Array Division- Tömb osztás

Megoldás

A feladat alapvetően arról szól, hogy úgy kell felosztanunk a tömböt k részre, hogy a legnagyobb rész-tömb összege a lehető legkisebb legyen. Ez egy klasszikus **Kétirányú keresési** probléma, amit dinamikus programozással vagy bináris kereséssel oldhatunk meg.

1. Bemenet beolvasása

```
n, k = map(int, input().split())

arr = list(map(int, input().split()))
```

- **Mi történik itt?**
  - Az első sorban beolvassuk az n (a tömb hossza) és a k (a kívánt rész-tömbök száma) értékeket.
  - A második sorban beolvassuk a tömb elemeit, és egy listává konvertáljuk őket arr változóba.

2. A can_divide függvény

```
def can_divide(arr, n, k, max_sum):
    current_sum = 0
    subarrays = 1  # Az első rész-tömb már kezdődik

    for num in arr:
        if current_sum + num > max_sum:
            subarrays += 1  # Új rész-tömb kezdődik
            current_sum = num  # Az új rész-tömb kezdődik az aktuális elemmel
            if subarrays > k:
                return False  # Túl sok rész-tömb lenne
        else:
            current_sum += num  # Hozzáadjuk a számot az aktuális rész-tömbhöz

    return True
```

- **Mi történik itt?**
  - A can_divide egy segédfüggvény, ami azt ellenőrzi, hogy lehetséges-e a tömböt k darab rész-tömbre felosztani úgy, hogy egyik rész-tömb összege sem haladja meg a max_sum értéket.

- **Hogyan működik?**

1. **current_sum és subarrays:**
    - current_sum tárolja az aktuális rész-tömb összegét.
    - subarrays nyomon követi, hány rész-tömböt hoztunk létre.
2. **Iterálás a tömb elemein:**
    - A tömb minden elemén végigmegyünk, és megnézzük, hogy hozzá tudjuk-e adni az aktuális elem értékét a jelenlegi rész-tömb összegéhez (current_sum).
    - Ha az új összeg meghaladná a max_sum értéket, akkor:
        - **Új rész-tömb kezdése:** Egy új rész-tömböt indítunk, és a subarrays változót növeljük.
        - Ha a subarrays meghaladja a kívánt k darab rész-tömböt, akkor a felosztás nem lehetséges, ezért False-t adunk vissza.
    - Ha a rész-tömb összeg nem haladja meg a max_sum-ot, hozzáadjuk az aktuális számot az aktuális rész-tömbhöz.
3. **Visszatérés:**

    Ha végigmentünk a tömbön, és sikerült a felosztás, akkor True-t adunk vissza.

- **Miért fontos ez?**
    - A can_divide függvény az alapja a bináris keresésnek. Segít eldönteni, hogy egy adott max_sum értékkel lehetséges-e a kívánt felosztás.

3. A minimize_max_subarray_sum függvény

```
def minimize_max_subarray_sum(n, k, arr):
    left, right = max(arr), sum(arr)

    while left < right:
        mid = (left + right) // 2
        if can_divide(arr, n, k, mid):
            right = mid  # Ha lehetséges, próbálkozunk egy kisebb maximális összeggel
        else:
            left = mid + 1  # Ha nem lehetséges, akkor növelnünk kell a maximális összeget

    return left
```

- **Mi történik itt?**
    - A minimize_max_subarray_sum a bináris keresést hajtja végre, hogy megtalálja a legkisebb maximális rész-tömb összeg értékét, amely lehetővé teszi a tömb k rész-tömbre történő felosztását.
- **Hogyan működik?**

1. **Keresési tartomány inicializálása:**
    - A left változó az alsó keresési határ, ami a legnagyobb szám a tömbben, mivel legalább ennyi kell, hogy bármelyik rész-tömb tartalmazzon egy számot.
    - A right változó az összes szám összegének a maximális értéke, mivel ennél nagyobb összeget nem szükséges keresni (ez lenne a legrosszabb eset, ha az egész tömb egyetlen rész-tömbbe kerül).
2. **Bináris keresés:**
    - A bináris keresésben a középső érték (mid) a keresett maximális összeg.
    - Meghívjuk a can_divide függvényt, hogy megnézzük, hogy a mid értékkel lehetséges-e a felosztás.
    - Ha a felosztás lehetséges (True), akkor próbálkozunk egy kisebb maximális összeggel, ezért a right értéket csökkentjük.
    - Ha a felosztás nem lehetséges (False), akkor növeljük a left értéket, hogy egy nagyobb max_sum-ot próbáljunk ki.
3. **Visszatérés:**
    - A bináris keresés addig folytatódik, amíg a left és right értékek meg nem egyeznek. A végső left érték lesz a keresett maximális összeg, amit a kimenetbe írunk.

- **Miért fontos ez?**
    - A bináris keresés hatékony módszer arra, hogy megtaláljuk a legkisebb maximális összegű rész-tömböt, amely lehetővé teszi a kívánt felosztást. A keresési tartomány folyamatosan szűkül, amíg el nem érjük a legjobb választ.

4. A fő program

python

Kód másolása

```
# Bemenet
n, k = map(int, input().split())
arr = list(map(int, input().split()))
```

Eredmény

print(minimize_max_subarray_sum(n, k, arr))

- **Mi történik itt?**
    - Beolvassuk a bemeneti adatokat.
    - Meghívjuk a minimize_max_subarray_sum függvényt, amely kiszámítja a legkisebb maximális rész-tömb összegét.
    - A kimenetben kiírjuk a választ.

5. Miért választottuk ezt a megoldást?

- **Bináris keresés és Greedy:**

  A bináris keresés és Greedy módszer kombinálása egy hatékony megoldást ad a problémára. A bináris keresés biztosítja, hogy az összes lehetséges válasz közül a legjobb eredményt találjuk meg logaritmikus időben, míg a Greedy megközelítés egyszerűen ellenőrzi, hogy egy adott max_sum érték mellett lehetséges-e a felosztás.

- **Hatékonyság:**
  A program működési ideje O(n log(sum)) — a bináris keresés logaritmikus lépései és a Greedy ellenőrzés miatt, amely egyszer végigmegy a tömbön. Ez elegendően gyors ahhoz, hogy a feladatot hatékonyan megoldjuk a maximális korlátozások mellett

  ($n \leq 2 * 10^5$).

Összegzés:

- **Bináris keresés:** A lehetséges maximális összeg közötti tartományt csökkentjük.
- **Greedy ellenőrzés:** Minden egyes próbált maximális összeg esetén ellenőrizzük, hogy lehetséges-e felosztani a tömböt k rész-tömbre úgy, hogy egyik sem haladja meg a próbált összeg értékét.

Példa végrehajtása

Bemenet:

Kód másolása

5 3

2 4 7 3 5

Kimenet:

Kód másolása

8

Ebben az esetben a legjobb felosztás a következő lehet: [2,4], [7], [3,5], és a legnagyobb összeg 8.