Cenni di Programmazione Funzionale

Parte Seconda

“To Iterate is Human, To Recurse, Divine”

4. Tail Recursion

```
def Sum(Ls):
    def SumRec(Ls, v):
        if Ls == []:
            return v
        else:
            return Ls[0] + SumRec(Ls[1:], v)
    return Sum(Ls[1:], Ls[0])
```

4. Tail Recursion

```
def Sum(Ls):
    def SumRec(Ls, v):
        if Ls == []:
            return v
        else:
            return Ls[0] + SumRec(Ls[1:], v)
    return Sum(Ls[1:], Ls[0])
```

```
def Sum(Ls):
    def TailRec(Ls, v):
        if Ls == []:
            return v
        else:
            return TailRec(Ls[1:], v+Ls[0])
    return TailRec(Ls[1:], Ls[0])
```

4. Tail Recursion

```
def ReverseTR(Ls):
    def ReverseRec(Ls, v):
        if Ls == []:
            return v
        else:
            return ReverseRec(Ls[1:], [Ls[0]] + v)
    return ReverseRec(Ls[1:], [Ls[0]])
```

4. Tail Recursion

```
def ReverseTR(Ls):
    def ReverseRec(Ls, v):
        if Ls == []:
            return v
        else:
            return ReverseRec(Ls[1:], [Ls[0]] + v)
    return ReverseRec(Ls[1:], [Ls[0]])

def Reverse2(Ls):
    def Cat(v, Ls):
        yield v
        for l in Ls:
            yield l

    def ReverseRec(Ls, v):
        if Ls == []:
            return [v]
        else:
            return Cat(Ls[-1], ReverseRec(Ls[:-1], v))
    return ReverseRec(Ls, [])

As = Reverse2([4,3,2,5,6,3])
print(next(As), next(As), next(As))
```

4. Tail Recursion

- **VANTAGGIO**: Una funzione Tail Recursive chima immediatamente se stessa con l nuovi parametri sino a quando non raggiunge la fine della lista. Si noti, che ad ogni chiamata di una funzione ricorsiva, tutti i valori delle chiamate precedente (lo stackframe) non sono più necessary: Questo permette ai compilari di ottimizare le funzioni Tail Recursive.

- **SVANTAGGIO**: Se la funzione della ricorsione (che non è tail recursive) può produrre una parte del suo output prima di processare tutta la lista, allora può essere implementata in modo da gestire le liste infinite e supportare la **LAZY EVALUATION**

SI PENSI ALLA DIFFERENZA TRA FoldRight e FoldLeft

5. Ricerca Lineare

Un **algoritmo di ricerca** (*search algorithm*) è un algoritmo per cercare un dato element con specifiche proprietà all'interno di una "collezione" di elementi.

La collezione di elementi viene di solito chiamato lo **spazio di ricerca** (*search space*).

Consideriamo ora 3 algoritmi di ricerca di elementi all'interno di una lista, che abbia la seguente **specifica:**

```
def Search(Ls, e):
    # Si assuma Ls è una lista
    # Restituisce True se "e" appartiene
    # alla lista, False altrimenti
```

5. Ricerca Lineare: Versione 0

```
def Search0(Ls, e):
      # Si assuma Ls è una lista
      # Restituisce True se "e" appartiene
      # alla lista, False altrimenti
      return e in Ls
```

5. Ricerca Lineare: Versione 1

```
def Search0(Ls, e):
    # Si assuma Ls è una lista
    # Restituisce True se "e" appartiene
    # alla lista, False altrimenti
    return e in Ls


def Search1(Ls, e):
    for i in range(len(Ls)):
        if Ls[i] == e:
            return True
    return False
```

DOMANDA: COMPLESSITÀ DI QUESTA IMPLEMENTAZIONE?

5. Ricerca Lineare: Versione 2

```
def Search1(Ls, e):
    for i in range(len(Ls)):
        if Ls[i] == e:
            return True
    return False
```

```
def Search2(Ls, e):
    for l in Ls:
        if l == e:
            return True
    return False
```

DOMANDA: DIFFERENZE?

5. Ricerca Lineare: Versione 2

```
def Search2(Ls, e):
    for l in Ls:
        if l == e:
            return True
    return False
```

DOMANDA: RICORSIVA? E SE LA LISTA FOSSE ORDINATA?

```
def SearchRec(Ls, e):
    if Ls == []
        return False
    if Head(Ls) == e:
        return True
    return SearchRec(Tail(Ls, e))
```

5. Ricerca Lineare: Versione 3

```
def Search2(Ls, e):
    for l in Ls:
        if l == e:
            return True
    return False
```

DOMANDA: SE LA LISTA FOSSE ORDINATA?

```
def Search3(Ls, e):
    for l in Ls:
        if l == e:
            return True
        if l > e:
            return False
    return False
```

NOTA: MIGLIORA L'AVERAGE RUNNING TIME, NON IL WORST CASE RUNNING TIME!

6. Ricerca Binaria

Come possiamo generalizzare l'idea di **Bisection Search** vista per trovare la radice quadrata di un numero?

6. Ricerca Binaria

Come possiamo generalizzare l'idea di **Bisection Search** vista per trovare la radice quadrata di un numero, supponendo che la lista sia ordinate in modo crescente?

IDEA (GROSSO MODO):

1. Si prenda l'elemento "mid" che sta in mezzo alla lista
2. Lo si confronta con l'elemento "e" che stiamo cercando . Se sono uguali abbiamo finite (True)
3. Confrontiamo "mid" con "e": se "mid" è minore, continuamo la ricerca nella metà sinistra della lista, altrimenti in quella destra

6. Ricerca Binaria

```
def BinarySearch(Ls, e):
      def bSearch(Ls, e, low, high):
            if low >= high
                  return e == low
            mid=(low+high)// 2
            if Ls[mid] == e:
                  return True
            if Ls[mid] < e:
                  return bSearch(Ls, e, mid+1, high)
            else:
                  return bSearch(Ls, e, low, mid-1)

      return bSearch(Ls, e, 0, len(Ls)-1)
```

DOMANDA: COMPLESSITÀ WORST-CASE?

6. Ricerca Binaria

```
def bSearch(Ls, e, low, high):
      if low >= high
            return e == low
      mid=(low+high)// 2
      if Ls[mid] == e:
            return True
      if Ls[mid] < e:
            return bSearch(Ls, e, mid+1, high)
      else:
            return bSearch(Ls, e, low, mid-1)

def BinarySearch(Ls, e):
      return bSearch(Ls, e, 0, len(Ls)-1)
```

DOMANDA: DIFFERENZA CON LA VERSIONE PRECEDENTE?

7. Selection Sort

Abbiamo supposto prima che la lista deve essere ordinata: ma quanto “costa” ordinare una lista?

IDEA DEL SELECTION SORT:

Si mantiene il seguente **INVARIANTE**: *data una suddivisione di una lista in un **prefisso** Ls[0:i] e un **suffisso** Ls[i+1:], il prefisso è sempre ordinato, e nessun elemento del prefisso è maggiore del più piccolo elemento nel suffisso.*

DOMANDA: RIUSCITE A PENSARE AD UN’IMPLEMENTAZIONE RICORSIVA e CON UN HIGH ORDER FUNCTION PER IL CONFRONTO?

7. Selection Sort

Abbiamo supposto prima che la lista deve essere ordinata: ma quanto “costa” ordinare una lista?

IDEA DEL SELECTION SORT:

Si mantiene il seguente **INVARIANTE**: *data una suddivisione di una lista in un **prefisso** Ls[0:i] e un **suffisso** Ls[i+1:], il prefisso è sempre ordinato, e nessun elemento del prefisso è maggiore del più piccolo elemento nel suffisso.*

DOMANDA: COMPLESSITÀ WORST-CASE?

8. Quick Sort

IDEA:

1. Scelgo il primo elemento della lista come pivot
2. Costruisco 3 liste:
    1. Pivots: tutti gli elementi uguali a quello scelto
    2. Smaller: tutti gli elementi inferiori a quello scelto
    3. Greater: tutti gli element superiori a quello scelto
3. Ordino la lista di Smaller e Greater
4. Restituisco la concatenazione:

    Smaller + Pivots + Greater

DOMANDA: COMPLESSITÀ WORST-CASE?

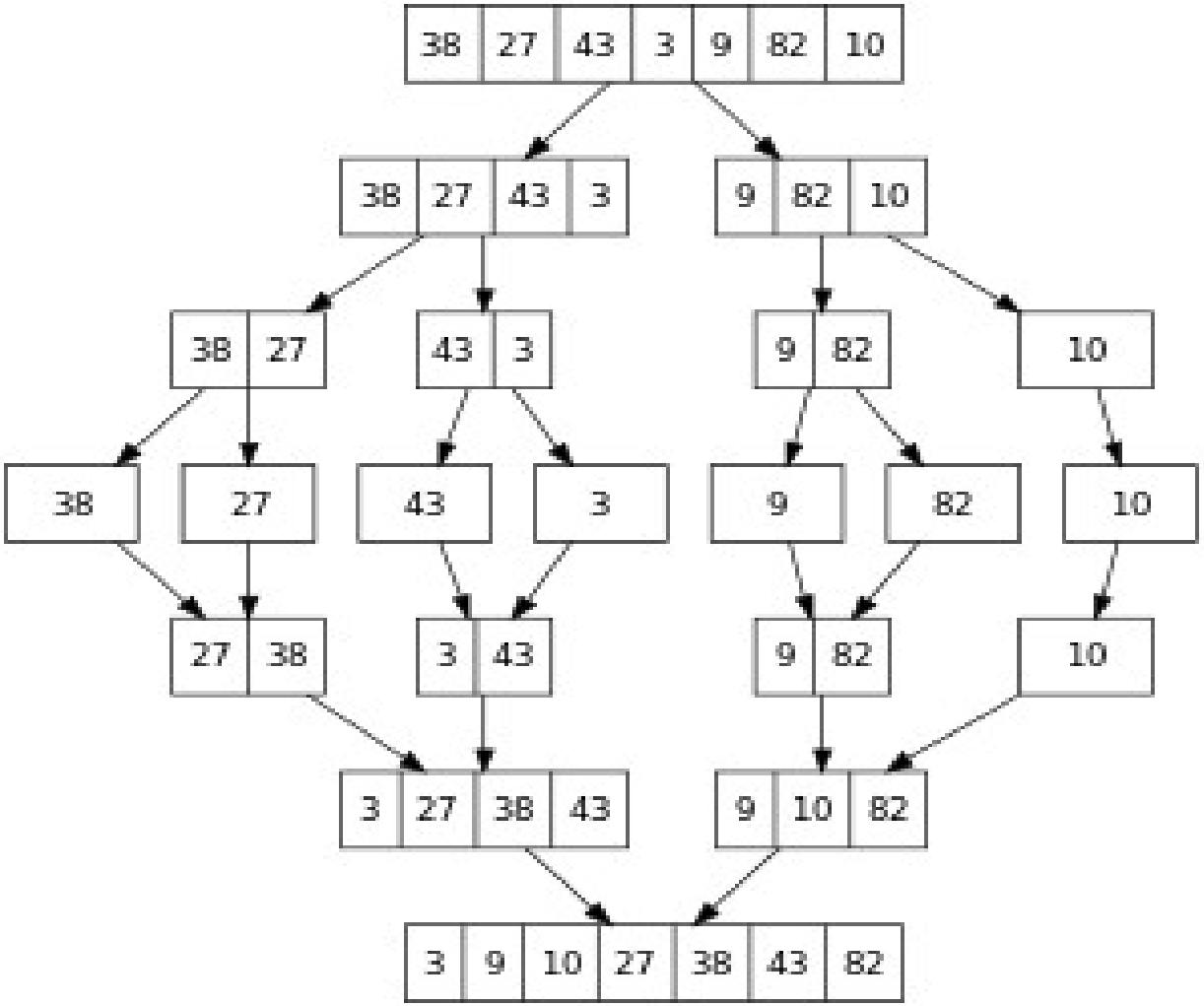
9. Merge Sort

Merge Sort è il classico esempio di "**divide-and-conquer algorithm**", in cui l'idea di base è di combinare soluzioni di istanze più semplici del problema iniziale.

IDEA DI MERGE SORT:

1. Se la lista ha lunghezza <= 1, è già ordinata
2. Altrimenti, divido la lista in due, e applico l'algoritmo su ciascuna delle due liste
3. Unisco in un'unica lista le due liste ordinate

9. Merge Sort (da Wikipedia)



10. Confronti

n	O(n^2)	O(n*log(n))	O(n^2) time	O(n*log(n)) time
1	1	0	1.66667E-05	0
10	100	10	0.001666667	0.000166667
100	10000	200	0.166666667	0.003333333
1000	1000000	3000	16.66666667	0.05
10000	100000000	40000	1666.666667	0.666666667
100000	10000000000	500000	166666.6667	8.333333333
1000000	1E+12	6000000	16666666.67	100

11. functools.lru_cache

- Vedere l’esempio sui notebooks
- Documentazione: https://docs.python.org/3/library/functools.html