BUKLET PEMBAHASAN SOAL

FINAL PEMROGRAMAN GEMASTIK 9
29 Oktober 2016

Soal-Soal

Kode	Judul	Penulis
A	Foto Wisuda	Ashar Fuadi
B	Membangun Gapura	Alham Fikri Aji
C	Balon Warna-Warni	Steven Halim
D	Papan Selancar	William Gozali
E	Irigasi Desa	Steven Halim
F	Permainan Jenius	Suhendry Effendy
G	Pohon Merah-Hitam	Ashar Fuadi
H	Panitia Gemastik	Irvan Jahja
I	Astik Sort	Suhendry Effendy
J	Tudung Cembung	William Gozali

A. Foto Wisuda

Oleh: Ashar Fuadi
Tema: *ad hoc*

Misalkan S adalah himpunan dari semua nama orang yang muncul pada masukan. Tentunya, jawabannya pasti setidaknya adalah |S|. Namun, pada kasus tertentu, |S| orang tidaklah cukup. Kasus tersebut adalah apabila terdapat foto “dibantu” yang terdiri atas semua orang pada S. Dalam hal ini, jawabannya adalah |S| + 1 karena harus terdapat satu orang lain di luar S yang memotret mereka.

Soal ini mudah, namun *tricky*. Kesalahan umum adalah anggapan bahwa apabila ada foto “dibantu”, maka jawabannya pasti adalah |S| + 1, padahal bisa saja yang memotret foto tersebut adalah orang yang termasuk dalam S.

B. Membangun Gapura

Oleh: Alham Fikri Aji
Tema: *meet-in-the-middle*, *binary search*

Setiap potongan besi dapat menjadi tiang kiri, kanan, atau penutup. Oleh karena itu, terdapat total 3^N kemungkinan penyusunan. Apabila semua kemungkinan dicoba, pasti akan menghasilkan TLE. Bagaimana cara mempercepatnya?

Kita bisa menggunakan teknik *meet-in-the-middle*. Bagi potongan-potongan besi menjadi dua buah bagian:

- A: terdiri atas potongan-potongan nomor 1 sampai dengan N/2
- B: terdiri atas potongan-potongan nomor N/2 + 1 sampai N

Pada setiap bagian, coba semua kemungkinan penyusunan. Suatu penyusunan pada bagian A dapat dinyatakan dengan triplet (a_kiri, a_atas, a_kanan) yang menyatakan panjang tiang kiri, penutup, dan kanan secara berurutan. Triplet (b_kiri, b_atas, b_kanan) digunakan untuk menyatakan hal yang serupa pada bagian B. Kemudian, suatu pasang penyusunan dari A dan B dapat membentuk gapura apabila:

a_kiri + b_kiri = a_kanan + b_kanan

atau:

a_kiri - a_kanan = b_kanan - b_kiri

Artinya, untuk setiap penyusunan (a_kiri, a_atas, a_kanan), kita bisa menemukan pasangan penyusunan pada B dengan bantuan rumus di atas. Caranya adalah dengan mengumpulkan semua penyusunan B dengan nilai (b_kanan - b_kiri) yang sama.

Apabila terdapat lebih dari satu penyusunan dengan nilai (b_kanan - b_kiri) yang sama, kita butuh penyusunan yang menghasilkan nilai (a_kiri + b_kiri) * (a_atas + b_atas) terbesar. Ini bisa dicari dengan menggunakan *binary search*. Detilnya diserahkan kepada pembaca,

dengan petunjuk sebagai berikut: jika terdapat segiempat-segiempat dengan keliling yang sama, segiempat dengan luas terbesar adalah yang bentuknya sedekat mungkin dengan persegi.

Kompleksitas waktunya turun dari $O(3^N)$ menjadi $O(3^{N/2} \cdot \lg 3^{N/2})$ atau $O(N \cdot 3^{N/2})$.

C. Balon Warna-Warni

Oleh: Steven Halim
Tema: *ad hoc, (very) simple math*

Jawabannya sudah jelas adalah N! mod M. Namun, N bisa bernilai sangat besar. Bagaimana cara mengakalinya?

Observasi kuncinya adalah, jika $N \geq M$, maka jawabannya pasti 0, karena terdapat faktor M dalam N!. Selebihnya, jika $N < M$, maka kita tinggal menghitung faktorial secara manual, yang cukup dalam batas waktu karena $M \leq 100.000$.

Kompleksitas waktunya adalah O(M).

D. Papan Selancar

Oleh: William Gozali
Tema: *ad hoc data structure, segment tree*

Untuk kemudahan pembahasan, kita anggap angin hanya bertiup ke arah kiri. Sehingga, setiap taburan bisa dinyatakan sebagai sebuah “trapesium”. Angin yang bertiup ke arah kanan bisa diselesaikan dengan cara yang mirip.

Kita siapkan dua buah *array of vector* yang menyatakan persebaran trapesium-trapesium pada papan selancar sebagai berikut:

- kiri[i]: berisi tinggi-tinggi sisi kiri trapesium yang sisi kirinya pada posisi i.
- kanan[i]: berisi tinggi-tinggi sisi kanan trapesium yang sisi kanannya pada posisi i.

Untuk setiap taburan, kita mendapatkan sebuah trapesium. Perbarui *array* kiri[] dan kanan[] sesuai dengan ukuran trapesium tersebut.

Setelah semua taburan selesai, ketinggian pasir untuk setiap posisi bisa dihitung sebagai berikut. Kita mulai dari posisi 1, lalu berjalan ke kanan sampai posisi N. Selama proses penghitungan ini, kita memiliki dua buah variabel sebagai berikut:

- banyak: banyaknya trapesium yang sedang kita punya saat ini
- tinggi: ketinggian pasir saat ini, yakni total tinggi dari setiap trapesium yang sedang kita punya, pada koordinat X yang bersangkutan

Misalkan kita memiliki nilai-nilai variabel banyak dan tinggi untuk posisi i-1. Saat maju ke posisi i, kita perbarui variabel banyak dan tinggi sebagai berikut:

- tinggi += total kiri[i]

- cetak tinggi
- tinggi -= total kanan[i]
- banyak += | kiri[i] |
- banyak -= | kanan[i] |
- tinggi += banyak

Kompleksitasnya adalah O(N + M).

Soal ini juga bisa diselesaikan dengan *segment tree* dengan ide yang mirip, yang merupakan solusi alternatif juri.

E. Irigasi Desa

Oleh: Steven Halim
Tema: *binary search, greedy*

Kita bisa melakukan *binary search* terhadap jawabannya.

Misalkan kita ingin menguji apakah waktu T jam mungkin sebagai jawaban. Ini bisa dilakukan dengan *greedy*. Observasinya adalah, setiap *leaf* pada pohon haruslah terairi dalam T jam. Oleh karena itu, kita perlu memasang pompa pada semua *node* yang berjarak T dari setiap *leaf*.

Setelah itu, kita simulasikan pengairan selama T jam. Lalu, kita "potong" semua *node* yang terairi. Maka, kita akan mendapat sebuah pohon baru. Lakukan hal yang sama sampai semua *node* habis (terairi). Apabila kita membutuhkan lebih dari K pompa untuk melakukan hal ini, maka nilai T tidak mungkin sebagai jawaban, dan kita perlu mencoba nilai yang lebih tinggi dari T. Selebihnya, T mungkin sebagai jawaban, dan kita dapat mencoba nilai yang lebih rendah dari T.

Detil simulasi pengairan untuk pengujian nilai T diserahkan kepada pembaca. Kompleksitasnya harus O(N), agar secara keseluruhan kompleksitasnya adalah O(N log N).

F. Permainan Jenius

Oleh: Suhendry Effendy
Tema: *game theory, grundy number, nim game*

Setiap bilangan bisa dinyatakan secara unik sebagai perkalian dari bilangan-bilangan prima. Kita bisa menganggap setiap bilangan sebagai sebuah "tumpukan" yang terdiri atas bilangan-bilangan prima. Dengan demikian, setiap langkah dapat dianggap sebagai pengambilan satu atau lebih bilangan prima dari sebuah tumpukan.

Jika kita perhatikan, permainan jenius yang dideskripsikan di atas tidak lain adalah sebuah *nim game*! *Grundy number* dari permainan ini adalah XOR dari dari *grundy number* setiap tumpukan, yang merupakan banyaknya faktor prima pada bilangan yang dinyatakan oleh

setiap tumpukan. Jika XOR-nya adalah 0, maka Anda bisa menang. Sebaliknya, Anda pasti kalah.

G. Pohon Merah-Hitam

Oleh: Ashar Fuadi
Tema: *dynamic programming on tree*

Tentu saja, soal ini bukan tentang *red-black tree* :)

Pertama-tama, bentuk ulang graf pada masukan agar menjadi pohon yang terakar pada *node* 1. Misalkan warna pada suatu *node* u adalah c, dan warna pada *parent*-nya adalah *p*. Observasinya adalah sebagai berikut:

- Jika c = p, maka **semua** *node* anak dari u tidak boleh berwarna c. Jika ada anak dari u yang berwarna c, maka akan terdapat jalur terdiri dari 3 simpul berwarna sama (c): *parent* u -> u -> anak u, yang mana tidak diperbolehkan.
- Jika c != p, maka banyaknya anak-anak dari *u* yang berwarna c harus **paling banyak satu**. Jika terdapat dua atau lebih, maka akan terdapat jalur terdiri dari 3 simpul berwarna sama (c): anak u -> u -> anak -> u, yang mana tidak diperbolehkan.

Maka, soal ini bisa diselesaikan dengan *dynamic programming*. Misalkan dp[u][c][p] adalah simpul merah minimum untuk mewarnai subpohon yang berakar pada u, dengan warna u adalah c dan warna *parent*-nya adalah p. Untuk kemudahan, anggap c=0 berarti hitam, dan c=1 berarti merah. Nilai dp[u][c][p] dapat dinyatakan sebagai berikut:

dp[u][c][p] = c + min (
- sum{dp[v][1-c][c] | v anak u}
- min{S - dp[v][1-c][c] + dp[v][c][c] | v anak u}, dengan S = sum{dp[v][1-c][c] | v anak u}

)

Jawabannya adalah min(dp[1][0][1], dp[1][1][0]). Kompleksitas waktunya adalah O(N).

H. Panitia Gemastik

Oleh: Irvan Jahja
Tema: *dynamic programming, probability, hypergeometric distribution*

Terdapat N+1 kemungkinan nilai X: 0, 1, 2, …, N. Misalkan cocok[x] = peluang terjadinya X=x. Maka, sesuai definisi, nilai harapan dari Indeks Jaccard adalah:

sum{ cocok[x] * x / (2N - x) | x in {0, 1, 2, …, N} }

Masalahnya, bagaimana menghitung nilai-nilai dari cocok[x]?

Ini bisa dihitung dengan menggunakan *dynamic programming*. Pertama-tama, kita butuh menghitung *array* kaos[i] = banyaknya kaos berwarna i.

Misalkan dp[i][k][x] adalah peluang terdapat x kaos yang cocok, jika kita sudah mengambil k kaos dari semua kaos yang berwarna 1 hingga (i-1). Pada mulanya, kita tahu bahwa dp[1][0][0] = 1.0 (yakni merupakan kondisi awal kita). Misalkan kita sudah mengetahui nilai dari dp[i][k][x]. Sekarang, kita ingin "memproses" warna ke-i. Kita akan mencoba mengambil sejumlah kaos berwarna ke-i, lalu memperbarui nilai dari dp[i+1][k+?][x+?] yang sesuai.

Pada saat memproses dp[i][k][x], misalkan banyaknya kaos yang kita "miliki" dinyatakan dengan M = P[i] + P[i+1] + … + P[N+1]. Lalu, kita memiliki fakta-fakta berikut:

- kita memiliki M kaos,
- P[i] di antaranya berwarna i,
- kita butuh mengambil m = N - k kaos,
- kita ingin mengetahui berapa peluang terambilnya k' kaos berwarna i, untuk k' = 0 hingga min(m, P[i]).

Misalkan terambil[k'] = peluang terambilnya k' kaos berwarna i. Maka, kita perbarui nilai dari:

dp[i+1][k+k'][x+x'] += terambil[k'] * dp[i][j][k]

dengan x' = min(kaos[i], k').

Sekarang, bagaimana menghitung nilai dari terambil[k']? Rupanya, perhitungan peluang ini merupakan aplikasi dari *hypergeometric distribution*. Singkatnya,

terambil[k'] = C(P[i], k') * C(M - P[i], m - k') / C(M, m).

Detil dari perhitungan kombinasi C(n, k) dengan nilai-nilai n dan k bisa mencapai 10^7 (sesuai dengan batasan pada soal) diserahkan kepada pembaca.

Akhirnya, bisa dihitung cocok[x] = dp[N+2][N][x]. Dengan ini, nilai harapan dari Indeks Jaccard dapat diketahui.

I. Astik Sort

Oleh: Suhendry Effendy
Tema: *inversion*

Kita definisikan *inversi* sebagai banyaknya pasangan (i, j) yang memenuhi i < j dan S[i] > S[j].

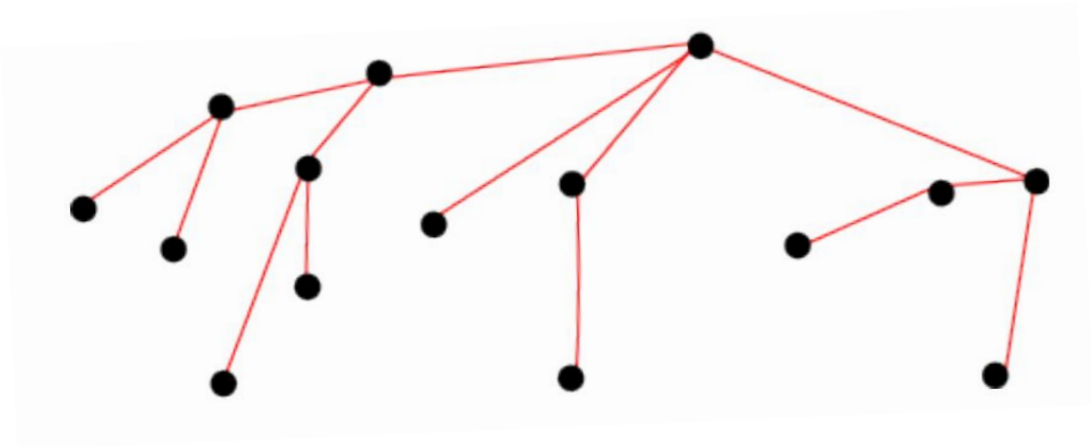
Perhatikan bahwa setiap rotasi akan mengubah inversi sebanyak 0, +2, atau -2. Karena barisan terurut 1, 2, 3, …, N memiliki inversi 0, maka apabila S bisa diurutkan, inversi S haruslah genap. Hal ini berlaku sebaliknya: apabila inversi S genap, maka S pasti bisa diurutkan. Buktinya diserahkan kepada pembaca.

Dengan demikian, kita tinggal mengecek apakah inversi S genap atau ganjil. Perhitungan inversi yang efisien dalam O(N log N) bisa dilakukan dengan *merge sort* atau Fenwick Tree (BIT). Ini adalah persoalan klasik; silakan merujuk pada internet untuk mengetahui detil implementasinya.

J. Tudung Cembung

Oleh: William Gozali
Tema: *convex hull, LCA-like data structure*

Apabila untuk setiap titik kita gambar *upper hull* dari semua titik di sebelah kanan titik tersebut dan titik tersebut itu sendiri, maka akan didapat gambar yang mirip seperti gambar di bawah:



Perhatikan bahwa gambar di atas sesungguhnya adalah struktur data pohon! Struktur pohon di atas dapat dibuat dengan menjalankan sedikit modifikasi dari algoritma *Andrew's monotone chain convex hull algorithm*, dengan mengambil hanya *upper hull*-nya (petunjuk: ketika kita melakukan *pop* pada *stack*, maka kita akan menemukan pasangan *node* dan *parent*-nya).

Setelah struktur pohon di atas, kita juga membutuhkan *suffix sum* dari luas *upper hull* yang diproyeksikan ke sumbu X, untuk setiap titik awal yang ada. Dengan demikian, untuk setiap pertanyaan X dan K, misalkan A adalah titik pertama yang ada di kanan atau sama dengan X, dan B adalah titik ke-K pada *upper hull* yang terbentuk dari A ke kanan, maka jawabannya adalah:

suffix_sum[A] - suffix_sum[B] - luas trapesium dari A dan B diproyeksikan ke sumbu X

Permasalahannya sekarang adalah bagaimana menentukan titik ke-K pada *upper hull* dengan cepat. Perhatikan bahwa titik ke-K sesungguhnya adalah *parent* ke-K dari A. Kita bisa menggunakan teknik yang mirip dengan LCA, yakni:

Buat *array* par[u][k] = *parent* ke-2^k dari u. Rekurensnya adalah:

- par[u][0] = *parent* dari u
- par[u][k] = par[par[u][k-1]][k-1]

Maka, kita bisa menentukan parent ke-K dari A dalam O(log N) dengan melihat representasi biner dari K. Misalnya, parent ke-11 dari A adalah par[par[par[A][3]][1]][0].

Kompleksitas waktu totalnya adalah O((N + Q) log N).