Zaino

Si consideri il problema dello zaino. Scrivere un algoritmo che prenda in input un vettore w di pesi di dimensione n e una capacità C, e stampi tutti i sottoinsiemi di indici corrispondente a sottoinsiemi di oggetti il cui peso sia inferiore o uguale a C.

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 caratteri R e da m caratteri G, per un totale di $n + m$ caratteri.
2. Scrivere un algoritmo che conta tutte queste possibile 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 ad 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, discutendo correttezza e complessità; ad esempio, con $n = 13$, stamperà:

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

k-cammini (Compito 25/7/18)

Scrivere un algoritmo che prende in input un grafo $G = (V, E)$, un nodo $s \in V$ e un intero k e stampa tutti i cammini contenuti nel grafo tali che:

- partano da s,
- abbiano lunghezza esattamente k,
- siano semplici (senza nodi ripetuti).

Discutere informalmente la correttezza della soluzione proposta e calcolare la complessità computazionale.

Spoiler alert!

Zaino

```
knapsackRec(int[] w, boolean [] S, int i, int c)
if i == 0 then
    processSolution(S, n)
else
    S[i] = false
    knapsackRec(w, S, i − 1, c)
    if w[i] ≤ c then
        S[i] = true
        knapsackRec(w, S, i − 1, c − w[i])
```

```
knapsack(int[] w, int n, int C)
int[] S = new int[1 . . . n]
knapsackRec(w, S, n, C)
```

Prima elementare – Backtracking

printRGRec(**char**[ ] S, **int** i, **int** n, **int** m)

```
if n == 0 and m == 0 then
    print S
else
    if n > 0 then
        V[i] = "R"
        printRGRec(S, i + 1, n − 1, m)
    if m > 0 then
        V[i] = "G"
        printRGRec(S, i + 1, n, m − 1)
```

printRG(**int** n, **int** m)

```
int[ ] S = new char[1 . . . n + m]
printRGRec(S, 1, n, m)
```

Prima elementare – Conteggio inefficiente

```
int countRGRec(int n, int m)
if n == 0 or m == 0 then
    return 1
else
    return countRGRec(n − 1, m) + countRGRec(n, m − 1)
```

```
int countRG(int n, int m)
return countRGRec(n, m)
```

Complessità: $O(2^{n+m})$

Prima elementare – Conteggio programmazione dinamica

int countRG(**int** n, **int** m)

```
int[][] DP = new int[0...n][0...m]
for i = 0 to n do
    DP[i][0] = 1
for j = 0 to m do
    DP[0][j] = 1
for i = 1 to n do
    for j = 1 to m do
        DP[i][j] = DP[i − 1][j] + DP[i][j − 1]
return DP[n][m]
```

Complessità: $O(nm)$

Somma di quadrati

Sia $DP[n]$ il minimo numero di quadrati necessari per esprimere n. Ovviamente il caso base è $DP[0] = 0$; per quanto riguarda la parte ricorsiva, consideriamo tutti i valori t tali per cui $1 \leq t^2 \leq n$; consideriamo quindi tutti i sottoproblemi $DP[n - t^2]$ e scegliamo fra questi quello che richiede il minor numero di quadrati, aggiungendo 1 per il quadrato considerato:

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

Somma di quadrati – Programmazione dinamica

int squareSum(**int** n)

int[ ] DP = **new int**$[0 \ldots n]$
$DP[0] = 0$
$DP[1] = 1$
for $i = 2$ **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 $DP[n]$

Somma di quadrati – Complessità

- Complessità: $\Theta(n\sqrt{n})$.
- 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[1 ... 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[1 ... n]
psRec(n, S, 1)
```

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 per il compito, ma una soluzione migliore evita le permutazioni imponendo un ordine ai valori che vengono sommati.
- Si aggiunge un parametro *limit* per imporre l'ordine
- I valori scelti sono limiti superiormente da *limit*
- Se si è scelto il valore t, tutte le scelte successive devono essere minori o uguali a quel valore

Somma di quadrati – Backtracking

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

if $n == 0$ **then**
    print $S[1 \dots i-1]$
else
    for $t = 1$ **to** $\min(\lfloor\sqrt{n}\rfloor, limit)$ **do**
        $s[i] \leftarrow t$
        psRec$(n - t^2, S, i+1, t)$

printSquare(**int** n)

int[ ] $S =$ **new int**$[1 \dots n]$
psRec$(n, S, 1, \lfloor\sqrt{n}\rfloor)$

k-cammini

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

boolean[ ] $visited$ = **new boolean**$[1 \ldots G.n]$ = { **false** } % Init to **false**
int[ ] $path$ = **new int**$[1 \ldots k+1]$
visitRec$(G, k, s, 1, path, visited)$

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

```
path[i] = u
if i == k + 1 then
    print path
else
    visited[u] = true
    foreach v ∈ G.adj(u) do
        if not visited[v] then
            visitRec(G, k, i + 1, v, path, visited)
    visited[u] = false
```

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:
  $(n-1) \cdot (n-2) \cdot \ldots (n-k) = \frac{(n-1)!}{(n-k-1)!} = O(n^k)$
- Stampa cammino: $\Theta(k)$
- Complessità: $O(n^k \cdot k)$