АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА КУЛИ

С цел избягване на дългите описания в изложението по-нататък, нека въведем следното определение:

Нека $p=(p_1, p_2,\ldots, p_N)$ е пермутация на числата *1, 2, 3, ..., N*. Масивът $L=(L_1, L_{2,\ldots,}L_N)$, където L_i е броя на покриваните от p_i елементи, ще наричаме *L-масив* на пермутацията *p* и ще означаваме с *L(p).*

Формално можем да дефинираме задачата така:

Да се генерира пермутация *p* на числата от 1 до *N*, така че *L(p)* да съвпада с въведения масив.

Решение с пълно изчерпване

Това е най-простото решение, което може да ни хрумне. Генерираме последователно всички пермутации на числата от 1 до *N*, за всяка пресмятаме нейния *L* масив и го сравняваме с въведения. Ако двата масива съвпаднат, то извеждаме съответната пермутация.

Това е решение от порядъка на *O(N!*f(N))*, където *f(N)* е сложността на пресмятането на *L-масива* на поредната пермутация. Това пресмятане можем да извършим по два начина:

- Обхождаме пермутацията отляво надясно и за всяко число търсим първото, надясно от него, което е по-голямо (то го покрива). В един масив натрупваме за всяко число колко елемента покрива и, когато обходим целия масив, ще сме получили *L-масива* на пермутацията. В този случай $f(N)=O(N^2)$,а цялото решение е със сложност $O(N!*N^2)$.
- Вторият начин е значително по-хитър – изчисляването на *f(N)* се извършва линейно по *N*. Това става като се използва стек, в който стоят елементите от масива, които чакат да бъдат „покрити”. Движим се отляво надясно по масива, като всеки елемент попада в този стек и след като бъде покрит се изважда от него, а към бройката покривани елементи на този, който го покрива, се добавя 1. Цялото решение е със сложност *O(N!*N).*

За съжаление *N!* расте толкова бързо, че чрез системата е невъзможно да се определи кой от двата начина е използван. Въпреки това си заслужава да запомните втория начин – той е класика за използването на стек в случаи, когато елементи на списък трябва да изчакат да се появи следващ ги елемент, който се намира в някакво стноше с тях. Както ще видим по-надолу този подход ще доведе до бързи решения.

Такива решения са реализирани във файлове **towersn!n2.cpp** и **towersn!n.cpp** от Руско Шиков**,** решават тестовете, в които $N\leq 10$ и ще получат 10 точки.

Решение с изчерпване с връщане назад (backtracking)

Друго изчерпващо решение може да бъде направено, използвайки техниката backtracking (изчерпване с връщане назад), чиято същност е рекурсивното построяване на търсената пермутация с подходящо „рязане“ на частичните решения, за които става ясно, че няма да доведат до правилно пълно решение. Задачата позволява доста добро „рязане“ и, въпреки, че сложността теоретично е *O(N!)*, това решение ще мине за тестовете, в които $N\leq 15$ и ще получи 30 точки. То е реализирано от Антон Шиков във файл **towersbtrck.cpp**.

Решение със сложност $O(N^2)$

Идеята на това решение е да добавяме елементите на пермутацията един по един отляво надясно, като в един масив поддържаме пермутация на числата от 1 до m, удовлетворяваща условието за покриване, където m е броят на числата в масива към текущия момент.

Когато добавяме нов елемент, знаем колко числа трябва да покрие той (тази бройка може да бъде и 0). Стойността на числото, което добавяме, смятаме линейно като тръгнем от края към началото на текущия масив. Лесно се вижда дали поредното число, което разглеждаме е покрито или не (трябва да няма нищо по-голямо от него до края масива, за да не е покрито). Освен това непокритите числа вървят в нарастващ ред при движението от края към началото. Като отброим толкова непокрити числа, колкото показва съответната бройка в *L-масива*, изчисляваме стойността на новия елемент, който добавяме, по формулата 1 + последното число, което ще покрие този елемент. Ако няма да покрие нито едно число (т.е. съответната стойност в *L-масива* е 0), то новият елемент получава стойност 1. Това може да доведе до повторения в нашия масив. Оправяме повторенията като увеличим с 1 всички числа по-големи или равни на новодобавената стойност. Както лесно може да се съобрази, това не променя отношението на покриване между елементите на масива.

Сложността на този алгоритъм е $O(N^2)$, той е реализиран във файл **towersn2.cpp** от Йордан Чапъров, ще мине на тестовете, в които $N \leq 6000$ и ще получи 60 точки.

Решение със сложност $O(NlogN)$

Решението на автора на задачата се базира на следната основна идея:

Нека сме успели да построим помощен масив *pom*, чийто елементи са цели числа и удовлетворяват следните две условия:

- Всички те са различни (но не са непременно в диапазона от 1 до *N*);
- *L-масивът* на *pom* съвпада с въведения масив.

За всеки елемент на *pom* помним и индекса му в масива (за целта елементите на *pom* са от структурен тип – стойност и индекс). Да сортираме *pom* по стойностите на елементите му (първоначалните индекси на елементите от *pom* се разместват заедно със съответните стойности). Да си представим какво се получи: на първо място в сортирания масив застана елементът с най-малка стойност. В пермутацията на числата от 1 до *N* най-малко е числото 1. Значи то трябва да отиде на онова място в търсената пермутация, където е стоял преди сортирането най-малкият елемент на построения масив. Тъй като пазим индексите преди сортирането, то знаем къде да запишем 1 в търсената пермутация. Продължаваме по същия начин с втория елемент от сортирания масив – индексът в него ни показва къде трябва да запишем 2 в търсената пермутация. По този начин с едно минаване през сортирания масив построяваме търсената пермутация.

Остана въпросът как да построим масив *pom*, удовлетворяващ горните две условия. За целта използваме стек, в който държим индексите на елементите, които още не са покрити. Трябва ни и едно число $d>N$. Преглеждаме въведения *L-масив* (нека да се нарича *L*) отляво надясно.В първия елемент на този масив винаги ще стои 0, тъй като първият елемент от масива *pom* нищо не покрива. Най-спокойно в *pom[1]* можем да запишем 0 и вкарваме 1 в стека. Продължаване да преглеждаме масива *L* и последователно за всяко $i \geq 2$ правим следното: ако *L[i]*=0, то значи, че *i-тият* елемент на масива ще бъде по-малък от елемента с номер *i-1* – в този случай записваме *pom[i]=pom[i-1]-d*; ако *L[i]>0*, то *i-тият* елемент ще покрива *L[i]* елемента, чийто индекси се намират във върха на стека. Забележете, че елементите с тези индекси вече са получили стойност, индексите в стека вървят в намаляващ ред от върха навътре, а

стойностите на елементите, които чакат покриване (чийто индекси са в стека) вървят в нарастващ ред при намаляващия ред на индексите. Вадим *L[i]* индекса от стека и нека последният от тях е със стойност *c*. Тогава правим *pom[i]=pom[c]+1*. При такова построяване *pom[i]* ще покрива точно *L[i]* елемента. И в двата случая (*L[i]=0* и *L[i]>0*) вкарваме *i* в стека. При избора на *d>N* лесно се съобразява, че в масива *pom* няма да се получат равни елементи.
Сложността на този алгоритъм се определя от сортирането на масива *pom* и следователно е *O(N*logN)*. Той е реализиран от Руско Шиков във файл **towers.cpp** и носи 100 точки.

Друг алгоритъм със същата сложност беше предложен от Александър Георгиев и Йордан Чапъров в близки версии. Той също използва стек, в който пази индексите на непокритите елементи, обхожда линейно масива *L* и при срещане на *L[i]*>0 вади *L[i]* индекса от стека, като определя стойността на *i-тия* елемент да бъде по-голям от стойността на елемента с последен изваден от стека индекс. В този алгоритъм се използва индексно дърво, за да се преизчисляват стойностите на вече определените елементи, така че да няма повтарящи се елементи в построената до момента пермутация (нещо като в алгоритъма с квадратична сложност, но в случая индексното дърво води до сложност *O(N*logN)*). Този алгоритъм е реализиран от Александър Георгиев във файл **towersTree.cpp** и също носи 100 точки.

Автори: *Руско Шиков*
Александър Георгиев
Йордан Чапъров
Антон Шиков