Упражнение №4 по САА

Разделяй и владей

Произход и идея на техниката. Части. Сложност на алгоритми, базирани на "Разделяй и Владей".

В темите до сега разгледахме няколко задачи за търсене и сортиране, в които итерирахме елементите на дадено множество. Не винаги се налагаше да итерираме всички елементи на множеството - например, в задачата за търсене на X в масив, понякога спирахме итерацията по-рано (ако намерим съвпадение). В много задачи ще ползваме това като оптимизация на алгоритъма ни - да не изброяваме всички елементи, когато това не е нужно.

Нещо повече, в тази тема ще покажем една хитра техника, която ни позволява да разглеждаме само тези от елементите на множеството, които са ни достатъчни за да изпълним поставената задача. Това понякога намаля броя разглеждани елементи в итерацията с огромен процент, като в резултат получаваме много по-бързи алгоритми.

Произход

Има няколко (що-годе) различни мита откъде произлиза израза "Разделяй и Владей", като може би най-близкият до идеята на алгоритмичната техника е следният. При разширяването на Римската империя, управниците (или генералите) са ползвали тактиката да разединят населението на дадена територия, преди да я нападнат. Чрез подкупи и фиктивни конфликти те са насъсквали местните племена едно срещу друго, като така са ги правили много по-слаб противник. В последствие Римляните са воювали само с отделни племена или части от разединеното население, като са печелили битките много по-лесно. Разделени, поданиците на империята не са могли да осъществят и големи бунтове, което е позволило да бъдат под римско владение векове наред.

Идея

Идеята на техниката "Разделяй и Владей" е същата - да разделим поставения ни проблем на по-малки такива, които можем да решим (завладеем) по-лесно, като в последствие обединим резултатите им и изградим отговор за целия проблем.

Разделяй

В процеса на разделяне ще гледаме да разбием задачата на *по-малки, незастъпващи се* подпроблеми, които можем да решим или тривиално (толкова са малки, че знаем отговора), или с допълнително раздробяване.

Защо по-малки?

Ако проблемите са по-малки, то най-вероятно ще са и по-лесни, като най-малките им варианти (базовите случаи) можем да сметнем и наум (или вече знаем отговора) и да хардкоднем в програмата си.

Защо незастъпващи се?

Ако подпроблемите се застъпват, то, най-вероятно, резултатът на единия ще зависи от резултата на другия. Това е нещо, което не можем да сметнем, и означава, че не сме разбили проблема като хората (ако той изобщо може да бъде разбит).

Дори ако резултатът на всеки от подпроблемите не зависи от резултата на никой от останалите такива, макар и да се застъпват, то е възможно застъпването да доведе до много повтаряща се работа, която изчисляваме отново и отново (правейки решението ни експоненциално -- тоест много бавно). Такива проблеми се решават с техниката динамично оптимиране.

Владей

Важно е след като сме изчислили резултатите на подпроблемите, да има начин, по който да ги съчетаем, за да получим отговора на проблема, който решаваме.

Ели, Крис и Станчо правят тест по математика. Всяка от задачите може да бъде решена от всеки от тримата приятели за 5 минути, но тестът се състои от 20 задачи за 60 минути. Те седят един до друг и са усъвършенствали едно от важните качества за това да бъдеш ученик - да можеш да преписваш. След като някой е сметнал отговора на дадена задача, на всеки от останалите двама му е нужна само една минута за да препише отговора. Могат ли тримата да направят теста успешно?

Както може би се досещате - да. Ще си разделят задачите на три (що-годе равни) части - 7, 7 и 6 задачи, като ще се нуждаят от, съответно, 35, 35, 30 минути за решаване на тяханата част, и 13, 13, 14 минути, за да препишат отговорите. Всеки от тях първо решава всички поставени му задачи и после почва да преписва останалите (вече решени) задачи. Така ще са им нужни 48 минути за целия тест.

Тук частта "Разделяй" е това, че си поделят задачите помежду си (по-лесно се решават 7 задачи, отколкото 20), а "Владей" е това да препишат отговорите на задачите, които не са решили (но са решени от друг).

След доброто си представяне на теста по математика, Ели, Крис и Станчо решиха да станат състезатели по информатика. Сайтът, където се подготвят, съдържа 90 теми, като всяка от темите отнема по приблизително една седмица за да бъде научена. До важното състезание, за което се подготвят, остава точно една година (52 седмици), като на него има много сложни задачи, изискващи теория и/или трикове от всяка една от темите. Могат ли да се подготвят те?

В тази (донякъде подобна) задача, номерът с разделяне на темите помежду им не

работи. Макар и те да могат да направят това, като научават по 30 теми всеки (за което са им нужни 30 седмици), то тъй като задачите изискват знания от *всички* теми, то никой от тях не може да реши никоя от задачите (тоест наученото не може да бъде комбинирано). Точно това, че не можем да приложим стъпката "Владей", не разрешава тази задача да бъде решена с тази техника.

Примери

Техниката е приложима в наистина много ситуации. Ще разгледаме два примера, които илюстрират как тя може да влезе в истинска задача, където разделянето и владеенето не е изобщо толкова очевидно, колкото при горните хипотетични задачи.

Ханойски кули

Дадени са ви три стълба (ляв, среден и десен) и N диска с различен диаметър. Дисковете имат дупка по средата, за да могат да се надяват на стълбовете. Всеки диск може да бъде сложен на всеки стълб, стига под него да няма друг диск с по-малък диаметър. В началото всички дискове са на най-левия стълб, като е спазено правилото по-горните дискове да са по-малки от по-долните. От вас се иска да ги преместите на най-десния стълб, като на всеки ход можете да местите само най-горния диск от един стълб на друг, като спазвате изискването под него да има само дискове с по-голям диаметър. Вижте тази снимка за пример (начална подредба) с $N = 8$.

Задачата за *Ханойските Кули* е много известна и със сигурност ще я срещнете и друг път (ако вече не сте). Съществува стратегия, която извършва местенето с доказан минимален брой ходове.

Съществува яка легенда, свързана с Ханойските кули. Тя гласи, че в Индия има храм, в който има стая с три такива стълба и 64 златни диска, като монасите ги местят по правилата на гореописаната задача. Когато преместят и последния диск, светът ще свърши.
Добрата новина е, че дори това да е вярно и те да ползват оптимална стратегия и местенето на един диск отнема само една секунда, то цялата задача ще им отнеме 585 милярда години.

Всъщност, ако не сте срещали задачата преди я помислете :) Колко ходя ви трябват за да преместите 5 диска?

Сега ще ви подскажем, че има решение, базирано на "Разделяй и Владей". Голяма подсказка, имайки предвид, че даваме задачата като пример в темата. Ако не сте измислили оптималното решение по-рано (отговорът за 5 е 31), то помислете как техниката може да бъде приложена тук.

Решението, всъщност, не е много трудно. Нека разгледаме по-общата задача, в която всички дискове са на някой от стълбовете (не задължително най-левия), който ще наричаме "начален". Искаме да ги преместим на някой друг (не задължително най-десния), който ще наричаме "краен". Третия стълб (който не е нито начален нито краен) ще наричаме "временен". Стратегията ни е да преместим горните *N-1* диска на временния, после да преместим *N*-тия диск (който е останал сам на началния стълб) на крайния, след което да преместим *N-1*-те от временния на крайния стълб.

Тук "разделяме" дисковете на две части - в едната са горните *N-1*, а в другата - само най-големият. Самата *задача* бива разделена на *три* части (три етапа на "владеене"), но в две от тях местим еднаква група дискове.

Да преместим *N-1* диска от началния стълб на временния (ползвайки крайния за временен) е *по-малка* задача *от същия тип*. И тъй като задачата с 1 диск е тривиална, то подобно свеждане на задача към по-малка подзадача е хубаво за нас (тъй като рано или късно ще стигнем до случай, който знаем как да решим). Преместването на най-големия диск (след като сме преместили по-малките) е тривиална задача. След това местим отново *N-1* диска, като при първата част.

Ще приложим примерна имплементация на алгоритъма. На функцията подаваме четири числа - първото е *N*, тоест колко диска имаме да преместим, а останалите три са кой е началният стълб, кой е временният такъв и кой е крайният. За простота сме означили левия с нула, средния - с едно и десния - с две.

```
const char* pegs[3] = {"left", "middle", "right"};
void hanoiTowers(int n, int initial, int temporary, int final) {
    if (n <= 0)
        return;
    // Местим по-малките дискове от началния стълб на временния, ползвайки
    // крайния за временен (накрая той остава пак празен).
    hanoiTowers(n - 1, initial, final, temporary);
    // Местим най-големия диск директно
    fprintf(stdout, "Moving disc from %s to %s.\n", pegs[initial], pegs[final]);
    // Местим по-малките дискове от временния стълб на крайния, ползвайки
    // началния за временен (накрая той остава пак празен).
    hanoiTowers(n - 1, temporary, initial, final);
}
```

Тримино

Друга интересна задача, свързана с техниката, е задачата за триминота. Тук стъпката "разделяй" изисква малко креативност, но създава особено красиво решение.

Даден е квадрат със страна 2^N, разделен на малки квадратчета (плочки) със страна *1*. Едно от тях е оцветено в черно, всички останали в бяло. На всеки ход ние можем да поставим trimino, успоредно на страните на квадрата, ако то е разположено само върху

бели плочки. Trimino е плочка от три квадратчета под формата на Г (евентуално завъртяно). Тоест квадрат с размер 2 на 2, на който липсва една от плочките. След поставянето му, трите бели плочки, върху които лежи, стават черни. Възможно ли е да се оцвети целият квадрат в черно с поставяне на trimino-та? Как?

Първо помислете сами малко.

Хайде де, още малко, нали целта е вие да се научите да измисляте хитри неща.

Така. Ако сте успели да го измислите, можете да се чувствате много горди от себе си. Ако не сте, ето и решението. Ако дъската е 1 на 1, очевидно има решение (без нито едно тримино). Ако дъската е 2 на 2, то също очевидно има решение. Нека разгледаме случая, в който дъската е 4 на 4.

```
.... => 1122 или .... => 5544 или #... => #122
.#.. => 1#32     .... => 5334     .... => 1132
.... => 4335     .... => 2311     .... => 4335
.... => 4455     ..#. => 22#1     .... => 4455
```

Намерихме решение на три от възможните начални разположения. След известно наблюдение се забелязва, че всяка дъска с размер 4 на 4 има решение чрез ротация и/или симетрия на някоя от горните три дъски. Бихме могли да предположим, че *всяка* дъска със страна 2^N има решение. Сега да намерим стратегия и как да намираме самите решения.

Нека разделим началната дъска на 4 равни части (два разреза: един през средата по X и един през средата по Y). В една от тях е първоначалното черно квадратче, а останалите 3 са изцяло бели. Но ако сложим едно тримино в центъра (там, където двата разреза се пресичат) така, че липсващата му плочка да застъпва частта, която съдържа началната черна плочка, то вече всяка от четирите части е с размер 2^{N-1} и има по една черна плочка. Но това е същата задача с по-малък размер! Следователно можем да решим рекурсивно със същия алгоритъм всяка от четирите подзадачи и да "съединим" решенията, получавайки решение и за големия квадрат. Дъното на рекурсията може да бъде дъска с размер 1 или 2, което предпочетем. Нека видим как би изглеждало първото разрязване на дъска с размер 8 на 8:

```
........    ........
.#......    .#......
........    ........    ....   ....   ...1   1...
........    ....1...    .#..   ....   ....   ....
........ => ...11... => .... , .... , .... , ....
........    ........    ....   1...   ....   ....
........    ........
........    ........
```

Яко, нали? Ако искате да напишете цяло решение на задачата, може да се пробвате да направите това в задачата **Gamers**.

Задачи

Съществуват много алгоритми и техники, базирани на техниката "Разделяй и Владей". Примери за известни такива са: двоично търсене, merge sort, бързо умножение на

матрици, бързо умножение на дълги числа, алгоритъм на Евклид и други.

В процеса на изучаване на следващите теми ще се сблъскате с много задачи, чиято идея е базирана на "Разделяй и Владей". Най-често те ще се решават с двоично търсене, но съществуват и по-сложни (и интересни) проблеми, които са податливи на тази техника.

Една още по-хитра от горните задачи, която може да се реши така, е задачата **Подслушване** от втория кръг на НОИ през 2011 година.

Двоично търсене

Задача за познаване на число. Идея на двоичното търсене. Имплементация. Изисквания. Други примери.

В темата за търсене и итерация разгледахме няколко примера, в които трябваше да намерим специфичен обект сред други такива. Там ползвахме оптимизацията, че веднага след като го намерим, можем да приключим итерацията и така да направим програмата си малко по-бърза. Това, обаче, в най-лошия случай не ни дава никакво подобрение. Примерно, ако елемента, който търсим, го няма, то нашата програма би минала през цялото множество без резултат. Нещо повече, в *средния* случай тази оптимизация прави програмата едва два пъти по-бърза, тъй като очакването е (ако го има) да го намерим, след като сме проверили половината стойности.

Тук ще разгледаме как с техниката "Разделяй и Владей" можем да направим решението си много, много по-бързо, ако са изпълнени определени изисквания.

Проблем

Нека разгледаме следната примерна постановка.

> Ели и Станчо играят на следната игра. Станчо си намисля число между 1 и 1000, включително, а Ели се опитва да го отгатне. След всяко нейно предположение, Станчо ѝ казва дали е уцелила, и ако не е - дали неговото число е по-голямо или по-малко ("нагоре" или "надолу"). За всяко питане тя трябва да пие шотче... натурален сок, а след като познае, Станчо трябва да допие останалото в бутилката. Тъй като от натуралния сок на нея ѝ се замайва главата, тя се стреми да познае числото на Станчо с възможно най-малко питания. Помогнете на Ели да измисли стратегия, с която да постигне това.

Тривиален подход

Очевидно, една възможна стратегия е да пита дали числото е 1, после 2, после 3 и т.н. Така, обаче, ако Станчо си е намислил 1000, то на нея ще ѝ отнеме цели 1000 питания за да го познае. Разбира се, тя може да почне от другия край (придвиждайки, че Станчо е хитър), но той може да придвиди, че тя ще придвиди, че той е хитър, и да си намисли 1. От където и да почне тя, в най-лошия за Ели развой на събитията, този подход задава 1000 въпроса.

Прескачане на числа и линейно търсене

При миналия подход Ели изобщо не използва това, че Станчо ѝ казва дали неговото число е по-голямо или по-малко. На нея ѝ хрумва, че може да пита само за нечетни числа, като по това, какво ѝ отговаря Станчо, тя може да се ориентира, дали току-що е прескочила неговото число. Например, тя пита за 1, 3, 5, … K*2+1, при което Станчо изведнъж казва, че неговото число е по-малко. Тя веднага знае, че неговото число е K*2. Така тя би намалила броя питания на половина (500 в най-лошия случай).

Доразвиване на стратегията с прескачане и линейно търсене

Добре, но тогава защо тя да не пита за не през едно число, ами през десет? Например за 1, 11, 21, … 10*K+1. В момента, в който Станчо ѝ каже, че неговото число е по-малко, тя ще пита за деветте числа между предходното и текущото ѝ питане, като със сигурност едно от тях ще е това на Станчо. С този подход тя прави 100 + 9 или общо 109 питания, в най-лошия случай.

Оптимална стратегия с прескачане и линейно търсене

Всъщност, защо пък десет? Колко числа да прескача тя, така, че общият брой питания да е минимален?

С малко мислене можем да съобразим, че оптималният брой числа, които трябва да пропуска Ели при тази стратегия, е корен квадратен от общия брой числа. Той е оптимален, тъй като балансира броя питания при прескачането и този при линейното търсене след това. В случая sqrt(1000) е приблизително 32. Така тя може да пита за 1, 33, 65, 97 и т.н., което са общо 32 питания. След това тя трябва да пита за (в най-лошия случай) всички числа между предходното и последното ѝ питане, които са още 31. Така тя има стратегия с най-много 63 питания.

Рекурсивна стратегия с прескачане

Някои от вас сигурно са забелязали, че тази стратегия има слаба част. След като намери интервала с 31 числа, в който се намира това на Станчо, Ели използва тривиалната стратегия, обсъдена съвсем в началото. Но както видяхме, тая е много далеч от оптимална. Защо, тогава, да не приложим стратегията с прескачане рекурсивно и в намерения малък интервал? Тоест, след като намери интервала с 31 елемента, където е числото на Станчо, тя прилага *същата* стратегия, но в много по-малък интервал. В случая, при първото викане стъпката ще е 32, при второто ще е sqrt(31) ≈ 5, след това sqrt(4) = 2 и накрая ще има едно единствено число, за което тя ще е сигурна, че е това на Станчо. Общият брой питания ще е 32 + 6 + 2 + 1 = 41.

Заключение за стратегията с прескачане

Тази стратегия, базирана на "Разделяй и Владей", *гарантирано* намира числото на Станчо, независимо кое си е намислил той. Нещо повече, то намира числото с най-много толкова питания, дори той да послъгва (променя намисленото си число), стига

отговорите му да са консистентни.

Въпреки последния фикс, в тази идея все още има дребна грешка. Ако я оправим, можем да постигнем дори по-добри резултати. За всяко рекурсивно викане прескачаме по корен квадратен от текущия брой числа. Защо? Защото казахме, че така балансираме броя въпроси при прескачане и броя такива при линейното търсене. Но нали го премахнахме (правейки алгоритъма рекурсивен)? Значи вече не зависим от него и може би има по-добра стъпка!

Двоично търсене

Вместо да прескачаме по корен квадратен на брой числа, ще прескачаме по... половината! На първата стъпка Ели пита за числото 500. Ако Станчо каже, че неговото е по-малко, то Ели трябва да го намери в интервала [1, 499]. В противен случай, тя трябва да го намери в [501, 1000]. Но и при двата случая тя елиминира *половината* числа с едно единствено питане! Както сами можете да видите, на втората стъпка тя ще елиминира половината на половината (тоест около 250 нови числа), и т.н. Такова търсене се нарича *двоично търсене*, тъй като на всяка стъпка разделяме интервала на две и захвърляме половината възможности.

Методът "Двоично търсене" се нарича "Binary Search" на английски, като много често ще го чуете като "Байнъри" от български състезатели. В литературата и в университета също така може да го срещнете като "Bisection Method" (метод на бисекцията) или "Dichotomy" (дихотомия).

При двоичното търсене, размерът на претърсваното пространство намалява на половина след всяка стъпка. Съществува модификация на техниката, при която той намалява само с една трета, но пък решава по-сложен проблем. Тази техника се нарича троично търсене и ще разгледаме по-нататък.

Да видим колко хода ще са нужни на Ели, за да познае числото на Станчо, с тази нова стратегия. Броят числа в интервала, в който търси тя, ще са (питайки за едно и изхвърляйки половината на всяка стъпка): {1000, 499, 249, 124, 62, 31, 15, 7, 3, 1}. Тоест Ели може да намери числото на Станчо с едва десет питания!

Идея

Идеята е, като имаме някакъв интервал, в който търсим някаква стойност, да разделяме интервала винаги на половина, като проверяваме какъв е резултатът за стойността, която се намира по средата. В зависимост от този резултат, решаваме дали текущата стойност е тази, която търсим, и ако не - дали да продължим търсенето в лявата или дясната половина.

Забележете, че стъпката "проверяваме какъв е резултатът за стойността, която се намира по средата" изобщо не е задължително да е тривиална. В примерната задача Ели

имаше Станчо, който да казва резултата, но в огромна част от реалните задачи това ще е отделен алгоритъм, който трябва да имплементирате. А той може да е почти всякакъв - от просто сравняване на числа до сложни и дълги алгоритми като, например, поток.

Тъй като, само по себе си, двоичното търсене е сравнително кратко и просто за писане, най-често в задачи то бива съчетано с друг, по-сложен алгоритъм или структура данни.

Брой стъпки на алгоритъма

Както видяхме в примерната задача, Ели се нуждаеше от 10 питания за да познае число от множество с 1000 елемента. От колко питания би се нуждала Ели, ако Станчовото число беше между 1 и 100,000?

Нека видим! Размерът на интервалите, при съответните питания, е: {100000, 49999, 24999, 12499, 6249, 3124, 1562, 781, 390, 195, 97, 48, 24, 12, 6, 3, 1}. Едва 17! Въпреки, че интервалът е сто *пъти* по-голям, броят питания не се увеличи дори двойно.

Това е така, тъй като стратегията ни игнорира огромен брой числа в началото - колкото по-голям е началният интервал, толкова повече игнорирани числа. И тъй като намаляме интервала на две всеки път, то броят питания е равен на *двоичния логаритъм* от големината на интервала. Наистина, $\log_2(1000) \approx 10$, $\log_2(100000) \approx 17$.

Колко питания биха били достатъчни на Ели, ако числото на Станчо беше между 1 и 1,000,000,000?

Имплементация

Имплементацията е проста и сравнително стандартна, в следствие на което много състезатели ползват един и същ шаблон винаги, когато пишат байнъри.

Първо, тъй като търсим нещо в някакъв интервал, е хубаво да си представим интервала по някакъв начин. Често ще търсим нещо в масив, като интервалът ще бъде представен чрез най-левия и най-десния валиден индекс. Понякога, обаче, (като например в тази задача) ще имаме друга функция, в която търсим отговора.

За сега ще разглеждаме само дискретни функции (тоест такива, които имат краен брой елементи), като например интервал от цели числа или индекс в масив. По-нататък ще обърнем специално внимание на недискретния вариант, тъй като там има няколко трика, на които искаме да ви научим :)

Бъдете внимателни за overflow ако ползвате съкратения вариант! Забележете, че дори целият интервал да се събира в даден тип, то *сумата* на лявата и дясната граница може да не се. Например, какво става ако имате интервала [1, 2000000000], границите са ви от тип int (съвсем логично, тъй като типът поддържа този интервал) и ползвате int mid = (left + right) / 2;? Ако отговорът е близък до горната граница, ще имате стъпки, в които (left + right) ще прехвърли възможностите на int и програмата ви ще даде грешен

резултат. В това отношение по-дългият запис е по-безопасен, тъй като няма този проблем.

Можем да представим интервала чрез две числа - най-лявата (малката) възможна стойност на интервала и най-дясната (голямата) такава, които ще наричаме *left* и *right* или за по-кратко *l* и *r*. На всяка стъпка ще изчисляваме една нова стойност - средата на интервала - в променлива *mid* или *m*. За да намерите средата на интервал по принцип взимате левия му край и прибавяте половината. Това, като код е:

```
mid = left + (right - left) / 2.
```

По-често, обаче, състезателите ползват малко по-кратък метод на записване на същото нещо:

```
mid = (left + right) / 2.
```

До кога въртим цикъла? Това частично зависи от задачата, но най-често *при дискретния вариант* - докато интервалът съдържа поне един елемент. Това лесно се изразява чрез while цикъл - например най-често ще срещнете while(left <= right) или while(left < right), в зависимост от предпочитанията на пишещия.

Примерна имплементация на задачата за Станчо и Ели би била:

```
// Приемаме, че функцията guess(X) връща 0, ако числото на Станчо е X,
// -1, ако неговото е по-малко и +1, ако е по-голямо.
int guessStanchosNumber(void) {
    int left = 1, right = 1000;
    while (left <= right) {
        int mid = (left + right) / 2;
        int ansForMid = guess(mid);
        if (ansForMid == 0)
            return mid;
        if (ansForMid < 0)
            right = mid - 1;
        else
            left = mid + 1;
    }
    return -1; // Everybody lies.
}
```

Горната задача, макар и чудесен пример за двоично търсене като цяло, не е хубав пример за различни негови имплементации, тъй като има само един единствен валиден отговор, който, очевидно, е и оптимален. В следващата задача ще има много стойности, които изпълняват изискването на задачата, но само една от тях ще е "оптимална".

Даден ви е сортиран масив с *N* (по-малко или равно на 1,000,000) цели числа числа с абсолютна стойност по-малка или равна на 1,000,000,000. Намерете индекса на най-малкия негов елемент, който е по-голям или равен на дадено цяло число *X*, отново по-малко или равно на 1,000,000,000 по абсолютна стойност. Ако такъв индекс не съществува, върнете *N*.

Както казахме, състезателите най-често ползват научен и добре трениран шаблон на

двоичното търсене. Има няколко различни такива, като тук ще покажем два от тях.

С обновяване на отговора

При този тип имаме много по-малко мислене какви да са границите и дали да имаме <= или < в while()-а. Аз лично го предпочитам, тъй като е по-труден за объркване. Идеята му е да имаме допълнителна променлива (*ans*) за отговора, която ъпдейтваме всеки път, когато срещнем "хубав" отговор.

```
int binarySearch(int* array, int arraySize, int x) {
    int ans = arraySize;
    int left = 0, right = arraySize - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (array[mid] < x)
            left = mid + 1;
        else
            right = mid - 1, ans = mid;
    }
    return ans;
}
```

С ползване на една от границите

Друг вариант, който със сигурност ще срещнете в чужд код (а може и вие да ползвате, ако ви харесва повече), е да ползваме една от границите за "отговор". Ако се вгледате в горния код ще забележете, че след излизане от while() цикъла, променливата *right* ще е отговорът минус едно.

```
int binarySearch(int* array, int arraySize, int x) {
    int left = 0, right = arraySize - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (array[mid] < x)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return right + 1;
}
```

Забележете, че всеки път, когато открием валиден отговор (тоест индекс, чиято стойност е по-голяма или равна на *X*), ние намаляме *right* да е с единица по-малко от това число. Така сме сигурни, че последният намерен валиден отговор (съответно и най-малкият такъв) е със стойност *right + 1*.

Това, обаче, не е много универсална конструкция. Например, представете си, че търсихме *най-големия* индекс, който е *по-малък* или равен на *X*. Тогава щяхме да ъпдейтваме *left* всеки път, когато намерим валиден индекс, като отговорът ни накрая щеше да бъде в *left - 1*.

Като цяло, и при двата подхода най-важното е, дали стойността в средата дава валиден (по някакви критерии) отговор или не. При първия вариант в този if (ако стойността е валидна), освен, че местим границата, обновяваме и отговора. При втория вариант

трябва да върнем границата, която местим във въпросния if (с +1 или -1, в зависимост дали е *right* или *left*).

Малко предимство на първия вариант е, че ако връщаме някаква по-странна стойност ако не намерим отговор (примерно ако връщахме *-1* вместо *N*, в горната задача), то щяхме да инициализираме *ans* с -1 в началото (вместо с *N*), като нямаше да променяме *нищо* по шаблона. При втория вариант трябва да внимаваме за това и да добавим допълнителен if, за да handle-нем този случай.

Изисквания

Очевидно, двоично търсене не е приложимо винаги. Хубаво е да знаем какви са изискванията за него, и да го ползваме само в задачи, когато би довело до верен отговор. В противен случай, резултатите могат да бъдат много лоши.

До сега си говорихме или за числото на Станчо, или за сортирани масиви. Какво е общото между двете? Нека разглеждаме връщаните стойности (отляво надясно), ако ги изчислим за *всички* стойности на началния интервал.

- В случая със Станчо и Ели, до едно време неговото число е по-голямо, после равно и после по-малко до края на интервала от възможни стойности.
- В случая с търсенето на *най-малкия* индекс в сортиран масив, чиито елемент е по-голям или равен на *X*, то до едно време отговорът е отрицателен, после става положителен (и остава такъв до края на интервала).
- В случая с търсенето на *най-големия* индекс в сортиран масив, чиито елемент е по-малък или равен на *X*, то до едно време отговорът е положителен, после става отрицателен (и остава такъв до края на интервала).

При втората и третата задача, интервалът бива разделен на две части - една със само положителни отговори и една със само отрицателни отговори. С малко наблюдение можем да видим, че и задачата за Станчо и Ели е (що-годе) от същия тип - в част от интервала местим единия индекс (можем да кажем, че това е "положителен" отговор), а в остатъка от него местим други индекс (което е аналогично на "отрицателен" отговор).

Много често отговорите ви за различни стойности на интервала няма да са въобще бинарни, но с операторите $<, >, \leq, \geq$ или други критерии ще ги свеждате до такива.

Бинарна (true/false) функция, чиито отговори са в началото само положителни (отрицателни), а после само отрицателни (положителни) се нарича *монотонна*. Техниката "двоично търсене" работи *само* при такива функции!

Работа в непрекъснат интервал

Всъщност, двоичното търсене изобщо не е ограничено до масиви или дискретни стойности. Стига да има начин, по който да свеждаме отговорите от "питанията" на всяка стъпка до монотонна двоична функция, то няма проблем да работим и с интервал

от реални числа. Единствената разлика е, че броят стъпки е по-сложен за определяне (на теория трябва да е безкрайност). Все пак, във всички състезателни *и* практически задачи отговорът се изисква с някаква точност (примерно 9 знака след десетичната точка), което прави броя възможни стойности *краен*.

Напишете функция, която намира корен трети на дадено неотрицателно реално число *N*.

Какъв ще е критерият ни за двоичното търсене? Търсим такова число X, за което е изпълнено, че X * X * X = N. Нека разгледаме възможните стойности на *X*, което са реалните числа в интервала [0, +INF). Колкото по-голямо е *X*, толкова по-голямо е произведението X * X * X. Следователно, тъй като *N* е неотрицателно, можем да твърдим, че до едно време X * X * X ще е по-малко или равно на *N*, а след това ще е по-голямо. Искаме да намерим най-голямата стойност на *X*, за която е изпълнено, че X * X * X <= N. Тази функция очевидно е бинарна и монотонна, следователно двоично търсене ще работи.

Стигаме до първата уловка в задачата. В какви граници ще бъде *X*? Както казахме, интервалът е [0, +INF), но тъй като не можем (лесно) да представим положителна безкрайност, то трябва да сложим някаква стойност. Ползвайки просто някакво много голямо число, обаче, не е хубава идея, тъй като каквато и да е тази стойност, *N* може да бъде тази стойност на трета степен плюс едно, като така нашата програма би дала грешен резултат.

Тъй като горният вариант (ползващ константа) се прецаква от това, че *N* може да е произволно голямо, то е логично да нагласим границите, ползвайки самото *N*. Стигайки до този извод, интуитивно [0, N] звучи като добра идея. Всъщност, обаче, не е. Тъй като *N* е реално число, то спокойно може да бъде, примерно, *0.5*, в който случай отговорът ще е по-голям от него. Това е *частен случай*, за който трябва да се погрижим допълнително. С малко мислене виждаме, че отговорът е в интервала [0, max(N, 1.0)], което вече е окей.

Втората уловка е, че след като проверим средата на интервала, не можем да сложим left = mid + 1 или right = mid - 1, както правихме при дискретния вариант. Вместо това ще ползваме left = mid и right = mid.

Това, обаче, прецаква условието на цикъла while (left <= right). Тъй като сравняваме реални числа, има случаи, в които това никога няма да стане. На теория, всъщност, би трябвало *никога* да не се случи, тъй като *mid* е винаги между двете и никога точно равно на което и да е от тях (ако в началото left != right).

Често начинаещите състезатели решават това с така наречените "епсилони" - тоест много малко число, което прибавят или изваждат за да се справят с този проблем. Например, би било донякъде интуитивно да ползваме едно от:

- `while(left + 0.000000001 < right)`
- `left = mid + 0.000000001`

- `right = mid - 0.000000001`

Това решава горния проблем, но създава нов такъв. Какво се случва, ако $N = 10^{-40}$? Най-вероятно няма да намерим точния отговор, тъй като цикълът ще спре много преди да сме стигнали до него. Това важи за произволен епсилон, който ползваме. Понякога задачите не разрешават толкова малък вход или биха зачели намерения с тази модификация отговор за верен, но за съжаление не всички са такива, и, съответно, това не винаги работи.

За наша радост има лесно решение, което се справя и с двата проблема (и, на всичкото отгоре, ни дава допълнителна сигурност). Знаем, че двоичното търсене работи за логаритъм на брой стъпки. Защо, тогава, да не направим логаритъм на брой стъпки? Заместваме while(left <= right) цикъла с for(int iter = 0; iter < 100; iter++) такъв и сме готови. Всичко останало вътре в него остава същото. Накрая отговорът ни ще е *left и right*, едновременно :) Това е така, тъй като след сто итерации техните стойности ще са толкова близки една до друга, че на практика няма да има разлика. Допълнителното предимство е, че сме *сигурни* колко итерации точно ще бъдат направени, като така можем да апроксимираме точно необходимото време за изпълнение на програмата ни.

Така кодът, който бихме ползвали за горната задача, би могъл да бъде:

```
double cubeRoot(double num) {
    double left = 0, right = max(num, 1.0);
    for (int iter = 0; iter < 100; iter++) {
        double mid = (left + right) / 2.0;
        if (mid * mid * mid < num)
            left = mid;
        else
            right = mid;
    }
    return right;
}
```

Още примери

Рандом генератор чрез монета

Имате на разположение генератор на случайни булеви величини (true, false), или с други думи - монета. Измислете начин, по който можете да генерира случайни реални числа в интервала [0, 1]. Докажете, че генерираните числа са равномерно разпределени.

Правим "двоично търсене" на числото. На всяка стъпка разделяме останалия ни интервал на две равни части, и в зависимост от това дали се падне ези или тура запазваме само лявата или само дясната част. Така, след определен брой хвърляния, останалият интервал ще е толкова малък, че можем да го счетем за едно единствено число.

Нека, например, при ези взимаме лявата страна, а при тура взимаме дясната. Нека също така са се паднали в този ред: ези, тура, ези, ези, тура, ези, тура, тура, тура, ези, тура. Нашият алгоритъм би стеснявал интервала по следния начин: [0, 1] -> [0, 0.5] -> [0.25,

0.5] -> [0.25, 0.375] -> [0.25, 0.3125] -> [0.28125, 0.3125] -> [0.296875, 0.3125] -> [0.3046875, 0.3125] -> [0.3046875, 0.30859375] -> [0.306640625, 0.30859375]. В крайна сметка можем да вземем просто средата на останалия интервал, в случая 0.3076171875 и да кажем, че това е нашето рандом число. Разбира се, прилагайки повече итерации бихме постигнали значително по-голяма точност.

Този метод можем да го разгледаме и по следния начин. В началото образуваме редицата 0.5, 0.25, 0.125, 0.0625, 0.03125, … За всеки член от тази редица хвърляме монетата, и ако се падне тура, го добавяме в сумата (която първоначално е била нула). Ако тази редица е безкрайна, резултатното число би било напълно случайно, равномерно разпределено в интервала [0, 1].

Липсващо число в сортиран масив

Даден ви е сортиран масив с N числа между 0 и N, включително, без повтарящи се числа, сортирани в нарастващ ред. Как бихте намерили лиспващото число?

Тази задача почти крещи "binary search, binary search". Винаги, когато имате нещо сортирано, погледнете защо ви го дават сортирано. Една от възможностите е да приложите двоично търсене (както е и в случая). На всяка стъпка от двоичното търсене ще проверявате дали на m-та позиция (където m е средата на текущо-разглеждания интервал) стои числото m. Ако не, то липсващото число е или на тази позиция, или наляво. Ако ли пък е, то значи липсващото число е на индекс, по-голям от m.

Прасета и отрова

Готвачът на краля приготви 1000 гозби за неговата сватба! Но кралските шпиони му съобщиха, че някой е сипал отрова в точно една от гозбите му. Отровата започва да се забелязва 2 часа след поемане на храната, а до сватбения пир остават... малко повече от два часа - тоест има време точно за една проба. За щастие, готвачът има на разположение известен брой прасета, върху които може да експериментира. На всяко от тях той може да забърка смесица от една или повече от гозбите и му я даде да я изяде. Така ако до два часа прасето умре, то в някоя от гозбите, които е изяло, е имало отрова. За да не става свинщина, той иска да ползва възможно най-малко на брой прасета. Колко е минималният такъв брой, който позволява да се определи в коя гозба е отровата в рамките на тези 2 часа?

Примерно решение е да даде на 1000 прасета по точно една от гозбите, но има решение с много по-малко на брой. Забележете, че той гледа да минимизира не броя умрели прасета, а броя ползвани такива.

Тази задача е значително по-хитра и нетривиална. Отговорът е 10 прасета (двоичен логаритъм от 1000). Тук по-скоро ще ползваме *идеята* на двоичното търсене, а не самия алгоритъм.

Готвачът ще даде на първото прасе от всяка от пърата половина от гозбите. На второто прасе ще даде от всяка от първата и третата четвъртина от гозбите. На третото ще даде

от 1-вата, 3-тата, 5-тата и 7мата осмина от гозбите и т.н.

За да покажем по-нагледно какво правим, нека считаме, че броят гозби е само 32 (но същата идея работи за произволен брой). Тогава (представено чрез битови маски, 1 за "даваме", 0 за "не даваме"), всяко от 5-те ни нужни прасета ще получи:

```
Прасе 1: 11111111111111110000000000000000
Прасе 2: 11111111000000001111111100000000
Прасе 3: 11110000111100001111000011110000
Прасе 4: 11001100110011001100110011001100
Прасе 5: 10101010101010101010101010101010
```

Ако първото прасе е умряло, то отровата е сред някоя от първите 16 гозби. В противен случай, тя е сред някоя от останалите 16. Тоест можем да определим в коя половина е отровата. В зависимост от това, дали второто прасе е умряло, можем да определим в коя четвъртина е отровата. Третото прасе определя осмината и т.н, като последното прасе еднозначно определя отровната гозба.

За малко по-голям пример, който не е точна степен на две, можете да разгледате този файл. Тук сме разпределили сто гозби между седем прасетата.

Намиране на корен на уравнение

Дадено ви е уравнение от типа A*X^5 + B*X^4 + C*X^3 + D*X^2 + E*X + F = 0, където *A, B, C, D, E*, и *F* са ненулеви, цели числа, с абсолютна стойност по-малка или равна на 100. Намерете стойност на *X*, която го удовлетворява (тоест намерете някой от *корените* на уравнението). Например, възможно решение за 3*X^5 - 13*X^4 - 42*X^3 + X^2 + X + 42 = 0 би било *-2.297692476*.

Тази задача е по-сложен вариант на примера, който разгледахме за недискретен интервал.

Основното наблюдение, което ще ползваме за да решим задачата, е, че лявата част от уравнението ще има различни знаци, когато *X* клони към минус безкрайност и когато клони към плюс безкрайност. Това е вярно, защото коефициентите са *ненулеви* и съответно най-старшият член A*X^5 променя знака си при отрицателен и положителен *X*. Следователно, някъде функцията приема стойност 0 (където и сменя знака си). С двоично търсене ще намерим тази точка.

Първо, нека проверим дали имаме монотонност. В минус безкрайност функцията приема отрицателна стойност. В нула функцията приема положителна стойност (42). В едно функцията приема отрицателна стойност (-8). В плюс безкрайност функцията приема положителна стойност. Ъъъъ. Отрицателна, положителна, отрицателна, положителна. Очевидно не е монотонна. И все пак в случая можем да ползваме двоично търсене. Тъй като търсим *който и да е* от отговорите, а във всеки интервал, в който функцията е с различни знаци в двата му края, има поне един корен. Следователно, рано или късно ще стигнем до подинтервал, в който функцията е

монотонна.

Следва да намерим интервала, в който ще се намират отговорите (или поне един от тях). Тъй като коефициентите на полинома са сравнително малки и при това цели, то можем да докажем, че тя ще има различен знак при X = -1,000,000,000,000 и X = +1,000,000,000,000. Следователно, това са хубави стойности за лява и дясна граница.

Остава да направим двоично търсене в този интервал, като местим тази граница, в която функцията има същия знак като в средата на интервала. Не забравяйте да ползвате фиксиран брой итерации вместо стандартната while(left <= right) конструкция!

```
#include <cstdio>
const double INF = 1000000000000.0;

double eval(int A, int B, int C, int D, int E, int F, double X) {
    return ((((A * X + B) * X + C) * X + D) * X + E) * X + F;
}

int sign(double num) {
    return num < 0.0 ? -1 : 1;
}

double solve(int A, int B, int C, int D, int E, int F) {
    double left = -INF, right = INF;
    double ansLeft = eval(A, B, C, D, E, F, left);
    for (int iter = 0; iter < 100; iter++) {
        double mid = (left + right) / 2.0;
        double val = eval(A, B, C, D, E, F, mid);
        if (sign(val) == sign(ansLeft))
            left = mid;
        else
            right = mid;
    }
    return right;
}

int main(void) {

    int A = 3, B = -13, C = -42, D = 1, E = 1, F = 42;
    fprintf(stdout, "%+d*X^5 %+d*X^4 %+d*X^3 %+d*X^2 %+d*X %+d has root %.9lf\n",
            A, B, C, D, E, F, solve(A, B, C, D, E, F));
    return 0;
}
```

Задачи

Като състезатели по информатика ще срещнете не малък брой задачи, които се решават (поне частично) с двоично търсене. Много често то ще е само малка част от задачата, но ще опростява остатъка от нея значително. Този метод е много важен и по друга причина - той е един от популярните въпроси на интервюта за работа. Редица софтуерни компании считат, че ако кандидатът за работно място не може да напише двоично търсене, то няма никакъв смисъл да го наемат като програмист.

За сега ще ви дадем само няколко задачи, върху които да тренирате. По-нататък в тренировката ще срещнете редица други такива, които изискват двоично търсене като стъпка в решението си. Можете да погледнете: **Sysadmin**, **Exam**, **Graze**, **Riddles**,

Article, **Watering**.

Източници:

1. Александър Георгиев, Разделяй и владей, http://informatika.bg/lectures/divide-and-conquer
2. Александър Георгиев, Двоично търсене, http://informatika.bg/lectures/binary-search