Глава 3

Конструкција алгоритама рекурзијом тј. индукцијом

Један од основних механизама конструкције алгоритама подразумева да се до решења проблема долази тако што се проблем сведе на решавање једног или више потпроблема истог облика, али мање димензије. Свођење, наравно, не може да тече у недоглед, већ је потребно да проблеме мале димензије умемо да решимо директно, без даљег свођења. На пример, проблем димензије 0 се решава директно, док се за свако $n > 0$, проблем димензије n своди на проблем димензије $n - 1$.

Имплементација овог поступка може бити реализована на два начина.

- Могуће је дефинисати рекурзивну функцију (функције која позива саму себе), којој се преко улазних параметара (уз евентуално коришћење додатних, глобалних променљивих) прослеђује опис проблема који се тренутно решава. Унутар функције се врши анализа да ли је прослеђени проблем довољно мале димензије да би се могао директно решити или се његово решење добија тако што функција позове сама себе да реши један или више мањих потпроблема.
- Могуће је дефинисати итеративни поступак, који подразумева да се у петљи променљиве ажурирају, кренувши од решења проблема мале димензије па проширујући решење мало по мало, све док се не дође до решења проблема тражене димензије.

Без обзира на то да ли се користи рекурзивна или итеративна имплементација, у основни оваквих алгоритама лежи исти поступак и њихова коректност следи на основу приниципа математичке индукције. Случај који се директно решава представља базу индукције. Индуктивна је претпоставка да су потпроблеми коректно решени и на основу те индуктивне претпоставке се доказује да се полазни проблем коректно решава. Стога ћемо често говорити о **индуктивно-рекурзивној** конструкцији и често ћемо приказивати и рекурзивну и нерекурзивну имплементацију (до које ћемо долазити било директно, било ослобађањем рекурзије).

3.1 Извођење итеративних алгоритама из рекурзивних

Задатак: Грејов код

Грејов кôд реда n подразумева ређање свих n-тоцифрених бинарних записа тако да се свака два суседна записа разликују тачно у једном биту (при чему ово важи и за први и последњи запис, тако да се може сматрати да су сви записи поређани у круг).

Грејов кôд дужине 0 садржи само један елемент и то празну ниску. Грејов кôд дужине $n+1$ се може добити од кода дужине n тако што се испред сваког броја у коду дужине n допише цифра 0, затим се редослед елемената у коду дужине n обрне и на сваком броју се на почетак допише цифра 1 и два тако добијена низа бројева се споје. Нпр. Грејов кôд реда 2 је

```
00
01
11
10
```

На основу претходног поступка добијамо Грејов кôд реда 3.

```
0 00  |                 k
0 01  |                 0:  000
0 11  |                 1:  001
0 10  v                 2:  011
            tj.         3:  010
1 10  ^                 4:  110
1 11  |                 5:  111
1 01  |                 6:  101
1 00  |                 7:  100
```

Напиши програм који за дату дужину кода n и дату позицију k ($0 \leq k < 2^n$) одређује бинарни број који се налази на позицији k у коду дужине n.

Улаз: Са стандардног улаза се учитава дужина кода n ($1 \leq n \leq 32$) и позиција k ($0 \leq k < 2^n$).

Излаз: На стандардни излаз исписати тражени бинарни број.

Пример 1

Улаз	*Излаз*
3 2	011

Пример 2

Улаз	*Излаз*
30 999999999	100110010101111010110100000000

Решење

Дефинишимо функцију која одређује k-ти по реду запис Грејовог кода дужине 2^n (подразумеваћемо да је $0 \leq k < 2^n$). Њу је једноставно дефинисати рекурзивно.

Ако је n нула, резултат је празна ниска. У супротном треба израчунати неки елемент Грејовог кода дужине $n-1$ и затим га допунити слева нулом или јединицом. Треба разликовати случај елемената у првој и у другој половини листе кодова. Пошто укупно има 2^n кодова, елементи у првој половини су на позицијама $0 \leq k < 2^{n-1}$, док су елементи у другој половини на позицијама $2^{n-1} \leq k < 2^n$.

- Када је $0 \leq k < 2^{n-1}$, тада се враћа k-ти елемент Грејовог кода дужине 2^{n-1} допуњен почетном нулом.
- Када је $2^{n-1} \leq k < 2^n$ тада се враћа $2^n - 1 - k$-ти елемент Грејовог кода дужине 2^{n-1} допуњен почетном јединицом. Изразом $2^n - 1 - k$ се позиција k своди у распон $[0, 2^{n-1})$ и уједно се обрће редослед бројева. Наиме, операцијом $k - 2^{n-1}$ вршимо редукцију интервала $[2^{n-1}, 2^n)$ на интервал $[0, 2^{n-1})$. Генерално, приликом обртања редоследа елемената, свака позиција p у интервалу $[0, m)$ се пресликава у позицију $m-p-1$ (позиција 0 се слика у $m-1$, док се $m-1$ слика у 0). Стога се приликом обртања интервала $[0, 2^{n-1})$ позиција $k-2^{n-1}$ слика у $2^{n-1}-(k-2^{n-1})-1$, но то је једнако 2^n-1-k.

Израчунавање степена двојке најједноставније се врши битовским операцијама (при чему треба обратити пажњу на потенцијално прекорачење).

Резултат можемо представити у облику ниске карактера. Иако дописивање карактера на почетак ниске може бити неефикасна операција, с обзиром на то да су ниске са којима радимо прилично кратке (најдужа има 32 карактера), о том не морамо да бринемо.

```
string grej(unsigned n, unsigned k) {
   if (n == 0)
      return "";
   if (k < (1u << (n - 1)))
      return "0" + grej(n - 1, k);
   else
      return "1" + grej(n - 1, (1ul << n) - 1 - k);
}
```

Функцију можемо реализовати и итеративно. Током итерације у променљивој `rez` налазиће се префикс траженог броја дужине $n_0 - n$, где је n_0 почетна, а n тренутна дужина кода. У зависности од тога да ли је текућа позиција k испод или изнад средине, префикс ћемо проширивати (здесна) нулом или јединицом, и уместо рекурзивног позива вредности k и n ћемо ажурирати вредностима које би се наводиле у рекурзивном позиву.

```
string grej(unsigned n, unsigned k) {
   string rez = "";
```

```
  while (n > 0) {
    if (k < 1u << (n-1))
      rez = rez + "0";
    else {
      rez = rez + "1";
      k = (1ul << n) - 1 - k;
    }
    n--;
  }
  return rez;
}
```

Напоменимо и да се тражени број у Грејовом коду може израчунати и директно, ако се представи у облику неозначеног броја изразом `k ^ (k >> 1)` (због водећих нула дужина n тада није битна). Овим се врши ексклузивна дисјункција позиције k и броја добијеног шифтовањем битова те позиције удесно. Неозначени број можемо затим претворити у ниску карактера и издвојити њен суфикс дужине n.

```
string grej(unsigned n, unsigned k) {
  return bitset<32>(k ^ (k >> 1)).to_string().substr(32 - n, n);
}
```

Задатак: Звезда

Особа је *звезда* (енгл. *superstar*) у некој групи људи ако њу сви остали познају, а она не познаје никога. Написати програм који одређује да ли у датој групи људи постоји звезда и која је то особа. Иако је за учитавање података потребно време које квадратно зависи од броја присутних особа, алгоритам треба да ради у времену које линеарно зависи од броја особа.

Улаз: Са стандардног улаза се учитава број особа присутних на забави n ($1 \leq n \leq 100$), а затим матрица димензије $n \times n$ која на позицији (i, j) тј. у врсти i и колони j садржи 1 ако особа i познаје особу j тј. 0 ако је не познаје. На дијагонали матрице се налазе јединице (свака особа познаје саму себе).

Излаз: На стандардни излаз исписати редни број особе која је звезда, ако звезда постоји (особе се броје од 0 до $n-1$) или -1 ако звезда не постоји.

Пример

Улаз	*Излаз*

`3`
`1 1 0`
`0 1 0`
`1 1 1`
	`1`

Објашњење

Особа на позицији 1 је звезда, јер она не познаје ни особу на позицији 0, ни особу на позицији 2 (у врсти 1 су све нуле, осим на позицији 1), а њу познају и особа на позицији 0 и особа на позицији 2 (у колони 1 су све јединице).

Решење

Директан начин је да за сваку особу проверимо да ли задовољава услов звезде.

Анализа сложености. Сложеност најгорег случаја овог алгоритма је $O(n^2)$, мада се тај најгори случај ретко јавља, јер је за очекивати да ће се обе провере у случају када кандидат није звезда прекинути много пре него што се стигне до краја низа.

```
bool poznajeNekog(const vector<vector<bool>>& poznaje, int i) {
  for (int j = 0; j < poznaje[i].size(); j++) {
    if (i != j && poznaje[i][j])
      return true;
  }
  return false;
}
```

```
bool sviJePoznaju(const vector<vector<bool>>& poznaje, int i) {
  for (int j = 0; j < poznaje.size(); j++) {
    if (i != j && !poznaje[j][i])
      return false;
  }
  return true;
}

int zvezda(const vector<vector<bool>>& poznaje) {
  for (int i = 0; i < poznaje.size(); i++)
    if (!poznajeNekog(poznaje, i) && sviJePoznaju(poznaje, i))
      return i;
  return -1;
}
```

Побољшање можемо покушати индуктивно-рекурзивним приступом. Прва идеја је да проблем проналажења звезде у скупу од n особа сводимо на проблем проналажења звезде у скупу од $n1$ особа.

- База индукције је празан скуп особа у коме не постоји звезда.
- Претпоставимо као индуктивну хипотезу да умемо да израчунамо звезду у скупу од $n-1$ особа. Раздвојмо скуп од n особа на подскуп од почетних $n-1$ особа и последњу особу o_{n-1}. Звезда целог скупа може бити или звезда тог подскупа или особа o_{n-1} (јер ако је особа звезда неког скупа онда је она уједно и звезда сваког подскупа којем припада).
  1. Ако у подскупу од $n-1$ особа постоји звезда, да би она била звезда целог скупа потребно је да је познаје особа o_{n-1} и да она не познаје особу o_{n-1}, то се може проверити веома брзо и једноставно.
  2. У супротном (ако у подскупу не постоји звезда или ако звезда постоји, али она познаје особу $_{n-1}$ или особа o_{n-1} не познаје њу) морамо још испитати да ли је особа o_{n-1} звезда. То питање никако не зависи од тога да ли у подскупу постоји звезда и да би се то испитало потребно је посебно проверити да ли свих претходних $n-1$ особа познаје особу o_{n1} и да ли она не познаје никога од њих. Нажалост, овај услов не можемо ефикасно проверити.

Анализа сложености. Ако скуп од првих $n-1$ особа садржи звезду, у времену $O(1)$ можемо проверити и да ли скуп од n особа садржи звезду. Међутим, ако скуп од $n-1$ особа не садржи звезду, тада је потребно време $O(n)$ да би се испитало да ли је особа o_{n-1} звезда. Тај се случај може догађати већ од тренутка када се појаве две особе, па је једначина која описује време извршавања $T(n) = T(n-1) + O(n)$, $(1) = O(1)$, па је сложеност алгоритма $O(n^2)$.

```
int zvezda(const vector<vector<bool>>& poznaje, int n) {
  if (n == 0)
    return -1;
  int z = zvezda(poznaje, n-1);
  if (z != -1 && poznaje[n-1][z] && !poznaje[z][n-1])
    return z;
  if (!poznajeNekog(poznaje, n-1) && sviJePoznaju(poznaje, n-1))
    return n-1;
  return -1;
}

int zvezda(const vector<vector<bool>>& poznaje) {
  return zvezda(poznaje, poznaje.size());
}
```

Из претходне рекурзивне конструкције лако је елиминисати рекурзију, али алгоритам остаје неефикасан.

```
int zvezda(const vector<vector<bool>>& poznaje) {
  int z = -1;
  for (int i = 0; i < poznaje.size(); i++) {
    if (z == -1 || !poznaje[i][z] || poznaje[z][i]) {
      if (!poznajeNekog(poznaje, i) && sviJePoznaju(poznaje, i))
        z = i;
```

```
      else
        z = -1;
    }
  }
  return z;
}
```

Основна идеја ефикасног решења је да се проблем посматра “уназад”. Број особа које нису звезде је сигурно много већи од броја особа које јесу звезде, па је идентификовање не-звезде много једноставније од идентификовања звезде. Кључна идеја ове ефикасне индуктивне конструкције је да веома брзо и једноставно (само једним питањем) из сваког скупа можемо уклонити особу за коју знамо да није звезда и на тај начин смањити димензију проблема. Када у скупу остане само једна особа, она је једини кандидат да буде звезда полазног скупа (јер су све остале особе елиминисане на основу тога што смо утврдили да не могу бити звезде). За ту једину преосталу особу онда можемо директно испитати да ли је звезда или није. Елиминација не-звезде из скупа се може извршити крајње једноставно. Одаберемо произвољне две особе у скупу и питамо се да ли особа A зна особу B. Ако је одговор потврдан, онда особа A не може бити звезда (јер звезда никога не познаје). Ако је одговор одричан, онда особа B није звезда (јер звезду сви познају).

Анализа сложености. Алгоритам заснован на овом поступку задовољава једначину $T(n) = O(1)+T(n-1)$, чије је решење $O(n)$. После овога, преостаје још да се провери да ли је преостали једини кандидат стварно звезда. То је могуће урадити грубом силом за шта нам је довољно $2n-1$ питања, тако да је сложеност и ове фазе $O(n)$, па је укупна сложеност обе фазе $O(n)$. Ипак, ако се у обзир узме и фаза учитавања матрице, за шта је потребно $O(n^2)$ операција, види се да је укупна сложеност програма $O(n^2)$, тако да се, нажалост, ова дивна оптимизација неће осетити у укупном времену извршавања.

Остаје питање техничке реализације, односно питање како да чувамо скуп кандидата који нису још елиминисани и како да из њега бирамо две особе које ћемо поредити. Једна могућност би била да су све особе сложене на један стек. Поредимо две особе са врха стека и на стек враћамо само ону која није елиминисана њиховим поређењем. Поступак настављамо док стек не постане једночлан.

Ипак, биће приказано још једноставније решење које користи два показивача. Први показивач, i, показује на особу која је тренутни кандидат за звезду, тј. на прву особу у низу за коју још није установљено да није звезда. Други показивач, j, показује на особу за коју се проверава да ли је текући кандидат за звезду познаје, тј. на прву особу након позиције i за коју још није утврђено да није звезда. На почетку су показивачи `i` и `j` на суседним позицијама ($i = 0$, $j = 1$). Особе се обрађују слева на десно секвенцијално.

- Уколико особа o_i не познаје особу o_j, тада је o_i и даље кандидат, а o сигурно није звезда (јер сви морају да познају звезду), па се показивач помера на следећу особу.
- Уколико особа i познаје особу j, онда i није звезда (јер звезда не познаје никога), али j можда јесте, па, пошто између i и j нико није звезда, први следећи кандидат за звезду је текућа особа j и показивач i се помера на j, а j на прву следећу особу (јер особе иза j још нису анализиране, па се за њих не зна да ли су звезде).

```
int zvezda(const vector<vector<bool>>& poznaje) {
  int i = 0, j = 1;
  while (j < poznaje.size()) {
    if (poznaje[i][j])
      i = j;
    j++;
  }
  if (!poznajeNekog(poznaje, i) && sviJePoznaju(poznaje, i))
    return i;
  return -1;
}
```

Доказ коректности. Докажимо формално коректност претходног поступка.

Лема: Инваријанта претходне петље да ниједан елемент у интервалу $[0, i)$ и ниједан елемент у интервалу (i, j) не може бити звезда (при чему је $0 \leq i < j \leq n$).

- На почетку је $i = 0$ и $j = 1$, па су оба интервала празна, а услов важи (под претпоставком да је $n > 0$).
- Претпоставимо да услов важи при уласку у петљу.

- Ако особа i познаје особу j тада она не може бити звезда. Пошто на основу претпоставке знамо да звезде не могу бити ни особе $[0, i)$ као ни особе (i, j) и како је $i' = j$, знамо да након тела петље звезде сигурно нису ни особе у интервалу $[0, i')$. Пошто је $j' = j + 1$, у том случају је интервал (i', j') празан, па тривијално задовољава услов.
- Ако особа i не познаје особу j тада знамо да j не може да буде звезда. Тада је $i' = i$, а $j' = j+1$, па на основу претпоставке знамо да звезде не могу бити особе из интервала $[0, i')$, знамо и да не могу бити особе из интервала (i', j), а пошто ни j не може бити звезда, знамо да звезде не могу бити особе из интервала (i', j'). Однос између променљивих тривијално остаје очуван у оба случаја.

Теорема: Функција или исправно проналази звезду или враћа -1 ако звезда не постоји.

Када се петља заврши није $j < n$, па на основу инваријанте важи да је $j = n$. Тада знамо да звезде не могу бити особе из интервала $[0, i)$ као ни особе из интервала (i, n). Дакле, једини кандидат за звезду је особа i. За њу се експлицитно проверава тражени услов, тако да функција враћа коректну вредност.

Задатак: Апсолутни победник на гласању

Апсолутни победник избора је онај ко освоји бар један глас више од половине изашлих бирача. Ако су позната сви гласачки листићи, одреди да ли постоји апсолутни победник избора и који је то кандидат (нагласимо да је апсолутни победник, ако постоји, јединствен тј. да није могуће да постоје два различита апсолутна победника).

Улаз: Са стандардног улаза се уноси број гласача n, а затим и гласови (сваки глас представља шифру неког кандидата - цео број из интервала $[0, 10^9]$).

Излаз: На стандардни излаз исписати број победника ако постоји апсолутни победник, тј. `nema` у супротном.

Пример 1

Улаз
```
10
342 123 342 756 123 756 123 756 756 756
```
Излаз
```
nema
```

Пример 2

Улаз
```
13
342 123 342 756 123 756 123 756 756 756 342 756 756
```
Излаз
```
756
```

Решење

Овај проблем се у литератури назива и *majority voting*.

Пребројавање гласова за сваког кандидата

Очигледан начин да се реши задатак је да се преброје сви гласови за сваког кандидата и да се пронађе да ли је неки кандидат освојио више од пола гласова. Бројање можемо извршити помоћу асоцијативног низа. Слична техника је коришћена у задацима Фреквенција знака и Фреквенције речи. У језику C++ можемо употребити колекцију `unordered_map` засновану на хеширању или `map` засновану на балансираним бинарним стаблима.

Анализа сложености. Сложеност приступа заснованог на бројању гласова зависи од сложености уметања у структуру података која пресликава кандидате у њихове освојене бројеве гласова. Ако је та структура података заснована на хеширању, та сложеност може бити константна у просеку (иако је у најгорем случају линеарна), а ако је заснована на балансираним стабилима, она је $\log n$. Зато укупна сложеност варира од $O(n)$ до $O(n \log(n))$, уз коришћење $O(m)$ меморије потребне за чување асоцијативног низа, где је m број различитих кандидата (обратимо пажњу на то да оригинални низ гласова није потребно чувати).

```
int apsolutniPobednik(const vector<int>& glasovi) {
  // brojimo glasove za svakog kandidata
  unordered_map<int, int> broj_glasova;
  for (int glas : glasovi)
    broj_glasova[glas]++;

  // proveravao da li neki kandidat ima vise od n/2 glasova
  int pobednik = -1;
  for (auto it : broj_glasova)
```

```
    if (it.second > glasovi.size() / 2) {
      pobednik = it.first;
      break;
    }

  return pobednik;
}
```

Сортирање

Још један начин може бити заснован на учитавању свих гласова, његовом сортирању, и затим провери броја узастопних једнаких елемената (анализирањем серије узастопних елемената). Разни начини сортирања описани су у задатку Сортирање бројева, а одређивање најдуже серије узастопних елемената описано је у задатку Најдужа серија победа.

Анализа сложености. Сложеност овог приступа зависи од сложености сортирања и једнака је $O(n\log(n))$ ако се користи библиотечко сортирање. Одређивање најдуже серије једнаких елемената врши се у времену $O(n)$.

```
int apsolutniPobednik(const vector<int>& glasovi) {
  // ukupan broj glasova
  int n = glasovi.size();
  // pravimo sortiranu kopiju niza glasova
  auto glasovi_s = glasovi;
  // sortiramo glasove za sve kandidate
  sort(begin(glasovi_s), end(glasovi_s));

  // duzina tekuce serije jednakih elemenata
  // element glasovi[0] zapocinje prvu seriju
  int broj_glasova = 1;
  for (int i = 1; i <= n; i++)
    if (i == n || glasovi_s[i] != glasovi_s[i-1]) {
      // serija jednakih elemenata je upravo zavrsena
      if (broj_glasova > n/2) {
        // ako u upravo zavrsenoj seriji ima vise od n/2 jednakih,
        // onda je u njoj apsolutni pobednik
        return glasovi_s[i-1];
      }
      // element glasovi[i] je zapoceo novu seriju
      broj_glasova = 1;
    } else
      // element glasovi[i] je nastavio tekucu seriju
      broj_glasova++;

  // ne postoji pobednik
  return -1;
}
```

Средишњи елемент

Неколико алгоритама подразумевају да сачувамо низ гласова, да у првом пролазу на неки начин одредимо потенцијалног кандидата за апсолутног победника, а да у другом пролазу проверимо да ли је тај кандидат заиста освојио потребан број гласова, тако што избројимо његов број гласова и проверимо да ли је већи од $\frac{n}{2}$.

Једна могућност да нађемо кандидата за апсолутног победника је да одредимо средишњи елемент у низу (када се низ сортира, то је онај елемент који се налази на позицији $\lfloor\frac{n}{2}\rfloor$ или на позицији $\lceil\frac{n}{2}\rceil$.

На пример, ако има 7 гласова, занима нас елемент на позицији 3.

```
0 1 2 3 4 5 6
```

```
. . . x . . .
```

Ако има 8 гласова, занима нас елемент на позицији 4 (а могли бисмо посматрати и елемент на позицији 3).

```
0 1 2 3 4 5 6 7
. . . x x . . .
```

Наиме, ако постоји апсолутни победник није могуће да он не заузме то средишње место (у случају парне димензије низа, заузимаће оба средишња места). На пример, ако има 7 гласова, апсолутни победник мора да има бар 4 гласова и они ће у сортираном редоследу бити узастопни. Ако би почињали на позицији 0, сигурно би захватили и позицију 3, ако би се завршавали на позицији 6, опет би сигурно захватили позицију 3, а слично важи и за било коју могућност између те две граничне ситуације. Слично, ако на пример има 8 елемената, апсолутни победник има бар 5 гласова, што значи да обухвата и позицију 6 и позицију 7 (било да почиње на позицији 0, да се завршава на позицији 7 или у било ком случају између та два гранична).

Средишњи елемент можемо у језику C++ можемо одредити библиотечком функцијом `nth_element` (чија је сложеност линеарна). Ручну имплементацију је најбоље реализовати алгоритмом QuickSelect. Тај алгоритам је описан у задатку Збир k најбољих.

```
int apsolutniPobednik(const vector<int>& glasovi) {
  // posto se redosled elemenata niza menja, moramo da napravimo kopiju
  auto Glasovi = glasovi;
  int n = Glasovi.size();
  // odredjujemo sredisnji element u sortiranom nizu - ako postoji
  // apsolutni pobednik on sigurno zauzima i sredisnju poziciju
  nth_element(begin(Glasovi), begin(Glasovi) + n / 2, end(Glasovi));
  int kandidat = Glasovi[n / 2];

  // proveravamo da li je kandidat ostvari vise od pola glasova
  if (count(begin(Glasovi), end(Glasovi), kandidat) > n / 2)
    return kandidat;
  else
    return -1;
}
```

Бојер-Муров алгоритам

У наставку ћемо прказати сасвим елементаран и изразито елегантан алгоритам за решење овог проблема који су увели Бојер и Мур у раду *"A fast majority vote algorithm"* (интересантно, у оригиналном раду алгоритам је уведен у циљу приказа могућности аутоматске формалне верификације софтвера).

Покушајмо да решимо проблем индуктивно-рекурзивном конструкцијом. Претпоставимо да желимо да испитамо постојање апсолутног победника у скупу од n гласова. Формално, наравно, ради се о мултискупу (јер може да садрже поновљене елементе), али ћемо једноставности ради у наредном опису користити термин скуп. Питање је како проблем свести на проблем мање димензије. Класично свођење проблема са димензије n на димензију $n-1$ не доводи до решења, јер избацивање било ког појединачног гласа може да промени решење (јер ако апсолутни победник има за један глас више од свих осталих, након избацивања тог гласа он више неће бити апсолутни победник). Самим тим, ако као резултат рекурзивног позива добијемо информацију да мањи скуп нема апсолутног победника, то нам никако не помаже да сазнамо да ли полазни, шири скуп има апсолутног победника.

Бојер-Муров алгоритам се заснива на следећој идеји. Замислимо поступак у којем би се гласови за различите кандидате међусобно потирали и нестали. Уколико постоји апсолутни победник тада ће након потирања остати бар један (а можда и више) гласова за тог водећег кандидата. Покажимо ово мало прецизније.

Ако из неког скупа избацимо било која два различита гласа, онда ће мањи скуп имати апсолутног победника само ако га је имао и већи скуп, и у питању ће бити исти апсолутни победник.

Доказ коректности. Докажимо ово. Ако постоји апсолутни победник p у ширем скупу од n гласова и он има m гласова, онда је $m > n/2$.

- Ако су из скупа избачена два гласа која нису за апсолутног победника онда n има и даље m гласова, а редуковани скуп има $n-2$ елемента, па важи $m > n/2 > (n-2)/2$ па апсолутни победник није промењен.

- Ако је избачен један глас за особу p и један који није за њу, тада p има $m-1$ глас, а у скупу има $n-2$ елемента, па је $m-1 > (n-2)/2 = n/2-1$, јер је $m > n/2$ и апсолутни победник није промењен.

Обратимо пажњу на то да постојање апсолутног победника у мањем скупу не имплицира то да у већем скупу постоји апсолутни победник.

Доказ коректности. Докажимо и ово. Претпоставимо да је p апсолутни победник скупа од $n-2$ елемента и да он има m_1 гласова. Он је апсолутни победник у мањем скупу ако и само ако је $m_1 > (n-2)/2$, тј. $m_1 > n/2-1$. То значи да је он апсолутни победник у мањем скупу и када је $m_1 = n/2$. Размотримо шта се дешава када се већи скуп (од n елемената) добија додавањем два различита гласа од којих ни један није гласао за p_1. У том већем скупу, број гласова за p_1 остаје $m_1 = n/2$ али неједнакост $m_1 > n/2$ није испуњена па p_1 није апсолутни победник у ширем скупу од n елемената.

Дакле, избацивањем два различита гласа, редукујемо димензију проблема. Ако у тако добијеном мањем скупу нема апсолутног победника, нема га ни у већем, а ако у мањи скуп има апсолутног победника, он је једини кандидат за апсолутног победника у већем скупу, међутим, потребно је накнадно експлицитно проверити да ли је тај кандидат заиста апсолутни победник у већем скупу.

Излаз из овог суштински рекурзивног поступка настаје када не можемо више избацивати парове различитих елемената. Празан скуп нема апсолутног победника, а непразан скуп у коме не постоје два различита елемента садржи гласове само за једног кандидата који је очигледно апсолутни победник.

Описали смо, дакле, индуктивно-рекурзивну конструкцију која нам може омогућити да добијемо кандидата за апсолутног победника (што ће се десити ако скуп гласова избацивањем парова различитих гласова сведемо на непразан скуп у ком су сви гласови за истог кандидата) или да утврдимо да апсолутни победник не постоји (што ће се десити ако скуп гласова избацивањем парова различитих гласова сведемо на празан скуп). Ова рекурзивна конструкција нам даје могућност имплементације решења, међутим, питање проналажења две различите особе у скупу и њиховог избацивања није рачунски тривијално, па ћемо имплементацију направити на други начин.

На основу ове рекурзивне конструкције, следи коректност следећег тврђења. Претпоставимо да је неки скуп гласова подељен на два дисјунктна подскупа, таква да се у једном од њих налазе парови гласова таквих да се у сваком пару налазе гласови за различите особе и да су сви гласови у другом скупу за једну исту особу. Избацивањем једног по једног пара гласова из првог скупа, проблем рекурзивно своди на све мању и мању димензију, све док не остану само елементи другог скупа. На основу описане рекурзивне конструкције следи да ако је други скуп празан, онда не постоји апсолутни победник у почетном скупу свих гласова, а ако није, онда је особа за коју су сви гласови у другом скупу једини кандидат за апсолутног победника.

Алгоритам ћемо добити индуктивном конструкцијом, тако што ћемо кренути од празног скупа особа подељеног на два празна подскупа и у скуп особа који тренутно обрађујемо додавати једну по једну особу полазног скупа, док их све не обрадимо. У сваком кораку ћемо претпоставити да знамо поделу текућег скупа на скуп парова различитих гласова и скуп гласова за неку (произвољну) особу (тај услов ће бити централна инваријанта петље). Кључни задатак сада је да на основу такве поделе текућег скупа направимо такву поделу скупа који се добија када се текућем скупу дода нека особа.

- Ако је други скуп празан или ако је нова особа једнака његовим елементима, други скуп ћемо проширити новом особом (у другом скупу ће се и даље налазити парови различитих гласова, а сви гласови у првом скупу ће и даље бити за исту особу).
- Ако је други скуп непразан и ако је нова особа различита од његових елемената (који су сви исти), тада један елемент тог другог скупа заједно са новом особом можемо пребацити у први скуп (у првом скупу ће и даље сви гласови бити за исту особу, а у другом ће бити сви ранији парови различитих гласова и овај новододати пар за који смо такође сигурни да је пар гласова за различите особе).

Приметимо да је уместо два скупа довољно само да памтимо особу o за коју су сви гласови из првог скупа, као и број тих гласова b.

- Ако је $b = 0$ или ако је глас за нову особу једнак o, увећаћемо b за 1. Ако је $b = 0$, ажурирамо вредност o.
- У супротном смањујемо b за 1.

Ако је $b = 0$ након обраде свих гласова апсолутни победник не постоји. У супротном је o једини кандидат за апсолутног победника. Пошто немамо гаранције да ће последњи кандидат за апсолутног победника бити стварно апсолутни победник, та се провера мора експлицитно извршити на крају програма.

Пример. Размотримо гласове $1, 2, 1, 3, 2, 2, 3, 2, 2$ и прикажимо како се два скупа (тј. њихова репрезентација помоћу променљивих o и b) мењају додавањем једног по једног елемента.

```
glas   prvi skup                  drugi skup     o      b
       []                         []             -      0
1      []                         [1]            1      1
2      [(1, 2)]                   []             1      0
1      [(1, 2)]                   [1]            1      1
3      [(1, 2), (1, 3)]           []             1      0
2      [(1, 2), (1, 3)]           [2]            2      1
2      [(1, 2), (1, 3)]           [2, 2]         2      2
3      [(1, 2), (1, 3), (2, 3)]   [2]            2      1
2      [(1, 2), (1, 3), (2, 3)]   [2, 2]         2      2
2      [(1, 2), (1, 3), (2, 3)]   [2, 2, 2]      2      3
```

Кандидат је, дакле, 2, а накнадна провера пребројавањем његових гласова показује да је он апсолутни победник (јер има 5 гласова, од укупних 9).

Анализа сложености. Обе фазе (и одређивање кандидата за апсолутног победника и провера да ли је он апсолутни победник) су сложености $O(n)$, док су меморијски захтеви такође $O(n)$ (јер морамо запамтити све гласове, да бисмо на крају проверили да ли је кандидат заиста апсолутни победник).

```
int apsolutniPobednik(const vector<int>& glasovi) {
  // odredjujemo kandidata za pobednika glasove delimo u dve grupe: u
  // prvoj grupi se svi glasovi mogu ponistiti tako sto se spajaju dva
  // po dva razlicita glasa, a u drugoj grupi su svi glasovi za
  // kandidata za pobednika

  // broj glasova u drugoj grupi
  int broj = 0;
  // kandidat za pobednika
  int kandidat;
  for (int glas : glasovi) {
    // druga grupa je prazna
    if (broj == 0) {
      // trenutni glas je kandidat za pobednika i ubacujemo ga u drugu
      // grupu
      kandidat = glas;
      broj = 1;
    } else if (glas == kandidat)
      // trenutni glas je za kandidata i ubacujemo ga u drugu grupu
      broj++;
    else
      // trenutni glas moze da se ponisti sa jednim glasom za
      // kandidata i oba ih prebacujemo u prvu grupu
      broj--;
  }

  // proveravamo da li postoji kandidat za pobednika i da li je
  // ostvario vise od n/2 glasova
  if (broj > 0 &&
      count(begin(glasovi), end(glasovi), kandidat) > glasovi.size() / 2)
    return kandidat;
  else
    return -1; // nema pobednika
}
```

Задатак: Циклично померање за k места улево

Дати низ од n целих бројева циклички померити (ротирати) за k места улево.

Улаз: У првој линији стандардног улаза налази се природан број n ($1 \leq n \leq 10^5$), који представља број елемената низа, у другој природан број k ($1 \leq k \leq 10^5$) који представља број места за која се низ помера улево, а затим се у следећих n линија налазе цели бројеви у границама од -1000 до 1000 који представљају елементе низа.

Излаз: У n линија стандардног излаза исписати елементе низа који се добија цикличким померањем учитаног низа за k места улево.

Пример

Улаз	*Излаз*
3	4
10	5
1 2 3 4 5 6 7 8 9 10	6
	7
	8
	9
	10
	1
	2
	3

Решење

Пошто се цикличним померањем за n места низ враћа у своју оригиналну позицију, није тешко уочити да је циклично померање за k места улево исто што и циклично померање за $k \bmod n$ места улево. Зато ћемо у свим наредним решењима претпоставити да се након учитавања броја `k` он мења вредношћу `k % n` и да важи да је $0 \leq k < n$.

Задатак се може решити на заиста много различитих начина, који се разликују по количини додатне меморије коју користе и по броју операција које је потребно применити. С обзиром на потенцијално велике димензије улазног низа, пожељно је користити она решења која низ трансформишу у месту тј. не користе помоћне низове, и код који је број операција линеаран (тј. пропорционалан броју n).

Наивна решења

k цикличких померања за једно место улево

Решење је засновано на k цикличких померања за једно место улево. Потребно је само циклично померање за једно место улево поновити k пута.

Анализа сложености. Ово решење не користи додатни низ (користи само једну помоћну целобројну променљиву), али мана му је велики број операција које се извршавају. Број додела које се извршавају отприлике је једнак nk, што може бити превише за велике вредности n и k (пошто број k може имати исти ред величине као и n, овај алгоритам у најгорем случају има квадратну сложеност).

```
void rotirajUlevoZaJednoMesto(vector<int>& a, int n) {
  // rotiranje za niza za jedno mesto ulevo
  int pom = a[0];
  for (int j = 0; j < n - 1; j++)
    a[j] = a[j + 1];
  a[n - 1] = pom;
}

void rotirajUlevoZaKMesta(vector<int>& a, int n, int k) {
  // k puta ponavljamo rotiranje za jedno mesto
  for (int i = 0; i < k; i++)
    rotirajUlevoZaJednoMesto(a, n);
}
```

Оптимална решења

Коришћење библиотечких функција

Језик C++ нуди функцију `rotate` која ротира елементе низа тако да дати елемент постане први. Функцији се прослеђују итератор који указује на почетак низа (или вектора), итератор који указује на елемент који треба да постане први и итератор који указује непосредно иза краја низа.

```
// rotiramo niz primenom bibliotecke funkcije
rotate(begin(a), next(begin(a), k), end(a));

// da je koriscen niz, a ne vektor, poziv bi bio
// rotate(a, next(a, k), next(a, n));
```

Размена блокова без коришења помоћне меморије

Ако би низ имао $2k$ елемената, ротација за k места би се вршила тако што би се разменили блокови елемената који чине прву и другу половину низа. У општем случају ситуација је мало компликованија, али се поново може свести на размену блокова елемената. Нека низ има n елемената и нека га ротирамо за k места улево. Нека првих k елемената чине блок који ћемо означити са L, а преосталих $n - k$ елемената чине блок који ћемо означити са D. Размотримо следеће случајеве, у зависности од односа дужина та два блока.

- Ако су блокови L и D једнаке дужине, њиховом разменом се добија тражено решење.
- Ако је блок L краћи, означимо са D_1 почетни део блока D дужине k, а са D_2, преостали део блока D. Елементи блока D_1 треба да буду почетни елементи у траженом решењу и да бисмо то постигли, можемо их разменити са елементима блока L. Тиме из ситуације LD_1D_2 долазимо у ситуацију D_1LD_2, док је тражено решење облика DL, тј. D_1D_2L. Да бисмо то постигли, потребно је да низ LD_2 заротирамо за дужину блока L улево (а то је опет k), тј. да разменимо блокове L и D_2, што је проблем истог облика, али мање димензије од полазног.
- Ако је блок L дужи, означимо са L_1 почетни део блока L дужине $n - k$, а са L_2, преостали део блока L. Елементи блока D треба да буду почетни елементи у траженом решењу и да бисмо то постигли, можемо их разменити са елементима блока L_1. Тиме из ситуације L_1L_2D долазимо у ситуацију DL_2L_1, док је тражено решење облика DL, тј. DL_1L_2. Да бисмо то постигли, потребно је да низ L_2L_1 заротирамо за дужину блока L_2 улево (а то је $2k - n$), тј. да разменимо блокове L_2 и L_1, што је проблем истог облика, али мање димензије од полазног.

Дакле, задатак се може решити индуктивно-рекурзивном конструкцијом. Први случај можемо подвести под други (или трећи) јер се комплетан леви блок замењује са десним, након чега остаје да се размене празни блокови, што не производи никакав ефекат, па излаз из рекурзије може бити случај када је било који од блокова празан.

Претпоставићемо да на располагању имамо функцију `razmeni` која у датом низу или вектору размењује блокове исте дужине који се не преклапају и који почињу на две дате позиције. Ту функцију је веома једноставно имплементирати. У језику C++ може се искористити библиотечка функција `swap_ranges` која као аргументе прима два итератора која ограничавају први блок и итератор који указује на почетак другог блока.

Решење се може испрограмирати рекурзивно.

```
// razmenjujemo u vektoru ”v” blok duzine ”duzina” koji pocinje na
// poziciji ”b1” sa blokom duzine duzina koji pocinje na poiciji ”b2”
void razmeni(vector<int>& v, int p1, int p2, int d) {
  swap_ranges(next(v.begin(), p1), next(v.begin(), p1 + d),
              next(v.begin(), p2));
}


// razmenjujemo blok L koji pocinje na poziciji pl i duzine je dl i
// blok D koji pocinje na poziciji pd i duzine je dd
void razmeniBlokove(vector<int>& v, int pl, int dl, int pd, int dd) {
  // ako je neki blok prazan nema potrebe za razmenom
  if (dl == 0 || dd == 0)
     return;
  if (dl <= dd) {
     // razmenjujemo kompletan levi blok sa pocetkom desnog
     razmeni(v, pl, pd, dl);
```

```
      // iz situacije L.D1.D2 dosli smo u situaciju D1.L.D2 i
      // da bismo dosli u zeljenu situaciju D1.D2.L moramo
      // da razmenimo L i D2
      razmeniBlokove(v, pl + dl, dl, pd + dl, dd - dl);
   } else {
      // razmenjujemo kompletan desni blok sa pocetkom levog
      razmeni(v, pl, pd, dd);
      // iz situacije L1.L2.D dosli smo u situaciju D.L2.L1 i
      // da bismo dosli u zeljenu situaciju D.L1.L2 moramo
      // da razmenimo L1 i L2
      razmeniBlokove(v, pl + dd, dl - dd, pd, dd);
   }
}

void rotiraj(vector<int>& v, int k) {
   // razmenjujemo pocetni blok duzine k i ostatak niza
   razmeniBlokove(v, 0, k, k, v.size() - k);
}
```

Пример. Размотримо следећи пример. Желимо да ротирамо следећи низ $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ за 4 места улево.

Након ротације требало би да добијемо $[5, 6, 7, 8, 9, 10, 1, 2, 3, 4]$.

- Ротација се врши тако што размењујемо леви блок $L = [1, 2, 3, 4]$ са десним блоком $D = [5, 6, 7, 8, 9, 10]$. Важи да је $p_l = 0$, $d_l = k = 4$, $p_d = k = 4$ и $d_d = n - k = 6$. Пошто је леви блок краћи од десног блока, можемо заменити цео блок L са блоком $D_1 = [5, 6, 7, 8]$. Тако добијамо $[5, 6, 7, 8, 1, 2, 3, 4, 9, 10]$.

  На тај начини бројеви $[5, 6, 7, 8]$ су дошли на своје место, а да би се од овог низа добио крајњи резултат, потребно је у делу низа иза њих разменити блок $L = [1, 2, 3, 4]$ и блок $D_2 = [9, 10]$.

- Сада је $L = [1, 2, 3, 4]$, а $D = [9, 10]$. Важи да је $p_l = 4$, $d_l = 4$, $p_d = 8$ и $d_d = 2$. Пошто је леви блок дужи од десног блока, размењујемо цео блок D са блоком $L_1 = [1, 2]$ и тако добијамо $[5, 6, 7, 8, 9, 10, 3, 4, 1, 2]$.

- До коначног решења сада можемо доћи тако што разменимо блокове $L_2 = [3, 4]$ и $L_1 = [1, 2]$. Важи да је $p_l = 6$, $d_l = 2$, $p_d = 8$ и $d_d = 2$. У овом случају је дужина оба блока једнака, па се након њихове размене добијам $[5, 6, 7, 8, 9, 10, 1, 2, 3, 4]$ и процедура се зауставља.

Анализа сложености. Докажимо да се алгоритам зауставља и проценимо му сложеност. Централна мера прогреса у алгоритму је укупна дужина блокова који се размењују. Укупна дужина креће од n и смањује се све док не дође до нуле. У првом случају се врши размена блокова дужине d_l за шта је потребно $O(d_l)$ корака и након тога се врши рекурзивни позив такав да је збир дужина блокова управо за d_l мањи од полазног збира дужина (нови збир је $d_l + (d_d - d_l) = d_d$, а полазни $d_l + d_d$). Слично, у другом случају се врши замена блокова дужине d_d за шта је потребно $O(d_d)$ корака и након тога се врши рекурзивни позив такав да је збир дужина блокова управо за d_d мањи од полазног збира дужина (нови збир је $(d_l - d_d) + d_d = d_l$, а полазни $d_l + d_d$). Дакле, рекурентна једначина је у оба случаја једнака $T(n) = T(n - d) + O(d)$, где је $d = \min(d_l, d_d)$ и њено решење је $T(n) = O(n)$.

Још једноставније, у сваком функције `razmeni` се помоћу d размена тачно d елемената доводи на своје финално место, одакле се више не мрда, а у рекурзивној функцији се осим тога (и рекурзивних позива) не врши никакав други посао. Пошто се по завршетку рекурзије сваки елемент налази на свом месту извшено је тачно n размена.

Рекурзија је репна и лако је се можемо ослободити. Могућа је и додатна оптимизација, јер почетне позиције блокова који се размењују не морамо посебно памтити, већ их можемо увек израчунати ако знамо дужине блокова који се размењују и знамо да се они налазе у завршном делу низа. Илустрације ради, приказаћемо мало другачије итеративно решење у ком ћемо претпоставити да у сваком кораку блок који треба да се нађе на крају низа доводимо на његово место, а да након тога размене преосталих блокова вршимо на почетку низа.

Претпоставимо да је у сваком кораку петље потребно разменити блокове L и D који се састоје од почетних d_l, па затим d_d елемената низа. Вредности d_l и d_d се иницијализују на k и $n - k$, а затим се у петљи понавља

следеће.

- Ако су d_l и d_d једнаки, тада се размењују блокови који почињу на позицјама 0 и d_l и који су дужине $d_l = d_d$ и петља се прекида.
- Ако је $d_l < d_d$, низ је облика LD_1D_2, где је $|D_2| = |L| = d_l$. Разменићемо блокове L и D_2. Први почиње на позицији 0, а други на позицији $|LD_1| = d_l + (d_d - d_l) = d_d$. Након тога ћемо добити низ D_2D_1L и преостаће нам да разменимо блокове D_2 и D_1, који су на почетку низа. Дужина блока D_2 је d_l, а блока D_1 је $d_d - d_l$. Зато се вредност d_d умањује за d_l, док се d_d не мења.
- Ако је $d_l > d_d$, низ је облика L_1L_2D, где је $|L_2| = |D| = d_d$. Разменићемо блокове L_2 и D. Први почиње на позицији $|L_1| = d_l - d_d$, а други на позицији $|L_1L_2| = d_l$. Након тога ћемо добити низ L_1DL_2 и преостаће нам да разменимо блокове L_1 и D. Дужина блока L_2 једнака је дужини блока D и износи d_d, па је дужина блока L_1 једнака $d_l - d_d$. Зато се d_l умањује за d_d док се d_d не мења.

Пример. Применимо описани алгоритам на још једном примеру. Нека је дат низ $[1, 2, 3, 4, 5]$ и $k = 3$.

- У почетку је $d_l = k = 3$ и $d_d = n - k = 2$, па треба разменити блокове $L = [1, 2, 3]$ и $D = [4, 5]$. Пошто је $d_d < d_l$, размењује се се блок $L_2 = [2, 3]$, који почиње на позицији $d_l - d_d = 1$ и блок $D = [4, 5]$, који почиње на позицији $d_l = 3$. Оба су дужине $d_d = 2$. Тако се добија низ $[1, 4, 5, 2, 3]$. Након тога се d_l умањује за d_d и постаје 1.
- Сада је $d_l = 1$ и $d_d = 2$, па треба разменити блокове $L = [1]$ и $D = [4, 5]$. Пошто је $d_l < d_d$, размењују се леви блок $L = [1]$, који почиње на позицији 0 и десни блок $D_2 = [5]$, који почиње на позицији $d_d = 2$ (оба су дужине $d_l = 1$). Тако се добија низ $[5, 4, 1, 2, 3]$. Након тога се d_d умањује за d_l и постаје 1.
- Сада је $d_l = d_d = 1$. Зато се размењују блокови $[5]$ и $[4]$, који почињу на позицији 0 тј. $d_l = 1$, чиме се низ доводи у жељену конфигурацију $[4, 5, 1, 2, 3]$.

Анализа сложености. Приметимо да је сложеност овог поступка линеарна, јер се у сваком кораку помоћу d размена бар d елемената доводи на своје коначно место и елиминише из даље обраде.

```
// razmenjujemo u vektoru "v" blok duzine "duzina" koji pocinje na
// poziciji "p1" sa blokom duzine d koji pocinje na poiciji "p2"
void razmeni(vector<int>& v, int p1, int p2, int d) {
  for (int i = 0; i < d; i++)
      swap(v[p1+i], v[p2+i]);
}

// funkcija rotira elemente niza a, duzine n za k mesta ulevo
void RotirajUlevo(vector<int>& a, int k) {
  // broj elemenata niza
  int n = a.size();

  // ciklicko pomeranje za k mesta je isto sto i ciklicko pomeranje za
  // k % n mesta
  k %= n;

  // vrsimo rotiranje niza
  int dl = k;
  int dd = n-k;

  while (true) {
    if (dl < dd) {
      // niz je oblika LD1D2, |L|=|D2| i razmenjujemo L i D2 tako da
      // dobijamo D2D1L
      razmeni(a, 0, dd, dl);
      dd -= dl;
    } else if (dl > dd) {
      // niz je oblika L1L2D, |L1|=|D| i razmenjujemo L2 i D tako da
      // dobijamo L1DL2
      razmeni(a, dl-dd, dl, dd);
      dl -= dd;
```

```
    } else {
      // niz je oblika LD, |L|=|D| i razmenjujemo L i D cime
      // zavrsavamo postupak
      razmeni(a, 0, dl, dl);
      break;
    }
  }
}
```

Базирано на идеји размене блокова, тј. на претходним индуктивно-рекурзивним конструкцијама можемо формулисати још један алгоритам (са доста елегантном имплементацијом). Претпоставимо да имамо на располагању променљиве l и d које указују на почетак блокова L и D и променљиве k и n које указују на њихове крајеве (прецизније, позиције иза њихових крајева).

На почетку је $l = 0$ и $d = k$.

У петљи размењујемо елементе на позицијама l и d, увећавајући, при том, оба индекса. Током поступка ће се догодити или да индкес l достигне k или да индекс d достигне n.

- Ако се обе ствари догоде истовремено, блокови су били исте дужине и успешно смо их разменили, чиме је посао завршен.
- Ако l достигне k пре него што d достигне n, тада је леви блок био краћи и у овој ситуацији је заправо размењен са почетним делом десног блока, чиме се дошло у конфигурацију D_1LD_2. Преостало је још разменити блокове L и D_2. Почетак блока L је на месту тренутног индекса l, а крај му је тик испред текућег индекса d и зато ажурирамо вредност k на вредност d. Почетак и крај дела D_2 су на својим исправним вредностима (d и n).
- Ако d достигне n пре него што l достигне k, тада је десни блок био краћи и у овој ситуацији је заправо размењен са почетним делом левог блока, чиме се дошло у конфигурацију DL_2L_1. Преостало је још разменити блокове L_2 и L_1. Почетак блока L_2 је на месту тренутног индекса l, а крај му је тик испред текућег индекса k. Међутим, почетак блока L_1 је на месту текућег индекса k и зато је потребно вредност d поставити на k, док је крај блока L_1 тик испред индекса n.

Дакле, алгоритам може имати наредну, веома елегантну имплементацију.

```
void RotirajUlevo(vector<int>& a, int k) {
  // broj clanova niza
  int n = a.size();

  // ciklicko pomeranje za k mesta je isto sto i ciklicko pomeranje za
  // k % n mesta
  k %= n;

  // vrsimo rotiranje niza razmenom blokova
  int l = 0; // pocetak levog bloka
  int d = k; // pocetak desnog bloka
  // razmenjujemo blokove L i D tj. blokove [l, k) u [d, n)
  while (true) {
    // menjamo tekuce elemente u blokovima
    swap(a[l++], a[d++]);

    // stigli smo do kraja oba bloka, pa je posao zavrsen
    if (l == k && d == n)
      break;

    if (d == n) // stigli smo do kraja desnog bloka
      // razmenili smo pocetak levog bloka sa desnim i dosli u
      // situaciju DL2L1 i jos treba da razmenimo blokove L2 i L1, a
      // to su blokovi [l, k) u [k, n), tako da d treba postaviti na k
      d = k;
```

```
    if (l == k) // stigli smo do kraja levog bloka
      // razmenili smo levi blok sa pocetkom desnog i dosli u situaciju
      // D1LD2 i jos treba da razmenimo blokove L i D2, a to su blokovi
      // [l, d) u [d, n), tako da k treba postaviti na d
      k = d;
  }
}
```

Анализа сложености. Сложеност је линеарна у односу на дужину низа и не користи се додатни меморијски простор осим помоћне променљиве у склопу размене два елемента низа.

Пример. Применимо описани алгоритам на примеру. Нека је дат низ $[1, 2, 3, 4, 5]$ и $k = 3$.

- Бројач l креће од вредности 0, бројач d од вредности $k = 3$, тј. креће се са обрадом блокова $[1, 2, 3]$ и $[4, 5]$. У тренутку када бројач d дође до краја (до вредности n), бројач l има вредност 2. Тада је конфигурација низа $[4, 5, 3, 1, 2]$.
- Тада се вредност d поставља на текућу вредност k а то је 3 и прелази се на обраду блокова $[3]$ и $[1, 2]$. Овај пут l стиже до вредности k када d има вредност 4 и тада је конфигурација $[4, 5, 1, 3, 2]$.
- Tada se k помера на 4 и разматрају се блокови $[3]$ и $[2]$. Након њихове размене долази се до краја и тражене конфигурације $[4, 5, 1, 2, 3]$.

Доказ коректности. Веома интересантно би било доказати директно да је овај алгоритам коректан. У наставку ћемо једино доказати да се претходни алгоритам зауставља (што није само по себи тривијално). Једна од инваријанти претходне петље је да је $0 \leq l \leq k \leq d \leq n$. Докажимо то.

- Пошто је $0 < n$, $0 < k$ и $k < n$, на почетку важи $0 = l < k = d < n$, па је инваријанта испуњена.
- Пошто је $0 < k$, на основу услова прекида петље, када се уђе у тело петље знамо да је $0 \leq l < k \leq d < n$. Након тога се l и d увећавају за 1, па након тога важи $0 \leq l \leq k < d \leq n$. Ако је $l = k$ и $d = n$, петља се завршава, а инваријанта важи и након прекида петље. У супротном, ако је $l = k$, тада је $d < n$, док се k поставља на d, па однос $0 \leq l < k \leq d < n$ важи и пре наредне итерације. Слично, ако је $d = n$, тада је $l < k$, док се d поставља на k, па однос $0 \leq l < k \leq d < n$ важи и пре наредне итерације. На основу овога знамо да ова инваријанта остаје испуњена и након извршавања тела петље.

Потребно је још доказати да се у неком тренутку мора догодити да је $l = k$ или да је $d = n$. Међутим, у сваком кораку петље променљива l се увећава за 1, док се горња граница n не мења. Стога знамо да ће се након највише n корака десити да је $l = n$. На основу инваријанте тада ће морати да важи да је $l = k = d = n$, па ће се петља зауставити. Уједно видимо и да се у најгорем случају петља извршава n пута, па је алгоритам сложености $O(n)$.

Задатак: Абацаба

Низ слова $ABACABADABACABAEABACABADABACABAFABACA...$ се може формирати на следећи начин:

1. Низ је на почетку празан.
2. На низ се допише прво велико слово енглеског алфабета које се не појављује у формираном делу низа, а иза тог слова се понове сва слова која су се појавила пре њега.
3. Корак 2 се понови потребан број пута

Тако после прве примене корака 2 добијамо низ A, после друге низ ABA, после треће низ $ABACABA$ итд.

Одредити слово које се појављује на n-том месту у низу, бројећи места од 1. Редослед слова у енглеском алфабету је ABCDEFGHIJKLMNOPQRSTUVWXYZ.

Улаз: Један природан број мањи од 67 108 864.

Излаз: Једно велико слово енглеског алфабета.

Пример 1

Улаз	*Излаз*
8	D

Пример 2

Улаз	*Излаз*
65	A

Пример 3

Улаз	*Излаз*
100	C

Решење

Решење грубом силом је да се у меморији креира цео низ карактера и да се затим прочита карактер са одговарајуће позиције. Ово решење троши превише меморије, а и времена док се дугачак низ карактера не изгради.

```
string s = ””;
char slovo = 'A';
while (s.size() <= n - 1) {
  s = s + slovo + s;
  slovo++;
}
cout << s[n-1] << endl;
```

Задатак се може решити без креирања дугачке ниске карактера, ако пажљиво проучимо правилност по којој се слова појављују. Прво слово `A` се налази на месту 1, прво слово `B` на месту `2`, прво слово `C` на месту `4`, прво слово `D` на месту `8` итд., па се може наслутити да се прва појављивања слова налазе на местима која су степени броја 2.

Заиста, ако дужину низа карактера пре уметања новог слова абецеде обележимо са d_k, важи да је $d_0 = 0$ (пре уметања слова `A` не налази се ниједан карактер) и да је $d_{k+1} = 2d_k + 1$ (пре уметања новог слова налази се ниска која је добијена тако што је претходно слово спојено са два појављивања ниске која се појављивала пре тог претходног слова). Зато је $d_k = 2^k - 1$ (важи да је $2^0 - 1 = 0$ и да је $2^{k+1} - 1 = 2 \cdot (2^k - 1) + 1$), док је прва позиција слова k (ако се слова броје од један) једнака 2^k.

Дакле, ако је унети број n неки степен двојке $n = 2^k$, онда је у питању место на ком се слово први пут појављује и важи да је $k = \log_2 n$ ($\log_2 n$ означава број k такав да је $2^k = n$, нпр. $\log_2 32 = 5$, јер је $2^5 = 32$). Знајући k слово лако можемо одредити сабирајући k са ASCII/UNICODE кодом слова `A`.

У супротном проблем можемо свести на проблем мање димензије. Нека је $2^k < n < 2^{k+1}$, тј. нека је k највећи степен двојке мањи од n. Карактер који тражимо налази се, дакле, иза позиције 2^k у делу низа који је добијен копирањем дела низа испред позиције 2^k. Зато је n-ти карактер у целом низу једнак $(n - 2^k)$-том карактеру у копираном делу, а пошто део који се копира почиње на почетку низа, једнак је $(n - 2^k)$-том карактеру у целом низу.

Овим смо описали рекурзивни поступак којим ефикасно можемо доћи до решења.

$$f(n) = \begin{cases} \log_2 n, \text{за } n = 2^k \\ f(n - 2^k), \text{за } 2^k < n < 2^{k+1} \end{cases}.$$

На пример, $f(23) = f(23 - 16) = f(7) = f(7 - 4) = f(3) = f(3 - 2) = f(1) = 0$, па се на месту 23 налази слово `A`. Слично, на пример, важи да је $f(40) = f(40 - 32) = f(8) = 3$, па се на месту 40 налази слово `D`.

На основу претходне дефиниције би се могла имплементирати рекурзивна функција (за то би било потребно испитати да ли је дати број степен броја 2, наћи логаритам таквог броја, и наћи највећи степен броја 2 од ког је дати број већи или једнак). Ипак, за тим нема потребе. Анализирајући рад такве рекурзивне функције можемо унапред закључити шта ће њен резултат бити, без потребе за њеним извршавањем. Претпоставимо да знамо бинарни запис броја n тј. да знамо да се број n представља као збир неких степенова двојке $2^{s_m} + 2^{s_{m-1}} + \ldots + 2^{s_0}$. Током рекурзије од броја ће се одузимати један по један степен двојке, све док не остане само 2^{s_0} и тада ћемо знати да је $k = s_0$. Дакле важи да је резултат једнак најмањем степену двојке који учествује разлагању броја на збир степенова двојке тј. да је тражени број k једнак позицији најдешње јединице у бинарном запису броја (претпостављамо да се позиције броје од 0, здесна). До траженог резултата је могуће доћи дељењем броја са 2_0^s, тј. узастопним дељењем броја са 2 све док последњи сабирак не постане 1, тј. све док је број паран (то ће се догодити тачно s_0 пута).

```
int k = 0;
while (n % 2 == 0) {
  n /= 2;
  k++;
}
cout << (char)('A' + k) << endl;
```

Сличан резултат можемо добити и баратајући директно са интерним записом броја у рачунару, коришћењем битовских оператора. Позицију крајње десне јединице једноставно можемо израчунати шифтовањем (поме-

рањем) бинарног записа броја удесно за једно место све док последња цифра не постане једнака 1 (последњу цифру можемо испитати битовском конјункцијом са 1).

```
int k = 0;
while ((n & 1) == 0) {
  n >>= 1;
  k++;
}
cout << (char)('A' + k) << endl;
```

Савремени хардвер често поседује и инструкцију *count trailing zeroes* којом се одређује број нула иза последње јединице у бинарном запису (што је баш позиција последње јединице). Иако програмски језици обично не стандардизују приступ овој операцији, неки компилатори нуде подршку за то (на пример, ако се користи GCC, може се употребити функција `__builtin_ctz`, а ако се користи Microsoft Visual C++, може се употребити функција `_BitScanReverse`).

```
cout << (char)('A' + __builtin_ctz(n)) << endl;
```

Задатак: Морзеов низ

Низ $1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, \ldots$, који се састоји од нула и јединица, гради се на следећи начин: први елемент је 1; други се добија логичком негацијом првог $NOT(1) = 0$, трећи и четврти логичком негацијом претходна два $NOT(1) = 0$, $NOT(0) = 1$, пети, шести, седми и осми логичком негацијом прва четири – добија се $0, 1, 1, 0$ итд. Дакле, кренувши од једночланог сегмента 1, сваком почетном сегменту који је дужине 2^k (k узима вредности $0, 1, 2, \ldots$) дописује се сегмент исте дужине добијен логичком негацијом свих елемената почетног сегмента. За задато n одредити n-ти члан низа (бројање креће од 1).

Улаз: У првој линији стандардног улаза налази се природан број n ($1 \leq n \leq 10^9$).

Излаз: На стандарном излазу приказати цифру (0 или 1) на позицији n

Пример 1		**Пример 2**		**Пример 3**	
Улаз	*Излаз*	*Улаз*	*Излаз*	*Улаз*	*Излаз*
15	0	1234	0	12345678	1

Решење

Формирање целог низа

Формулација задатка сугерише решење у коме би се почев од задатог првог елемента $x_1 = 1$ редом формирали сегменти по задатим правилима (x_2), (x_3, x_4), $(x_5 \ldots x_8), (x_9 \ldots x_{16}), (x_{17} \ldots x_{32}), \ldots$. Дакле, елементи текућег сегмента се формирају негацијом претходно формираних елемената $x_1 \ldots x_k$, који су на растојању k, где је k степен броја 2 (увећава се дупло након сваког преписаног сегмента).

Сложеност овог приступа је $O(n)$.

Ово уписивање се може реализовати помоћу угнежђених петљи где унутрашња петља дуплира садржај низа, а спољашња контролише колико пута садржај треба дуплирати (додатно осигуравајући да се уписивање прекине када се попуни потребних n елемената низа).

```
bool Morzeov(int n) {
  // Morzeov niz
  vector<bool> a(n+1);
  a[1] = true;
  int upisano = 1; // broj upisanih elemenata
  int duzina = 1; // duzina segmenta koji se trenutno negira
  while (upisano < n) {
    // negiramo segment trenutne duzine
    // prekidamo petlju ranije ako tokom toga dostignemo n upisanih elemenata
    for (int i = 1; i <= duzina && upisano < n; i++) {
      a[duzina + i] = !a[i];
      upisano++;
    }
    // naredni segment koji treba negirati je duplo duzi
```

```
    duzina *= 2;
  }

  // upisano je n elemenata niza, pa ocitavamo rezultat
  return a[n];
}
```

Низ се може попунити и помоћу само једне петље у којој се врши преписивање елемената док се не упише њих n тако што се на место i преписује елемент са позицијје $i-k$, увећавајући степен двојке k када се цео претходни подсегмент препише.

```
bool Morzeov(int n) {
  // Morzeov niz
  vector<bool> a(n+1);
  a[1] = true;
  // rastojanje elemenata koji se negiraju
  int k = 1;
  // popunjavamo niz zakljucno sa pozicijom n
  for (int i = 2; i <= n; i++) {
    // negiramo odgovarajuci element
    a[i] = !a[i - k];
    // negirali smo ceo segment, pa prelazimo na sledeci
    if (i == 2 * k)
      k *= 2;
  }
  // upisano je n elemenata niza, pa ocitavamo rezultat
  return a[n];
}
```

Веза између одговарајућих чланова

Директна решења задатка подразумевају формирање свих елемената низа који претходе n-том члану. Да би се израчунао 2001. члан мора се израчунати свих 2000 претходних чланова.

Уочимо следеће, како бисмо проблем решили без коришћења низа и без израчунавања свих чланова низа који претходе n-том: формирање елемената сегмента на основу претходног поступка, значи да се сваки пут дужина низа удваја, тј. представља степен броја 2.

- Приликом негирања почетног сегмента добијамо да је $x_2 = NOT(x_1)$. У овом случају важи да је $x_n = NOT(x_{n-1})$.
- Негирањем наредног сегмента добијамо да је $x_3 = NOT(x_1)$ и $x_4 = NOT(x_2)$. У овом случају важи да је $x_n = NOT(x_{n-2})$.
- Након тога добијамо да је $x_5 = NOT(x_1)$, $x_6 = NOT(x_2)$, $x_7 = NOT(x_3)$ и $x_8 = NOT(x_4)$. У овом случају важи да је $x_n = NOT(x_{n-4})$.

Дакле, за $n > 1$ важи рекурентна формула $x_n = NOT(x_{n-m})$, где је m - максимални степен броја 2, који је строго мањи од n. Ова рекурентна формула омогућава веома ефикасно израчунавање траженог члана низа. На пример,

$x_{15} = NOT(x_{15-8}) = NOT(x_7) = NOT(NOT(x_{7-4})) = x_{7-4} = x_3 = NOT(x_{3-2}) = NOT(x_1) = NOT(1) = 0.$

Тражени број се добија у неколико корака и за веће вредности броја n. На пример, за $n = 2001$:

$x_{2001} = NOT(x_{2001-1024}) = NOT(x_{977}) = x_{977-512} = x_{465} = NOT(x_{465-256}) = NOT(x_{209}) = x_{209-128} = x_{81} = NOT(x_{81-64}) = NOT(x_{17}) = x_{17-16} = x_1 = 1.$

Рекурентна формула нам указује да решење можемо веома једноставно реализовати уз помоћу рекурзивне функције. У имплементацији се користи помоћна функција за одређивање траженог степена двојке (ову функцију је могуће имплементирати и ефикасније, коришћењем операција над битовима, међутим и ова једноставна имплементација је сасвим довољна).

Пошто се сваки елемент низа израчунава на основу неког из претходне половине низа, у сваком кораку се n бар полови, тако да је сложеност овог приступа $O(\log n)$.

```
// vraca najveci stepen od 2, koji je manji od n
int MaxStepen2(int n) {
  int max = 1;
  while (max * 2 < n)
    max *= 2;
  return max;
}

// vraca n-ti element Morzeovog niza ako se pozicije broje od 1
bool Morzeov(int n) {
  if (n == 1)
    return true;
  return !Morzeov(n - MaxStepen2(n));
}
```

Рекурзију је могуће једноставно елиминисати и до решења можемо доћи и итеративно тако што ћемо полазећи од $x = 1$, при сваком преласку на индекс настао умањењем за највећи степен броја негирати текућу вредност x.

```
// vraca n-ti element Morzeovog niza ako se pozicije broje od 1
bool Morzeov(int n) {
  bool x = true; // prvi clan niza
  while (n > 1) {
    n -= MaxStep2(n);
    x = !x;
  }
  return x;
}
```

Приметимо да се током претходно описаног поступка уклања један по један бит броја, све док не остане само један бит (тј. док број не постане степен двојке). Након тога се тај бит помера за једно место надесно, све док број не постане 1.

На пример, низ

$$x_{44} = NOT(x_{12}) = x_4 = NOT(x_2) = x_1 = 1$$

се бинарно може представити помоћу

$$x_{101100} = NOT(x_{001100}) = x_{000100} = NOT(x_{000010}) = x_{000001} = 1.$$

Ова друга фаза би се могла избећи, а имплементација поједноставити ако би бројање позиција било од 0, а не од 1 (што лако можемо постићи ако одмах након учитавања умањимо вредност n за 1). Тада би важило да је $x_1 = NOT(x_0)$, затим $x_2 = NOT(x_0)$, $x_3 = NOT(x_1)$, затим $x_4 = NOT(x_0)$, $x_5 = NOT(x_1)$, $x_6 = NOT(x_2)$ и $x_7 = NOT(x_3)$. Разлика је, дакле, у томе што се од сваког индекса одузима највећи степен двојке који је већи или једнак од датог броја (разлика је значајна баш када је индекс који тренутно разматрамо степен двојке). У тој варијанти бисмо уместо x_{44} рачунали x_{43} и важило би

$$x_{43} = NOT(x_{11}) = x_3 = NOT(x_1) = x_0 = 1$$

тј. бинарно

$$x_{101011} = NOT(x_{001011}) = x_{000011} = NOT(x_{000001}) = x_{000000} = 1.$$

Дакле, на овај начин се у сваком кораку уклања један бит броја, све док број не постане 0, негирајући сваки пут резултујући бит. Увид који нас доводи до лепше имплементације је то да ће се исти резултат добити

и када се битови уклањају један по један, кренувши од битова најмање, уместо од битова највеће тежине. Уклањање последњег бита 1 у бинарном запису броја `n` се може урадити веома ефикасно коришћењем израза `n & (n - 1)`.

Можемо приметити да се у овом решењу потврђује да је бројање од 0 уместо од 1 природније и да има лепша математичка својства.

```
// vraca n-ti element Morzeovog niza ako se pozicije broje od 1
bool Morzeov(int n) {
  // prilagodjavamo brojanje tako da krene od 0 umesto od 1
  n--;

  int x = 1;
  while (n) {
    // uklanjamo poslednji bit iz binarnog zapisa broja
    n = n & (n-1);
    x = !x;
  }
  return x;
}
```

На крају, можемо увидети да је битан само број јединица у бинарном запису броја $n-1$. Ако је тај број паран, резултат је 1, а ако је непаран, резултат је 0. Постоје ефикасни алгоритми да се тај број израчуна, а неки рачунари имају и уграђену хардверску инструкцију за то. Иако програмски језик C++ не стандардизује приступ тој инструкцији, ако се користи компилатор GCC, могуће је употребити функцију `__builtin_popcount`, а ако се користи Microsoft Visual C++, могуће је употребити функцију `__popcnt`.

```
// vraca n-ti element Morzeovog niza ako se pozicije broje od 1
bool Morzeov(int n) {
   // prilagodjavamo brojanje tako da krene od 0 umesto od 1
   n--;
   // ispitujemo parnost ukupnog broja bitova jednakih 1 u binarnom
   // zapisu broja n
   return !(__builtin_popcount(n) & 1);
}
```

Задатак: Избацивање цифара на све начине

Напиши програм који за дати природан број n одређује збир свих бројева који се могу добити избацивањем неких цифара броја n.

Улаз: Са стандардног улаза се учитава број који може да има највише 1000 цифара.

Излаз: На стандардни излаз исписати тражени збир.

Пример

Улаз	*Излаз*
`123`	`177`

Објашњење

$123 + 12 + 13 + 23 + 1 + 2 + 3 + 0 = 177.$

Решење

До решења можемо доћи индуктивно-рекурзивном конструкцијом. Ако је број једнак нули, тада је тражени збир једнак нули. У супротном се тај број може разложити на своју последњу цифру и цифре које јој претходе (нпр. број 1234 се може разложити на број 123 и цифру 4). Претпоставимо да умемо да одредимо тражени збир за број коме је уклоњена последња цифра и размотримо како се комбинацијом тог броја и последње цифре може добити тражени збир. У текућем примеру, претпостављамо, дакле, да умемо да одредимо тражени збир за број 123. На сваки од сабирака који учествује у том збиру можемо додати четворку здесна. Сваки од тих сабирака се добија тако што се оригинални број помножи са 10 и дода се 4. Њихов се збир зато може добити тако што се полазни збир (177) помножи са 10 и затим увећа за онолико четворки колико има

сабирака (у текућем примеру их има 8). Пошто су овим покривене све могућности, укупан збир се добија сабирањем полазног и овако трансформисаног збира.

```
177     177*10 + 8*4 = 1802          177 + 1802 = 1979

123     1234
12      124
13      134
23      234
1       14
2       24
3       34
0       4
```

У општем случају, дакле, добијени збир префикса множимо са 10 и увећавамо производом последње цифре и броја бројева који се добијају прецртавањем цифара тог префикса. Тај број није тешко израчунати јер је прилично очигледно да k-тоцифрен број може да генерише 2^k бројева (свака од k цифара може бити или прецртана или непрецртана). Зато можемо ојачати индуктивну хипотезу и направити функцију која уз тражени збир враћа још додатно и број цифара броја (или још боље, одговарајући степен двојке).

Дакле, проблем једноставно можемо решити тако што дефинишемо рекурзивну функцију која прима број n а враћа тражени збир за тај број n и одговарајући степен двојке. Више вредности функција може да врати помоћу својих излазних параметара.

Додатни проблем представља величина бројева који су допуштени на улазу, тако да је јасно да имплементација алгоритма са библиотечким типовима података није исправна. Стога имплементација која користи само основне бројевне типове података не ради коректно за све могуће улазе (то се може решити коришћењем великих бројева).

```
void Zbir(int n, int& zbir, int& b) {
  if (n == 0) {
    zbir = 0;
    b = 1;
  } else {
    int zbirR, brojR;
    Zbir(n / 10, zbirR, brojR);
    zbir = zbirR +  zbirR*10 + brojR * (n % 10);
    b = brojR * 2;
  }
}

int Zbir(int n) {
  int zbir, b;
  Zbir(n, zbir, b);
  return zbir;
}
```

Овакве рекурзије је веома једноставно ослободити се и алгоритам се лако може имплементирати и итеративно. Размотримо како се извршава рекурзивни позив за аргумент $n = 123$.

```
Zbir(123)              177, 8
    Zbir(12)           15, 4
       Zbir(1)         1, 2
          Zbir(0)      0, 1
```

Примећујемо, дакле, да се израчунавања врше у повратку кроз рекурзију и то тако што се вредност претходног збира и претходног степена двојке ажурирају на основу описаног поступка. Исто израчунавање се може описати и итеративним поступком у ком се променљиве иницијализују на 0 и 1, а затим током итерације ажурирају коришћењем једне по једне цифре полазног броја, с лева надесно. Пролазак кроз цифре броја у том редоследу једноставнији је ако се број представи као ниска карактера.

```
int Zbir(const string& broj) {
  int zbir = 0, b = 1;
```

```
  for (char c : broj) {
    zbir += 10*zbir + b * (c - '0');
    b *= 2;
  }
  return zbir;
}
```

Задатак: Не садрже цифру 3

Напиши програм који одређује колико природних бројева из интервала $[0, n]$ не садрже цифру 3 у свом декадном запису.

Улаз: Прва линија стандардног улаза садржи природан број n ($n \leq 2 \cdot 10^9$).

Излаз: У првој линији стандардног излаза приказати тражени резултат.

Пример 1

Улаз	*Излаз*
15	14

Објашњење

У интервалу $[0, 15]$ постоји 16 бројева, а бројеви 3 и 13 једини садрже цифру 3.

Пример 2

Улаз

```
999
```

Излаз

```
729
```

Пример 3

Улаз

```
12345
```

Излаз

```
8262
```

Решење

Бројање бројева који не садрже цифру 3

Задатак можемо решити тако што за сваки број од 0 до n проверимо да ли садржи цифру 3, и ако не садржи увећамо бројач бројева који не садрже цифру 3 (тај бројач у почетку иницијализујемо на нулу). У том решењу примењује се алгоритам одређивања броја елемената серије који задовољавају дати услов, тј. алгоритам бројања филтриране серије. Проверу да ли број садржи цифру 3 можемо реализовати у посебној функцији.

```
bool sadrziCifru3(int n) {
  do {
    if (n % 10 == 3)
      return true;
    n /= 10;
  } while (n > 0);
  return false;
}

int brojeviBezCifre3(int n) {
  int br = 0;
  for (int i = 0; i <= n; i++)
    if (!sadrziCifru3(i))
      br++;
```

```
  return br;
}
```

Ефикасније израчунавање броја бројева

Задатак можемо решити и на много ефикаснији начин (али је решење у том случају доста комплексније). Нека $f(a, b)$ означава број бројева из интервала $[a, b]$ који у свом декадном запису не садрже цифру 3, а $f_0(n)$ број таквих бројева из интервала $[0, n]$. Размотримо пример у коме желимо да израчунамо вредност $f_0(4251)$. Све бројеве из интервала $[0, 4251]$ можемо поделити у неколико група тј. подинтервала. Важи да је

$$[0, 4251] = [0, 999] \cup [1000, 1999] \cup [2000, 2999] \cup [3000, 3999] \cup [4000, 4251].$$

Зато је

$$f_0(4251) = f(0, 999) + f(1000, 1999) + f(2000, 2999) + f(3000, 3999) + f(4000, 4251).$$

Важи да је $f(0, 999) = f_0(999)$. Такође, важи и да је $f(1000, 1999)$ једнак броју $f(0, 999)$ тј. $f_0(999)$. Заиста, између интервала $[0, 999]$ и $[1000, 1999]$ може се успоставити бијекција таква да слика у свом декадном запису садржи цифру 3 ако и само ако њен оригинал у свом декадном запису садржи цифру 3. Слично, важи и да је $f(2000, 2999) = f_0(999)$, док је $f(3000, 3999) = 0$ јер сви бројеви у интервалу $[3000, 3999]$ садрже цифру 3. На крају, важи и да је $f(4000, 4251)$ једнако $f(0, 251)$ тј. $f_0(251)$. Зато је

$$f_0(4251) = 3 \cdot f_0(999) + f_0(251).$$

Дакле, ако знамо бројеве $f_0(999)$ и $f_0(251)$ тада можемо израчунати и број $f_0(4251)$.

Иста техника се може применити и на израчунавање бројева $f_0(999)$ и $f_0(251)$.

Важи да је

$$[0, 999] = [0, 99] \cup [100, 199] \cup \ldots \cup [900, 999],$$

па је

$$f_0(999) = f(0, 99) + f(100, 199) + \ldots + f(900, 999).$$

Важи да је $f(0, 99) = f(100, 199) = f(200, 299) = f(400, 499) = \ldots = f(900, 999) = f_0(99)$, а да је $f(300, 399) = 0$. Стога је

$$f_0(999) = 8 \cdot f_0(99) + f_0(99) = 9 \cdot f_0(99).$$

Слично, важи да је

$$f_0(99) = 9 \cdot f_0(9),$$

док је $f_0(9) = 9$ (јер у интервалу $[0, 9]$ који има 10 елемената, једино елемент 3 садржи цифру 3).

Важи и да је

$$[0, 251] = [0, 99] \cup [100, 199] \cup [200, 251],$$

па је

$$f_0(251) = 2 \cdot f_0(99) + f_0(51).$$

Пошто је

$$[0, 51] = [0, 9] \cup [10, 19] \cup [20, 29] \cup [30, 39] \cup [40, 49] \cup [50, 51],$$

важи да је

$$f_0(51) = 4 \cdot f_0(9) + f_0(1).$$

Важи и да је $f_0(1) = 2$ јер су оба елемента интервала $[0, 1]$ такви да не садрже цифру 3.

Дакле, $f_0(4251) = 3 \cdot f_0(999) + f_0(251) = 3 \cdot (9 \cdot f_0(99)) + 2 \cdot f_0(99) + f_0(51) = 3 \cdot (9 \cdot (9 \cdot f_0(9))) + 2 \cdot (9 \cdot f_0(9)) + 4 \cdot f_0(9) + f_0(1) = 3 \cdot 9 \cdot 9 \cdot 9 + 2 \cdot 9 \cdot 9 + 4 \cdot 9 + 2 = 2387$.

Функцију f_0 је могуће рекурзивно дефинисати. Ако се број n разлаже на почетну цифру c и суфикс n' тада се $f_0(n)$ може израчунати на следећи начин, у зависности од цифре c (претпостављамо да број $9 \ldots 9$ има онолико деветки колико цифара има број n').

- Ако је $c < 3$ тада је $f_0(n) = c \cdot f_0(9 \ldots 9) + f_0(n')$,
- Ако је $c = 3$ тада је $f_0(n) = c \cdot f_0(9 \ldots 9)$,
- Ако је $c > 3$ тада је $f_0(n) = (c - 1) \cdot f_0(9 \ldots 9) + f_0(n')$.

Излаз из рекурзије може бити и само случај $f_0(0) = 1$ (0 је једини број у интервалу $[0, 0]$ и он не садржи цифру 3).

Проблем са имплементацијом овакве рекурзивне функције је то што се цифре одвајају слева надесно, што је компликованије него здесна налево, када број n лако разлажемо на $n \operatorname{div} 10$ и $n \bmod 10$. Стога пре уласка у рекурзију можемо обрнути цифре броја. Такође, за дати број n потребно је одредити одговарајући број који се састоји само од деветки. То једноставно можемо решити тако што конструишемо број који се од броја n добија заменом свих цифара цифром 9 и ако тај број прослеђујемо као други параметар рекурзије (тај број у сваком позиву садржи само деветке и то онолико деветки колико цифара има број n).

Приметимо да се од једног рекурзивног позива за број са k цифара најчешће добијају два рекурзивна позива за бројеве са $k-1$ цифара, што указује на то да је сложеност експоненцијална (за основу 2), што је допустиво, јер је број цифара мали. Ипак, с обзиром на то да се исти рекурзивни позиви преклапају (пре свега они облика $f_0(9 \ldots 9)$), имплементација се може убрзати динамичким програмирањем или тако што се примети да је да је $f_0(\underbrace{9 \ldots 9}_{k}) = 9^k$, па би се ови рекурзивни позиви могли специјализовати.

```
int brojeviBezCifre3(int n, int d) {
  if (n == 0)
    return 1;
  int c = n % 10;
  if (c < 3)
    return c * brojeviBezCifre3(d / 10, d / 10) + brojeviBezCifre3(n / 10, d / 10);
  else if (c == 3)
    return c * brojeviBezCifre3(d / 10, d / 10);
  else
    return (c - 1) * brojeviBezCifre3(d / 10, d / 10) + brojeviBezCifre3(n / 10, d / 10);
}

int brojeviBezCifre3(int n) {
  int nObratno = 0;
  int devetke = 0;
  do {
    nObratno = nObratno * 10 + n % 10;
    devetke = devetke * 10 + 9;
    n /= 10;
  } while (n > 0);

  return brojeviBezCifre3(nObratno, devetke);
}
```

Уместо рекурзије која ради одозго наниже, решење можемо синтетизовати одоздо навише.

Обрађиваћемо једну по једну цифру броја n, здесна налево и мало по мало ћемо проширивати обрађени суфикс n' броја n. Уведимо променљиву, нпр. br, која током итерације чува вредности $f_0(n')$ за суфиксе n' броја n које током итерације обрађујемо, и променљиву, нпр. t, која чува вредности бројева $f_0(9\ldots9) = 9^k$, за бројеве $9\ldots9$ који имају k цифара 9, колико укупно цифара има у тренуном суфиксу n'.

Променљиве t и br иницијализујемо на 1 (претпостављамо да је пре петље обрађен празан суфикс који одговара броју 0 и да је $9^0 = 1$). Затим у петљи обрађујемо цифру по цифру броја n, кренувши од цифре јединица. Променљиву br ажурирамо на следећи начин, у зависности од текуће цифре c.

- Ако је $c < 3$ тада је $br = c \cdot t + br$,
- Ако је $c = 3$ тада је $br = c \cdot t$,
- Ако је $c > 3$ тада је $br = (c-1) \cdot t + br$.

Променљиву t ажурирамо на вредност $9 \cdot t$.

Размотримо поново пример израчунавања $f_0(4251)$ и применимо претходно описани алгоритам на број 4251.

Претпоставимо да на почетку променљиве t и br имају вредност 1.

Прво обрађујемо последњу (прву с десна) цифру броја n, а то је цифра 1. Пошто је она мања од 3, ажурирамо br на вредност $c \cdot t + br = 1 \cdot 1 + 1 = 2$, а вредност t на вредност $9 \cdot t = 9$. Након овога, променљива br има вредност $f_0(1)$, а променљива t има вредност $f_0(9)$.

Наредна цифра је 5 и пошто је она већа од 3, ажурирамо вредност br на $(c-1) \cdot t + br = 4 \cdot 9 + 2 = 38$, а вредност t на вредност $9 \cdot t = 81$. Након овога, променљива br има вредност $f_0(51)$, а променљива t има вредност $f_0(99)$.

Наредна цифра је 2 и пошто је она мања од 3, ажурирамо вредност br на $c \cdot t + br = 2 \cdot 81 + 38 = 200$, а вредност t на вредност $9 \cdot t = 9 \cdot 81 = 729$. Након овога, променљива br има вредност $f_0(251)$, а променљива t има вредност $f_0(999)$.

На крају, почетна цифра је 4 и пошто је она већа од 4, ажурирамо вредност br на $(c-1) \cdot t + br = 3 \cdot 729 + 200 = 2387$. Вредност t се ажурира на $9 \cdot 729 = 6561$, али се та вредност даље не користи. Након овога, променљива br има вредност $f_0(4251)$, а променљива t има вредност $f_0(9999)$.

```
int brojeviBezCifre3(int n) {
  int t = 1, br = 1;
  while (n > 0) {
    int c = n % 10;
    if (c < 3)
      br = c*t + br;
    else if (c == 3)
      br = c*t;
    else
      br = (c-1)*t + br;
    t = 9*t;
    n /= 10;
  }
  return br;
}
```

Задатак: Максимални збир несуседних елемената низа

Напиши програм који одређује највећи збир подниза датог низа целих бројева који не садржи два узастопна члана низа.

Улаз: Са стандардног улаза се уноси број чланова низа n ($1 \leq n \leq 10^5$), а затим из наредног реда чланови низа раздвојени размацима.

Излаз: На стандардни излаз исписати тражени максимални збир.

Пример 1

Улаз	*Излаз*
6	16
7 3 2 4 1 5	

Објашњење

Максимални збир је збир сегмента $7 + 4 + 5 = 16$.

Пример 2

Улаз

```
12
3 -2 4 5 -1 3 -4 -5 4 5 2 -1
```

Излаз

```
17
```

Решење

Задатак можемо једноставно решити индуктивно-рекурзивном конструкцијом. Низ можемо разложити на последњи елемент и префикс пре њега. Да бисмо одредили максимални збир несуседних елемената низа, анализираћемо случај када је последњи елемент низа део тог максималног збира и када није (то су једине две могућности и једна од њих је сигурно тачна). Зато је максимални збир несуседних елемената низа максимум следећа два збира:

- првог, који се добија тако што се последњи елемент дода на максимални збир несуседних елемената префикса, при чему у том збиру не укључује претпоследњи елемент низа (тј. последњи елемент префикса) и
- другог, који се добија као максимални збир несуседних елемената префикса који може да укључи и претпоследњи елемент низа (тј. последњи елемент префикса).

Дакле, осим што претпостављамо да умемо да решимо проблем мање димензије тј. да за префикс умемо да одредимо максимални збир несуседних елемената, потребно је да ојачамо индуктивну хипотезу и да за префикс умемо да одредимо и максимални збир несуседних елемената тог префикса ако последњи елемент префикса није укључен у тај збир. Наравно, као и увек када се ојачава индуктивна хипотеза, "дуг" се мора вратити па за префикс проширен додатним елементом поред максималног збира несуседних елемената, морамо да умемо да одредимо и максимални збир несуседних елемената када последњи елемент није укључен. Међутим, то није тешко, јер је то управо максимални збир несуседних елемената префикса. Максимални збир проширеног низа (без обзира на то да ли укључује или не укључује последњи елемент) одређујемо као максимум два описана збира (која на основу индуктивне хипотезе лако израчунавамо).

Излаз из рекурзије је случај празног низа. Максимални збир његових несуседних елемената у свакој варијанти једнак је нули..

Дакле, обележимо са m_k максимални збир несуседних елемената првих k елемената датог низа. Циљ је да одредимо m_n. Нека је m'_k максимали збир несуседних елемената првих k елемената низа у који није укључен елемент a_{k-1}. Важе следеће рекурентне релације: $m_0 = m'_0 = 0$, док за $k > 0$, важи $m'_k = m_{k-1}$ и $m_k = \max(a_{k-1} + m'_{k-1}, m_{k-1})$.

Анализа сложености. Сложеност овог алгоритма је $O(n)$.

```
int maks_zbir_bez = 0;
int maks_zbir = 0;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  int maks_zbir_sa = maks_zbir_bez + x;
  maks_zbir_bez = maks_zbir;
  maks_zbir = max(maks_zbir_bez, maks_zbir_sa);
}
cout << maks_zbir << endl;
```

Још једна индуктивно-рекурзивна конструкција којом се проблем може решити подразумева да се за сваки префикс низа одреди максимални збир несуседних елемената када је последњи елемент префикса укључен и максимални збир несуседних елемената када последњи елемент префикса није укључен.

Обележимо са m_k^{sa} максимални збир несуседних елемената првих k елемената низа, који садржи и елемент a_{k-1} и са m_k^{bez} максимални збир несуседних елемената првих k елемената низа, који не садржи елемент a_{k-1}. База индукције може бити једночлан низ и важи $m_1^{sa} = a_0$, $m_1^{bez} = 0$. За сваку вредност $k > 1$, важи $m_k^{sa} = a_{k-1} + m_{k-1}^{bez}$ и $m_k^{bez} = \max(m_{k-1}^{sa}, m_{k-1}^{bez})$. Тражена вредност једнака је $\max(m_n^{sa}, m_n^{bez})$.

Анализа сложености. Сложеност овог алгоритма је $O(n)$.

```
// prvi element se ucitava van petlje zbog inicijalizacije
// to je ujedno i baza indukcije, jednoclani prefiks niza
int x;
cin >> x;
int maks_zbir_bez = 0;
int maks_zbir_sa = x;

// u petlji obradjujemo tekuci element
for (int i = 1; i < n; i++) {
  int x;
  cin >> x;
  int maks_zbir = max(maks_zbir_sa, maks_zbir_bez);
  maks_zbir_sa = maks_zbir_bez + x;
  maks_zbir_bez = maks_zbir;
}

cout << max(maks_zbir_bez, maks_zbir_sa) << endl;
```

Индуктивно-рекурзивна конструкција може тећи и на следећи начин. Претпостављамо да умемо да израчунамо максимални збир сваког префикса низа. Ако је префикс празан, максимални збир је нула, а ако је једночлан, тада је једнак већем од броја нула и првог елемента низа. Ако је префикс бар двочлан онда је максимални збир несуседних елемената тог префикса максимум максималног збира префикса без последњег елемента и збира последњег елемента и максималног збира без последња два елемента.

Дакле, добијамо да је $m_0 = 0$, $m_1 = \max(a_0, 0)$ и $m_i = \max(m_{i-1}, a_{i-1} + m_{i-2})$.

Ова рекурзивна конструкција се може испрограмирати рекурзивном функцијом. Нажалост, то решење је прилично неефикасно.

Анализа сложености. Сложеност овог решења задовољава једначину $T(n) = T(n-1) + T(n-2) + O(1)$, $(1) = T(0) = 1$, чије је решење експоненцијална функција (иста једначина се јавља и приликом директне рекурзивне имплементације израчунавања елемената Фибоначијевог низа).

```
int maksZbir(const vector<int>& a, int n) {
  if (n == 0)
    return 0;
  if (n == 1)
    return max(a[0], 0);
  return max(maksZbir(a, n-1), a[n-1] + maksZbir(a, n-2));
}
```

Рекурзију можемо уклонити и добити наредну итеративну имплементацију.

Анализа сложености. Сложеност овог алгоритма је $O(n)$.

```
int maks_zbir_p = 0;
int x;
cin >> x;
int maks_zbir = max(0, x);
for (int i = 2; i <= n; i++) {
  int x;
  cin >> x;
  int tmp = max(maks_zbir, maks_zbir_p + x);
```

```
  maks_zbir_p = maks_zbir;
  maks_zbir = tmp;
}
cout << maks_zbir << endl;
```

Напомена. Систематичан начин ослобађања неефикасности проузроковане рекурзијом овог типа долази у облику техника динамичког програмирања, које је описано у засебном поглављу.

Задатак: Максимални збир сегмента

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Максимални суфикси

С обзиром на то да се суфикси првих i елемената низа могу анализирати инкрементално (суфикси $i+1$ елемената низа се добијају проширивањем суфикса i елемената низа додатним елементом), проблем анализе свих сегмената је пожељно свести на проблем анализе суфикса. У овом конкретном проблему, можемо приметити да је максимални збир сегмента уједно и максимални збир суфикса који се завршава на позицији на којој се тај сегмент завршава. Стога се задатак може свести на то да се за сваку позицију у низу одреди максимални суфикс, а да се онда међу максималним суфиксима за сваку позицију пронађе онај који је глобално максимални.

Једини суфикс првих нула елемената низа је празан и његов збир је по дефиницији 0. Сви суфикси првих $i+1$ елемената низа, изузев празног суфикса добијају се додавањем елемента на позицији i на крај неког суфикса првих i елемената низа. Међу непразним суфиксима највећи збир има онај који је добијен додавањем последњег елемента управо на суфикс првих i елемената низа који има максимални збир. Од њега једино може бити већи збир празног суфикса (и то када се након проширивања последњим елементом добије негативни збир). Ако вредности максималног збира суфикса памтимо у низу, тада низ лако попуњавамо на основу веза $S_0 = 0$ и $S_{i+1} = \max{(S_i + a_i, 0)}$, где је са S_i обележена вредност максималног збира суфикса првих i елемената низа a.

На крају налазимо максимум низа S_i.

Пример. Прикажимо рад алгоритма на примеру низа `-2 3 2 -3 -3 -2 4 5 -8 3`. У таблици попуњавамо вредности S_i.

```
i    a_{i+1}   S_i
0              0
1    -2        0 = max(0+(-2), 0)
2     3        3 = max(0+3, 0)
3     2        5 = max(3+2, 0)
4    -3        2 = max(5+(-3), 0)
5    -3        0 = max(2+(-3), 0)
6    -2        0 = max(0+(-2), 0)
7     4        4 = max(0+4, 0)
8     5        9 = max(4+5, 0)
9    -8        1 = max(9+(-8), 0)
10    3        4 = max(1+3, 0)
```

Максимална вредност у колони S_i је 9.

Анализа сложености. Пошто користимо низ максималних збирова суфикса, меморијска сложеност је $O(n)$. Низ попуњавамо елемент по елемент, инкрементално, у једном пролазу за шта је довољно n корака, а затим максимум налазимо у новом пролазу тј. у нових n корака. Укупна сложеност је, дакле, линеарна тј. $O(n)$.

```
// maksimalni sufiks prvih i elemenata niza
vector<int> maxSufiks(n+1);
maxSufiks[0] = 0;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
```

```
  maxSufiks[i+1] = max(maxSufiks[i] + x, 0);
}

// maksimalni segment je maksimalni od svih maksimalnih sufiksa
cout << *max_element(begin(maxSufiks), end(maxSufiks)) << endl;
```

Каданов алгоритам

Максималне вредности збирова суфикса не морамо да памтимо у низу, ако њихов максимум одређујемо истовремено са одређивањем вредности максималних збирова суфикса. Овај алгоритам познат је под именом *Каданов алгоритам*.

Један начин да се дође до тог алгоритма је следећи. Покушавамо да алгоритам заснујемо на индуктивној конструкцији.

- За празан низ, једини сегмент је празан и његов је збир нула (то је уједно највећи збир који се може добити).
- Сматрамо да умемо да проблем решимо за произвољан низ дужине n и на основу тога покушавамо да решимо задатак за низ дужине $n+1$ (полазни низ проширен једним додатним елементом).

  Сегмент највећег збира у проширеном низу се или цео садржи у полазном низу дужине n или чини суфикс проширеног низа, тј. завршава се на последњој позицији (укључујући и могућност да је ту и празан сегмент).

  На основу индуктивне хипотезе знамо да израчунамо највећи збир сегмента низа дужине n и потребно је да још одредимо максимални збир суфкса проширеног низа. Један начин да се то уради је да приликом сваког проширења низа изнова анализирамо све сегменте који се завршавају на текућој позицији, али чак иако то радимо инкрементално (кренувши од празног суфикса, па додајући уназад један по један елемент) највише што можемо добити је алгоритам квадратне сложености (пробајте да се уверите да је то заиста тако). Кључни увид је то да највећи збир суфикса који се завршава на текућој позицији можемо инкрементално израчунати знајући највећи збир суфикса низа пре проширења. Највећи збир неког непразног суфикса који се завршава на текућој позицији је збир текућег елемента низа и највећег збира неког суфикса који се завршава на претходној позицији. Од њега може бити повољнији само празан суфикс (и то само ако је претходни збир негативан).

Дакле, ако са S_i обележимо максимални збир неког суфикса првих i елемената низа, а са M_i максимални збир неког сегмента прих i елемената низа, важи да је $M_0 = S_0 = 0$, да је $S_{i+1} = \max(S_i + a_i, 0)$ и $M_{i+1} = \max(M_i, S_{i+1})$.

Имплементацију можемо направити итеративним алгоритмом коме је инваријанта да у сваком кораку петље знамо ове две вредности (максимум сегмента и максимум суфикса).

Пример. Прикажимо рад алгоритма на примеру низа `-2 3 2 -3 -3 -2 4 5 -8 3`. У таблици попуњавамо вредности S_i и M_i.

```
i    a_{i+1}   S_i                      M_i
0              0                        0
1    -2        0 = max(0+(-2), 0)       0 = max(0, 0)
2     3        3 = max(0+3, 0)          3 = max(0, 3)
3     2        5 = max(3+2, 0)          5 = max(3, 5)
4    -3        2 = max(5+(-3), 0)       5 = max(5, 2)
5    -3        0 = max(2+(-3), 0)       5 = max(5, 0)
6    -2        0 = max(0+(-2), 0)       5 = max(5, 0)
7     4        4 = max(0+4, 0)          5 = max(5, 4)
8     5        9 = max(4+5, 0)          9 = max(5, 9)
9    -8        1 = max(9+(-8), 0)       9 = max(9, 1)
10    3        4 = max(1+3, 0)          9 = max(9, 4)
```

Анализа сложености. Пошто елементе учитавамо један по један и не памтимо их истовремено, меморијска сложеност је $O(1)$. Максимални збир сегмента и суфикса инкрементално израчунавамо једним проласком кроз задате елементе и временска сложеност је линеарна тј. $O(n)$.

Напомена. Приметимо да смо у претходном разматрању проширили индуктивну хипотезу претпостављајући да поред тражене вредности тј. максимума неког сегмента првих n елемената низа знамо додатно и вредност

максималног суфикса првих n елемената низа.

```
int maxSufiks = 0, max = maxSufiks;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  maxSufiks += x;
  if (maxSufiks < 0)
    maxSufiks = 0;
  if (maxSufiks > max)
    max = maxSufiks;
}
cout << max << endl;
```

Види другачија решења овог задатка.

Задатак: Број растућих сегмената

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Индуктивна конструкција

Најлепше решење можемо добити ако употребимо инкрементално проширивање сегмената једним по једним елементом здесна. Укупан број растућих сегмената у низу можемо добити тако што за сваку позицију низа израчунамо број R растућих сегмената који се завршавају на тој позицији и тако добијене бројеве саберемо.

На позицији 0 се не завршава ни један растући сегмент (јер они по дефиницији морају да буду бар двочлани). Дакле, важи $R_0 = 0$. Ако знамо број растућих сегмената који се завршавају на позицији j, тада веома једноставно можемо израчунати број растућих сегмената који се завршавају на позицији $j + 1$.

- Ако је $a_{j+1} > a_j$, онда се сваки растући сегмент који се завршава на позицији j може проширити елементом a_{j+1} и тако добити нови растући сегмент. Додатно, растући сегмент је и $[a_j, a_{j+1}]$, тако да је укупан број сегмената који се завршавају на позицији $j + 1$ за један већи од укупног броја растућих сегмената који се завршавају на позицији j и важи $R_{j+1} = R_j + 1$.
- Ако је $a_{j+1} \leq a_j$ онда се на позицији j не завршава ни један растући сегмент тј. важи $R_{j+1} = 0$.

Анализа сложености. Број растућих сегмената за сваки десни крај и њихов укупан број одређујемо једним проласком кроз низ, у сложености $O(n)$. Пошто памтимо само да узастопна члана низа, меморијска сложеност је $O(1)$.

```
int n;
cin >> n;
int prethodni;
cin >> prethodni;
// ukupan broj rastucih serija
long long ukupanBrojRastucih = 0;
// broj rastucih koji se zavrsavaju na tekucoj poziciji
long long brojRastucih = 0;
for(int i = 1; i < n; i++) {
  int tekuci;
  cin >> tekuci;
  if (tekuci > prethodni) {
    // tekuci element produzava sve rastuce segmente koji su se zavrsili na
    // prethodnoj poziciji i dodaje jos jedan nov dvoclan rastuci segment
    brojRastucih++;
    // dodajemo broj rastucih koji se zavrsavaju na poziciji i na
    // ukupan broj rastucih segmenata
    ukupanBrojRastucih += brojRastucih;
```

```
  } else {
    // na tekucoj poziciji se ne zavrsava ni jedna rastuca serija
    brojRastucih = 0;
  }
  prethodni = tekuci;
}
cout << ukupanBrojRastucih << endl;
```

Задатак: Фактори равнотеже бинарног дрвета

Дефинишимо висину празног дрвета као 0, а непразног дрвета као број чворова од корена тог дрвета до њему најудаљенијег листа (рачунајући и корен и тај лист). Дефинишимо фактор равнотеже чвора бинарног дрвета као апсолутну разлику између висина његовог левог и десног потомка. Напиши програм који одређује највећи фактор равнотеже неког чвора учитаног бинарног дрвета.

Улаз: Празно бинарно дрво се задаје карактером -, а непразно у облику [k <levo> <desno>], где је k цео број (вредност у корену), а <levo> и <desno> су лево и десно поддрво записани у истом формату.

Излаз: На стандардни излаз исписати вредност највећег фактора равнотеже.

Пример 1

Улаз	*Излаз*
[1 [6 - -] [3 [2 [7 - [8 - -]] -] [4 - [5 - -]]]]	3

Објашњење

Учитано дрво је приказано на слици.

Слика 3.1: Дрво

Највећи фактор равнотеже је у корену – лево подрво је висине 1, а десно висине 4, па је фактор једнак 3.

Пример 2

Улаз

```
[1 [2 [3 [4 [5 [6 [7 [8 [9 [10 - -] -] -] -] -] -] -] -] -] -]
```

Излаз

```
9
```

Решење

Директна индуктивно-рекурзивна конструкција у овом примеру не даје резултате јер фактор равнотеже чвора не зависи од фактора равнотеже левог и десног поддрвета (тј. њихових корених чворова), већ од висине тих дрвета. Са друге стране, висину дрвета веома једноставно можемо израчунати на основу индуктивно-рекурзивне конструције (висина празног дрвета је 0, а висина непразног дрвета је за један већа од максимума висина левог и десног поддрвета).

Са функцијом за израчунавање висине на располагању лако можемо дефинисати рекурзивну функцију која израчунава највећи фактор равнотеже. Највећи фактор равнотеже празног дрвета је 0, а највећи фактор

равнотеже непразног дрвета је максимум највећег фактора равнотеже левог подрвета, највећег фактора равнотеже десног поддрвета и фактора равнотеже корена, који израчунавамо као апсолутну разлику висина левог и десног поддрвета.

Анализа сложености. Ако претпоставимо да је дрво око сваког свог чвора уравнотежено тј. да је у сваком поддрвету пола чворова лево, а пола чворова десно од корена, функција која израчунава висину дрвета са n чворова задовољава једначину $T(n) = 2 \cdot T(n/2) + O(1)$, чије је решење на основу мастер теореме $O(n)$. Сложеност је таква и без те претпоставке (ако је дрво дегенерисано у листу, сложеност задовољава једначину $T(n) = T(n-1) + O(1)$ чије је решење такође $O(n)$). И интуитивно је јасно да функција у сваком позиву мора да обиђе и лево и десно подстабло да би одредила висину, тако да функција обилази све чворове (једном) и стога је линеарне сложености у односу на број чворова стабла.

Под претпоставком да је дрво око сваког чвора уравнотежено, сложеност функције која проналази највећи фактор равнотеже задовољава једначину $T(n) = 2 \cdot T(n/2) + O(n)$ и њена је сложеност, знамо на основу мастер теореме, $O(n \log n)$. Ако је дрво издегенерисано у листу (што се може десити у најгорем случају), добија се да време извршавања задовољава једначину $T(n) = T(n-1) + O(n)$ чије је решење $O(n^2)$. Рецимо да кључни проблем настаје услед тога што се висина делова дрвета израчунава изнова и изнова.

Нагласимо и да је сложеност учитавања и креирања дрвета, као и ослобађања линеарна и износи $O(n)$, али време може бити значајно услед великих константних фактора насталих услед коришћења релативно скупих операција алокације и деалокације меморије.

```
int visina(cvor* koren) {
  // visina praznog stabla je nula
  if (koren == nullptr)
    return 0;
  // visina untrašnjeg čvora
  return max(visina(koren->levo), visina(koren->desno)) + 1;
}

// izracunava faktore ravnoteze svih cvorova
int maxFaktorRavnoteze(cvor* koren) {
  if (koren == nullptr)
    return 0;

  int maxFaktorLevo = maxFaktorRavnoteze(koren->levo);
  int maxFaktorDesno = maxFaktorRavnoteze(koren->desno);
  int faktorKoren = abs(visina(koren->levo) - visina(koren->desno));

  return max({maxFaktorLevo, maxFaktorDesno, faktorKoren});
}
```

Алгоритам се може побољшати ако се висина и фактор равнотеже рачунају истовремено, тј. ако појачамо индуктивну хипотезу и претпоставимо да рекурзивним позивима можемо да израчунамо и факторе равнотеже и висину поддрвета. Базу чини случај празног стабла чија је висина нула и које не садржи чворове чији фактор равнотеже треба израчунати. Ако претпоставимо да умемо да израчунамо факторе равнотеже свих чворова поддрвета, као и њихове висине, тада фактор равнотеже корена можемо једноставно израчунати као апсолутну вредност разлике тих висина, а висину целог дрвета као број за један већи од максимума висине левог и десног поддрвета.

Анализа сложености. Под претпоставком да је дрво балансирано, број корака у извршавању овог алгоритма задовољава једначину $T(n) = 2 \cdot T(n/2) + O(1)$, чије је решење $O(n)$. Чак и када дрво није балансирано, сложеност је линеарна (јер је решење једначине $T(n) = T(n-1) + O(1)$ такође $O(n)$).

```
// izracunava faktore ravnoteze svih cvorova
void visinaIMaxFaktorRavnoteze(cvor* koren, int& visina, int& maxFaktor) {
  if (koren == nullptr) {
    visina = 0;
    maxFaktor = 0;
    return;
  }
```

```
  int visinaLevo, maxFaktorLevo;
  visinaIMaxFaktorRavnoteze(koren->levo, visinaLevo, maxFaktorLevo);
  int visinaDesno, maxFaktorDesno;
  visinaIMaxFaktorRavnoteze(koren->desno, visinaDesno, maxFaktorDesno);

  int faktorKoren = abs(visinaLevo - visinaDesno);

  maxFaktor = max({maxFaktorLevo, maxFaktorDesno, faktorKoren});
  visina = max(visinaLevo, visinaDesno) + 1;
}

int maxFaktorRavnoteze(cvor* koren) {
  int visina, maxFaktor;
  visinaIMaxFaktorRavnoteze(koren, visina, maxFaktor);
  return maxFaktor;
}
```

Задатак: Дијаметар бинарног дрвета

Дефинишимо дијаметар бинарног дрвета као број чворова на путањи између два најудаљенија чвора (то ће сигурно бити листови), рачунајући и те чворове. Напиши програм који учитава бинарно дрво и израчунава његов дијаметар.

Улаз: Празно бинарно дрво се задаје карактером `-`, а непразно у облику `[k <levo> <desno>]`, где је `k` цео број (вредност у корену), а `<levo>` и `<desno>` су лево и десно поддрво записани у истом формату.

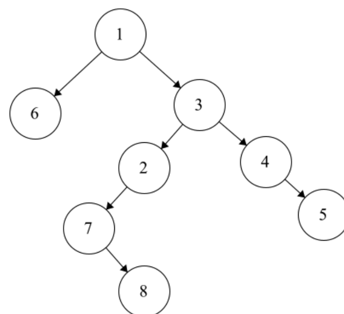
Излаз: На стандардни излаз исписати тражени дијаметар учитаног дрвета.

Пример

Улаз	*Излаз*
`[1 [6 - -] [3 [2 [7 - [8 - -]] -] [4 - [5 - -]]]]`	`6`

Објашњење

Учитано дрво је приказано на слици.



Слика 3.2: Дрво

Највеће растојање је 6 и оно се достиже између чворова 8 и 5.

Решење

Два најудаљенија чвора (листа) могу да буду оба у левој поддрвету, оба у десном подрвету или један може да буде у левом, а један у десном поддрвету. У прва два случаја пут између њих не садржи корен, а у трећем случају пут између њих садржи корен. У прва два случаја дијаметар дрвета једнак је дијаметру левог односно десног дрвета и може бити израчунат рекурзивно. У трећем случају знамо да најдужи пут садржи корен дрвета, као и путеве од корена левог дрвета до неког његовог листа и од корена десног дрвета до неког његовог листа, при чему ти путеви треба да буду што дужи. Међутим, најдужи пут од корена до листа у дрвету је управо висина тог дрвета, па је најдужи пут који пролази кроз корен дрвета за један већи од збира висина левог и десног поддрвета.

Висину дрвета лако можемо израчунати посебном рекурзивном функцијом (висина празног дрвета је 0, а висина непразног дрвета је за један већа од максимума висина левог и десног поддрвета).

Са функцијом за израчунавање висине на располагању, лако је дефинисати рекурзивну функцију која израчунава дијаметар. Дијаметар празног дрвета је 0, а у супротном је максимум дијаметра левог подрвета, дијаметра десног поддрвета и збира висина тих поддрвета увећаног за 1.

Анализа сложености. Ако претпоставимо да је дрво око сваког свог чвора уравнотежено тј. да је у сваком поддрвету пола чворова лево, а пола чворова десно од корена, функција која израчунава висину дрвета са n чворова задовољава једначину $T(n) = 2 \cdot T(n/2) + O(1)$, чије је решење на основу мастер теореме $O(n)$. Сложеност је таква и без те претпоставке (ако је дрво дегенерисано у листу, сложеност задовољава једначину $T(n) = T(n-1) + O(1)$ чије је решење такође $O(n)$). И интуитивно је јасно да функција у сваком позиву мора да обиђе и лево и десно подстабло да би одредила висину, тако да функција обилази све чворове (једном) и стога је линеарне сложености у односу на број чворова стабла.

Под претпоставком да је дрво око сваког чвора уравнотежено, сложеност функције која проналази дијаметар задовољава једначину $T(n) = 2 \cdot T(n/2) + O(n)$ и њена је сложеност, знамо на основу мастер теореме, $O(n \log n)$. Ако је дрво издегенерисано у листу (што се може десити у најгорем случају), добија се да време извршавања задовољава једначину $T(n) = T(n-1) + O(n)$ чије је решење $O(n^2)$. Рецимо да кључни проблем настаје услед тога што се висина делова дрвета израчунава изнова и изнова.

```
int visina(cvor* koren) {
  if (koren == nullptr)
    return 0;
  return 1 + max(visina(koren->levo), visina(koren->desno));
}

int dijametar(cvor* koren) {
  if (koren == nullptr)
    return 0;
  int dijametar_levo = dijametar(koren->levo);
  int dijametar_desno = dijametar(koren->desno);
  return max({dijametar_levo, dijametar_desno, 1 + visina(koren->levo) + visina(koren->desno)});
}
```

Ефикасније решење се добија ако се уместо засебних функција за израчунавање висине и дијаметра, индуктивна хипотеза ојача и дефинише се функција која истовремено израчунава висину и дијаметар.

Анализа сложености. Под претпоставком да је дрво балансирано, број корака у извршавању овог алгоритма задовољава једначину $T(n) = 2 \cdot T(n/2) + O(1)$, чије је решење $O(n)$. Чак и када дрво није балансирано, сложеност је линеарна (јер је решење једначине $T(n) = T(n-1) + O(1)$ такође $O(n)$).

```
void dijametar_i_visina(cvor* koren, int& dijametar, int& visina) {
  if (koren == nullptr) {
    dijametar = 0;
    visina = 0;
  } else {
    int dijametar_levo, visina_levo;
    dijametar_i_visina(koren->levo, dijametar_levo, visina_levo);
    int dijametar_desno, visina_desno;
    dijametar_i_visina(koren->desno, dijametar_desno, visina_desno);
    dijametar = max({dijametar_levo, dijametar_desno, 1 + visina_levo + visina_desno});
    visina = 1 + max(visina_levo, visina_desno);
  }
}

int dijametar(cvor* koren) {
  int d, v;
  dijametar_i_visina(koren, d, v);
  return d;
}
```