Progettazione di Algoritmi **Anno Accademico 2021–22**

Note per la Lezione 11

Ugo Vaccaro

Introduzione alla tecnica Programmazione Dinamica

Ricordiamo i passi fondamentali degli algoritmi basati sulla tecnica Divide-et-Impera per risolvere un dato problema algoritmico:

1. Risolvi il problema direttamente se esso è di dimensione sufficientemente piccola, altrimenti
2. Dividi il problema in sottoproblemi di dimensione inferiore
3. Risolvi (ricorsivamente) i sottoproblemi di dimensione inferiore
4. Combina le soluzioni dei sottoproblemi in una soluzione per il problema originale

Nei problemi algoritmici visti nelle lezioni precedenti, tipicamente i sottoproblemi che si ottenevano dalla applicazione del passo 2. dello schema erano diversi, pertanto, ciascuno di essi veniva individualmente risolto dalla relativa chiamata ricorsiva del passo 3. In molte situazioni, i sottoproblemi ottenuti al passo 2. potrebbero essere simili, o addirittura uguali. In tal caso, l'algoritmo basato su Divide-et-Impera risolverebbe lo stesso problema più volte, svolgendo lavoro inutile.

In situazioni siffatte (ovvero quando i sottoproblemi di un dato problema algoritmico tendono ad essere "simili", o addirittura "uguali"), è utile impiegare la tecnica della Programmazione Dinamica. Tale tecnica è essenzialmente simile alla tecnica Divide-et-Impera, con in più l' accortezza di risolvere ogni sottoproblema una volta soltanto. Gli algoritmi basati su Programmazione Dinamica risultano quindi essere più efficienti nel caso in cui i sottoproblemi di un dato problema tendono a ripetersi.

L'idea di Base della Programmazione Dinamica è quindi: *Calcola la soluzione a distinti sottoproblemi una volta soltanto, e memorizza tali soluzioni in una tabella, in modo tale che esse possano essere usate nel seguito, se occorre.*

Illustriamo questa idea su di un semplice esempio, il calcolo dei numeri di Fibonacci.

La sequenza $F_0, F_1, F_2, F_3, \ldots$ dei numeri di Fibonacci è definita dall'equazione di ricorrenza:

$$F_0 = F_1 = 1, \quad \text{e per ogni } \; n \geq 2 \;\; F_n = F_{n-1} + F_{n-2}.$$

Ad esempio, abbiamo che i primi termini della sequenza sono

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \ldots$$

I numeri di Fibonacci crescono rapidamente, si può provare infatti che $F_n \approx 2^{.694n}$.

Il primo algoritmo che viene in mente è basato su Divide-et-Impera e sulla definizione stessa di numeri di Fibonacci

```
Fib(n)
1.   IF ((n==0)||(n==1)) {
2.     RETURN 1
3.     } ELSE {
4.     RETURN Fib(n-1)+Fib(n-2)
       }
```

Sia $T(n)$ la complessità di `Fib`(n). Abbiamo che $T(0) = T(1) = 1$. In generale $\forall\ n \geq 2$ vale

$$T(n) = T(n-1) + T(n-2) + 1$$

Definendo $T'(n) = T(n) + 1$, otteniamo che i numeri $T'(n)$ hanno la proprietà che $T'(0) = T'(1) = 2$ e

$$\forall\ n \geq 2 \quad T'(n) = T'(n-1) + T'(n-2).$$

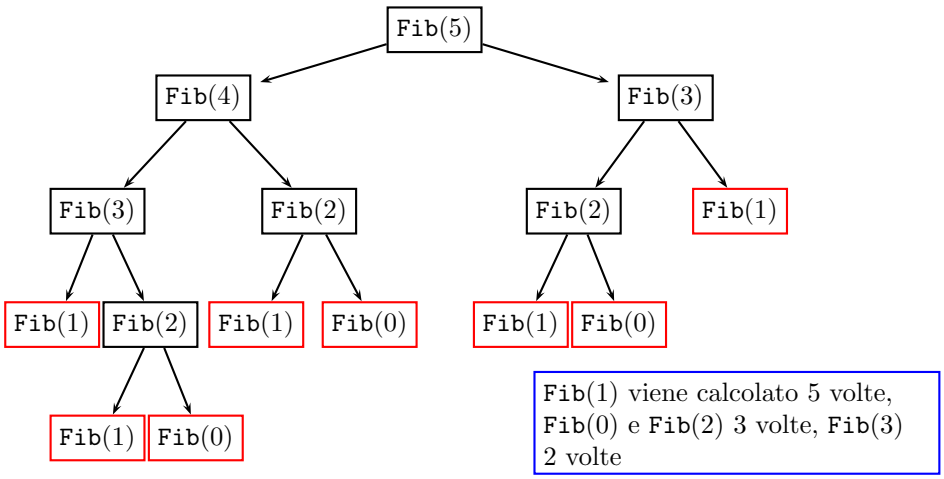
Quindi, una semplice induzione ci permette di provare che

$$T'(n) = 2F_n, \quad \text{ovvero} \quad T(n) = 2F_n - 1,$$

in altre parole, il tempo di esecuzione $T(n)$ di `Fib`(n) è pari a $T(n) = 2F_n - 1 \approx 2 \times 2^{.694n}$, troppo!

Dove sta il problema? Il problema sta nel fatto che l'algoritmo `Fib`(n) viene chiamato sullo stesso input molte volte, e ciò è chiaramente inutile.

Vediamo ad esempio lo sviluppo delle chiamate ricorsive per il calcolo di `Fib`(5).



Dall'esempio precedente vediamo che stesse quantità vengono calcolate più volte da differenti chiamate ricorsive sullo stesso input! Questo è il motivo per cui `Fib`(n) è inefficiente. Ovvero, i sottoproblemi in cui il problema originale viene decomposto non sono distinti, ma l'algoritmo ricorsivo si comporta come se lo fossero, e li risolve daccapo ogni volta.

Una volta che si è individuata la causa della inefficienza dell'algoritmo `Fib`(n), è facile individuare anche la cura. Basta memorizzare in un vettore i valori `Fib`$(i), i < n$ quando li si calcola per prima volta, cosicchè in future chiamate ricorsive a `Fib`(i) non ci sarà piú bisogno di calcolarli, ma basterà leggerli dal vettore. In altri termini, risparmiamo tempo di calcolo alle spese di un piccolo aumento di occupazione di memoria.

Il seguente algoritmo usa questa osservazione. I valori intermedi per il calcolo del numero di Fibonacci F_n vengono memorizzati in un array $a = a[0]a[1] \dots a[n]$.

```
MemFib(n)
1. IF ((n==0)||(n==1)) {
2.     RETURN 1
3.     } ELSE {
4.     IF(a[n] non è definito)
5.         a[n]= MemFib(n-1)+ MemFib(n-2)
6.      }
7. RETURN a[n]
```

L' aspetto importante dell'algoritmo `MemFib`(n) è che esso, prima di sviluppare la ricorsione per calcolare qualche quantità `MemFib`(i), per qualche $i < n$, controlla innanzitutto se essa è stata calcolata precedentemente e già posta in $a[i]$. Nel caso affermativo, la ricorsione *non* viene sviluppata e si legge semplicemente il valore da $a[i]$, con conseguente risparmio di tempo

Inoltre, se ad es. sviluppiamo le chiamate ricorsive di `MemFib`(5), ci rendiamo conto che le componento dell'array $a = [0] \dots a[5]$ vengono calcolate e assegnate nell'ordine $a[2], a[3], a[4], a[5]$. Possiamo quindi anche usare un algoritmo iterativo.

```
IterFib(n)
1. a[0]= 1
2. a[1]=1
3. FOR(i=2, i<n+1, i=i+1) {
4.   a[i]= a[i-1]+a[i-2]
5.    }
6. RETURN a[n]
```

L'algoritmo `IterFib(n)` richiede tempo $O(n)$ e memoria $O(n)$ per calcolare l'n-esimo numero di Fibonacci F_n, un miglioramento esponenziale rispetto al nostro primo algoritmo `Fib(n)`.

Un ulteriore miglioramento lo si può ottenere sulla memoria usata. Infatti, non è difficile notare che delle locazioni dell'array a ce ne occorrono solo le ultime due calcolate, e non tutte le n (i dettagli sono lasciati per esercizio).

Abbiamo detto che il primo algoritmo `Fib`(n) ha complessità $O(F_n) = O(2^{.694\dots n})$ e che gli algoritmi `MemFib`(n) e `IterFib`(n) hanno complessità $O(n)$.

In realtà abbiamo un pò imbrogliato, nel senso che ciò che abbiamo effettivamente provato è che gli algoritmi `Fib`(n), `MemFib`(n) e `IterFib`(n) eseguono $O(F_n) = O(2^{.694\dots n})$, $O(n)$, e $O(n)$ operazioni, rispettivamente. Ma quanto costa ciascuna di queste operazioni? Molto. Infatti, poichè in numeri di Fibonacci F_n crescono esponenzialmente ($\approx$ come $2^{.694\dots n}$), la loro rappresentazione in binario è un vettore lungo $O(n)$ bits, quindi ogni operazione che li coinvolge (addizione, sottrazione,...) costa tempo $O(n)$, e *non* tempo costante.

La "vera" complessità di `Fib`(n), è $O(nF_n) = O(n2^{.694\ldots n})$, e la "vera" complessità di `MemFib`(n) e `IterFib`(n) è $O(n \cdot n) = O(n^2)$. In ogni caso, un miglioramento esponenziale lo abbiamo ottenuto... E come abbiamo fatto ad ottenerlo?

1. Siamo partiti da un algoritmo ricorsivo (e non efficiente), ottenuto applicando la tecnica Divide-et-Impera al problema in questione,
2. Abbiamo scoperto che i motivi dell'inefficienza dell'algoritmo risiedevano nel fatto che l'algoritmo risolveva piú volte lo stesso sottoproblema,
3. Abbiamo quindi aggiunto una tabella all'algoritmo, indicizzata dai possibili valori input (= numero di sottoproblemi) alle chiamate ricorsive.
4. Abbiamo altresí aggiunto, prima di ogni chiamata ricorsiva dell'algoritmo su di un particolare sottoproblema, un *controllo* sulla tabella per verificare se la soluzione a quel sottoproblema era stata già calcolata in precedenza.
5. Nel caso affermativo, ritorniamo semplicemente la soluzione al sottoproblema (senza ricalcolarla). Se invece la soluzione al sottoproblema oggetto della chiamata ricorsiva non è stata precedentemente calcolata, chiamiamo ricorsivamente l'algoritmo per risolvere il (nuovo) sottoproblema.

Questa tecnica (chiamata Memoization) ha validità abbastanza generale, e ci permetterà di ottenere algoritmi efficienti per una varietà di problemi.

Applichiamo questa tecnica su di un altro semplcie esempio, il calcolo di combinazioni.

Denotiamo con $\binom{n}{r}$ il numero di modi con cui possiamo scegliere r oggetti da un insieme di n elementi. Come possiamo calcolare $\binom{n}{r}$? Innanzitutto, osserviamo che se $r = 0$, allora esiste un unico modo di scegliere "zero" oggetti da un insieme di n elementi, per cui varrà $\binom{n}{0} = 1$. Analogamente, se $r = n$ esiste un unico modo di scegliere n (ovvero tutti gli) oggetti da un insieme di n elementi, per cui varrà $\binom{n}{n} = 1$. In generale, per $0 < r < n$, per scegliere r oggetti da un insieme di n elementi, possiamo

- o scegliere di *prendere il primo oggetto dell'insieme* (in questo caso ci resterà il problema di scegliere i restanti $r-1$ oggetti da un insieme di $n-1$ elementi, e questo si potrà fare in $\binom{n-1}{r-1}$ modi),
- oppure scegliere di *non prendere il primo oggetto dell'insieme* (in questo secondo caso ci resterà il problema di scegliere tutti gli r oggetti da un insieme di $n-1$ elementi, e questo si potrà fare in $\binom{n-1}{r}$ modi).

Ciò equivale a dire che

$$\binom{n}{r} = \begin{cases} 1 & \text{se } r = 0 \text{ oppure } r = n, \\ \binom{n-1}{r-1} + \binom{n-1}{r} & \text{altrimenti .} \end{cases}$$

Possiamo quindi usare la tecnica Divide-et-Impera per calcolare $\binom{n}{r}$, ed ottenere il seguente algoritmo.

```
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))
        }
```

Ai fini dell'analisi dell'algoritmo `Choose`(n, r), denotiamo con $T(n, r)$ il numero di operazioni effettuate dall'algoritmo `Choose`(n, r), e sia $T(n) = \max_r T(n, r)$.

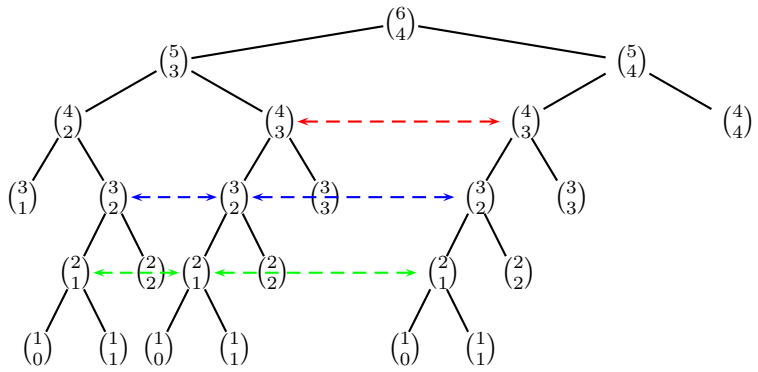
Abbiamo

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

per qualche costante c e d. Risolviamo l' equazione di ricorrenza.

$$\begin{aligned} T(n) &= 2T(n-1) + d \\ &= 2(2T(n-2) + d) + d \\ &= 4T(n-2) + 2d + d \\ &\dots \\ &= 2^i T(n-i) + d\sum_{j=0}^{i-1} 2^j \\ &= 2^{n-1} T(1) + d\sum_{j=0}^{n-2} 2^j \\ &= c2^{n-1} + d(2^{n-1} - 1) = (c+d)2^{n-1} - d. \end{aligned}$$

Quindi $T(n) = \Theta(2^n)$, e non stà bene. Perchè l'algoritmo è così inefficiente? Perchè l'algoritmo risolve gli stessi sottoproblemi piú volte. Diamo infatti un'occhiata all'albero delle chiamate ricorsive di `Choose`$(6, 4)$).



Si vede immediatamente che *stessi* sottoproblemi vengono risolti più volte. Ad esempio, `Choose`$(4, 3)$ viene chiamato due volte per risolvere lo stesso sottoproblema, `Choose`$(3, 2)$ viene chiamato tre volte per risolvere lo stesso sottoproblema, `Choose`$(2, 1)$ viene chiamato tre volte per risolvere lo stesso sottoproblema.

Situazione quindi già vista... Abbiamo cioè un algoritmo inefficiente `Choose`(n, r) per il calcolo dei numeri $\binom{n}{r}$, la cui inefficienza risiede nel fatto che esso ricalcola (mediante chiamate ricorsive ad esso stesso) quanto già calcolato in precedenza. Sappiamo già come ovviare a tale situazione: aggiungere all'algoritmo una tabella $T[i, j]$ che contenga i valori $\binom{i}{j}$ man mano che li calcoliamo. Faremo precedere ad ogni chiamata ricorsiva dell'algoritmo `Choose`(i, j) un controllo per verificare se numero $\binom{i}{j}$ è stato precedentemente computato; nel caso affermativo ci limiteremo a leggerlo dalla tabella in $T[i, j]$, altrimenti sviluppiamo la ricorsione per calcolarlo (ciò accadrà *una sola volta*), e lo memorizzeremo in $T[i, j]$.

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

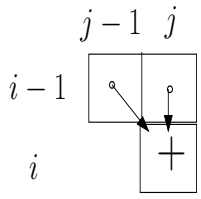
Espandendo in un albero tutte le chiamate ricorsive sull'esempio $n = 6$ e $r = 4$ (esercizio molto consigliato!) ci si rende conto che l'algoritmo `MemChoose`$(6, 4)$ riempie le entrate della tabella $T[1...6, 0...4]$ a partire da quelle che appaiono sulle foglie dell'albero, e poi via via dal basso in alto, fino all'entrata $T[6, 4]$ che conterrà l'output dell'algoritmo. Possiamo quindi anche pensare ad un algoritmo iterativo per il calcolo di $\binom{n}{r}$.

In altri termini, vogliamo calcolare tutti valori $\binom{i}{j}$, per $i = 0, \dots n$ e $j = 0, \dots r$, e metterli nella tabella $T[i, j]$ (noi siamo interessati al valore $T[n, r]$). Ricordiamo che la formula per $\binom{i}{j}$ è $\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j}$. con le condizioni iniziali $\binom{i}{i} = 1 = \binom{i}{0}$.

L'algoritmo iterativo è:

```
IterChoose(n,r)
1. FOR(i=0,i<n+1, i=i+1) {
2.   T[i,0]= 1
     }
3. FOR(i=0,i<r+1,i=i+1) {
4.   T[i,i]=1
     }
5. FOR(j=1,j<r+1,j=j+1) {
6.   FOR(i=j+1,i<n+1,i=i+1) {
7.     T[i,j]=T[i-1,j-1]+T[i-1,j]
      }
     }
RETURN T[n,r]
```

Al momento dell' assegnazione `T[i,j] = T[i-1,j-1] + T[i-1,j]` occorre che `T[i-1,j-1]` e + `T[i-1,j]` siano stati già calcolati (e infatti l'algoritmo correttamente procede in questo modo).



Vediamo un esempio di esecuzione, ovvero la tabella riempita dalla parte iniziale di una esecuzione di `IterChoose`(n, r).

1							
1	1						
1	2	1					
1	3	3	1				
1	4	6	4	1			
1	5	10	10	5	1		
1	6	15	20	15	6	1	
1	7	21	35	35	21	7	1

Analizziamo ora l'algoritmo `IterChoose(n,r)`. La tabella `T[0...n,0...r]` ha $n \cdot r \leq n^2$ entrate, ciascuna viene calcolata con una addizione dalle precedenti due (ciò richiede tempo $O(1)$). Quindi l'algoritmo ha complessità $O(n^2)$ (ricordiamo che il primo algoritmo ricorsivo aveva complessità $\Theta(2^n)$). L'algoritmo usa $O(n^2)$ spazio per memorizzare la tabella, ma poichè `T[i,j]=T[i-1,j-1]+T[i-1,j]`, ad ogni iterazione dell'algoritmo basta solo memorizzare la colonna $j-1$ e j. Quindi l'algoritmo necessita solo di $O(n)$ locazioni di memoria.

Morale della lezione.

- Quando, nella risoluzione di un problema, Divide et Impera genera molti sottoproblemi identici, la ricorsione porta ad algoritmi inefficienti.
- Conviene allora memorizzare la soluzione ai sottoproblemi in una tabella, e leggerli all'occorrenza.
- Questa è l'essenza della Programmazione Dinamica