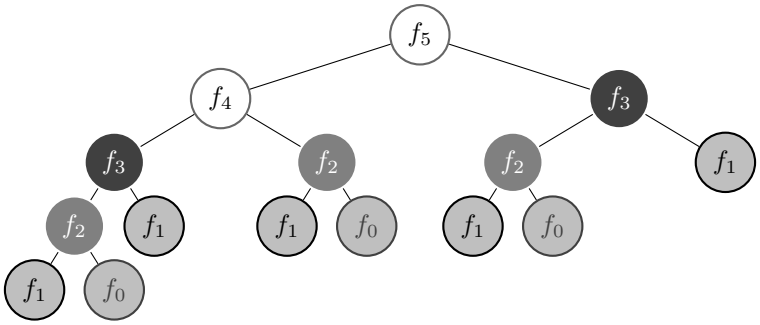
Giorno 9

1 Programmazione dinamica

Molti problemi risolti con il paradigma *divide et impera*, prendiamo ad esempio il calcolo dell'n-esimo numero di fibonacci, vi sono sottoproblemi sovrapposti che vengono risolti più di una volta.

Ad esempio, nel calcolare fib_5 con la definizione $fib_n = fib_{n-1} + fib_{n-2}$, con $fib_0 = fib_1 = 1$ come caso base, i sottoproblemi sono i seguenti (sono evidenziati quelli ripetuti):



La Programmazione Dinamica risolve ogni sottoproblema una sola volta.

I problemi che possono essere risolti dalla programmazione dinamica hanno determinate caratteristiche:

1. Sottoproblemi sovrapposti
2. Sottostruttura ottima.

Un problema ha *sottostruttura ottima* quando la sua soluzione ottima contiene le soluzioni ottime dei suoi sottoproblemi. Vedremo tra poco degli esempi di sottostruttura ottima.

1.1 Fibonacci DP

Ricordiamo che il costo di Fibonacci DeI è:

$$T(n) = T(n-1) + T(n-2) + O(1) > 2T(n-2) > 2^2T(n-4) > ... > 2^iT(n-2i)$$

Soluzione dipende da caso base se $n-2i = \begin{cases} 0 \\ 1 \end{cases}$, ossia se $\begin{cases} i = \frac{n}{2} & \implies T(n) > 2^{\frac{n}{2})} \cdot T(0) = O(2^{\frac{n}{2}}) \\ i = \frac{n-1}{2} & \implies T(n) > 2^{\frac{n-1}{2})} \cdot T(1) = O(2^{\frac{n-1}{2}}) \end{cases}$

Ci sono più approcci alla programmazione dinamica. Uno applicabile al probema dell'n-esimo numero di Fibonacci è l'*approccio bottom-up*.

1.1.1 Approccio bottom-up

```
1  Fibo(n):
2      a = 0, b = 1;
3      for(i = 0 to n):
4          c = a + b;
5          a = b;
6          b = c;
7      return b;
```

A differenza del paradigma divide et impera, che ha un approccio *top-down*, ossia parte dal caso che si vuole risolvere e scende ricorsivamente fino ai casi base, si parte dai casi base e si calcola iterativamente la soluzione (costo lineare).

1.1.2 Memoization

Un altro approccio è quello della *memoization*: si crea un dizionario di appoggio per tenere traccia di quali sottoproblemi hanno già soluzione. (costo lineare).

```
1  D = new dizionario;
2  Fibo(n):
3      if(n == 0): return 0;
4      if(n == 1): return 1;
5      if(n in D): return D.n;
6      else:
7          tmp = Fibo(n-1) + Fibo(n-2);
8          D.n = tmp;
9          return tmp;
```

1.2 Distanza di Levenshtein

La distanza di Levenshtein è una metrica $E_d(X, Y)$ che rappresenta il numero minimo di operazioni su caratteri (inserimenti, cancellazioni, sostituzioni) che trasformano una stringa X in un'altra Y.

Il seguente schema può essere usato per applicare la programmazione dinamica a vari problemi:

1. **Sottoproblemi**

   Date le stringhe X e Y *allineamento* è la struttura $\begin{matrix} X \\ Y \end{matrix}$, dove i caratteri della stringa X sono sovrapposti a quelli di Y.

   Siano $x_1...x_n$ i caratteri di X e $y_1...y_m$ i caratteri di Y.

   Definiamo un allineamento *ottimo* quando tutti i caratteri delle due stringhe coincidono.

   $$\begin{matrix} X \\ Y \end{matrix} = \begin{matrix} x_1, x_2, ..., x_n \\ y_1, y_2, ..., y_m \end{matrix} \text{ è un allineamento ottimo } \implies \begin{matrix} X[i, ..., j] \\ Y[i, ..., j] \end{matrix} = \begin{matrix} x_i, ..., x_j \\ y_i, ..., y_j \end{matrix} \text{ lo è per } 1 \leq i \leq j \leq n$$

   Quindi il problema ha sottostruttura ottima (si dimostra per assurdo obv).

   Per individuare i sottoproblemi concentriamoci sugli ultimi caratteri di X ed Y:

   $$\begin{matrix} X[1, n-1], x_n \\ Y[1, m-1], y_m \end{matrix}$$

   I due caratteri x_n e y_m possono essere uguali o diversi.

   Possiamo agire in quattro modi diversi:

   (a) Se sono uguali, possiamo **non fare niente**. Effetto:

   $$E_d(X, Y) = E_d(X[1, n-1], Y[1, m-1])$$

   (b) In caso contrario, possiamo **sostituire** y_m ad x_n in X:

   $$E_d(X, Y) = 1 + E_d(X[1, n-1], Y[1, m-1])$$

   (c) Se non sono uguali, possiamo anche **inserire** y_n in fondo ad X:

   $$E_d(X, Y) = 1 + E_d(X[1, n], Y[1, m-1])$$

   (d) Se non sono uguali, possiamo **cancellare** x_n da X:

   $$E_d(X, Y) = 1 + E_d(X[1, n-1], Y[1, m])$$

2. **Sottoproblemi elementari**

   (a) Per trasformare la stringa X, lunga n, nella stringa vuota, si fanno n cancellazioni.

   (b) Per trasformare la stringa vuota in una stringa Y lunga m, si fanno m inserimenti.

3. **Definizione ricorsiva del problema**: Siano i e j gli ultimi caratteri di X ed Y. Sia $DP(i, j)$ la funzione ricorsiva (**D**ynamic **P**rogramming).

   $$DP(i,j) = \begin{cases} DP(i-1, j-1) & \text{se } A[i] = A[j] \\ \min \begin{cases} 1 + DP(i-1, j-1) & \text{(sostituzione)} \\ 1 + DP(i, j-1) & \text{(cancellazione)} \\ 1 + DP(i-1, j) & \text{(inserimento)} \end{cases} & \text{o/w} \end{cases}$$

4. **Risoluzione (tabella)** Creiamo una tabella tale che la posizione i, j contenga $E_d(X[1, i], Y[1, j])$ (se i o $j = 0$: X o Y = stringa vuota, la tabella contiene i risultati dei sottoprob. elementari).

		y_1	y_2	$\dots$
	0	1	2	$\dots$
x_1	1	$DP(1,1)$	$DP(1,2)$	
x_2	2	$DP(2,1)$	$DP(2,2)$	
$\vdots$	$\vdots$			$\ddots$

   Segue un esempio che spiega come utilizzare questa tabella per il calcolo.

1.2.1 Esempio di calcolo di E_d

Vogliamo calcolare l'edit distance tra ALBERO e LABBRO.

$T =$

		L	A	B	B	R	O
	0	1	2	3	4	5	6
A	1						
L	2						
B	3						
E	4						
R	5						
O	6						

$T =$

		L	A	B	B	R	O
	0	1	2	3	4	5	6
A	1	1	1	2	3	4	5
L	2	1	2	2	3	4	5
B	3	2	2	2	2	3	4
E	4	3	3	3	3	3	4
R	5	4	4	4	4	3	4
O	6	5	5	5	5	4	**3**

Se ogni posizione della tabella T contiene la distanza tra le due sottostringhe, allora la posizionne $(n, m) = (6, 6)$ contiene il risultato del problema.
Ogni sottoproblema non elementare è definito in funzione di altri sottoproblemi, perciò vale la relazione:

$$T[i,j] = \begin{cases} T[i-1, j-1] & \text{se } X[i] = Y[j] \\ \min \begin{cases} 1 + T[i-1, j-1] & \text{(sostituzione)} \\ 1 + T[i, j-1] & \text{(cancellazione)} \\ 1 + T[i-1, j] & \text{(inserimento)} \end{cases} & \text{o/w} \end{cases}$$

che riflette la definizione ricorsiva. Applicando questa formula si ottiene la tabella a destra.
Complessità in tempo = Complessità in spazio = $\Theta(n \cdot m)$

1.3 Knapsack

Il Knapsack Problem, o *problema dello zaino*, è un probema di ottimizzazione. Sia A un insieme di n elementi $a_1 ... a_n$ che hanno associati un **valore** ed un **peso**, risp. $v_1, ..., v_n$ e $w_1, ..., w_n$. Qual è il valore totale del sottoinsieme S di A tale che:

1. Dato un peso massimo W, la somma dei pesi degli elementi è $\leq W$.
2. Il valore è massimo.

1.3.1 Problema dello zaino intero

Assumiamo ogni elemento possa essere incluso o meno in S (non si possono prendere frazioni di un elemento).
Se proviamo a risolvere questo problema senza fare uso della programmazione dinamica, utilizzando degli algoritmi greedy, che ad ogni scelta locale massimizzano, il valore o il rapporto v/w, possiamo facilmente renderci conto che questi non funzionano, poiché le soluzioni dipendono dalle scelte già prese.
Utilizziamo lo stesso schema:

1. **Sottoproblemi**

   Consideriamo gli elementi di A in ordine; per ognuno ci sono solo due possibilità: è incluso in S o no?

   (a) Se a_n è incluso, allora sottraiamo il suo peso da W e ci chiediamo se a_{n-1} è incluso.

   (b) In caso contrario, W rimane lo stesso, e ci chiediamo se a_{n-1} è incluso.

2. **Sottoproblemi elementari**: $W = 0$ o $n = 0 \implies$ risultato $= 0$.

3. **Definizione ricorsiva**: Sia i uguale all'indice dell'ultimo elemento della sequenza $a_1 \dots a_i$. Sia j uguale al peso massimo rimanente (inizialmente W, poi sottraggo il peso degli elementi inseriti)

$$DP(i,j) = \max \begin{cases} DP(i-1, j) & a_i \notin S \\ v_i + DP(i-1, j-w_i) & a_i \in S \end{cases}$$

4. **Tabella**: Segue dalla definizione ricorsiva e dai sottoproblemi elementari.

1.3.2 Problema dello zaino continuo (o frazionario)

Il problema dello zaino continuo è una versione del problema in cui si possono prendere frazioni degli elementi di A.

In questo caso si può usare un algoritmo greedy, che ordina gli elementi in base a v/w ed inserisce ordinatamente in S più elementi interi possibili, frazionando l'ultimo.

1.3.3 Complessità di Knapsack

La complessità in tempo del problema dello zaino intero è $\Theta(Wn)$. Questo è uno dei casi in cui la complessità sembra polinomiale, ma in realtà non lo è.

Se n è la dimensione di un insieme, W è un intero; la dimensione in memoria della rappresentazione di un intero è uguale al suo numero di cifre binarie. Questo è uguale a $b = O(\log W)$.

La complessità è perciò $O(2^b n)$, che non è un tempo polinomiale.

Un limite inferiore alla complessità della versione continua del problema è $O(\log_n)$, poiché si devono ordinare gli elementi, ma è possibile dimostrare che il problema è risolubile in $O(n)$.

1.4 Scacchiera

Il problema della scacchiera consiste nel trovare il numero di cammini che portano dalla casella $(0, 0)$ alla $(n-1, n-1)$ di una scacchiera $n \times n$. Le uniche mosse valide sono $\downarrow$ e $\rightarrow$.
Scriviamo nella tabella il numero di cammini fino a quella cella, seguendo la regola:

$$A[i, j] = A[i, j-1] + A[i-1, j]$$

e conoscendo il risultato dei sottoproblemi elementari: $A[0, j] = 1$, $A[i, 0] = 1$.

1	1	1	1	1	1
1	2	3	4	5	6
1	3	6	10	15	21
1	4	10	20	35	56
1	5	15	35	70	126
1	6	21	56	126	252

(heh)