Županijsko natjecanje iz informatike

Srednja škola

Druga podskupina (3. i 4. razred)

9. veljače 2018.

RJEŠENJA ZADATAKA

Napomena: kodovi za većinu opisanih algoritama dani su u Pythonu radi jednostavnosti i lakše čitljivosti. Zbog prirode jezika, rješenja u Pythonu sporija su od ekvivalentnih rješenja u C++u pa natjecateljima preporučujemo da u slučaju potrebe za bržim rješenjem koriste C++.

Ako su odabrani dani s rednim brojevima A, B i C, na kraju ćemo imati

10 / cijena[A] * cijena[B] / cijena[C]

ethereuma.

Isprobavajući sve moguće trojke A, B, C (koristeći tri for petlje jednu unutar druge) dobivamo složenost $O(N^3)$, što je presporo za veće test podatke.

Brže rješenje u $O(N^2)$ možemo dobiti isprobavanjem svih odabira A i B (dvije for petlje) ako znamo brzo naći element s najmanjom vrijednošću cijena[C] desno od odabranog B, jer se očito isplati podijeliti sa što manjim brojem. To možemo provesti ako, prije isprobavanja odabira, izračunamo tražene minimume za svaki B. Preciznije, prolazeći nizom s desna na lijevo pamtimo dosadašnji minimum i spremamo ga u pomoćni niz na trenutačnu poziciju B.

Za očekivano rješenje složenosti O(N) birat ćemo samo A. Potrebno je stoga unaprijed izračunati maksimalan omjer cijena[B] / cijena[C] desno od odabranog A. Taj pomoćni niz konstruiramo prolazeći nizom s desna na lijevo, pamteći dosadašnji maksimalan omjer i uspoređujući ga s onim koji dobivamo birajući trenutačni B i minimalni C desno od njega (a taj je spremljen u pomoćnom nizu iz prethodnog odlomka).

```
#!/usr/bin/python3
n = int(input())
a = [float(input()) for i in range(n)]

min_desno = [0 for i in range(n - 1)] + [a[n - 1]]
for i in range(n - 2, -1, -1):
    min_desno[i] = min(min_desno[i + 1], a[i])

max_omjer_desno = [0 for i in range(n)]
for i in range(n - 2, -1, -1):
    max_omjer_desno[i] = max(max_omjer_desno[i + 1], a[i] / min_desno[i + 1])

for i in range(n - 2):
    print(10 / a[i] * max_omjer_desno[i + 1])
```

U ulaznim podatcima zapravo nam je dano nekoliko komponenata hijerarhije koje možemo na različite načine povezati u jedno veliko stablo koje predstavlja hijerarhiju poduzeća. Načelno postoje dva slučaja za traženu najveću udaljenost:

A. ona se nalazi unutar jedne od zadanih komponenata (podstabala),
B. ili je riječ o udaljenosti stažista u različitim komponentama.

Za slučaj A, za svako manje stablo treba izračunati najveću udaljenost stažista unutar njega, što činimo rekurzivno računajući najprije visine svih vrhova podstabla. Potom za svakog mogućeg „zajedničkog šefa" zbrajamo dvije najveće visine njegove djece, pamteći maksimum takvih putova.

U drugom slučaju tražimo dva podstabla s najvećom visinom i gledamo koliko ona mogu biti međusobno udaljena u cijeloj hijerarhiji. U minimalnom slučaju bit će jedan do drugoga, pa će u tom slučaju najveća udaljenost njihovih stažista biti zbroj njihovih visina. U maksimalnom slučaju, između njih će na putu biti svi ne-stažisti, koje treba pribrojiti u najveću udaljenost. Na taj način dobivamo interval [B_min, B_max] za ovaj slučaj.

Na kraju uspoređujemo rezultate slučajeva A i B. Ako je udaljenost A veća ili jednaka B_max, rješenje je A. Inače najveća udaljenost pripada slučaju B. Tada ispisujemo -1 ako interval [B_min, B_max] sadrži više od jednog broja (tj. ako su min i max različiti), a inače ispisujemo B_min (ili B_max).

```
#!/usr/bin/python3
import sys
sys.setrecursionlimit(10**6)
n, m = map(int, input().split())
staz = [False for i in range(n)]
djeca = [[] for i in range(n)]
for i in range(m):
    x, y = map(int, input().split())
    staz[x - 1] = True
    djeca[y - 1].append(x - 1)

visina = [0 for i in range(n)]
max_put = 0

def dfs(i):
    global max_put
    a, b = -1, -1
    for y in djeca[i]:
        if b < dfs(y):
            b = visina[y]
            if a < b:
                a, b = b, a
    visina[i] = max(visina[i], 1 + a)
    if len(djeca[i]) == 1:
        max_put = max(max_put, visina[i] - 2)
```

```
    elif len(djeca[i]) > 1:
        max_put = max(max_put, a + b + 1)
    return visina[i]

a, b = -1, -1
for i in range(n):
    if staz[i] or len(djeca[i]) == 0:
        continue
    if b < dfs(i):
        b = dfs(i)
        if a < b:
            a, b = b, a

if b == -1:
    print(max_put)
    exit(0)

d_min = a + b
d_max = d_min - 2 + n - m
if max_put >= d_max:
    print(max_put)
elif d_min == d_max:
    print(d_min)
else:
    print(-1)
```

Ovaj zadatak rješavamo *brute-force* algoritmom, tj. isprobavanjem svih nizova poteza od kraćih prema duljima, uz pažnju da nizove koje smo već eliminirali ne isprobavamo dalje. Moguće su razne varijante implementacije, s rekurzijom ili bez rekurzije.

Provjeru je li igrač pobijedio možemo elegantno napraviti šireći se iz tek ubačenog žetona u svim smjerovima i brojeći uzastopne žetone njegove boje. Za širenje u svim smjerovima koristimo pomoćnu funkciju koja prima pomake po retku i stupcu (0, -1 ili 1) za odgovarajući smjer.

Rješenje u C++11:

```
#include <cstdio>
#include <vector>
#include <cstdlib>

const int N = 6;
const int M = 7;
const std::vector<int> NO_SOLUTION(8);

char check(const char board[N][M + 1], int y, int x, char player) {
  auto connected = [&board, &player](int y, int x, int dy, int dx) {
    int count = 0;
    while (y >= 0 && y < N && x >= 0 && x < M && board[y][x] == player &&
           count <= 4) {
      ++count;
      y += dy;
      x += dx;
    }
    return count;
  };

  if (connected(y, x, 1, 0) + connected(y, x, -1, 0) >= 5) return player;
  if (connected(y, x, 0, 1) + connected(y, x, 0, -1) >= 5) return player;
  if (connected(y, x, 1, 1) + connected(y, x, -1, -1) >= 5) return player;
  if (connected(y, x, 1, -1) + connected(y, x, -1, 1) >= 5) return player;
  return '0';
}

void solve(std::vector<int> &moves, char board[N][M + 1],
           std::vector<int> &win_moves) {
  if (moves.size() == 7U || moves.size() >= win_moves.size()) return;
  for (int i = 0; i < M; ++i) {
```

```
    int j = N - 1;
    while (j >= 0 && board[j][i] != '0') --j;
    if (j == -1) continue;
    moves.push_back(i);
    board[j][i] = (moves.size() & 1U) ? '1' : '2';
    auto winner = check(board, j, i, board[j][i]);
    if (winner == '1' && moves.size() < win_moves.size()) {
      win_moves = moves;
    }
    if (winner == '0') {
      solve(moves, board, win_moves);
    }
    moves.pop_back();
    board[j][i] = '0';
  }
}

int main() {
  char board[N][M + 1];
  for (int i = 0; i < N; ++i) {
    scanf("%s", board[i]);
  }

  std::vector<int> moves;
  std::vector<int> win_moves = NO_SOLUTION;
  solve(moves, board, win_moves);
  for (auto &move : win_moves) {
    printf("%d", move + 1);
  }
  printf("\n");
  return 0;
}
```