Prima elementare (Compito 01/02/12)

A mia figlia (prima elementare) è stato chiesto di disegnare tutte le possibili sequenze composte da tre pallini rossi e due pallini gialli.

1. Scrivere un algoritmo che stampa tutte le possibili stringhe composte da n_r caratteri R e da n_g caratteri G, per un totale di $n_r + n_g$ caratteri.
2. Scrivere un algoritmo che conta tutte queste possibili stringhe – ovviamente senza generarle tutte e poi contandole.
3. Calcolare la complessità computazionale degli algoritmi proposti.

Somma di quadrati (Compito 26/1/16)

Ogni intero positivo n può essere scritto come somma di quadrati di interi; ad esempio, $3 = 1^2 + 1^2 + 1^2$, $7 = 2^2 + 1^2 + 1^2 + 1^2$, mentre $13 = 3^2 + 2^2$.

Ovviamente, esistono più modi per esprimere un numero come somma di quadrati; 13 può essere espresso anche come $2^2 + 2^2 + 2^2 + 1^2$.

Scrivere un algoritmo che, preso in input n, restituisce il *numero minimo di quadrati* la cui somma è pari a n.

Ad esempio, nel caso di 13, $3^2 + 2^2$ richiede 2 quadrati, mentre $2^2 + 2^2 + 2^2 + 1^2$ richiede 4 quadrati, quindi l'algoritmo deve restituire 2.

Discuterne correttezza e complessità.

Somma di quadrati (Compito 26/1/16)

Scrivere un algoritmo che, dato n, stampa **tutti** i modi possibili per esprimere n come somma di quadrati, a meno dell'ordine degli addendi, discutendo correttezza e complessità; ad esempio, con $n = 13$, stamperà:

$$1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2$$
$$2^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2$$
$$2^2 + 2^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2$$
$$2^2 + 2^2 + 2^2 + 1^2$$
$$3^2 + 1^2 + 1^2 + 1^2 + 1^2$$
$$3^2 + 2^2$$

Tessere del domino

Il gioco del domino contiene tessere di dimensione 2×1. Si considerino le disposizioni di n tessere all'interno di un rettangolo $2 \times n$.
Scrivere un algoritmo che stampi tutte le disposizioni possibili e discutere la sua correttezza e complessità.
I casi (a)-(e) della figura rappresentano i cinque modi possibili con cui è possibile riempire un rettangolo 2×4.

(a)

1	2	3	4
1	2	3	4

(b)

1	2	3	3
1	2	4	4

(c)

1	1	3	4
2	2	3	4

(d)

1	2	2	4
1	3	3	4

(e)

1	1	3	3
2	2	4	4

Tessere del domino

Suggerimento: un modo semplice di stampare le disposizioni è di numerare le tessere, e stampare due righe di n interi l'una. Se queste due righe sono memorizzate in una matrice $S[1 \ldots 2, 1 \ldots n]$, siete autorizzati ad utilizzare il comando **print** S per stamparle. Se seguite questo suggerimento, è importante ricordare che siamo interessati alle disposizioni, e non ai numeri, quindi il caso (f) (permutazione di (a)) non deve essere contato/stampato. In ogni caso, metodi alternativi di stampa sono comunque accettati.

(a)

1	2	3	4
1	2	3	4

(b)

1	2	3	3
1	2	4	4

(c)

1	1	3	4
2	2	3	4

(d)

1	2	2	4
1	3	3	4

(e)

1	1	3	3
2	2	4	4

(f)

4	3	2	1
4	3	2	1

Spoiler alert!

Prima elementare – Backtracking

printRGRec(**char**[ ] S, **int** i, **int** n_r, **int** n_g)

```
if n_r == 0 and n_g == 0 then
    print S[0 ... i − 1]
else
    if n_r > 0 then
        S[i] = 'R'
        printRGRec(S, i + 1, n_r − 1, n_g)
    if n_g > 0 then
        S[i] = 'G'
        printRGRec(S, i + 1, n_r, n_g − 1)
```

printRG(**int** n_r, **int** n_g)

char[ ] S = **new char**$[0 \ldots n_r + n_g - 1]$
printRGRec$(S, 0, n_r, n_g)$

Prima elementare – Conteggio inefficiente

int countRGRec(**int** n_r, **int** n_g)

```
% Perché or qui?
if n_r == 0 or n_g == 0 then
    return 1
else
    return countRGRec(n_r − 1, n_g) + countRGRec(n_r, n_g − 1)
```

int countRG(**int** n_r, **int** n_g)

```
return countRGRec(n_r, n_g)
```

Complessità: $O(2^{n_r+n_g})$ se consideriamo che a ogni passo ho al più due chiamate ricorsive. Il numero di casi terminali è limitato superiormente da: $\binom{n_r+n_g}{n_r} = \frac{(n_r+n_g)!}{n_r!n_g!}$.

Prima elementare – Conteggio inefficiente

int countRGRec(**int** n_r, **int** n_g)

```
% Se resta un solo colore, esiste una sola continuazione
if n_r == 0 or n_g == 0 then
    return 1
else
    return countRGRec(n_r − 1, n_g) + countRGRec(n_r, n_g − 1)
```

int countRG(**int** n_r, **int** n_g)

return countRGRec(n_r, n_g)

Complessità: $O(2^{n_r+n_g})$ se consideriamo che a ogni passo ho al più due chiamate ricorsive. Il numero di casi terminali è limitato superiormente da: $\binom{n_r+n_g}{n_r} = \frac{(n_r+n_g)!}{n_r!n_g!}$.

Prima elementare – Conteggio programmazione dinamica

int countRG(**int** n_r, **int** n_g)

int[ ][ ] DP = **new int**$[0 \ldots n_r][0 \ldots n_g]$
for $i = 0$ **to** n_r **do**
  $DP[i][0] = 1$ % Non ci sono più G, si termina con R
for $j = 0$ **to** n_g **do**
  $DP[0][j] = 1$ % Non ci sono più R, si termina con G
for $i = 1$ **to** n_r **do**
  for $j = 1$ **to** n_g **do**
    $DP[i][j] = DP[i-1][j] + DP[i][j-1]$ % R oppure G
return $DP[n_r][n_g]$

$DP[i][j]$ conta le stringhe con i caratteri R e j caratteri G.
Complessità: $O(n_r \cdot n_g)$

Prima elementare – Calcolo combinatorio

int countRG(**int** n_r, **int** n_g)

return fact$(n_r + n_g)/(\text{fact}(n_r) \cdot \text{fact}(n_g))$

- Complessità (criterio uniforme): $O(n_r + n_g)$
- Complessità (criterio logaritmico): $O((n_r + n_g)^2 \log^2(n_r + n_g))$

Nota avanzata sulla complessità

Peter B. Borwein. *On the complexity of calculating factorials.* In Journal of Algorithms, 6(3):376-380, 1985. [Link]
Criterio logaritmico, fast multiplication, algoritmo efficiente:

$$O((n_r + n_g)(\log(n_r + n_g) \log\log(n_r + n_g))^2)$$

Somma di quadrati

Sia $DP[i]$ il minimo numero di quadrati necessari per esprimere i.

- Caso base: $DP[0] = 0$ (non consideriamo il caso $0^2 = 0$, in quanto è possibile ottenere 0 non sommando nulla)
- Ricorsione: per identificare l'ultimo quadrato,
  - consideriamo tutti i valori t^2 tali per cui $1 \leq t^2 \leq i$;
  - valutiamo i problemi $DP[i - t^2]$, ottenuti sottraendo t^2 da i;
  - scegliamo fra questi quello che richiede il minor numero di quadrati;
  - aggiungiamo 1 per il quadrato considerato.

$$DP[i] = \begin{cases} 0 & i = 0 \\ \min_{1 \leq t \leq \lfloor \sqrt{i} \rfloor} \{DP[i - t^2] + 1\} & i \geq 1 \end{cases}$$

Somma di quadrati – Programmazione dinamica

int squareSum(**int** n)

int[ ] DP = **new int**$[0 \dots n]$
$DP[0] = 0$
for $i = 1$ **to** n **do**
    $DP[i] = +\infty$
    for $t = 1$ **to** $\lfloor \sqrt{i} \rfloor$ **do**
        $DP[i] = \min(DP[i], DP[i - t^2] + 1)$
return $DP[n]$

Complessità?

Somma di quadrati – Programmazione dinamica

int squareSum(**int** n)

int[ ] DP = **new int**$[0 \dots n]$
$DP[0] = 0$
for $i = 1$ **to** n **do**
    $DP[i] = +\infty$
    for $t = 1$ **to** $\lfloor \sqrt{i} \rfloor$ **do**
        $DP[i] = \min(DP[i], DP[i - t^2] + 1)$
return $DP[n]$

Complessità? $\Theta(n\sqrt{n})$

Somma di quadrati – Note avanzate

- Lagrange ha dimostrato (Teorema dei quattro quadrati) che ogni intero positivo può essere espresso come somma di (al più) quattro quadrati perfetti

  `https://en.wikipedia.org/wiki/Lagrange%27s_four-square_theorem`

- Michael O. Rabin and Jeffrey Shallit hanno proposto algoritmi probabilistici di complessità polilogaritmica per identificare i quattro quadrati:
  - M. Rabin, J. Shallit. "Randomized Algorithms in Number Theory". Communications on Pure and Applied Mathematics. 39(S1):S239–S256 (1986)
- Un documento più recente è il seguente:
  - P. Pollack, E. Treviño. Finding the four squares in Lagrange's Theorem. [PDF]

Somma di quadrati – Backtracking

Per risolvere la seconda parte del problema, dobbiamo invece utilizzare la tecnica di backtracking. Una versione semplice per risolvere il problema potrebbe scegliere tutti i possibili valori di t tali per cui $1 \leq t^2 \leq n$, e provare ricorsivamente tutte le possibilità:

psRec(**int** n, **int**[ ] S, **int** i)

```
if n == 0 then
    print S[0 ... i − 1]
else
    for t = 1 to ⌊√n⌋ do
        S[i] = t
        psRec(n − t², S, i + 1)
```

printSquare(**int** n)

```
int[ ] S = new int[0 ... n − 1]
psRec(n, S, 0)
```

Nota: S contiene le basi dei quadrati e viene stampata "as is"

Somma di quadrati – Backtracking

- Il vettore S delle scelte ha dimensione n (somma di tutti 1)
- La complessità è superpolinomiale
- L'algoritmo appena visto, tuttavia, stampa anche tutte le permutazioni dei quadrati
- È accettabile se si interpreta l'ordine degli addendi come significativo; una soluzione migliore evita le permutazioni
- Imponiamo un ordine canonico: valori non decrescenti
- Si aggiunge un parametro *limit* per imporre l'ordine
- I valori scelti devono essere almeno pari a *limit*
- Se si è scelto il valore t, tutte le scelte successive devono essere maggiori o uguali a t

Somma di quadrati – Backtracking

psRec(**int** n, **int**[ ] S, **int** i, **int** *limit*)

```
if n == 0 then
    print S[0 ... i − 1]
else
    for t = limit to ⌊√n⌋ do
        S[i] = t
        psRec(n − t², S, i + 1, t)
```

printSquare(**int** n)

```
int[ ] S = new int[0 ... n − 1]
psRec(n, S, 0, 1)
```

k-cammini

visit(GRAPH G, **int** k, NODE s)

boolean[ ] $visited$ = **new boolean**$[0 \ldots G.n-1]$ = { **false** }
NODE[ ] $path$ = **new** NODE$[0 \ldots k]$
visitRec$(G, k, s, 0, path, visited)$

visitRec(GRAPH G, **int** k, NODE u, **int** i, NODE[] $path$, **boolean**[ ] $visited$)

```
path[i] = u
if i == k then
    print path[0...k]                    % Cammino con k+1 nodi, k archi
else
    visited[u] = true                    % Nella ricorsione, u non è più visitabile
    foreach v ∈ G.adj(u) do
        if not visited[v] then
            visitRec(G, k, v, i+1, path, visited)
    visited[u] = false                   % Nella ricorsione, u torna visitabile
```

k-cammini

- Struttura di una visita in profondità, con queste differenze:
  - Contatore per limitare profondità
  - Meccanismo di backtracking: $visited[u] =$ **false** quando si è completata la visita del nodo u
- Caso pessimo: grafo completo
- Numero di cammini semplici, con sorgente fissata, di lunghezza esattamente $k < n$:

$$(n-1) \cdot (n-2) \cdot \ldots (n-k) = \frac{(n-1)!}{(n-k-1)!} = O(n^k)$$

- Costo per cammino stampato: $\Theta(k)$
- Complessità: $O(n^k \cdot k)$

Tessere del domino

printDomino(**int**[ ][ ] S, **int** i)

if $i == 0$ **then**
    print S % Stampa la matrice $2 \times n$
if $i \geq 1$ **then** % Ultima colonna riempita verticalmente
    $S[0][i-1] = S[1][i-1] = i$
    printDomino$(S, i-1)$
if $i \geq 2$ **then** % Ultime due colonne riempite orizzontalmente
    $S[0][i-1] = S[0][i-2] = i$
    $S[1][i-1] = S[1][i-2] = i-1$
    printDomino$(S, i-2)$

printDomino(**int** n)

int[ ][ ] $S =$ **new int**$[0 \ldots 1][0 \ldots n-1]$
printDominoRec(S, n)

Tessere del domino

Complessità:

- Stima grossolana: $O(n \cdot 2^n)$, perché abbiamo al più due chiamate ricorsive per ogni chiamata e il costo di stampa è $O(n)$
- Tuttavia, dalla lezione su programmazione dinamica (Hateville) sappiamo che il numero delle disposizioni con n tessere è l'$(n+1)$-esimo numero di Fibonacci.
- Quindi il costo è $\Theta(n \cdot Fib(n+1)) = \Theta(n\phi^n)$, dove $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$.