АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА ДВОИЧНО ТЪРСЕНЕ

Първата част от анализа е на автора Антон Анастасов.

За учудване на журито много състезатели се справиха успешно с тази задача. Оказа се че има доста по-лесно решение на задачата, което те бяха измислили. Анализ на това решение, написан от Енчо Мишинев, се прилага след анализа на автора.То е със сложност O(N*logN). Вече след състезанието Енчо измисли и линейно решение на задачата, чийто анализ също се прилага по-долу.

Първото решение на задачата, което идва на ум, е да изпробваме всяка от N!-те пермутации и да проверим за кои тях двоичното търсене на Антон работи правилно. Токова решение има факториелна сложност и за него са предвидени общо 10% от тестовете. Съществуват и други решения, които имат експоненциална сложност, но могат да се справят с входове, при които $1 \leq N \leq 20$. За тях са предвидени общо 20% от тестовете. За да решим по-големи входове, трябва да стигнем до решение, което има полиномиална сложност относно *N*.

Задачи, които питат за брой на някакви комбинаторни структури, обикновено се решават с помощта на динамично програмиране. Тази задача също може да се реши с такъв подход. Първият проблем, с който се сблъскваме, е как да дефинираме структурата на подзадачите. Може да се забележи, че двоичното търсене на Антон е напълно правилно (с лекия пропуск, че не работи правилно върху празни масиви, което не представлява проблем за дадената задача). Нека помислим какво се случва докато правим двоично търсене върху пермутация на числата от 1 до *N*: избираме "средния" индекс *m* (кавичките са поради случаите, в които масивът има четна дължина и съответно трябва да изберем първия от двата "средни" елемента) и проверяваме стойността от масива с индекс *m*. В зависимост от стойността на *a*[*m*] има 3 случая:

- Случай 1: *a*[*m*] е равно на *value*: в този случай приключваме, защото сме открили стойността, която търсим. Забележете, че при този случай няма никакво значение как са разпределени другите елементи от масива.
- Случай 2: *a*[*m*] е по-голямо от *value*: в този случай, игнорираме всички индекси, които са по-големи от *m* и продължаваме да търсим двоично в лявата част от масива.
- Случай 3: *a*[*m*] е по-малко от *value*: в този случай, игнорираме всички индекси, които са по-малки от m, и продължаваме да търсим двоично в дясната част от масива.

От тук нататък, ще наричаме подмасив частта от масива *a*, която разглеждаме в даден момент от двоичното търсене; в началото, това е целият масив. След първата итерация на двоичното търсене, ще сме останали с интервал, който е "половината" от началния.

Наблюдение 1: Може да дефинираме подзадача с две цели числа: *smalls* — броя на числата в интервала, които са по-малки от *value*, и *bigs* — броя на числата в интервала, които са по-големи от *value*.

Това е най-важното наблюдение, което трябва да се направи. Такава подзадача е добре дефинирана. Нека първо да видим как такава подзадача всъщност ни решава задачата. Дефинираме *f*(*smalls*, *bigs*) да е броя на пермутациите на *smalls* + *bigs* + 1 елемента, където елементът, който търсим с двоично търсене е по-голям от всички *smalls* на брой елементи и по-малък от всички *bigs* на брой елементи. Например, ако *smalls* е 4, а *bigs* е 2, то *f*(*smalls*, *bigs*) съответства на броя на пермутациите между числата 1 и 7 (7 = 4 + 2 + 1 за елемента, който търсим), включително, за които двоичното търсене на Антон намира числото 5 (5, защото сме дефинирали, че числото, което се търси, е по-голямо от всички *smalls* = 4 на брой числа). Ако изначално получаваме на входа *N* и *K*, то задачата ни е да пресметнем *f*(*K* - 1, *N* - *K*), защото има точно *K* - 1 числа по-малки от *K*, и точно *N* - *K* по-големи от *K*.

Вече знаейки, че горните подзадачи ще ни помогнат за решението на задачата, трябва да изведем рекурентна зависимост, която да свързва различните подзадачи.

Рекурентна зависимост: Нека отново се замислим какво се случва, когато правим двоично търсене. Разглеждаме "средния" елемент както по-горе. Този среден елемент има стойност измежду 1 и *N*.

- Средният елемент е точно *smalls* + 1. Това означава, че директно ще открием елемента, който търсим и няма нужда да продължаваме. Останалите елементи от масива може да са пермутирани по всякакъв начин. Следователно, в този случай има точно ((*smalls* + *bigs* + 1) - 1)! пермутации, за които средният елемент е точно *smalls* + 1.
- Средният елемент е по-малък от този, който търсим. Това означава, че ще продължим двоичното търсене в дясната част от масива. Тъй като знаем колко елемента общо има в тази дясна част и искаме двоичното търсене да открие търсения елемент, остава единствено да преценим колко числа, по-малки от *smalls*+1, и колко – по-големи от *smalls*+1 ще бъдат в тази дясна част. За това, забелязваме, че за този случай трябва да изберем в дясната част да има измежду 0 и smalls-1 (защото средният е бил по-малък) елемента, по-малки от търсения, а останалите елементи в дясната част трябва задължително да са по-големи от търсения. Тъй като останалите числа трябва да са в лявата част, получаваме рекурентна зависимост, при която *f*(*smalls*, *bigs*) зависи от O(*smalls*) на брой подзадачи, защото, като фиксираме броя на по-малките числа в лявата част масива, еднозначно определяме броя на по-малките числа в десния. Тъй като знаем колко точно елемента има общо в лявата и в дясната част, знаем и еднозначно колко по-големи ще отидат в дясната и колко – в лявата част. За да изберем кои по-малки числа отиват в дясната част и кои в лявата след като сме фиксирали техния брой, трябва да използваме биномни коефициенти; също забелязваме, че, тъй като ще търсим двоично в дясната част, елементите в лявата може да бъдат пермутирани по всички възможни начини. Следователно за фиксиран брой по-малки елементи в лявата част, броят на валидните пермутации за двоично търсене е произведението на 4 числа: два биномни коефициента за избиране кои по-малки и кои по-големи числа в коя част отиват; един фактириел, за да пермутираме числата в частта, в която няма да търсим; число, съответстващо на валидните пермутации за двоично търсене в дясната част, което е просто *f*(*a*, *b*) за конкретни стойности на *a* и *b*.
- Средният елемент е по-голям от този, който търсим. Доста подобно на горния случай. За повече детайли, разгледайте предложените решения.

При така дефинираните подзадачи, имаме O(N^2) подзадачи, при всяка от които има *N* възможности за "средния елемент", и за всяка такава възможност зависи от най-много O(*N*) (по-точно O(*smalls*)) подзадачи, които вече сме решили. Общата сложност е O(N^4) и такова решение за определени 50% от тестовете. Забележете, че предварително може да пресметнем всички факториели за O(*N*) време, и всички биномни коефициенти за O(N^2), след което да ги използваме наготово.

Наблюдение 2: След като вече сме направили по-трудната част, остава да направим няколко по-лесни наблюдения, за да оптимизираме горното решение.

Може да забележим, че въпреки, че "средният" елемент има една от *N* възможни стойности, то има точно 3 различими случая — дали средния елемент е този, който търсим, дали е по-голям от този, който търсим, или по-малък. Наистина, ако разпишем рекурентната зависимост, ще видим, че ако числото на средната позиция е по-малко от това, което търсим, няма значение колко точно е то. Това свежда горното решение до O(N^2) подзадачи от 3 случая, всеки от които зависи от най-много O(*N*) подзадачи, което води до сложност O(N^3). Тази модификация трябва да реши близо до 65% от тестовете.

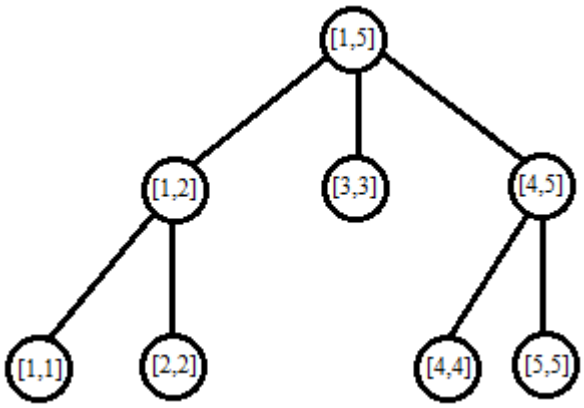
Наблюдение 3: Ако реализираме горното решение рекурсивно, ще видим, че то по-скоро работи като O(N^2), а не O(N^3). Реално се оказва, че броят на подзадачите, които е

нужно да решим, е точно O(N), а не O(N^2), което е доста изненадващо отначало. Причината обаче е доста интересна: след като разглеждаме масив с N елемента, свеждаме задачата до такава с N/2 елемента, защото правим двоично търсене. Съответно не е нужно да решаваме никакви подзадачи, за които броят на елементите в масива е N/2+2, N/2+3, …, N-1, защото нашата подзадача не зависи от тях! В такъв смисъл, ако разглеждаме правилно кои задачи са нужни, ще видим, че техният брой е O(N). Тъй като всяка подзадача зависи от най-много O(N) други, то цялата сложност е O(N^2). В зависимост от имплементацията, такова решение се очаква да реши 75% от тестовете.
Наблюдение 4: Оказва, се че един от лимитиращите фактори е използването на паметта. Най-лесно от начало беше да пресметнем всички биномини коефициенти, като използваме познатата рекурентна зависимост $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$. Такова пресмятане, обаче, изисква да пазим матрица O(N^2), което изисква твърде много памет. На помощ ни идва деленето по модул и наблюдението, че модулът, който разглеждаме (1 000 000 007), е просто число. Това, позволява бързо да делим (или с разширения алгоритъм на Евклид, или с бързо повдигане на степен). Използваме дефиницията на биномния коефициент, според която $C_n^k = \frac{n!}{k!(n-k)!}$, за което са ни нужни 3 факториела (които можем да изчислим предварително) и да извършим делене по зададения прост модул. Такова решение решава всички тестове.

Автор Антон Анастасов

Решение със сложност O(N*logN)

Нека за дадено N построим троично дърво показващо възможните действия на алгоритъма. Коренът на това дърво ще съответства на интервала [1; N]. Ако даден връх съответства на интервала [L; R], то децата му ще съответстват съответно на интервалите [L; M-1] (ляв син), [M; M] (среден син), и [M+1; R] (десен син) като M=$\left\lfloor \frac{L+R+1}{2} \right\rfloor$. Ако някой от интервалите се окаже празен, то приемаме, че този син не съществува. Така изпълнението на двоичното търсене всъщност се симулира като път от корена до дадено листо в дървото – преминаването от връх в левия му син означава, че в средата на текущия интервал е било число по-голямо от търсеното, преминаването към среден син означава, че сме намерили търсеното число и преминаването към десен син означава, че в средата на текущия интервал е било число по-малко от търсеното. Примерно дърво за N=5 е:



За да намери успешно стойността *value*, алгоритъмът трябва да завърши в листо съответстващо на индекса, на който се намира *value*. Нека за достигането до това листо е трябвало X пъти да слезем към ляв син и Y пъти към десен син. Искаме да намерим броя пермутации, при които алгоритъмът би слязъл точно до това листо, и на индекса съответстваш на него се намира *value*. Това очевидно значи, че позицията на *value* е

фиксирана. Остава да изберем *X* числа по-големи от *value* измежду останалите, които да са принудили слизането към леви синове и *Y* числа по-малко от value, които да са принудили слизането към десни синове. Измежду останалите числа има точно *value-1* по-малки от *value* и точно *N-value* по-големи. Тъй като реда на избиране има значение, то използваме вариации – броят начини да изберем X по-големи числа от *value* и Y по-малки е:

$\mathbf{V}(N\text{-}value,\ X) \times \mathbf{V}(value\text{-}1,\ Y) = \frac{(N-value)!}{(N-value-X)!} \times \frac{(value-1)!}{(value-1-Y)!}$

След това разпределяне остават *N-(X+Y+1)* числа, които можем да подредим по произволен начин тъй като няма да бъдат разгледани от двоичното търсене. Получаваме, че ако за достигане на дадено листо трябва да направим X слизания към ляв син и Y слизания към десен син – то броя пермутации, при които двоичното търсене завършва в това листо и намира стойността *value* е точно:

$\frac{(N-value)!}{(N-value-X)!} \times \frac{(value-1)!}{(value-1-Y)!} \times \big(N-(X+Y+1)\big)!$

При *X>N-value* или *Y>value-1* считаме стойността от формулата за равна на 0.

Тъй като дървото има точно N листа и всеки връх, който не е листо, има поне два сина, то в дървото има най-много *2N-1* върха. При предварително пресмятане на всички факториели на числата от 1 до N, бихме могли да сметнем стойностите на горната формула при фиксирано листо (и съответно X и Y) за O(log 10^9+7) тъй като трябва да "делим по модул".

Решение със сложност O(N)

Можем обаче да забележем, че тъй като при всяко придвижване увеличаваме X или Y с най-много 1, то можем да започнем с X=0,Y=0 в корена на дървото и при всяко придвижване за O(1) да пресмятаме получаващата се вариация, използвайки пресметнатите до сега стойности. Пресмятаме лесно използвайки:

$\mathbf{V}(N\text{-}value,\ X+1) = (N\text{-}value\text{-}X) \times \mathbf{V}(N\text{-}value,\ X)$

$\mathbf{V}(value\text{-}1,\ Y+1) = (value\text{-}1\text{-}Y) \times \mathbf{V}(value\text{-}1,\ Y)$

Така можем само с едно обхождане да пресметнем дадената формула за всяко листо за по O(1). Отговорът на задачата е сума от стойностите получени от всяко листо. Общата сложност се определя от големината на дървото и пресмятането на факториелите, и е съответно е O(*2N-1 + N*) = O(N).

Имплементацията се свежда до едно обхождане в дълбочина на даденото дърво, като не е нужно то реално да бъде строено.

Автор: Енчо Мишинев