Modifiche richieste

12.1 Hateville

Descrizione del problema Hateville è un villaggio particolare, composto da n case, numerate da 1 a n lungo una singola strada. Ad Hateville ognuno odia i propri vicini della porta accanto, da entrambi i lati. Quindi il vicino i odia i vicini $i-1$ e $i+1$ (se esistenti). Hateville vuole organizzare una sagra e vi ha affidato il compito di raccogliere i fondi. Ogni abitante i ha intenzione di donare una quantità $D[i]$, ma non intende partecipare ad una raccolta fondi a cui partecipano uno o entrambi i propri vicini.

Dobbiamo scrivere un algoritmo che restituisca la quantità massima di fondi che può essere raccolta.

Consegne del problema I problemi che possono esserci posti sono due.

1. scrivere un algoritmo che restituisca la quantità massima di fondi che può essere raccolta;
2. scrivere un algoritmo che restituisca il sottoinsieme di indici $S \subseteq \{1,\dots,n\}$ tale per cui la donazione totale $T = \sum_{i \in S} D[i]$ è massimale.

se risolviamo il primo siamo ad un passo dalla soluzione del secondo, mentre se risolviamo il primo abbiamo risolto necessariamente il primo.

Esempi di esecuzione Con un vettore di donazioni $D = [4,3,6]$ la raccolta massima è 10, dato dall'insieme di indici $\{1,3\}$. Mentre con un vettore di donazioni $D = [10,5,5,10]$ la raccolta massima è 20, dato dall'insieme di indici $\{1,4\}$.

La domanda che dobbiamo porci è la seguente: è possibile ridefinire una formula ricorsiva che ci permetta di calcolare il sottoinsieme di case che, se selezionate, dà origine alla maggior quantità di donazioni?

Ridefiniamo il problema caratterizzandolo matematicamente. Definiamo $HV(i)$ uno dei possibili insiemi di indici da selezionare per ottenere una donazione ottimale delle prime i case di Hateville, numerate $1,\dots,n$. $HV(n)$ è la soluzione del problema originale.

Andiamo per passi. Consideriamo il vicino i-esimo. Cosa succede se non accetto la sua donazione? Lo scarto. Proviamo ad esprimerlo in funzione dei problemi precedenti (dopotutto il problema viene risolto in maniera ricorsiva):

$$HV(i) = HV(i-1)$$

Cosa succede se accetto la sua donazione? Accetto la donazione i-esima e scarto i vicini.

$$HV(i) = \{i\} \cup HV(i-2)$$

A questo punto come faccio a decidere quale delle due opzioni scegliere? Semplicemente prendo quello che mi dà un guadagno maggiore. In simboli:

$$HV(i) = highest(HV(i-1), \{i\} \cup HV(i-2))$$

La funzione *highest* restituisce l'insieme di valore massimo.

12.1.1 Sottostruttura ottima

[EN 1]: aiuto con l'italiano Quando voglio provare che una soluzione di programmazione dinamica è corretta devo riuscire a dimostrare che le mie possibili scelte sono quelle giuste. Per dimostrarlo utilizziamo il teorema di sottostruttura ottima, che dice sostanzialmente che il modo in cui ho applicato la ricorsione è corretto. Proviamo quindi a dimostrare le scelte fatte.

Il problema dato dalle prime i case è indicato con $HV_p(i)$ e una (ce ne può essere più di una) soluzione ottima per questo problema è indicato con $HV_s(i)$. **[EN 2]: aiuto con l'italiano** Se abbiamo la soluzione

ottima, allora possiamo dimostrare che abbiamo la soluzione ottima per i rispettivi sottoproblemi (da qui sottostruttura).

Quindi se $i \in HV_s i$ allora $HV_s(i) = HV_s(i-1)$ (ossia la soluzione per il problema $HV_s i$ è la soluzione per il problema $HV_s(i-1)$), altrimenti se $i \notin HV_s i$ allora $HV_s(i) = HV_s(i-2) \cup \{i\}$.

Nota. Nella maggior parte dei casi non è necessaria una dimostrazione della soluzione, ma basta un'intuizione.

Dimostrazione

Indichiamo con $|HV_s(i)|$ l'ammontare di donazioni per la soluzione ottima $HV_s(i)$.

Nota. Sono due casi separati poiché sono mutualmente esclusivi: o prendiamo la donazione i-esima o non la prendiamo non ci sono altre possibilità.

Dimostrazione primo caso: $\boxed{i \notin HV_s(i)}$. Vogliamo dimostrare che $HV_s(i)$ è una soluzione ottima anche per $HV_p(i-1)$.

Se così non fosse esisterebbe una soluzione (migliore) $HV_s^{'}(i-1)$ per il problema $HV_p(i-1)$ tale che $|HV_s^{'}(i-1)| > |HV_s(i)|$.

Ma allora $HV_s^{'}(i-1)$ sarebbe una soluzione per $HV_p(i)$ (tale che $|HV_s^{'}(i-1)| > |HV_s(i)|$), che è assurdo. Quindi $HV_s(i)$ è una soluzione ottima anche per $HV_p(i-1)$. □

Dimostrazione secondo caso: $\boxed{i \in HV_s(i)}$. $i-1 \notin HV_s(i)$ (il precedente non appartiene alla soluzione ottima), altrimenti non sarebbe una soluzione ammissibile. Quindi, $HV_s(i) - \{i\}$ è una soluzione ottima per $HV_p(i-2)$.

Se così non fosse, esisterebbe una soluzione $HV_s^{'}(i-2)$ per il problema $HV_p(i-2)$ tale che $|HV_s^{'}(i-2)| > |HV_s(i) - \{i\}|$.

Ma allora $HV_s^{'}(i-2) \cup \{i\}$ sarebbe una soluzione per $HV_p(i)$ tale che $|HV_s^{'}(i-2) \cup \{i\}| > |HV_s(i)|$, il che è assurdo. □

12.1.2 Completare la ricorsione

Ragioniamo sui casi base. Se ho 0 case il mio guadagno è zero: $HV(0) = \varnothing$; se ho una casa prendo semplicemente la sua donazione: $HV(1) = \{1\}$.

Possiamo quindi scrivre una formula per calcolare la somma massima date i case:

$$HV(i) = \begin{cases} 0 & i = 0 \\ \{1\} & i = 1 \\ highest(HV(i-1), HV(i-2) \cup \{i\}) & i \geqslant 2 \end{cases}$$

Non vale la pena scrivere un algoritmo ricorsivo, basato su divide-etimpera, per risolvere il problema di Hateville poiché si risolverebbero molti sottoproblemi più volte.

12.1.3 Memorizzare una tabella

Facciamo qualche esempio di esecuzione. Nel primo il vettore delle donazioni è $D = [10, 5, 5, 8, 4, 7, 12]$, mentre nel secondo è $D = [10, 1, 1, 10, 1, 1, 10]$. Convincersi che gli insiemi risultanti sono corretti.

i	0	1	2	3	4	5	6	7
D		10	5	5	8	4	7	12
HV	$\varnothing$	{1}	{1}	{1,3}	{1,4}	{1,3,5}	{1,4,6}	{1,3,5,7}

i	0	1	2	3	4	5	6	7
D		10	1	1	10	1	1	10
HV	$\varnothing$	{1}	{1}	{1,3}	{1,4}	{1,4}	{1,4,6}	{1,4,7}

A questo punto dobbiamo risolvere ancora due problemi:

1. dobbiamo definire la funzione *highest* ma è banale;
2. dobbiamo memorizzare gli insiemi nella tabella, ma è costoso quindi lo non faremo, infatti noi andremo a costruire il valore della soluzione e non la soluzione e ci permetterà di ricostruire la soluzione a posteriori.

Tabella di programmazione dinamica

Indichiamo con $DP[i]$ il *valore* della massima quantità di donazioni che possiamo ottenere dalle prime i case di Hateville, e con $DP[n]$ il valore della soluzione ottima.

Possiamo quindi riempire la tabella di programmazione dinamica nel seguente modo:

$$HV(i) = \begin{cases} 0 & i = 0 \\ \{1\} & i = 1 \\ \max(DP[i-1], DP[i-2] + DP[i]) & i \geqslant 2 \end{cases}$$

Nota. Non memorizziamo più insiemi, ma valori. Infatti non effettuiamo più l'unione di insiemi ma la somma fra i valori contenuti all'interno della tabella.

Di seguito vediamo un algoritmo iterativo che risolve questo particolare problema, nel caso volessimo implementare un algoritmo ricorsivo allora dovremmo utilizzare la tecnica della memoization che vedremo più avanti.

Algoritmo 12.1.1: Algoritmo iterativo che risolve il problema Hateville

```
int hateville(int[ ] D, int n)
    // creo la tabella
    int[ ] DP ← new int[ ]0n
    // inserisco i casi base
    DP[0] ← 0
    DP[1] ← DP[1]
    // calcolo il valore i-esimo
    from i ← 2 until n do
        DP[i] ← max(DP[i − 1], DP[i − 2] + D[i])
    // restituisco il valore n-esimo
    return DP[n]
```

Stiamo calcolando la soluzione per ogni possibile sottoproblema $(n+1)$ qual è il valore massimo della soluzione. Questa soluzione ha complessità $\Theta(n)$ in quanto dobbiamo fare $\Theta(n)$ somme da fare per ottenere il risultato.

12.1.4 Soluzione con linguaggi di programmazione

Vediamo un paio di implementazioni con linguaggi di programmazione. Notiamo che gli indici iniziano da 1. Gli indici differiscono dalla notazione matematica.

Codice 12.1: Implementazione della soluzione in Java

```
public int hateville(int[] D, int n) {
    int[] DP = new int[n+1];
```

```
    DP[0] = 0;
    DP[1] = D[0];
    for (int i=2; i <= n; i++) {
        DP[i] = max(DP[i-1], DP[i-2] + D[i-1]);
    }

    return DP[n];
}
```

Codice 12.2: Implementazione della soluzione in Python

```
def hateville(D):
        DP = [ 0, D[0] ]

        for i in range(1,len(D)):
                DP.append( max(DP[-1], DP[-2] + D[i]) )

        return DP[-1]
```

12.1.5 Ricostruire la soluzione originale

Questi sono i possibili risultati che possiamo ottenere applicando l'algoritmo.

i	0	1	2	3	4	5	6	7
D		10	5	5	8	4	7	12
DP	0	10	10	15	18	19	25	31

i	0	1	2	3	4	5	6	7
D		10	1	1	10	1	1	10
DP	0	10	10	11	20	20	21	30

A questo punto abbiamo il valore della soluzione massimale, ma non abbiamo la soluzione (l'insieme degli indici)! Per riscostruire la soluzione guardiamo l'elemento i-esimo presente nella tabella nella posizione $DP[i]$, se la casa i-esima non è stata selezionata allora il valore di $DP[i]$ deriva da $DP[i-1]$, altrimenti (se la casa è stata selezionata) il suo valore deriva da $DP[i-2]+D[i]$. Utilizziamo quindi questa informazione per ricostruire la soluzione in modo ricorsivo: per ricostruire la soluzione fino ad i calcoliamo i valori fino a $i-1$ senza aggiungere nulla se la casa non è stata selezionata, altrimenti li calcoliamo fino a $i-2$ e aggiungiamo i.

Algoritmo 12.1.2: Ricostruire la soluzione generale di Hateville

```
int hateville(int[ ] D, int n)
    // Creo la tabella
    int[ ] DP ← new int[ ]0n
    // Inserisco i casi base
    DP[0] ← 0
    DP[1] ← DP[1]
    // Calcolo il valore i-esimo
    from i ← 2 until n do
        DP[i] ← max(DP[i − 1], DP[i − 2] + D[i])
    // Restituisco il valore n-esimo
    return solution(DP, D, n)
int solution(int[ ] DP, int[ ] D, int i)
    // i: indice di scorrimento
    if i == 0 then // caso base
        return ∅
    else if i == 1 then// caso base
        return {1}
    else if DP[i] == DP[i − 1] then// non seleziono la casa
        return solution(DP, D, i − 1)
    else // seleziono la casa
        Set sol = solution(DP, D, i − 2)
        sol.insert(i)
        return sol
```

Analisi della complessità La complessità computazionale di solution è $T(n) = \Theta(n)$, quella spaziale è $S(n) = \Theta(n)$.

Nota. Non è possibile migliorare la complessità spaziale di hateville poiché è necessario ricostruire la soluzione.