هوالعلیم

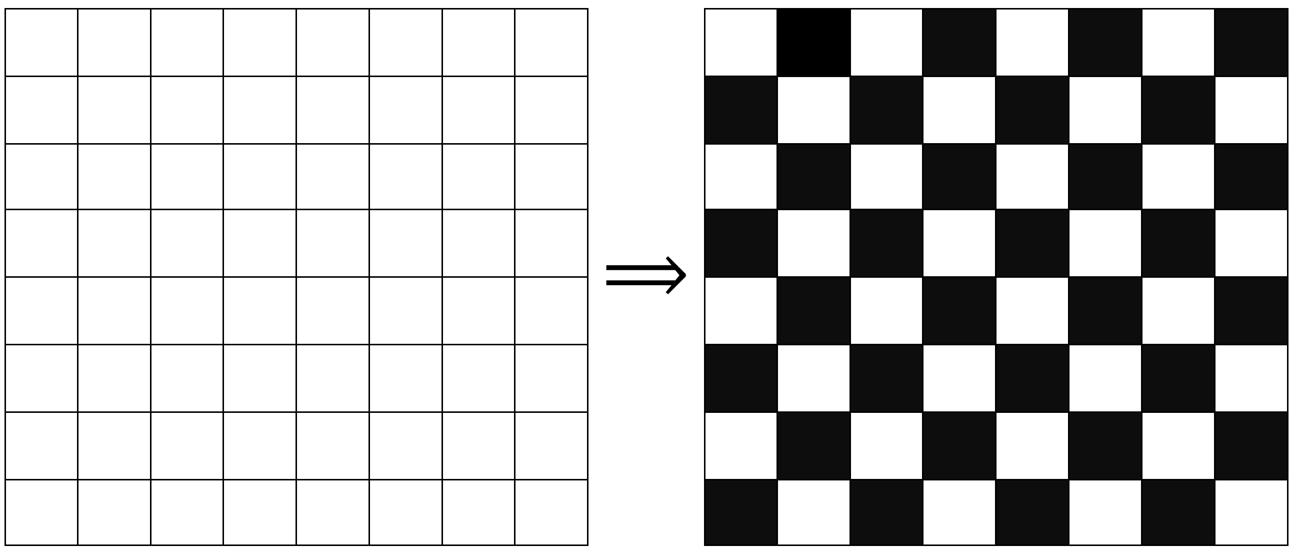
طراحی و تحلیل الگوریتم‌ها

نیمسال دوم سال تحصیلی ۱۴۰۱ – ۱۴۰۰

تمرینات نظری ۱

مریم رضائی

۱. در روزگاری، پادشاهی زندگی می‌کرد دوستدارِ شطرنج. او کاخی داشت که طرح کف آن مانند یک صفحه شطرنجی ۸×۸ بود و هر یک از ۶۴ اتاق کاخ، در هر یک از چهار دیوار اطرافش، یک در داشت. در ابتدا، کف همه اتاق‌های کاخ با رنگ سفید، رنگ شده بودند. پادشاه دستور داد که کف اتاق‌ها به گونه‌ای مجدداً رنگ‌آمیزی شوند که شبیه به مربعات یک صفحه شطرنجی، یک در میان، سیاه و سفید شوند. برای این منظور، نقاش او مجبور بود که در کاخ قدم بزند و به هر اتاقی که وارد شد، رنگ کف آن را (از سیاه به سفید یا از سفید به سیاه) تغییر دهد. البته نقاش مجاز بود که از یک در کاخ خارج شود و از دری دیگر به آن وارد شود. آیا نقاش راهی برای اجرای دستور پادشاه داشته است به گونه‌ای که از آن راه، بیشتر از ۶۰ بار اتاق‌ها را مجدداً رنگ نکند؟

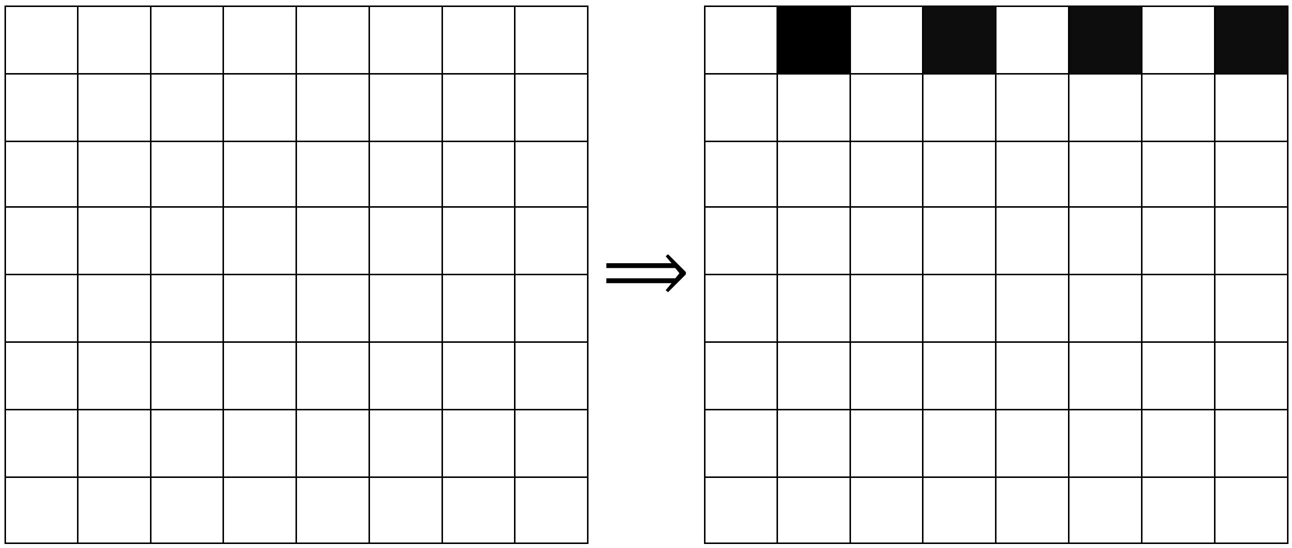


جواب:

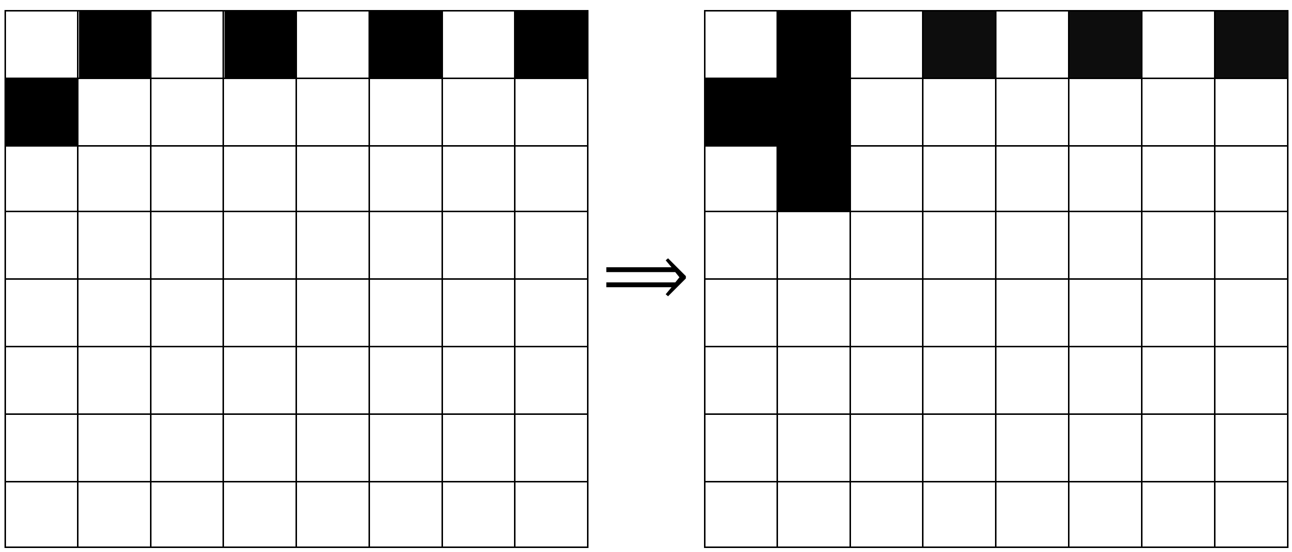
از آنجا که نقاش باید کمتر از ۶۰ بار وارد اتاق‌ها بشود و زمین کاخ در ابتدا سفید است، برای صرفه جویی در دفعات ورود به اتاق می‌توانیم ابتدا اتاق‌های کناری را که می‌توان به تنهایی به آن‌ها وارد و خارج شد یکی در میان از سفید به مشکی تغییر دهیم، سپس برای اتاق‌های میانی با ورود به یکی از اتاق‌های کناری و ادامه مسیر به درون کاخ پیش رویم به طوری که یکی در میان اتاق‌ها سیاه و سفید شود. برای این کار، با توجه به این که هر بار به اتاقی وارد می‌شویم باید رنگ آن را تغییر دهیم و در ابتدا همه سفید هستند، اگر میخواهیم اتاقی سفید باشد باید دوبار از آن گذر کنیم.

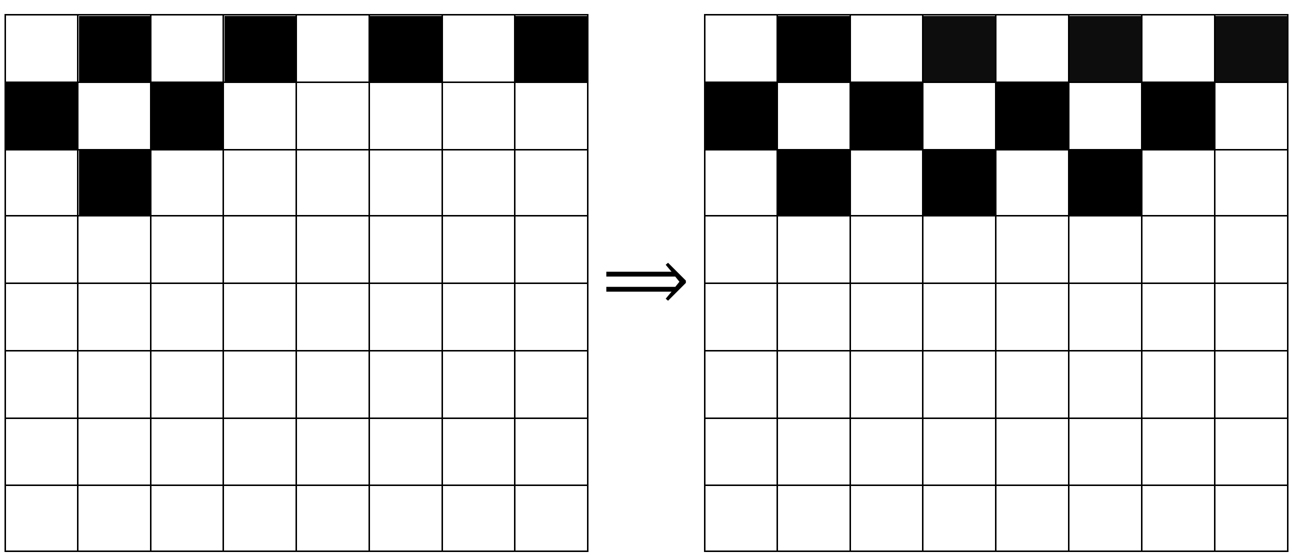
در ادامه مراحل این الگوریتم به ترتیب با شکل رسم شده‌اند.

۱) ابتدا اتاق‌های ضلع شمالی کاخ را با ورود و خروج یکی در میان به آن‌ها به مشکی تبدیل می‌کنیم (۴ ورود):

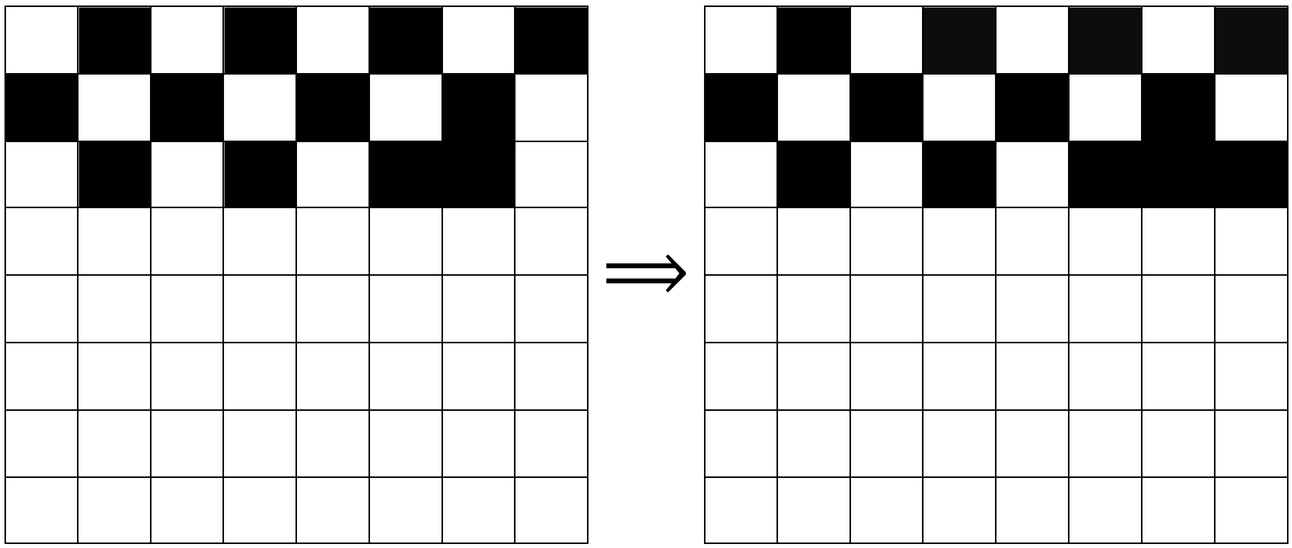


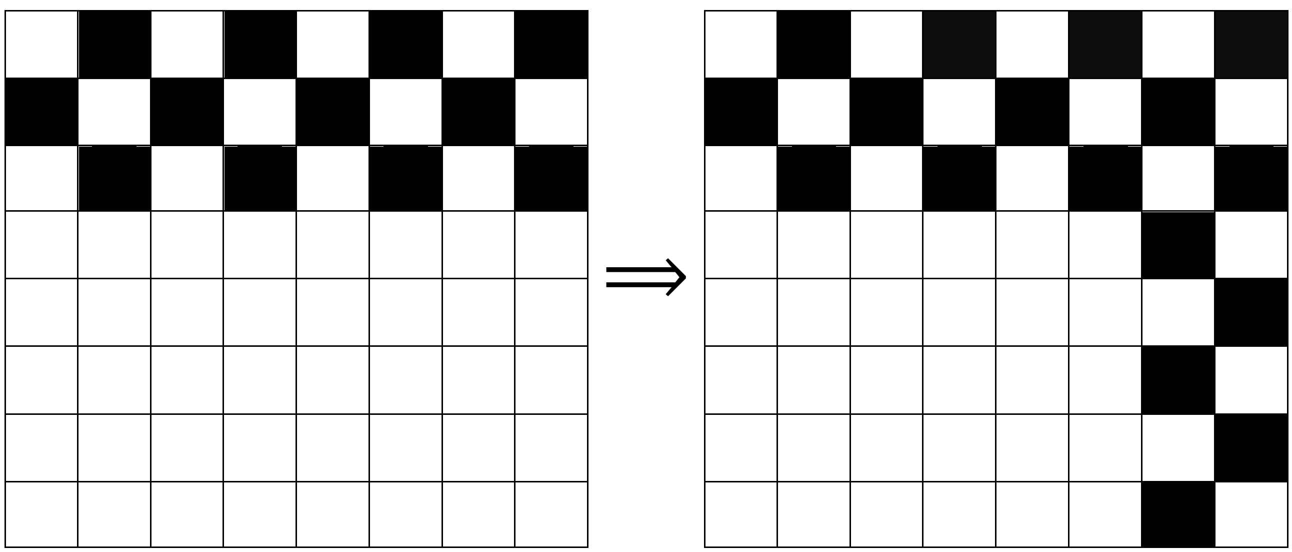
۲) سپس از اتاقی غربی وارد شده و به شرق رفته و مشکی می‌کنیم و برای سفید دوبار گذر می‌کنیم (۱۳ ورود):



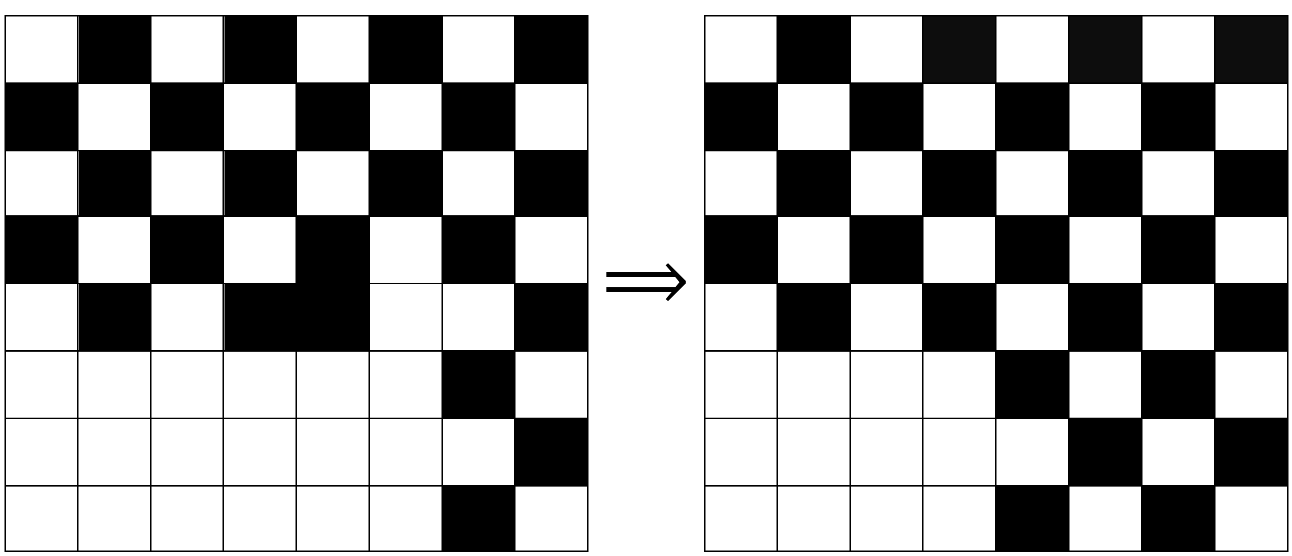


۳) مسیر را تغییر داده و به جنوب می‌رویم و مانند قبل یکی در میان مشکی می‌کنیم تا خارج شویم (۱۲ ورود):

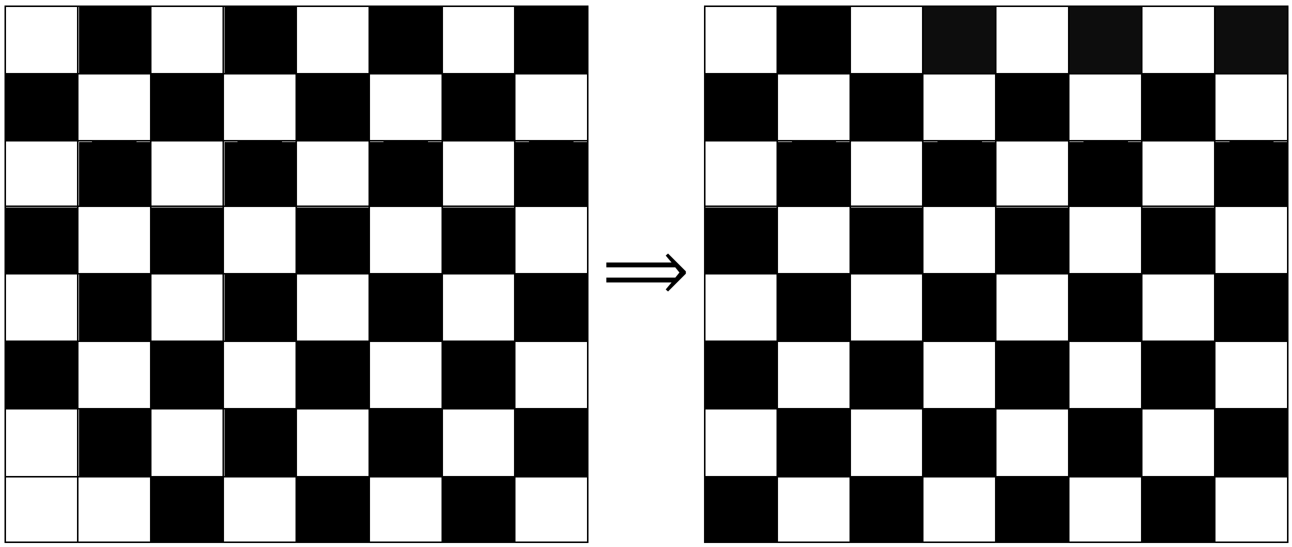




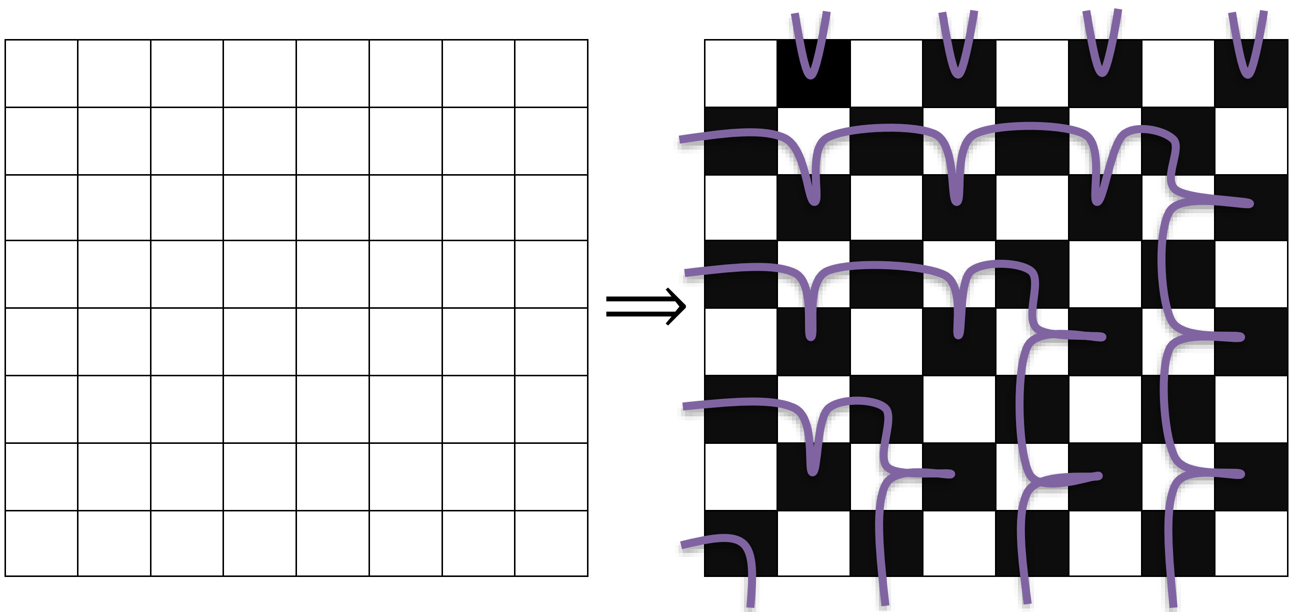
۴) از اتاق غربی بعدی وارد شده و مثل قبل به شرق و سپس جنوب حرکت می‌کنیم تا خارج شویم (۱۷ ورود):



۵) باز برای اتاق غربی بعد تکرار می‌کنیم تا خارج شویم و سپس آخرین اتاق را تنهایی مشکی می‌کنیم (۱+۹ ورود):



مسیر کلی نقاش در تمام مراحل را می‌توانیم در شکل زیر مشاهده کنیم:



با جمع تعداد ورودها در هر مرحله برای تعداد کل داریم:

۴ + ۱۳ + ۱۲ + ۱۷ + ۹ + ۱ = ۵۶ < ۶۰

مراجع: ---

۲. این الگوریتم نوشته شده است تا خارج‌قسمت و باقیمانده تقسیم یک عدد صحیح بر عدد صحیح دیگری را محاسبه کند.

```
Algorithm  Division(a, b)
// Computes recursively the quotient and remainder of a divided by b
// Input: Two large integers a and b, where b ≥ 1
// Output: The quotient and remainder of a divided by b
if  a = 0
     return  (0,0)
else
    (q, r) ← Division(⌊a/2⌋, b)
q ← 2 * q
r ← 2 * r
if  a is odd
    r ← r + 1
if  r ≥ b
    r ← r − b
    q ← q + 1
return  (q, r)
```

الف) آیا الگوریتم درست است؟ اگر درست است، درستی آن را ثابت کنید. اگر نادرست است، آن را درست و سپس درستی آن را ثابت کنید. (اثبات درستی الگوریتم، مستلزم آن است که تمام خطوط شبه‌کد را توجیه کنید.)

ب) کارایی زمانی الگوریتمِ درست چقدر است؟

جواب:

الف) الگوریتم برای مقسوم‌های بزرگ‌تر و مساوی صفر صحیح است اما برای مقسوم‌های منفی کار نمی‌کند؛ زیرا در قسمت کف گرفتن برای فراخوانی بازگشت هیچ وقت به صفر نمی‌رسد که بازگشت متوقف شد و عملا تا بی‌نهایت باز بازگشت فراخوانی می‌شود. برای تصحیح آن کافی‌ست الگوریتم را برای قدر مطلق مقسوم حل کنیم و سپس در صورت منفی بودن مقسوم اولیه با توجه به قانون تقسیم تغییرات زیر را انجام دهیم:

۱) اگر باقی‌مانده صفر بود، آنگاه خارج قسمت باید ضرب در -1 شود زیرا برای مثال داریم:

$$-9 \div 3 = -3 \text{ but } 9 \div 3 = 3$$

۲) اگر باقی‌مانده بزرگ‌تر از صفر بود، خارج قسمت ضرب و جمع با -1 شود و باقی مانده کسر آن از مقسوم‌علیه:

$$-9 \div 4 \longrightarrow q = -3, r = 3 \text{ but } 9 \div 4 \longrightarrow q = 2, r = 1$$

این تغییرات را در الگوریتم انجام می‌دهیم:

```
Algorithm: Division(a, b)
    // Computes recursively the quotient and remainder of a divided by b
    // Input: Two large integers a and b, where b ≥ 1
    // Output: The quotient and remainder of a divided by b
    c ← 1   // coefficient of a
    if a = 0 then
        return (0, 0)
    elif a < 0 then
        c ← −1   // saving negative coefficient
        a ← |a|   // making the dividend positive
    (q, r) ← Division(⌊a/2⌋, b)
    q ← 2 × q
    r ← 2 × r
    if a is odd then
        r ← r + 1
    if r ≥ b then
        r ← r − b
        q ← q + 1
    if c < 0 then   // if dividend was negative, change q
        if r = 0 then
            q ← cq
        elif r > 0 then
            q ← cq + c
            r ← b − r
    return (q, r)
```

حال برای اثبات درستی الگوریتم، پیروی آن از قانون تقسیم را بررسی می‌کنیم. در ابتدا صفر بودن مقسوم را داریم که حالت پایه است و برای توقف بازگشت در نظر گرفته شده است. پس از فراخوانی بازگشت نیز خطوط دو برابر کردن و فردیت سنجی مقسوم را داریم که به وضوح برای رفع تقسیم بر دوی مقسوم در فراخوانی بازگشتی الگوریتم است؛ زیرا تقسیم بر دو کف را گرفته است و این برای مقسوم‌های فرد یک واحد را از دست می‌دهد. همچنین بنا بر قانون تقسیم برای مقسوم a، مقسوم علیه b، خارج قسمت q و باقی‌مانده r می‌دانیم:

$$a = qb + r \xrightarrow{\times 2} 2a = 2qb + 2r$$

به شرط $r \geq b$ می‌رسیم که بخش اصلی الگوریتم است. این شرط عملا (با فراخوانی بازگشت) به طور حلقه‌ای برای هربار حذف مقسوم‌علیه از باقی‌مانده، یک عدد به خارج قسمت اضافه می‌کند. علت این بخش، رابطه‌ی درون عملیات تقسیم است. زیرا در معادله تقسیم برای مقسوم a، مقسوم علیه b، خارج قسمت q و باقی‌مانده r داریم:

$$a = qb + r = (b + b + \cdots + b) + r \to a - b - b - \cdots - b = r$$

که تعداد تکرار bها برابر با q بوده و در اصل تعداد وجود b در باقی‌مانده بعد از هر بار کم کردن میزان b از a می‌باشد. این کم کردن تا زمانی ادامه پیدا می‌کند که مقدار باقی‌مانده آنقدر کوچک شود که در آن دیگر عدد مقسوم علیه موجود نباشد (شرط $r \geq b$ را وارد نشود) و آنگاه برابر با r نهایی‌ست. بنابر تعریف داده شده برای خارج قسمت و باقی‌مانده مشاهده می‌کنیم که الگوریتم درست است و این تعریف را برای مقسوم‌های مثبت پیاده‌سازی می‌کند. همچنین این الگوریتم تغییر یافته برای مقسوم‌های کوچک‌تر از صفر نیز درست است زیرا پس از پیاده‌سازی برای قدر مطلق آن‌ها، تغییرات لازم را در خارج قسمت وارد می‌کند. این تغییرات را فرمول تقسیم می‌توانیم مشاهده کنیم:

$$\xrightarrow{r=0} a = qb \to -a = -qb$$

$$\begin{aligned}
\xrightarrow{r>0} a &= qb + r = (b + \cdots + b) + r \\
&= (b + \cdots + b) + (b + \cdots + b) - (b + \cdots + b) + b - b + r - r + r \\
&= -(b + \cdots + b) - b + (b - r) + 2\big((b + \cdots + b) + r\big) \\
&= -\big((b + \cdots + b) + b\big) + (b - r) + 2(qb + r) \\
&= -(q + 1)b + (b - r) + 2a \\
&\to -a = -(q + 1)b + (b - r)
\end{aligned}$$

پس تمام سطرهای الگوریتم جدید توجیه شده و درستی آن ثابت است.

ب) برای محاسبه‌ی کارایی زمانی الگوریتم در ابتدا به نظر می‌آید که به علت نصف شدن مقسوم در هر بازگشت، کارایی زمانی به شکل $\log a$ خواهد بود. اما نصف کردن عدد n بیتی بزرگ با نصف کردن یک آرایه متفاوت است زیرا در تقسیم بر دوی عدد، در اصل عملیات تقسیم یک بیت آن را حذف می‌کند و تعداد بازگشت‌ها با رشد عدد رابطه‌ی خطی $O(n)$ را دارد. در کنار تعداد بازگشت (یعنی عمق درخت بازگشت) زمان هر سطح بازگشت نیز اهمیت دارد. در هر سطح به طور موازی عملیات‌های جمع و تفریق را داریم. برای اعداد صحیح کوچک، زمان این عملیات‌ها ناچیز و ثابت $O(1)$ در نظر گرفته می‌شود. اما وقتی این اعداد رشد کرده و بزرگ هستند، زمان ثابت نبوده و در بدترین حالت این عمیات‌ها بر یک عدد n بیتی در زمان $O(n)$ انجام می‌شوند. از آنجا که عملیات‌ها موازی هستند، بدترین زمان یک سطح بازگشت $O(n)$ به دست می‌آید. پس کارایی زمانی کل الگوریتم برابر است با $O(n) \times O(n) = O(n^2)$.

مراجع:

- پاسخ قبل خود به سوال ۳ تمرینات سری ۳ درس طراحی و تحلیل الگوریتم‌ها ۱ (با استاد الماسی‌زاده)

۳. 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 $\mathrm{O}(n)$ خانه از خانه‌های آرایه C غیرصفر است؛ همه $\mathrm{O}(n^3)$ خانه دیگر آن صفر هستند. اگر بازیکن در طول بازی، به مکان (i,j,k) برود و $C[i,j,k]$ غیرصفر 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، بازیکن مکان دیگری مثل (i',j',k') را تصادفاً انتخاب می‌کند و ۱۰۰ امتیاز را به مقدار $C[i',j',k']$ اضافه می‌کند.

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

روش کارایی را برای نمایش آرایه C ، با استفاده از تنها $\mathrm{O}(n)$ خانه حافظه توصیف کنید. همچنین توضیح دهید که چگونه با الگوریتمی کارا مقدار هر خانه‌ی $C[i,j,k]$ را تعیین کنیم و ۱۰۰ امتیاز را به مقدار هر خانه‌ی $C[i,j,k]$ اضافه کنیم؛ مطلوب این است که بتوان مقدار هر خانه $C[i,j,k]$ را در بدترین حالت، در زمان $\Theta(\log n)$ تعیین کرد.

جواب:

از آنجا که همه خانه‌ها جز $O(n)$ خانه یعنی ضریبی از n خانه صفر هستند، به جای استفاده از آرایه‌ای ۳ بعدی برای ذخیره تمام خانه‌ها، می‌توانیم از آرایه‌ای یک بعدی برای ذخیره فقط خانه‌های دارای مقدار استفاده کنیم، که هر بار خانه‌ای مقدارش تغییر می‌کند به آن آرایه اضافه کنیم یا از آن حذف کنیم. برای سرچ پذیری بهینه آرایه، در طراحی آرایه در نظر می‌گیریم که آرایه با توجه به مختصات مرتب باشد؛ یعنی به ترتیب، اولویت با مرتب کردن بر اساس i های محور x، سپس j های محور y، و در آخر k های محور z باشد. پس فرم آرایه را داریم:

$$Data\ Array = [[i_1, j_1, k_1, C_1], [i_2, j_2, k_2, C_2], \dots, [i_n, j_n, k_n, C_n]]$$

$$\text{where } C_n > 0 \text{ and either (1) } i_{n-1} < i_n \text{ or (2) } i_{n-1} = i_n \text{ then } j_{n-1} < j_n \text{ (3) etc.}$$

حال برای تعیین مقدار هر خانه در زمان $\Theta(\log n)$، تنها کافی‌ست از الگوریتم جستجوی دودویی استفاده کنیم. با جستجوی دودویی به دنبال آن مختصات در آرایه می‌گردیم؛ اگر در آرایه موجود بود مقدار را استخراج کرده و آن مختصات را که حال مقدارش صفر شده از آرایه حذف می‌کنیم، اگر موجود نبود یعنی مقدار مختصات صفر است و به بازیکن مقداری اضافه نمی‌شود. از همین روش برای جستجو در آرایه می‌توان استفاده کرد تا به صورت بهینه مقدار ۱۰۰ را به یک خانه اضافه کنیم. یعنی با جستجوی دودویی در آرایه به دنبال مختصات می‌گردیم؛ اگر در آرایه موجود بود به آن ۱۰۰ امتیاز اضافه می‌کنیم، اگر موجود نبود در همانجا مختصات را (که مقدارش صفر بوده و حال قرار است ۱۰۰ شود) با کارایی زمانی خطی به آرایه اضافه کرده و مقدار آن را ۱۰۰ قرار می‌دهیم.

برای شبه کد جستجوی دودویی توضیح داده شده در آرایه‌ی تعریف شده داریم:

Algorithm: ***SearchGame*(*array*[0 … *n* – 1], *coordinates*)**

// Utilizes iterative binary search to find C of given coordinates in data array
// Input: Data array of the game and array of coordinates of a point
// Output: The C value of the given point in the game

$l \leftarrow 0$

$r \leftarrow n - 1$

while $l \geq r$ **do**

$m \leftarrow \lfloor (l + r)/2 \rfloor$

if *array*[*m*][0:3] = *coordinates* **then**

return *array*[*m*][3]

elif (*array*[*m*][0] > *coordinates*[0]) **or** (*array*[*m*][0] = *coordinates*[0] **and** *array*[*m*][1] > *coordinates*[1]) **or** (*array*[*m*][0] = *coordinates*[0] **and** *array*[*m*][1] = *coordinates*[1] **and** *array*[*m*][2] > *coordinates*[2]) **then**

$r \leftarrow m - 1$

else (*array*[*m*][0] < *coordinates*[0]) **or** (*array*[*m*][0] = *coordinates*[0] **and** *array*[*m*][1] < *coordinates*[1]) **or** (*array*[*m*][0] = *coordinates*[0] **and** *array*[*m*][1] = *coordinates*[1] **and** *array*[*m*][2] < *coordinates*[2]) **then**

$r \leftarrow m + 1$

// if while loop ends and coordinates are not found in data array

return 0

برای بررسی کارایی زمانی الگوریتم بالا برای تعیین مقدار هر مختصات، از آنجا که از الگوریتم جستجوی دودویی استفاده می‌شود، می‌دانیم که در بدترین حالت کارایی زمانی الگوریتم $\Theta(\log n)$ است زیرا بنا بر قضیه اصلی واکاوی الگوریتم‌ها (*Master theorem*) داریم:

$$T(n) = aT\left({}^{n}/_{b}\right) + O(n^c) = T\left({}^{n}/_{2}\right) + O(1)$$

$$\xrightarrow{yields} a = 1, b = 2, c = 0 \rightarrow \log_b a = \log_2 1 = 0 = c = 0$$

$$\xrightarrow{Case\ 2} T(n) = \Theta(n^c \log n) = \Theta(n^0 \log n) = \Theta(\log n)$$

از این الگوریتم می‌توان استفاده کرد تا به طور کارا مقدار یک مختصات را به بازیکن داد یا امتیاز ۱۰۰ را به یک مختصات اضافه کرد. در حالت اول کافی‌ست زمانی که مختصات در آرایه یافت می‌شود آن را از آرایه حذف کرد و سپس امتیاز را

برگرداند، و اگر یافت نشد، حذفی نیاز نیست و صفر برگردانده می‌شود؛ در این حالت کارایی زمانی الگوریتم تغییری نمی‌کند. در حالت دوم اگر مختصات در آرایه یافت شد تنها به آن مقدار ۱۰۰ اضافه می‌شود، اما اگر یافت نشد (یعنی اگر امتیاز مختصات صفر بود) در جایگاه m آخر یافت شده مختصات با امتیاز صد وارد می‌شود؛ در این حالت کارایی زمانی الگوریتم در بدترین حالت به $\Theta(n)$ تغییر می‌کند زیرا اضافه کردن در بدترین حالت در $O(n)$ انجام می‌شود و داریم:

$$T(n) = aT\left({}^{n}/_{b}\right) + O(n^c) = T\left({}^{n}/_{2}\right) + O(n)$$

$$\xrightarrow{yields} a = 1, b = 2, c = 1 \rightarrow \log_b a = \log_2 1 = 0 < c = 1$$

$$\xrightarrow{Case\ 3} T(n) = \Theta(n^c) = \Theta(n)$$

پس با این طرح آرایه و استفاده از جستجوی دودویی می‌توان بازی شکرستان را برای استفاده در موبایل بهینه‌سازی کرد.

مراجع: ---

۴. الف) فرض کنید $G = \langle V, E\rangle$ گرافی باشـد بی‌جهت. الگوریتمی کارا ارائه کنید برای آنکه بتوان تعیین کرد که آیا گراف G شامل حداقل دو دور هست یا خیر. کارایی زمانی الگوریتم خود را نیز اندازه بگیرید.

ب) فرض کنید $G = \langle V, E\rangle$ گرافی باشد بی‌جهت. الگوریتمی کارا ارائه کنید برای آنکه بتوان تعیین کرد آیا گراف G شامل دقیقاً دو دور مجزا (که یالی مشترک نداشته باشند) هست یا خیر. کارایی زمانی الگوریتم خود را نیز اندازه بگیرید.

جواب:

الف) برای حل این سوال و حرکت در گراف، بهترین و بهینه‌ترین راه استفاده از الگوریتم جستجوی عمقی (*DFS*) است. اما به جای استفاده از جستجوی عمقی عادی که هر رأس طی شده را علامت یکسان می‌زند، برای تشخیص دور از نوع علامت زنی سـه مرحله‌ای به جای دو مرحله‌ای اسـتفاده می‌کنیم. بدین صـورت هر رأس یا (۱) طی نشـده اسـت یعنی وجود دور یا بخشـی از دور بودن هنوز برای آن بررسـی نشـده اسـت، (۲) در حال پردازش است یعنی اگر رأس جدیدی همسایه آن باشد یک دور است، (۳) پشت سر گذاشته شده است یعنی پردازش شده و برای آن دوری ممکن نبوده است یا بوده‌است و شمرده شده است. نکته مهم این است که هدف پیدا کردن حداقل دو دور است. برای این‌کار شمارنده‌ای تعبیه کرده و زمانی که یک دور یافت می‌شـود، به آن شـمارنده اضـافه می‌شـود. اگر شـمارنده به دو رسـید الگوریتم با موفقیت متوقف می‌شـــود. اگر الگوریتم موفق به یافت دو دور نشـــد، در انتهای طی کردن گراف (وقتی همه‌ی رئوس با مرحله سه علامت گذاری شدند) توقف کرده و *False* را بر می‌گرداند.

در نوشتن الگوریتم چند نکته قابل به ذکر است:

۱) تمامی رئوس گراف در ابتدا با مرحله‌ی یک (یعنی *unvisited*) علامت گذاری شـــده‌اند. در جســـتجوی عمقی، هرگاه با رأسی طی نشده روبه‌رو می‌شویم آن را با مرحله‌ی دو (یعنی *processing*) علامت گذاری می‌کنیم و به همسایه‌هایش می‌رویم. اگر همسایه‌ای داشت که با علامت مرحله دو داشته باشد آن دور است، و اگر نداشت و دور به پایان رسید آن رأس را با مرحله سه (یعنی *visited*) علامت می‌زنیم.

۲) شـمارنده را متغیری کلی/جهانی (*global*) داریم که خودکار صفر اسـت و در صـورت یافتن دور به آن یک واحد اضافه می‌کنیم. هرگاه شمارنده حداقل ۲ شود جستجو را متوقف می‌کنیم. وگرنه ادامه می‌دهیم تا همه‌ی رئوس علامت مرحله‌ی یک را از دست بدهند.

۳) الگوریتم دارای دو بخش است؛ یکی تابع بازگشتی اصلی که برای هر رأس شروعی پیمایش رئوس متصل به آن را انجام می‌دهد تا دور پیدا کند، دیگری بدنه کلی الگوریتم که در رئوس گراف پیش رفته و آن‌هایی که بررسی نشده‌اند را به درون تابع بازگشتی می‌گذراند.

شـبه کد الگوریتم در صـفحه بعد قابل مشـاهده اسـت. برای کارایی زمانی آن می‌دانیم که جسـتجوی عمقی کارایی زمانی خطی دارد. زیرا هر رأس و هر یال تنها یک بار سـپری شـده و با علامت گذاری آنها و اجرای الگوریتم تنها برای رئوس با علامت مرحله‌ی یک (*unvisited*) رأسی تکرار نمی‌شود. پس کارایی زمانی الگوریتم $O(|V| + |E|)$ است.

```
Algorithm: CycleCounter(G, v)

    // Utilizes DFS to check whether undirected graph has at least two cycles
    // Input: Undirected graph G for cycle detection and beginning vertex v
    // Output: Returns True if at least two cycles exist, else returns False

    // global variable counter is automatically set to 0 outside function

    mark(v) ← processing
    for edge(v, w) in G.Edges(v) do   // for all w adjacent to v
        if mark(w) = processing and w ≠ parent(v) then
            counter ← counter + 1
            if counter ≥ 2 then
                return True
        elif mark(w) = unvisited then
            CycleCounter(G, w)
    mark(v) ← visited
    return False   // two cycles have not yet been detected from this starting v
--------------------------------------------------------------------------------
// to run the CycleCounter function on all vertices
global counter ← 0
for v in G do
    if mark(v) = unvisited then
        result ← CycleCounter(G, v)
        if result = True then
            break   // two cycles have been detected from the visited v's
print result   // will be True if two cycles were detected, False if none were
```

ب) به مانند قبل پیش می‌رویم اما این‌بار هر بار که دور یافت می‌شود، قبل از اضافه کردن به شمارنده باید مسیر طی شده این دور را ذخیره کنیم که اگر دور بعد یافت شد با این دور مقایسه کرده و در صورت نبود اشتراک هیچ یالی آن‌گاه به شمارنده اضافه کنیم. برای این کار نیاز به دو متغیر جهانی (*global*) دیگر داریم؛ یکی برای ذخیره رئوس گذشته شده در دور در حال شکل گرفتن (که ممکن است پذیرفته شود یا نه)، دیگری برای ذخیره دورهای تأیید شده. این دو متغیر جهانی را خارج از تابع تعریف کرده و به خصوص دقت می‌کنیم که متغیر اول برای رئوس طی شده (*parents*) هر بار که به *v* شروعی جدیدی می‌رویم و دور شکسته می‌شود از نو تعریف شود. سپس در تابع برای ذخیره‌ی دور طی شده، در لیست والدین عقب رفته و یال‌های هر جفت متوالی را می‌گیریم. شبه کد در دو صفحه بعد قابل مشاهده است.

Algorithm: *DistinctCycleCounter*(*G*, *v*)

```
// Utilizes DFS to check whether undirected graph has at least two distinct cycles
// Input: Undirected graph G for cycle detection and beginning vertex v
// Output: Returns True if at least two distinct cycles exist, else returns False

// global variables counter, path, and cycle are defined outside function

mark(v) ← processing

for edge(v, w) in G.Edges(v) do   // for all w adjacent to v
    if mark(w) = processing and w ≠ parent(v) then
        if cycle is empty then
            cycle.append([edge(v, w)])
            parent ← None and i ← 1
            while parent ≠ w do
                parent ← parents[−i] and i ← i + 1
                cycle[0].append(edge(parents[−i], parent))
            counter ← counter + 1
        else then
            tempcycle ← [edge(v, w)]
            parent ← None and i ← 1
            while parent ≠ w do
                parent ← parents[−i] and i ← i + 1
                tempcycle.append(edge(parents[−i], parent))
            if cycle[0] ∩ tempcycle = None then
                cycle.append(tempcycle)
                counter ← counter + 1
        if counter = 2 then
            return True
    elif mark(w) = unvisited then
        parents.append(v)
        DistinctCycleCounter(G, w)
mark(v) ← visited
return False   // two distinct cycles have not yet been detected from this v start
```

```
// to run the DistinctCycleCounter function on all vertices
global counter ← 0, cycle ← []   // cycle: confirmed cycles
for v in G do
    if mark(v) = unvisited then
        global parents ← [v]   // parents: resetting parents in current path
        result ← DistinctCycleCounter(G, v)
        if result = True then
            break   // two distinct cycles have been detected from visited v's
print result   // will be True if two distinct cycles were detected, False if none were
```

برای کارایی زمانی الگوریتم مشاهده می‌کنیم که حال جز جستجوی عمقی عادی، حلقه‌ای اضافه درون تابع داریم که رئوس طی شده متصل به هم در آن ذخیره شده‌اند. می‌دانیم که در بدترین حالت (زمانی که کل گراف یک دور باشد)، طول این لیست به اندازه تمام رئوس گراف بوده و حلقه‌ی پیش رفتن در آن زمان $O(|V|)$ را سپری می‌کند. از آنجا که در بدترین حالت هم ورود به این حلقه در هر بار فراخوانی تابع اتفاق نیافتاده و در تعداد معدود برای طول معدود، یا یک بار برای طول کل (زیرا با یک بار طی رئوس از حالت *unvisited* خارج می‌شوند و دوباره طی نمی‌شوند) اتفاق می‌افتد، در زمان جستجوی عمقی ضرب نشده و همچنان زمان خطی‌ست.

نکته: در مرور سوالات پیش از ارسال متوجه شدم که در سوال قسمت (ب) ۴ دقیقا دو دور خواسته شده و نه حداقل دو دور. برای تغییر الگوریتم تنها کافی‌ست شروط توقف پس از دو دور که برای بهینه‌سازی الگوریتم قرار داده شده‌اند حذف شده و لیست *cycle* به گرفتن دورهای ممکن ادامه دهد؛ بدین شکل لازم است هر بار پیش از پذیرش یک دور، اشتراک آن با دورهای پذیرفته شده در لیست دو به دو به کمک یک حلقه سنجیده شود و اگر اشتراکی نبود دور جدید پذیرفته شود. بنابراین اگر در آخر تعداد دورها (یعنی شمارنده) دو بود، *True* بدهد. البته برای بهینه کردن این الگوریتم می‌توان شرط توقف را که حذف کردیم بعد از ۳ شدن شمارنده قرار دهیم تا وقتی از ۲ گذشت خودکار *False* داده و به دنبال دورهای بیشتر نگردد.

از آنجا که در آخر وقت متوجه این تفاوت سوال شدم، موفق به اعمال تغییرات نشدم و آن‌هارا توضیح دادم. اما تغییرات کوچکی هستند و در آخر در کارایی زمانی بدترین حالت و ساختار کلی الگوریتم تفاوتی ایجاد نمی‌کنند.

مراجع:

- https://www.geeksforgeeks.org/print-all-the-cycles-in-an-undirected-graph

۵. فرض کنید که شما می‌خواهید برای حل مسأله خود، یکی از سه الگوریتم A یا B یا C را انتخاب کنید:

- الگوریتم A ، برای حل یک نمونه از مسأله با اندازه n ، آن را به پنج نمونه با اندازه نصف نمونه اصلی تقسیم می‌کند، به طور بازگشتی هر یک از نمونه‌های کوچک‌تر را حل می‌کند، و سپس در زمان خطی، جواب‌های آنها را با هم ترکیب می‌کند.
- الگوریتم B ، برای حل یک نمونه از مسأله با اندازه n ، دو نمونه کوچک‌تر با اندازه $n-1$ را به طور بازگشتی حل می‌کند، و سپس در زمان ثابت، جواب‌های آنها را با هم ترکیب می‌کند.
- الگوریتم C ، برای حل یک نمونه از مسأله با اندازه n ، آن را به نُه نمونه با اندازه $n/3$ تقسیم می‌کند، به طور بازگشتی هر یک از نمونه‌های کوچک‌تر را حل می‌کند، و سپس جواب‌های آنها را در زمان $O(n^2)$ ، با هم ترکیب می‌کند.

برحسب نماد مجانبی Θ، زمان اجرای هر یک از این الگوریتم‌ها چقدر است؟ شما کدام الگوریتم را انتخاب خواهید کرد؟

جواب:

ابتدا کارایی زمانی هر کدام از الگوریتم‌ها را محاسبه می‌کنیم:

- **الگوریتم A:** در این الگوریتم پنج عملیات بزرگ بازگشت داریم که اندازه هر کدام ${}^{n}/_{2}$ است. همچنین یک جمع در زمان n برای پنج بخش داریم. بنابراین برای هر سطح بازگشت به طور کلی داریم:

$$T(n) = T\left({}^{n}/_{2}\right) + T\left({}^{n}/_{2}\right) + T\left({}^{n}/_{2}\right) + T\left({}^{n}/_{2}\right) + T\left({}^{n}/_{2}\right) + T(n) = 5T\left({}^{n}/_{2}\right) + T(n)$$

اما از آنجا که الگوریتم بازگشتی‌ست و عمقی دارد (یعنی سطح تکرار می‌شود تا به پایان برسد، و در اصل عمق را برابر $\log_2 n$ داریم) برای محاسبه کارایی الگوریتم از قضیه اصلی واکاوی الگوریتم‌ها (*Master theorem*) در الگوریتم‌های بازگشتی استفاده می‌کنیم. از آنجا که اندازه کلی به ۵ بخش با اندازه ${}^{n}/_{2}$ تقسیم شده و تابع اضافه جمع اندازه n^1 دارد، می‌نویسیم:

$$T(n) = aT\left({}^{n}/_{b}\right) + O(n^c) = 5T\left({}^{n}/_{2}\right) + O(n)$$

$$\xrightarrow{yields} a = 5, b = 2, c = 1 \to \log_b a = \log_2 5 = 2.3219280949 > c = 1$$

$$\xrightarrow{Case\ 1} T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 5}\right) = \Theta(n^{2.3219280949})$$

بنابراین با استفاده از حالت اول قضیه، با فرمول کارایی زمانی بر حسب نماد مجانبی را داریم.

- **الگوریتم B:** در این الگوریتم تنها دو عملیات بزرگ بازگشت است که اندازه هر کدام $n-1$ است. جمع آخر در زمان ثابت بوده و اثری ندارد. از آنجا که تقسیمی در هر بخش انجام نشده، از قضیه قبل استفاده نمی‌توانیم کرده و با نوشتن حالت کلی یک سطح و سپس محاسبه آن برای عمق درخت بازگشت پیش می‌رویم. برای شکل محاسبه کارایی یک سطح درخت داریم:

$$T(n) = T(n-1) + T(n-1) + T(1) = 2T(n-1)$$

می‌دانیم که بازگشت با کم کردن یکی یکی تا جایی پیش می‌رود که n تمام شود. پس عمق درخت بازگشت برابر با n است. حال برای کارایی زمانی کل بایستی کارایی زمانی تمام سطح‌های درخت را جمع کنیم؛ یعنی شکل کلی یک سطح به تعداد عمق درخت بازگشت تکرار می‌شود. پس می‌نویسیم:

$$T(n) = T(n-1) \times n = \Theta\big(n(n-1)\big) = \Theta(n^2)$$

- **الگوریتم C:** از آنجا که هر بار الگوریتم برای بازگشت تقسیمی بر اندازه انجام می‌دهد، فرم قضیه اصلی واکاوی الگوریتم‌ها را برای الگوریتم‌های بازگشتی داشته و از آن استفاده می‌کنیم:

$$T(n) = aT\left({}^{n}/_{b}\right) + O(n^c) = 9T\left({}^{n}/_{3}\right) + O(n^2)$$

$$\xrightarrow{yields} a = 9, b = 3, c = 2 \rightarrow \log_b a = \log_3 9 = 2 = c = 2$$

$$\xrightarrow{Case\ 2} T(n) = \Theta(n^c \log n) = \Theta(n^2 \log n)$$

مشاهده کردیم که کارایی زمانی این الگوریتم با حالت دوم قضیه محاسبه شد.

از آنجا که الگوریتم کاراتر و بهینه‌تر همواره اولویت و کاربر بیشتری دارد، با مقایسه کارایی زمانی الگوریتم‌ها در میابیم که الگوریتم B انتخاب بهتری می‌باشد زیرا کارایی زمانی کمتری دارد.

مراجع: ----