Corso di laurea in Informatica
Progettazione d'algoritmi

Tecnica Programmazione Dinamica 1

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Dal *divide et impera* **alla** *programmazione dinamica*

Sappiamo che gli algoritmi basati sulla tecnica del Divide et Impera seguono i 3 passi di questo schema:

1. Dividi il problema in sottoproblemi di taglia inferiore.
2. Risolvi (ricorsivamente) i sottoproblemi di taglia inferiore.
3. Combina le soluzioni dei sottoproblemi in una soluzione del problema originale.

Negli esempi finora visti i sottoproblemi che si ottenevano dalla applicazione del passo 1 erano tutti **diversi**, pertanto ciascuno di essi veniva individualmente risolto dalla relativa chiamata ricorsiva del passo 2. In molte situazioni i sottoproblemi ottenuti al passo 1 possono risultare **uguali**. In tal caso, l'algoritmo basato sulla tecnica del Divide et Impera risolve lo stesso problema più volte svolgendo lavoro inutile.

ESEMPIO:

la sequenza $f_0, f_1, f_2 \ldots$ dei numeri di Fibonacci è definita dall'equazione di ricorrenza :

$$f_i = f_{i-1} + f_{i-2} \quad \text{con } f_0 = f_1 = 1$$

Ad esempio i primi termini della sequenza sono

$$1, 1, 2, 3, 5, 8, 13, \ldots$$

Problema: dato un intero n progettare un algoritmo che calcola f_n

- il primo algoritmo che viene in mente per risolvere il problema è basato sul divide et impera e sfrutta la definizione stessa di numero di Fibonacci.

```
def fib(n):
    if n<=1 : return 1
    a=fib(n-1)
    b=fib(n-2)
    return a+b
```

la relazione di ricorrenza per il tempo di calcolo $T(n)$ dell'algoritmo è:

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

Ovviamente:

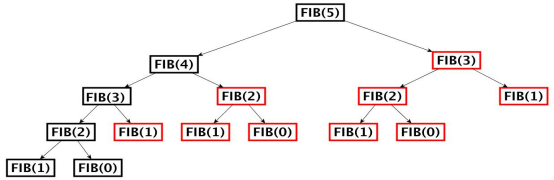
$$T(n) \geq 2T(n-2) + O(1)$$

e questa ricorrenza può essere facilmente risolta (ad esempio col metodo iterativo) e otteniamo:

$$T(n) = \Omega\left(2^{\frac{n}{2}}\right)$$

$$T(n) = \Omega\left(2^{\frac{n}{2}}\right)$$

Il motivo di una tale inefficienza stà nel fatto che il programma FIB viene chiamato sullo stesso input molte volte e ciò è chiaramente ridondante.



Esempio dello sviluppo delle chiamate ricorsive per $fib(5)$ dove sono evidenziate in rosso le chiamate "ridondanti"

- vediamo qui che, nel corso della scomposizione in sottoproblemi, stesse quantità vengono calcolate più volte da differenti chiamate ricorsive. I sottoproblemi in cui viene scomposto il problema non sono disgiunti (questo fenomeno prende il nome di overlapping di sottoproblemi) mentre l'algoritmo si comporta come se lo fossero e li risolve nuovamente.

- Individuata la causa dell'inefficienza dell'algoritmo fib è facile individuare la cura. Basta memorizzare in una lista i valori $fib(i)$ quando li si calcola la prima volta cosicché nelle future chiamate ricorsive a $fib(i)$ non ci sarà più bisogno di ricalcolarli ma potranno essere ricavati dalla lista (questa tecnica prende il nome di memoizzazione)

- si risparmia così tempo di calcolo al costo di un piccolo incremento di occupazione di memoria

```
def fib1(n):
    F=[-1]*(n+1)
    return memfib(n,F)


def memFib(n,F):
    if n<=1: return 1
    if F[n]==-1:
        a=memFib(n-1,F)
        b=memFib(n-2,F)
        F[n]=a+b
    return F[n]


>>> fib1(50)
20365011074
```



```
def fib1(n):
    F=[-1]*(n+1)
    return memfib(n,F)


def memFib(n,F):
    if n<=1: return 1
    if F[n]==-1:
        a=memFib(n-1,F)
        b=memFib(n-2,F)
        F[n]=a+b
    return F[n]


>>> fib1(50)
20365011074
```


- la novità dell'algoritmo $fib1$ è che esso, prima di attivare la ricorsione per il calcolo di qualche f_i, con $i < n$, controlla se quel valore è stato calcolato precedentemente e posto in $F[i]$. In caso affermativo la ricorsione non viene effettuata ma viene restituito direttamente il valore $F[i]$

- in questo modo l'algoritmo effettuerà esattamente n chiamate ricorsive (una sola chiamata per il calcolo di ogni f_i con $i < n$)

- Tenendo conto che ogni chiamata ricorsiva costa $O(1)$ il tempo di calcolo di $fib1$ è $\Theta(n)$, un miglioramento esponenziale rispetto alla versione da cui eravamo partiti.

Al punto a cui siamo giunti possiamo anche eliminare la ricorsione ed ottenere la versione iterativa dell'algoritmo

```
def fib2(n):
    F=[-1]*(n+1)
    F[0]=F[1]=1
    for i in range(2,n+1):
        F[i]=F[i-2]+F[i-1]
    return F[n]
```

- La complessità asintotica rimane $\Theta(n)$ ma abbiamo un risparmio di tempo e spazio (quello necessario alla gestione della ricorsione).

- La complessità di spazio è $O(n)$ a causa della lista F da memorizzare. Non è però difficile ridurre questa occupazione a $O(1)$ tenendo conto che dell'intera tabella basta conservare in memoria volta per volta solo gli ultimi due valori calcolati:

```
def fib3(n):
    if n<= 1: return 1
    a=b=1
    for i in range(2,n+1):
        a,b=b,a+b
    return b
```


Riassumendo:

- Siamo partiti da un algoritmo ricorsivo e non efficiente ottenuto applicando la tecnica del divide et impera al problema in esame.
- ci siamo accorti che il motivo dell'inefficienza era la presenza di overlapping di sottoproblemi.
- abbiamo risolto il problema del ricalcolo di soluzioni allo stesso sottoproblema mediante la tecnica della memoizzazione e quindi ricorrendo a "tabelle" per conservare i risultati a sottoproblemi gia calcolati.
- abbiamo sviluppato una versione dell'algoritmo iterativa che ha permesso di sbarazzarsi della ricorsione.
- abbiamo ottimizzare lo spazio di memoria mantenendo memorizzata nel corso dell'algoritmo solo la parte della tabella che sarebbe servita nel seguito.

Nota che:

- Nella versione ricorsiva dell'algoritmo si parte dal problema scomponendolo via via in sottoproblemi di dimensione sempre più piccola fino ad arrivare a problemi facilmente risolvibili. Si parla in questo caso di approccio top-down al problema.
- Nella versione iterativa dell'algoritmo si comincia col risolvere i sottoproblemi di dimensione sufficientemente piccola per poi passare a quelli di dimensione via via crescente fino ad arrivare alla soluzione del problema originario. Si parla in questo caso di approccio bottom-up al problema.

Esempio:

PROBLEMA: Abbiamo n file di varie dimensioni ed un disco di capacità C bisogna trovare il sottoinsieme di file che può essere memorizzato sul disco e massimizza lo spazio occupato.
Progettare un algoritmo che, dati C e $lista$ dove $lista[i]$ è la dimensione del file i, risolve il problema.

Ad esempio, per $C = 100$ e $lista = [82,\ 15,\ 40,\ 95,\ 31,\ 50,\ 40,\ 28]$ la risposta deve essere $\{2, 4, 7\}$ che occupa spazio $40 + 31 + 28 = 99$.

Non è difficile rendersi conto che algoritmi $greedy$ del tipo "finché c'è spazio memorizza sul disco il file di massima dimensione" non trovano sempre la soluzione ottima.

Per questo problema non si conoscono algoritmi efficienti. Migliori algoritmi si basano su un qualche tipo di ricerca esaustiva della soluzione

Per fortuna è altrettanto facile trovare algoritmi d'approssimazione efficienti con rapporti d'approssimazione costante.

Per semplicità di esposizione ci limiteremo ora a calcolare il valore della soluzione ottima, cioè il massimo spazio del disco che può essere occupato grazie agli n file.

Un algoritmo basato sul *divide et impera* può partire dalla seguenti osservazioni:

Sia $(lista, C)$ l'istanza che vogliamo risolvere:

- se la lista dei file risulta vuota o la capacità C vale 0 la soluzione ottima vale 0
- In caso contrario l'ultimo file della lista può appartenere o meno alla soluzione ottima.
  - se l'ultimo file non appartiene alla soluzione i rimanenti $n-1$ file devono essere una soluzione ottima per $(lista[:-1],\ \ C)$
  - se l'ultimo file appartiene alla soluzione i rimanenti $n-1$ file devono essere una soluzione ottima per $(lista[:-1],\ \ C - lista[n-1])$
- possiamo quindi ricondurci al calcolo della soluzione ottima di due sottoproblemi di dimensione inferiore in quanto manca all'appello l'ultimo file e una volta risolti questi due problemi ed ottenuti i due valori $v1$ e $v2$ come soluzione, la soluzione al problema di partenza sarà data da

$$\max(v1,\ v2 + lista[-1])$$

Implementazione 1:

```
def file(lista,C):
    if  len(lista)==0 or C==0:
        return 0
    lascio=file(lista[:-1],C)
    if lista[-1]>C:
        return lascio
    prendo=lista[-1]+file(lista[:-1],C-lista[-1])
    return max(lascio, prendo)


>>>lista,C=[80,15,40,95,30,50,40,30],100
100
```

Implementazione 2:

```
def file(i,lista,C):
    if  i==0 or C==0:
        return 0
    lascio=file(i-1,lista,C)
    if lista[i-1]>C:
        return lascio
    prendo=lista[i-1]+file(i-1,lista,C-lista[i-1])
    return max(lascio, prendo)


 >>> file(len(lista),lista,C)
100
```

- Con la prima implementazione per la complessità si avrà:

$$T(n) \ = \ T(n-1,\ C) \ + \ T(n-1,\ C - lista[n-1]) \ + \ \Theta(n)$$

il costo addittivo $O(n)$ è dovuto al passaggio della copia della lista.

- Con la seconda implementazione per la complessità si avrà:

$$T(n) \ = \ T(n-1, C) \ + \ T(n-1,\ C - lista[n-1]) \ + \ \Theta(1)$$

Considera l'istanza in cui $C = n-1$ e $lista[i] = 1,\ 0 \le i < n$.
Per quest'istanza per l'equazione di ricorrenza della seconda implementazione si ha:

$$\begin{aligned} T(n) \ &= \ T(n-1,\ C) \ + \ T(n-1,\ \ C - lista[n-1]) \ + \ \Theta(1) \\ &\ge \ 2 \cdot T(n-1,\ n-1) \ + \ \Theta(1) \\ &= 2 \cdot T(n-1) + \Theta(1) \end{aligned}$$

che risolta dà $T(n) = \Omega\left(2^n\right)$

Il motivo di questi tempi intrattabili è dovuto all'*overlapping* dei sottoproblemi generati dal partizionamento prodotto dal *divide et impera*

Di seguito parte dell'albero di ricorsione generato durante l'esecuzione dell'istanza dove $C = 10$ e $lista = [1,\ 5,\ 3,\ 4,\ 2,\ 2]$

- ciascun nodo è una coppia con al primo posto il numero di file dell'istanza ed al secondo la capacità del disco.
- in rosso sono marcate le chiamate ricorsive duplicate e che l'algoritmo ricalcola.
- più l'istanza è grande e più chiamate ricorsive duplicate saranno presenti.

- per evitare di fare questo inutile lavoro ricorriamo alla memoizzazione utiliziamo una tabella in cui salviamo i risultati via via calcolati per poterli riutilizzare se necessario senza doverli ricalcolare.

- La tabella T avrà dimensione $n \times (C+1)$ e nella cella $T[i][j]$ memorizzeremo il valore ottenuto dalla soluzione del sottoproblema in cui si hanno i soli primi i file e la dimensione del disco è j

Implementazione *divide et impera memoizzato*

```
def file(lista, C):
    T = [ [-1]*(C+1) for i in range(len(lista)+1) ]
    return mem_file(len(lista), lista, C, T)

def mem_file(i, lista, c, T):
    if T[i][c] == -1:
        if  i == 0 or C == 0:
            T[i][c] = 0
        else:
            lascio=mem_file(i-1, lista, c, T)
            T[i][c]=lascio
            if lista[i-1] <= c:
                prendo = lista[i-1] + mem_file(i-1, lista, c-lista[i-1], T)
                T[i][c] = max(lascio, prendo)
    return T[i][c]


>>> lista = [80,15,40,95,30,50,40,30]
>>> file(lista,20)
15
>>> file(lista,85)
85
>>> file(lista,345)
340
```

- Il tempo di calcolo della versione *memoizzata* è limitato dalla dimensione della tabella. Infatti, la funzione *mem_file()* esegue $O(1)$ operazioni in ogni chiamata e il numero totale di chiamate non può superare la dimensione della tabella.

- La complessità dell'implementazione memoizzata è dunque $O(nC)$ (in realtà è $\Theta(nC)$ perché la tabella va inizializzata con i valori -1).

La complessità della versione memoizzata è dunque $\Theta(nC)$, quella dell'algoritmo esaustivo è $O(2^n)$.

C'è stato un risparmio??? **Dipende!**

- Se C è molto grande, ad esempio quando supera 2^n, la versione memoizzata fa peggio. Però se C non è così grande, la versione memoizzata è più efficiente e a volte enormemente più efficiente.

- Consideriamo un caso abbastanza realistico: $n = 100.000$ file e capacità $C = 1.000.000$ (assumiamo che le dimensioni dei file e la capacità siano espresse in multipli di MB, così C equivale a 1 TB) con dimensione massima dei file 1000 (cioè 1 GB) di modo che qualsiasi sottoinsieme di 1000 file può essere memorizzato sul disco.

  - **L'algoritmo non memoizzato** eseguirà tutte le chiamate ricorsive (cioè sarà sempre possibile scegliere il k-esimo file) almeno relativamente ai primi 1000 file (da $n = 100.000$ a 99.000). Quindi la complessità dell'algoritmo non memoizzato sarà almeno dell'ordine di 2^{1000}.
  - **L'algoritmo memoizzato** ha una complessità dell'ordine di $nC = 100.000 \times 1.000.000 = 100.000.000.000$.

Quindi un computer con sufficiente memoria (dell'ordine delle centinaia di GB) potrò eseguire l'algoritmo memoizzato in meno di un'ora. Mentre con l'algoritmo non memoizzato, dovendo eseguire almeno 2^{1000} operazioni, anche usando tutti i computer del pianeta, non riusciremo a vedere il risultato del calcolo neanche aspettando fino alla fine all'età stimata dell'universo.

Come mostra l'esempio appena visto la profondità dell'albero delle chiamate ricorsive (in entrambe le versioni dell'algoritmo) può essere molto grande e portare a *stack overflow*. Nella versione memoizzata ci si può concentrare direttamente sul calcolo della tabella T eliminando così la ricorsione:

- Definiamo una tabella T di dimensioni $(n+1) \times (C+1)$ dove:
  $T[i, c]$ = spazio massimo con i primi i file su un disco di capacità c
  con $0 \leq i \leq n$ e $0 \leq c \leq C$
- la soluzione al problema originario la troviamo in $T[n][C]$

- Possiamo calcolare la tabella per righe grazie alla seguente formula ricorsiva:

$$T[i,\ c] = \begin{cases} 0 & \text{se } i = 0 \\ T[i-1,\ c] & \text{se } c < lista[i-1] \\ \max(T[i-1,\ c],\ lista[i-1] + T[i-1,\ c - lista[i-1]]) & \text{altrimenti} \end{cases}$$

- Definiamo una tabella T di dimensioni $(n+1) \times (C+1)$ dove:
  $T[i, c]$ = spazio massimo con i primi i file su un disco di capacità c
  con $0 \leq i \leq n$ e $0 \leq c \leq C$
- la soluzione al problema originario la troviamo in $T[n][C]$

- Possiamo calcolare la tabella per righe grazie alla seguente formula ricorsiva:

$$T[i,\ c] = \begin{cases} 0 & \text{se } i = 0 \\ T[i-1,\ c] & \text{se } c < lista[i-1] \\ \max(T[i-1,\ c],\ lista[i-1] + T[i-1,\ c - lista[i-1]]) & \text{altrimenti} \end{cases}$$

```
def iter_file(lista, C):
    n = len(lista)
    T = [ [0]*(C+1) for i in range(n+1) ]
    for i in range(1,n+1):
        for c in range(0, C+1):
            if c <  lista[i-1]:
                T[i][c] = T[i-1][c]
            else:
                T[i][c] = max( T[i-1][c], lista[i-1] + T[i-1][c- lista[i-1]] )
    return T[n][C]
```

Complessità $O(nC)$

- in termini dell'albero delle chiamate ricorsive, l'algoritmo iterativo, calcola i risultati a partire dalle foglie, cioè gli elementi $T[0, c]$, poi i risultati del livello superiore e così via risalendo l'albero di livello in livello fino alla radice $T[n, C]$ (approccio *bottom-up* al problema).

RIASSUMENDO:

- in termini generali la programmazione dinamica, al pari del divide et impera può essere vista come un metodo che risolve un problema risolvendo problemi di dimensione più piccola e grazie all'utilizzo di tabelle permette di non ricalcolare più volte lo stesso sottoproblema (*memoizzazione*).
- I sottoproblemi possono essere risolti a partire dal piú grande al piú piccolo (*approccio top-down*) tipico delle versioni ricorsive o dal piú piccolo al più grande (*approccio bottom-up*) tipico delle versioni iterative.

Algoritmi pseudopolinomiali

Viene detto *pseudopolinomiale* un algoritmo che risolve un problema in tempo polinomiale quando i numeri presenti nell'input sono codificati in unario.
L'algoritmo che abbiamo ottenuto di complessità $O(nC)$ è un esempio di algoritmo pseudopolinomiale.
Ricorda che la capacità C del disco data in input è codificata con $\log C$ bit e quindi per ottenere un algoritmo polinomiale nella complessità può comparire $\log^c C$ per una qualche costante c la presenza di C nella complessità mostra che questa può essere anche esponenziale nella reale dimensione dell'input.

Programmazione dinamica: dal valore della soluzione alla soluzione.

- La tabella che riporta le soluzioni di tutti i sottoproblemi è stata fin'ora usata solamente per calcolare il valore della soluzione. Può essere usata anche per ritrovare la soluzione.

- Come vedremo questa è una proprietà generale degli algoritmi di Programmazione Dinamica: in un primo momento ci si può concentrare sul calcolo del valore della soluzione e successivamente la tabella utilizzata per ottenere questo valore potrà essere usata per ritrovare la soluzione cui quel valore corrisponde.

- L'idea è di percorrere all'indietro la tabella, a partire dall'elemento che rappresenta il valore del problema, di elemento in elemento seguendo a ritroso le "decisioni" che hanno portato a determinare quel valore.

- Come esempio consideriamo la tabella T del problema FILE relativamente all'istanza $C = 10$ e 6 file di dimensioni $lista = [1, 5, 3, 4, 2, 2]$.

i \ c	0	1	2	3	4	5	6	7	8	9	10	
0	0	0	0	0	0	0	0	0	0	0	0	
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	1	1	1	5	6	6	6	6	6	5
3	0	1	1	3	4	5	6	6	8	9	9	3
4	0	1	1	3	4	5	6	7	8	9	10	4
5	0	1	2	3	4	5	6	7	8	9	10	2
6	0	1	2	3	4	5	6	7	8	9	10	2

- Tenendo conto che la tabella è stata calcolata per mezzo della seguente formula ricorsiva:

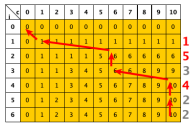
$$T[i,\ c] = \begin{cases} 0 & \text{se } i = 0 \\ T[i-1,\ c] & \text{se } c < lista[i-1] \\ \max(T[i-1,\ c],\ lista[i-1] + T[i-1,\ c - lista[i-1]]) & \text{altrimenti} \end{cases}$$

- Tenendo conto che la tabella è stata calcolata per mezzo della seguente formula ricorsiva:

$$T[i,\ c] = \begin{cases} 0 & \text{se } i = 0 \\ T[i-1,\ c] & \text{se } c < lista[i-1] \\ \max(T[i-1,\ c],\ lista[i-1] + T[i-1,\ c - lista[i-1]]) & \text{altrimenti} \end{cases}$$

- Nella figura qui sotto a partire da $T[n, C]$ da ogni elemento toccato della tabella $T[k, c]$ è tracciata una freccia verso l'elemento in base al quale è stato calcolato $T[k, c]$ (questo elemento sarà $T[k-1, c]$ o $T[k-1, c - lista[k]]$).
  Quindi se la freccia è verso $T[k-1, c]$ (una freccia verticale) vuol dire che il k-esimo file non è scelto mentre se la freccia è verso $T[k-1, c - lista[k]]$ (una freccia obliqua) vuol dire che il k-esimo file è stato scelto.

i \ c	0	1	2	3	4	5	6	7	8	9	10	
0	0	0	0	0	0	0	0	0	0	0	0	
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	1	1	1	5	6	6	6	6	6	5
3	0	1	1	3	4	5	6	6	8	9	9	3
4	0	1	1	3	4	5	6	7	8	9	10	4
5	0	1	2	3	4	5	6	7	8	9	10	2
6	0	1	2	3	4	5	6	7	8	9	10	2



- Le frecce rosse evidenziano le decisioni a partire dall'elemento $T[n, C]$ che fornisce il valore della soluzione ottima. In base a queste è possibile dire quali file sono stati scelti nella soluzione ottima (i file di dimensioni 4, 5, e 1).

- Quindi se $T[k, c] = T[k-1, c]$, il k-esimo file non è scelto (nella soluzione del sotto-problema $T[k, c]$), altrimenti $T[k, c] > T[k-1, c]$ e il k-esimo file è stato scelto.

Implementazione:

```
def iter_file1(lista,C):
    #restituisce il valore della soluzione ottima e gli indici dei file
    #da prendere per ottenerla
    T=[[0 for c in range(0,C+1)] for i in range(0, len(lista)+1)]
    for i in range(1,len(lista)+1):
        for c in range(0, C+1):
            if lista[i-1]<=c:
                T[i][c]=max(T[i-1][c],lista[i-1]+ T[i-1][c-lista[i-1]])
            else:
                T[i][c]=T[i-1][c]

    valore=T[len(lista)][C]
    sol=set()
    i=len(lista)
    while i>0:
        if T[i][valore]!=T[i-1][valore]:
            sol.add(i-1)
            valore-=lista[i-1]
        i-=1
    return T[len(lista)][C], sol

>>> iter_file1(lista1,C)
(10, {0, 1, 3})
```

- Nella prima parte viene pagato $O(nC)$ per il calcolo della tabella.
- Nella seconda parte, alla ricerca dei file presenti nella soluzione ottima, la tabella viene visitata a partire dall'ultima cella $T[n, C]$, un riga in meno per volta. Il costo di questa visita è $O(n)$.
- la complessità è $O(nC)$