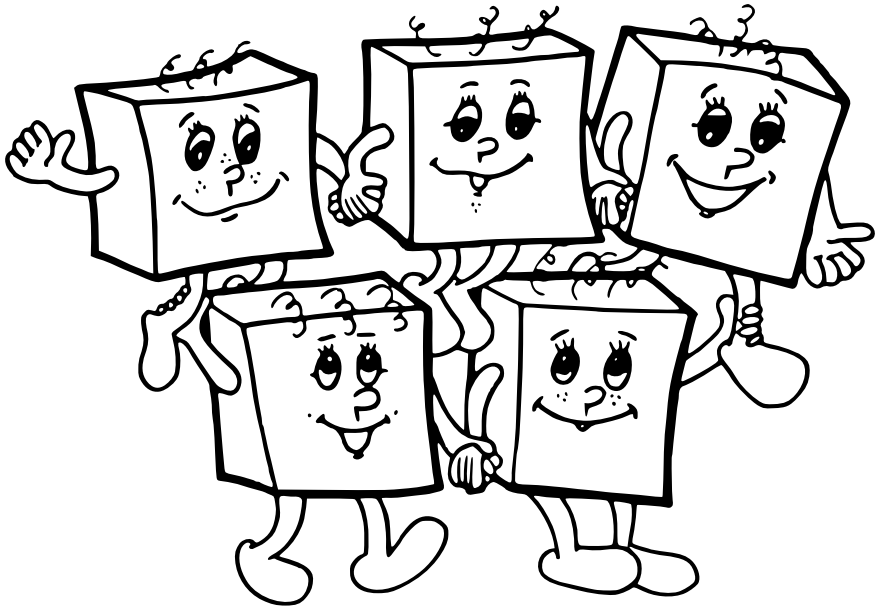
OLYMPIÁDA V INFORMATIKE

NA STREDNÝCH ŠKOLÁCH



23. ročník

školský rok 2007/08

riešenia domáceho kola
kategória A

- **Olympiáda v informatike** je od školského roku 2006/07 samostatnou súťažou. Predchádzajúcich 21 ročníkov tejto súťaže prebiehalo pod názvom **Matematická olympiáda, kategória P** (programovanie).
- Oficiálnu **webstránku** súťaže nájdete na `http://www.ksp.sk/oi/`.

Riešenia kategórie A

Tento materiál má pomôcť učiteľom na školách pri príprave riešiteľov domáceho kola. Taktiež slúži ako podklad pre opravovanie úloh členmi krajských výborov OI.

Žiakom možno tento materiál poskytnúť až po termíne stanovenom pre odovzdanie riešení úloh (15. novembra 2007). Po tomto termíne bude aj zverejnený na webstránke súťaže.

A-I-1 O zdanlivom kopci

Zadanou úlohou je vybrať z danej postupnosti čo najdlhšiu podpostupnosť, ktorá by najskôr rástla a potom klesala.

Keby sme vedeli, ktorý prvok je v našej podpostupnosti ten najväčší („vrchol kopca“), mali by sme ľahšiu úlohu: z časti postupnosti od začiatku po vrchol vybrať čo najdlhšiu rastúcu podpostupnosť končiacu „vrcholom“, a zo zvyšku zase vybrat čo najdlhšiu klesajúcu podpostupnosť. No a vyberať klesajúcu podpostupnosť je to isté ako vyberať rastúcu idúc sprava doľava.

Preto nám stačí riešiť nasledujúcu jednoduchšiu úlohu: K danej postupnosti pre každé i spočítame dĺžku d_i najdlhšej rastúcej podpostupnosti, ktorá končí členom a_i.

Ukážeme si najskôr pomalšie riešenie tejto úlohy, potom jeho efektívnejšiu verziu.

Pomalšie riešenie

Zjavne $d_1 = 1$. Ak už vieme hodnoty d_1 až d_k pre nejaké k, hodnotu d_{k+1} spočítame nasledovne:

Predstavme si, že už máme nájdenú najdlhšiu podpostupnosť končiacu prvkom a_{k+1}. Pozrime sa na ňu a zakryme si posledný člen. To, čo teraz vidíme, musí byť opäť nejaká rastúca podpostupnosť. A nie len tak hocijaká. Nech jej posledný člen je a_x. Potom to, čo vidíme, musí byť (jedna možná) najdlhšia

rastúca podpostupnosť končiaca a_x. Jej dĺžka je teda d_x, a dĺžka našej pôvodnej podpostupnosti je $d_x + 1$.

My síce nepoznáme x, ale tu je ľahká pomoc: Vyskúšame všetky možné x a vyberieme si tú najlepšiu možnosť. A ktoré sú to tie „všetky možné“ x? V prvom rade musí byť $1 \leq x \leq k$, no a navyše musí platiť $a_x < a_{k+1}$, aby bola nová podpostupnosť naďalej rastúca. Dostávame teda:

$$d_{k+1} = \max_{1 \leq x \leq k,\ a_x < a_{k+1}} d_x + 1$$

Algoritmus, ktorý spočíta hodnoty d_i použitím tohoto vzťahu, má časovú zložitosť $O(N^2)$, kde N je počet členov spracúvanej postupnosti. Takéto riešenie mohlo získať nanajvýš 7 bodov.

Lepšie riešenie

Na zrýchlenie vyššie popísaného algoritmu použijeme nasledujúce pozorovanie: Ak vieme vybrať dve rastúce podpostupnosti rovnakej dĺžky, „lepšia“ je tá, ktorá končí menšou hodnotou. Totiž ak vieme po pridaní nasledujúcich členov postupnosti nejako predĺžiť tú „horšiu“, vieme rovnako predĺžiť aj tú „lepšiu“.

V každom okamihu si teda stačí pre každú možnú dĺžku pamätať jednu rastúcu podpostupnosť – tú „najlepšiu“, čiže končiacu najmenšou možnou hodnotou.

Presnejšie, nech m_i je najmenšia hodnota, akou môže končiť i-prvková rastúca podpostupnosť vybratá z doteraz spracovaných prvkov.

Na začiatku je $m_0 = 0$ a $\forall i > 0;\ m_i = \infty$.

Teraz budeme postupne po jednom spracúvať prvky danej postupnosti. Ako sa hodnoty m_i zmenia po spracovaní jedného jej členu?

Všimnime si najskôr, že v každom okamihu platí, že hodnoty m_i (ktoré sú rôzne od ∞) sú rastúce. Totiž ak vieme spraviť rastúcu podpostupnosť dĺžky i, ktorá končí hodnotou m_i, tak jej prvých $i-1$ členov tvorí rastúcu podpostupnosť dĺžky $i-1$, ktorá končí členom menším ako m_i, preto nutne $m_{i-1} < m_i$.

Nech je práve spracúvaný člen postupnosti x. Potom zjavne existuje práve jedno a také, že $m_a < x \leq m_{a+1}$.

Čo toto znamená? V prvom rade vieme, že doteraz „najlepšia“ vybraná podpostupnosť dĺžky $a + 1$ (a viac) končila číslom väčším alebo rovným x. Žiadnu takúto postupnosť nevieme predĺžiť hodnotou x, a teda hodnoty od m_{a+2} ďalej sa meniť nebudú.

Podobne sa nebudú meniť ani hodnoty od m_0 po m_a, vrátane. Tieto sú všetky už teraz menšie ako x, takže ich zlepšiť nevieme.

Jediné, čo sa teda zmení, je, že teraz vieme vybrať rastúcu postupnosť dĺžky $a+1$, ktorá končí hodnotou x. Odteraz teda bude $m_{a+1} = x$. A zároveň vieme, že $a + 1$ je dĺžka najdlhšej vybranej rastúcej podpostupnosti končiacej práve spracúvaným prvkom.

No a už máme náš algoritmus skoro hotový. Teraz si stačí len uvedomiť, že hodnoty m_i sú utriedené, a teda vieme nájsť hodnotu a binárnym vyhľadávaním v čase $O(\log N)$. Zvyšné úpravy už spravíme v konštantnom čase.

Potrebujeme spracovať N prvkov, časová zložitosť teda bude $O(N \log N)$.

Listing programu:

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

#define NEKONECNO 987654321

vector<int> rastuce(const vector<int> &A) {
  int N = A.size();
  vector<int> result(N);
  vector<int> M(N+1,NEKONECNO); M[0] = 0;
  for (int i=0; i<N; i++) {
    // binarnym vyhladavanim najdeme "a"
    int a = lower_bound( M.begin(), M.end(), A[i] ) - M.begin() - 1;
    M[a+1] = A[i];
    result[i] = a+1;
  }
  return result;
}

int main() {
  // nacitame vstup
  int N; cin >> N; vector<int> A(N); for (int i=0; i<N; i++) cin >> A[i];

  // spocitame dlzky pre rastuce podpostupnosti
  vector<int> B = rastuce(A);

  // spocitame dlzky pre klesajuce podpostupnosti
  reverse(A.begin(),A.end());
```

```
  vector<int> C = rastuce(A);
  reverse(C.begin(),C.end());

  // spocitame vysledok
  int res = 0;
  for (int i=0; i<N; i++) res = max(res, B[i]+C[i]-1 );
  cout << res << endl;
  return 0;
}
```

A-I-2 Rezervácie miesteniek

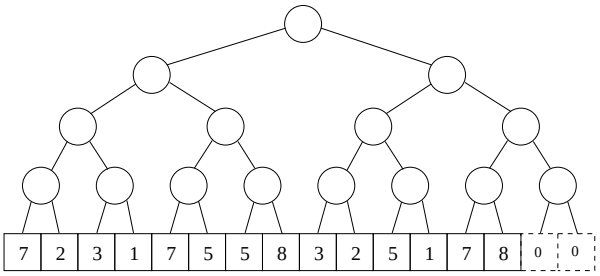
Naša železničná trať má N úsekov medzi stanicami. Keď nám príde nejaká požiadavka, potrebujeme sa pozrieť na úseky, ktoré obsahuje, a nájsť úsek, kde je voľného miesta najmenej. Ak je ho tam dosť, je ho dosť všade a požiadavku prijmeme. Naopak, ak ho tam dosť nie je, požiadavku musíme odmietnuť.

Na 4 body si stačilo pre každý úsek pamätať v poli počet obsadených miest. Na 7 bodov potrebujeme vedieť efektívnejšie odhaliť odmietané požiadavky (teda nájsť najplnší úsek), no a na 10 bodov budeme musieť efektívne spracúvať aj prijaté požiadavky.

Intervalový strom

Pre jednoduchosť budeme predpokladať, že N je mocnina dvoch. (Ak by nebolo, zväčšíme ho na najbližšiu mocninu dvoch. Uvedomte si, že tým sa zväčší menej ako na dvojnásobok pôvodnej hodnoty.)

Použijeme dátovú štruktúru známu pod menom *intervalový strom*. Ten bude vyzerať takto:

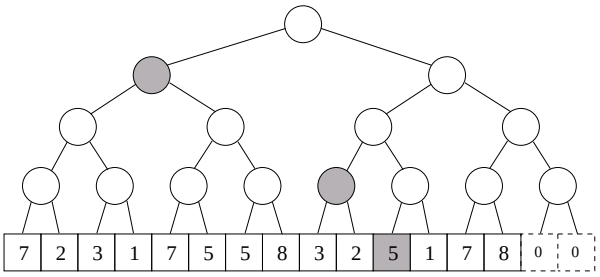


Listy intervalového stromu zodpovedajú jednotlivým úsekom trate. Všimnite

si, že vnútorný vrchol, ktorý je k úrovní nad listami, zodpovedá intervalu obsahujúcemu 2^k po sebe idúcich úsekov. Tie intervaly, ktoré zodpovedajú vrcholom nášho stromu, budeme volať *jednoduché*.

Načo je intervalový strom dobrý? Ukážeme, že ľubovoľný interval úsekov vieme šikovne „poskladať“ z jedoduchých intervalov.

Zaoberajme sa najskôr intervalom, ktorý obsahuje úseky od 1 po k. Tvrdíme, že tento vieme zložiť z najviac $\lg N$ jednoduchých intervalov. Toto dokážeme tak, že budeme z jeho ľavej strany odkrajovať čo najväčšie jednoduché intervaly, až kým sa neminie. Najjednoduchšie je to vidieť na príklade. Napr. interval „od 1 po 11“ vieme rozdeliť na „od 1 po 8“, „od 9 po 10“ a „od 11 po 11“.

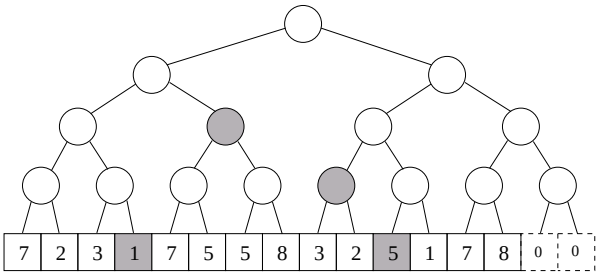


Vyznačené vrcholy zodpovedajú jednoduchým intervalom,
ktoré dokopy tvoria interval „od 1 po 11“.

Odhad počtu použitých intervalov vyplýva napríklad z toho, že v každom kroku odkrojíme viac ako polovicu intervalu.

Všimnite si, že postup krájania zodpovedá schádzaniu z koreňa dole po intervalovom strome. V každom vrchole sa pozrieme, či nerozkrájaná časť zadaného intervalu leží celá v ľavom podstrome. Ak áno, nič nekrájame a zlezieme doň. Ak nie, odkrojíme interval zodpovedajúci ľavému podstromu a zlezieme do pravého.

Vo všeobecnej situácii, keď chceme naskladať interval úsekov od k po l, budeme na tom podobne, vystačíme si s $2 \lg N$ jednoduchými intervalmi. Dôkaz je podobný, pôjdeme dole intervalovým stromom. Akonáhle niekedy zistíme, že zadaný interval zasahuje do oboch podstromov, rozkrojíme ho na dve časti. No a každá z častí už teraz zodpovedá jednoduchšiemu prípadu, ktorý sme rozobrali vyššie.

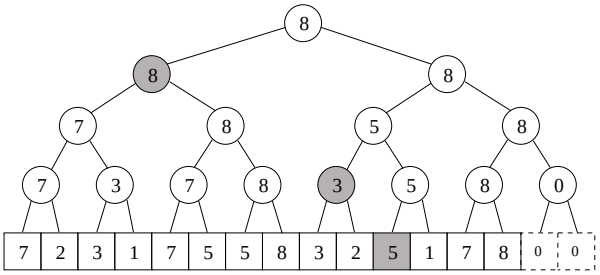


Všeobecný prípad: interval „od 4 po 11“.

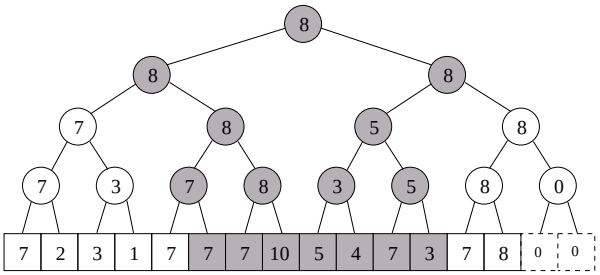
Ukázali sme teda, že v čase $O(\log N)$ vieme ľubovoľný interval rozdeliť na niekoľko častí, ktoré zodpovedajú vrcholom stromu.

Keby sme pre každý vrchol stromu vedeli, aké je maximum z hodnôt listov v jeho podstrome, vedeli by sme teda v čase $O(\log N)$ povedať maximum pre ľubovoľný interval úsekov. A v tom je celý trik intervalového stromu: Zvolili sme si niekoľko vhodných intervalov, z ktorých vieme efektívne naskladať odpoveď pre hocijaký iný interval.

Riešenie za 7 bodov je v tomto okamihu už triviálne. Spravíme si intervalový strom, v ktorom budú v listoch počty obsadených miest na jednotlivých úsekoch, a v každom vrchole si pamätáme maximum z hodnôt listov v jeho podstrome. Keď príde požiadavka „x y z“, v čase $O(\log N)$ zistíme maximum z hodnôt v intervale od $y+1$ po z. Ak je väčšie ako $M-x$, požiadavku odmietneme. V opačnom prípade potrebujeme strom upraviť. Zväčšíme hodnoty v listoch od $y+1$ po z a opravíme všetky vrcholy nad nimi. Toto vieme spraviť v čase $O(z-y)$, čo vieme zhora odhadnúť ako $O(N)$.



Intervalový strom s maximami pre jednoduché intervaly.
Maximum v intervale „od 1 po 11“ je rovné maximu zvýraznených políčok.

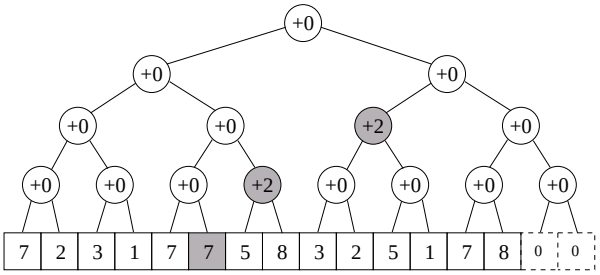


Situácia počas spracúvania prijatej požiadavky „2 5 12“:
Vo vyznačenom intervale (sivé štvorčeky) pribudli dvaja cestujúci.
Sivé krúžky označujú vrcholy, ktoré ešte treba (zdola nahor) prepočítať.

Celkovo má toto riešenie časovú zložitosť $O(P \log N + AN)$, kde P je počet všetkých a A je počet prijatých požiadaviek.

Efektívne spracovanie prijatej požiadavky

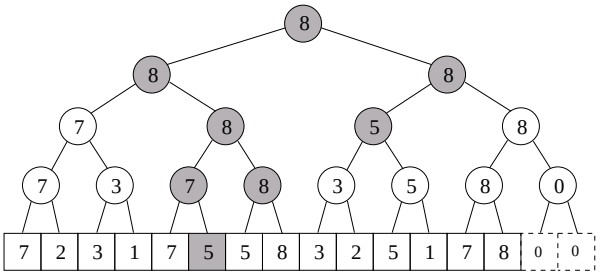
Zostáva ukázať, ako šikovnejšie upravovať informáciu o počtoch cestujúcich v prípade prijatia požiadavky. Trik je jednoduchý. Nebudeme ukladať informácie o počte cestujúcich len v listoch, ale v celom strome. Keď teda máme zväčšiť hodnoty v nejakom intervale, zväčšíme hodnoty v zodpovedajúcich jednoduchých intervaloch. V každom vrchole intervalového stromu si teda namiesto doterajšej jednej hodnoty (maxima) budeme pamätať hodnoty dve (maximum a zmenu počtu cestujúcich v ňom).



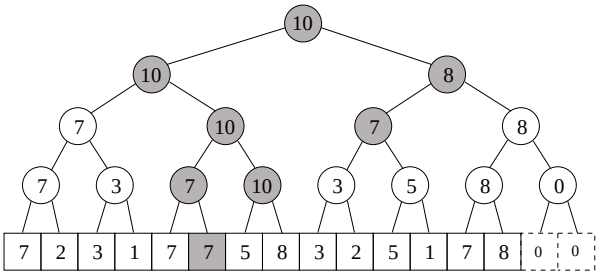
Pridanie 2 ľudí do úsekov 6 až 12.
Čísla predstavujú **počty cestujúcich**, sivé vrcholy sa menili.

Všimnite si, že pre každý list platí, že počet cestujúcich na zodpovedajúcom úseku dostaneme tak, že sčítame všetky počty cestujúcich na ceste z daného listu do koreňa stromu.

Upravili sme teda strom tak, že pamätané počty cestujúcich už sú správne. Zostáva upraviť pamätané maximá pre podstromy. Kde sa tie budú meniť? Nuž, vo vrcholoch, kde sa zmenili počty cestujúcich, a všade nad nimi:



Pridanie 2 ľudí do úsekov 6 až 12.
Čísla predstavujú **maximá**, sivé vrcholy treba prepočítať.



Pridanie 2 ľudí do úsekov 6 až 12.
Takto budú vyzerať maximá po prepočítaní.

Vrcholov, kde treba prepočítať maximum, je $O(\log N)$. Sú to totiž práve tie vrcholy, ktoré navštívime, keď delíme náš interval na jednoduché časti – a teda aj keď upravujeme počty cestujúcich. Vieme teda obe veci upraviť naraz v jednej jednoduchej rekurzívnej funkcii.

Celkovo vieme každú požiadavku spracovať v čase $O(\log N)$, preto celková časová zložitosť nášho riešenia je $O(P \log N)$.

Ešte jedna poznámka k implementácii: Intervalový strom si vieme pamätať v jednom statickom poli, podobne ako napríklad haldu. Koreň bude na políčku s indexom 1, synovia vrcholu x budú na políčkach $2x$ a $2x + 1$. Listy budú na políčkach N až $2N - 1$.

Listing programu:

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

#define MAXN 1000000

class vrchol { public: int cestujuci, maximum; };
vrchol strom[2*MAXN + 47];
int N,M,P;

int najdiMaximum(int x, int y, int kde, int left, int right, int uz) {
  // "kde" je cislo vrcholu v ktorom prave sme
  // "left" a "right" su konce jemu zodpovedajuceho jednoducheho intervalu
  // "uz" je pocet cestujucich, o ktorych uz vieme, ze su v intervale
  //   (dozvedeli sme sa o nich vyssie)

  if (x<=left && y>=right) return uz + strom[kde].maximum;
  if (y<left || x>right) return 0;
  int dlzka = (right-left+1)/2;
  uz += strom[kde].cestujuci;
  return max(
    najdiMaximum(x,y, 2*kde,   left,       left+dlzka-1, uz),
    najdiMaximum(x,y, 2*kde+1, left+dlzka, right,        uz) );
}

void pridaj(int x, int y, int kolko, int kde, int left, int right) {
  if (x<=left && y>=right) {
    strom[kde].maximum += kolko;
    strom[kde].cestujuci += kolko;
    return;
  }
  if (y<left || x>right) return;
  int dlzka = (right-left+1)/2;
  pridaj(x,y,kolko, 2*kde,   left,       left+dlzka-1);
  pridaj(x,y,kolko, 2*kde+1, left+dlzka, right       );
  strom[kde].maximum =
    strom[kde].cestujuci + max( strom[2*kde].maximum, strom[2*kde+1].maximum );
}

int main() {
  int _N;
  cin >> _N >> M >> P;
  N=1; while (N<_N) N *= 2;
  while (P--) {
    int k,x,y;
    cin >> k >> x >> y; x++;
    int m = najdiMaximum(x,y,1,1,N,0);
    if (k > M-m) {
      cout << "odmietnuta" << endl;
    } else {
      cout << "prijata" << endl;
      pridaj(x,y,k,1,1,N);
    }
  }
}
```

A-I-3 Fibonacciho sústava

Jednoznačnosť pekného zápisu

Nájsť pekný zápis prirodzeného čísla N vo Fibonacciho sústave vlastne znamená nájsť množinu Fibonacciho čísel, ktorých súčet je N, všetky sú navzájom rôzne a žiadne dve nenasledujú vo Fibonacciho postupnosti po sebe.

Na úvod si všimnime, že ak samotné N je Fibonacciho číslo, tak samo osebe tvorí hľadanú množinu. Tomu zodpovedá pekný zápis s práve jednou jednotkou: pre F_n (kde $n \geq 2$) je to „$1\underbrace{0\ldots0}_{n-2}$“.

Fibonacciho zápisy prirodzených čísel do 15 si môžeme ručne vypísať, aby sme odpozorovali nejakú zákonitosť. Dostaneme:

číslo	zápis
$F(2) = 1$	1
$F(3) = 2$	10
$F(4) = 3$	100
4	101
$F(5) = 5$	1000
6	1001
7	1010
$F(6) = 8$	10000
9	10001
10	10010
11	10100
12	10101
$F(7) = 13$	100000
14	100001
15	100010

Vidíme dve veci. V prvom rade sme si overili, že pre malé prirodzené čísla dokazované tvrdenie platí, v druhom rade začíname vidieť systém, akým pekný zápis funguje. Skúsime teraz dokázať, že to tak bude fungovať aj ďalej.

Matematickou indukciou dokážeme, že pre každé $n \geq 2$ platí tvrdenie $T(n)$: „Všetky prirodzené čísla z množiny $\{F_n, \ldots, F_{n+1} - 1\}$ majú práve jeden pekný zápis, a ten obsahuje práve $n - 1$ cifier. Navyše platí, že každé väčšie číslo už musí mať aspoň n-ciferný pekný zápis.“

Z tabuľky vidíme, že pre n do 6 toto tvrdenie platí. Zostáva teda dokázať indukčný krok. Nech pre nejaké n platia tvrdenia $T(2)$ až $T(n-1)$, dokážeme, že z toho vyplýva platnosť $T(n)$.

Zaujímajú nás teda zápisy čísel z množiny $M(n) = \{F_n, \ldots, F_{n+1} - 1\}$. Z tvrdenia $T(n-1)$ vieme, že ich zápis musí mať aspoň $n - 1$ cifier. No a viac cifier mať nemôže, najmenšie číslo s n-ciferným zápisom je predsa zjavne F_{n+1}. Pekný zápis každého z týchto čísel musí teda mať práve $n - 1$ cifier.

To znamená, že keď sa na pekný zápis dívame ako na množinu Fibonacciho čísel, táto množina musí obsahovať F_n. A keďže nemôžeme použiť dve po sebe idúce Fibonacciho čísla, táto množina nesmie obsahovať F_{n-1}. Inými slovami, pekný zápis čísla z $M(n)$ musí byť tvaru „$10\underbrace{?\ldots?}_{n-3}$“.

Zoberme si teraz nejaké číslo X z množiny $M(n)$. Ukážeme, že chýbajúce cifry pekného zápisu X sú určené jednoznačne. Prečo? Chýbajúce cifry zjavne musia tvoriť pekný zápis čísla $Y = X - F_n$. Platí $X < F_{n+1}$, preto $Y < F_{n+1} - F_n = F_{n-1}$. Z indukčného predpokladu teda vieme, že Y má práve jeden pekný zápis, a že ten má dostatočne málo cifier na to, aby sa zmestil na chýbajúce miesta. Preto má aj X práve jeden pekný zápis.

Posledný krok dôkazu: Najväčšie číslo s pekným zápisom dĺžky $n - 1$ má zjavne pekný zápis tvaru „$101010\ldots$“. Toto môžeme zapísať ako „$10Z$“, kde Z je maximálny pekný zápis dĺžky $n-3$. O tom už vieme z indukčného predpokladu, že zodpovedá číslu $F_{n-1} - 1$. Náš maximálny zápis teda zodpovedá číslu $F_n + F_{n-1} - 1 = F_{n+1} - 1$, q.e.d.

Nájdenie pekného zápisu

Náš dôkaz nám priamo dáva metódu na nájdenie pekného zápisu čísla X: Nájdeme najväčšie n také, že $F_n \leq X < F_{n+1}$. Toto Fibonacciho číslo sa v

zápise X bude vyskytovať. Zvyšok zápisu X je tvorený zápisom menšieho čísla $X - F_n$. Opakovaním postupu zostrojíme postupne celý zápis.

Listing programu:

```
var F : array[0..45] of longint;
    i,X,n : longint;
begin
   { spocitame Fibonacciho cisla }
   F[0] := 0;  F[1] := 1;
   for i:=2 to 100 do F[i] := F[i-1] + F[i-2];
   { nacitame vstup }
   read(X);
   { najdeme najvyssiu cifru }
   n:=2;
   while (F[n] <= X) do inc(n);
   dec(n);
   { postupne vypisujeme cifry }
   while (n >= 2) do begin
      if (X >= F[n]) then begin write(1); dec(X,F[n]); end else write(0);
      dec(n);
   end;
   writeln;
end.
```

Počítanie zaujímavých čísel

Označme $R(k, A, B)$ riešenie zadanej úlohy, teda počet čísel z množiny $\{A, A+1, \ldots, B\}$, ktoré majú vo svojom peknom zápise práve k jednotiek.

Začneme tým, že si zadanú úlohu zjednodušíme. Zadanú úlohu stačí vedieť riešiť pre prípad $A = 1$, lebo zjavne platí $R(k, A, B) = R(k, 1, B) - R(k, 1, A-1)$. Slovne: Aby sme spočítali vhodné čísla v zadanej množine, spočítame vhodné čísla neprevyšujúce B, a od nich odpočítame vhodné čísla menšie ako A.

Poďme teda ukázať, ako spočítať hodnotu $R(k, 1, N)$ pre dané N. Začneme tým, že si prevedieme N do Fibonacciho sústavy a zistíme jeho počet cifier c. Teraz vieme, že všetky hľadané pekné zápisy budú mať najviac c cifier. Môžeme si pre jednoduchosť predstaviť, že tie z nich, ktoré sú kratšie, doplníme zľava nulami na dĺžku presne c.

Na zadaný problém sa teda môžeme pozerať nasledovne: Máme postupnosti núl a jednotiek, ktoré majú dĺžku c. Potrebujeme spočítať, koľko z nich má všetky nasledujúce vlastnosti:

- Obsahuje práve k jednotiek.

- Je pekným zápisom, teda nemá dve jednotky po sebe.
- Vo Fibonacciho sústave predstavuje číslo neprevyšujúce N.

Len prvá podmienka

Spočítať postupnosti, ktoré spĺňajú prvú podmienku, je ľahké: Máme postupnosť dĺžky c, potrebujeme vybrať k miest, kde budú jednotky, toto sa dá spraviť $\binom{c}{k}$ spôsobmi.

Prvé dve podmienky

Tu to nebude omnoho zložitejšie. Zoberme ľubovoľnú postupnosť, ktorá spĺňa prvé dve podmienky. Tesne za každou z prvých $k-1$ jednotiek je v nej určite nula. Keď týchto $k-1$ núl vyhodíme, dostaneme novú postupnosť dĺžky $c-k+1$, v ktorej je k jednotiek. A naopak, z novej postupnosti vieme tú pôvodnú jednoznačne zrekonštruovať, stačí za každú jednotku okrem poslednej vložiť jednu nulu.

Tým sme dokázali, že postupností dĺžky c, ktoré spĺňajú prvé dve podmienky, je rovnako ako postupností dĺžky $c-k+1$, ktoré spĺňajú prvú podmienku – čiže $\binom{c-k+1}{k}$.

Jednoduché rekurzívne riešenie

Budeme rekurzívne zľava doprava generovať všetky možné postupnosti núl a jednotiek, ktoré spĺňajú prvé dve podmienky. Zároveň si budeme v každom okamihu pamätať, akému číslu zodpovedá práve vygenerovaná postupnosť, aby sme neprekročili N.

Toto riešenie je ľahko naprogramovateľné, ale má veľkú časovú zložitosť – počet krokov je aspoň tak veľký ako počet nájdených čísel. Presný odhad časovej zložitosti by bol náročný, uvedieme len náznak myšlienky: Každé číslo do N má v peknom zápise nanajvýš $\log_2 N$ jednotiek. Preto pre nejaké k ($1 \leq k \leq \log_2 N$) bude mať odpoveď veľkosť aspoň $N/\log_2 N$, a teda náš algoritmus má časovú zložitosť $\Omega(N/\log N)$.

Listing programu:

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

long long F[100]; // fibonacciho cisla

int generuj(int c, int k, int poz, int lim) {
  // c je dlzka postupnosti, k je pocet zostavajucich jednotiek
  // poz je pozicia ktoru doplname, lim je horny limit na velkost cisla
  if (lim<0) return 0;
  if (poz>=c) return k==0;
  // !!! sem pride optimalizacia !!!
  int result = generuj(c,k,poz+1,lim); // umiestnime 0
  if (k>0) result += generuj(c,k-1,poz+2,lim - F[c-poz+1]); // umiestnime 1
  return result;
}

int rataj(int k, int N) {
  int c=1; while (F[c+2]<=N) c++; // zistime pocet cifier c
  return generuj(c,k,0,N);
}

int main() {
  F[0]=0; F[1]=1; for (int i=2; i<100; i++) F[i]=F[i-1]+F[i-2];
  int k,A,B;
  cin >> k >> A >> B;
  cout << (rataj(k,B) - rataj(k,A-1)) << endl;
  return 0;
}
```

Optimalizácia rekurzívneho riešenia

Do programu pridáme dva riadky, ktoré ho priam zázračne urýchlia, napriek tomu, že oba budú veľmi jednoduché.

Prvé pozorovanie: Ak nám zostáva do konca použiť k jednotiek a máme už iba menej ako $2k-1$ pozícií, riešenie neexistuje a môžeme sa vrátiť o pozíciu späť.

Druhé pozorovanie: Ak máme doplniť posledných x miest, najväčšie číslo, ktoré vieme vyrobiť, je $F_{x+2}-1$. Pokiaľ vieme, že ešte ani týmto číslom neprekročíme hornú hranicu, nemusíme všetky možné čísla generovať, ale vieme ich rovno zarátať. Ako sme ukázali vyššie, je ich $\binom{x-k+1}{k}$.

V našej rekurzívnej funkcii teda pridáme nasledovné podmienky:

```
  if (c-poz < 2*k-1) return 0;
  if (lim >= F[c-poz+2]-1) return C[c-poz-k+1][k];
```

Kombinačné čísla si môžeme napríklad na začiatku jednoducho predrátať využitím známeho vzťahu $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$:

```
long long C[100][100]; // kombinacne cisla
for (int i=0; i<100; i++) for (int j=0; j<=i; j++)
  C[i][j] = (j==0||j==i) ? 1 : C[i-1][j-1]+C[i-1][j];
```

Prečo je takto vylepšené riešenie odrazu také efektívne? Preto, že situácia, kedy by sme nevedeli použiť žiadnu z podmienok, skoro nikdy nenastane.

Všimnime si jednu dôležitú vlastnosť pekného zápisu: Keď máme dva pekné zápisy rovnakej dĺžky, ten, ktorý predstavuje menšie číslo, je aj lexikograficky menší. (Hravo sa to dá dokázať indukciou.)

Všimnime si teraz ľubovoľnú situáciu počas behu nášho vylepšeného algoritmu. V tomto okamihu máme vygenerovanú nejakú postupnosť núl a jednotiek dĺžky najviac c, a ideme spočítať, koľkými spôsobmi sa dá doplniť. Porovnajme si doteraz vygenerovanú postupnosť s rovnako dlhým prefixom pekného zápisu N.

Ak je naša postupnosť väčšia, znamená to, že už sme N prekročili, a naša funkcia teda okamžite vráti nulu. Ak je naša postupnosť menšia, vieme, že nech už doplníme čokoľvek, N neprekročíme, a preto môžeme rovno vrátiť počet všetkých doplnení. Jediný prípad, kedy musíme spraviť dve rekurzívne volania (a teda generovať postupnosť ďalej) je ten, keď nastala rovnosť – čiže keď je naša postupnosť prefixom pekného zápisu N.

Ukážeme si to na príklade: Nech má N pekný zápis 100101001000 a nech $k = 5$. Keď sme v situácii 101..., aj keby sme už doplnili samé nuly, bude výsledok väčší ako N, preto takéto riešenie neexistuje. Keď sme v situácii 100100..., môžeme na zvyšných 6 miest doplniť ľubovoľný pekný zápis s troma jednotkami. V tomto prípade máme teda 4 riešenia.

Rekurzívne volania bude teda náš algoritmus robiť len raz pre každú dĺžku prefixu, teda rádovo c-krát. Preto je časová zložitosť samotnej rekurzie $O(c)$, alebo ekvivalentne $O(\log N)$. Celý algoritmus je o niečo pomalší kvôli predrátaniu kombinačných čísel. Dalo by sa s tým ešte niečo robiť, ale neoplatí sa. Ani pre N rovné miliarde by sme už žiadne viditeľné zrýchlenie nespozorovali.

Matematické riešenie

Na všetko je vzorec, a ani tento prípad nie je výnimkou. Naše riešenie bude vyzerať nasledovne: Prevedieme N na pekný zápis. Nájdeme najbližšie menšie alebo rovné číslo N', ktoré má v peknom zápise práve k jednotiek. (Rozmyslite si ako na to, nie je to také triviálne ako sa zdá na prvý pohľad.) Priamo z toho, ako tento zápis vyzerá, spočítame, koľký v poradí zaujímavý zápis to je.

Využijeme dve skutočnosti. Prvou bude pozorovanie z predchádzajúcej časti: Keď máme dva pekné zápisy rovnakej dĺžky, ten, ktorý predstavuje menšie číslo, je aj lexikograficky menší.

Druhou bude trik, ktorý sme použili, keď sme rátali pekné postupnosti: Keď zoberieme pekné postupnosti dĺžky c s k jednotkami a v každej za každou jednotkou okrem poslednej škrtneme nulu, dostaneme práve všetky postupnosti dĺžky $c - k + 1$ s k jednotkami.

Všimnime si teraz, že keď sme zobrali dve postupnosti a z oboch takto vyškrtali nuly, tak tá, ktorá bola menšia pred škrtaním, musela zostať menšia aj po ňom.

Takže pokojne môžeme zobrať zápis N', týmto spôsobom ho upraviť, a následne zodpovedať jednoduchšiu otázku: Koľká v poradí postupností s k jednotkami je toto?

No a toto vieme ľahko spočítať. V každom okamihu stačí rozlíšiť medzi dvoma prípadmi. Ak začína nulou, stačí túto nulu zahodiť. Ak začína jednotkou, sú pred ňou všetky postupnosti, ktoré začínajú nulou. No a tých je $\binom{d-1}{k}$, kde d je aktuálna dĺžka. Všetky tieto zarátame, jednotku zo začiatku zahodíme a zmenšíme k o jedna.

Skúste si uvedomiť, že toto riešenie, ktoré sme dostali, je takmer identické s riešením, ku ktorému sme sa z úplne opačného prístupu dopracovali v predchádzajúcej časti.

Listing programu:

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

long long F[100]; // fibonacciho cisla
long long C[100][100]; // kombinacne cisla
int jednotky[100];
int N;

long long rataj(int k, int N) {
  if (N==0) return 0;
  // spocitame zapis N
  int co=2; while (F[co+1]<=N) co++;
  int J=0; while (co>=2) { if (N>=F[co]) { jednotky[J++]=co-2; N-=F[co]; } co--;
}

  // zostrojime zapis N'
  if (J<k) {
    if (jednotky[0]<=2*k-2) return 0;
    int skus;
    for (skus=J-1; skus>=0; skus--) {
      int ostava=(jednotky[skus]-1)/2, treba=k-(skus+1);
      if (ostava >= treba) break;
    }
    jednotky[skus]--;
    for (int i=skus+1; i<k; i++) jednotky[i] = jednotky[i-1]-2;
  }
  J=k;

  // odstranime zo zapisu N' nuly a zostrojime si ho
  for (int i=0; i<J; i++) jednotky[i] -= J-1-i;
  int zapis[100];
  memset(zapis,0,sizeof(zapis));
  for (int i=0; i<J; i++) zapis[jednotky[i]]=1;

  // spocitame vysledok
  long long res = 1;
  for (int i=jednotky[0]; i>=0; i--) if (zapis[i]) { res += C[i][k]; k--; }
  return res;
}

int main() {
  F[0]=0; F[1]=1; for (int i=2; i<100; i++) F[i]=F[i-1]+F[i-2];
  for (int i=0; i<100; i++) for (int j=0; j<=i; j++)
    C[i][j] = (j==0||j==i) ? 1 : C[i-1][j-1]+C[i-1][j];
  long long k,A,B;
  cin >> k >> A >> B;
  cout << (rataj(k,B) - rataj(k,A-1)) << endl;
  return 0;
}
```

A-I-4 Prekladacie stroje

Podúlohy a+b

V prvej časti tohto vzorového riešenia ukážeme, že stroje B a C nerobia navzájom úplne presne inverzné operácie. Problém bude v tom, že dekódovanie morzeovky bez oddeľovačov nemusí byť jednoznačné – niektoré reťazce čiariek a bodiek sa dajú preložiť viacerými spôsobmi, a iné naopak vôbec.

Zoberme si napríklad jednoslovnú množinu $M_1 = \{i\}$. Keď túto preložíme strojom B do morzeovky, dostaneme $B(M_1) = \{\bullet\bullet\}$. Reťazec $\bullet\bullet$ je ale aj morzeovkovým zápisom reťazca ee. Preto keď $B(M_1)$ preložíme späť strojom C, dostávame $C(B(M_1)) = \{i, ee\} \neq M_1$, a teda prvé tvrdenie neplatí.

Podobne ľahko zistíme, že pre $M_2 = \{\blacksquare\}$ je $B(C(M_2)) = \emptyset \neq M_2$, a teda ani druhé tvrdenie neplatí.

Podúloha c

Prekladať budeme len niektoré reťazce, a to reťazce, v ktorých sú najskôr všetky a, a potom všetky b. Každú dvojicu a prepíšeme na jedno a, a každé b na tri b.

Formálne, náš prekladací stroj A bude pätica $(K, \Sigma, P, 0, F)$, kde $K = \{0, 1\}$, $F = \{1\}$ a prekladové pravidlá vyzerajú nasledovne:

$$P \;=\; \Big\{\ (0, aa, a, 0),\ (0, \varepsilon, \varepsilon, 1),\ (1, b, bbb, 1)\ \Big\}$$

Podúloha d

Najlepšie riešenie potrebuje tri operácie. Jedno takéto riešenie si ukážeme.

Čo všetko vieme dosiahnuť jedným prekladom? V prvom rade vieme z množiny M_1 vybrať len pre nás zaujímavé reťazce, teda tie, kde idú najskôr a a potom b. Keď sa dočítame na koniec slova, vieme ešte napísať aj nejaké c, už však nijak nevieme zabezpečiť, aby ich počet bol rovný počtu a a b.

Formálne, definujme prekladací stroj $A_1(K_1, \Sigma, P_1, A, F_1)$, kde $K_1 = \{A, B, C\}$, $F_1 = \{C\}$ a prekladové pravidlá vyzerajú nasledovne:

$$P_1 \quad = \quad \Big\{ \ (A, a, a, A),\ (A, \varepsilon, \varepsilon, B),\ (B, b, b, B),\ (B, \varepsilon, \varepsilon, C),\ (C, \varepsilon, c, C) \ \Big\}$$

Zjavne až na počet písmen c je $M_2 = A_1(M_1)$ presne to, čo hľadáme. Ako ale zabezpečiť, aby sa počty a, b aj c museli rovnať?

Trik je v tom, že rovnako, ako sme si práve vyrobili množinu reťazcov s rovnakým počtom a a b, vieme vyrobiť aj množinu reťazcov, ktoré budú mať rovnako b a c.

Formálne, definujme prekladací stroj $A_2(K_2, \Sigma, P_2, A, F_2)$, kde $K_2 = \{A, B, C\}$, $F_2 = \{C\}$ a prekladové pravidlá vyzerajú nasledovne:

$$P_2 \quad = \quad \Big\{ \ (A, \varepsilon, a, A),\ (A, \varepsilon, \varepsilon, B),\ (B, a, b, B),\ (B, \varepsilon, \varepsilon, C),\ (C, b, c, C) \ \Big\}$$

Aj tento prekladací stroj vyrobí z M_1 takmer presne to, čo treba: $M_3 = A_2(M_1)$ má nasledovné vlastnosti: Písmená v reťazcoch idú v správnom poradí a písmen b a c je rovnako veľa.

No a posledný krok je jednoduchý, $G = M_2 \cap M_3$. Slovne: V prieniku množín sú práve tie reťazce, ktoré majú obe dobré vlastnosti – počet a je rovný počtu b (lebo je to reťazec z M_2) a počet b je rovný počtu c (lebo je to zároveň reťazec z M_3).

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

DVADSIATY TRETÍ ROČNÍK OLYMPIÁDY V INFORMATIKE

Vydala IUVENTA s finančnou podporou Ministerstva školstva SR
Náklad: 400 výtlačkov
Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX