Esercizio 1 (10 punti) Realizzare una funzione `Prod(A,k)` che dato un array `A` di interi ≥ 0 ordinato in senso crescente e un valore intero `k` ≥ 0 verifica se esistono due indici `i` e `j` tali che `k = A[i]*A[j]`. Valutarne la complessità. Adattare la soluzione al caso in cui i valori nell'array possono essere negativi (assumendo ancora `k` ≥ 0).

Domanda A (6 punti) Dare la definizione formale della classe $O(f(n))$ per una funzione $f(n)$. Mostrare che se $f(n) = O(n^2)$ e $g(n) = O(n)$ allora $f(n) + g(n) = O(n^2)$.

Domanda B (7 punti) Dare la definizione di max-heap. Data la sequenza di elementi `17, 15, 22, 20, 19, 21, 30`, si specifichi il max-heap ottenuto inserendo, a partire da uno heap vuoto, uno alla volta questi elementi nell'ordine indicato e infine rimuovendo 15. Si descriva sinteticamente come si procede per arrivare al risultato.

Domanda C (6 punti) Sia T un albero binario i cui nodi x hanno i campi *x.left*, *x.right*, *x.key*. L'albero si dice un *sum-heap* se per ogni nodo x, la chiave di x è maggiore o uguale sia alla somma delle chiavi nel sottoalbero sinistro che alla somma delle chiavi nel sottoalbero destro.

Scrivere una funzione `IsSumHeap(T)` che dato in input un albero T verifica se T è un sum-heap e ritorna un corrispondente valore booleano. Valutarne la complessità.

Domanda B (4 punti) Si consideri una tabella hash di dimensione $m = 9$, e indirizzamento aperto con doppio hash basato sulle funzioni $h_1(k) = k \bmod m$ e $h_2(k) = 1 + k \bmod (m-2)$. Si descriva in dettaglio come avviene l'inserimento della sequenza di chiavi: 12, 3, 22, 14, 38.

Certo! Ecco le funzioni riscritte in pseudocodice formale:

1. Per la funzione Prod(A,k):

```
Algoritmo Prod(A, k)
Input: A (array di interi), k (intero ≥ 0)
Output: prodotto di A[k] per sé stesso

    if k < 0 OR k ≥ lunghezza(A) then
        return errore "k non valido"
    endif

    return A[k] * A[k]

Fine Algoritmo
```

Complessità: O(1) tempo e spazio

2. Domanda A (6 punti): Per dimostrare che $f(n) + g(n) = O(n^2)$ quando $f(n) = O(n^2)$ e $g(n) = O(n)$:

Dalla definizione di $O(n^2)$, esistono costanti positive c_1 e n_1 tali che: $f(n) \leq c_1n^2$ per ogni $n \geq n_1$

Dalla definizione di $O(n)$, esistono costanti positive c_2 e n_2 tali che: $g(n) \leq c_2n$ per ogni $n \geq n_2$

Quindi, per $n \geq \max(n_1, n_2)$: $f(n) + g(n) \leq c_1n^2 + c_2n \leq c_1n^2 + c_2n^2 = (c_1 + c_2)n^2$

Pertanto, $f(n) + g(n) = O(n^2)$

3. Domanda B (7 punti): Costruiamo il max-heap passo dopo passo con la sequenza 17, 15, 22, 20, 19, 21, 30:
4. Iniziamo con heap vuoto
5. Inseriamo 17: [17]
6. Inseriamo 15: [17, 15]
7. Inseriamo 22: [22, 15, 17]
8. Inseriamo 20: [22, 20, 17, 15]
9. Inseriamo 19: [22, 20, 17, 15, 19]
10. Inseriamo 21: [22, 20, 21, 15, 19, 17]
11. Inseriamo 30: [30, 20, 22, 15, 19, 17, 21]

Ora rimuoviamo 15:

1. Sostituiamo 15 con 21 (ultimo elemento)
2. Risistemare l'heap: [30, 20, 22, 21, 19, 17]
3. Domanda C (6 punti): Per verificare se un albero è un sum-heap:

```
Algoritmo GetSommaSubtree(nodo)
Input: nodo di un albero binario
Output: somma di tutte le chiavi nel subtree radicato in nodo

    if nodo = NULL then
        return 0
    endif

    return nodo.chiave + GetSommaSubtree(nodo.sinistro) +
GetSommaSubtree(nodo.destro)

Fine Algoritmo

Algoritmo IsSumHeap(T)
Input: T (radice di un albero binario)
Output: booleano (true se è un sum-heap, false altrimenti)

    if T = NULL then
        return true
    endif

    somma_sinistra ← GetSommaSubtree(T.sinistro)
```

```
    somma_destra ← GetSommaSubtree(T.destro)

    if T.chiave < somma_sinistra OR T.chiave < somma_destra then
        return false
    endif

    return IsSumHeap(T.sinistro) AND IsSumHeap(T.destro)

Fine Algoritmo
```

Complessità:

- Tempo: $O(n^2)$ dove n è il numero di nodi
- Spazio: O(h) dove h è l'altezza dell'albero (per lo stack di ricorsione)

3. Per l'inserimento nella tabella hash con doppio hashing:

```
Algoritmo Hash1(k, m)
Input: k (chiave), m (dimensione tabella)
Output: primo indice di hash

    return k mod m

Fine Algoritmo

Algoritmo Hash2(k, m)
Input: k (chiave), m (dimensione tabella)
Output: incremento per risoluzione collisioni

    return 1 + k mod (m-2)

Fine Algoritmo

Algoritmo InserisciHash(T, k, m)
Input: T (tabella hash), k (chiave da inserire), m (dimensione tabella)

    i ← 0
    pos ← Hash1(k, m)

    while T[pos] ≠ NULL do
        i ← i + 1
        if i = m then
            return errore "Tabella piena"
```

```
        endif
        pos ← (Hash1(k, m) + i * Hash2(k, m)) mod m
    endwhile

    T[pos] ← k

Fine Algoritmo
```

Complessità dell'inserimento:

- Caso medio: O(1)
- Caso peggiore: O(m) dove m è la dimensione della tabella

5. Domanda B (4 punti): Per la tabella hash con m=9 e le funzioni di hash: $h_1(k)$ = k mod m $h_2(k)$ = 1 + k mod (m-2)

Inseriamo la sequenza [12, 3, 22, 14, 38]:

1. k=12:
    - $h_1(12)$ = 3
    - posto in posizione 3
2. k=3:
    - $h_1(3)$ = 3 (collisione)
    - $h_2(3)$ = 4
    - posto in posizione 4
3. k=22:
    - $h_1(22)$ = 4 (collisione)
    - $h_2(22)$ = 2
    - posto in posizione 2
4. k=14:
    - $h_1(14)$ = 5
    - posto in posizione 5
5. k=38:
    - $h_1(38)$ = 2 (collisione)
    - $h_2(38)$ = 4 (collisione)
    - $h_1(38)$ + 2_$h_2(38)$ = 2 + 2_4 = 10 mod 9 = 1
    - posto in posizione 1

La tabella finale sarà: [-, 38, 22, 12, 3, 14, -, -, -]