АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА
Lightbulbs

По-опитните състезатели трябваше да се сетят, че задачата Lightbulbs е NP-пълна, съответно решението трябваше да е или някаква форма на meet-in-the-middle, или динамично, или bruteforce.

Тъй като meet-in-the-middle и динамично оптимиране са относително любими мои теми (давал съм не една или две такива задачи по състезания), някои от състезателите сигурно биха се насочили натам. Което, този път, би било грешка (или поне аз не знам за решение, базирано на някоя от тези две стратегии).

По-неопитните състезатели сигурно биха тръгнали към пълното изчерпване, което е и очакваното решение тук. За съжаление, с рандом тестове (каквито, що-годе са тези във feedback-а - първите 25) много от решенията изглеждат достатъчно бързи или верни, но всъщност не са.

Тук състезателите не само трябваше да напишат решение (което, поне, не е толкова трудно) а и да измислят тестове, на които то се държи възможно най-зле. Да се надяваме, че и аз съм измислил такива…

Една основна (относително очевидна) оптимизация, която може да се ползва във всички решения по-долу, е да забележим, че лампите са до 50 – тоест можем да пазим както включените, така и изгорелите такива в две 64-битови числа. Можем да ъпдейтваме кои са включени и кои са изгорели с битови операции, което забързва работата на програмата ни с около 50 пъти.

Ето няколко от решенията:

Пълно изчерпване, базирано на битова маска
Пробваме всички под-множества за O(2^N * N) – би хванало няколко теста и около 20 точки след групирането.

Пълно изчерпване, чрез бектрек
Същото като горното, само че реално работи малко по-бързо. Макар и да хваща малко повече тестове, след групирането отново остава с 20 точки.

Пълно изчерпване с първо грийди
Можем да направим наблюдението, че ако текущо се чудим дали да ползваме даден ключ, и най-важната лампа, която той контролира, е вече включена, то така единствено влошаваме нещата (тъй като ще я изключим, и после няма да има как да я включим, тъй като ще е изгоряла). Ако махнем този бранч от бектрека, вече хващаме значително повече тестове, и около 30-40 точки след групирането.

Пълно изчерпване с първо и второ грийди
Второто грийди наблюдение, което можем да направим, е че ако сортираме ключовете по най-старши бит, то от всички ключове с еднакъв най-старши бит ще ползваме най-много един (тъй като ползвайки два или повече ще изгорим крушката на този бит). Така решението ни вече хваща около 60 точки след групирането на тестовете.

Оптимизация на грийди решенията
Една оптимизация, която можем да направим, е да пропускаме битове, в които никой от ключовете, които все още не сме разгледали, няма най-старши бит. Реално това би било бит, в който не правим нищо, освен да продължим нататък. Така решението ни се забързва няколко пъти и води до още 10-20 точки отгоре на предните две решения.

Втора оптимизация на грийди решенията
Оказва се, че можем да правим *всички* неща с побитови операции – както да намерим кой е следващия бит, в който има ключ, който потенциално бихме ползвали, така и кои ключове има смисъл да ползваме на кои битове. За целта трябва много добре да знаем какво прави всяка побитова операция и как да ги изпълняваме правилно.

Сядайки да имплементираме тази оптимизация, стигаме до момента, в който виждаме, че ни трябва функция, която намира индекса на най-големия бит на дадено (до 50-битово) число; както и такава, която намира индекса на най-малкия бит на дадено (до 50-битово число). В GCC има вградени такива (вижте `__builtin_ffsll(num)` и `__builtin_clzll(num)`), но лесно можем да си напишем и ние такава (даже се оказва, че работи по-бързо!). За целта ще преизчислим позициите на най-малките и най-големите битове на всички числа до 2^13 (което са достатъчно малки масиви, за да се съберат в кеша на процесора). Имайки тези масиви (които ще наричаме lowest[] и highest[]) можем да ползваме следните две функции:

```
inline int lowestBit(unsigned long long num) {
    if ((num >> 0) & (MASK - 1)) return lowest[(num >> 0) & (MASK - 1)] + 0;
    if ((num >> 13) & (MASK - 1)) return lowest[(num >> 13) & (MASK - 1)] + 13;
    if ((num >> 26) & (MASK - 1)) return lowest[(num >> 26) & (MASK - 1)] + 26;
    return lowest[(num >> 39) & (MASK - 1)] + 39;
}

inline int highestBit(unsigned long long num) {
    if (num >> 39) return highest[num >> 39] + 39;
    if (num >> 26) return highest[num >> 26] + 26;
    if (num >> 13) return highest[num >> 13] + 13;
    return highest[num];
}
```

Забележете, че можете да направите масивите да са с размер 2^17 и да спестите по един if() във всяка от функциите. Така, обаче, рискувате кешът на процесора да не ви стигне, и реално да се влоши бързодействието.

Прилагайки всички от горните оптимизации, решението става достатъчно бързо за $N \leq 50$ и $L \leq 50$ и хваща 100 точки.

Интересен въпрос е каква точно е сложността на решението. Това е много сложно да се каже, но едно наблюдение, което можем да направим, е че имаме бранчове само когато бит-а, който разглеждаме не е умрял и в момента е 0. Тогава разглеждаме всички ключове, който го контролира с най-старшия си жив бит. Представете си вход от типа на:

```
1000XXXXXXXX
1000XXXXXXXX
1000XXXXXXXX
0100XXXXXXXX
0100XXXXXXXX
0100XXXXXXXX
0010XXXXXXXX
0010XXXXXXXX
0010XXXXXXXX
0001XXXXXXXX
0001XXXXXXXX
```

Тук с X сме отбелязали битове, които могат да са както нула, така и единица. За най-старшия бит имаме три варианта (като трябва да пробваме всеки от тях). За втория и третия бит също имаме по три варианта. За четвъртия бит имаме два варианта. Така до тук имаме 3 * 3 * 3 * 2 = 54 варианта. Ако имаме до 50 позиции (ключа), най-голямото нещо, което можем да постигнем, е 3^16 * 2 = 86,093,442 варианта, което не е чак толкова много (3 * 16 + 2 = 50).

Защо, обаче, TL-тът е толкова голям, ако трябва да направим едва под 100 милиона операции? Едно нещо, за което и аз не се бях сетил като почвах да правя задачата е, че в умрели битове все пак е възможно да ползваме някои от ключовете. Все пак, за да не получим комбинаторна експлозия, вместо това ще преместим тези ключове в най-старшия им бит, който е нула и не е мъртъв – така ще ги групираме с другите ключове, които просто имат този немъртъв нулев бит като най-старши и ще ползваме второто грийди наблюдение. Това "местене" обаче трябва също да се случи с побитови операции, и в зависимост колко често местим ключовете, добавя допълнителна тежест.

Тъй като нямаше как точно да сметна сложността на авторското решение, но от друга страна то беше относително просто, вместо това написах динамично, което да генерира възможно най-гадния тест за него. Така гарантирам, че (за дадените ограничения по време) няма тест, за който авторското решение да не върви достатъчно бързо.

Автор: Александър Георгиев