UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Algorithms and Data Structures

Homework 2

Anno Accademico 2024/2025

Studenti
Filomena Vigliotti
matr. M63001734
Antonio Sirignano
matr. M63001732

Indice

Capitolo 1

Problema 1

Data un vettore che può contenere numeri interi sia positivi che negativi, trovare il sotto-array di numeri contigui che ha la somma più grande, e riportare tale somma.
Ogni riga contiene un caso di test, rappresentato dagli elementi del vettore di cui si vuole calcolare la somma del massimo sotto-array. I casi di test terminano con una riga `END`.

1.1 Pseudocodice

Si è utilizzata una strategia di tipo *Bottom-up*, ovvero si ordinano i sottoproblemi per dimensioni e si risolvono a partire da quello più piccolo.
Nel caso in esame, si ha il seguente pseudocodice.

```
procedure MaxSubarraySum(Arr):
    max_sum = −∞
    current_sum = 0

    for i = 1 to Arr.lengh do
        current_sum = max (current_sum + Arr[i], Arr[i])
        max_sum = max (max_sum, current_sum)
    end for
return max_sum
end procedure
```

La procedura descritta calcola la somma massima di un sottoarray contiguo di un array Arr, utilizzando un approccio di programmazione dinamica. Nella procedura sono stati inizializzate due variabili: max_sum a -infinito e $current\_sum$ a 0. Nel ciclo for si pone la somma corrente al massimo tra la somma corrente più il valore i-esimo dell'array e la somma corrente; inoltre si pone la somma massima al massimo tra il valore attuale della somma massima e la somma corrente. Si ritorna poi il valore *max_sum*, ovvero la somma massima. La programmazione dinamica qui è implicita nell'uso di una relazione ricorsiva per calcolare *current_sum*: si divide il problema globale in sottoproblemi più piccoli (trovare la somma massima di un sottoarray che termina in ogni indice). Il calcolo di *current_sum* al passo i si basa su *current_sum* del passo $i-1$. Utilizzando questo approccio, non si ricalcolano somme per ogni possibile sottoarray (come accadrebbe in un approccio brute-force), invece, si aggiorna la soluzione in un singolo passaggio.

1.2 Implementazione

Il codice è stato implementato in *Python* come mostrato:

```python
def max_sub_sum(arr):
    max_sum = float('-inf')                          # si imposta come valore iniziale della somma massima valore -inf
    current_sum = 0                                  # valore della somma corrente

    for elem in arr:
        current_sum = max(elem, current_sum+elem)
        max_sum = max(max_sum, current_sum)

    return max_sum

def main():
    print("Max SubArray Sum\n")

    lines = []
    results = []

    while True:
        print("Insert new array: ")
        line = input().strip()

        if line == "END":
            break

        lines.append(line)
        array = list(map(int, line.split()))

        results.append(max_sub_sum(array))

    print("\nRESULTS:\n")
```

```python

    for i in range(len(lines)):
        print(lines[i], results[i], sep="\t")

if __name__ == "__main__":
    main()
```

La funzione `max_sub_sum(arr)` implementa lo pseudocodice precedente.

Il `main` permette di inserire da terminale i casi test che verranno presentati nella sezione successiva.

1.3 Test

I casi test che si presentano sono i seguenti:

Input	Output
-1 -3 4 2	6
-1 2 -5 7	7
1 -3 4 -2 6 8 -10	16
4 -6 2 -8 5	5
7 -5 1 3 9	15

Tabella 1.1: Casi test

Effettuando il test, i risultati sono i seguenti:

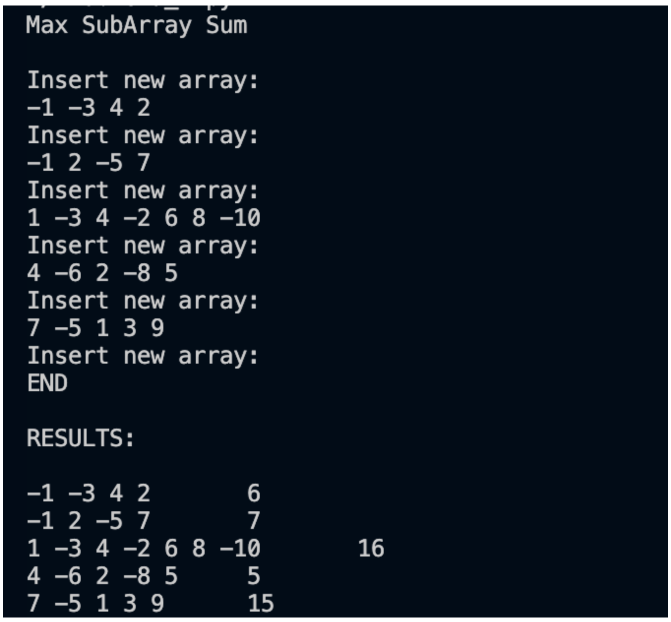


Figura 1.1: Output del codice

1.4 Complessità

Per quanto riguarda la complessità temporale, si calcola banalmente vedendo che il codice scorre l'array una sola volta.
Quindi supposto che l'array contenga n elementi si ha che:

$$T(n) = O(n)$$

. Per quanto riguarda la complessità spaziale è pari ad $O(1)$, poiché vengono utilizzate solamente due variabili di supporto per memorizzare lo stato.

Capitolo 2

Problema 2

Dato un certo importo da pagare di V centesimi ed una lista di n monete $coin[n]$ che possiamo usare (per esempio $n = 3$ monete, quelle da 1 centesimo ($coin[0] = 1$), da 5 centesimi ($coin[1] = 5$), da 10 centesimi ($coin[2] = 10$)), si scriva un algoritmo per determinare il numero minimo di monete che dobbiamo usare per arrivare all'importo esatto V o al più piccolo importo maggiore di V. Si assuma di avere un numero illimitato di monete di tutti i tipi.

2.1 Pseudocodice

Anche in questo caso è stata utilizzata una strategia di tipo *Bottom-up*, con la suddivisione del problema in problemi più piccoli e la corrispondente risoluzione seguendo un ordine di grandezza. La risoluzione è mostrata nel seguente pseudocodice:

```
1: procedure MinCoins(V, n, coins):
2:     max_value = V + max(coins)
3:     results[0 ... max_value + 1]                ▷ Vettore di lunghezza pari a max_value
4:     for k = 1 to max_value + 1 do
5:         results[k] = ∞
6:     end for
7:     results[0] = 0
8:
9:     for i = 1 to n do
10:        for j = coins[i] to max_value + 1 do
11:            results[j] = min(results[j], results[j − coins[i]] + 1)
12:        end for
13:    end for
14:
15:    closest_value = NIL
16:    closest_diff = ∞
17:    for i = 1 to results.lenght do
18:        if results[i]! = ∞ then
19:            diff = |i − V|
20:            if  diff < closest_diff OR (diff == closest_diff AND i > closest_value)
   then
21:                closest_value = i
22:                closest_diff = diff
23:            end if
24:        end if
25:    end for
        return results[closest_value]
26: end procedure
```

Dalla riga 2 alla riga 6 viene istanziato un vettore $results$ di dimensione $max\_value = V + max(coins)$, in modo da costruire tutti i valori possibili fino ad $V + max(coins)$, in modo da ritornare valutare anche i valori più grandi di V. Gli elementi del vettore vengono inizializzati a ∞ eccetto il primo che viene inizializzato a 0 poiché per raggiungere il valore 0 sono necessarie 0 monete.

Ogni elemento $results[i]$ rappresenterà il numero minimo di monete necessario per ottenere il valore i.

Dalla riga 8 alla 12, sono presenti due cicli for innestati permettendo il calcolo del risultato j-esimo valutandolo come il minimo tra il risultato j-esimo e il risultato in $j - coins[i]$ incrementato di 1.

Vengono successivamente istanziate due variabili di supporto, $closest\_value$ e $closest_{d}iff$, ovvero il valore più vicino a V e la differenza tra V e quest'ultimo.

Col successivo ciclo `for`, percorriamo tutto il vettore $results$, dove vengono valutati esclusivamente i valori diversi da ∞: viene effettuata la differenza tra i e V, e se la differenza è minore di $closest\_diff$ oppure la differenza è uguale a $closest\_diff$ e il valore i (iterazione del ciclo) è maggiore di $closest\_value$, allora si pone $closest\_value = i$ e $closest\_diff = diff$.

Alla fine si ritorna il $results[closest\_value]$, che qualora non esistesse, sarà pari ad ∞.

2.2 Implementazione

Di seguito vi è l'implementazione in Python del precedente pseudocodice.

```python
def min_coins(V, n, coins):
    max_value = V + max(coins)
    results = [float("inf")]*(max_value+1)
    results[0] = 0

    for i in range(n):
        for j in range(coins[i], max_value+1):
            results[j] = min(results[j], results[j-coins[i]]+1)

    closest_value = None
    closest_diff = float("inf")

    for i in range(len(results)):
        if results[i] != float("inf"):
            diff = abs(i-V)
            if diff < closest_diff or (diff == closest_diff and i > closest_value):
                closest_value = i
                closest_diff = diff

    return results[closest_value]

def main():
    print("Minimum number of coins to archieve value V")
    print("The first value is V, the second one is the number of coins, and the others
     are the values of each coin")
    print("If the result is -inf- there is no way toachieve the V or the nearest sum
    greater than V\n")

    lines = []
    results = []

    while True:
        print("Insert new array: ")
        line = input().strip()

```

```python
        if line == "END":
            break

        lines.append(line)
        array = list(map(int, line.split()))

        results.append(min_coins(array[0], array[1], array[2:]))

    print("\nRESULTS:\n")

    for i in range(len(lines)):
        print(lines[i], results[i], sep="\t")

if __name__ == "__main__":
    main()
```

La funzione `min_coins(V, n, coins)` implementa lo pseudocodice precedente.

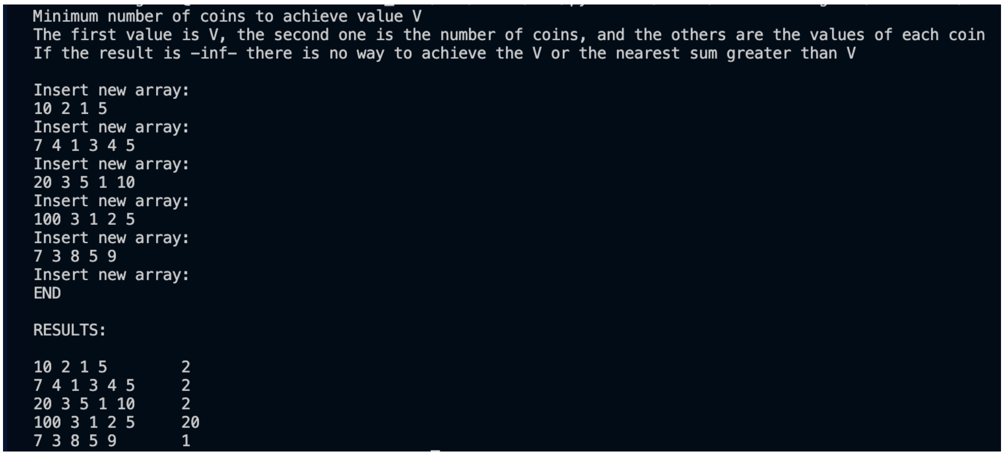
Il `main` permette di ottenere i parametri da passare in ingresso alla funzione `min_coins`, e di inserire da terminale i casi test che verranno presentati nella sezione successiva.

2.3 Test

Input	**Output**
10 2 1 5	2
7 4 1 3 4 5	2
20 3 5 1 10	2
100 3 1 2 5	20
7 3 8 5 9	1

Tabella 2.1: Casi test

Effettuando il test, i risultati sono i seguenti:



```
Minimum number of coins to achieve value V
The first value is V, the second one is the number of coins, and the others are the values of each coin
If the result is -inf- there is no way to achieve the V or the nearest sum greater than V

Insert new array:
10 2 1 5
Insert new array:
7 4 1 3 4 5
Insert new array:
20 3 5 1 10
Insert new array:
100 3 1 2 5
Insert new array:
7 3 8 5 9
Insert new array:
END

RESULTS:

10 2 1 5        2
7 4 1 3 4 5     2
20 3 5 1 10     2
100 3 1 2 5     20
7 3 8 5 9       1
```

Figura 2.1: Output del codice

2.4 Complessità

La complessità di questo algoritmo è data dalla somma delle complessità necessarie per la costruzione dell'array $results$ e della ricerca della somma più vicina.

Dalla riga 9 alla riga 13, si ha una complessità pari ad $O(n \cdot max\_value)$.

Dalla riga 17 alla riga 25, si ha una complessità pari ad $O(max\_value)$.

Complessivamente si ha quindi:

$$T(n) = O(n \cdot max\_value) + O(max\_value) = O(n \cdot max\_value)$$

Mentre per quel che riguarda la complessità spaziale, si ha che l'array $results$ ha complessità $O(max\_value)$ mentre le variabili $closest\_value$ e $closest\_diff$ hanno complessità costante $O(1)$. Quindi complessivamente

$$S(n) = O(max\_value)$$

Si ricorda che max_value è pari al valore massimo tra tutte le monete sommato al valore V. Quindi si ha che la complessità sia spaziale che temporale è dipendente sia al valore che si vuole raggiungere V che al valore delle monete utilizzate.