Глава 1

Анализа коректности алгоритама

Делови текста у наставку су преузети из скрипте "Програмирање 2", аутора Предрага Јаничића и Филипа Марића.

Исправност тј. коректност је суштинска особина алгоритама и програма. Иако се некада у пракси користе програми за које се зна да понекад могу да дају и нетачне резултате, то најчешће није случај и од програма се захтева да буде практично апсолутно непогрешив.

Једно од централних питања у развоју програма је питање његове исправности (коректности). Софтвер је у данашњем свету присутан на сваком кораку: софтвер контролише много тога — од банковних рачуна и компоненти телевизора и аутомобила, до нуклеарних електрана, авиона и свемирских летелица. У свом том софтверу неминовно су присутне и грешке. Грешка у функционисању даљинског управљача за телевизор може бити тек узнемирујућа, али грешка у функционисању нуклеарне електране може имати разорне последице. Најопасније грешке су оне које могу да доведу до великих трошкова, или још горе, до губитка људских живота. Неке од катастрофа које су општепознате су експлозија ракете Ариане 1996. узрокована конверзијом броја из шездесетчетворобитног реалног у шеснаестобитни целобројни запис која је довела до прекорачења, затим пад сателита Криосат (енгл. Cryosat) 2005. године услед грешке у софтверу због које није на време дошло до раздвајања сателита и ракете која га је носила коштао је Европску Унију око 135 милиона евра, затим грешка у нумеричком копроцесору процесора Pentium 1994. узрокована погрешним индексима у петљи `for` у оквиру софтвера који је радио дизајн чипа, као и пад орбитера послатог на Марс 1999. узрокован чињеницом да је део софтвера користио метричке, а део софтвера енглеске јединице. Међутим, фаталне софтверске грешке и даље се непрестано јављају и оне коштају светску економију милијарде долара. Ево неких од најзанимљивијих:

- Не нарочито опасан, али веома занимљив пример грешке је грешка у програму Microsoft Excel 2007 који, због грешке у алгоритму форматирања бројева пре приказивања, резултат израчунавања израза `77.1 * 850` приказује као `100 000` (иако је интерно коректно сачуван).
- У Лос Анђелесу је 14. септембра 2004. године више од четиристо авиона у близини аеродрома истовремено изгубило везу са контролом лета. На срећу, захваљујући резервној опреми унутар самих авиона, до несреће ипак није дошло. Узрок губитка везе била је грешка прекорачења у бројачу милисекунди у оквиру система за комуникацију са авионима. Да иронија буде већа, ова грешка је била откривена раније, али пошто је до открића дошло када је већ систем био испоручен и инсталиран на неколико аеродрома, његова једноставна поправка и замена није била могућа. Уместо тога, препоручено је да се систем ресетује сваких 30 дана како до прекорачења не би дошло. Процедура није испоштована и грешка се јавила после тачно 2^{32} милисекунди, односно 49,7 дана од укључивања система.
- Више од пет процената пензионера и прималаца социјалне помоћи у Немачкој је привремено остало без свог новца када је 2005. године уведен нови рачунарски систем. Грешка је настала због тога што је систем, који је захтевао десетоцифрени запис свих бројева рачуна, код старијих рачуна који су имали осам или девет цифара бројеве допуњавао нулама, али са десне уместо са леве стране како је требало.
- Једна бака у Америци је на свој 106. рођендан добила позив да мора да крене у школу, јер је систем бележио године помоћу две цифре.
- Компаније Dell и Apple морале су током 2006. године да корисницима замене више од пет милиона лап-

топ рачунара због грешке у дизајну батерије компаније Sony која је узроковала да се неколико рачунара запали.

1.1 Облици испитивања коректности

У развијању техника верификације програма, потребно је најпре прецизно формулисати појам коректности тј. исправности програма. Исправност програма почива на појму **спецификације**. Спецификација је, неформално, опис жељеног понашања програма који треба написати. Спецификација се обично задаје у терминима **предуслова** тј. услова које улазни параметри програма задовољавају, као и **постуслова** тј. услова које резултати израчунавања морају да задовоље. Када је позната спецификација, потребно је верификовати програм, тј. доказати да он задовољава спецификацију.

Коректност се огледа кроз два аспекта:

парцијална коректност: свака вредност коју алгоритам израчуна за улазне параметре који задовољавају спецификацију (тј. предуслов) мора да задовољи спецификацију (тј. постуслов).

заустављање алгоритам мора да се заустави за све улазе који задовољавају спецификацију (тј. предуслов).

Већина алгоритама које ћемо проучавати у овом курсу биће потпуни тј. заустављаће се за све допуштене улазе. За заустављајуће парцијално коректне алгоритме кажемо да су **тотално коректни**. Интересантно, доказано је да не постоје алгоритми којима би се испитивала горе наведена својства алгоритама.

Поступак показивања да је програм исправан назива се **верификовање програма**. Два основна приступа верификацији су:

динамичка верификација подразумева проверу исправности у фази извршавања програма, најчешће путем тестирања;

статичка верификација подразумева анализу изворног кода програма, често коришћењем формалних метода и математичког апарата.

Систематично тестирање је сигурно најзначајнији облик постизања високог степена исправности програма. Тестирањем на већем броју исправних улаза и упоређивањем добијених и очекиваних резултата може се открити велики број грешака. Нагласимо и да се тестирањем не може показати да је програм коректан, већ само да није коректан. Наиме, практично никада није могуће испитати понашање програма баш на свим исправним улазима. Већ програм који сабира два 32-битна броја има $2^{32} \cdot 2^{32} = 2^{64}$ исправних комбинација улазних параметара и исцрпно тестирање оваквог програма би трајало годинама. Зато се исцрпно тестирање скоро никада не спроводи, већ се програми тестирају тако да се улази бирају пажљиво, тако да покрију различите гране током извршавања програма. Обично се додатно посебно тестира понашање програма на неким граничним улазима (енгл. edge cases, corner cases), јер програми понекада не обрађују све специјалне случајеве како би требало. Многи системи за учење програмирања (такозвани грејдери, енгл. grader) оцењују ученичка решења тестирањем на већем броју тест-примера и савет почетницима је да током учења обавезно користе овакве системе.

О методама статичке верификације биће више речи у наставку овог поглавља.

1.2 Неке честе грешке у програмима

Сваки исправан програм мора да буде заснован на исправном алгоритму. Дакле, од неисправног алгоритма није могуће направити исправан програм и основна ствар приликом писања исправних програма је да се обезбеди исправност алгоритма који се примењује. Са друге стране, алгоритми се описују често на апстрактнијем нивоу него што су сами програми, и многи детаљи се занемарују. Зато се услед детаља имплементације од исправног алгоритма може добити неисправан програм. Наведимо неке честе грешке.

- Једна од најчешћих грешака представља **грешка прекорачења** (енгл. overflow). Наиме, ако се у имплементацији одабере бројевни тип података којим се не могу исправно представити сви подаци, тада програм даје неисправне резултате. Чест је случај да програмер одабере тип података који може да репрезентује исправно и улазне и излазне вредности, међутим, дешава се да међурезултати не могу да се репрезентују исправно, што се теже примети, а доводи до грешке.

- Честа грешка је **прекорачење граница низа** (енгл. buffer overflow). На пример, ако смо у низу одвојили место за 30 бројева, онда је могуће уписивати вредности само на позиције $0, 1, \ldots, 29$. Нарочито је критична позиција 30 (тј. у општем случају позиција n за низ од n елемената). Пошто у савременим програмским језицима бројање позиција у низовима креће од нуле, на позицију n није могуће уписивати вредности. У језику C++ се не врши провера опсега пре приступа елементима низа (тј. вектора) и одговорност је програмера да обезбеди да се не приступа ван граница - у супротном је понашање програма недефинисано, што значи да програм може да настави да ради неисправно и после одређеног броја инструкција да буде прекинут од стране оперативног система, али и да грешка може да прође неопажено. Уколико у петљи у низ уписујемо податке чији број не знамо унапред, неопходно је да пре сваког уписа проверимо да ли се упис врши унутар граница низа (или да користимо неки облик низа који допушта аутоматско проширивање додавањем нових елемената).
- Честа грешка је необраћање пажње на **специјалне случајеве**. На пример, ако у низу тражимо први елемент који задовољава неки услов, неопходно је да обезбедимо да програм коректно ради и у случају када ниједан елемент не задовољава тај услов. Треба пажљиво прецизирати да ли функција тада треба да врати број елемената низа или, на пример, -1, и треба осигурати да се у коду који позива ову функционалност добро реагује на ситуацију у којој тражени елемент не постоји. Специјални случајеви најчешће настају када неке вредности не постоје (када је неки скуп чије елементе разматрамо празан), затим када су улазне вредности у неком специјалном односу (на пример, да ли геометријски програм исправно ради ако су унете тачке колинеарне) и слично. При том, треба пажљиво прецизирати спецификацију задатка и одредити који специјални случајеви јесу, а који нису допуштени спецификацијом. Поново је потребно обратити пажњу на то да иако улазни параметри понекада не могу бити у неком специјалном односу, то не значи да међурезултати неће бити у том односу, па је онда потребно програме ипак прилагодити да обрате пажњу на све специјалне случајеве.

1.3 Индуктивно-рекурзивна конструкција

Кључна идеја у конструкцији алгоритама је то да је конструкција алгоритама веома тесно повезана са доказивањем теорема математичком индукцијом. **Математичка индукција**, у свом основном облику, је следећи начин доказивања особина природних бројева. Нека је P произвољно својство које се може формулисати за природне бројеве. Тада важи

$$(P(0) \wedge (\forall n)(P(n) \Rightarrow P(n+1))) \Rightarrow (\forall n)(P(n))$$

Дакле, да бисмо доказали да сваки природан број има неко својство P (тј. да бисмо доказали $(\forall n)(P(n))$), довољно је да докажемо да нула има то својство (тј. $P(0)$) и да докажемо да чим неки број има то својство, има га и његов следбеник (тј. да докажемо $(\forall n)(P(n) \Rightarrow P(n+1))$). Прво тврђење се назива **база индукције**, а друго **индуктивни корак**. Приницип математичке индукције је прилично јасан – на основу базе знамо да 0 има својство P, на основу корака да њен следбеник тј. 1 има својство P, на основу корака да његов следбеник тј. 2 има својство P итд. Интуитивно нам је јасно да на овај начин можемо стићи до било ког природног броја, који сигурно мора имати својство P. База се може формулисати и за веће вредности од нуле, али онда само можемо да тврдимо да елементи који су већи или једнаки од базе имају својство P.

Основни приступ конструкције алгоритама је тзв. **индуктивни** тј. **рекурзивни** приступ. Он у свом основном облику подразумева да се решење проблема веће димензије проналази тако што умемо да решимо проблем истог облика, али мање димензије и да од решења тог проблема добијемо решење проблема веће димензије (у напреднијим облицима је могуће и да се решава већи број проблема мање димензије). Притом за почетне димензије проблема решење морамо да израчунавамо директно, без даљег свођења на проблеме мање димензије. Ако се приликом свођења димензија проблема увек смањује, конструисани алгоритми ће се увек заустављати.

- Имплементација алгоритма може бити таква да промењиве унутар петље итеративно ажурирају своје вредности кренувши од вредности које представљају решења елементарних проблема, па до крајњих вредности које представљају решења задатог проблема. Пошто је ово прилично слично принципу математичке индукције, кажемо да је алгоритам дефинисан **индуктивно**.
- Имплементација може бити таква да функција која решава полазни проблем сама себе позива да би решила проблем истог облика, али мање димензије (осим у случају елементарних проблема, који се директно решавају) и тада кажемо да је алгоритам дефинисан **рекурзивно**.

Индуктивна конструкција лежи у основни практично свих итеративних алгоритама које смо до сада разматрали. На пример, алгоритам израчунавања збира серије бројева (на пример, збира елемената неког низа) почива на томе да знамо да израчунамо збир празне серије (то је 0) и да ако знамо збир серије од k елемената, тада умемо да израчунамо и збир серије која се добија проширивањем те серије додатним $k+1$-вим елементом (то радимо тако што дотадашњи збир увећамо за тај нови елемент).

```
int zbir = 0;
for (int i = 0; i < a.size(); i++)
    zbir = zbir + a[i];
```

Дакле, и у овом алгоритму имамо индуктивну базу (која одговара иницијализацији промењиве пре уласка у петљу) и индуктивни корак (који одговара телу петље, у ком се ажурира вредност резултујуће промењиве, у овом случају збира). База може одговарати и случају једночланог (а не обавезно празног) низа, али тада не можемо да гарантујемо да ће алгоритам радити исправно у случају празног низа. То одговара варијанти алгоритма у којој збир иницијализујемо на први елемент низа, па га увећавамо редом за један по један елемент од позиције 1 надаље.

Рекурзивна имплементација израчунавања збира елемената низа може бити следећа (у њој се приликом решавања проблема димензије $n > 0$ експлицитно захтева решавање проблема димензије $n - 1$).

```
int zbir(int a[], int n) {
    if (n == 0)
       return 0;
    else
       return zbir(a, n-1) + a[n-1];
}
```

Дефинисање алгоритама индуктивно-рекурзивом конструкцијом је у веома тесној вези са доказивањем њихове коректности. Иако постоје формални оквири за доказивање коректности императивних програма (пре свега *Хорова логика*), ми ћемо се бавити искључиво неформалним доказима и веза између логике у којој вршимо доказивање и (императивног) програмског језика у којем се програм изражава биће прилично неформална.

Рецимо и да ћемо приликом доказивања коректности програма обично игнорисати ограничења записа бројева у рачунару и подразумеваћемо да је опсег бројева неограничен и да се реални бројеви записују са максималном прецизношћу. Дакле, нећемо обраћати пажњу на грешке које могу настати услед прекорачења или поткорачења вредности током извођења аритметичких операција (иако реално то често може бити узрок грешака у програмима).

1.3.1 Доказ коректности рекурзивних функција

Проблем: Дефинисати функцију која одређује минимум непразног низа бројева и доказати њену коректност.

Алгоритам се веома једноставно конструише индуктивно-рекурзивном конструкцијом. Димензија проблема у овом примеру је број елемената низа.

База: Ако низ има само један елемент, тада је тај елемент уједно и минимум.

Корак: У супротном, претпоставимо да некако умемо да решимо проблем за мању димензију и на основу тога покушајмо да добијемо решење за цео низ. Дакле, претпоставимо да је дужина низа $n > 1$ и да умемо да нађемо број m који представља минимум првих $n - 1$ елемената низа. Минимум целог низа дужине n је мањи од бројева m и преосталог, n-тог елемента низа (ако бројање креће од 0, то је елемент a_{n-1}).

На основу овога можемо дефинисати рекурзивну функцију.

```
#include <iostream>
using namespace std;

int min2(int a, int b) {
    return a < b ? a : b;
}
```

```
int minNiza(int a[], int n) {
    if (n == 1)
       return a[0];
    else {
       int m = minNiza(a, n-1);
       return min2(m, a[n-1]);
    }
}

int main() {
    int a[] = {3, 5, 4, 1, 6, 2, 7};
    int n = sizeof(a) / sizeof(int);
    cout << minNiza(a, n) << endl;
}
```

Рецимо и да смо уместо дефинисања функције `min2` за одређивање минимума два броја могли користити и функцију `std::min` из заглавља `<algorithm>`. Ипак, у овим програмима нећемо користити специфичне могућности језика C++, да бисмо нагласили да технике које у овом поглављу уводимо нису ни по чему специфичне за тај језик.

Коректност претходног алгоритма се може формулисати у облику следеће теореме.

Теорема: За сваки непразан низ a (низ за који је дужина $|a| \geq 1$) и за свако $1 \leq n \leq |a|$ позив `minNiza(a, n)` враћа најмањи међу првих n елемента низа a (са a и n су обележене вредности низа `a` и променљиве `n`, а са $|a|$ дужина низа `a`).

Ту теорему можемо доказати индукцијом.

- Базу индукције представља случај $n = 1$, тј. позив `minNiza(a, 1)`. На основу дефиниције функције `minNiza` резултат је `a[0]` тј. први члан низа a_0 и тада тврђење тривијално важи (јер је он уједно најмањи међу првих 1 елемената низа).
- Као индуктивну хипотезу можемо претпоставити да ако важи $1 \leq n - 1 < |a|$, тада позив `minNiza(a, n-1)` враћа најмањи од првих $n-1$ елемената низа a. Из те претпоставке потребно је да докажемо да за n које задовољава $1 < n \leq |a|$ позив `minNiza(a, n)` враћа најмањи од првих n елемената низа a (при том је a непразан низ). На основу дефиниције функције `minNiza`, позив `minNiza(a, n)` ће вратити минимум бројева m (који представља резултат позива `minNiza(a, n-1)`) и a_{n-1}. Пошто су услови индуктивне хипотезе задовољени, на основу индуктивне хипотезе знамо да ће m бити најмањи међу првих $n-1$ елемената низа a. Зато ће минимум броја m и n-тог елемента низа (елемента a_{n-1}) бити најмањи међу првих n елемената низа a.

Примећујемо огромну сличност између рекурзивне конструкције алгоритма и индуктивног доказа његове коректности. Стога слободно можемо да кажемо да су рекурзија и индукција "две стране исте медаље" (индукцију користимо као технику доказивања, а рекурзију као технику дефинисања функција тј. конструкције алгоритама).

Рецимо и да је овај облик коришћења математичке индукције мало нестандардан, јер се не користи директно индукција по природним бројевима, већ се користи индукција по структури рекурзивне функције у којој се, из претпоставке да сваки рекурзивни позив враћа коректан резултат, доказује да функција враћа коректан резултат. Таква теорема индукције се може доказати на основу класичне математичке индукције по броју рекурзивних позива, под претпоставком да се докаже да се рекурзивна функција увек зауставља.

1.3.2 Доказ коректности итеративних алгоритама - инваријанте петље

Један од основних појмова у анализи и разумевању итеративних програма су **инваријанте петљи**. То су логички услови који важе непосредно пре петље, затим након сваког извршавања наредби у телу петље и непосредно након извршавања целе петље. Корисне инваријанте су оне које гарантују коректност алгоритма који та петља имплементира. Инваријанте суштински описују значење свих променљивих унутар петље. Илуструјмо појам инваријанте на једном једноставном примеру.

Размотримо следећу, класичну имплементацију алгоритма за одређивање минимума непразног низа бројева.

```
#include <iostream>
#include <algorithm>
```

```
using namespace std;

int min2(int a, int b) {
    return a < b ? a : b;
}

int minNiza(const vector<int>& a) {
    int m = a[0];
    for (int i = 1; i < a.size(); i++)
        m = min2(m, a[i]);
    return m;
}

int main() {
    vector<int> a{3, 5, 4, 1, 6, 2, 7};
    cout << minNiza(a) << endl;
}
```

У сваком кораку петље, део низа чији минимум знамо постаје дужи за по један елемент. Алгоритам креће од префикса низа дужине 1 и поставља променљиву `m` на вредност првог елемента низа a_0. У сваком кораку петље, претпостављамо да променљива m садржи вредност минимума првих i елемената низа, а онда у телу петље обрађени део низа проширујемо додајући $i+1$-ви елемент низа, на позицији i. Минимум проширеног низа се израчунава као минимум минимума првих i елемената низа (чија је вредност смештена у променљивој `m`) и додатног елемента низа a_i. Након извршавања тела петље, део низа чији минимум је познат је проширен на $i+1$ елемент. На крају петље је i једнако дужини низа, па променљива `m` садржи минимум целог низа.

Пре него што пређемо на формални доказ претходог разматрања, скренимо пажњу на то да именоване величине у математици (тачније алгебри) и у програмирању имају различите особине. Наиме, именоване величине у математици (параметри, непознате) означавају једну вредност док у (императивном) програмирању именоване величине имају динамички карактер и мењају своје вредности током извршавања програма по правилима задатим самим програмом. На пример, бројачка променљива `i` у некој петљи може редом имати вредности 1, 2 и 3. Да бисмо направили разлику између променљивих и њихових текућих вредности, користићемо различит фонт - променљиву програма ћемо обележавати са `i`, а њену вредност са i. Ако желимо да разликујемо стару и нову вредност променљиве `i`, користићемо ознаке i и i'. Ако желимо да нагласимо да је променљива редом узимала неку серију вредности, користићемо ознаке i_0 (почетна вредност променљиве `i`), i_1, i_2, … У ситуацијама у којима се вредност променљиве не мења (на пример, ако је дужина низа током целог трајања програма иста), нећемо обраћати пажњу на разлику између променљиве програма (нпр. `n`) и њене вредности (нпр. n). Елементе низова ћемо такође обележавати индексима и обично ћемо претпостаљати да бројање креће од нуле (нпр. a_0, a_1, …).

Формално, можемо доказати следећу теорему.

Теорема: Ако је низ a дужине $n \geq 1$, непосредно пре почетка петље, у сваком кораку петље (и на њеном почетку, непосредно након провере услова, али и на њеном крају, непосредно након извршавања тела), као и након извршавања целе петље важи да је $1 \leq i \leq n$ и да је m минимум првих i елемената низа (где је i текућа вредност променљиве `i`, а m текућа вредност променљиве `m`).

Ово тврђење можемо доказати индукцијом и то по броју извршавања тела петље (обележимо тај број са k). Напоменимо само да ћемо петљу `for` сматрати само скраћеницом за петљу `while`, тако да ћемо иницијализацију петље сматрати за кôд који се извршава пре петље, док ћемо корак петље сматрати као последњу наредбу тела петље.

```
    int n = a.size();
    int m = a[0];
    int i = 1;
    while (i < n) {
        m = min2(m, a[i]);
        i++;
    }
```

Такође, имплицитно ћемо подразумевати да се током извршавања петље низ ни у једном тренутку не мења (и то се експлицитно може доказати индукцијом). Ни променљива `n` не мења своју вредност.

Да бисмо у доказу били прецизнији, обележимо са $m_0, m_1, \ldots, m_k, \ldots$ вредности променљиве `m`, а са $i_0, i_1, \ldots, i_k, \ldots$ вредност променљиве `i` након $0, 1, \ldots, k, \ldots$ извршавања тела петље. Пошто променљива `n` не мења своју вредност, употребљаваћемо само ознаку n.

- Базу индукције чини случај $k = 0$ тј. случај када се тело петље није још извршило. Пре уласка у петљу променљива `i` се иницијализује на 1 (важи $i_0 = 1$). Пошто претпостављамо да је низ непразан, важи да је $1 \leq i = i_0 = 1 \leq n$. Променљива `m` се иницијализује на вредност `a[0]` (важи $m_0 = a_0$), што је заиста минимум једночланог префикса низа a. Дакле, услови су задовољени пре првог извршавања тела петље.

- Претпоставимо сада као индуктивну хипотезу да тврђење важи након k извршавања тела петље. Дакле, претпостављамо да услови теореме важе за вредности m_k и i_k тј. да је $1 \leq i_k \leq n$ и да је m_k једнако минимуму првих i_k елемената низа (са i_k и m_k обележавамо вредности променљивих након k извршавања тела петље). Ако је услов петље испуњен, то ће уједно бити и вредности променљивих на почетку тела петље, пре њеног $k+1$-вог извршавања. Након k извршавања тела петље важи да је $i_k = k+1$, јер је променљива `i` имала почетну вредност 1 и тачно k пута је увећана за 1 (и ово би се формално могло доказати индукцијом).

  Из индуктивне хипотезе и претпоставке да је услов петље `i < n` испуњен (тј. да је $i_k < n$) докажимо да након $k + 1$ извршавања тела петље услови теореме важе и за вредности m_{k+1} и i_{k+1} (са m_{k+1} и i_{k+1} обележавамо вредности променљивих након $k + 1$ извршавања тела петље). Вредности m_{k+1} и i_{k+1} се могу лако одредити на основу вредности m_k и i_k, анализом једног извршавања тела петље. Важи да је $i_{k+1} = i_k + 1 = k + 2$. Зато, пошто је $1 \leq i_k = k + 1 < n$, важи и да је $1 \leq i_{k+1} = k + 2 \leq n$, па је услов који се односи на распон вредности променљиве `i` очуван. Докажимо и да је m_{k+1} минимум првих i_{k+1} елемента низа. Важи да је m_{k+1} минимум вредности m_k и елемента a_{i_k}, тј. a_{k+1}. На основу индуктивне хипотезе знамо да је m_k минимум првих $i_k = k + 1$ елемената низа. Зато ће m_{k+1} бити минимум првих $k + 2$ елемената низа (закључно са елементом a_{k+1}), што је тачно i_{k+1} елемената низа, па и други услов остаје очуван.

Означимо са i и m вредности променљивих `i` и `m` након извршавања петље. На основу доказаног тврђења знамо да услови наведени у њему важе и након завршетка петље. Када се петља заврши, важи да је $i = n$ (јер на основу првог услова знамо да је $1 \leq i \leq n$, а услов петље `i < n` није испуњен). На основу другог услова знамо да је m минимум n чланова низа (што је заправо цео низ, јер је n његова дужина), тј. да променљива `m` садржи тражену вредност, чиме је доказана парцијална коректност. Заустављање се доказује једноставно тако што се докаже да се у сваком кораку петље ненегативна вредност $n - i$ смањује за по 1, док не постане 0.

Ако размотримо структуру претходног разматрања, можемо установити да смо идентификовали логичке услове који су испуњени непосредно пре и непосредно након сваког извршавања тела петље. Такви услови се називају **инваријанте петље**. Да бисмо доказали да је неки услов инваријанта петље, довољно је да докажемо:

(1) да тај услов важи пре првог уласка у петљу и

(2) да из претпоставке да тај услов важи пре неког извршавања тела петље и да је услов петље испуњен докажемо да тај услов важи и након извршавања тела петље.

Те две чињенице нам, на основу индуктивног аргумента, гарантују да ће услов бити испуњен пре и после сваке итерације петље, као и након извршавања целе петље (ако се она икада заустави), тј. да ће тај услов бити инваријанта петље (тај доказ се може спровести класичном математичком индукцијом на основу броја извршавања тела петље). Приметимо да први корак одговара доказивању базе индукције, а други доказивању индуктивног корака.

Свака петља има пуно инваријанти, међутим, од интереса су нам само оне инваријанте које у комбинацији са условом прекида петље (под претпоставком да петља није прекинута наредбом `break`) имплицирају услов који нам је потребан након петље. Ако је петља једина у неком алгоритму, обично је то онда услов коректности самог алгоритма. Дакле, након доказа леме која чини основу доказа да је неки услов инваријанта петље, потребно је да докажемо и

(3) да из тога да инваријанта важи након завршетка петље и да услов петље није испуњен следи коректност алгоритма.

Дакле, општа структура анализе програма коришћењем инваријанти се може описати на следећи начин.

```
<incijalizacija>
// ovde vazi <invarijanta>
```

```
while (<uslov>)
   // ovde vaze i <uslov> i <invarijanta>
   <telo>
   // ovde vazi <invarijanta>
// ovde ne vazi <uslov>, a vazi <invarijanta>
```

Изолујмо кључне делове претходног доказа и прикажимо их у формату који ћемо и убудуће користити приликом доказивања инваријанти петљи (индукција ће у тим доказима бити само имплицитна).

Лема: Ако је низ a дужине $n \geq 1$, услов да је $1 \leq i \leq n$ и да је m минимум првих i елемената низа је инваријанта петље (где са i обележавамо текућу вредност промењиве `i`, а са m текућу вредност промењиве `m`).

- Пре уласка у петљу промењива `i` се иницијализује на 1 (важи $i = 1$). Пошто претпостављамо да је низ непразан, важи да је $1 \leq i \leq n$. Промењива `m` се иницијализује на вредност `a[0]` (важи $m = a_0$), што је заиста минимум једночланог префикса низа a.
- Претпоставимо да тврђење важи након уласка у петљу тј. да је вредност промењиве `m` (означимо је са m) једнака минимуму првих i чланова низа (где је i вредност промењиве `i` на уласку у петљу), да је $1 \leq i \leq n$, као и да је услов петље испуњен тј. да је $i < n$.

  Пошто је након извршавања тела петље вредност промењиве `i` увећана за један, важи да је $i' = i + 1$ (где са i' обележавамо вредност промењиве i након извршавања тела и корака петље). Пошто је важи да је $i < n$ и $1 \leq i \leq n$, након извршавања тела петље, важиће да је $1 \leq i' \leq n$.

  Нова вредност промењиве `m` (означимо је са m') биће једнака мањој од вредности m и a_i. На основу претпоставке важи да је m једнако минимуму првих i елемената низа, тј. минимуму бројева a_0, …, a_{i-1}, па је m' једнако минимуму бројева a_0, …, a_i, што је управо минимум првих $i+1$ елемената низа, па је заиста m' минимум првих i' елемената низа.

Теорема: Након извршавања петље, промењива `m` садржи минимум целог низа.

На основу инваријанте важи да је $1 \leq i \leq n$, а пошто по завршетку петље њен услов није испуњен, важи да је $i = n$. На основу инваријанте важи и да промењива `m` садржи минимум првих i елемената низа, а пошто је $i = n$, где је n број чланова низа, то је заправо минимум целог низа.

У наставку овог поглавља видећемо још неколико примера примене технике инваријанте петље. Мора се признати да када се техника користи потпуно формално, да би се доказала коректност већ написаног програмског кода, то не делује нарочито инспиришуће (поготово, ако су програми једноставни и ако је једноставно интуитивно разумети разлоге њихове коректности). Ретко када се у практичном програмирању коректност заиста доказује потпуно формално (осим у случају софтвера који може да угрози велики број живота, попут, на пример, софтвера који управља метро-системом у Паризу, који јесте у потпуности формално верификован). Међутим, аргументе и инваријанте на којима коректност почива програмер често "проврти по глави". Видећемо и да се техника инваријанти може употребити и пре него што је програм написан у циљу извођења програмског кода из спецификације. Јасне инваријанте често једнозначно указују на то како програмски кôд треба да изгледа и на тај начин помажу у процесу програмирања.

Задаци који су одабрани нису ни по чему посебни – они ће бити поновљени у поглављима у којима се уводе опште програмерске технике које се у њима примењују.

Задатак: Тробојка

Написати програм који учитава низ целих бројева а затим га трансформише тако да елементи буду подељени у три дела у зависности од задатих вредности A и B. У првом делу су елементи мањи од задате вредности A (вредности из интервала $[-\infty, A)$), у другом елементи већи или једнаки задатој вредности A и мањи или једнаки задатој вредности B (вредности из интервала $[A, B]$), а у трећем елементи већи од задате вредности B (вредности из интервала $(B, +\infty)$). Није битно у ком се редоследу налазе елементи унутар делова. Учитати елементе у низ, а затим реорганизовати редослед елемената у том низу (не користити помоћне низове).

Улаз: У једној линији стандардног улаза налази се број елемената низа, N, а затим се, у наредној линији налазе елементи низа раздвојени размацима. У последње две линије се налазе цели бројеви A и B одвојени празнином, и при томе је $A < B$.

Излаз: Исписати елементе резултујућег низа на стандардни излаз (могуће је исписати елементе сваке од три групе у посебном реду, раздвојене размацима, а могуће је исписати и цео низ у једном реду или у више редова).

Пример

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

Решење

Један пролаз кроз низ

Задатак можемо решити помоћу само једног пролаза кроз низ и то "у месту" тј. без коришћења помоћног низа. Алгоритам у наставку познат је под називом "Холандска застава тробојка" (енгл. Dutch national flag) и приписује се чувеном информатичару Дајкстри (енгл. Edsger W. Dijkstra).

Одржаваћемо три променљиве l, d и i и током петље наметнућемо да важи $0 \leq l \leq i \leq d \leq n$ и да важе следећи услови.

- У интервалу позиција $[0, l)$ налазиће се елементи мањи од A тј. бројеви из интервала $(-\infty, A)$,
- у интервалу позиција $[l, i)$ налазиће се елементи из интервала $[A, B]$,
- у интервалу позиција $[i, d)$ налазиће се елементи који још нису испитани,
- у интервалу позиција $[d, n)$ налазиће се елементи који су већи од B тј. елементи из интервала $(B, +\infty)$.

Дакле, одржавамо распоред `<<<===???>>>`, где су са `<` обележени елементи прве групе, са `=` елементи друге, а са `>` елементи треће групе.

Да би инваријанта важила пре уласка у петљу, јасно је да мора да важи да је $i = 0$ и $d = n$ (јер су сви елементи из интервала $[i, d) = [0, n)$ неиспитани). Такође, да бисмо били сигурни да су и интервалу $[0, l)$ сви елементи мањи од A, тај интервал мора бити празан и мора да важи да је $l = 0$. Након овакве иницијализације и интервал $[l, i) = [0, 0)$ и интервал $[d, n) = [n, n)$ је празан, па задовољава наметнути услов.

Петља ће се извршавати док год има неиспитаних елемената, а то је док је $i < d$. Размотримо како треба да изгледа тело петље, да би услови били одржани.

- Ако је елемент на позицији i мањи од броја A тада ћемо га заменити са елементом на позицији l (првим елементом из интервала $[A, B]$), након чега можемо увећати и i и l.
- У супротном, ако је елемент на позицији i мањи или једнак од B он припада интервалу $[A, B]$ и већ је на свом допуштеном месту, па само можемо увећати вредност i.
- У супротном елемент је већи од B и тада можемо смањити вредност d и разменити елемент на позицији i са елементом на (умањеној) позицији d, не мењајући вредност i (да би се елемент који је управо доведен на позицију i могао испитати у наредној итерацији).

На крају петље важи да је $i = d$. Уз остале наметнуте услове тврђење одатле следи (елементи из интервала позиција $[0, l)$ су мањи од A, елементи из интервала позиција $[l, i) = [l, d)$ су између A и B, интервал непрегледаних елемената $[i, d)$ је празан, док су елементи из интервала $[d, n)$ већи од B. Дакле, низ је разбијен на надовезане сегменте $[0, l)$, $[l, d)$ и $[d, n)$ и у сваком сегменту се налазе одговарајући елементи.

Пример. Размотримо рад алгоритма на једном примеру. Нека је $A = 4$, $B = 7$ и нека низ има садржај `5 1 8 6 3 9 4 2`. У наставку ћемо приказати стање низа током извођења алгоритма.

```
l             d
5 1 8 6 3 9 4 2
i

l             d
5 1 8 6 3 9 4 2
  i

  l           d
```

```
1 5 8 6 3 9 4 2
    i

  l           d
1 5 2 6 3 9 4 8
    i

    l         d
1 2 5 6 3 9 4 8
      i

    l         d
1 2 5 6 3 9 4 8
        i

      l       d
1 2 3 6 5 9 4 8
          i

      l     d
1 2 3 6 5 4 9 8
          i

      l     d
1 2 3 6 5 4 9 8
            i
```

Анализа сложености. У сваком кораку петље се или увећава `i` или смањује `d`, док се не сусретну, што се дешава у n корака. Сложеност овог приступа је, дакле, $O(n)$.

```
// funkcija organizuje elemente vektora tako da se prvo nalaze elementi
// za koje vazi da su iz intervala (-Inf, A), nakon toga dolaze
// elementi iz intervala [A, B], i nakon toga elementi iz intervala
// (B, Inf)
void podelaNiza(vector<int>& niz, int A, int B) {
  // - u intervalu pozicija [0, l) su elementi iz intervala (-Inf, A)
  // - u intervalu pozicija [l, i) su elementi iz intervala [A, B]
  // - u intervalu pozicija [i, d) su jos neispitani elementi
  // - u intervalu pozicija [d, n) su elementi iz intervala (B, Inf)
  int l = 0, i = 0, d = niz.size();
  // dok god postoje neispitani elementi
  while (i < d) {
    if (niz[i] < A)
      // menjamo tekuci element sa prvim elementom iz intervala [A, B]
      swap (niz[i++], niz[l++]);
    else if (niz[i] <= B)
      // tekuci element ostaje na svom mestu
      i++;
    else
      // menjamo tekuci element sa poslednjim neispitanim
      swap(niz[i], niz[--d]);
  }
}
```

Задатак: Први који није дељив

Размотримо низ бројева $210, 2310, 390, 30, 510, 66, 6, 138, 46, 106, 59, 17, 23$. Он је интересантан из неколико разлога. На пример, првих пет бројева је дељиво са 10, а после ниједан број није дељив са 10. Првих десет бројева је парно, а после су сви бројеви непарни. Првих осам бројева је дељиво са 6, а после ниједан број

није дељив са 6. Прва два броја су дељива са 210, а после ниједан број није дељив са 210, итд. Покушај да пронађеш још оваквих правилности. Напиши програм који за сваки унети делилац одређује колико бројева је дељиво њиме. Сматрати да за сваки унети делилац важи наведена правилност.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^5$), а затим у наредном реду n природних бројева (мањих од 10^{18}) раздвојених по једним размаком. Након тога се до краја улаза уносе делиоци (сваки у посебном реду). За сваки делилац се сигурно зна (и то није потребно проверавати) да се у низу налазе прво бројеви који јесу, а затим бројеви који нису дељиви тим делиоцем.

Излаз: За сваки унети делилац у посебном реду исписати број елемената низа који су њиме дељиви.

Пример

Улаз	*Излаз*
13	5
210 2310 390 30 510 66 6 138 46 106 59 17 23	10
10	8
2	10
6	0
2	5
4	
15	

Решење

Бинарна претрага

Захваљујући интересантној особини низа, задатак ефикасно може бити решен применом алгоритма бинарне претраге. У питању је варијанта алгоритма бинарне претраге у ком се уместо позиције конкретне вредности у сортираном низу захтева проналажење прве позиције на којој се налази елемент који задовољава неки услов. Наиме, под претпоставком да се у низу прво налазе елементи који не задовољавају тај услов, а затим елементи који задовољавају тај услов, *преломну тачку* (тренутак када се из једне прелази у другу групу елемената) можемо наћи бинарном претрагом. Дакле, ако је низ облика `-------+++++++`, бинарном претрагом можемо пронаћи позицију последњег минуса, првог плуса, број минуса или број плусева, где смо са `-` означили оне елементе који не задовољавају, а са `+` оне елементе који задовољавају дати услов.

Ручно имплементирана бинарна претрага

Током рада алгоритма, одржавамо две променљиве l и d такве да важи инваријанта да је $0 \leq l \leq d + 1 \leq n$ и да су

- лево од l тј. у интервалу позиција $[0, l)$ елементи који не задовољавају услов,
- десно од d тј. у интервалу позиција (d, n) елементи који задовољавају услов.

У интервалу позиција $[l, d]$ налазе се елементи чији статус још није познат. На почетку су сви елементи непознати, па је јасно да интервал $[l, d]$ треба иницијализовати на $[0, n - 1]$, тј. променљиву l треба иницијализовати на нулу, а d на вредност $n - 1$. Интервали $[0, l)$ и (d, n) су празни, па је инваријанта очувана (услов $l \leq d + 1$ се своди на $0 \leq n$, што је тривијално испуњено).

Ако интервал позиција $[l, d]$ није празан тј. ако је $l \leq d$, проналазимо му средину $s = l + \left\lfloor \frac{d-l}{2} \right\rfloor$.

- Ако елемент на позицији s задовољава услов, тада на основу монотоности услов задовољавају и сви елементи десно од s. Зато померамо d за једно место лево од средине тј. вредност променљиве d постављамо на $s - 1$.
- Ако елемент на позицији s не задовољава услов, тада на основу монотоности услов не задовољавају ни сви елементи лево од s. Зато померамо l за једно место десно од средине тј. вредност променљиве l постављамо на $s + 1$.

Претрага траје све док се интервал $[l, d]$ не испразни, тј. док је $l \leq d$. Тада је $l = d + 1$ и елементи који не задовољавају услов се налазе на позицијама $[0, l) = [0, d]$, док се елементи који не задовољавају услов налазе на позицијама $(d, n) = [d + 1, n) = [l, n)$. Дакле, први елемент који задовољава услов је на позицији l, а последњи који не задовољва услов на позицији d.

Пример. Прикажимо рад алгоритма на једном примеру.

```
l               d
1 7 3 5 9 11 2 8 6
        s

          l     d
1 7 3 5 9 11 2 8 6
            s

          ld
1 7 3 5 9 11 2 8 6
          s

          d  l
1 7 3 5 9 11 2 8 6
```

Имплементација се може направити на следећи начин.

```
int prviKojiNijeDeljiv(const vector<long long>& a, long long k) {
  int n = a.size();
  int l = 0, d = n-1;
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (a[s] % k != 0)
      d = s - 1;
    else
      l = s + 1;
  }
  return l;
}
```

Доказ коректности. Докажимо формално коректност овог алгоритма. Уз поменуте услове инваријанта је и да важи $0 \leq l \leq d+1 \leq n$.

Након иницијализације $l = 0, d = n-1$, услови су испуњени (интервали $[0, l)$ и (d, n) су празни, док је услов $0 \leq l \leq d+1 \leq n$ еквивалентан услову $0 \leq 0 \leq n \leq n$ и тривијално је испуњен.

Ако интервал позиција $[l, d]$ није празан тј. ако је $l \leq d$, проналазимо му средину $s = l + \lfloor \frac{d-l}{2} \rfloor$. Пошто је $l \leq d$, важи и да је $l \leq s \leq d$.

- Ако елемент на позицији s задовољава услов, тада на основу монотоности услов задовољавају и сви елементи десно од s. Зато вредност промењиве d постављамо на $s-1$ (нове вредности промењивих су $l' = l$ и $d' = s-1$). Тиме инваријанта остаје на снази (посебно, сви елементи у интервалу позиција $(s-1, n) = [s, n)$ задовољавају услов). Важи и услов $0 \leq l' \leq d'+1 \leq n$, јер је он еквивалентан услову $0 \leq l \leq s \leq n$.
- Ако елемент на позицији s не задовољава услов, тада на основу монотоности услов не задовољавају ни сви елементи лево од s. Зато вредност промењиве l постављамо на $s+1$ (нове вредности промењивих су $l' = s+1$ и $d' = d$). Тиме инваријанта остаје одржана (посебно, ниједан елемент у интервалу позиција $(0, l) = [0, s]$ не задовољава услов). Важи и услов $0 \leq l' \leq d'+1 \leq n$ који је еквивалентан услову $0 \leq s+1 \leq d+1 \leq n$.

Када се интервал испразни, тада је $l > d$, па пошто важи $0 \leq l \leq d+1 \leq n$, важи и $l = d+1$. На основу инваријанте знамо да су елементи који задовољавају услов на позицијама $(d, n) = [l, n]$. Зато је први елемент који задовољава услов је на позицији l (што је уједно и број елемената који не задовољавају услов). Елементи који не задовољавају услов су на позицијама $[0, l) = [0, d+1) = [0, d]$, па је последњи елемент који не задовољава на позицији d.

Заустављање се лако доказује тако што се доказује да се у сваком кораку петље интервал $[l, d]$ тј. његова дужина $d-l+1$ смањује, што је прилично очигледно и када је $l' = l$ и $d' = s-1 < d$ и када је $l < l' = s+1$ и $d' = d$.

Анализа сложености. Пошто се у сваком кораку претраге ширина интервала $[l, d]$ преполови, пошто се иницијално креће од интервала $[0, n-1]$ који има n елемената и пошто се алгоритам завршава када се интервал

испразни, сложеност алгоритма је $O(\log n)$. Наиме, дужина интервала после k корака је $\left\lfloor \frac{n}{2^k} \right\rfloor$ и важи да је $\left\lfloor \frac{n}{2^k} \right\rfloor < 1$ када је $k > \log_2 n$.

Исправљање грешака на основу формалне анализе кода

Када је кôд коректан, доказ је обично неинформативан. Помаже нам да "мирно спавамо", али ништа више од тога. Много интересантнија ситуација се дешава у случају када нам формално резоновање о коду помаже да детектујемо и исправимо грешке у програму (тзв. багове). Погледајмо наредни покушај имплементације алгоритма.

```
int l = 0, d = n;
while (l < d) {
  int s = l + (d - l) / 2;
  if (a[s] % k != 0)
    d = s-1;
  else
    l = s+1;
}
cout << d+1 << '\n';
```

На основу инцијализације делује да покушавамо да претражимо полузатворени интервал $[l, d)$. Пошто је у питању бинарна претрага, изгледа да се намеће инваријанта да је $0 \leq l \leq d \leq n$ и да су:

- сви елементи из $[0, l)$ дељиви са k,
- ниједан елемент из интервала $[d, n)$ није дељив са k.

На почетку су оба та интервала празна, па инваријанта за сада добро функционише. Ако погледамо услов петље, делује да петља ради док се интервал непознатих елемената $[l, d)$ не испразни (заиста, када је $l \geq d$, тај интервал је празан). За сада све ради како треба. Покушамо сада да проверимо да ли извршавање тела петље одржава инваријанту.

- Ако a_s није дељив са k, тада се променљива `d` поставља на вредност $d' = s - 1$. На основу инваријанте треба да важи да ниједан елемент у интервалу $[d', n)$ није дељив са k. Међутим, ми то не знамо, јер само знамо да је a_s није дељив са k, али не знамо да a_{s-1} није дељив са k. Дакле, овде се сигурно крије грешка у коду. Ако доделу `d = s-1` заменимо са `d = s`, тада ће инваријанта бити одржана (јер знамо да a_s није дељив са k, па са k неће бити дељив ниједан елемент иза њега).
- Ако a_s јесте дељив са k, тада се променљива `l` поставља на вредност $l' = s + 1$. На основу инваријанте треба да важи да су сви елементи у интервалу $[0, l')$ дељиви са k, међутим, то ће овде бити испуњено, јер је a_s дељив са k, па су са k дељиви и сви елементи испред њега. Дакле, у овом случају је кôд коректан и инваријанта остаје одржана.

На крају, када се петља заврши можемо закључити да важи да је $l = d$ (јер све време важи да је $l \leq d$, а након петље не важи да је $l < d$). У коду се за позицију првог елемента који није дељив са k проглашава позиција $d + 1$. Иако је у оригиналној варијанти кода `l` могло без проблема да се замени са `d+1`, у овој варијанти то није могуће. Наиме, ми на основу инваријанте овог кода знамо да се на позицији $l = d$ налази елемент који није дељив са k, а да се на позицији $l - 1$ налази елемент који јесте дељив са k (осим када је $l = 0$ и тада нема елемената дељивих са k). Зато крајњи резултат није коректан и потребно га је заменити са `d`, јер се први елемент који није дељив са k налази на позицији d (осим када су сви елементи дељиви са k, када је $d = n$, но и тада је d исправна повратна вредност). Дакле, формалном анализом смо открили и исправили две грешке.

Програмери често програм исправљају тако што насумице покушавају да помере индексе за 1 лево или десно, да замене мање са мање или једнако и слично. Већ на овако кратким програмима се види да је простор могућих комбинација велики, а да је могућност за грешку приликом таквог експерименталног приступа веома велика. Стога је увек боље застати, формално анализирати шта је потребно да кôд ради и исправити га на основу резултата формалне анализе.

На крају, скренимо пажњу на још један детаљ исправљеног програма. Парцијална коректност је јасна на основу анализе коју смо спровели, међутим, заустављање може бити доведено у питање, с обзиром на наредбу `d = s`. Заустављање доказујемо тако што показујемо да се у сваком кораку смањује број непознатих елемената, тј. да дужина интервала $[l, d)$ која је једнака $d - l$ у сваком кораку петље опада. Пошто је $l \leq d$ инваријанта, смањивање не може трајати довека, па се у неком тренутку програм зауставља. Поставља се питање да ли се $d - l$ смањује и у измењеном коду у коме се јавља наредба `d=s`. Одговор је потврдан, а образложење је

суптилно. Прво, на основу услова петље важи да је $l < d$. Даље, вредност s се израчунава наредбом `s = l + (d - l) / 2` што нам да је $s = \left\lfloor \frac{l+d}{2} \right\rfloor$. Због заокруживања наниже, важи да је $s < d$ и зато се након одређивања $d' = s$, $l' = l$ вредност $d' - l'$ смањује у односу на $d - l$. Важи и да је $l \leq s$, али пошто је у другој грани $l' = s + 1$ и $d' = d$, вредност $d' - l'$ се опет смањује у односу на $d - l$. Да је заокруживање којим случајем вршено навише (нпр. `s = l + (d - l + 1) / 2`), програм би могао упасти у бесконачну петљу.

```
int prviKojiNijeDeljiv(const vector<long long>& a, long long k) {
  int n = a.size();
  int l = 0, d = n;
  while (l < d) {
    int s = l + (d - l) / 2;
    if (a[s] % k != 0)
      d = s;
    else
      l = s + 1;
  }
  return d;
}
```

Види другачија решења овог задатка.

Задатак: Најмањи број који није збир елемената скупа

Дат је скуп природних бројева (задат у облику сортираног низа). Одредити најмањи природан број који није збир неких елемената тог скупа (сваки елемент скупа може само једном учествовати у збиру).

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^3$), а затим у наредном реду сортиран низ од n различитих природних бројева мањих од 10^4.

Излаз: На стандардни излаз исписати тражени најмањи природан број који није збир неких елемената тог скупа.

Пример

Улаз	*Излаз*
`8` `1 2 4 7 15 32 35 48`	`30`

Решење

Чињеница да су елементи сортирани олакшава решење задатка. Обрађиваћемо елемент по елемент и одржаваћемо границу до које смо сигурни да се сваки број може представити као збир неког подскупа. Можда мало изненађујуће, та граница је у сваком кораку једнака збиру свих тренутно учитаних елемената. Ако је нови учитани елемент строго већи од збира свих претходних елемената увећаног за један, онда се тај увећани збир не може добити као подскуп. У супротном можемо бити сигурни да се сви бројеви од 0 па до збира свих елемената (у који је укључен и нови елемент) могу добити као збир неког подскупа. Наиме, пошто је у претходном кораку било могуће добити све бројеве од 1 до збира свих елемената без тог новог, када у све те подскупове укључимо нови елемент добићемо све бројеве од тог новог елемента, па до збира свих елемената са тим новим елементом.

Пример. На пример, нека је дат низ $1, 2, 3, 5, 14, 20, 27$.

- 0 можемо добити као збир празног скупа $\{\}$.
- 1 можемо добити као збир скупа $\{1\}$.
- Када у претходне скупове укључимо и 2, можемо добити све бројеве закључно са 3 (2 као $\{2\}$ и 3 као $\{1, 2\}$).
- Када у претходне скупове укључимо и 3, можемо добити све бројеве закључно са 6 (4 као $\{1, 3\}$, 5 као $\{2, 3\}$ и 6 као $\{1, 2, 3\}$).
- Када у претходне скупове укључимо 5 можемо добити све бројеве закључно са 11 (7 као $\{2, 5\}$, 8 као $\{1, 2, 5\}$ и 9 као $\{1, 3, 5\}$, 10 као $\{2, 3, 5\}$ и 11 као $\{1, 2, 3, 5\}$).
- Пошто је наредни број 14, јасно је да се број 12 не може никако добити.

```
int n;
cin >> n;
```

```
// sabiranjem elemenata trenutnog sukpa mogu se dobiti svi elementi
// iz intervala [0, mozeDo]
int mozeDo = 0;
for (int i = 0; i < n; i++) {
  int x; cin >> x;
  if (x > mozeDo + 1)
    break;
  mozeDo += x;
}
cout << mozeDo + 1 << endl;
```

Анализа сложености. Програм се решава једним проласком кроз низ бројева и сложеност је прилично очигледно $O(n)$.

Доказ коректности. Докажимо и формално коректност, овог алгоритма тј. програма датог у прилогу.

Лема: Нека је m вредност променљиве `mozeDo`. Инваријанта петље је да је $0 \leq i \leq n$, да је m збир првих i елемената низа и да се сваки број из интервала $[0, m]$ може добити као збир неког подскупа првих i елемената низа.

Пре уласка у петљу је $i = 0$ и $m = 0$. Збир првих $i = 0$ елемената низа је по дефиницији нула (тј. m). Број 0 је једини елемент интервала $[0, m] = [0, 0]$ и он се може добити као збир празног подскупа (тј. 0 елемената полазног низа).

Претпоставимо да тврђење важи пре уласка у петљу.

- Ако је $a_i > m + 1$, тврдимо да је $m + 1$ тражени најмањи број. На основу инваријанте знамо да су сви бројеви из интервала $[0, m]$ покривени, тако да мањи број од $m + 1$ не може бити решење. Докажимо да број $m + 1$ не може бити збир подскупа. Пошто је низ сортиран, сви елементи од a_i до a_{n-1} су строго већи од $m+1$. Дакле, ни један од тих елемената не сме бити укључен у подскуп јер би њиховим укључивањем збир већ премашио $m + 1$. Подскуп се мора састојати само од елемената a_0 до a_{i-1}, међутим, пошто је m њихов збир, збир сваког њиховог подскупа је мањи или једнак m. Дакле, $m + 1$ се не може постићи и он је тражено решење.
- Ако је $a_i \leq m + 1$, тада је $m' = m + a_i$, $i' = i + 1$ и тврдимо да је m' збир свих елемената a_0, …, a_i и да се сваки број из интервала $[0, m']$ може представити као збир неког подскупа првих $i' = i + 1$ елемената низа. Прва тврдња је прилично очигледна, јер је по претпоставци m збир свих елемената a_0, …, a_{i-1}, а $m' = m + a_i$. На основу претпоставке знамо да сви бројеви из $[0, m]$ могу бити збирови подскупова првих i елемената низа. Слично и сви бројеви из интервала $[a_i, a_i + m]$ се могу добити као збир неког подскупа првих $i' = i + 1$ елемента низа. Наиме, тај подскуп ће бити унија елемента a_i и оног подскупа првих i елемената низа чији је збир једнак разлици између тог броја и броја a_i – он је из $[0, m]$, па на основу претпоставке такав подскуп постоји. Пошто је $a_i \leq m + 1$ унија интервала $[0, m]$ и $[a_i, a_i + m]$ је $[0, a_i + m] = [0, m']$. Зато је сваки елемент из $[0, m']$ једнак збиру неког подскупа првих i' елемената низа, па инваријанта остаје очувана.

Теорема: Случај када се петља заврши прекидом, јер је $a_i > m+1$ је већ размотрен. Када се петља заврши, важи да је $i = n$. На основу инваријанте m је збир свих елемената низа, и сваки број из $[0, m]$ јесте збир неког подскупа првих $i = n$ елемената низа, тј. целог низа. Зато је $m + 1$ најмањи елемент који није могуће добити (јер се укључивањем свих елемената добија највише m) и исписано решење је исправно.

Задатак: Бинарни запис

Напиши програм који на основу неозначеног целог броја n формира и исписује његов 32-битни бинарни запис.

Улаз: Са стандардног улаза се уноси број n ($0 \leq n \leq 2^{32} - 1$).

Излаз: На стандардни излаз исписати 32-битни бинарни запис броја n.

Пример 1

Улаз	*Излаз*
123456	00000000000000011110001001000000

Пример 2

Улаз	*Излаз*
`16777215`	`00000000111111111111111111111111`

Решење

Нека је низ од 32 логичке вредности попуњен вредношћу `false`. Бинарни запис одређујемо тако што одређујемо једну по једну бинарну цифру броја, здесна налево. У сваком кораку петље одређујемо остатак при дељењу броја n са 2, и на наредно место у низу (у кораку i на место i) уписујемо `true` ако је тај остатак једнак 1. На крају петље, исписујемо садржај низа уназад.

Пример. Прикажимо извршавање алгоритма на примеру превођења броја 38 у бинарни запис. Приказаћемо само кораке који се изводе док n не постане 0 (од тог тренутка надаље низ се само попуњава нулама).

```
n      niz b
38
19       0
9       10
4      110
2     0110
1    00110
0   100110
```

Имплементација се може направити на следећи начин.

```
// broj koji se prevodi
unsigned long n;
cin >> n;

// niz binarnih cifara, redom, od cifre najmanje do cifre najvece
// tezine
bool binarneCifre[32] = {false};
// prevodjenje
for (int i = 0; n > 0; i++, n /= 2)
  binarneCifre[i] = n % 2;

// ispisujemo rezultat (od cifre najmanje tezine
for (int i = 31; i >= 0; i--)
  cout << (binarneCifre[i] ? '1' : '0');
cout << endl;
```

Доказ коректности. Докажимо формално коректност овог алгоритма.

Да бисмо лакше одредили инваријанту проширимо пример извршавања програма вредношћу бинарног броја тренутно записаног у низу b и одговарајућим степеном двојке.

```
n      niz b    b     2^i
38              0     1
19       0      0     2
9       10      2     4
4      110      6     8
2     0110      6    16
1    00110      6    32
0   100110     38    64
```

Сада се лако може приметити да у сваком реду важи да је $2^i \cdot n + b = 38$ (заиста, важи да је $1 \cdot 38 + 0 = 2 \cdot 19 + 0 = 4 \cdot 9 + 2 = 8 \cdot 4 + 6 = 16 \cdot 2 + 6 = 32 \cdot 1 + 6 = 64 \cdot 0 + 38 = 38$.

Лема: Услов $2^i \cdot n + b = n_0$ је инваријанта петље, где је b број тренутно кодиран низом бинарних цифара (ако логичка вредност на позицији k у низу одговара цифри b_k, нека је $b = \sum_{k=0}^{31} b_k 2^k$), где је i текућа вредност променљиве `i`, док је n_0 почетна, а n текућа вредност неозначеног броја `n`.

- Заиста на почетку је $n = n_0$, $i = 0$ и $b = 0$ па тврђење важи.

- Претпоставимо да тврђење важи при уласку у петљу. Промењиве се током извршавања тела и корака петље мењају на следећи начин. $n' = n \operatorname{div} 2$, $b' = b + 2^i \cdot (n \bmod 2)$ и $i' = i + 1$. Тада је $2^{i'} \cdot n' + b' = 2^{i+1} \cdot (n \operatorname{div} 2) + b + 2^i \cdot n \bmod 2 = 2^i \cdot (2 \cdot (n \operatorname{div} 2) + n \bmod 2) + b$. На основу дефиниције целобројног дељења важи да је $2 \cdot (n \operatorname{div} 2) + n \bmod 2 = n$, па је вредност претходног израза једнака $2^i \cdot n + b$, а на основу претпоставке о томе да инваријанта важи на уласку у тело петље знамо да је то једнако n_0.

Теорема: По завршетку алгоритма низ садржи бинарни запис неозначеног броја `n`.

Како је по изласку из петље $n = 0$, на основу инваријанте важи да је $b = n_0$ тј. да низ садржи бинарни запис полазног броја.

Заустављање је прилично очигледно јер је `n` ненегативан број који се у сваком кораку стриктно смањује (све док не достигне нулу).

Анализа сложености. У овом облику алгоритма се увек спровди 32 корака, па је сложеност константна. У општем случају можемо сматрати да је линеарна у односу на број цифара у бинарном запису (она логаритамски зависи од величине броја n).

Задатак: Број формиран од датих цифра с лева на десно

Написати програм којим се формира природан број од учитаних цифара, ако се цифре броја учитавају слева на десно (редом од цифре највеће тежине до цифре јединица).

Улаз: Свака линија стандарног улаза, њих највише 9, садржи по једну цифру.

Излаз: На стандарном излазу приказати формирани број.

Пример

Улаз	*Излаз*

`4`
`1`
`0`
`9`
	`4109`

Решење

Читамо цифру по цифру са стандардног улаза, све док не дођемо до краја и формирамо број додавајући му прочитану цифру на десну страну (као цифру најмање тежине). Алгоритам се назива *Хорнерова шема* и на основу тог алгоритма број се гради тако што се у сваком кораку претходна вредност броја помножи са 10 и добијени производ се увећа за вредност наредне цифре (тако се цифра допише на десни крај претходног броја).

Пример. Анализирајмо пример у којем учитавамо редом цифре 3, 2, 7 и 5 и треба да добијемо број 3275. На основу дефиниције позиционог записа тај број је једнак вредности израза $3 \cdot 10^3 + 2 \cdot 10^2 + 7 \cdot 10 + 5$. Међутим, претходни израз можемо израчунати на следећи начин $((3 \cdot 10 + 2) \cdot 10 + 7) \cdot 10 + 5$, што доводи до Хорнеровог поступка. Ако се он имплементира итеративно, промењива `n` којом се представља вредност броја узима редом вредности 0, 3, 32, 327 и 3275.

Пошто број цифара није унапред познат, цифре ћемо учитавати у петљи којом учитавамо бројеве до краја стандардног улаза.

Број који формирамо памтићемо у промењивој `n` коју иницијизујемо на нулу. Када прочитамо цифру, додајемо је као цифру јединица на до сада формирани број. Додавање цифре јединица на број `n`, постижемо тако што помножимо броја `n` са 10 и додамо му учитану цифру. Ако унесемо k цифара, на описан начин цифру коју смо прву прочитали множићемо са 10 тачно $k - 1$ пут, другу прочитану цифру множимо са 10 тачно $k - 2$ пута, и тако редом, последњу прочитану цифру не множимо са 10 (то је цифра јединица).

```
int cifra;
int n = 0;
while (cin >> cifra)
  n = n * 10 + cifra;
cout << n << endl;
```

Доказ коректности. Докажимо и формално коректност Хорнерове схеме. Претпоставимо да ће се редом учитавати цифре a_{k-1}, a_{k-2}, …, a_1, a_0. Инваријанта петље је то да је n вредност броја који се добија

записом до сада прочитаних и обрађених цифара. Након i извршавања тела петље то су цифре од a_{k-1} до a_{k-i} и тврдимо да важи

$$n = (a_{k-1} \dots a_{k-i})_{10} = a_{k-1}10^{i-1} + a_{k-2}10^{i-2} + \dots + a_{k-i}10^0.$$

- *База*. Након 0 извршавања тела петље важи да је $n = 0$ и $i = 0$. Број $(a_{k-1} \dots a_{k-i})_{10} = (a_{k-1} \dots a_k)_{10}$ нема ниједну цифру и вредност му је нула.
- *Корак*. Претпоставимо да инваријанта важи на уласку у тело петље. Нека је $i' = i + 1$. Из тела петља види се да је $n' = 10 \cdot n + a_{k-i-1}$. Пошто на основу индуктивне хипотезе важи $n = a_{k-1}10^{i-1} + a_{k-2}10^{i-2} + \dots + a_{k-i}10^0$, важи и да је

  $n' = 10(a_{k-1}10^{i-1}+a_{k-2}10^{i-2}+\dots+a_{k-i}10^0)+a_{k-i-1} = a_{k-1}10^i+a_{k-2}10^{i-1}+\dots+a_{k-i}10+a_{k-i-1}$.

  Пошто је $i' = i + 1$, важи да је

  $n' = a_{k-i}10^{i'-1} + a_{k-2}10^{i'-2} + \dots + a_{k-i'+1}10 + a_{k-i'}$,

  па је инваријанта очувана.

Када се петља заврши, биће учитано k цифара, па ће важити да је $i = k$, тј.

$$n = (a_{k-1} \dots a_{k-i})_{10} = a_{k-1}10^{i-1} + a_{k-2}10^{i-2} + \dots + a_1 \cdot 10^1 + a_0 \cdot 10^0,$$

што значи да n садржи декадну вредност броја формираног од свих k цифара.

1.4 Додатни задаци за вежбу

Задатак: Аритметички троугао

Колики је збир бројева у датом реду следећег троугла?

```
         1
      2  3  4
   5  6  7  8  9
10 11 12 13 14 15 16
        ...
```

Улаз: Редни број k ($1 \leq k \leq 5 \cdot 10^5$), реда троугла чији збир треба израчунати (бројање редова почиње од 1).

Излаз: Збир вредности у задатом реду троугла.

Пример

Улаз	*Излаз*
3	35

Решење

Итерација

До решења се може доћи коришћењем петљи. У првој петљи одређујемо први елемент k-тог реда, а уз то одређујемо и број елемената у њему. Први елемент одређујемо тако што саберемо број елемената у претходних $k - 1$ редова, док број елемената сваког реда тако што у сваком кораку број елемената претходног реда увећавамо за два (сваки наредни ред има тачно два елемента више од претходног). Дакле, у петљи одржавамо почетак и број елемената текућег реда (иницијализујемо их на један, јер први ред почиње од један и има тачно један елемент) и $k - 1$ пута почетак реда увећавамо за број елемената текућег реда, а број елемената текућег реда за два.

Пример. Прикажимо ово на примеру одређивања првог елемента реда 5.

k	pocetak	brojElemenata
1	1	1
2	2	3

```
3   5         5
4   10        7
5   17        9
```

Након тога, у другој петљи одређујемо збир елемената у k-том реду, тако што на збир који иницијализујемо на нулу додајемо један по један елемент тог реда, коришћењем алгоритма сабирања серије бројева – елементи су узастопни природни бројеви, па их је лако набројати.

```
long long zbirRedaTrougla(long long k) {
  // odredjujemo prvi broj u k-tom redu trougla
  long long pocetak = 1;
  long long brojElemenata = 1;
  for (int i = 1; i < k; i++) {
    pocetak += brojElemenata;
    brojElemenata += 2;
  }

  // odredjujemo zbir elemenata u k-tom redu trougla
  long long zbir = 0;
  for (long long i = pocetak; i < pocetak + brojElemenata; i++)
    zbir += i;
  return zbir;
}
```

Анализа сложености. У првој петљи обилазимо k редова којима одређујемо почетак помоћу две операције сабирања. Пажљивом анализом можемо закључити да у реду k има $2k - 1$ елемената (мада ту чињеницу нисмо употребили у програму), који се у другој фази сабирају. Сложеност обе фазе, па и укупног решења је, дакле, $O(k)$. Задатак, наравно, може да се реши и у мањој сложености од ове, без коришћења петљи.

Доказ коректности. Докажимо формално коректност овог поступка. Инваријанта петље је да након m извршавања њеног тела важи да је $i = m+1$ и $1 \leq i \leq k$, као и да променљива `pocetak` садржи први елемент реда $i = m + 1$, а променљива `brojElemenata` садржи број елемената реда $i = m + 1$.

- Пошто се променљива `i` иницијализује на вредност 1, након $m = 0$ корака важи да је $1 = 0 + 1$ и $1 \leq 1 \leq k$. Пошто су и обе променљиве иницијализоване на вредност 1, након $m = 0$ корака петље, променљиве заиста садрже први елеменат и број елемената реда $i = m + 1 = 0 + 1 = 1$.
- Претпоставимо да инваријанта важи након m корака петље и да је услов петље испуњен тј. да је $i < k$.

  Докажимо да тврђење важи и након $m' = m + 1$ извршавања тела и корака петље. Пошто након извршавања корака важи $i' = i + 1$, а пошто је на основу индуктивне хипотезе важило да је $i = m + 1$, важи и да је $i' = m' + 1$. Пошто је услов петље био испуњен, важило је $i < k$, па уз индуктивну претпоставку $1 \leq i \leq k$, важи $1 \leq i' \leq k$.

  Означимо са p и b вредности променљивих `pocetak` и `brojElemenata` на улазу у тело петље, а са p' и b' њихове нове вредности, након извршења тела и корака петље. Анализом додела у телу петље, јасно видимо да је $p' = p+b$ и $b' = b+2$. На основу претпоставке важи да је p први елемент реда $i = m+1$, а да је b број елемената реда $i = m+1$. Сабирањем првог елемента било ког реда у троугла и броја елемената тог реда троугла добија се први елемент наредног реда троугла. Дакле, важи да је $p + b$ први елемент реда $i+1 = m+2$ троугла, па, важи да је $p' = p+b$ први елемент реда $i' = i+1 = m+2 = m'+1$. На основу дефиниције троугла важи јасно и да сваки наредни ред има и два елемента више него претходни. Зато, пошто је b број елемената реда $m + 1$, важи и да је $b' = b + 2$ број елемената реда $i' = m' + 1$.
- На крају петље услов није испуњен, па важи да је $i \geq k$. Уз услов $1 \leq i \leq k$, мора да важи да је $i = k$. На основу инваријанте знамо да променљива `pocetak` садржи број елемената реда $i = m + 1 = k$. Дакле, петља је коректно извршила свој задатак и у променљиву `pocetak` сместила први елемент реда k.

На сличан начин се може формално доказати и коректност друге петље (инваријанта је да након m њених корака променљива `zbir` садржи збир првих m елемената реда k, а да променљива i садржи вредност елемента на позицији m у том реду, ако се позиције броје од 0).

Напомена. Приметимо да смо у овој петљи увели и нову променљиву којом смо регистровали број елемената у текућем реду троугла. То што смо ту променљиву имали на располагању нам је помоглно да ажурирамо

вредност почетног елемента наредног реда, међутим, "кредит" који смо добили на почетку тела петље морали смо да вратимо на крају тако што смо морали да ажурирамо и вредност те променљиве (и тако је припремимо за наредну итерацију). Ова техника је позната под именом **ојачавање индуктивне хипотезе** и често се користи приликом конструкције алгоритама.

Задатак: Двобојка

Напиши програм који организује елементе низа тако да прво иду сви парни елементи, а затим непарни, при чему међусобни редослед парних и непарних елемената није битан. Елементе прво учитати у низ, а затим тај низ трансформисати у линеарном времену (само једним пролазом кроз низ).

Улаз: У првој линији стандардног улаза унети природан број n ($1 \leq n \leq 50000$) - број елемената низа, а у наредној линија унети n природних бројевава у границама од 1 до 1000.

Излаз: На станардни излаз исписати елементе низа уређене на тражени начин, раздвојене са по једним размаком.

Пример

Улаз	*Излаз*
`10`	`2 6 8 10 4 5 3 1 9 11`
`2 5 3 6 1 8 9 10 11 4`	

Решење

Трансформације низа "у месту" - два показивача

Постоји неколико начина да се низ "у месту" трансформише само једним проласком кроз низ (ти приступи имају линеарну временску сложеност). Приказаћемо неколико могућности, све засноване на техници два показивача.

Парни лево, непарни десно - размена наопаких

Претпоставићемо да су у сваком кораку петље познати индекси l и d тако да су елементи низа груписани тако да су:

- сви елементи на позицијама из интервала $[0, l)$ парни,
- сви елементи на позицијама из интервала $[l, d]$ још непрегледани и
- сви елементи на позицијама из интервала (d, n) непарни.

Инваријанта је, дакле, да је распоред елемената у низу облика `ppp???nnn`, где су `l` и `d` позиција првог тј. последњег непознатог елемента (обележеног упитником).

На почетку иницијализујемо l на нулу, а d на $n-1$ (тада су интервали $[0, l)$ и (d, n) празни, а сви елементи у интервалу $[l, d] = [0, n-1]$ су још непрегледани). Петљу у приниципу извршавамо док још има непрегледаних елемената тј. док је $l \leq d$, међутим, у овом задатку можемо је завршити и корак раније. Петља се извршава док је $l < d$ и завршава када је $l = d$, јер какав год да је тај последњи непрегледани елемент на позицији $l = d$, он ће се моћи припојити или левом или десном делу низа и неће бити потребе премештати га.

У телу петље радимо следеће:

- Прво испитујемо да ли је елемент на позицији l паран и ако јесте, онда само увећавамо вредност l за 1.
- У супротном, проверавамо да ли је елемент на позицији d непаран и ако јесте, онда само умањујемо вредност d за 1.
- На крају, ако ниједна од претходне две провере није успела, знамо да је елемент на позицији l непаран, а елемент на позицији d паран, размењујемо их, увећавамо l за 1 и умањујемо d за 1.

Када се петља заврши, елементи су у жељеном редоследу.

Пример. Прикажимо рад алгоритма на једном примеру.

```
l                 d
3  8  7  4  5  1  6  2

   l              d
2  8  7  4  5  1  6  3
```

```
      l           d
2  8  7  4  5  1  6  3

         l     d
2  8  6  4  5  1  7  3

            l  d
2  8  6  4  5  1  7  3

            ld
2  8  6  4  5  1  7  3
```

Имплементација се може направити на следећи начин.

```
// odrzavamo uslov
// [0, l) - parni
// (d, n) - neparni
// [l, d] - nepoznati

// u pocetku su svi nepoznati
int l = 0, d = n-1;
// dok jos ima nepoznatih elemenata
while (l < d) {
  // ako je na mestu l paran, ostavljamo ga na svom mestu i pomeramo
  // se na naredni element
  if (a[l] % 2 == 0)
    l++;
  // ako je na mestu d neparan, ostavljamo ga na svom mestu i
  // pomeramo se na prethodni element
  else if (a[d] % 2 != 0)
    d--;
  else
    // na mestu l je neparan, a na mestu d je paran broj, pa ih
    // razmenjujemo i pomeramo se po oba kraja
    swap(a[l++], a[d--]);
}
```

Анализа сложености. Променљива `l` се увећава, а променљива `d` се умањује све док се не сусретну, што се догађа у тачно n корака, па је укупна сложеност алгоритма $O(n)$.

Доказ коректности. Докажимо и формално коректност претходног алгоритма. Током извршавања важи раније описана инваријанта, а важи и да је $0 \leq l \leq d + 1 \leq n$.

Иницијализацијом вредности l на нулу, а вредности d на $n - 1$ постижемо да су ови услови на почетку задовољени (јер су сви елементи у интервалу $[l, d] = [0, n - 1]$ још непрегледани).

У телу петље радимо следеће:

- Прво испитујемо да ли је елемент на позицији l паран и ако јесте, онда само увећавамо вредност l за 1 чиме наметнута инваријанта остаје да важи. Заиста, парни су били сви елементи из интервала $[0, l)$, паран је и елемент на позицији l, па су парни сви елементи из интервала $[0, l] = [0, l')$, где је $l' = l + 1$, нова вредност променљиве l. Пошто је $d' = d$, елементи на позицијама из интервала (d', n) су непарни.
- У супротном, проверавамо да ли је елемент на позицији d непаран и ако јесте, онда само умањујемо вредност d за 1 чиме наметнута инваријанта опет остају на снази (аргументација је слична оној у претходном случају).
- На крају, ако ниједна од претходне две провере није успела, знамо да је елемент на позицији l непаран, а елемент на позицији d паран, размењујемо их, увећавамо l за 1 и умањујемо d за 1 чиме инваријанта остаје да важи. Заиста, важи да је $l' = l + 1$ и $d' = d - 1$. Сви елементи из интервала позиција $[0, l)$ су парни, а паран је елемент на позицији l (јер је разменом паран елемент са позиције d доведен на позицију l). Зато су парни сви елементи у интервалу $[0, l')$. Слично, пошто су сви елементи у интервалу

позиција (d, n) били непарни, а пошто је након размене непарни елемент са позиције l доведен на позицију d, непарни су и сви елементи из интевала $[0, d')$.

Када се петља заврши, распоред је коректан. Наиме, пошто на крају петље услов $l < d$ није испуњен, а пошто је $l \leq d + 1$, тада је $l = d$ или је $l = d + 1$.

- Ако је $l = d + 1$, знамо да је низ разбијен на сегмент парних елемената на позицијама $[0, l)$ и непарних на позицијама $(d, n) = [d + 1, n) = [l, n)$.
- Размотримо случај $l = d$.
  - Ако је елемент на позицији l паран, пошто су на основу инваријанте парни и сви елементи на позицијама $[0, l)$, знамо да ће парни бити сви елементи на позицијама $[0, l]$, док ће елементи на позицијама $(d, n) = [l + 1, n)$ бити непарни.
  - Слично, ако је елемент на позицији l непаран, тада су парни сви елементи на позицијама $[0, l)$, а непарни су сви елементи на позицијама $[l, n)$ (јер на основу инваријанте знамо да су још непарни и елементи на позицијама $(d, n) = (l, n)$).

  У оба случаја је, дакле, постигнут жељени распоред елемената.

Приметимо да смо прекидом петље када је $l = d$, уштедели једну итерацију петље (што није нарочито значајно), али смо закомпликовали доказ коректности, па се природно поставља питање колико је та оптимизација имала смисла.

Парни, непарни, па непознати

У овом решењу инваријанта је мало другачија. Памтимо индекс k и текући индекс i и претпостављамо да су

- елементи на позицијама из интервала $[0, k)$ парни,
- елементи из интервала $[k, i)$ непарни,
- елементи из интервала $[i, n)$ још непрегледани.

Дакле, намећемо инваријанту да је распоред облика `pppnnn???`, где је i позиција првог непознатог, а k позиција првог непарног елемената.

Размотримо како да из инваријанте закључимо како треба иницијализовати променљиве. Пошто су сви елементи из интервала $[i, n)$ непрегледани, променљиву i морамо инцијализовати на 0. Пошто су сви елементи из интервала $[0, k)$ парни, а $[k, i)$ непарни, и k морамо поставити на 0.

Инваријанта јасно диктира и услов петље. Наиме, петља се извршава док још има непрегледаних елемената тј. док је $i < n$. У сваком кораку петље i се увећава за 1 (користимо класичну бројачку петљу `for` по променљивој i), чиме се сужава интервал непрегледаних елемената.

Размотримо како треба да изгледа тело петље да би инваријанта остала испуњена.

- Ако је елемент на текућој позицији i паран, онда га размењујемо са првим непарним елементом, а то је елемент на позицији k. Изузетак је случај када је $k = i$, када заправо не долази до размене (елемент се мења сам са собом). У оба случаја се k увећава за 1.
- У супротном, елемент на позицији i је непаран он остаје на свом месту и у телу петље није потребно ништа урадити (грану `else` у коду није потребно наводити).

Пример. Прикажимо рад алгоритма на једном примеру.

```
k                   n
3  8  7  4  5  1  6  2
i

k                   n
3  8  7  4  5  1  6  2
   i

   k                n
8  3  7  4  5  1  6  2
      i
```

```
   k                 n
8  3  7  4  5  1  6  2
      i

      k              n
8  4  7  3  5  1  6  2
         i

      k              n
8  4  7  3  5  1  6  2
            i

      k              n
8  4  7  3  5  1  6  2
               i

         k           n
8  4  6  3  5  1  7  2
                  i

            k         n
8  4  6  2  5  1  7  3
                     i
```

Имплементација се може направити на следећи начин.

```
// odrzavamo uslov
// [0, k) - parni
// [k, i) - neparni
// [i, n) - nepoznati

// u pocetku su svi elementi nepoznati
int k = 0;
for (int i = 0; i < n; i++)
  // ako je element paran razmenjujemo ga sa prvim neparnim
  if (a[i] % 2 == 0)
    swap(a[i], a[k++]);
  // a ako je neparan, ne pomeramo ga
```

Анализа сложености. У програму се користи класична бројачка петља `for` која се завршава у n корака, па је укупна сложеност алгоритма $O(n)$.

Доказ коректности. Докажимо и формално коректност претходног поступка. Уз описану инваријанту важи и да је $0 \leq k \leq i \leq n$.

Пошто је на почетку $i = k = 0$, важи да су сви елементи у интервалу $[i, n) = [0, n)$ непрегледани. Интервали $[0, k) = [k, i) = [0, 0)$ су празни, па задовољавају услове, а тривијално Важи и да је $0 \leq k \leq i \leq n$.

У телу петље се врше следеће акције.

- Ако је елемент на текућој позицији i паран, онда га размењујемо са првим непарним елементом, а то је елемент на позицији k. Изузетак је случај када је $k = i$, када заправо не долази до размене (елемент се мења сам са собом). У оба случаја се k увећава за 1. Дакле, на основу инваријанте знамо да су елементи на позицијама $[0, k)$ парни, да је након размене елемент на позицији k паран, па су парни и сви елементи на позицијама $[0, k') = [0, k + 1)$. Елементи на позицијама $[k', i') = [k + 1, i + 1)$ су непарни. Наиме, ако је $k = i$, овај интервал је празан, а ако је $k < i$, тада је пре размене елемент на позицији k био непаран (јер на основу инваријанте знамо да су сви елементи на позицијама $[k, i)$ били непарни, па самим тим и елемент на позицији k, који је сада доведен на позицију i).
- У супротном, елемент на позицији i је непаран он остаје на свом месту. Пошто се у телу петље не дешава ништа, важи $k' = k$ и $i' = i$, а инваријанта прилично очигледно остаје на снази.

По завршетку петље услов $i < n$ није испуњен, па пошто на основу инваријанте важи $0 \leq k \leq i \leq n$, важи да

је $i = n$. Зато је интервал непознатих $[i, n)$ празан, сви елементи из интервала $[0, k)$ су парни, из интервала $[k, i) = [k, n)$ су непарни и постигнут је тражени распоред.

Заустављање се једноставно доказује јер се у сваком кораку сужава интервал непознатих $[i, n)$.

Задатак: Разлика сума до мах и од мах

Уноси се масе предмета, одредити разлику суме маса предмета до првог појављивања предмета највеће масе и суме маса предмета после првог појављивања предмета највеће масе (предмет највеће масе није укључен ни у једну суму).

Улаз: У првој линији стандардног улаза налази се број предмета n ($1 \leq n \leq 50000$). Свака од наредних n линија садржи по један природан број из интервала $[1, 50000]$, ти бројеви представљају масе сваког од n предмета.

Излаз: У првој линији стандарног излаза приказати тражену разлику маса.

Пример

Улаз	*Излаз*
5	-14
10	
13	
7	
13	
4	

Објашњење

Предмет највеће масе је 13. Збир маса пре његовог првог појављивања је 10, а после његовог првог појављивања је 24. Зато је тражена разлика -14.

Решење

Директно решење

Решење до којег се најједноставније долази је да се масе свих предмета учитају у низ, да се након тога пронађе позиција највећег елемента у низу и да се затим одреди збир елемената низа пре те позиције и збир елемената низа после те позиције.

Оптимизовано решење

Задатак можемо решити и без коришћења низова. У овом решењу задатка ћемо комбиновати алгоритам одређивања максимума серије бројева и алгоритам сабирања серије учитаних бројева. У петљи ћемо учитавати једну по једну масу предмета, одржавајући при том збир маса пре прве појаве максималне масе, збир маса после прве појаве максималне масе и саму вредност максималне масе предмета.

На почетку учитавамо масу првог предмета (он постоји по условима задатка), максимум иницијализујемо на ту учитану вредност, а два збира на нулу (јер није виђен ни један предмет ни пре, ни после тог првог предмета који је уједно и прво појављивање до тада максималне масе).

Затим, у петљи учитавамо остале предмете, један по један. Могућа су два случаја.

- Ако је маса учитаног предмета строго већа од до тада максималне масе, онда тај предмет представља прво појављивање предмета максималне масе (он је максимални од свих до тада виђених). Зато је потребно да израчунамо збир маса свих предмета пре њега. Међутим, ми знамо вредност збира маса свих предмета пре ранијег предмета максималне масе, знамо његову масу и знамо збир маса предмета након њега (у тај збир још није укључен текући предмет), тако да је збир предмета пре текућег предмета (новог максимума) једнак збиру ове три вредности. Максимум се ажурира на масу текућег предмета, а збир маса предмета после новог максимума се ажурира на нулу (јер за сада нисмо видели ни један такав предмет).
- У супротном, текући предмет не представља прво појављивање максимума (он је или мањи од максимума, или је једнак максимуму, али није његово прво појављивање). Зато се предмет максимум не мења, не мења се серија предмета пре њега, док се серија предмета након њега продужава текућим предметом. Зато је у овом случају само потребно ажурирати збир маса предмета после максимума тако што се тај збир увећа за масу текућег предмета.

Доказ коректности. Докажимо формално коректност овог алгоритма. Инваријанту петље чине следећи услови:

- $1 \leq i \leq n$.
- Променљива `max` садржи вредност максимума првих i елемената низа (елемената $a_0, \ldots, a_{i-1}$). Претпостављамо да се прво појављивање те вредности јавља на некој позицији m између 0 и $i-1$ тј. да су сви елементи $a_0, \ldots, a_{m-1}$ строго мањи од вредности променљиве `max`, да a_m има вредност `max`, а да сви елементи $a_{m+1}, \ldots, a_{i-1}$ имају вредност већу или једнаку од променљиве `max`.
- Променљива `zbirPreMax` садржи збир свих елемената пре првог појављивања максималног елемента `max` тј. вредност $a_0 + \ldots + a_{m-1}$.
- Променљива `zbirPosleMax` садржи збир свих елемената после првог појављивања максималног елемента `max` тј. вредност $a_{m+1} + \ldots + a_{i-1}$.

Докажимо да су услови заиста инваријанта петље.

- Пре уласка у петљу, променљива `max` има вредност a_0, променљиве `zbirPreMax` и `zbirPosleMax` имају вредност 0, а променљива `i` вредност 1. Тада је $m = 0$, па су сви услови испуњени (скупови елемената $a_0, \ldots, a_{m-1}$ и $a_{m+1}, \ldots, a_{i-1}$ су празни).
- Претпоставимо да услови важе пре уласка у петљу.
  - Претпоставимо да је a_i строго веће од вредности `max`. Тада на основу инваријанте знамо да је a_i строго веће од свих вредности $a_0, \ldots, a_{i-1}$, па се нова вредност максимума јавља на позицији $m' = i$.

    Тада је нова вредност променљиве `zbirPreMax` једнака збиру старих вредности променљивих `zbirPreMax`, `zbirPosleMax` и `max`, па на основу варијанте важи да је та нова вредност једнака $(a_0 + \ldots + a_{m-1}) + a_m + (a_{m+1} + \ldots + a_{i-1})$, што је једнако $a_0 + \ldots + a_{m'-1}$. Ово је тачно збир свих елемената пре нове позиције максимума за коју смо утврдили да је позиција $m' = i$. Уједно за све ове елементе важи да су строго мањи од `max`.

    Нова вредност променљиве `max` једнака је a_i што јесте нова вредност максимума.

    Нова вредност променљиве `zbirPosleMax` једнака је 0, што је једнако збиру вредности $a_{m'+1} + \ldots + a_{i-1}$ (јер је $m' = i$, па је тај скуп елемената празан).
  - Претпоставимо да је a_i мање или једнако од вредности `max`. Тада се прво појављивање максимума у делу низа $a_0, \ldots, a_i$ поклапа са првим појављивањем максимума у делу низа $a_0, \ldots, a_{i-1}$ и налази се такође на позицији m, па је $m' = m$.

    Променљиве `max` и `zbirPreMax` не мењају своје вредности (што је у реду, јер је $m' = m$, па је вредност `max` једнака $a_m = a_{m'}$, док је вредност променљиве `zbirPreMax` једнака $a_0 + \ldots + a_{m'-1} = a_0 + \ldots + a_{m-1}$).

    Променљива `zbirPosleMax` увећава своју вредност за a_i, па, пошто је њена стара вредност била једнака $a_{m+1} + \ldots a_{i-1}$, пошто је $m' = m$ и $i' = i + 1$, важи да је њена нова вредност једнака $a_{m+1} + \ldots + a_i = a_{m'+1} + \ldots + a_{i'-1}$.

На крају петље не важи да је $i < n$, па пошто је $i \leq n$, важи да је $i = n$. Зато на основу инваријанте знамо да променљиве `zbirPreMax` `zbirPosleMax` заиста имају вредности збира свих елемената пре првог и после првог појављивања максималног елемента у делу низа $a_0 \ldots a_{i-1}$, што је заправо цео низ (јер је $i = n$, а n је укупан број елемената).

```
// ukupan broj predmeta
int n;
cin >> n;

// ucitavanje mase prvog predmeta
int m;
cin >> m;
// najveca do sada ucitana masa
int max = m;
// zbir masa pre najvece do sada ucitane
int zbirPreMax = 0;
// zbir masa posle najvece do sada ucitane
```

```
int zbirPosleMax = 0;
for (int i = 1; i < n; i++) {
  // ucitavanje mase sledeceg predmeta
  cin >> m;
  if (m > max) {
    // korekcija najvece mase i zbirova
    zbirPreMax += max + zbirPosleMax;
    max = m;
    zbirPosleMax = 0;
  } else
    zbirPosleMax += m;
}

// prikaz rezultata
cout << zbirPreMax - zbirPosleMax << endl;
```

Задатак: Обртање цифара

Улаз: Са стандардног улаза се уноси природан број n ($0 \leq n \leq 10^9$).

Излаз: На стандардни излаз исписати број који се добија обртањем цифара броја n.

Пример 1

Улаз	*Излаз*
12345	54321

Пример 2

Улаз	*Излаз*
12000	21

Решење

У овом поступку се комбинују два алгоритма. Први је алгоритам за одређивања цифара у декадном запису броја, здесна налево, одређивањем остатка при целобројном дељењу. Други алгоритам је Хорнерова схема за одређивање вредности броја на основу његових цифара слева надесно. Тај алгоритам је описан у задатку Број формиран од датих цифра с лева на десно. Скидамо једну по једну цифру броја n здесна и дописујемо је на резултат, такође здесна.

Пример. Размотримо поступно рад алгоритма на једном примеру.

n	r	i
123000	0	0
12300	0	1
1230	0	2
123	0	3
12	3	4
1	32	5
0	321	6

Коректност овог алгоритма се заснива на следећој инваријанти. Након i корака петље број n садржи број који се добије када се од почетног броја n_0 обрише последњих i цифара, док број r садржи број који се добије обртањем тих последњих i цифара броја n_0.

Имплементација се може направити на следећи начин.

```
int obrni(int n) {
  int r = 0;
  while (n > 0) {
    r = 10*r + n % 10;
    n /= 10;
  }
  return r;
}
```

Доказ коректности. Пошто је програм написан у терминима математичких операција, ако желимо да спроведемо потпуно прецизан, математички формалан доказ, и инваријанту морамо изразити у терминима математичких операција. Нажалост, испоставиће се да крајње нуле у запису броја, које постају водеће нуле у

резултату значајно компликују прецизно изражавање инваријанте и наставак доказа чине прилично мукотрпним. Наиме, број који се обрће може се завршити нулама, тако да на пример бројеви 123, 1230, 12300 сви имају исту вредност броја који се записује истим цифрама као оригинални број али у обрнутом поретку (321). Али број корака петље који ће бити потребан да се та вредност израчуна ће се разликовати у тим ситуацијама (и биће једнак броју цифара броја n). Отуда је битно приметити да је број завршних нула у запису броја битан за предложени алгоритам.

Лема: Услов да је $n_0 = n \cdot 10^i + rev(r) \cdot 10^s$ (где је s број завршних нула у запису броја n_0) је инваријанта петље. При том је број r једнак 0 док је $i \leq s$ (има 0 цифара у запису), односно има тачно $i - s$ цифара у запису када је $i > s$ и водећа цифра му је различита од нуле. Важи и да је $n \geq 0$.

Докажимо ову лему.

- Пре првог уласка у петљу је $n = n_0$, $r = 0$ и $i = 0$, па услов очигледно важи.
- Претпоставимо да тврђење важи на уласку у петљу. Ефекат тела петље је да је $n' = n \operatorname{div} 10$, да је $r' = 10r + n \bmod 10$ и да је $i' = i + 1$. Потребно је да докажемо да вредност $n' \cdot 10^{i'} + rev(r') \cdot 10^s$ остаје једнака n_0. Након извршавања тела петље, она је једнака $(n \operatorname{div} 10) \cdot 10^{i+1} + rev(10r + n \bmod 10) \cdot 10^s$. Ову вредност даље израчунавамо у зависности од односа променљивих i и s.
  - Ако је $i < s$, петља скида нуле са краја записа броја n, тада је $r = 0$ и $n \bmod 10 = 0$. Зато је претходна вредност једнака $(n \operatorname{div} 10) \cdot 10^{i+1} = (10 \cdot (n \operatorname{div} 10) + n \bmod 10) \cdot 10^i = n \cdot 10^i$, што је на основу претпоставке једнако n_0 (јер је и у претходном кораку $r = 0$).
  - Ако је $i = s$ петља је стигла до прве цифре различите од нуле, па је још увек $r = 0$, а $n \bmod 10 \neq 0$. Пошто је $n \bmod 10$ једноцифрен број, важи да је $rev(n \bmod 10) = n \bmod 10$. Пошто је $i = s$, вредност израза који фигурише у инваријанти једнака је $(n \operatorname{div} 10) \cdot 10^{i+1} + (n \bmod 10) \cdot 10^i = (10 \cdot (n \operatorname{div} 10) + n \bmod 10) \cdot 10^i = n \cdot 10^i$, а то је на основу претпоставке једнако n_0 (јер је и у претходном кораку $r = 0$). Важи и да је $r' = n \bmod 10$, па је $0 < r' < 10$ и r' је једноцифрен број. Зато важи да r' има тачно $i' - s = i + 1 - i = 1$ цифара и да му је водећа цифра различита од нуле.
  - Ако је $i > s$, пошто r има тачно $i - s$ цифара, тада важи да је $rev(10r + n \bmod 10) = (n \bmod 10) \cdot 10^{i-s} + rev(r)$. Заиста, израз са леве стране означава број који се добија обртањем редоследа цифара броја који се од броја r добија дописивањем цифре $n \bmod 10$ (она може бити и нула) с десне стране. Што се тиче израза са десне стране, на основу инваријанте знамо да је r број који има тачно $i - s$ цифара и оне одговарају степенима од 10^0 до 10^{i-s-1}. Зато се цифра $n \bmod 10$ множењем са фактором 10^{i-s} поставља као прва цифра иза које следе све цифре записа броја r у обрнутом редоследу. Дакле, леви и десни израз имају исту вредност. На основу тога, знамо да је вредност израза који фигурише у инваријанти након извршавања тела једнака $n' \cdot 10^{i'} + rev(r') \cdot 10^s$ = $(n \operatorname{div} 10) \cdot 10^{i+1} + rev(10r + n \bmod 10) \cdot 10^s$ = $(n \operatorname{div} 10) \cdot 10^{i+1} + ((n \bmod 10) \cdot 10^{i-s} + rev(r)) \cdot 10^s$ = $(10 \cdot (n \operatorname{div} 10) + n \bmod 10) \cdot 10^i + rev(r) \cdot 10^s$ = $n \cdot 10^i + rev(r) \cdot 10^s$, што је једнако n_0 на основу претпоставке. Пошто је $r' = 10r + n \bmod 10$, онда важи да r' има једну цифру више него r (тј. има $i - s + 1 = i' - s$ цифара), а водећа цифра је различита од нуле (водећа цифра се није променила).

Теорема: Након извршавања кода важи да је $r = rev(n_0)$.

Пошто је $n \geq 0$ и није $n > 0$ (јер се петља завршила), важи да је $n = 0$. Зато је $n_0 = rev(r) \cdot 10^s$. тј. $rev(r) = \frac{n_0}{10^s}$. Пошто је водећа цифра броја r различита од нуле и пошто је последња цифра броја $\frac{n_0}{10^s}$ такође различита од нуле (јер n_0 има тачно s завршних нула које се дељењем са 10^s бришу), важи да је $r = rev(\frac{n_0}{10^s})$. Међутим, важи и да је $rev(n_0) = rev(\frac{n_0}{10^s})$, јер $rev(n_0)$ има тачно s водећих нула које се могу обрисати без утицаја на коначни резултат.

Заустављање се доказује прилично једноставно јер је $n \geq 0$ и константно се смањује, па мора доћи до нуле.

Задатак: Растављање на просте чиниоце

Ако је дато неколико простих бројева, њихов производ се може веома лако и брзо одредити. Међутим, ако је дат производ, често је веома тешко одредити просте бројеве који га сачињавају. Напиши програм који што ефикасније решава тај проблем.

Улаз: Са стандардног улаза се уноси један природан број n ($1 \leq n \leq 2 \cdot 10^9$).

Излаз: На стандардни излаз исписати просте чиниоце броја n, уређене од најмањих до највећих, раздвојене размаком.

Пример

Улаз	*Излаз*
900	2 2 3 3 5 5

Решење

Алгоритам факторизације

Потенцијални чиниоци f броја n се испитују редом, у петљи, кренувши од броја 2. У сваком кораку испитује се да ли је број n дељив бројем f и док год јесте дељив, у унутрашњој петљи, он се дели бројем f приј-ављујући при том чинилац f (пошто се у склопу услова унутрашње петље врши провера дељивости n са f, није потребна посебна провера дељивости наредбом гранања пре те петље). Након тога прелази се на следећи потенцијални чинилац (за један већи од претходног). Иако се може помислити да је за сваки потенцијални чинилац потребно проверити да ли је он прост број (јер нас занимају само прости чиниоци), то није потребно радити. Наиме, у поступку претраге који смо навели, ако текући кандидат f није прост, он не може да дели број n, јер смо све његове просте чиниоце већ дељењем уклонили из броја n. На пример, када f достигне вредност 6, број n не може бити дељив њиме јер је претходно исцрпно издељен бројем 2 (а касније и бројем 3). Заиста, ако претпоставимо супротно да f дели n и да је f сложен број, тада би f имао неки прости чинилац мањи од њега и то би уједно био прост чинилац броја n. Међутим, то није могуће јер смо пре увећања броја f на његову текућу вредност утврдили да текући број n не може бити дељив ни једним бројем мањем од f (иначе бисмо га делили са f, а не увећавали f). Дакле сложени чиниоци се елиминишу тако што се утврди да текући број n није дељив њима, што је много ефикасније него примењивати на њих тест простости.

У најједноставнијој имплементацији, описани поступак траје све док се број n дељењем својим чиниоцима не сведе на број 1.

Анализа сложености. Иако коректан, алгоритам који се завршава свођењем броја на 1 је прилично неефикасан и за бројеве који имају велике просте чиниоце ради веома споро (покушајте извршавање програма нпр. за број 1000000007). Проблем настаје јер се делиоци последњег простог чиниоца испитују све док се не дође до самог тог броја. С обзиром на ограничење бројевног типа, број чинилаца можемо сматрати практично константним (не може их бити више од 32), па је сложеност $O(f_k)$, где је f_k највећи прост фактор полазног броја, што је $O(n)$ када је n прост број.

На срећу, алгоритам је једноставно поправити, тако што се растављање заустави чим се утврди да је текућа вредност променљиве n прост број (а видећемо да за то није потребно чекати да вредност f достигне n).

Пример. Прикажимо рад алгоритма на једном примеру.

n	f	cinioc
3300	2	2
1650	2	2
825	2	-
825	3	3
275	3	-
275	4	-
275	5	5
55	5	5
11	5	-
11	6	-
11	7	-
11	8	-
11	9	-
11	10	-
11	11	11
1	11	-

Имплементација се може направити на следећи начин.

```
// ucitavamo broj koji treba rastaviti na proste cinioce
int n;
cin >> n;
int f = 2; // prvi potencijalni prost cinilac je 2
// dok se broj deljenjem sa svojim prostim ciniocima ne svede na 1
```

```
while (n > 1) {
  while (n % f == 0) {   // dok je n deljivo sa f
    cout << f << " ";    //   prijavljujemo pronađeni prost cinilac
    n /= f;              //   delimo broj njime
  }
  f = f + 1;             //   prelazimo na sledeceg kandidata
}
cout << endl;
```

Доказ коректности. Докажимо коректност претходног алгоритма и формално, коришћењем технике инваријанти петље. Централна инваријанта петље је то да текућа вредност промењљиве n није дељива ни једним бројем из интервала $[2, f)$, као и да је почетни број n_0 производ до сада исписаних простих бројева и текуће вредности промењљиве n.

Пре уласка у петљу је $f = 2$, па је интервал $[2, f)$ празан и први део инваријанте тривијално важи. Важи и да је $n = n_0$ и да ниједан број није исписан, па и други део инваријанте важи.

Претпоставимо да инваријанта важи на уласку у петљу.

- Ако је n дељив бројем f, исписује се број f. Он је очигледно чинилац броја n. Претпоставимо да је сложен и да се може раставити као $f = f_1 \cdot f_2$, за $f_1 > 1$ и $f_2 > 1$. Тада би број n био дељив са f_1 и са f_2 који припадају интервалу $[2, f)$, што је супротно инваријанти, па закључујемо да d мора бити прост. Дељењем броја n са f добија се нови број n', који такође није дељив ни са једним бројем из $[2, f)$, па први део инваријанте остаје очуван. Почетна вредност n_0 остаје једнака производу исписаних бројева и текуће вредности n (јер је то важило на основу инваријанте, исписано је f, а n је подељено са f), па и други део инваријанте остаје очуван.
- Ако n није дељив бројем f, тада се f увећава за 1 (нека је $f' = f + 1$). Да би први део инваријанте био одржан, треба да важи да n није дељив ни са једним бројем из интервала $[2, f') = [2, f]$, но то важи, јер на основу инваријанте знамо да n није дељив ни са једним бројем из интервала $[2, f)$, а на основу експлицитне провере услова знамо да n није дељиво ни са f. Ниједан број није исписан, нити је број n промењен, па други део инваријанте тривијално наставља да важи.

На основу инваријанте знамо да је n_0 једнак производу свих исписаних простих чинилаца и тренутне вредности броја n. Пошто је када се алгоритам заврши она једнака 1, исписана је проста факторизација броја n.