Ricorsione

Giovanni Falco

November 15, 2024

Contents

1 Ricorsione

la ricorsione consiste nel svolgere un problema chiamando più volte una funzione su se stessa. Detta cosi potrebbe sembrare un qualcosa di astratto, in fatti per comprenderlo meglio vedremo degli esempi di codice e dei disegni che chiariscono il tutto. Pensiamo ad un semplice problema, lo svolgimento di un fattoriale, questo si fa moltiplicando il numero stesso, per se stesso -1, finchè non si arriva "al caso base", n=1.

```
factorial(int n) {
if(n == 1 || n == 0)
return 1 ;
else
return n* factorial(n-1) ;

}
```

Potrebbe sembrare molto semplice ma dietro ci sono tante altre cose che non vediamo. Quindi scendiamo nel particolare.
PRIMO STEP
supponiamo di avere n = 4.

```
 factorial(4) {
if(n == 1 || n == 0)
return 1 ;
else
return 4* factorial(4-1) ;

}
```

quindi viene saltato il primo if e verrà eseguita nuovamente la funzione factorial.
SECONDO STEP:
n verrà incrementato di 1

```
 factorial(3) {
if(n == 1 || n == 0)
return 1 ;
else
return 3* factorial(3-1) ;

}
```

TERZO STEP:
n verrà incrementato di 1

```
 factorial(2) {
if(n == 1 || n == 0)
return 1 ;
else
```

```
return 2* factorial(2-1) ;

}
```

QUARTO STEP:
n verrà incrementato di 1

```
 factorial(1) {
if(n == 1 || n == 0)
return 1 ;
else
return 1* factorial(1-1) ;

}
```

Infine verrà n di 0 e finisce.
Ora cosa accade come ottengo 4! ?
Ora dobbiamo immaginare una pila, quindi possiamo avere una pila dove accade ciò : CHIAMATE :
factorial(4);
factorial(3);
factorial(2);
factorial(1);
factorial(0);
ora cosi avviene il ritorno :
1
1*1
1*1*2
1*1*2*3
1*1*2*3*4
cosi si ottiene il valore desiderato cioè 4!=24.
per essere più precisi ecco un immagine che descrive meglio il procedimento:
NOTA: è importante il caso base in maniera tale che puoi "finire" le "n" chiamate, se non avessi aggiunto l'if sul caso n=1 che ritornasse 1 avrei avuto un loop infinito.

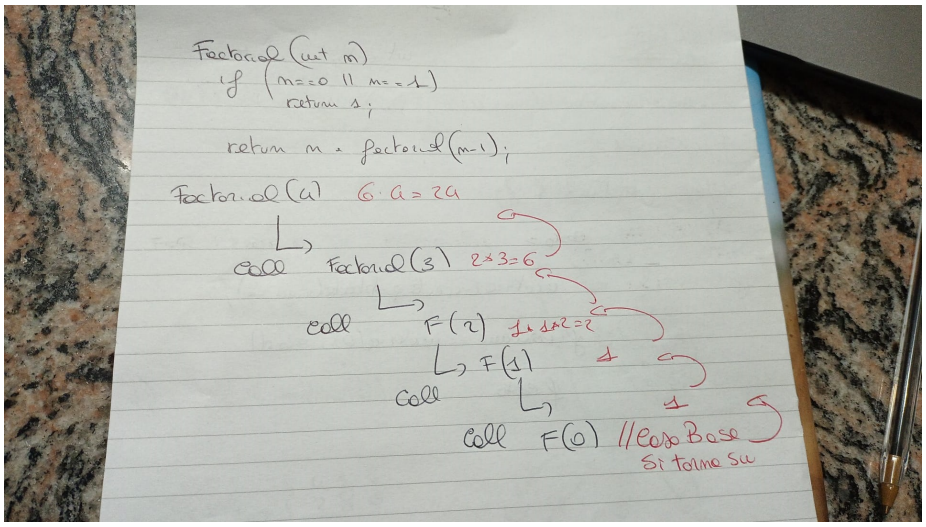

Figure 1: Stack delle chiamate

2 Divide et Impera

è un approccio per risolvere i problemi. Lo si utilizza in quanto spesso questo approccio riduce la complessità temporale dello svolgimento di un algoritmo. Significa che un approccio, spesso accompagnato con la ricorsione che migliora il tempo di esecuzione.
E' basato su 3 approcci:

- Divide: dividi il problema in sottoproblemi
- impera: risolvi il problema, o in maniera ricorsiva o iterativa.
- combina: unisci le soluzioni.

Ci concentreremo sulla maniera iterativa e faremo capire questo approccio. Vediamo ora il mergeSort, per capire come viene suddiviso il problema e basta, non ordineremo il vettore ma dimostreremo che si potranno fare qualsiasi operazione, però bisogna essere in grado di sapere cosa si sta facendo.

2.1 Merge

```
int/void ... MergeSort( int vettore, int inizio, int fine)
if inizio> fine
return vettore[inizio]

mid = inizio + (fine-inizio)/2
elementoSinistra = MergeSort(vettore, inizio, mid);
elementoDestra = MergeSort(vettore,mid+1, fine);

// QUALSIASI ALTRA OPERAZIONE.
```

Ora bisogna sapere cosa si fa e come vengono sviluppate queste chiamate. Quindi smetterò di chiamare le funzioni ricorsive e potrò lavorare col singolo elemento, e potrò fare qualsiasi macchingegno.
Come illustrato nella foto. quindi uno degli approcci è dividere, per poi lavorare singolarmente sugli elementi. E' l'unico approccio ? no, ce ne sono altri, ma questo è uno dei tanti approcci.

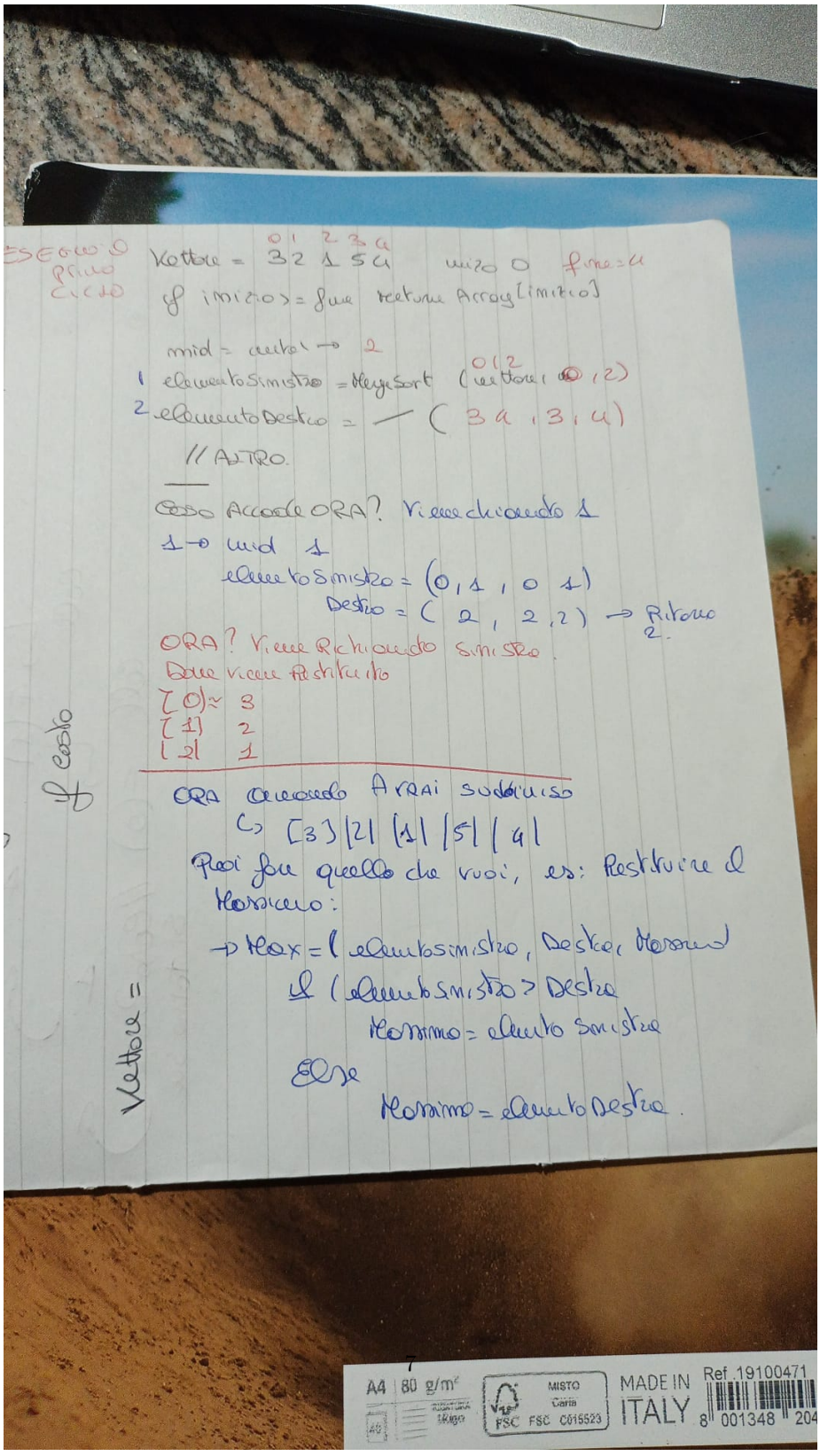

Figure 2: COSA ACCADE?

3 BackTracking

è un paradigma di programmazione, che fa uso della ricorsione ai fini di risolvere un problema. A prima battuta potrebbe sembrare ostico ma è molto più semplice del metodo precedente perchè ha un paradigma di base che verrà sviluppato in base al caso.
PARADIGMA DI BASE :

```
// CASO BASE :
if(...)
processoSolution() // fa qualcosa

altrimenti :

for( i from 0 to n)
vettore.push_back(element) // AGGIUNGI ELEMENTO
backtracking(...)
vettore.pop_back() // RIMUOVI ELEMENTO.
```

Cosi non è di grande aiuto, quindi andiamo ad analizzare due casi in cui viene utilizzato per chiarire meglio le idee.

3.1 Permutazioni

Le conoscete?
le permutazioni non sono altro che tutte le possibili combinazini degli "n" numeri.
ESEMPIO: dato 1,2,3.

- 1,2,3
- 1,3,2
- 2,1,3
- 2,3,1
- 3,1,2
- 3,2,1

Quante ne ho ?
dato un insieme di n elementi ho n! insiemi. Infatti abbiamo ottenuto : 3*2*1 = 6 elementi. Come funziona quindi ? devi iniziare da un elemento vuoto e iniziare a fare delle decisioni. In questo caso la decisione iniziale è capire da dove si inizia
1, 2 o 3. quindi :

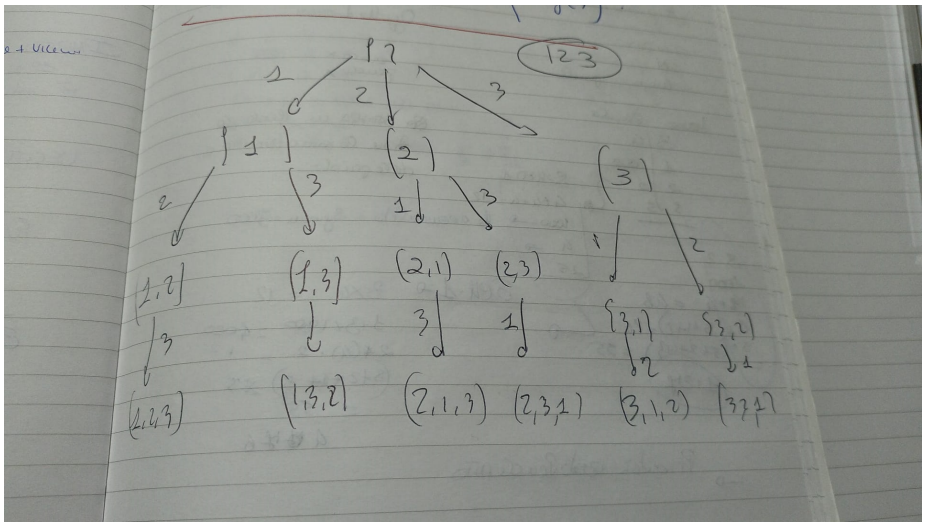

Figure 3: ALBERO DELLE DECISIONI

```
for i from 0 to n .
contenitore.push_back(vettore[i])
Backtracking...
contenitore.pop_back()

che accade ?
for i =0 ;
contenitore aggiungi 1,
chiama ricorsivamente la funzione.

ora i = 1
aggiunge 2,
e chiama ricorsivamente la funzione

aggiunge 3
e finisce e ritorniamo il primo insieme.
torniamo indietro
fino ad ottenere ancora l'inseme 1.
viene rimosso due e viene richiamata di nuovo la funzione per poi ottenere
1 3 2
.
ORA FINALEMENTE SI RITORNA INDIETRO CON 2 INSIEMI : 1,2,3 E 1,3,2.

ORA SI RIPETE IL TUTTO INIZIANDO DA 2.
```

ECCO il codice completo :

```
    #include <iostream>
#include <vector>
using namespace std;

// Funzione di utilità per stampare una permutazione
void printPermutation(const vector<int>& permutation) {
    for (int num : permutation) {
        cout << num << " ";
    }
    cout << endl;
}

// Funzione di backtracking per generare tutte le permutazioni
void permute(vector<int>& nums, vector<int>& currentPermutation, vector<bool>& used, vector<
    // Se la permutazione è completa (lunghezza uguale a nums), aggiungila al risultato
    if (currentPermutation.size() == nums.size()) {
        result.push_back(currentPermutation);
        return;
    }

    // Prova ogni elemento non utilizzato
    for (int i = 0; i < nums.size(); ++i) {
        // Se l'elemento è già stato utilizzato, salta
        if (used[i]) continue;

        // Aggiungi l'elemento corrente alla permutazione
        currentPermutation.push_back(nums[i]);
        used[i] = true; // Marca l'elemento come utilizzato

        // Ricorsivamente genera le permutazioni per il prossimo elemento
        permute(nums, currentPermutation, used, result);

        // Annulla l'aggiunta (backtrack)
        currentPermutation.pop_back();
        used[i] = false;
    }
}

int main() {
    // Definire l'array di numeri
    vector<int> nums = {1, 2, 3};
    vector<vector<int>> result;
    vector<int> currentPermutation; // Permutazione corrente
    vector<bool> used(nums.size(), false); // Vettore per tenere traccia degli elementi usat
    // Genera tutte le permutazioni
    permute(nums, currentPermutation, used, result);
```

```
    // Stampa tutte le permutazioni generate
    cout << "Permutazioni: " << endl;
    for (const auto& perm : result) {
        printPermutation(perm);
    }

    return 0;
}
```

Nota viene usato il vettore di booleani per tenere traccia degli elementi presi.

3.2 Enumerazione

L'enumerazione differisce di poco dalla permutazione, consiste nel prendere dei sottinsieme degli n elementi.
Dato un vettore : 1,2,3 otteniamo :

1

2

3

1,2

1,3

2,3

Complessità: $2^n = 2^3 = 8$.
Qui non prendo tutte le volte l'elemento "i", ma devo decidere se prende un elemento o meno.
Ecco il codice.

```
    // Funzione di backtracking per generare i sottinsiemi
void enumerateSubsets(vector<int>& set, vector<int>& currentSubset, int index) {
    // Stampa il sottinsieme corrente
    if()
    QUALCOSA


    // Esplora tutti gli altri sottinsiemi includendo o meno ogni elemento
    for (int i = index; i < set.size(); i++) {
        currentSubset.push_back(set[i]); // Aggiungi l'elemento corrente
```

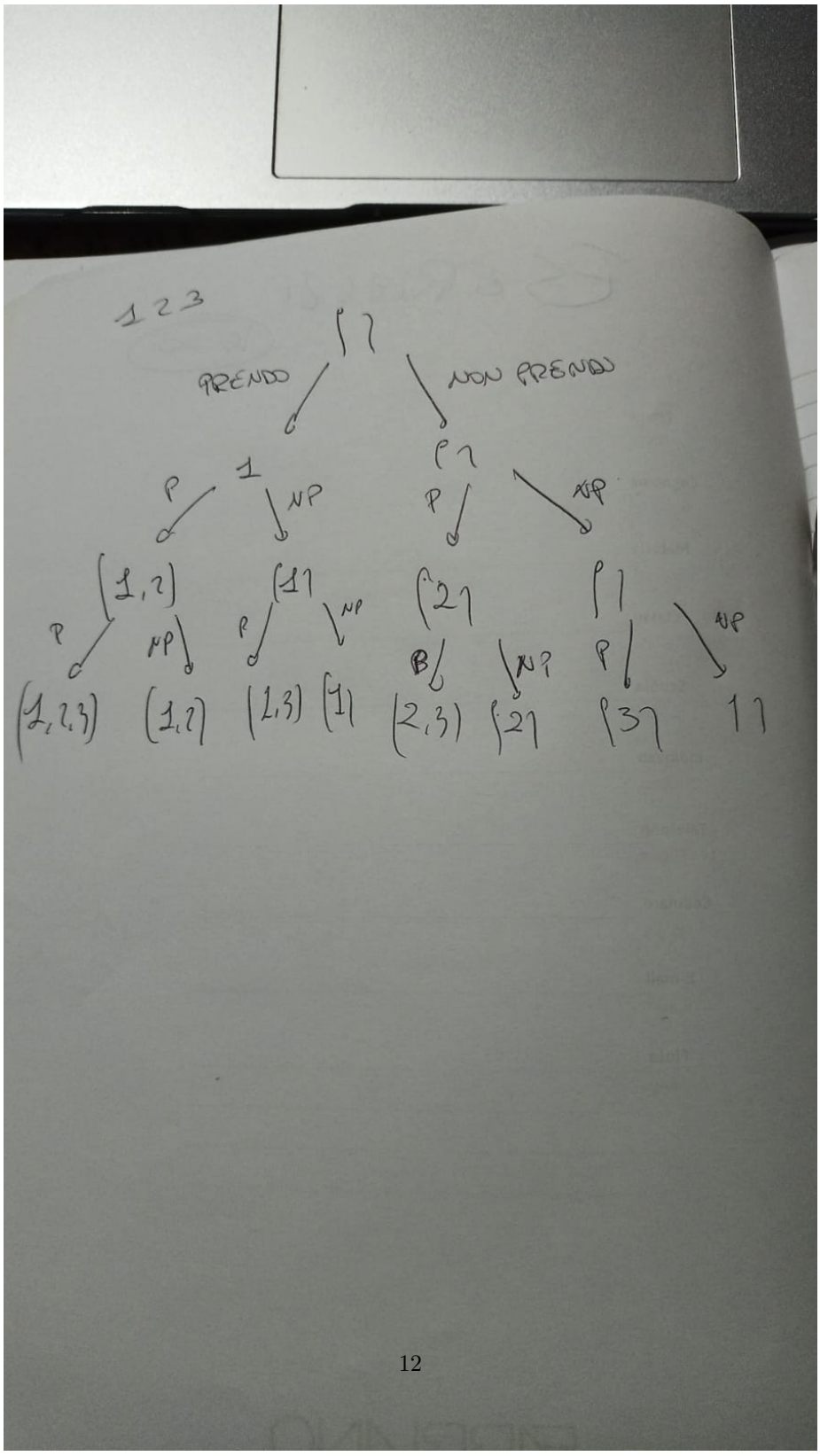

Figure 4: ALBERO DELLE DECISIONI

```
        enumerateSubsets(set, currentSubset, i + 1); // Ritorna per il prossimo elemento
        currentSubset.pop_back(); // Rimuovi l'elemento corrente (backtrack)
    }
    NOTA GLI PASSI INDICE COSA CHE DI LA NON FACEVI.
}
```