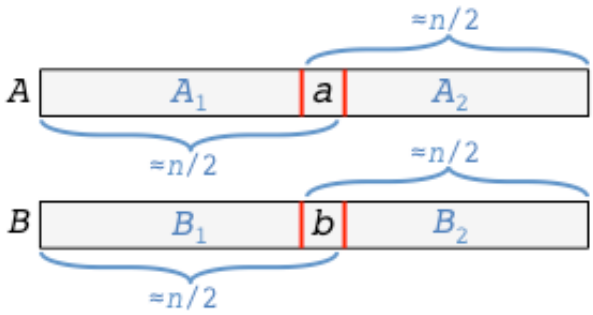
Progettazione di Algoritmi - lezione 16

Discussione dell'esercizio [mediana]

Siano A e B due array di n interi, ordinati in senso non decrescente. Dobbiamo descrivere un algoritmo che trovi la mediana dei $2n$ elementi in tempo O(log$n$). Chiaramente, potremmo fondere i due array ordinati in un array ordinato di $2n$ elementi e poi prendere l'elemento centrale, però questo richiede tempo O(n). Per farlo in tempo O(log$n$) dovremmo usare la tecnica Divide et Impera similmente alla ricerca binaria.

Essendo i due array A e B ordinati possiamo determinare facilmente il valore mediano a di A e il valore mediano b di B. Confrontando a e b possiamo vedere dove cercare il valore mediano c dei $2n$ elementi.



Se $a \le b$, allora deve essere $a \le c \le b$. Perchè se fosse $c < a$, allora c sarebbe minore di tutti gli elementi in A_2 e B_2 che sono più di n (vedremo i dettagli dopo) e se fosse $c > b$, c sarebbe maggiore di tutti gli elementi in A_1 e B_1 che sono più di n. In questo caso quindi continuiamo la ricerca di c come mediana di A_2 e B_1. Se invece $a > b$, in modo del tutto simmetrico deve essere $b \le c \le a$. E continuiamo la ricerca di c come mediana di A_1 e B_2.

Quindi a seconda dell'esito del confronto delle due mediane di A e B, continuiamo la ricerca della mediana ricorsivamente o nei sottoarray A_2 e B_1 o nei sottoarray A_1 e B_2. In entrambi i casi i sottoarray hanno dimensione circa $n/2$. Per garantire però che i due sottorray abbiano sempre la stessa dimensione (come i due array iniziali A e B), dobbiamo scegliere con accortezza le mediane nel caso n pari.

Poniamo $ma = \lceil n/2 \rceil$, la posizione della mediana in un array ordinato, e sia $mb = ma$ se n è dispari e $mb = ma + 1$ se n è pari (se n è pari anche $ma + 1$ è la posizione di un valore mediano). I valori delle mediane sono $a = A[ma]$ e $b = B[mb]$. Rivediamo nei dettagli i due casi. Sia $n = 2k - p$, dove $p = 0$ se n è pari e $p = 1$ altrimenti. Si osservi che $ma = k$ e $mb = k + 1 - p$.

- Se $a \le b$, allora $a \le c \le b$. Infatti, se fosse $c < a$, per ogni elemento x in $A[ma \cdots n]$ o in $B[mb \cdots n]$ si avrebbe $c < x$ e quindi ci sarebbero

  $n - ma + 1 + n - mb + 1 = 2k - p - k + 1 + 2k - p - k - 1 + p + 1 = 2k - p + 1 = n + 1$

  elementi strettamente maggiori di c (e quindi gli elementi minori o uguali a c sarebbero al più $n - 1$, cioè meno della metà), in cotraddizione con l'ipotesi che c è il valore mediano. Analogamente se fosse $c > b$, ci sarebbero almeno $n + 1$ elementi strettamente minori di c, che ancora una volta è in cotraddizione con l'ipotesi che c è il valore mediano. Quindi c è la mediana dei sottoarray $A[ma \cdots n]$ e $B[1 \cdots mb]$. La dimensione del primo è $n - ma + 1 = 2k - p - k + 1 = k + 1 - p = mb$, che è uguale a quella del secondo.
- Se $a > b$, allora $b \le c \le a$ con un ragionamento simmetrico a quello dell'altro caso. Quindi c è la mediana dei sottoarray $A[1 \cdots ma]$ e $B[mb \cdots n]$. La dimensione del secondo è $n - mb + 1 = 2k - p - k - 1 + p + 1 = k = ma$, che è uguale a quella del primo.

Ecco allora lo pseudo-codice:

```
MED(A: array di interi, B: array di interi, n: dimensione dei due array)
    IF n ≤ 2 THEN
        RETURN mediana di A e B, calcolata esaustivamente
    ELSE
        ma <- ⌈n/2⌉
        IF n è pari THEN mb <- ma + 1
        ELSE mb <- ma
        IF A[ma] ≤ B[mb] THEN
            RETURN MED(A[ma···n], B[1···mb], mb)
        ELSE
            RETURN MED(A[1···ma], B[mb···n], ma)
```

Siccome la dimensione degli array all'incirca si dimezza ad ogni chiamata ricorsiva, la complessità $T(n)$ soddisfa $T(n) = T(n/2) + O(1)$ e quindi è $\Theta(\log n)$.

Dalla Memoizzazione…

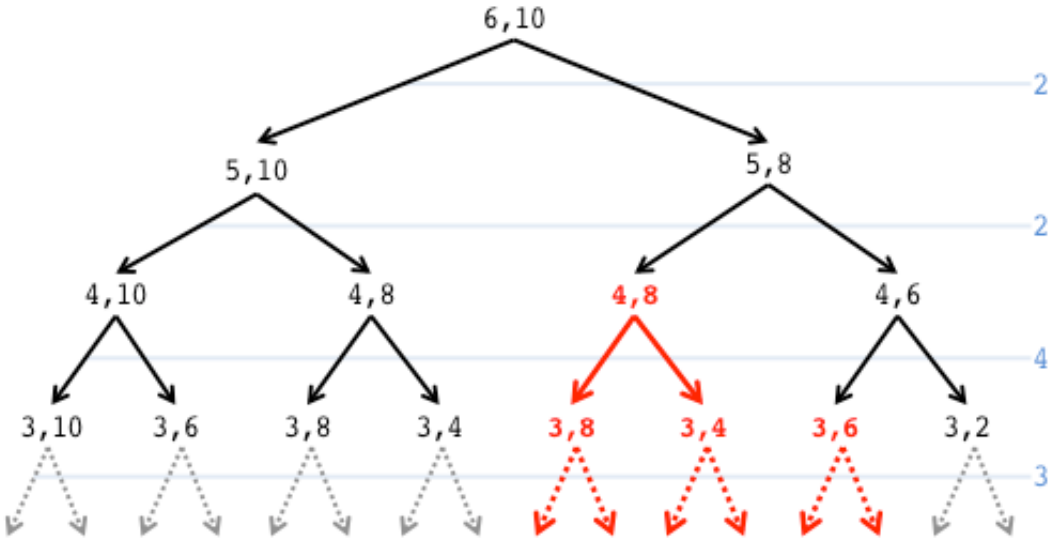
Ricordiamo il problema *file*:

> Dati n file di dimensioni $s_1,\ldots,s_n$ e un disco di capacità C, vogliamo trovare un sottoinsieme dei file che può essere memorizzato sul disco e che massimizza lo spazio usato.

Sappiamo che è un problema difficile, ovvero non si conoscono algoritmi efficienti che trovano sempre la soluzione ottima. I migliori algoritmi si basano sulla ricerca esaustiva della soluzione ottima. Un algoritmo che effettua una ricerca esaustiva per il problema *file* lo possiamo facilmente descrivere in modo ricorsivo. Per semplicità ci limiteremo a calcolare il valore di una soluzione ottima, cioè il massimo spazio del disco che può essere usato. La ricerca esaustiva deve quindi esaminare tutte le possibili soluzioni, ovvero tutti i possibili sottoinsiemi degli n file. Per ogni file abbiamo solamente due possibilità: o lo scegliamo (se è possibile memorizzarlo) o non lo scegliamo. Per ognuna delle due possibilità la ricerca continua ricorsivamente relativamente all'insieme dei file rimasti e la capacità residua (se il file è stato scelto).

```
FILE_REC(S: array delle dimensioni, k: numero file, c: capacità)
    IF k = 0 THEN
        RETURN 0
    ELSE
        max <- FILE_REC(S, k - 1, c)                  /* Il k-esimo file non è scelto */
        IF S[k] ≤ c THEN
            m <- S[k] + FILE_REC(S, k - 1, c - S[k])  /* Il k-esimo file è scelto */
            IF m > max THEN max <- m
        RETURN max
```

La chiamata iniziale è `FILE_REC(S, n, C)`. La complessità, nel caso peggiore, è molto elevata potendo arrivare a $\Theta(2^n)$ (ovviamente non può superare $O(2^n)$ dato che 2^n è il numero di tutti i sottoinsiemi degli n file). Per avere una migliore intuizione su come viene effettuata la ricerca, consideriamo un'istanza e tracciamo l'albero (parziale) delle chiamate ricorsive: $n = 6$, $C = 10$ e dimensioni dei file 1,5,3,4,2,2 (in celeste la dimensione del file considerato a quel passo)



In rosso sono marcate le chiamate ricorsive duplicate, cioè quelle che sono già state calcolate ma l'algoritmo comunque le ricalcola. Più l'istanza è grande e più è probabile che ci siano un gran numero di chiamate ricorsive duplicate. Possiamo evitare di fare questo inutile lavoro di ricalcolo? Basta usare una tabella in cui salviamo i risultati delle chiamate ricorsive evitando così di ricalcolarle:

```
T: tabella (n+1)x(C+1) inizializzata a -1

FILE_REC_M(S: array delle dimensioni, k: numero file, c: capacità, T: tabella)
    IF T[k, c] = -1 THEN    /* Se non è stato ancora calcolato */
        IF k = 0 THEN
            T[k, c] <- 0
        ELSE
            max <- FILE_REC_M(S, k - 1, c, T)
            IF S[k] ≤ c THEN
                m <- S[k] + FILE_REC_M(S, k - 1, c - S[k], T)
                IF m > max THEN max <- m
            T[k, c] <- max
    RETURN T[k, c]
```

Come si vede, un risultato viene calcolato solo se non è già stato calcolato e memorizzato nella tabella `T` e quando un nuovo risultato è calcolato, viene memorizzato in `T`. Questa tecnica si chiama **memoizzazione** dal termine inglese *memoization* coniato alla fine degli anni '60 e derivato dal latino *memorandum* con il significato di "trasformare una funzione in qualcosa da memorizzare". La tecnica è generale e può essere applicata a tantissimi algoritmi per una grande varietà di problemi.

La memoizzazione ha un costo che è la memoria per mantenere la tabella. Ma quanto tempo fa risparmiare? È difficile fare una stima perchè, in generale, dipende fortemente dall'istanza. Però possiamo dire che il tempo di calcolo della versione memoizzata, almeno per il nostro problema, è limitato dalla dimensione della tabella. Infatti, l'agoritmo `FILE_REC_M` esegue O(1) operazioni in ogni chiamata e il numero totale di chiamate è limitato dalla dimensione della tabella, quindi la complessità è limitata da $O(nC)$, in realtà è proprio $\Theta(nC)$ perchè la tabella va inizializzata. Rispetto al tempo di calcolo $\Theta(2^n)$ della versione non memoizzata, abbiamo risparmiato? In alcuni casi sì e in altri no. 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 la capacità è C = 1.000.000 (assumiamo che le dimensioni dei file e la capacità siano espresse in multipli di MB, così C equivale a 1TB). Se la dimensione massima dei file è 1000 (cioè 1GB), un qualsiasi sottoinsieme di 1000 file può essere memorizzato sul disco. L'algoritmo non memoizzato `FILE_REC` eseguirà tutte le chiamate ricorsive (cioè sarà sempre possible scegliere il `k`-esimo file) almeno relativamente ai primi 1000 file (da n = 100.000 a 99.000). Quindi la complessità sarà almeno dell'ordine di 2^{1000}. L'algoritmo memoizzato `FILE_REC_M` invece ha una complessità dell'ordine di nC = 100.000x1.000.000 =

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

...alla Programmazione Dinamica

Come l'istanza d'esempio con n = 100.000 mostra, la profondità dell'albero delle chiamate ricorsive (in entrambe le versioni) può essere molto grande e portare a stack overflow. Nella versione memoizzata possiamo concentrarci direttamente sul calcolo della tabella T eliminando così la ricorsione. Per ogni k = 0,1,…,n e per ogni c = 0,1,…,C:

$$T[k,c] = \text{massimo spazio usabile dai primi } k \text{ file (di dimensioni } S[1],\ldots,S[k]) \text{ su un disco di capacità } c$$

È chiaro che, per ogni c, $T[0, c] = 0$. Inoltre, per ogni $k \geq 1$ e per ogni $c \geq 0$,

$$T[k,c] = \begin{cases} T[k-1,c] & \text{se } S[k] > c \\ \max\{T[k-1,c], S[k] + T[k-1, c-S[k]]\} & \text{altrimenti} \end{cases}$$

Così l'algoritmo `FILE_REC_M` può essere trasformato in uno che non usa la ricorsione perchè calcola direttamente gli elementi della tabella T:

```
FILE_T(S: array delle dimensioni, n: numero file, C: capacità)
    T: tabella (n+1)x(C+1)
    FOR c <- 0 TO C DO T[0, c] <- 0
    FOR k <- 1 TO n DO
        FOR c <- 0 TO C DO
            T[k, c] <- T[k - 1, c]
            IF S[k] ≤ c AND T[k, c] < S[k] + T[k - 1, c - S[k]] THEN
                T[k, c] <- S[k] + T[k - 1, c - S[k]]
    RETURN T[n, C]
```

In termini dell'albero delle chiamate ricorsive, l'algoritmo `FILE_T`, 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]`.

L'algoritmo `FILE_T` è un esempio di algoritmo che usa la tecnica della **Programmazione Dinamica** (dall'inglese *Dynamic Programming*). Il termine Dynamic Programming fu introdotto negli anni '40 da Richard Bellman per intendere un procedimento in cui bisogna prendere delle decisioni (ottime) basate sui risultati di decisioni precedenti. Il termine Programmazione (Programming) non ha nulla a che fare con la programmazione dei computer ma è inteso come sinonimo di pianificazione, organizzazione (scheduling, planning).

La Programmazione Dinamica (DP) può essere descritta in termini generali come un metodo che risolve un problema P partendo dalle soluzioni dei problemi più piccoli dello stesso tipo di P, usando tali soluzioni per risolvere problemi un po' più grandi le cui soluzioni sono poi usate per risolvere problemi ancora più grandi e così via fino ad arrivare al problema originale P. Nel caso del problema *file*, i problemi più piccoli sono i $T[0, c]$ e i problemi $T[k, \cdot]$ sono più grandi dei problemi $T[k - 1, \cdot]$. La proprietà cruciale che una tale stratificazione in sotto-problemi deve avere è che ci deve essere un modo per calcolare la soluzione di un problema a partire dalle soluzioni dei problemi più piccoli, che sono stati già risolti.

Il fatto che la DP si basi sulla soluzione di problemi più piccoli o sotto-problemi, potrebbe far pensare che ci sia una certa somiglianza con la tecnica Divide et Impera. In realtà la somiglianza è solo superficiale, nella tecnica Divide et Impera i sotto-problemi sono generalmente disgiunti mentre nella DP si sovrappongono ampiamente. Gli algoritmi Divide et Impera si prestano naturalmente ad essere implementati in modo ricorsivo, mentre quelli di DP sono iterativi perchè calcolano le soluzioni dei sotto-problemi organizzati in un qualche tipo di tabella.

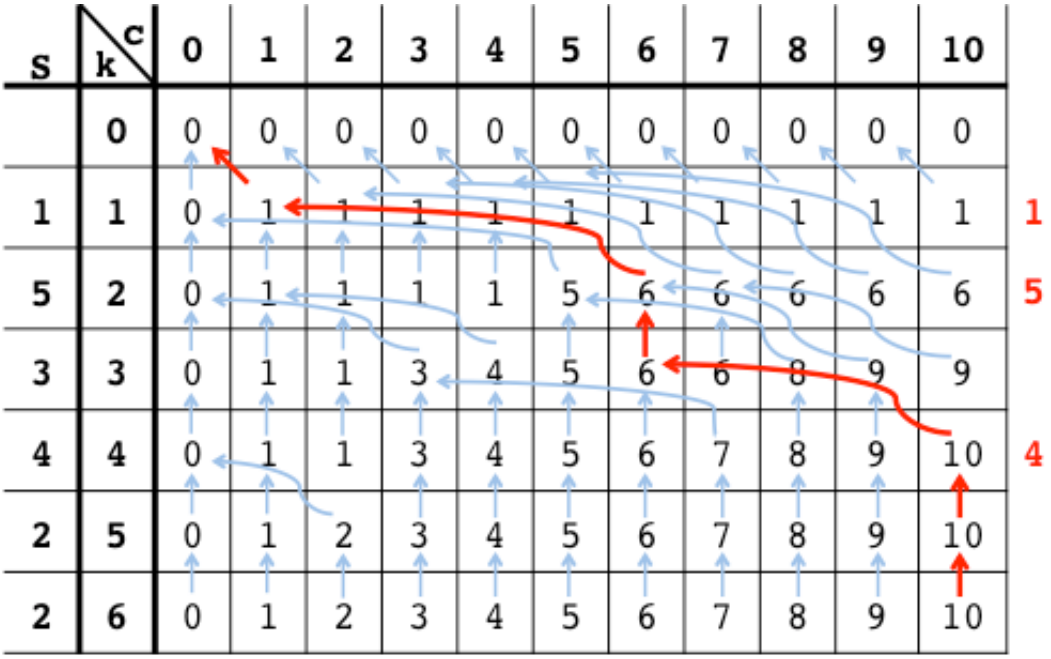
La DP è una delle tecnica algoritmiche più generali e potenti. Spesso permette di ottenere algoritmi ottimali laddove tutte le altre tecniche falliscono. Inoltre, una volta che un algoritmo di DP è stato ben definito la correttezza

discende quasi direttamente dalla sua stessa definizione e l'analisi della complessità è anch'essa diretta. Di contro è piuttosto difficile applicare la tecnica, almeno tipicamente. I casi, come l'esempio che abbiamo visto del problema *file*, in cui l'applicazione della DP discende da un algoritmo di ricerca esaustiva (tramite la memoizzazione), fanno eccezione. Purtroppo, la maggior parte delle applicazioni di successo della DP non ricadono in quella casistica. Solamente una buona esperienza basata sulla conoscenza approfondita di parecchie applicazioni della DP può portare all'acquisizione della capacità di saper applicare la tecnica.

DP: dal valore ottimo alla soluzione ottima

La tabella che riporta le soluzioni di tutti i sotto-problemi è stata usata solamente per calcolare il valore della soluzione ottima. Però può essere usata per ritrovare anche la soluzione ottima. Come vedremo questa è una proprietà generale degli algoritmi di DP. Nel senso che ci si può concentrare sul calcolo del valore ottimo e successivamente la tabella potrà essere usata per ritrovare anche la soluzione ottima. L'idea è di percorrere all'indietro la tabella, a partire dall'elemento che rappresenta il valore ottimo del problema, di elemento in elemento seguendo le "decisioni" che hanno portato a determinare il valore (ottimo) dell'elemento successivo.

Come esempio consideriamo la tabella T del problema *file* relativamente all'istanza $n = 6$, $C = 10$ e dimensioni dei file 1,5,3,4,2,2. Nella figura qui sotto da ogni elemento $T[k, c]$ è tracciata una freccia verso l'elemento precedente (o $T[k - 1, c]$ o $T[k - 1, c - S[k]]$) in base al quale è calcolato $T[k, c]$. 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 $T[k - 1, c - S[k]]$ vuol dire che il k-esimo file è scelto.



Le freccie 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 4,5,1). Quindi se $T[k, c] = T[k - 1, c]$, il k-esimo file non è scelto (nella soluzione del sotto-problema k, c), altrimenti, cioè $T[k, c] > T[k - 1, c]$, il k-esimo file è scelto.

```
FILE_SOL(S: array delle dimensioni, T: tabella, n: numero file, C: capacità)
    SOL <- insieme vuoto
    c <- C
    FOR k <- n DOWNTO 1 DO
        IF T[k, c] > T[k - 1, c] THEN
            SOL <- SOL ∪ {k}
            c <- c - S[k]
    RETURN SOL
```

Quindi `FILE_SOL(S, T, n, C)` ritorna l'insieme degli indici dei file della soluzione ottima calcolata dalla tabella

`T`. Per l'istanza dell'esempio sopra, ritorna {1,2,4}. Per il problema *file* è stata sufficiente la tabella che permette di calcolare il valore ottimo per ritrovare anche la soluzione ottima ma, come vedremo, potrebbero servire anche altre informazioni.

Se non si è interessati a ritrovare la soluzione ottima si può risparmiare memoria. Infatti, osservando il programma `FILE_T` ci accorgiamo che non è necessario mantenere l'intera tabella. Per calcolare la *k*-esima riga della tabella è sufficiente avere la riga precedente. Ma nel nostro caso procedendo da destra verso sinistra, cioè per *c* decrescenti, è sufficiente un solo array:

```
FILE_T2(S: array delle dimensioni, n: numero file, C: capacità)
    R: array di dimensione C+1, inizializzato a 0
    FOR k <- 1 TO n DO
        FOR c <- C DOWNTO 0 DO
            IF S[k] ≤ c AND R[c] < S[k] + R[c - S[k]] THEN
                R[c] <- S[k] + R[c - S[k]]
    RETURN R[C]
```

Esercizio [resto]

Disponendo di un numero illimitato di banconote di un paese straniero di n tagli diversi $1 = v_1 < v_2 < \cdots < v_n$, bisogna dare un resto $R \geq 1$. Descrivere un algoritmo che in tempo $O(nR)$ calcola il numero minimo di banconote necessarie per produrre il resto R.