آرش حاجی صفی «تمرین سری پنجم DS»

Year. Month. Date. Subject: ۹۶۳۱۰۱۹

ج ۱)

الگوریتم	بدترین حالت	حالت میانگین	بهترین حالت
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
Quick Sort	$O(n^2)$	$O(n\log n)$	$O(n\log n)$
Heap Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$

از آنجایی که آرایه‌ی داده‌شده از قبل مرتب‌شده‌ی صعودی است، پس الگوریتم‌های Insertion Sort و Bubble Sort بهتر عمل می‌کنند؛ چون که هردو در زمان خطی این آرایه را Sort می‌کنند (best case است به‌دلیل از قبل مرتب بودن)

ج ۲) حداکثر تعداد مقایسه برابر با $r+p-1$ مقایسه خواهد بود به این صورت که فرض کنیم مثلاً همه‌ی $r-1$ عنصر اول A، از عنصر اول B کوچکتر باشند که به $r-1$ مقایسه لازم می‌شود، حالا عنصر آخر (r ام) A از همه‌ی عناصر B بیشتر باشد که به p مقایسه (مقایسه عنصر آخر A با همه‌ی عناصر B) لازم می‌شود. هیچ حالتی بیشتر از این حالت مقایسه نخواهد داشت، پس حداکثر $r+p-1$ مقایسه خواهیم داشت

Year. Month. Date. Subject:

ج۳) الف) نادرست، در بدترین حالت پیچیدگی زمانی Quick Sort، $O(n^2)$ است که از $O(n\log n)$ برای Merge Sort بدتر است.

ب) نادرست، حداکثر درجه‌ی پیچیدگی هردو $O(n^2)$ است Binary Insertion Sort درست است از Insertion Sort مقایسه‌های کمتری انجام می‌دهد ولی در بدترین حالت به دلیل تعداد swap همچنان $O(n^2)$ است مثل بدترین حالت برای Quick Sort.

ج) نادرست، حداکثر درجه پیچیدگی Quick Sort، $O(n^2)$ است ولی حداکثر برای HeapSort، $O(n\log n)$ است.

د) درست، Quick Sort به طور میانگین $O(n\log n)$ زمان می‌برد و Insertion Sort، $O(n^2)$ پس برای ورودی‌های بزرگ مثل $n \geq 23$ حتماً Quick Sort به طور میانگین بهتر عمل می‌کند.

ه) درست، چون اگر از حافظه‌ی کمکی استفاده نکنند و از مقایسه استفاده کنند ثابت می‌شود نمی‌توانند سریع‌تر از $O(n\log n)$ عمل کنند.

ج۴) الف)

m=0	۰
m=1	۱۳۱
m=2	۱۴۴
m=3	۵۳
m=4	۱۳۴
m=5	۱۳۵
m=6	۳۹
m=7	۴۴
m=8	
m=9	۲۲
m=10	۱۵۲
m=11	
m=12	

۳۹ ← ۷
۵۳ ۱۴۴
۴۴
۱۵۲

Yasha

m = ۰	۵۰	۱۶ امانه ← ۳۹
m = ۱	۱۳۱	۱ امانه ← ۱۴۴ ، ۹ امانه ← ۵۳
m = ۲	۱۴۴	
m = ۳	۳۹	
m = ۴	۱۳۴	
m = ۵	۱۳۵	۱ امانه ← ۴۴
m = ۶	۴۴	
m = ۷		
m = ۸		
m = ۹	۲۲	۱۶ امانه ← ۱۵۲
m = ۱۰	۵۳	
m = ۱۱		
m = ۱۲	۱۵۲	

ج ۵) یک راه مناسب این است که به جای linked list در جایی که collision ایجاد می شود، از red black tree استفاده کنیم. در این صورت دسترسی به هر عنصر، حذف و اضافه از O(n) به O(logn) کاهش می یابد. در جایی هم که collision نداریم همه ی این عملیات ها O(1) باقی می ماند.

راه دیگر این است از Binary Search Tree استفاده کنیم که زمان ها در جایی که collision داریم به O(logk) کاهش می یابد.

روش دیگر این است که از یک تابع Hashing دیگر هم علاوه بر تابع اصلی استفاده کنیم و جایی که تصادم رخ می دهد یک آرایه کوچک دیگر برای آن با تابع Hashing جدید بسازیم که همه ی زمان ها O(1) بماند.

Year. Month. Date. Subject:

ج ۶)

```
BuildHeap_D&D(A, i, j)

if (j > i)  return
if j == i  return pointer to node A[i]
Find i ≤ m ≤ j such that x = A[m] is the min element in A[i]...A[j]
Swap A[m] with A[i]   در اینجا root به اول زیرآرایه می رود
left_child = BuildHeap_D&D(A, i+1, ⌊(i+1+j)/2⌋)

right_child = "  "  "  (A, ⌊(i+1+j)/2⌋ + 1, j)

r = new Node(x);
x.left = left_child;
x.right = right_child;
return x;   ← در اینجا یک minHeap در A[i, ..., j] ساخته شده و
              ریشه ی آن برگردانده می شود.
```

ج ۷) الگوریتم به این صورت عمل می کند که آرایه را نصف می کند و بزرگترین مجموع زیرآرایه سمت چپ، بزرگترین مجموع زیرآرایه سمت راست و بزرگترین مجموع زیرآرایه که از وسط می گذرد را بدست می آورد و ماکزیمم این ۳ تا، بزرگترین مجموع زیرآرایه ی آرایه می شود. فراخوانی ها به صورت رابطه ی بازگشتی انجام می شود.

ادامه →

Year. Month. Date. Subject:

ادامه‌ی V: برای پیدا کردن بزرگترین مجموع زیرآرایه که از وسط می‌گذرد، از این تابع کمکی استفاده می‌کنیم:

```
int maxSumIncludingMid(A, start, mid, end):
int Sum = 0;
int left_Sum = right_Sum = Integer.Min_Value;
for i = mid down to start do:
    sum = sum + A[i];
    if (sum > left_Sum)
        left_Sum = sum;
sum = 0;
for i = mid + 1 upto end do:
    sum = sum + A[i]
    if (sum > right_Sum)
        right_Sum = sum;

return left_Sum + right_Sum;
```

حالا تابع اصلی را می‌نویسیم:

```
int maxSubArraySum(A, start, end)
if start > end  return;
if start == end  return A[start];
int mid = (start + end) / 2;

return Maximum of maxSubArraySum(A, 1, mid)
                  and maxSubArraySum(A, mid+1, end)
                  and maxSumIncludingMid(A, start, mid, end)
```

Year. Month. Date. Subject:

ج ۸) این مساله ویژگی زیر ساختار بهینه (optimal substructure) دارد. پس به فکر اینکه هر زیر مساله را فقط یکبار حل کنیم از D.P استفاده می‌کنیم. به این صورت که

Case 1

می دانیم در هر زیر آرایه $S[i...j]$ اگر $i = j$ باشد یعنی طول زیر آرایه ۱ باشد،

Case 2

طول زیر دنباله خود واردن ۱ می‌شود و ۲- چنانچه طول زیر آرایه ۲ باشد و هر دو کاراکتر با هم برابر باشند، طول زیر رشته خود واردن ۲ می‌شود

Case 3

۳- چنانچه هیچ یک از حالات بالا نباشد و $S[i] = S[j]$ باشد، طول زیر رشته خود واردن برابر با طول زیر رشته خود واردن زیر رشته‌ای $S[i+1...j]$ + ۲ می‌شود

Case 4

۴- اگر هیچ یک از حالات بالا نباشد، طول زیر رشته خود واردن ماکسیمم بین خود رشته زیر واردن بین $S[i...j-1]$ و $S[i+1...j]$ است. با استفاده از D.P کد را می‌نویسیم

```
int LPS(String S):
int n = S.length();
int i, j, l;                    ← l طول زیر رشته است.
for i = 0 to n do:
    table[i][i] = 1;            ← Case 1
for l = 2 to n do:
    ① for i = 0 to n-l+1 do:
        ② j = i + l - 1;
        if(S[i] == S[j] and l == 2)
            L[i][j] = 2;        ← Case 2
```


ادامه صفحه بعد

Year. Month. Date. Subject:

ادامه ۸:

```
else if (S[i] == S[j])
    L[i][j] = L[i+1][j-1] + 2;      ← Case 3
else
    L[i][j] = max(L[i][j-1], L[i+1][j]);   ← Case 4

return L[0][n-1];
```

پیچیدگی زمانی: $T(n) = O(n^2)$ ← زمان اجرا از $O(n^2)$ است،

میزان حافظه‌ی اضافی هم $O(n^2)$ است.

ج ۹) برای این منظور از radix sort استفاده می‌کنیم

radix sort از $O(d \times (n+b))$ است که خود d از $O(\log_b(n))$ است، پس اگر

مبنا را n بگیریم ($b=n$) ← $d = O(1)$ ← radix sort از $O(n)$ می‌شود.

برای این کار به این صورت عمل می‌کنیم:

ابتدا اعداد را به مبنای n می‌بریم، در این صورت چون بازه‌ی اعداد از ۰ تا n^2-1 است

هر عدد حداکثر ۲ رقم در مبنای n دارد.

ابتدا اعداد را براساس LSD با آنها با یک الگوریتم مرتب سازی از $O(n)$ مثل

Counting Sort مرتب کرده (چون هر رقم حداکثر از ۰ تا $n-1$ است)

سپس اعداد را براساس MSD آنها دوباره با $O(n)$ مرتب می‌کنیم.

حالا آرایه‌ای مرتب شده در $O(n)$ داریم.

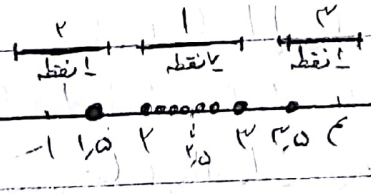
ادامه ←

Yasha

کد سوال ۹:

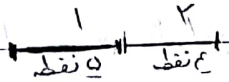
```
Void countSort (arr, n, exp):
    int output[] = new int[n];
    int i, count = new int[n];
    for i=0 to n do
        count[i] = 0;
    for i=0 to n do:                    ← تعداد هر رقم مشخص می‌شه
        count[ (arr[i]/exp)%n ]++;
    for i=1 to n do:                    ← جایی که هر رقم باید در خروجی آرایه قرار بگیره مشخص می‌شه
        count[i] += count[i-1]
    for i=n-1 to 0 do:                  ← ساختن output
        output[ count[(arr[i]/exp)%n] - 1 ] = arr[i]
        count[ (arr[i]/exp%n] --;
    for i=0 to n do:
        arr[i] = output[i]              ← کپی کردن خروجی sort شده در آرایه اصلی

Void sort(arr, n)
    countSort(arr, n, 1);               ← اول براساس LSD، sort می‌کنیم
    countSort(arr, n, n);               ← سپس براساس MSD، sort می‌کنیم
```

ج ۱۰) الف) نادرست است؛ مثال نقض:



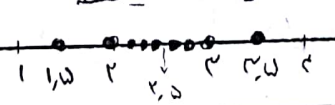
۳ پاره‌خط ←

این الگوریتم



نمونه ۲ پاره‌خط ←

حالت بهینه



Year. Month. Date. Subject:

ادامه‌ی ۱۰) ب) درست است.

اثبات: در حالت بهینه حتماً سمت چپ‌ترین نقطه باید انتهای سمت چپ چپ‌ترین پاره خط انتخابی باشد. چون با استفاده از برهان خلف اگر همچین چیزی نباشد و حالت بهینه باشد، یعنی چپ‌ترین پاره‌خط نقاط بیشتری را از حالت گفته شده بپوشاند. نقطه‌ی مذکور چپ‌ترین نقطه نبوده!

به همین صورت اول که سمت چپ را پاره خط گذاشتیم و نقاط آن قسمت حذف شد، باز هم حالت بهینه باتوجه به استدلال گفته شده این است که از سمت چپ‌ترین نقطه‌ی باقیمانده پاره‌خطی به طول l بکشیم (سمت چپ‌ترین قسمت پاره خط روی نقطه باشد) تا همه‌ی نقاط تمام شوند.

ج ۱۱) در اصل یک ماتریس متقارن نسبت به قطر اصلی داریم که می‌خواهیم با جابجایی سطرها و ستون‌ها قرینه‌ی آن نسبت به قطر اصلی ماتریس حاصل شود.

به عنوان مثال:

rows \ Columns	1	2	3
1	0	-1	1
2	1	0	-1
3	-1	1	0

تبدیل شود به

	1	2	3
1	0	1	-1
2	-1	0	1
3	1	-1	0

TransposeMatrix(M) → آرایه دوبعدی M[n][n]

این الگوریتم پیشنهاد می‌شود.

یعنی باید سطرهای $i=0$ تا n جوری سطرها با هم، ستون‌ها با هم را جابجا کنیم که سطر i ام با ستون n ام جابجا شود.

```
for i=1 upto n do:
    for j=1 upto i/2 do.
        swap row(j) with row(n-j) ;
        swap column(j) with column(n-j)
```

Yasha

Year. Month. Date. Subject:

ج ۱۲) با استفاده از جستجوی کامل فضای حالات به روش back tracking می‌توان ماکزیمم تعداد کارمندان را بدست آورد. ابتدا حالتی را بررسی می‌کنیم که همه‌ی کارمندان دعوت شده باشند، اگر شرط مخالف بودن یکی بین هر دو نفر برقرار بود که همه کارمندان می‌توانند دعوت شوند، یعنی n نفر، اگر برقرار نبود تمام حالت هایی را بررسی می‌کنیم که همه جز یک نفر دعوت شود باشند و حالت بعدی همه جز ۲ نفر و ... الی آخر تا جایی که یک حالت جواب پیدا شود که چون از بزرگ به کوچک می‌رویم اولین حالت بدست آمده ماکزیمم خواهد بود.

یک روش دیگر استفاده از این تابع است:

آرایه‌ی کارمندان

```
int findMaxInvitations(A, i, j):
if(i==j) return 1
if(j==i+1)
    if((Pi>Pj && Ci>Cj) || (Pj>Pi && Cj>Ci))
        return 2;
    else return 0;
fmax = findMaxInvitations(A, i, j-1); boolean canBeInvited;
for K=i upto j-1 do
    if((Pj>Pk && (Cj>Ck) || (Pj>Pk && Cj>Ck))
        canBeInvited = true;
    else
        canBeInvited = false;
        break;
if(canBeInvited)
    return max+1
else
    return max
```

Yasha

با استفاده از تابع بالا نقطه‌ی شروع را به ازای هر کارمند جدا حساب می‌کنیم (i) و ماکزیمم آنها را حساب کرده که ماکزیمم قابل دعوت ها می‌شود.