دانشکدهٔ ریاضی و آمار

ساختمان داده ها و الگوریتم ها — دکتر جعفر الماسی زاده

تمرینات ۱

محمد ملائی، محسن محمودآبادی، داوود نصرتی امیرآبادی

ویرایش اول — نیم‌سال اول ۱۴۰۳-۱۴۰۲

۱ تمرین

Algorithm 1: Example(a, b)

Input: Two integers a and $b > 0$
Output: To be answered
  $x \leftarrow 0$
  $y \leftarrow |a|$
  while $y \geq b$ **do**
    $y \leftarrow y - b$
    $x \leftarrow x + 1$
  end while
  if $a < 0$ and $y = 0$ **then**
    $x \leftarrow -x$
  end if
  if $a < 0$ and $y > 0$ **then**
    $y \leftarrow b - y$
    $x \leftarrow -(x + 1)$
  end if
  return (x, y)

۱.۱ الف

با یک مثال عددی، مسأله‌ای را که الگوریتم حل می‌کند، مشخص کنید.

جواب

مثال زیر را به ازای ورودی های $a = 15$ و $b = 7$ بررسی می‌کنیم:

step	a	b	x	y	$y \geq b$
0	15	7	0	15	true
1	15	7	1	8	true
2	15	7	2	1	false

طبق جدول، خروجی $(2, 1)$ خواهد بود.

در واقع الگوریتم فوق همان الگوریتم تقسیم است که در آن x همان خارج قسمت و y باقی‌مانده است.

۲.۱ ب

درستی الگوریتم را ثابت کنید.

جواب

(a,b) را به صورت زوج مرتب به الگوریتم می‌دهیم. در هر بار اجرای حلقه‌ی while، به میزان b تا از y کم شده و یک واحد به x افزوده می‌شود. حلقه‌ی while به اندازه‌ی n بار تکرار می‌شود تا جایی که شرط $y \geq b$ برقرار نباشد. یعنی:

$$y = |a| - nb$$
$$x = n - 1 + 1 = n$$

حال با فرض اینکه $|a| - b < b$ باشد و حلقه while به پایان برسد داریم:

$$y = |a| - nb$$
$$x = n$$

در اینجا نیز دو حالت داریم:

$$a \geq 0, y = a - nb \Rightarrow a = xb + y$$
$$a < 0, y = -a - xb \Rightarrow a = -xb - y$$

و همچنین داریم:

$$y \geq 0,\ y < b \Rightarrow 0 \leq y < b$$

طبق قضیه تقسیم از کتاب ریاضیات گریمالدی داریم:

$$a = xb + y$$
$$0 \leq y < b$$

که همان الگوریتم تقسیم است.

۳.۱ پ

با این فرض که $a > b$ باشد، درستی این ادعا را که کارایی زمانی الگوریتم $O(x \log a)$ است، توجیه کنید.

جواب

مشخص است که بیشترین عملیات انجام شده مربوط به محتویات حلقه‌ی $while$ می‌باشد. با توجه به اینکه متغیر x نقش counter را بازی می‌کند، می‌دانیم که در نهایت عدد x نشانگر تعداد دفعات اجرای حلقه‌ی while خواهد بود. (شروطی که بعد از حلقه‌ی while آمده اند و در مقدار x تغییر ایجاد می‌کنند، تنها در صورتی اجرا می‌شوند که مقدار $a < 0$ باشد. اما در صورت سوال فرض بر این گرفته شده که $a > b$ و می‌دانیم که $b > 0$ پس شروط مذکور هرگز اجرا نخواهند شد.)

از طرفی، سنگین ترین عملیات موجود در حلقه‌ی $while$ عملیات تفریق است که هربار رخ می‌دهد. برای a و b های به اندازه‌ی کافی بزرگ، عملیات تفریق هزینه‌ای برابر با تعداد بیت های عدد بزرگتر (یعنی a) خواهد داشت. چرا که این عملیات به ازای هر بیت اجرا خواهد شد. در ضمن، میدانیم که تعداد بیت های یک عدد ده‌دهی (دسیمال) مانند a برابر با $\log_2^a$ است.

در نتیجه، عملیات تفریق که خود با کارایی $\log_2^a$ انجام می‌شود، x بار رخ می‌دهد. پس کارایی الگوریتم فوق $O(x \log a)$ است.

مراجع

۲ تمرین

درستی هر یک از این ادعاهای ریاضی را ثابت کنید.

۱.۲ الف

$$\sum_{i=0}^{n} a^i \in \Theta(a^n), a \geq 1$$

جواب

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + ... + a^n \tag{۱}$$

$$= 1 \cdot \left(\frac{a^n - 1}{a - 1}\right) \tag{۲}$$

$$\Rightarrow \lim_{n\to\infty} \frac{1 \cdot \left(\frac{a^n-1}{a-1}\right)}{a^n} \tag{۳}$$

$$= \lim_{n\to\infty} \frac{a^n \left(1 - \frac{1}{a^n}\right)}{a^n(a-1)} \tag{۴}$$

$$= \frac{1}{a-1} \tag{۵}$$

از جایی که a عدد ثابت است، حاصل نیز عددی ثابت بوده و در نتیجه:

$$\sum_{i=0}^{n} a^i \in \Theta(a^n), a \geq 1$$

۲.۲ ب

$$\sum_{i=1}^{n} i^k \in \Theta(n^{k+1})$$

جواب

$$\begin{aligned}
\sum_{i=1}^{n} i^k &= 1^k + 2^k + \ldots + \left(\frac{n}{2}\right)^k + \ldots + (n-1)^k && (۶)\\
&\geq (\frac{n}{2})^k + \ldots + (n-1)^k && (۷)\\
&\geq (\frac{n}{2})^k + \ldots + (\frac{n}{2})^k && (۸)\\
&\geq (\frac{n}{2})^k \cdot (\frac{n}{2}) && (۹)\\
&\geq \frac{n^{k+1}}{2^{k+1}} && (۱۰)\\
&\approx n^{k+1} && (۱۱)\\
&\Rightarrow S \in \Omega(n^{k+1}) && (۱۲)
\end{aligned}$$

از طرفی داریم:

$$\begin{aligned}
\sum_{i=1}^{n} i^k &= 1^k + 2^k + \ldots + (n-1)^k && (۱۳)\\
&\leq (n-1)^k + \ldots + (n-1)^k && (۱۴)\\
&\leq (n-1)(n-1)^k && (۱۵)\\
&= (n-1)^{k+1} && (۱۶)\\
&\leq n^{k+1} && (۱۷)\\
&\Rightarrow S \in O(n^{k+1}) && (۱۸)
\end{aligned}$$

از معادله‌ی ۱۲ و ۱۸ نتیجه می‌شود که:

$$\sum_{i=1}^{n} \in \Theta(n^{k+1})$$

۳.۲ پ

$$\sum_{i=1}^{n} \frac{1}{i} \in \Theta(\ln n)$$

جواب

$f(x) = \frac{1}{x}$ یک تابع نزولی است. پس مجموع ریمان آن از چپ، از مقدار انتگرال تابع بیشتر خواهد بود.

$$\int_1^n \frac{1}{x} dx \leq \sum_{i=1}^{n} \frac{1}{i} \tag{۱۹}$$

$$\ln(x) \leq \sum_{i=1}^{n} \frac{1}{i} \tag{۲۰}$$

$$\sum_{i=1}^{n} \frac{1}{i} \in \Omega(\ln x) \tag{۲۱}$$

از طرفی، چون $f(x)$ یک تابع نزولی است، مجموع ریمان آن از راست از انتگرال تابع کمتر خواهد بود.

$$\sum_{i=1}^{n} \frac{1}{i+1} \leq \int_1^n \frac{1}{x} dx \tag{۲۲}$$

$$1 + \sum_{i=1}^{n} \frac{1}{i+1} \leq \int_1^n \frac{1}{x} dx + 1 \tag{۲۳}$$

$$\sum_{i=1}^{n} \frac{1}{i} \leq \int_1^n \frac{1}{x} dx + 1 \tag{۲۴}$$

$$\sum_{i=1}^{n} \frac{1}{i} \leq \ln n + 1 \tag{۲۵}$$

$$\sum_{i=1}^{n} \frac{1}{i} \in O(\ln n) \tag{۲۶}$$

از معادله ۲۱ و ۲۶ نتیجه میشود که:

$$\sum_{i=1}^{n} \frac{1}{i} \in \Theta(\ln n)$$

۴.۲ ت

$$\sum_{i=0}^{n} \binom{n}{i} i \in \Theta(n2^n)$$

جواب

$$
\begin{aligned}
\sum_{i=0}^{n} \left(\frac{n}{i}\right) i &= \sum_{i=1}^{n} \left(\frac{n}{i}\right) i && (۲۷)\\
&= \sum_{i=1}^{n} \frac{n!i}{i!(n-i)!} && (۲۸)\\
&= \sum_{i=1}^{n} \frac{n(n-1)!i}{i(i-1)!(n-i)!} && (۲۹)\\
&= n \cdot \sum_{i=1}^{n} \binom{n-1}{i-1} && (۳۰)\\
&= n \cdot \sum_{i=0}^{n-1} \binom{n-1}{i} && (۳۱)\\
&= n2^{n-1} && (۳۲)\\
&\approx n2^{n} && (۳۳)\\
&\Rightarrow \sum_{i=0}^{n} \binom{n}{i} i \in \Theta(n2^n) && (۳۴)
\end{aligned}
$$

مراجع

۳ تمرین

این سه عملیات را در نظر بگیرید:

- $Push(S, e)$: عنصری را در انتهای یک ساختارداده درج می‌کند.
- $Pop(S)$: عنصری را از انتهای یک ساختار داده حذف می‌کند.
- $Find - Min(S)$: کوچکترین عنصر را در یک ساختارداده برمی‌گرداند (بدون آنکه آن را حذف کند).

در اینجا ما به دنبال طراحی الگوریتم‌هایی برای انجام سریع سه عملیات Push و Pop و Find – Min هستیم و قسمت عمده طراحی این الگوریتم‌ها چیزی نیست جز طراحی ساختارداده‌ای برای چینش مناسب داده ها.

ساختار داده‌ای طراحی کنید که با آن بتوان، هر سه عملیات Push و Pop و Find – Min را در زمان $\Theta(1)$ انجام داد. بعد از طراحی ساختارداده، شبه‌کد الگوریتم‌های $Push(S, e)$ و $Pop(S)$ و $Find - Min(S)$ را بنویسید

۱.۳ جواب

نحوه عملکرد دو عملیات $Push$ و Pop یادآور خصوصیات حافظه پشته $(Stack)$ است که به صورت $(First\,In\,Last\,Out)$ عمل می‌کند.

روند حل و یافتن الگوریتم مورد نظر با کارایی $O(1)$ در انجام سه عملیات مذکور:
در ابتدا ساده‌اندیشانه ترین حالتی که به نظر می‌رسد بدین صورت است که اندیس مقدار مینیمم همواره در یک متغیر جداگانه ذخیره شود و در هربار انجام عملیات $Push$ ،بررسی شود که مقدار افزوده شده از مقدار مینیمم بیشتر است یا خیر. در صورتی که بیشتر بود، صرفا به $Stack$ افزوده می‌شود، اما در صورتی که مقدار جدید از مقدار مینمم کمتر باشد، متغیری که حاوی اندیس مقدار مینمم است اپدیت شده و برابر با $len(Stack) - 1$ می‌شود. اما راهبرد مذکور دارای یک ایراد اساسی است. اگر مقدار مینیمم، آخرین عضو پشته باشد، با انجام عملیات Pop ، این مقدار مینمم حذف خواهد شد، اما ما به دومین کمترین مقدار پشته دسترسی نداریم. فلذا برای یافتن مقدار مینمم جدید، مجبور هستیم تا یک بار دیگر پشته را پیمایش کنیم که این عملیات در بهینه ترین حالت برای پشته‌ی نامرتب ما، پیچیدگی‌ای از مرتبه $O(n)$ خواهد داشت. در نتیجه برای حل این مشکل، نیاز داریم تا در هر مرحله از انجام عملیات های Pop و $Push$ ، به اصطلاح آمار مینمم ها دستمان باشد. بدین منظور، به پشته ای دیگر برای ذخیره‌سازی مینمم ها در هر مرحله نیاز داریم.

```python
class SuperStack:
    def __init__(self) -> None:
        self.main_stack = list()
        self.aux_stack = list()
        self.top = -1

    def is_empty(self):
        return self.top == -1

    def add_to_aux(self, x: int):
        if self.is_empty():
            self.aux_stack.append(x)
        elif self.aux_stack[self.top] <= x:
            last_aux_stack = self.aux_stack[self.top]
            self.aux_stack.append(last_aux_stack)
        else:
            self.aux_stack.append(x)

    def find_min(self):
        if self.is_empty():
            return None
        else:
            return self.aux_stack[self.top]

    def push(self, x: int):
        self.main_stack.append(x)
        self.add_to_aux(x)
        self.top += 1

    def pop(self):
        if self.is_empty():
            return None
        else:
            last_main_stack = self.main_stack[self.top]
            self.main_stack.pop(self.top)
            self.aux_stack.pop(self.top)
            self.top -= 1
            return last_main_stack
```

Listing 1: Python Implementation of SuperStack

توضیحات مربوطه:

- متغیر $main\_stack$: لیستی شامل مقادیر اصلی است.
- متغیر aux_stack: لیستی شامل مقدار مینیمم پشته در هر مرحله است. از این پس با نام پشته‌ی کمکی از آن یاد خواهیم کرد.
- متغیر top: طول پشته را مشخص می‌کند.
- متد is_empty: اگر طول پشته 0 باشد ($top == -1$) مقدار $true$ و در غیر این صورت $false$ را برمی‌گرداند.
- متد add_to_aux: متدی است برای افزودن مقدار جدید به پشته‌ی کمکی. در صورتی که پشته خالی باشد، مقدار ورودی را به آن می‌افزاید. در غیر این صورت، دو حالت داریم:

ورودی از مینیمم پشته بیشتر است: در این صورت صرفا مینمم قبلی پشته (آخرین عنصر پشته)، دوباره به پشته افزوده می‌شود.
ورودی از مینمم پشته‌ی کمکی کمتر است (یک مینیمم جدید داریم): مقدار ورودی به پشته‌ی کمکی افزوده می‌شود.

- متد $find\_min$: اگر پشته خالی نباشد، اخرین عضو پشته‌کمکی را برمی‌گرداند.
- متد $push$: ورودی را به پشته‌ی اصلی و سپس به پشته‌کمکی می‌افزاید. سپس، طول پشته را یک واحد بیشتر می‌کند.
- متد pop: اگر پشته خالی نباشد، ابتدا آخرین مقدار افزوده شده به پشته را در متغیر $last\_main\_stack$: ذخیره کرده و سپس با توجه به طول پشته ($self.top$) ، آخرین اعضای هر دو پشته را حذف میکنیم. در نهایت $last\_main\_stack$: را برمی‌گردانیم.

با توجه به اینکه در هربار از انجام عملیات $push$ و pop، تعداد مشخصی (ثابت عددی) فرخوانی صورت می‌گیرد، عملیات‌های مذکور از مرتبه اعداد ثابت هستند و نماد مجانبی آنها $O(1)$ خواهد بود.

مراجع

1. Geeks For Geeks - Special Stack
2. StackOverFlow - Get Min Max in O(1) from a Queue
3. StackOverFlow - Keep track of the minimum effieciently

۴ تمرین

فرض کنید A مجموعه‌ای متنهای باشد، مثلا مجموعه‌ی $A = \{1, 2, \ldots, n\}$ و f تابعی باشد از A به A:

$$f : A \to A$$

۱.۴ الف

الگوریتمی کارا طراحی کنید که با آن بتوان تعیین کرد که آیا چنین توابع f ای، «یک - به - یک» هستند یا خیر. الگوریتمتان را با شبه‌کد توصیف کنید و کارایی زمانی آن را نیز اندازه بگیرید.

جواب

الگوریتم:

Algorithm 2: IsInjective($A, f(x)$)

Input: A set A, and a function $f(x)$
Output: returns $true$ if f(x) is injective, $false$ otherwise.

```
Y = [ ]
for i = 0 to n − 1 do
    if f(A[i]) is in Y then
        return false
    else
        Y.push(A[i])
    end if
end for
return true
```

تحلیل کارایی الگوریتم:

$M(n)$ را تعداد مقایسه‌های الگوریتم در بدترین حالت در نظر می‌گیریم.

حلقه‌ی for برای f های «یک - به - یک» n بار اجرا می‌شود و در هربار اجرای حلقه $i - 1$ بار مقایسه انجام می‌شود. بنابراین داریم:

$$M(n) = \sum_{i=1}^{n} i - 1 = \sum_{i=0}^{n-1} = \frac{n(n-1)}{2} \Rightarrow M(n) \in \Theta(n^2)$$

۲.۴ ب

الگوریتمی کارا طراحی کنید که با آن بتوان بزرگترین زیرمجموعه $S \subseteq A$ را به گونه‌ای که تابع $f : S \to S$ «یک - به - یک» باشد، تعیین کرد. الگوریتمتان را با شبه‌کد توصیف کنید و کارایی زمانی آن را نیز اندازه بگیرید.

جواب

Algorithm 3: BiggestInjectiveSubset($A, f(x)$)

```
Input: A set A, and a function f(x)
Output: returns the biggest injective subset of A
  X = [ ]
  Y = [ ]
  for i = 0 to n − 1 do
      if f(A[i]) is not in Y then
          Y.push(f(A[i]))
          X.push(A[i])
      end if
  end for
  return X
```

توضیح الگوریتم:

در ابتدا X و Y را لیست هایی خالی در نظر می‌گیریم. سپس از 0 تا $n-1$ (تمام اعضای مجموعه‌ی A) بررسی می‌کنیم که در صورت عدم وجود $f(A[i])$ در Y، اندیس آن را به X و مقدار $f(A[i])$ آن را به Y اضافه می‌کنیم. در نهایت X را برمی‌گردانیم که همان مجموعه‌ی اعضایی از A است که تابعی «یک - به - یک» می‌سازند. تحلیل کارایی الگوریتم:

همانند الگوریتم قبل، $M(n)$ را تعداد مقایسه‌های الگوریتم در بدترین حالت در نظر می‌گیریم.

حلقه‌ی for برای f های «یک - به - یک» n بار اجرا می‌شود و در هربار اجرای حلقه $i-1$ بار مقایسه انجام می‌شود. بنابراین داریم:

$$M(n) = \sum_{i=1}^{n} i - 1 = \sum_{i=0}^{n-1} = \frac{n(n-1)}{2} \Rightarrow M(n) \in \Theta(n^2)$$

مراجع

۵ تمرین

این الگوریتم بازگشتی برای مسأله یکتایی عناصر را در نظر بگیرید.

Algorithm 4: UniqueElements($A[0,\ldots,n-1]$) Determines whether all the elements in a given array are distinct

Input: An array $A[0,...,n-1]$
Output: Returns ”true” if all the elements in A are distinct and ”false” otherwise

```
if n = 1 then
    return true
else if not UniqueElements(A[1, ..., n − 2] then
    return false
else if not UniqueElements(A[0, ..., n − 1] then
    return false
else
    return A[0] ≠ A[n − 1]
end if
return (x, y)
```

و سپس به سوالات زیر پاسخ دهید.

۱.۵ الف

چرا این الگوریتم بازگشتی، جواب درست مسأله را برمی‌گرداند؟

جواب

k را تعداد عناصر یک آرایه فرض می‌کنیم. اگر اندازه‌ی $A = 1$ باشد، الگوریتم $true$ را برمی‌گرداند که خروجی صحیحی برای ورودی مذکور است. (حالت پایه) فرض می‌کنیم الگوریتم برای آرایه‌ای با اندازه k درست باشد، نشان می‌دهیم اگر اندازه‌ی آرایه $k+1$ باشد، خروجی الگوریتم درست خواهد بود.

برای تعیین یکتا بودن عناصر، الگوریتم آرایه را به دو آرایه‌ی کوچکتر با اندازه k تقسیم می‌کند:

$$A_1 = A[0,\ldots,n-2], A_2 = A[1,\ldots,n-1]$$

بدین صورت، A_1 یکتایی عناصر را از 0 تا $n-2$ و A_2 یکتایی عناصر را از 1 تا $n-1$ بررسی می‌کند. برای یکتایی عناصر، کافیست عنصر اول $(k=0)$ و عنصر n ام $(k=n-1)$ را مقایسه کنیم تا یکتایی A مشخص شود که در الگوریتم همین اتفاق می‌افتد. به عبارت دیگر، چون دو آرایه به طول k داریم، بنابر فرض استقرا الگوریتم می‌تواند یکتایی عناصر این دو آرایه را تشخیص دهد. چون اشتراک این دو آرایه عناصر عبارت است از: $A[1,\ldots,n-2]$، در نتیجه در صورت ازگذر از سه شرط اول تابع، مطمئن خواهیم بود که اعضای مذکور مشابه نیستند. در اینجاست که با ایجاد شرط چهارم و بررسی عدم یکتایی عناصری که در اشتراک حضور نداشتند، به پاسخ نهایی مسأله می‌رسیم. در نتیجه بنابر استقرای ریاضی، این الگوریتم برای هر آرایه A با اندازه‌ی $n > 0$ درست خواهد بود.

۲.۵ ب

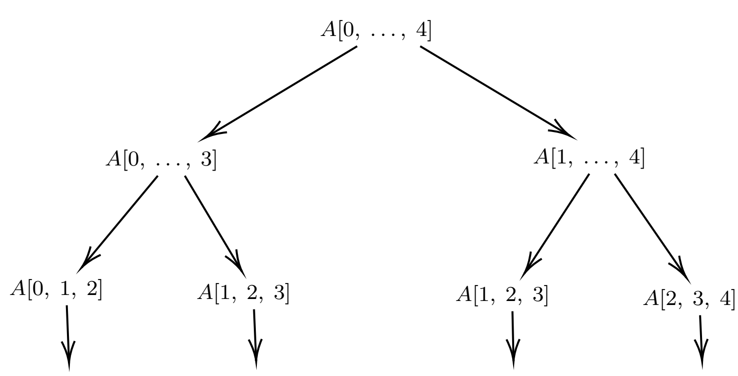
کارایی زمانی الگوریتم چقدر است؟ چرا الگوریتم ناکارا است؟

جواب

$M(n)$ را تعداد مقایسه‌های الگوریتم در بدترین حالت در نظر می‌گیریم:

$$
\begin{aligned}
M(1) &= 1, && (۳۵)\\
M(n) &= M(n-1) + M(n-1) + 4 && (۳۶)\\
&= 2M(n-1) + 4 && (۳۷)\\
&= 2(2M(n-2) + 4) + 4, && (۳۸)\\
&= 2^i M(n-i) + 4i && (۳۹)\\
&= 2^{n-1} M(1) + 4(n-1) && (۴۰)\\
&= 2^{n-1} + 4n - 4 && (۴۱)\\
&\approx 2^{n-1} \approx 2^n && (۴۲)\\
&\Rightarrow M(n) \in \Theta(2^n) && (۴۳)
\end{aligned}
$$

درخت فرخوانی های بازگشتی الگوریتم برای $A[0, 1, 2, 3, 4]$:



از روی درخت فراخوانی‌های بازگشتی، میتوانیم ببینیم که الگوریتم روی آرایه‌های یکسانی چند بار اجرا میشود و نماد مجانبی آن 2^n است، پس الگوریتم کارا نیست.

۳.۵ پ

یک الگوریتم بازگشتی کارا برای مسأله طراحی کنید و کارایی زمانی آن را نیز اندازه بگیرید.

جواب

شبه کد:

Algorithm 5: UniqueElements($A[0,\dots,n-1]$) Determines whether all the elements in a given array are distinct

Input: An array $A[0,...,n-1]$
Output: Returns "true" if all the elements in A are distinct and "false" otherwise

```
if n = 1 then
    return true
else if A[0] is in A[1,...,n-1] then
    return false
else
    return UniqueElements(A[1,...,n-1])
end if
```

کارایی الگوریتم:

$$\begin{aligned}
M(1) &= 1, && (۴۴)\\
M(n) &= M(n-1) + (n-1) + 1 && (۴۵)\\
&= M(n-2) + (n-2) + 1 + (n-1) + 1 && (۴۶)\\
&= M(n-i) + \sum_{k=1}^{i} n-k+1 && (۴۷)\\
&= M(1) + \sum_{k=1}^{n-1} n-k+1 && (۴۸)\\
&= n(n-1) + \frac{1}{2}(n-1)(n-2) + (n-1) && (۴۹)\\
&\approx \frac{n^2}{2} \approx n^2 && (۵۰)\\
&\Rightarrow M(n) \in \Theta(n^2) && (۵۱)
\end{aligned}$$

مراجع