```
// Struttura di un algoritmo ricorsivo generico

void recursive (... , level) {

  // E -- sequenza di istruzioni che vengono eseguite sempre
  // Da usare solo in casi rari (es. Ruzzle)
  doAlways();

  // A
  if (condizione di terminazione) {
    doSomething;
    return;
  }

  // Potrebbe essere anche un while ()
  for () {

    // B
    generaNuovaSoluzioneParziale;


    if (filtro) { // C
      recursive (..., level + 1);
    }

    // D
    backtracking;
  }

}
```

Algoritmi Ricorsivi:

1) permutazioni di una parola

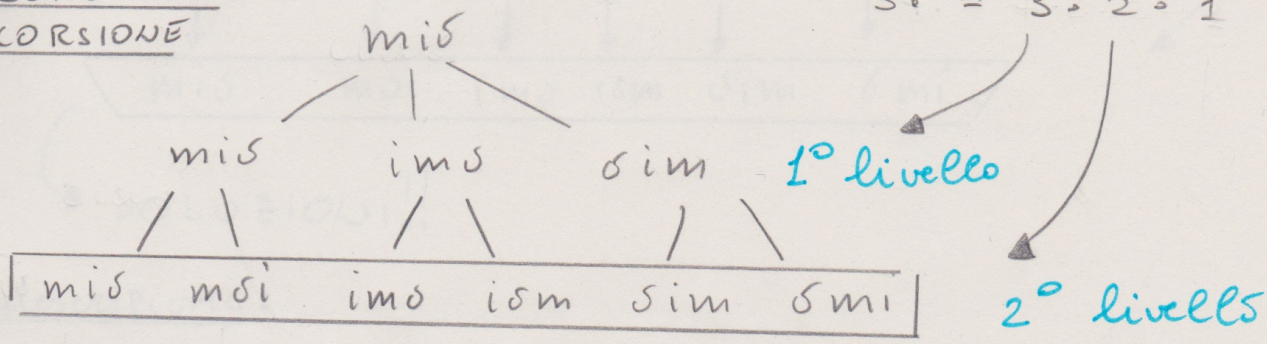
es. mis : imo, smi, moi, ...

1ª soluzione:

mis → 3 lettere = 3!

3! = 3 · 2 · 1

ALBERO DELLA RICORSIONE



descrizione:

Al 1° livello scambio la prima lettera con le altre. Al 2° livello scambio la seconda lettera con le altre ...

- data una parola di lunghezza n
- dimensione dello spazio di ricerca: n!
- n° di livelli per la ricorsione: n

alg. ricorsivo:

A • condizione di terminazione: livello/passo == n

B • generazione di una nuova soluzione:
scambio lettera i-esimo livello con le successive

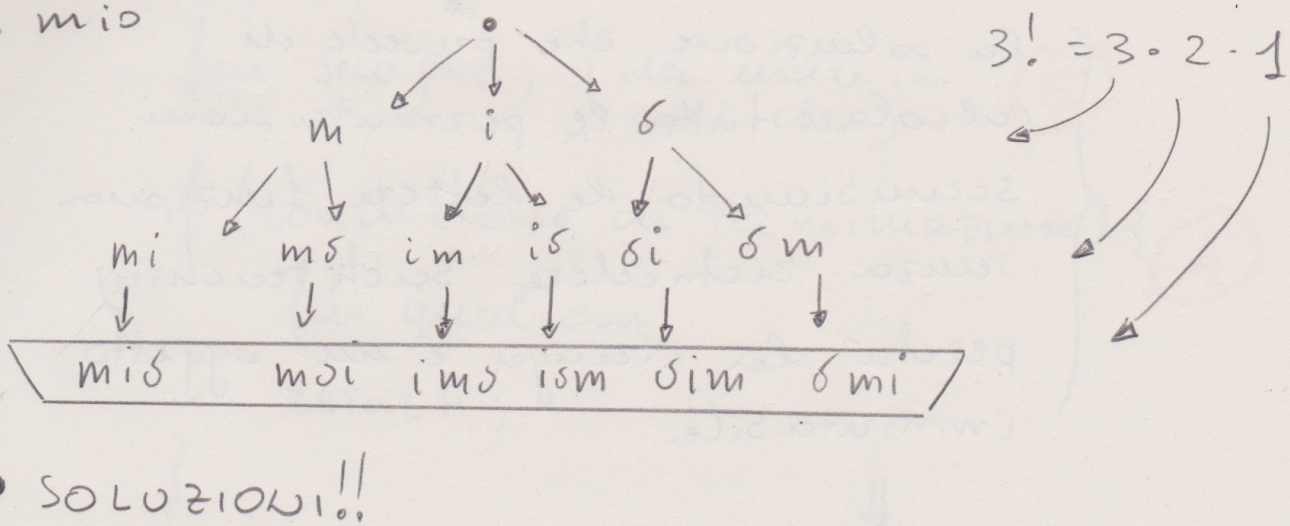
C • filtro sulla chiamata ricorsiva: NO

D • back tracking

2ª soluzione

costruisco la soluzione in modo incrementale!

es: mio



descrizione:

al 1° livello aggiungo la prima lettera (3 possibilità). Al 2° livello aggiungo la seconda lettera (2 possibilità). Al 3° livello aggiungo la 3a ed ultima lettera.

- data una parola di lunghezza: n
- numero di soluzioni: n!
- profondità dell'albero della ricorsione: n

alg ricorsivo:

A. condizione di terminazione: [passo == n]

B. generazione di una nuova soluzione:
[al passo i-esimo, aggiunta i-esima lettera]

C. filtro sulla chiamata ricorsiva: [NO]

D. bachtracking: [NO]

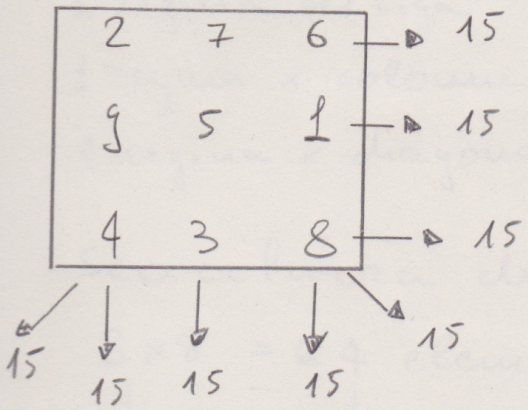
NOTA: ricorsive (string permutazione, livello)

la soluzione che prevede di calcolare tutte le permutazioni scambiando le lettere funziona senza richiedere backtracking perché la stringa è un oggetto immutabile.

⇓

nel caso si usi una lista di Character o un array di char, bisogna fare backtracking.

2) quadrato magico



somma delle righe ==
somma delle colonne ==
somma diag princ/second

Il problema è molto simile al #1. La matrice può essere vista come un array:

⇒ [2,7,6, 9,5,1, 4,3,8]

descrizione:

- data una matrice n x n (es 3 x 3)
- numero delle soluzioni: $(n^2)!$
- livelli della ricorsione: (n^2)

alg ricorsivo:

A. condizione di terminazione: passo == (n^2)
   → qui controllo se è una soluzione!!

B. generazione di una nuova soluzione:
   al passo i-esimo
   1) scambio i-esima elemento ARRAY con tutti i successivi
   2) aggiungo i-esimo numero

C. filtro sulla chiamata ricorsiva: NO

D. backtracking: SI → al passo i-esimo tolgo i-esimo numero.

3) 8 regine

1 regina per riga

1 regina x colonna

1 regina x diagonale

Scacchiera di

8×8 = 64 elementi

↓ ↓

matrice array

(come nell'esercizio n°2 si sfrutta la corrispondenza tra matrice e array)

2 strutture dati

- array di int di 8 elementi ogni elemento è un numero da 0 a 7 che indica la posizione della regina nella colonna.
- array di int da 64
  0 → casella vuota
  1 → regina.

descrizione

- data una matrice 8×8 (n×n)
- numero delle soluzioni 8^8
  - → utilizzando il vincolo di 1 regina per riga
  - → non verrà esplorato l'intero spazio delle soluzioni perché si utilizza il trucco del filtro sulla chiamata ricorsiva ☺

NON DEVO ESPLORARE 64! SOLUZIONI!!

- livelli della ricorsione: n

alg ricorsivo:

- condizione di terminazione:

  (A) passo == n ; se arrivo fin qui questa è una soluzione!

- generazione di una nuova soluzione:

  (B) al passo i-esimo posiziono una regina sulla riga i-esima.

- filtro sulla chiamata ricorsiva

  (C) SI: vado avanti solo se le i regine posizionate fino al passo i-esimo rappresentano una soluzione parziale VALIDA!!

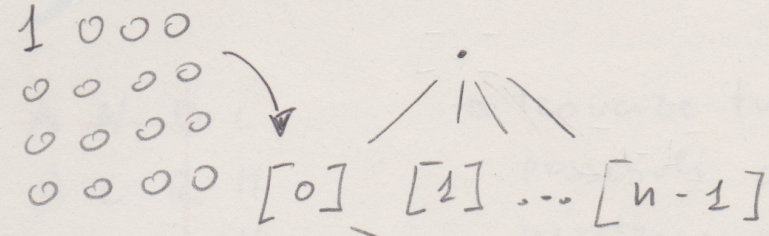
  -> utilizzo una funzione check_regine (matrix)

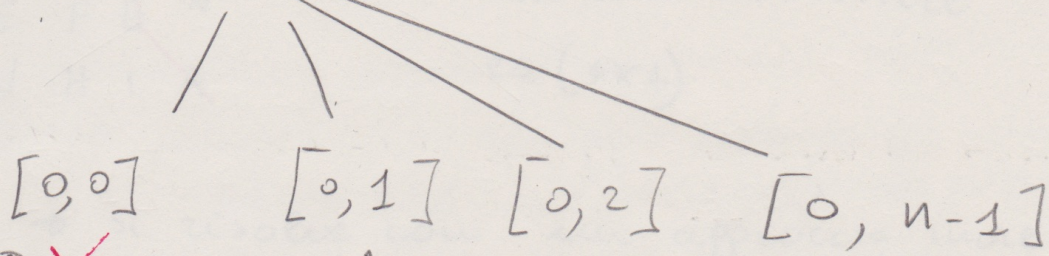
- backtracking: SI!

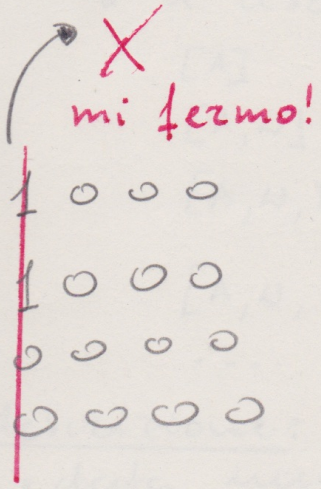
  (D) al passo i-esimo elimino la i-esima regina inserita.

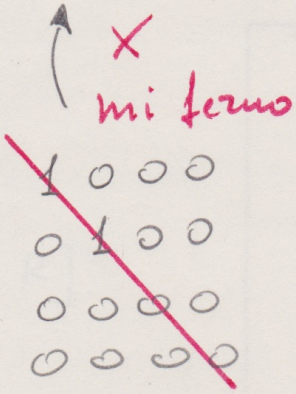
esempio (4x4) → 4 regine

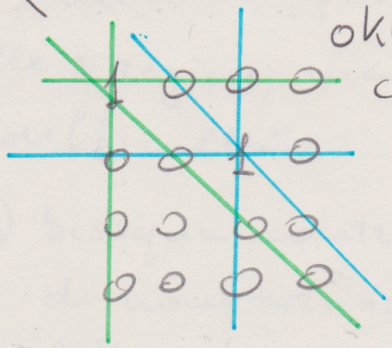
ESEMPIO ALBERO DELLA RICORSIONE





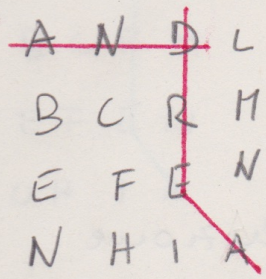






PS: manca le diagonale secondaria.

4) Zuzzle



→ trovare tutte le parole possibili data la matrice di lettere

es (4×4)

→ si risolve con un approccio incrementale

[A]
[A, N]
[A, N, D]
[A, N, D, R]
....

descrizione:

- data una matrice 4×4 (n×n)
- soluzioni possibili:

  TANTE!

è simile al problema delle 8 regine, ma aggiunge 2 difficoltà:

1) bisogna controllare di non tornare sulle lettere già scelte
2) le soluzioni possono avere lunghezze differenti

- livelli di ricorsione: teoricamente $16 = (n^2)$

[in realtà mi fermo quando la sequenza di lettere considerata non esiste in nessuna parola nel dizionario.]

↝ (E)

alg ricorsivo:

- condizione di terminazione
  - passo == (n^2)

A

- generazione di una nuova soluzione

B

al passo i-esimo aggiungo l'i-esima lettera ad un array (lista) di char che:
- si deve trovare tra i vicini rispetto alla posizione corrente
- non deve essere tra quelle per cui sono già passato.

- filtro sulla funzione ricorsiva: SI

C

la sequenza generata deve esistere nel dizionario!!

- backtrack: elimino i-esima lettera aggiunta!!

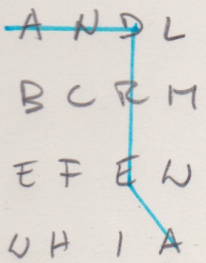
D

- operazioni da svolgere sempre:

E

se passo > 1 (più di due lettere)
controllo se la sequenza esiste come PAROLA nel dizionario.

ESEMPIO
ALBERO
DELLA RICORSIONE
SO ROZZE!!



albero parziale!!

[A]

[AN]

[AND]

[ANDL] [ANDR]

[ANDLM] [ANDLR] [ANDRE]

[ANDREA]

[ANDREAI]

non esiste!

non esiste

✓ ESISTE!
AGGIUNGO alle SOLUZ.

✗ non esiste