ASD 2

alessandro.manfucci@studenti.unitn.it

20/07/2023

Table of contents

1 Programmazione Dinamica

1.0.1 `domino`

- **tempo** $O(n)$
- **spazio** $O(n)$

```
int domino(int n)
---
int DP''=1
int DP'=2
int DP
for int i=3 to n do
 | DP = DP' + DP''
 | DP'' = DP'
 | DP' = DP
 -
 return DP
```

1.0.2 `hateville`

```
int hateville(int[] D, int n)
---
int[] DP = new int[0...n] % Tabella delle soluzioni
DP[0]=0
DP[1]=D[1]
for (i=2 to n) do
 | DP[i] = max(DP[i-1], DP[i-2]+D[i])
 -
return DP[n]
```

1.0.3 `knapsack`

- **tempo** $O(nC)$
- **spazio** $O(nC)$

```
int knapsack(int[] w, int[] p, int n, int C)
---
int[][] DP = new int[0...n][0...n] = {0}
for int i = 1 to n do
 | for j=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]
 |  |  -
 |  -
 -
 return DP[n][C]
```

1.0.4 `knapsack 0-k`

- **tempo** $O(nC)$
- **spazio** $O(C)$

```
int knapsack(int[] w, int[] p, int n, int C)
---
int[] DP = new int[0...C] = {0}
for c=1 to C do
 | for i=1 to n do
 |  | if w[i] <= c then
 |  |  | DP[c] = max(DP[c], DP[c-w[i]]+p[i])
 |  -
 -
 return DP[C]
```

1.0.5 `lcs`

- **tempo** $O(nm)$
- **spazio** $O(nm)$

```
int lcs(int[] X, int[] Y, int n, int m)
---
int[][] DP = new int[0...n][0...m]
for i=0 to n do DP[i][0]=0
for j=0 to m do DP[0][j]=0
for i=1 to n do
 | for j=1 to m do
 |  | if X[i]==Y[j] then % MATCH
 |  |  | DP[i][j] = DP[i-1][j-1]+1
 |  | else % no MATCH
 |  |  | DP[i][j] = max(
 |  |  | DP[i-1][j], % rimuovo X[i]
 |  |  | DP[i][j-1]) % rimuovo Y[j]
 |  |  -
 |  -
 -
return DP[n][m]
```

- portare l'esempio di ricostruzione...

1.0.6 stringMatching(k-mismatch/levenshtein distance)

- **tempo** $O(nm)$
- **spazio** $O(nm)$

```
int stringMatching(Item[] P, Item[] T, int m, int n)
---
int[][] DP = new int[0...m][0...n]
for j=0 to n do DP [0][j] = 0
for i=0 to m do DP [i][0] = i
for i=1 to m do
 | for j=1 to n do
 |   | DP [i][j]=min(
 |   | DP [i-1][j-1]+iif(P[i]==T[j],0,1), % MATCH o rimuovo P[i],T[j]
 |   | DP [i-1][j]+1, % rimuovo P[i]
 |   | DP [i][j-1]+1) % rimuovo T[j]
int pos = 0 % Calcola minimo ultima riga
for j = 1 to n do
 | if DP [m][j] < DP [m][pos] then
 |   | pos = j
 |   -
 -
return DP[m][pos]
```

- il k-mismatch del pattern che finisce in T[n] è complementare alla LCS?no

1.0.7 matrix-chain mult

- **tempo** $O(n^3)$
- **spazio** $O(n^2)$

```
int recPar(int[] C, int i, int j, int[][] DP)
---
if i == j then
| return 0
else if DP[i][j] == | then
 | int minSofar = +infty
 | for int k=i to j-1 do
 |   | minSoFar = min(minSoFar, recPar(c, k+1, j, DP)+c[i-1]*c[k]*c[j])
 |   -
 | DP[i][j] = minSoFar
 | return DP[i][j]
 else
 | return DP[i][j]
 -
```

1.0.8 SPSP

1.0.8.1 Dijkstra

- **tempo** $O(n^2)$
- **spazio**

```
% (predecessor, distance)
(int[], int[]) shortestPath(Graph G, Node s)
---
PriorityQueue Q = PriorityQueue(); Q.insert(s, 0)
while not Q.isEmpty() do
```

```
 | u = Q.deleteMin()
 | b[u] = false
 | foreach v   G.adj(u) do
 |  | if d[u]+G.w(u,v)<d[v] then
 |  |  | if not b[v] then
 |  |  |  | Q.insert(v,d[u]+G.w(u,v))
 |  |  |  | b[v] = true
 |  |  | else
 |  |  |  | Q.decrease(v, d[u]+G.w(u,v))
 |  |  |  -
 |  |  | T[v]=u
 |  |  | d[v]=d[u]+G.w(u,v)
 |  |  -
 |  -
 -
return (T,d)
```

- vettore

1.0.8.2 `Johnson`

- **tempo** $O(m \log n)$
- **spazio**
- heap binario

1.0.8.3 `Fredman-Tarjan`

- **tempo** $O(m + n \log n)$
- **spazio**
- heap di Fibonacci

1.0.8.4 `Bellman-Ford-Moore`

- **tempo** $O(nm)$
- **spazio**

```
% (predecessor, distance)
(int[], int[]) shortestPath(Graph G, Node s) - Corpo principale
---
Queue Q = Queue(); Q.enqueue(s)
while not Q.isEmpty() do
 | u = Q.dequeue()
 | b[u] = false
 | foreach v   G.adj(u) do
 |  | if d[u] + G.w(u, v) < d[v] then
 |  |  | if not b[v] then
 |  |  |  | Q.enqueue(v)
 |  |  |  | b[v] = true
 |  |  |  -
 |  |  | T[v] = u
 |  |  | d[v] = d[u]+G.w[u,v]
 |  |  -
 |  -
 -
return (T,d)
```

- se siamo su un dag possiamo rilassare gli archi secondo l'ordinamento topologico con costo $O(m+n)$

1.0.9 APSP

1.0.9.1 `Floyd-Warshall`

- **tempo** $O(n^3)$
- **spazio** $O(n^3)$

```
% (distance, predecessor)
(int[][], int[][]) floydWarshall(Graph G)
---
int[][] d = new int[1...n][1...n]
int[][] T = new int[1...n][1...n]
foreach u, v   G.V() do
 | d[u][v] = +∞
 | T[u][v] = nil
 -
foreach u   G.V() do
 | foreach v   G.adj(u) do
 |   | d[u][v] = G.w(u, v)
 |   | T[u][v] = u
 |   -
 -
for k = 1 to G.n do
 | foreach u   G.V() do
 |   | foreach v   G.V() do
 |   |   | if d[u][k] + d[k][v] < d[u][v] then
 |   |   |   | d[u][v] = d[u][k] + d[k][v]
 |   |   |   | T [u][v] = T [k][v]
 |   |   |   -
 |   |   -
 |   -
 -
return (d,T)
```

1.0.10 Chiusura Transitiva

- mettere true/false nel precedente

1.1 Esercizi

1.1.1 `Massima copertura`

1.1.2 `Palindroma`

```
minpal(ITEM[] S, int n) = n-lcs(S[1...n], S[n...1], n, n)
```

1.1.3 d20

1.1.4 Costo partizione

1.1.5 Quadrato binario

1.1.6 Ottimizza somma

1.1.7 Mosse scacchiera

1.1.8 Promessi sposi

1.1.9 zaino

1.1.10 sottocres

1.1.11 pillole

1.1.12 lcs

1.1.13 mincover

1.1.14 mincoverpesato

1.1.15 LCSubstring

```
int LCSubstring(Item[] P , Item[] T , int n, int m)
---
int[][] DP = new int[0...n][0...m] = {0}
int maxSoFar = 0
for i = 1 to n do
 | for j = 1 to m do
 |   | if P [i] == T [j] then
 |   |   | DP [i][j] = DP [i - 1][j - 1] + 1
 |   |   | maxSoFar = max(maxSoFar , DP [i][j])
 |   | else
 |   |   | DP [i][j] = 0
 |   |   -
 |   -
 -
return maxSoFar
```

2 Greedy

2.0.1 MST

2.0.1.1 Kruskal

- **tempo** $O(m \log n)$
- **spazio**

```
% insieme di archi
SET kruskal(EDGE[] A, int n, int m)
---
SET T = SET()
MFSET M = MFSET()
sort(A, x,y -> x.weight <= y.weight)
int count=0
int i=0
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])
|   -
```

```
| i = i + 1
-
return T
```

2.0.1.2 Prim

- **tempo** $O(m \log n)$
- **spazio**

```
% predecessori
int[] prim(Graph G, Node r)
---
PriorityQueue Q = MinPriorityQueue()
PriorityItem[ ] pos = new PriorityItem[1 . . . G.n]
int[] p = new int[1 . . . G.n]
foreach u  G.V() - {r} do
 | pos[u] = Q.insert(u, +∞)
 -
pos[r] = Q.insert(r, 0)
p[r]=0
while not Q.isEmpty() do
 | Node u = Q.deleteMin()
 | pos[u] = nil
 | foreach v  G.adj(u) do
 |  | if pos[v] 6 = nil and w(u, v) < pos[v].priority then
 |  |  | Q.decrease(pos[v], w(u, v))
 |  |  | p[v] = u
 |  |  -
 |  -
 -
return p
```

2.1 Esercizi

2.1.1 Sciatori

2.1.2 Sfilatino

2.1.3 HighLine

2.1.4 CostoPartizione

3 Ricerca Locale

3.0.1 Flusso Massimo

3.0.1.1 Ford-FulkerSon

- **tempo** $O(f^*(V + E))$ se le capacità sono intere; altrimenti può non terminare

```
int[][] maxFlow(Graph G, Node s, Node t, int[][] c)
---
int[][] f = new int[][] % Flusso parziale
int[][] g = new int[][] % Flusso da cammino aumentante
int[][] cf = new int[][] % Rete residua
foreach x, y  G.V() do
 | f [x][y] = 0 % Inizializza un flusso nullo
 | cf [x][y] = c[x][y] % Copia c in r
```

```
 -
repeat
 | g = flusso associato ad un cammino aumentante in Gf, oppure f0
 | foreach x, y   G.V() do % se f[x][y] > 0 allora f[y][x] < 0 (o g)
 |   | f [x][y] += g[x][y] % f += g
 |   | cf [x][y] -= g[x][y] % Calcola cf
 |   -
until g = f0
return f
```

- i flussi aumentanti sono creati con in aggiunta degli archi all'indietro con flusso negativo
- ogni cammino aumentante è trovato con una visita in ampiezza; attraversando archi con capacità residua non nulla e considerando il flusso dato dall'arco con capacità residua minima in modulo;
- il senso di attraversare archi negativi: sono arrivato in u con un flusso c; ora vado in v con un flusso -c; ovvero quel flusso che andava da v in u lo sto già portando io e quel c in v lo ridireziono su un altro percorso
- con numeri reali il fatto di avere questi back edges nelle reti residue può portare a dei loop

3.0.1.2 Edmonds-Karp

- **tempo** $O(VE^2)$

3.1 Esercizi

3.1.1 Cammini indipendenti

Il problema del flusso massimo in un grafo con capacità tutte ad 1 è equivalente a trovare il massimo numero di cammini indipendenti da s a t

3.1.2 Torre di Controllo

Possiamo trasformarlo in un problema di flusso aggiungendo una sorgente che da capacità 1 a tutti gli aerei ed un pozzo che prende capacità L da tutte le torri. Ogni aereo è collegato con capacità 1 a tutte le torri a distanza massima d.

4 BackTrack

4.0.1 enumeration

```
enumeration( dati problema , Item[] S, int i,  dati parziali )
---
% Verifica se S[1 . . . i-1] contiene una soluzione ammissibile
if accept( dati problema , S, i,  dati parziali ) then
 | % "Processa" la soluzione (stampa, conta, etc.)
 | processSolution( dati problema , S, i,  dati parziali )
else
 | % Calcola l'insieme delle scelte in funzione di S[1 . . . i − 1]
 | Set C = choices( dati problema , S, i,  dati parziali )
 | % Itera sull'insieme delle scelte
 | foreach c   C do
 |   | S[i] = c
 |   | % Chiamata ricorsiva
 |   | enumeration( dati problema , S, i + 1,  dati parziali
 | -
 -
```

4.0.2 `sottoinsiemi`

- **tempo** $O(n2^n)$

```
subsetsRec(int n, int[] S, int i)
---
% S ammissibile dopo n scelte
if i > n then
 | processSolution(S, n)
else
 | % Non presente / presente
 | Set C = {0, 1}
 | foreach c   C do
 |   | S[i] = c
 |   | subsetsRec(n, S, i + 1)
 |   -
 -
```

4.0.3 `sottoinsiemi`

- **tempo** $O(n2^n)$
- **spazio** $O(\log n)$ oppure $O(1)$

```
subsets(int n)
---
for j = 0 to 2^n-1 do
 | for i = 0 to n-1 do
 |   | if (j && 2^i) != 0 then % Bitwise and
 |   |   | print i
 |   |   -
 |   -
 -
```

- sommo uno; esploro le rappresentazioni in base 2; ad ogni cifra della rappresentazione associo un elemento dell'insieme – che può essere o non essere nel sottoinsieme

4.0.4 `permutazioni`

- **tempo** $O(n^2 n!)$

```
permRec(Set A, Item[ ] S, int i)
---
% Se A è vuoto, S è ammissibile
if A.isEmpty() then
 | print S
else
 | % Copia A per il ciclo foreach
 | Set C = copy(A)
 | foreach c   C do
 |   | S[i] = c
 |   | A.remove(c)
 |   | permRec(A, S, i + 1)
 |   | A.insert(c)
 |   -
 -
```

- n! foglie di ricorsione => O(n n!) nodi di ricorsione
- ad ogni nodo pago O(n) per la copia (ed il foreach? – rimuovendo la copia avrei sempre O(n) credo...)

4.0.5 `permutazioni`

- **tempo**: $O(n!)$

```
permRec(Item[] S, int i)
---
% Caso base, un elemento
if i == 1 then
 | println S
else
 | for j = 1 to i do
 |   | swap(S, i, j)
 |   | permRec(S, i - 1)
 |   | swap(S, i, j)
 |   -
 -
```

- proviamo a mettere tutti gli elementi nell' n-esima posizione; quindi la fissiamo e proviamo a mettere i rimanenti nella n-1 esima...e poi la fissiamo

4.0.6 `k-sottoinsiemi`

- **tempo** $O(n2^n)/10^{-5...-1}$

```
kssRec(int n, int missing, int[] S, int i)
---
if missing == 0 then
 | processSolution(S, i-1)
else if i   n and 0 < missing   n-(i-1) then
 | foreach c   {0, 1} do
 |   | S[i] = c
 |   | kssRec(n, missing - c, S, i + 1)
 |   -
 -
```

- se ne ho già presi abbastanza assumi i restanti elementi nulli
- pruning ulteriore se ne ho già esclusi troppi

4.0.7 `SubSet Sum`

- **tempo** O(2^n)

```
boolean ssRec(int[] A, int n, int missing, int[] S, int i)
---
if missing == 0 then
 | processSolution(S, i - 1) % Stampa gli indici della soluzione
 | return true
else if i > n or missing < 0 then
 | return false % Terminati i valori o somma eccessiva
else
 | foreach c   {0, 1} do
 |   | S[i] = c
 |   | if ssRec(A, n, missing - A[i] · c, S, i + 1) then
 |   |   | return true
 |   |   -
 |   -
 |  return false
 -
```

4.0.8 queens

- **tempo** O(n!)

```
queens(int n, int[] S, int i)
---
if i>n then
 | print S
else
 | for j=1 to n do %Prova a piazzare la regina nella colonna j
 |   | boolean legal = true
 |   | for k=1 to i-1 do %Verifica le regine precedenti
 |   |   | if S[k]==j or k+S[k]==i+j or k-S[k]==i-j then
 |   |   |   | legal = false
 |   |   |   -
 |   |   | if legal then
 |   |   |   | S[i]=j
 |   |   |   | queens(n, S, i+1)
 |   |   |   -
 |   |   -
 |   -
 -
```

- verifico la colonna; la diagonale destra (r+c = cost); la diagonale sinistra (r-c = cost) #### sudoku

```
boolean sudoku(int[][] S, int i)
---
if i==81 then
 | processSolution(S, n)
 | return true
 -
int x = i mod 9
int y = i//9
Set C = moves(S, x, y)
int old = S[x][y]
foreach c   C do
 | S[x][y] = c
 | if sudoku(S, i+1) then
 | | return true
 | -
 -
S[x][y] = old
return false
```

```
Set moves(int[][] S, int x, int y)
---
Set C = Set()
if S[x][y] != 0 then
 | % Numero pre-inserito
 | C.insert(S[x][y])
else
 | % Verifica conflitti
 | for c = 1 to 9 do
 |   | if check(S, x, y, c) then
 |   |   | C.insert(c)
 |   |   -
```

```
 |  -
 -
return C
```

4.0.9 `tri-omini`

- un tri-omino è mancante
- si ricorre sui quattro quadrati mettendo un tri-omino al centro; il caso base è risolvibile

4.0.10 `knight-tour`

```
boolean knightTour(int[ ][ ] S, int i, int x, int y)
---
% Se i = 64, ho fatto 63 mosse e ho completato un tour (aperto)
if i == 64 then
 | processSolution(S)
 | return true
 -
Set C = moves(S, x, y)
foreach c   C do
 | S[x][y] = i
 | if knightTour(S, i + 1, x + mx[c], y + my[c]) then
 |   | return true;
 | S[x][y] = 0
 -
return false
```

- problema del ciclo hamiltoniano, non polinomiale; faccio una visita dfs con pruning quando tutti i vicini sono già stati visitati
- esistono algoritmi di costo lineare basatia sul divide et imperta

4.0.11 Inviluppo convesso

wiki

4.0.11.1 `spigolo`

- **tempo** $O(n^3)$
- uno spigolo è tale che tutti i punti non nello spigolo sono nello stesso semipiano definito dalla retta passante per quello spigolo
- provo tutti i segmenti congiungenti due punti e controllo se vale la proprietà dello spigolo

4.0.11.2 `jarvis`

- **tempo** $O(nR)$

1. si parte dal punto più a sx, p_{o_1}
2. si calcolano gli angoli degli spigoli (p_{o_1}, p_i) - angolo int rispetto alla verticale
3. si sceglie p_{o_2} t.c. l'angolo è maggiore
4. si itera fino a che l'angolo maggiore è con p_{o_1} - angolo int rispetto allo spigolo precedente

4.0.11.3 `graham`

gif

- tempo $O(n \log n)$

1. si parte dal punto più in basso, p_{o_1}

2. si ordinano i punti in base all'angolo formato con p_{o_1} - rispetto all'orizzontale[1]
3. si cancellano quei punti che hanno stesso angolo ma distanza non massima
4. si pone $p_{o_2} := p_2$
5. pongo $p_{o_3} := p_3$; controllo se per (p_{o3}, p_4) vale la regola spigolo
   - se non vale; elimino p_{o_3}, backtrack e controllo se vale la regola spigolo
   - se vale, pongo $p_{o_4} := p_4$
6. mi fermo quando sull'ultimo p_{o_j} vale la regola spigolo anche per p_n; l'ultimo spigolo è (p_{o_h}, p_{o_1})

4.0.12 `closestPair`

- **tempo** $O(n \log n)$
- approccio divide et impera

1. ordiniamo X, Y; troviamo il mediano e ricorriamo sui due sottoinsiemi di punti dati dal mediano
2. troviamo $\delta = \min(\delta_L, \delta_R)$
3. o usiamo come soluzione δ o troviamo una nuova coppia (p_L, p_R) con distanza minore
   1. la distanza dalla retta mediana L è al più δ; ci limitiamo a questo sottoinsieme X' Y'
   2. controlliamo i punti come dall'ordinamento di Y'; possiamo limitarci per ogni punto p al rettangolo successivo $\delta, 2\delta$ per questioni geometriche
   3. inoltre scomponendo il rettangolo in quadrati $\delta/2$, vi sono 16 celle ed in ognuna al più un punto (per la definizione di δ)
   4. quindi possiamo limitarci per ogni punto a controllare le distanze dei 15 successivi in Y'

4.1 Esercizi

4.1.1 `prima elementare`

4.1.2 `somma di quadrati`

5 Metodi per intrattabili

5.1 Probabilistici

5.1.1 Montecarlo

5.1.2 Las Vegas

[1] OSS: l'ordinamento definisce un inviluppo concavo