Pembahasan Red Blue Election Manipulation

Deskripsi Singkat

Terdapat *array* A yang menyimpan suara *voting* dari setiap pulau dengan R adalah *vote* untuk Partai Red dan B adalah *vote* untuk Partai Blue. Berikut ini merupakan peraturan *voting* yang digunakan.

- A akan dibagi menjadi 1 atau lebih kelompok pulau.
  - Sebuah kelompok pulau dapat terdiri atas **1 atau lebih pulau.**
  - Contoh:
    - 1…n ⟶ 1 kelompok pulau
    - 1…1, 2…5, 6...n ⟶ 3 kelompok pulau
- Hasil akhir *voting* adalah jumlah *vote* dari seluruh kelompok pulau. Perhitungan jumlah *vote* di dalam setiap kelompok pulau memiliki ketentuan:
  - Jika **jumlah R** $>$ **jumlah B**, hasil akhir *voting*-nya adalah **B = 0** dan **R = jumlah R + jumlah B**
  - Jika **jumlah R** $\leq$ **jumlah B**, hasil akhir *voting*-nya adalah **B = jumlah R + jumlah B** dan **R = 0**

Carilah **jumlah *vote* Partai Red maksimal** yang dapat diperoleh dengan konfigurasi pembagian kelompok pulau / *subarray* yang optimal.

Ide

Fungsi `getMaxRedVotes(start,end)` adalah sebuah fungsi rekursif yang akan mengembalikan *vote* maksimal yang akan didapatkan untuk Partai Red pada *subarray* $A[start,\ end]$ (inklusif).

Pada saat menganalisa sebuah subarray $A[start,\ end]$, maka kita diberikan dua pilihan:

1. Menjadikan *subarray* tersebut sebagai sebuah kelompok pulau.
2. Membagi *subarray* tersebut menjadi *subarray* $A[start,\ cut]$ dan $A[cut\ +\ 1,\ end]$. Lalu masing-masing *subarray* tersebut kembali dianalisa.

Opsi 1: Menjadikan *subarray* $A[start,\ end]$ sebagai satu kelompok pulau

i	…		start				end		…
A[i]	…	B	R	R	B	B	B	R	…

Opsi 2: Membagi *subarray* tersebut menjadi *subarray* $A[start,\ cut]$ dan $A[cut\ +\ 1,\ end]$. Lalu masing-masing *subarray* tersebut kembali dianalisa.

i	…		start		cut	cut+1	end		…
A[i]	…	B	R	R	B	B	B	R	…

Perhatikan bahwa masalah $A[start, end]$ artinya dapat diselesaikan dengan menyelesaikan sub-masalah $A[start, cut]$ dan $A[cut + 1, end]$. Sehingga permasalahan ini bisa diselesaikan dengan teknik *dynamic programming.*

Pada pembahasan ini, fungsi `getMaxRedVotes(start,end)` akan digunakan untuk menerapkan *dynamic programming top-down* menggunakan rekursif. Oleh karena itu, kita bisa memulai dengan mencari tahu bagaimana *base case* dari fungsi tersebut, kemudian dilanjutkan dengan mencari *recursive case* yang sesuai.

Base Case

Pada *base case*, `getMaxRedVotes` akan menjadikan *subarray* $A[start, end]$ sebagai sebuah kelompok pulau. Fungsi akan langsung mengembalikan sebuah nilai dan tidak akan melakukan rekursi ke kasus yang lebih kecil lagi.

Terdapat dua buah *base case*:

1. **Ketika hanya ada 1 pulau dalam kelompok pulau**
   Kondisi ini terjadi ketika nilai `start == end`. Saat kondisi ini tercapai, maka rekursi ke kasus yang lebih kecil tidak lagi dapat dilakukan. `getMaxRedVotes` akan menjadikan *subarray* ini sebagai sebuah kelompok pulau dan mengembalikan nilai jawaban.

   Jika *subarray* tersebut berisi 'R', maka kembalikan 1

i	…		start, end		…
A[i]	…	R	R	B	…

   Jika *subarray* tersebut berisi 'B', maka kembalikan 0

i	…		start, end		…
A[i]	…	B	B	R	…

2. **Pada saat jumlah *vote* Red lebih banyak dari jumlah *vote* Blue pada *subarray* tersebut**
   Pada saat kondisi ini tercapai, kita tidak perlu untuk melakukan rekursi ke kasus yang lebih kecil lagi karena suara maksimal bisa langsung didapatkan. `getMaxRedVotes` akan menjadikan *subarray* ini sebagai sebuah kelompok pulau dan mengembalikan nilai jawaban.

i	…		start				end		…
A[i]	…	B	R	R	B	R	R	R	…

   Misalkan `nR` adalah jumlah *vote* Red pada *subarray* tersebut dan `nB` adalah jumlah *vote* Blue. Karena `nR > nB`, maka `getMaxRedVotes(start, end) = nR + nB`

Recursive Case

Pada *recursive case*, kita akan mencari berapa *vote* maksimal yang bisa didapatkan oleh Red dengan melakukan analisa ke kasus-kasus yang lebih kecil lagi.

Misalkan kita ingin membagi *subarray* $A[start, end]$ menjadi dua bagian yang lebih kecil, yaitu $A[start, cut]$ dan $A[cut + 1, end]$, di mana `cut` adalah index pemotongan *subarray* tersebut. Karena kita ingin nilai yang dikembalikan semaksimal mungkin, maka kita harus mengecek setiap nilai `cut` yang mungkin agar `getMaxRedVotes(start,end)` dapat menghasilkan jumlah *vote* Red yang maksimal.

i	…		start		…		cut	cut+1		…		end		…
A[i]	…	R	R	R	…	B	B	R	B	…	R	R	B	…

$$getMaxRedVotes(start, end) = max_{cut}\{ getMaxRedVotes(start, cut) + getMaxRedVotes(cut + 1, end) \}$$

Berdasarkan dari uraian tersebut, berikut ini adalah *code* dari `getMaxRedVotes(start,end)`.

```
public static int getMaxRedVotes(int start, int end) {
    // Base cases
    // -- Case 1: Hanya ada 1 pulau dalam kelompok pulau ini
    if (start == end) {
        return A[start] == 'R' ? 1 : 0;
    }

    // -- Case 2: Pada kelompok pulau ini, jumlah Red lebih banyak dari Blue
    // Hitung jumlah vote Red dan Blue
    int nR = countRedVotes(start, end);
    int nB = (end - start + 1) - nR;
    if (nR > nB) {
        return nR + nB;
    }

    // Recursive Case
    int maxVotes = -1;
    for (int cut = start; cut < end; cut++) {
        int temp = getMaxRedVotes(start, cut) + getMaxRedVotes(cut + 1, end);
        maxVotes = Math.max(maxVotes, temp);
    }
    return maxVotes;
}
```

Code dari fungsi pembantu `countRedVotes(start,end)` yang memiliki kompleksitas $O(n)$ adalah sebagai berikut.

```
public static int countRedVotes(int start, int end) {
   int ans = 0;
   for (int i = start; i <= end; i++) {
       ans += A[i] == 'R' ? 1 : 0;
   }
   return ans;
}
```

Kompleksitas waktu dari solusi tersebut adalah $O(2^n)$, namun perhitungan untuk mendapatkan kompleksitas tersebut diserahkan kepada pembaca sebagai latihan :D

Solusi di atas sendiri masih belum optimal karena masih terdapat kemungkinan fungsi dengan parameter yang sama terpanggil secara berulang-ulang. Untuk menghindari hal ini, kita bisa menggunakan memoisasi agar fungsi yang sudah pernah dihitung sebelumnya tidak perlu dihitung kembali ketika dipanggil lagi.

Berikut adalah *code* solusi dengan menggunakan memoisasi. Perlu diperhatikan bahwa `memo[a][b]` merupakan sebuah *array* dua dimensi yang menyatakan hasil perhitungan dari `getMaxRedVotes(a,b)`. Pada awal program, seluruh nilai `memo[a][b]` diinisiasi dengan nilai -1. Sehingga jika belum dilakukan perhitungan atau belum dimemoisasi, nilainya adalah -1.

```
public static int getMaxRedVotes(int start, int end) {
   // Jika sudah pernah dihitung, maka hasil sudah
   // dimemoisasi dan bisa langsung dikembalikan
   if (memo[start][end] != -1)
       return memo[start][end];

   // Base cases
   // -- Case 1: Hanya ada 1 pulau dalam kelompok pulau ini
   if (start == end) {
       memo[start][end] = A[start] == 'R' ? 1 : 0;
       return memo[start][end];
   }

   // -- Case 2: Pada kelompok pulau ini, jumlah Red lebih banyak dari Blue
   // Hitung jumlah vote Red dan Blue
   int nR = countRedVotes(start, end);
```

```
    int nB = (end - start + 1) - nR;
    if (nR > nB) {
        memo[start][end] = nR + nB;
        return memo[start][end];
    }

    // Recursive Case
    int maxRedVotes = -1;
    for (int cut = start; cut < end; cut++) {
        int temp = getMaxRedVotes(start, cut) + getMaxRedVotes(cut + 1, end);
        maxRedVotes = Math.max(maxRedVotes, temp);
    }
    memo[start][end] = maxRedVotes;
    return memo[start][end];
}
```

Kompleksitas untuk mengisi `memo[N][N]` adalah $O(n^2)$ dan setiap mengisi suatu `memo[i][j]` terdapat iterasi dari `i` sampai `j` yang kompleksitasnya adalah $O(n)$. Dengan demikian, penggunaan memoisasi akan membuat kompleksitas algoritma rekursif di atas berkurang menjadi $O(n^3)$.

Ide Optimasi

Solusi yang dijelaskan di atas sudah dapat memperoleh nilai 100 pada *grader*. Namun sebenarnya solusi tersebut masih dapat dioptimasi lebih lanjut lagi. Berikut ini adalah beberapa hal yang dapat dilakukan untuk menurunkan *running time* maupun kompleksitasnya.

- Fungsi pembantu `countRedVotes(start,end)` dapat dioptimasi menjadi $O(1)$ dengan menerapkan *prefix sum* dalam perhitungan jumlah *vote* Red. Hal ini tidak akan menurunkan keseluruhan kompleksitas program, namun akan menurunkan *running time*.
- `getMaxRedVotes(start,end)` dapat memiliki kompleksitas $O(n^2)$ apabila memoisasi hanya dilakukan pada *array* satu dimensi `memo[]` dengan `memo[i]` menyatakan jumlah *vote* maksimal yang bisa didapatkan oleh Partai Red pada subarray $A[0, i]$.