    lenLZS ← 1
    for i = 2 to n do
        max_p, max_d, h_p, h_d ← 0
        for h = 1 to i − 1 do
            if pari(C[h]) ∧ C[h] > max_p ∧ x_h > x_i then
                max_p ← C[h]
                h_p ← h
            else if dispari(C[h]) ∧ C[h] > max_d ∧ x_h < x_i then
                max_d ← C[h]
                h_d ← h
            end if
        end for
        if max_p > max_d then
            C[i] ← max_p + 1
            H[i] ← h_p
        else
            C[i] ← max_d + 1
            H[i] ← h_d
        end if
        if C[i] > lenLZS then
            lenLZS ← C[i]
        end if
    end for
    return lenLZS
end procedure
```

```
procedure RICOSTRUISCI(X, C, H)
    i ←trova-indice(C, lenLZS)
    while i > 0 do
        print(x_i)
        i ← H[i]
    end while
end procedure
```

Complessita' La ricerca del massimo sottoproblema richiede $T_s(n) = (n)$. Quindi l'algoritmo ha complessita' totale $T(n) = (n^2)$.

Chapter 22

Problema LCZS

22.1 Introduzione

Il problema Longest Common Zig-Zag Subsequence consiste nel trovare una sottosequenza piu' lunga comune tale che:

$$\begin{cases} z_i < z_{i+1} & dispari(i) \\ z_i > z_{i+1} & pari(i) \end{cases}$$

22.2 Ragionamento

E' una sovrastruttura del concetto base di LCS e al tempo stesso di LZS. Di conseguenza si replica la sue basi per il ragionamento di LZCS.

Ragiono per prefissi. Siano X_i, Y_j i prefissi i,j-esimi della sequenze di input. Sia $Z_k = LCZS(X, Y)$ e quindi $S_{i,j} = LCZS(X_i, Y_j)$.

Se $i = 0 \vee j = 0$ allora $Z_k = X_i$ e $S_i = X_i = Z_k$.
Questo perche' banalmente:
LCZS(<α>, <>) = <>
LCZS(<>, <α>) = <>
LCZS(<>, <>) = <>.

Seguendo il ragionamento LCS e' vero che:

$$x_i \neq y_j \Rightarrow LCZS(X_i, Y_j) = max\{LCZS(X_{i-1}, Y_j), LCZS(X_i, Y_{j-1})\}$$

Tuttavia sara' necessario sapere solo il suo opposto, ovvero quanto segue.
Allo stesso modo di prima si asserisce che $x_i = y_j$ implica che x_i sia un candidato per LCZS.

Mi rifaccio ai due sotto problemi $LCZS_{p:i,j}$, $LCZS_{d:i,j}$ che risolvono rispettivamente la LCZS pari vincolata a terminare con $x_i = y_j$ e la LCZS dispari vincolata a terminare con $x_i = y_j$.

Questo implica necessariamente che:

$$x_i \neq y_j \Rightarrow LCZS_{p:i,j} = LCZS_{d:i,j} = <>$$

Quindi assegno $S_{i,j} = max\{LCZS_{p:i,j}, LCZS_{d:i,j}\}$. Trovare $LCZS_{p:i,j}$ e $LCZS_{d:i,j}$ a questo punto coincide con il cercare il $S_{i,j}$ compatibile piu' lungo.
Quindi, $Z_k = max_{length}\{S_{i,j} \mid \forall i \in [1,n], j \in [1,m]\}$.

Caso Base

$$\begin{cases} <> & i = 0 \vee j = 0 \end{cases}$$

Caso Ricorsivo

$$\begin{cases} LCZS_{p:i} & i,j \geq 1 \wedge x_i = y_j \wedge z_k < x_i \wedge |LCZS_{p:i}| > |LCZS_{d:i}| \\ LCZS_{d:i} & i,j \geq 1 \wedge x_i = y_j \wedge z_k > x_i \wedge |LCZS_{p:i}| \leq |LCZS_{d:i}| \end{cases}$$

22.3 Procedura

```
procedure LCZS(X, Y)
    lenLCZS ← 0
    for i = 1 to n do
        for j = 1 to m do
            H[i][j] ← (−1, −1)
            if x_i ≠ y_j then
                C[i][j] ← 0
            else
                max_p, max_d ← 0
                head_p, head_d ← (0, 0)
                for h = 1 to i − 1 do
                    for k = 1 to i − 1 do
                        if pari(C[h][k]) ∧ C[h][k] > max_p ∧ x_h > x_i then
                            max_p ← C[h][k]
                            head_p ← (h, k)
                        else if dispari(C[h][k]) ∧ C[h][k] > max_d ∧ x_h < x_i then
                            max_d ← C[h][k]
                            head_d ← (h, k)
                        end if
                    end for
                end for
                if max_p > max_d then
                    C[i][j] ← max_p + 1
                    H[i][j] ← head_p
                else
                    C[i][j] ← max_d + 1
                    H[i][j] ← head_d
                end if
                if C[i][j] > lenLCZS then
                    lenLCZS ← C[i][j]
                end if
            end if
        end for
    end for
    return lenLZS
end procedure
```

Complessita' La ricerca del massimo sottoproblema richiede $T_s(n, n) = (n \cdot m)$. Quindi l'algoritmo ha complessita' totale $T(n) = (n^2 \cdot m^2)$.

---

procedure RICOSTRUISCI(X, C, H)
    $(i, j) \leftarrow$trova-indici$(C, lenLZS)$
    while $(i, j) \neq (0, 0)$ **do**
        $print(x_i)$
        $(i, j) \leftarrow H[i][j]$
    end while
end procedure

Chapter 23

Problema LCSL

23.1 Introduzione

LCSL sta per Longest Common Subsequence $\leq$ L. E' il problema decisionale che intende stabilire se la LCS tra X e Y ha lunghezza al piu' L.

23.2 Ragionamento

per $i > 0, j > 0, l = 0, x_i = y_j$, $c_{i,j,l}$ = false

per $i = 0, j > 0, l \geq 0$, $c_{i,j,l}$ = true

per $i > 0, j = 0, l \geq 0$, $c_{i,j,l}$ = true

per $i > 0, j > 0, l > 0, x_i = y_j$, $c_{i,j,l} = c_{i-1,j-1,l-1}$

per $i > 0, j > 0, l \geq 0, x_i \neq y_j$, $c_{i,j,l} = c_{i-1,j,l} \wedge c_{i,j-1,l}$

$$c_{i,j,l} = \begin{cases} false & i > 0 \wedge j > 0 \wedge l = 0 \wedge x_i = y_j \\ true & i = 0 \wedge j > 0 \wedge l \geq 0 \\ true & i > 0 \wedge j = 0 \wedge l \geq 0 \\ c_{i-1,j-1,l-1} & i > 0 \wedge j > 0 \wedge l > 0 \wedge x_i = y_j \\ c_{i-1,j,l} \wedge c_{i,j-1,l} & i > 0 \wedge j > 0 \wedge l \geq 0 \wedge x_i \neq y_j \end{cases}$$

23.3 Procedura

```
procedure RIEMPI-MATRICI(X, Y, L)
    for l = 0 to L do
        for i = 0 to |X| do
            C_l[i, 0] ← true
        end for
        for j = 0 to |Y| do
            C_l[0, j] ← true
        end for
        for i = 1 to |X| do
            for j = 1 to |Y| do
                if x_i = y_j then
                    if l = 0 then
                        C_l[i, j] ← false
                    else
                        C_l[i, j] ← C_{l-1}[i − 1, j − 1]
                    end if
                else
                    C_l[i, j] ← C_l[i − 1, j] ∧ C_l[i, j − 1]
                end if
            end for
        end for
    end for
end procedure
```

Chapter 24

Problema KC

24.1 Introduzione

il problema Knapsack Colorato e' una variante di Knapsack in cui ad ogni oggetto si associa un colore (rosso o blu) in cui la soluzione deve contenere al piu' oggetti rossi.

24.2 Ragionamento

Definizione del generico sottoproblema Sottoproblema $d_{i,c,r} \Rightarrow KC(X_i, c \leq C, r \leq R)$

Individuazione del caso base per $i = 0, c \geq 0, r \geq 0$, $d_{i,c,r} = 0$

per $i \geq 0, c = 0, r \geq 0$, $d_{i,c,r} = 0$

Individuazione del caso ricorsivo per $i > 0, c > 0, r \geq 0, w_i > c$, $d_{i,c,r} = d_{i-1,c,r}$

per $i > 0, c > 0, r = 0, col(i) = rosso$, $d_{i,c,r} = d_{i-1,c,r}$

per $i > 0, c > 0, r > 0, col(i) = rosso$, $d_{i,c,r} = \max\ (d_{i-1,c-w_i,r-1} + w_i,\ d_{i-1,c,r})$

per $i > 0, c > 0, r \geq 0, col(i) \neq rosso$, $d_{i,c,r} = \max\ (d_{i-1,c-w_i,r} + w_i,\ d_{i-1,c,r})$

24.3 Procedura

Chapter 25

Problema LCSDP

25.1 Introduzione

LCSDP e' una LCS in cui i valori dispari sono in posizione dispari e i valori pari sono in posizione pari.

Part III

Laboratorio

Chapter 26

Weighted Interval Scheduling

Data: 11-10-2022

26.1 Scheduling di Attivita'

Un esempio di Programmazione per Intervalli Pesati.
Siano date n attivita' da svolgersi nello stesso spazio fisico. Determinare un sottoinsieme di attivita' che non si sovrappongono e che sia il massimo possibile.

i	p(i)	v
1	0	10
2	0	2
3	2	8
4	2	1
5	1	1
6	4	3

Ad occhio si ricava $< 1, 3, 6 >$.

L'algoritmo naive e' quello combinatorio, ma e' estremamente inefficiente. Il tempo e' $T(n) = \Omega(2^n)$.

26.2 Soluzione PD

- $n \leftrightarrow X = 1, .., n$
- $\forall i \in 1, .., n$, s_i e' il tempo di inizio dell'attivita' i
- $\forall i \in 1, .., n$, f_i e' il tempo di fine dell'attivita' i
- $\forall i \in 1, .., n$, v_i e' il valore dell'attivita' i

Definisco la funzione:
$COMP : \mathbb{P}(\{1, .., n\}) \to \{true, false\}$
$\forall i, j \in A \mid i = j \vee attivitaCompatibili(A_i, A_j) \Rightarrow COMP(A) = true$

Dette poi i, j due attivita' si dice:

$$attivitaCompatibili(i,j) = \begin{cases} true & [s_i, f_i) \cap [s_j, f_j) = \emptyset \\ false & altrimenti \end{cases}$$

Quindi si definisce:
$V : \mathbb{P}(\{1, .., n\}) \to \mathbb{R}$

$$V(i,j) = \begin{cases} \Sigma_{i \in A} v_i & A \neq \emptyset \\ 0 & A = \emptyset \end{cases}$$

La Soluzione e' $(S \subseteq X \mid COMP(S) = true) \wedge (\forall A \subseteq X \mid V(S) \geq V(A))$

Processo Detto $S_n \Leftrightarrow sol(X_n)$, e quindi $S_{n-k} \Leftrightarrow sol(X_{n-k})$. Nella soluzione di S_n si assume di conoscere:

- $\forall k \in 1, .., n \; S_{n-k}$
- $\forall k \in 1, .., n \; sol(X_{n-k})$

Detto $OPT(i) = V(S_i)$. Dividendo in sottoproblemi e potendo disporre a piacimento di ognuno di questi, riesco facilmente a individuare il caso base. E' immediato sapere $sol(\emptyset)$ e $sol(1)$, quindi questi possono essere i casi base.

Ragionamento

Caso Base

$$S_0 \Leftrightarrow X_0 = \emptyset \wedge V(X_0) = 0 \tag{26.1}$$
$$S_1 \Leftrightarrow X_1 = x_1 \wedge V(X_1) = 1 \tag{26.2}$$

Caso Ricorsivo Voglio risolvere S_i e $OPT(i)$.
Assumo di avere gia' risolto $\forall j \in \{0, .., i-1\} \mid S_j$.

Se sapessi che $i \notin S_i$ allora dovrei risolvere S_{i-1}.
Se sapessi al contrario che $i \in S_i$ il sotto problema da considerare sarebbe:
$sol(\{j \mid \forall j \in X_{i-1} \mid attivitaCompatibile(i,j)\})$.
Questo si traduce nel risolvere S_j, dove j e' il massimo indice di una attivita' compatibile con i.

Detto $p(i) : X \to X$, la funzione che associa ad ogni attivita', l'indice dell'attivita' compatibile precedente piu' vicina. Potendo approssimare, per questioni di prestazione dei confronti, il problema S_j ad al problema leggermente piu' grande $S_{p(i)}$, mi riduco a:

$$V(p(i)) + v_i \geq V(i-1) \Rightarrow S_{p(i)} \cup \{x_i\} \tag{26.3}$$
$$V(p(i)) + v_i < V(i-1) \Rightarrow S_{i-1} \tag{26.4}$$
$$\tag{26.5}$$

Che verranno dimostrati successivamente. Il tutto si traduce in:

$$S_i = \begin{cases} \emptyset & i = 0 \\ \{x_1\} & i = 1 \\ S_{p(i)} \cup \{x_i\} & V(p(i)) + v_i \geq V(i-1) \\ S_{i-1} & V(p(i)) + v_i < V(i-1) \end{cases}$$

Ovvero, riprendo i casi base esattamente come scritti sopra, e in piu' vincolo la scelta della soluzione (e di fatto del sottoproblema S_{i-1} o $S_{p(i)}$) all'unico criterio che e' in grado di stabilire la buona riuscita dell'ottimizzazione.

Dimostrazione Per dimostrare il teorema mi avvalgo dei sottoproblemi di S.
Assumo quindi che $\forall j \in \{0, .., i-1\} \mid S_j$.
Inoltre assumo che il problema per X_i abbia almeno una soluzione.

1 Assumo che $i \in S_i$, devo mostrare che $S_i = S_{p(i)} \cup i$.
Si supponga per assurdo che $S_{p(i)} \cup i \neq sol(X_i)$.
Chiamo S^I la soluzione S_i.
Posso affermare con certezza che $V(S^I) > V(S_{p(i)}) + v_i$.
Visto che $i \in S_i$, deduco che allora $S^I = S^{II} \cup i$.
Ragionevolmente S^{II} contiene attivita' compatibili insieme a i.
Per la riduzione di qualche paragrafo precedente posso affermare che $S^{II} \subseteq \{1, .., p(i)\}$.
Ma allora $V(S^{II}) > V(S_{p(i)})$.
Il che e' impossibile, perche' sappiamo che $S_{p(i)} = sol(X_{p(i)})$.
Dovendo essere necessariamente $S_{p(i)} = sol(X_{p(i)})$, ne ricaviamo che $S^{II} = S_{p(i)}$. Quindi $S_i = S_{p(i)} \cup i$.

2 Assumo che $i \notin S_i$, devo mostrare che $S_i = S_{i-1}$.
Supponiamo per assurdo che $S_i \neq S_{i-1}$
Allora necessariamente $\exists S^I \neq S_{i-1} \mid V(S^I) > V(S_{i-1})$.
Visto che $S^I = S_i$ allora $i \notin S^I$, dato che $i \notin S_i$.
Ma quindi allora $S^I \subseteq 1, .., i-1 \mid V(S^I) > V(S_{i-1})$. Il che e' impossibile, perche' sappiamo che $S_{i-1} = sol(X_{i-1})$.
Dovendo essere necessariamente $S_{i-1} = sol(X_{i-1})$, ne ricaviamo che $S^I = S_{i-1}$.
Quindi $S_i = S_{i-1}$.

26.3 Procedura TOP-DOWN

```
procedure WIS-OPT(i)
    if i = 0 then
        return 0
    else if i = 1 then
        return 1
    else
        Z1 ← append(WIS − OPT(p(i)), x_i)
        Z2 ← WIS − OPT(i − 1)
        if OPT(Z1) ≥ Z2 then
            return Z1
        else
            return Z2
        end if
    end if
end procedure
```

Complessita' E' immediato intuire che l'esplorazione prolissa dei due casi paralleli porta $T(n) = O(2^n)$.

26.4 Procedura BOTTOM-UP

```
procedure INIZIALIZZA-VETTORI
    OPT[0] ← 0
    OPT[1] ← 1
    WIS[0] ← X_0
    WIS[1] ← X_1
end procedure
```

```
procedure WIS-OPT-ITER(i)
    INIZIALIZZA − VETTORI()
    for i = 2 to n do
        Z1 ← append(WIS[p(i)], x_i)
        Z2 ← WIS[i − 1]
        if OPT[p(i)] < OPT[i − 1] then
            OPT[i] ← OPT[p(i)] + v_i
            WIS[i] ← Z1
        else
            OPT[i] ← OPT[i − 1]
            WIS[i] ← Z2
        end if
    end for
    return WIS[n]
end procedure
```

Il vettore WIS e' implementabile ragionevolmente con una matrice di booleani che caratterizzano la presenza di un elemento nell'insieme. Visto che questo richiederebbe una quantita' di spazio non indifferente, una cosa comoda potrebbe essere codificare le righe o le colonne in un numero intero decimale.

Complessita' La procedura WIS-OPT-ITER comprende un solo ciclo di $\Theta(n)$ iterazioni. Il calcolo di $p(i)$ richiede $O(n)$, perche' e' un ciclo inverso semplice che dipende dalla disposizione degli elementi in X. Quindi l'algoritmo e' $T(n) = (n^2)$ nel caso medio.

Osservazioni E' possibile scrivere una procedura che esplori linearmente l'array OPT per verificare i passi che sono stati effettuati per costruire OPT. Quindi si puo' risparmiare lo spazio occupato da WIS.

26.5 Consegna Esercizio

Da risolvere per 18-10-2022.

L'istanza e' simile ma con le case al posto delle attivita'. Ci sono n case adiacenti in linea retta. Ad ogni casa e' associato un valore d, la donazione che l'abitante e' disposto a fare. Trovare un sottoinsieme di case il cui valore totale sia massimo e in cui le case non siano adiacenti.

Ragionamento Definisco la funzione:
$COMP : \mathbb{P}(\{1, .., n\}) \to \{true, false\}$
$\forall i, j \in A \mid i = j \vee caseCompatibili(A_i, A_j) \Rightarrow COMP(A) = true$

Dette poi i, j due attivita' si dice:

$$caseCompatibili(i, j) = \begin{cases} true & |i - j| > 1 \\ false & altrimenti \end{cases}$$

Quindi si definisce:
$V : \mathbb{P}(\{1, .., n\}) \to \mathbb{R}$

$$V(i, j) = \begin{cases} \Sigma_{i \in A} v_i & A \neq \emptyset \\ 0 & A = \emptyset \end{cases}$$

La Soluzione e' $(S \subseteq X \mid COMP(S) = true) \wedge (\forall A \subseteq X \mid V(S) \geq V(A))$

In questo caso $p(i)$ scorre l'array di case fino a individuare la prima non adiacente. Questa operazione e' logicamente a tempo $T(n) = \Theta(1)$, perche' si parla di spostarsi a sinistra di due case.

Dimostrazione Per dimostrare il teorema mi avvalgo dei sottoproblemi di S.
Assumo quindi che $\forall j \in \{0, .., i - 1\} \mid S_j$.
Inoltre assumo che il problema per X_i abbia almeno una soluzione.

1 Assumo che $i \in S_i$, devo mostrare che $S_i = S_{p(i)} \cup i$.
Si supponga per assurdo che $S_{p(i)} \cup i \neq sol(X_i)$.
Chiamo S^I la soluzione S_i.
Posso affermare con certezza che $V(S^I) > V(S_{p(i)}) + v_i$.
Visto che $i \in S_i$, deduco che allora $S^I = S^{II} \cup i$.
Ragionevolmente S^{II} contiene attivita' compatibili insieme a i.
Per la riduzione di qualche paragrafo precedente posso affermare che $S^{II} \subseteq \{1, .., p(i)\}$.
Ma allora $V(S^{II}) > V(S_{p(i)})$.
Il che e' impossibile, perche' sappiamo che $S_{p(i)} = sol(X_{p(i)})$.
Dovendo essere necessariamente $S_{p(i)} = sol(X_{p(i)})$, ne ricaviamo che $S^{II} = S_{p(i)}$. Quindi $S_i = S_{p(i)} \cup i$.

2 Assumo che $i \notin S_i$, devo mostrare che $S_i = S_{i-1}$.
Supponiamo per assurdo che $S_i \neq S_{i-1}$
Allora necessariamente $\exists S^I \neq S_{i-1} \mid V(S^I) > V(S_{i-1})$.
Visto che $S^I = S_i$ allora $i \notin S^I$, dato che $i \notin S_i$.

Ma quindi allora $S^I \subseteq 1,..,i-1 \mid V(S^I) > V(S_{i-1})$. Il che e' impossibile, perche' sappiamo che $S_{i-1} = sol(X_{i-1})$.
Dovendo essere necessariamente $S_{i-1} = sol(X_{i-1})$, ne ricaviamo che $S^I = S_{i-1}$.
Quindi $S_i = S_{i-1}$.

Equazione di Ricorrenza

$$S_i = \begin{cases} \emptyset & i = 0 \\ \{x_1\} & i = 1 \\ S_{p(i)} \cup \{x_i\} & V(p(i)) + v_i \geq V(i-1) \\ S_{i-1} & V(p(i)) + v_i < V(i-1) \end{cases}$$

26.6 Procedura Esercizio

```
procedure INIZIALIZZA-VETTORI
    OPT[0] ← 0
    OPT[1] ← 1
    WIS[0] ← X_0
    WIS[1] ← X_1
end procedure
```

```
procedure ESERCIZIO(i)
    INIZIALIZZA − VETTORI()
    for i = 2 to n do
        Z1 ← append(WIS[p(i)], x_i)
        Z2 ← WIS[i − 1]
        if OPT[p(i)] < OPT[i − 1] then
            OPT[i] ← OPT[p(i)] + v_i
            WIS[i] ← Z1
        else
            OPT[i] ← OPT[i − 1]
            WIS[i] ← Z2
        end if
    end for
    return WIS[n]
end procedure
```

Considerazioni Esercizio Valgono le considerazioni di WIS-OPT-ITER, ma questa volta $T(n) = \Theta(n)$, perche' la complessita' di $p(i)$ e' mutata. Inoltre si ricorda che entrambi gli esercizi godono di una proprieta' non scontata, ovvero la positivita' dei valori v_i. Se questa proprieta' non fosse stata garantita avremmo dovuto introdurre un'ulteriore caso di confronto. Quindi avremmo avuto $max(V(p(i)) + v_i, V(i-1), v_i)$.

Chapter 27

Interleaving

Data: 18-10-2022

27.1 Interleaving

Introduzione

Esempio X = $<C, I, A, O>$
Y = $<M, A, M, M, A>$
W = $<C, I, M, A, A, M, M, A, O>$

W e' un interleaving. L'algoritmo deve decidere se W e' un interleaving. Nel problema si assume che $|W| = i + j$, perche' altrimenti la soluzione nel caso specifico $|W| \neq i + j$ sarebbe triviale.

Ragionamento Definisco i prefissi X_i, Y_j, W_{i+j}. Vero sse W_{i+j} e' un interleaving.

Caso Base

$$\begin{cases} true & i = 0 \land j = 0 \\ false & i = 0 \land j > 0 \land y_j \neq w_{i+j} \\ false & j = 0 \land i > 0 \land x_i \neq w_{i+j} \\ false & i, j > 0 \land y_j \neq w_{i+j} \land x_i \neq w_{i+j} \end{cases}$$

Caso Ricorsivo

$$\begin{cases} IL(X_{i-1}, Y_j, W_{i+j-1}) & i > 0 \land (j = 0 \lor y_j \neq w_{i+j}) \land x_i = w_{i+j} \\ IL(X_i, Y_{j-1}, W_{i+j-1}) & j > 0 \land (i = 0 \lor x_i \neq w_{i+j}) \land y_j = w_{i+j} \\ IL(X_{i-1}, Y_j, W_{i+j-1}) \lor IL(X_i, Y_{j-1}, W_{i+j-1}) & i, j > 0 \land x_i = w_{i+j} \land y_j = w_{i+j} \end{cases}$$

Caso Generale

$$\begin{cases} true & i = 0 \wedge j = 0 \\ false & i = 0 \wedge j > 0 \wedge y_j \neq w_{i+j} \\ false & j = 0 \wedge i > 0 \wedge x_i \neq w_{i+j} \\ false & i, j > 0 \wedge y_j \neq w_{i+j} \wedge x_i \neq w_{i+j} \\ IL(X_{i-1}, Y_j, W_{i+j-1}) & i > 0 \wedge (j = 0 \vee y_j \neq w_{i+j}) \wedge x_i = w_{i+j} \\ IL(X_i, Y_{j-1}, W_{i+j-1}) & j > 0 \wedge (i = 0 \vee x_i \neq w_{i+j}) \wedge y_j = w_{i+j} \\ IL(X_{i-1}, Y_j, W_{i+j-1}) \vee IL(X_i, Y_{j-1}, W_{i+j-1}) & i, j > 0 \wedge x_i = w_{i+j} \wedge y_j = w_{i+j} \end{cases}$$

27.2 Procedura Bottom-Up

```
procedure INIZIALIZZA(n, m)
    M[0][0] ← true
    for i = 1 to n do
        M[i][0] ← x_i = w_i
    end for
    for j = 1 to m do
        M[0][j] ← y_j = w_j
    end for
end procedure
```

```
procedure IL(X, Y, W)
    INIZIALIZZA(n, m)
    for i = 1 to n do
        for j = 1 to m do
            if x_i = w_{i+j} then
                if y_j = w_{i+j} then
                    M[i][j] ← M[i − 1][j] ∨ M[i][j − 1]
                else
                    M[i][j] ← M[i − 1][j]
                end if
            else
                if y_j = w_{i+j} then
                    M[i][j] ← M[i][j − 1]
                else
                    M[i][j] ← false
                end if
            end if
        end for
    end for
end procedure
```

Complessita' La complessita' computazionale e' banalmente $\Theta(n \cdot m)$.

Chapter 28

Stringhe Palindrome

Data 25-10-22

28.1 Introduzione

Data una stringa S voglio il numero minimo di caratteri da aggiungere per renderla palindroma. I caratteri possono essere aggiunti in qualsiasi posizione.

Esempio 1 $S =$ ""
$PAL(S) = 0$.
caso base

Esempio 2 $S =$ "ADDA"
$PAL(S) = 0$.
E' gia' palindroma

Esempio 3 $S =$ "ARDA"
$PAL(S) = 1$.
Non e' palindroma

28.2 Ragionamento

Un approccio banale e' accodare alla stringa di partenza l'inverso di se stessa. Ma cosi' si aggiungono n caratteri. Io voglio il minimo numero possibile.

Definisco una funzione $f : \Sigma^* \to \mathbb{N}$
$\forall S \in \Sigma^*, f(S) =$ minimo numero di caratteri da aggiungere per rendere la stringa palindroma.

$S =$ "" $\Rightarrow f(S) = 0$
$\forall \alpha \in Sigma \mid S = \alpha \Rightarrow f(S) = 1$

Se $|S| > 1$ allora $S = \alpha \cdot S^I \cdot \beta$. Se $\alpha = \beta$ allora $f(S) = f(S^I)$. Se invece $\alpha \neq \beta$, $f(S) = min(f(S^I \cdot \beta), f(\alpha \cdot S^I)) + 1$.

In questo problema quindi non bastera' mantenere un indice per il prefisso, ma trattandosi di sottostringhe avremo bisogno di due indici i, j.

28.3 Algoritmo

Caso Base

$$\begin{cases} 0 & i \geq j \end{cases}$$

Caso Ricorsivo

$$\begin{cases} f(S, i+1, j-1) & i < j \wedge s_i = s_j \\ f(S, i+1, j) + 1 & i < j \wedge s_i \neq s_j \wedge f(S, i+1, j) \leq f(S, i, j-1) \\ f(S, i, j-1) + 1 & i < j \wedge s_i \neq s_j \wedge f(S, i+1, j) > f(S, i, j-1) \end{cases}$$

Caso Generale

$$\begin{cases} 0 & i \geq j \\ f(S, i+1, j-1) & i < j \wedge s_i = s_j \\ f(S, i+1, j) + 1 & i < j \wedge s_i \neq s_j \wedge f(S, i+1, j) \leq f(S, i, j-1) \\ f(S, i, j-1) + 1 & i < j \wedge s_i \neq s_j \wedge f(S, i+1, j) > f(S, i, j-1) \end{cases}$$

28.4 Procedura

```
procedure PAL(S)
    for i = n to 1 do
        for j = 1 to n do
            if i >= j then
                M[i][j] ← 0
            else if s_i = s_j then
                M[i][j] ← M[i+1][j-1]
            else if s_i ≠ s_j then
                if M[i+1][j] ≤ M[i][j-1] then
                    M[i][j] ← M[i+1][j] + 1
                else
                    M[i][j] ← M[i][j-1] + 1
                end if
            end if
        end for
    end for
    return M[|S|][|S|]
end procedure
```

Complessita' $T(n) = \Theta(n^2)$

Chapter 29

Esercizio Genoa

Determinare la lunghezza di una piu' lunga sottosequenza comune di X e Y con al massimo R simboli colorati di rosso. $COL : \Sigma \to \{rosso, blu\}$. Chiamo la sottosequenza in questione $SEQ(X, Y)$.

29.1 Ragionamento

Caso Base

$$\begin{cases} 0 & i = 0 \vee j = 0 \end{cases}$$

Caso Ricorsivo Uso una funzione $ROS : \Sigma^* \cdot \Sigma^* \to \mathbb{N}$, con $ROS(X, Y) = |\{\alpha \mid \forall \alpha \in SEQ(X, Y) \mid COL(\alpha) = rosso\}|$

$$\begin{cases} GEN(X_{i-1}, Y_{j-1}) + 1 & x_i = y_j \wedge (ROS(X_{i-1}, Y_{j-1}) < R \vee COL(x_i) \neq rosso) \\ GEN(X_{i-1}, Y_{j-1}) & x_i = y_j \wedge ROS(X_{i-1}, Y_{j-1}) = R \wedge COL(x_i) = rosso \\ GEN(X_{i-1}, Y_j) & x_i \neq y_j \wedge GEN(X_{i-1}, Y_j) \geq GEN(X_i, Y_{j-1}) \\ GEN(X_i, Y_{j-1}) & x_i \neq y_j \wedge GEN(X_{i-1}, Y_j) < GEN(X_i, Y_{j-1}) \end{cases}$$

Caso Generale

$$\begin{cases} 0 & i = 0 \vee j = 0 \\ GEN(X_{i-1}, Y_{j-1}) + 1 & x_i = y_j \wedge (ROS(X_{i-1}, Y_{j-1}) < R \vee COL(x_i) \neq rosso) \\ GEN(X_{i-1}, Y_{j-1}) & x_i = y_j \wedge ROS(X_{i-1}, Y_{j-1}) = R \wedge COL(x_i) = rosso \\ GEN(X_{i-1}, Y_j) & x_i \neq y_j \wedge GEN(X_{i-1}, Y_j) \geq GEN(X_i, Y_{j-1}) \\ GEN(X_i, Y_{j-1}) & x_i \neq y_j \wedge GEN(X_{i-1}, Y_j) < GEN(X_i, Y_{j-1}) \end{cases}$$

Chapter 30

Esercizio a Casa

Part IV

Esami

Chapter 31

Esame 10 Nov 2020

31.1 E2

Si considerino due sequenze X e Y di caratteri sull'alfabeto Σ e sia $K \in \mathbb{N}$. Sia data la funzione $\phi : \Sigma \to \mathbb{N}$ che associa ogni carattere ad un numero naturale. Si vuole calcolare mediante la programmazione dinamica la lunghezza di una LCS di X e Y nella quale la somma dei numeri associati ai caratteri sia inferiore o uguale a K. (32 punti)

Sottostruttura ottima Chiamo il problema P. Sia $L_{i,j,k} = P(X_i, Y_j, k \leq K)$.

$$LASTCHAR(L_{i,j,k}) = x_i \Leftrightarrow x_i = y_j \wedge \phi(x_i) \leq k$$

Sottoproblemi e variabili per essi Mi avvalgo del sottoproblema PR di dimensione (i, j, k) che calcola la lunghezza della LCS di prefissi X_i e Y_j con somma massima $k \leq K$. Il numero di sotto problemi e' $(i+1) \times (j+1) \times (k+1)$. Suppongo di avere a disposizione quindi $k+1$ matrici M di dimensione $(i+1) \times (j+1)$

Casi base

$$\begin{cases} 0 & i \geq 0 \wedge j = 0 \\ 0 & i = 0 \wedge j \geq 0 \end{cases}$$

Caso ricorsivo

$$\begin{cases} max(M_k[i-1, j-1], M_{k-\phi(x_i)}[i-1, j-1] + 1) & i > 0 \wedge j > 0 \wedge k >= 0 \wedge x_i = y_j \\ max(M_k[i-1, j], M_k[i, j-1]) & i > 0 \wedge j > 0 \wedge k >= 0 \wedge x_i \neq y_j \end{cases}$$

Composizione del risultato La cella $M_k[n, m]$, con $n = |X|, m = |Y|$, contiene la lunghezza della LCS di peso massimo k su X e Y. Se affianco ad essa una serie di $k+1$ matrici S di dimensione $(i+1) \times (j+1)$ e memorizzo in essa l'indicazione della mossa effettuata al passo (i,j,k).
Se nel passo (i, j, k) prendo la cella $(i-1, j, k)$ scrivo UP.
Se nel passo (i, j, k) prendo la cella $(i, j-1, k)$ scrivo LEFT.
Se nel passo (i, j, k) prendo la cella $(i-1, j-1, k-\phi(x_i))$ scrivo TAKE.
Se nel passo (i, j, k) prendo la cella $(i-1, j-1, k)$ scrivo NOTTAKE.

```
procedure FILL(X, Y, K)
    for k = 0 to K do
        for i = 0 to n do
            for j = 0 to m do
                if i = 0 then
                    M_k[i, j] ← 0
                else if j = 0 then
                    M_k[i, j] ← 0
                else if x_i = y_j then
                    if M_k[i − 1, j − 1] ≥ M_{k−φ(x_i)}[i − 1, j − 1] + 1 then
                        M_k[i, j] ← M_k[i − 1, j − 1]
                        S_k[i, j] ← NOTTAKE
                    else
                        M_k[i, j] ← M_{k−φ(x_i)}[i − 1, j − 1] + 1
                        S_k[i, j] ← TAKE
                    end if
                else if x_i ≠ y_j then
                    if M_k[i − 1, j − 1] ≥ M_k[i − 1, j − 1] + 1 then
                        M_k[i, j] ← M_k[i − 1, j]
                        S_k[i, j] ← UP
                    else
                        M_k[i, j] ← M_k[i, j − 1]
                        S_k[i, j] ← LEFT
                    end if
                end if
            end for
        end for
    end for
end procedure
```

Pseudocodice

```
procedure BUILD(X, Y, K)
    i, j, k ← n, m, K
    RES ←<>
    while i > 0 ∧ j > 0 ∧ k > 0 do
        if S_k[i, j] = TAKE then
            RES ← x_i • RES
            i, j ← i − 1, j − 1
        else if S_k[i, j] = NOTTAKE then
            i, j ← i − 1, j − 1
        else if S_k[i, j] = UP then
            i ← i − 1
        else if S_k[i, j] = LEFT then
            j ← j − 1
        end if
    end while
    print(RES)
end procedure
```