1 Laboratori

1.1 Laboratorio 7 - 04/03/20

Zaino
Avete uno zaino di capacità C ed un insieme di N oggetti. Per ognuno di questi oggetti, sapete il peso (Pi) e il valore (Vi). Dovete scegliere quali oggetti mettere nello zaino in modo da ottenere il maggior valore possibile senza superare la capacità dello zaino.

Soluzione proposta:

Algorithm 1 Zaino (ottimizzazione memoria)

```
%Inizializzazione dei vettori soluzione
%Questi vettori indicano ad un particolare elemento la sua soluzione con le relative capacita'. Le soluzioni ottime per capacita' sono ottenute a tale indice
%Usiamo C + 1 per un motivo di coerenza con l'algoritmo usato e per leggibilita' del codice
int DP_precedente[C]
int DP_successivo[C]
%Vettori Peso e Valore dell'oggetto considerato
int P[N]
int V[N]

int knapsack(int[] DP_precedente, int[] DP_successivo, int[] P, int[] V, int C, int N){
        %Inizializzo il vettore dell'elemento precedente, il successivo verra' calcolato in seguito
        for c = 0 to C do
                DP_precedente[c] = 0
        int taken, not_taken
        %Puntatore per lo swap dei due vettori (swap di indirizzi)
        int *swap
        %Costruzione della soluzione
        %Noto: salvo solo la riga dell'elemento i-1 esimo con tutte le possibili capacita' (le righe ancora precedenti non le uso piu')
        for i = 0 to N do
                for c = 0 to C do
                        %Se l'elemento ci sta nello zaino con capacita' considerata
                        if P[i] <= c then
                                taken = DP_precedente[c - P[i]] + V[i]
                                not_taken = DP_precedente[c]
                                DP_successivo[c] = max(taken, not_taken)
                        else
                                DP_successivo[c] = DP_precedente[c]
        %Effettuiamo uno scambio di vettori (scambiamo gli indirizzi che referenziano il primo elemento del vettore)
        %Ora il vettore relativo all'oggetto precedente referenzia l'elemento successivo, mentre il vettore che riguarda l'oggetto successivo referenzia quello precedente che sovrascrivera'
                swap = DP_successivo
                DP_successivo = DP_precedente
                DP_precedente = swap
        %Per via della logica dello scambio la soluzione si trova nell'oggetto precedente (ovvero oggetto i-esimo) con capacita' C considerata
        return DP_precedente[C];
```

L'algoritmo riprende la logica della versione dello zaino (non illimitato) implementato con programmazione dinamica aggiungendo un'ottimizzazione a livello di memoria: non salvo tutta la matrice DP, ma considero solo due righe di tale matrice (salvate tramite array paralleli): la riga dell'elemento i-esimo per tutte le capacità possibili e la riga dell'elemento i-1 per tutte le capacità e conterrà il profitto massimo calcolato fino ad ora. Questa ottimizzazione nasce dal fatto che la formula usata per l'implementazione considera solamente la riga DP[i-1][...] per effettuare i calcoli successivi e nessun altra riga tra le precedenti e le successive.

Sottoinsieme Crescente di Somma Massima(sottocres)
Vi viene dato in input un array di N interi $A_1 ... A_n$. Per ogni elemento potete scegliere se includerlo nell'insieme soluzione, o se ignorarlo. Se un elemento A_i fa parte dell'insieme, tutti gli elementi che lo succedono nell'array e che hanno valore minore di A_i non possono essere inclusi nell'insieme. In altre parole, gli elementi dell'insieme, quando stampati nell'ordine in cui si trovavano nell'array, devono formare una sequenza crescente. Vogliamo massimizzare la somma degli elementi dell'insieme.

Soluzione proposta: il seguente problema può essere risolto mediante Programmazione Dinamica poiché è possibile notare che, provando a cercare la massima sequenza di numeri crescenti, continuiamo a risolvere gli stessi sottoproblemi più e più volte.

Algorithm 2 Sottoinsieme Crescente di Somma Massima

```
1  int sottocres(int n, int[] A)
2          int[] DP = new int[n]
3          DP[1] = A[1]
4          for i = 2 to n do
5                  DP[i] = A[i]
6                  for j = 1 to i do
7                          if A[i] ≥ A[j] and DP[i] < DP[j] + A[i] then
8                                  DP[i] = DP[j] + A[i]
9          int massimo = −∞
10         for i = 0 to n do
11                 massimo = max(massimo, DP[i])
12         return massimo
```

Spiegazione: ad ogni iterazione del ciclo for andiamo ad inizializzare l'elemento $DP[i]$ con $A[i]$; questa operazione viene fatta per due motivi: il primo è che magari l'emento che stiamo analizzando è di per sè già la più grande sottosequenza crescente che possiamo ottenere all'interno del nostro input, il secondo è perchè, non avendo inizializzato il vettore DP, la nostra funzione farebbe degli accessi non leciti. All'interno del primo ciclo si ha un secondo ciclo (questo ci suggerisce che la complessità della nostra funzione è $\mathcal{O}(n^2)$) che serve per controllare nuovamente tutti gli elementi che abbiamo inserito fino ad ora ed inserire con certezza il massimo in quella casella. Il controllo che viene eseguito si occupa di verificare non solo che $A[i] \geq A[j]$ (notare il fatto che sono ritenuti validi anche numeri uguali perchè definito in questi termini il problema) ma anche che $DP[i] < DP[j]+A[i]$ e che quindi, come dicevamo prima, ci sia effettivamente un guadagno rispetto al numero a cui inizializzamo la cella, oppure che, nel caso di più possibilità, consideriamo effettivamente quella che porta il guadagno massimo.