پاسخنامه تمرین شماره ۴

Hash

and

Sort

دانشکده مهندسی برق و کامپیوتر

ساختمان های داده و الگوریتم -

پاییز ۱۴۰۱

استاد: دکتر هشام فیلی

طراح تمرین : **علی اخگری**

---

۱. الف) اعداد ۱۲، ۲۱ و ۲۵ بدون اتفاق افتادن تصادم به جدول اضافه می شوند. اما اضافه کردن ۹ و ۱۳ باعث ایجاد تصادم می شود که در هر کدام باید i را یکی زیاد کنیم.

0	1	2	3	4	5	6	7	8	9	10	11
12											
12									21		
12	25								21		
12	25								21	9	
12	25	13							21	9	

ب) در درهم سازی دوگانه، طبق رابطه $h(k,\ i)\ =\ (h_1(k)\ +\ i\ h_2(k))\ \%\ m$ باید عمل کنیم.

0	1	2	3	4	5	6	7	8	9	10	11
12											
12									21		
12	25								21		
12	25								21		11
12	25		13						21		

۲.الف) ابتدا آرایه را با هر روش مرتب سازی ای که هزینه $O(nlogn)$ دارد، سورت می کنیم. سپس روی عناصر این آرایه پیمایش می کنیم و به ازای هر عنصر مقدار $S\ -\ arr[i]$ را در آرایه با استفاده از binary search جستجو می کنیم، که هزینه این کار نیز $O(nlogn)$ است. بنابراین هزینه کلی این الگوریتم $O(nlogn)$ خواهد بود.

۲.ب) روی عناصر آرایه پیمایش می کنیم و به ازای هر عنصری که می بینیم، آن را در یک جدول درهم سازی ذخیره می کنیم و سپس وجود مقدار $S - arr[i]$ را در جدول درهم سازی جستجو می کنیم. این روند را تا جایی ادامه می دهیم که مقدار $S - arr[i]$ در جدول موجود باشد. هزینه اضافه کردن و جستجو در جدول درهم سازی $O(1)$ است و هزینه پیمایش روی آرایه $O(n)$ است، بنابراین هزینه کلی این الگوریتم $O(n)$ است.

۳. در ابتدا متغیر های زیر را تعریف می کنیم:

current_sum: نشان دهنده جمع عناصر آرایه از ابتدا تا ایندکس iام (در ابتدا مقدار صفر دارد.)

max_len: نشان دهنده طول بزرگترین زیر آرایه که مجموع عناصر آن صفر است. (در ابتدا مقدار صفر دارد.)

hash_table: یک جدول درهم سازی که کلید آن current_sum و مقدار آن ایندکس i است.

الگوریتم به این صورت است که روی آرایه پیمایش می کنیم:

اگر به جایی رسیدیم که current_sum برابر با صفر شد، مقدار max_len را برابر با i+1 می گذاریم.

حال فرض کنید که به خانه j ام از این آرایه رسیده باشیم، چک می کنیم که آیا مجموع اعضای آرایه از ابتدا تا ایندکس j در hash_table موجود است یا خیر. اگر موجود باشد، نتیجه می گیریم که مجموع اعضای i+1 تا j از آرایه برابر با صفر است. حال اگر این مقدار (j-i) بیشتر از max_len بود، مقدار max_len را آپدیت می کنیم، در غیر این صورت مقدار i را با کلید current_sum به hash_table اضافه می کنیم.

(hash_table[current_sum]=i)

با توجه به اینکه هزینه اضافه کردن و جستجو در جدول درهم سازی $O(1)$ است، بنابراین فقط یک پیمایش روی آرایه داریم که در نهایت هزینه زمانی ما $O(n)$ می شود.

پیاده سازی این الگوریتم به این صورت است:

```python
def find_largest_length_subarray_zero_sum(arr: list) -> int:
    hash_table = {}
    max_len = 0
    current_sum = 0

    for i in range(len(arr)):
        current_sum += arr[i]
        if current_sum == 0:
            max_len = i + 1
        if current_sum in hash_table:
            max_len = max(max_len, i - hash_table[current_sum])
        else:
            hash_table[current_sum] = i

    return max_len
```

.۴

مرتب سازی شمارشی، الگوریتمی است که از آن برای مرتب سازی عناصر یک آرایه با شمارش و ذخیره فرکانس هر عنصر در یک آرایه کمکی استفاده می شود. پیچیدگی زمانی این الگوریتم $O(n + k)$ می باشد که n تعداد عناصر آرایه می باشد و k رنج این عناصر می باشد.

برای مرتب سازی آرایه داده شده، ابتدا تعداد تکرار هر عنصر را پیدا می کنیم:

Input:

4	8	4	2	9	9	6	2	9

Counts:

0's	1's	2's	3's	4's	5's	6's	7's	8's	9's	10's
0	0	2	0	2	0	1	0	1	3	0

سپس جمع انباشته (cummulative) تعداد تکرار این عناصر را به دست می آوریم:

Cumm:

0	0	2	2	4	4	5	5	6	9	9

سپس با پیمایش روی آرایه input، با دیدن عنصر x آن را در ایندکس cumm[x]-1 از آرایه result قرار می دهیم و مقدار cumm[x] را یکی کم می کنیم. در نهایت آرایه مرتب شده ما به صورت زیر می شود:

2	2	4	4	6	8	9	9	9

۵. جلسه ها را بر اساس زمان پایان مرتب می کنیم و آنها را در یک آرایه ذخیره می کنیم. $O(nlogn)$ سپس این آرایه را پیمایش می کنیم، اولین جلسه را انتخاب می کنیم و سپس به ازای هر جلسه ای که می بینیم، اگر زمان شروع آن از آخرین جلسه ای که انتخاب کردیم بزرگ تر بود، آن را نیز انتخاب می کنیم و در غیر این صورت این روند را ادامه می دهیم. با این روش ما بیشترین تعداد جلسه را انتخاب خواهیم کرد.

هزینه زمانی این الگوریتم به اندازه یکبار سورت کردن ($O(nlogn)$) و یکبار پیمایش است($O(n)$) که در نهایت برابر با $O(nlogn)$ می شود.

۶. از ایده merge-sort استفاده می کنیم، به این صورت که به طور بازگشتی هر بار آرایه را به دو نیمه تقسیم می کنیم و جواب را برای نیمه چپ و راست محاسبه می کنیم. در اینجا جواب به صورت جمع جواب نیمه اول به اضافه جمع جواب نیمه دوم به اضافه تعداد وارونگی هایی است که در آن ها عددی در نیمه چپ بزرگتر از عددی در نیمه راست است. برای بدست آوردن جواب حالت سوم، متغیر های i و j را در نیمه های چپ و راست در نظر می گیریم.

اگر $arr[i] > arr[j]$ باشد، تعداد mid-1 تا وارونگی وجود دارد زیرا نیمه های چپ و راست مرتب شده اند و تمام عناصر نیمه چپ بزرگتر از $arr[j]$ هستند. این روند تقسیم آرایه را تا انتها ادامه می دهیم تا به حالت پایه برسیم.

هزینه نهایی این الگوریتم نیز همان هزینه merge-sort است. $O(nlogn)$