تمرینات فصل دو

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

1. برای هر زوج تابع (A, B) در جدول، مشخص کنید که رابطه تابع A و تابع B با کدام یک از نمادهای مجانبی O ، o ، Ω ، ω و Θ قابل بیان است. فرض کنید $k \geq 1$ ، $\varepsilon > 0$ و $c > 1$ همه ثابت هستند. پاسخ‌های نهایی خود را به شکل «بله» یا «خیر» در هر یک از خانه‌های جدول بنویسید.

جواب:

	A	B	O	o	Ω	ω	Θ
1	$\lg^k n$	n^ε	بله	بله	خیر	خیر	خیر
2	n^k	c^n	بله	بله	خیر	خیر	خیر
3	$\sqrt{n}$	$n^{\sin n}$	خیر	خیر	خیر	خیر	خیر
4	2^n	$2^{n/2}$	خیر	خیر	بله	بله	خیر
5	$n^{\lg c}$	$c^{\lg n}$	بله	خیر	بله	خیر	بله
6	$\lg(n!)$	$\lg(n^n)$	بله	خیر	بله	خیر	بله

1) $A = (\log n)^k = m^k \quad m \in o(n),\ B = n^\varepsilon \ \rightarrow \ \lim_{n \to \infty} \frac{(\log n)^k}{n^\varepsilon} = \frac{m^k}{n^\varepsilon} = 0 \ \rightarrow \ A \in o(B)$

2) $A = n^k,\ B = c^n \ \xrightarrow{exponential\ always\ grows\ faster} \ \lim_{n \to \infty} \frac{n^k}{c^n} = 0 \ \rightarrow A \in o(B)$

3) از آنجا که $A = \sqrt{n} = n^{\frac{1}{2}}$ و $B = n^{\sin n}$ و برای بی‌نهایت n، مقدار $\sin n$ می‌تواند کوچکتر یا بزرگتر از $\frac{1}{2}$ باشد، نه $\frac{A}{B}$ و نه $\frac{B}{A}$ هیچ کدام کراندار نیستند و بنابراین حدشان در بی‌نهایت ناموجود است.

پس این دو تابع نمی‌توانند با هم به طور حدی مقایسه شوند و ارتباطی ندارند.

4) $A = 2^n,\ B = 2^{\frac{n}{2}},\ we\ know\ \frac{n}{2} \in o(n) \ \rightarrow \ \lim_{n \to \infty} \frac{2^n}{2^{\frac{n}{2}}} = \infty \ \rightarrow \ A \in \omega(B)$

5) $n^{\log_2 c} = \left(2^{\log_2 n}\right)^{\log_2 c} = \left(2^{\log_2 c}\right)^{\log_2 n} = c^{\log_2 n} \ \rightarrow \ \lim_{n \to \infty} \frac{n^{\log_2 c}}{c^{\log_2 n}} = 1 \ \rightarrow \ A \in \Theta(B)$

6) $\xrightarrow{Taylor\ series,\ \ x \in \mathbb{R}^+\ \ n \in \mathbb{N}} e^x = 1 + \sum_{n=1}^{\infty} \frac{x^n}{n!} \ \rightarrow\ e^x \geq \frac{x^n}{n!} \xrightarrow{x=n} e^x \geq \frac{n^n}{n!} \rightarrow n^n \geq \left(\frac{n}{e}\right)^n$

$\xrightarrow{we\ know} \log(n!) = \sum_{k=1}^{n} \log(k) \geq 0 = n \log\left(\frac{n}{n}\right) = n\log(n) - n$

$\xrightarrow{and} \log(n!) = \sum_{k=1}^{n} \log(k) \leq \sum_{k=1}^{n} \log(n) = n\log(n)$

$\xrightarrow{so\ together} n\log(n) - n \leq \log(n!) \leq n\log(n)$

$\xrightarrow{finally} \exists\ c_1 > 0{:}\ c_1 \log(n!) \geq\ n\log(n)\ \ and\ \ \exists\ c_2 > 0{:}\ c_2 \log(n!) \leq\ n\log(n)$

$\xrightarrow{\log(n^n) = n\log(n)}\ c_2 \log(n!) \leq \log(n^n) \leq c_1 \log(n!) \rightarrow\ B \in \Theta(A) \rightarrow\ A \in \Theta(B)$

---

2. درستی هر یک از این ادعاها را ثابت کنید. **جواب:**

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

$$A = \sum_{i=0}^{n} a^i = a^n + \sum_{i=0}^{n-1} a^i\ = a^n + B \xrightarrow{\sum_{j=0}^{n} a^j = \frac{a^{n+1}-1}{a-1}} B = \sum_{i=0}^{n-1} a^i = \frac{a^n - 1}{a - 1}$$

$$\rightarrow \lim_{n\rightarrow\infty} \frac{\frac{a^n-1}{a-1}}{a^{n-1}} = 1\ \rightarrow B \in \Theta(a^{n-1})\ \rightarrow\ A = a^n + \Theta(a^{n-1}) =\ \Theta(a^n) + \Theta(a^{n-1})$$

$$\xrightarrow{a^{n-1} \in \Theta(a^n)} A = \Theta(a^n)\ \rightarrow \sum_{i=0}^{n} a^i \in \Theta(a^n)$$

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

$$\xrightarrow{upper\ bound:\ all\ i=n} A = \sum_{i=1}^{n} i^k =\ \sum_{1}^{n} n^k = n.n^k = n^{k+1}\ \rightarrow\ A \in O(n^{k+1})$$

$$\xrightarrow{lower\ bound}\ n^k < \sum_{i=1}^{n} i^k \rightarrow A \in \omega(n^k) \xrightarrow{1^k = (n-n+1)^k\ so\ A\ is\ polynomial}\ A \in \Omega(n^{k+1})$$

$$\xrightarrow{upper\ and\ lower\ bound} A \in \Theta(n^{k+1})$$

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

$$\sum_{1}^{n} \frac{1}{i} = \int_{1}^{n+1} \frac{1}{x} dx = \ln(x+1) = \ln\,(n+1) \in \Theta(\ln n)$$

ت) $\sum_{i=1}^{n} \frac{1}{i^2} \in \Theta(1)$

$$A = \sum_{i=1}^{n} \frac{1}{i^2} = \int_1^n \frac{dx}{x^2} = 1 - \frac{1}{n} \ \rightarrow \lim_{n \to \infty} \frac{-1}{n} = 0 \ \ and \lim_{n \to \infty} 1 = 1 \ \rightarrow \ A \in \Theta\left(1 - \frac{1}{n}\right)$$

$$\rightarrow A \in \Theta\left(1 - \frac{1}{n}\right) \ \rightarrow \lim_{n \to \infty} \frac{-1}{n} = 0 \ \rightarrow \frac{-1}{n} \in \omega(1) \ \rightarrow \ A \ \in \ \Theta(1)$$

$$\rightarrow \ \exists\, c_1 > 0{:}\, c_1 \times 1 \geq \ A \ \ and \ \exists\, c_2 > 0{:}\, c_2 \times 1 \leq \ A \xrightarrow{based\ on\ def} A \in \Theta(1)$$

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

$$\xrightarrow{n = 2k,\ \ k \in \mathbb{N}} A = \sum_{i=0}^{n} \binom{n}{i} i = 1 \times \binom{n}{1} + 2 \times \binom{n}{2} + \cdots + (n-1) \times \binom{n}{n-1} + n\binom{n}{n}$$

$$\xrightarrow{\binom{n}{r} = \binom{n}{n-r}\ so\ we\ factorize\ duplicates} A = \binom{n}{1}(n - 1 + 1) + \cdots + \binom{n}{n/2}\left(n - \frac{n}{2} + \frac{n}{2}\right)$$

$$\xrightarrow{\left(n - \frac{n}{2} + \frac{n}{2}\right) = n\ so\ we\ again\ factorize} A = n\left(\binom{n}{1} + \cdots + \binom{n}{n/2}\right) = n \times \frac{[2^n]}{2}$$

$$\xrightarrow{n = 2k+1,\ \ k \in \mathbb{N}} A = 1 \times \binom{n}{1} + \cdots + \left(\left[\frac{n}{2}\right] + 1\right) \times \binom{n}{\left[\frac{n}{2}\right] + 1} + \cdots + (n-1) \times \binom{n}{n-1}$$

$$\xrightarrow{like\ before} A = n\left(\binom{n}{1} + \cdots + \binom{n}{n/2}\right) + \left(\left[\frac{n}{2}\right] + 1\right) \times \binom{n}{\left[\frac{n}{2}\right] + 1}$$

$$\rightarrow \ A = n\left(\frac{2^n - \binom{n}{\left[\frac{n}{2}\right]+1}}{2}\right) + \binom{n}{\frac{n+2}{2}}\left(\frac{n+2}{2}\right) \ \rightarrow \ \ A = \frac{n \times 2^n}{2} + C \ \xrightarrow{C\, \in\, o(n2^n)} A \in \Theta(n2^n)$$

---

3. یا درســـتی هر یک از این ادعاها را با اســـتفاده از تعاریف نمادهای مجانبی، ثابت کنید یا با یک مثال نقض، نادرستی یک ادعا را نشان دهید. فرض کنید مقادیر هر یک از توابع، از جایی به بعد، همیشه مثبت خواهند بود.

الف) اگر $f_1(n) \in \Theta\big(g_1(n)\big)$ و $f_2(n) \in \Theta\big(g_2(n)\big)$ باشد، آنگاه $f_1(n) * f_2(n) \in \Theta\big(g_1(n) * g_2(n)\big)$ خواهد بود.

ب) اگر $f_1(n) \in O\big(g_1(n)\big)$ و $f_2(n) \in O\big(g_2(n)\big)$ باشد، آنگاه $f_1(n) - f_2(n) \in O\big(g_1(n) - g_2(n)\big)$ خواهد بود.

پ) اگر $f(n) \in \Theta\big(g(n)\big)$ و $g(n) \in \Theta\big(h(n)\big)$ باشد، آنگاه $f(n) \in \Theta\big(h(n)\big)$ خواهد بود.

ت) اگر $f(n) \in O\big(g(n)\big)$ باشد، آنگاه $\log f(n) \in O(\log g(n))$ خواهد بود.

ث) اگر $f(n) \in O\big(g(n)\big)$ باشد، آنگاه $2^{f(n)} \in O\big(2^{g(n)}\big)$ خواهد بود.

جواب:

3 – الف) بنابر تعریف تتای بزرگ و مثبت بودن مقادیر داریم:

$$f_1(n) \in \Theta\big(g_1(n)\big) \;\to\; c_1, c_2 > 0 \;\to\; c_2 g_1(n) \le f_1(n) \le c_1 g_1(n)$$
$$f_2(n) \in \Theta\big(g_2(n)\big) \;\to\; c_3, c_4 > 0 \;\to\; c_4 g_2(n) \le f_2(n) \le c_3 g_2(n)$$
$$\xrightarrow{f_1(n)\cdot f_2(n)} c_2 c_4 g_1(n) g_2(n) \le f_1(n) f_2(n) \le c_1 c_3 g_1(n) g_2(n)$$
$$\xrightarrow{c_1\ c_3 = c_5 > 0,\ \ c_2\ c_4 = c_6 > 0} c_6 g_1(n) g_2(n) \le f_1(n) f_2(n) \le c_5 g_1(n) g_2(n)$$
$$\xrightarrow{based\ on\ \theta\text{−}notation\ definition} f_1(n) f_2(n) \in \Theta\big(g_1(n) g_2(n)\big)$$

3 – ب) بنابر تعریف اُی بزرگ و همواره مثبت بودن توابع از جایی به بعد، می‌توانیم بنویسیم:

$$f_1(n) \in O\big(g_1(n)\big) \;\to\; c > 0 \;\to\; f_1(n) \le c g_1(n)$$
$$f_2(n) \in O\big(g_2(n)\big) \;\to\; c > 0 \;\to\; f_2(n) \le c g_2(n)$$
$$\xrightarrow{f_1(n) - f_2(n)} f_1(n) - f_2(n) \le c g_1(n) - c g_2(n) = c\big(g_1(n) - g_2(n)\big)$$
$$\xrightarrow{based\ on\ O\text{−}notation\ definition} f_1(n) - f_2(n) \in O\big(g_1(n) - g_2(n)\big)$$

3 – پ) بنابر تعریف نماد تتای بزرگ داریم $f(n) \in \Theta\big(g(n)\big)$ زمانی صـــحت دارد که $f(n)$ برای مقادیر بزرگ n از بالا و پایین کرانی برابر ضرب‌های ثابت مثبتی از $g(n)$ باشد. یعنی اگر ثابت‌های $c_1, c_2 > 0$ را داشته باشیم و عدد صـــحیح $n_0 \ge 0$، برای تمامی $n \ge n_0$ آنگاه $c_2 g(n) \le f(n) \le c_1 g(n)$. به همین شـــکل برای $g(n) \in \Theta\big(h(n)\big)$ داریم $c_3, c_4 > 0$ که $c_4 h(n) \le g(n) \le c_3 h(n)$. از آنجا که $c_4 h(n) \le g(n)$ و بوده و

$f(n)$ هم از ضرب ثابت مثبت آن بزرگتر مساوی‌ست و این برای نیمه‌ی دیگر نامساوی‌ها نیز صدق می‌کند:

$$c_2 g(n) \le f(n) \le c_1 g(n) \ \rightarrow \ c_2 c_4 h(n) \le f(n) \le c_1 c_3 h(n) \ \rightarrow \ c_6 h(n) \le f(n) \le c_5 h(n)$$

که $c_1\, c_3 = \ c_5 > 0$ و $c_2\, c_4 = \ c_6 > 0$ و بنابر تعریف تتای بزرگ $f(n) \in \Theta\big(h(n)\big)$.

3 – ت) با توجه به تعریف اُی بزرگ داریم:

$$\log f(n) \in \mathrm{O}(\log g(n)) \xrightarrow{c>0} \ \log f(n) \le c \log g(n)$$

در صــورت داشــتن $g(n) = 1$، به علت شــرط ادعا یعنی $f(n) \in \mathrm{O}\big(g(n)\big)$ باید تابع $f(n)$ نیز ثابت باشــد و می‌تواند $f(n) = 2$ باشد. آنگاه $\log f(n) = 1$ اما $\log g(n) = 0$ که با هیچ ضریب $c \ > \ 0$ بزرگتر یا مساوی با یک نمی‌شود و با مثال این ادعا نقض می‌شود. فقط در صورت داشتن شرط $g(n) \in \omega(1)$ ادعا صحت دارد.

در حالت وجود شرط بالا، برای اثبات ادعا داریم:

$$f(n) \in \mathrm{O}\big(g(n)\big) \xrightarrow{c>0} f(n) \le cg(n) \ \rightarrow \log\big(f(n)\big) \le \log\big(cg(n)\big)$$
$$\rightarrow \log\big(f(n)\big) \le \log\big(g(n)\big) + \log(c) \xrightarrow{\log(c) is\ unimportant\ in \lim} \log\,(f(n)) \le c' \log\big(g(n)\big)$$
$$\xrightarrow{based\ on\ O-notation\ definition} \log\big(f(n)\big) \in O\big(\log\big(g(n)\big)\big)$$

3 – ث) بنـابر تعریف اُی بزرگ می‌دانیم کـه برای $f(n) = 3n$ و $g(n) = n$ آنگـاه $f(n) \in \mathrm{O}\big(g(n)\big)$ زیرا بـا ضریب ثابت $c > 0$ برای $g(n)$ مرتبه رشد $f(n)$ هیچگاه بزرگتر از $g(n)$ نمی‌شود. اما $2^{f(n)} \notin \mathrm{O}\big(2^{g(n)}\big)$ زیرا مرتبه رشد $2^{3n} = 8^n$ همواره از 2^n بیشتر است. پس ادعا با مثال نقض رد می‌شود.

---

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

```
ALGORITHM  UniqueElements(A[0..n − 1])
// Determines whether all the elements in a given array are distinct
// Input: An array  A[0..n − 1]
// Output: Returns “true” if all the elements in A are distinct and “false” otherwise
if  n = 1
    return  true
else if not  UniqueElements(A[0..n − 2])
    return  false
else if not  UniqueElements(A[1..n − 1])
    return  false
else
    return  A[0] ≠ A[n − 1]
```

الف) چرا این الگوریتم بازگشتی، جواب درست مسأله را برمی‌گرداند؟

ب) کارایی زمانی الگوریتم چقدر است؟ چرا الگوریتم ناکارا است؟

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

جواب:

4 – الف) برای اثبات درستی الگوریتم از استقرا استفاده می‌کنیم.

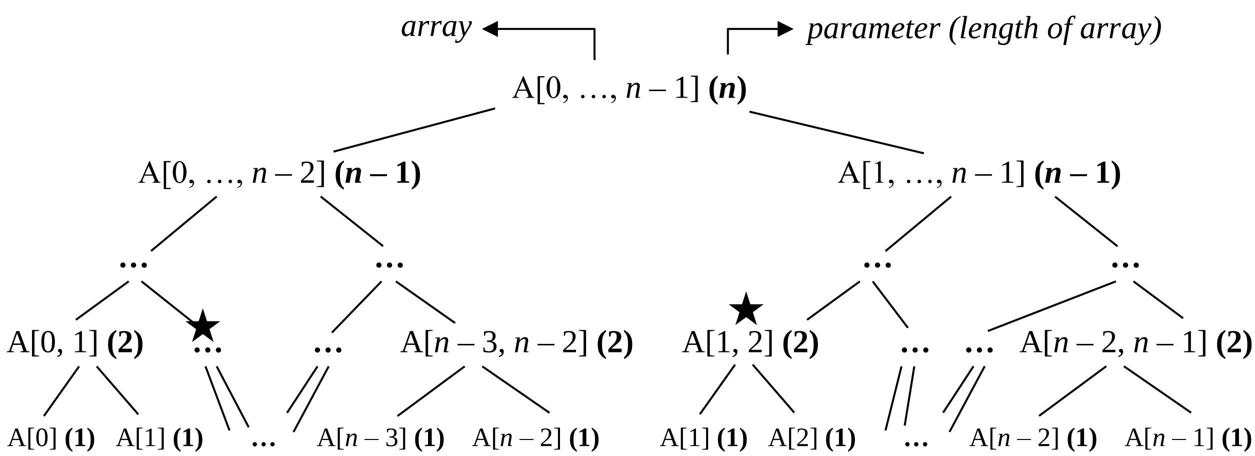
- ابتدا برای حالت پایه، آرایه‌ای با یک عضو (یعنی $n = 1$) را می‌سنجیم که چون تنها عضو آن حتما یگانه است، خروجی *True* صحیح است.

- سپس برای ورودی‌ای با دو عضو ($n = 2$)، الگوریتم از شرط خط اول رد می‌شود و خط سوم را بررسی می‌کند؛ یعنی الگوریتم را برای آرایه‌ای متشکل از تنها عضو اول آرایه بررسی می‌کند. ار آنجا که خروجی آن حتما *True* است، شرط برقرار نشده و الگوریتم خط پنجم را برای آن آرایه‌ی دو عضوی می‌سنجد. در آن خط به صورت بازگشتی، الگوریتم را برای آرایه‌ای متشکل ار تنها عضو دوم آرایه‌ی اصلی بررسی می‌کند که خروجی آن حتما *True* است. بنابراین شرط برقرار نشده و الگوریتم از این خط هم می‌گذرد و در خط آخر با مقایســه‌ی عنصــر اول و دوم (آخر) آرایه، در صــورت برابری *False* و نابرابری *True* برمی‌گرداند که پاسخ مورد نظر ما برای سنجش یکتا بودن عناصر آرایه است.

- به همین شکل برای آرایه با سه عضو ($n = 3$)، الگوریتم آن را از دو طرف به آرایه‌های دو عضوی و بعد تک عضوی تقسیم کرده و سنجش یکتایی اعضا را انجام می‌دهد و در صورت یکتا بودن عناصر آرایه‌های دو عضوی آن (یعنی دو آرایه‌ی $A[0, n – 2]$ و $A[n – 2, n – 1]$) عناصر اول و آخر آرایه‌ی سه عضوی را مقایسه می‌کند که قبلا با هم مقایسه نشده بودند و نتیجه را برمی‌گرداند. پس بر اساس استقرا، درستی الگوریتم برای آرایه‌ی n عنصری اثبات می‌شود.

4 – ب) در محاسبه‌ی کارایی زمانی الگوریتم، باید تعداد فراخوانی عملیات پایه (که در اینجا مقایسه است) را برای ورودی با پارامتر n به دست آوریم. برای سنجش کارایی الگوریتم‌های بازگشتی که بیش از یکبار خود را فراخوانی می‌کنند، رسم یک درخت می‌تواند به راحتی کارایی زمانی را به ما نشان دهد.

پس برای ورودی $A[0, \ldots, n-1]$ که تعداد اعضای آرایه برابر با n می‌باشد، پارامتر را n در نظر گرفته و می‌بینیم که برای هر بازگشت به خود الگوریتم، از مقدار n یکی کم می‌شود.



برای $C(n)$ یعنی تعداد تکرارهای بازگشت، با شمارش گره‌های درخت با l که طبقه گره در درخت باشد داریم:

$$C(n) = \sum_{l=0}^{n-1} 2^l = 2^n - 1$$

که این کارایی زمانی در مرتبه رشد $O(2^n)$ قرار دارد و نشانگر الگوریتم نمایی‌ست که برای مقادیر کوچک هم در زمانی طولانی اجرا می‌شود. این نتیجه را می‌توانیم با بررسی درخت هم مشاهده کنیم زیرا همانطور که در درخت بالا می‌بینیم، این الگوریتم هر آرایه را چند بار بررسی می‌کند؛ برای مثال، در دو مکان که علامت ★ قرار داده شده، آرایه‌ی $A[1, 2]$ بررسی می‌شود. پس الگوریتم کند و ناکاراست.

4 – پ) برای الگوریتم بازگشتی کارای مسئله می‌توانیم فقط از یک فاراخوانی بازگشتی استفاده کنیم و به جای پیش رفتن در آرایه از دو طرف، فقط از یک طرف جلو رویم و دنبال عنصر در آرایه بگردیم و در صورت وجود آن را False بازگردانیم، در غیر این صورت تا جایی پیش رویم که در آرایه فقط یک عضو بماند، و در آن حالت True برگردانیم.

```
Algorithm: UniqueElements(A[0…n – 1])
    // determines whether all the elements in a given array are distinct
    // Input: array A[0, …, n – 1]
    // Output: returns True if all the elements in A are distinct and False if otherwise
    if n = 1 then
        return True
    for i ← 1 to n – 1 do
        if A[i] = A[0] then   // checks if current element exists in the rest of the array
            return False
    else then
        return UniqueElements(A[1:])   // moving to next element
```

به وضوح می‌توان دید که به علت یک بار فراخوانی بازگشتی، این الگوریتم از الگوریتم قبلی کارایی زمانی بهتری دارد. برای اثبات ریاضی آن، تعداد تکرار عملیات پایه (مقایسه درون حلقه) برای پارامتر n (طول آرایه) را باید در بدترین حالت ممکن محاسبه کنیم زیرا برای یک آرایه با طول یکسان، الگوریتم با تعداد فراخوانی‌های بازگشتی متفاوت می‌تواند اجرا می‌شود.

اگر $C_{worst}(n)$ تکرار عملیات پایه در بدترین حالت ممکن باشد، مشاهده می‌کنیم که تعداد فراخوانی بازگشتی یکی کمتر از طول آرایه است زیرا برای عنصر اول فراخوانی انجام نمی‌شود، پس $C(n - 1)$. همچنین برای هر فراخوانی، حلقه‌ای بررسی می‌شود که تعداد تکرار آن، یکی کمتر از تعداد اعضای پارامتر ورودی در آن مرحله‌ی بازگشتی‌ست یعنی $(n - 1)$. از آنجا که برای پارامتر 1 (یعنی آرایه‌ی ورودی با یک عضو) عملیات پایه (مقایسه عناصر) انجام نمی‌شود، برای حالت اولیه‌ی رابطه‌ی بازگشتی داریم $C(1) = 0$.

پس رابطه‌ی بازگشتی را به شکل زیر می‌نویسیم:

$$C_{worst}(n) = \underbrace{C_{worst}(n-1)}_{\textit{recursive calls}} + \underbrace{(n-1)}_{\textit{loop}}$$

حال با عقب آمدن و جایگزینی آن را حل می‌کنیم:

$$
\begin{aligned}
C_{worst}(n) &= C(n-1) + (n-1) \\
&= C(n-2) + (n-2) + (n-1) = C(n-2) + 2n - (1+2) \\
&= C(n-3) + (n-3) + 2n - (1+2) = C(n-3) + 3n - (1+2+3) \\
&= \ldots \\
&= C(n-i) + i \times n - (1 + \ldots + i) \\
&= \ldots \\
&= C(n-(n-1)) + (n-1) \times n - (1 + \ldots + (n-1)) \\
&= C(1) + n(n-1) - \frac{n((n-1)+1)}{2} \\
&= \frac{2n(n-1) - n(n)}{2} = \frac{2n^2 - 2n - n^2}{2} = \frac{n(n-2)}{2}
\end{aligned}
$$

پس کارایی زمانی در مرتبه رشد $O(n^2)$ قرار دارد که نسبت به الگوریتم پیشین سریع‌تر است.

---

5. این الگوریتم را در نظر بگیرید.

```
ALGORITH  Example(A[0..n − 1], B[0..n − 1])
// Input: Two arrays A and B , each of n elements

count ← 0
for i ← 0 to n − 1 do
    total ← 0
    for j ← 0 to n − 1 do
        for k ← 0 to j do
            total ← total + A[k]
    if B[i] = total
        count ← count + 1
return count
```

الف) الگوریتم، چه مسأله‌ای را حل می‌کند؟

ب) عمل پایه‌ای الگوریتم چیست؟ تعداد دفعات تکرار عمل پایه‌ای (به شکل تابعی از n) چیست؟

پ) کارایی زمانی الگوریتم چقدر است؟

ت) در صورت امکان، الگوریتم را با هدف بهبود کارایی زمانی آن، بازنویسی کنید و کارایی زمانی الگوریتم جدید را مشخص کنید. اگر نمی‌توان الگوریتم را بهتر کرد، ثابت کنید که بهینه است؛ یعنی نمی‌توان الگوریتمی طراحی کرد که جواب مسأله را با تعداد عملیات کمتری به دست آورد.

جواب:

5 – الف) برای مقدار *total* که از عناصر A به دست می‌آید، با ردگیری جمع درون حلقه‌ی کد به ازای یک آرایه نمونه مانند A = [10, 5, 11, 9] در تکرارهای 1 و 2 و 3 و 4 (به تعداد طول آرایه) داریم:

1) total = total + 10
2) total = total + 10 + 5
3) total = total + 10 + 5 + 11
4) total = total + 10 + 5 + 11 + 9

$\rightarrow$ total = 4 × 10 + 3 × 5 + 2 × 11 + 1 × 9 $\rightarrow$ total = $\sum_{i=1}^{n} i \times A[-i]$

این الگوریتم پس از محاسبه مقدار *total* برای *A* به اعضای هر عضو *B*، الگوریتم عضو حال حاضر *B* را با مقدار حاصل مقایسه می‌کند و در صورت برابری، آن را می‌شمارد. به زبانی ساده، الگوریتم بررسی می‌کند که مقدار *total* چندبار در آرایه‌ی *B* تکرار شده است و آن تعداد را در خروجی برمی‌گرداند.

5 – ب) عمل پایه‌ای الگوریتم، درونی‌ترین عملیات در حلقه‌هاست که جمع در محاسبه‌ی total می‌باشد. برای تعداد تکرار آن به شکل تابعی از *n*، از آنجا که هر سه حلقه به طور کامل پیش می‌روند، می‌نویسیم:

$$C(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{j} 1 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(j - 0 + 1) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} j + \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1$$
$$= \sum_{i=0}^{n-1}\frac{n\big((n-1)+0\big)}{2} + \sum_{i=0}^{n-1}\big((n-1) - 0 + 1\big) = \sum_{i=0}^{n-1}\frac{n(n-1)}{2} + \sum_{i=0}^{n-1} n$$
$$= \big((n-1) - 0 + 1\big)\frac{n(n-1)}{2} + \big((n-1) - 0 + 1\big)n = \frac{n^2(n-1)}{2} + n^2$$
$$= \frac{n^2(n-1) + 2n^2}{2} = \frac{n^3 + n^2}{2} = \frac{n^2(n+1)}{2}$$

5 – پ) کارایی زمانی الگوریتم برای عملیات پایه‌ای جمع و پارامتر *n* (طول یکی از آرایه‌های ورودی)، بنابر تعداد تکرارهای عملیات پایه‌ای محاسبه شده در بخش ب برابر است با $C(n) \in O(n^3)$.

5 – ت) در الگوریتم داده شده، مقدار *total* به تعداد *n* بار محاسبه می‌شود در حالی که مقدار آن همواره یکسان است. از طرفی به علت وجود سه حلقه‌ی تودرتو، الگوریتم بسیار کند و ناکاراست. با جداکردن حلقه‌ها و حذف حلقه‌ی اضافه و جایگزینی آن با فرمول یافت شده، می‌توانیم الگوریتمی کارا بنویسیم.

```
Algorithm: Example(A[0…n – 1], B[0…n – 1])
    // calculates the result of formula for A and compares it with elements of B
    // Input: arrays A[0, …, n – 1] and B[0, …, n – 1]
    // Output: returns number of times the result is in B
    total ← 0
    for i ← 1 to n do
        x = n – i
        total ← total + (i × A[x])
    count ← 0
    for j ← 0 to n – 1 do   // moving through elements of B
        if B[ j] = total do
            count ← count + 1
    return count
```

همان‌طور که مشاهده می‌کنیم، این الگوریتم متشکل از دو بخش مستقل است که هرکدام از یک حلقه تشکیل شده‌اند. پس برای محاسبه کارایی زمانی بایستی max دو بخش را بیابیم. از آنجا که عملیات پایه‌ی جمع در بخش اول حتما به تعداد پارامتر n تکرار می‌شود، $f(n) = n$. همچنین در حلقه‌ی دوم، عملیات پایه‌ی مقایسه برای پارامتر n در بدترین حالت n تکرار دارد پس $g(n) = n$. بنابر قانون داریم:

$$\max\big(f(n), g(n)\big) \in \Theta\big(f(n) + g(n)\big) \rightarrow \max(n, n) \in \Theta(n + n) = \Theta(2n) \in \Theta(n)$$

---

6. الف) فرض کنید $n - 1$ عدد صحیح یکتا در محدوده 1 تا n در آرایه $A[0..n-2]$ ذخیره شده است. الگوریتمی با کارایی زمانی $O(n)$ برای پیدا کردن عدد صحیحی در محدوده 1 تا n که در A نیست، طراحی کنید. کارایی فضایی الگوریتم باید $O(1)$ باشد؛ یعنی الگوریتم علاوه بر فضای اشغال شده توسط آرایه A ، مجاز به استفاده از بیشتر از چند واحد حافظه نخواهد بود.

ب) فرض کنید $n-2$ عدد صحیح یکتا در محدوده 1 تا n در آرایه $A[0..n-3]$ ذخیره شده است. الگوریتمی با کارایی زمانی $O(n)$ برای پیدا کردن دو عدد صحیحی در محدوده 1 تا n که در آرایه A نیستند، طراحی کنید. کارایی فضایی الگوریتم باید $O(1)$ باشد؛ یعنی الگوریتم علاوه بر فضای اشغال شده توسط آرایه A، مجاز به استفاده از بیشتر از چند واحد حافظه نخواهد بود.

جواب:

6 – الف) می‌توانیم به سادگی با کم کردن مجموع اعداد درون آرایه از مقدار جمع n عدد (که با داشتن طول آرایه به دست می‌آید)، عدد ناموجود را بیابیم. البته لازم به ذکر است که در صورت بزرگ بودن لیست، محاسبه جمع سخت‌تر می‌شود؛ پس با استفاده از اصل جابه‌جایی در جمع و تفریق، اعمال را همزمان انجام دهیم. این الگوریتم همواره $n-1$ بار عملیات پایه‌ی درون حلقه یعنی جمع را تکرار می‌کند که در مرتبه رشد $O(n)$ است.

```
Algorithm: MissingNumber(A[0…n – 2])
    // finds the missing number in array of numbers from 1 to n
    // Input: array A[0, …, n – 2]
    // Output: returns missing number
    sum ← 0
    for i ← 1 to n do   // n is length of array plus one
        sum ← sum + i
        sum ← sum – A[i – 1]
    return sum
```

6 – ب) به همین شکل، با تعمیم الگوریتم قسمت الف می‌توانیم دو عدد را پیدا کنیم. ابتدا براساس الگوریتم قبل، جمع دو عدد را از تفریق جمع اعداد موجود از جمع کل اعداد تا n پیدا می‌کنیم. می‌دانیم که به علت نابرابر بودن دو عدد، یکی از آن‌ها برابر یا کوچکتر از میانگینشان و دیگری حتما از میانگینشان بزرگتر خواهد بود. پس با کسر جمع اعداد موجودی که کمتر یا مساوی میانگین هستند از جمع کل اعداد طبیعی از 1 تا میانگین، می‌توانیم عدد اول را بیابیم. سپس با کم کردن آن مجموع از جمع دو عدد، عدد دوم را پیدا می‌کنیم. از آنجا که این الگوریتم دارای سه حلقه‌ی مستقل است، کارایی زمانی آن به مقدار *max* میان آن‌ها بستگی دارد که در حلقه‌ی اول با تعداد تکرار n اتفاق می‌افتد، پس کارایی زمانی آن $O(n)$ و کارایی فضایی آن $O(1)$ می‌باشد.

```
Algorithm: MissingNumbers(A[0…n – 3])
    // finds the two missing numbers in array of numbers from 1 to n
    // Input: array A[0, …, n – 3]
    // Output: returns stack of the two missing numbers
    sumTwo ← 0   // will be sum of the two missing numbers
    for i ← 1 to n do   // n is length of array plus two
        sumTwo ← sumTwo + i
        sumTwo ← sumTwo – A[i – 1]
    average ← sumTwo / 2
    totalHalf ← 0   // sum of 1 to average
    for i ← 1 to average do
        totalHalf ← totalHalf + i
    sumHalf ← 0   // sum of numbers in array smaller than average of the two
    for i ← 1 to n – 2 do   // n – 2 is length of array
        if A[i] ≤ average then
            sumHalf ← sumHalf + A[i]
    first ← totalHalf – sumHalf
    second ← sumTwo – first
    return  first, second
```

---

7. الف) فرض کنید A ماتریسی است $n \times n$ و تنها شامل 0 و 1 است، و اینکه در هر سطر A ، 1 ها قبل از 0 ها قرار گرفته‌اند. اگر ماتریس A در حافظه ذخیره شده باشد، الگوریتمی با کارایی زمانی $O(n)$ طراحی کنید که با آن بتوان سطری از A را که شامل بیشترین تعداد 1 باشد، پیدا کرد.

ب) فرض کنید A ماتریسی است $n \times n$ و فقط شامل 1 و 0 است، و اینکه در هر سطر A ، 1 ها قبل از 0 ها قرار گرفته‌اند. به علاوه، فرض کنید که برای هر $i = 0, 1, \cdots, n-2$ ، تعداد 1 ها در سطر i ، برابر یا بیشتر از تعداد 1 ها در سطر $i + 1$ است. اگر ماتریس A در حافظه ذخیره شده باشد، الگوریتمی با کارایی زمانی $O(n)$ طراحی کنید که با آن بتوان تعداد 1 ها را در ماتریس A پیدا کرد.

<u>جواب:</u>

7 – الف) برای یافتن سطر دارای بیشترین تعداد 1 در این ماتریس خاص، می‌توانیم الگوریتمی طراحی کنیم که از سطر اول خانه‌ی اول شروع کرده و به راست حرکت کند تا بزرگترین index که در آن عدد 1 وجود دارد را بیابد، سپس در سطر بعد از این index به بعد را مقایسه کند و درصورت وجود عدد 1 در indexهای بزرگتر از آن، سطر جدید را به عنوان سطر دارای بیشترین 1 ذخیره کند. کد پایتون در پیوست 1 قابل مشاهده است.

Algorithm: MaxRow($A[n \times n]$)

// finds row of special matrix with most 1s

// Input: matrix $A_{n \times n}$

// Output: returns index of row

index ← 0   // intended as max index of 1 in a row

row ← *None*   // intended as index of row with max 1s

for *i* ← 0 **to** *n* – 1 **do**

// preventing error if max index has passed last index of row

// and there are no further indexes to check for more 1s

while *index* ≠ *n* **and** *A*[*i*][*index*] = 1 **do**

index ← *index* + 1

row ← *i*

return *row*   // will return *None* if matrix has no 1s

در بررسی کارایی زمانی این الگوریتم، نیاز به بررسی بدترین حالت داریم زیرا برای ماتریس‌های $n \times n$ با اشکال متفاوت، کارکرد الگوریتم یکسان نیست. اگر تمامی ماتریس متشکل از اعداد یک باشد، در اولین اجرای حلقه این الگوریتم n بار عملیات پایه‌ی جمع درون حلقه را انجام می‌دهد و سپس در اجرای دوم حلقه، سعی می‌کند وجود عدد یک را بعد از آن *index* برای سطر بعد بسنجد که به علت برابر بودن $index = n - 1$ و داشتن بیشترین تعداد 1، حلقه شکسته شده و پایان میابد. می‌بینیم که همواره این الگوریتم خطی‌ست زیرا از ابتدای سطر شروع کرده و به جلو می‌رود و در سطرهای بعدی از بعد از نقطه‌ی قبل شروع به حرکت می‌کند، گویا که در یک سطر با n عنصر در حرکت است. پس کارایی زمانی $O(n)$ می‌باشد. کد پایتون آنالیز آن در پیوست 1 درج شده است.

7 – ب) با پیاده‌سازی تغییرات کوچکی در الگوریتم ارائه شده در قسمت الف، می‌توانیم به جای حرکت از چپ بالا به راست پایین در ماتریس، از راست بالا به چپ پایین حرکت کنیم. این روند به ما کمک می‌کند تا بزرگترین *index* عدد 1 در سطر اول (که بنابر فرض سوال حتما بیشترین تعداد 1ها را در میان سطرها دارد) بیابیم و هر بار در سطر پایینی به چپ پیش رویم و در هر مرحله تعداد 1ها در آن سطر را (که یکی بیشتر از *index* حال حاضر است) در متغیری ذخیره و با هم جمع کنیم. کارایی زمانی این الگوریم دقیقا مانند الف، $O(n)$ می‌باشد.

Algorithm: OneRepetition(*A*[*n* × *n*])

// finds number of 1s in special matrix

// Input: matrix $A_{n\times n}$

// Output: returns quantity of 1s

index ← *n* – 1 // intended as largest index of 1

quantity ← 0 // intended as quantity of 1s

for *i* ← 0 **to** *n* – 1 **do**

// preventing error if max index has passed first index of row

// and there are no further indexes to check for more 1s

while *index* ≠ –1 **and** A[*i*][*index*] = 0 **do**

index ← *index* – 1

quantity ← *quantity* + *index* + 1 // because index starts from 0 and not 1

return *quantity* // will return 0 if matrix has no 1s

---

8. فرض کنید n قلم داده در گره‌های درخت دودویی n گره‌ای T ذخیره شده باشند.

الف) الگوریتمی بازگشتی با کارایی زمانی $\Theta(n)$ طراحی کنید که درخت دودویی T را به عنوان ورودی بگیرد و مقادیر ذخیره شده در هر گره‌ی درخت را چاپ کند. کارایی فضایی الگوریتم خود را نیز اندازه بگیرید.

ب) اکنون با استفاده از یک پشته، یک الگوریتم غیربازگشتی با کارایی زمانی $\Theta(n)$ طراحی کنید که درخت دودویی T را به عنوان ورودی بگیرد و مقادیر ذخیره شده در هر یک از گره‌های درخت را چاپ کند. کارایی فضایی الگوریتم خود را نیز اندازه بگیرید.

پ) یک الگوریتم غیربازگشـــتی با کارایی زمانی $\Theta(n)$ طراحی کنید که درخت دودویی T را به عنوان ورودی بگیرد و مقادیر ذخیره شـده در هر یک از گره‌های درخت را چاپ کند. کارایی فضـایی الگوریتم‌تان باید $\Theta(1)$ باشد.

<u>جواب:</u>

8 – الف) فرض را بر این می‌گیریم که کلاس درخت تعریف شده است و می‌توانیم از *attribute*های آن اسـتفاده کنیم. از آنجایی که تنها از متغیرهای درون‌تابعی *root* اسـتفاده شـده اسـت، با کمک *In-order tree traversal*، الگوریتم از ریشـه شـروع کرده تا آخرین گره سـمت چپ یعنی برگ حرکت کرده و چاپ می‌کند. کارایی فضـایی این الگوریتم (که در زیر قابل مشاهده است) به علت ایجاد نکردن متغیر بزرگ جدید، همواره O(1) است.

Algorithm: RecursivePrintNode(*Tree.root*)

// prints nodes using recursive method and In-order tree traversal

// Input: root of tree

// Output: prints nodes

// attributes of a node in the *Tree* class:

node.right → the right child of node

node.left → the left child of node

node.data → the data saved on the node

root ← *Tree*.root

if *root* **is not** *None* **then**

// first visit the left children until arriving at leaf

PrintNodeR(*root*.left)

// print the data saved in the node

print(*root*.data, end = “ ,”)

// then visit the right children until arriving at leaf

PrintNodeR(*root*.right)

8 – ب) به مانند الف از حرکت *In-order* در درخت استفاده کرده و مقادیر را در پشته‌ی کمکی ذخیره می‌کنیم. سپس داده‌های درون پشته (که فرزندان چپی هستند) را چاپ کرده و وجود فرزند راستی را برای آن در درخت بررسی می‌کنیم؛ اگر فرزند موجود بود، در فرزندهای چپی آن پیش رفته و به همین شکل به بالا حرکت کرده و چاپ می‌کنیم. در بدترین حالت پشته دارای نیمی از تعداد کل گره هاست، پس کارایی فضایی $O(n)$ می‌باشد.

Algorithm: PrintNodeStack(*Tree*)

// prints nodes of tree with the help of an auxiliary stack and in-order tree traversal
// Input: *Tree*
// Output: prints nodes
// attributes of a node in the *Tree* class:

node.right → the right child of node
node.left → the left child of node
node.data → the data saved on the node

current ← *Tree*.root // the current node

S ← stack() // the auxiliary that will be used for the algorithm's processes

while *True* **do**

if *current* **is not** *None* **then**

// push all the left children into stack to arrive at the leaf

S.push(*current*)

current ← *current*.left

elif *S*.pop() **is not** *None* **then**

// print the left child saved in stack

current ← *S*.pop()

print(*current*.data, end = " ,")

// then visit the ones on the parent's right

current ← *current*.right // is *None* the first time

else then

// after all the prints, stack *S* becomes empty

break

8 – پ) همچنان با *In-order tree traversal* می‌توانیم این مسئله را حل کنیم. تفاوت این االگوریتم با قسمت ب این است که به جای استفاده از پشته برای ذخیره‌ی گره‌هایی از درخت که در آن‌ها پیش رفته‌ایم، از ویژگی *visited* استفاده می‌کنیم که به ما کمک می‌کند تا گره‌های بررسی شده را علامت زده و مشخص کنیم. از آنجایی که به جای پشته تنها با استفاده از یک تک متغیر تمام عملیات‌ها را انجام داده‌ایم، کارایی فضایی در مرتبه رشد O(1) قرار می‌گیرد.

Algorithm: PrintNode(*Tree*)

    // prints nodes without the help of stack, using In-order tree traversal method
    // Input: tree
    // Output: prints nodes
    // attributes of a node in the *Tree* class:
        node.right → the right child of node
        node.left → the left child of node
        node.data → the data saved on the node
    current ← *Tree*.root   // the current node
    while *current* **is** *True* **and** *current*.visited **is** *False* **do**
        if *current*.left **is True and** *curent*.left.visited **is False then**
            // first visit the left children in all subtrees of the root
            current ← *current*.left
        elif *current*.right **is** *True* **and** *curent*.right.visited **is** *False* **then**
            // then visit the right children until leaf
            current ← *current*.right
        else then
            print(*current*.data, end = “ ,”)
            current.visited ←*True*

9. مربع یک گراف جهت‌دار $G = (V, E)$ ، گراف $G^2 = (V, E^2)$ است؛ با این تعریف که $(u, v) \in E^2$ است اگر و فقط اگر، در G مسیری با حداکثر دو یال بین u و v وجود داشته باشد.

الف) با این فرض که G با لیست (های) مجاورت نمایش داده شده باشد، الگوریتم کارایی را برای محاسبه G^2 از روی G طراحی و کارایی زمانی آن را اندازه بگیرید.

ب) با این فرض که G با ماتریس مجاورت نمایش داده شده باشد، الگوریتم کارایی را برای محاسبه G^2 از روی G طراحی و کارایی زمانی آن را اندازه بگیرید.

<u>جواب:</u>

9 – ب) برای ماتریس *G.matrix* از گراف G، از آنجایی که اگر دو یال بین دو گره قرار داشته باشد مقدار G^2 در خانه متناظر آن در *G^2.matrix* نیز برابر 1 می‌شود، برای به دست آوردن گراف G^2 الگوریتم زیر به دست می‌آید که از ضرب ساده ماتریس‌ها استفاده می‌کند و کارایی زمانی آن $O(|V|^3)$ می‌باشد. البته الگوریتم‌های متفاوتی برای ضرب ماتریس‌ها وجود دارند مانند الگوریتم *Coppersmith-Winograd* و یا الگوریتم‌های تعبیه شده در زبان‌های برنامه‌نویسی مثل *matmul* در ماژول *numpy* پایتون، که هرکدام دارای کارایی زمانی متفاوتی هستند.

Algorithm: G^2AdjMatrix(*G*)

// returns graph G^2 for graph *G* as a matrix

// Input: graph *G*

// Output: graph G^2

// attributes of *Matrix* class:

Matrix.power(*n*) → multiplies the matrix ‘n’ times by itself

result ← *G* + *G*.matrix.power(2)

return *result*

9 – الف) برای به دست آوردن مقدار G^2 از گراف G با گره‌های V و یال‌های E، در ابتدای کار تمام یال‌های G را به لیست پیوندی خالی اضافه کرده و سپس یال بین گره‌هایی که با مسیری شامل یک گره‌ی واسطه به گره‌ی اول متصل می‌شوند را اضافه می‌کنیم، یعنی اگر گره اول به گره دوم و گره دوم به گره سوم وصل باشد، با یک یال گره‌ی اول را به گره‌ی سوم متصل می‌کنیم. در این عملیات امکان دارد که یال‌هایی تکراری شکل بگیرند، پس در آخر با دو حلقه یال‌های تکراری گراف حاصل را حذف می‌کنیم.

Algorithm: G^2AdjList(G)

```
// returns graph G^2 for graph G which is an adjacency list
// Input: graph G
// Output: graph G^2
// attributes of Graph class:

    Graph.V → list of graph's vertices
    Graph.E → list of graph's edges
    Graph.adj[v] → list of vertices u in graph connected to v with (v, u) edge
    Graph.V.add() and Graph.E.add() → add element to attributes

G^2 ← graph()   // as adjacency lists

for u in G.V do

    for v in G.adj[u] do

        G^2.E.add((v, u))

        for w in G.adj[v] do

            G^2.E.add((u, w))

// remove duplicate edges in graph

for edge in G^2.E do

    while G^2.E.count(edge) > 1 do   // count and remove are methods of list

        G^2.E.remove(edge)

return G^2
```

از آنجا که برای تشـــکیل یال‌های (u,v) کارایی زمانی O(EV) بوده و در حذف تکرارها با وجودِ داشـــتن دو حلقه، کارایی زمانی O(E) اســـت (زیرا در بدترین حالت همه‌ی اعضـــای لیســـت بررســـی می‌شـــوند و هیچ عضـــوی دوبار بررسی نمی‌شود مگر اینکع قبلا در حلقه حذف شده باشد، که در آن صورت دفعه‌ی دوم دیگر تکرار نمی‌شود). از آنجا که این دو بخش الگوریتم مستقل هستند، باید بیشرین مقدار را در نظر بگیریم. پس کارایی زمانی کلی برابر با O(EV) می‌باشد.

10. این الگوریتم مرتب‌ســـازی معروف را (که بعداً آن را در کتاب مطالعه خواهیم کرد) در نظر بگیرید. یک شمارنده در الگوریتم، برای شمارش تعداد مقایسه‌های آن، درج شده است:

```
ALGORITHM  SortAnalysis(A[0..n − 1])
// Input: An array A[0..n − 1] of n orderable elements
// Output: The total number of key comparisons made
count ← 0
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        count ← count + 1
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
return count
```

الف) آیا شمارنده مقایسه‌ها، در مکان درست درج شده است؟ اگر فکر می‌کنید بله، این را اثبات کنید؛ اگر فکر می‌کنید خیر، شمارنده را در مکان درست درج کنید.

ب) برنامه‌ای را برای پیاده‌سازی الگوریتم بنویسید. برنامه را (البته، مشروط بر آنکه شمارنده یا شمارنده‌های مقایسه‌های الگوریتمT در مکان درست درج شده باشند) روی 20 آرایه‌ی کاملاً تصادفی با اندازه‌های $1000, 2000, 3000, \dots, 20000$ اجرا کنید.

پ) داده‌های به دست آمده از اجراهای برنامه را تحلیل کنید و فرضیه‌ای درباره کارایی میانگین حالت الگوریتم، طرح کنید.

ت) تعداد مقایسه‌هایی را که باید از الگوریتم، برای مرتب‌سازی آرایه‌ای با اندازه 25000 (که به طور تصادفی تولید شده باشد) انتظار داشته باشیم، تخمین بزنید.

ث) قسمت‌های (ب) و (پ) و (ت) را با معیار قرار دادن «زمان اجرا» به عنوان شاخص کارایی الگوریتم و اندازه‌گیری زمان اجرای برنامه بر حسب میلی‌ثانیه تکرار کنید؛ یعنی برنامه را به گونه‌ای بنویسید که با آن کارایی الگوریتم بر مبنای زمان اجراهای آن، تخمین زده شود نه بر مبنای تعداد مقایسه‌های آن.

جواب:

10 – الف) خیر، این شمارنده در جای درست درج نشده است زیرا عمل مقایسه در خط *while* انجام می‌شود و در صورت عدم برقراری شرط، حلقه‌ی *while* اجرا نشده و به شمارنده مقداری اضافه نمی‌شود در حالی که مقایسه انجام شده بوده است. این شمارنده در اصل تعداد جابه‌جایی‌های عناصر آرایه را اندازه گیری می‌کند که

در بهترین حالت ممکن برای الگوریتم (یعنی برای زمانی که آرایه مرتب باشد) حلقه‌ی *while* اجرا نشده و هیچ جابه‌جایی‌ای صورت نمی‌گیرد. برای شمردن تعداد مقایسه‌ها در این الگوریتم باید خارج از حلقه‌ی *while* نیز به شمارنده اضافه کنیم تا در صورت برقرار نبودن شرط مقایسه برای جابه‌جایی، انجام آن همچنان شمرده شود.

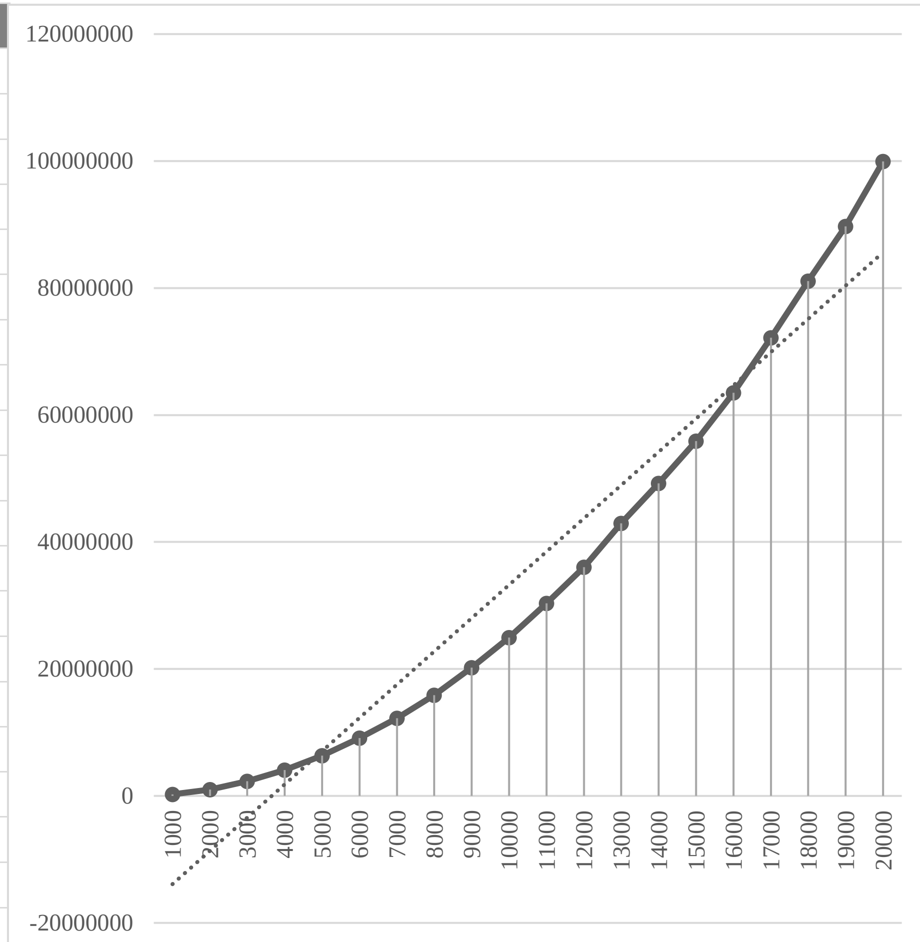
```
Algorithm: SortAnalysis(A[0…n – 1])
    // counts number of comparisons made in algorithm
    // Input: array A[0, …, n – 1] of n orderable elements
    // Output: returns the total number of key comparisons made
    count ← 0
    for i ← 1 to n – 1 do
        v ← A[i]
        j ← i – 1
        count ← count + 1
        while j ≥ 0 and A[ j] > v do
            A[j + 1] ← A[ j]
            j ← j – 1
            count ← count + 1
        A[j + 1] ← v
    return count
```

10 – ب) به منظور سنجش تعداد تکرار مقایسه‌ها در این الگوریتم برای 20 آرایه تصادفی با اندازه‌هایی از 1000 تا 20000، نیاز است علاوه بر نوشتن برنامه‌ای که الگوریتم را پیاده‌سازی کند، 20 آرایه‌ی تصادفی با طول‌های مشخص نیز به وجود آوریم تا تعداد مقایسه‌های الگوریتم را در مرتب کردن آن‌ها بسنجیم. برای حل این مسئله از زبان برنامه نویسی پایتون استفاده شده است که کد آن در پیوست 2 قرار داده شده است.

در ابتدا می‌توانیم به روش‌های مختلف آرایه‌ی تصادفی مورد نظر را بسازیم. از آنجا که طراحی الگوریتم رندوم مقصود ای مسئله نیست، از ماژول *random* پایتون استفاده می‌کنیم که سرعت بالاتری داشته و به کمک *random.sample()* آن می‌توانیم لیستی تصادفی با طول دلخواه، تشکیل دهیم. سپس الگوریتم را به روی آن

اجرا کرده و مقادیر حاصل را در یک فایل CSV ذخیره می‌کنیم. مقادیر حاصل عبارتند از:

Length	Comparisons
1000	247806
2000	987161
3000	2307416
4000	4084564
5000	6324102
6000	9134746
7000	12263414
8000	15871425
9000	20221245
10000	24945372
11000	30334547
12000	36014907
13000	42936024
14000	49257013
15000	55892224
16000	63511613
17000	72169709
18000	81076226
19000	89691235
20000	99948293



10 – پ) برای تحلیل داده‌های حاصل، رسم یک نمودار می‌تواند نتیجه‌گیری درمورد کارایی زمانی میانگین را آسان‌تر کند. پس با استخراج یک نمودار از فایل CSV حاصل (که در بالا نمایش داده شده)، مشاهده می‌کنیم که شیب نمودار شکلی شبیه به شیب نمودارهای $\log n$ و $n\log n$ و n^2 و n^3 دارد، اما با مقایسه‌ی مقادیر درمیابیم که روند نمودار حاصل از مرتبه رشد $\log n$ و $n\log n$ بسیار بیشتر و از n^3 بسیار کمتر است؛ پس نزدیک‌ترین نمودار n^2 می‌باشد. بر اساس این تحلیل، مرتبه رشد الگوریتم مرتب‌سازی داده شده $O(n^2)$ می‌باشد.

10 – ت) در قسمت پیشین نتیجه گرفتیم که مرتبه رشد این الگوریتم $O(n^2)$ است پس برای $n = 25000$ تعداد مقایسه‌ها نمی‌تواند بیشتر از 25000^2 باشد. به منظور تخمین مقدار مقایسه میانگین برای آرایه‌ای با اندازه‌ی 25000 بنابر اصول تخمین مقدار توان و داده‌های به دست آمده داریم:

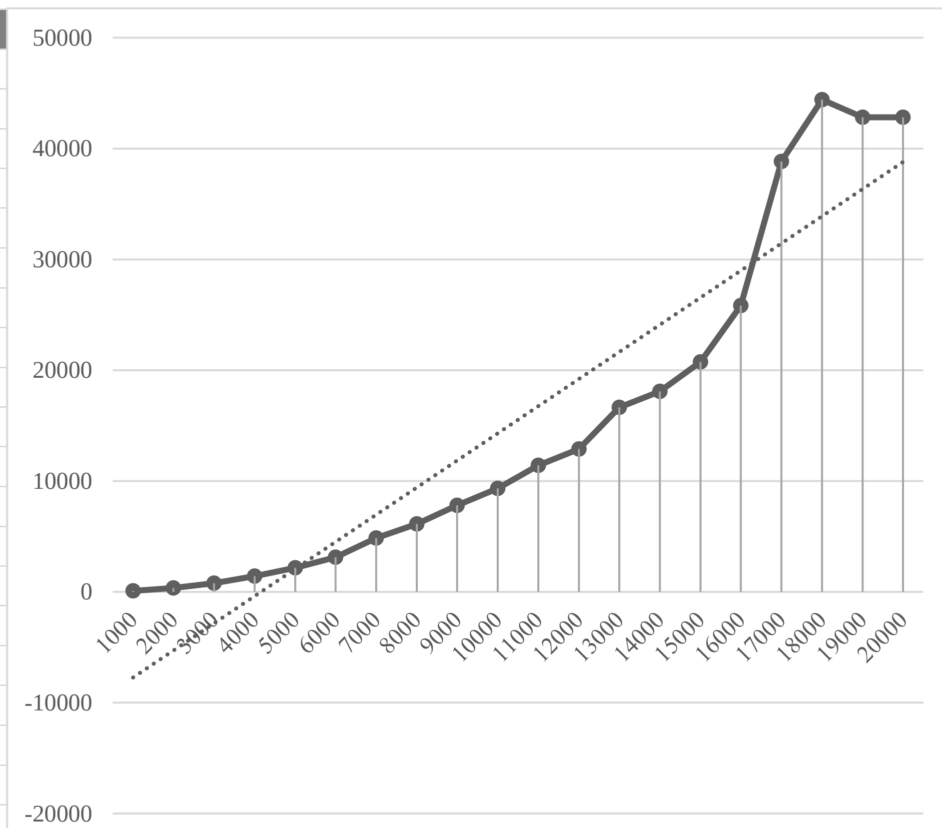
$$x^n \to y \Rightarrow \left(x + \frac{1}{m}x \right)^n \to \frac{1}{\sqrt[n]{p}} y \Rightarrow 20000^2 \to 100000000$$

$$\Rightarrow \left(20000 + \frac{20000}{4} \right)^2 \to 100000000 + \frac{1}{\sqrt[2]{4}} 100000000$$

$$\Rightarrow 25000 \to 150000000$$

10 – ث) با کمک ماژول *timeit* پایتون می‌توانیم زمان یکبار اجرای الگوریتم برای هر کدام از 20 آرایه‌ی تصادفی را برحسب میلی‌ثانیه به دست آوریم. اگر قصد بر تکرار اجرای الگوریتم بر آرایه بود، به علت مرتب شدن آرایه بعد از یکبار اجرای الگوریتم بایستی حلقه‌ای قرار می‌دادیم که برای هر طول آرایه، چند بار آرایه‌ی تصادفی تشکیل دهد و سپس زمان اجرای الگوریتم را درون حلقه برای یکبار اجرا بر آن محاسبه می‌کردیم و یک تقسیم میانگین می‌گرفتیم. به علت سادگی الگوریتم و بزرگ بودن مقدار پارامتر، نیازی به این کار دیده نشد و زمان اجرای الگوریتم برای یکبار بر هر طول آرایه به دست آمد (کد برنامه در پیوست 3 قابل مشاهده است). بنابر نتایج یافت شده (که در زیر نمایش داده شده‌اند)، مشاهده می‌کنیم که شیب نمودارد به $n\log n$ نزدیک است پس برای $n = 25000$ که لگاریتم آن 14.6 است، زمان تقریبی 365000 میلی‌ثانیه می‌باشد.

Length	Time (ms)
1000	84
2000	358
3000	778
4000	1410
5000	2157
6000	3127
7000	4857
8000	6137
9000	7811
10000	9341
11000	11422
12000	12890
13000	16656
14000	18077
15000	20741
16000	25840
17000	38839
18000	44411
19000	42830
20000	42814



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). It used 1.8 GB RAM space.

پیوست 1

---

question 7 code

```
def MaxRow(matrix):

    index = 0
    row = None
    n = len(matrix)
    for i in range(n):
        while index != n and matrix[i][index] == 1:
            index += 1
            row = i
    return row

def MaxRowAnalysis(matrix):

    index = 0
    row = None
    n = len(matrix)
    count = 0
    for i in range(n):
        while index != n and matrix[i][index] == 1:
            index += 1
            row = i
            count += 1
    return count
```

پیوست 2

```
import csv
import random

def SortAnalysis(array):
    # to count number of comparisons
    # made by insertion sort algorithm
    count = 0
    for i in range(1, len(array)):
        key = array[i]
        j = i – 1
        count += 1
        while j >= 0 and key < array[j]:
            array[j + 1] = array[j]
            j -= 1
            count += 1
        array[j + 1] = key
    return count

lengths = list()  # list of intended 20 array lengths
for i in range(1, 21):
    lengths.append(i * 1000)

# main code for running SortAnalysis for 20 arrays
csvlines = [['Length', 'Comparisons']]

for l in lengths:
    # creating a shuffled orderable list
    # with elements as numbers from 0 to (length - 1)
    array = random.sample(range(l), l)
    count = SortAnalysis(array)
    csvlines.append([l, count])

with open('sortcount.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerows(csvlines)
```

پیوست 3

```
import csv
import random

def InsertionSort(array):
	for i in range(1, len(array)):
		key = array[i]
		j = i – 1
		while j >= 0 and key < array[j]:
			array[j + 1] = array[j]
			j -= 1
		array[j + 1] = key

# list of intended 20 array lengths
lengths = list()
for i in range(1, 21):
	lengths.append(i * 1000)

csvlines = [['Length', 'Time (ms)']]
import timeit
code = '''
from __main__ import InsertionSort
from __main__ import array'''

for l in lengths:
	# creating a shuffled orderable list
	# with elements as numbers from 0 to (length - 1)
	array = random.sample(range(l), l)
	time = timeit.timeit(setup = code,
				stmt = 'InsertionSort(array),'
				number = 1)
	# finding time based on milliseconds
	time = round(time, 3) * 1000
	csvlines.append([l, time])

with open('sorttime.csv', 'w', newline='') as csvfile:
	writer = csv.writer(csvfile)
	writer.writerows(csvlines)
```