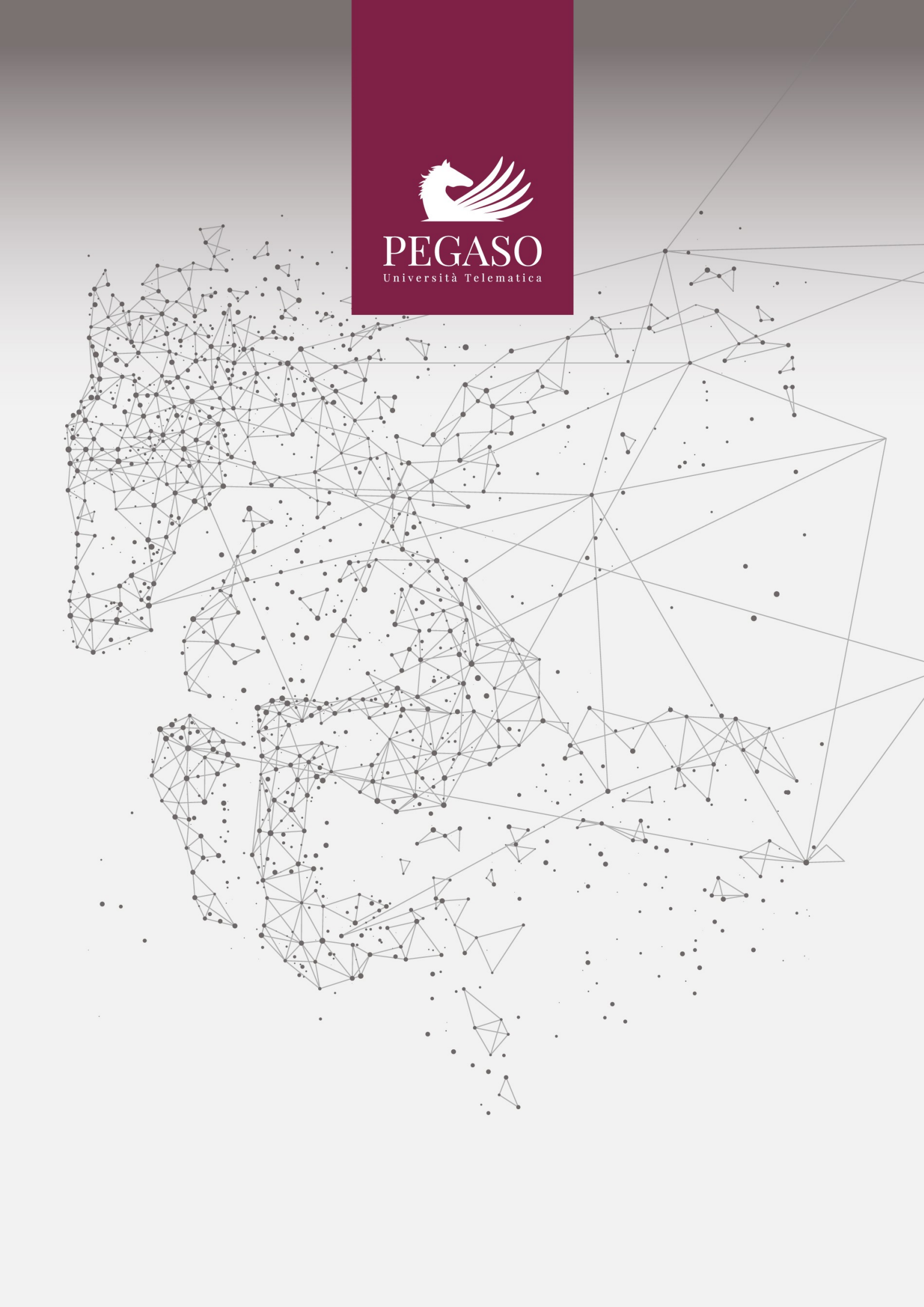

PEGASO
Università Telematica

Indice

1. Programmazione dinamica

Quando dobbiamo affrontare un problema possiamo approcciare diverse tecniche di risoluzione:

- Divide-et-impera
- Programmazione dinamica / memoization
- Tecnica greedy
- Ricerca locale
- Backtrack
- Algoritmi probabilistici
- Tecniche di soluzione per problemi intrattabili

Focalizziamo la nostra attenzione sulla **programmazione dinamica**.

La tecnica di programmazione dinamica, simile al metodo divide et impera, permette di risolvere problemi combinando le soluzioni dei sottoproblemi. Tuttavia, mentre il metodo divide et impera suddivide il problema in sottoproblemi indipendenti, la programmazione dinamica si applica quando i sottoproblemi hanno in comune dei sotto-sottoproblemi. In questo modo, un algoritmo divide et impera può svolgere più lavoro del necessario, risolvendo ripetutamente i sotto-sottoproblemi comuni.

Per evitare questo problema, la programmazione dinamica risolve ogni sottoproblema una sola volta e salva la sua soluzione in una tabella. Ciò permette di evitare il lavoro di ricalcolare la soluzione ogni volta che si presenta il sottoproblema. La programmazione dinamica si applica in particolare ai problemi di ottimizzazione, dove esistono molte soluzioni possibili, ciascuna con un valore associato. L'obiettivo è trovare la soluzione ottima, ovvero quella con il valore massimo o minimo.

Il processo di sviluppo di un algoritmo di programmazione dinamica può essere suddiviso in quattro fasi:

1. nella prima fase si cerca di caratterizzare la struttura di una soluzione ottima
2. nella seconda fase si definisce in modo ricorsivo il valore di una soluzione ottima
3. nella terza fase si calcola il valore di una soluzione ottima secondo uno schema bottom-up, ovvero partendo dal basso verso l'alto
4. nella quarta fase si costruisce la soluzione ottima dalle informazioni calcolate

Le prime tre fasi formano la base per risolvere un problema applicando la programmazione dinamica, mentre la quarta fase può essere omessa se si cerca soltanto il valore di una soluzione ottima.

Tuttavia, a volte durante la fase 4 si inseriscono informazioni aggiuntive per ottimizzare ulteriormente la soluzione.

Sintetizzando:

- Si divide un problema ricorsivamente in sottoproblemi
- Ogni sottoproblema viene risolto una volta sola
- La sua soluzione viene memorizzata in una tabella
- Nel caso un sottoproblema debba essere risolto **nuovamente**, si ricerca la sua soluzione dalla tabella (essendo già stato risolto, la soluzione è stata scritta in tabella).

ATTENZIONE: la tabella è facilmente indirizzabile – lookup in $O(1)$

L'approccio generale è quello per cui abbiamo a che fare con problemi di ottimizzazione per i quali cerchiamo la definizione della soluzione in maniera ricorsiva, in particolare il valore della soluzione in maniera ricorsiva (essendo un problema di ottimizzazione, il valore sarà ad es. un minimo, un massimo ecc.); a questo punto, se vi sono dei problemi "ripetuti" posso utilizzare la **programmazione dinamica** o una sua specializzazione che è la **memoization**. Ovviamente, se non vi sono problemi ripetuti posso utilizzare un classico approccio **divide et impera**.

Sia la programmazione dinamica che la memoization producono una tabella delle soluzioni.

Ricordiamo che il termine programmazione dinamica (Dynamic Programming) è stato coniato da Richard Bellman agli inizi degli anni '50, nell'ambito dell'ottimizzazione matematica: inizialmente, si riferiva al processo di risolvere un problema compiendo le migliori decisioni una dopo l'altra:

- "Dynamic" doveva dare un senso "temporale"
- "Programming" si riferiva all'idea di creare "programmazioni ottime", per esempio nel campo della logistica

2. Il problema del domino

Il problema del domino può essere formulato nella seguente maniera: si hanno delle tessere del domino di dimensione 2×1; si vuole scrivere un algoritmo efficiente che prenda in input un intero n e restituisca il numero di possibili disposizioni di n tessere in un rettangolo $2 \times n$.

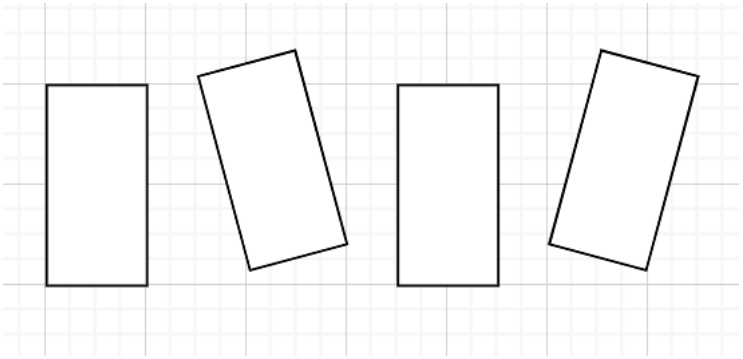


Figura 1: Tessere del domino

Se ad es. $n = 1$, cioè abbiamo 1 tessere, dovendo costruire un rettangolo $2\ x\ n\ =\ 2\ x\ 1$ abbiamo 1 sola disposizione possibile: quella in verticale.

Se ad es. $n = 2$, cioè abbiamo 2 tessere, dovendo costruire un rettangolo $2\ x\ n\ =\ 2\ x\ 2$ abbiamo 2 disposizioni possibili:

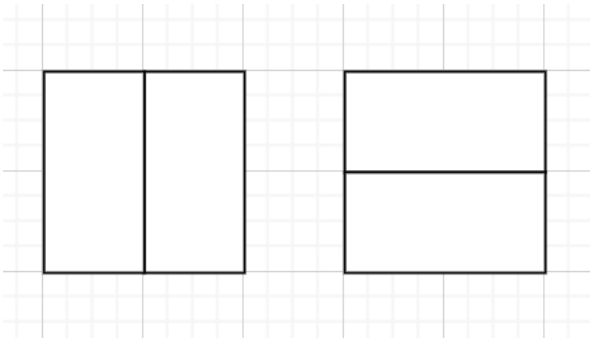


Figura 2: disposizioni

Possiamo cioè posizionare il primo tassello in verticale, dopodiché l'unica opzione possibile per il completamento è il posizionamento anche del secondo in verticale; oppure possiamo posizionare il primo tassello in orizzontale, dopodiché l'unica opzione possibile per il completamento è il posizionamento anche del secondo in orizzontale.

Se ad es. $n = 4$, cioè abbiamo 4 tessere, dovendo costruire un rettangolo $2\ x\ n\ =\ 2\ x\ 4$ abbiamo 5 disposizioni possibili:

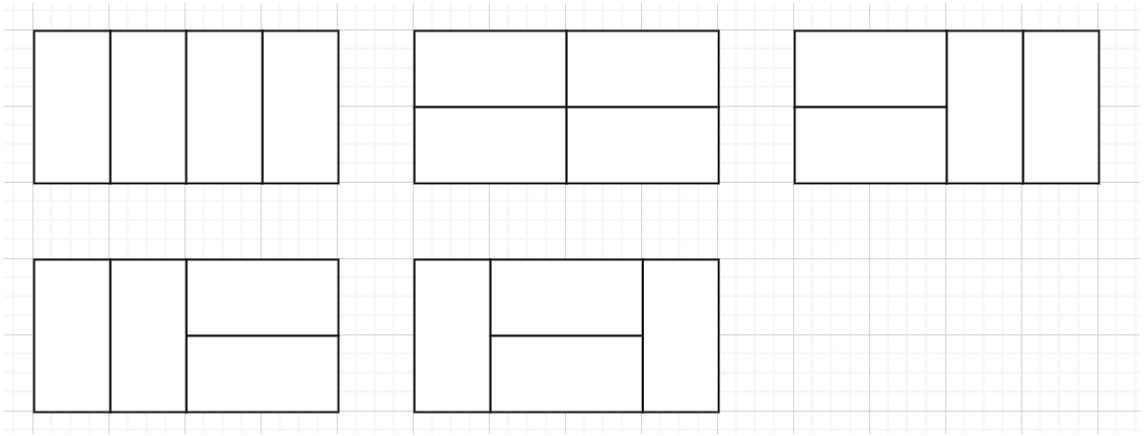


Figura 3: disposizioni

ATTENZIONE: se non avessi alcun tassello, "quante" disposizioni avrei a disposizione? La risposta potrebbe sembrare essere 0 ma in realtà anche in questo caso è 1, perché "non mettere alcun tassello del domino", equivale a tutti gli effetti ad una disposizione possibile.

Quindi abbiamo che:

- n=0 → 1 disposizione
- n=1 → 1 disposizione
- n=2 → 2 disposizioni
- n=4 → 5 disposizioni

L'obiettivo è creare una formula ricorsiva $DP[n]$ che mi permetta di trovare il numero di disposizioni dato un cero numero di tessere n ($DP\ =\ Dynamic\ Programming$).

Ragioniamo nella seguente maniera.

Se l'ultimo tassello che metto è verticale, mi resterà poi da affrontale il problema per le $n - 1$ tessere rimanenti. Quindi in sostanza potrei dire che $DP[n] = DP[n-1]$, cioè il numero di disposizioni per un certo numero n di tasselli, se scelgo come ultimo tassello quello verticale, sarà dato dal numero delle disposizioni dei tasselli rimanenti, cioè $DP[n-1]$.

Se l'ultimo tassello che metto è orizzontale, sicuramente dovrò metterne un altro orizzontale, e poi mi resterà poi da affrontale il problema per le $n - 2$ tessere rimanenti. Quindi in sostanza potrei dire che $DP[n] = DP[n-2]$, cioè il numero di disposizioni per un certo numero n di tasselli, se scelgo come ultimo tassello quello orizzontale, sarà dato dal numero delle disposizioni dei tasselli rimanenti, cioè $DP[n-2]$.

ATTENZIONE: potremmo avere la "tentazione" di scrivere $1 + DP[n-1]$ e $2 + DP[n-2]$, cioè aggiungere il valore costante relativo alla disposizione del tassello in verticale (nel primo caso) e dei 2 orizzontali (nel secondo caso); ma qui non sto contando il numero di tasselli, sto contando il numero di "disposizioni" e pertanto la disposizione che prevede che l'ultimo tassello sia verticale (ad es.) avrà quello come tassello finale "insieme" agli altri $n-1$ tasselli (configurabili in $DP[n-1]$ disposizioni) e quindi non abbiamo "una disposizione in più": è esattamente la stessa.

Quindi, riassumendo, potremmo dire che DP[n]= $DP[n-1] + DP[n-2]$ cioè devo "sommare" le due casistiche perché sono 2 scenari completamente diversi:

- o scelgo l'ultima verticale (con di seguito tutte le disposizioni che sono previste, cioè $DP[n-1]$)
- o scelgo l'ultima orizzontale (e dunque un'altra orizzontale con tutte le disposizioni che sono previste, cioè $DP[n-2]$)

Abbiamo dunque 2 "scenari" disgiunti che si vanno a sommare e determinano dunque:

$$DP[n] = DP[n-1] + DP[n-2]$$

Se proviamo a scrivere tutta la serie otteniamo i seguenti valori:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Questi rappresentano la serie di **Fibonacci**. Nello specifico: $DP[n]$ è pari all'$(n+1)-esimo$ numero della serie di **Fibonacci**.

Consideriamo dunque la seguente implementazione:

```
int domino_v1(int n)
if n ≤ 1 then
     return 1
else
     return domino1(n - 1) + domino1(n - 2)
```

Si può verificare che la complessità di questo algoritmo è pari a $\Theta(2^n)$

Se analizziamo i "sotto-problemi" generati, ci rendiamo facilmente conto che ve ne sono molti ripetuti: se ad es. volessimo risolvere il domino con $n = 4$ dovremo risolvere il domino con $n = 3$ e quello con $n = 2$; ma a loro volta, il domino con $n = 3$ va risolto con il domino con $n = 2$ e quello con $n = 1$ e per quello con $n = 2$ dovremo risolvere il domino con $n = 1$ e quello con $n = 0$ e così via…

Se rappresentassimo da un punto di vista grafico il tutto tramite un albero avremmo:

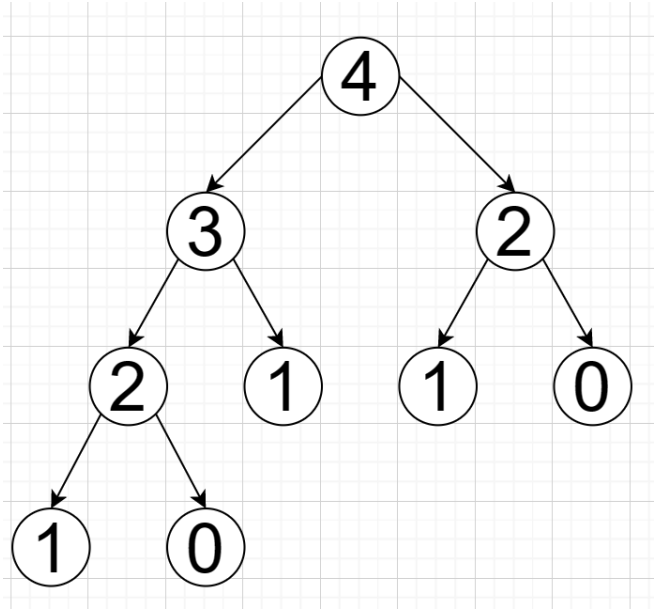


Figura 4: albero del domino con n=4

Vediamo chiaramente che l'albero presenta dei nodi "identici" in sottoalberi differenti, ad es. nel sotto-albero di sinistra abbiamo un 2 ed in quello di destra un 2 quindi entrambi vanno a risolvere lo stesso problema.

La soluzione può essere l'utilizzo di una tabella DP (Dynamic Programming): memorizziamo cioè il risultato ottenuto risolvendo un particolare problema in una tabella DP (vettore, matrice, dizionario); la tabella deve contenere un elemento per ogni sottoproblema che dobbiamo risolvere.

La strategia è di tipo "bottom-up": si parte dai problemi risolubili come basi base e si sale verso problemi via via più grandi fino a raggiungere il problema originale.

```
int domino_v2(int n)
DP = new int[0…n]
DP[0] = DP[1] = 1
for i = 2 to n
     DP[i] = DP[i – 1] + DP[i – 2]
return DP[n]
```

Quello che si ottiene è appunto la serie di Fibonacci:

- *DP[0] = 1*
- *DP[1] = 1*
- *DP[2] = DP[0] + DP[1] = 2*
- *DP[3] = DP[2] + DP[1] = 3*
- *DP[4] = DP[3] + DP[2] = 5*

- […]

Quindi la tabella (in questo caso un vettore) si costruisce partendo dal "basso" inserendo il valore 1 nelle prime 2 celle (0 ed 1) e poi inserendo nella cella successiva la somma delle 2 precedenti.

La tabella conterrà dunque le soluzioni per ogni dato valore di n.

Si può verificare che la complessità di questo algoritmo è pari a $\Theta(n)$ ed anche la complessità in termini di spazio è pari a $\Theta(n)$.

L'ulteriore miglioramento che possiamo fare è in termini di occupazione di spazio. L'idea alla base è la seguente: per quale motivo dobbiamo memorizzare tutti i risultati precedenti? Ci basta memorizzare gli ultimi 2 risultati.

```
int domino_v3(int n)
int DP0 = 1
int DP1 = 1
int DP2 = 1
for i = 2 to n
      DP0 = DP1
      DP1 = DP2
      DP2 = DP0 + DP1
return DP2
```

Come si evince dal codice precedente, sto utilizzando di fatto un numero costante di variabili per gestire il tutto: mi bastano cioè solo DP0, DP1 e DP2 per gestire il processo portando quindi l'occupazione di spazio a $\Theta(1)$.

Vediamo dal punto di vista pratico cosa accade:

n	0	1	2	3	4	5
DP0	-	-	1	1	2	3
DP1	1	1	1	2	3	5
DP2	1	1	2	3	5	8

Per $n = 0$ ed $n = 1$ inseriamo in DP1 e DP2 il valore 1.

Dopodichè a partire da $n = 2$, "spostiamo" i precedenti DP1 e DP2 in DP0 e DP1, e sommiamo.

Pertanto:

- *DP0[2] = DP1[1]*
- *DP1[2] = DP2[1]*
- *DP2[2] = DP0[2] + DP1[2]*

E così via per le colonne successive.

ATTENZIONE: qualora si faccia crescere il valore di n, in realtà la "somma" di 2 numeri di Fibonacci consecutivi ha un costo $O(n)$, pertanto le varie complessità che abbiamo calcolato in precedenza, per n grade, prevedono che si debba moltiplicare per n il valore ottenuto.

Funzione	Complessità temporale n piccolo	Complessità temporale n grande	Complessità spaziale n piccolo	Complessità spaziale n grande
domino_v1	$O(2^n)$	$O(2^n)$	$O(n)$	$O(n^2)$
domino_v2	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$
domino_v3	$O(n)$	$O(n^2)$	$O(1)$	$O(n)$

3. Fibonacci e sezione aurea

Ricordiamo che la serie di Fibonacci è una sequenza di numeri in cui ogni numero è la somma dei due numeri precedenti.

Esiste un legame tra la serie di Fibonacci e la **sezione aurea**: questa è denotata dalla lettera greca $phi\ (\varphi)$ ed il suo valore approssimativo è 1,6180339887.

Il legame tra la serie di Fibonacci e la sezione aurea è che la proporzione tra due numeri consecutivi della serie di Fibonacci si avvicina sempre di più al valore di φ, quando i numeri diventano più grandi. Ad esempio, se si divide 8 per 5, si ottiene 1,6, che è un'approssimazione di φ. Se si divide 13 per 8, si ottiene 1,625, che è ancora più vicino a phi. In sostanza, più ci si sposta nella serie di Fibonacci, più la proporzione tra i numeri consecutivi si avvicina al valore di φ.

Inoltre, il rapporto tra due numeri qualsiasi della serie di Fibonacci si avvicina alla sezione aurea quando i numeri diventano grandi: ad esempio, se si divide 89 per 55, si ottiene 1,6181818182, che è ancora una volta un'approssimazione molto vicina di φ.

Cos'è la sezione aurea?

Immaginiamo di avere una linea retta di lunghezza L. La si può dividere in due parti in modo tale che la parte più lunga sia di lunghezza a e la parte più corta sia di lunghezza b. La sezione aurea a/b è definita come il rapporto tra la parte più lunga e la parte più corta, e deve soddisfare la seguente relazione:

$$a/b = (a+b)/a$$

Questa relazione può essere riscritta come:

$$a^2 = ab + b^2$$

oppure come:

$$b/a = (\sqrt{5} - 1)/2$$

Quest'ultima equazione fornisce il valore esatto del rapporto della sezione aurea, che è rappresentato da φ ed è approssimativamente 1,6180339887.

In pratica, questo significa che se si divide una linea retta in due parti in modo tale che il rapporto tra la parte più lunga e la parte più corta sia uguale a φ, allora si ottiene una proporzione che è considerata armoniosa e gradevole alla vista. Questo concetto è stato ampiamente utilizzato nell'arte e nell'architettura per creare opere esteticamente piacevoli.

Come disegniamo una "spirale aurea", la cui immagine è abbasta "familiare"?

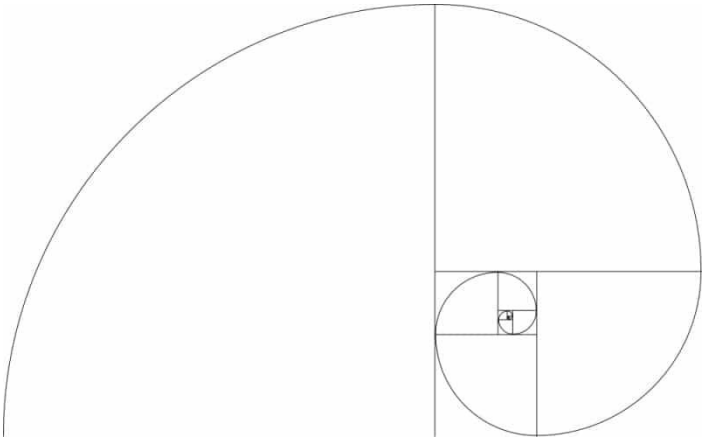
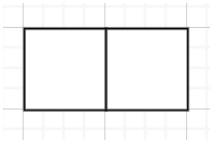


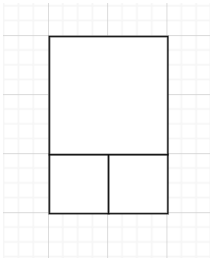
Figura 5: Spirale aurea

Immaginiamo di cominciare con un quadrato di lato 1.

Per costruire la spirale aurea, si disegna un altro quadrato adiacente al primo, dove la dimensione del nuovo quadrato è uguale alla dimensione della linea che forma uno dei lati del quadrato precedente, cioè 1.

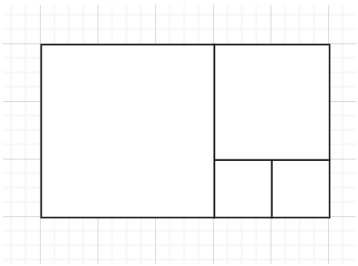


Quindi, si disegna un altro quadrato adiacente a questi due, dove la dimensione del nuovo quadrato è uguale alla somma delle dimensioni dei due quadrati precedenti, cioè 1 + 1 = 2. Tale quadrato si posizione "sopra" ai primi 2.

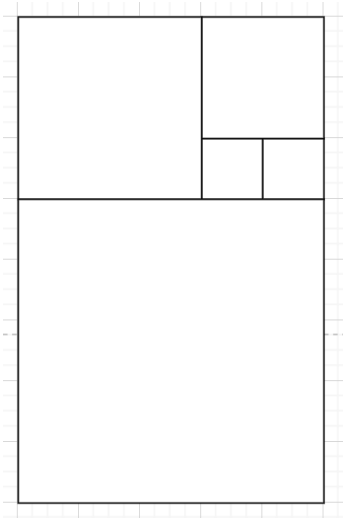


Si continua a disegnare quadrati adiacenti in modo simile, dove la dimensione di ogni nuovo quadrato è la somma delle dimensioni dei due quadrati precedenti.

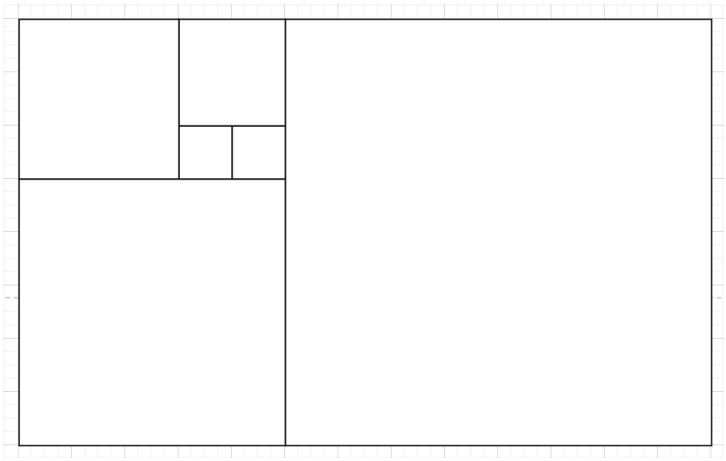
Il terzo quadrato ha una dimensione di 3 (2 + 1) e lo si posiziona sulla sinistra.



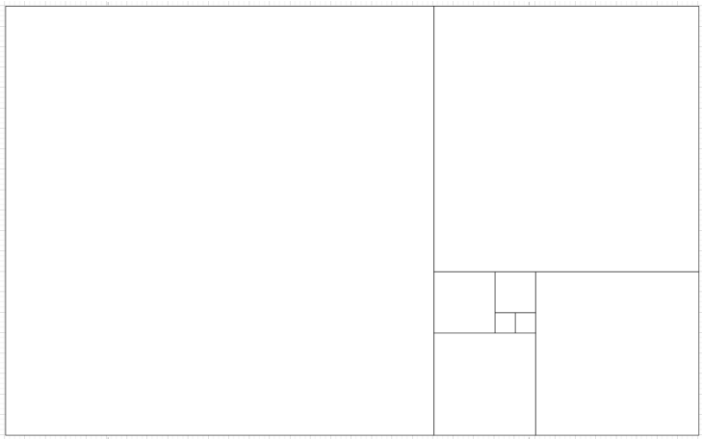
Il quarto quadrato ha una dimensione di 5 (3 + 2) e lo si posiziona in basso.



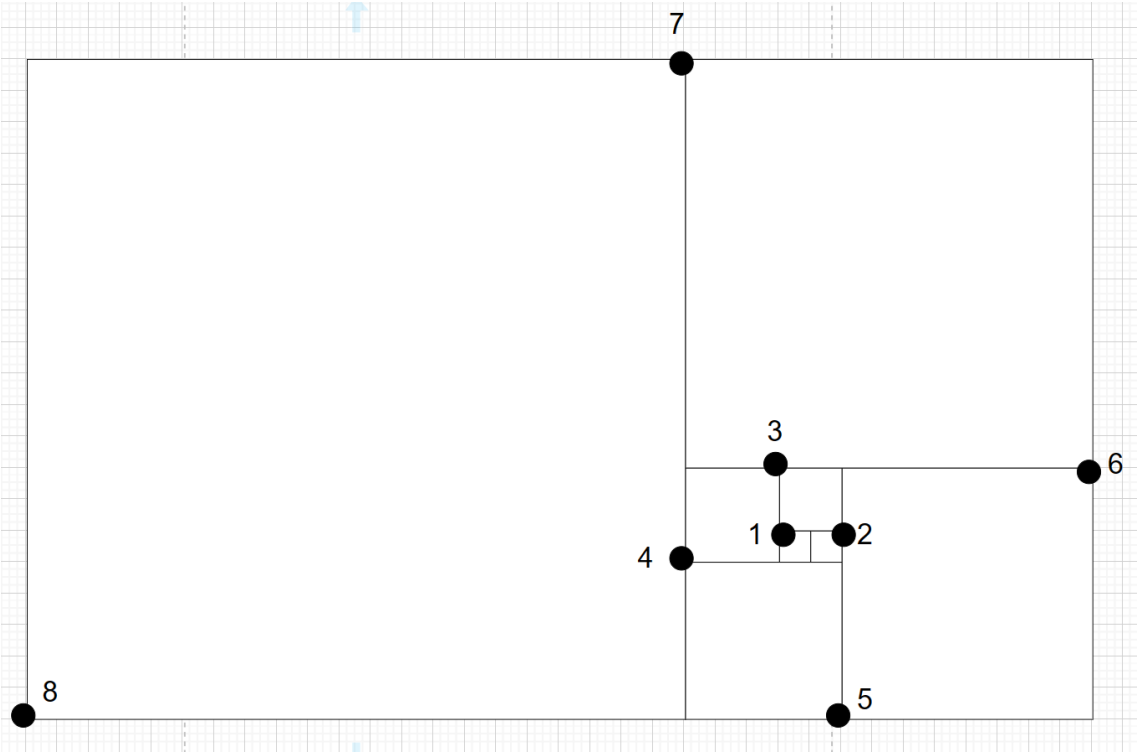
Il quinto quadrato ha una dimensione di 8 (5 + 3) e lo si posiziona a destra.



E così via; fermiamoci con la successione a questo schema:



Individuiamo i seguenti "punti":



Individuo i primi 2 vertici come in figura (1) e (2) e poi il (3) sarà l'opposto del (2) nel terzo quadrato; il (4) sarà l'opposto del (3) nel quarto quadrato; il (5) sarà l'opposto del (4) nel quinto quadrato ecc.

A questo punto unisco con degli archi di circonferenza; ad es. nel terzo quadrato faccio l'arco tra (2) e (3) avendo come centro il vertice tra il (2) ed il (3); nel quarto quadrato faccio l'arco tra (3) e (4) avendo come centro il vertice tra il (3) ed il (4) e così via…

Alla fine, otteniamo come risultato la "spirale aurea".

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione;
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione;
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley;
- Sedgewick: Algoritmi in C, Pearson, 2015;
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione.