Použitie hrubej sily

Obsah:

- cieľ tejto časti
- n!
- 2^n
- orezávania

Úlohy čo tu chcem využit:
Máme číslo n. Nájdite najmenšie číslo menšie ako n, ktoré má všetky cifry rôzne.
Máme n čísiel kladných aj záporných. Koľko je takých poradí preskúmavania, aby hodnota nikdy neklesla pod 0.
Koniec úloh
Ako sa vraví: Keď to nejde silou, používate jej primálo.

Častokrát sa nám stane, že nevieme vymyslieť rýchle riešenie, jediné čo nás napadne je hrubé priamočiare skúšanie všetkých možností. Napríklad ak hľadáme všetky čísla menšie ako 1000, ktoré sú deliteľné 5 a ich druhá mocnina je deliteľná 4. Jedna možnosť je zamyslieť sa, ako také čísla vyzerajú a vypísať tie správne. Druhá možnosť je naozaj vyskúšať všetkých 1000 čísiel či nevyhovujú zadaniu.

Takéto riešenie je pre nás ľudí nevyhovujúce. Umocňovať 1000 čísiel na druhú a ešte kontrolovať či sú deliteľné 4? V žiadnom prípade. Ale počítač je na tom inak. Jeho hlavná výhoda je výpočtová sila. Počítač nemá problém umocniť číslo a zistiť deliteľnosť v priebehu milisekúnd. A vyskúšať to 1000 krát ... brnkačka.

V tejto kapitole sa teda porozprávame, ako dostať silu počítaču pod našu kontrolu, naučíme sa kedy a ako používať hrubú silu a zopár trikov, aby to ani tomu počítaču netrvalo tak dlho. A ak sa nudíte, môžete sa rovno vrhnúť na úlohu **magic**.

Permutácie

Prvou možnosťou na generovanie všetkých možností sú permutácie. Zoberme si napríklad úlohu **pamiatky**. V tejto úlohe máme n pamiatok a na svojej výhliadkovej túre, si ich chceme všetky postupne pozrieť. Každá pamiatka nám však pridá alebo uberie radosť z výletu. Samozrejme nechceme, aby naša radosť bola pod 0. Otázka je, koľko je takých postupností návštevy pamiatok, že naša radosť neklesne pod 0.

Vieme, že existuje najviac $n!$ rôznych pozretí pamiatok, keďže každá ich permutácia určuje jednu postupnosť, ktorou sa môžeme vybrať. Takisto vieme, že v najväčšom možnom vstupe, je $n = 9$, takže rôznych permutácií je $9! = 362\,880$. A to vôbec nie je tak veľa.

No a úplne najlepšia správa je, že ak vám dám konkrétnu permutáciu pamiatok, viete veľmi jednoducho zistiť, či táto permutácia poruší podmienku zo zadania a teda radosť z jej prejdenia klesne pod 0. Na to stačí simulovať prechádzku po pamiatkach a stále si pamätať, akú máme momentálne radosť. Ak niekedy klesne pod 0, daná postupnosť je zlá, ak sa to nikdy nestane, môžeme si pripočítať 1 k výsledku.

Ten najdôležitejší krok však zostáva. Ako si vygenerovať postupne všetky permutácie čísiel od 1 po n. Poprípade to ani nemusia byť prvky od 1 po n, môžu to byť aj čísla oveľa väčšie alebo opakujúce sa.

Prvou možnosťou je spraviť si vlastnú rekurzívnu funkciu. Ako parametre dostane táto funkcia dve polia: prvé obsahuje prvky, ktoré ešte nepoužila a treba ich zaradiť do vytváranej permutácie, druhá obsahuje prvky, ktoré vytvárajú aktuálnu permutáciu. To čo treba spraviť ako ďalší krok, je vybrať prvok, ktorý bude ležať na ďalšom mieste v našej permutácii. A vhodný kandidáti sú všetky ešte nezaradené prvky. No a keďže chceme vytvoriť permutácie všetky, postupne skúšame dosadiť všetky nezaradené prvky na ďalšie voľné miesto. Keď si zvolíme prvok x, vyberieme ho z nezaradených prvkov, pridáme ho do permutácie a znova sa rekurzívne vnoríme.

Keď spotrebujeme všetky prvky ostane nám výsledná permutácia. A tento prístup ich postupne vygeneruje všetky. Tu je k tomu názorný program:

Listing programu (C++)

```
//rekurzivna funkcia tvoriaca permutacie
void permutacie(vector<int> nezaradene, vector<int> permutacia) {
    if(nezaradene.size()==0) {
        //už som zaradil všetky prvky, spravím niečo s výslednou permutáciou
        return ;
    }
    //postupne dám na ďalšie miesto každý prvok
    for(int i=0; i<nezaradene.size(); i++) {
        //v nove_nezaradene nie je prvok nezaradene[i]
        vector<int> nove_nezaradene;
        for(int j=0; j<i; j++) nove_nezaradene.push_back(nezaradene[j]);
        for(int j=i+1; j<nezaradene.size(); j++) nove_nezaradene.push_back(nezaradene[j]);
        //v nova_permutacia pribudne na koniec prvok nezaradene[i]
        vector<int> nova_permutacia = permutacia;
        nova_permutacia.push_back(nezaradene[i]);
        //rekurzivne sa zavolám
        permutacie(nove_nezaradene, nova_permutacia);
    }
}
```

Nevýhodou na tomto riešení je, že si posúva dve polia ako parametre funkcie. To ho dosť spomaľuje. Samozrejme, dá sa toho zbaviť, ale napriek tomu, potrebujeme napísať sami zbytočne veľa kódu, v ktorom je veľká šanca, že sa pomýlime. Niekto však ten kód už napísal za nás, tak prečo ho nepoužiť.

V STL existuje funkcia `next_permutation()`, ktorá ako napovedá názov vytvorí z poľa, ktoré jej dáme ako parameter ďalšiu jeho permutáciu. A toto má množstvo výhod. Nemusíme skoro nič programovať, všetko dostaneme zadarmo. Túto funkcia môžeme používať opakovane a prejsť tak postupne cez všetky možné permutácie. A taktiež vždy nájde permutáciu, ktorá je lexikograficky väčšia. To znamená, že ak pole obsahuje niekoľko rovnakých prvkov, táto funkcia vie preskočiť všetky také, ktoré sa líšia len výmenou niektorých dvoch rovnakých. Ak teda máme n prvkov a p z nich sú rovnaké a ostatné sú rôzne, tak namiesto $n!$ možností vygeneruje len $\frac{n!}{p!}$.

Jediné na čo si treba dať pozor je, že na začiatku chcete dať do tejto funkcie prvky usporiadané od najmenšieho po najväčší. Toto je totiž lexikograficky najmenšia permutácia a ako som spomínal `next_permutation()` vytvára vždy väčšiu. Tu si môžete pozrieť najčastejšie použitie tejto funkcie:

Listing programu (C++)

```
//potrebný include
#include <algorithm>

//prvky, ktoré chcem permutovať
vector<int> permutacia;
//musím si prvky zoradiť
sort(permutacia.begin(),permutacia.end());
do{
    //v poli permutacia je ďalšia permutácia, tak ju spracujeme
}while(next_permutation(permutacia.begin(),permutacia.end()))
```

Bude veľmi dobré, ak sa tento kúsok kódu naučíte, zíde sa vám, keď budete potrebovať generovať všetky permutácie.

Množiny

Množiny sú ďalší matematický objekt, ktorý nám vie pomôcť. Zoberme si úlohu **zlodej**. V tejto úlohe ste zlodej, ktorý sa snaží z domu ukradnúť predmety s čo najväčšou spoločnou váhou, má však batoh, ktorý unesie najviac hmotnosť w. Chcete teda vybrať takú sadu vecí, že ich spoločná váha je čo najbližšie k w.

Táto úloha je od predchádzajúcej trochu odlišná. Je pravda, že by sme sa stále mohli pýtať na všetky permutácie prvkov, pridávať ich v tom poradí do batohu a keď batoh naplníme, prestaneme a zrátame, koľko váži dokopy veci v batohu.

To však ani zďaleka nie je to, čo by sme chceli robiť, dokonca to od nás nevyžaduje ani úloha. Nás totiž vôbec nezaujíma, v akom poradí vložíme veci do batoha. Nás zaujíma, ktoré veci budú na konci v batohu. Predstavte si, že viete vložiť k vecí do batohu. Existuje $k!$ možností ako ich tam vložiť, ale stále dostaneme to isté – tieto predmety sa do batohu zmestia a zaberú stále rovnako miesta. Bolo by fajn teda vyskúšať túto množinu len raz.

A tu prichádzame na koreň problému – množiny. Chceli by sme vedieť rýchlo prezerať všetky množiny vecí (množina je reprezentovaná tým, či danú vec zoberiem, alebo nie) a potom overiť, či daná množina vyhovuje.

Výhodou je, že všetkých množín z n prvkov je 2^n, čo je stále veľké číslo, ale oproti $n!$ nám to umožňuje spracovať všetky množiny až z 25 prvkov. Opäť existuje niekoľko prístupov na riešenie tohoto problému.

Prvým je znova rekurzia. V tomto prípade opäť budeme mať množinu vecí, ktoré ešte chceme zadeliť a množinu vecí, ktoré už patria do našej množiny. Na začiatku volania zoberieme prvú vec, ktorá je voľná a rozhodneme sa, či ju zobrať alebo nie. Samozrejme, keďže sa snažíme nájsť všetky možnosti, treba skúsiť obe rozhodnutia. Keď prvok vyberieme, zaradíme ho do množiny vecí čo už máme, vyhodíme ju z nerozhodnutých a rekurzívne sa zavoláme. Ak si ju neberieme, aj tak ju vyhodíme z nerozhodnutých vecí, lebo sme sa už rozhodli, že ju neberieme a zavoláme sa rekurzívne. Dostaneme takýto jednoduchý kúsok kódu:

Listing programu (C++)

```
//rekurzívna funkcia na generovanie množín
void mnoziny(vector<int> na_spracovanie, vector<int> mnozina) {
    if(na_spracovanie.size()) {
        //už som spracoval všetky prvky, v mnozina je ďašia množina, spracujem ju
        return ;
    }
    int x=na_spracovanie[0];
    //nove_na_spracovanie obsahuje všetky okrem prvého prvku, ktorý akurát spracujeme
    vector<int> nove_na_spracovanie;
    for(int i=1; i<na_spracovanie.size(); i++)
        nove_na_spracovanie.push_back(na_spracovanie[i]);
    //prvý prvok nezoberiem
    mnoziny(nove_na_spracovanie, mnozina);
    //vyberiem prvý prvok
    mnozina.push_back(x);
    mnoziny(nove_na_spracovanie, mnozina);
}
```

Opäť to však nie je najľahší prístup. Existuje aj prístup oveľa jednoduchší, samozrejme, ale viac trikovejší. Majme našich n vecí a dajme im nejaké poradie, napríklad podľa toho, kedy sa vyskytli na vstupe, nultý, prvý až $n-1$-vý. Každú množinu si teraz viem predstaviť ako reťazec dlhý n znakov, tvorených zo znakov 0 a 1. 1 znamená, že prvok sa v tejto množine nachádza, 0 je opak. Dobre, ale toto je vlastne spôsob akým sa zapisujú binárne čísla. Stačí i-temu prvku v poradí priradiť hodnotu 2^i a z nášeho reťazca sa razom stane celé číslo. A presne toto bude princíp nášho riešenie.

Môžeme si uvedomiť, že čísla od 0 po 2^n-1 nám postupne v dvojkovej sústave kóduju všetky možné množiny n prvkov, začínajúc prázdnou a končiac takou, čo obsahuje všetky prvky. A s číslami sa v C++ pracuje veľmi pohodlne a keďže sú interne reprezentované ako 0 a 1, tak s nimi vieme robiť veľa peknych operácií. Tu je program, ktorý ukazuje, ako sa dá postupne prejsť cez všetky množiny a tiež zistiť, ktoré čísla do nich patria:

Listing programu (C++)

```
//prvky, z ktorých idem robiť množiny
vector<int> prvky;
for(int i=0; i<(1<<prvky.size()); i++) {
    //spracujem ďalšiu množinu reprezentovanú číslom i
    for(int j=0; j<prvky.size(); j++) {
        if(i&(1<<j)) {
            //množina i obsahuje j-ty prvok
        }
        else {
            //množina i neobsahuje j-ty prvok
        }
    }
}
```

Niektoré operácie ste možno ešte nestretli. $1 << n$ je takzvaný bitový posun doľava. V princípe to znamená, že k číslu naľavo (1) v bitovom zápise pridá číslo napravo (n) núl. Teda z tohoto vznikne číslo 2^n. Ďalšia neznáma je $i\&(1 << j)$. Už vieme, že sa vlastne pýtame na to, že $i\&2^j$. Čo to ale je? Znak & predstavuje bitový and, ktorý zoberie bitový zápis čísla i, bitový zápis čísla 2^j a bit po bite vykoná and. Ako vieme $1 and 1 = 1$ a $0 and hocio = 0$. Číslo 2^j obsahuje jedinú jednotku na j-tom mieste, všade inde sú 0. To znamená, že tento výraz bude mať hodnotu 0, ak sa na j-tom mieste v množine i nachádza 0 a hodnotu 2^j, ak sa tam nachádza 1. A keďže v C++, 0 je $false$ a všetko ostatné je $true$, podmienka bude splnená práve vtedy, keď množina i obsahuje j-ty prvok. Šikovné nie?