# تمرینات فصل سه

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

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

```
ALGORITHM  BruteForceStringMatch(T [0..n − 1], P[0..m − 1])
// Implements brute − force string matching
// Input: An array T [0..n − 1] of n characters representing a text and
// an array P[0..m − 1] of m characters representing a pattern
// Output: The index of the first character in the text that starts a
// matching substring or − 1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T [i + j] do
        j ← j + 1
    if j = m
        return i
return − 1
```

**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 $O(n)$ 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 متفاوت 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:

```
ALGORITHM  MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])
// Multiplies two square matrices of order n by the definition − based algorithm
// Input: Two n × n matrices A and B
// Output: Matrix C = AB

for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

**الف)** اگر $A$ ماتریســـی $m \times n$ و $B$ ماتریســـی $n \times p$ باشـــد، الگوریتم را به نحوی تغییر دهید که با آن بتوان حاصل‌ضرب دو ماتریس مستطیلی $A$ و $B$ را به دست آورد. تعداد دقیق ضرب‌ها و تعداد دقیق جمع‌های الگوریتم جدید چیستند؟ کارایی زمانی الگوریتم چقدر است؟

**ب)** اگر $A$ یک ماتریس بالامثلثی $n \times n$ بوده و $B$ نیز یک ماتریس بالامثلثی $n \times n$ باشد، الگوریتم را به نحوی تغییر دهید که با آن بتوان حاصل‌ضـــرب دو ماتریس بالامثلثی $A$ و $B$ را به دســـت آورد؛ الگوریتم شـــما نباید عناصـری از ماتریس حاصل‌ضـرب را که پیشـاپیش صـفر بودن آنها مشـخص اسـت، محاسـبه 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 $A_{n\times m}$ و $B_{m\times p}$ باشـــند، تنها کافی‌ســـت 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 A_{n×m} and B_{m×p}
// Output: matrix C_{n×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 ضـرب و جمع در یک خط انجام می‌شـوند و پس تعداد تکرارشـان یکی‌سـت. 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:

$$C(n,m,p) = \sum_{i=0}^{n-1}\sum_{j=0}^{p-1}\sum_{k=0}^{m-1} 1 = \sum_{i=0}^{n-1}\sum_{j=0}^{p-1}\big((m-1)-0+1\big) = \sum_{i=0}^{n-1}\big((p-1)-0+1\big)m$$

$$= \big((n-1)-0+1\big)pm = nmp$$

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 $i \le j$ 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 A_{n×n} and B_{n×n}
// Output: matrix C_{n×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:

$$C(n)=\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}\sum_{k=0}^{j}1=\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j-0+1)=\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}j+\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}1$$

$$=\sum_{i=0}^{n-1}\frac{(n-i)\big((n-1)+i\big)}{2}+\sum_{i=0}^{n-1}(n-i)$$

$$=\sum_{i=0}^{n-1}\frac{n^2}{2}-\sum_{i=0}^{n-1}\frac{i^2}{2}-\sum_{i=0}^{n-1}\frac{n}{2}+\sum_{i=0}^{n-1}\frac{i}{2}+\sum_{i=0}^{n-1}n-\sum_{i=0}^{n-1}i$$

$$=\sum_{i=0}^{n-1}\frac{n^2+n}{2}-\sum_{i=0}^{n-1}\frac{i^2}{2}-\sum_{i=0}^{n-1}\frac{i}{2}$$

$$=n\left(\frac{n^2+n}{2}\right)-\frac{n\left(\frac{(n-1)^2}{2}+\frac{0^2}{2}\right)}{2}-\frac{n\left(\frac{n-1}{2}+\frac{0}{2}\right)}{2}=\frac{n^3+5n^2-2n}{4}$$

با مقایسه‌ی حاصل با بخش قبل مشاهده 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:

**ALGORITHM** $Example(a, b)$

// Input: Two integers $a$ and $b > 0$

$x \leftarrow 0$
$y \leftarrow |a|$

**while** $y \geq b$ **do**
    $y \leftarrow y - b$
    $x \leftarrow x + 1$

**if** $a < 0$ **and** $y = 0$
    $x \leftarrow -x$

**if** $a < 0$ **and** $y > 0$
    $y \leftarrow b - y$
    $x \leftarrow -(x + 1)$

**return** $(x, y)$

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

**ب)** درستی الگوریتم را ثابت کنید.

**پ)** با این فرض که $a > b$ باشد، این ادعا را که کارایی زمانی الگوریتم $O(x \log a)$ 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:

$$a = xb + y = (b + b + \cdots + b) + y \rightarrow a - b - b - \cdots - b = y$$

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:

$$\xrightarrow{y=0} a = xb \rightarrow -a = -xb$$

$$\begin{aligned}
\xrightarrow{y>0} a &= xb + y = (b + \cdots + b) + y \\
&= (b + \cdots + b) + (b + \cdots + b) - (b + \cdots + b) + b - b + y + y - y \\
&= -(b + \cdots + b) - b + (b - y) + 2\big((b + \cdots + b) + y\big) \\
&= -\big((b + \cdots + b) + b\big) + (b - y) + 2(xb + y) \\
&= -(x+1)b + (b - y) + 2a \\
&\rightarrow a - 2a = -(x+1)b + (b - y) \\
&\rightarrow -a = -(x+1)b + (b - y)
\end{aligned}$$

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 $O(x\log a) \in O(a)$، پس با فرض درستی آن می‌نویسیم:

$$\left[\frac{a}{b}\right]\log a \le a \xrightarrow{\log a = c} \frac{2^c}{b} \times c \le 2^c \longrightarrow \frac{2^c}{b} \times \frac{1}{2^c} \times c \le 1 \longrightarrow \frac{c}{b} \le 1 \longrightarrow c \le b$$
$$\xrightarrow{we\ know\ a > b > 0} \log a < \log b \xrightarrow{\log b < b} \log a < b \longrightarrow c < b$$

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

---

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

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

**<u>جواب:</u>**

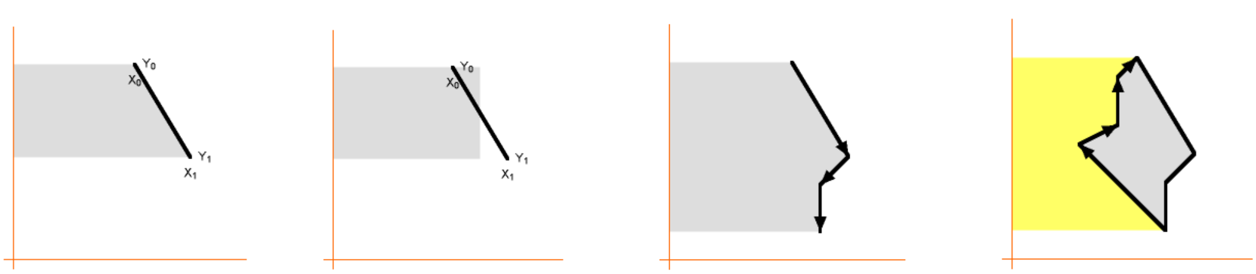
**4 – الف)** برای محاسبه مساحت چندضلعی محدب $n$ راسی از اصل هرون در هندسه برای محاسبه مساحت مثلث استفاده می‌کنیم که با آن بدون داشتن ارتفاع می‌توانیم مساحت را بیابیم. برای این کار ابتدا نیاز است چندضلعی را مثلث‌بندی کنیم و سپس با محاسبه مساحت هرکدام با اصل هرون، مساحت‌ها را جمع کنیم. شبه کد الگوریتم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

$$max(f(n), g(n)) \in O(f(n) + g(n)) \rightarrow O(n - 2 + n - 2) = O(2n - 4) \in O(n)$$

```
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 $((x_1 + x_2) \times (y_1 - y_2)) / 2$، 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 $n$ مجموعه $S_1, S_2, \cdots, S_n$ ، خود زیرمجموعه‌ای از مجموعه $\{1, 2, \cdots, n\}$ 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 O 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(S1, S2, …, 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:

$$C_{worst}(n) = \sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{i=1}^{n}\sum_{i=1}^{n} 1 = \sum_{i=1}^{n}(n-i+1)n^2 = n(n^3+n^2) - \sum_{i=1}^{n} i.n^2$$
$$= n(n^3+n^2) - \frac{n(n.n^2+n^2)}{2} = \frac{n^4+n^3}{2}$$

**5 – پ)** از آنجایی که تعداد تکرار عمل پایه یعنی عمل مقایســـه، 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 کاســـت. برای این کار می‌توانیم به جای دو حلقه‌ی درونی الگوریتم که اعضـــای 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 جســـتجوی دودویی در مجموعه‌ی مرتب شـــده درون حلقه، با اعضـــای مجموعه‌ی اول مقایســـه می‌کنیم (که این حلقه‌ی درونی کارایی $\log n$ دارد). پس دو حلقه‌ی درونی الگوریتم از $n^2$ به $2n\log n$ تغییر میابد.

$$O(n^2.2n\log n) = O(2n^3\log n) \in O(n^3\log n)$$

---

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

یکی از تالارهای دانشــگاه بزرگ‌تر از بقیه اســـت و به همین دلیل، برگزار کنندگان می‌خواهند بیشـــترین 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 بیشـترین تعداد از سـخنرانی‌ها است به گونه‌ای که هیچ دو سـخنرانی 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 $n$ را (که تعداد سخنرانی‌ها اســـت) ، مجموعه‌ی $S = \{s_1, s_2, \cdots, s_n\}$ را (که شـــامل زمان‌های شـــروع ســـخنرانی‌ها اســـت) و مجموعه‌ی $F = \{f_1, f_2, \cdots, f_n\}$ را (که شامل زمان‌های پایان سخنرانی‌ها است) به عنوان ورودی می‌گیرد.

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

**جواب:**

**6 – الف)** برای نوشــتن الگوریتم ســاده‌اندیشــانه با فرض اینکه عضــو اول دو مجموعه، زمان شــروع و پایان یک ســخنرانی یکســان باشــند، با اســتفاده از الگوریتم ســاده‌اندیشــانه‌ی 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 ســخنرانی و ســپس گزینش اولین ســخنرانی‌ای که بعد از زمان پایان ســخنرانی اول شــروع می‌شــود، حلقه را پیش برده و مجموعه‌ای از بیشــترین ســخنرانی‌های ممکن ایجاد می‌کنیم. این عمل را برای انتخاب‌های متفاوت به عنوان ســـخنرانی اول روز و یا ســـخنرانی‌های 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 $\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1}\sum_{k=j+1}^{n-1} 1$ دارد در بدترین حاله (که شــرط همواره برقرار باشــد، دارای کارایی زمانی $O(n^3)$ بوده و بخش چهارم حلقه‌ای‌ســت که به تعداد حالت‌های کلی ممکن برای ســـخنرانی‌ها تکرار می‌شـــود و در هر حال خطی می‌باشـــد. حال می‌دانیم که برای محاسبه کارایی الگوریتمی با چند بخش مستقل، باید *max* مقادیر کارایی را در نظر بگیریم؛ یعنی:

$$max( f(n), g(n), h(n), k(n)) \in O( f(n) + g(n) + h(n) + k(n)) \rightarrow O(n^2 + n + n^3 + n) \in O(n^3)$$

**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 $n$ باشد و تعداد خیابان‌های شهر $m$ باشد، کارایی الگوریتم‌تان را برحسب $n$ و $m$ تعیین کنید.

**<u>جواب:</u>**

**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.** شـــما قرار اســـت به گروهی از تحلیلگران امنیت که در حال نظارت بر مجموعه‌ای از رایانه‌های یک شـــبکه و ردگیری مسیرهای انتشار یک ویروس در شبکه هستند، کمک کنید. $n$ رایانه موجود در شبکه، با برچسب‌های $C_1, C_2, \cdots, C_n$ 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 $(C_i, C_j, t_k)$ است که مشخص می‌کند دو رایانه $C_i$ و $C_j$ ، در زمان $t_k$ ، بیت‌هایی را با یکدیگر تبادل کرده‌اند. در مجموع، $m$ 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 $x$ به رایانه $C_a$ تزریق شده باشد، آیا می‌توانسته است که رایانه $C_b$ را تا زمان $y$ آلوده کرده باشد؟

فرایند آلودگی ساده است: اگر رایانه آلوده $C_i$ ، در زمان $t_k$ ، با رایانه سالم $C_j$ ارتباط برقرار کرده باشد (یعنی اگر یکی از سه‌تایی‌های $(C_i, C_j, t_k)$ یا $(C_j, C_i, t_k)$ در داده‌های ردگیری وجود داشته باشد) پس رایانه $C_j$ نیز، از زمان $t_k$ 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 $C_j$ تا زمان $t_k$ آلوده شده باشد و سه‌تایی‌های $(C_i, C_j, t_k)$ و $(C_j, C_q, t_r)$ در داده‌های ردگیری وجود داشته باشد و $t_k \leq t_r$ باشد، پس می‌توان نتیجه گرفت که $C_q$ نیز از طریق $C_j$ آلوده شده است. (توجه کنید که $t_k$ می‌تواند با $t_r$ برابر باشد: این برابری، به آن معناست که $C_j$ همزمان با $C_i$ و $C_q$ در حال ارتباط بوده است؛ و در نتیجه، ویروس می‌توانسته از $C_i$ به $C_q$ منتقل شود.)

به عنوان مثال، وضعیتی را در نظر بگیرید که $n = 4$ رایانه در شبکه وجود داشته است و مجموعه داده‌های ردگیری، شامل سه‌تایی‌های

$$(C_1, C_2, 4), (C_2, C_4, 8), (C_3, C_4, 8), (C_1, C_4, 12)$$

باشد و ویروس در زمان 2 به رایانه $C_1$ تزریق شده باشد.

در چنین وضعیتی، می‌توان نتیجه گرفت که بعد از 3 گام و در زمان 8 ، رایانه $C_3$ نیز آلوده شده است: در زمان 4 ، $C_2$ آلوده شده است؛ سپس در زمان 8 ، $C_4$ ویروس را از $C_2$ گرفته است؛ و سپس در زمان 8 ، $C_3$ ویروس را از $C_4$ گرفته است.

از طرف دیگر، اگر مجموعه داده‌های ردگیری، به صورت

$$(C_2, C_3, 8), (C_1, C_4, 12), (C_1, C_2, 14)$$

باشـند و ویروس در زمان 2 به رایانه $C_1$ تزریق شـده باشـد، می‌توان نتیجه گرفت که در طول دوره نظارت، $C_3$ آلوده نشده است. گرچه $C_2$ در زمان 14 آلوده شده، ولی می‌بینیم که $C_3$ تنها در وقتی که $C_2$ هنوز آلوده نشده اســـت، با آن ارتباط برقرار کرده اســـت. بنابراین، در این نمونه، دنباله‌ای از ارتباطات رو به جلو در محور زمان وجود ندارد که از طریق آن، ویروس بتواند از $C_1$ به $C_3$ 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 $x$ به رایانه $C_a$ تزریق شده باشد، می‌توانسته است که رایانه $C_b$ را تا زمان $y$ آلوده کرده باشد یا خیر؟ الگوریتم شما باید در زمان $O(n + m)$ اجرا شود.

**<u>جواب:</u>**

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 اســت که باید در حین جســتجو در گراف، حتماً زمان ارتباط حال حاضــر 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 $C_a$ است.

می‌توانیم با استفاده از الگوریتم پیمایش عمقی به صورت بازگشتی از کامپیوترِ ویروسی شروع کرده و با تغییر وضـــعیت (یعنی ویژگی *status* برای شـــیء راس در گراف) از آن راس پیش رویم و ارتباطات آن را در محدوده زمانی داده شـده بررسـی کنیم تا مشـاهده کنیم آیا با کامپیوتر مقصـود ارتباط داشـته اسـت 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** $n > 1$ 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:**

ریاضـــیدانان برای اینکه حتماً بازی را برنده شـــوند می‌توانند اســـتراتژی 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 خاصـی داشـته باشـد (که آن را $m$ می‌نامیم). زمانی که بازی آغاز می‌شـود می‌دانیم که هر فرد می‌تواند اعداد دیگران را مشـاهده کند و جمع آن‌ها را محاسـبه 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 بنویســـد و این عدد باید برای حداقل یک نفر درست بوده و $x=y$ باشد.

با یک اثبات ساده‌ی ریاضـی متوجه می‌شـویم که اگر هر فرد حدس خود یعنی $y$ را بر اسـاس باقی‌مانده‌ی تقسـیم $sum$ بر $n$ بیابد، حداقل یک نفر حدس درست خواهد زد:

می‌خواهیم $x=y$ شـــود. می‌دانیم که باقی‌مانده‌ی تقســـیم 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 $r$ باشـد، باقی‌مانده‌ی تقسـیم $r$ 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 ریاضـیدانان پیش از جشـن 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 شـده‌ی $m$ خواهد بود. آنگاه برای این فرد باید اعداد $sum$ و $m$ در تقسـیم بر $n$ هم کلاس باشند، که این بر اساس فرض پیشین بدیهی‌ست پس اثبات کامل است. یعنی:

$$y-x\equiv 0\ (mod\ n)\longrightarrow y-x+r\equiv r\ (\mathrm{mod}\ n)\xrightarrow{sum\equiv r\ (\mathrm{mod}\ n)} y-x+sum\equiv r\ (\mathrm{mod}\ n)$$

$$\xrightarrow{in\ at\ least\ one\ case:\, r=m} y-x+sum\equiv m\ (\mathrm{mod}\ n)\xrightarrow{y=x} sum\equiv m\ (\mathrm{mod}\ n)$$

بنابراین دیدیم که برای حداقل یک فرد عدد حدســی $y$ قطعاً با عدد روی کلاه وی یعنی $x$ برابر خواهد شـد. پس افراد باید پیش از جشـــن هر کدام عددی را انتخاب کنند ســـپس با محاســـبه‌ی 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:

```
ALGORITHM  Sieve(n)
// Implements the sieve of Eratosthenes
// Input: A positive integer n > 1
// Output: Array L of all prime numbers less than or equal to n
for p ← 2 to n do
    A[p] ← p
for p ← 2 to √n do    // see note before pseudocode
    if A[p] ≠ 0    // p hasn′ t been eliminated on previous passes
        j ← p * p
        while j ≤ n do
            A[j] ← 0    // mark element as eliminated
            j ← j + p
// copy the remaining elements of A to array L of the primes
i ← 0
for p ← 2 to n do
    if A[p] ≠ 0
        L[i] ← A[p]
        i ← i + 1
return L
```

**الف)** نشان دهید کارایی زمانی این الگوریتم $O(n \log\log n)$ است. آیا این الگوریتم را می‌توان مثالی از یک الگوریتم کارا دانست؟

**ب)** نشان دهید کارایی فضایی این الگوریتم $O(n)$ 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 $\frac{n}{p}$ عمل پایه‌ی حذف مضارب عدد $p$ را انجام می‌دهیم. پس برای کارایی زمانی الگوریتم داریم:

$$C(n) = \frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \cdots = n\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \cdots\right) = n\left(\sum_{p \text{ prime}} \frac{1}{p}\right)$$

برای محاســبه مقدار مجموع بالا نیاز اســت از اثبات ســری‌های هامونیک اســتفاده کرده و واگرایی جمع وارون ضــربی‌های اعداد اول را نشــان دهیم. اثبات‌های متفاوتی برای این واگرایی و یافتن مقدار حاصــل 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 شـک برانگیز اسـت. این اثبات که از *product formula* اویلر برای سـری‌های هارمونیک و از سری تیلور استفاده می‌کند به شکل زیر است:

$$\sum_{n=1}^{\infty} \frac{1}{n} = \infty \xrightarrow{product\ formula} \sum_{n=1}^{\infty} \frac{1}{n} = \prod_{p}\left(1 + \frac{1}{p} + \frac{1}{p^2} + \cdots\right) = \prod_{p} \frac{1}{1 - p^{-1}}$$

$$\xrightarrow{taking\ \log} \log\left(\sum_{n=1}^{\infty} \frac{1}{n}\right) = \log\left(\prod_{p} \frac{1}{1 - p^{-1}}\right) = \sum_{p} \log\left(\frac{1}{1 - p^{-1}}\right)$$

$$\xrightarrow{Taylor\ series\ expansion} \log\left(\sum_{r=1}^{n} \frac{1}{r}\right) = \sum_{p} \sum_{r=1}^{n} \frac{1}{r \times p^{r}} = \sum_{p \text{ prime}} \frac{1}{p}$$

$$\xrightarrow{also\ based\ on\ Taylor\ series,\quad x=1} \log\left(\frac{1}{1-x}\right) = \sum_{n=1}^{\infty} \frac{x^n}{n} = \sum_{r=1}^{n} \frac{1}{r} = \log(n)$$

$$\xrightarrow{substituting\ into\ previous\ eq.} \log\left(\sum_{r=1}^{n} \frac{1}{r}\right) = \log(log(n)) = \sum_{p \text{ prime}} \frac{1}{p}$$

$$\xrightarrow{finally} C(n) = n\left(\sum_{p \text{ prime}} \frac{1}{p}\right) = n(\log(log(n)))$$

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

همانطور که مشــاهده می‌کنیم، الگوریتم از نظر زمانی نســبت به الگوریتم عادی تعیین اعداد اول (یعنی با روش تقسـیم کردن) کاراتر است زیرا نه تنها عمل تقسـیم از اعمال جمع و ضرب گران‌تر و زمان‌بر تر است، بلکه تعداد اعمالی که الگوریتم انجام می‌دهد نیز کمتر می‌باشــد؛ از آنجا که در زمان رســیدن به حد $n$ متوقف شــده 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:

$$RAMusage(A) = RAMusage([2, \dots, n]) = RAMusage\left(\sum_{i=2}^{n} 1\right) \rightarrow Space\ Complexity = O(n)$$

با اینکه کارایی فضایی الگوریتم برابر $\mathrm{O}(n)$ است اما فضای استفاده شده از رم برای اعداد بزرگ بسیار زیاد است، برای مثال اگر $n = 10^{12}$ 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.

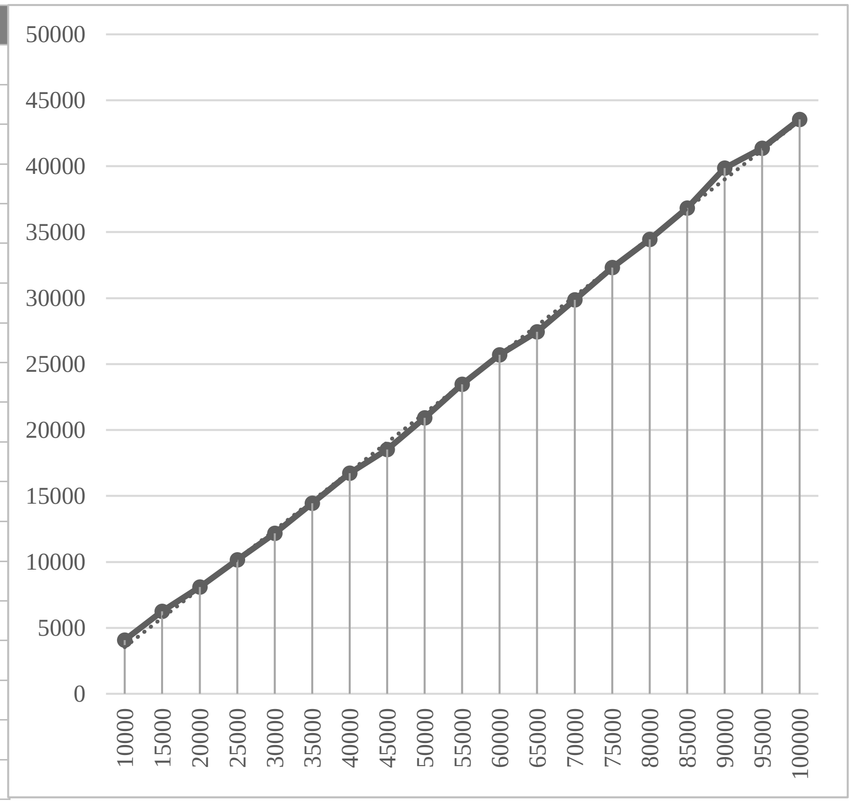
**10 – پ)** برنامه‌های پایتونی که برای اندازه‌گیری تجربی مصرف زمان و حافظه‌ی الگوریتم اراتوستنس نوشته شده و کد آن‌ها در پیوست قرار داده شده است، به ترتیب از ماژول‌های *time* و *tracemalloc* پایتون استفاده می‌کنند. با ماژول *time* زمان اجرای 10 بارِ تابع غربال اراتوستنس را اندازه‌گیری کرده و آن را برای 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 $\mathrm{O}(n)$ 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.

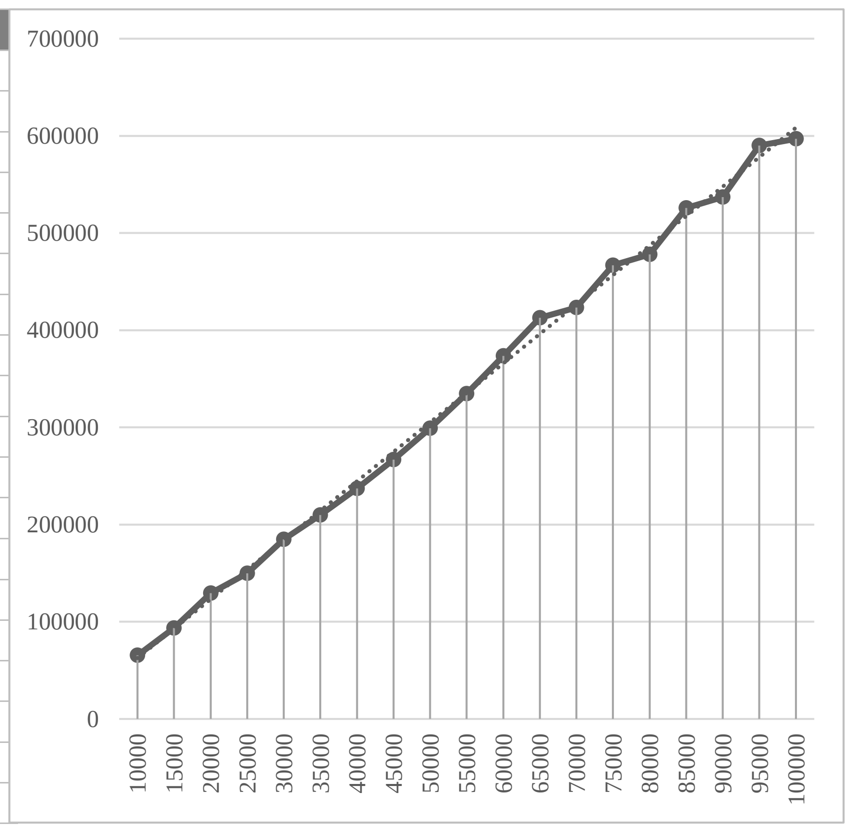
نتایج بررسی تجربی الگوریتم از اجرای برنامه‌ها بر روی کامپیوتری با مشخصات زیر جمع‌آوری شدند:

*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)
```