Pembahasan Los Polos Armanos Restaurant Service

Deskripsi Singkat

Terdapat 6 kueri yang perlu diselesaikan untuk menjawab permasalahan pada tugas pemrograman kali ini.

1. **Advanced Scanning**, bagaimana cara menghitung banyak pelanggan '+' dan '-' dalam range R_{ij} ?
2. **P,** bagaimana cara mendapatkan koki dengan pelayanan minimal berdasarkan tipe dari makanan tersebut?
3. **L,** bagaimana cara mendapatkan pesanan terawal yang belum dilayani dan meningkatkan banyak layanan yang dilakukan koki?
4. **B,** bagaimana cara menentukan total harga pesanan yang dilakukan pelanggan dengan ID tertentu?
5. **C,** bagaimana cara mendapatkan **Q** koki dengan pelayanan paling sedikit dan menghiraukan tipe dari koki tersebut?
6. **D,** bagaimana cara pelanggan mendapatkan total harga paling murah dengan daftar harga paket yang telah diberikan?

Advanced Scanning

Solusi Naif

Cara paling mudah adalah dengan mengiterasi dari pelanggan ke-($j - R_{ij}$) hingga ($j - 1$) sambil dihitung banyaknya status '+' dan banyaknya status '-', sehingga kompleksitasnya adalah $O(R_{ij})$.

Perlu diperhatikan bahwa jumlah pelanggan maksimal yang akan datang pada hari ke-i adalah 100.000 pelanggan. Jika untuk setiap pelanggan diperlukan Advanced Scanning dengan $R_{ij} = j$, maka diperlukan sebanyak $P_i(P_i + 1)/2$ atau $O(P^2)$. Sehingga solusi di atas akan memakan waktu yang sangat lama ketika pelanggan yang datang cukup besar.

Solusi ini hanya akan *accepted* untuk *test case* 1-40.

Solusi Optimal

Cara yang dapat dilakukan adalah dengan menerapkan Prefix Sum, yaitu dengan menyimpan jumlah '+' dan '-' dari pelanggan ke-1 hingga pelanggan ke-j. Solusi ini memiliki kompleksitas $O(1)$.

Berikut contoh visualisasinya.

j	1	2	3	4	5
Status	+	-	-	+	?
Jumlah '+'	1	1	1	2	
Jumlah '-'	0	1	2	2	

Misalkan R_{i5} adalah 3. Untuk mendapatkan jumlah '+' dan jumlah '-' pada pelanggan ke-2 hingga ke-4, kita hanya perlu mengurangi jumlah masing-masing status hingga pelanggan ke-4 dan jumlah status hingga pelanggan ke-1.

- Jumlah '+' = positif[4] - positif[1] = 2 - 1 = 1
- Jumlah '-' = negatif[4] - negatif[2] = 2 - 0 = 2

Karena jumlah '+' pada pelanggan ke-2 hingga ke-4 adalah 1 sedangkan jumlah '-' adalah 2, maka pelanggan ke-5 memiliki status '-'.

Solusi ini hanya membutuhkan 1 operasi pengurangan, sehingga kompleksitasnya adalah $O(1)$ dan menggunakan memori sebanyak $O(P)$.

Kueri P

Setiap pesanan yang diterima memiliki sifat **FIFO (First In First Out)** dan struktur data yang mendukung ini adalah **Queue**. Sehingga agar pesanan yang terawal bisa dilayani terlebih dahulu, setiap menerima kueri P, kita harus menyimpan datanya di queue.

Selanjutnya, kita perlu menentukan koki mana yang akan melayani pesanan ini. Koki yang akan melayani adalah koki dengan jumlah pelayanan minimal berdasarkan tipe dari makanan tersebut.

Solusi Naif

Untuk query ini, cara paling mudah adalah dengan menganggap suatu koki minimal, K_{min} , kemudian mengiterasi koki tersedia, K_i , (sesuai dengan tipe-nya) dan membandingkan banyak layanan dengan koki minimal sementara.

Akan ada 4 kemungkinan yang terjadi ketika membandingkan banyak layanan tersebut.

1. $K_{min}.banyakLayanan < K_i.banyakLayanan$, maka K_{min} tidak berubah
2. $K_{min}.banyakLayanan > K_i.banyakLayanan$, maka $K_{min} = K_i$
3. $K_{min}.banyakLayanan == K_i.banyakLayanan$, maka yang akan dibandingkan ID dari koki.
   a. $K_{min}.id < K_i.id$, maka K_{min} tidak berubah
   b. $K_{min}.id > K_i.id$, maka $K_{min} = K_i$

Cara ini memiliki komplekstias $O(jumlahKoki)$ atau $O(V)$ untuk setiap pelayanan dan $O(XV)$ untuk X pelayanan. Karena diketahui

- $1 \leq V \leq 100\,000$ koki
- $1 \leq X_i \leq 200\,000$ pelayanan

Maka solusi ini tidak akan efisien ketika nilai V dan X cukup besar. Solusi ini hanya akan *accepted* untuk *test case* 1-40.

Solusi Optimal

Kesalahan dari cara tersebut adalah **penggunaan struktur data yang tidak tepat**. Dengan menggunakan struktur data tertentu, pencarian jumlah pelayanan minimal dapat lebih cepat. Struktur data yang mendukung ini adalah **Priority Queue.**

Implementasi *underlying* dari **priority queue** tidak akan dibahas, tetapi perlu dipahami bahwa dengan menggunakan **priority queue,** pencarian elemen minimal hanya membutuhkan $O(1)$. Namun untuk melakukan *poll* di priority queue yang memiliki N elemen memerlukan $O(log\ N)$. Sehingga kompleksitas dari solusi optimal adalah $O(log\ X)$.

Perhatikan dari segi kompleksitas cara ini sudah tepat, tetapi untuk kebenaran belum sepenuhnya benar karena adanya kueri L. Solusi optimal dari kueri P akan dibahas pada penjelasan kueri **L.**

Kueri L

Urutan pesanan yang dilayani akan sesuai dengan queue yang telah diisi oleh kueri P. Setiap kali dilakukan *query* **L**, banyak pelayanan dari koki yang melayani akan ditambah 1.

Observasi Kueri

Misalkan ada 3 koki dan barisan kueri $(Q_1, Q_2, .. Q_n)$ dengan $Q_i \in \{ P, L \}$. Untuk observasi ini, ID pelanggan, ID makanan, dan tipe koki dapat diabaikan karena tujuan utamanya adalah memahami **pola** dari kedua kueri tersebut.

Pada kondisi awal, banyak pelayanan setiap koki adalah 0. $Koki_{min}$ adalah daftar koki yang telah terurut berdasarkan banyak pelayanan dan nomor ID, $Koki_{layan}$ adalah daftar koki yang harus melayani jika kueri L dilakukan, terurut dari koki pertama yang harus melayani.

		Banyak Pelayanan			Daftar Koki	
Langkah	**Kueri**	**Koki ID 1**	**Koki ID 2**	**Koki ID 3**	$Koki_{min}$	$Koki_{layan}$
1	P	0	0	0	Koki 1, Koki 2, Koki 3	Koki 1
2	L	1	0	0	Koki 2, Koki 3, Koki 1	-
3	P	1	0	0	Koki 2, Koki 3, Koki 1	Koki 2
4	P	1	0	0	Koki 2, Koki 3, Koki 1	Koki 2
5	L	1	1	0	Koki 3, Koki 1, Koki 2	Koki 2
6	L	1	2	0	Koki 3, Koki 1, Koki 2	-
7	P	1	2	0	Koki 3, Koki 1, Koki 2	Koki 3
8	P	1	2	0	Koki 3, Koki 1, Koki 2	Koki 3
9	L	1	2	1	Koki 1, Koki 3, Koki 2	Koki 3
10	P	1	2	1	Koki 1, Koki 3, Koki 2	Koki 3, Koki 1
11	L	1	2	2	Koki 1, Koki 2, Koki 3	Koki 1
12	P	1	2	2	Koki 1, Koki 2, Koki 3	Koki 1
13	L	2	2	2	Koki 1, Koki 2, Koki 3	Koki 1
14	P	2	2	2	Koki 1, Koki 2, Koki 3	Koki 1

Dari hasil observasi di atas, maka hal yang dapat kita peroleh adalah

1. Setiap kueri P, $Koki_{min1}$ yang sama saja dengan $Koki_{layan}$ terakhir akan melayani pesanan
2. Setiap kueri L, banyak layanan koki yang menerima pesanan ($Koki_{layan1}$) ditambah 1
3. Karena pesanan terawal selalu dilayani lebih dulu, dapat dipastikan ketika kueri L banyak pelayanan yang akan bertambah adalah milik koki dengan pelayanan minimal di **saat itu** ($Koki_{layan1}$) selama belum semua pesanan milik koki tersebut dilayani.
4. Selama pesanan yang harus dilayani koki minimal **saat itu** ($Koki_{layan1}$) belum habis, pesanan selanjutnya hanya akan diberikan ke koki minimal **saat ini** ($Koki_{min1}$). $Koki_{min1}$ dan $Koki_{layan1}$ bisa memiliki nilai yang sama
5. Ketika banyak layanan koki minimal **saat itu** ($Koki_{layan1}$) bertambah 1, maka salah satu kemungkinan ini dapat terjadi:
   a. $Koki_{min1}$ tidak lagi menjadi koki dengan jumlah pelayanan minimal → langkah 2
   b. $Koki_{min1}$ masih minimal, namun ID-nya lebih besar dari $Koki_{min2}$ → langkah 9
   c. $Koki_{min1}$ masih minimal dan ID-nya masih paling kecil → langkah 13

   Ketika kemungkinan a atau b terjadi, $Koki_{min1} = Koki_{min2}$

Dari sini, permasalahan dari penggunaan priority queue dapat terlihat.

- Kita tidak dapat meng-*update* banyak pelayanan koki **non minimal** (bukan $Koki_{min1}$) yang berada di dalam priority queue secara langsung (pada langkah 11, kita perlu meng-*update* data $Koki_3$ walaupun $Koki_{min1} = Koki_1$)
- Jika ingin meng-*update* data koki, kita perlu me-*remove* koki tersebut dari priority queue, meng-*update* data koki, lalu memasukkan kembali koki ke priority queue.
  a. Operasi *remove* pada priority queue terdiri dari mencari elemen yang ingin di-*remove* $O(V)$ lalu me-*remove*-nya sebesar $O(log(V)$, sehingga total membutuhkan $O(V)$.
  b. Operasi *insert* pada priority queue membutuhkan $O(log\ V)$

  Sehingga total membutuhkan $O(V + log\ V) = O(V)$, masih belum optimal.

Solusi Permasalahan

Kita akan mengadakan konsep **hutang layanan** terhadap koki minimal. Kita akan menambahkan **hutang pelayanan** daripada layanan itu sendiri.

		Banyak Pelayanan			Hutang Pelayanan			Daftar Koki	
Langkah	**Kueri**	**ID 1**	**ID 2**	**ID 3**	**ID 1**	**ID 2**	**ID 3**	$Koki_{min}$	$Koki_{layan}$
1	P	0	0	0	1			Koki 1, Koki 2, Koki 3	Koki 1
2	L	1	0	0	0			Koki 2, Koki 3, Koki 1	-
3	P	1	0	0		1		Koki 2, Koki 3, Koki 1	Koki 2
4	P	1	0	0		2		Koki 2, Koki 3, Koki 1	Koki 2
5	L	1	1	0		1		Koki 3, Koki 1, Koki 2	Koki 2

		Banyak Pelayanan			Hutang Pelayanan			Daftar Koki	
6	L	1	2	0		0		Koki 3, Koki 1, Koki 2	-
7	P	1	2	0			1	Koki 3, Koki 1, Koki 2	Koki 3
8	P	1	2	0			2	Koki 3, Koki 1, Koki 2	Koki 3
9	L	1	2	1			1	Koki 1, Koki 3, Koki 2	Koki 3
10	P	1	2	1	1		1	Koki 1, Koki 3, Koki 2	Koki 3, Koki 1
11	L	1	2	2	1		0	Koki 1, Koki 2, Koki 3	Koki 1
12	P	1	2	2	2			Koki 1, Koki 2, Koki 3	Koki 1
13	L	2	2	2	1			Koki 1, Koki 2, Koki 3	Koki 1
14	P	1	2	2	3			Koki 1, Koki 2, Koki 3	Koki 1

Dari hasil observasi di atas, maka hal yang dapat kita peroleh adalah

6. Setiap kueri P, hutang pelayanan selalu ditambahkan ke $Koki_{min1}$ yang sama saja dengan $Koki_{layan}$ terakhir
7. Setiap kueri L, banyak pelayanan ditambahkan dan hutang pelayanan dikurangkan ke $Koki_{layan}$ pertama

Berdasarkan kedua poin di atas, maka kita akan menggunakan struktur data *auxiliary*, yaitu **deque**.

Ketika kueri P, $Koki_{min1}$ harus sudah berada di **tail deque** (hasil observasi poin 6)**.** Sehingga jika belum ada, kita perlu me-*remove* $Koki_{min1}$ dari priority queue dan memasukkan koki tersebut ke dalam deque**.** Operasi `poll()` priority queue membutuhkan $O(log\ V)$ dan operasi `add()` deque hanya membutuhkan $O(1)$, sehingga total hanya membutuhkan $O(log\ V)$. Selanjutnya hutang pelayanan dapat ditambahkan ke koki yang berada di **tail deque** (hasil observasi poin 6)**.**

Maka, ketika ada kueri L, kita hanya perlu menambahkan banyak pelayanan koki dan mengurangi hutang pelayanan koki yang berada di **head deque** (hasil observasi poin 3 dan 7)**.**

Sekarang, timbul beberapa pertanyaan.

1. **Kenapa deque?**
   Sebenarnya struktur data apa saja dapat digunakan, seperti array contohnya. Hanya saja deque akan lebih memudahkan untuk mengubah nilai banyak layanan dan hutang pelayanan koki sesuai hasil observasi poin 6 dan 7.

2. **Kapan kita perlu memasukkan $Koki_{min2}$ dari priority queue ke dalam deque?**
   Kita akan memasukkan $Koki_{min2}$ ke dalam deque ketika kasusnya sesuai dengan poin 5a, sesuai dengan poin 5b, atau ketika deque kosong.

3. **Kapan kita akan meremove koki di dalam deque? Koki urutan keberapa yang akan di remove?**
   Berdasarkan hasil observasi poin 3, kita akan me-*remove* koki pertama dari deque ketika hutang layanannya = 0.

4. **Kenapa aman memasukkan koki ke dalam deque? Bukannya kebenaran dari priority queue bisa jadi salah?**
   Dari hasil observasi poin 4, 5, dan 6 kita tahu bahwa koki yang perlu dipertimbangkan untuk menerima pesanan hanya $Koki_{min1}$ atau $Koki_{min2}$. Sehingga memasukkan koki tersebut ke dalam deque tidak akan mengubah kebenaran priority queue.

Detail Algoritma

Misalkan D = deque dari koki dan PQ = priority queue dari koki dengan `PQ.peek()` adalah koki dengan banyak pelayanan terkecil pada PQ.

Algoritma Kueri P

1. Jika salah satu dari kondisi-kondisi ini memenuhi
   a. D kosong
   b. D tidak kosong dan `banyakPelayanan tail D > banyakPelayanan PQ.peek()`
   c. D tidak kosong, `banyakPelayananani tail D == banyakPelayanan PQ.peek()`, dan `ID tail D < ID PQ.peek()`

   maka tambahkan $Koki_{min}$ dari PQ ke D → `D.add(pq.poll())`
2. Tambahkan hutang pelayanan koki paling belakang → `D.getLast().hutangPelayanan++`

Berdasarkan algoritma di atas, jika salah satu dari ketiga kondisi yang dijabarkan pada nomor 1 terpenuhi, artinya nilai dari $Koki_{min1}$ pada adalah `PQ.peek()`. Oleh karena itu koki tersebut dipindahkan ke deque.

Algoritma Kueri L

1. Tambahkan `banyakPelayanan` head deque → `D.getFirst().banyakPelayanan++`
2. Kurangkan `hutangPelayanan` head deque → `D.getFirst().hutangPelayanan–`
3. Jika `hutangPelayanan head deque = 0`, *remove* head deque lalu *insert* ke priority queue → `PQ.add(D.removeFirst())`

Sehingga kompleksitas untuk query P dan L masing-masing $O(log\ V)$.

Kueri B

Pada kueri L, setiap pesanan dilayani, kita perlu menambahkan jumlah uang yang perlu dibayar pelanggan tersebut. Sehingga ketika menjalankan kueri B, kita cukup mengecek apakah jumlah uang pelanggan mencukupi untuk membayar atau tidak. Hanya ada 2 kasus yang akan terjadi dari kueri ini:

- `jumlahUangHarusDibayar` $\leq$ `jumlahUangPelanggan` $\rightarrow$ pelanggan membayar
- `jumlahUangHarusDibayar` $>$ `jumlahUangPelanggan` $\rightarrow$ pelanggan di-*blacklist*

Kueri C

DIketahui PQ_S, PQ_G, dan PQ_A adalah priority queue yang menyimpan data koki sesuai dengan tipenya, terurut berdasarkan jumlah pelayanan dari terkecil ke terbesar. Priority queue ini digunakan juga pada kueri P dan kueri L yang sebelumnya telah dijelaskan.

Berikut ini adalah langkah-langkah pengerjaan kueri C:

1. Duplikat ketiga PQ tersebut ke priority queue *temporary* PQT_S, PQT_G, dan PQT_A
2. Bandingkan jumlah pelayanan koki yang berada di head dari setiap PQT (PQT`.peek()`) untuk menentukan koki dengan jumlah pelayanan terendah berada di tipe apa. Jika nilai terendah ada lebih dari satu, maka pilih berdasarkan tipe S, tipe G, lalu tipe A.
3. Keluarkan koki terpilih dari PQT sesuai tipenya (PQT`.poll()`), lalu cetak ID dari koki tersebut
4. Lakukan langkah 2 dan 3 sejumlah Q kali, sesuai dengan banyak koki yang diminta kueri

Kueri D

Di bagian ini, kita akan menggunakan pendekatan rekursif. Pertama kita perlu membangun fungsi rekursifnya terlebih dahulu, kurang lebih seperti berikut.

Index dimulai dari 1 hingga n, sedangkan makna nilai dari statusS, statusG, dan statusA adalah:

- 0: paket **belum** terbuat
- 1: paket **sedang** dalam proses pembuatan
- 2: paket **sudah** terbuat

```
int costS, costG, costA;

long rec(int index, int statusS, int statusG, int statusA) {

   // -- BASE CASE
   if (index > n) {
       // Dicek valid tidaknya, yaitu ketika seluruh status nilainya != 1
```

```
        // Jika valid
        return 0;

        // Jika tidak valid, maka diset infinity atau
        // highest value agar tidak mungkin menjadi jawaban
        return (long)1e18;
    }

    // Penyimpanan costMin sementara, karena ingin mendapatkan
    // nilai terkecil, maka di awal diset sebagai highest value
    long costMin = (long)1e18;

    // -- RECURSIVE CASE
    if (statusS == 1) {
        // Jika sedang dalam proses pembuatan paket S

        // CASE 1: Melanjutkan membuat paket S
        costMin = Math.min(costMin, rec(index + 1, statusS, statusG, statusA) + costS);

        // CASE 2: Menyelesaikan paket S (jika makanan di index ini bertipe S)
        // statusS berubah menjadi 2
        if (tipe[index]=='S') {
            costMin = Math.min(costMin, rec(index + 1, 2, statusG, statusA) + costS);
        }
    } else if (statusG == 1) {
        // Implementasi mirip dengan S
    } else if (statusA == 1) {
        // Implementasi mirip dengan S
    } else {
        // Jika sekarang sedang tidak dalam proses pembuatan paket apapun

        // CASE 1: Memulai untuk membuat paket bertipe S jika paket tersebut
        // belum pernah terbuat dan tipenya sesuai. statusS berubah menjadi 1
        if (statusS == 0 && tipe[index] == 'S')
            costMin = Math.min(costMin, rec(index + 1, 1, statusG, statusA) + costS);

        // CASE 2-3: Memulai untuk membuat paket bertipe A / G jika paket
        // tersebut belum pernah terbuat dan tipenya sesuai
        // Implementasi mirip dengan S

        // CASE 4: Membeli langsung secara satuan tanpa membuat paket
        costMin = rec(index + 1, statusS, statusG, statusA) + harga[index];
```

```
    }

    return costMin;
}
```

Ternyata solusi di atas tidak cukup untuk menyelesaikan permasalahan kita, dikarenakan kompleksitas yang terlalu tinggi sehingga program kita akan berjalan lambat. Kompleksitas kode di atas yaitu kurang lebih $O(2^N)$. Karena itu perlu kita optimasi lagi, salah satu caranya yaitu menggunakan pendekatan *dynamic programming* atau DP.

Sebelum kita implementasikannya, kita perlu mengecek beberapa hal untuk mengetahui apakah masalah ini dapat diselesaikan DP, untuk nilai dari parameter yang sama apakah fungsi pasti akan mengembalikan nilai yang sama. Hal tersebut dapat dibuktikan dengan melihat variable luar yang digunakan selain parameter yaitu `costA, costG, costS, n,` dan `tipe`. Variable tersebut **hanya** berubah untuk **kueri D yang berbeda**, sehingga DP yang kita implementasikan perlu di-*reset* untuk setiap kueri D yang berbeda. Pada kondisi awal tiap menjalankan kueri D, table DP dapat kita set dengan nilai -1 dan berikut penambahan kode di fungsinya.

```
int costS, costG, costA;

long rec(int index, int statusS, int statusG, int statusA) {

    // -- BASE CASE
    if (index > n) {
        // Dicek valid tidaknya, yaitu ketika seluruh status nilainya != 1

        // Jika valid
        return 0;

        // Jika tidak valid, maka diset infinity atau
        // highest value agar tidak mungkin menjadi jawaban
        return (long)1e18;
    }

    // Jika sudah pernah dihitung maka tinggal menggunakan jawaban
    // yang telah disimpan dalam table DP
    if (dp[index][statusS][statusG][statusA] != -1) {
        return dp[index][statusS][statusG][statusA];
    }

    // Penyimpanan costMin sementara, karena ingin mendapatkan
    // nilai terkecil, maka di awal diset sebagai highest value
    long costMin = (long)1e18;
```

```
    // -- RECURSIVE CASE
    if (statusS == 1) {
        // Jika sedang dalam proses pembuatan paket S

        // CASE 1: Melanjutkan membuat paket S
        costMin = Math.min(costMin, rec(index + 1, statusS, statusG, statusA) + costS);

        // CASE 2: Menyelesaikan paket S (jika makanan di index ini bertipe S)
        // statusS berubah menjadi 2
        if (tipe[index]=='S') {
            costMin = Math.min(costMin, rec(index + 1, 2, statusG, statusA) + costS);
        }
    } else if (statusG == 1) {
        // Implementasi mirip dengan S
    } else if (statusA == 1) {
        // Implementasi mirip dengan S
    } else {
        // Jika sekarang sedang tidak dalam proses pembuatan paket apapun

        // CASE 1: Memulai untuk membuat paket bertipe S jika paket tersebut
        // belum pernah terbuat dan tipenya sesuai. statusS berubah menjadi 1
        if (statusS == 0 && tipe[index] == 'S')
            costMin = Math.min(costMin, rec(index + 1, 1, statusG, statusA) + costS);

        // CASE 2-3: Memulai untuk membuat paket bertipe A / G jika paket
        // tersebut belum pernah terbuat dan tipenya sesuai
        // Implementasi mirip dengan S

        // CASE 4: Membeli langsung secara satuan tanpa membuat paket
        costMin = rec(index + 1, statusS, statusG, statusA) + harga[index];
    }

    // Menyimpan hasil perhitungan dalam table DP
    dp[index][statusS][statusG][statusA] = costMin;

    return costMin;
}
```

Kompleksitas kode kita menurun menjadi $O(54N)$ atau dapat disederhanakan menjadi $O(N)$ saja.