دانشگاه اصفهان
دانشکده علوم ریاضی و کامپیوتر

Design and Analysis of Algorithms

طراحی و تحلیل و الگوریتم ها

محمد ملائی

عنوان:

تمرینات ۱

نیم‌سال دوم ۱۴۰۲-۱۴۰۳

نام استاد درس

جعفر الماسی زاده

تمرین ۱

مساله کوله پشتی را درنظر بگیرید : n عنصر با وزن های معلوم $w_1, w_2, w_3, ...w_n$ و ارزش های معلوم $v_1, v_2, v_3, ...v_n$ و یک کوله پشتی با ظرفیت W، داده شده اند؛ با ارزش‌ترین زیرمجموعه‌ای از این عناصر را پیدا کنید که بتوان آن‌ها را درون کوله پشتی جا داد.

الف) با تحلیلی ریاضی، کارایی زمانی الگوریتم جستجوی کامل برای مسأله کوله‌پشتی را با نماد مجانبی Θ بیان کنید.

ب) برنامه‌ای برای پیاده‌سازی الگوریتم جستجوی کامل بنویسید. و بزرگترین مقداری از n را که به ازای آن، رایانه در کمتر از ۱ دقیقه قادر به اجرای برنامه باشد، پیدا کنید.

جواب

الف

برای حل این مسئله با روش جستجوی کامل باید تمام زیرمجموعه‌های مجموعه عناصر را تولید کنیم و شرط مسئله را در هریک از آن ها بررسی کنیم و بهترین حالت را پیدا کنیم. اگر تولید هر یک از زیرمجموعه‌های این مجموعه در زمان $O(1)$ اتفاق بیفتد و بررسی شرط مسئله در زمان ثابت انجام شود، کارایی زمانی الگوریتم $\Theta(2^n)$ خواهد بود. در زیر روشی برای تولید زیرمجموعه‌های یک مجموعه آورده شده است:

```python
def powerset(elements: list):
    p = [[]]
    for element in elements:
        for i in range(len(p)):
            p.append(p[i] + [element])
    return p
```

Listing 1: Python Powerset Implementation

بدین صورت که ابتدا با مجموعه که تنها مجموعه تهی را دارد شروع کرده و عضو اول مجموعه را به آن اضافه کرده و به زیرمجموعه‌های مجموعه این تک عضوی دست می‌یابیم، حال با اضافه کردن عضو بعدی به این مجموعه‌ها و اضافه کردن آن‌ها به لیست مجموعه‌ها، لیست زیرمجموعه‌های این دوعضو را داریم و با ادامه این روند، تمام زیر مجموعه های این مجموعه تولید خواهند شد.

ب

برای پیاده سازی الگوریتم جستجوی کامل برای این مساله، کافیست پس از اضافه کردن عناصر به محموعه powerset شرایط مسئله را روی آن‌ها بررسی کنیم و جواب مسئله را پیدا کنیم.

```python
def find_most_valuable_subset(elements, W):
    p = [[]]
    subset = []
    subset_value = 0
    for element in elements:
        for i in range(len(p)):
            new_subset = p[i] + [element]
            p.append(new_subset)

            if sum([x[1] for x in new_subset]) <= W:
                new_subset_value = sum([x[0] for x in new_subset])
                if new_subset_value > subset_value:
                    subset_value = new_subset_value
                    subset = new_subset

    return subset, subset_value
```

Listing 2: Python Knapsack Problem Implementation

در اینجا پس از ساخت هر زیرمجموعه، با بررسی شرایط مسئله، در صورت با ارزش تر بودن زیرمجموعه جدید آن را انتخاب کرده و در آخر بهترین زیرمجموعه به همراه ارزش آن بازگردانده می‌شود. (عناصر باید به صورت دوتایی های ارزش و وزن به تابع داده شوند).
این تابع با ارزش‌ترین زیرمجموعه مجموعه‌هایی که تعداد اعضای آن‌ها تا ۲۵ باشد را کمتر از یک دقیقه محاسبه می‌کند

با استفاده از توابع آماده در پایتون برای تولید زیرمجموعه‌ها که با زبان C نوشته شده‌اند می‌توان این عدد را تا ۲۶ یا ۲۷ نیز افزایش داد. البته این روش مزیت استفاده بهینه از حافظه را به علت استفاده از generator ها را نیز داراست :

```
from itertools import chain, combinations

def faster_powerset(elements):
    s = list(elements)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def faster_most_valuable_subset(elements, W):
    best_subset = []
    best_subset_value = 0
    for subset in faster_powerset(elements):
        if sum([x[1] for x in subset]) <= W:
            subset_value = sum([x[0] for x in subset])
            if subset_value > best_subset_value:
                best_subset_value = subset_value
                best_subset = subset
    return best_subset, best_subset_value
```

Listing 3: Python Faster Knapsack Problem Implementation

مراجع

python documentations – itertools

تمرین ۲

هر یک از این رابطه‌های بازگشتی را (که در تحلیل کارایی زمانی الگوریتم‌های بازگشتی پیش می‌آیند) با هر روشی که می‌دانید، حل کنید. تعیین مرتبه رشد توابع جواب کافی است.

الف)

$$T(n) = T(k) + T(n-k-1) + d \quad for \quad n > 0, \qquad T(0) = c$$

ب)

$$T(n) = n + max_{1 \le k \le n-1}[T(k) + T(n-k)], \quad for \quad n > 1, \qquad T(1) = 0$$

پ)

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$$

ت)

$$T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + n$$

جواب

الف

با محاسبه $T(n)$ برای n های کوچک، میتوانیم این رابطه بازگشتی را به صورت $T(n) = (n+1)c + nd$ حدس بزنیم:

$$T(0) = c \tag{۱}$$

$$\begin{aligned} T(1) &= T(k) + T(k) + d \rightarrow k = 0 \\ T(1) &= T(0) + T(0) + d = 2c + d \end{aligned} \tag{۲}$$

$$\begin{aligned} T(2) &= T(k) + T(1-k) + d \rightarrow k = 0, 1 \\ k = 0 : T(2) &= T(0) + T(1) + d = 3c + 2d \\ k = 1 : T(2) &= T(1) + T(0) + d = 3c + 2d \end{aligned} \tag{۳}$$

$$\begin{aligned} T(3) &= T(k) + T(2-k) + d \rightarrow k = 0, 1, 2 \\ k = 0 : T(3) &= T(0) + T(2) + d = 4c + 3d \\ k = 1 : T(3) &= T(1) + T(1) + d = 4c + 3d \\ k = 1 : T(3) &= T(2) + T(0) + d = 4c + 3d \end{aligned} \tag{۴}$$

از آنجایی که این رابطه برای $T(1)$ درست است آن‌را حالت پایه در نظر می‌گیریم و فرض می‌کنیم این رابطه برای $n-1$ رابطه قبلی درست باشد آنگاه نشان می‌دهیم که برای n نیز درست است:

$$T(n) = T(k) + T(n-1-k) + d; \quad k = 0, 1, \ldots, n-1 \tag{۵}$$

$$k = i < n : T(n) = T(i) + T(n-1-i) + d \tag{۶}$$

از آنجا که طبق فرض این رابطه برای اعداد کوچکتر از n درست است، داریم:

$$T(i) = (i+1)c + id \quad (۷)$$
$$T(n-1-i) = (n-1-i+1)c + (n-1-i)d \quad (۸)$$
$$T(i) + T(n-1-i) = ((n-1-i+1) + (i+1))c + ((n-1-i) + i)d \quad (۹)$$
$$T(i) + T(n-1-i) = (n+1)c + (n-1)d \quad (۱۰)$$
$$\Rightarrow T(n) = T(i) + T(n-1-i) + d = (n+1)c + nd \quad (۱۱)$$

بنابراین حکم ثابت شد و از آنجایی که c و d اعداد ثابت اند، داریم:

$$T(n) = (n+1)c + nd = (c+d)n + c \Rightarrow T(n) \in \theta(n)$$

ب

با محاسبه $T(n)$ برای n های کوچک در می‌یابیم که $T(n) = n + (n-1) + (n-2) + \cdots + 2$. برای اثبات این موضوع از استقرای ریاضی استفاده می‌کنیم:

$$T(2) = 2 + max\{T(1) + T(1)\} = 2$$

حالت پایه این تساوی برقرار است بنابراین فرض می‌کنیم این تساوی برای $k-1$ برقرار است آنگاه نشان می‌دهیم این تساوی برای k نیز برقرار است:

$$\begin{aligned} T(k) &= k + max\{T(i) + T(j) : i + j = k, i, j > 0\} \\ &= k + max\{T(1) + T(k-1), T(2) + T(k-2), \ldots, T(k-1) + T(1)\} \end{aligned} \quad (۱۲)$$

از آنجایی که $T(1) = 0$ کافی است نشان دهیم :

$$T(k-1) > T(i) + T(j);\ \ i + j = k\ \ \&\ \ i, j < k-1$$

$$\begin{aligned} T(k-1) &= (k-1) + (k-2) + \cdots + 2 \\ T(i) &= i + (i-1) + (i-2) + \cdots + 2 \\ T(j) &= j + (j-1) + (j-2) + \cdots + 2 \end{aligned} \quad (۱۳)$$

$$T(i) + T(j) = 2 \times (2 + 3 + \cdots + min(i,j)) + min(i,j) + 1 + \cdots + max(i,j) \quad (۱۴)$$

$$T(k-1) - [T(i) + T(j)] = (max(i,j) + 1) + \cdots + (k-1) - (2 + 3 + \cdots + min(i,j)) \quad (۱۵)$$

با توجه به اینکه $i + j = k$ بنابراین تعداد اعداد مثبت و منفی در این معادله برابر است و از آنجایی که هر عدد مثبت از هر عدد منفی بزرگتر است حکم ثابت شد و برای هر n بزرگتر از 1 داریم:

$$\begin{aligned} T(n) = n + T(n-1) = n + (n-1) + (n-2) + \cdots + 2 &= \frac{(n-1)(n+2)}{2} \\ &\to T(n) \in \theta(n^2) \end{aligned} \quad (۱۶)$$

پ

برای حل این بخش می‌توانیم از قاعده همواری استفاده کنیم. از آنجایی که در هر مرحله جذر n را محاسبه می‌کنیم و از آن کف می‌گیریم، می‌توانیم فرض کنیم که $n = 2^{2^k}$، در این صورت می‌توانیم این رابطه بازگشتی را ساده‌تر کنیم و آن‌را حل کنیم:

$$T(2^{2^k}) = 2T(2^{2^k-1}) + log(2^{2^k}) = 2T(2^{2^{k-1}}) + 2^k log(2) = 2T(2^{2^{k-1}}) + 2^k \quad (۱۷)$$

$$= 2 \times [2T(2^{2^{k-2}}) + 2^{k-1}] + 2^k = 2^2 T(2^{2^{k-2}}) + 2 \times 2^k \quad (۱۸)$$

$$= 2^3 T(2^{2^{k-3}}) + 3 \times 2^k = \cdots = 2^k T(2^{2^{k-k}}) + k \times 2^k \quad (۱۹)$$

$$= 2^k T(2) + k \times 2^k \quad (۲۰)$$

از طرفی داریم :

$$n = 2^{2^k} \rightarrow 2^k = log(n) \quad (۲۱)$$

$$\rightarrow k = log(log(n)) \quad (۲۲)$$

بنابراین:

$$T(n) = log(n) \times T(2) + log(log(n)) \times log(n) \quad (۲۳)$$

$$\rightarrow T(n) \in \theta(log(log(n)) \times log(n)) \quad (۲۴)$$

روش دوم می‌توانیم برای حل این سوال از قضیه اصلی نیز استفاده کنیم بدین صورت که با استفاده از تغییر متغیر، شکل رابطه بازگشتی را عوض کرده و آن را با قضیه اصلی حل می‌کنیم. برای این کار ابتدا فرض می‌کنیم $n = 2^k$ بنابراین داریم:

$$T(n) = T(2^k) = S(k) \quad (۲۵)$$

$$S(k) = 2S(k/2) + k \quad (۲۶)$$

$$\xrightarrow{master\ theorem} S(k) \in k \log(k) \quad (۲۷)$$

$$\rightarrow T(n) \in log(log(n)) \times log(n) \quad (۲۸)$$

ت

می‌توانیم با استفاده از درخت فراخوانی‌های بازگشتی، مرتبه رشد این تابع بازگشتی را مشخص کنیم. در ابتدا این مسئله به دوقسمت نامساوی با اندازه‌های یک سوم و دو سوم مسئله اصلی تقسیم شده‌اند. دو مرحله از این تابع بازگشتی به صورت زیر خواهد بود:

$$T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + n \quad (۲۹)$$

$$= T(\frac{n}{9}) + T(\frac{2n}{9}) + \frac{n}{3} + T(\frac{2n}{9}) + T(\frac{4n}{9}) + \frac{n}{3} + n \quad (۳۰)$$

همانطور که می‌بینیم در هر مرحله به اندازه تابع، n واحد اضافه می‌شود و این کار تا زمانی که تقسیم های ما به پایان برسد ادامه می‌یابد. از آنجایی که در این مسئله، تقسیم به زیر مسئله‌ها به صورت نامتوازن است، طبیعتا درخت متناظر آن نیز همینطور خواهد بود. اضافه شدن n تا زمانی که به انتهای کوچکترین مسیر برسیم ادامه خواهد یافت و این امر هنگامی که مسئله را به یک سوم مسئله اصلی تقسیم می‌کنیم اتفاق می‌افتد و انجام این کار $log_3(n)$ بار اتفاق می‌افتد. بنابراین اندازه تابع حداقل $nlog_3(n)$ خواهد بود و پس از آن مقادیری کمتر از آن به تابع اضافه می‌شوند بنابراین داریم:

$$T(n) \in \Omega(nlog_3(n))$$

مراجع

solving recursion - statckexchange

تمرین ۳

الف) فرض کنید A و B ، دو مجموعه ای باشند هر یک متشکل از n عدد صحیح که هر یک از آنها در محدوده 1 تا $2n$ واقع باشد. الگوریتمی را با کارایی زمانی O(n) توصیف کنید که که دو مجموعه A و B را به عنوان ورودی بگیرد و تعیین کند که آیا دو مجموعه A و B با هم برابر هستند یا خیر؛ یعنی آیا شامل عناصر کاملاً یکسانی (گرچه با ترتیب متفاوت) هستند یا خیر.

برنامه‌ای برای پیاده‌سازی الگوریتم بنویسید و با درج شمارنده‌ای (یا شمارنده‌هایی) در آن، تعداد عملیات پایه‌ای الگوریتم را به عنوان تابعی از اندازه دو مجموعه A و B محاسبه کنید. با هر یک از مقادیر $n = 10, 10^2, 10^3, 10^4, 10^5, 10^6$ آزمایش کنید.

ب) فرض کنید A و B ، دو مجموعه‌ای باشند هر یک متشکل از n عدد صحیح که هر یک از آنها در محدوده‌ی 1 تا n^4 واقع باشد. الگوریتمی را با کارایی زمانی $O(n)$ توصیف کنید که دو مجموعه A و B و عدد صحیح x را به عنوان ورودی بگیرد و تعیین کند که آیا عدد صحیح a در مجموعه A و عدد صحیح b در مجموعه B وجود دارند که که رابطه $a + b = x$ برقرار باشد یا خیر.

برنامه‌ای برای پیاده‌سازی الگوریتم بنویسید و با درج شمارنده‌ای (یا شمارنده‌هایی) در آن، تعداد عملیات پایه‌ای الگوریتم را به عنوان تابعی از اندازه دو مجموعه A و B محاسبه کنید. با هریک از مقادیر $n = 10, 10^2, 10^3, 10^4, 10^5, 10^6$ آزمایش کنید.

جواب

الف

از آنجایی که اعضای A در محدوده 1 تا $2n$ هستند، میتوان آرایه‌ای به انداز $2n$ ایجاد کرد و اعداد را براساس اندازه در این آرایه دخیره کرد. بدین صورت که اگر عنصر i ام این آرایه 1 باشد آنگاه این عدد در آرایه A وجود دارد. پس از ذخیره کردن A در این آرایه با بررسی اعضای B یکسان بودن یا نبودن این آرایه‌ها را بررسی می‌کنیم.

```python
def are_equal(A: list[int], B: list[int]):
    n = len(A)
    helper_list = [0] * (2 * n + 1)
    for number in A:
        helper_list[number] = 1
    for number in B:
        if helper_list[number] == 0:
            return False
    return True
```

در این تابع پس از ذخیره اطلاعات در `helper_list` اگر عددی در B وجود داشته باشد که مقدار آن در `helper_list` 1 نباشد آنگاه این دو آرایه یکسان نیستند و اگر چنین عددی وجود نداشت باهم برابرند.

اگر عملیات پایه‌ای را انتصاب در نظر بگیریم، آنگاه تعداد عملیات های پایه‌ای تنها به اندازه A وابسته خواهد بود و برابر اندازه A خواهد بود. اگر عملیات پایه‌ای را مقایسه در نظر بگیریم آنگاه تنها به اندازه B وابسته خواهد بود و در بدترین حالت برابر با اندازه B خواهد بود اما اگر A و B به صورت تصادفی انتخاب شده باشند به صورت میانگین بین ۱ تا ۳ مقایسه برای نشان دادن اینکه یکسان نیستند کافی‌است زیرا احتمال برابر بود آنها بسیار کم است و اگر عملیات پایه‌ای را دسترسی به عنصری در هریک از آرایه‌ها در نظر بگیریم، آنگاه تابع نشان‌دهنده تعداد عملیات های پایه‌ای به صورت $T(n) = n + k$ به طوری که k، تعداد مقایسه‌های ما خواهد بود که دسترسی به عناصر B را نشان می‌دهد: در زیر نمونه‌ای از خروجی برنامه برای مجموعه‌های تصادفی A و B آورده شده است:

```
    length of A and B is 10^1
    A and B are not equal
    a_access: 10, b_access: 2, assigns: 10
    helper_access: 2, comparisons: 2

    length of A and B is 10^2
    A and B are not equal
    a_access: 100, b_access: 1, assigns: 100
    helper_access: 1, comparisons: 1

    length of A and B is 10^3
    A and B are not equal
    a_access: 1000, b_access: 2, assigns: 1000
    helper_access: 2, comparisons: 2

    length of A and B is 10^4
    A and B are not equal
    a_access: 10000, b_access: 2, assigns: 10000
    helper_access: 2, comparisons: 2

    length of A and B is 10^5
    A and B are not equal
    a_access: 100000, b_access: 1, assigns: 100000
    helper_access: 1, comparisons: 1

    length of A and B is 10^6
    A and B are not equal
    a_access: 1000000, b_access: 1, assigns: 1000000
    helper_access: 1, comparisons: 1
```

ب

برای حل این بخش به دلیل زیادبودن فاصله بین کوچک‌ترین و بزرگ‌ترین عددهای لیست، استفاده از روش قسمت قبل کاربردی نخواهد بود و با استفاده از جدول درهم سازی نوشته شده در سوال ۴ می‌توانیم به صورت تقریبی به کارایی مورد نظر دست یابیم. کارایی زمانی دسترسی به عناصر با استفاده از جدول درهم‌سازی در بدترین حالت یعنی زمانی که تعداد زیادی از کلیدها hash یکسان داشته باشند می‌تواند $O(n)$ باشد اما به صورت میانگین از آنجایی که اندازه جدول چهار برابر تعداد عناصر انتخاب شده است، هزینه دسترسی به عناصر جدول $O(1)$ خواهد بود و می‌توانیم جواب قسمت ب را در $O(n)$ پیدا کنیم.

```python
def find_a_b(x: int, A: list[int], B: list[int]):
    n = len(A)
    hash_table = hash_table_module.HashedTable(4 * n)
    for number in A:
        hash_table.add(number)
    for number in B:
        if hash_table.search(x - number):
            return True
    return False
```

تمرین ۴

الف) با تحلیل ریاضی، مشخص شده است که اگر اندازه یک جدول درهم‌سازی بسته n باشد و اگر تعداد کلیدها m باشد و اگر $\alpha = \frac{n}{m}$ باشد، آنگاه میانگین تعداد مقایسه‌های لازم برای جستجوهای ناموفق (یا درج‌ها) در چنین جدولی با راهبرد کاوش خطی، تقریباً $\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$ خواهد بود.

برنامه‌ای بنویسید که با آن بتوان یک جدول درهم‌سازی بسته با اندازه n ساخت و سپس با راهبرد کاوش خطی $\frac{n}{2}$ عدد صحیح (کلید) تصادفی را در جدول درج کرد. سپس شمارنده‌ای (یا شمارنده‌هایی) را در برنامه درج کنید و با آن، میانگین تعداد مقایسه‌هایی را که برای جستجوهای ناموفق (درج‌ها) در جدول لازم است محاسبه کنید. آزمایش را با هر یک از مقادیر $n = 10, 10^2, 10^3, 10^4, 10^5, 10^6$ انجام دهید و نتایج تجربی را با نتایج نظری مقایسه کنید.

ب) این فرضیه مطرح شده است که تعداد مقایسه‌های لازم برای درج m کلید تصادفی با راهبرد کاوش خطی در یک جدول درهم‌سازی بسته با اندازه m ، تقریباً برابر با $\sqrt{\frac{\pi}{2}}(m^{\frac{3}{2}})$ است. با نوشتن برنامه‌ای، درستی این ادعا را تحقیق کنید.

جواب

برای ساختن جدول درهم سازی بسته در python می توان از class استفاده کنیم. در زیر نحوه ساخت این کلاس آمده است.

```python
class HashedTable:
    def __init__(self, size: int) -> None:
        self.size = size
        self.table = [None] * size
        self.filled_cells = 0
        self.insertion_counter = 0
        self.search_counter = 0
```

در تابع innit که در زمان ساخت یک شی از این کلاس صدا زده خواهد شد مقادیر اولیه این جدول را مانند اندازه آن، یک آرایه برای ذخیره اطلاعات و تعداد خانه‌های پر را ذخیره می‌کنیم.

```python
class HashedTable:
    ...
    def add(self, value: str):
        if self.filled_cells >= self.size:
        raise Exception("HashTable is full")
        self.table[self.get_insertion_index(value)] = value
        self.filled_cells += 1
```

تابع بعدی، اضافه کردن یک مقدار به این جدول را انجام می‌دهد البته پس از بررسی پر نبودن جدول. تابع `get_insertion_index` پس از این تابع تعریف خواهد شد و اندیس خانه‌ای که می‌توانیم مقدار جدید را در آن ذخیره کنیم به ما خواهد داد.

```python
class HashedTable:
    ...
    def get_insertion_index(self, value: int):
        hash_index = self.hash(value)
        start_index = -(len(self.table) - hash_index)
        for i in range(start_index, hash_index):
            if self.table[i] is None or self.table[i] is False:
                self.insertion_counter += 2 * abs(start_index - i)
                return i
```

این تابع همانطور که گفته شد، اندیس خانه‌ای که این کلید می‌تواند در آن ذخیره شود را به ما خواهد داد. این کار با یک حلقه و بررسی خانه‌ها، به ترتیب و از خانه‌ای شروع خواهد شد که تابع hash به ما داده است، زیرا این اولین گزینه ما برای ذخیره این کلید است و اگر

پر باشد، خانه‌های پس از آن بررسی خواهند شد. خانه‌هایی که از جدول حذف شده‌اند با $False$ نشانه گذاری شده‌اند پس ما آن‌ها را نیز خالی در نظر می‌گیریم زیرا این نشانه‌گذاری برای جستجو است، نه اضافه کردن.

```python
class HashedTable:
    ...
    def hash(self, value: int):
        return value % self.size
```

برای hash کردن یا درهم سازی نیز از باقیمانده تقسیم عدد بر اندازه جدول استفاده می‌کنیم.

```python
class HashedTable:
    ...
    def search(self, value: int):
        hash_index = self.hash(value)
        start_index = -(len(self.table) - hash_index)
        for i in range(start_index, hash_index):
            if self.table[i] is None:
                self.search_counter += abs(start_index - i) + 1
                return False
            elif self.table[i] == value:
                if i >= 0:
                    return i
                else:
                    return len(self.table) + i
        self.search_counter += len(self.table)
        return False
```

برای جستجو در این جدول از گزینه اولمان که از تابع hash گرفته‌ایم شروع کرده و خانه‌ها را به ترتیب بررسی می‌کنیم. اگر مقدار خانه None باشد یعنی در این خانه هرگز مقداری ذخیره نشده‌است و نتیجه جستجو منفی خواهد بود. اگر مقدار خانه برابر False باشد یعنی مقدار این خانه قبلا حذف شده است و به جستجو ادامه می‌دهیم و اگر مقدار خانه برابر کلید جستجو باشد، جستجو موفقیت آمیز بوده‌است و اندیس خانه به عنوان نتیجه جستجو بازگردانده خواهد شد. و اگر جستجو بی‌نتیجه باشد False بازگردانده خواهد شد.

```python
class HashedTable:
    ...
    def delete(self, value: str):
        index = self.search(value)
        if index:
            self.table[index] = False
            self.filled_cells -= 1
```

و به عنوان آخرین تابع این کلاس، برای حذف کردن مقادیر ابتدا آن را جستجو کرده و اگر این جستجو موفقیت آمیز بود، آن را با قراردادن False به جای مقدار اصلی حذف کرده و یک واحد از خانه‌های اشغال شده کم می‌کنیم.

حال با اضافه کردن همه‌ی توابع کلاس HashedTable به صورت زیر خواهد بود:

```python
class HashedTable:
    def __init__(self, size: int) -> None:
        self.size = size
        self.table = [None] * size
        self.filled_cells = 0
        self.insertion_counter = 0
        self.search_counter = 0

    def add(self, value: int):
        if self.filled_cells >= self.size:
            raise Exception("HashTable is full")
        self.table[self.get_insertion_index(value)] = value
        self.filled_cells += 1

    def get_insertion_index(self, value: int):
        hash_index = self.hash(value)
        start_index = -(len(self.table) - hash_index)
```

```python
        for i in range(start_index, hash_index):
            if self.table[i] is None or self.table[i] is False:
                self.insertion_counter += 2 * abs(start_index - i)
                return i

    def hash(self, value: int):
        return value % self.size

    def search(self, value: int):
        hash_index = self.hash(value)
        start_index = -(len(self.table) - hash_index)
        for i in range(start_index, hash_index):
            if self.table[i] is None:
                self.search_counter += abs(start_index - i) + 1
                return False
            elif self.table[i] == value:
                if i >= 0:
                    return i
                else:
                    return len(self.table) + i
        self.search_counter += len(self.table)
        return False

    def delete(self, value: str):
        index = self.search(value)
        if index:
            self.table[index] = False
            self.filled_cells -= 1
```

Listing 4: HashedTable Implementation

با بررسی تعداد مقایسه‌های لازم برای جستجوی ناموفق و درج m عنصر در جدول درهم سازی به نتایج زیر دست می‌یابیم که نشان می‌دهد نتایج تجربی، نتایج نظری بخش‌های الف و ب را تایید می‌کند:

```
Testing unseccussfull search comparisons ...

>> n is 10^1 and alpha is 0.5:
average uncessufull search comparisons is: 1.6
in theory the average for search should be: 2.5
>> n is 10^2 and alpha is 0.5:
average uncessufull search comparisons is: 2.12
in theory the average for search should be: 2.5
>> n is 10^3 and alpha is 0.5:
average uncessufull search comparisons is: 2.405
in theory the average for search should be: 2.5
>> n is 10^4 and alpha is 0.5:
average uncessufull search comparisons is: 2.4926
in theory the average for search should be: 2.5
>> n is 10^5 and alpha is 0.5:
average uncessufull search comparisons is: 2.48968
in theory the average for search should be: 2.5
>> n is 10^6 and alpha is 0.5:
average uncessufull search comparisons is: 2.497765
in theory the average for search should be: 2.5


Testing unseccussfull add comparisons ...

>> n is 10^1:
uncessufull add comparisons are 12
in theory it should be: 39.633272976060105
>> n is 10^2:
uncessufull add comparisons are 820
```

```
in theory it should be: 1253.3141373155001
>> n is 10^3:
uncessufull add comparisons are 52682
in theory it should be: 39633.27297606011
>> n is 10^4:
uncessufull add comparisons are 2145782
in theory it should be: 1253314.1373155
>> n is 10^5:
uncessufull add comparisons are 35076828
in theory it should be: 39633272.97606011
>> n is 10^6:
uncessufull add comparisons are 1190299018
in theory it should be: 1253314137.3155
```

تمرین ۵

فرض کنید A مجموعه‌ای بزرگ از n عنصر باشد؛ آنقدر بزرگ که نتوان آن را به طور کامل در حافظه اصلی نگهداری کرد و لازم باشد که آن را روی دیسک نگهداری کرد. یک راه برای مرتب کردن چنین مجموعه‌ای، طراحی گونه‌ای از الگوریتم مرتبسازی ادغامی است: مجموعه A را به k مجموعه کوچک‌تر $A_1, A_2, \ldots, A_k$ تقسیم کنید، هر یک از آن مجموعه‌ها را به طور بازگشتی مرتب کنید و سپس همه مجموعه‌های مرتب را با هم ادغام کنید تا مجموعه مرتب اصلی تشکیل شود.

الف) فرض کنید اندازه هر صفحه دیسک b باشد و اندازه حافظه اصلی m باشد. با یک مثال توضیح دهید که مقدار k (برحسب b و m) چند باشد تا بتوان با $O(\frac{n}{b})$ عملیات انتقال صفحه (خواندن یک صفحه از دیسک یا نوشتن یک صفحه روی دیسک) ، k مجموعه مرتب را با هم ادغام کرد.

ب) با تشکیل و حل یک رابطه بازگشتی برای تعداد عملیات انتقال صفحه، ثابت کنید که الگوریتم مرتبسازی ادغامی مجموعه A را با $O((\frac{n}{b})\frac{\log(\frac{n}{b})}{\log(\frac{m}{b})})$ عملیات انتقال صفحه مرتب میکند.

جواب

الف

پس از مرتب کردن همه‌ی زیر آرایه‌های آرایه A برای ادغام آن ها و ساخت زیرآرایه بزرگتر می‌توانیم از الگوریتم k_way_merge استفاده کنیم در این الگوریتم هر صفحه اشغال شده توسط این آرایه را یکبار برای خواندن از دیسک می‌گیریم و برای به اندازه‌ی تعداد صفحات اشغال شده نیز برای نوشتن به دیسک می‌فرستیم بنابراین در هر مرحله ادغام $\frac{2n}{b}$ عملیات انتقال صفحه خواهیم داشت. برای اینکه با $O(\frac{n}{b})$ عملیات انتقال صفحه این آرایه را مرتب کنیم، باید تعداد مراحل اندازه‌ای ثابت و یا در حالت ایده‌ال یک مرحله باشند. با در نظر گرفتن این موضوع که اندازه هر زیرآرایه باید از اندازه حافظه اصلی کوچکتر باشد، بنابراین $k > \frac{n}{m}$ خواهد بود. از طرفی برای اینکه بتوانیم k زیرآرایه را در یک مرحله ادغام کنیم، باید بتوانیم حداقل یک صفحه از هر یک آز آن‌ها را در هر لحظه در حافظه اصلی داشته باشیم و این به این معناست که $k < \frac{m}{b}$. می‌توانیم مقدار k را $\frac{m}{b} - 1$ در نظر بگیریم تا حداکثر تعداد زیرآرایه را در هر مرحله باهم ادغام کنیم.

ب

اگر تعداد صفحاتی که در هرلحظه می‌توانیم برای ادغام کردن در حافظه اصلی داشته باشیم، B باشد، از آنجایی که در هر مرحله مسئله به B قسمت تقسیم می‌شود که با $\frac{2n}{b}$ عملیات انتقال صفحه ادغام می‌شوند، رابطه بازگشتی به صورت زیر خواهد بود:

$$T(n) = BT(n/B) + 2n/b$$

بااستفاده از قضیه اصلی و درخت فراخوانی های بازگشتی این مسئله داریم:

$$T(n) \in O((\frac{n}{b})\log_B(\frac{n}{b}))$$
$$\Rightarrow T(n) \in O((\frac{n}{b})\frac{\log(\frac{n}{b})}{\log(B)})$$

اگر بخواهیم کارایی الگوریتم حداکثر شود B را همان $\frac{m}{b} - 1$ در نظر می‌گیریم و از آنجایی که $B \in O(m/b)$ داریم:

$$T(n) \in O(\frac{n}{b}\frac{\log(\frac{n}{b})}{\log(\frac{m}{b})})$$

مراجع

-stackoverflow- -opendsa- -youtube-