Друштво математичара Србије
Адреса: Кнез Михаилова 35/4,
11000 Београд
сајт: www.dms.rs
е-маил: pom@dms.rs

Решење проблема Б за Децембар 2011
Бинарна претрага

Задаци овог типа, где је дат опис неког алгоритма и потребно је испитати његово понашање за дати улаз, су нестандардни за Б категорију. Међутим, ми смо се ипак одлучили да дамо овакав проблем, како би се ученици ухватили у коштац и са овим типом задатка и увидели да он није ништа компликованији од стандардних типова. У овом конкретном проблему описан је алгоритам бинарне претраге. Као што је и у тексту проблема напоменуто, познавање самог алгоритма није потребно, довољно је анализирати описани псеудо код. Но, препоручујемо читаоцу да се мало више упозна са овом методом уколико то до сада није учинио (ово је јако познат и користан алгоритам). Примљена решања су базирана на истој почетној идеји идеји. Прво ћемо изнети главну идеју, а касније и две њене могуће имплементације. Друго решење нам је послао ученик Математичке гимназије из Београда, Никола Јовановић.

Главна идеја: За овакве типове проблема, уколико немате идеју, најбоље је симулирати сам алгоритам на неколико примера. У сваком кораку алгоритма се на случајан начин[a] бира елемент *pivot* из тренутног низа. Уколико је његова вредност једнака траженој вредности v, алгоритам враћа *true* и завршава. У супротном, у зависности од тога да ли је његова вредност већа или мања од v из низа избацујемо све елементе пре или после пивота. Одавде видимо да у свакој итерацији алгоритма имам подниз почетног низа a.

Алгоритам никада неће вратити *true* за вредност који се не налази у низу[b]. Зато ћемо испитивање усмерити на вредности из самог низа: које елементе можемо увек пронаћи а које не? Претпоставимо да претражујемо вредност v која се заиста налази у нашем низу a. Покушајмо да нађемо карактеризацију низа, односно, коју особину он треба да има, која би нас спречила да увек пронађемо вредност v у њему. Вредност v нећемо наћи, уколико за неки избор *pivota*-а избацимо баш онај део низа (префикс или суфикс) у коме се он налази. Ово се може десити у два случаја:

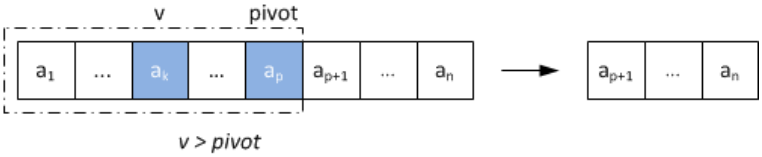
- уколико је $pivot < v$ и вредност v се налази пре *pivot*-а у низу a (у алгоритму се овде претпоставља да се вредност v налази у "левом" делу низа)

[a] Бирање елемента на случајан начин подразумева да сваки елемент има исту вероватноћу да буде изабран. У нашем случају вредност вероватноће није битна, битно је да за сваки елемент постоји позитивна вероватноћа да буде изабран.

[b] Овакав тип грешке, уколико би постојао, се назива *false positive*.

- уколико је $pivot > v$ и вредност v се налази после $pivot$-а у низу a (у алгоритму се овде претпоставља да се вредност v налази у "десном" делу низа)

Како се $pivot$ бира на случајан начин, довољно је да за дати елемент који претражујемо $a[k] = v$ постоји неки елемент $a[p] = pivot$ за који је $k < p$ (v је пре $pivot$-а у низу) и $v > pivot$ или $k > p$ (v је после $pivot$-а у низу) и $v < pivot$, како би v избацили из низа. Заиста, уколико је $k < p$ и $a[k] = v > a[p]$ избором $pivot = a[p]$ избацујемо подниз пре $pivot$-а коме припада тражена вредност v. Аналогно и за други случај.



Слика 1. Пример једног случаја избацивања тражене вредности v из низа.

Коначно, добијамо следећи закључак: вредност $a[k] = v$ можемо увек наћи описаним алгоритмом у низу a једино ако је већи од свих елемената пре њега у низу[c] и ако је мањи од свих елемената после њега. Другим речима, не сме постојати елемент $a[p]$ са горе описаним условима који га може избацити. Овим смо у потпуности описали алгоритам.

Input: низ a дужине n
Output: број елемената који се могу пронаћи алгоритмом бинарне претраге

```
1 toReturn = 0;
2 for k ← 1 to n do
3 |   if (a[k] је већи од a[1], ..., a[k − 1] и мањи од a[k + 1], ..., a[n]) then
4 |   |   toReturn = toReturn + 1;
5 |   end
6 end
7 return toReturn
```

Алгоритам 1: Тривијални приступ, где за сваки елемент $a[k]$, $k \in [1, n]$, испитујемо да ли је већи од свих елемената пре њега односно мањи до свих после њега, неће задовољавати временско ограничење (сложеност те имплементације је квадратна, а та сложеност је превелика за наше ограничење $n \leq 10^5$ много). Зато покушајмо да услов описан у алгоритму рачунамо мало "паметније".

Испитивање за елемент $a[k]$ делимо у два независна услова која испитујемо аналогно: да је већи од свих "лево" и мањи од свих "десно". Услов да је $a[k]$ већи од $a[k-1], \ldots, a[1]$ можемо записати елегантније као

$$a[k] > \max\{a[k-1], \ldots, a[1]\} := d[k-1]$$

Уколико би са $d[k]$ означили максимум првих k елемената, за први услов би било довољно испитати да ли је $a[k]$ веће од $d[k-1]$. Поставља се питање да ли низ d можемо иницијализовати брже од сложености тривијалног решења? Да, ово можемо урадити уколико

[c] Под "пре" и "после" елемента у низу подразумевамо елементе са мањим односно већим индексима.

искористимо везу између елемената $d[k]$ и $d[k-1]$. Конкретније имамо да важи:

$$d[k] = \max\{a[k], a[k-1], \ldots, a[1]\} = \max\{a[k], d[k-1]\} \ , \ k > 1$$

На овај начин, уз почетни услов да је $d[1] = a[1]$, низ d можемо иницијализовати у линеарном времену. Аналогно можемо посматрати и низ $q[k] = \min\{a[k], \ldots, a[n]\}$. Након овога, почетни услов можемо записати као

$$(a[k] > d[k-1]) \ and \ (a[k] < q[k+1])$$

Међутим овде имамо малу замку. Наиме за индексе $k = 1$ и $k = n$ су нам потребне вредности $d[0]$ и $q[n+1]$ које немамо дефинисане. Ово можемо решити на два начина:

- иницијализација ових вредности посебно (убацивање if услова за вредност индекса k)
- дефинисањем "вештачких" вредности $d[0] = 0$ и $q[n+1] = +\infty$ [d], чиме сигурно имамо да су услови $a[1] > d[0]$ и $a[n] < q[n+1]$ задовољени. Такође, горњу рекурентну везу можемо дефинисати (уз ове нове почетне услове) за вредности $k \in [1, n]$ (како сада постоји и $d[0]$ рекурентна веза важи и за $k = 1$).

  Додавање *"dummy"* вредности је стандарни трик када радимо са неким граничним условима. Оне не утичу на временску сложеност али нам могу доста поједноставити код. Ово посебно помаже када имамо вишедимензионалне низове (ово се посебно изражава у проблемима динамичког програмирања).

Временска сложеност, као и меморијска, ове имплементације је линеарна - $O(n)$.

Input: низ a дужине n
Output: број елемената који се могу пронаћи алгоритмом бинарне претраге

```
1  d[0] = 0;
2  for k ← 1 to n do
3  |   d[k] = max{a[k], d[k − 1]};
4  end
5  q[n + 1] = +∞;
6  for k ← n to 1 do
7  |   q[k] = min{a[k], q[k + 1]};
8  end
9  toReturn = 0;
10 for k ← 1 to n do
11 |   if (a[k] > d[k − 1]) and (a[k] < q[k + 1]) then
12 |   |   toReturn = toReturn + 1;
13 |   end
14 end
15 return toReturn
```

Напомена: У описаној имплементацији користили смо два додатна низа d и q. Међутим, описани алгоритам можемо имплементирати у линеарном времену уз помоћ само једног би-

[d] Под бесконачношћу подразумевамо било коју вредност која ће сигурно бити већа од свих елемената низа. У нашем случају за бесконачност можемо узети вредност $2 \cdot 10^9 + 1$.

нарног низа. Ова побољшања не утичу на временску сложеност алгоритма, али нам могу уштедети доста меморије.[e]

Дефинишимо са $mark[]$ бинарни низ, за који је $mark[k] = true$ уколико елемент $a[k]$ задовољава горње услове, $false$ иначе. На почетку иницијализујемо низ $mark$ на позитивне вредности (претпоставимо да сви задовољавају услове). Прво проверавамо само први услов, да је већи од претходних елемената, и мењамо вредност низа $mark$ уколико услов није испуњен. Како се крећемо од првог елемента у промењивој $currentMax$ чувамо тренутни максимум за елементе које смо обишли. Сада услов постаје $a[k] > currentMax$. Када пређемо на наредни елемент $a[k+1]$, вредност $currentMax$ постављамо на $\max\{a[k], currentMax\}$. Након овога пролазимо још једном кроз низ, сада од последњег ка првом елементу и, слично претходном, проверавамо за минимуме суфикса.

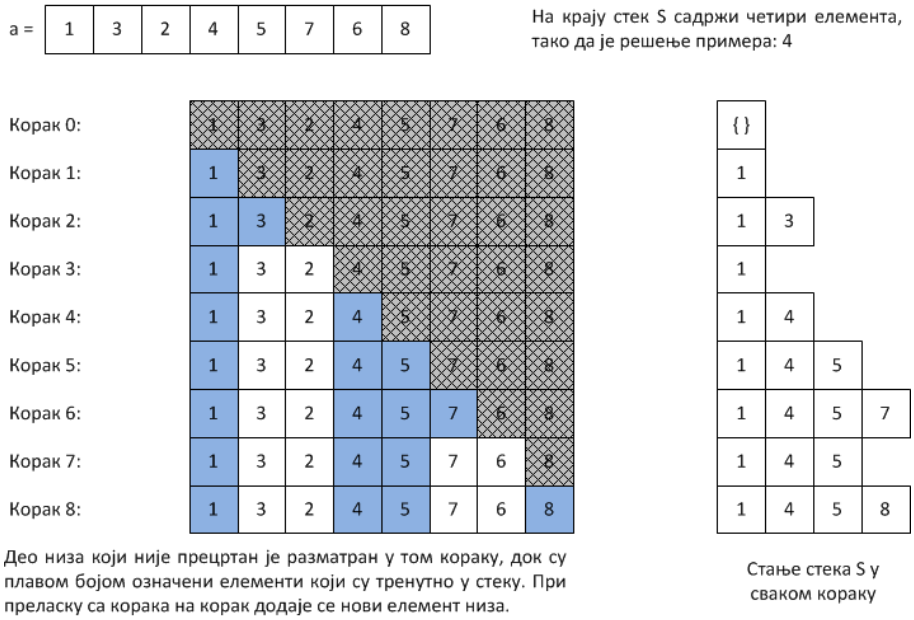
Алгоритам 2: Ово решење, које је такође базирано на почетној идеји, нам је послао ученик Никола Јовановић. У овој имплементацији ћемо користити појам стека (енг. *stack*). Стек представља структуру података која је базирана на $LIFO$ принципу: елемент који је задњи убачен у структуру је први који се избацује (енг. *last in first out*). Ово можемо имплементирати преко низа у коме додајемо елементе на крај низа, а са краја их и скидамо (леп пример ове структуре је држач за ПЕЗ бомбоне). Више о овој структури и њеним модификацијама можете прочитати у свакој озбиљнијој књизи из програмирања. Наравно, већина програмских језика има ову структуру имплементирану у стандардним библиотекама.

Означимо стек са S. На почетку ће бити празан и редом ћемо обилазити елементе низа и модификовати стек S (избацивати неке елементе и убацити нове). У сваком тренутку S ће садржати елементе који задовољавају тражене услове: када обрађујемо елемент $a[k]$ стек S садржи све елементе који ће сигурно бити нађени алгоритмом над поднизом $a[1], a[2], \ldots, a[k-1]$. Другим речима, када обрађујемо елемент $a[k]$, стек S садржи све елементе који су већи од претходних (све претходне смо већ обишли) а мањи од свих после њега до елемента $a[k-1]$. Дакле, разматрање новог елемента може довести до тога да неки елемент који је био у стеку више не задовољава други услов. Заиста, нови елемент $a[k]$ може бити мањи од њега. Сви елементи у стеку задовољавају први услов, а други услов само до елемента који се обрађује.

Стек S у сваком тренутку представља растући низ бројева. Уколико он не би био растући, тада би имали елемент који је мањи од неког који је испред њега, што би значило да он не задовољава тражене услове (па не сме бити у стеку S). Дакле, елемент $a[k]$ убацујемо у стек ако је већи од свих претходних, а избацујемо елементе ако су већи од тренутног елемента којег испитујемо. Овим смо добили два основна корака када испитујемо елемент $a[k]$:

- Са стека S скидамо све елементе који су већи од елемента $a[k]$. Пошто је стек S сортиран у растућем поретку, можемо стати са испитивањем чим наиђемо на први елемент који задовољава услов да је мањи од $a[k]$ (сви после њега су мањи од њега а самим тим и од $a[k]$). Овде је битно нагласити да уколико не би стали са испитивањем тј. уколико би у сваком кораку испитивали све елементе стека, добили би квадратну сложеност.
- На стек S убацујемо елемент $a[k]$ ако је већи од свих претходних. Ово можемо имплементирати тако што памтимо тренутни максимум и при преласку на наредни елемент ажурирамо ову вредност.

[e]У нашем случају, два низа дужине $n = 10^5$ заузимају $2 \cdot 4 \cdot 10^5$ бајтова, односно око 0.8 мегабајта, док један бинарни низ исте дужине заузима око 0.1 мегабајт.



Слика 2. Кораци Алгоритма 2 на датом примеру

На крају, стек S садржи управо елементе који задовољавају тражени услов. Тражено решење је број елемената стека. Како сваки елемент највише једном убацујемо и избацујемо из стека, сложеност овог алгоритма је линеарна, $O(n)$. Имплементација овог решења је једноставна и можете је наћи у пропратном материјалу.

Напомена: Проблем можемо урадити и на јако елегантан начин, али у сложености $O(n \log n)$ (која и даље задовољава наша ограничења из проблема). Уколико елемент $a[k]$ задовољава горње услове (односно увек се налази алгоритмом бинарне претраге) тада је он већи од свих претходних и мањи од свих наредних елемената. Он би се у сортираном низу налазио на позицији са индексом k: већи је од $(k-1)$-ог елемента низа а мањи од свих осталих. Проблем смо свели на број елемената низа чија се позиција поклапа са позицијом у сортираном низу (ово важи само у случају када су елементи низа различити). Одавде добијамо нови алгоритам: направимо копију b низа a, сортирамо је и вратимо број индекаса $k \in [1, n]$ за које је $a[k] = b[k]$.

Решење задатка припремио:
Андреја Илић,
Природно математички факултет, Ниш