COMPFEST 9

Pembahasan Coder Class

SCPC - Minggu 3

COMPFEST 9

Daftar Soal

A. Toni Vs Tere
B. Flow
C. Next Balanced Parentheses
D. Efek Domino
E. Truel
F. Jumlahin Jumlahnya Jumlah Faktor
G. Friend-Score
H. Huruf Ajaib

A. Toni Vs Tere

Tag

ad hoc

Pembahasan

Jika selisihnya kurang dari 2, maka terjadi voting ulang. Kenapa begitu? Asumsikan $Y < X$, maka $Y = X - d$. Sehingga:

$$X \geq \frac{(X+Y)}{2} + 1$$

$$X \geq \frac{2X - d}{2} + 1$$

$$X \geq X - \frac{d}{2} + 1$$

$$\frac{d}{2} \geq 1$$

$$d \geq 2$$

Oleh karena itu, hanya ada 3 kasus yang bisa terjadi, yaitu:

1. $Y - X \geq 2$, pemenangnya Toni
2. $X - Y \geq 2$, pemenangnya Tere
3. Jika bukan dua kasus di atas, voting ulang

Namun, angka yang diberikan pada masukan sangat besar. Maka, jangan lupa untuk menggunakan **64-bit *unsigned integer***. Selain itu, $X - Y$ atau $Y - X$ memiliki potensi menyebabkan *underflow*, sehingga harus dilakukan pengecekan lagi. Berikut *pseudocode* untuk menyelesaikan soal ini:

```
if(X < Y):
     if(Y - X > 1):
          print “Toni”
     else:
          print “Voting Ulang”
else:
     if(X - Y > 1):
          print “Tere”
     else:
          print “Voting Ulang”
```

Kompleksitas: $O(1)$.

B. Flow

Tag

bruteforce

Pembahasan

Soal ini dapat diselesaikan dengan *complete search.* Anggap terdapat C warna yang dinomori $1, 2, \ldots, C$. *Complete search* dilakukan dengan cara mencoba mencari jalur untuk warna 1, lalu mencoba mencari jalur untuk warna 2, dan seterusnya hingga mencari jalur untuk warna C. Jika pernah ditemukan suatu konfigurasi yang valid untuk ke-C warna, maka keluarkan SOLVABLE atau UNSOLVEABLE jika sebaliknya.

Perlu diperhatikan bahwa *complete search* yang dilakukan perlu dilengkapi dengan beberapa *pruning* sederhana, seperti tidak mencoba menempatkan suatu warna di atas petak yang telah berwarna, meletakkan warna namun tidak terhubung dengan sumber warna itu sendiri, dll.

C. Next Balanced Parantheses

Tag

greedy

Pembahasan

Pada pembahasan ini, akan digunakan *1-based indexing*. Misal buka-tutup kurung yang diberikan di soal adalah X, panjang X adalah N, dan buka-kurung tutup yang leksikografis terkecil setelah X adalah Y. Maka, observasinya adalah terdapat sebuah indeks i, sehingga i sebesar mungkin serta:

- Untuk setiap $1 \leq j < i, X_j = Y_j$
- $X_i =$ '(', dan $Y_i =$ ')'

Maka, sebenarnya persoalan ini berubah menjadi: Carilah indeks i, sehingga $X_i =$ '(', dan terdapat suatu konfigurasi buka kurung dan tutup kurung untuk setiap $j > i$, sehingga jika $X'_i =$ ')', ia menjadi konfigurasi buka-tutup kurung yang valid. Bagaimana cara memeriksanya?

Pertama, kita buat karakter '(' bernilai +1, dan ')' bernilai -1. Misal, didefinisikan P_i adalah *prefix sum* ke-i dari pemberian nilai tersebut. Maka, pada konfigurasi buka-tutup kurung yang valid, nilai $P_i \geq 0$ untuk $1 \leq i \leq N$, serta $P_N = 0$. Perhatikan apa yang terjadi apabila kita mengubah karakter X_i dari '(' menjadi ')'. Tentunya, P_i' bernilai $P_i - 2$. Apabila $P_i' \geq 0$,

ternyata pasti terdapat suatu cara menyusun $N - i$ karakter sisanya, sehingga membentuk suatu buka-tutup kurung Y, yang leksikografis terkecil lebih besar dari X. Nah, bagaimana menyusun $N - i$ karakter sisanya?

Pertama, kita catat jumlah dari nilai karakter, simpan dalam variabel yang sebut saja sum. Untuk $i - 1$ karakter pertama, tidak ada pengubahan karakter. Untuk karakter ke-i, karakter berubah dari '(' menjadi ')'. Lalu, untuk menyusun karakter ke-j, $i < j \leq N$, kita lakukan berturut-turut:

- Seandainya $sum + 1 \leq N - j$, maka karakter ke-j adalah '('. Ini karena kita masih bisa "menutup" belakangan.
- Jika tidak, maka karakter ke-j adalah ')'.

Tentunya, sambil memasang karakter, kita juga meng-*updatesum* sesuai penambahan karakter yang terjadi.

Perhatikan bahwa ada kasus dimana X adalah buka-tutup kurang yang leksikografis terbesar. Pada kasus tersebut, tentunya tidak terdapat indeks i, $1 \leq i \leq N$, sehingga $P_i \geq 2$ dan $X_i = $ '('. Jika terjadi, maka keluarkan "TIDAK ADA".

Kompleksitas: $O(N)$

D. Efek Domino

Tag

Math, big integer

Pembahasan

Mula-mula perhatikan contoh susunan untuk N = 5 berikut

1 3 5 2 4

Susunan ini merupakan susunan dengan efek domino. Apabila kita ingin meletakkan tambahan domino maka kita dapat meletakkannya pada celah yang ditandai dengan garisbawah berikut

_ 1 _ 3 _ 5 _ 2 _ 4 _

Jika domino yang akan kita tambahkan memiliki tinggi 6cm maka kita tidak dapat meletakkannya persis di depan 1 dan 2

_ 1 3 _ 5 _ 2 4 _

Untuk setiap susunan 5 domino tersebut terdapat 4 tempat yang mungkin untuk meletakkan domino setinggi 6cm. Dengan kata lain, apabila kita mendefiniskan F(N) sebagai banyaknya

susunan domino dengan tinggi 1 hingga N yang menghasilkan efek domino maka kita dapat mendefinisikannya dengan relasi rekurensi $F(N) = \left(floor(\frac{N}{2}) + 1\right) * F(N-1)$.

Kesulitan pada soal ini terdapat pada penggunaan BigInteger karena soal tidak menggunakan modulo. Peserta dapat menggunakan class BigInteger pada Java atau mengimplementasikannya sendiri dengan bahasa pemrograman lain.

Kompleksitas: $O(N * (kompleksitas\ perkalian\ bigInteger))$

E. Truel

Tag

Graph, parsing

Pembahasan

Perhatikan bahwa relasi mendominasi-didominasi dalam soal ini dapat dipetakan ke dalam sebuah *tournament graph*. Sehingga, permasalahan dalam soal ini dapat direduksi menjadi mencari tahu apakah terdapat setidaknya sebuah *cycle* dengan panjang 3 (untuk selanjutnya akan disebut sebagai *triangle*) pada graf tersebut.

Pencarian sebuah *triangle* pada graf paling cepat berjalan dalam kompleksitas $\frac{N^3}{64}$ dan itu pun masih mendapat putusan TLE jika digunakan untuk menjawab permasalahan ini.

Lemma 1

Jika terdapat sebuah *cycle* dengan panjang lebih dari tiga pada suatu *tournament graph*, maka pada graf tersebut dapat ditemukan pula *cycle* dengan panjang tepat tiga.

Bukti Lemma 1

Anggap dalam graf tersebut terdapat *cycle* terpendek C (*cycle* terpendek berarti *cycle* dengan banyak $vertex$ tersedikit). Apabila C memiliki panjang tiga, maka telah ditemukan *cycle* dengan panjang tiga. Namun, apabila $|C| > 3$, anggap terdapat tiga buah $vertexx, y, z$ pada *cycle* tersebut sehingga $x \to y \to z$ merupakan salah satu *path* dari *cycle* tersebut.

Karena graf merupakan *tournament graph*, antara x dan z pasti terdapat suatu edge/sisi. Apabila sisi tersebut berbentuk $z \to x$, maka C bukan merupakan *cycle* terpendek (karena terdapat *cycle* $x \to y \to z \to x$). Apabila sisi tersebut berbentuk $x \to z$, maka C juga bukan merupakan cycle terpendek karena sisi $x \to z$ dan sisi-sisi lainnya dalam C selain yang melewati y membentuk *cycle* lain yang panjangnya tepat lebih pendek satu *path* dari C sendiri.

Oleh karena itu, dapat dideduksi bahwa tidak mungkin terdapat suatu *cycle* terpendek C dalam suatu *tournament graph* sehingga $|C| > 3$. Perlu diperhatikan juga bahwa $|C|$ tidak mungkin satu atau dua karena *tournament graph* tidak menginginkan adanya *self-loop* atau *cycle* dalam bentuk $x \to y \to x$.

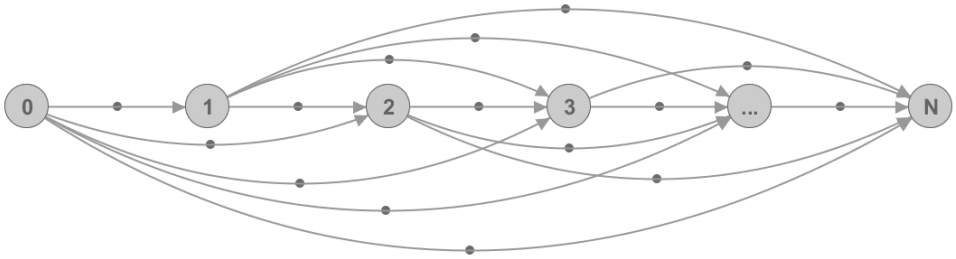
Dari **Lemma 1**, dapat ditarik kesimpulan lagi bahwa permasalahan soal ini dapat direduksi kembali menjadi mencari setidaknya satu *cycle* (panjang berapapun) dari graf yang diberikan.

Lalu, bagaiman mencari *cycle*-panjang-berapapun dari graf yang diberikan?

Permasalahan ini jawabannya dapat dicari dengan mencari kebenaran dari komplemennya: Apakah tidak terdapat *cycle* pada graf yang diberikan?

Perhatikan bahwa *tournament graph* tidak mengandung *cycle* apabila himpunan nilai-nilai *in-degree* dari *vertex*-*vertex*-nya ekivalen dengan $\{0, 1, 2, \ldots, |V| - 1\}$ dengan V adalah

himpunan *vertex* dan $|V|$ menyatakan banyaknya *vertex*, atau dengan kata lain, *tournament graph* tersebut juga merupakan *Directed Acyclic Graph (DAG)*. Hal ini benar karena untuk suatu *tournament graph*, hanya ada satu bentuk yang merupakan DAG, yaitu seperti berikut:



Atau dengan kata lain, untuk suatu *tournament graph* yang asiklis dengan banyaknya *vertex*$|V|$:

- Terdapat tepat satu *vertex*$v_0$yang memiliki *in-degree* 0: untuk semua vertex $v_i \neq v_0$, terdapat sisi $v_i \rightarrow v_0$
- Terdapat tepat satu *vertex*$v_1$yang memiliki *in-degree* 1: untuk semua vertex $v_i \neq v_1$,$v_i \neq v_0$, terdapat sisi $v_i \rightarrow v_1$
- Dan seterusnya, hingga *vertex*v_n yang memiliki *in-degree* sebesar $|V| - 1$.

Kompleksitas: $O(V^2)$

F. Jumlahin Jumlahnya Jumlahan Faktor

Tag

math

Pembahasan

Kita perlu merubah rumus yang diberikan di soal. Perubahannya adalah sebagai berikut:

$$\sum_{i=1}^{A}\sum_{j=1}^{B}\sum_{d|\gcd(i,j)} d$$

$$= \sum_{d=1}^{A}\sum_{i=1}^{\lfloor \frac{A}{d} \rfloor}\sum_{j=1}^{\lfloor \frac{B}{d} \rfloor} d$$

$$= \sum_{d=1}^{A} d \sum_{i=1}^{\lfloor \frac{A}{d} \rfloor}\sum_{j=1}^{\lfloor \frac{B}{d} \rfloor} 1$$

Mengapa hal ini benar? Intuitifnya, $\sum_{i=1}^{\lfloor \frac{A}{d} \rfloor}\sum_{j=1}^{\lfloor \frac{B}{d} \rfloor} 1$ merupakan banyaknya pasangan yang berbentuk (di, dj). Tentunya, FPB dari keduanya merupakan kelipatan dari d, sehingga

d merupakan faktor dari FPB pasangan tersebut. Jika diamati, kita bisa mengganti formula itu menjadi $\left\lfloor \frac{A}{d} \right\rfloor \cdot \left\lfloor \frac{B}{d} \right\rfloor$, sehingga rumus akhirnya menjadi:

$$\sum_{d=1}^{A} d \cdot \left\lfloor \frac{A}{d} \right\rfloor \cdot \left\lfloor \frac{B}{d} \right\rfloor$$

Kompleksitas: $O(A)$

G. Friend-Score

Tag

Disjoint set, small-to-large

Pembahasan

Perhatikan bahwa Friend-Score dari perusahaan gabungan tempat A dan B bekerja dapat dihitung sebagai gabungan nilai-nilai berikut:

1. Friend-Score perusahaan tempat A sebelumnya bekerja
2. Friend-Score perusahaan tempat B sebelumnya bekerja
3. Banyak pertemanan yang baru saja terjadi setelah penggabungan

Untuk mencari tahu perusahaan dan Friend-Score perusahaan tempat X (untuk sembarang X) bekerja, kita dapat memanfaatkan struktur data Disjoint Set. Selanjutnya, bagian yang cukup sulit adalah menghitung nilai ketiga, yaitu banyak pertemanan yang baru saja terjadi.

Untuk mencarinya, anggap setiap pertemanan dinomori dari 1 sampai M. Kita buat array of set, sebut saja *pendingFriend[]*, yang mana *pendingFriend[X]* menyatakan pertemanan-pertemanan pada perusahaan X yang temannya berada pada perusahaan lain. Maka, ketika menggabungkan perusahaan, misal tempat X dan Y bekerja, tentunya yang terjadi adalah sebagai berikut:

1. Periksa seluruh pertemanan pada *pendingFriend[X]*, Apabila terdapat pertemanan yang terdapat pada *pendingFriend[Y]*, hapus pertemanan pada kedua set tersebut, dan ini menambah satu pada nilai Friend-Score.

2. Pindahkan seluruh pertemanan sisanya pada *pendingFriend[X]* ke *pendingFriend[Y]*.

Apabila dilakukan secara naif, hal ini tentunya berjalan sebesar $O(M^2 logM)$, yaitu $O(M^2)$ pemindahan dan dikali $O(logM)$ karena kita menggunakan set. Namun, ada satu optimisasi mudah yang bisa mempercepat: kita memproses *pendingFriend[]* yang berukuran lebih kecil, lalu memindahkannya ke yang berukuran lebih besar. (Dengan kata lain, bisa saja kita memindahkan dari Y ke X.) Walaupun terlihat mudah dan tidak berpengaruh, ternyata hal ini dapat mengurangi kompleksitas menjadi $O(Mlog^2 M)$! Pembuktiannya diserahkan kepada pembaca sebagai latihan. Sebagai petunjuk, dengan cara tersebut, suatu pertemanan pasti dipindahkan tidak lebih dari $O(logM)$ kali.

Sebagai catatan, terdapat solusi lain yang menggunakan teknik *parallel binary search*, yang memiliki kompleksitas kurang lebih sama, namun konstantanya lebih ringan sehingga lebih cepat, namun *code*-nya lebih panjang.

Kompleksitas: $O(N + Q + M\ log^2 N)$

H. Huruf Ajaib

Tag

String, greedy

Pembahasan

Kita dapat mengerjakan soal ini dengan *greedy* terhadap karakter terkecil pada string S. Kita misalkan c sebagai karakter terkecil pada string S dan count(c) sebagai jumlah karakter c pada string S. Perhatikan bahwa kita harus menemukan subsekuens dengan panjang K yang leksikografis terkecil, tentunya semakin banyak karakter c dalam subsekuens tersebut membuat subsekuens semakin kecil.

Kita dapat melakukan *binary search* untuk mencari berapa banyak karakter c maksimal yang bisa kita letakkan pada awal subsekuens yang memiliki panjang minimal K. Terdapat count(c) * M buah karakter c yang dapat kita gunakan. Selanjutnya kita misalkan pos(i) sebagai posisi karakter c yang ke-i pada string R dan len(S) sebagai panjang string S. Perhatikan bahwa apabila kita meletakkann Y buah karakter c di awal, maka kita dapat membentuk subsekuens sepanjang $(Y + len(R) - pos(Y) - 1)$ karakter.

Setelah mendapatkan jumlah karakter c maksimal yang bisa kita letakkan di awal maka kita dapat membentuk subsekuens dengan panjang K yang memiiliki 3 bagian misal bagian p1, p2 dan p3.

p1p2p3

Bagian p1 terdiri atas semua karakter c yang terletak di awal subsekuens, bagian p3 merupakan substring dari karakter c pertama yang tidak termasuk bagian p1 sepanjang mungkin selama subsekuens belum lebih panjang dari K. Bagian p2 adalah bagian karakter tambahan yaitu subsekuens leksikografis terkecil dari substring pos(len(p1)) hingga pos(len(p1) + 1) dengan panjang K - len(p1) - len(p3).

Dengan membuat subsekuens dengan panjang K dan terdiri atas 3 bagian tersebut maka terdapat 3 kasus yang mungkin untuk setiap X.

1. X berada di bagian p1, maka output sudah pasti karakter c

2. X berada di bagian p2, maka kita perlu mencari p2. Karena p2 hanya memiliki panjang maksimal N karakter, untuk mencari subsekuens terkecil dengan panjang K karakter kita dapat melakukan *depth first search* (DFS) untuk setiap subsekuens dengan mengutamakan karakter-karakter terkecil, sehingga kita dapat berhenti DFS setelah memproses X karakter.

3. X berada di bagian p3, perhatikan bahwa p3 hanya merupakan substring R dari pos(len(p1) +1) maka karakter X dapat dicari dengan mengambil karakter ke-$(X - len(p1) - len(p2) + 1 + pos(len(p1) + 1)$

Terdapat beberapa kasus *tricky* yaitu kasus-kasus dimana kita dapat membentuk subsekuens yang hanya terdiri dari 1 atau 2 bagian saja, detail diserahkan kepada pembaca :).

Kompleksitas: $O(\log R + N)$