14 Programmazione dinamica

Si applica negli algoritmi ricorsivi in cui i sotto problemi ottenuti dalla tecnica *Divide et impera* si ripropongono più volte, causando uno spreco di risorse considerevole. Si dice che questi sotto problemi non sono **indipendenti**.
La tecnica consiste nel memorizzare le soluzioni in una **tabella** dei sotto problemi in modo da potervi accedere quando li si incontra di nuovo senza doverli risolvere nuovamente.

Esempio 14.1 (Fibonacci)**.** Generare la sequenza di Fibonacci, che ricordiamo essere definita come:

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n \geq 2 \end{cases}$$

```
Fib(n)
  if(n==0) return 0;
  if(n==1) return 1;
  return Fib(n-1)+Fib(n-2);
```

Listing 29: Fibonacci

In questa soluzione di codice le somme eseguite sono in numero esponenziale:

$$T(n) = \begin{cases} 0 & n \leq 1 \\ 1 + T(n-1) + T(n-2) & n > 1 \end{cases}$$

Per calcolare la successione di Fibonacci con **memoization** di tipo **top down** usiamo il seguente algoritmo, sempre ricorsivo:

```
Fib(n)
  F: new array(0:n) // Dimensione n+1
  for i=0 to n
    F(i) = -1
  return FibRic(n, F)

FibRic(n, F)
  if(n==0 || n==1) return n;
  if(F(n) != -1) return F[n]
  else
    F[n] = FibRic(n-1, F) + FibRic(n-2, F);
    return F[n];
```

Listing 30: Fibonacci dinamico top-down

Utilizzando invece il metodo **bottom-up**:

```
Fib(n)
  F: new array(0:n) // Dimensione n+1
  F[0] = 0;
  F[1] = 1;
  for i=2 to n
    F(i) = F[i-1] + F[i-2]
  return F[n]
```

Listing 31: Fibonacci dinamico bottom-up

La complessità di questi algoritmi è $\Theta(n)$ in tempo e $\Theta(n)$ in spazio, a differenza dell'algoritmo non dinamico che ha una complessità di ϕ^n in tempo e Per ottimizzare l'algoritmo in spazio possiamo fare nel modo seguente:

```
Fib(n)
  if(n==0) return 0;
  if(n==1 || n==2) return 1;
  a = 1
  b = 1
  for i=3 to n
    c = a+b
    a = b
```

```
      b = c
    return b;
```

Listing 32: Fibonacci spazio costante

Osservazione 14.0.1 (Complessità in spazio)**.** Come sappiamo il numero di cifre necessarie per rappresentare un numero n è $\log_x n$ dove x è la base in cui scriviamo il numero. Di conseguenza in quest'ultimo algoritmo in realtà è **pseudo-polinomiale** in quanto passiamo da un'istanza di input logaritmica ad una complessità lineare.

14.1 Struttura di un algoritmo

La programmazione dinamica si applica a problemi di ottimizzazione con queste caratteristiche:

- **Sottostruttura ottima**: la soluzione ottima del problema si può costruire dalle soluzioni ottime dei sottoproblemi
- **Sovrapposizione dei sottoproblemi** (o ripetizione)

Gli algoritmi sono costituiti da 4 fasi:

1. Definizione dei sotto problemi e dimensionamento della tabella.
2. Soluzione diretta dei sotto problemi elementari e inserimento di questi nella tabella (approccio **bottom-up**)
3. Definizione della regola ricorsiva per ottenere la soluzione di un sotto problema a partire dalle soluzioni dei sotto problemi già risolti (già nella tabella)
4. Restituzione del risultato relativo al problema di dimensione n

Esempio 14.2 (Taglio della corda)**.** Data una corda di lunghezza n e una tabella di prezzi (un pezzo di dimensione diversa ha un valore diverso), trovare il taglio ottimale della corda per massimizzare il guadagno.
Un possibile modo di affrontare il problema è tramite **brute-force**, che comporta analizzare ogni possibile taglio della corda. Notiamo che possiamo dividere la corda in 2^{n-1} possibili modi.
Un approccio **ricorsivo** al problema è quello di tagliare la corda al punto i. In questo modo abbiamo che il ricavo massimo ottenibile sarà il prezzo del pezzo tagliato $p(i)$ e il ricavo massimo della corda restante r_{n-i}. In generale il ricavo massimo quindi sarà $r_n = max_{1 \leq i \leq n}(p(i) + r(n-i))$.

```
  CUT_ROD(P, n)
    if(n==0) return 0;
    q = -∞
    for i=1 to n
      q = max{q, p[i] + CUT_ROD(P, n-i)}
    return q;
```

Listing 33: Taglio della corda

Questo algoritmo, per quanto breve, è estremamente inefficiente (esponenziale) a causa della ripetizione degli stessi sotto problemi. È quindi necessaria la *programmazione dinamica*.

Esempio 14.3 (Longest common subsequence)**.**

Esempio 14.4 (Edit distance)**.** Date due parole X, Y determinarne la distanza.
La prima fase è quella dell'**allineamento**: ad ogni carattere o spazio di X corrisponde un carattere o uno spazio di Y.
Una volta eseguito l'allineamento, si calcola la distanza con queste regole:

- **MATCH** (caratteri corrispondenti uguali) $\rightarrow$ distanza + 0
- **MISMATCH** (caratteri corrispondenti diversi) $\rightarrow$ distanza + 1
- **SPACE** (carattere e spazio) $\rightarrow$ distanza + 1

Il problema della edit-distance è determinare la distanza *minima*.

14.2 Tecnica Greedy

Esempio 14.5 (Zaino frazionario)**.** Il problema è quello dello zaino visto in precedenza ma in questo caso il ladro può prendere anche frazioni di oggetti.

Correttezza: occorre dimostrare che il problema soddisfi gli elementi della tecnica greedy:

- *Sottostruttura ottima*
- *Proprietà della scelta greedy*: per assurdo, supponiamo che qualche scelta non sia greedy (localmente ottima). Anziché scegliere un oggetto m di valore specifico $\frac{v_m}{w_m}$, scegliamo p kg dell'oggetto q di valore specifico $\frac{v_q}{w_q}$.

$$\frac{v_q}{w_q} < \frac{v_m}{w_m}$$

  Sia $r = min\{p, w_m\}$

Esempio 14.6 (Scheduling delle attività)**.** Ogni attività inizia e finisce in due istanti di tempo diversi. Il problema consiste nel massimizzare il numero di attività eseguibili rispettando il vincolo di *non*

i	1	2	3	4	5	6	7	8	9	10	11
S_i											
T_i											

sovrapposizione.

Utilizziamo la strategia greedy:

- Seleziona l'attività che finisce prima
- Elimina le attività che non sono compatibili, ovvero che si sovrappongono con quella selezionata
- Ripeti