Napredni algoritmi u natjecateljskom programiranju

Iletić, Karlo

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike i računarstva

Karlo Iletić

Napredni algoritmi u natjecateljskom programiranju

Završni rad

Osijek, 2021.

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike i računarstva

Karlo Iletić

Napredni algoritmi u natjecateljskom programiranju

Završni rad

Mentor: izv. prof. dr. sc. Slobodan Jelić

Osijek, 2021.

Advanced algorithms in competitive programming

Sažetak

U ovom radu ćemo opisati tri naprednija algoritma koja se pojavljuju u natjecateljskom programiranju. Za svaki od algoritama ćemo dati primjer zadatka sa natjecanja na kojem ćemo demonstrirati kako rješiti zadatak koristeći taj algoritam.

Ključne riječi

Natjecateljsko programiranje, algoritmi, najniži zajednički predak, dinamičko programiranje, slomljeni profil, Mo, korjenska dekompozicija, c++.

Abstract

In this paper we will describe three advanced algorithms that appear in competitive programming. For each of the algorithms we will give an example of a competition task where we will demonstrate how to solve the task using that algorithm.

Key words

Competitive programming, algorithms, lowest common ancestor, dynamic programming, broken profile, Mo, sqrt decomposition, c++

Sadržaj

Uvod

Na natjecanjima u programiranju cilj je rješiti što veći broj zadataka u što bržem vremenu. Zadaci koji se pojavljuju su algoritamskog tipa, odnosno potrebno je osmisliti algoritam koji za dani ulaz vraća točan izlaz te natipkati implementaciju tog algoritma u nekom od programskih jezika koje natjecanje podržava. Svaki zadatak ima zadano vremensko i memorijsko ograničenje unutar kojih se natjecatelj mora držati.

Tipični zadaci koji se pojavljuju mogu pripadati nekima od sljedećih kategorija: kombinatorika, teorija brojeva, teorija grafova, teorija igara, računarska geometrija (engl. computational geometry), procesuiranje stringova, strukture podataka, dinamičko programiranje, linearna algebra ili numeričke metode. Težina zadataka varira od natjecanja do natjecanja, ali u pravilu najteži zadaci na natjecanju predstavljaju veliki izazov čak i najboljim natjecateljima.

Neki od najpopularnijih natjecanja su *Međunarodna olimpijada u informatici* (engl. skraćenica IOI), *Međunarodno fakultetsko natjecanje u programiranju* (engl. skraćenica ICPC), te natjecanja koje organiziraju velike tvrtke poput Googlea - *Google Code Jam*, i Facebooka - *Facebook Hacker Cup*. Svako od tih natjecanja se održava jednom godišnje.

Također, postoje mnoge natjecateljske platforme koje imaju veliku zajednicu s ciljem vježbanja za natjecanja u programiranju. Jedne od najpoznatijih su *codeforces*[1], *atcoder*[2], *codechef*[3] na kojima se održavaju online natjecanja barem jednom tjedno.

Postoji ogroman broj algoritama i struktura podataka s kojim rješavamo zadatke, stoga ćemo u ovom radu obraditi samo tri odabrane teme — problem pronalaska najnižeg zajedničkog pretka, dinamičko programiranje na slomljenom profilu i Moov algoritam.

[1] `https://codeforces.com`
[2] `https://atcoder.jp`
[3] `https://codechef.com`

1 Osnovni pojmovi

U ovom poglavlju ćemo definirati nekoliko pojmova koje ćemo koristiti u ostatku rada.

Definicija 1.1. ***Graf*** *G je uređeni par (V, E) gdje je V skup elemenata koje nazivamo vrhovima, a E skup uređenih parova vrhova čije elemente nazivamo bridovima.* ***Neusmjereni graf*** *je onaj graf za koji vrijedi ako $(u, v) \in E \implies (v, u) \in E$, gdje su $u, v \in V$.* ***Usmjereni graf*** *je onaj graf koji nije neusmjeren.*

Definicija 1.2. *Neka je $G = (V, E)$ graf. Konačna šetnja je niz bridova $(e_1, e_2, \ldots, e_{n-1})$ za koje postoji niz vrhova $(v_1, v_2, \ldots, v_n)$ takvih da vrijedi $e_i = (v_i, v_{i+1})$, za $i = 1, 2, \ldots, n-1$.* ***Jednostavan put*** *je šetnja u kojoj su svi bridovi i vrhovi različiti.*

Definicija 1.3. ***Stablo*** *je neusmjeren graf u kojem za bilo koja dva vrha postoji jedinstven jednostavan put.* ***Ukorijenjeno stablo*** *je stablo u kojem postoji jedan poseban vrh kojeg nazivamo korijen stabla.*

Definicija 1.4. ***Udaljenost*** *između dva vrha u stablu je broj bridova na jednostavnom putu između ta dva vrha.*

Definicija 1.5. ***Potomak*** *(engl.* ***descendant****) vrha v u ukorijenjenom stablu je bilo koji vrh koji je ili dijete od v ili (rekurzivno) potomak od bilo kojeg djeteta od v.*

Definicija 1.6. ***Predak*** *(engl.* ***ancestor****) vrha v u ukorijenjenom stablu je bilo koji vrh kojem je v potomak.*

Definicija 1.7. *U ukorijenjenom stablu,* ***roditelj*** *vrha v je onaj vrh s kojim je v direktno povezan i koji se nalazi na putu vrha v do korijena stabla.*

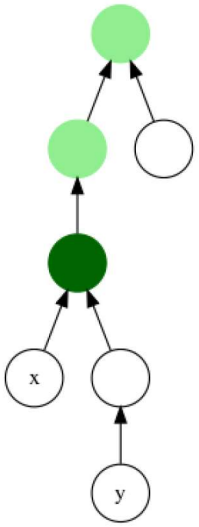
2 Najniži zajednički predak

U ovom poglavlju ćemo govoriti o algoritmima za pronalazak najnižeg zajedničkog pretka (skraćeno, NZP) za bilo koja dva vrha u ukorijenjenom stablu.

Uvedimo neke oznake kojih ćemo se pridržavati u ovom poglavlju. Neka je dano ukorijenjeno stablo $T = (V, E)$ sa korijenom kojeg ćemo označiti sa r. Broj vrhova u stablu označimo sa n. Definirajmo dubinu d_v vrha $v \in T$ kao udaljenost od v do korijena stabla r, a dubina korijena neka bude 0. Radi jednostavnosti uzmimo da je roditelj korijena sam korijen. Nadalje, neka su $x, y \in T$ proizvoljna dva vrha u stablu za koje želimo pronaći NZP.

Prije svega, definirajmo najniži zajednički predak.

Definicija 2.1. *Neka je T ukorijenjeno stablo. Najniži zajednički predak dvaju vrhova u i v iz stabla T je najniži (odnosno najdublji) vrh koji je predak i od u i od v, uz pretpostavku da je svaki vrh sam sebi predak.*



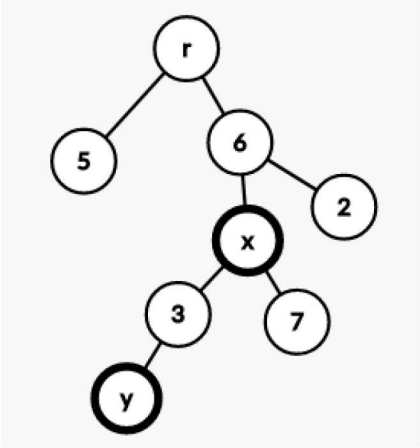
Slika 1: NZP vrhova x i y je označen tamno zelenom bojom. Ostali zajednički pretci su označeni svijetlo zelenom bojom.[4]

2.1 Naivni pristup

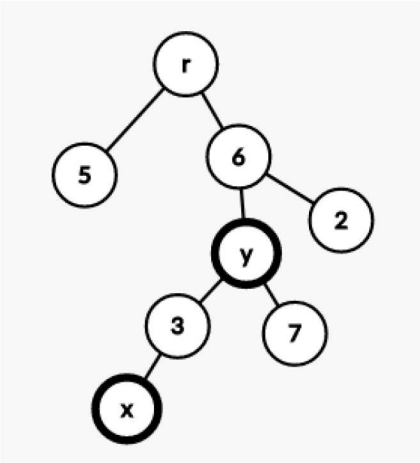
Opišimo prvo algoritam koji pronalazi NZP na naivan način. Kada promatramo dva vrha x i y mogu se desiti sljedeći slučajevi:

- x je predak od y ili je y predak od x (tretiramo ih na isti način)
- x nije predak od y, niti je y predak od x

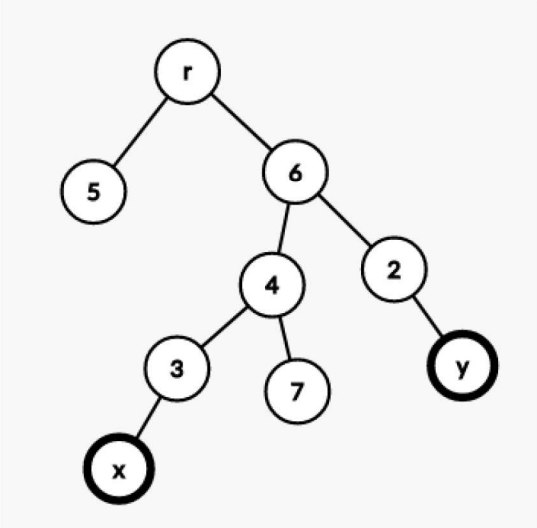
[4]Izvor slike: `https://en.wikipedia.org/wiki/Lowest_common_ancestor#/media/File:Lowest_common_ancestor.svg`



Slika 2: Primjer za prvi slučaj kada je x predak od y



Slika 3: Primjer za prvi slučaj kada je y predak od x



Slika 4: Primjer za drugi slučaj

Bez smanjenja općenitosti uzmimo da vrijedi $d_x \geq d_y$, odnosno vrh x je dublji ili jednake dubine kao y. Ovaj algoritam tada radi na sljedeći način. Prvo, iz vrha x penjemo se prema korijenu stabla sve dok dubina trenutnog vrha nije jednaka d_y. Penjanje vršimo na način da vrh x postavimo na njegovog roditelja. Vrh do kojeg smo se popeli iz vrha x označimo sa x'. Ako su sada x' i y jednaki to znači da se desio prvi slučaj i NZP je onda x'. Inače, imamo dva vrha x' i y u stablu koja su na istoj dubini, ali nijedan od njih nije predak drugome. Sada se možemo iz oba vrha istovremeno penjati prema korijenu stabla sve dok vrhovi ne postanu isti. Ovaj algoritam će se sigurno zaustaviti jer u najgorem slučaju će vrhovi doći do samog korijena koji je predak svim vrhovima u stablu.

2.1.1 Pseudokod

Sljedećim pseudokodom opisujemo dani naivan pristup za pronalazak NZP.

Algorithm 1: Naivni način pronalaska NZP

Data: Stablo T, dubina vrha d_i, roditelj vrha p_i, te neka dva vrha iz stabla x i y
Result: NZP od x i y

```
1 if d_x < d_y then
2     swap(x, y);
3 while d_x > d_y do
4     x ← p_x
5 while x ≠ y do
6     x ← p_x
7     y ← p_y
8 return x;
```

2.1.2 Analiza složenosti

Analizirajmo prvo vremensku složenost. Linije 1 do 2 se izvršavaju u $O(1)$. While petlja na liniji 3 ovisi o razlici $d_x - d_y$, odnosno dubini stabla. Kako nam je dano proizvoljno stablo njegova dubina ovisi o broju vrhova u stablu. Linija 4 se izvrsava u $O(1)$ pa je ukupno izvršavanje te white petlje jednako $O(n)$, gdje je n broj vrhova u stablu. Linije 5 do 7 također ovise o dubini stabla, odnosno broju vrhova u stablu pa je složenost isto $O(n)$. Nadalje, linija 8 se izvršava u $O(1)$. Sveukupna složenost ovog algoritma je tada $O(n)$.

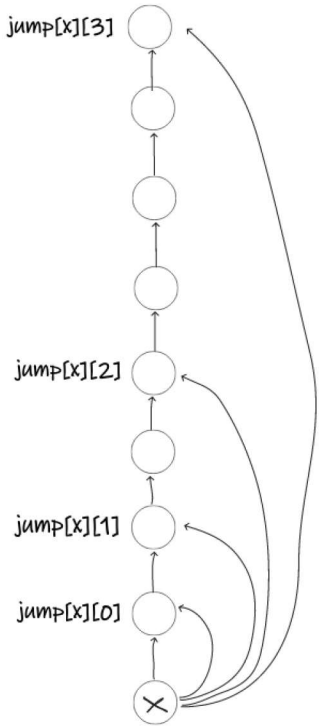
Što se tiče prostorne složenosti, potrebna nam je reprezentacija stabla, te dva polja duljine jednake broju vrhova u stablu za dubine i roditelje. Stoga, prostorna složenost je $O(n)$, gdje je n broj vrhova u stablu.

2.2 Binarno skakanje

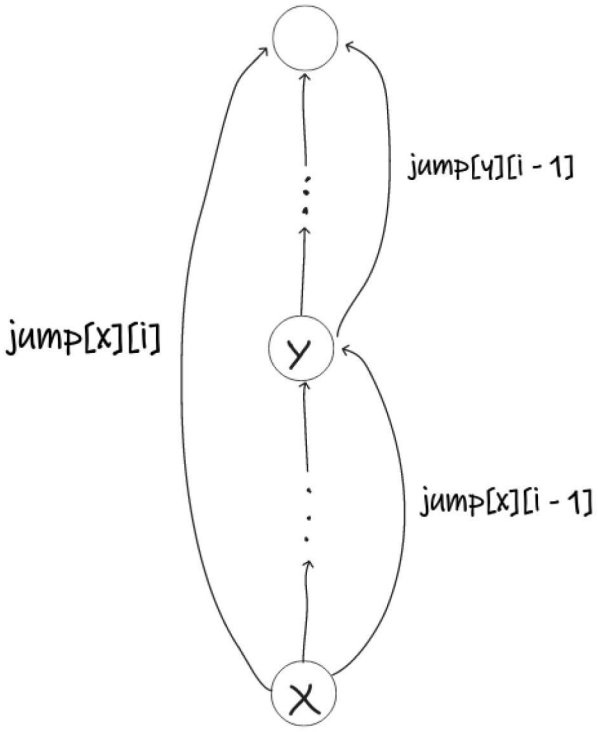
Sada kada imamo jednostavan algoritam za pronalazak NZP postavlja se pitanje da li je moguće osmisliti algoritam koji koristi istu ideju ali to radi na brži način. Problem kojeg želimo riješiti binarnim skakanjem (engl. binary lifting) je putovanje iz bilo kojeg vrha u stablu prema nekom od njegovih predaka brže nego samo iteriranjem po svim pretcima. To možemo postići na način da predračunamo neke informacije koje će nam pomoći.

Uvedimo dvodimenzionalno polje `jump` čija je prva dimenzija veličine n, a druga dimenzija veličine $\lceil \log_2 n \rceil$. Definirajmo vrjednost `jump[u][k]` kao predak vrha u koji se nalazi na dubini $d_u - 2^k$ (za primjer vidi sliku 5). Ako je $d_u - 2^k < 0$ postavimo tu vrijednost na korijen stabla.

Ideja iza izgradnje ove tablice je da gradimo sloj po sloj. Bazni sloj ove tablice se odnosi na skok za jedan iz svakog vrha, a to je upravo jednako polju roditelja p. Zato, možemo postaviti `jump[u][0]` $= p_u$. Za proizvoljni i-ti sloj, $i \geq 1$, vrijednost `jump[u][i]` možemo dobiti vrlo lako iz informacije prethodnog sloja. Naime, ako iz vrha u skočimo za 2^{i-1} i onda opet za 2^{i-1}, to je isto kao da smo iz vrha u skočili za 2^i (vidi sliku 6). Sada imamo da je `jump[u][i]` = `jump[jump[u][i - 1]][i - 1]`.



Slika 5: Primjer čemu će prve četiri vrijednosti za vrh x u jump tablici biti jednake



Slika 6: Za izračun skoka 2^i možemo dva puta uzastopno skočiti za 2^{i-1}

Algorithm 2: Inicializacija jump tablice

Data: Roditelj vrha p_i
Result: Jump tablica
for $i \leftarrow 1$ **to** n **do**
    $jump[i][0] \leftarrow p_i$;
for $k \leftarrow 1$ **to** $\lceil \log_2 n \rceil$ **do**
    for $i \leftarrow 1$ **to** n **do**
        $jump[i][k] \leftarrow jump[jump[i][k-1]][k-1]$;
return jump;

2.2.1 k-skok

Pokažimo kako bi iz vrha x skočili do nekog pretka koji se nalazi na dubini D. Skok koji trebamo napraviti iznosi $d_x - D$. Problem nastaje kada taj skok nije potencija broja 2 jer ne možemo to učiniti u jednom skoku. Označimo skok sa $k = d_x - D$ i pokažimo kako ga možemo napraviti koristeći tablicu skakanja `jump`.

Promotrimo binarni zapis broja $k \in \mathbb{N}$. Postoje $b_1, b_2, \ldots, b_m$, gdje je $b_i \in \mathbb{N}_0$, takvi da je $k = 2^{b_1} + 2^{b_2} + \cdots + 2^{b_m}$. To znači da ako želimo napraviti skok za k možemo napraviti manje uzastopne skokove za $2^{b_1}, 2^{b_2}, \ldots, 2^{b_m}$.

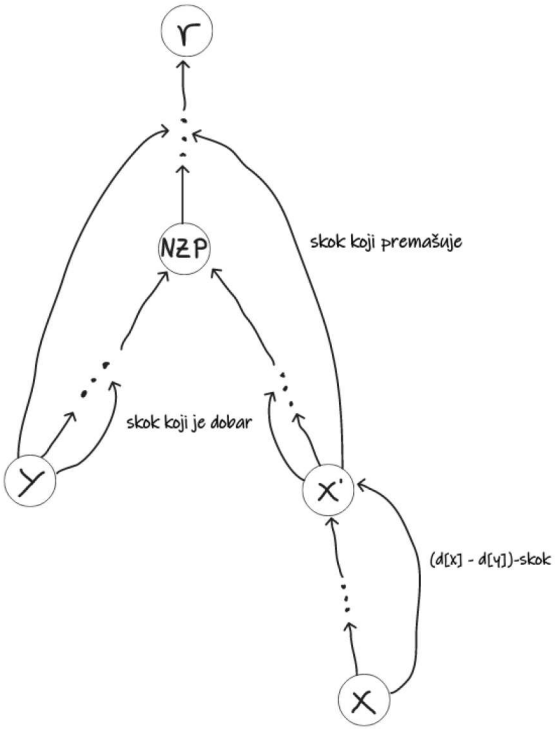
Algorithm 3: k-skok

Data: Jump tablica $jump$, skok k, vrh iz kojeg skačemo x
Result: k-ti predak vrha x
for $i \leftarrow 0$ **to** $\lceil \log_2 k \rceil$ **do**
    if *i-ti bit uključen u broju k* **then**
        $x \leftarrow jump[x][i]$;
return x;

Sada možemo opisati algoritam za pronalazak NZP pomoću binarnog skakanja. Sjetimo se naivnog pristupa gdje smo iz vrha x prvo radili $d_x - d_y$ skok. Sada umjesto skakanja na roditelja sve dok ne dođemo do dubine d_y možemo to napraviti sa k-skokom. Nakon toga imat ćemo vrhove x i y na istoj dubini. Također, može se desiti da su sada x i y jednaki. To znači da je vrh y bio predak od x pa je on ujedno i NZP. Inače, x i y nisu isti i potrebno je pronaći njihov NZP. Sada možemo iskoristiti ideju koja se zasniva na binarnom traženju. Kada napravimo neki k-skok iz vrha x, odnosno y, mogu se desiti dva slučaja:

- preskočili smo NZP (ili barem skočili do njega)
- skočili smo do nekog vrha kojemu je NZP od x i y predak

Uočimo da kada se prvi slučaj desi x i y će skočiti na istog pretka. Upravo iz tog razloga možemo održavati invarijantu da su x i y različiti. Krenit ćemo od najveće potencije broja 2 i ići sve do najmanje, te pokušati napraviti skok na svakoj. Ako nas skok odvede do različitog vrha to znači da se NZP nalazi još iznad njega pa možemo napraviti taj skok, inače ako nas skok odvede do istog vrha to znači da se NZP nalazi ispod njega pa ne želimo napraviti taj skok. Nakon završetka skakanja ćemo imati neka dva vrha koja nisu jednaka i svaki skok potencije broja 2 iz njih vodi u isti vrh. To znači da je čak i skok od $2^0 = 1$ vodi do istog vrha pa je onda očito da će NZP od njih biti njihov roditelj.



Slika 7: Slika koja prikazuje skokove koje radimo u algoritmu

Algorithm 4: Pronalazak NZP preko binarnog skakanja

Data: Jump tablica $jump$, vrhovi za koje tražimo NZP x i y, broj vrhova u stablu n
Result: NZP od x i y

```
1  if d_x < d_y then
2      swap(x, y);
3  x ← (d_x − d_y)-skok iz vrha x;
4  if x = y then
5      return x;
6  for i ← ⌈log_2 n⌉ to 0 do
7      x' ← jump[x][i];
8      y' ← jump[y][i];
9      if x' ≠ y' then
10         x ← x';
11         y ← y';
12 return jump[x][0];
```

2.2.2 Analiza složenosti

Linije 1 do 2 izvršavaju se u $O(1)$. Linija 3 se izvršava u $O(\log_2 n)$ jer unutar te funkcije imamo petlju koja ide od 0 do $\lceil \log_2 k \rceil$, a k ovisi o dubini stabla, odnosno broju vrhova u stablu n. Unutar te funkcije provjeravama da li je i-ti bit uključen što možemo napraviti u $O(1)$ sa bitshift operacijama. Nadalje, linije 4 do 5 se izvršavaju u $O(1)$. Linije 6 do 11 se izvršavaju u $O(\log_2 n)$ jer imamo $\lceil \log_2 n \rceil + 1$ iteracija petlje u kojoj su sve operacije $O(1)$. Stoga, konačno vrijeme izvršavanja je $O(\log_2 n)$.

Što se tiče prostorne složenosti, prije ovog algoritma moramo napraviti predračun tablice

jump koja je veličine $O(n \log_2 n)$, pa je ukupna prostorna složenost $O(n \log_2 n)$.

2.3 Zadatak - Tourists

U nastavku ćemo dati primjer zadatka u kojem ćemo koristiti NZP.

Problem[5]**:** Dano je stablo s n vrhova ($n \leq 2 \times 10^5$). Vrhovi u stablu su označeni redom od 1 do n. Potrebno je naći sumu duljina puteva između svaka dva vrha u i v za koje vrijedi da $v < u$, te v dijeli u. Duljina puta se definira kao broj vrhova na putu (uključujući oba vrha na krajevima).

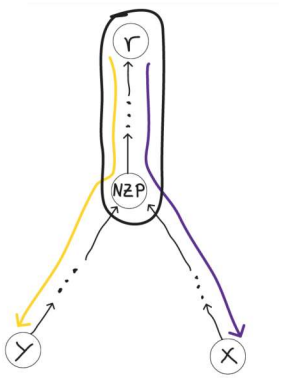
Rješenje: Iteriranje po svim parovima vrhova (u, v) iz stabla i računanjem udaljenosti između ta dva vrha nam daje $O(n^3)$ rješenje jer postoji $O(n^2)$ parova vrhova, a izračun udaljenosti između nekog para možemo napraviti obilaskom stabla u $O(n)$. Ovo rješenje je pre sporo.

Pokažimo prvo kako možemo izračunati udaljenost između neka dva vrha brže od $O(n)$. Neka su u i v vrhovi za koje želimo izračunati udaljenost. Ukorijenimo naše stablo u proizvoljnom vrhu (npr. uzmimo da korijen stabla bude vrh s oznakom 1), te izračunajmo udaljenost svakog vrha do korijena stabla. To možemo postići obilaskom stabla u dubinu (engl. depth first search). Uočimo kako put između bilo koja dva vrha prolazi kroz NZP od ta dva vrha. To je trivijalno za dokazati jer se iz nekog vrha penjemo prema gore sve dok nismo naišli na vrh koji je predak oba vrha, te se iz tog vrha onda spuštamo prema drugom vrhu. Taj vrh je upravo po definiciji NZK od ta dva vrha. Sada udaljenost između u i v možemo dobiti na način da zbrojimo udaljenosti tih vrhova do korijena stabla te oduzmemo duplu udaljenost do NZP od u i v (vidi sliku 8). Na kraju je još potrebno nadodati 1 na taj izraz jer smo oduzimanjem duple udaljenosti do NZP izbacili NZP no on se nalazi na putu i želimo ga ubrajati. Tako smo optimizirali računanje udaljenosti između proizvoljna dva vrha sa $O(n)$ na $O(\log n)$.

Sada više nemamo $O(n^3)$ rješenje nego $O(n^2 \log n)$ što je još uvijek pre sporo. Postoji bolji način za pronalazak svih parova (u, v) gdje v dijeli u (ili obratno) od iteriranja po svim parovima vrhova. Koristimo ideju iz algoritma Eratostenovog sita[1] gdje ćemo u vanjskoj petlji iterirati po svih vrhovima stabla, a u unutarnjoj petlji iterirati po svim njegovim višekratnicima. Tako će vrijednost na vanjskoj petlji uvijek djeliti vrijednost u unutarnjoj petlji pa smo izbacili puno kandidata parova koji ne zadovoljavaju uvjet dijeljenja. Može se pokazati da je kompleksnost algoritma Eratostenovog sita $O(n \log \log n)$[6], pa ukupna kompleksnost našeg rješenja postaje $O(n \log n \log \log n)$ što je dovoljno dobro da zadatak prođe sve testne slučajeve.

[5]Ovaj zadatak možete pronaći na `https://open.kattis.com/problems/tourists`

[6]Dokaz na linku `https://cp-algorithms.com/algebra/sieve-of-eratosthenes.html#toc-tgt-1`



Slika 8: Kada promatramo udaljenost od korijena do vrhova x i y imamo duplo ubrojavanja vrhova na putu od korijena do NZP koji ne pripadaju putu od x do y.

Implementacija u C++ programskom jeziku

```cpp
#include<bits/stdc++.h>
using namespace std;

const int maxn = 2e5 + 5;
vector <int> g[maxn];
int jump[maxn][22], exists[maxn], d[maxn], n;

void dfs(int u, int p) {
  d[u] = d[p] + 1;
  jump[u][0] = p;
  for (int v : g[u])
    if (v != p)
      dfs(v, u);
}

int k_jump(int k, int u) {
  for (int b = 0; b < ceil(log2(n)); b++)
    if (k & (1 << b))
      u = jump[u][b];
  return u;
}

int lca(int u, int v) {
  if (d[u] < d[v])
    swap(u, v);
  u = k_jump(d[u] - d[v], u);
  if (u == v)
    return u;
  for (int k = ceil(log2(n)); k >= 0; k--)
    if (jump[u][k] != jump[v][k]) {
      u = jump[u][k];
      v = jump[v][k];
```

```
    }
  return jump[u][0];
}

int dist(int u, int v) {
  return d[u] + d[v] - 2 * d[lca(u, v)];
}

int main() {
  cin >> n;
  for (int i = 0; i < n - 1; i++) {
    int u, v;
    cin >> u >> v;
    g[u].push_back(v);
    g[v].push_back(u);
    exists[u] = exists[v] = 1;
  }
  dfs(1, 1);
  for (int k = 1; k <= ceil(log2(n)); k++)
    for (int u = 1; u <= n; u++)
      jump[u][k] = jump[jump[u][k - 1]][k - 1];
  long long res = 0;
  for (int u = 1; u <= n; u++)
    for(int v = 2 * u; v <= n; v += u)
      if (exists[u] && exists[v])
        res += dist(u, v) + 1;
  cout << res;
}
```

3 Dinamičko programiranje na slomljenom profilu

Zadaci[7] koje rješavamo dinamičkim programiranjem na slomljenom profilu (engl. dynamic programming on broken profile) uključuju:

- pronaći broj načina za popuniti prostor (npr. šahovnicu/rešetku) sa nekim figurama (npr. dominama)
- pronaći način za popuniti prostor sa najmanjim mogućim brojem figura
- pronaći način za djelomično popuniti prostor sa najmanjim mogućim prostorom koji je ostao nepopunjen

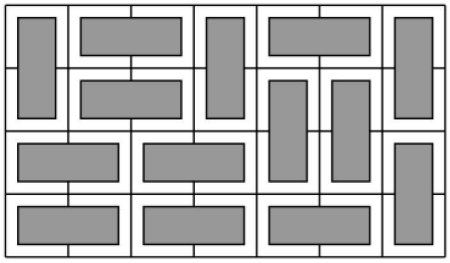
Svojstva[6] koja zadaci koji spadaju u ovu kategoriju generalno imaju su:

- jedna dimenzija rešetke je puno manja od druge
- kada popunjavamo rešetku, svaka ćelija ovisi samo o susjednim ćelijama
- ćelije obično nemaju puno mogućih vrijednosti

Detaljnije ćemo objasniti ovu tehniku na primjerima.

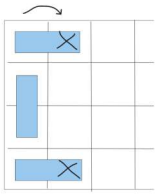
3.1 Zadatak - Counting Tilings

Problem[7]**:** Trebamo izračunati broj načina za popuniti $n \times m$ ($n \leq 10$, $m \leq 1000$) rešetku koristeći 1×2 i 2×1 pločice.

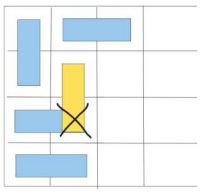


Slika 9: Jedan od načina kako popuniti rešetku dimenzija 4×7. [8]

Rješenje: Koristit ćemo dinamičko programiranje za ovaj zadatak. Popunjavat ćemo rešetku stupac po stupac, odozgo prema dolje, krećući iz ćelije u gornjem lijevom kutu. Na svakoj čeliji možemo pokušati postaviti ili 1×2 ili 2×1 pločicu. Očito, nećemo uvijek moći to učiniti. Ako smo u prošlom stupcu na istom retku stavili 1×2 pločicu to znači da na trenutnoj ćeliji ne možemo staviti nijednu pločicu (vidi sliku 10). Inače, ćelija je prazna pa možemo pokušati postaviti neki tip pločice. Prvo, recimo da smo pokušali postaviti 1×2 pločicu. To ćemo uspjeti napraviti samo ako se ne nalazimo u zadnjem stupcu (inače bi pločica izlazila iz rešetke što nije dozvoljeno). Stoga, za postavljanje pločice 1×2 nam je bitno samo u kojem stupcu se nalazimo. S druge strane, kada pokušamo staviti 2×1 pločicu potrebno je biti malo oprezniji. Slično kao i za 1×2 pločicu kada stavljamo 2×1 pločicu trebamo paziti nalazimo li se na zadnjoj čeliji u retku jer ako da tada bi pločica koju stavimo izlazila iz rešetke što nije dopušteno. Nadalje, ako se ne nalazimo na zadnjoj čeliji potrebno je gledati da li se na čeliji ispod nalazi već zauzeto mjesto nastalo postavljanjem 1×2 pločice iz prethodnog stupca (vidi sliku 11). Ako smo prošli prethodne dvije provjere možemo uspješno postaviti 2×1 pločicu.



Slika 10: Popunili smo prvi stupac i krećemo popunjavati drugi. Čelije označene sa X moramo preskočiti jer one sadržavaju 1×2 pločice iz prethodnog stupca



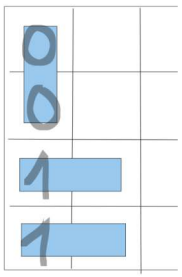
Slika 11: Prvi stupac je popunjen. Na prvoj čeliji drugog stupca smo stavili 1×2 pločicu. Sada na drugoj čeliji drugog stupca pokušamo staviti 2×1 pločicu ali ona se preklapa sa 1×2 pločicom iz prethodnog stupca.

Sada kada smo u teoriji pokrili sve slučaje na koje moramo paziti pitanje je kako ćemo to programski riješiti. Sjetimo se iz opisa problema zadatka da broj redaka može biti najviše

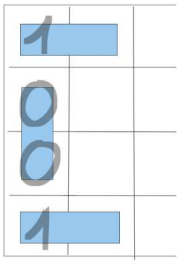
[7]Ovaj zadatak možete pronaći na `https://cses.fi/problemset/task/2181`

[8]Izvor slike: [1], 81. stranica, figure 6.6

10. Ova informacija je vrlo bitna jer nam omogućuje da reprezentiramo niz od n ćelija kao neki cijelobrojni broj. Definirajmo niz od n čelija kao broj od n bitova, gdje je svaki bit dodijeljen jednoj čeliji. Neka je i-ti bit upaljen (postavljen na jedan) ako je na ćeliji kojoj taj bit odgovara postavljena 1×2 pločica, a u svakom drugom slučaju će taj bit biti ugašen (postavljen na nulu, za primjere vidi slike 12, 13).

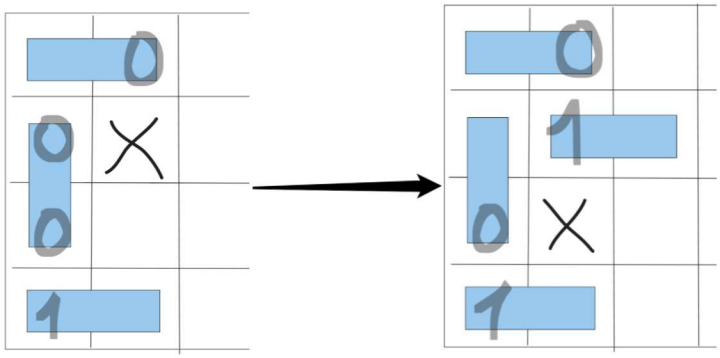


Slika 12: U prvom stupcu imamo 4 ćelije koje smo reprezentirali kao binarni broj 1100 što je jednako broju 12 u dekadskom zapisu.



Slika 13: U prvom stupcu imamo 4 ćelije koje smo reprezentirali kao binarni broj 1001 što je jednako broju 9 u dekadskom zapisu.

Radi jednostavnosti broj koji reprezentira niz od n ćelija nazovimo `mask`. Sada kada smo definirali reprezentaciju niza od n ćelija možemo opisati rješenje zadatka. Održavat ćemo broj `mask` kako se krećemo kroz rešetku. Kada se nalazimo na nekoj čeliji (i, j) znamo koje smo tipove pločica postavili na prethodnih n ćelija. Održavanje samo prethodnih n ćelija će nam biti dovoljno jer je trenutnoj ćeliji bitno samo koji tip pločice smo postavili na ćeliji u prethodnom stupcu na istom retku (vidi sliku 14 za dodatno pojašnjenje).



Slika 14: Na slici lijevo želimo staviti pločicu na ćeliju označenu sa X. Za to nam je potrebna informacija o tipu pločice iz ćelije iz istog retka u prethodnom stupcu koju imamo kao najmanji bit. Stavimo pločicu tipa 1×2 pa izbacimo zadnju ćeliju (na poziciji (0, 1)) jer nam ona više nije potrebna, te dodamo novu ćeliju (na poziciji (1, 2)). Ovakva tranzicija mijenja naš broj `mask` iz 0100 u 1010.

Izbacivanje zadnje čelije i dodavanje nove postižemo na način da broju `mask` napravimo bitshift u desno, te uključimo najveći bit ukoliko je to potrebno (ako novododana čelija sadrži 1×2 pločicu).

Postoje samo dvije tranzicije (postavljanje 1×2, odnosno 2×1 pločice) koje možemo napraviti u $O(1)$ vremenu. Broj različitih stanja je veličina dp tablice što je jednako $O(nm2^n)$. Vremenska kompleksnost je broj tranzicija puta broj stanja što je jednako $O(nm2^n)$.

Također, valja napomenuti da postoji polinomijalno rješenje koje se može pronaći u [1], stranica 82, poglavlje 6.2.5. Counting Tilings.

Implementacija u C++ programskom jeziku

```cpp
#include <bits/stdc++.h>
using namespace std;
const int mod = 1e9 + 7;

int add(int a, int b) { return (a += b) < mod ? a : a - mod; }

int n, m;
int dp[10][1000][1 << 10];

int rek(int i, int j, int mask) {
  if (j == m) return mask == 0;
  if (i == n) return rek(0, j + 1, mask);
  if (dp[i][j][mask] != -1) return dp[i][j][mask];
  if (mask & 1) {
    return dp[i][j][mask] = rek(i + 1, j, mask >> 1);
  }
  int ret = 0;
  // pokušali smo staviti 1 x 2 pločicu
  ret = add(ret, rek(i + 1, j, (mask >> 1) + (1 << (n - 1))));
  if (!(mask & 2) && n > 1 && i <= n - 2){
    // pokušali smo staviti 2 x 1 pločicu
    ret = add(ret, rek(i + 2, j, mask >> 2));
  }
  return dp[i][j][mask] = ret;
}

int main() {
  cin >> n >> m;
  memset(dp, -1, sizeof dp);
  cout << rek(0, 0, 0);
}
```

4 Moov algoritam

U nastavku ćemo pokazati kako odgovoriti na upite na rasponu (engl. range queries) polja duljine n u vremenu $O((n+q)\sqrt{n}f(n))$, gdje je q broj upita, a $f(n)$ neka nepoznata funkcija koju ćemo u nastavku objasniti. Ovaj algoritam se zasniva na ideji korijenske dekompozicije (engl. sqrt decomposition, vidi [1] poglavlje 15.1. Square Root Techniques). Možemo ga

primjeniti samo na upite koje možemo riješiti offline[9].

Ideja iza ovog algoritma je da odgovaramo na upite u posebnom poretku koji ovisi o poziciji upita. Podijelit ćemo naše polje nad kojim radimo upite na blokove jednake duljine (osim zadnjeg bloka koji može biti manje ili jednake veličine), te prvo odgovarati na upite koji imaju početak u bloku 0, zatim u bloku 1 i tako dalje. Unutar zasebnog bloka ćemo isto odgovarati upite u posebnom poretku a to je sortiranom po kraju upita. Definirat ćemo strukturu podataka u koju ćemo pohraniti informacije o trenutnom rasponu. Kada želimo odgovoriti na sljedeći upit (na početku raspon je prazan) želimo se pomaknuti do raspona kojeg taj upit pokriva. To ćemo napraviti tako da dodajemo/brišemo elemente s obje strane našeg trenutnog raspona sve dok nismo zadovoljili raspon sljedećeg upita. Na taj način trebamo definirati strukturu podataka u kojoj možemo dodavati/brisati element po element.

4.1 Struktura podataka

U ovom poglavlju ćemo detaljnije definirati od čega se sve naša struktura podataka sastoji.

Za održavanje trenutnog raspona definirat ćemo dva pokazivača (zapravo su to samo indeksi u polju) od kojih `left` predstavlja početak, a `right` kraj raspona.

Nadalje, definirat ćemo tri funkcije koja struktura podataka koristi:

- `add` - prima samo jedan parametar — indeks elementa u polju. Ova funkcija dodaje novi element na danom indeksu u našu strukturu podataka.
- `remove` - prima samo jedan parametar — indeks elementa u polju. Ova funkcija briše element na danom indeksu iz naše strukture podataka.
- `get_ans` - ne prima nijedan parametar. Ova funkcija daje odgovor na upit određen sa trenutnim rasponom na kojoj se struktura podataka nalazi.

Poslije poziva funkcija `add` i `remove` moramo popraviti naš raspon. Funkcija `add` proširuje trenutni raspon — ako se novododani element nalazi lijevo od početka našeg raspona tada mičemo pokazivač `left` u lijevo, a inače mičemo pokazivač `right` u desno. Funkcija `remove` sužava trenutni raspon — ako se novoobrisani element nalazi na lijevom kraju našeg raspona tada mičemo pokazivač `left` u desno, a inače mičemo pokazivač `right` u lijevo.

Nabrojane varijable i funkcije su one koje svaka struktura podataka mora sadržavati, no može se desiti da su nam potrebne još neke informacije koje su specifične za zadatak kojeg pokušavamo riješiti. Unutar ove strukture se mogu nalaziti nekoliko različitih struktura kao što su balansirano binarno stablo[2], segmentno stablo[5] i mnoge druge koje će funkcije `add`, `remove`, `get_ans` koristiti.

4.2 Analiza složenosti

Sortiranje upita možemo napraviti u $O(q \log q)$ koristeći neki od algoritama koji sortiraju polje duljine q u $O(q \log q)$ vremenu (vidi [2]).

Uzmimo da je veličina bloka jednaka S. Tada će biti sveukupno $\lceil \frac{n}{S} \rceil$ blokova koje indeksiramo počevši od nule. Fiksirajmo neki blok $0 \leq i \leq \lceil \frac{n}{S} \rceil - 1$ i promotrimo koliko puta ćemo pomaknuti naše pokazivače `left` i `right` da odgovorimo na sve upite koji se nalaze u tom bloku.

[9]Offline znači da možemo učitati sve upite te dati odgovor na njih u proizvoljnom poretku.

Svi upiti u trenutnom bloku imaju početak u tom bloku, no kraj im ne mora nužno biti unutar. Prisjetimo se da upite koji pripadaju istom bloku sortiramo uzlazno po kraju upita. To znači da ćemo desni pokazivač pomaknuti najviše za $n - iS = O(n)$ puta. To je zato što i-ti blok počinje od indeksa iS što je najmanja vrijednost kraja upita, a do kraja polja ima točno $n - iS$ elemenata i ne vraćamo se unazad jer su upiti sortirani. Sada pogledajmo koliko puta ćemo pomaknuti lijevi pokazivač. Unutar bloka upiti se ne pojavljuju i nikakvom poretku s obzirom na početak upita. To znači da se može desiti da početak upita alternira između početka i kraja bloka. Označimo sa u_i broj upita koji se nalaze u i-tom bloku. Kako je duljina bloka jednaka S to znači da lijevi pokazivač pomičemo najviše $O(u_i S)$ puta.

Preostaje još slučaj kada smo gotovi sa zadnjim upitom u bloku i idemo na upit koji se nalazi u sljedećem bloku. U najgorem slučaju za lijevi pokazivač imat ćemo da se nalazimo na početku prijašnjeg bloka i trebamo ići na kraj sljedećeg bloka. U tom slučaju ćemo lijevi pokazivač pomaknuti točno $2S - 1 = O(S)$ puta. U najgorem slučaju za desni pokazivač imat ćemo da se nalazimo na samom kraju polja i trebamo ići na početak sljedećeg bloka. U tom slučaju ćemo desni pokazivač pomaknuti točno $n - (i + 1)S = O(n)$ puta.

Stoga, imamo da za i-ti blok pomičemo naše pokazivače $O(u_i S + n)$ puta.

Pogledajmo sada koliko puta ćemo pomaknuti pokazivače kroz sve upite zajedno. Kako je sveukupno q upita imamo da vrijedi:

$$\sum_{i=0}^{\lceil \frac{n}{S} \rceil - 1} u_i = q$$

.

Ukupna složenost će biti jednaka sumi složenosti pojedinih blokova:

$$\sum_{i=0}^{\lceil \frac{n}{S} \rceil - 1} O(u_i S + n) = O(qS + \frac{n}{S} n)$$

Uzmemo li da je veličina bloka jednaka $S \approx \sqrt{n}$ dobivamo konačnu vremensku složenost $O((q + n)\sqrt{n})$. No, ova analiza odgovara samo broju pomicanja naših pokazivača ali prilikom pomicanja se pozivaju funkcije `add` ili `remove`. Radi lakše analize uzmimo da se pozivaju obje funkcije bilo kada se neki od pokazivača pomakne i označimo složenost tog poziva sa $O(f(n))$. Funkciju $f(n)$ ne znamo unaprijed jer ona ovisi o zadatku kojeg rješavamo. Tada je konačna složenost jednaka $O((q + n)\sqrt{n} f(n))$.

Za kraj valja napomenuti da možemo postići drukčiju vremensku složenost ako izaberemo drukčiju veličinu bloka. Na primjer, ako uzmemo $S \approx \frac{n}{\sqrt{q}}$ možemo postići $O(n\sqrt{q})$ pomicanja pokazivača (za detalje posjetite link [10]). Također, možemo značajno smanjiti konstantu ako sortiramo na posebne načine (za detalje posjetite linkove [11] [12]).

Predložak

U nastavku ćemo dati predložak za Moov algoritam koji je napisan u programskom jeziku `c++`.

[10] `https://codeforces.com/blog/entry/61203?#comment-451304`
[11] `https://codeforces.com/blog/entry/61203`
[12] `https://cp-algorithms.com/data_structures/sqrt_decomposition.html#toc-tgt-7`

```
1  int BLOCK_SIZE;
2
3  void add(int index);
4  void remove(int index);
5  int get_ans();
6
7  struct Query {
8    int index, l, r;
9
10   bool operator<(const Query& other) const {
11     if (l / BLOCK_SIZE == other.l / BLOCK_SIZE)
12       return r < other.r;
13     return l < other.l;
14   }
15 };
16
17 std::vector <int> mo_s_algorithm(std::vector <Query>& queries) {
18   std::sort(queries.begin(), queries.end());
19   std::vector <int> answer(queries.size());
20   int l = 0;
21   int r = -1;
22   for (const Query& query : queries) {
23     while (r < query.r) add(++r);
24     while (l > query.l) remove(l--);
25     while (l < query.l) add(++l);
26     while (r > query.r) remove(r--);
27     answer[query.index] = get_ans();
28   }
29   return answer;
30 }
```

Na linijama 20 i 21 incijalizirane su varijable gdje `l` odgovara `left`, a `r` odgovara `right` pokazivaču naše strukture. Desni pokazivač je postavljen na vrijednost -1 jer ćemo tako prvim pomakom u desno dodati prvi element u našu strukturu. Preostaje samo postaviti `BLOCK_SIZE` na veličinu bloka, učitati upite i kreirati listu koja ih sadržava (koristimo `std::vector` za to), te implementirati funkcije `add`, `remove` i `get_ans`.

4.3 Zadatak - Powerful array

Problem[13]**:** Dano je polje pozitivnih cijelih brojeva $a_1, a_2, \ldots, a_n$ ($n \leq 2 \times 10^5$). Promotrimo podniz $a_l, a_{l+1}, \ldots, a_r$, gdje su $1 \leq l \leq r \leq n$, $1 \leq a_i \leq 10^6$. Za svaki pozitivni cijeli broj s označimo sa K_s broj pojavljivanja broja s u tom podnizu. Snagu podniza računamo kao sumu produkta $K_s \times K_s \times s$ za svaki pozitivni cijeli broj s. Dodatno, dano je $q \leq 2 \times 10^5$ upita gdje svaki upit sadrži dva cijela broja l i r te je potrebno ispisati snagu podniza od

[13]Ovaj zadatak možete pronaći na `https://codeforces.com/contest/86/problem/D`

elementa a_l do a_r.

Rješenje: Definirajmo strukturu podataka kojoj je baza struktura podataka iz Moovog algoritma (ima varijable `left, right` i funkcije `add, remove, get_ans`). Uz sve nabrojane atribute sadrži još i polje K čija vrijednost K_i predstavlja broj ponavljanja broja i na trenutnom rasponu $[left, right]$ na kojem se struktura nalazi, te nenegativni cijeli broj ans koja predstavlja snagu podniza od a_{left} do a_{right}. Veličina polja K ovisi o najvećem mogućem broju u polju koji je najviše 10^6.

Preostaje još samo definirati funkcije `add, remove, get_ans`:

- `add(i)` - dodaje element na indeksu i u trenutni podniz. Kada proširujemo naš podniz sa elementom a_i snaga našeg podniza se mijenja i potrebno ju je ažurirati. Kako je snaga podniza definirana kao suma produkata ono što možemo napraviti je oduzeti od ans vrijednost $K_{a_i} \times K_{a_i} \times a_i$ i nadodati vrijednost $(K_{a_i}+1) \times (K_{a_i}+1) \times a_i$. Također, moramo povećati K_{a_i} za jedan. Tako u konstantnom vremenu možemo ažurirati snagu nakon dodavanja novog elementa u podniz.
- `remove(i)` - briše element na indeksu i iz trenutnog podniza. Slično kao i kod funkcije `add` ono što moramo napraviti je prvo od ans oduzeti vrijednost $K_{a_i} \times K_{a_i} \times a_i$ i nadodati vrijednost $(K_{a_i}-1) \times (K_{a_i}-1) \times a_i$. Također, moramo smanjiti K_{a_i} za jedan. Tako u konstantnom vremenu možemo ažurirati snagu nakon brisanja elementa iz podniza.
- `get_ans` - vraćamo samo vrijednost ans

Implementacija u C++ programskom jeziku

```
#include <bits/stdc++.h>
using namespace std;

int BLOCK_SIZE;
long long ans;
int a[200005], cnt[1000005];

void add(int index) {
  ans -= 1ll * cnt[a[index]] * cnt[a[index]] * a[index];
  cnt[a[index]]++;
  ans += 1ll * cnt[a[index]] * cnt[a[index]] * a[index];
}

void remove(int index) {
  ans -= 1ll * cnt[a[index]] * cnt[a[index]] * a[index];
  cnt[a[index]]--;
  ans += 1ll * cnt[a[index]] * cnt[a[index]] * a[index];
}

long long get_ans() {
```

```
  return ans;
}

struct Query {
  int index, l, r;

  Query(int _index, int _l, int _r) : index(_index), l(_l), r(_r) {}

  bool operator<(const Query& other) const {
    if (l / BLOCK_SIZE == other.l / BLOCK_SIZE)
      return r < other.r;
    return l < other.l;
  }
};

std::vector <long long> mo_s_algorithm(std::vector <Query> queries) {
  std::sort(queries.begin(), queries.end());
  std::vector <long long> answer(queries.size());
  int l = 0;
  int r = -1;
  for (const Query& query : queries) {
    while (r < query.r) add(++r);
    while (l > query.l) add(--l);
    while (l < query.l) remove(l++);
    while (r > query.r) remove(r--);
    answer[query.index] = get_ans();
  }
  return answer;
}

int main() {
  int n, q;
  cin >> n >> q;
  BLOCK_SIZE = sqrt(n) + 1;
  for(int i = 0; i < n; i++)
    cin >> a[i];
  vector <Query> queries;
  for(int i = 0; i < q; i++) {
    int l, r;
    cin >> l >> r;
    queries.push_back(Query(i, l - 1, r - 1));
  }
  auto sol = mo_s_algorithm(queries);
  for(int i = 0; i < q; i++)
    cout << sol[i] << "\n";
}
```

Literatura

[1] A. Laaksonen, Guide to Competitive Programming: Learning and Improving Algorithms Through Contests (Undergraduate Topics in Computer Science), Springer 2nd edition, 2020.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, MIT Press; 3rd edition (September 1, 2009)

[3] S. Halim, Competitive Programming 3: The New Lower Bound of Programming Contests, lulu; Third Edition (January 1, 2013)

[4] Code Submission Evaluation System - https://cses.fi/
(pristupljeno 28.9.2021.)

[5] Lista algoritama za natjecateljsko programiranje - https://cp-algorithms.com/
(pristupljeno 28.9.2021.)

[6] https://usaco.guide/adv/dp-more?lang=cpp#dp-on-broken-profile
(pristupljeno 28.9.2021.)

[7] https://cp-algorithms.com/dynamic_programming/profile-dynamics.html
(pristupljeno 28.9.2021.)