Korešpondenčný seminár z programovania
XXXIII. ročník, 2015/16
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. série letnej časti

Mário

1. Zjedená pizza

(max. 6 b za popis, 4 b za program)

Aby sme vyriešili túto úlohu, stačí nám robiť to, čo robil Adam v rozprávke v zadaní. Začneme od najväčších krabíc – vždy do nich vložíme toľko menších, koľko sa zmestí, pričom zvyšné krabice musia zostať vonku.

Aby sme si v našom riešení udržali poriadok, v jednej premennej `mozem_zabalit` si budeme pamätať, koľko aktuálne spracúvaných krabíc vieme zabaliť do väčších. Môžeme si všimnúť, že aj keď sú v krabici 8×8 štyri krabice 4×4, stále sa tam zmestí 16 krabíc 2×2. Teda väčšie krabice neovplyvňujú tie menšie, ak sú spoločne v jednej krabici. Na spočítanie, koľko menších krabíc vieme zabaliť, nám preto stačí vynásobiť premennú `mozem_zabalit` štyrmi .

Postupovať budeme od najväčších krabíc po najmenšie a v každom kroku bude našou úlohou „zabaliť krabice s hranou dĺžky 2^i". Zakaždým spočítame, koľko takýchto krabíc musí zostať vonku a koľko miesta bude pre menšie krabice.

Krabíc zabalíme najviac `mozem_zabalit`. Ak sa všetky krabice zmestia do väčších, veľkosť voľného miesta pre ešte menšie krabice bude `4 * mozem_zabalit`. Ak sa niektoré krabice nezmestia dnu, zostávajú vonku a teda ich pripočítame ku `krabice_vonku`. Tiež ale vytvoria nový priestor pre ešte menšie krabice, teda menších krabíc budeme vedieť zabaliť `4 * (mozem_zabalit + ostali_vonku)`.

Krabice jednej veľkosti „zabalíme" v konštantnom čase $O(1)$, teda ak máme k veľkostí krabíc, riešenie bude potrebovať $O(k)$ času. Keďže vstup spracúvame odzadu, musíme ho celý načítať do poľa, teda aj pamäťová zložitosť bude $O(k)$. V tejto úlohe boli ale všetky vstupy s $k = 20$, tak ste mohli tvrdiť, že beh programu nezávisí od veľkosti čísel na vstupe a prehlásiť časovú zložitosť za konštantnú (takéto tvrdenie je ale vždy potrebné vysvetliť, a predsalen, časový odhad $O(k)$ je informatívnejší).

Listing programu (C++)

```
#include <iostream>

using namespace std;

int main(){
  int pocty[20];
  for(int i = 0; i < 20; i++)
    cin >> pocty[i];
  int krabice_vonku = 0;
  long long mozem_zabalit = 0;

  for(int i = 19; i >= 0; i--){
    if (pocty[i] > mozem_zabalit){
      krabice_vonku += pocty[i] - mozem_zabalit;
      mozem_zabalit += pocty[i] - mozem_zabalit;
    }
    mozem_zabalit *= 4;
  }
  cout << krabice_vonku << endl;
}
```

Niektorí z vás skúšali aj opačný prístup – vkladať krabice od najmenšej. Často ste si ale nevšimli, že nám vonku ostávajú krabice rôznych veľkostí. Ak totiž príde veľká krabica (napr. taká, do ktorej sa zmestia všetky menšie), počet tých, ktoré zostanú vonku sa nedá vyrátať bez toho, aby sme sa pozreli na každú menšiu veľkosť zvlášť.

Šandyna

2. Zázračné karty

(max. 6 b za popis, 4 b za program)

Na začiatok je dôležité si všimnúť niekoľko vecí. Prvé pozorovanie je, že keď x kariet presunieme na vrch a potom x kariet presunieme na spodok, budú v rovnakom poradí, ako na začiatku. Podobne môžeme vidieť, že ak presunieme n kariet jedným smerom, budú opäť všetky na pôvodnom mieste.

Predstavme si, že karty rozložíme do kruhu a medzi vrchnú a spodnú kartu vložíme zarážku tak, aby vrchná karta bola napravo od zarážky. Čo sa stane, ak presunieme vrchnú kartu na spodok balíčka a opäť ich rozložíme

do kruhu? Jediné, čo sa zmení je pozícia zarážky, ktorá bude mať naľavo od seba vrchnú kartu, ktorá sa posunula a napravo bude druhá karta zvrchu. To znamená, že zarážka sa nám po kruhu posunula o jedno miesto doprava. A samozrejme, presunutie spodnej karty navrch je len posunutie zarážky o jedno miesto doľava.

Ak teda iba presúvame karty zvrchu naspodok a naopak, stačí si pamätať, kam sme presunuli našu zarážku. A keď máme následne vypísať aktuálny stav balíčka, tak pôjdeme postupne po kruhu kariet, pričom začínať budeme na karte napravo od zarážky (aktuálna vrchná karta) až kým nenarazíme opäť na zarážku.

Čo sa ale stane ak otočíme celý balíček? Vrchná karta bude zrazu naľavo od zarážky, kým spodná bude napravo. Samotné usporiadanie kariet sa teda nezmenilo a dokonca ani zarážka sa neposunula, iba sa zmenila strana, na ktorej je vrchná karta. Je jasné, že ak v takomto stave budeme presúvať vrchnú kartu naspodok, zarážka sa posunie o jedno miesto **doľava**. Takisto výsledok sa bude vypisovať v opačnom smere ako predtým, lebo vrchná karta je na opačnej strane zarážky.

A ako to naprogramujeme? Budeme si pamätať dve premenné. Premenná *reverz* určuje, na ktorej strane od zarážky sa nachádza vrchná karta. Premenná *zarazka* potom hovorí pozíciu zarážky v našom kruhu. My ale nemáme kruh, iba obyčajné pole. To nevadí. Hodnota $zarazka = 0$ bude znamenať, že zarážka sa nachádza medzi prvou a n-tou kartou balíčka. $zarazka = 1$ znamená, že zarážka je medzi prvou a druhou kartou atď. Ak teda budeme musieť otočiť balíček, jednoducho upravíme hodnotu *reverz* a keď budeme musieť presunúť kartu, tak podľa hodnoty *reverz* zmeníme hodnotu *zarazka* – ak je vrchná karta napravo od zarážky a presúvame túto kartu naspodok, k hodnote *zarazka* pričítame číslo 1, ak je vrchná karta naľavo, tak k hodnote *zarazka* pričítame -1. A naopak pri presúvaní spodnej karty.

Samozrejme, môže sa nám stať, že hodnota *zarazka* bude záporná, alebo privysoká. Vtedy však využijeme naše prvé pozorovanie – ak posunieme n kariet jedným smerom, dostaneme rovnakú situáciu. To znamená, že k hodnote *zarazka* môžeme kedykoľvek pričítať alebo odčítať číslo n bez toho, aby sme zmenili situáciu, ktorú popisujeme. Vďaka tomu bude hodnota *zarazka* vždy v intervale 0 až $n - 1$.

Nakoniec, keď budeme chcieť vypísať výsledok, tak si z hodnôt *reverz* a *zarazka* zistíme, kde sa nachádza vrchná karta a v správnom smere (závislom od *reverz*) vypíšeme postupne všetky čísla. Časová aj pamäťová zložitosť bude lineárna od n, čo zapíšeme ako $O(n)$.

Listing programu (C++)

```
#include<bits/stdc++.h>

using namespace std;

int main() {
  int n, q;
  cin >> n >> q;
  vector<int> karty(n);
  for(int i = 0; i < n; i++) {
    cin >> karty [i];
  }

  char op;
  int reverz = 1, zarazka = 0;
  for(int i = 0; i < q; i++) {
    cin >> op;
    if (op == 'D')
      zarazka -= reverz;
    else if (op == 'H')
      zarazka += reverz;
    else if (op == 'R')
        reverz *= -1;
  }

  if (reverz == -1) {
    reverse(karty.begin(), karty.end());
    zarazka *= -1;
  }

  zarazka = ((zarazka % n) + n) % n;

  for (int i = zarazka; i < n; i++)
    printf("%d%c", karty[i], (zarazka==0 && i==n-1) ? '\n' : ' ');

  for (int i = 0; i < zarazka; i++)
    printf("%d%c", karty[i], (i == zarazka-1) ? '\n' : ' ');
}
```

Listing programu (Python)

```
n, q = map(int, input().split())
karty = input().split()
kroky = input().split()

zarazka, reverz = 0, 1

for k in kroky:
```

```
    if k == 'D':
        zarazka -= reverz
    if k == 'H':
        zarazka += reverz
    if k == 'R':
        reverz = -reverz

if reverz == -1:
    karty = list(reversed(karty))
    zarazka = -zarazka

zarazka = ((zarazka % n) + n) % n

print("␣".join(karty[zarazka:] + karty[:zarazka]))
```

Vlejd

3. Zárobkom do nového roka

(max. 6 b za popis, 4 b za program)

Najjednoduchšie, čo môžeme spraviť, je celú situáciu simulovať. Budeme si jednoducho pamätať všetky darčeky, ktoré ešte Hilbert nepredal, a postupne ich predávať. Otázkou je, ako to vieme robiť rýchlo.

Riešenie hrubou silou

Stačí mať jedno pole (C++ vector), v ktorom si udržiavame ceny ešte nepredaných darčekov. Na ňom potom potrebujeme robiť nasledovné operácie:

1. Nájdenie darčeku s najvyššou cenou neprevyšujúcou nejakú hranicu
2. Predanie (odstránenie) darčeka

Operáciu 1 vieme robiť triviálne v čase $O(n)$. Stačí prejsť všetky darčeky a pamätať si, aký najdrahší sme zatiaľ videli. Odstraňovanie darčeku je za normálnych okolností tiež lineárne, lebo musíme posunúť všetky darčeky za ním. Dá sa ale urobiť finta na skonštantnenie tohoto času. Môžeme totiž najprv vymeniť chcený darček s posledným a potom iba vector o jedna skrátiť. Podobne by sme daný darček mohli len prepísať na 0. S časovou zložitosťou si ale aj tak veľmi nepomôžeme.

Pre každého človeka, čo príde do obchodu, musíme nájsť darček v $O(n)$ a teda dostaneme riešenie v čase $O(nm)$ s pamäťou $O(n)$, ktoré nám získa 2 body.

Listing programu (C++)

```
#include <cstdio>

int darceky [1000000];              //vsetky darceky

int main () {
  int n, m;
  scanf("%d␣%d␣", &n, &m);
  for(int i = 0; i < n; i++)
    scanf("%d␣", &darceky[i]);

  bool je_prve = true;

  for(int i = 0; i < m; i++){
    int clovek;
    scanf("%d␣", &clovek);
    int max = 0;
    int kde = 0;
    for(int j = 0; j < n; j++){     //najdeme najdrahsi vyhovujuci darcek
      if (max < darceky[j] && darceky[j] <= clovek){
        max = darceky[j];
        kde = j;
      }
    }

    if(!je_prve)                    //pekne vypisovanie
      printf("␣");
    je_prve = false;
    printf("%d", max);

    if(max > 0){
      darceky[kde] = 0;             //odstranime darcek
  }

  printf("\n");
  return 0;
}
```

Mohlo by nám napadnúť, že operácia 1 sa by sa mohla dať vyriešiť binárnym vyhľadávaním. Potom by sme ju vedeli robiť v $O(\log(n))$. Problém ale je, že pre operáciu 2 by sme nevedeli použiť našu fintu (tá totiž nezachováva usporiadanie) a stále by sme ostali na čase $O(nm)$.

Riešenie hrubou STL[1] silou

Binárne vyhľadávanie nám síce veľmi nepomohlo, **binárny vyhľadávací strom** nám ale pomôcť môže. Je to štruktúra, ktorá zvláda pridávanie nových prvkov, vymazávanie a vyhľadávanie prvkov a to všetko v čase $O(\log(n))$. Takáto funkcionalita je už implementovaná v štandardnej C++kovej knižnici pod krycím názvom **multiset**. Stačí túto čarovnú štruktúru inicializovať, naplniť ju darčekmi, a potom v nej len vyhľadávať pomocou metódy **lower_bound** a vymazávať pomocou **erase**. Nakoľko pridávanie je logaritmické, vieme ju naplniť v čase $O(n\log(n))$ a pre každého človeka vyhľadávať a mazať tiež v $O(\log(n))$. Dostaneme teda riešenie v čase $O(n\log(n)+m\log(n))$. Vďaka tomu, že sú darčeky na vstupe usporiadané, je vkladanie možné aj v konštantnom čase a teda výsledná časová zložitosť by bola len $O(n+m\log(n))$, čo ale nie je podstatné zlepšenie. Pamäťovú zložitosť máme stále $O(n)$. Toto riešenie je dostatočne rýchle pre všetky testovacie sady, no plný počet za popis neprinesie.

Listing programu (C++)

```
#include <cstdio>
#include <set>

using namespace std;

int main () {
  int n, m;
  int pom;
  multiset<int> darceky;
  multiset<int>::iterator it;
  bool je_prve = true;
  scanf("%d %d", &n, &m);
  for(int i = 0; i < n; i++){
    scanf("%d ", &pom);
    darceky.insert(-pom);              //lebo set vie vracat len vacsie veci
  }

  for(int i = 0; i < m; i++){
    int clovek;
    scanf("%d ", &clovek);
    it = darceky.lower_bound(-clovek);

    if(!je_prve)
      printf(" ");
    je_prve = false;

    if(it != darceky.end()){           //toto vrati lower_bound ked nieco v sete nie je
      printf("%d", -(*it));
      darceky.erase(it);
    }
    else{
      printf("0");
    }
  }
  printf("\n");

  return 0;
}
```

Optimálne riešenie

Zatiaľ sme takmer vôbec nevyužili usporiadanie darčekov a ľudí na vstupe. To by nám mohlo napovedať, že ešte nemáme optimálne riešenie. Predstavme si nasledujúcu situáciu: do obchodu príde Ferko a kúpi nejaký darček. Keď po ňom príde Jožko, aký darček kúpi? Jožko je určite aspoň tak bohatý ako Ferko. Buď si kúpi niečo, čo mohol kúpiť aj Ferko, alebo kúpi niečo, čo si Ferko nemohol dovoliť. Inak povedané, keď prišiel do obchodu Ferko, mal množinu darčekov, ktoré si mohol dovoliť. Z nej si vybral ten najdrahší. Keď potom prišiel Jožko, tiež mal množinu darčekov, ktoré si mohol dovoliť. V nej boli **všetky** tie darčeky, čo si mohol dovoliť Ferko (okrem toho, ktorý si Ferko kúpil) a všetky, ktoré si Ferko kúpiť nemohol, no Jožko už áno.

To znamená, že nám stačí udržiavať množinu kúpiteľných a množinu nekúpiteľných darčekov. Keď príde nový človek, presunieme z nekúpiteľných do kúpiteľných všetky darčeky, ktoré si daný človek už môže dovoliť. Teda niekoľko najlacnejších nekúpiteľných darčekov prehlásime za kúpiteľné. Potom zistíme, či je množina kúpiteľných darčekov prázdna a ak nie je, predáme z nej najdrahší darček. Na začiatku sú všetky darčeky nekúpiteľné a vieme, že na vstupe sú **vzostupne** usporiadané. Vďaka tomu si môžeme nekúpiteľné darčeky pamätať vo fronte[2], do ktorej ich najprv dáme. Kúpiteľné si potom môžeme pamätať v zásobníku[3]. Potom, keď budeme presúvať nejaký darček D z nekúpiteľných do kúpiteľných, tak vieme, že D bol najlacnejší z nekúpiteľných a bude najdrahší z kúpiteľných.

[1] Standard library: http://www.cplusplus.com/reference/stl/

[2] Jednoduchá dátová štruktúra – zoznam, do ktorého na jednom konci vkladáme prvky a na druhej strane ich vyberáme.

[3] Zoznam, do ktorého vkladáme a z ktorého vyberáme na rovnakom konci

Nakoľko každé presunutie robíme v konštantnom čase a každý darček presunieme maximálne raz, dostávame riešenie s časovou zložitosťou $O(m+n)$ a potrebnou pamäťou $O(n)$.

Všetky spomenuté dátové štruktúry sa dajú pekne implementovať pomocou poľa.

Listing programu (C++)

```
#include <cstdio>

int darceky [1000001];                  //vsetky darceky

int main () {
  int n, m;
  scanf("%d %d", &n, &m);
  for(int i = 0; i < n; i++){
    scanf("%d",&darceky[i]);
  darceky[n] = 1000000001;              // na koniec pridame zarazku (velky darcek, ktory si nikto nemoze dovolit)
  int zasobnik[n];                      // zasobnik na darceky
  int v_zasobniku = 0;
  int spracovanych_darcekov = 0;
  bool je_prve = true;

  for(int i = 0; i < m; i++){
    int clovek;
    scanf("%d", &clovek);
    while(darceky[spracovanych_darcekov] <= clovek){ // pridame do zasobnika vsetky potencialne kupitelne darceky
      zasobnik[v_zasobniku] = darceky[spracovanych_darcekov];
      v_zasobniku++;
      spracovanych_darcekov++;
    }

    if(!je_prve)
      printf(" ");
    je_prve = false;
    if(v_zasobniku > 0){                //predame najdrahsi darcek
      v_zasobniku--;
      printf("%d", zasobnik[v_zasobniku]);
    }
    else{
      printf("0");
    }
  }
  printf("\n");

  return 0;
}
```

Prípadne vieme použiť aj knižničné funkcie a štruktúry ako queue (fronta) a stack (zásobník).

Listing programu (C++)

```
#include <cstdio>
#include <stack>
#include <queue>

using namespace std;

int main () {
  int n, m;
  stack<int> kupitelne;
  queue<int> nekupitelne;

  scanf("%d %d", &n, &m);
  int pom;
  for(int i = 0; i < n; i++){
    scanf("%d", &pom);
    nekupitelne.push(pom);
  }
  nekupitelne.push(1000000001);         // pridame si na koniec zarazku (velky darcek, ktory si nikto nemoze dovolit)

  bool je_prve = true;
  for(int i=0; i<m; i++){
    int clovek;
    scanf("%d",&clovek);
    while(nekupitelne.front() <= clovek){
      kupitelne.push(nekupitelne.front());
      nekupitelne.pop();
    }

    if(!je_prve){
      printf(" ");
    }
    je_prve = false;

    if(!kupitelne.empty()){             //predame najdrahsi darcek
      printf("%d", kupitelne.top());
      kupitelne.pop();
    }
    else{
      printf("0");
    }
  }
  printf("\n");
```

```
  return 0;
}
```

Kubo

4. Zapeklitá situácia

(max. 9 b za popis, 6 b za program)

Nad touto úlohou sa bolo treba zamyslieť, možno si nakresliť pár príkladov a odpozorovať niekoľko vlastností, ktoré nám pomôžu ju vyriešiť. No najprv z rozprávky vybaľme trocha terminológie. Banditi a ich mierenie nám predstavuje vrcholy a hrany orientovaného grafu, v ktorom z každého vrchola vychádza práve jedna hrana.

Úvodné pozorovania

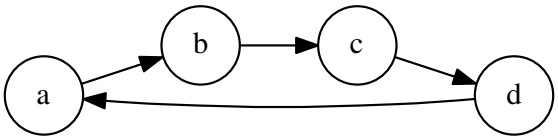
Je jasné, že ak sa náš graf skladá z viacerých komponentov, tie sa navzájom neovplyvňujú, lebo medzi nimi nevedie hrana označujúca strieľanie. Môžeme preto úlohu riešiť iba pre jeden súvislý komponent. Aj keď vo výslednom riešení túto skutočnosť nemusíme priamo použiť, zjednoduší nám jeho vymýšľanie.

Ďalej si môžeme všimnúť, že v každom komponente bude aspoň jeden šťastný bandita, ktorý prežije. Niekto v komponente totiž strieľa ako posledný a na to, z pochopiteľných dôvodov, musí byť nažive. Samozrejme, týmto výstrelom nevie zabiť sám seba.

Po chvíli kreslenia môžeme odpozorovať, že každý komponent v sebe obsahuje aspoň jeden cyklus. Ak totiž začneme v ľubovoľnom vrchole, ktorý si označíme v_0 a prechádzame sled vychádzajúcich hrán, tak postupne navštevujeme vrcholy $v_1, v_2, \ldots$ Keďže z každého vrchola vychádza práve jedna hrana, a vrcholov máme len konečne veľa, týmto spôsobom navštívime vrchol, v ktorom sme už boli (teda pre nejaké $j > i : v_j = v_i$), a teda nájdeme cyklus.

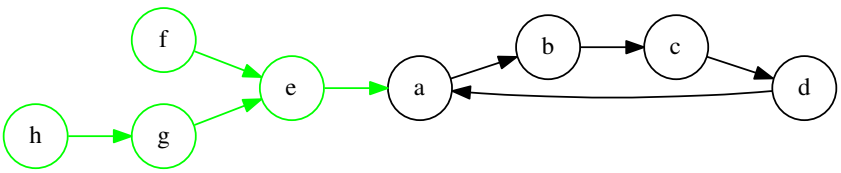
Riešenie pre jeden komponent

Najskôr sa vrhnime na najjednoduchší prípad - jednoduchá kružnica. Na nej platí, že na každého banditu mieri práve jeden iný bandita.



Pozrime sa, ako by vyzerala optimálna streľba. Keďže všetky vrcholy na kružnici sú rovnocenné a niekto musí začať strieľať, tak nech začne bandita c výstrelom na d. Teraz ale bandita b, ktorý naňho mieril, nemá stlačením spúšte čo pokaziť a tak vystrelí. Teraz však môže bez problémov vystreliť bandita a. Ten už si môže vydýchnuť, lebo naňho mieriaci d už bol zastrelený, a teda prežije. Lepšie takýto graf už nevieme vyriešiť, lebo aspoň jeden bandita nám musí zostať nažive. Ľahko vidieť, že tento postup môžeme použiť aj na väčšie kružnice, ako aj najmenší možný prípad (dvaja na seba mieriaci banditi).

V samotnom programe takéto riešenie vieme vyrobiť tak, že si najskôr nájdeme jedného banditu na takejto kružnici (v našom príklade by to bol a). Z neho obehneme kružnicu po hranách, pričom si budeme vrcholy ukladať do pomocného vektora, a zastavíme keď narazíme na pôvodný vrchol. V našom príklade by v ňom zostali hodnoty $[a, b, c, d]$. Keď sa na tento vektor pozrieme od konca, získame správnu postupnosť výstrelov. Treba si však dať pozor na posledného banditu (d), ktorý nevystrelí, ale je zastrelený ako prvý. Jeho teda vypisovať nechceme.



Nuž a čo ak komponent nie je len obyčajná kružnica? V takom prípade komponent vyzerá ako cyklus, do ktorého vedie niekoľko stromov (na obrázku je takýto strom vyznačený zelenou). V obyčajnom cykle nám musel zostať jeden bandita nažive, lebo strieľal ako posledný a už ho nemal kto ďalší zastreliť. Teraz však môžeme využiť to, že do kružnice mieri niekto iný.

Pozrime sa znova na náš obrázok a vezmime si riešenie predchádzajúcej časti. Vtedy sa nám postrieľali banditi b, c, d a nažive nám zostal a. Tentokrát však máme banditu e, ktorý ho môže zastreliť, a jeho zas môže zastreliť bandita f atď. Na kružnici teda neprežije nikto. Kto ale prežije na strome?

Vrcholy stromu, ktoré majú len jednu hranu nazývame *listami*. Je zjavné, že každý bandita, ktorý je v liste, prežije – nikto naňho totiž nemieri. Ďalej si ukážeme, že vieme nájsť postupnosť výstrelov, v ktorej zastrelíme každého iného banditu v strome.

Označme si vrchol, ktorým sa strom napája na kružnicu (e) ako *koreň* stromu. Pre každý list v strome vedie práve jedna cesta z listu do koreňa. Ak si vyberieme ľubovoľnú takúto cestu a postrieľame na nej postupne všetkých banditov (samozrejme, okrem listu), rozpadne sa nám strom na niekoľko menších stromov a stále bude platiť, že banditi, na ktorých nikto nemieri sú len pôvodné listy. Zmenšili sme strom na niekoľko menších a opakovaním tohto postupu nám zostanú len stromy s jedným vrcholom – pôvodné listy – preživší.

Naprogramovať sa to dá pomerne jednoducho. Vyberieme si náhodný list (f) a postrieľame banditov na ceste ku koreňu. Rovnako ako na kružnici si môžeme uložiť postupnosť vrcholov na ceste do poľa a pole potom obrátiť a pridať ku výslednej postupnosti. Strom sa nám rozpadne na dva menšie stromy – $f, h \to g$.

Ak by sme teraz chceli postrieľať banditov v podstrome, tiež by sme vystrieľali cestu od niektorého listu ku koreňu (teda napríklad na ceste h, g). Môžeme si ale uvedomiť, že táto cesta leží na ceste od listu ku koreňu aj v pôvodnom strome (teda, h, g je súčasťou h, g, e). V implementácii si teda vôbec nemusíme pamätať menšie stromy, stačí pospúšťať cesty (streľby) z každého listu do koreňa a vždy zastreliť len tých banditov, ktorí ešte žijú.

Dokonca nemusíme vedieť ani ktoré vrcholy sú koreňmi. Jednoducho spustíme streľbu z listu a keď sa strom napojí na kružnicu, vystrieľa ju celú. Ďalšie postupnosti výstrelov už kružnicu nenavštívia, lebo strieľame len dovtedy, kým na ceste nenarazíme na mŕtveho banditu.

Výsledné riešenie

Naše riešenie teda vyzerá tak, že prejdeme cez všetky listy a postrieľame cestu, ktorá z nich vychádza, a následne prejdeme cez všetky kružnice.

Už potrebujeme len zistiť, ktoré vrcholy sú listy a ktoré sú na kružnici. O listoch vieme, že nebudú mať žiadne vstupné hrany, takže takže nám stačí spočítať si vstupné hrany každého vrcholu pri načítaní vstupu. O vrcholoch na kružnici zase vieme, že majú práve jednu vstupnú hranu. Tu už ale neplatí, že každý vrchol s jednou vstupnou hranou leží na kružnici (napríklad vrchol g). Avšak každý z takýchto vrcholov zastrelíme pri strieľaní od listov. Stačí nám teda najskôr prejsť vrcholy a strieľať od listov, a potom ich prejsť ešte raz a strieľať len živé vrcholy s jednou vstupnou hranou.

Pamäťová zložitosť bude $O(n)$. Pre každého banditu si pamätáme len jeho cieľ, počet naňho mieriacich banditov a či je ešte nažive.

Časová zložitosť je tiež $O(n)$. Pri strieľaní listov sa na každý vrchol pozrieme najviac dvakrát. Raz, keď sa pozrieme či to nie je list a raz, keď ho zastrelíme. Obdobne pri strieľaní kružnice.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

vector<bool> nazive;
vector<int> ciel;
vector<int> vstupne;
vector<int> result;

void strielaj(int a) {
  if(!nazive[a]) return;

  vector<int> temp;   // na ukladanie vystrelov
  temp.push_back(a);  // ukladame od posledneho vystrelu po prvy
  int i = ciel[a];

  // zapiseme si zastrelenie dalsieho banditu a pokracujeme
  while(i != a && nazive[i]) {
    nazive[i] = false;
    temp.push_back(i);
    i = ciel[i];
  }
```

```
  // posledny striela na mrtveho alebo povodneho, to nechceme
  temp.pop_back();

  // vystrely mame ulozene od posledneho, ale chceme ich od prveho
  reverse(temp.begin(), temp.end());
  result.insert(result.end(), temp.begin(), temp.end());
}

int main(){
  int n;
  cin >> n;

  ciel.resize(n);
  nazive.resize(n, true);
  vstupne.resize(n, 0);

  // nacitame vstup, precislujeme vrcholy od 0, spocitame vstupne hrany
  for(int i=0; i<n; i++) {
    cin >> ciel[i];
    ciel[i]--;
    vstupne[ciel[i]]++;
  }

  // zastrelime od volnych koncov
  for(int i=0; i<n; i++)
    if (vstupne[i] == 0) strielaj(i);

  // zastrelime kruznice
  for(int i=0; i<n; i++)
    if (vstupne[i] == 1) strielaj(i);

  int count = 0;
  for(int i=0; i<n; i++)
    if (nazive[i]) count++;

  cout << count << endl;

  cout << result.size() << endl;
  for(int i=0; i<result.size(); i++)
    if (i != result.size()-1)
      cout << result[i]+1 << "␣";    // kedze sme precislovali po nacitani, tak precislujeme spat
    else
      cout << result[i]+1 << endl;

  return 0;
}
```

Buj

5. Obmedzená slovná zásoba

(max. 9 b za popis, 6 b za program)

Táto úloha bola zaujímavá tým, že na jej vyriešenie nebolo potrebné poznať žiadne dátové štruktúry, ani techniky riešenia úloh ako napríklad dynamické programovanie. Ba dokonca, ak ste v úlohe hľadali nejaké tie štandardné štruktúry a algoritmy, tak ste pravdepodobne zabili veľa času :)

Prvý krok k úspešnému vyriešeniu je rozobrať si niekoľko malých prípadov ručne, a snáď pritom odpozorovať niečo o tom, ako sa to celé správa.

Na začiatok si všimneme, že ak niektoré slovo na vstupe je prázdne, tak odpoveď je zrejme 0. (Nemá prefix ani sufix dĺžky aspoň 1.) V ďalšom texte predpokladáme, že obe slová majú dĺžku aspoň 1.

Hrubá sila

Každé slovo vieme nájsť tak, že zoberieme nejaký prefix A, nejaký sufix B a spojíme ich. Naskytá sa nám tak nasledujúce riešenie:

Vyskúšame všetky možné prefixy A. Pre každú možnosť vyskúšame všetky sufixy B. Spojíme ich do jedného slova, a uložíme do vreca. Po odskúšaní všetkých možností z vreca odstránime všetky duplikáty. Počet zvyšných prvkov vo vreci je hľadaný výsledok.

Jediný problém môžeme mať pri vyhadzovaní duplikátov – ako to spraviť? Zamyslime sa, ako by sme odstránili duplikáty z postupnosti čísel – postupnosť utriedime, a následne vieme, že rovnaké prvky sú v postupnosti hneď za sebou. Prejdeme teda postupnosť od začiatku po koniec, a každý prvok, ktorý je rôzny od predchádzajúceho, hodíme do výsledného vreca (ktoré už neobsahuje duplikáty).

To isté spravíme so slovami. Slová budeme porovnávať **lexikograficky** (takým spôsobom sú slová usporiadané napríklad v slovníkoch). To znamená, že sa najprv pozrieme na prvý znak. Ak sa slová na ňom nezhodujú, za **lexikograficky menšie** prehlásime to, ktorého znak je v abecede skôr. Ak sa slová na ňom zhodujú, pokračujeme ďalším písmenom.

Čo sa ale stane, ak niektoré slovo už nemá nasledujúci znak? Jednoducho ho prehlásime za menšie. Teda ak je jedno zo slov prefixom druhého, tak je od neho lexikograficky menšie.

Napríklad $jablko < jablkoahruska$, $a < bcdef$, prázdny reťazec je menší od ľubovoľného slova, ...

Čo sa týka implementácie, väčšina programovacích jazykov štandardne vie porovnávať dva reťazce, a tiež väčšinou majú štandardnú funkciu na triedenie. Stačí ju teda zavolať na naše pole všetkých kombinácii prefixov a sufixov.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main () {
  string A, B;
  cin >> A >> B;
  int sa = A.size(), sb = B.size();

  vector<string> vrece;

  for (int i=1; i<=sa; i++)
    for (int j=1; j<=sb; j++)
      vrece.push_back(A.substr(0, i) + B.substr(sb-j, j));

  sort(vrece.begin(), vrece.end());

  int res = 0;
  for (int i=0; i < vrece.size(); i++) {
    if (i==0)
      res++;
    else if (vrece[i] != vrece[i-1])
      res++;
  }
  cout << res << "\n";
  return 0;
}
```

Časová zložitosť tohto riešenia je $O(n^3 \log n)$, a pamäťová $O(n^3)$, kde n je veľkosť vstupu. (Máme $O(n^2)$ reťazcov, a každý má dĺžku $O(n)$. Triedenie spraví $O(n^2 \log n^2)$ porovnaní, každé v čase $O(n)$, takže celková zložitosť je $O(n^2 \log n^2 \cdot n) = O(n^3 \log n)$.) V špeciálnych prípadoch je to aj menej (napríklad keď $|A| = 0$), ale v najhoršom prípade je to toľkoto.

Jemnejšia sila

Na „odstránenie" duplikátov sa dá použiť aj iný postup – zabezpečíme, že ich nikdy do nášho „vreca" nevložíme.

Na začiatku budeme mať prázdny slovník a postupne vyskúšame pridať všetky kombinácie prefixov A a sufixov B, tak, ako v predchádzajúcom riešení. Ak práve skúšaná kombinácia ešte nie je v slovníku, pridáme ju doň a zvýšime si počítadlo počtu slov v slovníku o $+1$. V opačnom prípade ju zahodíme.

Slovník môžeme implementovať ako písmenkový strom (po anglicky trie) alebo ako hashovaciu tabuľku (`unordered_map` v štandartnej knižnici C++). Vkladanie prvkov a zisťovanie, či už sa tu takýto prvok nachádza trvá týmto štruktúram lineárny čas od dĺžky reťazca.

Oproti predchádzajúcemu riešeniu sme si vylepšili čas o $\log n$ – časová zložitosť je $O(n^3)$, a pamäťová je rovnaká.

Riešenie postupnými zmenami

Najprv si všimneme, že má zmysel hľadať duplikáty iba v slovách rovnakej dĺžky, nakoľko slová rôznych dĺžok určite nie sú rovnaké. Vieme tak znížiť pamäťovú zložitosť na $O(n^2)$ – namiesto toho, aby sme si spravili zoznam všetkých kombinácii naraz, stačí nám spraviť si osobitný zoznam pre každú dĺžku. Ten môžeme po spracovaní (vyhodení duplikátov) zahodiť, a tak si nikdy nepamätáme viac ako $O(n)$ reťazcov dĺžky $O(n)$ **naraz**.

Pozrime sa teraz, ako vyzerajú kombinácie s jednou konkrétnou dĺžkou, keď ich spracúvame „sprava doľava" (tak, že začneme kombináciou s najdlhším prefixom A, a postupne zmenšujeme prefix A a zväčšujeme sufix B). Napríklad ak máme kombinácie slov $jablko, hruska$ dĺžky 8, tak ich budeme spracúvať v tomto poradí:

$$jablkoka, jablkska, jabluska, jabruska, jahruska$$

Zameriame sa na zmeny. Čo sa zmení, keď sa posuniem v zozname o 1 ďalej? Posledný znak prefixu sa nahradí znakom sufixu. (Napríklad v treťom posune sme mali prefix $jabl$ a sufix $uska$. Posledný znak prefixu, l, sme nahradili znakom sufixu, ktorý v $hruska$ predchádza $uska$, teda r.) Ak sú tieto znaky rovnaké, dostaneme to isté slovo – našli sme duplikát. Čo ale ak sú tieto znaky rôzne? Nemôže sa stať, že by sme dostali slovo, ktoré bolo spracované ešte skôr?

Všimnime si ale, že zmeny sa postupne dejú na skorších pozíciách. Prvý posun $jablkoka \to jablkska$ nám zmenil znak na pozícii 6, druhý posun ovplyvňuje znak na pozícii 5, ... Teda **každá pozícia sa zmení v**

najviac jednom posune. Ak sa znak na pozícii k zmenil v nejakom posune z α na β, potom všetky kombinácie pred tým posunom majú na danej pozícii α, a všetky kombinácie po tom posune majú na tej pozícii β. Napríklad v presune $jabluska \rightarrow jabruska$ sa zmenila pozícia 4:

$$jab\mathbf{l}koka, jab\mathbf{l}kska, jab\mathbf{l}uska \rightarrow jab\mathbf{r}uska, jah\mathbf{r}uska$$

To ale znamená, že ak je nahradený znak rôzny od pôvodného, tak výsledné slovo určite nebolo spracované skôr, lebo všetky slová spracované skôr majú na tejto pozícii pôvodný znak. Našli sme tak novú kombináciu.

Vyskúšame teda všetky možné dĺžky kombinácií, a pre každú spracujeme kombinácie s tou dĺžkou. Pritom nemusíme kombinácie vytvárať (nemusíme *materializovať* stringy), ale pri spracovaní sa pozeráme iba na 2 znaky – pôvodný (pred posunom), a nový (po posune). Časová zložitosť je preto $O(n^2)$, a pamäťová $O(n)$.

Listing programu (C++)

```
#include <iostream>
#include <string>
using namespace std;

int main () {
  string A,B;
  cin >> A >> B;

  long long res = (long long) A.size() * B.size();
  int maxdlzka = A.size() + B.size();
  for (int dlzka=2; dlzka<=maxdlzka; dlzka++) {
    int a = A.size();
    int b = dlzka - a;
    while (b<=0) {            // dlzka sufixu B predsa nemoze byt zaporna!!!
      a--;
      b++;
    }
    while (a > 1 && b < B.size()) {
      // prefix tvoria znaky A[0], A[1], ..., A[a-1]
      char povodny = A[a - 1];

      // sufix tvoria znaky B[B.size()-1], B[B.size()-2], ..., B[B.size()-b]
      // a teda znak predchadzajuci tomu sufixu je B[B.size()-b-1]
      char novy = B[B.size() - b - 1];

      if (povodny == novy)    // objavili sme duplikat
        res--;
      a--;
      b++;
    }
  }
  cout << res << "\n";
  return 0;
}
```

Možno ste si všimli detail v implementácii: Všetkých kombinácií je $|A| \cdot |B|$ a odčítavame počet duplikátov. Tento detail sa nám bude hodiť za chvíľku.

Počítanie toho istého iným spôsobom

Čo vlastne predchádzajúce riešenie robí? Pozerá sa na niektoré dvojice [pozícia v A, pozícia v B], a ak sú znaky na týchto pozíciách rovnaké, odráta od výsledku 1. Zamyslime sa nad tým, koľkokrát pozrie ktoré dvojice. Zvolíme si nejakú konkrétnu dvojicu $[a, b]$ a chceme zistiť, koľko posunov spôsobí nahradenie znaku na pozícii a znakom, ktorý je v B na pozícii b?

Každá pozícia v A nám jednoznačne určí, aký má byť prefix pred posunom, a aký je po posune. (Napríklad pozícia 2 v *jablko* vraví, že pred posunom je prefix ja, a po posune je j.) Pozíciu 1 ale neberieme do úvahy, lebo po posune bude prefix prázdny. Zaujímajú nás len kombinácie tvaru [neprázdny prefix A] zreťazený s [neprázdny sufix B].

Taktiež každá pozícia v B nám jednoznačne určí, aký má byť sufix pred a po posune. Nerátame pozíciu $|B|$. Dostávame tak, že na každú dvojicu $[a, b]$, kde $a > 1, b < |B|$ sa pozrieme **práve raz**. Čo to ale znamená? Zvoľme si nejakú pevnú pozíciu a, a povedzme, že na nej je znak α. Potom odrátame 1 za každý znak v [B okrem posledného znaku], ktorý je rovný α. Celkovo si odrátame k, kde k je počet znakov α v [B bez posledného znaku]. Ak máme v [A bez prvého znaku] l znakov α, od výsledku dokopy odrátame $l \cdot k$.

Dopracovali sme sa tak k nasledujúcemu riešeniu: spočítame si počty jednotlivých znakov v [A bez prvého znaku], [B bez posledného znaku]. Zo získaných počtov ľahko vypočítame výsledok. Časová zložitosť je $O(n+\sigma)$, kde σ je veľkosť abecedy.

Listing programu (C++)

```
#include <iostream>
#include <string>
```

```
using namespace std;

int main () {
  string A,B;
  cin >> A >> B;

  long long pocA[26], pocB[26];
  for (int i=0; i<26; i++) {
    pocA[i]=0;
    pocB[i]=0;
  }

  for (int i=1; i<(int)A.size(); i++)
    pocA[A[i]-'a']++;
  for (int i=0; i<(int)B.size()-1; i++)
    pocB[B[i]-'a']++;

  long long res= (long long)A.size()*(long long)B.size();
  for (int i=0; i<26; i++)
    res-= pocA[i]*pocB[i];

  cout << res << "\n";
  return 0;
}
```

Pamäťová zložitosť riešenia je $O(n+\sigma)$, ale vedeli by sme ju zlepšiť aj na $O(\sigma)$, ak by sme vstup načítavali po znakoch.

Žaba

6. O sústredkových pozvánkach

(max. 12 b za popis, 8 b za program)

Keďže zadanie je trochu komplikovanejšie, začnime tým, že si ho zopakujeme. Máme čísla 1 až n a m podmienok, ktoré musíme splniť. Každá podmienka nám o nejakom čísle hovorí, že sa vo výslednej permutácii musí objaviť pred ktorýmsi iným číslom. Naviac, nechceme nájsť ľubovoľnú permutáciu, ktorá spĺňa všetky podmienky, ale takú, kde sa číslo 1 nachádza čo najskôr, potom číslo 2 čo najskôr atď. Uvedomme si, že toto nie je to isté ako hľadať lexikograficky najmenšiu takúto permutáciu, lebo permutácia $(3,1,2)$ je síce lexikograficky väčšia ako $(2,3,1)$, ale číslo 1 sa v nej nachádza skôr.

Pri riešení takýchto úloh je dobré si vstup zakresliť, aby sa nám nad ním lepšie rozmýšľalo – napríklad formou orientovaného grafu. Každé číslo bude mať priradený jeden vrchol a hrana povedie z vrchola x do vrchola y, ak má byť x v permutácii pred y. Keďže zadanie nám zaručuje, že existuje aspoň jedna vhodná permutácia, v takomto grafe sa nemôžu nachádzať orientované cykly (rozmyslite si prečo by sme ich nevedeli celé splniť) a takýto graf sa preto volá orientovaný acyklický graf, alebo tiež DAG[4].

Poďme sa teraz pozrieť na to, ako by sme vedeli vytvoriť vhodnú permutáciu, ktorá by spĺňala všetky podmienky. Ktoré číslo sa môže nachádzať na prvom mieste? Každé, ktoré nemá byť predbehnuté iným číslom. Keď sa teda pozrieme na náš graf, zistíme, že sú to tie vrcholy, do ktorých nevedie žiadna hrana – hrana vedie z vrchola, ktorý má byť skôr, do takého, ktorý má byť neskôr. Ak do vrchola vedie hrana, musíme najskôr do permutácie vložiť prvok na začiatku tejto hrany.

Ak teda vyberieme nejaké číslo a vložíme ho do permutácie, z nášho grafu si môžeme príslušný vrchol odstrániť. A rovnako všetky hrany, ktoré z neho viedli von, lebo tieto hrany už určite budú splnené. Ich začiatok sa v permutácii nachádza skôr ako ich koniec. Na druhom mieste permutácie môže byť opäť ľubovoľný vrchol, do ktorého nevchádza žiadna hrana.

Postupne by sme teda vedeli vytvoriť permutáciu, ktorá spĺňa všetky podmienky. Ako však vyberieme takú, kde bude 1 čo najskôr? Nemôžeme sa rozhodovať podľa veľkosti čísel, lebo nejaké väčšie číslo nám možno odkryje lepšie menšie a vie nastať pomerne veľa komplikovaných prípadov.

Keď to teda nevyšlo pre začiatok, skúsme vytvárať permutáciu od konca. Ktoré čísla môžu byť na poslednej pozícii? No predsa všetky, po ktorých už nemusí ísť žiadne číslo. V grafe sú to teda vrcholy, z ktorých nevychádza žiadna hrana. A opäť, ak si niektorý z nich vyberieme a odstránime ho z grafu, dostaneme graf, z ktorého môžeme vyberať číslo na pozíciu, ktorá je druhá od konca. Robíme v podstate to isté, len opačným smerom. Vieme však teraz zaručiť aj zvyšné požiadavky, teda aby sa 1 nachádzala vo výslednej permutácií čo najskôr?

Čo keby sme vždy z čísel, ktoré máme k dispozícii na poslednú pozíciu vybrali to, ktoré je najväčšie? Znie to ako nápad, ktorý nič nepokazí. Predsalen, umožníme všetkým menším číslam, aby sa nachádzali skôr. A síce je fajn, že nám intuícia napovedá, že tento postup je správny, musíme si ho aj dokázať.

Predstavme si, že máme optimálnu permutáciu, ktorá na posledné miesto umiestnila číslo x, ale na posledné miesto môže ísť aj číslo y, ktoré je väčšie ako x. Čo sa stane, ak z tejto permutácie vyberieme číslo y a dáme ho na koniec, pričom všetky čísla, čo boli pôvodne za ním posunieme o jedno dopredu? Vo výslednej permutácii sa dozadu posunulo iba číslo y. Všetky ostatné buď zostali na svojom mieste, alebo sa posunuli dopredu. A hlavne

[4] Z anglického Directed Acyclic Graph

sa dopredu posunulo číslo x. A keďže je menšie ako y, tak táto permutácia je lepšie ako pôvodná, s ktorou sme začínali. To je ale spor s tým, že tá permutácia mala byť optimálna. Dokázali sme teda, že náš postup je správny.

Otázkou zostáva už len to, ako takéto riešenie naprogramovať. Musíme vedieť odstrániť ľubovoľný vrchol z grafu a tiež všetky hrany, ktoré z neho viedli. Takisto si musíme udržiavať množinu čísel, z ktorých nevedie žiadna hrana a z tejto množiny rýchlo vybrať číslo, ktoré je najväčšie. Na druhú časť použijeme dátovú štruktúru halda, do ktorej vieme vkladať prvok v čase $O(\log n)$ a vyberať najväčší prvok v čase $O(\log n)$, kde n je počet prvkov v halde.

Na začiatku načítame vstup a pre každý vrchol vytvoríme zoznam hrán, ktoré doň vchádzajú. Takisto si v poli $P[]$ budeme pamätať, koľko hrán z neho vychádza. Všetky čísla x, pre ktoré je $P[x]$ rovné 0 vložíme do haldy, keďže sú to vrcholy, z ktorých nevychádza žiadna hrana. Z haldy vyberieme najväčšie číslo, označme si ho ako x. Vrchol x chceme odstrániť z grafu. Preto prejdeme všetky hrany vychádzajúce z tohto vrchola a každému koncovému vrcholu zmenšíme hodnotu v $P[]$, lebo už z neho vychádza o jednu hranu menej. Ak počas týchto úprav klesne niektoré $P[y]$ na nulu, tak y vložíme do haldy. Na konci vypíšeme čísla v obrátenom poradí ako sme ich vyberali z haldy.

Každý vrchol vložíme aj vyberieme z haldy najviac raz. Takisto raz odstránime každú hranu. Preto je výsledná časová zložitosť tohto riešenia $O(n \log n + m)$. Pamäťová zložitosť je $O(n + m)$.

Listing programu (C++)

```
#include <cstdio>
#include <algorithm>
#include <vector>
#include <cmath>
#include <queue>
using namespace std;

#define For(i, n) for(int i=0; i<int(n); i++)

vector<vector<int> > G;
vector<int> P;
int n, m;

int main() {
  scanf("%d %d", &n, &m);
  P.resize(n, 0);
  G.resize(n);
  For(i, m) {
    int x, y;
    scanf("%d %d", &x, &y);
    x--; y--;
    G[y].push_back(x);
    P[x]++;
  }
  vector<int> Res;
  priority_queue<int> Q;
  For(i, n)
    if(P[i] == 0)
      Q.push(i);
  while(!Q.empty()) {
    int v = Q.top(); Q.pop();
    Res.push_back(v);
    For(i, G[v].size()) {
      int w = G[v][i];
      P[w]--;
      if(P[w] == 0) Q.push(w);
    }
  }
  reverse(Res.begin(), Res.end());
  For(i, Res.size())
    printf("%d%c", Res[i]+1, ((i != Res.size()-1) ? ' ' : '\n'));
}
```

Mário

7. Odolateľná reklama

(max. 12 b za popis, 8 b za program)

V tomto vzoráku si ukážeme viacero všeobecných techník na riešenie úloh a optimalizovanie programov. Ak sa ich naučíte, pomôžu vám vyriešiť veľa rôznych problémov.

Najprv si ukážeme *bruteforce* riešenie. Ďalej bruteforce zoptimalizujeme pomocou *memoizácie*, prevedieme rekurziu na *dynamické programovanie* a ukážeme si, kedy použiť *binárne vyhľadávanie*. Takto získame riešenie so zložitosťou $O(n^2 \log n)$ hodné 5/8 bodov s veľmi malou námahou.

Na záver ešte trošku porozmýšľame a dopracujeme sa k $O(n \log n)$ riešeniu, v ktorom zoptimalizujeme dynamické programovanie pomocou deque.

Riešenie hrubou silou

Ak neviete, kde začať alebo ak ste už unavení z úvodného uvažovania, vždy je dobré začať s naprogramovaním riešenia hrubou silou – najjednoduchšie naprogramovateľné, funkčné riešenie, čo nám napadne – väčšinou pomerne neefektívne. V našej úlohe takéto riešenie vyskúša pre každý lístok, či sa s ním dá cieľ dosiahnuť s čakaním menším ako t a z vyhovujúcich lístkov vyberie ten najlacnejší.

Kostra programu by mohla vyzerať nasledovne:

Listing programu (Python)

```
n, t = map(int, input().split())
# Na začiatok polí pridáme nuly, aby sme mali zastávky očíslované od 1
cena  = [0, ]  + list(map(int, input().split()))
cakaj = [0, 0] + list(map(int, input().split()))

# Nejak spočítame, čí sa dá dostať z 1 na n s k-lístkom
def dasa(k):
    ...

odpoved = cena[-1]

for k in range(1, n):
    if cena[k] < odpoved and dasa(k):
        odpoved = cena[k]

print(odpoved)
```

Uvedomíme si, že **zo zastávky s číslom x sa nám nikdy neoplatí vrátiť späť** (ísť na zastávku s menším číslom) a teda sa budeme hýbať len na zastávky $x+1, x+2, \ldots, x+k$.

To, či s k-lístkom vieme dosiahnuť cieľ, vieme zistiť jednoduchou rekurziou. Ak sme na nejakej zastávke, to či sa dá odtiaľto dostať do cieľa zistíme tak, že sa skúsime pohnúť na ďalšiu a spýtame sa, či sa odtiaľto vieme dostať do cieľa v časovom limite. Musíme ale vyskúšať všetky susedné zastávky.

Funkcia f ako parameter dostane číslo zastávky x, odkiaľ sa chceme dostať do cieľa a zvyšný čas, ktorý máme na čakanie t. Vyskúša všetkých k pohybov na zastávky $x+1, \ldots, x+k$ – zavolá $f(x+1, t-t_x), \ldots, f(x+k, t-t_x)$ a vráti `True` ak sa dá do cieľa dostať z niektorej zo zastávok $x+1, \ldots, x+k$. Inak fukcia vráti `False`.

Listing programu (Python)

```
def dasa(k):
    def f(x, t):
        if t < 0:
            return False
        if x >= n:
            return True
        for dalsi in range(x+1, x+k+1):
            if f(dalsi, t-cakaj[x]):
                return True
        return False

    # Z 1 na n sa dá dostať k-lístkom práve vtedy, ak
    return f(1, t) == True
```

Rekurzia vyskúša každú cestu zo zastávky 1 do n najviac raz a ciest je najviac toľko, koľko je všetkých podmnožín čísel $1, 2, \ldots, n$. Časovú zložitosť tohto riešenia vieme zhora odhadnúť ako $O(n^3 \cdot 2^n)$: Pre jeden z n lístkov over 2^n ciest. Každá cesta má najviac n zastávok (je potrebných najviac n volaní) a každé volanie zbehne v čase $O(k)$.

Nepočítajme nič dvakrát 1 - Memoizácia

V predošlom riešení sa môže stať, že rekurzívnu **funkciu voláme viackrát s tými istými parametrami**. Počítame teda viackrát to isté. Tomuto sa môžeme vyhnúť, ak si budeme spočítané výsledky rekurzie pamätať. Ak potom zavoláme funkciu s tými istými parametrami, namiesto výpočtu sa len pozrieme do pamäte. Vypočítané výsledky si môžeme pamätať vo viacrozmernom poli (každý rozmer zodpovedá jednému parametru), alebo v mape, kde je kľúčom k-tica parametrov.

Listing programu (Python)

```
def dasa(k):
    memo = {}

    def f(x, t):
        if t < 0:
            return False
        if x >= n:
            return True
        if (x, t) in memo:
            return memo[(x, t)]
```

```
        for dalsi in range(x+1, x+k+1):
            if f(dalsi, t-cakaj[x]):
                memo[(x, t)] = True
                return True

        memo[(x, t)] = False
        return False

    return f(1, t) == True
```

Pre každý z n lístkov je počet rôznych volaní funkcie (rôznych dvojíc parametrov (zastávka, čas)) najviac $n \cdot t$. Každé volanie trvá najviac $O(k)$ času. Pre jeden lístok zistíme $dasa(k)$ v čase $O(n \cdot t \cdot k)$ a teda celý náš program bude mať čas behu zhora ohraničený $O(n^3 \cdot t)$.

Použitie memoizácie si môžete pozrieť aj v riešení úlohy z minulej série Optimálna šifrovačka.

Nepočítajme nič dvakrát 2

Napriek tomu, že pre každú dvojicu parametrov voláme rekurziu len raz, ak máme spočítaný výsledok pre nejaké parametre, môžeme už vedieť výsledok pre iné: Ak vieme, že cestu stihneme so zvyšným časom 47, cestu určite stihneme aj s hocijakým väčším zvyšným časom. Stačí nám teda zapamätať si ten najmenší čas, s ktorým sa vieme dostať do cieľa z x. Rôznych volaní funkcie f tak už nebude $n \cdot t$ ale len n.

Namiesto toho, aby sme pre dvojicu (zastávka x, čas t) počítali, len či sa dá dosiahnuť cieľ (`true/false`), budeme pre zastávku x počítať, aký je **najmenší čas potrebný na dosiahnutie cieľa, keď začneme v** x. Toto opäť vieme rátať podobnou rekurziou ako vyššie – pozrieme sa na najbližšie zastávky, pre ne rekurzívne spočítame hodnoty $f(x+1), f(x+2), \ldots, f(x+k)$ a vyberieme si, kam sa chceme pohnúť – na zastávku, odkiaľ sa najskôr dostaneme do cieľa. Teda $f(x) = \min\{f(x+1), f(x+2), \ldots, f(x+k)\} + t_x$ a $f(n) = 0$

Listing programu (Python)

```
def dasa(k):
    memo = {}

    def f(x):
        if x >= n:
            return 0
        if x in memo:
            return memo[x]
        memo[x] = min( f(dalsi) for dalsi in range(x+1, x+k+1) ) + cakaj[x]
        return memo[x]

    # Z 1 na n sa dá dostať k-lístkom práve vtedy, ak
    return f(1) <= t
```

Jedno volanie stále zbehne v čase $O(k)$, ale rôznych vstupov funkcie f je len $O(n)$. Výpočet pre jedno k teda trvá $O(n \cdot k)$, a vďaka tomu bude celková zložitosť programu $O(n^3)$.

Nepočítajme nič zbytočne - Binárne vyhľadávanie

Skúsme teraz zoptimalizovať inú časť programu – skúšanie všetkých lístkov. Pri tejto úlohe je dobré si uvedomiť, že rôzne ceny lístkov sú tu hlavne na zmätenie. To podstatné je zistiť minimálne k také, že vieme trasu prejsť s čakaním menším ako t. Ak máme minimálne k, na výpočet výsledku stačí nájsť najmenšiu cenu z $c_k, c_{k+1}, \ldots, c_n$.

Ak už totiž vieme, že sa dá trasa prejsť s k-lístkom, určite sa bude dať prejsť aj s $k+1$-lístkom a s hocijakým lístkom s väčším dosahom. Ak sa trasa nedá prejsť s k-lístkom, nebude sa dať prejsť so žiadnym lístkom s menším dosahom. Výsledky funkcie $dasa(k)$ budú teda s rastúcim k najskôr len `False` a od istej hranice budú len `True`.

Vieme sa teda pozrieť najskôr na $dasa(n/2)$, ak je to `False`, $k > n/2$, inak $k \leq n/2$. Vylúčili sme polovicu možností a pokračujeme pýtaním sa na $dasa(n/4)$ alebo $dasa(3n/4)$ ... Po približne $\log_2 n$ krokoch nám zostane jediná možnosť.

Binárne vyhľadávanie sa dá využiť vždy, keď hľadáme hodnotu v monotónnej postupnosti (nerastúcej alebo neklesajúcej). Vieme hľadať najľavejší výskyt aj najpravejší výskyt hodnoty. Špeciálnym prípadom sú funkcie ako $dasa()$, kde sú hodnoty `False/True`, alebo `0/1`.

Listing programu (Python)

```
def najmensie_k():
    l = 0       # este sa to neda stihnut
    r = n-1     # s takymto dosahom sa to stiha

    while r - l > 1:
        piv = (l + r) // 2    # prvok v strede intervalu (l, r>
        if not dasa(piv):
            l = piv
```

```
        else:
            r = piv
    return r

print(min(cena[k] for k in range(najmensie_k(), n)))
```

Chuťovka: Pre vyhnutie sa chybám v kóde binárneho vyhľadávania môže byť vhodné použiť binárne vyhľadávanie štandardnej knižnice jazyka. V pythone sa dá dokonca zneužiť aj pre náš účel.

Listing programu (Python)

```
from bisect import bisect_left

def najmensie_k():
    class vyhladavatko:
        def __getitem__(self, key):
            return dasa(key)

    return bisect_left(vyhladavatko(), True, 1, n)
```

V našej úlohe teda $(\log n)$-krát zavoláme funkciu $dasa()$ a dosiahneme čas $O(n^2 \log n)$.

Toto riešenie nám umožní dosiahnuť 5/8 bodov (riešenie v C++ určite, s týmto pythonovským kódom 4) a ani sme nemuseli veľa rozmýšľať (pokiaľ poznáte tieto všeobecné techniky, všetky predošlé úvahy dokopy vám zaberú len niekoľko minút).

Dynamické programovanie

Vráťme sa naspäť k optimalizovaniu našej rekurzívnej funkcie $f(x)$ – najmenší potrebný čas na dosiahnutie cieľa zo zastávky x. Na vypočítanie $f(x)$ sme potrebovali mať spočítané hodnoty $f(x+1), f(x+2), \ldots, f(x+k)$, a na ich spočítanie sme potrebovali mať vypočítané hodnoty až po $f(x+k+k)$, atď. Od začiatku ale vieme, že $f(n) = 0$. Na spočítanie $f(n-1)$ potrebujeme len $f(n)$, na spočítanie $f(n-2)$ len $f(n-1), f(n)$, atď.

Hodnoty $f(x)$ teda nemusíme počítať v rekurzívnej funkcii, ale stačí použiť jednoduché cykly. Vytvoríme si pole $f[]$, kde $f[x]$ označuje minimálny čas potrebný na dosiahnutie cieľa zo zastávky x. $f[n] = 0$ Pre znižujúce sa x postupne spočítame $f[x] = \min\{f[x+1], f[x+2], \ldots, f[x+k]\} + t_x$, teda presne to isté, čo počítala rekurzívna f.

Toto spôsobí len konštantné zrýchlenie programu, ale povedie nás to k optimálnemu riešeniu. V niektorých prípadoch dokážeme pomocou dynamického programovania aj znížiť pamäťovú zložitosť programu.

Listing programu (Python)

```
def dasa(k):
    # pole velkosti n+k je praktickejsie, nemusime specialne osetrovat pripad: dalsi > n
    f = [0] * (n+k)
    for x in range(n-1, 0, -1):
        f[x] = min( f[dalsi] for dalsi in range(x+1, x+k+1)) + cakaj[x]
    return f[1] <= t
```

Nepočítajme nič dvakrát, časť tretia

Ešte stále sa opakujeme? Áno:

$$f[x] = \min\{f[x+1], f[x+2], f[x+3], \ldots, f[x+k]\} + t_x$$

$$f[x+1] = \min\{f[x+2], f[x+3], \ldots, f[x+k], f[x+k+1]\} + t_{x+1}$$

Ak už vieme minimum pre $x+1, \ldots, x+k$, minimum pre $x+2, \ldots, x+k+1$ by sme mohli vedieť spočítať rýchlejšie. Totiž, ak je minimom jedno z čísel $f[x+2], f[x+3], \ldots, f[x+k]$, tak

$$\min\{f[x+2], \ldots, f[x+k+1]\} = \min\{\min\{f[x+1], \ldots, f[x+k]\}, f[x+k+1]\}$$

a teda na spočítanie nového minima by nám stačilo jedno porovnanie (s $f[x+k+1]$).

Jediný nepríjemný prípad je, ak je $\min\{f[x+1], \ldots, f[x+k]\} = f[x+1]$. Vtedy by sme potrebovali vedieť, aký bol druhý najmenší prvok z $f[x+1] \ldots, f[x+k]$

Budeme postupovať sprava doľava, tak ako doteraz a počítať hodnoty $f[x]$ – minimálny čas potrebný na presun z x do cieľa aj s časom čakania t_x. Chceli by sme si pamätať niekoľko zastávok, ktoré sa môžu niekedy stať minimom a vedieť tieto hodnoty aktualizovať pre $f[x-1]$.

Budeme si udržovať pozície zastávok, na ktorých je závislá hodnota $f[x]$, teda niektoré z $x+1, \ldots, x+k$ a pamätať si ich budeme usporiadané podľa ich hodnôt f v štruktúre **deque** – obojsmerný spájaný zoznam.

Vieme z neho vyberať a vkladať do neho prvky z oboch strán. Nech sú naľavo tie zastávky, ktoré majú najmenšie f, teda tie, z ktorých sa najrýchlejšie dostaneme do cieľa.

Na začiatku algoritmu sme na zastávke $n-1$, v deque si pamätáme zastávku n, $f[n]=0$.

Novú hodnotu $f[x]$ spočítame vždy jednoducho, ako minimálny čas, potrebný na cestu do cieľa z jednej z k zastávok napravo od x a čas čakania na x, čo je $f[deque.left]+t_x$.

Po výpočte $f[x]$ musíme deque aktualizovať. Chceme do nej niekam vložiť zastávku x (lebo hodnota $f[x]$ bude potrebná pre zastávky naľavo od x). Keďže žiadnu zastávku, ktorú máme v deque nebudeme vo výpočtoch používať dlhšie ako x, môžeme z deque vyhodiť všetky zastávky, ktoré potrebujú väčší čas na dosiahnutie cieľa – žiadna z týchto zastávok by sa už totiž nemohla stať novým minimom. Takisto zastávky, ktoré sú ďalej ako k od x už nikdy nepoužijeme a potrebujeme ich odstrániť z konca aj zo začiatku deque.

Vyhodíme tak všetky nepotrebné zastávky z konca a zo začiatku a nakoniec vložíme x.

Listing programu (Python)

```
from collections import deque

def dasa(k):
    f = [0] * (n+1)
    mozne_mini = deque([n])

    for x in range(n-1, 0, -1):
        f[x] = f[mozne_mini[0]] + cakaj[x]

        # odstranime zastavky z konca, ktore nikdy nebudu minimom
        while len(mozne_mini) > 0 and (f[mozne_mini[-1]] > f[x] or mozne_mini[-1] - (x-1) > k):
            mozne_mini.pop()
        # odstranime zastavky zo zaciatku, ktore nikdy nebudu minimom
        while len(mozne_mini) > 0 and mozne_mini[0] - (x-1) > k:
            mozne_mini.popleft()
        # pridame nove potencialne minimum
        mozne_mini.append(x)

    return f[1] <= t
```

V tomto riešení každú zastávku najviac raz vložíme do deque a najviac raz ju vyberieme. Preto bude mať funkcia $dasa(k)$ časovú zložitosť $O(n)$. Ak použijeme binárne vyhľadávanie na nájdenie najmenšieho k, celý program potrebuje len čas $O(n \log n)$.

Jano

8. Odrezané dážďovky

(max. 12 b za popis, 8 b za program)

Táto úloha bola dosť o premýšľaní a to bolo na nej super. Navyše, málokedy sa nám v KSP podarí taká pekná úloha s množstvom rôznych možných prístupov od nesprávnych riešení po správne riešenia so zložitosťami od $O(2^n)$ cez $O(n^3)$, $O(n^2)$, $O(n\sqrt{n})$, $O(n^{1.25})$, $O(n \log n)$ až po $O(n)$. Premennou n označujeme v celom vzorovom riešení dĺžku vstupu (dĺžku dážďovky). Všetky tieto riešenia so sebou nesú nejaké ponaučenie, tak vám odporúčame pozorne čítať. Text je možno trochu dlhší, pretože sa snažíme poriadne vysvetliť všetky detaily. Nemusíte ho celý čítať naraz.

Medzi nesprávne riešenia patrí množstvo pažravých prístupov, napríklad *idem od začiatku dážďovky a vždy useknem čo najkratší kus, aby som ich mal na konci čo najviac.* Protipríkladom je napríklad postupnosť 54638291 ktorú by sme pažravo nasekali 5,46,38291 a správne je 54,63,82,91. Pažravé riešenia v tejto úlohe nefungujú.

Pomalé riešenie $O(n^2)$

Určite by ste všetci zvládli naprogramovať riešenie, ktoré rekurzívne skúša všetky možnosti. V tomto rekurzívnom riešení sa dookola pýtame takéto otázky: *Na koľko najviac kusov môžeme nasekať prvých i-cifier vstupu, ak posledný kus má j-cifier?* Pričom riešenie úlohy je maximum pre všetky možné j a pre $i=n$. Odpoveď na takúto otázku si označíme `P[i][j]` a vieme ju ľahko spočítať z iných hodnôt P, pre iné i a j. `P[i][j]` je maximum z `P[i-j][k]+1` pre všetky $k \geq 1$ také, že posledný kus kratšej dážďovky je menší ako posledný kus dlhšej. Očividne môžeme použiť všetky $k \leq j$, pretože menejciferné číslo je istotne menšie ako viacciferné. A to, či môžeme uvažovať aj hodnotu `P[i-j][j]` zistíme porovnaním dvoch reťazcov dĺžky j.

Inak povedané, hodnotu `P[i][j]` vieme zistiť v čase $O(j)$[5], na základe iných hodnôt P. A takto vieme spočítať všetky P pre všetky $1 \leq j \leq i \leq n$, čo je $O(n^2)$ možností. Naťukáme túto myšlienku do počítača a dostaneme riešenie so zložitosťou $O(n^3)$.

Ha! S minimálnou úpravou kódu vieme dostať riešenie so zložitosťou $O(n^2)$. Stačí uvažovať také sekania, kde dĺžka každého kusu dážďovky bude $O(\sqrt{n})$, presnejšie $j \leq \sqrt{2n}+1$ (čoskoro si povieme, prečo to stačí) Potom zložitosť tohto riešenia je $O(n\sqrt{n}\sqrt{n}) = O(n^2)$.

[5] *ouj é :D*

Listing programu (C++)

```
#include<iostream>
#include<algorithm>
#include<vector>
#include<string>
#include<cmath>
using namespace std;
#define NEDASA -47

string dazdovka; // vstup
vector<vector<int> > P; // pocet kusov do pozicie i, ak posledny kus bol dlhy j

// porovnaj cisla, ktore maju 'dlzka' cifier a zacinaju na poziciach i1, i2
bool mensie(int i1, int i2, int dlzka) {
    for(int i = 0; i < dlzka; ++i)
        if (dazdovka[i1+i] != dazdovka[i2+i])
            return dazdovka[i1+i] < dazdovka[i2+i];
    return false;
}

// rekurzivne spocitame P[i][j]
int pocet(int i, int j) {
    if (i == j) return 1;
    if (i < j) return NEDASA;
    // ak sme uz odpoved spocitali, nebudeme pocitat znova
    if (P[i][j] != -1) return P[i][j];

    int najlepsie = NEDASA;
    for(int k = 1; k<j; k++)
        najlepsie = max(najlepsie, pocet(i-j, k)+1);
    if (i>=2*j && !mensie(i-j, i-2*j, j))
        najlepsie = max(najlepsie, pocet(i-j, j)+1);

    if (najlepsie < 0) najlepsie = NEDASA;
    return P[i][j] = najlepsie;
}

int main() {
    cin >> dazdovka;
    int n = dazdovka.size();
    int odmocnina = sqrt(2*n)+2;
    P = vector<vector<int> > (n+1, vector<int>(odmocnina+1, -1));
    int najlepsie = 0;
    for(int j = 1; j < odmocnina; ++j)
        najlepsie = max(najlepsie, pocet(n, j));
    cout << najlepsie << endl;
}
```

Príprava na rýchlejšie riešenie

Na rýchlejšie riešenia ako $O(n^2)$ si potrebujeme spraviť to sľubované premýšľanie. Ukážeme si, že stačí uvažovať len niektoré sekania dážďovky.

Mimochodom, dĺžkou kusu/čísla rozumieme počet jeho cifier. Často budeme využívať fakt, že ak číslo B je dlhšie ako A, tak B je určite väčšie.

Pozorovanie 1.

Stačí nám uvažovať také riešenia, kde sa dĺžky susedných kusov dážďovky líšia najviac o 2.

Ak by sa totiž niektoré dva susedné kusy A, B líšili o 3 alebo viac, môžeme ľavý kus A predĺžiť o 1 a pravý kus B skrátiť o 1. Tým pádom všetky ostatné kusy ostanú nezmenené a určite sme nespôsobili žiadny problém pri porovnaniach ktorýchkoľvek z kusov. Číslo A je naďalej väčšie od kusov naľavo, pretože už predtým bolo väčšie a teraz sme ho len zväčšili. Podobne B je menšie od kusov napravo. Napokon A je menšie ako B, pretože B má aspoň o jednu cifru viac. Opakovaním takýchto posunov o 1 dosiahneme, že rozdiely dĺžok sú všade najviac dva.

Podobne, nikdy v optimálnom riešení nebude prvý úsek dlhší ako dve cifry, pretože ak by mal viac ako dve cifry, tak ho určite môžeme rozdeliť na prvú cifru a zvyšok, čím dostaneme lepšie riešenie.

Teraz vieme spraviť prvý odhad na dĺžku kusov. Keďže prvý kus je dlhý najviac dva a každý ďalší môže byť najviac o dva dlhší, najväčšie možné dĺžky dosiahneme, ak budú postupne počty cifier v úsekoch $2, 4, 6, 8, 10, 12, ..., 2k$.

Celková dĺžka dážďovky potom bude $k(k+1)$. Opačne, ak vieme, že dĺžka dážďovky je n, tak $k < \sqrt{n}$ a dĺžka najväčšieho úseku v nejakom optimálnom riešení bude najviac $2\sqrt{n}$. Neskôr si ukážeme, že je to dokonca najviac $\sqrt{2n} + 1$.

Zároveň ľahko nájdeme vstupy, kde v optimálnom riešení má najväčšie číslo viac ako $\sqrt{2n} - 1$ cifier, takže tento odhad už sa veľmi nedá zlepšiť.

Pozorovanie 2.

V skutočnosti nemusíme hľadať čo najvyšší počet kusov dážďovky, ako nabáda zadanie, ale môžeme nájsť také nasekanie, aby posledný kus bol čo najmenší. Je to totiž takmer to isté.

Kľúčovým pozorovaním tejto úlohy je to, že nám stačí uvažovať len také riešenia, ktoré končia najmenším možným číslom, resp. najkratším možným kusom[6]. Predstavme si, že sme nasekali dážďovku tak, aby bol posledný kus čo najkratší. Nech má posledný kus dĺžku m. Zároveň spomedzi všetkých nasekaní, ktoré majú posledný kus dlhý m vyberieme to, ktoré má najväčší počet kusov. Toto nasekanie volajme A a označíme si jednotlivé pozície rezov $i_1, i_2, i_3, \ldots, i_k$.

Čo by sa stalo, keby toto nasekanie nebolo optimálne z hľadiska počtu kusov? Ak sa nám podarí prísť ku sporu, zistíme, prečo toto nasekanie muselo byť optimálne.

Pre spor teda predpokladajme, že existuje iné nasekanie B na pozíciách $j_1, j_2, \ldots, j_{k+1}$, ktoré má viac kusov. Z predpokladov vieme, že posledný kus B je dlhší ako posledný kus A, preto $j_{k+1} < i_k$. Navyše si označme myslený nultý rez úplne na začiatku dážďoviek $i_0 = j_0 = 0$.

Pozrime sa na najmenšiu dvojicu indexov (a, b), pre ktoré platí, že $1 \leq a \leq b \leq k+1$ a zároveň $i_a > j_b$. (Pri výbere najmenšej najprv zoraďujeme podľa a, potom podľa b.) Keďže existuje aspoň jedna taká dvojica (k, k+1) a možných dvojíc je konečne veľa, tak určite existuje najmenšia.

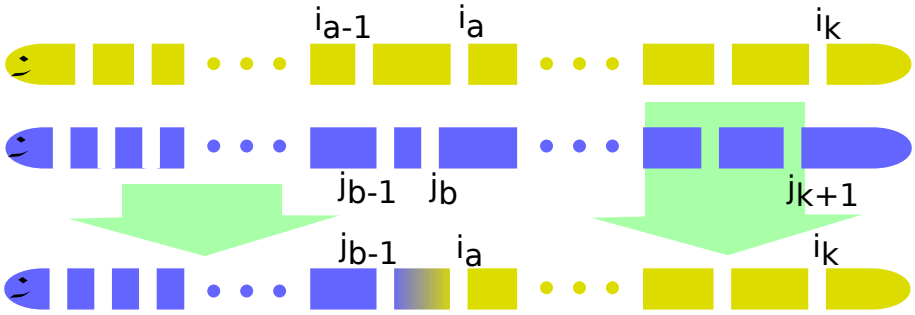
Vieme tiež, že $i_{a-1} \leq j_{b-1}$, inak by (a, b) nebola najmenšia dobrá dvojica. Taktiež $j_{b-1} < i_a$, pretože $i_b < i_a$.

No ale potom vieme vyrobiť nové nasekanie C, ktoré bude pozostávať z rezov $j_1 \ldots j_{b-1}$ a rezov $i_a \ldots i_k$. O tomto nasekaní vieme povedať, že

- Je korektné a spĺňa nerovnosti so zadania. Totiž, číslo pozostávajúce z cifier medzi rezmi j_{b-1} a i_a je väčšie ako číslo medzi rezmi j_{b-1} a j_b a preto je aj väčšie od čísla pred ním. Taktiež je toto číslo menšie alebo rovné od čísla medzi rezmi i_{a-1} a i_a a tým pádom aj menšie ako číslo za ním. Všetky ostatné nerovnosti platia, pretože platili v pôvodných dážďovkách.
- Má $k+1$ kusov, čiže je dlhšie ako nasekanie A.
- Končí rovnakým číslom ako nasekanie A.

A to je spor s tým ako sme vybrali nasekanie A. □

Ukázali sme, že nasekanie A je optimálne, pretože ak by existovalo nasekanie s väčším počtom kusov, dostali by sme sa do sporu. Pre lepšiu predstavu si pozrite ilustračný obrázok ku dôkazu.



Dôsledky

Vďaka druhému pozorovaniu, vieme, že nasledujúci algoritmus funguje správne. Spočítame, akým najkratším kusom vieme ukončiť nasekanie dážďovky. Odsekneme ten najkratší možný kus. Opakujeme postup s kratšou dážďovkou.

Druhý dôsledok je tesnejší odhad pre počet cifier všetkých čísel. Keby mali jednotlivé kusy dĺžky postupne $1, 2, 3, 4, \ldots$ prípadne s jedným vynechaným číslom aby to pasovalo na dĺžku dážďovky, tak dostaneme riešenie, ktoré končí číslom kratším ako $\sqrt{2n}+1$. Tým pádom riešenie, ktoré končí najkratším možným kusom, končí tiež kusom kratším ako $\sqrt{2n}+1$. A tým pádom existuje aj optimálne riešenie s kratším kusom na konci. A keďže posledný kus je zo všetkých najdlhší, tak sme dostali horný odhad na dĺžku všetkých kusov.

Tretí dôsledok je rýchlejšie riešenie úlohy.

Rýchlejšie riešenie – $O(n\sqrt{n})$

Pamätáte si ešte pôvodné rekurzívne riešenie, kde sme sa snažili pre každé i, j zistiť, *na koľko najviac kusov môžeme nasekať prvých i-cifier vstupu, ak posledný kus má j-cifier*? Tak teraz už vieme, že nemusíme rozlišovať, koľko cifier má posledný kus – stačí uvažovať rozsekanie s najkratším možným koncom.

Takže po novom už budeme počítať $P[i]$ – na koľko najviac kusov môžeme nasekať prvých i cifier dážďovky a $D[i]$, koľko najmenej cifier môže mať posledný kus.

[6]Pozor, je to niečo úplne iné, ako snažiť sa začať najmenším možným číslom. To by nefungovalo.

Celková zložitosť algoritmu bude $O(n\sqrt{n})$, pretože každé $P[i]$ a $D[i]$ spočítame v čase $O(D[i])$ z týchto hodnôt pre menšie i. Najprv prejdeme $O(D[i])$ menších hodnôt $P[j], D[j]$ a potom ešte potrebujeme porovnať dve $D[i]$-ciferné čísla, aby sme zistili, či môžu byť posledné dva úseky rovnako dlhé. A už vieme, že $D[i] \leq \sqrt{2n} + 1$.

Ak chceme program ďalej zrýchliť, potrebujeme vedieť rýchlo porovnávať dlhé čísla a tiež potrebujeme prechádzať menej hodnôt pri počítaní jednej dvojice $P[i], D[i]$. Už sme si dokázali, že nemá zmysel, aby sa dĺžky susedných čísel líšili o viac ako dva. Takže nemusíme skúšať všetky možnosti, stačí pozrieť len hodnoty $j \in \{D[i] - 2, D[i] - 1, D[i]\}$. Bohužiaľ, keď nepoznáme $D[i]$, tak nevieme, ktoré možnosti skúšať.

Preto budeme tieto hodnoty počítať trocha inak – odpredu. Pekne od začiatku začneme počítať P a D pre hodnoty $1, 2, 3, 4, \ldots$ A vždy, keď spočítame nejakú hodnotu, $P[i]$ a $D[i]$, vieme, ktoré vyššie hodnoty ovplyvnia. Aktualizujeme teda, hodnoty o $D[i]$, $D[i] + 1$ a $D[i] + 2$ vyššie, čo už môžeme spraviť, lebo už poznáme $D[i]$. Teda napríklad vieme, že $P[i + D[i] + 1]$ ani $P[i + D[i] + 2]$ nebude menej ako $P[i] + 1$. A zároveň vieme, že ak číslo pozostávajúce z cifier $i - D[i]$ až i (polouzavretý interval) je menšie ako číslo pozostávajúce z cifier i až $i + D[i]$, tak aj $P[i + D[i]] > P[i] + 1$.

Tieto myšlienky vieme zapísať do nasledujúceho programu, ktorý už je pomerne rýchly a získa 7 z 8 bodov.

Listing programu (C++)

```
#include<iostream>
#include<algorithm>
#include<vector>
#include<string>
using namespace std;
#define For(i, n) for(int i = 0; i<int(n); ++i)

string dazdovka; // vstup

// porovnaj cisla, ktore maju 'dlzka' cifier a zacinaju na poziciach i1, i2
bool mensie(int i1, int i2, int dlzka) {
    For(i, dlzka)
        if (dazdovka[i1+i] != dazdovka[i2+i])
            return dazdovka[i1+i] < dazdovka[i2+i];
    return false;
}

int main() {
    cin >> dazdovka;
    // pristupujeme do poli aj za n-tu poziciu, ale nikdy nie dalej ako rezerva
    int rezerva = dazdovka.size() + dazdovka.size()/10 + 100;
    vector<int> P (rezerva); // pocet kusov do pozicie i
    vector<int> D (rezerva); // dlzka kusa konciaceho na pozicii i
    P[0] = 0;
    D[0] = 1;

    For(i, dazdovka.size()+1) {
        if (!mensie(i, i-D[i], D[i])) {
            D[i+D[i]] = D[i];
            P[i+D[i]] = P[i]+1;
        }
        // vieme, ze predosle hodnoty neboli mensie,
        // lebo sme ich mohli najst len s mensim i
        D[i+D[i]+1] = D[i]+1;
        D[i+D[i]+2] = D[i]+2;
        P[i+D[i]+1] = P[i]+1;
        P[i+D[i]+2] = P[i]+1;
    }
    cout << P[dazdovka.size()] << endl;
}
```

Vzorové riešenie

Jediná vec, ktorá teraz spomaľuje beh programu je porovnávanie reťazcov. V najhoršom prípade porovnáme všetkých $O(D[i])$ znakov, čo trvá $O(\sqrt{n})$ času. Ale my predsa vieme porovnávať aj rýchlejšie, ak si dážďovku trocha predspracujeme.

Možností, ako zrýchliť porovnávanie je mnoho. Jeden z tých jednoduchších spôsobov je spraviť väčšie bloky cifier, ktoré budeme schopní porovnávať naraz.

Zvolíme si veľkosť bloku k, následne zoberieme všetky možné kusy dážďovky dĺžky k. Tieto kusy si radix-sortom utriedime a spočítame indexy – pre každú k-ticu si zapamätáme, koľká najmenšia k-tica zo všetkých to je.

Potom, keď budme chcieť porovnať dve dlhé čísla, tak môžeme najprv porovnať prvú k-ticu cifier, potom druhú k-ticu cifier a tak ďalej. Každú k-ticu porovnáme jedným pozretím do poľa indexov. Možno nám ešte na konci ostane niekoľko (najviac k) cifier, ktoré budeme musieť porovnať po jednej.

Časová zložitosť predpočítania indexov je $O(nk)$ a časová zložitosť jedného porovnania reťazcov dĺžky $\sqrt{n}$ je $O(k + \sqrt{n}/k)$, čo je najmenej pre $k = \sqrt[4]{n}$.

Takže dostaneme algoritmus s časovou zložitosťou $O(n\sqrt[4]{n}) = O(n^{1.25})$. Pri implementácii si treba dávať pozor, že keď porovnávame dve k-tice, potrebujeme rozlišovať nielen dve možnosti menší/nie menší, ale tri

možnosti väčší/menší/rovnaký.

V praxi je radix-sort pomerne pomalý, má zlú konštantu, takže v zdrojovom kóde sme nastavili konštantu k na 10 pre lepší čas (teoreticky by bolo lepšie $k = 30$, pretože to je približne $\sqrt[4]{10^6}$). Takto vyzerá funkcia `predpocitaj`, ktorú zavoláme raz hneď po načítaní vstupu a funkcia `mensie`, ktorá nahradí pôvodnú funkciu `mensie`.

Listing programu (C++)

```
// skok by mal byt priblizne stvrta odmocnina z n
#define skok 10
vector<int> poradie;

void predpocitaj(int n) {
    vector<int> D(n+skok+47,0);
    For(i,n) D[i] = dazdovka[i]-'0';

    vector<int> kusy (n);
    vector<int> vrecka[10];
    For(i, n) kusy[i] = i;

    For(i, skok) {
        For(j, n) vrecka[D[kusy[j]+skok-i-1]].push_back(kusy[j]);
        int p = 0;
        For(j, 10) {
            for(const auto &x : vrecka[j]) kusy[p++] = x;
            vrecka[j].clear();
        }
    }
    poradie.resize(n);
    For(i, n) {
        bool rovnake = (i > 0);
        if (rovnake) For(j, skok)
            if (D[kusy[i-1]+j] != D[kusy[i]+j]) {
                rovnake = false;
                break;
            }
        if (rovnake) poradie[kusy[i]] = poradie[kusy[i-1]];
        else poradie[kusy[i]] = i+1;
    }
    poradie.resize(n+n/100+47, 0);
}

bool mensie(int i1, int i2, int dlzka) {
    int i = 0;
    for(; i < dlzka-skok; i+=skok) {
        if (poradie[i1+i] != poradie[i2+i])
            return poradie[i1+i] < poradie[i2+i];
    }
    for(; i < dlzka; ++i) {
        if (dazdovka[i1+i] != dazdovka[i2+i])
            return dazdovka[i1+i] < dazdovka[i2+i];
    }
    return false;
}
```

Táto myšlienka sa dala posunúť ešte ďalej a namiesto jednej sady blokov dĺžky k sme mohli mať bloky dĺžok $1, 2, 4, 8, 16, 32, \ldots, 2^{\log n}$. Predpočítať poradie blokov všetkých dĺžok vieme spraviť v čase $O(n \log n)$, pretože vieme zistiť poradie blokov dĺžky $2k$ v $O(n)$, ak poznáme poradie blokov dĺžky k.

Porovnávanie čísel, resp. funkcia `mensie` by potom bežala tiež v čase $O(\log n)$, čím by sme dosiahli výbornú celkovú zložitosť $O(n \log n)$.

Takéto riešenie stačilo na plný počet bodov aj za popis aj za program. Predošlé riešenie $O(n^{1.25})$ by dostalo za program tiež plný počet a za dobrý popis by dostalo 11 bodov z 12.

Iný spôsob, ako dosiahnuť rýchle riešenie je hashovanie. Môžeme jednoduchým cyklom spočítať hash pre každý blok dlhý k cifier a potom zasa vieme porovnávať v čase $O(k + \sqrt{n}/k)$.

Dobrá hash je napríklad $\sum_{i=1}^{k} c_i 47^i \mod p$, kde c_i je i-ta cifra čísla a p je nejaké veľké prvočíslo, napr. $10^9 + 9$.

Spočítanie hash-hodnôt vieme spraviť v $O(n)$ a tak je to v praxi oveľa rýchlejšie ako radix-sort. Nevýhoda je, že pri použití hashe nemáme istotu, že program odpovie správne, pretože dve rôzne k-ciferné čísla môžu mať rovnakú hash. Taktiež hash čísla nevie povedať, ktorá k-tica je menšia, vie len overovať rovnakosť. To nám však príliš nevadí.

Celková zložitosť tohto algoritmu by bola tiež $O(n^{1.25})$.

Pomocou hash-funkcií vieme spraviť aj jednoduché riešenie so zložitosťou $O(n \log n)$. Najprv si spočítame hash pre každý prefix dážďovky $H_i = \sum_{i=1}^{k} c_i 47^i \mod p$, kde c_i je i-ta cifra dážďovky.

Potom, ak chceme zistiť, či sú nejaké dva intervaly dážďovky (a, b) a (c, d) rovnaké, stačí zistiť, či $((H_b - H_a) \cdot 47^c - (H_d - H_c) \cdot 47^a) \mod p == 0$.

Potom, ak ideme porovnávať dve dlhé čísla, vieme binárne vyhľadať prvú cifru v ktorej sa líšia a porovnať len tú. Takto dosiahneme zložitosť predpočítania $O(n)$ a zložitosť jedného porovnania $O(\log n)$.

Listing programu (C++)

```
typedef long long ll;
vector<ll> H; // hash prefixov dazdovky
vector<ll> powp; // mocniny cisla 47
ll prime = 47;
ll mod = 1000000009;

void predpocitaj(int n, int rezerva) {
    H.resize(rezerva+1);
    powp.resize(rezerva+1);
    powp[0] = 1;
    For(i, rezerva) powp[i+1] = (powp[i]*prime) % mod;
    H[0] = 0;
    For(i, rezerva) {
        int c = (i>n)?0:(dazdovka[i]-'0');
        H[i+1] = (H[i] + (powp[i]*c)) % mod;
    }
}

bool mensie(int i1, int i2, int dlzka) {
    // binarne vyhladame prvu cifru, kde sa lisia
    int b = 0, e = dlzka, m;
    while(e-b > 1) {
        m = (b+e)/2;
        ll h1 = ((H[i1+m] - H[i1]) * powp[i2]) % mod;
        ll h2 = ((H[i2+m] - H[i2]) * powp[i1]) % mod;
        if ((h1-h2)%mod == 0) b = m;
        else e = m;
    }
    return dazdovka[i1+b] < dazdovka[i2+b];
}
```

Bonusové riešenie

Všetkých vás určite zaujíma, či sa úloha nedala vyriešiť aj v čase $O(n)$. Áno dala, ale zložitosť riešenia výrazne presahuje kategóriu O.

Bolo potrebné dotiahnuť do konca zrýchlenie porovnávania. Povieme si aspoň hlavnú myšlienku pre skúsených čitateľov. Nezúfajte, ak teraz neporozumiete skratkám a názvom algoritmov, keď budete starší, môžete sa k tomuto vzoráku vrátiť a skúsiť to znova.

Zostrojíme si pre dážďovku Suffixové pole. Potom, keď budeme chcieť porovnať dva rovnako dlhé kusy dážďovky, najprv overíme, či náhodou nie sú úplne rovnaké. Ak nie sú, tak vieme, že ich poradie je rovnaké ako poradie k nim prislúchajúcich suffixov. Takže ich porovnáme pomocou Suffixového poľa v $O(1)$.

Ako však v $O(1)$ zistiť, či sú kusy dážďovky rovnaké? Jednou možnosťou by bolo opäť hashovať, ale to nechceme, lebo hash nezaručuje správnosť riešenia. Namiesto toho si na začiatku spočítame LCP pole ku Suffixovému poľu a overíme, či minimum zo všetkých LCP medzi suffixami prislúchajúcimi porovnávaným kusom dážďovky je väčší alebo rovný ako dĺžka porovnávaných kusov. Ak áno, tak sú rovnaké, inak sú odlišné.

Takže potrebujeme rýchlo počítať minimum z nejakých intervalov v nejakom poli. Inak povedané, potrebujeme riešiť známy problém Range-Minimum-Query v čase $O(n)$ na predspracovanie a $O(1)$ na query. Pekné riešenie RMQ spomínajú páni Fischer a Heun vo svojom článku z roku 2006. Vraveli sme, že to presahuje kategóriu O.

Ničmenej, dá sa to celé nakódiť, ale v praxi pre $n \leq 10^6$ je už len zostrojenie Suffixového poľa pomalšie ako pôvodné $O(n\sqrt{n})$ riešenie. Takže máme pekný teoretický výsledok, že sa to celé dá v $O(n)$ ale ponechajme radšej tento výsledok v teoretickej rovine.