



دانشگاه اصفهان
دانشکده علوم ریاضی و کامپیوتر

Data Structures and Algorithms

ساختمان داده ها و الگوریتم ها

عنوان:

تمرینات ۴

اعضای گروه

محمد ملائی

محسن محمودآبادی

داوود نصرتی امیرآبادی

نیم سال اول ۱۴۰۲-۱۴۰۳

نام استاد درس

جعفر الماسی زاده

تمرین ۱

فرض کنید H_0, H_1, H_2, \dots دنباله‌ای نامتناهی از ماتریس‌ها باشد. در این دنباله، $H_0 = 1$ است و برای هر $k > 0$ ماتریسی است $2^k \times 2^k$ با این تعریف بازگشتی:

$$H = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}$$

اگر v برداری ستونی با طول $n = 2^k$ باشد، الگوریتمی با کارایی $O(n \log n)$ برای محاسبه‌ی حاصل ضرب ماتریس در بردار $H_k v$ ارائه کنید. فرض کنید که همه اعداد آنقدر کوچک باشند که عملیات حسابی جمع و ضرب روی آنها در زمان ثابت قابل انجام باشد.

جواب

برای ضرب ماتریس H_k در بردار ستونی V باید درای‌های هر سطر ماتریس را در تمام درایه‌های بردار ضرب کرده و با هم جمع کنیم تا درایه‌های ماتریس حاصل ضرب را به دست آوریم. در این مرحله واضح است که درایه‌های ستون اول ماتریس تنها در درایه اول بردار ضرب خواهند شد و درایه‌های ستون دوم ماتریس در درایه دوم بردار ضرب خواهند شد و این روند برای تمام ستون‌های ماتریس تکرار خواهد شد، از این نکته برای حل سوال به شیوه تقسیم و حل استفاده خواهیم کرد.

برای بردار V داریم: $V = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$. نیمه بالایی بردار V را به صورت $V_t = \begin{bmatrix} v_1 \\ \vdots \\ v_{n/2} \end{bmatrix}$ و نیمه پایینی این بردار را به صورت $V_b = \begin{bmatrix} v_{n/2+1} \\ \vdots \\ v_n \end{bmatrix}$ تعریف می‌کنیم. ماتریس H_k به صورت بازگشتی و از روی H_{k-1} تعریف شده است که ماتریس را به چهار

بلوک مشابه تبدیل می‌کند. برای ضرب ماتریس در بردار V کافی است نیمه بالایی ماتریس را در بردار ضرب کرده و در نیمه بالایی بردار حاصل ضرب قرار دهیم و این کار را برای نیمه پایینی نیز تکرار کنیم. با توجه به توضیحات داریم:

$$H_k V = \begin{bmatrix} H_{k-1} V_t + H_{k-1} V_b \\ H_{k-1} V_t - H_{k-1} V_b \end{bmatrix}$$

بدین ترتیب مسئله به دو مسئله کوچکتر که $H_{k-1} V_t$ و $H_{k-1} V_b$ هستند تبدیل خواهد شد و پس از پیدا کردن جواب آنها، با جمع ماتریس‌ها، جواب کلی را پیدا می‌کنیم.

HadamardTimesV(V, k)

Input: Vector V which will be multiplied with the corresponding Hadamard matrix

Output: Vector $M = H_k V$

```

if k = 0 then
    return V
else
     $M_1 = \text{HadamardTimesV}(V_t, k - 1)$ 
     $M_2 = \text{HadamardTimesV}(V_b, k - 1)$ 
    return  $\begin{bmatrix} M_1 + M_2 \\ M_1 - M_2 \end{bmatrix}$ 
```

```

1  def add_vectors(v1, v2, coefficient=1):
2      return [v1[i] + (coefficient) * v2[i] for i in range(len(v1))]
3
4
5  def hadamard_times_v(v):
6      n = len(v)
7      if n == 1:
8          return v
9      else:
10         first_half = hadamard_times_v(v[: n // 2])
11         second_half = hadamard_times_v(v[(n - 1) // 2 + 1 :])
12         top = add_vectors(first_half, second_half)
13         bottom = add_vectors(first_half, second_half, coefficient=-1)
14         return top + bottom
15

```

Listing 1: Python Implementation

در روند تقسیم و حل، ماتریس H_k با اندازه 2^k به دو ماتریس کوچکتر H_{k-1} با اندازه های 2^{k-1} تقسیم می شود و برای ترکیب جواب ها $2n$ جمع خواهیم داشت، بنابر این اگر $T(n)$ تابعی باشد که زمان اجرای این الگوریتم را نشان می دهد خواهیم داشت:

$$T(n) = 2T(n/2) + 2n$$

طبق قضیه اصلی، زمان اجرای الگوریتم $\theta(n \log(n))$ خواهد بود.

مراجع

تمرین ۲

الف) الگوریتمی بنویسید که به عنوان ورودی، اشاره‌گری به ریشه درخت دودویی T را بگیرد و تعداد گره‌های T را به عنوان خروجی چاپ کند. کارایی زمان الگوریتم خود را تعیین کنید.

ب) الگوریتمی بنویسید که به عنوان ورودی، اشاره‌گری به ریشه درخت دودویی T را بگیرد و عمق هر یک از گره‌های T را به عنوان خروجی چاپ کند. (عمق ریشه را صفر بگیرید.) کارای زمان الگوریتم خود را تعیین کنید.

جواب

الف

برای پیدا کردن تعداد گره‌های T می‌توانیم به صورت بازگشتی تعداد گره‌های فرزندان چپ و راست گره را پیدا کرده و در صورت پر بودن گره ریشه آن را یک واحد اضافه می‌کنیم تا تعداد همه ی گره‌های T را به دست آوریم. (طبق تعریف گره پر یا دو فرزند دارد یا فرزند ندارد)

$\text{CountFullNodes}(P)$

Input: Pointer p to tree T

Output: Count of full nodes in the tree

```
get tree T pointed by P
count = 0
if T is empty then
    return 1
else
    countLeft = CountFullNodes( $T_{left}$ )
    countRight = CountFullNodes( $T_{right}$ )
    if T has left and right child then
        count += 1
    count = count + countLeft + countRight
    return count
```

```
1 def count_full_nodes(root):
2     count = 0
3
4     if not root.left and not root.right:
5         return 1
6
7     if root.left:
8         count += count_full_nodes(root.left)
9     if root.right:
10        count += count_full_nodes(root.right)
11
12    if root.left and root.right:
13        count += 1
14
15    return count
```

Listing 2: Python Implementation

الگوریتم، مسئله را به دو مسئله کوچکتر تبدیل می‌کند و پس از حل آنها با انجام یک مقایسه و جمع آنها را باهم ترکیب می‌کند بنابراین اگر $T(n)$ نشان دهنده زمان اجرای الگوریتم باشد داریم:

$$T(n) = 2T(n/2) + \theta(1)$$

طبق قضیه اصلی، زمان اجرای الگوریتم $\theta(n)$ خواهد بود.

ب

برای حل این مسئله می توانیم از الگوریتم جستجوی سطح-ترتیب استفاده کنیم و گره ها را به ترتیب در هر سطح چاپ کنیم. برای این کار از صف استفاده می کنیم که اولین عضو آن ریشه با عمق صفر است. درون حلقه فرزندان ریشه را به صف اضافه کرده و آن ها عمق یک را نسبت می دهیم تا در تکرار های بعدی همراه با عمقشان چاپ شوند. این روند را برای هر گره تکرار می کنیم تا به برگ ها برسیم و پیمایش سطح ترتیب به اتمام برسد.

PrintLevelOrder(*root*)

Input: Root of tree T

Output: Prints all nodes and their depth

```
queue = []
root.depth = 0
append root to the queue
while queue is not empty do
    node = queue.pop()
    print node value and its depth
    if node has left child then
        node.left.depth = node.depth + 1
        append node left child to queue
    if node has right child then
        node.right.depth = node.depth + 1
        append node right child to queue
```

```
1 def print_depths(root):
2     queue = []
3     queue.append((root, 0))
4
5     while len(queue) > 0:
6         (node, depth) = queue.pop(0)
7         print(node.data, depth)
8         if node.right:
9             queue.append((node.right, depth + 1))
10        if node.left:
11            queue.append((node.left, depth + 1))
12
```

Listing 3: Python Implementation

این الگوریتم تمام گره های درخت را یک بار و فقط یک بار بررسی و چاپ می کند بنابراین کارایی زمانی آن $\theta(n)$ خواهد بود.

مراجع

- Binary Tree Level Order Traversal - Medium -

تمرین ۳

Sort($A[i \dots j]$)

Input: Subarray of array $A[0 \dots n-1]$, defined by its indices i^j

Output: Subarray $A[i \dots j]$ sorted in non-decreasing order

```

n = j - i + 1
if n = 2 and A[i] > A[j] then
    swap(A[i], A[j])
else if n > 2 then
    m = ⌊n/3⌋
    Sort(A[i ... j - m])
    Sort(A[i + m ... j])
    Sort(A[i ... j - m])

```

(الف) ثابت کنید که این الگوریتم، به درستی آرایه ورودی خود، $A[0 \dots n-1]$ را مرتب می‌کند.
 (ب) با تشکیل و حل یک رابطه بازگشتی برای شمارش تعداد عملیات مقایسه الگوریتم، کارایی زمانی آن را تعیین کنید.
 (پ) ثابت کنید که تعداد عملیات جابه‌جایی الگوریتم حداکثر $\frac{n(n-1)}{2}$ است.

جواب

الف

حالت پایه الگوریتم، آرایه ای به طول ۲ است که به درستی مرتب می‌شود. بنابر استقرا ریاضی فرض می‌کنیم که الگوریتم آرایه ای به طول $n-1$ را به درستی مرتب می‌کند. نشان می‌دهیم که الگوریتم آرایه به طول n را نیز مرتب خواهد کرد.
 آرایه ای به طول n مانند $A = [a_0, a_1, \dots, a_{n-1}]$ را در نظر می‌گیریم و با استفاده از الگوریتم دو سوم ابتدایی آرایه را مرتب می‌کنیم و داریم:

$$m = \lfloor \frac{n}{3} \rfloor \rightarrow A = [a_0, \dots, a_{n-m}, \dots, a_n]; a_0 < \dots < a_m < \dots < a_{n-m}$$

در مرحله ی بعدی دو سوم پایانی آرایه مرحله قبل را مرتب می‌کنیم و داریم :

$$m = \lfloor \frac{n}{3} \rfloor \rightarrow A = [a_0, \dots, a_m, a'_{m+1}, \dots, a'_n]; a'_{m+1} < \dots < a'_{n-m} < \dots < a'_n$$

در این مرحله اطمینان داریم که یک سوم پایانی این آرایه در جای درست خود قرار گرفته. برای اثبات این گزاره از برهان خلف استفاده می‌کنیم و فرض می‌کنیم که در یک سوم ابتدایی عددی وجود دارد که حداقل از یکی از اعداد یک سوم پایانی بزرگ تر است (از اعداد ۱ و ۳ برای نشان دادن یک سوم ابتدایی و پایانی آرایه به دست آمده در مرحله دوم استفاده شده است):

$$\exists x \in 1 \quad \exists y \in 3 : x > y \xrightarrow{x \leq a_m} \exists y \in 3 : a_m > y \xrightarrow{a_m < a_{m+1}} \exists y \in 3 : y < a_{m+1} < \dots < a_{n-m} \rightarrow y \notin 3$$

نتیجه می‌گیریم y از یک سوم میانی آرایه پس از مرحله اول کوچکتر است و پس از مرحله دوم باید قبل از همه آن‌ها قرار بگیرد پس y نمی‌تواند بعد از مرحله دوم در یک سوم پایانی باشد که این تناقض است بنابر این حکم درست است و اعداد در یک سوم پایانی به درستی در جای خود قرار گرفته‌اند.

در مرحله سوم دوباره دو سوم ابتدایی آرایه را مرتب می‌کنیم و این باعث مرتب شدن تمام آرایه خواهد شد. بنابر استقرا درستی این الگوریتم برای آرایه‌های به طول n که n عددی صحیح و مثبت است اثبات شد.

ب

برای حل، این مسئله به سه مسئله کوچکتر با اندازه های $2/3$ مسئله اصلی تبدیل می شود که پس از حل شدن آن ها مسئله اصلی بدون نیاز به ترکیب جواب ها حل می شود. بنابراین اگر $T(n)$ را زمان اجرای الگوریتم در نظر بگیریم، خواهیم داشت :

$$T(n) = 3T(2/3n) + \theta(1)$$

طبق قضیه اصلی کارایی الگوریتم $\theta(n^{\log_{1.5} 3}) = \theta(n^{2.7})$ است.

پ

بدترین حالت در این الگوریتم زمانی اتفاق می افتد که آرایه نزولی باشد. در مرحله دوم همین اتفاق برای عناصر یک سوم میانی و یک سوم پایانی رخ خواهد داد و در آخر همین اتفاق برای یک سوم ابتدایی و میانی می افتد. بدین ترتیب هر دو عنصر ای آرایه دقیقا یکبار باهم جابجا می شوند که داریم :

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

بنابراین حداکثر تعداد جابجایی های الگوریتم $\frac{n(n-1)}{2}$ خواهد بود.

مراجع

تمرین ۴

فرض کنید یک بازی رایانه‌ای به نام **شکرستان** تولید شده است که بازیکن آن در دنیایی سه بعدی حرکت می‌کند. مکان‌های بازیکن در آن دنیای سه بعدی را خانه‌های آرایه سه بعدی C با ابعاد $n \times n \times n$ مشخص می‌کنند و علاوه بر آن، مقدار هر خانه $C[i, j, k]$ ، تعداد امتیازاتی را که بازیکن شکرستان با بودن در مکان (i, j, k) به دست می‌آورد، مشخص می‌کند. در شروع بازی، تنها $O(n)$ خانه از خانه‌های آرایه C غیر صفر است؛ همه $O(n^3)$ خانه دیگر آن صفر هستند. اگر بازیکن در طول بازی، به مکان (i, j, k) برود و $C[i, j, k]$ غیر صفر باشد، آنگاه به مقدار $C[i, j, k]$ ، امتیاز به عنوان جایزه به او داده می‌شود و سپس مقدار $C[i, j, k]$ صفر می‌شود. در ادامه، رایانه مکان دیگری مثل (i, j, k) را تصادفاً انتخاب می‌کند و ۱۰۰ امتیاز را به مقدار $C[i, j, k]$ اضافه می‌کند.

مسئله این است که بازی شکرستان، برای اجرا روی رایانه‌ای بزرگ طراحی شده است و چون حالا قرار است که آن را برای اجرا روی تلفن هوشمند (که حافظه بسیار کمتری دارد) سازگار کنیم، شما نمیتوانید مانند نسخه رایانه‌ای بازی، از $O(n^3)$ خانه حافظه برای نمایش آرایه C در حافظه تلفن استفاده کنید.

توضیح دهید که چگونه می‌توانیم محتویات آرایه C را با استفاده از تنها $O(n)$ واحد حافظه نمایش دهیم و اینکه چگونه می‌توانیم با الگوریتم‌هایی با کارایی $O(\log n)$

- مقدار هر خانه $C[i, j, k]$ را تعیین کنیم؛
- مقدار هر خانه غیر صفر $C[i, j, k]$ را صفر کنیم؛
- و ۱۰۰ امتیاز را به مقدار هر خانه $C[i, j, k]$ اضافه کنیم.

جواب

از آنجایی که در هر لحظه $O(n)$ خانه در این بازی پر شده اند، تنها خانه‌هایی که مقدار آن‌ها غیر صفر است را ذخیره می‌کنیم و با این کار می‌توانیم دنیای بازی را در $O(n)$ واحد حافظه ذخیره کنیم. با توجه به اینکه برای عملیات‌های این بازی به ساختار و الگوریتم‌های کارا نیازمندیم، می‌توانیم از درخت‌های خود متوازن استفاده کنیم که در این سوال، استفاده از درخت AVL با توجه به شرایط مسئله مناسب است. در این ساختار با توجه به متوازن بودن آن، کارایی الگوریتم‌های جستجو، حذف و اضافه کردن، $O(\log n)$ خواهد بود.

برای ذخیره سازی این دنیای سه بعدی که از مولفه‌های i, j, k تشکیل شده است، کافی است به هر یک از این خانه‌ها یک عدد یکتا نسبت دهیم تا بتوانیم با استفاده از آن، درخت AVL مورد بحث را تشکیل دهیم. اگر فرایند اندیس گذاری را از خانه $(1, 1, 1)$ شروع کنیم و در راستای محور x حرکت کنیم، خواهیم داشت:

$$index(1, 1, 1) = 1, index(2, 1, 1) = 2, \dots, index(n, 1, 1) = n$$

سپس یک واحد به مولفه j اضافه کرده و این روند را تکرار می‌کنیم. برای خانه‌هایی که مولفه j آنها ۲ است داریم:

$$index(1, 2, 1) = 1 + n, index(2, 2, 2) = 2 + n, \dots, index(n, 2, 2) = n + n$$

با تکرار این روند و پوشش کامل خانه‌هایی که مولفه k آنها ۱ و تعداد آنها n^2 است، k را یک واحد افزایش داده و این روند را تا انتها تکرار می‌کنیم. بنابراین اندیس هر خانه را می‌توانیم به صورت تابعی از مولفه‌ها بنویسیم:

$$index(i, j, k) = i + n(j - 1) + n^2(k - 1)$$

بدین ترتیب با مجموعه اندیس و مقدار خانه‌های غیر صفر درخت دودویی مورد نظر را می‌سازیم که عملیات‌های آن به صورت زیر تعریف شده اند:

- تعیین مقدار هر خانه: این کار را با جستجو در درخت انجام می‌دهیم و در صورت ناموفق بودن جستجو مقدار آن خانه را صفر برمی‌گردانیم
- صفر کردن مقدار خانه: با حذف کردن گره متناظر در درخت، مقدار خانه را صفر می‌کنیم.
- اضافه کردن امتیاز: ابتدا اندیس خانه را جستجو کرده و در صورت موفق بودن، مقدار آن را ۱۰۰ واحد افزایش می‌دهیم در غیر این صورت یک گره به درخت اضافه می‌کنیم.

مراجع

تمرین ۵

گاهی با داده‌هایی عددی کار می‌کنیم و ممکن است که تعیین میانه آن مجموعه اعداد، پیوسته در طول زمان لازم باشد. فرض کنید S مجموعه‌ای از اعداد باشد که در ابتدا تهی باشد و ممکن است لازم باشد که در طول زمان بارها با عملیات $insert(S, x)$ عدد x در آن درج شود یا با عملیات $median(S)$ میانه آن تعیین شود. روشی را پیشنهاد کنید که اگر n عدد در مجموعه S باشد، بتوان مبتنی بر آن، الگوریتمی با کارایی $O(\log n)$ برای انجام عملیات $insert(S, x)$ و الگوریتمی هم با کارایی $O(\log n)$ برای انجام عملیات $median(S)$ طراحی کرد.

جواب

برای حل این سوال به یک درخت AVL نیاز خواهیم داشت با این تفاوت که هر گره، علاوه بر داشتن اشاره‌گرهایی به گره پدر و گره فرزند راست و گره فرزند چپ، مقداری تحت عنوان $Size$ را نیز نگهداری می‌کند. $Size$ گره c تعداد تمام گره‌هایی است که از c منشعب می‌شوند. برای مثال $Size$ ریشه درخت، برابر با یک واحد کمتر از کل تعداد گره‌هاست. از طرفی، فرض کنید دنبال k امین کوچکترین عنصر در درخت هستیم. می‌دانیم تمام عناصر کوچکتر از ریشه، در سمت چپ آن قرار دارند. به عبارت دیگر، اگر $Size$ زیردرخت چپ برابر با $k - 1$ باشد (یعنی $k - 1$ عنصر در زیردرخت چپ قرار دارند) و به دنبال k امین کوچکترین عنصر باشیم، پاسخ همان ریشه است. چرا که $k - 1$ عنصر کوچکتر از ریشه وجود دارند و خود ریشه k امین عنصر است. حال اگر کلید k کوچکتر از $Size$ زیردرخت چپ باشد، می‌توان نتیجه گرفت که عنصر مورد نظر ما جایی در زیردرخت چپ قرار دارد. در غیر این صورت، اگر k از $Size$ زیردرخت چپ بیشتر باشد، باید در زیردرخت راست دنبال آن بگردیم. میانه‌ی یک لیست که تعداد عناصر آن فرد باشد، اندیس $\lceil \frac{N+1}{2} \rceil$ ام آن است. در صورتی که تعداد عناصر آن زوج باشد، میانگین دو عنصر $\lceil \frac{N}{2} \rceil$ ام و $\lceil \frac{N+2}{2} \rceil$ ام آن است. پس با استفاده از الگوریتمی که در بالا شرح داده شد، به یافتن عناصر مورد نظر می‌پردازیم و میانه را حساب می‌کنیم.

Find_Kth($T, K, root$)

Input: T : A tree, K : The K _th element to be found, $root$: The root to begin with

Output: The K _th smallest element in T

```
size = root.left.size + 1
if k == size then
    return root.value
else if k < size then
    return Find_Kth(T, K, root.left)
else
    return Find_Kth(T, K - size, root.right)
```

Find_Median(T)

Input: T : The AVL tree

Output: Median in T

```
size = T.size
if size%2 == 1 then
    return T.Find_Kth((T, size + 1, T.root) // 2)
first_med = T.Find_Kth(T, size // 2, T.root)
second_med = T.Find_Kth(T, (size + 2) // 2, T.root)
return (first_med + second_med) / 2
```

با توجه به اینکه درختی که استفاده کردیم از نوع AVL بوده است، کارایی زمانی افزودن یک مقدار به آن از مرتبه $O(\log n)$ خواهد بود. بعلاوه، همانطور که در الگوریتم $Find_Median$ مشاهده کردیم، هر بار درخت را به دو نیم تقسیم کرده و یک نیم را حذف می‌کنیم. در نتیجه کارایی یافتن میانه از مرتبه $O(\log n)$ خواهد بود.

مراجع