Dispensa ASD 2

Bonmassar Ivan

June 12, 2022

Contents

Chapter 1

Programmazione dinamica

1.1 Hateville

Ad Hateville viene organizzata una sagra. Per la raccolta fondi la casa i donera' n soldi solo se non doneranno entrambi i suoi vicini i-1 e i+1. Scrivere un algoritmo che restituisca il numero maggiore di soldi.

Algorithm 1 Hateville(int[] DP, int n)

```
int [] D = new int[0...n]
DP[0] = 0;
DP[1] = D[1];
for ( doi=2 to n)
    DP[i] = max(DP[i-2] + D[i], DP[i-1])
end for
```

Questo ritornera' una tabella DP dalla quale dovrebbe essere ricavabile la soluzione.

1.2 Knapsack

Dato un insieme di oggetti con peso e con un loro valore e data una capacita' C di uno zaino, si calcoli il valore massimo trasportabile dallo zaino.

Algorithm 2 Knapsack(int[] w, int[] p, int C, int n)

```
DP = new int[0...n][0...C];
for i = 0 to n do
    DP[i][0] = 0;
end for
for c = 0 to C do
    DP[0][c] = 0;
end for
for i=1 to n do
    for c=1 to C do
        if w[i] ≤ c then
            DP[i][c] = max(DP[i-1][c-w[i]] + p[i], DP[i-1][c]);
        else
            DP[i][c] = DP[i-1][c];
        end if
    end for
end for
```

This should return the correct matrix containing the solution in the bottom right corner.

C'e' anche una versione ricorsiva dello zaino con la memoization. La memoization e' l'approccio top-down, in pratica si controlla prima se quel problema e' gia stato risolto. DP e' inizializzata nella funzione wrapper con tutti gli elementi posti a -1.

Algorithm 3 Knapsack(int[] w, int[] p, int C, int n)

```
if c < 0 then
    return -∞
else if i == 0 or c == 0 then
    return 0
else

    if DP[i][c] < 0  then
        int notTaken = knapsackRec(w,p,i-1,c,DP);
        int taken = knapsackRec(w,p, i-1,c-w[i],DP) + p[i];
        return max(taken,notTaken);
    end if
end if
return DP[i][c];
```

La versione dello zaino senza fondo presenta invece un array DP e non una matrice. Lo si puo' trovare nelle slide di Montresor.

1.3 Sottosequenza comune massimale

Per SCM (d'ora in avanti LCS per longest common subsequence) s'intende la sottosequenza piu' lunga che due parole hanno in comune. Per esempio AAAATTGA e AAATA, LCS coincide con AAATA, in quanto la sottosequenza non deve essere di fila.

Algorithm 4 int LCS(ITEM[] T, ITEM[] U, int n, int m)

```
int[][] DP = new int[1...n][1...m];
for i = 0 to n do
    DP[i][0] = 0;
end for
for j = 0 to m do
    DP[0][m] = 0;
end for
for i=1 to n do
    for j=1 to m do
        if T[i] == U[i] then DP[i][j] = DP[i][j]+1;
        else
            DP[i][j] = max(DP[i-1][j],DP[i][j-1];
        end if
    end for
end for
     return DP[n][m];
```

In pratica quello che viene fatto e' calcolare la LCS e nel caso la lettera preas in considerazione non sia uguale si controlla togliendo una lettera dalla prima parola e poi dalla seconda. Da notare che questo non da la soluzione, in quanto da solo la lunghezza massima della LCS.

1.4 String matching approssimativo

Calcolare il minor numero k necessario per un pattern per essere trovato in una stringa. Le operazioni possibili sono, cancellazione, sostituzione e inserimento. Esempio:
BAB e' in ABABAB con k=0 modifiche.
unesempio e' contenuto in questoeunoscempio con k = 2 modifiche.

Algorithm 5 stringMatching()

```
int [][] DP = new int [0...n][0...m] ;
for j = 0 to n do DP[0][j] = 0;
end for
for i = 0 to m do DP[i][0] = i;
end for
for i = 0 to m do
    for j = 0 to n do
        DP[i][j] =min(
        DP[i-1][j-1] + iff(P[i] == T[i], 0,1),
        DP[i-1][j]+1,
        DP[i][j-1])
    end for
end for
```

1.5 Insieme di intervalli pesati

Questo problema puo' essere spiegato con una sala riunioni e un'organizzazione degli appuntamenti che massimizza i profitti.

Per risolverlo, l'algoritmo fa uso di "predecessori" ovvero dell'intervallo di tempo appena prima quello selezionato.

Algorithm 6 Set maxSet(int[] a, int[] b, int[]w, int n)

```
ordina gli intervalli per estremi di fine crescenti
int[] pred = computePred(a,b,n);
int[] DP = new int[0...n];
DP[0] = 0;
for i = 1 to n do
DP[i] = max(DP[i-1], w[i]+DP[pred[i]]);
end for
i = n;
Set s = Set();
while i > 0 do
    if DP[i-1] > w[i]+DP[pred[i]] then
        i = i-1
    elseS.insert(i) i = pred[i]
    end if
end while
```

Chapter 2

Greedy

2.1 Insieme di intervalli

Nella versione greedy del problema non sono piu' pesati. Per greedy si intende una "tattica" da utilizzare per scegliere la soluzione migliore senza dover calcolare le altre. In questo caso la scelta migliore e' quella di scegliere gli intervalli man mano con il minor tempo di fine.

Algorithm 7 Set indipendentSet(int[] a, int[] b)

```
ordina a e b in modo che b[1] < b[2]...  Set S = Set(); S.insert(1) int last = 1
for i = 2 to n do
    if a[i] ≥ b[last] then
        S.insert(i)
        last = i;
    end if
end for
```

2.2 Compressione di Huffman

La compressione di caratteri di Huffman si basa sull'idea di creare una codifica per ogni file. Questo per fare in modo che ogni file abbia una sua specifica e corretta frequenza dei caratteri. I caratteri con minor frequenza avranno prefissi piu' lunghi.

Algorithm 8 huffman(int[]c,int[]f, int n)

```
PriorityQueue Q = minPriorityQueue();
for i = 1 to n do
    Q.insert(f[i],c[i]);
end for
for i=1 to n-1 do
    z_1 = Q.deleteMin();
    z_2 = Q.deleteMin();
    z = Tree(z_1.f+z_2.f, nil);
    z.left = z_1;
    z.right = z_2;
    Q.insert(z.f,z);
end for
```

2.3 Albero di copertura minimo

Questo problema viene risolto da due algoritmi.

L'idea del primo (Kruskal) e' quella di ingrandire sottoinsiemi disgiunti di un albero, connettendoli tra loro fino ad avere l'albero complessivo.

Algorithm 9 Kruskal(Edge[]A, int n, int m)

```
Set T = Set();
MFSET M = Mfset(n);
ordina A in ordine crescente di pesi;
int count = 0;
int i = 1;
while count < n-1 and i ≤ m do
    if M.find(A[i].u ≠ M.find(A[i].v then
        M.merge(A[i].u,A[i].v)
        T.insert(A[i])
        count++
    end if
    i++
end while
```