| ZADATAK | NAFTA | VALUTA | IZOMORFA | SUTOMI |
|---|---|---|---|---|
| **ulazni podaci** | standardni ulaz | | | |
| **izlazni podaci** | standardni izlaz | | | |
| **vremensko ograničenje** | 1 sec | 1 sec | 2 sec | 1 sec |
| **memorijsko ograničenje** | 32 MB | 32 MB | 32 MB | 32 MB |
| **broj bodova** | **100** | **100** | **100** | **100** |
| | **400** | | | |

Rješenje zadatka neće koristiti brojevne tipove podataka za pamćenje stare i nove cijene već nizove znakova. U tom slučaju nećemo raditi razliku između realnih i prirodnih brojeva.

Primjećujemo da za rješenje, odnosno broj poteza, nije bitno koji se niz znakova upisuje prvi. To koristimo u primjeru rješenja ispod.

Mogući potezi su:

- skinemo pločicu i ostavimo mjesto na displeju prazno
- zamijenimo pločicu s nekom drugom
- stavimo pločicu na prazno mjesto na displeju

Budući da je potez 2 ekvivalentan uzastopnom izvršavanju poteza 1 i 3, u optimalnom rješenju se nikada neće naći potez 3 neposredno nakon poteza 1.

Za dio displeja koji je popunjen i prije i poslije promjene ćemo izvršavati potez 2 samo na mjestima gdje se početna i završna znamenka razlikuju.

Za dio displeja koji je popunjen samo prije promjene izvršiti ćemo poteze 1 ako je završna cijena kraća, odnosno poteze 3 ako je početna cijena kraća i to onoliko puta kolika je razlika u duljini ovih nizova.

Funkcija za izračun broja poteza može izgledati ovako u jeziku C++:

```
int solve(string a, string b) {
  if (a.size() > b.size()) swap(a, b); // neka je a kraci niz.
  int razlika = b.size() - a.size(); // razlika u duljini nizova
  int sol = b.size() - a.size(); // potezi koji rjesavaju prazna mjesta
  for (int i = 0; i < a.size(); ++i) // za svaki znak u podnizovima
    if (a[i] != b[razlika + i]) // koji su zajednicki kada se poravnamo desno
      ++sol; // povecaj rjesenje za jedan.
  return sol;
}
```

Vremenska i memorijska složenost ovog problema je **O(n)**.

### Korištene strukture podataka

U zadatku koristimo dva polja dimenzije $N \times N$ od kojih jedan sadrži realne brojeve, a drugi cjelobrojne vrijednosti ili konstantu ∞.

U polju $put_{a,b}$ zapisujemo najkraći put između valuta s oznakoma $a$ i $b$ ili ∞ ako put ne postoji.

Ako put postoji, tada vrijednost $c_{a,b}$ kazuje koliko je valuta s oznakom $a$ cjenovno jača od valute s oznakom $b$, odnosno među pravilima imamo $1 \bullet a = c_{a,b} \bullet b$.

### Bitni zaključci

Istaknimo nekoliko tvrdnji:

- $put_{a,a}$ je uvijek $0$ i $c_{a,a}$ je uvijek $1$.
- ako je $put_{a,b}$ konačan tada znamo da je konačan i $put_{b,a}$ i vrijedi $c_{a,b}=1/c_{b,a}$.
- ako postoje $put_{i,k}$ i $put_{k,j}$ tada znamo da postoji i $put_{i,j}$ te da vrijedi $c_{i,j} = c_{i,k} \cdot c_{k,j}$

### Algoritam

Na temelju gornjih tvrdnji koje je trebalo zaključiti iz teksta zadatka konstruiramo rješenje.

Niz ***put*** i ***c*** na početku inicijalizirajmo tako da vrijedi prva tvrdnja, tj. na glavne dijagonale polja stavljamo vrijednosti **0** za duljinu puta odnosno **1** za odnos između valuta.

Prilikom učitavanja, za svako pravilo oblika $1 \bullet a = k \bullet b$ postavljamo:

- $put_{a,b} = put_{b,a} = 1$
- $c_{a,b} = k$
- $c_{b,a} = 1/k$

Glavni dio problema je generirati nova pravila na temelju prethodnih. Rješavanju ovog problema ćemo pristupiti slično kao kada rješavamo problem svih najkraćih puteva u grafu. Naime, ako niz **put** shvatimo kao matricu susjedstva nekog grafa tada će svake dvije valute između kojih postoji put biti u odnosu.

Nakon što pronađemo put između neke dvije valute, vrijednost odnosa računamo množeći sve vrijednosti iz polja **c** na tom putu.

Jedan od algoritama koji pronalazi sve najkraće puteve u grafu zove se Floyd-Warshall i upravo ćemo njegovu modifikaciju koristiti kako bi rješili ovaj problem.

```
int N; // broj valuta
double c[maxn][maxn]; // c[a][b] govori koliko je puta a jaca od b
int put[maxn][maxn]; // ako je put[a][b] != INF tada postoji put od a do b.
void prosiri() {
  for (int k = 0; k < N; ++k)
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j)
        // ako imam kraci put izmedju i,j preko k
        // osvjezavam duljinu tog puta i zapisujem
        // u c[i][j] precizniju vrijednost za odnos izmedju i,j
        if (put[i][j] > put[i][k] + put[k][j]) {
          put[i][j] = put[i][k] + put[k][j];
          c[i][j] = c[i][k] * c[k][j];
        }
```

```
}
```

Dakle algoritam za traženje najkraćeg puta smo modificirali tako da uz svaki najkraći put izračunamo i umnoške svih vrijednosti iz polja .

Ovo rješenje je ujedno i najpreciznije zato što koristimo minimalno operacija množenja od kojih svaka unosi određenu grešku u rješenje.

Nakon proširenja grafa za svaki upit ***(a, b)*** ispitujemo vrijednost $put_{a,b}$ te ispisujemo $c_{a,b}$ u slučaju da postoji put i **-1** u suprotnom.

Memorijska složenost rješenja je $O(n^2)$, a vremenska složenost je $O(n^3)$.

**Druga rješenja**

Problem je bilo moguće rješiti i koristeći algoritam BFS (engl. breadth first search) tako da se proširimo samo iz onih vrhova čija nas rješenja zanimaju. U najgorem slučaju, zanimati će nas odnosi svih vrhova pa će se BFS zvati **N** puta što se opet svodi na vremensku složenost od $O(n^3)$.

Rješenja koja koriste algoritam DFS (engl. depth first search) matematički daju točna rješenja, no zbog toga što neće koristiti najkraći put između dvije valute postoji mogućnost da će zbog akumulacije greške prikaza decimalnog broja u računalu odstupanje biti preveliko.

U zadatku izomorfa bilo je potrebno ispitati izomorfnost dvaju zadanih neusmjerenih jednostavnih grafova.

**Pojmovi**

Jednostavan graf je onaj graf u kojem svaka veza spaja dva **različita** vrha te između svaka dva vrha postoji najviše jedna veza. Relacija prijateljstvo se najintuitivnije prikazuje neusmjerenim jednostavnim grafom.

U daljnjem tekstu riječ graf ćemo koristiti kao sinonim za **jednostavan neusmjeren graf**. Svaki takav graf ima binarnu matricu susjedstva koja je simetrična i ima nule na glavnoj dijagonali.

Dva grafa su **identični** ukoliko su im matrice susjedstva identične.

Dva grafa **A** i **B** su **izomorfna** ako je svaki vrh u prvom grafu moguće preimenovati u jedinstven vrh iz drugog grafa na način da grafovi nakon toga postaju identični.

**Algoritam**

Budući da je veličina grafa u zadatku svega 10, rješenje možemo dobiti iscrpnom pretragom. Preimenovanje vrhova iz jednog grafa možemo prikazati kao permutaciju niza **{1, 2, ..., N}**. Za svako preimenovanje provjeravamo jesu li prvi graf i preimenovani drugi graf identični.

```
int graph[2][maxn][maxn];
int idx[maxn];

int check() {
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      if (graph[0][i][j] != graph[1][idx[i]][idx[j]])
        return 0;
    return 1;
}

int solve() {
  for (int i = 0; i < N; ++i)
    idx[i] = i;
  do {
    if (check()) return 1;
  } while (next_permutation(idx, idx + N));
  return 0;
}
```

Vremenska složenost je $\mathbf{O(n^2 \bullet n!)}$.

U zadatku Sutomi potrebno je ispisati broj puteva koji vode to smrti Super Tomija.

**Algoritam**

Prvo primjećujemo da novi put ne može nastati na mjestima gdje nema ili početka ili kraja daske. Dakle, bitni su nam samo rubovi dasaka.

Promatrajmo 2 slučaja:

1. Daska započinje

**Y** = visina daske koja je započela

Ako je Tomi na dasci ispod, može ili skočiti na dasku ako je bio na visini **Y - 1**, ili nastaviti šetati na **Y – 1** visini.

2. Daska završava

Tomi nema pretjerano izbora, mora pasti. Pada ili na neku dasku ili u ponor.

To navodi na **dinamičke relacije**. Probajmo konstruirati dinamičke relacije.

Primijetimo prije pokušavanja kreiranja dinamike da je 2. slučaj malo problematičan. Moram znati u svakom trenutku dasku ispod mene. Trpajmo sve daske koje trenutno imamo u **set S**. Ujedno primjećujemo da zbilja nema smisla gledati hoće li Tomi pasti na dasku koja je već odavno završila. Zaključak: u 1. dodajemo dasku u set, a u 2. brišemo dasku iz seta.

**X, Y** = koordinate trenutnog ruba daske koji promatramo

**S** = set dasaka

1.
***DP(X, Y) = DP(X - 1, Y - 1)***

***DP(X, Y - 1) = DP(X - 1, Y - 1)***

ubaci u S trenutnu dasku

2. Postoje dvije mogućnosti:

2.1. Ne postoji daska ispod trenutne. To znači da Tomi pada u ponor. Dakle, možemo samo povećati rješenje za ***DP(X - 1, Y)***

2.2. postoji daska ispod trenutne

***DP(X, Y od daske to Tomija (za to pitamo S)) = DP(X - 1, Y)***

Ova nije potpuna, jer Tomi može s različitih visina pasti na istu dasku u istoj točki, pa se mora povećavati umjesto izjednačavati.

***DP(X, Y od daske to Tomija (za to pitamo S)) += DP(X - 1, Y)***

Primjetimo da kad bismo računali dinamiku **iterativno**, ne moramo se gnjaviti s moram li pisati = ili +=, već stalno pišemo += .Već smo primjetili da možemo gledati samo rubove dasaka, ali koje rubove bismo trebali gledati prve ?

Ako dvije daske nastaju u istom trenu (pod tren se misli X koordinata), prvo moramo obraditi gornju. Ako dvije daske završavaju u istom trenu, svejedno nam je pa recimo da ćemo prvo obraditi gornju. Objašnjenje: svi padovi vode do iste situacije, a ako Tomi prvo padne na dasku ispod pa sa nje padne na neku dasku koja ne završava, opet isto.

Ako 1. daska počinje, a 2. završava, prvo obradimo 1. dasku jer Tomi može pasti sa 2. na 1., a ako nismo obradili 1., pasti će u ponor.

Po tim pravilima možemo sortirati eventove (rubovi dasaka) te ih obrađivati po redu.

Vremenska složenost jedne obrade: ***O(log(n))*** (zbog seta)

Broj eventova: ***2n***

Vremenska složenost: ***O(n log(n))***

**Potrebna predznanja**

- dinamičko programiranje
- sweep line metoda