Dispensa ASD 2

Bonmassar Ivan

June 20, 2022

Contents

1 Programmazione dinamica **3**
- 1.1 Hateville . . . 3
- 1.2 Knapsack . . . 4
- 1.3 Sottosequenza comune massimale . . . 5
- 1.4 String matching approssimativo . . . 6
- 1.5 Insieme di intervalli pesati . . . 7

2 Greedy **8**
- 2.1 Insieme di intervalli . . . 8
- 2.2 Compressione di Huffman . . . 9
- 2.3 Albero di copertura minimo . . . 10

3 Programmazione dinamica - esami **11**
- 3.1 Mosse su scacchiera . . . 11
- 3.2 profit . . . 12
- 3.3 Minimo palindromo . . . 12
- 3.4 print Bits . . . 12
- 3.5 Max Sum Even . . . 13
- 3.6 Sequenza k-contigua massimale . . . 14
- 3.7 Max Sum Increasing . . . 14
- 3.8 Small sum . . . 16
- 3.9 Count Rec . . . 16
- 3.10 Paths . . . 17

4 Backtracking - esami **18**
- 4.1 print All . . . 18
- 4.2 printFlags . . . 18
- 4.3 Count paths . . . 19
- 4.4 Palindromi . . . 20
- 4.5 Octals . . . 21
- 4.6 Binary . . . 22
- 4.7 Sequenza k-limitata . . . 23
- 4.8 PrintBits . . . 23
- 4.9 Primes . . . 24

5 Misc - esami **25**
5.1 Skipass 25
5.2 Largest cross 25
5.3 Meeting point 26
5.4 Ciclo hamiltoniano 27

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.

Complessita' $O(n)$

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. Complessita' $O(n2^k)$

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.

Complessita' $O(2^n(m+n)$

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.

Complessita' $O(n^2)$

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
    z1 = Q.deleteMin();
    z2 = Q.deleteMin();
    z = Tree(z1.f+z2.f, nil);
    z.left = z1;
    z.right = z2;
    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
```

Chapter 3

Programmazione dinamica - esami

3.1 Mosse su scacchiera

Scrivre un algoritmo che ritorna il massimo profitto basato sulla matrice $n \times n$ con un pedone che puo' muoversi di uno in diagonale oppure andando dritto

Complessita' $O(n^2)$

$$DP = \begin{cases} -\infty & c < 1 \mathbf{or} c > n \\ P[1, c] & r = 1 \mathbf{and} 1 \leq c \leq n \\ max\Big(DP[r-1][c-1], DP[r-1][c], DP[r-1][c+1]) + P[r][c] & otherwise \end{cases} \tag{3.1}$$

Algorithm 10 searchPath(int[][] P, int n

```
int DP = new int[1...n][1...n]
for int c = 1 to n do
    DP[1][c] = P[1][c]
end for
for r = 2 to n do
    for c=1 to n do
        DP[r][c] = max(DP[r][c],DP[r-1][c] + P[r][c],DP[r-1][c+1] + P[r][c],
DP[r-1][c-1] + P[r][c]
    end for
end for
return max(DP[n])
```

3.2 profit

Scrivere un algoritmo che da il profitto massimo dipendentemente se si decide di vendere o comprare un'azione in un determinato giorno V[i].

Complessita' $O(N^2)$

$$DP = \begin{cases} k \cdot V[i] & i = n \\ DP[i][j] = max(DP[i+1][j], DP[i+1][j+1] - V[i], DP[i+1][0] + k * V[i]) & otherwise \end{cases} \tag{3.2}$$

Algorithm 11 profit(int[] V, int n)

```
int[][] DP = new int[1...n][1...n]
for i = 1 to n do
    for j = 1 to n do
        DP[i][j] = -1
    end for
end for
    profitRec(V,n,1,0,DP)
```

Algorithm 12 profitRec(int[] V, int n,int i, int j, int[][]DP)

```
if i == n then return V[i] · j
end if
if DP[i][j] ¡ 0 then
    DP[i][j] = max(profitRec(V,n,i+1,j,DP),profitRec(V,n,i+1,j+1,DP)-V[i],
profitRec(V,n,i+1,0,DP)+j*V[i])
end ifreturn DP[i][j]
```

3.3 Minimo palindromo

Scrivere un algoritmo che presa in input una stringa, restituisca quante modifiche vanno fatte per renderla palindroma.

Complessita' $O(n^2)$

$$DP = \begin{cases} 0 & i \geq j \\ DP[i][j] = min(DP[i-1][j] + 1, DP[i][j-1] + 1, DP[i-1][j-1] + iffS[i] == S[j], 0, 1) \end{cases} \tag{3.3}$$

3.4 print Bits

Algoritmo che conti il numero di array di n bit senza 1 consecutivi.

Algorithm 13 minPalindroma(Item[] S, int n)

```
int[][] DP = new int[1...n][1...n]
for i = 1 to n do
    for j = 1 to n do
        DP[i][j] = -1;
    end for
end for
minPalRec(S,n,1,1)
```

Algorithm 14 minPalRec(Item[] S, int n, int i, int j)

```
if i ≥ j then return 0
end if
if DP[i][j] < 0 then
    DP[i][j] = min(DP[i-1][j] + 1, DP[i][j-1]+1, DP[i-1][j-1] + iff S[i]==S[j],0,1)
end if
```

Complessita' : $O(n^2)$

$$DP[n] = \begin{cases} 1 & n = 0 \\ 2 & n = 1 \\ DP[i] = DP[i-1] + DP[i-2] & n \geq 2 \end{cases} \tag{3.4}$$

Algorithm 15 printBits(int n)

```
int DP = new int [0...n]
DP[0] = 1
DP[1] = 2
for i=2 to n do
    DP[i] = DP[i-1] + DP[i-2]
end for
```

3.5 Max Sum Even

Dato un vettore A di n interi, calcolare la somma massima di sottovettori pari.
Esempio:

A= [9,8,-8,9,10] = 20 (9,-8,9,10)

$$DP[i] = \begin{cases} 0 & i \leq 1 \\ maxDPDP[i-2] + A[i-1] + A[i], 0 & otherwise \end{cases} \tag{3.5}$$

Complessita' : $O(n)$

Algorithm 16 maxSumEven()

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

3.6 Sequenza k-contigua massimale

Una sequenza k-contigua e' una sottosequenza che deriva dalla cancellazione di al piu' k elementi consecutivi dalla sequenza originale.

Esempio:

k=1 A[1,2,3,4,5,6] =¿ A_c[1,3,5]

Scrivere un algoritmo che trova quella massimale.

Algorithm 17 kContigua(int[]v, int n, int k)

```
int[][] DP = -1;
return kContiguaRec(V,n,k,DP)
```

Algorithm 18 kContiguaRec(int[]v, int i, int j,int[][] DP)

```
if i ≤ 0 then
    return 0
end if
if DP[i][j] < 0 then
    if j==0 then
        DP[i][j] = kContiguaRec(V,i,j-1,DP) + V[i]
    else
        DP[i][j] = max(kContiguaRec(V,i-1,j-1,DP),kContiguaRec(V,i-1, j,
DP)+V[i])
    end if
end ifreturn DP[i][j]
```

3.7 Max Sum Increasing

Scrivere un algoritmo che prenda in input un array di numeri e che calcoli il valore massimo della somma tra i numeri crescenti. Esempio:
A=[2, 102, 3, 4, 101, 5, 6] =¿ 2+3+4+101 = 110;

Complessita' $O(n^2)$

L'idea e' la seguente:

$$DP = \begin{cases} A[i] & \forall j, 1 \leq j \leq i : A[i] \leq A[j] \\ maxDP[j] : 1 \leq j \leq i-1 \land A[j] < A[i] + A[i] & otherwise \end{cases} \tag{3.6}$$

Algorithm 19 maxSumIncreasing(int[] A, int n)

```
int[] DP = new int[1..n]
for i = 1 to n do
    DP[i] = A[i]
    for j = 1 to i-1 do
        if A[j] < A[i] and DP[j] + A[i] > DP[i] then
            DP[i] = DP[j]+A[i];
        end if
    end for
end for
```

3.8 Small sum

Trovare il sottoinsieme con meno elementi per riempire uno zaino di capacita' C, dati un vettore di pesi W di dimensione n Complessita' $O(nC)$

L'idea e' la seguente:

$$DP = \begin{cases} \infty & c < 0 \\ \infty & i = 0 \\ 0 & c = 0 \\ min(DP[i-1][c], DP[i-1][c-W[i]]+1) & c > 0 \land i > 0 \end{cases} \tag{3.7}$$

Algorithm 20 ssRec(int[][] DP, int c, int[]W, int i)

```
if c < 0  then
    return ∞
else if i = 0 then
    return ∞
else if c = 0 then
    return 0
else
    if DP[i][c] > 0 then
        DP[i][c] = min(ssRec(DP, c, W, i-1), ssRec(DP, c-W[i], W,i-1)+1)
    end if
end if
```

3.9 Count Rec

CHIEDI A DAVE

3.10 Paths

Data una matrice $n \times n$ calcolare il numero di sentieri per arrivare nella casella in basso a destra potendo andare solo in basso e a destra

Complessita': $O(n^2)$

Algorithm 21 paths(int n)

```
int[][] DP = new int [0...n][0...n];
for i = 1 to n do
    DP[i][0] = DP[0][i] = 1
end for
for i = 1 to n do
    for j = 1 to n do
        DP[i][j] = DP[i-1][j] + DP[i][j-1];
    end for
end forreturn DP[n][n];
```

Chapter 4

Backtracking - esami

4.1 print All

Scrivere un algoritmo che prenda in input un testo T[] e un pattern P. Elencare tutti gli indici i di P che compongono la sottosequenza P in T.

Algorithm 22 printAll(Item[] T, int n, Item[] P, int m)

```
Stack S = Stack()
printAllRec(S,T,n,P,m)
```

Algorithm 23 printAllRec(Stack S, Item[] T, int i, Item[] P, int j)

```
if j=0 then print S
else if i ≥ j then
    Caso in cui si ignora l'ultimo carattere di P
    printAllRec(S,T, i-1, P, j)
    if T[i] == P[j] then
        S.push(i)
        printAllRec(S,T,i-1,P,j-1)
        S.pop
    end if
end if
```

4.2 printFlags

Stampare il numero di bandiere con n colori e k colori tali che siano delle abbinazini accettabili date dalla matriche k×k.

Algorithm 24 printFlags(boolean[][] A, int n, int k)

```
int[] S = new int [1...n]
return printFlagsRec(A,S, 1, k,n)
```

Algorithm 25 printFlagsRec(boolean[][] A, int[] S, int i, int k, int n)

```
if then
    print S
else
    for j = 0 to k do
        if i == 1 or A[j][S[i-1] then
            S[i] = j printFlagsRec(A,S,i+1,k,n)
        end if
    end for
end if
```

4.3 Count paths

Data una matrice contare il numero di modi per arrivare dalla casella (1,1) alla casella (n,n). Le mosse possibili sono spostarsi di A[i][j] caselle a destra sinistra, giu' o in alto, purche' si visitino solo una volta.

Complessita' $O(4^{n^2})$

Algorithm 26 countPaths(int[][] A, int n)

```
boolean visted = new boolean[1...n][1...n] = false;
countPathsRec(A,n,
```

Algorithm 27 countPathsRec(int[][] A, int n, boolean visited[][],int i, int j)

```
if i==n and j=n then return 1
else if 1 ≤ i ≤ n and 1 ≤ j ≤ n and not visited[i][j] then
    visited[i][j] = true
    int res =
    countPathsRec(A,n,visited, i+A[i][j],j) +
    countPathsRec(A,n,visited, i-A[i][j],j) +
    countPathsRec(A,n,visited, i,j+A[i][j]) +
    countPathsRec(A,n,visited, i,j-A[i][j])
    visited[i][j] = false;
    return ret
else
    return 0
end if
```

4.4 Palindromi

Scrivere tutte le permutazioni di serie di numeri palindrome tali che la loro somma sia uguale a n. Esempio:

n =6 => [1,2,2,1], [1,1,1,1,1,1], [2,2,2] ecc.

Algorithm 28 palindrome(int n)

```
int[] S = new int[1...n/2]
palRec(S,1,n)
```

Algorithm 29 palRec(int[] S, int i, int missing)

```
for k = 1 to i-1 do
    print S[k]
end for
if missing > 0 then
    print missing
end if
for k=i-1 downto 1 do
    print S[k]
end for
for j=1 to missing/2 do
    S[i] = j; palRec(S, i + 1, missing − 2 · j)
end for
```

4.5 Octals

Scrivere un algoritmo che stampa tutti gli ottali (numeri da 0 a 7) con n cifre.

Viene riportato solamente la funzione ricorsiva. Il wrapper inizializzava l'array S soluzione e chiamava la funzione con i seguenti valori: S,n,-1

Algorithm 30 octalsRec(int n)

```
if i==0 then
    print S
else
    for d==0 to 7 do
        if d ≠ prev then
            S[i] = d;
            octalsRec(S,i-1,d);
        end if
    end for
end if
```

4.6 Binary

Scrivere un algoritmo che prenda in input n, n_0 e n_1. Dovra' stampare le permutazioni di n cifre con massimo n_0 0 consecutivi e massimo n_1 1 consecutivi.

Qui viene riportato soltanto la funzione ricorsiva. La funzione wrapper crea l'array S e poi chiama la funzione ricorsiva

Complessita' $O(n \cdot 2^n)$

Algorithm 31 binaryRec(int[] S, int n, int n_0, int n_1, int i_0, int $_1$)

```
if n==0 then
    print S
end if
if i_0 > 0 then
    S[n] =0;
    binaryRec(S,n-1,n_0,n_1,i_0-1, n_1)
end if
if i_1 > 0  then
    S[n] =1;
    binaryRec(S,n-1, n_0, i_1-1,i_1)
end if
```

4.7 Sequenza k-limitata

Scrivere un algoritmo che prenda in input un vettore A[n] e un valore k e stampi tutte le sequenze k-limitate. Esempio:

A = [4,6,3,1,4] e k = 1

le sottosequenze sono :

[1][3][6][4][][4, 3][4, 3, 4], [4, 4]...

Complessita' $O(n \cdot 2^n)$

Algorithm 32 k-SequenzeRec(int[] A, Stack S, int i, int k)

```
if i==0 then
    print S
else
    if S.isEmpty() or |A[i] - S.top() leq k| then
        S.push(A[i])
        k-SequenzeRec(A, S, i-1, k);
        S.pop()
    end if
    k-SequenzeRec(A, S, i-1, k)
end if
```

4.8 PrintBits

Scrivere un algoritmo che stampi le sequenze di n bit senza 1 consecutivi.

Algorithm 33 printBitsRec(int[]S, int n, int i)

```
if i == n+1 then
    print S
end if
if  i == 1 or S[i-1] ≠ 1 then
    S[i] = 1
    printBitRec(S, n, i+1);
else
    S[i] = 0;
    printBitsRec(S,n,i+1);
end if
```

4.9 Primes

Scrivere un algoritmo che prende in input un array P con i primi p numeri primi, un intero n e un intero k. Stampare tutte le somme di primi possibili per raggiungere n con esattamente k addendi. Complessita' $O(n^k)$

Algorithm 34 primesRec(int[] S, int[] P, int p, int n, int k)

```
if k == 0 and n == 0 then
    print S
end if
if k > 0 and n > 0 and p > 0 then
    primesRec(S,P,p-1,n,k)
    S[k] = P[p]
    primesRec(S, P, p - 1, n - P[p], k - 1)
end if
```

Chapter 5

Misc - esami

5.1 Skipass

Algorithm 35 minSkipass(int[] start, int[] end, int n

```
sort(start)
sort(end)
int available = 0;
int bought = 0;
int s, e = 1;
while s ≤ n do
    if start[s] < end[e] then
        if available > 0 then
            available++
        else
            bought++
        end if
        s++
    else
        available++
        e++
    end if
end while
return bought
```

5.2 Largest cross

Trovare la dimensione della croce di 1 contenuta all'interno della matrice $n \times n$
Complessita' $O(n^3)$

Algorithm 36 cross(int[][] M, int n)

```
int maxSoFar = 0;
for i = 1 to n do
    for j = 1 to n do
        if M[i][j] = 1 then int dim = largestCross(M,n,i,j) maxSoFar =
max(maxSoFar, dim)
        end if
    end for
end for
```

Algorithm 37 largestCross(int[][] M, int n, int i, int j)

int k = 1;
while $i + k \leq n$**and**$j + k \leq n$**and**$i - k \geq 1$**and**$j - k \geq 1$**and**$M[i][j + k] + M[i + k][j] + M[i - k][j] + M[i][j - k] == 4$ **do**
    k++
end while
return k

5.3 Meeting point

Scrivere un algoritmo che calcola il punto di incontro tra due nodi tale che questo sia a distanza uguale tra i due. Grafo pesato.

Basta utilizzare shortestPath visto nella prima parte e poi ciclare sui due array di risultato per vedere se ce n'e' uno uguale: Complessita' $O(mlogn)$

Algorithm 38 meetingPoint(Graph G, Node u, Node v)

int[] d_u = shortestPath(G,u);
int[] d_v = shortestPath(G,v);
for each u in G **do**
    if $d_u == d_v$ and $d_u \neq \infty$ **then**
        return true
    end if
end for

5.4 Ciclo hamiltoniano

Scrivere un algoritmo che prende in input un grafo G e che ritorni tutti i cicli hamiltoniani. Complessita' $O(n!)$

Algorithm 39 printHamilton(Graph G)

```
int[] path = new int [1...G.n]
boolean[] visited = new int [1...G.n] = false;
visitRec(G,1,1 path, visited)
```

Algorithm 40 visitRec(Graph G, Node u, int i, int [] path, boolean[] visited)

```
path[i] = u;
if i == G.n  then
    if path[1] ∈ G.adj(u) then
        print path
    end if
else
    visited[u] = true;
    for each v ∈ G.adj(u) do
        if  thennot visited[v]
            visitRed(G,v, i+1, path, visited)
        end if
        visited[u] = false;
    end for
end if
```