

Merge sort

Data una lista **lst** di **n** numeri, l'algoritmo ripete ricorsivamente le seguenti operazioni:

- spezza **lst** in due sotto liste $l1 = lst[:n/2]$, $l2 = lst[n/2:]$
- riordina $l1$ e $l2$
- unisce le liste $l1$, $l2$ in modo che la lista risultante sia ordinata e restituisce tale lista

L'operazione di riordinamento avviene ricorsivamente (il caso base e' dato da una lista di lunghezza 1)

Esercizio

i. implementare la funzione **merge_sorted_lists(l1,l2)** (eventualmente ricorsiva) che restituisce una lista ordinata a partire dalle liste ordinate **l1**, **l2**

ii. con l'ausilio della funzione **merge_sorted_list(l1,l2)** implementare la funzione ricorsiva **merge_sort(lst)** che ordina una lista secondo l'algoritmo sopra riportato

Confrontate le prestazioni dei tre algoritmi implementati sulle seguenti liste:

```
In [ ]: l1 = range(100)
        swap(l1,50,53)
        swap(l1,87,90)

        l2 = range(1000)
        random.shuffle(l2)

        l3 = range(1000)
        l3.reverse()

        l4 = range(0,1000,2)+range(1,1000,2)

        # Per ogni lista, si ripeta l'operazione di ordinamento per 1000 volte
        # e si confrontino i tempi medi di esecuzione
```

Esercizio (OOP)

Creare un file account.py e definirvi la classe Account.

La classe Account ha i seguenti campi dati:

- il nome del proprietario (una stringa)
- la quantita' di denaro (un float)

La classe Account ha i seguenti metodi:

- un costruttore che prende il nome del cliente come parametro e inizializza il saldo a 0
- un metodo get_nome che restituisce il contenuto del campo nomecliente
- un metodo get_saldo che restituisce il contenuto del campo saldo
- un metodo set_saldo() che prende come parametro un float ed assegna al campo saldo il valore del parametro
- un metodo deposita() che prende una variabile denaro come parametro e deposita "denaro" sul conto
- prelievo(float denaro) che prende una variabile denaro come parametro e preleva dal conto la quantita' di denaro richiesta. Il conto puo' andare in rosso di 100 euro al massimo, oltre tale soglia l'operazione di prelievo fallisce stampando un messaggio di errore
- stampasaldo che visualizza il saldo corrente con il seguente formato: "nome cliente: saldo"

Aggiungere un campo transazioni alla classe Account con lo scopo di mantenere un log delle transazioni effettuate: di ogni operazione si deve salvare la data e il saldo che si ottiene dopo aver effettuato l'operazione. Quindi la variabile transazioni e' una tupla (data, saldo) dove data e' un intero (risultato di time.gmtime()) e saldo un float. Valgono le seguenti osservazioni:

- La creazione dell'account, il deposito ed il prelievo quando non fallisce, sono transazioni
- l'operazione di stampa del saldo non e' una transazione
- Ogni transazione ha un costo di 1 euro se il conto e' in attivo, oppure di 5 euro se il conto e' in rosso (dopo l'operazione stessa)
- Definire un metodo stampatransazioni che stampi, per ogni operazione andata a buon fine una stringa come la seguente: "In data (data dell'operazione) Il cliente (nome del cliente) ha prelevato/depositato xxx euro". Per stampare la data utilizzare il seguente comando: time.strftime("%X %d-%m-%Y",self.getdata())

Pattern Matching

Dato una sequenza **ptr** e una sequenza **lst**, diciamo che il pattern **ptr** ha un match in **lst** se **ptr** compare come sotto-sequenza di **lst**

i. Implementare la funzione **match_begin(ptr,lst)** che ritorna True se il pattern **ptr** compare all'inizio della lista **lst**

Bonus: implementare sia una versione ricorsiva che una versione iterativa di ****match_begin****

ii. Scrivere una funzione ricorsiva **match_num(ptr,lst)** che calcoli il numero di match di un pattern **ptr** in una lista **lst**, con l'ausilio della funzione definita al punto precedente

Hint: I booleani True e False vengono valutati 1 e 0 rispettivamente all'occorrenza.

```
False - True # -> -1
```

Esempio

```
ptr = ['a','b']
lst = ['c','a','b','b','a','b','c']
print match_num(ptr,lst) # -> 2
```

Aggiungere un campo transazioni alla classe Account con lo scopo di mantenere un log delle transazioni effettuate: di ogni operazione si deve salvare la data e il saldo che si ottiene dopo aver effettuato l'operazione. Quindi la variabile transazioni e' una tupla (data, saldo) dove data e' un intero (risultato di time.gmtime()) e saldo un float. Valgono le seguenti osservazioni:

- La creazione dell'account, il deposito ed il prelievo quando non fallisce, sono transazioni
- l'operazione di stampa del saldo non e' una transazione
- Ogni transazione ha un costo di 1 euro se il conto e' in attivo, oppure di 5 euro se il conto e' in rosso (dopo l'operazione stessa)
- Definire un metodo stampatransazioni che stampi, per ogni operazione andata a buon fine una stringa come la seguente: "In data (data dell'operazione) Il cliente (nome del cliente) ha prelevato/depositato xxx euro". Per stampare la data utilizzare il seguente comando: time.strftime("%X %d-%m-%Y",self.getdata())

iii. Definire una funzione ricorsiva **best_match_begin_len(a,b)** che calcoli la lunghezza del prefisso più lungo in comune tra le liste a e b.

Es.

```
a=['a','d','b']
```

```
b=['a','d','f','g','a','d','f','d','b','f','a','d','b','r','e','a','g']
```

```
print best_match_begin_len(a,b) # -> 2
```

iv. Definire una funzione ricorsiva **best_match_len(a,b)** che calcoli la lunghezza del prefisso più lungo di a contenuto nella lista b

Es.

```
a=['a','d','b']
```

```
b=['a','d','f','g','a','d','f','d','b','f','a','d','b','r','e','a','g']
```

```
print best_match_len(a,b) # -> 3
```

Esercizio (Backtracking)

Scrivere un programma che calcoli tutte le soluzioni del problema delle n regine (per un n fissato). Il problema, data una scacchiera di dimensioni n x n, consiste nel piazzarvi n regine in modo che non possano attaccarsi a vicenda. Più in dettaglio: data una matrice n x n, piazzare n elementi su tale matrice in modo che, per ogni coppia (i,j) di elementi, NON si verifichino le seguenti condizioni:

1) i e j sono sulla stessa riga

2) i e j sono sulla stessa colonna

3) i e j sono sulla stessa diagonale.

L'esempio seguente mostra una regina (O) su una scacchiera e tutte le posizioni che NON possono essere occupate da altre regine (X)

```
_ X X X
X X O X
_ X X X
X _ X _
```

Due regine non possono stare sulla stessa riga ed abbiamo n righe ed n regine, di conseguenza ogni riga sarà occupata da una e una sola regina. Utilizziamo questa osservazione per rappresentare la matrice tramite un vettore m dove m[i]=j significa che la regina i si trova nella riga i e nella colonna j

Proseguiamo nel seguente modo per costruire il programma. Potete utilizzare le seguenti intestazioni di funzioni

```
def inizializzascacchiera(n): #crea un vettore di dimensione n e lo inizializza appropriatamente
```

```
def stampascacchiera(m): # stampa la scacchiera come nell'esempio sopra
```

```
def posizionevalida(m,x,y): # controlla se e' possibile mettere una regina nella # scacchiera m in posizione x,y
```

```
def posizionaregina(m,x): # funzione ricorsiva che posiziona la x-esima regina sulla scacchiera
```

In []: