Contents

1 ALGORITMI

1.1 Programmazione dinamica

1.1.1 Introduzione

L' idea è quella di non dover ripetere calcoli già fatti in precedenza, cosa che succede per molto di frequente in un albero di chiamate ricorsive. Una soluzione è quella di "salvare" soluzioni a problemi intermedi in una struttura di supporto è di controllare se un particolare calcolo è già stato fatto. I problemi risolti dalla programmazione dinamica hanno la caratteristica di essere facilmente risolvibili per "piccole" istanze del problema, praticamente l'analogo del caso base in una soluzione ricorsiva. Combinando soluzioni intermedie piu semplici da risolvere si ottiene la soluzione dell' istanza completa (e più complessa) del problema originale.

1.1.2 Approccio al problema

Cominciare definendo l' istanza del problema e la soluzione del problema in maniera formale. Può essere utile, ma non è obbligatorio, definire l' equazione di ricorrenza del problema. Anche se in seguito si utilizzerà un approccio di programmazione dinamica questo passaggio ci permette di individuare il caso base e lo step per ipoteticamente raggiungere la soluzione.

Una volta individuata la struttura di supporto più appropriata per il problema definire cosa la struttura contiene ad una particolare posizione. Generalmente contiene la soluzione fino a il/i carattere/i della sequenza/sequenze corrispondente alla posizione della struttura di supporto. Molto importante è definire se la soluzione include o meno il carattere.

1.1.3 Algoritmi visti in classe

Longest increasing subsequence (LIS)

Definizione del problema Avendo una sequenza **X** di lunghezza **n** con indice **I** = {1,. . . ,n},
una sottosequenza **Z** di **X** viene ritenuta valida se gli indici degli elementi di **X** che la compongono rispettano un ordine strettamente crescente.

Esempio: **X** = < a, b, c, d > **Z** = < a, c , d > è valida
X = < a, b, c, d > **Z** = < d, a, b, c > non è valida

Vogliamo trovare la più lunga sottosequenza dove ogni elemento è maggiore del precedente (LIS).
L' algoritmo visto in classe si limita a trovare la lunghezza della LIS.

Istanza del problema:
Una sequenza **X** di elementi, la lunghezza di **X** è **n**
Soluzione del problema:

- Se voglio la LIS:
  Una permutazione di **Z** di **X** tale che:
  1. Tra tutte le permutazioni possibili di **X** (insieme delle parti) **Z** risulta la sottosequenza (deve rispettare le proprietà di una sottosequenza) quello con lunghezza (cardinalità) massima.
  2. Gli elementi della sottosequenza "ottima" **Z** devono essere ordinati in senso strettamente crescente.
- Se voglio la lunghezza della LIS:
  La lunghezza l ∈ **N** della sottosequenza **Z** definita sopra.

Pseudocodice con spiegazione Come struttura di supporto viene utilizzato un array **L** di dimensione uguale alla lunghezza **n** della sequenza **S**. Alla posizione **i** il vettore **L** contiene la lunghezza massima della LIS fino al carattere **i** della sequenza **S** con S[i] compreso.

Genericamente L[i] può avere valore:

- L[i] = 1 se i = 1
  cioè al primo elemento della sequenza la lunghezza massima possibile è 1.

- L[i] = 1 + MAX{L[i]} se $1 \leq j \leq$ i-1 AND S[j] $<$ S[i]
  cioè se l'elemento **i** di **S** risulta maggiore dei precedenti il valore in **L** alla posizione **i** è uguale al valore massimo precedentemente trovato + 1, appunto perchè è stato possibile aggiungere un nuovo elemento alla LIS fino a **i**.

Praticamente quello che l' algoritmo fa è:

1. imposto a 1 la lunghezza massima della LIS fino a **i** = 1.

2. scorro S e per ogni elemento controllo se è maggiore dell' elemento precedente che mi permette di ottenere la LIS massima, questo dato lo trovo in **L**.

3. Se non trovo nessun elemento precedente L[i] sarà 1, se lo trovo sarà max + 1.

4. Dopo aver controllato ogni elemento controllo se la LIS terminante con lui ha lunghezza maggiore della LIS massima trovata fino ad ora, se è maggiore aggiorno il valore massimo.

Soluzione vista a lezione:

```
 1  int LIS (S[1,...,n])
 2  L[1] = 1;
 3  max_tot = 1;
 4  for i = 2 to n
 5    max = 0;
 6    for j = 1 to i-1
 7    if S[j] < S[i] AND L[j] > max
 8      max = L[J]
 9    L[i] = 1 + max;
10    if L[i] > max_tot
```

```
11      max_tot = L[i]
12
13  return max_tot
```

Implementazione C++

```
 1  int lis(int sequence[], int size){
 2    // support array, at pos i will hold longest lis including i
 3    int l[size];
 4    int max_length = 1;
 5    l[0] = 1;
 6
 7    for(int i = 1; i < size; ++i)
 8      {
 9        int max = 0;
10        for(int j = 0; j <= i-1; ++j)
11   if(sequence[i] > sequence[j] && l[j] > max)
12     max = l[j];
13
14        l[i] = 1 + max;
15        if(l[i] > max_length)
16   max_length = l[i];
17      }
18
19    return max_length;
20  }
```

Longest common subsequence (LCS)

Knapsack problem

Definizione del problema Abbiamo un knapsack (zaino) con una capacità $\mathbf{L} \in \mathbf{N}$. Abbiamo un insieme di oggetti **O** di lunghezza $\mathbf{n} \in \mathbf{N}$. Ad ogni oggetto è associato un Valore $v_i \in \mathbf{V}$ la cui lunghezza è **n**. Ad ogni oggetto è associato un Peso $w_i \in \mathbf{W}$ la cui lunghezza è **n**. Definisco una funzione valore(X) $\{1,\ldots,n\} \rightarrow$ N che mi restituisce il valore totale di un insieme. La soluzione $\mathbf{S} \subseteq \mathbf{O}$ con $\sum w_i$ con $1 \leq i \leq L$ tale che valore(**S**) = MAX(valore($\wp$(O)))

L'algoritmo visto in classe ritorna il valore massimo ottenibile.

Pseudocodice e spiegazione Come struttura di supporto viene utilizzata una matrice di dimensione **n+1** x **L+1**. Viene aggiunto 1 perchè rappresenteremo anche l'oggetto e la capacità 0.

Su ogni colonna mettiamo tutti i numeri da 0 a **L**. Su ogni riga tutti i numeri da 0 a **n**. Alla posizione [i][j] troviamo il massimo valore ottenibile prendendo i oggetti avendo capacità j.

I passi dell'algoritmo:

1. Inizializzo la matrice iniziale
2. Se la capacità che sto considerando (j) è minore del peso dell'oggetto significa che non posso prendere l'oggetto i. Il valore [i][j] sarà allora il massimo valore che posso ottenere con la stessa capacità j ma con un oggetto in meno (i-1).
3. Se la capacità è invece maggiore devo considerare se ci "guadagno" a prendere il nuovo oggetto. Controllo il valore massimo ottenibile senza prendere l'oggetto e lo confronto con il valore ottenuto aggiungendo il nuovo oggetto.
4. Ritorno il valore massimo ottenibile.

```
 1   for i=0 to n
 2     OPT[i][0] = 0
 3   for j=0 to L
 4     OPT[0][j] = 0
 5
 6   for i=1 to n
 7     for j = 1 to L
 8       if j < W[i]
 9         OPT[i][j] = OPT[i-1][j]
10       else if OPT[i-1][j] > ( OPT[i-1][j - W[i]] + V[i] )
11         OPT[i][j] = OPT[i-1][j]
12       else
13         OPT[i][j] = OPT[i-1][j - W[i] + V[i]
14
15  return OPT[n][L]
```

Implementazione C++

```
1  int knapsack(int sackCapacity, char objects[], int oSize,
2  int values[], int vSize, int weigth[], int wSize){
```

```
 3
 4  assert(oSize == vSize);
 5  assert(vSize == wSize);
 6  assert(sackCapacity >= 0);
 7
 8  // add 0 value in matrix
 9  oSize = oSize + 1;
10  sackCapacity = sackCapacity + 1;
11  int matrix[oSize][sackCapacity];
12
13  // initialize matrix
14  for(int i = 0; i < oSize; ++i)
15     matrix[i][0] = 0;
16  for(int i = 0; i < sackCapacity; ++i)
17     matrix[0][i] = 0;
18
19  // access to values and weight has to be offset by -1 because we added a
20  // 0 row and column to the matrix
21
22  for(int i = 1; i < oSize; ++i)
23    for(int j = 1; j < sackCapacity; ++j)
24      if(j < weigth[i-1])
25        matrix[i][j] = matrix[i-1][j];
26      else if (matrix[i-1][j] > (matrix[i-1][j - weigth[i-1]]) + values[i-1])
27        matrix[i][j] = matrix[i-1][j];
28      else
29        matrix[i][j] = matrix[i-1][j - weigth[i-1]] + values[i-1];
30
31      std::cout << "value matrix:" << std::endl;
32      for(int i = 0; i < oSize; ++i)
33        {
34   for(int j = 0; j < sackCapacity; ++j)
35     std::cout << matrix[i][j] << " ";
36   std::cout << std::endl;
37        }
38
39      return matrix[oSize-1][sackCapacity-1];
40
41    }
```

1.2 Programmazione greedy