Quiz 01 Answers

پاسخنامه آزمون شماره 1

پاسخ سوال یک :

الف)الگوریتم مورد نظر **Bubble Sort(مرتب‌سازی حبابی)** است.

۱ :در مرحله اول، بزرگ‌ترین عدد (19) به انتهای آرایه منتقل شد. این ویژگی بارز Bubble Sort است که در هر Pass ، بزرگ‌ترین عنصر به مکان نهایی خود در انتهای آرایه می‌رود.

۲ :در همین مرحله، عدد 0 از موقعیت سوم به اول رسید (7,1,0,6 → 0,7,1,6). این نشان‌دهنده حرکت تدریجی عناصر کوچک‌تر به سمت چپ است که از مشخصات Bubble Sort می‌باشد.

۳ :در مرحله دوم، عدد 1 به موقعیت دوم رسید و 8 جایگزین 19 شد. این نشان می‌دهد که الگوریتم به صورت تکرارشونده (Pass به Pass) عمل کرده و در هر مرحله یک عنصر مرتب دیگر به انتها اضافه می‌شود.

ب)

:1	0,7,1,4,6,19,8,13
:2	0,1,7,4,6,8,19,13
:3	0,1,4,7,6,8,13,19
:4	0,1,4,6,7,8,13,19

پ)

0,1,7,6,19,8,13,4

0,1,7,6,19,8,13,4

0,1,4,6,19,8,13,7

0,1,4,6,19,8,13,7

0,1,4,6,7,8,13,19

0,1,4,6,7,8,13,19

0,1,4,6,7,8,13,19

پاسخ سوال دو :

اگر عناصر *Distinct باشند و مقدار هر عنصر در بازهٔ [min, min + n·k] محصور باشد، می‌توانیم از Counting Sort بهینه‌شده یا Bucket Sort استفاده کنیم تا زمان بهتر از O(n log n) به دست بیاوریم.

1) ایدهٔ اصلی راه‌حل

وقتی تفاوت مقدارها محدود باشد، تعداد بازه‌های ممکن نیز محدود است.

بنابراین اگر مقدارها در محدودهٔ نسبتاً کوچکی باشند، می‌توان با ساختن تعداد محدودی سطل (bucket) یا جدول شمارشی (count array) آرایه را سریع مرتب کرد.

الگوریتم Bucket Sort با اندازهٔ کنترل‌شده

فرض مهم

چون $|A\_sorted[i] - A\_sorted[i-1]| \le k$ است، تمام n عدد در بازه‌ای با اندازهٔ حداکثر n·k قرار دارند.

پس:

$$range\_size \le n \cdot k$$

اگر از Counting Sort عادی استفاده کنیم، هزینهٔ زمان و حافظه برابر خواهد شد با:

$$O(n + n \cdot k)$$

که در صورتی که k کوچک باشد از $O(n \log n)$ بهتر است.

2) روش کامل الگوریتم

مرحله ۱: پیدا کردن حداقل مقدار

```
minVal = min(A)
```

مرحله ۲: تعیین بازهٔ سطل‌ها

```
k range = n
```

مرحله ۳: ایجاد آرایهٔ شمارش با اندازهٔ کوچک

```
count[0 ... range]
```

مرحله ۴: شمارش عناصرهر عنصر A[i] در index زیر قرار می‌گیرد:

```
index = A[i] – minVal
```

```
count[index]++
```

مرحله ۵: بازسازی آرایهٔ مرتب‌شده

```
pos = 0
```

```
for i from 0 to range:
```

```
repeat count[i] times:
```

```
A[pos] = i + minVal
```

```
pos++
```

3) اثبات درستی الگوریتم

1. چون عناصر distinct هستند، هر مقدار دقیقاً یک بار در بازه قرار می‌گیرد. 2. چون range = n·k است و اختلاف مقادیر در آرایهٔ مرتب‌شده هرگز بیش از k نیست، تمام اعداد قطعاً داخل بازهٔ minVal تا minVal+n·k قرار می‌گیرند. 3. Counting Sort ترتیب مقادیر را دقیقاً بر اساس مقدار عدد بازسازی می‌کند، پس نتیجه ۱۰۰٪ مرتب است.

4) تحلیل زمانی الگوریتم

زمان اجرا- ساخت count: O(n·k)- شمارش n عنصر: O(n)- بازسازی خروجی: O(n·k)

در مجموع:O(n + n·k)

حالت‌های خاص:- *بهترین حالت: اگر k = O(1)

زمان O(n)

میانگین: اگر k خیلی بزرگ نباشد و نزدیک به ثابت باشد

زمان خطی

بدترین حالت: اگر k ≈ n

زمان $O(n^2)$ و بهینه نیست

5) آیا می‌توان همیشه به O(n)رسید؟

فقط وقتی امکان‌پذیر است که k مقدار کوچکی باشد بازهٔ مقادیر محدود باشد .

اگر k بزرگ باشد، مجبوریم طیفی به اندازهٔ n·k بسازیم، و در این صورت پیچیدگی خطی دستیابی‌پذیر نیست. پس O(n)* فقط زمانی ممکن است که k محدود و کوچک باشد.

پاسخ سوال سه :

توابع:

1. $f(n) = n * \log_2(n)$ (رشد لگاریتمی-ضربدر-خطی)

2. $g(n) = 50n + 1000$ (رشد خطی)

3. $h(n) = n^{(1.5)}$ (رشد توانی)

1. رتبه‌بندی و 2. مقایسه جفتی (شهودی):

$f(n)$ در مقابل $g(n)$:

$g(n)$ یک تابع خطی است (با ضریب ۵۰).

$f(n)$ شامل یک بخش خطی (n) است که در لگاریتم ($\log_2(n)$) ضرب شده.

می‌دانیم که لگاریتم، حتی در پایه‌های کوچک، نسبت به رشد خطی بسیار کندتر است. اما وقتی همین تابع کند، در یک تابع خطی دیگر ضرب شود، رشد کلی آن از رشد خطی محض سریع‌تر خواهد شد.

برای مقادیر بزرگ n ، عبارت $n * \log_2(n)$ از $50n$ سریع‌تر رشد می‌کند. مثلاً اگر $n=1024$ (که $\log_2(1024) = 10$ است):

$$g(1024) = 50 * 1024 + 1000 = 51200 + 1000 = 52200$$

$$f(1024) = 1024 * \log_2(1024) = 1024 * 10 = 10240$$

بیایید دوباره چک کنیم. در مقادیر بزرگ، $\log_2(n)$ هرچند کند، اما رشد می‌کند. پس $n * \log_2(n)$ باید از n سریع‌تر باشد. بیایید $n=100$ را امتحان کنیم:

$$\log_2(100) \approx 6.64$$

$$g(100) = 50 * 100 + 1000 = 5000 + 1000 = 6000$$

$$f(100) = 100 * \log_2(100) \approx 100 * 6.64 = 664$$

باز هم اشتباه! این نشان می‌دهد که شهود اولیه در مورد کند بودن لگاریتم در برابر ضریب ۵۰، درست بوده است. عبارت n50 در ابتدا خیلی قوی‌تر از $n * \log_2(n)$ است. اما باید رفتار بلندمدت را در نظر گرفت.

نکته کلیدی: وقتی n خیلی بزرگ شود، $\log_2(n)$ هم مقداری می‌شود و ضرب آن در n، از n50 پیشی می‌گیرد. بیایید n بسیار بزرگتری را در نظر بگیریم، مثلاً $n = 2^{16} = 65536$.

$$\log_2(65536) = 16$$

$$g(65536) = 50 * 65536 + 1000 \approx 3,276,800$$

$$f(65536) = 65536 * 16 = 1,048,576$$

هنوز هم g(n) بزرگتر است! این نشان می‌دهد که ضریب ۵۰ بسیار تاثیرگذار است. اما بیایید n را باز هم بزرگتر کنیم. مثلاً $n = 2^{20} \approx 10^6$

$$\log_2(2^{20}) = 20$$

$$g(10^6) = 50 * 10^6 + 1000 \approx 50 * 10^6$$

$$f(10^6) = 10^6 * 20 = 20 * 10^6$$

بالاخره! برای n های بسیار بزرگ، f(n) از g(n) سریع‌تر رشد می‌کند. رشد خطی با ضریب ثابت، نهایتاً مغلوب رشد $n * \log(n)$ می‌شود.

h(n) در مقابل f(n) و g(n):

$n = (1.5)^{h(n)}$. این یک تابع توانی است.

توابع توانی (حتی با توان غیرصحیح مثل ۱.۵) تقریباً همیشه از توابع خطی (g(n)) و توابع (f(n)) $n * \log(n)$ سریع‌تر رشد می‌کنند.

برای n=100:

$f(100) \approx 664$

$g(100) = 6000$

$h(100) = 100^{(1.5)} = 100 * \text{sqrt}(100) = 100 * 10 = 1000$

اینجا h(n) کوچکتر از g(n) است! دوباره، این نشان‌دهنده اهمیت ضریب و ثابت در رشد اولیه است. اما باز هم باید به رفتار بلندمدت نگاه کرد.

برای n=1000:

$\log_2(1000) \approx 10$

$f(1000) = 1000 * \log_2(1000) \approx 1000 * 10 = 10000$

$g(1000) = 50 * 1000 + 1000 = 51000$

$h(1000) = 1000^{(1.5)} = 1000 * \text{sqrt}(1000) \approx 1000 * 31.6 \approx 31600$

در این مقیاس، h(n) هنوز از g(n) کندتر است، اما از f(n) سریع‌تر شده.

برای n = 65536:

$f(65536) \approx 1{,}048{,}576$

$g(65536) \approx 3{,}276{,}800$

$h(65536) = 65536^{(1.5)} = 65536 * \text{sqrt}(65536) = 65536 * 256 = 16{,}777{,}216$

اکنون h(n) از هر دو f(n) و g(n) سریع‌تر رشد کرده است.

نتیجه‌گیری:

کندترین: $f(n) = n * \log_2(n)$ (رشد لگاریتمی-ضربدر-خطی)

متوسط: $g(n) = 50n + 1000$ (رشد خطی)

سریع‌ترین: $h(n) = n^{(1.5)}$ (رشد توانی)

3. رتبه‌بندی نهایی (از کندترین به سریع‌ترین):

1. $f(n) = n * \log_2(n)$

2. $g(n) = 50n + 1000$

3. $h(n) = n^{(1.5)}$

نکته مهم: این مقایسه‌ها برای مقادیر بزرگ n (رفتار مجانبی) صادق هستند. در مقادیر کوچک n، ممکن است تابع خطی با ضریب بزرگ یا تابع توانی با توان کوچک، از توابع دیگر بزرگتر باشد. اما در تحلیل الگوریتم‌ها، تمرکز اصلی روی رفتار تابع برای n های بسیار بزرگ است.

پاسخ سوال چهار:

1) Quick Sort : ناپایداری Quick Sort معمولاً به دلیل نحوه جابجایی (swap) عناصر در طی فرآیند پارتیشن‌بندی (partitioning) رخ می‌دهد. در پیاده‌سازی‌های رایج، وقتی عنصری با عنصر محوری (pivot)جابجا می‌شود، اگر این دو عنصر برابر باشند، ترتیب نسبی آن‌ها ممکن است تغییر کند. همچنین، نحوه انتخاب pivot و چگونگی قرارگیری عناصر مساوی نسبت به آن نیز می‌تواند بر پایداری تأثیر بگذارد.

Merge Sort: پایداری Merge Sort به دلیل ماهیت ادغام (merge) دو زیرآرایه مرتب شده است. در مرحله ادغام، هنگامی که عناصر دو زیرآرایه با هم مقایسه می‌شوند، اگر مقادیر برابر باشند، همیشه

ابتدا عنصری از زیرآرایه چپ (که زودتر آمده) انتخاب می‌شود. این رویه تضمین می‌کند که ترتیب نسبی عناصر برابر حفظ شود.

2) برای اطمینان از اینکه پس از مرتب‌سازی نهایی بر اساس نام و سپس نمره، ترتیب کلی صحیح باشد، الگوریتم مرتب‌سازی مرحله دوم (بر اساس نمره) باید پایدار باشد.

توضیح :فرض کنید ابتدا لیست دانش‌آموزان را بر اساس نام مرتب کرده‌ایم (فرض کنید الگوریتم ما پایدار است). حالا می‌خواهیم بر اساس نمره مرتب کنیم. اگر دو دانش‌آموز نمره یکسان داشته باشند (مثلاً هر دو 18)، یک الگوریتم مرتب‌سازی پایدار تضمین می‌کند که ترتیب نسبی آن‌ها که از مرحله مرتب‌سازی نام به دست آمده بود، حفظ شود. یعنی اگر در مرتب‌سازی نام، دانش‌آموز "علی" قبل از "رضا" آمده بود و هر دو نمره 18 دارند، در مرتب‌سازی نهایی نیز "علی" قبل از "رضا" قرار خواهد گرفت. اگر الگوریتم ناپایدار باشد، ترتیب "علی" و "رضا" با نمره یکسان ممکن است پس از مرتب‌سازی بر اساس نمره، برعکس شود و ترتیب نهایی مطلوب حاصل نگردد.

3) Bubble Sort و Insertion Sort (جزئی از دسته جابجایی زوجی و درج) با وجود سادگی پیاده‌سازی، در بدترین حالت دارای پیچیدگی زمانی O(n^2) هستند که آن‌ها را در مقایسه با الگوریتم‌های O(n log n) (مانند Merge Sort یا Quick Sort ، کندتر می‌کند. البته Insertion Sort در داده‌های تقریباً مرتب شده، کارایی خوبی دارد.