تمرینات فصل سه

|  |  |  |
| --- | --- | --- |
| نام: امیرحسام بهمن‌خواه | نام: مریم رضائی | نام: محمدحسین فرخی |

**1.** الگوریتم ساده­اندیشانه برای مسأله تطابق رشته را در نظر بگیرید:

**الف)** الگوریتم را به گونه­ای تغییر دهید که در صورت وجود الگو در متن، مکان نویسه­ای از متن را که با سمت راست­ترین نویسه الگو تطابق یافته است، برگرداند.

**ب)** الگوریتم را به گونه­ای تغییر دهید که نویسه­های الگو را از راست به چپ پردازش ­کند، نه از چپ به راست.

**پ)** مبتنی بر این الگوریتم، الگوریتمی بنویسید که با آن بتوان تعداد دفعات وقوع یک الگو در یک متن را تعیین کرد؛ الگوها نباید با یکدیگر اشتراک داشته باشند. کارایی زمانی الگوریتم خود را نیز تعیین کنید.

**ت)** با این فرض که همه نویسه­های الگو متفاوت با یکدیگر باشند، الگوریتم را به گونه­ای تغییر دهید که کارایی زمانی آن شود.

**جواب:**

**1 – الف)** می‌خواهیم که به جای اولین کاراکتر، آخرین کاراکترِ اولین الگوی یافت شده در متن را برگرداند، پس باید *i* یافت شده را با طول الگوی P جمع کنیم. در نتیجه خط 6 که return *i* است باید return *i* + *m* بشود.

**2 – ب)** برای برعکس کردن حرکت بایستی نقطه شروع حلقه را از پایان متن قرار داده و به طور کلی حلقه‌ها را برعکس کنیم تا به جای حرکت چپ به راست الگوریتم در متن و الگو، از راست به چپ حرکت کنیم. یعنی:

**Algorithm: BruteForceStringMatch(*T*[0...*n* – 1], *P*[0...*m* – 1])**

// implements brute-force string matching right to left

**for** *i* ← *n* **to** *m* **do**  // we know *m* < *n* so we are moving with – 1 steps

*j* ← *m* – 1

**while** *j* ≥ 0 **and** *P* [ *j* ] = *T* [*i* – (*m* – *j*)] **do**

*j* ← *j* – 1

**if** *j* = 0 **then**

**return** *i* – *m*

**return** – 1

**1 – پ)** به جای بازگرداندن جایگاه کاراکتر، هر بار که یک الگو در متن یافت شد به شمارنده یکی اضافه کرده و به اندازه‌ی طول الگو به جلو پرش می‌کنیم تا جستجو را از بعد از پایان الگو ادامه دهیم و در الگوهای یافت شده کاراکتری مشترک نبوده و دو بار بررسی نشود. یعنی:

**Algorithm: BruteForceStringMatch(*T*[0...*n* – 1], *P*[0...*m* – 1])**

// returns number of repetitions of pattern *P* in text *T*

*i* ← 0

*count* ← 0

**while** *i* ≤*n* –*m* **do**

*j* ← 0

**while** *j* < *m* **and** *P* [ *j* ] = *T* [*i* + *j*] **do**

*j* ← *j* + 1

**if** *j* = *m* **then**

*count* ← *count* + 1

*i* ← *i* + *m* – 1

*i* ← *i* + 1

**return** *count*

**1 – ت)** کارایی زمانی الگوریتم موجود در سوال در بدترین حالت که تمامی کاراکترهای متن و الگو یکی باشند اتفاق می‌افتد. اگر بدانیم تمام کاراکترهای الگو متفاوفت هستند، می‌توانیم پس از هربار بررسی وجود الگو رسیدن و موفق نشدن، با پرش کردن به اندازه‌ی طول کاراکترهای مقایسه شده الگورتم را کاراتر کنیم؛ زیرا می‌دانیم که کاراکتر اول الگو در قطعاً در این چند کاراکتر یکی شده موجود نخواهد بود.

**Algorithm: BruteForceStringMatch(*T*[0...*n* – 1], *P*[0...*m* – 1])**

// provided that all the characters of the pattern are different

*i* ← 0

**while** *i* <*n* –*m* **do**

*j* ← 0

**while** *j* < *m* **and** *P* [ *j* ] = *T* [*i* + *j*] **do**

*j* ← *j* + 1

**if** *j* = *m* **then**

**return** *i*

**elif** *j* ≠ 0 **then** // comparisons started but failed to match full string

*i* ← *i* + *j* – 1

*i* ← *i* + 1

**return** – 1

**2.** الگوریتم تعریف – مبنا (ساده­اندیشانه) برای ضرب دو ماتریس مربعی را در نظر بگیرید:

**الف)** اگر ماتریسی و ماتریسی باشد، الگوریتم را به نحوی تغییر دهید که با آن بتوان حاصل­ضرب دو ماتریس مستطیلی و را به دست آورد. تعداد دقیق ضرب­ها و تعداد دقیق جمع­های الگوریتم جدید چیستند؟ کارایی زمانی الگوریتم چقدر است؟

**ب)** اگر یک ماتریس بالامثلثی بوده و نیز یک ماتریس بالامثلثی باشد، الگوریتم را به نحوی تغییر دهید که با آن بتوان حاصل­ضرب دو ماتریس بالامثلثی و را به دست آورد؛ الگوریتم شما نباید عناصری از ماتریس حاصل­ضرب را که پیشاپیش صفر بودن آنها مشخص است، محاسبه کند. تعداد دقیق ضرب­ها و تعداد دقیق جمع­های الگوریتم جدید چیستند؟ کارایی زمانی الگوریتم چقدر است؟

**جواب:**

**2 – الف)** برای تغییر الگوریتم ضرب دو ماتریس به صورتی که ماتریس‌های مستطیلی را با آن بتوان ضرب کرد، اگر ماتریس‌های ورودی *An×m* و *Bm×p* باشند، تنها کافی‌ست که محدوده‌ی حلقه‌ها را به طوری تغییر دهیم تا درونی‌ترین حلقه از 0 تا یکی کمتر از بُعد یکسان ماتریس‌ها پیش رود.

**Algorithm: MatrixMultiplication(*A*[0...*n*-1][0...*n*-1], *B*[0...*n*-1][0...*n*-1])**

// multiplies two non-square matrices provided that they have one equal dimension

// Input: two matrices *An×m* and *Bm×p*

// Output: matrix *Cn×p*

*C*← matrix(*n* × *p*)

**for** *i* ← 0 **to** *n* – 1 **do**

**for** *j* ← 0 **to** *p* – 1 **do**

*C*[*i*][*j*] ← 0

**for** *k* ← 0 **to** *m* – 1 **do**

*C*[*i*][*j*] ← *C*[*i*][*j*] + *A*[*i*][*k*] × *B*[*k*][*j*]

**return** *C*

می‌بینیم که ضرب و جمع در یک خط انجام می‌شوند و پس تعداد تکرارشان یکی‌ست. از طرفی هر دو اعمال ساده هستند و به همین علت در سیگما عدد یک را قرار می‌دهیم. پس تعداد تکرار هرکدامشان برابر است با:

از آنجا که تعداد دقیق تکرار عمل جمع یا ضرب هر کدام برابر با *nmp* است، هرکدام را که به عنوان عمل پایه در نظر بگیریم بدترین حالت زمانی رخ می‌دهد که *n* = *m* = *p* و بنابراین مقدار *nmp* = *n*3 باشد؛ پس کارایی زمانی الگوریتم O(*n*3) می‌باشد با فرض اینکه که پارامتر *n* بزرگترین میان سه بعد *n* و *m* و *p* است.

**2 – ب)** برای محاسبه حاصل‌ضرب دو ماتریس مربعی بالامثلثی بدون محاسبه‌ی صفرها، می‌دانیم که برای عنصر در جایگاه (*i*, *j*) هرگاه *i* > *j* باشد آن عنصر صفر است، پس قطعا اجرای اعمال ضرب و جمع بر آن‌ها همواره حاصلی برابر با صفر دارد. بر این اساس فقط زمانی که *i ≤ j* باشد وارد حلقه‌ی سوم می‌شویم. همچنین برای عناصر، هرگاه *k* > *j* باشد، به شکل قبل عمل ضرب با صفر حاصلی جز صفر ندارد. بنابراین فقط برای *k*های کوچکتر مساوی با *j* محاسبات را انجلم می‌دهیم. در نتیجه:

**Algorithm: UpperTriangularMatrixMultiplication(*A*[0...*n*-1][0...*n*-1], *B*[0...*n*-1][0...*n*-1])**

// multiplies two upper-triangular matrices

// Input: two matrices *An×n* and *Bn×n*

// Output: matrix *Cn×n*

*C*← matrix(*n* × *n*)

**for** *i* ← 0 **to** *n* – 1 **do**

**for**  *j* ← 0 **to** *n* – 1 **do**

*C*[*i*][*j*] ← 0

**if** *i* ≤ *j* **then**

**for** *k* ← 0 **to** *j* **do**

*C*[*i*][*j*] ← *C*[*i*][*j*] + *A*[*i*][*k*] × *B*[*k*][*j*]

**return** *C*

به مانند بخش الف، تکرار عمل جمع و ضرب برابر می‌باشد. تنها تفاوت در محاسبه‌ی تعداد آن‌ها در محدوده‌ی سیگماها است که برای دومین و سومین سیگما، به ترتیب به *i* تا *n* – 1 و 0 تا *j* تغییر یافته‌اند:

با مقایسه‌ی حاصل با بخش قبل مشاده می‌کنیم که تعداد تکرار عمل پایه از حالت قبل کمتر است (که باید اینطور باشد زیرا به علت صفرها، اعمال کمتری انجام می‌شوند) اما نرخ رشد هردو همواره یکی بوده و کارایی زمانی باز O(*n*3) می‌باشد.

**3.** این الگوریتم را، که با راهبردی ساده­اندیشانه مسأله­ای را حل می­کند، در نظر بگیرید:

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

**ب)** درستی الگوریتم را ثابت کنید.

**پ)** با این فرض که باشد، این ادعا را که کارایی زمانی الگوریتم است، توجیه کنید.

**جواب:**

**3 – الف)** با فرض *a* = 7 و *b* = 3 و ردیابی برنامه مشاهده می‌کنیم که *x* = 2 و *y* = 1 می‌شود؛ بنابراین الگوریتم، خارج قسمت و باقی‌مانده‌ی تقسیم *a* بر *b* را محاسبه می‌کند و آن‌ها را به ترتیب به صورت زوج مرتبی خروجی می‌دهد که *x* خارج قسمت و *y* باقی‌مانده است.

**3 – ب)** بنابر معادله‌ی تقسیم برای مقسوم *a*، مقسوم علیه *b*، خارج قسمت *x* و باقی‌مانده *y* داریم:

که تعداد تکرار *b*ها برابر با *x* بوده و در اصل تعداد وجود *b* در عدد 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 عدد دیگر مقسوم علیه موجود نباشد و آنگاه برابر با *y* نهایی‌ست. بنابر تعریف داده شده برای خارج قسمت و باقی‌مانده مشاهده می‌کنیم که حلقه‌ی درون الگوریتم درست است و این تعریف را برای مقسوم‌های مثبت پیاده‌سازی می‌کند. همچنین این الگوریتم برای مقسوم‌های کوچک‌تر از صفر نیز درست است زیرا:

می‌بینیم که در الگوریتم، برای مقسوم‌های منفی پس از محاسبه‌ی خارج قسمت و باقی‌مانده‌ی قدر مطلق آن‌ها، در شرط اول و دوم به ترتیب دو حالت اثبات شده در بالا پیاده‌سازی می‌شود. پس الگوریتم درست است.

**3 – پ)** می‌دانیم تعداد تکرارهای حلقه‌ی *while* به تعداد تفریق‌های انجام شده در آن است، یعنی به تعداد *x* که همان حاصل تقسیم صحیح *a* بر *b* می‌باشد؛ بنابراین الگوریتم خطی بوده و کارایی زمانی آن هم مرتبه با O(*n*) می‌باشد که پارامتر *n* در اصل بزرگترین عدد میان *a* و *b* است و چون *a* > *b* پس *n* = *a* و O(*n*) = O(*a*). حال می‌خواهیم ثابت کنیم O(*x*log*a*) ∈ O(*a*)، پس با فرض درستی آن می‌نویسیم:

و بنابراین از هر دو جهت به یک نامساوی رسیده و اثبات صحت ادعا کامل است.

**4. الف)** فرض کنید یک چندضلعی رأسیِ محدب باشد. الگوریتمی با کارایی زمانی طراحی کنید که را به عنوان ورودی بگیرد و مساحت آن را به عنوان خروجی برگرداند.

**ب)** فرض کنید یک چندضلعی رأسیِ ساده باشد؛ ممکن است محدب باشد یا نباشد. الگوریتمی با کارایی زمانی طراحی کنید که را به عنوان ورودی بگیرد و مساحت آن را به عنوان خروجی برگرداند.

**جواب:**

**4 – الف)** برای محاسبه مساحت چندضلعی محدب *n* راسی از اصل هرون در هندسه برای محاسبه مساحت مثلث استفاده می‌کنیم که با آن بدون داشتن ارتفاع می‌توانیم مساحت را بیابیم. برای این کار ابتدا نیاز است چندضلعی را مثلث‌بندی کنیم و سپس با محاسبه مساحت هرکدام با اصل هرون، مساحت‌ها را جمع کنیم. شبه کد الگوریتم در صفحه‌ی بعد قابل مشاهده است.

برای کارایی زمانی آن مشاهده می‌کنیم که دو حلقه‌ی مستقل در الگوریتم وجود دارد، پس کارایی زمانی هم مرتبه با بزرگترین میان آن مقادیر خواهد بود. از آنجا که به ازای پارامتر *n* (تعداد رئوس چندضلعی) حلقه‌ی اول *n* – 2 بار تکرار می‌شود و حلقه‌ی دوم نیز به تعداد مثلث‌های تشکیل شده در حلقه‌ی اول تکرار می‌شود (که آن هم برابر با *n* – 2 است)، برای کارایی زمانی الگوریتم داریم:

*max*( *f*(*n*), *g*(*n*)) ∈ O( *f*(*n*) + *g*(*n*)) →O(*n* – 2 + *n* – 2) = O(2*n* – 4) ∈ O(*n*)

**Algorithm: ConvexPolygonArea(*Polygon*)**

// finds area of convex polygon using triangulation and Heron’s formula

// Input: intended *Polygon* which is an array of *Points*

// Output: area of polygon

// class *Point* is an ordered pair with attributes *Point*.x and *Point*.y

*triangles* ← list()

*points* ← *Polygon*

**while** length(*points*) ≥ 3 **do**

*triangles*.add(list(*points*[0], *points*[1], *points*[2]))

*points*.remove(2)

*area* ← 0

**for** *T* **in** *triangles* **do**

*a* ← ((*T*[0].x – *T*[1].x)2 + (*T*[0].y – *T*[1].y)2)1/2

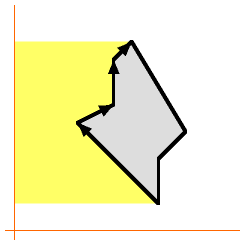
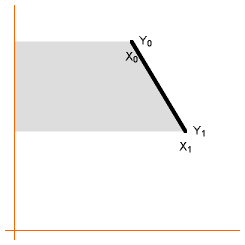
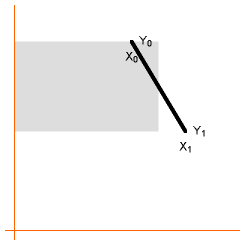
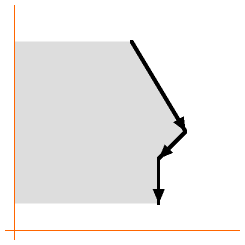
*b* ← ((*T*[2].x – *T*[1].x)2 + (*T*[2].y – *T*[1].y)2)1/2

*c* ← ((*T*[2].x – *T*[0].x)2 + (*T*[2].y – *T*[0].y)2)1/2

*p* ← (*a* + *b* + *c*) / 2

*area* ← *area* + (p × (p – *a*) × (p – *b*) × (p – *c*))1/2

**return** *area*

**4 – ب)** طبق اصل محاسبه گراف‌های هندسیTحاصل ضرب اختلاف عرض دو نقطه در نصف اجتماع طول آن‌ها یعنی ((*x*1 + *x*2) × (*y*1 – *y*2)) / 2، مساحت سایه‌ی خط واصل آن دو نقطه بر محور عرض‌ها را می‌دهد. در نتیجه، اگر مساحت مذکور را برای تمام جفت راس‌های مجاور یک چندضلعی در جهت ساعت گرد محاسبه کنیم، مساحت کل چندضلعی محاسبه می‌شود. تکرار حلقه برابر تعداد راس‌هاست، پس کارایی زمانی برابر با O(*n*) می‌باشد. شکل‌های زیر از چپ به راست الگوریتم را نشان می‌دهند (ناحیه‌ی خاکستری محاسبه می‌شود):

**Algorithm: PolygonArea(*Polygon*)**

// finds area of any polygon

// Input: intended *Polygon* which is an array of *Points*

// Output: area of polygon

// class *Point* is an ordered pair with attributes *Point*.x and *Point*.y

// and *Polygon.point.next* returns the next point clockwise

*area* ← 0

**for** *point* **in** *Polygon* **do**

*area* ← *area* + ((*point*.x + *point*.next.x) × (*point*.y – *point*.next.y)) / 2

**return** *area*

**5. الف)** فرض کنید هر یک از مجموعه ، خود زیرمجموعه­­ای از مجموعه باشد. الگوریتمی ساده­اندیشانه طراحی کنید که با آن بتوان مشخص کرد که آیا دو زیرمجموعه مجزا از یکدیگر، در میان این زیرمجموعه­ها وجود دارد یا خیر.

**ب)** عمل پایه­ای الگوریتمی که طراحی کرده­اید، چیست؟ حداکثر تعداد دفعات تکرار این عمل چقدر است؟

**پ)** کارایی زمانی الگوریتم ساده­اندیشانه را با نماد مجانبی بیان کنید.

**ت)** چگونه می­توان برای حل مسأله، الگوریتمی کاراتر از الگوریتم ساده­اندیشانه طراحی کرد؟

**جواب:**

**5 – الف)** از آنجایی که الگوریتم ساده‌اندیشانه است، با دو حلقه در مجموعه‌ها پیش رفته و با انتخاب دو تا از آن‌ها هر بار تمامی اعضایشان را مقایسه می‌کنیم. همچنین چون قصدمان پیدا کردن فقط دو مجموعه‌ی مجزا است، با اولین برخورد به مجموعه‌‌هایی که مجزا هستند الگوریتم را متوقف می‌کنیم. این عمل را به کمک متغیر *disjoint* انجام می‌دهیم که در ابتدا قبل از پیش رفتن در اعضای دو مجموعه‌ی انتخابی آن را برابر با رشته‌ی *yes* قرار می‌دهیم و در صورت یافتن اعضای یکسان به *no* تغییر می‌دهیم و در پایان هر یک از دو حلقه‌ی مقایسه، مقدار آن را بررسی می‌کنیم. شبه کد الگوریتم در صفحه‌ی بعد قابل مشاهده می‌باشد:

**Algorithm: AreSetsDisjoint(*S*1, *S*2, …, *Sn*)**

// determines whether two disjoint sets exist between the entries

// Input: intended *n* sets

// Output: returns *True* if two are disjoint, else returns *False*

**for** *i* ← 1 **to** *n* **do**

**for** *j* ← *i* **to** *n* **do**

*disjoint* ← ‘yes’

**for** *x* **in** *S*[*i*] **do**

**for** *y* **in** *S*[*j*] **do**

**if** *x* = *y* **then**

*disjoint* ← ‘no’

**break**

**if** *disjoint* = ‘no’ **then**

**break**

**if** *disjoint* = ‘yes’ **then**

**return** *True*

**return** *False*

**5 – ب)** از آنجا که الگوریتم اعضای مجموعه‌ها را مقایسه می‌کند، عمل مقایسه در خط شش را که درونی‌ترین عمل الگوریتم است به عنوان عمل پایه در نظر می‌گیریم. بنابراین برای حداکثر تعداد تکرار آن عمل نیاز است که بدترین حالت را محاسبه کنیم، یعنی زمانی که هیچ دو مجموعه‌ی مجزایی میان مجموعه‌ها وجود نداشته باشند (تا الگوریتم مجبور شود تمامی اعضا را بررسی کند) و همه‌ی مجموعه ها حداکثر تعداد عضو که *n* *می‌باشد (زیرا زیرمجموعه‌ی مجموعه‌ای n عضوی هستند) را داشته باشند.* پس برای تعداد تکرار داریم:

**5 – پ)** از آنجایی که تعداد تکرار عمل پایه یعنی عمل مقایسه، در بدترین حالت چندجمله‌ای از مرتبه‌ی چهار به دست آمد، بنابراین کارایی زمانی الگوریتم ساده‌اندیشانه هم مرتبه با O(*n*4) می‌باشد.

**5 – ت)** برای الگوریتمی کاراتر از الگوریتم کند ساده‌اندیشانه قطعا باید از تعداد حلقه‌های تودرتوی الگوریتم و یا تعداد تکرار آن‌ها کاست. برای این کار می‌توانیم به جای دو حلقه‌ی درونی الگوریتم که اعضای دو مجموعه‌ی انتخابی را با همدیگر مقایسه می‌کنند، از روش دیگری استفاده کنیم. یعنی ابتدا یکی از مجموعه‌ها را با الگوریتم Merge Sort مرتب کرده (که کارایی زمانی آن *n*log*n* می‌باشد)، سپس با حلقه‌ای در مجموعه‌ی دیگر پیش رفته (که تکرار آن به اندازه‌ی طول مجموعه است که در بدترین حالت آن را *n* فرض می‌کنیم) و هر عضو آن را با یک جستجوی دودویی در مجموعه‌ی مرتب شده درون حلقه، با اعضای مجموعه‌ی اول مقایسه می‌کنیم (که این حلقه‌ی درونی کارایی log*n* دارد). پس دو حلقه‌ی درونی الگوریتم از *n*2 به 2*n*log*n* تغییر میابد.

O(*n*2.2*n*log*n*) = O(2*n*3log*n*) ∈ O(*n*3log*n*)

**6.** قرار است در یک روز خاص، تعداد زیادی سخنرانی علمی که زمان­­های شروع و زمان­های پایان هر یک از آنها مشخص است، در تالارهای یک دانشگاه برگزار شود.

یکی از تالارهای دانشگاه بزرگ­تر از بقیه است و به همین دلیل، برگزار کنندگان می­خواهند بیشترین تعداد ممکن از سخنرانی‌ها را در آن تالار برگزار کنند. از آنجا که زمان­های برگزاری بعضی از سخنرانی­ها با هم تداخل دارند، مسأله برگزار کنندگان، انتخاب بیشترین تعداد از سخنرانی­ها است به گونه­ای که هیچ دو سخنرانی با هم تداخل نداشته باشند. گرچه هر سخنرانی باید بی‌وقفه برگزار شود، اما به محض پایان یافتن یک سخنرانی می­توان سخنرانی بعدی را شروع کرد.

**الف)** الگوریتمی ساده­اندیشانه برای این مسأله طراحی کنید. فرض کنید الگوریتم، عدد را (که تعداد سخنرانی‌ها است) ، مجموعه­‌ی را (که شامل زمان­های شروع سخنرانی­ها است) و مجموعه‌ی را (که شامل زمان­های پایان سخنرانی­ها است) به عنوان ورودی می­گیرد.

**ب)** کارایی زمانی الگوریتم ساده­اندیشانه را با نماد مجانبی بیان کنید.

**جواب:**

**6 – الف)** برای نوشتن الگوریتم ساده‌اندیشانه با فرض اینکه عضو اول دو مجموعه، زمان شروع و پایان یک سخنرانی یکسان باشند، با استفاده از الگوریتم ساده‌اندیشانه‌ی Bubble Sort مجموعه‌ی *S* را مرتب کرده و همزمان همان اعمال را بر مجموعه‌ی *F* انجام می‌دهیم تا جایگاه شروع و پایان هر سخنرانی در مجموعه‌ها حفظ شوند. سپس اعضای هر دو را به ترتیب به در زوج مرتب‌هایی قرار می‌دهیم تا لیستی از زوج مرتب‌هایی به دست آوریم که هر عضو، زمان شروع و پایان سخنرانی را شامل می‌شود؛ این زوج مرتب‌ها همچنین بر اساس زودترین زمان شروع به دیرترین، منظم در لیست قرار دارند.

حال با انتخاب اولین زوج مرتب به عنوان اولین سخنرانی و سپس گزینش اولین سخنرانی‌ای که بعد از زمان پایان سخنرانی اول شروع می‌شود، حلقه را پیش برده و مجموعه‌ای از بیشترین سخنرانی‌های ممکن ایجاد می‌کنیم. این عمل را برای انتخاب‌های متفاوت به عنوان سخنرانی اول روز و یا سخنرانی‌های بعد از آن تکرار می‌کنیم تا تمامی حالات ممکن بررسی شود و هر یک مجموعه‌های حاصل را در لیست نهایی قرار می‌دهیم. در آخر با یافتن بزرگترین مجموعه (مجموعه‌ای که بیشترین تعداد سخنرانی‌ها را دارد)، آن را خروجی می‌دهیم.

شبه کد الگوریتم پاسخ، در صفحه‌ی بعد قابل مشاهده است.

**6 – ب)** این الگوریتم ساده‌اندیشانه دارای بخش‌های مستقل و تعداد حلقه‌های تودرتوی زیادی‌ست. اولین بخش مستقل الگوریتم، مرتب‌سازی حبابی‌ست که با دو حلقه کارایی آن O(*n*2) است. دومین بخش، حلقه‌ای با کارایی O(*n*) است. سومین و بزرگترین بخش الگوریتم که سه حلقه به شکل دارد در بدترین حاله (که شرط همواره برقرار باشد، دارای کارایی زمانی O(*n*3) بوده و بخش چهارم حلقه‌ای‌ست که به تعداد حالت‌های کلی ممکن برای سخنرانی‌ها تکرار می‌شود و در هر حال خطی می‌باشد. حال می‌دانیم که برای محاسبه کارایی الگوریتمی با چند بخش مستقل، باید *max* مقادیر کارایی را در نظر بگیریم؛ یعنی:

*max*( *f*(*n*), *g*(*n*), *h*(*n*), *k*(*n*)) ∈ O( *f*(*n*) + *g*(*n*) + *h*(*n*) + *k*(*n*)) →O(*n*2+ *n* + *n*3 + *n*) ∈ O(*n*3)

**Algorithm: Schedule(*n*, *S*[0…*n* – 1], *F*[0…*n* – 1])**

// finds best schedule with most presentations

// Input: *n* number of presentations, *S* set of start time, *F* set of end time

// Output: returns list of most possible presentations

**for** *i* ← 0 **to** *n* – 1 **do** // Bubble Sort

**for** *j* ← 0 **to** *n* – *i* – 2 **do**

**if** *S*[*j*] > S[*j* + 1] **do**

*saved* ← *S*[*j*]

*S*[*j*] ← *S*[*j* + 1]

*S*[*j* + 1] ← *saved*

*dualsaved* ← *F*[*j*]

*F*[*j*] ← *F*[*j* + 1]

*F*[*j* + 1] ← *dualsaved*

*all* ← list()

**for** *i* ← 0 **to** *n* – 1 **do**

*all*.append( [*S*[*i*], *F*[*i*]] )

*superset* ← list()

**for** *i* ← 0 **to** *n* – 1 **do**

*main* ← *all*[*i*]

**for** *j* ← *i* + 1 **to** *n* – 1 **do**

*subset* ← list(*main*)

**if** *all*[*j*][0] ≥ *main*[1] **then**

*subset*.append(*all*[*j*])

*temp* ← *all*[*j*]

**for** *k* ← *j* + 1 **to** *n* – 1 **do**

**if** *all*[*k*][0] ≥ *temp*[1] **then**

*temp* ← *all*[*k*]

*subset*.append(*temp*)

*superset*.append(*subset*)

*maxList* ← *superset*[0][0]

*maxNumber* ← length(*maxList)*

**for** *subset* **in** *superset* **do**

**if** length(*subset*) ≥ *maxNumber* **then**

*maxNumber* ← length(*subset*)

*maxList* ← *subset*

**return** *maxList*

**7.** شهرداری یک شهر بزرگ، تصمیم گرفته است که با انجام پروژه­ای بی­نظیر، همه تقاطع­های شهر را، آن هم در سریع­ترین زمان ممکن، تعمیر و زیباتر کند. در زمان تعمیر هر تقاطع، آن تقاطع کاملاً بسته خواهد شد؛ و علاوه بر آن، هر خیابانی نیز که از دو طرف به دو تقاطع بسته منتهی شده باشد، کاملاً بسته خواهد شد. بنابراین، یکی از موانع بر سر راه انجام پروژه، این است که با بسته شدن همزمان بعضی از تقاطع­ها، رفتن خودروها از نقاطی از شهر به نقاطی دیگر (البته از طریق خیابان­ها نه کوچه­ها!) غیرممکن خواهد شد.

فرض کنید همه خیابان­های شهر دو طرفه باشند و اگر خیابانی فقط از یک طرف بسته شده باشد، خیابانی باز به حساب خواهد آمد؛ یعنی می­توان با خودرو، از تقاطع باز، وارد خیابان شد و تا تقاطع بسته حرکت کرد و از تقاطع بسته به طرف تقاطع باز برگشت! مسأله کارشناسان شهرداری، تعیین بزرگ­ترین مجموعه (از نظر تعداد) از تقاطع­هایی است که در صورت بسته شدن همزمان آنها، باز بتوان با خودرو، از هر نقطه­ای از شهر به هر نقطه دیگری طی طریق کرد. با در دست داشتن چنین مجموعه­ای، می­توان در مرحله اول پروژه و به طور همزمان، به تعمیر آن تقاطع­ها پرداخت.

**الف)** الگوریتمی برای این مسأله طراحی کنید.

**ب)** با این فرض که تعداد تقاطع­های شهر باشد و تعداد خیابان­های شهر باشد، کارایی الگوریتم­تان را برحسب و تعیین کنید.

**جواب:**

**7 – الف)** می‌توانیم با کمک جستجوی عمقی و تعریف ویژگی‌هایی جدید برای گراف (یعنی شهر که راس‌های آن تقاطع‌ها و یال‌های آن خیابان‌ها باشند)، نقاط همبندی گراف را پیدا کنیم که در صورت حفظ آن‌ها و بستن بقیه‌ی نقاط، گراف همچنان همبند باشد. برای این‌کار باید نقاطی را بیابیم که یا به آن‌ها فقط برگ وصل شده است و یا مسیری برای برگشت از آن نقاط به نیاکان نقطه نباشد.

سپس با بررسی نقاط به دست آمده، تعیین کنیم که آیا دو راس دو سر یک یک یال بسته شده‌اند یا خیر و اگر این طور هست راسی را باز کنیم که بیشترین تعداد یال بسته از آن خارج شده است و در این صورت کمترین تعداد ممکن برای راس باز را بیابیم. الگوریتم کلی که از دو بخش مستقل غیربازگشتی و یک الگوریتم بازگشتی تشکیل شده است در سه بخش به عنوان سه الگوریتم در صفحه‌های بعد قرار گرفته است.

**7 – ب)** کارایی زمانی یافتن نقاط همبندی که بر اساس پیمایش عمقی نوشته شده است برابر با O(*m* + *n*) و کارایی زمانی بررسی رفع مشکل حذف یال بنابر وجود یک حلقه‌ی دارای شرطی با *any* که دو حلقه است، برابر با O(*n*(*m* – *APs*)2) می‌باشد که *Aps* نقاط همبندی یافت شده است.

**Algorithm: LeastOpenIntersections(*G*)**

// finds list of least open intersections

// Input: graph *G* of city with intersections as vertices and streets as edges

// Output: returns list of least possible number of closed intersections

*Esolated* ← list() // isolated edges

*Visole* ← Dict() // vertices of isolated edges

*G*.ArticulationPointsSearch() // algorithm is written in next code box

*result* ← *G*.APs()

**for** *edge* **in** *G*.edges() **do**

**if** *G*.APs(*edge*[0]) = *False* **and** *G*.APs(*edge*[1]) = *False* **then**

*Esolated*.add(*edge*)

**if** *edge*[0] **in** *Visole* **then**

*Visole*[*edge*[0]] ← *Visole*[*edge*[0]] + 1

**else then**

*Visole*.add(*edge*[0]:0)

**if** *edge*[1] **in** *Visole* **then**

*Visole*[*edge*[1]] ← *Visole*[*edge*[1]] + 1

**else then**

*Visole*.add(*edge*[1]:0)

*Visole*.valuesort()

**for** *v* **in** *Visole* **do**

**if** **any** *Esolated* **in** *G*.edges(*v*) **then**

*result*.add(*v*)

**for** *edge* **in** *Esolated* **do**

**if** edge[0] = *v* **or** *edge*[1] = *v* **then**

**delete** *edge*

**return** *result*

**Algorithm: ArticulationPointsSearch(*G*)**

// finds articulation points and edits their attributes

*G*.visited ← [*False*] × (*G*.vertices()) // mark all vertices as not visited (list of *V* falses)

*G*.discovery ← [float(“Inf”)] × ( *G*.vertices()) // initializing discovery times

*G*.low = [float(“Inf”)] × (*G*.vertices()) // will contain min of two discovery times

*G*.parent = [ – 1] × (*G*.vertices())

*G*.APs = [*False*] × (*G*.vertices())

**for** *u* ← 0 **to** *G*.vertices() **do**

**if** *G*.visited[ u ] = *False* **then**

*G*.Articulation(*G*, *u*)

**Algorithm: Articulation(*G*, *u*)**

// implements depth first search algorithm

*children* ← 0

*G*.visited[*u*] ← *True*

*G*.discovery[*u*] ← G.time // *time* is an attribute of graph marking visiting time

*G*.low[*u*] ← *G*.time

G.time ← *G*.time + 1

**for** *v* **in** *G*.vertices(*u*) **do**

**if** *G*.visited[*v*] = *False* **then**

*G*.parent[*v*] ← *u*

*children* ← *children* + 1

Articulation(*G* , *v*)

*G*.low[*u*] ← min(*G*.low[*u*], *G*.low[*v*])

**if** *G*.parent[*u*] = – 1 **and** *children* > 1 **then**

*G*.APs[u] ← *True*

**if** *G*.parent[*u*] ≠ – 1 **and** Glow[*v*] ≥ *G*.discovery[*u*] **then**

*G*.APs[u] ← *True*

**elif** *v* ≠ *G*.parent[*u*] **then**

*G*.low[*u*] = min(*G*.low[*u*], *G*.discovery[*v*])

**8.** شما قرار است به گروهی از تحلیگران امنیت که در حال نظارت بر مجموعه­ای از رایانه­های یک شبکه و ردگیری مسیرهای انتشار یک ویروس در شبکه هستند، کمک کنید. رایانه­ موجود در شبکه، با برچسب­های مشخص شده­اند. مجموعه­ای از داده­های ردگیری به شما داده شده است که زمان­هایی را که دو رایانه با یکدیگر ارتباط داشته­اند، مشخص می­کنند. مجموعه داده­ها به شکل سه­تایی­های مرتب است که مشخص می­کند دو رایانه و ، در زمان ، بیت­هایی را با یکدیگر تبادل کرده­اند. در مجموع، سه تایی این چنینی وجود دارد. فرض کنید که سه­تایی­ها، قبلاً به ترتیب صعودی مؤلفه زمان­شان، مرتب شده باشند. و برای سادگی، فرض کنید که هر دو رایانه، در بازه زمانی که شما فعالیت آنها را نظاره می­کنید، حداکثر یک بار با یکدیگر تبادل اطلاعات خواهند داشت.

تحلیلگران امنیت که شما با آنها کار می­کنید، می­خواهند به چنین سؤالاتی پاسخ دهند: اگر ویروس در زمان به رایانه تزریق شده باشد، آیا می­توانسته است که رایانه را تا زمان آلوده کرده باشد؟

فرایند آلودگی ساده است: اگر رایانه آلوده ، در زمان ، با رایانه سالم ارتباط برقرار کرده باشد (یعنی اگر یکی از سه­تایی­های یا در داده­های ردگیری وجود داشته باشد) پس رایانه نیز، از زمان به بعد آلوده خواهد بود. بنابراین، آلودگی می­تواند از طریق دنباله­ای از ارتباطات، از یک رایانه به رایانه­ای دیگر انتقال پیدا کند؛ با این شرط که هر گام در این دنباله، بیانگر حرکت رو به جلوی ویروس در طول زمان باشد. اگر مثلاً رایانه تا زمان آلوده شده باشد و سه­تایی­های و در داده­های ردگیری وجود داشته باشد و باشد، پس می­توان نتیجه گرفت که نیز از طریق آلوده شده است. (توجه کنید که می‌تواند با برابر باشد: این برابری، به آن معناست که همزمان با و در حال ارتباط بوده است؛ و در نتیجه، ویروس می­توانسته از به منتقل شود.)

به عنوان مثال، وضعیتی را در نظر بگیرید که رایانه در شبکه وجود داشته است و مجموعه داده­های ردگیری، شامل سه­تایی­های

باشد و ویروس در زمان 2 به رایانه تزریق شده باشد.

در چنین وضعیتی، می­توان نتیجه گرفت که بعد از 3 گام و در زمان 8 ، رایانه نیز آلوده شده است: در زمان 4 ، آلوده شده است؛ سپس در زمان 8 ، ویروس را از گرفته است؛ و سپس در زمان 8 ، ویروس را از گرفته است.

از طرف دیگر، اگر مجموعه داده­های ردگیری، به صورت

باشند و ویروس در زمان 2 به رایانه تزریق شده باشد، می­توان نتیجه گرفت که در طول دوره نظارت، آلوده نشده است. گرچه در زمان 14 آلوده شده، ولی می­بینیم که تنها در وقتی که هنوز آلوده نشده است، با آن ارتباط برقرار کرده است. بنابراین، در این نمونه، دنباله­ای از ارتباطات رو به جلو در محور زمان وجود ندارد که از طریق آن، ویروس بتواند از به برسد.

الگوریتمی طراحی کنید که با آن بتوان به سؤالاتی از این دست پاسخ داد: با توجه به داده­های ردگیری، آیا اگر ویروس در زمان به رایانه تزریق شده باشد، می­توانسته است که رایانه را تا زمان آلوده کرده باشد یا خیر؟ الگوریتم شما باید در زمان اجرا شود.

**جواب:**

ارتباطات کامپیوترها را که در سه‌تایی‌ها داده شدند به صورت گرافی غیر جهت‌دار و وزن‌دار با لیست مجاورت در نظر می‌گیریم که کامپیوترها رئوس آن و یال‌ها ارتباط میان آن‌ها باشند. آنگاه زمان ارتباطشان به عنوان وزن آن یال قرار گرفته و در لیست مجاورت هر راس، همراه راسی که راس *v* با آن ارتباط دارد ذکر شده است، به شکلی که اعضای لیست مجاورت دوتایی‌های مرتبی هستند که عضو اول راس متصل و عضو دوم زمان ارتباط است.

نکته قابل توجه این است که باید در حین جستجو در گراف، حتماً زمان ارتباط حال حاضر را با زمان ارتباط قبلی کامپیوترمان بررسی کنیم تا همواره زمان‌ها به صورت صعودی پیش رفته و در زمان عقب نرویم. برای این کار، وزن یال میان راس‌های قابل بررسی را با وزن یال قبلاً بررسی شده (که به عنوان ورودی زمان *x* به الگوریتم بازگشتی داده شده) مقایسه کنیم تا از آن بیشتر و از زمان پایانی یعنی *y* کمتر باشد. در اول فراخوانی الگوریتم، مقدار *x* مقدار ویروسی شدن کامپیوتر *Ca* است.

می‌توانیم با استفاده از الگوریتم پیمایش عمقی به صورت بازگشتی از کامپیوترِ ویروسی شروع کرده و با تغییر وضعیت (یعنی ویژگی *status* برای شیء راس در گراف) از آن راس پیش رویم و ارتباطات آن را در محدوده زمانی داده شده بررسی کنیم تا مشاهده کنیم آیا با کامپیوتر مقصود ارتباط داشته است یا خیر. از آنجا که الگوریتم هر راس را حداکثر یک بار بررسی می‌کند و از الگوریتم خطی *DFS* استفاده می‌کند، خطی می‌باشد.

شبه کد الگوریتم در صفحه‌ی بعد قابل مشاهده است.

**Algorithm: MalwareDetection(*G*, *x*, *y*, *Ca*, *Cb*)**

// determines whether computer *Cb* has been infected by *Ca* from time *x* to *y*

// implements depth-first search traversal on a weighted graph

// Input: graph *G* of data, time *x* when *Ca* was infected, time *y* until which to check *Cb*

// Output: returns *True* if *Cb* was infected, else returns *False*

*Ca*.visited *←* *True* // vertex is infected

**for** *pair* **in** *G*.adjacencyList(*Ca*) **do**

// element in the adjacency list of vertex is an ordered pair: (*adj vertex*, *weight*)

**if** *pair*[0].visited **is not** *True* **and** *x* ≤ *pair*[1] ≤ *y* **then**

**if** *pair*[0] = *Cb* **then**

**return** *True*

**else then**

*MalwareDetection*(*G*, *pair*[1], *y*, *pair*[0], *Cb*)

**return** *False*

**9.** ریاضیدان که در جشن سال نوی دانشگاه خود شرکت کرده­اند، می­خواهند با همدستی کردن، رئیس دانشگاه را که نیز در جشن شرکت کرده است، به چالش بکشانند: رئیس، روی کلاه­ هر یک از ریاضیدانان که در جشن به سر کرده است، عددی از 0 تا *n* − 1 را خواهد نوشت؛ اعداد نوشته شده روی کلاه­ها، ممکن است یکسان باشند یا متفاوت. آنگاه هر یک از ریاضیدانان، بدون هیچ گونه ارتباطی با ریاضیدانان دیگر و تنها بعد از دیدن همه اعداد نوشته شده روی کلاه­های دیگران (جز کلاه خودش) ، عدد کلاه خودش را حدس خواهد زد و آن را روی تکه­ای کاغذ خواهد نوشت و به رئیس تحویل خواهد داد. البته، هیچ یک از ریاضیدانان نباید اعداد نوشته شده توسط دیگران را ببیند. اگر حداقل یکی از اعدادی که ریاضیدانان روی تکه­های کاغذ نوشته­اند، درست باشد، آنها شرط­بندی جمعی را برنده خواهند شد و رئیس، بودجه سال آینده دانشکده آنها را 5 درصد افزایش خواهد داد. اما اگر حدس هیچ یک از آنها درست نباشد، بودجه پنج سال آینده آنها معلق خواهد ماند!

آیا ریاضیدانان ادعایی واهی کرده­اند یا آنکه راهی برای برنده شدن دارند؟

**جواب:**

ریاضیدانان برای اینکه حتماً بازی را برنده شوند می‌توانند استراتژی بردی طراحی کنند به طوری که قبل از جشن هر کدام عددی از میان 0 تا *n* – 1 را انتخاب کنند تا حتماً همه‌ی اعداد توسط یک نفر انتخاب شود و هر فرد عدد خاصی داشته باشد (که آن را *m* می‌نامیم). زمانی که بازی آغاز می‌شود می‌دانیم که هر فرد می‌تواند اعداد دیگران را مشاهده کند و جمع آن‌ها را محاسبه کند؛ اگر این جمع را *sum* نامیده و عدد روی کلاه هر فرد را مجهول *x* بنامیم، فرد باید حدس خود یعنی عدد *y* را روی برگه بنویسد و این عدد باید برای حداقل یک نفر درست بوده و *x* = *y* باشد.

با یک اثبات ساده‌ی ریاضی متوجه می‌شویم که اگر هر فرد حدس خود یعنی *y* را بر اساس باقی‌مانده‌ی تقسیم *sum* بر *n* بیابد، حداقل یک نفر حدس درست خواهد زد:

می‌خواهیم *x* = *y* شود. می‌دانیم که باقی‌مانده‌ی تقسیم مجموع اعدادِ روی کلاه‌ها یعنی *sum* (مانند هر عدد دیگری) بر *n* حتما یکی از اعدد 0 تا *n* – 1 خواهد بود. اگر این عدد باقی مانده *r* باشد، باقی‌مانده‌ی تقسیم *r* بر *n* نیز خود *r* می‌شود و این دو بر اساس باقی‌مانده تقسیمشان بر *n* یعنی *mod n* هم کلاس هستند. حال می‌دانیم که هر یک از ریاضیدانان پیش از جشن یکی از اعداد 0 تا *n* – 1 را انتخاب کرده بودند، پس عدد *r* برای حداقل یکی از آن‌ها برابر با عدد از پیش تعیین شده‌ی *m* خواهد بود. آنگاه برای این فرد باید اعداد *sum* و *m* در تقسیم بر *n* هم کلاس باشند، که این بر اساس فرض پیشین بدیهی‌ست پس اثبات کامل است. یعنی:

بنابراین دیدیم که برای حداقل یک فرد عدد حدسی *y* قطعاً با عدد روی کلاه وی یعنی *x* برابر خواهد شد. پس افراد باید پیش از جشن هر کدام عددی را انتخاب کنند سپس با محاسبه‌ی جمع اعدادِ روی کلاه دیگران، با تقسیم آن بر *n* عدد حدسی را به دست آورده و روی کاغذ بنویسند تا سر رئیس دانشگاه به اصلاح کلاه بگذارند.

**10.** الگوریتم غربال اِراتوستِنس، الگوریتمی ساده­اندیشانه است برای پیدا کردن تمام اعداد اول کوچک­تر یا مساوی با یک عدد طبیعی خاص:

**الف)** نشان دهیدکارایی زمانی این الگوریتم است. آیا این الگوریتم را می­توان مثالی از یک الگوریتم کارا دانست؟

**ب)** نشان دهیدکارایی فضایی این الگوریتم است. آیا ممکن است الگوریتم، در عمل، به دلیل کمبود حافظه رایانه­، قادر به تولید خروجی نباشد؟

**پ)** برنامه­ای بنویسید برای تحلیل تجربی الگوریتم؛ کارایی زمانی و کارایی فضایی الگوریتم را با آزمایش تعیین کنید. نتایجی را که از انجام آزمایش به دست خواهید آورد با نتایجی که از تحلیل­ ریاضی به دست آورده­اید، مقایسه کنید. آیا نهایتاً استفاده از الگوریتم را در عمل توصیه خواهید کرد؟

**جواب:**

**10 – الف)** اگر عمل علامت زدن به عنوان عدد غیر اول در درونی‌ترین حلقه‌ی الگوریتم را عمل پایه در نظر بگیریم، مشاهده می‌کنیم که برای هر عدد حذف نشده‌ی اول *p* از اعداد 2، تعداد عمل پایه‌ی حذف مضارب عدد *p* را انجام می‌دهیم. پس برای کارایی زمانی الگوریتم داریم:

برای محاسبه مقدار مجموع بالا نیاز است از اثبات سری‌های هامونیک استفاده کرده و واگرایی جمع وارون ضربی‌های اعداد اول را نشان دهیم. اثبات‌های متفاوتی برای این واگرایی و یافتن مقدار حاصل وجود دارد که کوتاه‌ترین و معروف‌ترین آن‌ها اثبات اویلر می‌باشد.

لازم به ذکر است که با وجود درست بودن پاسخ پیدا شده توسط اویلر، منطق اثبات وی به علت نتیجه‌گیری‌های ناگهانی‌اش اندکی شک برانگیز است. این اثبات که از *product formula* اویلر برای سری‌های هارمونیک و از سری تیلور استفاده می‌کند به شکل زیر است:

پس اثبات کامل است و بدین ترتیب کارایی زمانی الگوریتم محاسبه می‌شود.

همانطور که مشاهده می‌کنیم، الگوریتم از نظر زمانی نسبت به الگوریتم عادی تعیین اعداد اول (یعنی با روش تقسیم کردن) کاراتر است زیرا نه تنها عمل تقسیم از اعمال جمع و ضرب گران‌تر و زمان‌بر تر است، بلکه تعداد اعمالی که الگوریتم انجام می‌دهد نیز کمتر می‌باشد؛ از آنجا که در زمان رسیدن به حد *n* متوقف شده و همچنین درحالی که اعداد مرکب بیشتر از اعداد اول هستند، آن‌ها در اصل تعداد کمی ریشه‌ی اول دارند و برای الگوریتم تقسیم، اعداد اول زیادی کوچکتر از جزر حد قرار دارند.

**10 – ب)** کارایی فضایی الگوریتم بر اساس متغیرهایی که خارج از و اضافه بر ورودی(ها) به وجود می‌آیند تایین می‌شود که در الگوریتم غربالِ اراتوستنس داده شده در سوال این متغییر همان لیست *A* ایجاد شده از اعداد 2 تا *n* است. در نتیجه پیچیدگی فضایی به شکل زیر است:

با اینکه کارایی فضایی الگوریتم برابر O(*n*) است اما فضای استفاده شده از رم برای اعداد بزرگ بسیار زیاد است، برای مثال اگر *n* = 1012 باشد حجم مورد نیاز حدود 1GB می‌شود. پس کاملا ممکن است که الگوریتم پیش از آنکه بخواهد با کمبود زمان مواجه شود، به علت کمبود حافظه از کار افتاده و متوقف شود، به طوری که برای *n*های بیشتر از 14 رقم کامپیوترهای روزمره قابلیت اجرای برنامه را از دست خواهند داد.

**10 – پ)** برنامه‌های پایتونی که برای اندازه‌گیری تجربی مصرف زمان و حافظه‌ی الگوریتم اراتوستنس نوشته شده و کد آن‌ها در پیوست قرار داده شده است، به ترتیب از ماژول‌های *time* و *tracemalloc* پایتون استفاده می‌کنند. با ماژول *time* زمان اجرای 10 بارِ تابع غربال اراتوستنس را اندازه‌گیری کرده و آن را برای یک بار اجرا بر حسب میکروثانیه محاسبه می‌کنیم و در فایل *csv* قرار می‌دهیم. به کمک ماژول *tracemalloc* پایتون نیز می‌توانیم بیشترین میزان فضای استفاده شده توسط تابع را در زمان اجرا به دست بیاوریم؛ هرچند این ماژول از سرعت اجرا کم می‎‌کند، اما بر خلاف ماژول *sys* می‌تواند فضای مصرفی هر بخش برنامه را با دقت ضبط و محاسبه کند.

جداول حاصل در صفحه‌ی بعد درج شده‌اند. با بررسی آن‌ها می‌بینیم که مرتبه رشد هر دو نمودار به O(*n*) شباهت بسیاری داشته که این مشاهده با نتیجه‌های ریاضی در قسمت‌های قبل سوال همخوانی دارد. پس با وجود اینکه الگوریتم از نظر زمانی کارا است، برای اعداد خیلی بزرگ در حافظه دچار مشکل می‌شود. توصیه نمی‌شود. بنابراین بسته به موقعیت و دستگاه مورد استفاده، با توجه به اینکه آیا زمان برای ما اهمیت دارد یا حافظه، استفاده از این الگوریتم می‌تواند مفید باشد.

نتایج بررسی تجربی الگوریتم از اجرای برنامه‌ها بر روی کامپیوتری با مشخصات زیر جمع‌آوری شدند:

*Results extracted from program run on:* Python 3.9.2; Windows 10; AMD FX-9830p CPU (quad core with single tread and 3.4 G Hertz processing speed).

|  |  |
| --- | --- |
| **Number** | **Time** (microsecond) |
| 10000 | 4081.3 |
| 15000 | 6262.51 |
| 20000 | 8107.1 |
| 25000 | 10154.6 |
| 30000 | 12159.33 |
| 35000 | 14445.63 |
| 40000 | 16738.03 |
| 45000 | 18527.58 |
| 50000 | 20935.4 |
| 55000 | 23476.45 |
| 60000 | 25691.21 |
| 65000 | 27452.08 |
| 70000 | 29879.48 |
| 75000 | 32319.49 |
| 80000 | 34460.92 |
| 85000 | 36817.85 |
| 90000 | 39857.04 |
| 95000 | 41367.92 |
| 100000 | 43551.39 |

|  |  |
| --- | --- |
| **Number** | **RAM usage** (bite) |
| 10000 | 65364 |
| 15000 | 93602 |
| 20000 | 129280 |
| 25000 | 149956 |
| 30000 | 184878 |
| 35000 | 209710 |
| 40000 | 237010 |
| 45000 | 266798 |
| 50000 | 299126 |
| 55000 | 334594 |
| 60000 | 373710 |
| 65000 | 412886 |
| 70000 | 423390 |
| 75000 | 466926 |
| 80000 | 477906 |
| 85000 | 525694 |
| 90000 | 537014 |
| 95000 | 589898 |
| 100000 | 596930 |

**پیوست**

# EMPIRICAL ANALYSIS OF TIME COMPLEXITY

import timeit

import csv

def SieveOfEratosthenes(n):

prime = [True for i in range(n + 1)]

p = 2

while p \* p <= n:

if prime[p] == True:

for i in range(p \* 2, n + 1, p):

prime[i] = False

p += 1

prime[0] = False

prime[1] = False

Plist = []

for p in range(n + 1):

if prime[p]:

Plist.append(p)

return Plist

Test = []

for i in range(10000, 100001, 5000):

Test.append(i)

csvlines = [['Number', 'Time (microsecond)']]

code = '''

from \_\_main\_\_ import SieveOfEratosthenes

from \_\_main\_\_ import Number'''

for num in Test:

Number = num

time = timeit.timeit(setup = code,

stmt = 'SieveOfEratosthenes(Number)',

number = 10)

time = round((time\*1000000)/10, 2)

csvlines.append([num, time])

with open('Time.csv', 'w', newline='') as file:

writer = csv.writer(file)

writer.writerows(csvlines)

# EMPIRICAL ANALYSIS OF SPACE COMPLEXITY

import tracemalloc

import csv

def SieveOfEratosthenes(n):

prime = [True for i in range(n + 1)]

p = 2

while p \* p <= n:

if prime[p] == True:

for i in range(p \* 2, n + 1, p):

prime[i] = False

p += 1

prime[0] = False

prime[1] = False

Plist = []

for p in range(n + 1):

if prime[p]:

Plist.append(p)

return Plist

Test = []

for i in range(10000, 100001, 5000):

Test.append(i)

csvlines = [['Number', 'RAM usage (bite)']]

code = '''

from \_\_main\_\_ import SieveOfEratosthenes

from \_\_main\_\_ import Number'''

for num in Test:

tracemalloc.start()

SieveOfEratosthenes(num)

current, peak = tracemalloc.get\_traced\_memory()

tracemalloc.stop()

csvlines.append([num,peak])

with open('Space.csv', 'w', newline='') as file:

writer = csv.writer(file)

writer.writerows(csvlines)