Pembahasan Ladang Boba Izuri

Deksripsi Singkat

Diberikan *N* buah ladang yang masing-masing memiliki *Ai* boba. Kemudian juga disediakan *M* buah keranjang yang masing-masing memiliki kapasitas awal sebesar *Ci* dan fleksibilitas sebesar *Fi*. Pada setiap ladang, sebuah keranjang dapat memanen boba atau melakukan ekspansi kapasitas. Setelah itu, terdapat H hari dimana masing-masing hari selain hari pertama berisi satu query spesial (Izuri) dan, *Y* query biasa yang diikuti dengan angka *O* yang menandakan jumlah query yang akan dilayani pada hari tersebut. Hitung hasil panen optimal yang dapat dilakukan masing-masing keranjang dan tampilkan secara terurut dari terbesar hingga terkecil berdasarkan hasil panen. Jika hasil panen sama, urutkan sesuai leksikografis dari yang terkecil hingga terbesar.

Ide

Tipe data yang akan digunakan adalah sebagai berikut:

```
class Keranjang implements Comparable<Keranjang> {
    String nama;
    int C, F, bobaCollected;

    Keranjang(String nama, int c, int f) {
        this.nama = nama;
        C = c;
        F = f;
        bobaCollected = 0;
    }

    public int compareTo(Keranjang lain) {
        if (bobaCollected > lain.bobaCollected) return -1;
        else if (bobaCollected < lain.bobaCollected) return 1;
        else return nama.compareTo(lain.nama);
    }
}

class Query {
    String custName;
    String operation;

    Query(String custName, String operation) {
        this.custName = custName;
        this.operation = operation;
    }
}
```

Struktur data yang akan digunakan adalah sebagai berikut:

```
private static final int INF = 1000000000;
private static final int MAXN = 105;
private static int N, M, H, Y, O;
private static int[] ladangBoba = new int[MAXN];
private static HashMap<String, Keranjang> keranjangMap = new HashMap<>();
private static Queue<Query> queryQueue = new LinkedList<>();
private static int dp[][] = new int[MAXN][101*MAXN];
private static List<Keranjang> listKeranjang = new ArrayList<>();
```

Pembahasan akan dibagi menjadi tiga bagian, yaitu proses pencarian hasil panen optimal, proses pengurutan keranjang berdasarkan hasil panen, dan pemrosesan query.

1. **Pencarian Hasil Panen Optimal**

   Secara naif (mencoba semua kemungkinan), pencarian hasil panen paling optimal dapat dicari dengan cara mencoba setiap kemungkinan panen atau ekspansi kapasitas di setiap ladang. Pencarian ini dapat dilakukan dengan cara rekursif tanpa memo ataupun dengan memo. Berikut method untuk pencarian hasil panen paling optimal.

   - Naif rekursif tanpa memo

```
private static int hitungTotalBoba(int pos, int c, int f) {
        if (pos == N) return 0;

        int tidakAmbil = hitungTotalBoba(pos+1, c+f, f);
        int ambil = hitungTotalBoba(pos+1, c-min(c, ladangBoba[pos]), f) +
min(c, ladangBoba[pos]);

        return max(ambil, tidakAmbil);
}
```

   Base case pada kasus ini adalah ketika pengecekan sudah melewati ladang terakhir. Untuk *recursive case* terdapat dua kemungkinan, yaitu ketika melakukan **ekspansi (tidak panen)** dan ketika **memanen** boba. Ketika melakukan ekspansi, kapasitas saat ini (c) ditambah dengan fleksibilitas (f). Ketika memanen boba, kapasitas yang tersedia berkurang sesuai dengan jumlah boba yang dipanen. `min(c, ladangBoba[pos])` menangani kasus ketika boba di suatu ladang lebih besar dari kapasitas, sehingga jumlah boba yang dipanen hanya sebesar kapasitas keranjang yang tersedia.

   Kompleksitas waktu untuk potongan kode di atas adalah O(2^N).

   - Naif rekursif dengan memo

```
private static int hitungTotalBoba(int pos, int cap, int f) {
        if (pos == N) return 0;

        if (dp[pos][cap] == -1) {
            int ambil = min(cap, ladangBoba[pos]);
            dp[pos][cap] = hitungTotalBoba(pos+1, cap-ambil, f) + ambil;
            dp[pos][cap] = max(dp[pos][cap], hitungTotalBoba(pos+1, cap+f, f));
        }
        return dp[pos][cap];
}

private static int hitungTotalBoba(int c, int f) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < 101*N; j++) {
                dp[i][j] = -1;
            }
        }
        return hitungTotalBoba(0, c, f);
    }
```

   Untuk menggunakan loop, tetap membutuhkan **pencatatan jumlah boba yang dipanen** pada sebuah *array*. **Posisi dan kapasitas** dapat berubah-ubah seiring dengan berjalannya panen sehingga perlu dijadikan indeks pada *array*. Pada awalnya, semua elemen pada *array* `dp` bernilai -1, yang menandakan bahwa elemen tersebut belum diproses. Rekursif pada algoritma ini sama dengan rekursif tanpa

memo tetapi hasil perhitungan akhir disimpan ke dalam *array* pada indeks yang sesuai.

Jika digambarkan pada sebuah tabel, berikut isi *array* `dp` sebelum dilakukan kalkulasi.

pos \ cap	0	1	2	...	N*F
0	-1	-1	-1	...	-1
1	-1	-1	-1	...	-1
2	-1	-1	-1	...	-1
3	-1	-1	-1	...	-1
...	...	...	...	...	...
N	-1	-1	-1	...	-1

Baris menandakan posisi ladang saat ini (dimulai dari indeks 0). Kolom cap menandakan kapasitas saat ini. Setiap pergeseran kolom, kapasitas saat ini ditambah dengan nilai fleksibilitas.

Kompleksitas waktu untuk potongan kode tersebut adalah O(N^2F) dan kompleksitas memori O(N^2F). Potongan kode ini dapat berjalan optimal untuk nilai N dan F yang tidak terlalu besar.

Untuk menurunkan kompleksitas, dibutuhkan pendekatan *dynamic programming* dengan cara mencatat dan menggunakan hasil kalkulasi yang sudah dilakukan sebelumnya. Dengan begini, operasi yang dilakukan berulang kali dapat dipangkas. Algoritma *dynamic programming* dapat diadaptasi dari algoritma naif rekursif dengan memo sebelumnya. Berikut potongan kode *dynamic programming*.

```
private static int hitungTotalBoba(int pos, int remExp, int c, int f) {
        if (pos == -1) return remExp == 0? 0: -INF;

        if (dp[pos][remExp] == -1) {
            dp[pos][remExp] = min(hitungTotalBoba(pos-1, remExp, c, f) +
ladangBoba[pos], c+remExp*f);
            if (remExp > 0) dp[pos][remExp] = max(dp[pos][remExp],
hitungTotalBoba(pos-1, remExp-1, c, f));
        }
        return dp[pos][remExp];
}

private static int hitungTotalBoba(int c, int f) {
        for (int i = 0; i < MAXN; i++) {
            for (int j = 0; j < MAXN; j++) dp[i][j] = -1;
        }
        int hasilOpt = -INF;
        for (int exp = 0; exp <= N; exp++) {
            int hasilKeranjang = hitungTotalBoba(N-1, exp, c, f);
            hasilOpt = max(hasilOpt, hasilKeranjang);
        }
        return hasilOpt;
}
```

Fungsi `hitungTotalBoba` pertama berfungsi sebagai fungsi rekursif untuk perhitungan saja sedangkan fungsi `hitungTotalBoba` kedua yang akan dipanggil pada `main`. Bentuk *array* `dp` pada algoritma ini berbeda dari *array* sebelumnya karena indeks kedua dari *array* berupa **berapa kali ekspansi**, bukan kapasitas asli dari keranjang. Berikut bentuk *array* `dp` jika direpresentasikan dalam tabel.

pos \ remExp	0	1	2	...	N
0	-1	-1	-1	...	-1
1	-1	-1	-1	...	-1
2	-1	-1	-1	...	-1
3	-1	-1	-1	...	-1
...	...	...	...	...	...
N	-1	-1	-1	...	-1

Baris menandakan posisi ladang saat ini (dimulai dari indeks 0). Kolom remExp keranjang sudah berapa kali melakukan ekspansi. Kalkulasi kapasitas keranjang dapat dihitung dengan rumus **C + remExp * F**. Dimensi *array* yang digunakan pasti sebesar N x N karena ekspansi tidak dapat dilakukan lebih dari jumlah ladang.
Kompleksitas waktu dan memori untuk potongan kode di atas adalah O(N^2)

2. **Proses Pengurutan Keranjang Berdasarkan Hasil Panen**

   Proses pengurutan dapat dilakukan dengan dua metode, ***sorting* konvensional** atau ***insertion sort* yang dimodifikasi** dengan mengikuti permasalahan pada soal. *Sorting* konvensional yang dapat memenuhi batasan soal yaitu *sorting* dengan **kompleksitas O(N log N)**. Contoh sorting dengan kompleksitas *average case* O(N log N) yang dapat digunakan yaitu *merge sort*, *quick sort*, dan *heap sort*.

   Solusi alternatif yaitu *insertion sort* yang dimodifikasi merupakan penyisipan keranjang ke dalam sebuah *array* yang selalu *sorted*. *Insertion sort* ini dilakukan **setiap dibuatnya keranjang baru** dan **ketika ada keranjang yang berubah**, baik nama (S), kapasitas awal (C), maupun fleksibilitas (F). Jika *sorting* dilakukan dengan cara ini, setiap ada elemen baru yang ingin disisipkan dalam *array*, *array* akan selalu *sorted*. Kompleksitas *sorting* akan berkurang menjadi **O(N) untuk setiap keranjang**, atau dapat dinyatakan dalam bentuk O(M*N). Berikut potongan kode dari *insertion sort* yang dimodifikasi.

```
private static void insertToListKeranjang(Keranjang keranjang) {
        int idx = listKeranjang.size();
        while (idx > 0 && listKeranjang.get(idx-1).compareTo(keranjang) > 0)
idx--;
        listKeranjang.add(idx, keranjang);
}
```

3. **Pemrosesan Query**

Y query yang masuk tiap hari dapat disimpan dalam struktur data queue sementara query Izuri dapat disimpan secara terpisah. Setiap harinya, sebanyak O query akan di*poll* dari queue untuk diproses. Query-query yang belum dilayani akan tetap tersimpan dalam queue dan di*poll* pada hari-hari selanjutnya untuk diproses. Query Izuri diproses setelah memproses O query yang sebelumnya.

Untuk tiap operasi yang mungkin, proses-proses yang dilakukan adalah sebagai berikut:

- ADD
  Periksa apakah terdapat objek Keranjang dengan nama yang diinput dengan memeriksa pada keranjangMap. Jika tidak, buat objek Keranjang baru dan hitung jumlah boba optimal yang dapat dipanennya. Kemudian tambahkan objek Keranjang ke dalam keranjangMap dengan key nama dan listKeranjang. Penambahan ke listKeranjang.
- SELL
  Periksa apakah terdapat objek Keranjang dengan nama yang diinput dengan memeriksa pada keranjangMap. Jika ya, hapus objek tersebut dari keranjangMap dan listKeranjang.
- RENAME
  Periksa apakah terdapat objek Keranjang dengan nama lama dan baru yang diinput dengan memeriksa pada keranjangMap. Jika ada nama lama dan tidak ada nama baru, hapus objek Keranjang pada keranjangMap dengan key nama lama dan tambahkan objek Keranjang dengan key nama baru. Kemudian, hapus objek Keranjang dari listKeranjang, ganti nama dari objek Keranjang, dan tambahkan kembali ke listKeranjang.
- UPDATE
  Periksa apakah terdapat objek Keranjang dengan nama yang diinput dengan memeriksa pada keranjangMap. Jika ya, hapus objek Keranjang dari listKeranjang, ganti C dan F dari objek Keranjang, hitung kembali jumlah boba optimal, dan tambahkan kembali ke listKeranjang.

```
private static void handleQuery(String query){
    String[] split = query.split(" ");
    Keranjang keranjang;
    switch (split[0]){
        case "ADD":
            if (keranjangMap.containsKey(split[1])) break;
            keranjang = new Keranjang(split[1],
Integer.parseInt(split[2]),Integer.parseInt(split[3]));
            keranjang.bobaCollected = hitungTotalBoba(keranjang.C, keranjang.F);
            keranjangMap.put(split[1], keranjang);
            insertToListKeranjang(keranjang);
            break;
        case "SELL":
            if (!keranjangMap.containsKey(split[1])) break;
            keranjang = keranjangMap.get(split[1]);
            keranjangMap.remove(split[1]);
            listKeranjang.remove(keranjang);
            break;
        case "RENAME":
            if (!keranjangMap.containsKey(split[1]) ||
keranjangMap.containsKey(split[2])) break;
```

```
            keranjang = keranjangMap.get(split[1]);
            keranjang.nama = split[2];
            keranjangMap.remove(split[1]);
            keranjangMap.put(split[2], keranjang);
            listKeranjang.remove(keranjang);
            insertToListKeranjang(keranjang);
            break;
        case "UPDATE":
            if (!keranjangMap.containsKey(split[1])) break;
            keranjang = keranjangMap.get(split[1]);
            keranjang.C = Integer.parseInt(split[2]);
            keranjang.F = Integer.parseInt(split[3]);
            keranjang.bobaCollected = hitungTotalBoba(keranjang.C, keranjang.F);
            listKeranjang.remove(keranjang);
            insertToListKeranjang(keranjang);
            break;
        default:
            break;
    }
}

private static String readQuery() {
    String query = in.next();
    switch (query) {
        case "ADD":
            return query + " " + in.next() + " " + in.next() + " " + in.next();
        case "SELL":
            return query + " " + in.next();
        case "RENAME":
            return query + " " + in.next() + " " + in.next();
        case "UPDATE":
            return query + " " + in.next() + " " + in.next() + " " + in.next();
        default:
            return query;
    }
}

// Pada main
for (int i = 0; i < H-1; i++) {
    String queryIzuri = readQuery();
    Y = in.nextInt();
    for (int j=0; j < Y; j++) {
        String custName = in.next();
        String query = readQuery();
        queryQueue.add(new Query(custName, query));
    }

    O = in.nextInt();
    out.println("Hari ke-" + (i+2) + ":");
    out.println("Permintaan yang dilayani");
    for (int j = 0; j < O; j++) {
        if (queryQueue.isEmpty()) break;
        Query query = queryQueue.poll();
        out.print(query.custName + " ");
        handleQuery(query.operation);
    }
    out.println("IZURI");
    handleQuery(queryIzuri);

    out.println("Hasil Panen");
    for (Keranjang keranjang: listKeranjang) {
        out.println(keranjang.nama + " " +keranjang.bobaCollected);
    }
    if (i < H-2) out.println();
}
```

Solusi Optimal (DP + Insertion Modifikasi)

N (input ladang) + ($M*N^2$) (input keranjang dan DP) + M (sorting keranjang) + $H(N^2 + (M+H*O))$ (query setiap hari) + (M+H*O) (print)

Big OH

$O(M*N^2 + H(N^2 + (M+H*O)))$