دانشگاه اصفهان
دانشکده علوم ریاضی و کامپیوتر

Data Structures and Algorithms

ساختمان داده ها و الگوریتم ها

عنوان:

تمرینات ۳

اعضای گروه

محمد ملائی

محسن محمودآبادی

داوود نصرتی امیرآبادی

نیم‌سال اول ۱۴۰۲-۱۴۰۳

نام استاد درس

جعفر الماسی زاده

تمرین ۱

این الگوریتم را برای مسأله تولید تمام $n!$ جایگشت n عنصر در نظر بگیرید:

LexicographicPermute(n) Generates permutations in lexicographic order

Require: A positive integer n
Ensure: A list of all permutations of $\{1, \ldots, n\}$ in lexicographic order

```
initialize the first permutation with {12...n}
while last permutation has two consecutive elements in increasing order do
    let i be its largest index such that a_i < a_i + 1          ▷ a_i + 1 > a_i + 2 > ... > a_n
    find the largest index j such that a_i < a_j                 ▷ j ≥ i + 1 since a_i < a_i + 1
    swap a_i with a_j                          ▷ a_{i+1} a_{i+2} ... a_n will remain in decreasing order
    reverse the order of the elements from a_{i+1} to a_n inclusive
```

الف) درستی الگوریتم را ثابت کنید؛ یعنی ثابت کنید که الگوریتم $n!$ جایگشت متفاوت با هم تولید می‌کند.
ب) کارایی زمانی الگوریتم را تعیین کنید.

جواب

الف

ابتدا رابطه ترتیب را برای دو جایگشت P و P' تعریف می کنیم و داریم :

$$P_i = (A_{i0}, A_{i1}, \ldots, A_{in-1})$$

$$P_j = (A_{j0}, A_{j1}, \ldots, A_{jn-1})$$

فرض کنیم A_{im} اولین عنصری باشد که این دو جایگشت در آن باهم متفاوتند، گوییم $P_i > P_j$ اگر و تنها اگر $A_{im} > A_{jm}$ باشد. بنابرین اولین و آخرین جایگشت های این n تایی که کوچکترین و بزرگترین جایگشت ها نیز هستند را به صوت زیر داریم :

$$P_{first} = (p_0, p_1, \ldots, p_{n-1}) \qquad \forall i > j : p_i > p_j$$
$$P_{last} = (p_0, p_1, \ldots, p_{n-1}) \qquad \forall i > j : p_i < p_j$$

فرض کنیم در قسمتی از برنامه جایگشتی مانند $P = (a_0, a_1, \ldots, a_{n-1})$ ایجاد شده است، حال رابطه بین P و P' را که جایگشت بعدی P است را پیدا می کنیم. فرض می کنیم اولین عنصری که P و P' که در آن باهم تفاوت دارند m امین عنصر آرایه باشد که آن را b می نامیم. بنابرین $b \neq a_{m-1}$ و از آنجایی که $P' > P$ نتیجه می گیریم $a_{m-1} < b$.
از آنجایی که P بزرگترین جایگشت بین جایگشت هایی است که m عنصر اول آنها با P برابر است می توانیم درباره $n-m$ عنصر باقیمانده نتیجه بگیریم که :

$$a_m > a_{m+1} > \ldots > a_{n-1} \tag{۱}$$

از طرفی $m-1$ عنصر اول P' با P یکسان است و $b \neq a_{m-1}$ پس می توانیم نتیجه بگیریم که b یکی از $n-m$ عنصر باقیمانده خواهد بود که مجموعه آنها را S می نامیم :

$$S = \{a_m, a_{m+1}, a_{m+2}, ..., a_{n-1}\}$$

میدانیم که P' کوچکترین جایگشت از بین جایگشت های بزرگتر از P است پس b باید کوچکترین عضو S باشد به طوری که $b > a_{m-1}$ و $a_k = b$. چنین b زمانی وجود دارد که $a_m > a_{m-1}$.
حال با پیدا کردن b ، آن را با عنصر m ام جابجا می کنیم و $n - m$ و مجموعه جدید را S' می نامیم و داریم :

$$S' = (S \setminus \{a_k\}) \cup \{a_{m-1}\}$$

از آنجایی که P' از تمام جایگشت های بعد از خود کوچکتر است پس باید از اعضای آن از m به بعد به صورت نزولی مرتب شده باشند و از (1) داریم که اعضای S' به صورت صعودی مرتب شده اند پس کافی است ترتیب آن ها را برعکس کنیم تا به P' برسیم.

الگوریتم از P_{first} که اولین جایگشت است شروع می شود

حال بنابر استقرا فرض می کنیم که الگوریتم تا جایگشت P را محاسبه کرده است، نشان دادیم که الگوریتم با انجام مراحل بالا می تواند جایگشت بعدی P را تا هنگام برقرار بودن شرط محاسبه کند پس الگوریتم تمام جایگشت های این n تایی را از P_{first} تا P_{last} محاسبه می کند. بنابراین حکم ثابت شد و الگوریتم درست است.

ب

در بدترین حالت، الگوریتم برای پیدا کردن بزرگترین توالی نزولی از اعداد باید n مقایسه انجام دهد و از آنجایی که این روند برای هر جایگشت اتفاق می افتد که $n!$ هستند بنابراین این کارایی این الگوریتم $O(n \times n!)$ خواهد بود.

مراجع

Mathsanew - Generating Permutations

تمرین ۲

پلیس شهر رایانستان، همه خیابانهای شهر را یک طرفه کرده است. با وجود این، شهردار رایانستان ادعا می‌کند که هنوز راهی برای رانندگی قانونی از هر تقاطعی در شهر به هر تقاطع دیگر در آن، وجود دارد. چون مخالفان شهردار قانع نمیشوند و شهر هم بسیار بزرگ است، برای تعیین درستی یا نادرستی ادعای شهردار، برنامه‌ای رایانه ای لازم است. از آنجا که انتخابات شورای شهر به زودی برگزار خواهد شد، شهردار خدمتگزار، که نگران از دست دادن فرصت عظیم خدمت به شهروندان است، برای حل مسأله‌اش به سراغ شما آمده است. شما میدانید که با توجه به بزرگی شهر و زمان اندکی که شهردار دارد، تنها باید به دنبال الگوریتمی خطی برای اثبات درستی ادعای شهردار بود.

الف) مسأله شهردار را به شکل یک مسأله گراف بیان کنید و آنگاه الگوریتمی برای حل آن ارائه کنید که زمان اجرای آن (برحسب تعداد رأس‌ها و تعداد یال‌های گراف ورودی) خطی باشد. الگوریتم خود را با شبه کد توصیف کنید.

ب) شهردار باهوش رایانستان، به این هم فکر کرده است که در صورتی که با الگوریتم شما، نادرستی ادعای او مشخص شد، ادعای ضعیفتری را طرح کند: اینکه اگر شما از ساختمان شهرداری رانندگی را شروع کنید، خیابان‌های یک طرفه را بپیمایید و به هر جایی که ممکن بود برسید، باز همیشه راهی برای آنکه به طور قانونی رانندگی کنید تا به ساختمان شهرداری برگردید، خواهید داشت. این ادعای ضعیفتر شهردار را هم به شکل یک مسأله گراف بیان کنید و باز الگوریتمی خطی ارائه کنید که با آن بتوان درستی یا نادرستی ادعای او را تحقیق کرد. الگوریتم خود را با شبه کد توصیف کنید.

جواب

الف

برای حل این مسئله می توانیم خیابان ها و تقاطع های شهر را مانند رئوس و یال های گراف در نظر بگیریم. با توجه به یکطرفه بودن خیابان ها، گراف مورد نظر جهت دار خواهد بود و برای اینکه نشان دهیم بین هر دو راس، مسیری وجود دارد باید نشان دهیم که این گراف قویا همبند است.

```
dfs
Require: Graph G = <V, E> and v, a vertex of G
Ensure: Marks all the vertices connected to v with 1
  mark v with 1
  for each vertex w in V adjacent to v do
      if w is marked with 0 then
          dfs(w)
```

IsStronglyConnected

Require: Graph G = <V, E>
Ensure: Boolean representing if the graph is strongly connected or not

```
mark each vertex in V with 0 as a mark of being "unvisited"
G_1 = dfs(G, V_0)
for each vertex v in V_1 do
    if v is marked with 0 then
        return false

G' is transposed graph of G
G_2 = dfs(G', V_0)
for each vertex w in V_2 do
    if v is marked with 0 then
        return false

return true
```

الگوریتم اول، تمام راس های متصل به ورودی را پیمایش می کند. در الگوریتم دوم ابتدا از یکی از راس های گراف شروع کرده و تمام راس های متصل به آن را علامت گذاری می کنیم. اگر تمام رئوس علامت داشته باشند، آن گاه حداقل یک مسیر بین راس انتخاب شده و بقیه رئوس وجود دارد، در غیر اینصورت این گراف قویا همبند نیست.

حال باید نشان دهیم که از هر راس دیگر گراف به راس انتخاب شده نیز یک مسیر وجود دارد. این کار را با بررسی وجود مسیر بین راس انتخاب شده و رئوس گراف ترانهاده انجام می دهیم. به عبارتی اگر در گراف ترانهاده بین راس انتخاب شده و هر راس دیگر یک مسیر وجود داشته باشد پس در گراف اصلی، یک مسیر از هر راس به راس انتخاب شده وجود خواهد داشت.

مانند قبل گراف ترانهاده را از راس انتخاب شده پیمایش می کنیم و تمام رئوس مجاور آن را به صورت بازگشتی علامت گذاری می کنیم، اگر همه گراف علامت گذاری شده باشد پس می توانیم نتیجه بگیریم که گراف قویا همبند است.

این الگوریتم از دو جستجوی عمقی و دو حلقه استفاده می کند، پس کارایی زمانی آن

$$O(4|V| + 2|E|) = O(|V| + |E|)$$

خواهد بود.

ب

قسمت ب را نیز مشابه با قسمت الف می توانیم حل کنیم، با این تفاوت که پیمایش گراف را از راس نشان دهنده شهرداری شروع کرده و مولفه همبندی گراف را که شامل این راس است را پیدا کرده و سپس با پیمایش گراف ترانهاده آن، قویا همبند بودن آن را بررسی می کنیم.

Algorithm 1 GetConnectedComponent

Require: Graph G = <V, E> which is traversed by dfs algorithm, Vertex v
Ensure: Connected Component of G which includes v

```
connectedComponent = < V_2, E_2 >
for each vertex v in V do
    if v is marked with 0 then
        add v to V_2 and its edges to E_2
return connectedComponent
```

Algorithm 2 IsStronglyConnected

Require: Graph G = <V, E>, Vertex v
Ensure: Boolean representing if the connected component of G which includes v is strongly connected or not

```
G1 = dfs(G, v)
G2 = GetConnectedComponent(G1, v)
G' is transposed graph of G2
G3 = dfs(G', v)
for each vertex w in V3 do
    if v is marked with 0 then
        return false

return true
```

کارایی زمانی این الگوریتم نیز مانند قسمت قبل خطی خواهد بود.

مراجع

GeeksForGeeks - Connectivity in a directed graph
GeeksForGeeks - Strongly connected components

تمرین ۳

آرایه $A[0 \cdots n-1]$ شامل «عنصر غالب» است، اگر بیشتر از $\lfloor \frac{n}{2} \rfloor$ عنصر در آن، یکسان باشند. الگوریتمی کارا طراحی کنید که آرایه ای را به عنوان ورودی بگیرد و مشخص کند که آیا آرایه، عنصر غالب دارد یا خیر، و در صورت وجود عنصر غالب، آن را تعیین کند. در حالت کلی، عناصر آرایه، لزوماً اشیاء ترتیب پذیر مانند اعداد صحیح نیستند و به همین دلیل، نمیتوانید برای حل مسأله، از مقایسه هایی به شکل «آیا $A_i > A_j$ است؟» استفاده کنید. (مثلاً تصور کنید که عناصر آرایه، فایلهای تصویری GIF باشند.) اما شما میتوانید در الگوریتم از پرسشهایی به شکل «آیا $A_i = A_j$ است؟» استفاده کنید.

الف) الگوریتمی ساده اندیشانه برای این مسأله طراحی کنید و آن را با شبه کد توصیف کنید. کارایی زمانی الگوریتم‌تان باید $\Theta(n^2)$ باشد و کارایی فضایی آن $\Theta(1)$.

ب) الگوریتمی تقلیل و حل برای این مسأله طراحی کنید و آن را با شبه کد توصیف کنید. کارایی زمانی الگوریتم‌تان باید $\Theta(n)$ باشد و کارایی فضایی آن $\Theta(1)$.

جواب

الف

اولین و ساده اندیشانه ترین راهکار موجود برای حل مسأله، آن است که از ابتدای لیست شروع به پیمایش کرده و بررسی کنیم هر عنصر چند بار در لیست تکرار شده است.

```
simple_find_major(A)
Require: a list A[0, ..., n − 1] containing n objects
Ensure: returns the major object whose count is more than ⌊n/2⌋, -1 if no major object is found
  for i in A do
    count = 0
    for j in A do
      if i == j then
        count += 1
    if count > len(A) // 2 then
      return i
  return -1
```

همانطور که مشخص است، به ازای هر عضوِ i در A ، یک بار پیمایش صورت می گیرد و تعداد تکرار i در لیست A مشخص شده، در متغیر $count$ ذخیره می گردد. در انتهای پیمایش و قبل از پرداختن به عضو بعدی $(i+1)$ مقدار متغیر $count$ بررسی شده و در صورت بیشتر بودن از $len(A)//2$ مقدار i به عنوان پاسخ مسئله خروجی داده می شود. در انتهای پیمایش اعضای A اگر تا آن زمان مقداری به عنوان خروجی مشخص نشده باشد، -1 به عنوان خروجی داده می شود.

به ازای هر یک از اعضای لیست، تمام اعضای لیست یک بار پیمایش می شوند. از آنجایی که عمل پایه در حلقه اول و دوم، یک واحد مقایسه است، کارایی زمانی آن $\Theta(n \times n) = \Theta(n^2)$ خواهد بود.

کارای فضایی نیز، به دلیل وجود صرفا ۳ متغیر و عدم استفاده از لیستی جداگانه برای یافتن جواب مسأله برابر با $\Theta(1)$ خواهد بود.

ب

از آنجایی که کارای زمانی الگوریتم باید از مرتبه $\Theta(n)$ باشد، مشخص است که تعداد پیمایش های روی لیست باید تعدادی ثابت باشد. الگوریتم بویر–مور با چنین کارایی زمانی و همچنین کارایی فضایی $\Theta(1)$ چنین عمل کرده که با یک بار پیمایش، کاندیدی را

با عنوان عضو *غالب احتمالی* مشخص می کند. اما چون ما از وجود عضو غالب در لیست اطمینان نداریم، پیمایشی دیگر در لیست را آغاز کرده و از غالب بودن عنصری که کاندید شده است، اطمینان حاصل می کنیم.

```
find_candidate(A)
```

Require: a list $A[0,\ldots,n-1]$ containing n objects
Ensure: returns a candidate object whose count is possibly more than $\lfloor \frac{n}{2} \rfloor$

```
candidate = 0
votes = 0
for i from 0 to n − 1 do
    if votes = 0 then
        candidate = A[i]
        votes = 1
    else
        if A[i] == candidate then
            votes += 1
        else
            votes -= 1
return candidate
```

الگوریتم فوق، از اولین عضو لیست پیمایش را آغاز می کند. سپس با رویکردی هوشمندانه، به خنثی کردن اعضای لیست می پردازد. بدین صورت که با رسیدن به عنصری مشابه با متغیر *candidate* ، یک واحد به مقدار متغیر *votes* می افزاید و در صورت عدم تشابه مقدار متغیر را یک واحد می کاهد. در هر مرحله از پیمایش، در صورتی که مقدار متغیر *votes* صفر شده باشد، این مفهوم منتقل می شود که تعدادی عضو پیش از این مرحله از پیمایش خنثی شده اند و اکنون نوبت به انتخاب عنصر بعدی به عنوان کاندید رسیده است. در نهایت عنصری که در متغیر *candidate* باقی می ماند، کاندید و جواب احتمالی ماست.

در واقع عملکرد الگوریتم را می توان به گونه ای دیگر نیز توجیه کرد: ما با آغاز پیمایش، عنصر اول را به عنوان کاندید احتمالی خود در نظر می گیریم. در صورتی که عضو بعدی در لیست مشابه کاندید ما باشد، یک واحد به تعداد رای های کاندید اضافه می کنیم. از طرفی دیگر، در صورت مشاهده عنصری که متفاوت با کاندید ماست،، تعداد آرای کاندید را یک واحد کاهش می دهیم. زمانی که تعداد آراء صفر شود، می توان گفت که پیش از این تعدادی رأی دهنده حضور داشته اند که آرای آنها با یکدیگر خنثی شده است. پس عملا حضور یا عدم حضور آنها در لیست تغییری در برنده یا بازنده شدن کاندید نهایی نخواهد داشت. در نتیجه به بررسی باقی لیست می پردازیم و به دیگر سخن می توان گفت که آرای قبلی را نادیده می گیریم. از این جهت، الگوریتم فوق یک الگوریتم تقلیل و حل به شمار می رود چرا که با هر بار صفر شدن متغیر *votes* مسأله را به مسأله ای جدید با اندازه $n-(2votes)$ تبدیل می کند.

```
is_candidate_major(A, candidate)
```

Require: a list $A[0,\ldots,n-1]$ containing n objects, and a *candidate*
Ensure: returns candidate if it's a major element, -1 if not

```
count = 0
for i in A do
    if i == candidate then
        count += 1
if count > n // 2 then
    return candidate
else
    return -1
```

الگوریتم فوق نیز، تنها یک بار در لیست پیمایش کرده و تعداد دفعات تکرار عنصر کاندید را در متغیر *count* ذخیره می کند.

در نهایت اگر $count$ از کف نیمی از تعداد اعضا بیشتر باشد، آن را به عنوان عنصر غالب بر می گرداند. در غیر این صورت -1 خروجی داده می شود.

در پایان با ترکیب دو الگوریتم، به پاسخ نهایی می رسیم.

---

find_major(A)

Require: a list $A[0, \ldots, n-1]$ containing n objects
Ensure: returns major element if it exists, -1 if not
  $candidate = find\_candidate(A)$
  return $is\_candidate\_major(A, candidate)$

در اینجا، ابتدا کاندید را یافته و سپس به بررسی غالب بودن آن می پردازیم.

کارایی زمانی الگوریتم فوق $\Theta(2n) = \Theta(n)$ است. به این دلیل که از دو حلقه جداگانه تشکیل شده که هرکدام تمام لیست را پیمایش می کنند. همچنین کارایی فضایی آن نیز به دلیل عدم استفاده از لیستی دیگر به جز لیست ورودی، و صرفا تعدادی ثابت از متغیر ها، $\Theta(1)$ است.

مراجع

GeeksForGeeks - Boyer-Moore Majority Voting Algorithm
YouTube - LeetCode 169. Majority Element - Interview Prep Ep 73 | Boyer–Moore majority vote algorithm

تمرین ۴

تصور کنید که آرایه‌ای داریم به نام A و به طول n. و اینکه ما می‌دانیم که از ابتدای آرایه تا جایی (که ما نمی‌دانیم کجاست!) اعداد صحیح به ترتیب صعودی قرار گرفته‌اند و از آنجا به بعد تا انتهای آرایه، اعداد صحیح به ترتیب نزولی قرار گرفته‌اند. (در حالات خاص، ممکن است آرایه قسمت صعودی یا قسمت نزولی نداشته باشد.)
الف) الگوریتمی با کارایی $O(\log n)$ برای یافتن بزرگ‌ترین عدد صحیح در آرایه A بیابید.
ب) تصور کنید که آرایه‌ای بسیار طولانی داریم به نام A و به طول m . و اینکه ما می‌دانیم که از ابتدای آرایه تا جایی (که ما نمی‌دانیم کجاست!) اعداد صحیح به ترتیب صعودی قرار گرفته‌اند و از آنجا به بعد تا انتهای آرایه، که قسمت اعظم آرایه است، نمادهای ∞ در آرایه ذخیره شده‌اند.
فرض کنید که می‌خواهیم کلید K را که عددی صحیح است در آرایه A جستجو کنیم. اگر n (که مقدار آن برای ما معلوم نیست) طول قسمت کوچکی از ابتدای آرایه باشد که اعداد صحیح در آن قسمت ذخیره شده باشند، الگوریتمی با کارایی $O(\log n)$ برای جستجوی کلید K طراحی کنید. (توجه کنید که حتی اگر آرایه را بی‌انتها تصور کنیم باز می‌توان الگوریتمی با کارایی $O(\log n)$ طراحی کرد.)

جواب

الف

با توجه به اینکه کارایی زمانی الگوریتم باید $O(\log n)$ باشد، به نظر می رسد که با نوعی الگوریتم جستجوی دودویی مواجه هستیم که در هر مرحله از اجرا، فضای جستجو رو نصف می کند.

```
find_biggest(A)
Require: a list A[0, ..., n − 1] containing n numbers
Ensure: returns maximum
  left = 0
  right = len(A) - 1
  while left ≤ right do middle = (left + right)//2
      if middle == 0 then
          return max(A[0], A[1])
      else if middle == len(A) - 1 then
          return max(A[middle − 1], A[middle])
      else if A[middle − 1] < A[middle] > A[middle + 1] then
          return A[middle]
      else if A[middle − 1] < A[middle] < A[middle + 1] then
          left = middle + 1
      else if A[middle − 1] > A[middle] > A[middle + 1] then
          right = middle − 1
```

الگوریتم فوق تا حدود زیادی مشابه الگوریتم جستجوی دودویی است. به طور کلی، در هر مرحله عضو مورد نظر را با عنصر قبلی و بعدی اش مقایسه کرده و در صورتی که از هر دو بیشتر باشد، آن را به عنوان عنصر ماکسیمم معرفی می کنیم. اما در صورتی که اشاره گر $middle$ روی عنصری قرار بگیرد که در شیب صعودی قرار دارد، اشاره گر سمت چپ را به عنصر بعدیِ $middle$ منتقل می کنیم. (چون اولا مطمئن هستیم خود $middle$ جواب مسئله نیست، و دوما می دانیم جواب قطعا بعد از $middle$ قرار دارد. چون روی شیب صعودی قرار گرفته ایم ولی هنوز به ماکسیمم نرسیده ایم) به همین شکل، در صورتی که اشاره گر $middle$ روی عناصر با شیب نزولی قرار بگیرد، اشاره گر $right$ روی عنصر قبلیِ $middle$ قرار می گیرد (چون اولا مطمئن هستیم خود $middle$ جواب

مسئله نیست، و دوما می دانیم جواب قطعا قبل از $middle$ قرار دارد. چون روی شیب نزولی قرار گرفته ایم و ماکسیمم را رد کرده ایم) همچنین، در صورتی که اشاره گر $middle$ روی اولین را آخرین عنصر قرار بگیرد، می دانیم که فاصله $left$ و $right$ یک واحد است و صرفا دو عنصر برای مقایسه باقی مانده اند، پس ماکسیمم آنها را به عنوان جواب مسأله باز می گردانیم.
از طرفی، چون هر بار بازه، مسأله را به دو نیم تقسیم کرده و یک نیم را در نظر نمی گیریم، و عمل پایه را مقایسه در نظر می گیریم، چون تعداد ثابتی عمل پایه ای داریم، کارایی زمانی الگوریتم مضربی ثابت از $O(\log n)$ خواهد بود که با توجه به در نظر نگرفتن ثوابت، کارای زمانی الگوریتم همان $O(\log n)$ خواهد بود.

ب

الگوریتم زیر را در نظر بگیرید:

```
find_k(A, k)
Require: a list A[0, . . . , m − 1]
Ensure: returns index of k in A
  left = 1
  right = len(A) - 1
  while A[left] < k do
      left× = 2
  right = left
  left// = 2
  while left ≤ right do middle = (left + right) // 2
      if A[middle] == k then
          return middle
      else if A[middle] < k then
          left = middle + 1
      else
          right = middle − 1
  return -1
```

به طور خلاصه، آنقدر نشانه گر سمت چپ را دوبرابر می کنیم تا به مقداری برسیم که از کلید k بیشتر است (خواه این مقدار عدد باشد، خواه بینهایت). آنگاه، اشاره گر راست را برابر با اشاره گر چپ قرار داده و سپس اشاره گر چپ را نصف می کنیم (ما هربار اشاره گر چپ را دوبرابر کرده بودیم، پس بعد از یافتن اشاره گر راست، مطمئن هستیم که قبل از $left//2$ عنصری نبوده که از کلید k بیشتر باشد، چون در این صورت حلقه همان جا متوقف می شد.)
اگر دقت کنیم، تا اینجا ما به اندازه $\lceil \log k \rceil$ یا همان $\lfloor \log k \rfloor + 1$ عمل مقایسه انجام داده ایم، پس کارایی الگوریتم تا اینجا برابر $O(\log n)$ خواهد بود چون در بدترین حالت k همان n خواهد بود.
اکنون که چپ و راستِ بازه ای که k در آن قرار دارد را یافته ایم، با اعمال الگوریتم جستجوی دودویی به یافتن محل دقیق کلید k می پردازیم.
در بدترین حالت، $right = n$ و $left = n//2$ است که باعث می شود طول بازه ی جستجو $n - \frac{n}{2} = \frac{n}{2}$ باشد و کارایی زمانی الگوریتم جستجوی دودویی $O(\log \frac{n}{2})$ شود که همان $O(\log n)$ است.

مراجع

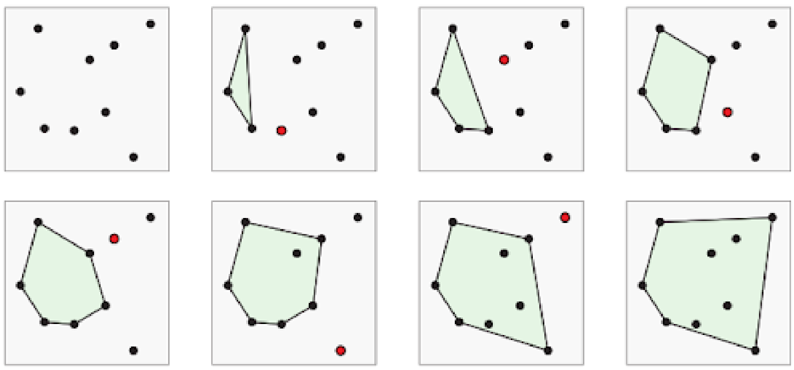
تمرین ۵

فرض کنید $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ مجموعه‌ای از n نقطه در صفحه باشد با این قید که دو یا بیشتر از دو نقطه از آن نقاط روی یک خط عمودی قرار نداشته باشند و سه یا بیشتر از سه نقطه از آن نقاط روی یک خط قرار نداشته باشند. با این فرض، این توصیف متنی از الگوریتمی برای ساخت پوسته محدب مجموعه P را در نظر بگیرید:

۱. مجموعه نقاط P را به ترتیب صعودی مؤلفه x آنها مرتب کنید تا مجموعه مرتب P' به دست آید.

۲. پوسته محدب سه نقطه اول در مجموعه مرتب P' را که یک مثلث است، بسازید.

۳. این عملیات را تا زمانی که تمام نقاط مجموعه P' پردازش شوند تکرار کنید: نقطه بعدی در مجموعه P' را به پوسته محدب فعلی اضافه کنید تا پوسته محدب بزرگ‌تری ایجاد شود. (با اضافه کردن یک نقطه به پوسته فعلی، ممکن است نقاطی که رأسی از پوسته بوده‌اند درون پوسته بعدی قرار گیرند.)



الف) الگوریتم را با شبه‌کد آنقدر دقیق توصیف کنید که بتوان به راحتی مبتنی بر آن شبه‌کد، برنامه‌ای برای پیاده‌سازی الگوریتم نوشت. توصیف دقیق الگوریتم، مستلزم آن است که مشخص کنیم در هر مرحله، نقطه بعدی در لیست مرتب نقاط به کدام یک از دو نقطه پوسته محدب فعلی وصل می‌شود.

ب) الگوریتم را چگونه باید غنی کرد تا در حالاتی هم که دو یا بیشتر از دو نقطه روی یک خط عمودی قرار داشته باشند، درست کار کند؟ الگوریتم را چگونه باید غنی کرد تا در حالاتی هم که سه یا بیشتر از سه نقطه روی یک خط قرار داشته باشند، درست کار کند؟

پ) ثابت کنید که می‌توان کارایی زمانی الگوریتم برای ساخت پوسته محدب هر مجموعه n نقطه‌ای را به صورت $O(n^2)$ بیان کرد.

جواب

الف

برای اضافه کردن نقطه جدید، باید نقاطی را بیابیم که در پوسته ایجاد شده به این نقطه متصل اند. این کار را با دانستن این نکته که اگر دو نقطه جزئی از پوسته محدب باشند، آنگاه تمام نقاط مرزی این پوسته در طرف دیگر خط گذرنده از این دو نقطه خواهند بود انجام خواهیم داد.

همان طور که در صورت سوال نیز یادآوری شده است، پس از اضافه کردن این نقطه ممکن است تعدادی از نقاط درون این پوسته قرار گیرند که باید آن ها را از لیست نقاط مرزی حذف کنیم. این کار را با بررسی نقاط دیگر این پوسته انجام می دهیم به طوری که اگر p۱ و p۲ نقاطی باشند که در قسمت قبل پیدا کرده ایم، آنگاه اگر یکی از نقاط مرزی، سمت راست خط گذرنده از این دو نقطه باشد باید آن را حذف کنیم زیرا درون مثلثی از نقاط مرزی خواهند بود.

Algorithm 3 GetOrientation

Require: Points P_1, P_2 and Q
Ensure: Orientation of Q with respect to P_1 and P_2
1: Right or top
2: Bottom or left
0: In the same line
A = P_1P_2
B = P_1Q
return $sign(A \times B)$

Algorithm 4 IsBoundaryLine

Require: Points Q_1, Q_2 and List Of Points $P = \{p_1, p_2, ..., p_n\}$
Ensure: Boolean representing if Q_1Q_2 is a boundary line or not
orientation = GetOrientation(Q_1, Q_2, P_1)
for each point p in P **do**
  if orientation * GetOrientation(Q_1, Q_2, p) < 0 **then**
    return false
return true

Algorithm 5 GetConvexHull

Require: A list of points $P = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$
Ensure: Convex Hull of P
P = sortByX(P)
boundaryPoints = $\{p_1, p_2, p_3\}$
for each p in P from p_4 to p_n **do**
  connectedPoints = {}
  for each point b in boundaryPoints **do**
    if IsBoundaryLine(p, b) **then**
      connectedPoints.append(b)
  for each point c in boundaryPoints **do**
    orientation = GetOrientation(connectedPoints[0], connectedPoints[1], c)
    if c is not in connectedPoints and orientation > 0 **then**
      boundaryPoints.remove(c)
  boundaryPoints.append(p)
return boundaryPoints

مراجع