Corso di laurea in Informatica
Introduzione agli Algoritmi

Algoritmi ricorsivi

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

FUNZIONI RICORSIVE

Una funzione matematica si definisce **ricorsiva** quando la sua definizione fa riferimento a se stessa.
Ecco un esempio di definizione ricorsiva per la funzione fattoriale:

$$0! = 1$$
$$n! = n \cdot (n-1)! \quad \text{se } n > 0$$

- La prima riga della definizione identifica il **caso base**: specifica che il fattoriale di 0 è uguale a 1.
- La seconda riga descrive il **meccanismo ricorsivo**: definisce come calcolare il valore della funzione per un numero intero n utilizzando il valore della funzione per $n-1$.

Nel campo dell'informatica, esiste un concetto analogo: quello degli **algoritmi ricorsivi**. Un algoritmo si dice ricorsivo quando soddisfa le seguenti caratteristiche:

- Viene definito in termini di se stesso.
- La soluzione del problema complessivo viene costruita risolvendo (ricorsivamente) uno o più sottoproblemi di dimensione minore e combinando successivamente le soluzioni di tali sottoproblemi per ottenere la soluzione finale.
- La sequenza dei sottoproblemi, sempre più piccoli, deve convergere a un **caso base** (o **condizione di terminazione**) che interrompe la ricorsione.

Di seguito è riportato un algoritmo ricorsivo per il calcolo del fattoriale:

```
def fattorialeR(n):
    if n==0:
        return 1
    return n * fattorialeR (n-1)
```

Equazioni di ricorrenza:

```
def fattorialeR(n):                    T(n) =
    if n==0:                               Θ(1)
        return 1                           Θ(1)
    return n * fattorialeR (n-1)           Θ(1) + T(n − 1)
```

Indicando con $T(n)$ il tempo di esecuzione dell'algoritmo su un input di dimensione n, otteniamo la seguente definizione ricorsiva per il tempo di calcolo:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0, \\ T(n-1) + \Theta(1) & \text{altrimenti.} \end{cases}$$

L'espressione sopra descritta prende il nome di equazione di ricorrenza. Le **equazioni di ricorrenza** sono strumenti matematici utilizzati per descrivere il comportamento temporale (e talvolta anche spaziale) di algoritmi ricorsivi. Queste equazioni esprimono il tempo di esecuzione di un algoritmo in funzione del numero di sottoproblemi risolti ricorsivamente e dei costi associati a ciascuna chiamata.

Nel seguito, vedremo come risolvere queste equazioni di ricorrenza per determinare esplicitamente la complessità dell'algoritmo.

- É fondamentale assicurarsi che le funzioni ricorsive abbiano almeno un caso base, altrimenti la loro esecuzione genererebbe una catena illimitata di chiamate ricorsive che non terminerebbe mai (provocando ben presto la terminazione forzata dell'elaborazione a causa dell'esaurimento delle risorse di calcolo, per ragioni che vedremo fra breve).
- Bisogna quindi essere assolutamente certi che il caso base non possa essere mai "saltato" durante l'esecuzione.
- Ogni funzione, sia essa ricorsiva o no, richiede per la sua esecuzione una certa quantità di memoria, per:
  - caricare in memoria il codice della funzione;
  - passare i parametri alla funzione;
  - memorizzare i valori delle variabili locali della funzione.
- Quindi le funzioni ricorsive in generale hanno maggiori esigenze, in termini di memoria, delle funzioni non ricorsive (dette anche funzioni iterative).

La ricorsione e l'iterazione sono due approcci per risolvere problemi ripetitivi:

- **Ricorsione**: utilizza chiamate successive alla funzione stessa e sfrutta lo stack delle chiamate.
- **Iterazione**: utilizza strutture di controllo come cicli *for* o *while*.

Va detto che ogni algoritmo ricorsivo può essere riscritto in forma iterativa. In quali situazioni è dunque consigliabile utilizzare un algoritmo ricorsivo anziché uno iterativo?
Queste considerazioni possono essere d'aiuto nella decisione:

- è conveniente utilizzare la ricorsione quando essa permette di formulare la soluzione del problema in un modo che è chiaro ed aderente alla natura del problema stesso, mentre la soluzione iterativa è molto più complicata o addirittura non evidente;
- non è conveniente utilizzare la ricorsione quando esiste una soluzione iterativa altrettanto semplice e chiara;
- non è conveniente utilizzare la ricorsione quando l'efficienza è un requisito primario.

I Vantaggi e gli Svantaggi della Ricorsione possono essere riassunti come segue:

Vantaggi:

- Codice più semplice e leggibile per problemi naturalmente ricorsivi.
- Riduzione della complessità logica in certi casi.

Svantaggi:

- Maggior utilizzo di memoria a causa delle chiamate in attesa.
- Rischio di overflow se il caso base non è raggiunto.

Il seguente codice iterativo per il calcolo del fattoriale indica che, per questo problema, non c'è alcun vantaggio ad usare la versione ricorsiva:

```
def fattorialeI(n):
    f=1
    for i in range(1,n+1):
        f *= i
    return f
```

1. **Progettare una procedura ricorsiva per calcolare la somma di una lista di n interi.**
   L'idea di base è la seguente:

   - la somma degli elementi della lista vuota è 0
   - negli altri casi, la somma è data dal primo elemento della lista sommato al risultato della somma degli elementi della lista privata del primo elemento.

   Un'implementazione che utilizza il costrutto *slicing* per realizzare questa idea è riportata nella figura seguente:

```
def Sommalista(A):                          T(n) =
    if len(A)==[]:                              Θ(1)
        return 0                                Θ(1)
    return A[0] + SommaLista(A[1:])         Θ(n) + T(n − 1)
```

   a formula ricorsiva che descrive il calcolo della complessità di questa implementazione è

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(n) & \text{altrimenti} \end{cases}$$

   risolvendo questa relazione otterremo che $T(n) = \Theta(n^2)$.

Una versione alternativa più efficiente, che evita l'uso dello slicing, è riportata nella figura seguente:

```
def Sommalista(A, i = 0):                    T(n) =
    if len(A)== i:                                Θ(1)
        return 0                                  Θ(1)
    return A[i] + SommaLista(A, i + 1)            Θ(1) + T(n − 1)
```

In questo caso, la formula ricorsiva per la complessità è

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(1) & \text{altrimenti} \end{cases}$$

risolvendo questa relazione si otterrà $T(n) = \Theta(n)$.
Nota bene: è sempre meglio evitare slicing su liste di grandi dimensioni in operazioni ricorsive o cicliche, poiché il costo $O(b\text{-}a)$ di $A[a : b]$ può accumularsi e diventare significativo. Per operazioni più efficienti, è preferibile iterare direttamente sulla lista con un range di indici.

2. **Progettare una procedura ricorsiva che calcola il valore x^n, dove x è un numero reale e n è un intero non negativo.**
Si propone una procedura ricorsiva che, per $n > 0$, utilizza la relazione

$$x^n = x * x^{(n-1)}$$

Il codice corrispondente è illustrato nella figura seguente:

```
def power(x, n):
    # Caso base: qualsiasi numero elevato a 0 è 1
    if n == 0:
        return 1
    return x * power(x, n - 1)
```

La complessità di questa procedura è descritta dalla seguente relazione di ricorrenza:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(1) & \text{altrimenti} \end{cases}$$

risolvendo questa relazione si otterrà

$$T(n) = \Theta(n).$$

É possibile migliorare la procedura osservando che, per $n > 0$, possiamo distinguere due casi:

- **Caso pari:** Se n è pari, possiamo sfruttare la proprietà: $x^n = x^{\frac{n}{2}} \cdot x^{\frac{n}{2}}$ In questo caso il problema viene ridotto dimezzando il valore di n, con conseguente riduzione logaritmica della complessità.
- **Caso dispari:** Se n è dispari, utilizziamo la relazione: $x^n = x \cdot x^{n-1}$, che permette di ricondursi al caso pari.

Il codice ottimizzato è mostrato nella figura seguente:

```
def power(x, n):
    # Caso base: qualsiasi numero elevato a 0 è 1
    if n == 0:
        return 1
    a = power(x, n // 2)
    if n % 2 == 0:
        # Se n è pari: x^n = a*a
        return a * a
    #  n è dispari
    return x * a * a
```

```python
def power(x, n):
    # Caso base: qualsiasi numero elevato a 0 è 1
    if n == 0:
        return 1
    a = power(x, n // 2)
    if n % 2 == 0:
        # Se n è pari: x^n = a*a
        return a * a
    #  n è dispari
    return x * a * a
```

In questa versione, a ogni chiamata ricorsiva il valore di n viene dimezzato. La complessità è descritta dalla seguente relazione di ricorrenza:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(\lfloor \frac{n}{2} \rfloor) + \Theta(1) & \text{altrimenti} \end{cases}$$

Risolvendo questa relazione si otterrà

$$T(n) = \Theta(\log n).$$

3. Una lista è palindroma se è composta da al più un elemento o i suoi estremi sono uguali la sottolista privata degli estremi è ancora palindroma. Ecco un esempio di lista palindroma: $[1, 5, 4, 8, 4, 5, 1]$.
Progettare una procedura ricorsiva che, data una lista di n interi, verifica se la lista è palindroma.
Applicando direttamente la definizione otteniamo il codice nella figura che segue:

```
def palindroma(A, inizio, fine):                    T(n) =
    if inizio >= fine:                                Θ(1)
        return True                                   Θ(1)
    if A[inizio] != A[fine]:                          Θ(1)
        return False                                  Θ(1)
    return palindroma(A, inizio + 1, fine - 1)        Θ(1) + T(n − 2)
```

la formula ricorsiva per la complessità è

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T(n-2) + \Theta(1) & \text{altrimenti} \end{cases}$$

risolvendo questa relazione si otterrà $T(n) = \Theta(n)$.

4 **progettare una procedura ricorsiva che genera tuttte le permutazioni di una stringa di n caratteri distinti.**

L'algoritmo funziona scegliendo un carattere alla volta, rimuovendolo dalla stringa e ricorsivamente generando le permutazioni dei caratteri rimanenti. Infine, combina il carattere scelto con ogni permutazione ottenuta.

```
def permutazioni(s):
    '''Restituisce una lista con tutte le permutazioni
    di una stringa data'''
    if len(s) == 1:
        # Se la lunghezza della stringa è 1, restituire
        #la lista contenente la stringa stessa
        return [s]
    # Lista per memorizzare tutte le permutazioni
    risultato = []
    # Cicla su ogni carattere della stringa
    for i in range(len(s)):
        # Carattere corrente che viene separato dalla stringa
        c = s[i]
        # Crea il resto della stringa senza il carattere
            corrente
        resto = s[:i] + s[i+1:]
        # Chiamata ricorsiva per ottenere le permutazioni del
            resto della stringa
        PR = permutazioni(resto)
        for p in PR:
            # Aggiungi il carattere corrente al risultato della
                permutazione
            risultato.append(c + p)
    # Restituisce tutte le permutazioni trovate
    return risultato
```

```
>>> permutazioni('abc')
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
```

```python
def permutazioni(s):
    if len(s) == 1:
        return [s]
    risultato = []
    for i in range(len(s)):
        c = s[i]
        resto = s[:i] + s[i+1:]
        PR = permutazioni(resto)
        for p in PR:
            risultato.append(c + p)
    return risultato
```

Complessità:

- Se la stringa è lunga 1 il costo è $\Theta(1)$
- Il ciclo più esterno viene eseguito esattamente n volte.
- Il calcolo della sottostringa resto ha un costo di $\Theta(n)$, poiché viene creata una copia della stringa originale di lunghezza n, privandola di un carattere.
- Il costo per calcolare la lista PR=permutazioni(resto) è $T(n-1)$.
- Il ciclo più interno viene eseguito $(n-1)!$ volte, poiché questa è la cardinalità della lista PR, ovvero il numero di permutazioni di una stringa di lunghezza $n-1$.
- Ogni iterazione del ciclo più interno ha un costo di $\Theta(n)$, in quanto viene creata una nuova stringa di lunghezza n.

L'equazione di ricorrenza per la generazione delle permutazioni di una stringa di lunghezza n è:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ n \cdot T(n-1) + \Theta(n \cdot n!) & \text{altrimenti} \end{cases}$$

Questa relazione risolta darà $\Theta(n \cdot n!)$.

5. La definizione dei numeri di Fibonacci è la seguente:

   - $F(0) = 0$
   - $F(1) = 1$
   - $F(i) = F(i-1) + F(i-2)$ se $i > 1$

   A partire da questa formula, otteniamo la seguente sequenza numerica: $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots\ldots$
   Esiste una formula esplicita per il calcolo di questi numeri (formula di Binet), che però introduce errori numerici dovuti agli arrotondamenti causati dai calcoli in virgola mobile:

$$F(n) = \frac{\Phi^n}{\sqrt{5}} - \frac{(1-\Phi)^n}{\sqrt{5}} \text{ dove } \Phi = \frac{1+\sqrt{5}}{2}$$

   É naturale impostare il calcolo di questi numeri mediante la funzione ricorsiva illustrata nell'immagine seguente:

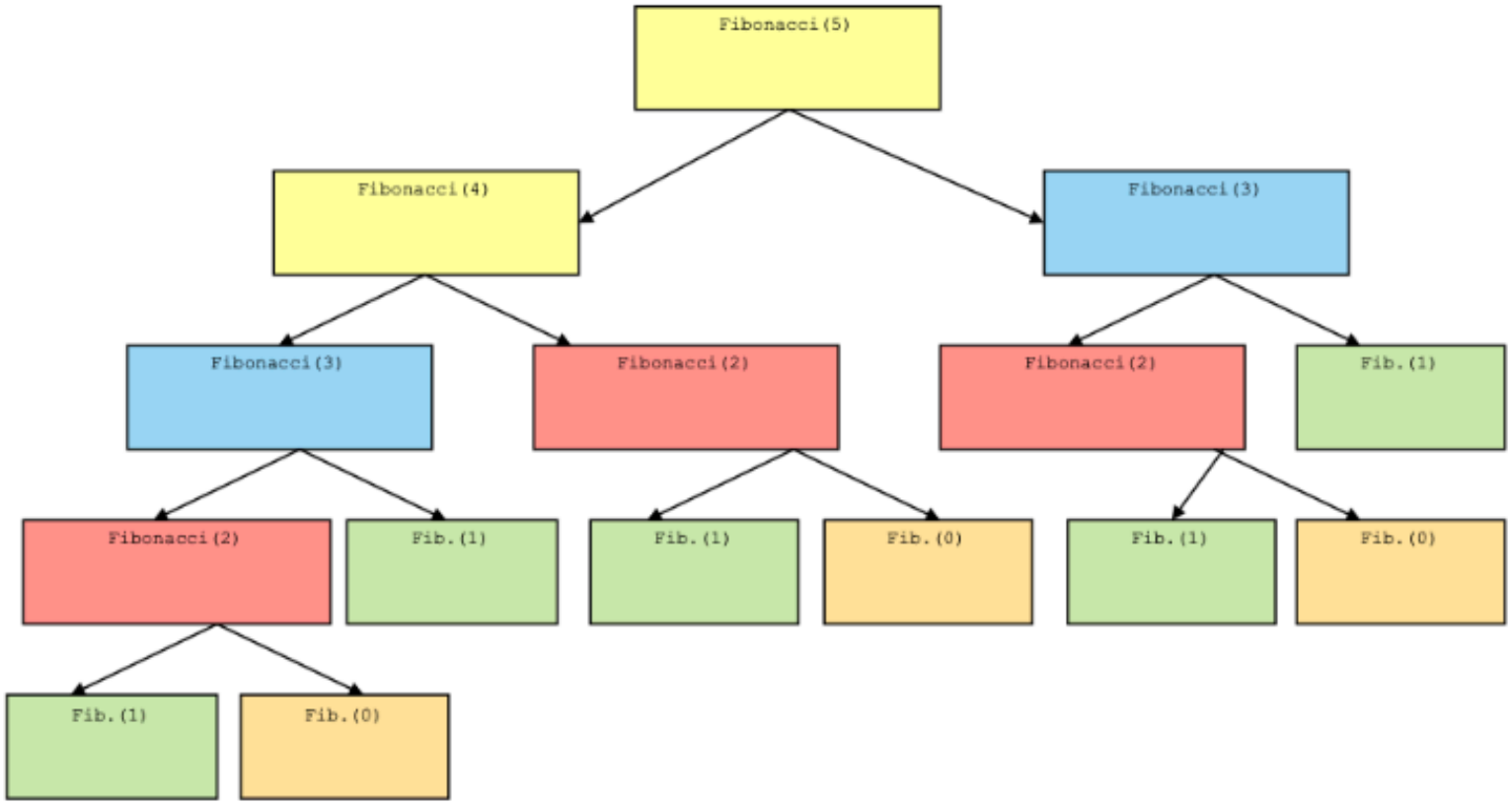
```
def FibonacciR(n):
    if n<=1:
        return n
    return Fibonacci(n-1) + Fibonacci(n-2)
```

   La ricorrenza che descrive il costo computazionale è:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) & \text{altrimenti} \end{cases}$$

   questa ricorrenza è simile alla formula che definisce i numeri di Fibonacci, che crescono molto velocemente. Vedremo che risolta quest'equazione produce un **costo esponenziale**.

La complessità esponenziale di questa funzione ricorsiva deriva dal fatto che, ad ogni chiamata, vengono eseguite due nuove chiamate ricorsive (una per $F(n\text{-}1)$ e una per $F(n\text{-}2)$). Questo porta a una crescita esponenziale del numero di chiamate, ed, in particolare, molti valori vengono calcolati più volte.
Ad esempio nella figura che segue, sono tracciate le chiamate effettuate dalla funzione per il calcolo di $F(5)$ e possiamo osservare che, i valori di $F(3), F(2), F[1]$ ed $F(0)$ vengono calcolati ripetutamente:



Si può porre rimedio a questo problema con un implementazione ricorsiva più accorta ma non è difficile pensare ad un algoritmo iterativo di costo lineare $\Theta(n)$. Ecco il codice:

```
def FibonacciI(n):
    F=[None]*(n+1)
    F[0], F[1] = 0, 1
    for i in range(2, n+1):
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

In questa versione, utilizziamo una lista per memorizzare i valori di Fibonacci da $F(0)$ fino a $F(n)$. La complessità temporale è $O(n)$, ma la complessità spaziale è (n) poiché conserviamo tutti i valori calcolati in una lista.

Per ottimizzare ulteriormente lo spazio, possiamo notare che, per calcolare $F(n)$, ci bastano solo gli ultimi due valori calcolati, $F(n\text{-}1)$ e $F(n\text{-}2)$. Quindi, non è necessario conservare l'intera sequenza, ma possiamo farlo con due variabili.
Ecco il codice:

```
def FibonacciI(n):
    a,b = 0,1
    for i in range(2, n+1):
        a, b = b, a + b
    return b
```

In questa versione, utilizziamo solo due variabili (a e b) per tenere traccia degli ultimi due numeri di Fibonacci. Questo riduce la complessità spaziale a $\Theta(1)$, poiché usiamo un numero costante di variabili, indipendentemente dal valore di n. La complessità temporale rimane $O(n)$, poiché iteriamo da 2 fino a n.

Nell'immagine seguente riportiamo i tempi di calcolo delle due versioni (quella ricorsiva $FibR$ e quella iterativa $FibI$) dell'algoritmo di Fibonacci.

```
def  FibR(n):
    if n <= 1:
        return n
    return FibR(n-1) + FibR(n-2)

from time import time

def TempoFibR(n):
    a=time()
    FibR(n)
    return time()-a


>>> TempoFibR(20)
0.00424599647521972 66
>>> TempoFibR(30)
0.5020561218261719
>>> TempoFibR(32)
1.225944995880127
>>> TempoFibR(35)
5.374284029006958
>>> TempoFibR(37)
14.031291723251343
>>> TempoFibR(40)
59.66544198989868
>>> TempoFibR(50)
?
```

```
def  FibI(n):
    a,b=0,1
    for i in range(2,n+1):
        a, b = b, a+b
    return b

from time import time

def TempoFibI(n):
    a=time()
    FibI(n)
    return time()-a

>>> TempoFibI(50)
1.2636184692382812e-05
>>> TempoFibI(500)
7.796287536621094e-05
>>> TempoFibI(5000)
0.0007832050323486328
```

Corso di laurea in Informatica
Introduzione agli algoritmi

Esercizi

SAPIENZA
UNIVERSITÀ DI ROMA

1. Per ciascuno dei seguenti problemi:

   - Progettate in Python una procedura ricorsiva per risolverli, assicurandovi che la complessità sia descritta dalla seguente relazione di ricorrenza:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(1) & \text{altrimenti} \end{cases}$$

   - Fornite successivamente una soluzione iterativa con complessità $\Theta(n)$.

problemi da risolvere:

(a) Data una lista di numeri interi, calcolare la somma di tutti i numeri pari presenti nella lista.
Ad esempio per $lista = [3, 7, -6, 2, 3, -1]$ la risposta è -4.

(b) Data una lista non vuota di numeri, determinare il valore minimo e il valore massimo nella lista. Da esempio per $lista = [3, 7, -6, 2, 3, -1]$ la risposta è $(-6, 7)$.

(c) Date due stringhe binarie della stessa lunghezza, determinare il numero di posizioni in cui differiscono.
Ad esempio per $s_1 =$ "1101" ed $s_2 =$ "1001" la risposta è 2.

(d) Stampare gli elementi di una lista in ordine inverso. Ad esempio per la lista $[1, 2, 3]$, l'output dovrebbe essere $3, 2, 1$.

2. Scrivete una procedura ricorsiva in Python per convertire un numero decimale n in una stringa che rappresenti il suo equivalente in binario.
   Ad esempio, per $n = 23$, la funzione dovrebbe restituire la stringa "10111"
   La complessità della soluzione proposta deve essere descritta dalla seguente equazione di ricorrenza: $T(n) = T(\lfloor \frac{n}{2} \rfloor) + \Theta(\log n)$.

3. Scrivete una procedura ricorsiva in Python che, data una lista A di numeri interi, restituisca il numero di coppie (i, j) tali che $A[i] = A[j]$ e $i < j$).
   Ad esempio, per la lista $[1, 1, 2, 1, 2]$, la risposta dovrebbe essere 4 (le coppie sono $(0, 1)$, $(0, 3)$, $(1, 3)$, $(2, 4)$).
   La complessità della soluzione proposta deve essere descritta dalla seguente equazione di ricorrenza: $T(n) = T(n\text{-}1) + \Theta(n)$.

4. Scrivete una funzione ricorsiva in Python che, data una lista di n numeri interi, restituisca il numero di sottoliste (sequenze contigue di elementi) in cui la somma degli elementi è pari.
   Ad esempio, per la lista $A = [1, 2, 3, 4]$, la risposta dovrebbe essere 4 (le sottoliste con somma pari sono: $([2],\ [4],\ [1, 2, 3],\ [1, 2, 3, 4])$.
   La complessità della soluzione proposta deve essere descritta dalla seguente equazione di ricorrenza: $T(n) = T(n\text{-}1) + \Theta(n)$.

5. Scrivete una procedura ricorsiva in Python che, data una lista di n numeri interi e un numero intero x, determini se esiste un sottoinsieme degli elementi della lista la cui somma sia esattamente x.
   Ad esempio, per la lista $[3, -34, 4, -12, 5, 2]$ e $x = -28$ la risposta dovrebbe essere $True$, (un sottoinsieme che somma a -28 è $\{-34, 4, 2\}$)
   La complessità della soluzione proposta deve essere descritta dalla seguente equazione di ricorrenza: $T(n) = 2T(n\text{-}1) + O(1)$.

- Progettare una procedura ricorsiva che, data una stringa s, i cui caratteri non sono necessariamente distinti, restituisca l'elenco di tutte le possibili stringhe ottenibili permutando i caratteri di s.
  Ad esempio per $s =' anna'$ la procedura deve restituire la lista:

$$[' anna',' anan',' aann',' nana',' naan',' nnaa'].$$

  Ottenere poi l'equazione di ricorrenza dell'algoritmo proposto.