**Бинарна претрага**

Бинарна претрага је један од основних алгоритама у рачунарству. Да бисмо је разумели, прво ћемо изградити теоријску основу, а затим њу употребити за правилну реализацију алгоритма и избегавајње уобичајених грешака.

* **Проналажење вредности у сортираном низу**

Најједноставнији облик бинарне претраге користи се за брзо проналажење дате вредности у сортираном низу. Тражену вредност назовимо *циљна вредност* због лакшег споразумевања. Бинарна претрага оперише у ограниченом поднизу почетног низа, у коме се циљна вредност сигурно налази. Назовимо га *простор претраге*. Простор претраге је првобитно цео низ. У сваком кораку, алгоритам упоређује вредност медиана простора претраге са циљном вредношћу. На основу поређења и зато што је низ сортиран, простор претраге се може преполовити. Понављајући шроцес више пута, простор претраге ће се на крају састојати од једног елемента, циљне вредности.

На пример, размотримо следећи низ бројева у растућем поретку, на пример, у потрази за бројем 55:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 5 | 13 | 19 | 22 | 41 | 55 | 68 | 72 | 81 | 98 |

Нама се тражи локација циљне вредности у низу, тако да нам простор претраге представљају бројеви одређени индексима. У почетном тренутку, простор претраге садржи бројеве са индексима од 1 до 11. Пошто је простор претраге заиста интервал, довољно је памтити само два броја, најмањи и највећи индекс. Као што смо већ напоменули, сада бирамо медијан низа, што је вредност члана са индексом 6 (средина између 1 и 11): односно 41, и он је мањи од циљне вредности. Из овога можемо закључити не само да елемент са индексом 6 није циљна вредност, већ и да ниједан од елемената са индексима од 1 до 5 такође не може да буде циљна вредност, јер су сви они мањи од 41, што је мање од циљне вредности. Тиме сужавамо простор претраге на бројеве са индексима од 7 до 11:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 55 | 68 | 72 | 81 | 98 → | 55 | 68 |

Понављајући процес на исти начин низ ћемо преполовити још два пута. Сада, у зависности од тога како рачунмо медијану низа са парним бројем елемената, или ћемо у следећем кораку наићи на 55 или ћемо одсећи 66 и тако свели простор претраге на један елемент. У сваком случају закључујемо да је тражени индекс 7.

Када тражени број не би био присутан у датом низу, бинарна претрага би испразнила простор претраге. Овај услов се лако проверава и једноставан је за употребу.

|  |  |
| --- | --- |
| function binarna\_pretraga (a:niz; target:longint):longint;  var mid : longint;  begin  lo:=1; hi:=sizeof(a);  while lo<=hi do  begin  mid : = lo + (hi – lo) div 2;  if a[mid]=target then binarna\_pretraga : = mid  else if a[mid]<target then lo : = mid+1  else hi : = mid – 1;  //target was not found | binary\_search(a, target):  lo = 1, hi = size(a)  while lo <= hi:  mid = lo + (hi-lo)/2  if a[mid] == target:  return mid  else if a[mid] < target:  lo = mid+1  else:  hi = mid-1  // target was not found |

* **Сложеност**

Пошто при сваком поређењу бинарна претрага користи само половину простора претраге, можемо тврдити и лако доказати да бинарна претрага никада неће користити више од О (log N) поређења да би пронашла циљну вредност.

Логаритам је ужасно споро растућа функција. У случају да нисте свесни колико је бинарна претрага ефикасна, замислите да тражите име у телефонском именику који садржи милион имена. Бинарна претрага вам омогућава да систематски нађете било које име користећи највише 21 поређења. Ако бисте поседовали листу свих људи на свету сортираних по имену, могли бисте да пронађете било коју особу у мање од 35 корака. Ово можда тренутно не изгледа изводљиво или корисно, али ускоро ћемо то променити.

Имајте на уму да уколико имамо приступ неуређеном низу или повезаној листи нема смисла користити бинарну претрагу и да је, у таквим ситуацијама, боље користити обичану линеарну претрагу.

* **Бинарна претрага у стандардним библиотекама**

*Standard Template Library* у C++-у реализује бинарну претрагу у алгоритмима: *lower\_bound, upper\_bound, binary\_search* и *equal\_range*, у зависности од тога шта нам је тачно потребно да урадимо. Јава има уграђени *Arrays.binary\_search* метод за претрагу по низовима, а .NET Framework има *Array.BinarySearch*.

Најбоље би било да користите функције из библиотека када год сте у могућности, јер, као што ћете видети, креирање бинарне претраге може бити захтевно.

* **Изван низова: дискретна бинарна претрага**

Ово је место где ћемо почети да апстрактном претрагу бинарни. Секвенца (низ) је заиста само функција која сарадника целих (индекси) са одговарајућим вредностима. Међутим, не постоји разлог да се ограничи нашу коришћење бинарне претраге на опипљиве секвенци. У ствари, можемо користити исти алгоритам описано на било монотоне функције ф чији домен је скуп целих бројева. Једина разлика је у томе што смо заменили проналажење низа са функцијом евалуације: ми смо сада у потрази за неко к такав да је ф (к) је једнака циљне вредности. Потрага простор је сада формално субинтервал домена функције, а циљна вредност је елемент цодомаин. Моћ бинарног претраживања почиње да сада показују: не само да нам је потребна на већини О (лог Н) поређењима наћи циљну вредност, али такође не треба да процени функцију више него да је много пута. Поред тога, у овом случају није ограничена практичним количинама, као што су расположиве меморије, као што је био случај са низовима.  
Узимајући га даље: главни теорема  
Када наиђете на проблем који мислите да се реши применом бинарну претрагу, морате на неки начин доказивања да ће радити. Сада ће представити још један ниво апстракције, који ће нам омогућити да реши више проблема, да доказује бинарне претраге решења веома лако и такође помажу њихово спровођење. Овај део је тад формално, али не дајте обесхрабрити, није тако лоше.  
Размислите п предикатно дефинисан у неком наредио сета С (тражи простор). Тражи простор се састоји од кандидата решења проблема. У овом чланку, предикат је функција која враћа боолеан вредност, прави или лажни (такође ћемо искористити да и не, као боолеан вредности). Ми користимо предикат да провери да ли кандидат решење је правно (не крши неку препреку), према дефиницији проблема.  
Оно што можемо назвати главне државе теорема која бинарна претрага може да се користи ако и само ако за свако к у С, п (к) подразумева п (и) за све и> к. Ова имовина је оно што ми користимо када смо одбацили другу половину простору претраге. То је еквивалент на то да је ¬ п (к) подразумева ¬ п (и) за све и <к (симбол ¬ означава логичку не оператора), што је оно што ми користимо када смо одбацили прву половину простору претраге. Теорема се лако може доказати, иако ћу прескочити доказ овде да смање гужву.  
Иза криптичне математике сам заиста наводећи да, ако сте имали да или не питање (предикат), добијање иес одговор за неких потенцијалних решења к значи да можете да добијете одговор иес за било који елемент након к. Слично томе, ако имаш без одговора, да ћеш добити одговор за било који елемент пре к. Као последица тога, ако сте били да поставим питање за сваки елемент у простору претраге (у реду), да би добили низ без одговора праћен серијом Да одговора.  
Пажљиви читаоци могу приметити да бинарна претрага може да се користи када предикат даје низ одговора ДА затим серијом без одговора. То је истина и допуна тог предикат ће задовољити првобитно стање. Због једноставности ћемо се бавити само предиката описаним у теореме.  
Уколико стање у главном теорема задовољан, можемо користити бинарни претрагу да пронађете најмању законско решење, односно најмањи к за које је П (к) је истина. Први део разрађује решење засновано на бинарном претраге је дизајнирање предикат који се може оцењивати и за које то има смисла користити бинарну претрагу: ми треба да изабере оно што је алгоритам треба пронаћи. Можемо да га нађемо било први к за које је П (к) је истинита или последњи к за које је П (к) је лажна. Разлика између њих је само мала, као што ћете видети, али је неопходно да се населе на један. За почетак, хајде да тражимо прво одговорите (прва опција).  
Други део се докаже да бинарна претрага може да се примени на предиката. То је место где ми користимо главни теореме, проверава да ли су испуњени услови наведени у теореме. Доказ не треба да буде превише математичка, ви само треба да се увери да је п (к) подразумева п (и) за све и> к или да ¬ п (к) подразумева ¬ п (и) за свако и <к . То често може да се уради применом здравог разума у ​​реченици или две.  
Када домен предиката су цели бројеви, довољно је доказати да је п (к) подразумева П (к +1) или да ¬ п (к) подразумева ¬ п (к-1), затим следи остатак индукцијом.  
Ова два дела су најчешће прошаране: када мислимо проблем може решити бинарном претрагом, циљ нам је да дизајнира предикат тако да задовољава услов у главном теореме.  
Могло би се запитати зашто смо изабрали да користите ову апстракцију уместо једноставнији изглед алгоритма користили смо тако далеко. То је зато што многи проблеми не могу бити моделиран као потрази за одређену вредност, али је могуће да се дефинишу и вредновати предикат као што је "Да ли постоји задатак који кошта Кс или мање?", Када смо у потрази за неку врсту Задатак са најнижим трошковима. На пример, уобичајени трговачки путник проблем (ТСП) изгледа за најјефтинији тура које посећује сваки град тачно једном. Ево, циљна вредност није дефинисана као таква, али можемо дефинисати предикат "Да ли постоји повратна кошта Кс или мање?" а затим применити бинарну претрагу да пронађете најмању к која задовољава предикат. То се зове смањење оригинални проблем са одлуком (да / не) проблем. Нажалост, знамо ни начин ефикасно оцењивање овај посебан предикат па ТСП проблем није лако решити бинарном претрагом, али многи проблеми оптимизације.  
Хајде сада конвертовали једноставну претрагу на бинарну разврстаних низовима описане у уводу овог апстрактног дефиницији. Прво, хајде да преформулишем проблем као: "Имајући у виду низ и циљна вредност, врати индекс првог елемента једнака или већа од циљне вредности." Узгред, ово је више или мање како ЛОВЕР\_БОУНД понаша у Ц + +.  
Желимо да нађемо индекс циљне вредности, чиме свака индекс у низу је кандидат решење. Тражи простор С је скуп свих кандидата решења, чиме интервала који садржи све индексе. Размислите предикат "Да ли је [к] већа или једнака циљне вредности?". Ако смо да пронађемо први к за које предикат каже да, ми бисмо добили оно што смо одлучили тражили у претходном пасусу.

Стање у главном теорема је задовољан због тога што низ је сортиран у растућем редоследу: ако [к] је већа или једнака циљне вредности, сви елементи након што су сигурно и већа или једнака циљне вредности.

Ако узмемо секвенцу узорак од раније:  
0 5 13 19 22 41 55 68 72 81 98  
Са тражи простор (индекси):  
1 2 3 4 5 6 7 8 9 10 11  
И примењују нашу предикат (са циљном вредношћу 55) на њега добијамо:  
не не не не не не да да да да да  
Ово је серија без одговора затим низ Да одговора, као што смо очекивали. Обратите пажњу како индекс 7 (где се налази циљна вредност) је први за који предикатна приноси, да, па то је оно што наша потрага бинарни наћи.

Примена дискретног алгоритма  
Једна важна ствар коју треба запамтити пре него што почињу да код да се населе на шта су два броја ти одржавамо (доња и горња граница) значе. Вероватно је одговор затворен интервал који сигурно садржи прве к за које је П (к) је истина. Све кода требало би бити усмерене на одржавање ове инваријанта: то вам говори како да правилно померање границе, где се грешка може лако наћи свој пут у вашем коду, ако нисте пажљиви.  
Још једна ствар коју треба бити опрезан са је колико високо да поставите границе. Под "високе" Ја заиста мислим "широка", јер постоје две границе да бринете о томе. Сваки тако често се дешава да кодер закључује током кодирања да границе он или она поставили су довољно широка, само да пронађе цоунтерекампле током паузе (када је касно). Нажалост, мало помоћи савет може се дати овде осим да увек двоструко и троструко проверите своје границе! Такође, од времена извршења повећава логаритамски са границама, увек можете да их поставите већи, док се не разбије евалуацију предиката. Пази се за преливне грешке свуда, а посебно у обрачуну медијану.  
  
Сада смо коначно доћи до кода који имплементира бинарну претрагу као што је описано у овом и претходном одељку:

binary\_search(lo, hi, p):

while lo < hi:

mid = lo + (hi-lo)/2

if p(mid) == true:

hi = mid

else:

lo = mid+1

if p(lo) == false:

complain // p(x) is false for all x in S!

return lo // lo is the least x for which p(x) is true

Две су кључне линије су хи = средине и гле = средином +1. Када је п (средина) је истина, можемо одбацити другу половину претрагу простора, јер предикат важи за све елементе у њој (поред главног теореме). Међутим, не можемо да одбацимо средине себи, јер то може бити први елемент за који је п тачно. То је разлог зашто се креће горња граница до срeдине је као агресивна као што можемо урадити без увођења грешке.  
На сличан начин, ако је п (средина) је лажна, можемо одбацити прву половину претрагу простора, али овај пут, укључујући средином. п (средина) је нетачно тако да ми не треба у нашем простору претраге. То практично значи да можемо кренути доња граница до средине +1.  
  
Ако бисмо желели да нађемо последњи к за које је П (к) је лажан, ми би осмислити (користећи сличну логику као горе) нешто као:

// warning: there is a nasty bug in this snippet!

binary\_search(lo, hi, p):

while lo < hi:

mid = lo + (hi-lo)/2 // note: division truncates

if p(mid) == true:

hi = mid-1

else:

lo = mid

if p(lo) == true:

complain // p(x) is true for all x in S!

return lo // lo is the greatest x for which p(x) is false

Можете да проверите да ли је овај задовољава наше услов да елеменат тражимо увек бити присутан у интервалу (гле, хи). Међутим, постоји још један проблем. Размислите шта се дешава када покренете овај код на неком простору претраге за које предикат даје:  
не да  
Код ће се заглави у петљи. Она ће увек изабрати први елемент средином, али онда неће померити доња граница јер жели да задржи у свом бр простору претраге. Решење је да се промени средина = ло + (хи-ло) / 2 до средине = ло + (Хи-Ло +1) / 2, односно тако да се заокружује се уместо доле. Постоје и други начини за добијање око проблема, али ово је вероватно најчистији. Само запамтите да увек тестирати свој код на два елемента скупа где предикат је лажна за први елемент, а важи и за други.  
Такође можете запитати зашто је средином се израчунава МИД = гле + (хи-ло) / 2 уместо уобичајеног средине = (ло + хи) / 2. Ово је да се избегне још један потенцијални буг заокруживања: у првом случају, желимо да увек подела око доле, ка доња граница. Али подела Скраћује, па кад гле + хи бити негативан, он ће почети заокруживање ка више везан. Цодинг обрачун на овај начин обезбеђује да број подељен је увек позитивна и тиме увек рунди јер желимо да. Иако буг не изронити када тражи простор се састоји само од бројева или реалних бројева, одлучио сам да га кодирати на овај начин током чланку за доследности.  
  
Реал бројеви  
Бинарна претрага може се користити и на монотона функција чији домен је скуп реалних бројева. Имплементација бинарну претрагу на реалних бројева је обично лакше него на целе бројеве, јер не морате да пазите како се крећу границе:

binary\_search(lo, hi, p):

while we choose not to terminate:

mid = lo + (hi-lo)/2

if p(mid) == true:

hi = mid

else:

lo = mid

return lo // lo is close to the border between *no* and *yes*

Пошто скуп реалних бројева је густа, требало би да буде јасно да нећемо обично моћи да пронађете тачан циљну вредност. Међутим, можемо брзо пронаћи неки к такав да је ф (к) у некој толеранције границе између не и да. Имамо два начина одлучивања када престаје: прекинути када тражи простор постаје све мања него неки предодређени сигурно (рецимо 10-12) или да фиксни број итерација. На ТопЦодер, најбоље је да користите само неколико стотина итерација, то ће вам дати најбољи могући прецизност, без превише размишљања. 100 итерација ће смањити простор за претрагу на око 10-30 своје почетне величине, који би требало да буде довољно за већину (ако не и све) проблеме.  
Ако вам је потребно да урадите као пар итерација као могуће, можете прекинути када интервала добија мали, али покушајте да релативни поређење граница, не само апсолутни један. Разлог за ово је да дубл никада не могу вам дати више од 15 децималних цифара прецизности тако да ако тражи простор садржи велике бројеве (рецимо по налогу милијардама), никада не можете добити апсолутну разлику од мање од 10-7.  
  
Пример  
У овом тренутку ћу показати како се све ово говори да се користи за решавање ТопЦодер проблем. За ово сам изабрао умерено тежак проблем, ФаирВорклоад, што је подела 1 ниво 2 проблем у СРМ 169.  
У проблему, број радника треба да испита број подношења кабинета. Кабинети нису све исте величине, а ми смо рекли за сваки кабинет колико фолдери она садржи. Од нас се тражи да се пронађе такав задатак да сваки радник добије секвенцијални низ кабинета да прођу и да минимизира максимални износ фасциклама које би радник морати да погледате.  
Након упознавања са проблемом, додир креативности је потребно. Замислите да имамо неограничен број радника на располагању. Кључна примедба је да, за одређени број МАКС, можемо израчунати минимални број радника потребних тако да сваки радник мора да испита не више од МАКС фолдера (ако је могуће). Хајде да видимо како ћемо то урадити. Неки радник треба да испита прву владу па смо доделили раднику било до њега. Али, пошто су кабинети морају бити додељена у секвенцијалном редоследу (радник не може да испита кабинете 1 и 3, без разматрања 2 као добро), увек је оптимално да га распореди на други кабинет, као и, ако то не да га преузме границе ми уведен (МАКС). Ако би га преко границе, можемо закључити да је његов посао је завршен и доделите новог радника до другог кабинета. Ми смо наставили на сличан начин све док сви кабинети су додељена и тврде да смо користили минимални број радника је могуће, са вештачким лимита смо увели. Обратите пажњу да број радника је обрнуто пропорционална МАКС већа поставили смо наш лимит, мање радника ћемо требају.  
Сада, ако се вратимо и пажљиво испитати шта смо тражили у изјави проблема, можете видети да смо заиста тражили најмањи МАКС такве да број радника потребних је мања или једнака броју радника на располагању . Имајући то у виду, ми смо скоро готови, само треба повезати тачке и видети како се све ово уклапа у кадру смо постављен за решавање проблема помоћу бинарне претраге.  
Са проблемом преформулисана тако да боље одговарају нашим потребама, сада можемо испитати предикат Може се шири обим посла, тако да сваки радник мора да испита не више од к фолдера, са ограниченим бројем радника на располагању? Можемо користити описани похлепног алгоритам за ефикасно оценити овај предикат за било к. Ово закључује први део изградње бинарну претрагу решење, ми смо тек сада треба да докаже да је стање у главном теорема је задовољан. Али приметити да се повећањем к заправо опушта лимит на максималном оптерећењу, тако да можемо само да треба исти број радника или мање, не више. Дакле, ако предикат казе да за неко к, такође ће рећи да за све већим к.  
  
Да га завршити, ево СТЛ погон фрагмент који решава проблем:

int getMostWork( vector folders, int workers ) {

int n = folders.size();

int lo = \*max\_element( folders.begin(), folders.end() );

int hi = accumulate( folders.begin(), folders.end(), 0 );

while ( lo < hi ) {

int x = lo + (hi-lo)/2;

int required = 1, current\_load = 0;

for ( int i=0; i<n; ++i ) {

if ( current\_load + folders[i] <= x ) {

// the current worker can handle it

current\_load += folders[i];

}

else {

// assign next worker

++required;

current\_load = folders[i];

}

}

if ( required <= workers )

hi = x;

else

lo = x+1;

}

return lo;

}

Обратите пажњу на пажљиво одабране доње и горње границе: можете да замените горње границе са било довољно великим цео број, али доња граница не сме да буде мања од највеће кабинета да се избегне ситуација у којој један кабинет ће бити превелика за било радника, случај који неће бити исправно руковати предиката. Алтернатива би била да подесите доња граница на нулу, а онда обради превише мали да к као посебан случај у предиката.  
Да бисте проверили да решење не закључава се, користио сам малу НЕ / ДА пример са фасциклама = {1,1} и радника = 1.  
Укупна комплексност решења је О (н лог ВЕЛИЧИНА), где величина је величина простору претраге. Ово је веома брз.  
Као што видите, ми смо користили похлепног алгоритма за процену предикат. У другим проблемима, процене предикат може доћи до било чега из једноставног математичког израза проналажењу максималну кардиналност подудара у бипартитног графикону.  
  
Закључак  
Ако сте довде, без одустајања, требало би да будете спремни да реше све што се може решити са бинарном претрагом. Покушајте да задржите неке ствари на уму:  
• Дизајнирање предикат који се може ефикасно проценити и да бинарна претрага може да се примени  
• Одлучите шта тражите и код тако да тражи простор увек садржи тога (ако постоји)  
• Ако претрага простор се састоји само од бројева, тестирали алгоритам на два елемента-постављен да би били сигурни да не закључа  
• Проверите да ли су доње и горње границе не претерано ограничен: обично је боље да их опусти док се не разбије предикат