P-III-4 Tlustý Santa

Hledání nejkratší cesty v grafu

Kdykoliv máme určit nejmenší počet kroků, jimiž lze dosáhnout nějakého cíle, měli bychom uvažovat o možnosti sestrojit vhodný graf a v tomto grafu nalézt nejkratší cestu. Graf vznikne pokaždé stejně: jeho vrcholy odpovídají stavům, v nichž se můžeme nacházet, a hrany vycházející z vrcholu odpovídají tahům, které můžeme v dané situaci provést.

Jak to bude vypadat v našem případě? Stavy, tedy vrcholy grafu, budou odpovídat jednotlivým pozicím, na nichž se Santa může nacházet. Přesněji, budeme uvažovat pozice, na kterých může ležet levý horní roh Santy – tím je jeho poloha jednoznačně určená. Každý vrchol grafu můžeme jednoznačně popsat dvěma čísly: jeho souřadnicemi. Protože $r_1, s_1 \leq 2\,000$, náš graf bude mít nejvýše 4 miliony vrcholů.

Z každého vrcholu povedou hrany odpovídající krokům, které v té chvíli může Santa provést. Na výběr máme jen 4 směry pohybu, takže z libovolného vrcholu budou vycházet nejvýše 4 hrany.

Abychom věděli, jaké hrany v grafu máme, potřebujeme zjistit, na kterých pozicích může Santa stát a na kterých (kvůli překážkám) stát nemůže. Tím se budeme zabývat později. Zatím předpokládejme, že to umíme zjistit, a vraťme se zpět k našemu grafu.

Když už máme sestrojený graf, zbývá v něm nalézt nejkratší cestu. K tomu použijeme algoritmus prohledávání do šířky (breadth-first search, BFS). Toto prohledávání představuje způsob, jak by se v daném grafu šířila voda z počátečního vrcholu rovnoměrně do všech směrů. Nejprve zpracujeme počáteční vrchol (vzdálenost 0), potom postupně všechny jeho sousedy (každý z nich má vzdálenost 1 od počátečního vrcholu), potom sousedy jeho sousedů (vzdálenost 2), atd.

Prohledávání do šířky snadno implementujeme pomocí fronty:

```
prohledej(vrchol v):
    Označ v jako navštívený.
    Vzdálenost do v = 0.
    Vytvoř frontu Q obsahující jediný prvek v.
    Dokud fronta Q není prázdná:
        Vyber ze začátku fronty Q prvek x.
        Pro každou hranu vedoucí z x:
            Jestliže její koncový vrchol w není navštívený:
                Označ w jako navštívený.
                Vzdálenost do w = 1 + vzdálenost do v.
                Zapamatuj si, že do w jsme se dostali z x.
                Vlož w na konec fronty Q.
```

Časová i paměťová složitost tohoto řešení je lineární vzhledem k velikosti místnosti, tedy $\mathcal{O}(r_1 s_1)$. Proč tomu tak je? V první řadě si uvědomíme, že máme $r_1 s_1$

vrcholů a nejvýše $4r_1s_1$ hran, tedy dohromady máme $\mathcal{O}(r_1s_1)$ vrcholů a hran. Každý vrchol nejvýše jednou vložíme do fronty (v okamžiku, když se do něj poprvé dostaneme) a nejvýše jednou ho potom z fronty vyzvedneme. Každou hranu také zpracujeme nejvýše jednou: tehdy, když zpracováváme vrchol, z něhož tato hrana vede. Celkový počet kroků, které náš algoritmus vykoná, je proto přímo úměrný počtu vrcholů a hran v grafu.

Při implementaci si nepotřebujeme samostatně značit, které vrcholy jsou již navštívené. Poznáme to jednoduše tak, že navštívený vrchol má vzdálenost jinou než nekonečno.

Rekonstrukce cesty

V naší úloze máme také vypsat, kudy vede nalezená nejkratší cesta. Pro každé políčko si proto zapamatujeme, odkud jsme se na něj během prohledávání poprvé dostali. Následně můžeme pomocí těchto údajů od konce zrekonstruovat cestu.

Předvýpočet

Před spuštěním vlastního prohledávání do šířky potřebujeme zjistit, na které pozice může Santa vstoupit. Jsou to takové pozice, na kterých Santa překryje nejvýše M překážek (tolik jich dokáže pomocí magie schovat.)

Do tabulky T si chceme na políčko $T[y, x]$ zapsat, kolik překážek je v obdélníku $[y, x]$ až $[y + r_2 - 1, x + s_2 + 1]$. Jestliže $T[y, x] \leq M$, Santa může vstoupit na pozici y, x.

Jednoduchým řešením je pro každé políčko se podívat na $r_2 \times s_2$ políček vpravo dolů od něho a spočítat překážky. Toto řešení ale vykoná celkem $\Theta(r_1s_1r_2s_2)$ kroků, což je pro velké vstupy příliš.

Při hledání rychlejšího řešení se zkusíme nejprve podívat na jednorozměrný případ. V jednorozměrném případě nám stačí spočítat, kolik překážek je na políčkách $[y, x]$ až $[y, x + s_2 - 1]$. To vypočítáme postupně pro každé x:

```
T[y,0] = počet překážek od [y,0] do [y,s2-1]
T[y,1] = T[y,0] + (1 když [y,s2] je 'X') - (1 když [y,0] je 'X')
...
T[y,i] = T[y,i-1] + (1 když [y,i+s2-1] je 'X') - (1 když [y,i-1] je 'X')
..
T[y,s1-s2] = T[y,s1-s2-1] + (1 když [y,s1-1] je 'X') - (1 když [y,s1-s2-1] je 'X')
```

Výpočet $T[y, 0]$ nám sice bude trvat s_2 kroků, ale potom už každé ze zbývajících $s_1 - s_2$ políček spočítáme v konstantním čase. Celkem tedy provedeme při zpracovaní jednoho řádku jen $\mathcal{O}(s_1)$ kroků.

Dále je to už jednoduché. Pokud nám 6 bodů nestačí, stačí si uvědomit, že dvojrozměrný případ vyřešíme tak, že provedeme ještě jednou totéž, ale tentokrát po sloupcích. Přitom použijeme hodnoty, které jsme právě vypočítali průchodem po řádcích. Tento předvýpočet zvládneme v čase $\mathcal{O}(r_1s_1)$. Totéž platí i pro BFS, takže dostáváme řešení se zjevně optimální časovou složitostí $\mathcal{O}(r_1s_1)$. Paměťová složitost je také $\mathcal{O}(r_1s_1)$.

```
#include <cstdio>
#include <algorithm>
#include <vector>
#include <queue>
using namespace std;
#define INF 1023456789

int R,S,r,s,M;
char v;
int vstup[3][2047][2047];
int dist[2047][2047];
int prev[2047][2047];

int dy[] = {1,-1,0,0};
int dx[] = {0,0,1,-1};
char dc[] = "JSVZ";

int main() {
    scanf("%d %d %d %d %d", &R, &S, &r, &s, &M);
    // načtení vstupu a vytvoření tabulky
    for(int i = 0; i<R; ++i)
        for(int j = 0; j<S; ++j) {
            scanf(" %c", &v);
            vstup[0][i][j] = (v=='X')?1:0;
            dist[i][j] = INF;
        }
    int sum;
    for(int i = R-1; i>=0; --i) {
        for(int j = S-1; j>=0; --j) {
            vstup[1][i][j] = vstup[1][i][j+1] + vstup[0][i][j]
                             - ((j+s<S) ? vstup[0][i][j+s] : 0);
            vstup[2][i][j] = vstup[2][i+1][j] + vstup[1][i][j]
                             - ((i+r<R) ? vstup[1][i+r][j] : 0);
        }
    }

    // BFS
    queue<int> F;
    dist[0][0] = 0;
    F.push(0);
    F.push(0);
    int x,y, xx, yy;
    while(!F.empty()) {
        y = F.front(); F.pop();
        x = F.front(); F.pop();
        if (y==R-r && x==S-s)  {
            // zrekonstruování cesty
            vector<char> vc;
            while( y!=0 || x!=0 ) {
                vc.push_back(dc[prev[y][x]]);
                yy = y-dy[prev[y][x]];
                x = x-dx[prev[y][x]];
                y = yy;
            }
            for(int i = vc.size()-2; i>=0; --i)
                printf("%c",vc[i]);
            printf("\n");
```

```
            return 0;
        }
        for(int d = 0; d<4; d++) {
            yy = y+dy[d];
            xx = x+dx[d];
            // jsem vně místnosti, je tam moc překážek nebo jsem tam už byl
            if (xx<0 || yy<0 || xx>S-s || yy>R-r ||
                vstup[2][yy][xx]>M || dist[yy][xx]<=dist[y][x]+1) continue;

            dist[yy][xx] = dist[y][x]+1;
            prev[yy][xx] = d;
            F.push(yy);
            F.push(xx);
        }
    }
    printf("Santa je chudak.\n");
    return 0;
}
```

Jiné řešení

Místo předvýpočtu výše uvedeným způsobem můžeme přímo použít dvojrozměrné prefixové součty: pro každé y, x určíme počet překážek $P[y, x]$ v obdélníku, který má v protilehlých rozích políčka $[0, 0]$ a $[y-1, x-1]$. To dokážeme vhodným trikem spočítat v čase $\mathcal{O}(r_1 s_1)$. Pomocí těchto hodnot pak už umíme v konstantním čase o libovolném obdélníku říci, kolik překážek obsahuje.

```
Předvýpočet:
    P[y,x] = P[y-1,x] + P[y,x-1] - P[y-1,x-1] + (1 když [y-1,x-1] je 'X')
Počet překážek v obdélníku od [y1,x1] do [y2-1,x2-1]:
    počet = P[y2,x2] - P[y2,x1] - P[y1,x2] + P[y1,x1]
```

P-III-5 Úřad

Naším cílem samozřejmě je nalézt řešení, které bude efektivnější než přímočará simulace každého příkazu. Struktura ministerstva popsaná na vstupu představuje strom. A protože na obecném stromě se předepsané operace provádějí obtížně, prvním krokem řešení bude převedení úlohy na trochu jinou – na operace s intervaly a s prvky v poli. Každému vrcholu stromu tedy přiřadíme jeden prvek pole tak, aby každému oddělení v úřadu odpovídal souvislý úsek pole. Příkaz se potom bude týkat buď jednoho prvku pole, nebo nějakého intervalu.

Vhodné přiřazení pozic v poli vrcholům stromu získáme snadno: stačí libovolným způsobem prohledat strom do hloubky a každému vrcholu přiřadit index rovný pořadí, v němž byl poprvé navštíven. (Existuje více vyhovujících očíslování, toto je jen jedno z nich, které lze snadno a systematicky sestrojit.)

Během prohledávání si zároveň můžeme pro každý vrchol zapamatovat, kde začíná a kde končí interval, který obsahuje jeho podřízené. (Při prohledávání do hloubky interval odpovídající vrcholu v začíná tímto vrcholem samotným. Konec intervalu zjistíme tehdy, když prohledávání opouští vrchol v.)

Tuto část řešení dokážeme vykonat v čase $\mathcal{O}(n)$.

V tuto chvíli tedy už můžeme na strom zapomenout, máme jen obyčejné pole. Abychom mohli provádět příkazy ze zadání, potřebujeme s ním efektivně provádět následující operace:

- Zvyš hodnotu prvku a o k
- Zjisti součet v intervalu $[a, b]$.
- Zvyš hodnotu všech prvků v intervalu $[a, b]$ o k.
- Zjisti minimum v intervalu $[a, b]$.
- Změň hodnotu všech prvků v intervalu $[a, b]$ na k.

Intervalové stromy

Začneme efektivní implementací prvních dvou operací. (Za to získáme 9 bodů. Stačí navíc doplnit, že vždy, když budeme potřebovat zvýšit mzdu celému oddělení, projdeme všechny zaměstnance oddělení a každému zvýšíme jeho mzdu.)

Nad polem si postavíme binární strom. Hodnota každého vrcholu bude rovna součtu hodnot jeho synů, tedy zároveň součtu v celém intervalu pole, který je pokryt tímto vrcholem.

Změnu hodnoty implementujeme snadno: změníme hodnotu v poli a potom postupujeme strommeme nahoru a upravujeme hodnotu všech předků daného vrcholu. Časová složitost je úměrná hloubce tohoto stromu, tzn. $\mathcal{O}(\log n)$. Zjištění součtu v úseku bude trochu komplikovanější, ale stejně rychlé. Potřebujeme sečíst ty vrcholy stromu, které interval pokrývá celé a zároveň nepokrývá jejich otce. Těchto vrcholů bude maximálně $\mathcal{O}(\log n)$ – uvědomte si, že z každé úrovně stromu sčítáme maximálně dva vrcholy. Najdeme je jednoduchou rekurzivní procedurou, která prochází strom shora dolů:

```
zjisti_součet(vrchol x, sčítaný interval I):
  Je-li interval I disjunktní s intervalem, který pokrývá podstrom ve vrcholu x:
    Vrať 0.
  Je-li interval I nadmnožinou intervalu, který pokrývá podstrom ve vrcholu x:
    Vrať součet ve vrcholu x.
  Vrať zjisti_součet(levý syn x, I) + zjisti_součet(pravý syn x, I).
```

Tento průchod má také časovou složitost $\mathcal{O}(\log n)$. Analogicky, pomocí rekurze, můžeme implementovat i první operaci: začneme v kořeni stromu, sestoupíme dolů na správné políčko pole, to upravíme a při vynořování se z rekurze přepočítáme hodnoty v jeho předcích. U tohoto řešení to ještě není podstatné, ale právě takováto rekurzivní implementace bude potřebná při modifikacích popsaných v následující části řešení.

Líné provádění operací

Nyní si ukážeme, jak efektivně provádět zbývající operace. Začneme třetí operací a na ní si ukážeme princip řešení. Zbývající dvě operace potom už dořešíme analogicky.

Hlavní myšlenkou bude „nedělej nic předčasně“. Představte si, že chceme zvýšit o k všechny hodnoty v intervalu, který odpovídá nějakému vrcholu v našeho binárního stromu. Místo toho, abychom procházeli celý podstrom a zvyšovali každý jeho

prvek, jednoduše si ve vrcholu v upravíme jeho součet (to můžeme udělat hned) a poznačíme si: „všechny prvky pod tímto vrcholem je třeba o k zvýšit“. Jeho potomky tedy zatím necháme tak, budou mít dočasně nesprávné hodnoty. To nám ale nevadí – jestliže někdy v budoucnosti budeme chtít přistupovat k potomkům vrcholu v, musíme přitom projít vrcholem v. No a právě až v tomto okamžiku vykonáme zaznamenaný příkaz. Přesněji, pokaždé když chceme z nějakého vrcholu u sestupovat po stromu dolů, nejdříve se podíváme, zda v u nemáme „odloženou“ nějakou operaci. Pokud ano, nejprve tuto operaci přesuneme z u do obou jeho synů, a až potom se posuneme o krok níže (a tam celý postup opakujeme, je-li třeba).

Obecně bude zvyšování intervalu o x vypadat podobně jako zjišťování součtu. Jenom do všech třech operací (zvyšování prvku, zvyšování intervalu, zjišťování součtu intervalu) musíme doplnit část, v níž zkontrolujeme a případně vyřešíme odložené operace. Potřebujeme také umět „skládat“ zvýšení: máme-li již v nějakém vrcholu zaznamenané zvýšení o a a přijde další zvýšení o b, jednoduše změníme zaznamenanou hodnotu na $a + b$.

Zbývají poslední dvě operace – zjišťování minima a úprava intervalu na konkrétní hodnotu. Zjišťování minima bude jednoduché: v každém vrchole si budeme udržovat navíc minimum příslušného intervalu. Úprava intervalu na konkrétní hodnotu proběhne podobně jako zvýšení celého intervalu o k. Provedeme ji líně jen ve vrcholech, které jsou celé pokryty intervalem, který měníme. Když provádíme tuto operaci a náhodou se ve vrcholu předtím objevila operace přidání, tuto starší operaci zahodíme (neboť nová operace všechny hodnoty stejně přepíše). Naopak, máme-li odloženou změnu na hodnotu a a přijde nám zvýšení o b, můžeme si například změnit zapamatovanou operaci na „změň hodnotu na $a + b$“.

Díky línému provádění dokážeme každou operaci vykonat v čase $\mathcal{O}(\log n)$.

```
#include <cstdio>
#include <vector>
#include <algorithm>

using namespace std;

#define INF 200000000000000000ll

int n;
vector<vector<int> > sons;
vector<int> salary;
vector<long long> sizes;

// Interval <a,b>
vector<pair<int, int> > postorder;

int postord(int x, int num) {
  int last = num;
  int size = 1;
  for (int i = 0; i < sons[x].size(); i++) {
    last = postord(sons[x][i], last);
    size += sizes[sons[x][i]];
  }
  sizes[x] = size;
  postorder[x] = make_pair(num, last);
```

```
  return last+1;
}

int is = 131072;

typedef long long ll;

struct Inter {
  ll sum;
  ll min;

  ll pending_change;
  ll pending_add;

  Inter() : sum(0), min(0), pending_change(-1), pending_add(0) {}
};

Inter strom[1234567];

pair<ll, ll> add(ll a, int my_begin, int my_end, int a_begin, int a_end, int v);
pair<ll, ll> change(ll ch, int my_begin, int my_end, int a_begin, int a_end, int v);

void push_down(int my_begin, int my_end, int v) {
  if (strom[v].pending_change != -1) {
    change(strom[v].pending_change, my_begin, (my_begin+my_end)/2,
           my_begin, my_end, 2*v);
    change(strom[v].pending_change, (my_begin+my_end)/2, my_end,
           my_begin, my_end, 2*v+1);
    strom[v].pending_change = -1;
  }
  if (strom[v].pending_add != 0) {
    add(strom[v].pending_add, my_begin, (my_begin+my_end)/2,
           my_begin, my_end, 2*v);
    add(strom[v].pending_add, (my_begin+my_end)/2, my_end,
           my_begin, my_end, 2*v+1);
    strom[v].pending_add = 0;
  }
}

// Vrací <součet, minimum>
pair<ll, ll> add(ll a, int my_begin, int my_end, int a_begin, int a_end, int v) {
  if (my_begin >= a_end || my_end <= a_begin)
    return make_pair(strom[v].sum, strom[v].min);

  if (my_begin >= a_begin && my_end <= a_end) {
    strom[v].pending_add += a;
    strom[v].min += a;
    strom[v].sum += (my_end - my_begin) * a;
    return make_pair(strom[v].sum, strom[v].min);
  }

  push_down(my_begin, my_end, v);

  pair<ll, ll> r1 = add(a, my_begin, (my_end+my_begin)/2, a_begin, a_end, 2*v);
  pair<ll, ll> r2 = add(a, (my_end+my_begin)/2, my_end, a_begin, a_end, 2*v+1);
  strom[v].sum = r1.first+r2.first;
  strom[v].min = min(r1.second, r2.second);
  return make_pair(strom[v].sum, strom[v].min);
}

void add(ll a, int a_begin, int a_end) {
```

```
  add(a, 1, is, a_begin, a_end, 1);
}

pair<ll, ll> change(ll ch, int my_begin, int my_end, int a_begin, int a_end, int v) {
  if (my_begin >= a_end || my_end <= a_begin)
    return make_pair(strom[v].sum, strom[v].min);

  if (my_begin >= a_begin && my_end <= a_end) {
    strom[v].pending_add = 0;
    strom[v].pending_change = ch;
    strom[v].min = ch;
    strom[v].sum = ch*(my_end - my_begin);
    return make_pair(strom[v].sum, strom[v].min);
  }

  push_down(my_begin, my_end, v);

  pair<ll, ll> r1 = change(ch, my_begin, (my_end+my_begin)/2, a_begin, a_end, 2*v);
  pair<ll, ll> r2 = change(ch, (my_end+my_begin)/2, my_end, a_begin, a_end, 2*v+1);
  strom[v].sum = r1.first+r2.first;
  strom[v].min = min(r1.second, r2.second);
  return make_pair(strom[v].sum, strom[v].min);
}

void change(ll ch, int a_begin, int a_end) {
  change(ch, 1, is, a_begin, a_end, 1);
}

pair<ll, ll> get(int my_begin, int my_end, int a_begin, int a_end, int v) {
  if (my_begin >= a_end || my_end <= a_begin)
    return make_pair(0, INF);
  if (my_begin >= a_begin && my_end <= a_end) {
    return make_pair(strom[v].sum, strom[v].min);
  }

  push_down(my_begin, my_end, v);

  pair<ll, ll> r1 = get(my_begin, (my_end+my_begin)/2, a_begin, a_end, 2*v);
  pair<ll, ll> r2 = get((my_end+my_begin)/2, my_end, a_begin, a_end, 2*v+1);
  return make_pair(r1.first + r2.first, min(r1.second, r2.second));
}

ll get_min(int a_begin, int a_end) {
  return get(1, is, a_begin, a_end, 1).second;
}

ll get_sum(int a_begin, int a_end) {
  return get(1, is, a_begin, a_end, 1).first;
}

int main() {
  scanf("%d", &n);
  salary.resize(n);
  sons.resize(n);
  sizes.resize(n);
  postorder.resize(n);
  for (int i = 0; i < n; i++) {
    int p;
    scanf("%d %d", &salary[i], &p);
    sons[i].resize(p);
```

```
    for (int j = 0; j < p; j++) {
      scanf("%d", &sons[i][j]);
      sons[i][j]--;
    }
  }
  postord(0, 1);
  for (int i = 0; i < n; i++)
    change(salary[i], postorder[i].second, postorder[i].second+1);

  int q; scanf("%d", &q);

  for (int o = 0; o < q; o++) {
    int a, u, x, y;
    scanf("%d %d", &a, &u);
    if (a != 3) scanf("%d %d", &x, &y);
    u--;
    if (a == 1) {
      if (get_sum(postorder[u].second, postorder[u].second+1) < x) {
        add(y, postorder[u].second, postorder[u].second+1);
      }
    } else if (a == 2) {
      if (get_sum(postorder[u].first, postorder[u].second+1) < x * sizes[u]) {
        add(y, postorder[u].first, postorder[u].second+1);
      }
    } else {
      ll mi = get_min(postorder[u].first, postorder[u].second+1);
      change(mi, postorder[u].first, postorder[u].second+1);
    }
  }
  for (int i = 0; i < n; i++)
    printf("%lld\n", get_min(postorder[i].second, postorder[i].second+1));
  return 0;
}
```