دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

درس تحلیل و طراحی الگوریتم‌ها

راه‌حل تکالیف:

سری اول - تحلیل زمانی

نام استاد درس:

دکتر بهروز شاهقلی

نام دستیاران آموزشی درس:

رضا رستگاری

میلاد محمدی

زهرا تاکی

سارا کهتری

جواد جعفری

امیرحسین عرب‌پور

نیم‌سال دوم تحصیلی ۱۴۰۰-۱۴۰۱

فهرست

تمرین اول: پیچ سورت

$$n^{\frac{1}{\log n}} < (\log n)^2 < \sqrt{n} < 4^{\log n} < \log(n!) < n * \log(n) < n * Ln(n) < n^2$$

$$5n^2 + 7n < \log(n)! < n^{\log(\log(n))} < n * 2^n < n^{\frac{5}{2}} < n^3 < \frac{3^n}{2} < 4^n <$$

$$\text{n}! < n^n < n^n + \ln(n) < 2^{2^n} < 2^{n!}$$

با توجه به قانون حد تقسیم دو تابع، روابط زیر به دست آمده است. اگر در پاسخ شما یکسری اختلاف جزئی با جواب وجود دارد، نمره ای کم نمیشود. استدلال و بررسی توابع توسط شما هدف اصلی این سوال بوده است.

تمرین دوم: پیچ سالو

الف)

طبق کد داده شده، while داخل به تعداد ثابت تکرار می‌شود. و حلقه بیرونی n بار تکرار دارد، پس در مجموع kn، بار این کد تکرار می‌شود.

ب)

باتوجه به forهای تو در تویی که داریم، به این شکل می‌توانیم بنویسیم.

$$\sum_{k=1}^{n-1} \frac{k\,n!}{k!\,(n-k)!} = \frac{1}{2}\,(2^n - 2)\,n$$

ج)

در الگوریتم غربال اراتستن، برای پیدا کردن اعداد اول کمتر مساوی n ،اعداد مرکب را در آرایه prime عالمت false میزنیم. برای این کار، باید مضارب عدد اول فعلی که بزرگر مساوی مربع عدد اول فعلی هستند را false کنیم. با عدد ۲ شروع میکنیم و از مربع عدد ۲، یعنی ۴ مضارب ۲ را false می‌کنیم. در این مرحله n/2 عدد false میشوند. سپس همین کار را برای عدد اول بعدی یعنی ۳ می‌دهیم:

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \cdots = n\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \cdots\right) = n\log(\log(n))$$

تمرین سوم: پیچ‌مث

$n\sqrt{n} \in O(n^n)$

اطور که می‌دانیم $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$ (صفر) $\longrightarrow$ $g(n) \in \underset{\text{small o}}{o}(f_n)$

همانطور که مدام طبق تعریف و نمودار دیدیم

small O = big O — theta

یعنی در big O باید کوچکتر یا مساوی باشه اما در small o باید اکیداً یک تابع از دیگری در بی‌نهایت کوچکتر شود.

طبق این مقایسه می‌توان نتیجه گرفت:

$g(n) \in \underset{\text{small}}{o}(f_n) \longrightarrow g(n) \in \underset{\text{big}}{O}(f_n)$ *

$g(n) = n\sqrt{n}$, $f(n) = n^n$ $\rightarrow$ $\lim_{n\to\infty} \frac{n\sqrt{n}}{n^n} = 0 \longrightarrow n\sqrt{n} \in \underset{\text{small}}{o}(n^n) \xrightarrow{*} n\sqrt{n} \in \underset{\text{big}}{O}(n^n)$

2.3) $\sum_{i=1}^{n}\sum_{j=1}^{i} j \in \Theta(n^3)$

لازم به ذکر است:

از پیوست اول کتاب روابط زیر را می‌دانیم: (با استقرا قابل اثبات اند)

1. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

2. $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

حال داریم:

$$\sum_{i=1}^{n}\sum_{j=1}^{i} j \xRightarrow{\text{طبق } 1} \sum_{i=1}^{n} \frac{i(i+1)}{2} \Rightarrow \frac{1}{2}\sum_{i=1}^{n} i^2+i$$

$$\Rightarrow \frac{1}{2}\left(\sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i\right) \xRightarrow{\text{طبق } 2,1} \frac{1}{2}\left(\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2}\right)$$

$$\Rightarrow \frac{1}{2}\, n(n+1)\left(\frac{2n+1}{6} + \frac{1}{2}\right) \Rightarrow \frac{n(n+1)(n+2)}{6}$$

حال باید ثابت کنیم n_0 وجود دارد که برای $n \geq n_0$، ثابت‌های c_1 و c_2 وجود داشته باشند:

$$c_1 n^3 \leq \frac{n(n+1)(n+2)}{6} \leq c_2 n^3$$

(I) نامساوی سمت چپ برای $n \geq 1$ و $c_1 = \frac{1}{6}$ برقرار است.

$$\frac{1}{6} n^3 \leq \frac{1}{6}(n^3+3n^2+2n)$$

(II) نامساوی برای $n \geq 1$ و $c_2 = 6$ برقرار است.

$$\frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n \leq 6n^3$$

بنابراین n_0 و c_1 و c_2 وجود دارند و $\sum_{i=1}^{n}\sum_{j=1}^{i} j \in \Theta(n^3)$

تمرین چهارم: سینیو یا جونیور، مسئله این است

برای حل این مسئله دو قدم اصلی را باید برداریم. اول باید هزینه‌های ثابت برای یک‌بار برنامه‌نویسی و کامپایل برنامه (هزینه‌های پیاده‌سازی) بر روی سرور به ازای هرکدام را بدست آوریم. سپس معادله k بار اجرای هرالگوریتم بر روی سرور با احتساب پیچیدگی زمانی داده شده را بنویسیم.

الگوریتم دوم Senior	الگوریتم اول Junior	
60	20	حقوق برنامه‌نویسی بر ساعت
5	4	میزان ساعت مورد نیاز برنامه‌نویسی
5 * 60 = 300	**4 * 20 = 80**	**هزینه برنامه‌نویسی**
(6/60)	(2/60)	زمان کامپایل (ساعت)
50		هزینه سرور
50 * (6/60) = 5	**50 * (2/60) = 1.67**	**هزینه کامپایل**
300 + 5 = 305	**80 + 1.67 = 81.67**	**هزینه نهایی پیاده‌سازی**

در نتیجه می‌توانیم بگوییم هزینه پیاده‌سازی الگوریتم دوم 223.33 = 81.67 – 305 بیشتر از الگوریتم اول است.

برای سادگی نوشتن معادلات، هزینه اجرای برنامه بر روی سرور بر حسب میکروثانیه را c در نظر می‌گیریم. در نتیجه معادله زمانی که هزینه پیاده‌سازی و اجرای دو الگوریتم به ازای k و n برابر باشد به شکل زیر است:

$$k * (n^2) * c = k * (100 * n)*c+ 223.33$$

الف)

مقدار n برابر با 50 است. پس:

$$k * (50^2) * c = k * (100 * 50)*c + 223.33$$

$$k * 2500 * c = k * 5000 *c + 223.33$$

واضح است که به ازای هیچ مقدار مثبت k این معادله جواب ندارد. پس در شرایط قسمت الف، هیچ وقت الگوریتم دوم از لحاظ هزینه به صرفه‌تر از الگوریتم دوم نیست!

ب)

مقدار n برابر با 500 است. پس:

$$k * (500^2) * c = k * (100 * 500)*c + 223.33$$

$$k * 250000 * c = k * 50000 *c + 223.33$$

$$20000 * c * k = 223.33$$

که با بدست آوردن مقدار c بر حسب هزینه سرور داده شده و حل معادله ساده شده خواهیم داشت که تقریبا به ازای 80500 بار به بالای برنامه، الگوریتم دوم بهینه‌تر خواهد بود.

تمرین پنجم: تفکر الگوریتمی تخم‌مرغی

شما ابتدا بایستی در این سوال الگوریتم‌های مورد استفاده را تحلیل زمانی کرده، و تابع پیچیدگی زمانی آنها را بدست می‌آوردید. تابع پیچیدگی زمانی برخی از الگوریتم‌های مرتب‌سازی و جستجو در جداول زیر آورده شده است.

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

نکته: زمانی که از تابع O استفاده می‌کنیم و Log داریم، در جهت سادگی محاسبات و باتوجه به تاثیر ثابت پایه‌ی Log، ما پایه آن را ۱۰ در نظر می‌گیریم. هرچند اگر پایه لگاریتم را ۲ در نظر گرفته باشید از شما غلط گرفته نمی‌شود.

الف)

حالت Worst Case :

در حالت Worst Case اگر از جستجوی خطی استفاده کنیم:

O(n) ➔ 100

اگر از جستجوی دودویی استفاده کنیم، باید ابتدا عناصر آرایه را مرتب کنیم. هر دو الگوریتم مرتب‌سازی Quick Sort و Insertion Sor، در حالت Worst Case پیچیدگی زمانی O(n^2) دارند. و جستجوی باینری هم پیچیدگی زمانی O(log n) پس در مجموع می‌توان گفت:

O(n^2) + O(log n) ➔ 10002

حالت Average Case:

در حالت Average Case جستجوی خطی پیچیدگی O(n) دارد. که مانند قسمت الف می‌شود.

حالت میانگین جستجوی دودویی در Quick Sort برابر O(n*log n) است که بهینه‌تر از Insertion Sort است. پس در مجموع:

O(n*log n) + O (log n) ➔ 202

بنابراین استفاده از الگوریتم جستجوی خطی بهینه‌تر است.

ب)

حالت Worst Case :

در هر بار جستجو، پیچیدگی زمانی O(n) دارد.

k * O(n) ➔ k * n ➔ 100k

در جستجوی دودویی، تنها یک‌بار مرتبه مرتب‌سازی نیاز است که انجام دهیم که تابع پیچیدگی زمانی آن O(n^2) خواهد بود. و سپس برای هربار جستجو پیچیدگی زمانی O(log n)

100k = k * log(100) + (100^2) ➔ 100k = 2k + 10000

که به طور تقریبی می‌توان گفت اگر بیش‌از ۱۰۰ جستجو داشته باشیم، استفاده از شیوه‌دوم بهینه‌تر است.

حالت Average Case:

100k = k * log (100) + 100 * log (100) ➔ 100k = 2k + 200

به طور تقریبی می‌توان گفت بیش از ۲بار اجرای شیوه دوم، این شیوه بهینه‌تر از شیوه اول خواهد بود.

ج)

دقیقا به مانند قسمت الف سوال خواهد بود. با تفاوت در n. اما چون تابع پیچیدگی زمانی در جستجوی خطی برابر O(n) و در جستجوی باینری در حالت Worst برابر O(log n)+O(n^2) و در حالت میانگین به صورت O(n*logn) + O(log n) است، جستجوی خطی بهینه‌تر است.

د)

حالت Worst Case :

همان معادله‌های قسمت ب را باید برای n جدید حل کنیم.

10000k = k * log(10000) + 10000^2 ➔ 10000k = 4k + 100000000

به طور تقریبی می‌توان گفت اگر بیش‌از ۱۰۰۰۰ بار جستجو نیاز داشته باشیم شیوه دوم بهینه‌تر است.

حالت Average Case:

10000k = k * log(10000) + 10000 * log (10000) ➔ 10000k = 4k + 40000

یعنی به طور تقریبی اگر بیشتر از ۴ جستجو نیاز داشته باشیم، الگوریتم دوم بهینه است.

ه)

الگوریتم‌ها علاوه بر time complexity نیازمند تحلیل space complexity هستند. اجرای الگوریتم‌ها به جز داده‌های ورودی الگوریتم، ممکن است نیازمند متغییرهای temporary باشد یا نیازمند حافظه stack در الگوریتم‌های بازگشتی باشد. برای تحلیل حافظه الگوریتم باید به این موارد در کد اجرا الگوریتم توجه کرد. این مبحث کمتر از مبحث time complexity به طور عمومی اهمیت دارد، چون هم‌اکنون با پیشرفت تکنولوژی کامپیوترهای ما مشکل حافظه ذخیره‌سازی برای اجرا کدها ندارند. اما در استفاده‌های خاص و در داده‌های حجیم، این مبحث همانند مبحث time complexity اهمیت بالایی دارد. مخصوصا بعد از گذراندن درس معماری کامپیوتر متوجه خواهید شد که هرچقدر نیاز خودتان را به حافظه کمتر کنید، محاسبات سریع‌تری خواهید داشت. در جدول ابتدای فایل space complexity الگوریتم‌ها آورده شده‌است. باتوجه به آن اطلاعات به تحلیل خواهیم پرداخت.

حالت Worst Case :

تمام الگوریتم‌های مرتب‌سازی و جستجویی که در این سوال آورده شده، پیچیدگی فضایی برابر O(1) دارند به جز الگوریتم Quick Sort که پیچیدگی فضایی برابر O(log n) دارد. فرض می‌کنیم داده‌های داخل آرایه تراکنش‌ما از نوع long int باشد که هر عنصر از آرایه ما 8 بایت فضا اشغال می‌کند. پس می‌توانیم معادله فضای اشغال شده را به این شکل بنویسیم:

(original data + auxiliary space needed)

= (n * 8) + ((log n) * 8) = (10000 * 8) + ((log 10000) * 8) = 80000 + 32

می‌توان گفت برای ذخیره‌سازی خود داده‌ها نیازمند ۸۰کیلوبایت فضا هستیم و برای اجرای الگوریتم نیازمند ۳۲ بایت فضای کمکی. که مقدار بسیار کمی است.

و)

در جستجوی دودویی در هر مرحله از الگوریتم، عنصر وسط آن سگمنت از آرایه را مد نظر قرار می‌دهد. اگر داده هدف از این عنصر وسط سگمنت بزرگتر باشد، روبه جلو حرکت کرده و سراغ عنصر وسط سگمنت بعدی خواهد رفت. اما اگر داده هدف، از عنصر وسط کوچکتر باشد، باید عنصر وسط سگمنت پایینی (قبلی) را جستجو کند. که نیازمند بازگشت رو به عقب بر روی داده هاست. و باتوجه به شرایط مسئله و نوع عملکر دیسک سخت، ممکن است باعث بهینه‌نبودن زمان اجرا الگوریتم ما شود.

در این شرایط می‌توانیم از الگوریتم Jump Search استفاده کنیم. این الگوریتم با سگمنت کردن داده‌های آرایه به m بخش و جستجو به شکل روبه جلو بر روی آنها، باعث می‌شود حداکثر یک‌بار بازگشت روبه عقب بر روی داده‌ها داشته باشیم. با اینکه پیچیدگی زمانی این الگوریتم بیشتر از الگوریتم باینری سرچ در حالت عمومی است، اما در این شرایط (یا هر شرایط مشابه)، باعث بهینگی سرعت اجرا الگوریتم می‌شود.

ز)

اضافه شدن داده‌ها در حالت جستجو خطی، تفاوتی ایجاد نمی‌کند. زیرا در هر صورت این الگوریتم نیازمند به مرتب بودن داده‌ها ندارد. پس در هر شرایط زمان اجرا این الگوریتم O(n) خواهد بود.

اما در جستجوی دودویی و یا جستجوی پرشی، نیازمند مرتب بودن داده‌ها هستیم. و اضافه شدن یک داده جدید به عناصر داده‌ها به انتهای آن، می‌تواند مرتب بودن داده‌ها را بهم بریزد. در این حالت شاید اولین راهکار مرتب‌کردن کل داده‌ها پس از اضافه شدن هر داده باشد. که طبق بررسی که در قسمت الف این سوال انجام دادیم هیچ وقت این شیوه بهینه نخواهد بود.

اما با استفاده از همان شیوه‌های جستجو، مثل جستجوی باینری، در زمان اضافه شدن داده جدید به آرایه مرتب، می‌توانیم جای مناسب داده را پیدا کرده و داده را در آن index قرار دهیم. در این صورت پس از اضافه شدن هر داده جدید یک O(log n) پیچیدگی زمانی خواهیم داشت که برابر پیچیدگی زمانی هر جستجو در این حالت است. به این شیوه اگر حجم داده‌ها متوسط و بالا باشد، ما یکبار مرتب‌سازی اولیه لازم داریم و سپس هر عملیات بر روی داده‌ها O(log n) زمان از ما خواهد گرفت. که در اکثر مواقع (به جز موارد خاص که اضافه شدن، حذف شدن و تغییر داده‌ها بسیار بیشتر از تعداد داده‌های اولیه و یا میزان جستجو باشد) بهینه‌تر از جستجو خطی خواهد بود.

تمرین ششم: تخم‌مرغ‌های فولادی

بدیهی است که استفاده از الگوریتم جستجویی دودویی می‌تواند باعث شکستن تعدادی تخم مرغ (حداکثر log n) شود. زیرا در جستجوی دودویی بجای آنکه به ترتیب پیش‌رویم، از وسط سگمنت شروع می‌کنیم. اگر حد تحمل تخم‌مرغ بیشتر باشد و نشکند که به سراغ وسط سگمنت نیمه بالایی می‌رویم. اما اگر از حد تحمل تخم‌مرغ بیشتر باشد، باید تخم‌مرغ جدید گذاشته شود و به وسط سگمنت نیمه پایینی برویم.

در این سوال و در این شرایط باید توجه داشته باشید که باتوجه به مکانیکی بودن بازوی روبات‌های تست ارتفاع، هزینه زمانی بالایی داریم. از طرفی ممکن است هزینه زمانی حرکت رو به بالای روبات، و هزینه زمانی حرکت روبه پایین روبات متفاوت باشد.

الف)

چون فقط یک تخم‌مرغ داریم، تنها الگوریتمی که می‌توانیم مطمئن باشیم به جواب خواهیم رسید، جستجوی خطی خواهد بود. به ترتیب پله‌ها را حرکت می‌کنیم و هر پله‌ای که تخم‌مرغ بر روی آن شکست، پله قبلی آخرین حد تحمل تخم‌مرغ و جواب مورد نیاز ما خواهد بود.

ب)

اگر بیایم و ابتدا تخم‌مرغ را از پله وسطی نردبان ارتفاع امتحان کنیم، در صورتی که با شکستن تخم‌مرغ مواجه شویم خواهیم فهمید حد آستانه تحمل تخم‌مرغ در نیمه پایینی خواهد بود. و اگر نشکست، می‌دانیم جواب در نیمه بالایی خواهد بود. در صورت نشکستن تخم‌مرغ بازهم می‌توانیم تخم‌مرغ را از نردبان وسط نیمه به دست (سگمنت) بیاندازیم. هر زمان که با شکستن تخم‌مرغ مواجه شدیم، با توجه به باقی ماندن فقط یک تخم مرغ برای جستجو، از شیوه قسمت الف استفاده کنیم. البته فقط نیاز است تا همان سگمنتی که می‌دانیم جواب در آن هست را جستجوی خطی کنیم.

ج)

اگر مقدار k از log n بیشتر باشد، با خیال راحت می‌توان الگوریتم جستجوی دودویی را به شکل کامل اجرا کرد و به جواب رسید. اگر مقدار k از log n کمتر باشد، می‌توانیم تا k-1 بار شکستن تخم‌مرغ، الگوریتم را به شیوه جستجو دودویی انجام دهیم و سپس مرحله آخر باتوجه به آنکه می‌دانیم جواب در کدام سگمنت است، بر روی آن سگمنت جستجوی خطی انجام دهیم.

د)

یک راه دیگر می‌تواند استفاده از Jump Search باشد. در این حالت و الگوریتم، سگمنت کردن نردبان شبیه به باینری سرچ خواهد بود، اما به طور کلی جستجو خطی خواهیم داشت. و در این حالت هم ما می‌توانیم حداکثر با دوبار شکستن تخم‌مرغ به جواب برسیم. یک تخم مرغ زمانی می‌شکند که حد آستانه سگمنت را پیدا کنیم. و یک تخم در زمان جستجو خطی بر روی آن سگمنت.

اگر روبات‌های ما قابلیت انداختن چندین تخم‌مرغ همزمان را داشته باشند، و فرض کنیم m تخم مرغ را می‌توانیم همزمان بیاندازیم (و ریسک و بودجه مالی شکستن m تخم‌مرغ را داشته باشیم)، می‌توانیم نردبان را به m سگمنت جدا کنیم. و سپس تخم‌مرغ‌ها از حد بالای هر سگمنت انداخته شوند. در این حالت با یک‌حرکت همزمان روبات سگمنت جواب را پیدا می‌کنیم و با جستجوی خطی بر روی آن سگمنت می‌توانیم به جواب برسیم.

تمرین هفتم: تابع بد

توضیح:

برای امتیاز ۵۰ تنها کافیست که خود تابع گفته شده را با یک زبان برنامه‌نویسی پیاده‌سازی کنید.

برای امتیاز ۱۰۰، نمی‌توان از پیاده سازی تابع استفاده کرد و جواب دادن مثل حالت ۵۰ امتیازی، باعث Time limit exceeded می‌شود.

در این حالت اگر کمی به تابع دقت کنیم، خواهیم دید که می توان به جای جفت عدد زوج و فرد کنار هم، عدد یک را در نظر گرفت، بنابر این با توجه به زوج یا فرد بودن n، میتوان سوال را به این شکل حل کرد:

کد:

```python
n = int(input())
if(n%2):
    print((n-1)//2-n)
else:
    print(n//2)
```

تمرین هشتم: آب سیب

توضیح:

عدد اولیه شامل بیش از ۱۰۰۰۰۰ رقم نیست. بنابراین پس از تبدیل اول، عدد حاصل بیش از ۹۰۰۰۰۰ نخواهد بود و سپس از ۶ رقم بیشتر تشکیل نمی‌شود. بنابراین بعد از تبدیل بعدی عدد بیشتر از ۵۴ نخواهد بود و بنابراین دو رقمی یا یک رقمی خواهد بود. بنابرین شبیه سازی عملیات وقت چندانی نمی‌گیرد.

کد:

```python
n = input()
ans =0
d=0
while len(n)>1:
   for i in range(len(n)):
      d+= (int(n[i]))
   n = str(d)
   d=0
   ans = ans +1
print(ans)
```

تمرین نهم: آنتن بد

توضیح:

ممکن است در ابتدا تلاش کرده باشید معکوس عملیات ذکر شده در سوال را انجام دهید، و یا با راه‌های دیگر و بررسی ویژگی اعداد ورودی و خروجی به جواب رسیده باشید. همگی کار های خوبی هستند، ولی نکته‌ای که در این سوال وجود دارد این است که، گاهی با توجه به تعداد ورودی، شاید بدیهی‌ترین راه حل هم به اندازه کافی خوب باشد و نیازی به بررسی راه‌های پیچیده‌تر نباشد!

برای پیدا کردن مقدار اولیه x در این سوال تنها کافیست همه اعداد بازه 0 تا 255 را بررسی کنیم و عملیات شرح داده شده در صورت سوال، یعنی x^(x<<1) را روی همه اعداد بازه اجرا کنیم و چنانچه عددی یافتیم که با ورودی یکی است، آن را به عنوان جواب خروجی دهیم . در این راه حل حداکثر به 256n عملیات نیاز داریم که برای محدودیت زمانی گفته شده با توجه به اندازه n کافیست.

کد:

```cpp
#include <bits/stdc++.h>
using namespace std;

int val (int a) {
   for (int i = 0; i <= 255; ++i) {
      if (((i ^ (i << 1)) % 256) == a)
         return i;
   }
   return 0;
}

int main() {
   int n;
   cin >> n;

   for (int i = 0; i < n; ++i) {
      int temp;
      cin >> temp;

      cout << val(temp) << ' ';
   }

   cout << endl;
}
```