Progettazione di Algoritmi **Anno Accademico 2021–2022**

Note per la Lezione 12

Ugo Vaccaro

Nella lezione scorsa abbiamo appreso che la tecnica Divide-et-Impera, per quanto utile ed importante, può portare in certi casi ad algoritmi inefficienti. Ciò può accadere se, ad esempio, durante la risoluzione di un problema, la ricorsione genera sottoproblemi identici. In tal caso l'acritica applicazione della tecnica Divide-et-Impera porta ad algoritmi inefficienti, dato che si spreca tempo a risolvere più volte lo *stesso* sottoproblema.

Abbiamo altresì appreso che un semplice accorgimento ci può evitare la risoluzione ripetuta di sottoproblemi identici. Ci basta, infatti, memorizzare la soluzione ai sottoproblemi in una tabella, e recuperare tali soluzioni all'occorrenza. Questo approccio è alla base della tecnica di progetto di algoritmi che và sotto il nome di Programmazione Dinamica (PD).

Per progettare un algoritmo basato con la PD occorre quindi:

- dare una formulazione ricorsiva della soluzione al problema;
- far precedere ogni chiamata ricorsiva per la soluzione di sottoproblemi con un controllo, ciò al fine di verificare se il sottoproblema corrente è già stato risolto;
- nel caso positivo (ovvero nel caso in cui il sottoproblema è stato già risolto in precedenza) limitarsi a leggere dalla tabella la soluzione precedentemente calcolata
- nel caso negativo (ovvero nel caso in cui il sottoproblema non è stato già risolto in precedenza), risolvere il problema e memorizzare la soluzione in una opportuna entrata della tabella, per futuri usi.

In certi casi, potrebbe essere preferibile dare una versione iterativa e non ricorsiva dell'algoritmo. In questo casi, occorrerà:

- usare le condizione di base per riempire l'inizio della tabella
- riempire la tabella contenente le sottosoluzioni ai problemi seguendo un opportuno ordine, ovvero con l'accortezza che il calcolo di nuove entrate della tabella venga effettuato usando valori della tabella già riempiti in precedenza.

Come si può vedere nell'esempio del calcolo di $\binom{n}{r}$, la strutura di algoritmi basati sulla tecnica Divide-et-Impera e sulla tecnica Programmazione Dinamica è molto simile. L'unica differenza consta nel controllo preliminare (prima della chiamata ricorsiva) per verificare se il sottoproblema corrente è stato già risolto, come si vede chiaramente dai due relativi algoritmi di sotto riportati.

```
Choose(n, r)
1. IF (r = 0||n = r) {
2.      RETURN(1)
3.      } ELSE {
4.      RETURN(Choose(n − 1, r − 1)+ Choose(n − 1, r))
        }
```

```
MemChoose(n, r)
1. IF(r = 0||n = r) {
2.    RETURN(1)
3.    } ELSE {
4.       IF(T[n, r] non é definito) {
5.            T[n, r] =MemChoose(n − 1, r − 1) +MemChoose(n − 1, r)
6.         }
7.    }
8. RETURN(T[n, r])
```

◇

Nel caso in cui il problema da risolvere ha una naturale formulazione ricorsiva (come nel caso del calcolo dei numeri di Fibonacci o del calcolo dei numeri $\binom{n}{r}$), l'algoritmo di Programmazione Dinamica è agevole da derivare, in quanto esso naturalmente eredita la struttura ricorsiva del problema. In generale, però, la formulazione (della soluzione) di un problema algoritmico può non avere una immediata formulazione ricorsiva, ed è compito del progettista di algoritmi escogitarne una. In questa lezione vediamo due esempi di problemi algoritmici: per il primo problema la formulazione ricorsiva della soluzione è immediata, per il secondo occorre derivarne una opportuna.

Per descrivere il primo problema, supponiamo di avere n dadi, ciascuno con m facce. Ogni faccia di ciascun dado è etichettata con un distinto intero $i \in \{1, \ldots, m\}$. Sia inoltre dato un intero X. Il problema in questione è: determinare qual è il numero dei possibili lanci di tutti gli n dadi che danno somma pari a X. Ad esempio, supponiamo che $X = 8$, abbiamo 3 dadi, ciascuno con 6 facce etichettate con gli interi da 1 a 6. Vi sono svariati lanci dei 3 dadi che possono avere somma totale 8, ad es.

I dado	II dado	III Dado
6	1	1
5	2	1
5	1	2
4	3	1
4	2	2
4	1	3
3	3	2
3	2	3
⋮	⋮	⋮

Denotiamo con Modi(m, n, X) il numero che vogliamo calcolare. É evidente che esso soddisfa l'equazione di ricorrenza:

$$\begin{aligned} \text{Modi}(m, n, X) = &\text{Modi}(m, n-1, X-1) + \\ &\text{Modi}(m, n-1, X-2) + \\ &\cdots \\ &\text{Modi}(m, n-1, X-m). \end{aligned} \tag{1}$$

Ovviamente, varrà anche che Modi$(m, 1, j) = 1$, per tutti i valori di $j = 1, \ldots, m$. Se usassimo la ricorrenza (1) per il calcolo di Modi(m, n, X), otterremmo un algoritmo di complessità esponenziale. Infatti, molti sottoproblemi si ripetono e l'algoritmo li risolverebbe da capo (un utile esercizio è verificare queste affermazioni!). Potremmo usare la tecnica della Programmazione Dinamica per ottenere il seguente algoritmo, che fà uso di una tabella ausiliaria `M[i,j]` di dimensione $n \times X$.

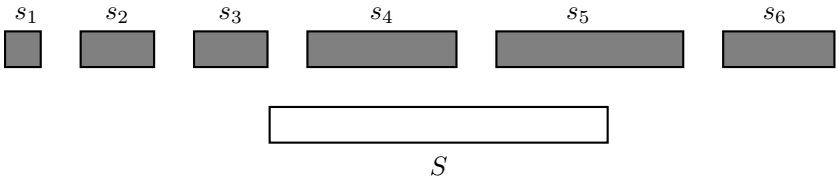
```
CalcolaModi(m,n,X)
1. IF ((n==1))&&(X<=m)) {
2.   return 1
3.     } ELSE {
4.     IF(M[n,X] non è definito){
5.       M[n,X]=0
6.       FOR(k=1;k<m+1;k=k+1){
7.       M[n,X]=M[n,X]+CalcolaModi(m,n-1,X-k)
    }
}
return M[n,X]
```

Per il calcolo di ciascuna delle $n \times X$ entrate della tabella `M` l'algoritmo impiega tempo $O(n)$, per cui la complessità dell'algoritmo è $O(n \times X \times m)$.

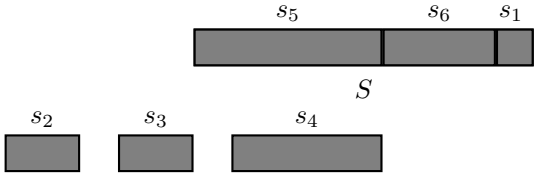
Vediamo ora un'applicazione della tecnica Programmazione Dinamica ad un problema in cui, in via preliminare, occorrerà dare una formulazione ricorsiva alla sua soluzione. Qui si suppone di avere un contenitore (zaino) di capacità S, ed n oggetti di dimensione $s_1, \ldots, s_n$, rispettivamente. Vogliamo scoprire se è possibile selezionare un sottoinsieme degli n oggetti la cui somma delle dimensioni sia *esattamente* pari a S (si immagini il caso di un CD di capacità S, e dei file di dimensione $s_1, \ldots, s_n$, e vogliamo sapere se è possibile memorizzare un sottoinsieme dei file sul CD in modo tale che la capacità del CD sia completamente ed esattamente esaurita). Formalmente, il problema è definito nel modo seguente.

Input: n oggetti di lunghezza $s_1, \ldots, s_n$, valore S
Output: `True` se e solo se esiste un sottoinsieme di tali oggetti di lunghezza totale S.

Esempio:

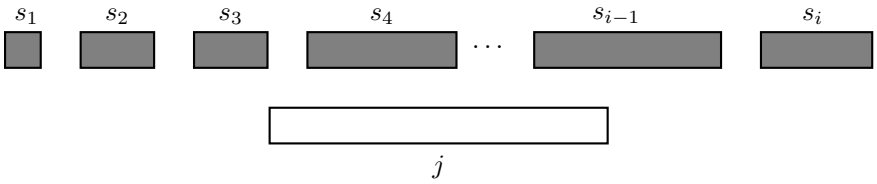


In questo caso si vede che la risposta è affermativa, in quanto la somma delle lunghezze s_1, s_5 e s_6 è esattamente pari a S.



Per motivi storici, questo problema viene chiamato Problema dello Zaino (versione semplice) in cui S rappresenta la dimensione di un ipotetico zaino e ci chiediamo se possiamo trovare un sottoinsieme di oggetti (scelti tra gli n disponibili di dimensione $s_1, \ldots, s_n$) la cui somma delle dimensioni è proprio pari a S.

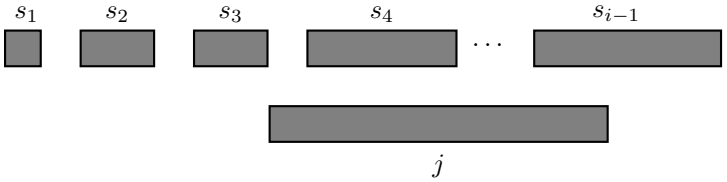
Come abbiamo detto in precedenza, occorre dare innanzitutto una formulazione ricorsiva del problema, anche se il problema in questione non è formulato (inizialmente) in forma ricorsiva. Per fare ciò, nell'esempio in questione procediamo nel seguente modo. Stabiliamo che un generico sottoproblema del problema di partenza è individuato da un sottoinsieme $s_1, \ldots, s_i$ $i \leq n$, delle lunghezze in input, e da un numero $j \leq S$ che rappresenta, concettualmente, la dimensione ipotetica di un sottozaino dello zaino di partenza. Vogliamo progettare un algoritmo `Zaino`(i, j) che risolva il sottoproblema individuato dalla coppia $(\{s_1, \ldots, s_i\}, j)$, per arbitrari $i = 1, \ldots, n$ e $j = 1, \ldots, S$. Ovvero, vogliamo un algoritmo `Zaino`(i, j) che su input $(\{s_1, \ldots, s_i\}, j)$ restituisca valore `True` se e solo se esiste un *sottoinsieme* dei primi i oggetti la cui somma delle lunghezze sia pari a j (alla fine ci servirà `Zaino`(n, S)).



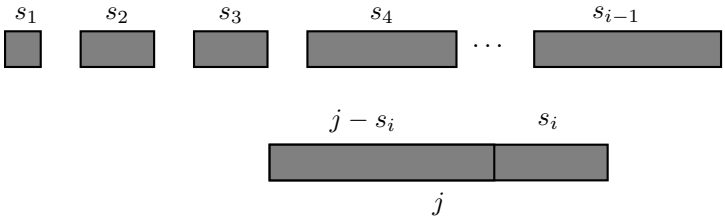
Quando `Zaino`(i, j) restituirà valore `True`? Distinguiamo i due casi:

1. o usiamo l'oggetto i-esimo (e quindi la sua lunghezza) s_i per arrivare alla lunghezza totale j,
2. oppure non usiamo l'oggetto i-esimo.

Nel caso 2., se l'oggetto i-esimo non viene usato per arrivare alla lunghezza totale j, allora `Zaino`(i, j) restituirá valore `True` se e solo se `Zaino`$(i-1, j)$ restituisce valore `True`. Detto in altri termini, la soluzione la cerchiamo usando solo le lunghezze $s_1, \ldots, s_{i-1}$. Cioè, siamo in una situazione del tipo di sotto riportata:



Se invece l'oggetto i-esimo s_i viene usato per arrivare alla lunghezza totale j, allora `Zaino`(i, j) restituirá valore `True` se e solo se `Zaino`$(i-1, j-s_i)$ restituirà valore `True`. Detto in altri termini, siamo prima riusciti a trovare un sottoinsieme dei primi $i-1$ oggetti di lunghezza totale $j-s_i$ con cui, aggiungendo poi l'i-esimo oggetto, arriviamo ad una lunghezza totale pari a $(j - s_i) + s_i = j$. Cioè, siamo in una situazione del tipo di sotto riportata:



Ma noi non sappiamo *a priori* se usare o no l'oggetto i-esimo! Come ce la caviamo, allora? Semplicemente, proviamo entrambe le alternative (ovvero proviamo a riempire uno zaino di capacità j usando l'oggetto i-esimo e proviamo anche a riempire lo zaino di capacità j senza usare l'oggetto i-esimo, restituendo `True` se riusciamo a riempirlo con almeno una delle due alternative). In altri termini, possiamo usare il seguente algoritmo.

```
Zaino(i, j)      %ritorna True se e solo se con un sottoinsieme di s_1, ..., s_i si può riempire tutto j
1. IF(i == 0) {
3.    IF(j == 0) {
4.       RETURN True
5.       } ELSE {
6.       RETURN False
    } ELSE {
7. IF(s_i ≤ j) {
8.    RETURN Zaino(i − 1, j)||Zaino(i − 1, j − s_i)
9.    } ELSE {
10. RETURN Zaino(i − 1, j)
     }
    }
```

Sia $T(n)$ = tempo di esecuzione di `Zaino`(n, S). Ovviamente vale che

$$T(n) = \begin{cases} c & \text{se } n = 1, \\ 2T(n-1) + d & \text{se } n > 1 \end{cases}$$

Come abbiamo già visto in precedenza, la soluzione è $T(n) = \Theta(2^n)$, e questo non ci piace. Sappiamo, però, già come risolvere la questione. Possiamo pensare di usare una tabella $t[\cdot, \cdot]$ dove memorizzare soluzioni a sottoproblemi e, prima di ciascuna chiamata ricorsiva, verificare se la soluzione che cerchiamo in quella chiamata ricorsiva, è stata o meno già computata in precedenza. Se è stata già computata, non effettuiamo la chiamata ricorsiva e ci limitiamo a ritornare il valore della soluzione già computata e memorizzata precedentemente nella tabella.

```
MemZaino(i, j)
1. IF(i == 0) {
3.    IF(j == 0) {
4.       RETURN True
5.       } ELSE {
6.       RETURN False
    } ELSE {
7. IF(t[i, j] non è definito) {
8.    IF(s_i ≤ j) {
```

```
9.        t[i,j] = MemZaino(i-1,j)||MemZaino(i-1,j-s_i)
10.       } ELSE {
11.       t[i,j] = MemZaino(i-1,j)
        }
      }
    }
12. RETURN t[i,j]
```

É ovvio che ogni entrata della tabella $t[\cdot,\cdot]$ viene computata una ed una sola volta, ed ogni entrata richiede tempo costante per essere computata. Per cui, essendo il numero di entrate della tabella $t[\cdot,\cdot]$ pari ad nS, otteniamo che il numero totale di operazioni elementari eseguite da `MemZaino`(n,S) è $\Theta(nS)$.

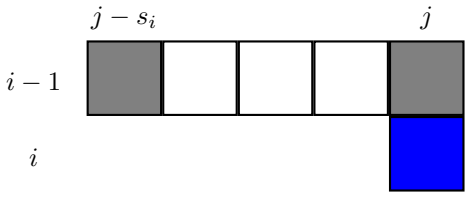
Possiamo anche dare una versione iterativa dell'algoritmo in cui memorizziamo `Zaino`(i,j) in una tabella $t[i,j]$, dove il valore di $t[i,j]$ é posto a `True` se e solo se

- o vale che $t[i-1,j]$ è `True`,
- oppure $t[i-1,j-s_i]$ ha senso (ovvero $j-s_i \geq 0$) ed è stato posto in precedenza a `True`

Ciò viene fatto col seguente codice

```
t[i,j] = t[i-1,j]
IF(j - s_i ≥ 0) {
    t[i,j] = (t[i,j]||t[i-1,j-s_i])
}
```

La figura di sotto illustra come vengono riempite le entrate della tabella $t[\cdot,\cdot]$, dove l'entrata di colore blu viene calcolata sulla base dei valori delle due entrate di colore grigio, poste nella riga precedente.



per calcolare $t[i,j]$ occorre che la riga $i-1$ sia stata giá calcolata, quindi calcoleremo la matrice $t[\cdot,\cdot]$ riga per riga, da sinistra a destra, e dall'alto in basso

L'algoritmo completo di Programmazione Dinamica, versione iterativa, sarà quindi:

```
IterZaino(s_1,...,s_n,S)
1. t[0,0] =True
2. FOR(j = 1; j < S+1; j = j+1) {
3.       t[0,j] =False
         }
```

```
4. FOR(i = 1; i < n + 1; i = i + 1) {
5.    FOR(j = 0; j < S + 1, j = j + 1) {
6.       t[i, j] = t[i − 1, j]
7.       IF(j − s_i ≥ 0) {
8.          t[i, j] = (t[i, j] ∨ t[i − 1, j − s_i])
            }
         }
      }
9. RETURN(t[n, S])
```

Analizziamo la complessità di `IterZaino`$(s_1, \ldots, s_n, S)$. Le linee 1. e 9. costano $O(1)$. Il `FOR` sulla linea 2. costa $O(S)$. Le linee 6–8 costano $O(1)$. Il `FOR` della linea 4 costa $O(n)$. Il `FOR` sulle linee 5 costa $O(S)$ per ogni sua completa esecuzione, quindi costa $O(nS)$ in totale. Pertanto, il tempo di esecuzione di `IterZaino`$(s_1, \ldots, s_n, S)$ è $O(nS)$.

Un esempio di (inizio di) esecuzione dell'algoritmo `IterZaino` sull'input $s_1 = 1, s_2 = 2, s_3 = 2, s_4 = 4, s_5 = 5, s_6 = 2, s_7 = 4, S = 15$ è riportato nella seguente tabella. Un utile esercizio consiste nel completare il calcolo di tutte le entrate della tabella, in accordo all'algoritmo `IterZaino`.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	T	T	T	F	F	F	F	F	F	F	F	F	F	F	F
3																
4																
5																
6																
7																