Gatavošanās LIO

Artis Vijups, Lunda, 2023

2023. gada 22. februāra versija
Ja plāno dalīties ar šo materiālu, lūdzu mani par to informēt - artis@vijups.eu

Saturs

1. Implementācijas idejas

1.1. Datu tipi

1.1.1. Programmas sākums

Olimpiādēs būtu jāizmanto nevis `int` un `double`, bet gan `long long` un `long double`. Šo var darīt vienmēr, jo atmiņas ierobežojumi ļoti reti izraisa problēmas. Pie tam, neizmantojot `long long` un `long double`, ir iespējams pavisam viegli zaudēt punktus.

Īsāka pieraksta nolūkos, pieņemam, ka visas programmas, kas tālāk sekos, satur šīs rindas:

```
-7  #include <bits/stdc++.h> // iekļaujam visu
-6
-5  using namespace std;
-4
-3  typedef long long ll;
-2  typedef long double ld;  // šo var izlaist, ja double nevajag
-1
```

Pie tam, vienmēr pieņemsim, ka rinda ar kārtas numuru 0 satur `int main()`. Attiecīgi, ja rindai ir negatīvs kārtas numurs, tā atrodas pirms `int main()`, citādi tā atrodas iekš `int main()`.

Tad mēs varam ieviest dažus mainīgos pavisam īsā veidā:

```
1  ll a = 743; // vesels skaitlis
2  ld b = -0.132923; // decimālskaitlis
```

Olimpiādes sistēmā var atrast saiti "Standarta ievads un izvads" ar šādām rindām:

```
1  ios::sync_with_stdio(0);
2  cin.tie(NULL);
```

Tās vērts izsaukt `int main()` sākumā, jo īpaši, ja uzdevumā ir jāielasa daudz datu.

1.1.2. Vektori un simbolu virknes

Tips `vector` ir saraksts ar maināmu elementu skaitu. Lūk, piemērs, kura beigās tiek izvadīti visi vektora elementi:

```
1  vector<ld> v(3);
2  v = {-34.3, 2.334, 83.1};
3  for(auto el:v) cout << el << " ";
```

Mēs varam pievienot elementu vektora beigām ar `v.push_back()`, un izņemt pēdējo elementu ar `v.pop_back()`. Noteiktā brīža elementu skaitu var iegūt ar `v.size()`.

Simbolu virkne `string` pēc savas būtības ir līdzīga `vector<char>`. Tai darbojas visas iepriekšminētās funkcijas, kā arī:

```
4  ll a = stoll("743"); // pārvērš string par ll
5  ld b = stold("-0.132923"); // pārvērš string par ld
6  string s = to_string(b); // pārvērš ll vai ld par string
7  cout << s.substr(2, 4) << "\n";
```

Funkcija `s.substr(i, n)` atgriež tādu `string`, kas sākas ar `s[i]` un ir n simbolus gara. Tātad, `s.substr(2, 4)` atgriež `".132"`, jo `s[2]=='.'`, un virknei jābūt 4 simbolus garai.

1.1.3. Vārdnīcas

Viens ļoti vērtīgs datu tips ir `map`. Aplūkosim šādu piemēru:

```
1  map<string, ld> m;
2  m["Aija"]=7.3;
3  m["Zane"]=-2;
```

Šī vārdnīca ļauj jebkurai `string` vērtībai piešķirt atbilstošu `ld` vērtību. Mēs varam uzzināt, vai `string s` ir vārdnīcā, ar `m.count(s)`, un mēs varam to izvadīt ar `m[s]`.

Nedaudz jāuzmanās ar to, ka, ja mēs mēģināsim piekļūt kādai vērtībai, kas neeksistē, tad `map` to automātiski izveidos. Piemēram, var aplūkot šīs rindas:

```
4  cout << m["Rita"] << "\n";
5  cout << m.count("Rita") << "\n";
```

Mēs it kā neesam iestatījuši vērtību priekš `m["Rita"]`, tāpēc gribētos domāt, ka pēdējā rinda izvadīs 0. Taču tā izvada 1, jo mēs iepriekš esam mēģinājuši piekļūt šim elementam.

Visbeidzot, `map` var definēt arī ar citiem datu tipiem. Lūk, daži interesanti varianti:

```
1  map<char, ll> m1;
2  m1['a'] = 2;
3  cout << m1['a'] << " ";
4
5  map<string, vector<ll> > m2;
6  m2["teksts"].push_back(4);
7  for(auto el:m2["teksts"]) cout << el << " ";
8
9  map<pair<ll, ll>, bool> m3;
10 m3[{7,4}] = true;
11 cout << m3[{7,4}] << "\n";
```

Šī programma attiecīgi izvada 2 4 1.

1.1.4. Rindas

Mums svarīgs ir viens rindu tips - `priority_queue`.

```
priority_queue<ll> Q;
Q.push(5);
Q.push(8);
Q.push(2);
```

Šāda rinda automātiski visus elementus saglabā tādā secībā, lai "augšējais" elements būtu vislielākais. Taču piekļūt var tikai šim vienam elementam - to var nolasīt ar `Q.top()`, un to var izņemt ar `Q.pop()`. Lūk, piemērs:

```
cout << Q.top() << " ";
Q.pop();
cout << Q.top() << "\n";
```

Šī programma izvada 8 5.

Mums ir iespējams definēt arī citu datu tipu rindas, kā `priority_queue<string>` vai arī `priority_queue<pair<ll, ll> >`.

Nedaudz sarežģītāk ir definēt rindu, kuras "augšējais" elements ir nevis vislielākais elements, bet vismazākais elements.

```
priority_queue<ll, vector<ll>, greater<ll> > Q;
Q.push(7);
Q.push(3);
cout << Q.top() << "\n";
```

Šī programma izvada 3.

1.1.5. Uzdevumi

Izmanto šos datu tipus, lai atrisinātu uzdevumus 1703B un 652B mājaslapā Codeforces.

1.2. Binārā meklēšana

1.2.1. Motivācija

Teiksim, ka mums ir dots augošā secībā sakārtots vektors ar skaitļiem:

```
vector<ll> s(15);
s = {10,11,12,14,17,18,19,22,24,28,33,37,39,40,43};
```

Kurā pozīcijā atrodas skaitlis 18? Protams, mēs varam pamēģināt kaut ko naivu:

```
ll i=0;
while(s[i]!=18) i++;
cout << i << "\n";
```

Šis ir ātri, jo, tā kā `s[5]==18`, atbilde ir 5. Taču, ja mums būtu saraksts ar tūkstošiem pozitīvu un negatīvu skaitļu, šis pēkšņi kļūtu par ļoti lēnu risinājumu.

1.2.2. Ideja

Vispirms, aplūkosim vidējo elementu.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	11	12	14	17	18	19	22	24	28	33	37	39	40	43

Vidējais elements ir lielāks par 18, tātad 18 atrodas kaut kur pa kreisi no vidējā elementa.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	11	12	14	17	18	19	22	24	28	33	37	39	40	43

Aplūkojam atlikušās daļas vidējo elementu.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	11	12	14	17	18	19	22	24	28	33	37	39	40	43

Šis elements ir mazāks par 18, tātad 18 jābūt kaut kur pa labi no tā.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	11	12	14	17	18	19	22	24	28	33	37	39	40	43

Atkal aplūkojam atlikušās daļas vidējo elementu.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	11	12	14	17	18	19	22	24	28	33	37	39	40	43

Mēs esam atraduši skaitli 18.

Kā redzams, katrā solī mūsu meklēšanas apgabals saruka vismaz uz pusi. Tātad, piemēram, sarakstam ar 1000 elementiem pietiktu ar $\log_2 1000 \leq 10$ soļiem, lai atrastu vajadzīgo elementu. (Ar $\log_2$ apzīmē, cik reizes skaitlis jādala ar 2, lai paliktu 1.)

Ja mēs gribam šo ideju implementēt koda formā, to var panākt, piemēram, šādi:

```
6  ll k=0;  // meklēšanas apgabala kreisais indekss
7  ll l=14; // meklēšanas apgabala labais indekss
8  ll v=7;  // meklēšanas apgabala vidējā pozīcija
9  while(s[v]!=18){
10     if(s[v]>18) l=v-1;
11     else k=v+1;
12     v=(k+l)/2;
13 }
14 cout << v << "\n";
```

1.2.3. Iebūvētās funkcijas

Ir trīs ar bināro meklēšanu saistītas iebūvētas funkcijas. Jāatzīst, ka visas šīs funkcijas izskatās ļoti, ļoti dīvaini. Mēs gan par to neuztrauksimies - mūsu nolūkos tas īsti nav svarīgi, kāpēc tās ir tik dīvainas pēc izskata, kamēr tās atgriež to, ko mēs gribam.

Piemēra pēc, izmantosim šīs funkcijas ar to pašu vektoru `s`:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	11	12	14	17	18	19	22	24	28	33	37	39	40	43

Pirmā no funkcijām ir `binary_search`:

```
15 cout << binary_search(s.begin(), s.end(), 18) << " ";
16 cout << binary_search(s.begin(), s.end(), 20) << "\n";
```

Šī funkcija izmanto binārās meklēšanas ideju, lai pārbaudītu, vai skaitlis atrodas sarakstā. Attiecīgi, šīs rindas izvada 1 0, jo 18 ir sarakstā, bet 20 sarakstā nav.

Otrā funkcija ir ar nosaukumu `lower_bound`:

```
17 cout << lower_bound(s.begin(), s.end(), 18) - s.begin() << " ";
18 cout << lower_bound(s.begin(), s.end(), 20) - s.begin() << "\n";
```

Tā ļauj atrast pirmo elementu, kura vērtība ir *vismaz* meklētā vērtība. Šīs rindas izvada 5 7, jo pirmais elements, kura vērtība ir vismaz 18, ir `s[5]`, bet pirmais elements, kura vērtība ir vismaz 20, ir `s[7]`.

Kā izrādās, `lower_bound` ir gandrīz identiska funkcija tam kodam, ko mēs izveidojām paši. Mums būtu jānomaina vien divas rindiņas:

```
9  while(k<l){
10     if(s[v]>=18) l=v;
```

Ieteicams patstāvīgi izspēlēt situāciju ar 18 un 20, lai saprastu, kāpēc šī izmaiņa darbojas.

Trešā un pēdējā funkcija ir upper_bound:

```
19  cout << upper_bound(s.begin(), s.end(), 18) - s.begin() << " ";
20  cout << upper_bound(s.begin(), s.end(), 20) - s.begin() << "\n";
```

Šī funkcija ļauj atrast pirmo elementu, kura vērtība *pārsniedz* meklēto vērtību. Šīs rindas izvada 6 7, jo pirmais elements, kura vērtība pārsniedz 18, ir `s[6]`, bet pirmais elements, kura vērtība pārsniedz 20, ir `s[7]`.

1.2.4. Uzdevums

Atrisini uzdevumu 1613C mājaslapā Codeforces.

1.3. Kārtošana un sarežģītība

1.3.1. Motivācija

Teiksim, ka mums ir dots vecais vektors s, bet nejaušā secībā:

```
1  vector<ll> s(15);
2  s = {17,39,12,14,28,19,33,22,18,40,37,10,43,11,24};
```

Cik skaitļi vektorā ir mazāki par 18?

Kā jau iepriekš, mēs varētu naivi iet cauri visam sarakstam un saskaitīt, cik elementi ir mazāki par 18.

1.3.2. Ideja

Ja mēs varētu sakārtot vektoru augošā secībā, tad ar `lower_bound` varētu atrast pirmo elementu, kura vērtība ir vismaz 18. Tad visi elementi pirms šī indeksa ir mazāki par 18. Mēs atrastu indeksu 5, līdz ar to ir 5 skaitļi (no indeksa 0 līdz 4), kas ir mazāki par 18.

Lai sakārtotu vektoru, pastāv iebūvēta funkcija, kuru mēs vienmēr varēsim pielāgot un izmantot, tāpēc kārtošanas algoritmu paši mēs neveidosim.

```
3  sort(s.begin(), s.end()); // sakārtošanas funkcija
```

Šis kods vektoru s sakārto augošā secībā. Tad mēs varētu pabeigt programmu ar:

```
4  cout << lower_bound(s.begin(), s.end(), 18) - s.begin() << "\n";
```

1.3.3. Sarežģītība

Vai mūsu ideja tiešām ir labāka par naivo variantu?

Teiksim, ka mums ir saraksts ar 1000 skaitļiem, un mēs vēlamies uzzināt, cik no šiem skaitļiem ir mazāki par K, simts dažādām K vērtībām.

Naivajā variantā mēs katrai no 100 vērtībām veiktu 1000 salīdzinājumus, līdz ar to mums tas prasītu 100000 operācijas.

Mūsu ideja vispirms liek sakārtot 1000 skaitļu vektoru. Izrādās, ka mūsu izmantotās `sort` funkcijas "sarežģītība" - tas, cik operācijas funkcija veic, lai sakārtotu 1000 skaitļu vektoru - ir 1000 x $\log_2 1000 \leq$ 1000 x 10 = 10000.

Visbeidzot, mēs izsaucam `lower_bound`, kas, kā mēs atminamies, sarakstam ar 1000 skaitļiem pieprasa 10 soļus. Taču mums tas būs jādara vienu reizi katrai no simts K vērtībām, tātad tas kopā prasīs 10 x 100 = 1000 operācijas.

Rezultātā, mūsu ideja veic ne vairāk kā 11000 operācijas, bet naivais variants veic 100000 operācijas, kas ir apmēram deviņas reizes vairāk.

Šis ir atsevišķs piemērs, taču to var vispārināt. Ja mums ir saraksts ar N skaitļiem, un ir M dažādas K vērtības, tad naivais variants prasa N x M operācijas, bet mūsu ideja prasa N x $\log_2$N + M x $\log_2$N operācijas.

Šo mēs apzīmējam ar lielo burtu O - naivā varianta sarežģītība ir O(N x M), bet mūsu idejas sarežģītība ir O(N x $\log_2$N + M x $\log_2$N).

Sarežģītību tik detalizēti tālāk vairs neapskatīsim, taču cerams, ka tas ir skaidrs, ka šis ir svarīgi. Pirmkārt, jo tas mums iedod īsu veidu, kā pierakstīt, cik ātra ir programma, un otrkārt, jo tas mums ļauj saprast, kāpēc viens risinājums ir labāks par citu risinājumu.

1.3.4. Iebūvētā funkcija

Kā zināms, vektoru augošā secībā sakārto:

```
5  sort(s.begin(), s.end());
```

Ja mēs vēlamies vektoru sakārtot dilstošā secībā, tad darbojas:

```
6  sort(s.begin(), s.end(), greater<ll>());
```

Taču, varbūt mums ir kāda īpaša vēlme. Piemēram, mēs gribam sakārtot sarakstu augošā secībā pēc pirmajiem cipariem, bet tos skaitļus, kuriem pirmais cipars sakrīt, mēs gribam sakārtot dilstošā secībā.

Tātad, mēs gribam iegūt:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
19	18	17	14	12	11	10	28	24	22	39	37	33	43	40

Mēs vispirms varam nodefinēt īpašu funkciju, kas salīdzina divus skaitļus a un b, un atgriež true, ja skaitlim a jāatrodas pirms skaitļa b:

```
7  auto seciba = [&](ll a, ll b) { // funkcija iekš int main()
8      ll ap = (a - a%10) / 10;
9      ll bp = (b - b%10) / 10;
10     if(ap!=bp){ // ja pirmie cipari ir dažādi
11         return ap<bp;
12         // tad atgriezt true, ja a pirmais cipars ir mazāks
13     }else{
14         return a>b;
15         // citādi atgriezt true, ja a ir lielāks
16     }
17 };
```

Tad, lai iegūtu augstāk doto vektoru, atliek izsaukt:

```
18 sort(s.begin(), s.end(), seciba);
```

1.3.5. Uzdevums

Datu tipa `priority_queue` metožu `push()` un `pop()` sarežģītība ir $O(\log_2 n)$.

Tas nozīmē, ka (cerams) iepriekš atrastajam uzdevuma 652B atrisinājumam sarežģītība visticamāk ir aptuveni $O(2n \times \log_2 n)$ (veikto `push()` un `pop()` izsaukumu skaits, kas reizināts ar $\log_2 n$).

Izmantojot `sort`, pastāv, piemēram, tāds atrisinājums, kura sarežģītība, rupji rēķinot, ir $O(n \times \log_2 n)$. Tas tātad ir nedaudz labāks risinājums. Uzraksti un iesniedz jaunu risinājumu šim uzdevumam!

1.4. Matemātikas uzdevumi

1.4.1. Zilonis

Drīz mēs pievērsīsimies dažādiem sarežģītākiem algoritmiem, taču ir svarīgi neaizmirst, ka pirms vispār ķeras klāt pie programmēšanas, par uzdevumu ir jāpadomā uz papīra.

Viens ekstrēms piemērs ir Codeforces uzdevums 617A. Zilonis dzīvo uz skaitļu ass punkta 0, un viņš grib apciemot savu draugu, kurš dzīvo uz skaitļu ass punkta x. Šeit x ir naturāls skaitlis, kas nepārsniedz vienu miljonu.

Vienā gājienā zilonis spēj veikt 1, 2, 3, 4 vai 5 soļus uz priekšu. Mērķis ir noteikt mazāko gājienu skaitu, kurā zilonis var sasniegt savu draugu.

Protams, mēs varam veidot programmu, kas aplūko visas iespējamās 1, 2, 3, 4 vai 5 soļu kombinācijas, kuru summa ir x, un nosaka, kurā no tām ir vismazāk gājienu. Taču, ja x = 1000000, tad šo kombināciju skaits būs milzīgs. Šis nešķiet efektīvi.

Tāpēc pāriesim pie domāšanas uz papīra.

Viena no iespējamajām gājienu kombinācijām ir k reizes veikt 5 soļus, līdz mēs esam ne vairāk kā 5 soļu attālumā no drauga. Tad pietiek ar vēl vienu gājienu, lai sasniegtu draugu.

Tas ir pavisam skaidrs, ka kombinācijas ar mazāk nekā k + 1 gājieniem nav, jo k gājienos lielāku distanci par 5 x k mērot nav iespējams.

Tagad varam pieiet atpakaļ pie datora un minūtes laikā izdomāt un iesniegt atrisinājumu.

1.4.2. Dalītāji un noderīgas funkcijas

Viens ļoti svarīgs fakts par dalītājiem ir - skaitlis n dalās ar d tad un tikai tad, ja tas dalās ar n/d. Piemēram, 24 dalās ar 4 tad un tikai tad, ja 24 dalās ar 6.

Šis fakts mums ļauj izveidot funkciju, kas izvada visus kāda skaitļa n dalītājus ar aptuvenu sarežģītību O(√n):

```
-8  void dalitaji(ll n){
-7      for(ll i=1; i<=sqrt(n); i++){
-6          if (n%i==0){
-5              cout << i << " ";
-4              if (n/i!=i) cout << n/i << " ";
-3          }
-2      }
-1  }
```

Ja n ir 36, tad šis izvada 1 36 2 18 3 12 4 9 6. Ja dalītājus vajag secībā, tad var ievadīt dalītājus vektorā un izsaukt `sort`. Vektorā būs ne vairāk kā 2 x √n skaitļi. Ja šis vektors satur tikai divus skaitļus - skaitli 1 un skaitli n - tad n ir pirmskaitlis.

Dažreiz mums der noskaidrot divu skaitļu lielāko kopīgo dalītāju, vai arī mazāko kopīgo dalāmo. Lielākajam kopīgajam dalītājam pastāv iebūvēta funkcija:

```
1  cout << __gcd(6, 15) << "\n";
```

Šī rinda izvada 3. Tikmēr mazāko kopīgo dalāmo var iegūt, abu skaitļu reizinājumu dalot ar lielāko kopīgo dalītāju:

```
2  cout << 6*15/__gcd(6, 15) << "\n";
```

Šī rinda izvada 30.

Visbeidzot, eksistē dažādas iebūvētas funkcijas priekš kāpināšanas, logaritmiem, absolūtās vērtības iegūšanas, noapaļošanas, vai arī trigonometrijas - attiecīgi pow; `log2` vai `log10`, vai `log`; `abs`; `round`, `ceil` un `floor`; `sin` un citas.

1.4.3. Matemātika informātikas olimpiādēs

Informātikas olimpiādēs matemātika ir sastopama ļoti bieži, un daudzas tēmas - modulārā aritmētika, ģeometrija un daudzstūri, skaitļi dažādās bāzēs - ir ļoti aktuālas.

Tā kā šīs tēmas drīzāk pieder pie gatavošanās matemātikas olimpiādēm, tās šajā materiālā netiek īpaši apskatītas. Taču, ja ir sajūta, ka matemātikas olimpiāžu tēmas nav diez ko labi pazīstamas, tas ir ieteicams aplūkot šīs tēmas atsevišķi, kā arī pamēģināt plašāku ar matemātiku un ģeometriju saistītu Codeforces uzdevumu klāstu.

1.4.4. Uzdevumi

Atrisini uzdevumus 1328A un 588B mājaslapā Codeforces.

2. Dinamiskā programmēšana

2.1. Viendimensionāli piemēri

2.1.1. Trīs jautājumi

Teiksim, ka mums ir dots uzdevums. Tad, mēs varam uzdot trīs jautājumus:

1. *Vai šo uzdevumu var sadalīt apakšuzdevumos?*
2. *Vai mēs varam noskaidrot pirmo apakšuzdevumu rezultātus?*
3. *Vai mums ir veids, kā izmantot iepriekšējo apakšuzdevumu rezultātus, lai iegūtu nākamā apakšuzdevuma rezultātu?*

Ja atbilde uz šiem trim jautājumiem ir jā, tad mums ar to arī pietiek, lai atrisinātu uzdevumu. Šādu risinājumu mēs saucam par dinamiskās programmēšanas jeb DP risinājumu.

2.1.2. Faktoriāls

Kā vienkāršu piemēru, mēģināsim izveidot programmu, kuras mērķis ir noteikt skaitļa n faktoriālu `n!` = 1 x 2 x 3 x `...` x n.

1. *Vai šo uzdevumu var sadalīt apakšuzdevumos?* Jā.

   Mēs varam aplūkot `0!`, `1!`, `2!`, `3!`, un tā tālāk līdz `n!`.

2. *Vai mēs varam noskaidrot pirmo apakšuzdevumu rezultātus?* Jā.

   Mēs zinām, ka `0!` = 1.

3. *Vai mums ir veids, kā izmantot iepriekšējo apakšuzdevumu rezultātus, lai iegūtu nākamā apakšuzdevuma rezultātu?* Jā.

   Ja mēs zinām `(i-1)!` vērtību, tad mēs varam noteikt `i!`, pareizinot šo `(i-1)!` vērtību ar `i`.

Šos slēdzienus kodā var implementēt šādi:

```
1  ll dp[n+1];              // saraksts ar stingri n+1 elementu
2  dp[0]=1;                 // 0! = 1
3  for(ll i=1; i<=n; i++){
4      dp[i]=dp[i-1]*i;     // i! = (i-1)! * i
5  }
6  cout << dp[n] << endl;
```

Ja n = 10, piemēram, tad šī programma izvada 3628800, kas tiešām ir `10!`.

2.1.3. Fibonači virkne

Tagad aplūkosim Fibonači virkni. Tā sākas ar $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, un tā tālāk, un katrs nākamais elements ir iepriekšējo divu elementu summa. Mērķis ir noteikt F_n vērtību.

1. *Vai šo uzdevumu var sadalīt apakšuzdevumos?* Jā.

   Mēs varam noteikt pirmo Fibonači skaitli, tad otro, tad trešo, un tā tālāk līdz n-tajam Fibonači skaitlim.

2. *Vai mēs varam noskaidrot pirmo apakšuzdevumu rezultātus?* Jā.

   Pirmais skaitlis ir 1 un otrais skaitlis ir 1.

3. *Vai mums ir veids, kā izmantot iepriekšējo apakšuzdevumu rezultātus, lai iegūtu nākamā apakšuzdevuma rezultātu?* Jā.

   Ja mēs zinām gan F_{i-2}, gan F_{i-1}, tad mēs varam noteikt F_i, saskaitot F_{i-2} un F_{i-1} kopā.

Šos slēdzienus kodā var implementēt šādi:

```
ll dp[n+1];
dp[1]=1; dp[2]=1;           // pirmais un otrais skaitlis ir 1
for(ll i=3; i<=n; i++){
    dp[i]=dp[i-2]+dp[i-1]; // saskaitām pēdējos divus skaitļus
}
cout << dp[n] << endl;
```

Ja n = 10, piemēram, tad šī programma izvada 55, kas ir desmitais Fibonači skaitlis.

2.1.4. Uzdevums

Atrisini uzdevumu 1180A mājaslapā Codeforces.

2.2. Daudzdimensionāli piemēri

2.2.1. Apakšvirkne

Par virknes A apakšvirkni mēs saucam tādu virkni, ko var iegūt, izdzēšot kaut kādu skaitu elementu no A, bet nemainot elementu secību. Lūk, piemērs ar {3,6,5,9,7}:

- {6,5,7} ir apakšvirkne. To var iegūt, izdzēšot 2 elementus, attiecīgi 3 un 9.
- {3,6,5,9,7} ir apakšvirkne. To var iegūt, izdzēšot 0 elementus.
- {} ir apakšvirkne. To var iegūt, izdzēšot 5 elementus.
- {5,7,6} nav apakšvirkne. Šie skaitļi virknē šādā secībā neparādās.

2.2.2. Garākā kopīgā apakšvirkne

Kā piemēru tam, kā dp masīvs var būt daudzdimensionāls, aplūkosim uzdevumu - noteikt, cik gara ir divu virkņu A un B garākā kopīgā apakšvirkne. Piemēram, virknēm FDDEG un FGDFE garākā kopīgā apakšvirkne ir FDE, tātad meklētais garums ir 3.

1. *Vai šo uzdevumu var sadalīt apakšuzdevumos?* Jā.

   Mēs varam aplūkot katras virknes pirmos dažus elementus. Piemēram, mēs varam aplūkot virknes FD un FGD, vai arī virknes FDDE un FG. Garākās kopīgās apakšvirknes ir $\{A_2, B_3\}$ = FD un $\{A_4, B_2\}$ = F, bet to garumi ir $|A_2, B_3| = 2$ un $|A_4, B_2| = 1$.

2. *Vai mēs varam noskaidrot pirmo apakšuzdevumu rezultātus?* Jā.

   Ja kādai no virknēm mēs paņemam pirmos 0 elementus, tad attiecīgais garākās kopīgās apakšvirknes garums noteikti būs 0.

3. *Vai mums ir veids, kā izmantot iepriekšējo apakšuzdevumu rezultātus, lai iegūtu nākamā apakšuzdevuma rezultātu?* Jā.

   Ja A_i un B_j pēdējie elementi ir vienādi, tad virkni $\{A_i, B_j\}$ var iegūt, $\{A_{i-1}, B_{j-1}\}$ pievienojot šo vienādo elementu. Līdz ar to, $|A_i, B_j| = |A_{i-1}, B_{j-1}| + 1$.

   Ja A_i un B_j pēdējie elementi nav vienādi, tad iespējami divi gadījumi:

   - Ja A_i pēdējais elements nepieder virknei $\{A_i, B_j\}$, tad $\{A_i, B_j\}$ ir tas pats, kas $\{A_{i-1}, B_j\}$.
   - Ja A_i pēdējais elements pieder virknei $\{A_i, B_j\}$, tad B_j pēdējais elements tai nepieder, līdz ar to $\{A_i, B_j\}$ ir tas pats, kas $\{A_i, B_{j-1}\}$.

   Tātad $\{A_i, B_j\}$ noteikti ir viena no virknēm $\{A_{i-1}, B_j\}$ un $\{A_i, B_{j-1}\}$. Lai virkne $\{A_i, B_j\}$ būtu pēc iespējas garāka, mēs izvēlamies garāko no šīm divām virknēm, tāpēc $|A_i, B_j| = \max(|A_{i-1}, B_j|, |A_i, B_{j-1}|)$.

Kodā šo ideju var implementēt šādi:

```
ll As = A.size(), Bs = B.size();
ll dp[As+1][Bs+1]{}; /* divdimensionāls saraksts jeb tabula, kur
                        {} nozīmē, ka sākumā visi elementi ir 0 */
for(ll i=1; i<=As; i++){
    for(ll j=1; j<=Bs; j++){
        if(A[i-1]==B[j-1]){
            dp[i][j] = dp[i-1][j-1] + 1;
        }else{
            dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }
}
cout << dp[As][Bs] << "\n";
```

Īpatnēja var izskatīties piektā rinda: kāpēc tur ir `A[i-1]==B[j-1]`, nevis `A[i]==B[j]`? Svarīgi atcerēties, ka mēs gribam salīdzināt A_i un B_j pēdējos elementus. Tā kā A_i garums ir `i`, tā pēdējais elements patiesībā ir `A[i-1]`, un līdzīgi ir ar B.

Ja A un B ir attiecīgi `FDDEG` un `FGDFE`, šī programma izvada 3. Programma darbojas pat tad, ja A un B ir `vector` tipa mainīgie (`string` pats par sevi ir līdzīgs `vector<char>`).

2.2.3. Uzdevumi

Atrisini uzdevumus 1097B un 118D mājaslapā Codeforces.

2.3. Rekonstruēšana

2.3.1. Virknes izvadīšana

Līdz šim visos uzdevumos beigās ir bijis nepieciešams izvadīt vienu vienīgu skaitli. Taču, diemžēl, vienmēr tā nav, un mums mēdz nākties rekonstruēt to, kā mēs ieguvām kaut kādu skaitli kā rezultātu.

Aplūkosim diezgan sarežģītu piemēru: kā iegūt ne tikai garākās kopīgās apakšvirknes garumu, bet arī pašu virkni?

Turpināsim aplūkot virknes FDDEG un FGDFE. Vispirms, vizuāli attēlosim mūsu garākās kopīgās apakšvirknes meklējumus. Sākumā, dp masīvs izskatījās šādi:

		F	D	D	E	G	
	0	0	0	0	0	0	0
F	0	0	0	0	0	0	1
G	0	0	0	0	0	0	2
D	0	0	0	0	0	0	3
F	0	0	0	0	0	0	4
E	0	0	0	0	0	0	5
	0	1	2	3	4	5	

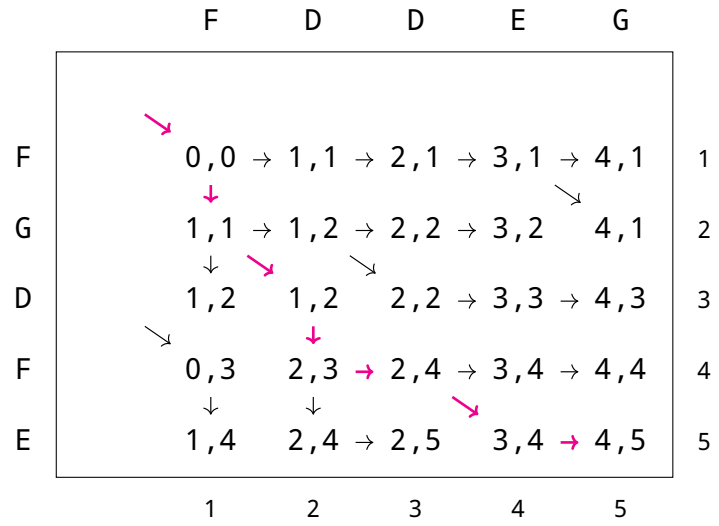
Mēs zinām, ka virknēm A_1 = F un B_1 = F atbilst trešais gadījums - abu virkņu pēdējais elements ir šo virkņu garākajā kopīgajā apakšvirknē. Tātad, `dp[1][1]=dp[0][0]+1`, kas ir 1. Mēs ieliekam vērtību 1 tabulā, un ar bultiņu attēlojam, ka `dp[1][1]` tika iegūts, izmantojot `dp[0][0]`. Šis is redzams tabulā pa kreisi.

		F	D	D	E	G	
	0	0	0	0	0	0	0
F	0	1	0	0	0	0	1
G	0	0	0	0	0	0	2
D	0	0	0	0	0	0	3
F	0	0	0	0	0	0	4
E	0	0	0	0	0	0	5
	0	1	2	3	4	5	

		F	D	D	E	G	
	0	0	0	0	0	0	0
F	0	1	→ 1	→ 1	→ 1	→ 1	1
G	0	1	→ 1	→ 1	→ 1	2	2
D	0	1	2	2	→ 2	→ 2	3
F	0	1	2	→ 2	→ 2	→ 2	4
E	0	1	2	→ 2	3	→ 3	5
	0	1	2	3	4	5	

Pa labi redzams iznākums, šādi aizpildot visu tabulu. Ja vērtību var iegūt vairākos veidos, tad dosim priekšroku diagonālajām bultiņām, tad bultiņām pa labi, tad bultiņām uz leju.

Mēs varam izveidot jaunu tabulu, kurā norādīts, no kurienes iziet tā bultiņa, kas nonāk katrā pozīcijā. Papildus, rozā krāsā iezīmēsim maršrutu līdz apakšējam labajam stūrim.



Šo tabulu `pair<ll, ll> iz[As+1][Bs+1]` ar visām bultiņu iziešanas pozīcijām mēs varam iegūt, iepriekšējo kodu pamainot šādi:

```
6  if(A[i-1]==B[j-1]){
7      dp[i][j] = dp[i-1][j-1] + 1;
8      iz[i][j] = {i-1,j-1};
9  }else{
10     if(dp[i-1][j]>=dp[i][j-1]){
11         dp[i][j] = dp[i-1][j];
12         iz[i][j] = {i-1,j};
13     }else{
14         dp[i][j] = dp[i][j-1];
15         iz[i][j] = {i,j-1};
16     }
17 }
```

Šī tabula mums atļauj atveidot augstāk rozā iekrāsoto maršrutu:

pozīcija x,y:	5,5	4,5	3,4	2,4	2,3	1,2	1,1	0,0
dp[x][y]:	3	**3**	2	2	**2**	1	**1**	0
A[x-1]:	G	E	D	D	D	F	F	
B[y-1]:	E	E	F	F	D	G	F	
iz[x][y]:	4,5	3,4	2,4	2,3	1,2	1,1	0,0	

Skaidrs, ka tad, kad palielinājusies `dp[x][y]` vērtība, salīdzinot ar bultiņas iziešanas pozīciju, ka mēs esam pievienojuši elementu apakšvirknei. Pie tam, `dp[x][y]` nevarētu palielināties no `-1` uz 0, tātad tajā brīdī apakšvirknes veidošanu var pabeigt.

Visbeidzot, mēs varam iegūt vektoru s = EDF šādi:

```
ll x=As, y=Bs;
while(dp[x][y]!=0){
    ll ix = iz[x][y].first;
    ll iy = iz[x][y].second; // bultiņa uz x,y iziet no ix, iy
    if(dp[x][y]>dp[ix][iy]){
        s.push_back(A[x-1]); // citādi pievienot var B[y-1]
    }
    x=ix;
    y=iy;
}
```

Taču garākā kopīgā apakšvirkne ir otrādākā secībā - FDE - tāpēc izvadīšanu veicam šādi:

```
for(ll i=dp[As][Bs]-1; i>=0; i--){ // s garums ir dp[As][Bs]
    cout << s[i];
}
```

Beidzot, mums ir izdevies izvadīt garāko kopīgo apakšvirkni!

2.3.2. Uzdevums

Atrisini uzdevumu 56D mājaslapā Codeforces. Šis uzdevums ir sarežģīts, taču notikumu gaita ir tieši identiska tai, kādu mēs tagad esam apskatījuši:

- izveidot dp masīvu, kur katra virkne nosaka attiecīgi rindu un kolonnu skaitu, un iegūt uzdevuma atbildē nepieciešamo skaitli;
- pielāgot kodu, lai papildus iegūtu `iz` masīvu ar bultiņu iziešanas pozīcijām;
- atveidot maršrutu, un apmest to otrādākā secībā, lai izvadītu atlikušo nepieciešamo uzdevumā.

2.4. Rekursīva implementācija

2.4.1. Rekursīvas funkcijas

Līdz šim, kad mēs vēlējamies noskaidrot kāda apakšuzdevuma rezultātu, mēs to jau bijām ieguvuši kādā brīdī iepriekš - tā bija *imperatīva* implementācija.

Taču ir alternatīva pieeja. Kad mēs vēlamies noskaidrot kāda apakšuzdevuma rezultātu, mēs varam izsaukt funkciju, kas to mums noskaidros - tā būs *rekursīva* implementācija.

Šī funkcija visiem apakšuzdevumiem būs identiska; pie tam, tā kā paši apakšuzdevumi gribēs uzzināt citu apakšuzdevumu rezultātus, šī funkcija šad un tad izsauks pati sevi, tikai ar citiem argumentiem.

Piemēram, lai rekursīvi implementētu faktoriālu, atminamies mūsu trīs jautājumus.

1. *Vai šo uzdevumu var sadalīt apakšuzdevumos?* Jā.

   Mēs varam aplūkot `0!`, `1!`, `2!`, `3!`, un tā tālāk līdz `n!`.

   Šis mūs aicina definēt funkciju ar vienu parametru `fact(i)`.

2. *Vai mēs varam noskaidrot pirmo apakšuzdevumu rezultātus?* Jā.

   Mēs zinām, ka `0! = 1`.

   Līdz ar to, `fact(0)` jāatgriež `1`.

3. *Vai mums ir veids, kā izmantot iepriekšējo apakšuzdevumu rezultātus, lai iegūtu nākamā apakšuzdevuma rezultātu?* Jā.

   Ja mēs zinām `(i-1)!` vērtību, tad mēs varam noteikt `i!`, pareizinot šo `(i-1)!` vērtību ar `i`.

   Citādā secībā, vajag, lai `fact(i)` izsauktu `fact(i-1)` un atgrieztu `fact(i-1) x i`.

Varam to implementēt pavisam vienkārši:

```
-4  ll fact(ll i){
-3      if(i==0) return 1;
-2      else return fact(i-1)*i;
-1  }
```

Tagad, izsaucot `fact(6)`, mēs iegūsim 720. Varam arī attēlot, kuras funkcijas procesā tika izsauktas:

`fact(0)` ← `fact(1)` ← `fact(2)` ← `fact(3)` ← `fact(4)` ← `fact(5)` ← `fact(6)`

Šis liekas maznozīmīgi, jo situācija ir diezgan primitīva, tāpēc aplūkosim nedaudz interesantāku gadījumu.

2.4.2. Memoizācija

Identiskā manierē, rekursīvi implementēsim Fibonači virkni.

1. *Vai šo uzdevumu var sadalīt apakšuzdevumos?* Jā.

   Mēs varam noteikt pirmo Fibonači skaitli, tad otro, tad trešo, un tā tālāk līdz n-tajam Fibonači skaitlim.

   Šis mūs aicina definēt funkciju ar vienu parametru `fib(i)`.

2. *Vai mēs varam noskaidrot pirmo apakšuzdevumu rezultātus?* Jā.

   Pirmais skaitlis ir 1 un otrais skaitlis ir 1.

   Līdz ar to, `fib(1)` jāatgriež 1, un `fib(2)` arī jāatgriež 1.

3. *Vai mums ir veids, kā izmantot iepriekšējo apakšuzdevumu rezultātus, lai iegūtu nākamā apakšuzdevuma rezultātu?* Jā.

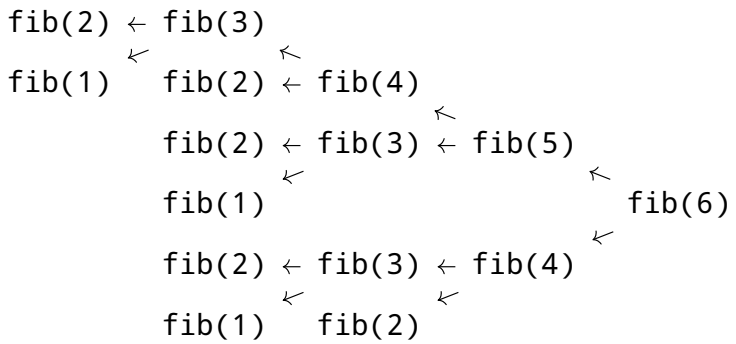
   Ja mēs zinām gan F_{i-2}, gan F_{i-1}, tad mēs varam noteikt F_i, saskaitot F_{i-2} un F_{i-1} kopā.

   Citādā secībā, vajag, lai `fib(i)` izsauktu `fib(i-1)` un `fib(i-2)`, un atgrieztu to summu.

Tātad, funkciju varam uzrakstīt šādi:

```
-4  ll fib(ll i){
-3      if(i==1||i==2)return 1;
-2      else return fib(i-1)+fib(i-2);
-1  }
```

Izsaucot `fib(6)`, mēs iegūsim 8. Taču šoreiz, attēlojot procesu, paveras nelāgs skats:



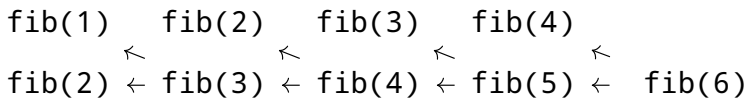
Vairākas no funkcijām mēs izsaucam vairākkārt, un sarežģītība pārsniedz pat $O(F_n)$.

Lai glābtu situāciju, mums ir vienkāršs triks - ja mēs kādu rezultātu jau esam noskaidrojuši, tad saglabāsim šo rezultātu. Ja mums vajadzēs to pašu rezultātu vēlāk, mums nebūs jāveic viss aprēķins pa jaunam, jo mēs varēsim vienkārši nolasīt saglabāto rezultātu.

Šo triku, ko sauc par memoizāciju, bieži vien ir ērti ieviest ar `map`:

```
-5  map<ll, ll> f = {{1,1}, {2,1}};
-4  ll fib(ll i){
-3      if(!f.count(i)) f[i]=fib(i-1)+fib(i-2);
-2      return f[i];
-1  }
```

Sākumā vārdnīca satur tikai vērtības `f[1]=1` un `f[2]=1`. Ja vārdnīca nesatur `f[i]`, tad mēs to noskaidrojam tāpat, kā iepriekš; ja `f[i]` jau ir saglabāts, tad to atgriežam uzreiz. Tagad skats izskatās ievērojami labāk:



2.4.3. Uzdevumi

Atrisini uzdevumus 768B un 559B mājaslapā Codeforces.

3. Grafi

3.1. Ievads par grafiem

3.1.1. Struktūras izveidošana

Pirms mēs pievēršamies grafiem, mums ir jāiepazīstas ar `struct`. Piemēram, mēs varam izveidot šādu `struct`:

```
-5  struct Item{
-4      string s;
-3      ll n;
-2      bool b;
-1  }; // šeit jābūt semikolam aiz figūriekavas
```

Šajā brīdī, `Item` uzvedās kā mūsu pašu izdomāts datu tips, un mēs varam izveidot `Item` tipa mainīgo a:

```
1  Item a = {"Teksts", 15, 1};
```

Mēs varam piekļūt mūsu mainīgā a vērtībām ar attiecīgi `a.s`, `a.n` un `a.b`:

```
2  a.n = 12;
3  cout << a.s << " " << a.n << " " << a.b << "\n";
```

Pēdējā rindiņa šeit izvada `Teksts 12 1`.

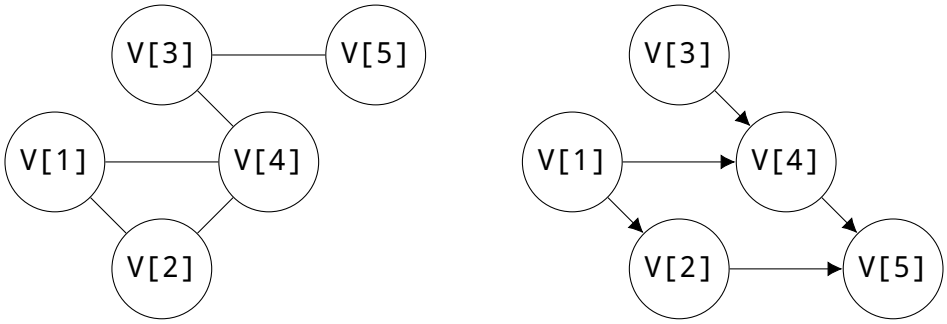
Datu struktūrai var arī piešķirt sākotnējas vērtības. Piemēram, mēs varam nodefinēt:

```
-4  struct Item{
-3      bool b = true;
-2      vector<ll> v;
-1  };
```

Jebkuram izveidotam `Item` a uzreiz piederēs vērtība `a.b == true` un tukšs vektors `a.v`.

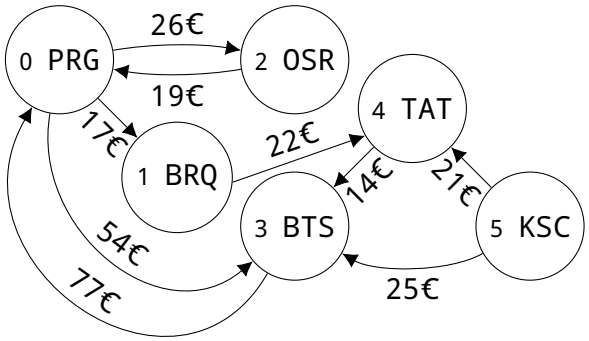
3.1.2. Grafu veidi

Grafs sastāv no virsotnēm un savienojumiem. Lūk, divi piemēri:



Pa kreisi ir redzams neorientēts grafs (savienojumus var šķērsot abos virzienos), bet pa labi redzams orientēts grafs (savienojumus var šķērsot vienā virzienā).

Grafi var ātri kļūt diezgan sarežģīti. Piemēram, mēs varam izveidot orientētu grafu ar lidojumiem starp lidostām, lidojumu cenām un lidostu nosaukumiem:



3.1.3. Grafa implementēšana

Izveidosim šo lidostu grafu no šādiem ievaddatiem:

```
6 9
PRG BRQ OSR BTS TAT KSC
0 1 17
0 2 26
0 3 54
1 4 22
2 0 19
3 0 77
4 3 14
5 3 25
5 4 21
```

Mēs varam radīt sarakstu ar lidostām. Katrai lidostai mums vajag saglabāt nosaukumu, izejošo lidojumu galamērķus, un šo lidojumu cenas, tāpēc ieviešam šādu `struct`:

```
-5  struct Lidosta{
-4      string n;       // nosaukums
-3      vector<ll> g;   // galamērķi
-2      vector<ll> c;   // cenas
-1  };
```

Vispirms izveidojam tukšu sarakstu:

```
1  ll L, M;        // lidostu un maršrutu skaiti
2  cin >> L >> M;
3  Lidosta l[L];   // saraksts ar L lidostām
```

Tad nolasām visu lidostu nosaukumus:

```
4  for(ll i=0; i<L; i++) cin >> l[i].n;
```

Un visbeidzot, ievadām visus lidojumus:

```
5  for(ll i=0; i<M; i++){
6      ll S, G, C; // sākumpunkts, galapunkts, cena
7      cin >> S >> G >> C;
8      l[S].g.push_back(G);
9      l[S].c.push_back(C);
10 }
```

Tagad, ja mēs gribam, piemēram, katram lidostas #5 galamērķim izvadīt nosaukumu, mēs varam izsaukt:

```
11 for(auto G:l[5].g) cout << l[G].n << " ";
```

Šis izvadīs BTS TAT.

Tas, kā mēs implementējam grafu, ir pilnībā atkarīgs no tā, kāds ir uzdevums. Piemēram, ar šo lidostu sarakstu atbildēt uz jautājumu "Cik lidojumi ierodas lidostā TAT?" varētu būt diezgan sarežģīti, un nepieciešama būtu kāda cita pieeja.

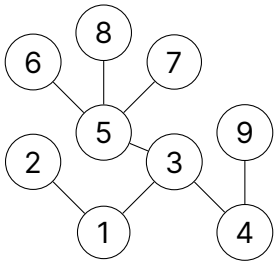
3.1.4. Uzdevums

Atrisini uzdevumus 129B un 707B mājaslapā Codeforces.

3.2. Koki

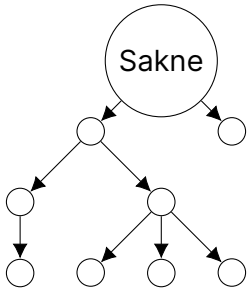
3.2.1. Definīcijas

Par koku sauc tādu saistītu neorientētu grafu, kuram savienojumu skaits ir par 1 mazāks nekā virsotņu skaits. Attiecīgi, par koku parasti sauc, piemēram, šādu grafu:



Koki olimpiādēs parādās ļoti bieži, jo tiem ir jauka īpašība - tajos nav ciklu. Tas, ka savienojumu skaits ir par 1 mazāks nekā virsotņu skaits, bieži vien parādās kā apakšuzdevums.

Taču šī materiāla ietvaros izmantosim citu definīciju. Par koku sauksim tādu saistītu *orientētu* grafu, kuram katrā virsotnē ienāk savienojums no tieši vienas citas virsotnes, ar vienu izņēmumu - "saknes virsotni", kurā neienāk neviens savienojums. Lūk, piemērs:



3.2.2. Summas masīvs

Iedomāsimies šādu uzdevumu: mums ir dots skaitļu saraksts K. Vienā vaicājumā mēs varam vai nu aizstāt kādu skaitli sarakstā ar kaut ko citu, vai arī noteikt kāda saraksta intervāla skaitļu summu.

Ja vispirms tiek doti aizstāšanas vaicājumi, un tad tiek doti intervālu summu vaicājumi, tad šo uzdevumu var viegli atrisināt ar summas masīvu.

Teiksim, ka sākumā ir šāds saraksts:

0	1	2	3	4	5	6	7
1	8	8	2	3	9	9	2

Nomainīt vērtības ir pavisam elementāri. Ja mēs gribam, lai `K[0]` kļūst par 4 un `K[6]` kļūst par 7, tad mēs tos tā arī aizstājam:

0	1	2	3	4	5	6	7
4	8	8	2	3	9	7	2

Pēc visiem aizstāšanas vaicājumiem, mēs varam izveidot summas masīvu S - tas ir tāds saraksts, ka `S[i]` ir vienāds ar `K[0] + K[1] + ... + K[i-1] + K[i]`.

Lai šo sarakstu izveidotu, atliek pamanīt, ka `S[0]` ir tā pati vērtība, kas ir `K[0]`, un katru nākamo `S[i]` var iegūt ar izteiksmi `S[i-1] + K[i]`. Rezultātā iegūst šādu S:

0	1	2	3	4	5	6	7
4	12	20	22	25	34	41	43

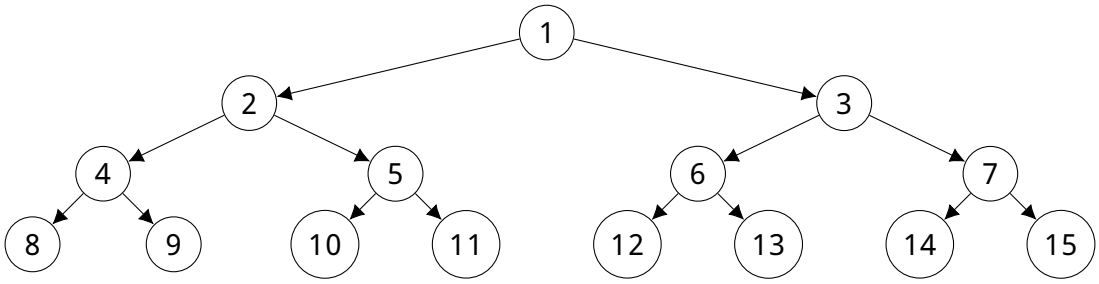
Tagad, ja mēs gribam noskaidrot visu skaitļu no `K[i]` līdz `K[j]` summu, novērojam, ka `K[i] + ... + K[j]` ir tik, cik `(K[0] + ... + K[j]) - (K[0] + ... + K[i-1])`, kas ir attiecīgi `S[j] - S[i-1]`.

Līdz ar to, piemēram, skaitļu no `K[2]` līdz `K[6]` summa ir `S[6] - S[1] = 41 - 12 = 29`.

Tikmēr skaitļu no `K[0]` līdz `K[5]` summa ir vienkārši `S[5] = 34`.

3.2.3. Segmentu koks

Par segmentu koku (segment tree) sauksim tādu koku, kuram no 2^a virsotnēm neiziet savienojumi, bet no pārējām 2^a-1 virsotnēm iziet tieši divi savienojumi. Piemēram, ja saknes virsotne ir ar indeksu 1 un a ir 3, tad segmentu koks ir šāds:



Segmentu koks ir pavisam īpašs, jo to var implementēt kā sarakstu ar 2^{a+1} elementiem.

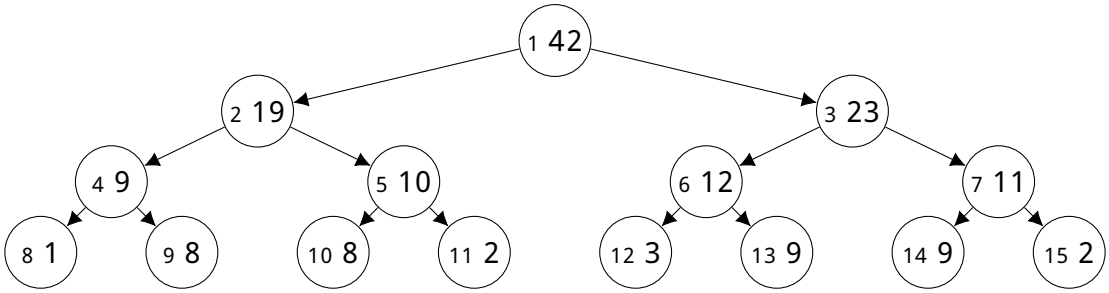
Papildus varam pamanīt, ka no virsotnes ar indeksu $i<2^a$ iziet savienojumi uz virsotnēm ar indeksiem $2i$ un $2i+1$. Līdzīgi, virsotnē ar indeksu $j>1$ ienāk savienojums no virsotnes ar indeksu `floor(j/2)`.

Šie divi novērojumi nozīmē, ka šim grafam nemaz nav atsevišķi jāsaglabā savienojumi.

3.2.4. Vērtības aizstāšana

Mūsu iepriekšējais uzdevuma risinājums ar summas masīvu kļūst ļoti lēns, ja vaicājumi ir doti sajauktā secībā, jo summas masīvs pēc katra jaunā aizstāšanas vaicājuma kļūtu nepareizs un būtu jāveido no jauna.

Tāpēc aplūkosim jaunu ideju. Saglabāsim sarakstu segmentu koka apakšējā rindā, bet pārējās virsotnēs ieliksim abu zemāk esošo virsotņu summu.

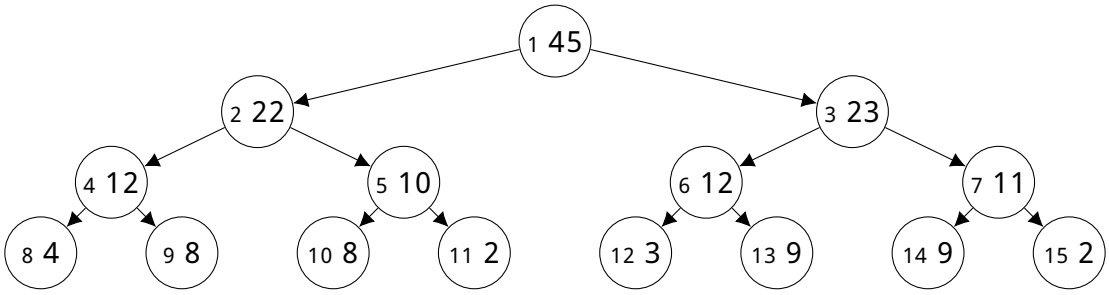


Sauksim šo grafu par `st`. Tad, `st` var izveidot šādi:

```
1  ll K[8] = {1, 8, 8, 2, 3, 9, 9, 2}, st[16];
2  for(ll i=8; i<=15; i++) st[i] = K[i-8];
3  for(ll i=7; i>=1; i--) st[i] = st[2*i] + st[2*i+1];
```

Aplūkosim, kā mēs varētu veikt aizstāšanas vaicājumu. Ja mēs vēlamies aizstāt `K[0]` ar 4, tad vispirms nosakām, ka `K[0]` atbilst pozīcijai `st[8]`.

Pie tam, aizstājot vērtību 1 ar 4, izmaiņa būs 3. Par šo vērtību mainīsies gan `st[8]`, gan arī `st[4]`, `st[2]`, `st[1]`:



Līdzīgi, ja mēs gribētu `K[6]`, kas ir 9, aizstāt ar 7, tad izmaiņa būtu `-2`, un tā attiektos uz `st[14]`, `st[7]`, `st[3]`, `st[1]`.

Pārtaisīsim `K` un `st` par globāliem sarakstiem - paņemsim pirmo rindiņu, kur mēs ieviesām `K` un `st`, un pārliksim to pirms `int main()`. Tad aizstāšanas vaicājumu `av` var ieviest šādi:

```
-8  void av(ll i, ll v){ // indekss, jaunā vērtība
-7      v-=K[i];         // vērtības izmaiņa
-6      i+=8;            // indekss grafā st
-5      while(i!=0){
-4          st[i]+=v;
-3          i/=2;        // C++ automātiski noapaļo uz leju
-2      }
-1  }
```

3.2.5. Uzdevums

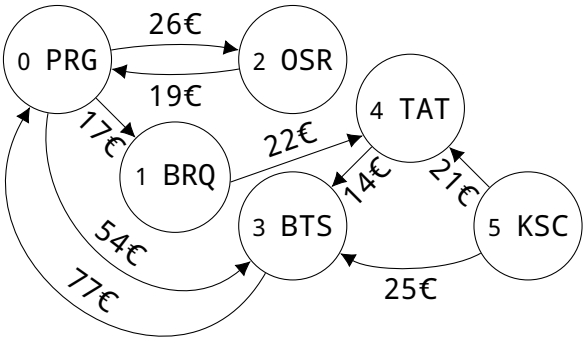
Atrisini uzdevumu 722C mājaslapā Codeforces.

3.3. Meklēšana dziļumā

3.3.1. Ideja

Lai mēs varētu noteikt atbildes ar segmentu koku uz intervāla summas vaicājumiem, mums būtu ieteicams iepazīties ar meklēšanu dziļumā.

Kā piemēru izmantosim mūsu lidostu grafu:



Teiksim, ka mēs gribam noskaidrot, kuras lidostas var sasniegt no lidostas #3, veicot jebkādu lidojumu skaitu.

Līdzīgi kā iepriekš, varam izveidot sarakstu `Lidosta l[6]` un to aizpildīt:

```
-16  struct Lidosta{
-15      string n;        // nosaukums
-14      vector<ll> g;    // galamērķi
-13      vector<ll> c;    // cenas
-12      bool a = false; // vai lidosta ir apmeklēta
-11  };
-10  Lidosta l[6];        // globāli definēts saraksts
```

Tad jautājuma atbildi mēs varam iegūt ar rekursīvu funkciju:

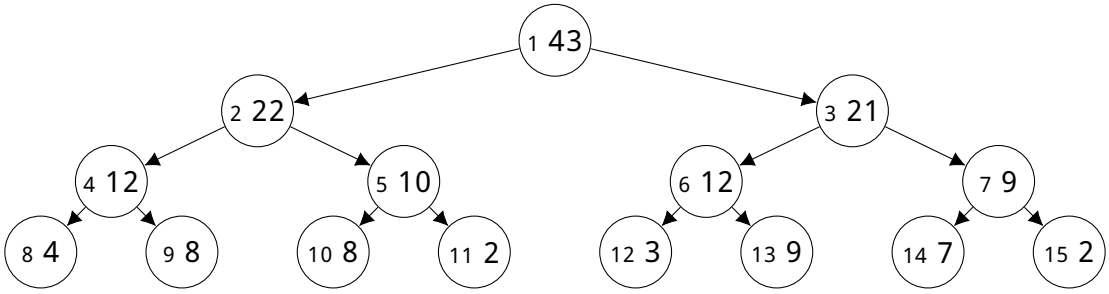
```
 -9  void dfs(ll K){
 -8      cout << K << " ";     // izvada, ka apmeklē lidostu K
 -7      l[K].a=true;          // atzīmē, ka lidosta ir apmeklēta
 -6      for(auto G:l[K].g){   // iet cauri visiem lidostas galamērķiem
 -5          if(!l[G].a){      // ja galamērķis vēl nav apmeklēts
 -4              dfs(G);       // tad apmeklē galamērķi
 -3          }
 -2      }
 -1  }
```

Tagad, izsaucot `dfs(3)`, programma izvada 3 0 1 4 2.

Šo pieeju sauc par meklēšanu dziļumā (depth-first search), jo sasniedzot lidostu, mēs turpinām doties tālāk caur šo sasniegto lidostu ("iet dziļāk"), līdz tas vairs nav iespējams.

3.3.2. Intervāla summas noteikšana

Aplūkosim, kā mēs varam izmantot meklēšanu dziļumā, lai noteiktu intervāla summu. Pieņemsim, ka esam izsaukuši `av(0, 4)` un `av(6, 7)`, un segmentu koks izskatās šādi:



Mēs vēlamies uzzināt `K[i] + ... + K[j]`.

Mēs zinām, ka `st[1]` ir skaitļu `K[0] + ... + K[7]` summa - kopā 8 saskaitāmie. Teiksim vispārīgi, ka jebkurš `st` elements `st[e]` ir skaitļu `K[k] + ... + K[l]` summa, un kopā ir `s` saskaitāmie. Tad mēs varam `dfs` motivēt šādi:

- ja `i<=k` un `l<=j`, tad pilnīgi visi saskaitāmie `K[k] + ... + K[l]` ir atrodami summā `K[i] + ... + K[j]`, tāpēc pieskaitām kopsummai `st[e]` un neturpinām iet dziļāk;
- ja `l<i` vai `j<k`, tad neviens no saskaitāmajiem `K[k] + ... + K[l]` nav atrodams summā `K[i] + ... + K[j]`, līdz ar to neturpinām iet dziļāk;
- citādi, mums jāiet dziļāk par elementu `st[e]` uz elementiem `st[2e]` un `st[2e+1]`, un katram no šiem elementiem piederēs attiecīgi pirmie `s/2` un pēdējie `s/2` elementa `st[e]` saskaitāmie.

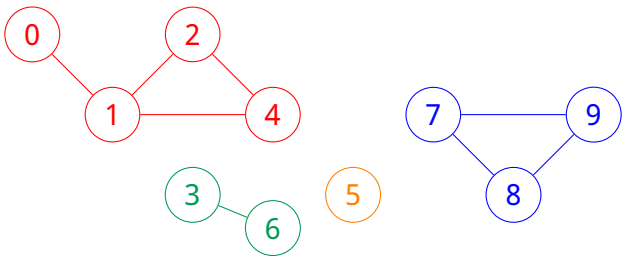
Kodā šī funkcija izskatās šādi:

```
-13  ll dfs(ll e, ll k, ll l, ll s, ll i, ll j){
-12      if(i<=k and l<=j){
-11          return st[e];
-10      }else if(l<i or j<k){
 -9          return 0;
 -8      }else{
 -7          return dfs(2*e, k, k+s/2-1, s/2, i, j)
 -6             + dfs(2*e+1, l-s/2+1, l, s/2, i, j);
 -5      }
 -4  }
 -3  void sv(ll i, ll j){
 -2      cout << dfs(1, 0, 7, 8, i, j) << "\n";
 -1  }
```

Tagad, `sv(2, 6)` un `sv(0, 5)` attiecīgi izvada 29 un 34. Sarežģītība gan `sv`, gan `av` ir $O(\log_2 n)$, kur n ir saraksta K elementu skaits.

3.3.3. Komponentes

Apskatīsim tādu neorientētu grafu, kas nav saistīts.



Mēs virsotņu kopu saucam par atsevišķu komponenti tad, ja no jebkuras šīs kopas virsotnes var sasniegt jebkuru citu kopas virsotni, bet nevar sasniegt nevienu citu virsotni grafā. Attiecīgi, augstāk attēlotajā grafā katra komponente ir savā krāsā.

Teiksim, ka mums ir šādi ievaddati:

```
10 8
0 1
1 2
2 4
4 1
3 6
7 9
8 9
8 7
```

Kā mēs varētu noskaidrot, cik komponentes ir šajā grafā? Vispirms izveidojam `struct` un ieviešam `dfs` līdzīgi, kā iepriekš:

```
-14 struct Virsotne{
-13     vector<ll> k;   // kaimiņi
-12     bool a = false; // vai virsotne ir apmeklēta
-11 };
-10 Virsotne v[10];
 -9
 -8 void dfs(ll N){
 -7     v[N].a = true;
 -6     for(auto K:v[N].k){
 -5         if(!v[K].a){
 -4             dfs(K);
 -3         }
 -2     }
 -1 }
```

Tad, nolasām grafu:

```
ll V, S; // virsotnes un savienojumi
cin >> V >> S;
for(ll i=0; i<S; i++){
    ll v1, v2; // savienotās virsotnes
    cin >> v1 >> v2;
    v[v1].k.push_back(v2);
    v[v2].k.push_back(v1);
}
```

Lai noteiktu, cik komponentes ir, mums ir vienkāršs plāns:

- ja virsotne vēl nav apmeklēta, mēs izsaucam `dfs`, kas par apmeklētām atzīmēs visas pārējās komponentei piederošās virsotnes, un palielinām komponenšu skaitu par 1;
- ja virsotne jau ir apmeklēta, mēs to varam izlaist, jo tā pieder kādai no jau iepriekš pieskaitītajām komponentēm.

Šo plānu implementēt var šādi:

```
ll C = 0; // cik komponenšu ir
for(ll i=0; i<V; i++){
    if(!v[i].a){
        dfs(i);
        C++;
    }
}
cout << C;
```

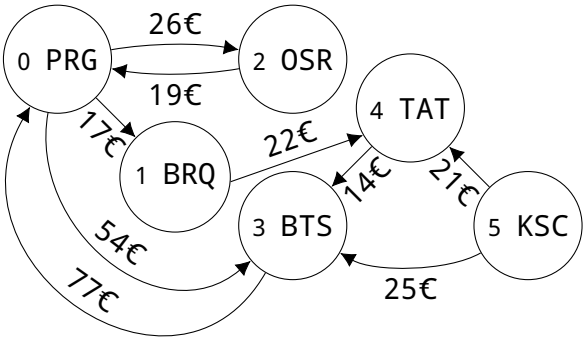
Pēdējā rinda atgriež 4.

3.3.4. Uzdevumi

Atrisini uzdevumus 429A un 1167C mājaslapā Codeforces.

3.4. Meklēšana plašumā

3.4.1. Ideja



Kad mēs iepriekš izvadījām, kuras lidostas var sasniegt no lidostas #3, mums īpaši neinteresēja šo lidostu secība.

Taču teiksim, ka mēs gribam lidostas izvadīt nepieciešamo lidojumu skaita secībā. Attiecīgi, mēs vēlamies izvadīt nevis 3 0 1 4 2, bet, piemēram:

0	1	2	2	3
3	0	1	2	4

Šim nolūkam mēs varam izmantot meklēšanu plašumā (breadth-first search). Lūk, viens piemērs, kā implementēt meklēšanu plašumā:

```
-13  void bfs(ll K){
-12      vector<ll> v = {K}; // apmeklēto lidostu vektors
-11      l[K].a = true;      // sākumā apmeklēta tikai lidosta K
-10      for(ll i=0; i<v.size(); i++){
 -9          cout << v[i] << " ";     // izvada lidostu
 -8          for(auto G:l[v[i]].g){   // iet cauri galamērķiem
 -7              if(!l[G].a){         // ja galamērķis nav vektorā
 -6                  v.push_back(G);  // pievieno to vektoram
 -5                  l[G].a = true;   // un atzīmē to kā apmeklētu
 -4              }
 -3          }
 -2      }
 -1  }
```

Tagad, `bfs(3)` izvada 3 0 1 2 4.

Īsumā, ja mēs vispirms atzīmējam visus lidostas galamērķus, tā ir meklēšana plašumā. Ja mēs pie pirmās iespējas dodamies uz jaunu galamērķi, tad tā ir meklēšana dziļumā.

3.4.2. Struktūru rinda

Iepriekš, mēs implementējām `bfs` ar vektoru, taču, kā mēs redzēsim vēlāk, visbiežāk šim nolūkam ir nepieciešams ieviest `priority_queue`.

Teiksim, ka mēs gribam ieviest `priority_queue<Lidosta>`, kurā lidostas ir sakārtotas alfabētiski pēc nosaukuma. Pēc datu nolasīšanas izsaucam:

```
priority_queue<Lidosta> Q;
for(ll i=0; i<L; i++) Q.push(l[i]);
```

Mēs uzreiz saņemam:

```
error:  no operator "<" matches these operands
operand types are:  const Lidosta < const Lidosta
```

Programmai nav skaidrs, pēc kāda principa salīdzināt divas lidostas.

Tāpēc mēs varam uzrakstīt salīdzināšanas funkciju, kas atgriež `true`, ja lidostai a jābūt virs lidostas b, un tad izveidot `priority_queue`, kas izmanto šo funkciju:

```
auto sec = [&](Lidosta a, Lidosta b){
    return a.n > b.n; // vai a.n ir pirms b.n alfabētiski
};
priority_queue<Lidosta, vector<Lidosta>, decltype(sec)> Q(sec);
for(ll i=0; i<L; i++) Q.push(l[i]);
cout << Q.top().n;
```

Šī programma atgriež BRQ. Ja mēs divpadsmitajā rindā nomainītu > uz <, programma atgrieztu TAT.

Šeit tas, kā mēs ieviešam Q, ir nedaudz dīvaini. Taču par laimi, sacensību sistēmas dokumentācijā pie `priority_queue` līdzīgu piemēru var ātri atrast. Ieteicams šo gadījumu tagad arī sameklēt.

3.4.3. Lētākais ceļš

Kā pēdējo aplūkosim svarīgu jautājumu - kas ir zemākā cena, par kuru mēs varam no vienas lidostas tikt uz citu?

Piemēram, teiksim, ka mēs gribam uzzināt lētāko ceļu no lidostas #2 uz lidostu #3. Īsākais ceļš, caur PRG un tad pa taisno uz BTS, maksā, 73€, taču, lidojot caur PRG, BRQ un TAT, mēs varam ietaupīt 1€ un sasniegt BTS par 72€.

Teiksim, ka mēs katrai lidostai esam saglabājuši galamērķu skaitu `s`. Kā pirmo soli, mēs varam uztaisīt jaunu `struct`, kurā mēs saglabātu veikumu - kuru lidostu mēs esam sasnieguši, un par kādu cenu mēs to esam paveikuši.

```
-25  struct Veikums{
-24      ll l; // sasniegtā lidosta
-23      ll c; // cena
-22  };
```

Tad mēs varam izveidot salīdzināšanas funkciju, lai lētākais variants būtu virs dārgākiem variantiem:

```
-21  auto sec = [&](Veikums a, Veikums b){
-20      return a.c > b.c;
-19  };
```

Visbeidzot, mēs varam izveidot pašu `bfs` funkciju:

```
-18  void bfs(ll i, ll j){
-17      priority_queue<Veikums,vector<Veikums>,decltype(sec)> Q(sec);
-16      Q.push({i, 0});              // lidosta i sasniegta par 0€
-15      while(!Q.empty()){
-14          Veikums v = Q.top();
-13          Q.pop();
-12          if(l[v.l].a) continue;   // izlaižam apmeklētu lidostu
-11          l[v.l].a = true;         // citādi to apmeklējam
-10          if(v.l == j){            // ja sasniegta meklētā lidosta
 -9              cout << v.c << "\n"; // tad izvadām cenu
 -8              return;
 -7          }
 -6          for(ll i=0; i<l[v.l].s; i++){
 -5              Q.push({l[v.l].g[i], v.c+l[v.l].c[i]});
 -4              // rindai pievienojam galamērķus, sasniegšanas cenas
 -3          }
 -2      }
 -1  }
```

Šobrīd ieguvums no jaunā `struct` ir diezgan neliels - patiesībā bez tā var pilnībā iztikt, lietojot `pair<ll, ll>`.

Taču, ja mēs gribētu ieviest, piemēram, kaut kāda veida rekonstruēšanu, lai iegūtu ne tikai cenu, bet arī visas lidostas, kurās vajadzēja pārsēsties, mums būtu daudz vieglāk pamainīt `Veikums` saturu nekā atmest `pair<ll, ll>` datu tipu pilnībā.

3.4.4. Uzdevums

Atrisini uzdevumu 1037D mājaslapā Codeforces.

Epilogs

Visbeidzot, aplūkosim, ko vērts paturēt prātā, piedaloties Latvijas informātikas olimpiādē.

- **Dokumentācija ir vislabākais draugs!**

Šis materiāls, un C++ valoda kā tāda, satur ļoti daudz dažādus datu tipus, iebūvētas funkcijas un konstrukcijas. Tās visas atcerēties nav prātīgi. Tāpēc noteikti ir ieteicams iepazīties ar dokumentāciju un saprast, kur ko var atrast.

- **Apakšuzdevumos var atrast vērtīgus punktus!**

Atšķirībā no Codeforces uzdevumiem, LIO uzdevumiem ir apakšuzdevumi. Tas ir svarīgi tos izlasīt. Vienmēr ir tādi uzdevumi, kuros 100 punktus dabūt nav lemts, tomēr tas nenozīmē, ka attiecīgi jāsaņem 0 punkti. Apakšuzdevumos bieži var sakrāt pat lielāko daļu iespējamo punktu.

- **Publiski redzamais rezultāts nav pilnais rezultāts!**

No visiem maniem olimpiāžu gadiem, šis ir mans mīļākais ekrānšāviņš. Tīri teorētiski, redzot patīkamo zaļo krāsu, es varēju apstāties, taču man šķita, ka risinājums nebija diez ko ātrs. Tā arī bija - un tie 40 sakrātie punkti togad izšķīra zelta medaļas ieguvējus.

- **Skaitās tikai pēdējais iesūtījums!**

Vismaz, laikam. Šo noteikti vērts pārjautāt vai sameklēt nolikumā.

- **Ierobežojumi mēdz kaut ko pateikt priekšā!**

Kā tāds nerakstīts noteikums, tas nav iespējams uztaisīt sarakstu ar vairāk kā miljons elementiem (vektori gan var būt lielāki). Līdz ar to, LIO ļoti tipisks ierobežojums ir ap 10^5. Ja ierobežojums ir ievērojami zemāks, kā 1000, tad tas paver iespēju taisīt tabulu. Bet, ja ierobežojums ir daudz lielāks, kā 10^{18}, tad sarakstu ar tādu izmēru taisīt noteikti nevar.

- **Materiāls un olimpiāde ir divas dažādas vides!**

Šajā materiālā katrs uzdevums ir sekojis uzreiz pēc tā tēmas. Līdz ar to, kaut kāds pirmais iespaids par to, kas uzdevumā būs jādara, jau ir. Taču olimpiādē tēmas nav dotas.

Lai palīdzētu attīstīt prasmi atpazīt metodes, es esmu izveidojis uzdevumu komplektu. Uzdevumus var mēģināt atrisināt, taču tos var arī vienkārši izlasīt un mēģināt izprast, kurai tēmai katrs varētu piederēt.

- **Šis materiāls nav visaptverošs!**

Šis materiāls, cerams, ļauj labi sagatavoties LIO, taču eksistē daudzas citas idejas un algoritmi. IOI mājaslapā var uzzināt, kādas tēmas mēdz parādīties vispasaules olimpiādē.

Tas viss! Vēlu veiksmi Latvijas informātikas olimpiādē!