ზაზა გამეზარდაშვილი

ალგორითმები

ქუთაისი, 2004

1. ელემენტარული ამოცანები

1.1. მოცემულია ორი a და b რიცხვი (ვთქვათ a=11, b=7). გაუცვალეთ მათ მნიშვნელობები მესამე ცვლადის გამოყენების გარეშე.

მითითება. ცხადია, რომ უშუალოდ a=b; b=a; ბრძანებები სასურველ შედეგამდე ვერ მიგვიყვანენ, რადგან პირველი ბრძანების შემდეგ ორივე ცვლადის მნიშვნელობა 7-ის ტოლი გახდება და ამავე მნიშვნელობებს შეინარჩუნებენ ცვლადები მეორე ბრძანების შემდეგაც. დამატებითი (ვთქვათ, c) ცვლადის გამოყენებით ამოცანის ამოხსნა მარტივია: c=a; a=b; b=c. თუმცა ამოცანის ამოხსნა მის გარეშეც შეიძლება: a=a-b; b=a+b; a=b-a.

1.2. დაწერეთ პროგრამა, რომელიც მოცემული სამი a, b და c რიცხვებისათვის შეამოწმებს, შესაძლებელია თუ არა შესაბამისი სიგრძეების მონაკვეთებით სამკუთხედის აგება. დადებითი პასუხის შემთხვევაში პროგრამამ გამოიტანოს – "YES", თუკი სამკუთხედი არ აიგება – "NO".

მითითება. სამი მონაკვეთით სამკუთხედის აგებისთვის აუცილებელია, რომ ნებისმიერი ორი გვერდის ჯამი მესამეს აღემატებოდეს. პროგრამირების ენაზე ეს ასე ჩაიწერება:

```
IF ((a+b>c) AND (a+c>b) AND (b+c>a))  THEN  WRITE ("YES")  ELSE WRITE ("NO")
```

1.3. დაწერეთ პროგრამა, რომელიც მოცემული სამი a, b და c რიცხვებისათვის შეამოწმებს, შესაძლებელია თუ არა შესაბამისი სიგრძეების მონაკვეთებით მართკუთხა სამკუთხედის აგება. დადებითი პასუხის შემთხვევაში პროგრამამ გამოიტანოს – "YES", თუკი სამკუთხედი არ აიგება – "NO".

მითითება. სამკუთხედი მხოლოდ მაშინაა მართკუთხა, თუ მისი გვერდები აკმაყოფილებენ პითაგორას თეორემას:

```
IF ((a*a+b*b=c*c) OR (a*a+c*c=b*b) OR (b*b+c*c=a*a)) THEN WRITE("YES")   ELSE WRITE("NO")
```

1.4. დაწერეთ პროგრამა, რომელიც მოცემული სამი განსხვავებული a, b და c რიცხვებისათვის ეკრანზე გამოიტანს მხოლოდ მათ შორის უდიდესს.

მითითება.

```
IF ((a>b) AND (a>c)) THEN WRITE(a)
IF ((b>a) AND (b>c)) THEN WRITE(b)
IF ((c>a) AND (c>b)) THEN WRITE(c)
```

1.5. იპოვეთ 200-დან 400-მდე ყველა კენტი რიცხვის ჯამი.

მითითება.

```
s=0
FOR  i=200 TO 400 {
        IF (i MOD 2)=1 THEN s=s+i  }
WRITE (s)
```

1.6. იპოვეთ 100-დან 800-მდე ყველა იმ რიცხვის ჯამი, რომელიც არაა 7-ის ჯერადი.

მითითება.

```
s=0
FOR  i=100 TO 800 {
        IF (i MOD 7)<>0  THEN s=s+i  }
WRITE (s)
```

1.7. იპოვეთ ყველა სამნიშნა რიცხვი, რომლებიც უნაშთოდ იყოფიან საკუთარი ციფრების ჯამზე.

მითითება.
```
FOR  i=100 TO 999 {
a=i MOD 10;   b=i DIV 100
c=(i MOD 100) DIV 10
IF i mod (a+b+c)=0 THEN WRITE (i)  }
```

1.8. იპოვეთ ყველა ექვსნიშნა რიცხვი, რომელთა პირველი 3 და ბოლო 3 ციფრის ჯამი ტოლია.

მითითება.
```
FOR  i=100000 TO 999999 {
a1=i MOD 10; a2=(i mod 100) DIV 10; a3=(i MOD 1000) DIV 100
b1=(i MOD 10000) DIV 1000; b2=(i MOD 100000) DIV 10000; b3=i DIV 100000
IF a1+a2+a3=b1+b2+b3 THEN WRITE (i)  }
```

1.9. მოცემულია მთელი რიცხვი n (1<n<2000000000). იპოვეთ მისი ციფრთა ჯამი.

მითითება.
```
READ (n)
WHILE n>0  {
        s=s+n mod 10
        n=n div 10  }
WRITE (S)
```

1.10. მოცემულია ორი მთელი რიცხვი n1 და n2 (1920<n1<n2<2020), რომლებიც აღნიშნავენ გარკვეულ წლებს. იპოვეთ დღეების რაოდენობა ამ წლების 1 იანვრებს შორის.

მითითება. აქ გასათვალისწინებელია, რომ ნაკიან და ჩვეულებრივ წელიწადს დღეების განსხვავებული რაოდენობა აქვთ. ცნობილია, რომ ნაკიანი წლების აღმნიშვნელი რიცხვები უნაშთოდ იყოფიან 4-ზე.

```
READ (n1, n2)
FOR  i=n1  to  n2  {
        IF i mod 4=0 THEN s=s+366  ELSE s=s+365
WRITE (s)
```

1.11. მოცემულია განსხვავებული სიგრძის მქონე ორი სიტყვა (არცერთის სიგრძე 10 სიმბოლოზე მეტი არაა). დაწერეთ პროგრამა, რომელიც გააერთიანებს ამ სიტყვებს ისე, რომ წინ უფრო გრძელი სიტყვა მოთავსდეს.

შემავალი მონაცემები:	გამომავალი მონაცემები:
ter compu	computer

მითითება.
```
READ (word1)
READ(word2)
k1=LENGTH(word1); k2=LENGTH(word2);
IF k1>k2 THEN word=word1+word2 ELSE word=word2+word1
WRITE (word)
```

1.12. მოცემულია კორექტული ტექსტი, რომლის სიგრძე არ აღემატება 250 სიმბოლოს. იპოვეთ ტექსტში სიტყვების რაოდენობა.

შემავალი მონაცემები:	გამომავალი მონაცემები:
This is a correct text	5

მითითება. ტექსტის კორექტულობა ნიშნავს, რომ ყოველი ორი მეზობელი სიტყვა თითო ჰარითაა დაშორებული, ამიტომ სიტყვების რაოდენობა ტექსტში ერთით მეტი იქნება მასში ჰარების რაოდენობაზე.

```
READ (txt)
k=LENGTH(txt);   n=0
FOR i=1 TO k {
        IF txt[i]=' '  THEN n=n+1  }
WRITE (n+1)
```

1.13. მოცემულია სიტყვა, რომლის სიგრძე 10 სიმბოლოზე ნაკლები არაა, წაშალეთ მასში პირველი სამი და ბოლო სამი სიმბოლო.

შემავალი მონაცემები: გამომავალი მონაცემები:
mmmcomputerhhh computer

მითითება.
```
READ (word);
k=LENGTH(word)
DELETE (word, k-2, 3)
DELETE (word,1,3)
WRITE (word)
```

1.14. მოცემულია ორი სიტყვა, რომელთაგან პირველი ლუწი რაოდენობის სიმბოლოებისაგან შედგება. დაწერეთ პროგრამა, რომელიც მეორე სიტყვას პირველის შუაში ჩასვამს.

შემავალი მონაცემები: გამომავალი მონაცემები:
matabeli
re matarebeli

მითითება.
```
READ (word1)
READ(word2)
k=LENGTH(word1) div 2
INSERT (word2, word1, k)
WRITE (word1)
```

1.15. მოცემულია ტექსტი, რომლის სიგრძე არ აღემატება 250 სიმბოლოს. სიმბოლო 'a' ამ ტექსტში შეცვალეთ 'o' სიმბოლოთი.

შემავალი მონაცემები: გამომავალი მონაცემები:
gaad camputer good computer

მითითება.
```
READ (txt)
k=LENGTH(txt);  ww='o'
FOR i=1 TO k {
        IF txt[i]='a' THEN {     txt[i]='o'     }
WRITE (txt)
```

1.16. რევერსი

დაწერეთ პროგრამა, რომელიც კლავიატურიდან შემოტანილ სიტყვას ეკრანზე შებრუნებულად გამოიტანს. სიტყვის სიგრძე არ აღემატება 100 სიმბოლოს.

შემავალი მონაცემები: გამომავალი მონაცემები:
computer retupmoc

მითითება.
```
READ (word)
k=LENGTH(word)
FOR i=k TO 1 {
        WRITE (word[i])  }
```

1.17. პალინდრომი

(ამიერკავკასიის სტუდენტთა პირადი პირველობა, 2002 წელი)

პალინდრომი ეწოდება ისეთ სიტყვას, რომელიც ერთნაირად იკითხება წინიდან და უკნიდან. მაგალითად, სიტყვა deed პალინდრომია, ხოლო სიტყვა read – არა. დაწერეთ პროგრამა, რომელიც გაარკვევს, პალინდრომია თუ არა მოცემული სიტყვა.

შემავალი ფაილი შეიცავს ერთადერთ სიტყვას, რომელიც არაა 254 სიმბოლოზე გრძელი. სიტყვა შედგება მხოლოდ ლათინური მთავრული ასოებისაგან. გამომავალი ფაილი შეიცავს მხოლოდ ერთ სიტყვას: Yes ან No.

შემავალი მონაცემები: გამომავალი მონაცემები:
DEED Yes
READ No

მითითება. რაიმე d ცვლადს მივანიჭოთ 0 და ციკლში წყვილ-წყვილად შევადაროთ სიტყვის ბოლოებიდან თანაბრად დაშორებული სიმბოლოები: პირველი – ბოლოს, მეორე – ბოლოსწინას და ა.შ.

თუკი ვიპოვით ერთ მაინც განსხვავებულ წყვილს, d-ს მივანიჭოთ 1 და ციკლი შევწყვიტოთ. პასუხი გამოვიტანოთ d-ს მნიშვნელობის მიხედვით ციკლის შემდეგ.

```
READ (word)
k=LENGTH(word);    d=0;   j=k div 2
FOR i=1 TO j   {
        IF word[i]<>word[k-i+1] THEN {d=1; break}   }
IF d=0 THEN WRITE ('yes') ELSE WRITE ('no')
```

1.18. ტოლდიდი მართკუთხედები

მართკუთხედებს ეწოდებათ ტოლდიდი, თუ მათი ფართობები ტოლია. დაწერეთ პროგრამა, რომელიც მოცემული მთელი მნიშვნელობის მქონე ფართობისათვის იპოვის ყველა განსხვავებულ ტოლდიდ მართკუთხედს, რომელთა გვერდებიც ასევე მთელი რიცხვებით აღიწერება. გვერდების გაცვლით მიღებული მართკუთხედებს განსხვავებულად ნუ ჩათვლით.

შემავალი მონაცემები: ერთადერთი მთელი რიცხვი n – მართკუთხედის ფართობი (1≤n≤50000).

გამომავალი მონაცემები: n ფართობის მქონე ყველა განსხვავებული ტოლდიდი მართკუთხედის გვერდები.

შემავალი მონაცემების მაგალითი:	გამომავალი მონაცემები ნაჩვენები მაგალითისათვის:
12	1*12 2*6 3*4

მითითება. მართკუთხედის ფართობი წარმოადგენს მისი მიმდებარე ორი გვერდის ნამრავლს, ამიტომ ამოცანა შეგვიძლია განვიხილოთ როგორც რაიმე მთელი რიცხვის შემადგენელ მამრავლთა ყველა შესაძლო წყვილის პოვნა. მამრავლთა პოვნისას განვიხილოთ მხოლოდ ის წყვილები, რომლებშიც პირველი მამრავლი მეტია ან ტოლი მეორე მამრავლზე. ამ შემთხვევაში ცხადია, რომ პირველი მამრავლი არასოდეს გადააჭარბებს კვადრატულ ფესვს მოცემული ფართობიდან.

```
READ (n)
k=INT(SQRT(n))
FOR i=1 TO k {
        IF n MOD i=0 THEN { write(i,'*',n DIV i)   }
```

1.19. დაწერეთ პროგრამა, რომელიც მოცემული ნატურალური n (n<10000) რიცხვისათვის ეკრანზე გამოიტანს ამ რიცხვის ჩანაწერს ორობით სისტემაში.

მითითება. ამოცანის პირობის თანახმად n<10000, ამიტომ ჩვენს მიერ გამოსატანი ორობითი რიცხვი არ იქნება 14 თანრიგზე მეტი (10000-თან ნაკლებობით ყველაზე ახლოს მყოფი 2-ის ხარისხი არის 213=8192).

```
READ (n)
k=8192
FOR i=1 TO 14 {
        IF n>=k THEN { write('1'); n=n-k } ELSE WRITE ('0')
        k=k div 2 }
```

1.20. დაწერეთ პროგრამა, რომელიც წაიკითხავს 0-ებისა და 1-ებისაგან შედგენილ სიმბოლოთა ჯაჭვს (სიგრძე 14 სიმბოლო) და ეკრანზე გამოიტანს შესაბამისი რიცხვის ჩანაწერს ათობით სისტემაში.

მითითება.

```
READ (w)
n=0
FOR i=14 TO 1 {
        IF w[i]=1 THEN n=n+214-i  }
WRITE (n)
```

1.21. დაწერეთ პროგრამა, რომელიც მოცემული ნატურალური n (n<10000) რიცხვისათვის ეკრანზე გამოიტანს ყველა მარტივ რიცხვს 2-დან n-მდე.

მითითება. ამ ამოცანის ამოხსნა დიდი n-ებისათვის მოითხოვს რთულ ალგორითმს, რომელიც აღწერილია "რიცხვთა თეორიის" თავში. მცირე n-ებისათვის კი ამოცანა ასე შეიძლება ამოიხსნას:

```
READ (n)
FOR i=2 TO n {
        k=0;       p=INT(SQRT(i))
        FOR j=1 TO p {
                IF i mod j=0 THEN {k=1; break}   }
        IF k=0 THEN WRITE (i) }
```

1.22. დაწერეთ პროგრამა, რომელიც მოცემულ ნატურალური n (n<10000) რიცხვს დაშლის მარტივ მამრავლებად.

მითითება. ეს ამოცანა დიდი n-ებისათვის რეალურ დროში ამოუხსნელად ითვლება. მცირე n-ებისათვის კი ამოცანა ასე შეიძლება ამოიხსნას:

```
READ (n)
p=n div 2;    i=2
WHILE i<=p {
        IF n mod i=0 THEN {n=n div i; WRITE (i) }  ELSE i=i+1 }
```

1.23. იპოვეთ ორი მოცემული ნატურალური m და n რიცხვის უდიდესი საერთო გამყოფი.

მითითება. ამ ამოცანის ამოხსნის ეფექტური ალგორითმი (ევკლიდეს ალგორითმი) აღწერილია "რიცხვთა თეორიის" თავში. მცირე n-ებისათვის კი ამოცანა ასე შეიძლება ამოიხსნას:

```
READ (m.n)
FOR i=m TO 1 {
        IF ((m mod i=0) AND (n mod i=0)) THEN WRITE (i)
```

1.24. მინიმიზაცია

ვთქვათ, ნატურალურ რიცხვ N-ისათვის აწარმოებენ შემდეგ ორ ოპერაციას: ა) თუ რიცხვი ლუწია, ჰყოფენ 2-ზე. ბ) თუ რიცხვი კენტია, ამრავლებენ 3-ზე და უმატებენ ერთს. შემდეგ იგივე ოპერაციას იმეორებენ მიღებულ რიცხვზე მანამ, ვიდრე რომელიმე ოპერაციის შემდეგ 1-ს არ მიიღებენ (მათემატიკურად დამტკიცებულია, რომ 1 ნებისმიერი ნატურალური რიცხვისაგან დაწყების შემთხვევაში მიიღება).

ოპერაციათა რაოდენობას, რომელიც საჭიროა ნატურალური რიცხვ N-ისაგან 1-ის მისაღებად, ვუწოდოთ ნატურალურ რიცხვ N-ის მინიმიზაციის სიგრძე. დაწერეთ პროგრამა, რომელიც იპოვის N1-დან N2-მდე ნატურალურ რიცხვებს შორის რომელ K რიცხვს აქვს მინიმიზაციის უდიდესი სიგრძე.

შემავალი მონაცემები:	გამომავალი მონაცემები:
N1	K
N2 (<N1<N2<20000)	K რიცხვის მინიმიზაციის სიგრძე

მაგ.:

შემავალი მონაცემები:	გამომავალი მონაცემები:
100	129
160	121

მითითება.

```
READ (n1,n2)
    FOR i=n1 to n2 DO {
        numb=i
        numb_min=0
        WHILE numb>1 DO {
```

```
        IF numb MOD 2=0 THEN numb=numb DIV 2 ELSE numb=3*numb+1
        numb_min=numb_min+1 }
    IF numb_min>max THEN max=num_min; numb_max=i}
WRITE  (numb_max, max)
```

1.25. წილადის პერიოდი

განვიხილოთ m/n წესიერი წილადი, სადაც m და n ნატურალური რიცხვებია. თუკი m-ის n-ზე გაყოფისას გაყოფა შეწყდა, იტყვიან, რომ m/n წილადის პერიოდი არის 0, ხოლო თუ გაყოფის პროცესი უსასრულოდ გაგრძელდა, მაშინ განაყოფში აუცილებლად მოიძებნება რიცხვთა განმეორებადი მიმდევრობა, რომელსაც წილადის პერიოდს უწოდებენ.

დაწერეთ პროგრამა, რომელიც იპოვის წესიერი წილადის პერიოდს.

შემავალი მონაცემები:	გამომავალი მონაცემები:
M	m/n წილადის პერიოდის სიგრძე
N	m/n წილადის პერიოდი
1<M<N<1000	
შემავალი მონაცემების მაგალითი:	გამომავალი მონაცემები მოცემული მაგალითისთვის:
6	6
7	0.(857142)

მითითება. შემოვიღოთ 1000-ელემენტიანი ერთგანზომილებიანი k მასივი და შევავსოთ 0-ებით. რადგან მოცემული წილადი წესიერია, თავიდან m რიცხვი, შემდეგში კი გაყოფის შედეგად დარჩენილი ნაშთი ციკლში გავამრავლოთ 10-ზე და გავყოთ n-ზე. თითოეულ ბიჯზე მიღებული ნაშთის შესაბამის ინდექსზე k მასივში ჩავწეროთ 1-იანი – თუკი იქ 0 დაგვხვდება, ხოლო თუკი 1-იანი დაგვხდება – პროცესი შევწყვიტოთ, რადგან ეს ნიშნავს, რომ ნაშთი განმეორდა და ყველაფერი თავიდან იწყება. ქვემოთ მოყვანილ პროგრამაში ასეთი ციკლი ორჯერ მეორდება (ანუ k მასივის ელემენტები მეორე ციკლში 2-მდე იზრდებიან და პროცესი მხოლოდ ნაშთის მესამედ განმეორებისას წყდება), ეს დაკავშირებულია ე.წ. "წინაპერიოდთან", რომელმაც შეიძლება ხელი შეგვიშალოს პერიოდის სიგრძის განსაზღვრისას. მაგალითად, წილად 11/12-ში მიიღება 0,9166666..., აქ ციფრები 9 და 1 არ შედიან პერიოდში, მაგრამ k მასივის შევსებისას მათ მისაღებად საჭირო ნაშთების შესაბამისი ელემენტები 1-ის ტოლი გახდებიან. ამიტომ თუკი პერიოდის მიღების შემდეგ პროცესს გავაგრძელებთ პერიოდის კიდევ ერთხელ განმეორებამდე, წინაპერიოდი პასუხზე გავლენას უკვე ვეღარ მოახდენს.

```
 READ (m, n)
j=0;  c=m;
FOR i=1 TO 1000 {
c=c*10;    c=c MOD n;
        IF  k[c]=1 then BREAK else k[c]=1; }
FOR i=1 TO 1000 {
        c=c*10;     d=c DIV n;  c=c MOD n;
        IF c=0 THEN { j=0; break; }
        IF  k[c]=2 THEN BREAK ELSE { j=j+1; k[c]=k[c]+1; WRITE (d); } }
WRITE (j);
```

1.26. კალენდარი

დაწერეთ პროგრამა, რომელიც იპოვის მოცემული თარიღის შესაბამისი დღის დასახელებას. თარიღი მოცემული იქნება ფორმატით "დდ.თთ.წწწწ" და წლების აღმნიშვნელი რიცხვი მოთავსებულ იქნება 2000..2100 დიაპაზონში.

შემავალი მონაცემები:	გამომავალი მონაცემები:
22.03.2002	პარასკევი

მითითება. ამოცანის ამოსახსნელად საჭიროა რაიმე თარიღის აღება ათვლის წერტილად. მოცემული დიაპაზონისათვის უმჯობესი იქნება ავიღოთ 2000 წლის 1 იანვარი, რომელიც შაბათი იყო. ახლა უნდა დავთვალოთ დღეების რაოდენობა 2000 წლის 1 იანვრიდან შემავალ მონაცემში ნაჩვენებ თარიღამდე. ნაკიანი წლების რაოდენობის გამოსათვლელად უნდა გამოვიყენოთ ის ფაქტი, რომ ასეთი წლების აღმნიშვნელი რიცხვები უნაშთოდ იყოფიან 4-ზე. დღეთა მიღებული რაოდენობა უნდა გავყოთ 7-ზე და დარჩენილი ნაშთის შესაბამისად გადავთვალოთ დღის დასახელება (თუ ნაშთი 0-ია, მოცემული დღე შაბათი ყოფილა, თუ ნაშთი 1-ია – კვირა და ა.შ).

1.27. "ნავსი" დღეები

დაწერეთ პროგრამა, რომელიც იპოვის თუ როდის ემთხვევა XXI საუკუნეში თვის 13 რიცხვი ორშაბათს.

მითითება. წინა ამოცანის მსგავსად აქაც საჭიროა რაიმე თარიღის აღება ათვლის წერტილად. უმჯობესი იქნება ავიღოთ 2001 წლის 1 იანვარი – ორშაბათი (ვისაც მიაჩნია, რომ XXI საუკუნე 2000 წლის 1 იანვარს დაიწყო, შეუძლია შესაბამისი თარიღით ისარგებლოს). სხვა მოქმედებებიც წინა ამოცანის მსგავსია, იმ განსხვავებით, რომ აქ თარიღს ჩვენვე ვუკეთებთ გენერაციას (13.xx.xxxx), შემდეგ კი ვამოწმებთ, არის თუ არა ეს თარიღი ორშაბათი.

1.28. იპოვეთ ერთგანზომილებიანი n-ელემენტიანი (n<1000) მასივის ყველაზე დიდი ელემენტის მნიშვნელობა. მასივის ელემენტები მოთავსებულია დიაპაზონში: -30000..30000.

მითითება.
```
READ (n, a[n])
max=-maxint
FOR i=1 TO n {
        IF a[i]>maxint THEN max=a[i]  }
WRITE [max]
```

1.29. ერთგანზომილებიანი n-ელემენტიანი (n<1000) მასივი შევსებულია ნატურალური რიცხვებით 1-დან 1000-მდე. დაწერეთ პროგრამა, რომელიც იპოვის თუ რამდენი რიცხვია 300-ზე მეტი.

მითითება.
```
READ (n, a[n])
raod=0
FOR i=1 TO n {
        IF a[i]>300  THEN raod=raod+1  }
WRITE [raod]
```

1.30. ერთგანზომილებიანი n-ელემენტიანი (n<1000) მასივი შევსებულია ნატურალური რიცხვებით 1-დან 1000-მდე. იპოვეთ იმ ელემენტების რაოდენობა, რომლებიც აღემატებიან საკუთარ ინდექსს.

მითითება.
```
READ (n, a[n])
raod=0
FOR i=1 TO n {
        IF a[i]>i  THEN raod=raod+1  }
WRITE [raod]
```

1.31. ერთგანზომილებიანი n-ელემენტიანი (n<1000) მასივი შევსებულია ნატურალური რიცხვებით 1-დან 1000-მდე. იპოვეთ ყველაზე გრძელი ზრდადი მიმდევრობის სიგრძე და პირველი წევრის ნომერი.

მითითება.
```
READ (n, a[n])
max=1; nom=1; max1=1; nom1=1
FOR i=2 TO n {
        IF a[i]>a[i-1] THEN max1=max1+1 ELSE { max1=1; nom1=i }
        IF max1>max THEN { max=max1; nom=nom1}
WRITE (max, nom)
```

1.32. ერთგანზომილებიანი n-ელემენტიანი (n<1000) მასივი შევსებულია ნატურალური რიცხვებით 1-დან 100-მდე. რომელი რიცხვი გვხდება მასივში ყველაზე მეტად და რამდენჯერ.

მითითება.
```
READ (n, a[n])
FOR i=1 TO n {
        b[a[i]]=b[a[i]]+1 }
max=0; nom=0
FOR i=1 TO n {
        IF b[i]>max THEN {max=b[i]; nom=i}
WRITE (max, nom)
```

1.33. ძროხათა სადგომი

(USACO, 2000-01 წელი, ზამთრის პირველობა, ორივე დივიზიონი)

ფერმერ ჯონს ჰყავს N (1≤N≤5000) ძროხა, რომლებიც ისვენებენ K სადგომის მქონე ბოსელში. სადგომები გადანომრილია 0-დან (K-1)-მდე. i-ურ ძროხას აქვს უნიკალური დამღა, რომელიც წარმოადგენს დადებით მთელ რიცხვს – S_i (1≤ S_i ≤1000000). ძროხებმა იციან, რომ მათი სადგომის ნომერია S_i mod K. ძროხებს, რა თქმა უნდა, არ სურთ, რომ სადგომით ერთდროულად რამდენიმე მათგანმა ისარგებლოს, ამიტომ ძროხათა რაოდენობისა და მათი დამღების გათვალისწინებით იპოვეთ ისეთი მინიმალური K, რომ ძროხების არცერთი წყვილი ერთ სადგომში არ მოხვდეს.

შემავალი მონაცემები: პირველ სტრიქონში მოცემულია ერთი მთელი რიცხვი N, მეორიდან (N+1) სტრიქონამდე მოცემულია თითო მთელი რიცხვი, რომლებიც წარმოადგენენ ძროხათა დამღებს.

გამომავალი მონაცემები: ერთადერთ სტრიქონში ერთადერთი მთელი რიცხვი K.

შემავალი მონაცემების მაგალითი (ფაილი bed.in)	შესაბამისი გამომავალი მონაცემები (ფაილი bed.out)
5 4 6 9 10 13	8

მითითება. საზოგადოდ ასეთი რიცხვის პოვნა სირთულეს არ წარმოადგენს: ავიღებთ S_i-ის უდიდეს მნიშვნელობაზე ერთით მეტ რიცხვს და მიმდევრობის თითოეული წევრის მასზე გაყოფისას განსხვავებულ ნაშთებს მივიღებთ, მაგრამ რადგან საუბარია ამ პირობის დამაკმაყოფილებელ უმცირეს ნატურალურ რიცხვზე, უფრო მცირე პასუხი უნდა ვეძებოთ. აქვე შევნიშნოთ, რომ აზრი არა აქვს n-ზე ნაკლებ რიცხვებში პასუხის ძებნას, რადგან შეუძლებელია n-ზე ნაკლებმა რიცხვმა n ან მეტი რაოდენობის განსხვავებული ნაშთი მოგვცეს.

```
READ (n, S[n]);
FOR i=n TO S[n]+1 {
        a=0
        FOR j=1 TO n {
                k=S[j] MOD i
                IF b[k]=0 THEN b[k]=1 ELSE { a=1; break }   }
        IF a=0 THEN { WRITE (i); break }
        FOR j1=1 TO n {
                b[j1]=0 }       }
```

1.34. ერთგანზომილებიანი n-ელემენტიანი (n<1000) მასივი შევსებულია ნატურალური რიცხვებით 1-დან 1000-მდე. იპოვეთ იმ 10-ელემენტიანი ქვემიმდევრობის პირველი წევრი, რომლის ელემენტთა ჯამი მაქსიმალურია.

მითითება. შემოვიღოთ b მასივი, რომლის i-ურ ელემენტში ჩავწერთ საწყისი a მასივის i-დან (i+9)-მდე ელემენტების ჯამს. შემდეგ ვიპოვით b მასივის უდიდეს ელემენტს, რომელიც

საკუთარ ინდექსთან ერთად ჩვენი ამოცანის პასუხი იქნება. რეალურად b მასივი 9 ელემენტით მცირე იქნება a მასივზე, მაგრამ რადგან წინასწარ არ ვიცით a მასივში ელემენტების რაოდენობა, ისინი ერთნაირად უნდა აღვწეროთ.

ამოცანის ამოხსნა დამატებითი მასივის გარეშეც შეიძლება:

```
READ (n, a[n])
max=0; nom=0
FOR i=1 TO n-9 { s=0
        FOR j=i TO i+9 { s=s+a[j] }
        IF max<s THEN { max=s; nom=i}  }
WRITE (max, nom)
```

არსებობს უფრო ეკონომიური გზაც, რომელიც ასევე არ საჭიროებს დამატებით მასივს:

```
READ (n, a[n])
FOR i=1 TO 10 { s=s+a[i] }
max=s; nom=1
FOR i=2 TO n-9 {
        s=s+a[i+9]-a[i-1]
        IF max<s THEN { max=s; nom=i}  }
WRITE (max, nom)
```

1.35. მოცემულია ”(” და ”)” მრგვალი ფრჩხილებისაგან შემდგარი სიმბოლოთა ჯაჭვი, რომლის სიგრძე არ აღემატება 250-ს. დაწერეთ პროგრამა, რომელიც გაარკვევს კორექტულია თუ არა ეს მიმდევრობა მათემატიკური თვალსაზრისით. კორექტულობის შემთხვევაში პროგრამამ დაბეჭდოს “YES”, წინააღმდეგ შემთხვევაში – “NO”.

მითითება. თუ “(“-ს შევუსაბამებთ 1-ს, ხოლო “)”-ს – -1-ს, მივიღებთ სიმბოლური მიმდევრობის ანალოგიურ რიცხვით მიმდევრობას, რომლის კორექტულობა დაიყვანება ორი პირობის შესრულებაზე: ა) ყველა რიცხვითი მნიშვნელობის მარცხნიდან მარჯვნივ აჯამვისას ჯამი არც ერთ მომენტში არ უნდა გახდეს უარყოფითი; ბ) ჯამის საბოლოო მნიშვნელობა 0-ის ტოლი უნდა იყოს.

```
READ (a)
k=LENGTH(a)
s=0
FOR i=1 TO k {
        IF A[i]='(' THEN s=s+1 ELSE s=s-1
        IF s<0 THEN BREAK }
IF s=0 THEN WRITE ('YES') ELSE WRITE ('NO')
```

1.36. დაზიანებული კლავიატურა

კომპიუტერის კლავიატურა იმგვარადაა დაზიანებული, რომ ზოგჯერ კლავიშზე ხელის დაჭერისას ერთის ნაცლად რამდენიმე სიმბოლოს ბეჭდავს. ცნობილია, რომ ამ კლავიატურიდან შეტანილ ტექსტში ერთმანეთის მეზობლად ერთნაირი სიმბოლოები არ გვხდება. დაწერეთ პროგრამა, რომელიც შეასწორებს ტექსტს და წაშლის მასში ზედმეტ სიმბოლოებს. ტექსტის სიგრძე არ აღემატება 250 სიმბოლოს.

შემავალი მონაცემი	გამომავალი მონაცემი
kkkkeybbooooarrrdd	keyboard

მითითება.

```
        READ (w)
        k=LENGTH(w);    i=1
L1:     IF i=k THEN GOTO L2
        IF w[i]=w[i+1] THEN  { DELETE (w,i,1); k=k-1 }  ELSE i=i+1
        GOTO L1
L2:     WRITE (w)
```

1.37. დიდი რიცხვები

გამოთვალეთ 100!

მითითება.100! დაახლოებით 150-ნიშნა რიცხვია. ასეთი სიდიდის მთელ რიცხვად მოცემა არცერთ პროგრამულ ენას არ შეუძლია. არსებობენ მონაცემთა ტიპები, რომლებიც აღწერენ მსგავს სიდიდეებს, მაგრამ ვერ უზრუნველყოფენ სიზუსტეს. ასეთ შემთხვევებში იყენებენ ე.წ. დიდი რიცხვების არითმეტიკას. ავიღოთ 200-ელემენტიანი მასივი, რომლის თითოეული ელემენტი განვიხილოთ, როგორც გამოსათვლელი რიცხვის შესაბამისი თანრიგი, ხოლო ოპერაციები ვაწარმოოთ არა მთლიანად რიცხვზე, არამედ ცალკეულ თანრიგებზე. შედეგად მივიღებთ გამოსათვლელ სიდიდეს, რომელიც ჩაწერილი იქნება მასივში ციფრებად (თითო ციფრი – მასივის თითო ელემენტში).

```
a[200]=1
FOR  i=2 TO 100 {
        FOR j=1 TO 200 {
                A[j]=a[j]*i }
        FOR j=200 TO 3 {
                k1=a[j] mod 10; k2=(a[j] mod 100) div 10; k3=a[j] div 100
                a[j]=k1; a[j-1]=a[j-1]+k2; a[j-2]=a[j-2]+k3  }       }
FOR j=1 TO 200 {
        WRITE A[j] }
```

1.38. ორი ლაზიერი

საჭადრაკო დაფაზე მოთავსებულია ორი სხვადასხვა ფერის ლაზიერი. დაწერეთ პროგრამა, რომელიც გაარკვევს შეუძლიათ თუ არა ამ ლაზიერებს ერთმანეთის მოკვლა. დადებითი პასუხის შემთხვევაში პროგრამამ გამოიტანოს "YES", წინააღმდეგ შემთხვევაში – "NO".

შემავალი მონაცემები: *(x1,y1) – თეთრი ლაზიერის და (x2,y2) – შავი ლაზიერის კოორდინატები დაფაზე.*

მითითება. READ (x1,y1,x2,y2)

IF (X1=x2) OR (y1=y2) OR (ABS(x1-x2)=ABS(y2-y1)) THEN WRITE ('YES') ELSE WRITE ('NO')

1.39. ორი მხედარი

საჭადრაკო დაფაზე (8×8) მოთავსებულია ორი სხვადასხვა ფერის მხედარი. შავი მხედარი არ მოძრაობს. დაწერეთ პროგრამა, რომელიც გაარკვევს შეუძლია თუ არა თეთრ მხედარს შავი ფიგურის მოკვლა არაუმეტეს სამ სვლაში. დადებითი პასუხის დროს პროგრამამ გამოიტანოს "YES", წინააღმდეგ შემთხვევაში – "NO".

შემავალი მონაცემები: *(x1,y1) – თეთრი მხედრის და (x2,y2) – შავი მხედრის კოორდინატები დაფაზე.*

მითითება. გავანულოთ (8×8)-ზე მასივი. თეთრი მხედრის საწყისი პოზიცია აღვნიშნოთ -1-ით, ხოლო შავი მხედრის – -2-ით. მოცემული პოზიციიდან მხედარს შეუძლია გადაადგილდეს მაქსიმუმ 8 უჯრაზე (თუკი ის დაფის არცერთ კიდესთან არ დგას ორ უჯრაზე ახლოს). ყველა იმ უჯრაში, სადაც მხედარს შეუძლია გადასვლა ერთი სვლით, ჩავწეროთ ერთიანები. შემდეგ ყველა იმ უჯრიდან, სადაც 1-იანები წერია, მოვახდინოთ გადასვლა ახალ უჯრებში და ჩავწეროთ იქ 2-ები, თუკი უჯრაში 0 ეწერა. შემდეგ იგივე გავიმეოროთ 3-ებისათვის. ყოველ ახალ უჯრაზე გადასვლისას შევამოწმოთ ხომ არ წერია მასში -2.

1.40. იპოვეთ n×n (n<100) მასივის მთავარი დიაგონალების ჯამი.

მითითება. მასივს აქვს ორი მთავარი დიაგონალი. პირველ მათგანში, რომლის ბოლოებსაც წარმოადგენენ ელემენტები (1,1) და (n,n), ორივე ინდექსი თანაბრად იზრდება და ყოველთვის ტოლია, ხოლო მეორე დიაგონალში, რომლის წვეროებსაც წარმოადგენენ (1,n) და (n,1) – მარცხენა ინდექსი იზრდება, ხოლო მარჯვენა მცირდება.

```
sum=0
FOR i=1 TO n {
        sum=sum+a[i, i]+a[i, n+1-i]  }
WRITE [sum]
```

1.41. იპოვეთ n×n (n<100) მასივის მთავარ დიაგონალს ზემოთ მდგარი წევრების ჯამი.

მითითება. მთავარ დიაგონალში იგულისხმება ის დიაგონალი, რომლის ბოლოებსაც წარმოადგენენ ელემენტები (1,1) და (n,n). ამ დიაგონალის ზემოთ მდებარე ნებისმიერი ელემენტისათვის მარცხენა ინდექსი მარჯვენაზე ნაკლებია, ხოლო დიაგონალის ქვემოთ მყოფი ელემენტებისათვის – პირიქით.

```
sum=0
FOR i=1 TO n {
        FOR j=1 TO n {
                IF i<j THEN sum=sum+a[i, j]   }}
WRITE [sum]
```

1.42. ჯვრები და ნოლები

3×3 ზომის დაფაზე ორი მოთამაშე რიგ-რიგობით წერს შესაბამისად ჯვარს ან ნულს. მოგებულია ის მოთამაშე, რომელიც საკუთარ ფიგურას განალაგებს რომელიმე ჰორიზონტალის, ვერტიკალის ან დიაგონალის სამივე უჯრაში. მოცემულია დაფის საბოლოო მდგომარეობა, სადაც "+" აღნიშნავს ჯვარს, "0" – ნულს, ხოლო "*" – ცარიელ უჯრას (თუკი ასეთი არის). დაწერეთ პროგრამა, რომელიც იპოვის გამარჯვებულ მოთამაშეს.

შემავალი მონაცემები: +0*
+00
+*0

გამომავალი მონაცემები: თუ პირველმა მოთამაშემ მოიგო პროგრამამ დაბეჭდოს სიტყვა "FIRST", თუ მეორემ – "SECOND", თუ ვერცერთმა ვერ მოიგო – "NO ONE". საწყისი მონაცემები კორექტულია.

მოცემული მაგალითისთვის: FIRST

მითითება. 3×3 ზომის დაფაზე.თითოეული მოთამაშისათვის მომგებიანი სიტუაციის 8 განლაგება არსებობს: 3 ვერტიკალურად, 3 ჰორიზონტალურად და ორი დიაგონალურად. მცირე ზომის დაფებისათვის შესაძლებელია ყველა ვარიანტის ჩამოწერა (ჩვენს შემთხვევაში 16-ის) და თუ არცერთი არ შესრულდა, მაშინ ყაიმი იქნება. თუმცა დიდი ზომის მასივებისათვის ასეთი მიდგომა გაუმართლებელია. n×n ზომის სიმბოლური მასივი უმჯობესია გადავწეროთ ამავე ზომის რიცხვით მასივში, სადაც '+'-ის ნაცვლად ჩავწეროთ 1, 'O'-ს ნაცვლად -1, ხოლო ''-ის ნაცვლად 0. შემდეგ ცალ-ცალკე ავჯამოთ ყველა ვერტიკალი, ყველა ჰორიზონტალი და ორივე დიაგონალი. თუკი რომელიმე ამ ჯამთაგან n-ის ტოლია – გაუმარჯვია პირველ მოთამაშეს, თუ –n-ის ტოლია – მეორე მოთამაშეს, ხოლო თუკი არცერთი ჯამი არ უდრის n-ს ან –n-ს, თამაში ყაიმით დამთავრებულა.*

1.43. რიცხვები დაფაზე

n×n (3≤n≤100) ზომის დაფაზე, რომელიც საჭადრაკო დაფის ანალოგიურადაა დაყოფილი შავ და თეთრ უჯრედებად, ყოველ უჯრედში ჩაწერილია ორნიშნა რიცხვი. იპოვეთ შავ უჯრედებში ჩაწერილი რიცხვების ჯამი, თუკი დაფის ზედა მარცხენა უჯრედი თეთრია.

შემავალი მონაცემები: პირველ სტრიქონში მოცემულია ერთი მთელი რიცხვი – n, მეორე სტრიქონდან n+1 სტრიქონამდე თითოეულში მოცემულია n ცალი ორნიშნა რიცხვი.

გამომავალი მონაცემები: ერთადერთ სტრიქონში ერთი მთელი რიცხვი – შავ უჯრედებში ჩაწერილი რიცხვთა ჯამი.

შემავალი მონაცემების მაგალითი	მაგალითის მონაცემების	შესაბამისი	გამომავალი
3 12 25 78 19 38 91 37 42 63	177		

მითითება. თითოეული შავი უჯრედის ინდექსთა ჯამი არის კენტი, ხოლო თითოეული თეთრი უჯრედის ინდექსთა ჯამი – ლუწი. მასვიც ამ პირობით უნდა გადავამოწმოთ.

```
sum=0
FOR i=1 TO n {
        FOR j=1 TO n {
                IF (i+j) MOD 2=1  THEN sum=sum+a[i, j]   }}
WRITE (sum)
```

1.44. რიცხვების ჩაწერა სიტყვებით

სანოტარო ბიუროს დოკუმენტებში საჭიროა ყველა რიცხვითი ჩანაწერი შეიცვალოს მისი შესაბამისი სიტყვიერი ანალოგით. მაგ. თუ დოკუმენტში ჩაწერილია რიცხვი 1357, საჭიროა იგი შეიცვალოს სიტყვით “ATAS SAMAS ORMOCDA CHVIDMETI”. დაწერეთ პროგრამა, რომელიც დაეხმარება ბიუროს ამ დავალების შესრულებაში.

შემავალი მონაცემები: ერთადერთ სტრიქონში მოცემულია ერთადერთი რიცხვი N (0<N<10000).

გამომავალი მონაცემები: ერთადერთ სტრიქონში გამოტანილი უნდა იქნას N რიცხვის სიტყვიერი ანალოგი.

მითითება. ამოცანის ამოსახსნელად საჭიროა ყველა საკვანძო სიტყვის (ATASI, CXRAASI, RVAASI, ..., ASI, OTXMOCDAATI, OTXMOCI, ..., ATI, CXRA, RVA, ... ORI, ERTI) დამახსოვრება და რაიმე ცვლადისათვის მინიჭება.

1.45. რიცხვების არაბული და რომაული ჩაწერა

რიცხვების რომაულ ჩანაწერში გამოიყენება სიმბოლოები I, V, X, L, C, D და M, რომლებსაც არაბულ ჩანაწერში შეესაბამებიან 1, 5, 10, 50,100, 500 და 1000. დაწერეთ, პროგრამა, რომელიც კლავიატურიდან შეტანილ რომაულად ჩაწერილ რიცხვს გადაიყვანს არაბულ ჩანაწერში.

შემავალი მონაცემები: ერთადერთ სტრიქონში მოცემულია სიმბოლოთა მიმდევრობა, რომელიც წარმოადგენს 3000-ზე ნაკლები რიცხვის რომაულ ჩანაწერს. შემავალი მონაცემები ყოველთვის კორექტულია.

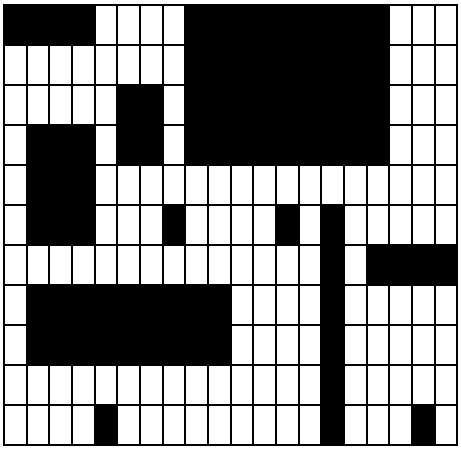
გამომავალი მონაცემები: ერთადერთ სტრიქონში – შემომავალი რიცხვის შესაბამისი არაბული ჩანაწერი.

შემავალი მონაცემების მაგალითი	მაგალითის შესაბამისი გამომავალი მონაცემები
LXXXIX	89
CMLXIV	964

მითითება. რომაული ჩაწერის თავისებურებიდან გამომდინარე, უნდა ვიპოვოთ ყველაზე დიდი რიცხვის შესაბამისი სიმბოლო და განვიხილოთ მის მარცხნივ და მარჯვნივ განლაგებულ სიმბოლოთა ჯგუფები. თავის მხრივ ისინიც ანალოგიურად გავაანალიზოთ და ა.შ.

1.46. მართკუთხედების დათვლა

m×n მართკუთხა ფორმის მინდორი დაყოფილია m*n უჯრედად. ზოგიერთი უჯრედი შეღებეს შავად. ცნობილია, რომ შავი ფერის უჯრედები ჰქმნიან მართკუთხედებს, რომლებიც ერთმანეთს არც კვეთენ და არც ეხებიან. შემავალ მონაცემებში 0 აღნიშნავს თეთრ ფერს, 1 – შავს. იპოვეთ მართკუთხედების რაოდენობა.



1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0
0 0 0 0 0 1 1 0 1 1 1 1 1 1 1 1 1 0 0 0
0 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 0 0 0
0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1
0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 0
0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0

მითითება. ვიმოძრაოთ მასივში მარცხნიდან მარჯვნივ და ზემოდან ქვემოთ. პირველივე შემხვედრი ერთიანი, ბუნებრივია იქნება რომელიღაც მართკუთხედის ზედა მარცხენა წერტილი. დავთვალოთ რამდენი 1-ია ზედიზედ მის მარჯვნივ დ მის ქვემოთ. მიღებული რიცხვების შესაბამისად დატრიალებული ორი ჩადგმული ციკლით გავანულოთ ეს მართკუთხედი, რაიმე მთვლელს დავუმატოთ 1 და გავაგრძელოთ ძებნა არჩეული მიმართულებით.

1.47. სპირალი

დაწერეთ პროგრამა, რომელიც მოცემული n ნატურალური რიცხვისათვის (3≤n≤18), 1-დან n^2-მდე რიცხვებს სპირალურად განალაგებს n×n ზომის ცხრილში.

შემავალი მონაცემები

5

გამომავალი მონაცემები

1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

მითითება. მასივის შესავსებად მოძრაობა ხორციელდება ოთხი სხვადასხვა მიმართულებით, რომლებიც თანმიმდევრულად ენაცვლებიან ერთმანეთს. თითოეული მიმართულებისთვის საჭიროა განვსაზღვროთ ინდექსების ცვლილების წესი და მოძრაობის დასრულების პირობა.

```
read(n);
k=1;  i=1;  j=1;
while k<n*n {
        while (i+j<n+1)  {  a[i,j]=k;  k=k+1;  j=j+1;  }
        while (i<j)        { a[i,j]=k;  k=k+1;  i=i+1;   }
        while (i+j>n+1) {  a[i,j]=k; k=k+1;  j=j-1;  }
        while (i>j+1)    {  a[i,j]=k;  k=k+1; i=i-1;   }        }
 a[i,j]=k
 write (a[i,j])
```

1.48. მანათობელი ციფრები

A და B პუნქტებს შორის ყოველ კილომეტრში აღმართული საგზაო ნიშნულებიდან ხუთის ჯერადებზე (5,10,15,...) მანძილის მაჩვენებელი რიცხვები მითითებულია პლასტმასისაგან საგანგებოდ დამზადებული მანათობელი ციფრების საშუალებით. საგზაო სამსახურმა მიიღო დავალება, რომ ასეთივე ციფრები გააკრას ყველა დანარჩენ ნიშნულზედაც. მანძილის ათვლა იწყება A პუნქტიდან და B პუნქტამდე 4000 კმ-ია. საგზაო სამსახურის ცალკეულ თანამშრომლებს დაევალათ ციფრების გაკვრა გარკვეულ m და n კილომეტრებს შორის(0<m<n<4000).

დაწერეთ პროგრამა, რომელიც გამოთვლის, თუ რა რაოდენობით თითოეული ციფრი უნდა წაიღოს დავალების შესასრულებლად წასულმა თანამშრომელმა.

შემავალი მონაცემები: m და n(მიეწოდება კლავიატურიდან).

მაგ.: 157 181

გამომავალი მონაცემები: ათივე ციფრი გვერდით მიწერილი შესაბამისი რაოდენობით

მაგ.(მოცემული რიცხვებისათვის): 0-0, 1-23, 2-2, 3-2, 4-2, 5-3, 6-10, 7-11, 8-4, 9-3.

მითითება.

```
read(n,m);
for k=n to m  {
    if (k mod 5<>0) then  {
        c=k div 100; a[c]=a[c] + 1;
        c=(k mod 100) div 10; a[c]=a[c] + 1;
        c=k mod 10;  a[c]=a[c] + 1; }   }
for c=0 to 9  { write (c,a[c])     }
```

1.49. კოდის კორექცია

კავშირგაბმულობის რომელიღაც არხით გადაეცემა შეტყობინება, რომელსაც აქვს 0-ებისა და 1-ებისაგან შედგენილი მიმდევრობის სახე. არხში არსებული ხარვეზების გამო ზოგიერთი სიგნალი შესაძლოა შეცდომით იქნას მიღებული: 0 აღქმული იქნას 1-ად და პირიქით. სიგნალების გადაცემის საიმედოობის გასაზრდელად გადაწყდა, რომ თითოეული სიგნალი სამჯერ გაიგზავნოს. გადამცემი 1-ის ნაცვლად აგზავნის 111-ს, ხოლო 0-ის ნაცვლად 000-ს.

დაწერეთ პროგრამა, რომელიც აღადგენს საწყის შეტყობინებას. რადგან გადაცემისას მოსალოდნელი იყო ხარვეზები, ციფრთა ყოველი სამეულის ნაცვლად პროგრამამ უნდა გამოიტანოს ის ციფრი, რომელიც ამ სამეულში ორჯერ მაინც გვხვდება.

შემავალი მონაცემების ფორმატი: შემავალ ფაილში მოცემულია ერთადერთი სტრიქონი, რომელიც შედგება “0”-ებისა და “1”-ებისაგან. სტრიქონის სიგრძე 3-ის ჯერადი რიცხვია, მეტია 2-ზე და ნაკლებია 760-ზე.

გამომავალი მონაცემების ფორმატი: ერთადერთ სტრიქონში უნდა გამოიტანოთ გაშიფრული შეტყობინება.

შემავალი მონაცემების ნიმუში	გამომავალი მონაცემების ნიმუში
110111010001	1100

მითითება. ამოცანა იოლად იხსნება ორი ჩადგმული ციკლით. გარე ციკლში ხდება საწყისი სტრიქონის კითხვა სამ-სამ სიმბოლოდ, ხოლო შიგა ციკლში მიღებული სამეულის ყოველი სიმბოლოს გარდაქმნა რიცხვით ცვლადად და მათი ჯამის პოვნა. თუკი ჯამი მეტია ან ტოლი 2-ის, გამომავალ ფაილში გვექნება 1, ხოლო თუ 2-ზე ნაკლებია – 0.

1.50. ლურსმნები

(რუსეთის რაიონული ოლიმპიადა, 1996-97 წ.წ.)

ხის გრძელ ფიცარზე ერთ მწკრივად ჩააჭედეს ლურსმნები, რომლებიც შემდეგ წყვილ-წყვილად შეაერთეს თოკებით იმგვარად, რომ სრულდება შემდეგი პირობები:

1) ყოველ ლურსმანზე მიბმულია ერთი მაინც თოკი;

2) თოკების სიგრძეთა ჯამი მინიმალურია.

დაწერეთ პროგრამა რომელიც გაარკვევს, თუ რომელი ლურსმნებია შეერთებული.

შემავალი მონაცემების ფორმატი: პირველ სტრიქონში მოცემულია ერთი ნატურალური n (n<10000) რიცხვი, რომელიც აღნიშნავს ლურსმნების რაოდენობას. მეორე სტრიქონში მოცემულია ზრდადობით დალაგებული n ცალი მთელი დადებითი რიცხვი, რომლებიც წარმოადგენენ ლურსმნების კოორდინატებს. თითოეული კოორდინატის მნიშვნელობა არ აღემატება 100000-ს.

გამომავალი მონაცემების ფორმატი: პირველ სტრიქონში პროგრამამ უნდა გამოიტანოს თოკის მინიმალური ჯამი, ხოლო მეორე სტრიქონში ზრდადობით დალაგებული თითოეულ წყვილში შემავალი ლურსმნების ნომრები.

შემავალი მონაცემების ნიმუში	გამომავალი მონაცემების ნიმუში
5	3
11 12 13 16 17	1 2 2 3 4 5

მითითება. ცხადია, რომ პირველი და ბოლო ლურსმნებისათვის უახლოესი იქნება შესაბამისად მეორე და ბოლოსწინა ლურსმნები, ხოლო ამ ოთხი წევრის გარდა ყველა დანარჩენი უნდა მიეურთდეს მარცხენა და მარჯვენა მეზობლებიდან უახლოესს.

1.51. თამაში

(მერვე საერთაშორისო ოლიმპიადა, უნგრეთი, 1996 წელი)

განვიხილოთ შემდეგი თამაში ორი მოთამაშისათვის: სათამაშო დაფაზე დაწერილია მთელი დადებითი რიცხვების მიმდევრობა. მოთამაშეები სვლებს აკეთებენ რიგ-რიგობით. სვლა მდგომარეობს შემდეგში: მოთამაშე ირჩევს მიმდევრობის რომელიმე კიდურა წევრს (მარცხენას ან მარჯვენას). არჩეული რიცხვი დაფიდან იშლება. თამაში დამთავრდება მაშინ, როცა დაფაზე ყველა რიცხვი წაიშლება. პირველი მოთამაშე იმ შემთხვევაში გაიმარჯვებს, თუ მეორე მოთამაშის მიერ არჩეული ჯამი მეტი არ იქნება მის მიერ არჩეული რიცხვების ჯამზე. გაითვალისწინეთ, რომ მეორე მოთამაშე საუკეთესო ალგორითმით თამაშობს.

თამაშს ყოველთვის პირველი მოთამაშე იწყებს.

ცნობილია, რომ თუ დაფაზე თავიდან დაწერილი მიმდევრობის წევრთა რაოდენობა ლუწია, მაშინ პირველ მოთამაშეს მოგების შანსი აქვს. დაწერეთ პროგრამა, რომელშიც რეალიზებული იქნება პირველი მოთამაშის მომგებიანი სტრატეგია. საწყისი მიმდევრობის წევრთა რაოდენობა N ყოველთვის ლუწია და $2 \le N \le 100$.

მითითება. საწყისი მიმდევრობა პირობითად შეიძლება გავყოთ ორ ნაწილად: კენტ ადგილებზე მდგომი წევრები და ლუწ ადგილებზე მდგომი წევრები. მათი ცალ-ცალკე აჯამვა სირთულეს არ წარმოადგენს. პირველი მოთამაშე ირჩევს ამ ორი ჯამიდან უდიდესს (თუ ჯამები ტოლია, მნიშვნელობა არა აქვს რომელს ამოირჩევს) და სვლებს აკეთებს იმგვარად, რომ მხოლოდ შესაბამისი ქვემიმდევრობის წევრებს წაშლის. თუ ლუწინდექსიანი ელემენტების ჯამია მეტი, მაშინ პირველად წაშლის ბოლო ელემენტს, ხოლო თუ კენტინდექსიანი ელემენტების ჯამია მეტი – პირველად წაშლის პირველ ელემენტს. შემდეგ სვლებზე კი ორივე შემთხვევაში წაშლის ელემენტს იმ მხრიდან, საიდანაც წინა სვლაზე წაშალა რიცხვი მეორე მოთამაშემ.

1.52. ორთოგრაფია

მოსწავლეთა ჯგუფს ინგლისურ ენაზე უნდა დაეწერა მზის სისტემაში შემავალი პლანეტების სახელები: "Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Neptune", "Pluto", "Uranos".

მოსწავლეებმა სიტყვების ჩაწერისას დაუშვეს სამი ტიპის შეცდომა: 1) ჩაწერეს სიტყვაში ზედმეტი სიმბოლო; 2) დააკლეს სიტყვას რომელიმე სიმბოლო; 3) ერთი სიმბოლოს ნაცვლად ჩაწერეს სიტყვაში სხვა სიმბოლო. არცერთ სიტყვაში ერთ შეცდომაზე მეტი არ ყოფილა დაშვებული.

შეადგინეთ პროგრამა, რომელიც გააანალიზებს კლავიატურიდან შეტანილ სიტყვას. თუ სიტყვა სწორადაა შეტანილი, პროგრამამ უნდა დაბეჭდოს სიტყვა "TRUE". შეცდომის შემთხვევაში პროგრამამ უნდა დაბეჭდოს შეცდომის ტიპი, ე.ი. ციფრები "1", "2" ან "3". ამასთან, თუ დაშვებულია "1" ტიპის შეცდომა – პროგრამამ დაბეჭდოს ზედმეტი სიმბოლო, "2" ტიპის შეცდომისას დაიბეჭდოს დაკლებული სიმბოლო, ხოლო "3" ტიპის შეცდომისას დაიბეჭდოს მცდარი და მის ნაცვლად ჩასაწერი სიმბოლოები. სიმბოლოთა რეგისტრს ნუ გაითვალისწინებთ.

მაგალითი:

შეტანილი სიტყვა	პროგრამის პასუხი
Mars	TRUE
Marss	1 – s
Jupter	2 – i
Jypiter	3 – y u

მითითება. თავდაპირველად უნდა განვსაზღვროთ მოცემული ცხრა სიტყვიდან რომელთან გვაქვს საქმე. ამისათვის საკმარისია შევამოწმოთ პირველი ორი ან ბოლო ორი სიმბოლოს დამთხვევა, რომლებიც ყველა მოცემულ სიტყვას განსხვავებული აქვს. თუკი ერთერთი მათგანი დაემთხვა – სიტყვა გარკვეულია. ამის შემდეგ შეცდომის ტიპი განისაზღვრება სიტყვათა სიგრძეების შედარებით, ხოლო გამოსატანი სიმბოლოების გასარკვევად მოხდება ორი სიტყვის შესაბამისი სიმბოლოების შედარება.

1.53. ახარისხება

მოცემულია ორი დადებითი a და b რიცხვი. დაწერეთ პროგრამა, რომელიც გაარკვევს თუ რომელი ციფრით მთავრდება a^b.

შემავალი მონაცემების ფორმატი: ერთადერთ სტრიქონში – ჰარით გაყოფილი a და b რიცხვები (1≤a,b≤10000).

გამომავალი მონაცემების ფორმატი: ერთადერთ სტრიქონში გამოიტანეთ ციფრი, რომლითაც მთავრდება a^b.

შემავალი მონაცემების მაგალითი:	გამომავალი მონაცემი ნაჩვენები მაგალითისათვის
3 4	1

მითითება: ეს ამოცანა გარკვეულწილად წააგავს 29-ე ამოცანას, თუმცა მასზე გაცილებით იოლია და მასივის გარეშეც ამოიხსნება. ჩვენ გვჭირდება a რიცხვის მხოლოდ უკანასკნელი ციფრი, რომელსაც საკუთარ თავზე გადავამრავლებთ b-ჯერ, თუმცა თუკი ნამრავლში ორნიშნა რიცხვი მივიღეთ, დავიტოვებთ მხოლოდ ერთეულების თანრიგში მყოფ ციფრს.

```
READ (a,b)
k= a MOD 10; p=k;
    for c = 2 to b { p=p*k;  p=p MOD 10 }
    WRITE (p)
```

1.54. ტექსტის შესწორება

მოცემულია წერტილით დამთავრებული ტექსტი. ტექსტის სიმბოლოთა შორის განსაკუთრებულ როლს თამაშობს "#" სიმბოლო, რომლის გამოჩენაც ტექსტში ნიშნავს წინა სიმბოლოს წაშლას. შესაბამისად, "#" სიმბოლოს ზედიზედ k-ჯერ გამოყენება ნიშნავს ტექსტში წინა k სიმბოლოს წაშლას, თუკი მიმდინარე სტრიქონში ისინი მოიძებნებიან.

დაწერეთ პროგრამა, რომელიც გარდაქმნის ტექსტს "#" სიმბოლოს აღნიშნული თვისების შესაბამისად.

შენიშვნები:

1) თუკი "#" სიმბოლოს წინ მიმდინარე სტრიქონში სიმბოლო აღარ დარჩა – "#" სიმბოლო იგნორირდება.

2) გამომავალ ფაილში "#" სიმბოლო არცერთ შემთხვევაში არ გამოიტანოთ.

3) თუკი გარდაქმნის შედეგად მიმდინარე სტრიქონში ყველა სიმბოლო წაიშალა, გამომავალ ფაილში ამ სტრიქონის ნაცვლად უნდა გამოიტანოთ ცარიელი სტრიქონი.

შემავალი მონაცემების ფორმატი: შემავალ ფაილში მოცემულია არანაკლებ ერთისა და არაუმეტეს 100 სტრიქონზე მეტი ტექსტი. ტექსტის დასასრულს აღნიშნავს წერტილი (“.” სიმბოლო). თითოეულ სტრიქონში ტექსტის სიგრძე არ აღემატება 200 სიმბოლოს.

გამომავალი მონაცემების ფორმატი: შემავალი ფაილის თითოეული სტრიქონისათვის შესაბამის გამომავალ ფაილში უნდა გამოიტანოთ მისი გარდაქმნილი ვარიანტი. ტექსტის დამამთავრებელი წერტილი გამომავალ ფაილში არ ჩაწეროთ.

შემავალი მონაცემების მაგალითი:	გამომავალი მონაცემი ნაჩვენები მაგალითისათვის
Hello ww#orld! # a##abc# the##he end.	Hello world! ab the end

მითითება. ამოცანის ამოხსნისათვის გამოვიყენოთ 200-ელემენტიანი სიმბოლური მასივი. ყოველი სტრიქონი წავიკითხოთ თითო-თითო სიმბოლოდ, დავუმატოთ წაკითხული სიმბოლო ჩვენს მასივს და გავზარდოთ მთვლელი. თუკი შეგვხდება "#" სიმბოლო, მთვლელი შევამციროთ 1-ით (წინასწარ შევამოწმებთ, არის თუ არა ის 1-ზე მეტი). როცა წასაკითხი სტრიქონი ამოიწურება – გამოვიტანოთ ჩვენი მასივის ელემენტები 1-დან მთვლელამდე და გადავიდეთ შემდეგ სტრიქონზე. წერტილის წაკითხვის შემთხვევაში, პროგრამა დავასრულოთ.

1.55. კვადრატები

100×100 ზომის სიბრტყეზე მოცემულია N ცალი (1<N<100) კვადრატი, რომელთა გვერდები კოორდინატთა ღერძების პარალელურია. განსაზღვრეთ იმ არის ფართობი, რომელიც ამ კვადრატებს ერთად უკავიათ (წერტილი ეკუთვნის ამ არეს, თუკი ის მდებარეობს ერთი კვადრატის შიგნით მაინც). კვადრატთა წვეროები მდებარეობენ მთელი მნიშვნელობის მქონე კოორდინატებში.

შემავალი მონაცემების ფორმატი:პირველ სტრიქონში მოცემულია კვადრატების რაოდენობა N. მომდევნო N სტრიქონიდან თითოეულში მოცემულია ჰარით გაყოფილი სამი რიცხვი a, b და c – კვადრატის მახასიათებლები, სადაც a და b კვადრატის ზედა მარცხენა წვეროს კოორდინატებია, ხოლო c კვადრატის გვერდის სიგრძე. 1≤a+c,b+c≤100).

გამომავალი მონაცემების ფორმატი: ერთადერთ სტრიქონში – კვადრატების მიერ დაფარული არის ფართობი.

შემავალი მონაცემების მაგალითი:	გამომავალი მონაცემი ნაჩვენები მაგალითისათვის
3 4 6 10 7 8 3 12 9 4	108

მითითება: 39-ე ამოცანის მსგავსად ცარიელი არე წარმოვიდგინოთ 0-ებად, ხოლო კვადრატების მიერ დაფარული არე 1-ებით შევავსოთ. ყველა კვადრატის აგების შემდეგ დავთვალოთ 1-იანების რაოდენობა.

1.56. ივანოვები და პეტროვები

(რუსეთის სასკოლო ოლიმპიადის დასკვნითი ტური ინფორმატიკაში, 1990წ.)

ქალაქის ერთ-ერთ ქუჩაზე ცხოვრობენ მხოლოდ ივანოვები და პეტროვები. დაწერეთ პროგრამა, რომელიც გამოითვლის გაცვლათა იმ მინიმალურ რაოდენობას, რომელიც საჭიროა იმისათვის, რომ ქუჩის ერთ-ერთი ბოლოდან ცხოვრობდნენ მხოლოდ ივანოვები, ხოლო მეორიდან – პეტროვები.

შემავალი მონაცემების ფორმატი: პირველ სტრიქონში n – სახლების რაოდენობა ქუჩაზე (n<10000), მეორე სტრიქონში n წევრისაგან შემდგარი ჰარებით გაყოფილი 1-ებისა და 2-ებისაგან შემდგარი მიმდევრობა, სადაც 1 აღნიშნავს ივანოვებს, ხოლო 2 – პეტროვებს.

გამომავალი მონაცემების ფორმატი: გაცვლათა მინიმალური რაოდენობა.

შემავალი მონაცემების მაგალითი:	გამომავალი მონაცემი ნაჩვენები მაგალითისათვის
24 1 1 2 2 2 2 1 2 1 2 2 1 1 2 2 2 1 2 2 1 2 2 1 2	5

(გაცვლების ერთ-ერთი ვარიანტი სახლთა ნომრების მიხედვით: 3↔23,4↔20,5↔17,6↔13,8↔12)

მითითება: ამოცანის ამოსახსნელად საჭიროა დავთვალოთ რომელიმე ციფრის (ვთქვათ, 1-იანის) საერთო რაოდენობა მთელს მიმდევრობაში. 1-იანების საერთო რაოდენობაა – 9. მიმდევრობის პირველ ცხრა და ბოლო ცხრა წევრებში დავითვალოთ 2-ების რაოდენობა. შესაბამისად მივიღებთ 5-ს და 6-ს. ამ ორ რიცხვში უმცირესი წარმოადგენს ამოცანის პასუხს.

თუ 1-იანებს გადავალაგებთ მიმდევრობის ბოლოში, დაგვჭირდება 6 გაცვლა, ხოლო თუ გადავალაგებთ მიმდევრობის თავში – 5 გაცვლა.

1.57. სატვირთო მატარებელი

(რუსეთის სასკოლო ოლიმპიადის დასკვნითი ტური ინფორმატიკაში, 1991წ.)

სატვირთო მატარებელი შედგება ერთი ან ორი ლოკომოტივისაგან შემადგენლობის თავში, ერთი ან მეტი სატვირთო ვაგონისაგან და დაცვის ვაგონისაგან შემადგენლობის ბოლოში. დაწერეთ პროგრამა, რომელიც შემავალი მონაცემების მიხედვით გამოიცნობს, არის თუ არა ეს მატარებელი სატვირთო.

შემავალი მონაცემების ფორმატი: 1-ების, 2-ების და 3-ებისაგან შემდგარი სიმბოლოთა ჯაჭვი, სადაც 1 აღნიშნავს ლოკომოტივს, 2 – სატვირთო ვაგონს და 3 – დაცვის ვაგონს.

გამომავალი მონაცემების ფორმატი: YES – თუ სიბოლოთა მიმდევრობა აღწერს სატვირთო მატარებელს, წინააღმდეგ შემთხვევაში – NO.

შემავალი მონაცემების მაგალითი:	გამომავალი მონაცემი ნაჩვენები მაგალითისათვის
1122222222222223 12222222222221222222223	YES NO

1.58. ახალწვეულთა მწკრივი

(რუსეთის სასკოლო ოლიმპიადის დასკვნითი ტური ინფორმატიკაში, 1991წ.)

ახალწვეულთა მწკრივს ზემდეგმა უბრძანა: “მარჯვნისაკენ!”, რის შემდეგაც თითოეული ახალწვეული შებრუნდა 900-ით, მაგრამ ზოგი – მარჯვნივ და ზოგი – მარცხნივ. ყოველი ახალწვეული, რომელიც დაინახავს მეზობლის სახეს, მაშინვე ტრიალდება 1800-ით. დაწერეთ პროგრამა, რომელიც გაარკვევს მოცემული საწყისი მდგომარეობისათვის (ზემდეგის ბრძანების შემდეგ) შეწყვეტენ თუ არა ბრუნვას ახალწვეულები და თუ შეწყვეტენ – რამდენი ბრუნის შემდეგ.

შემავალი მონაცემების ფორმატი: პირველ სტრიქონში n – ახალწვეულთა რაოდენობა (n<10000), მეორე სტრიქონში n წევრისაგან შემდგარი ჰარებით გაყოფილი 1-ებისა და 2-ებისაგან შემდგარი მიმდევრობა, სადაც 1 აღნიშნავს ზემდეგის ბრძანების შემდეგ მარჯვნივ მიბრუნებულებს, ხოლო 2 – მარცხნივ მიბრუნებულებს.

გამომავალი მონაცემების ფორმატი:ერთადერთი რიცხვი, რომელიც აღნიშნავს რამდენი ბრუნის შეწყვეტენ ტრიალს ახალწვეულები, ან თუკი ტრიალს ვერ შეწყვეტენ – “NO” .

შემავალი მონაცემების მაგალითი:	გამომავალი მონაცემი ნაჩვენები მაგალითისათვის
11 1 2 1 2 2 1 1 2 1 2 1	5

ქვემოთ მოცემულია ახალწვეულთა მდგომარეობა თითოეული ბრუნის შემდეგ:

1 ბრუნი – 2 1 2 1 2 1 2 1 2 1 1

2 ბრუნი – 2 2 1 2 1 2 1 2 1 1 1

3 ბრუნი – 2 2 2 1 2 1 2 1 1 1 1

4 ბრუნი – 2 2 2 2 1 2 1 1 1 1 1

5 ბრუნი – 2 2 2 2 2 1 1 1 1 1 1

მითითება. ბრუნვის პროცესი, ცხადია, ყოველთვის შეწყდება, რადგან როგორც კი განაპირა ახალწვეულები ზურგით დადგებიან დანარჩენი რიგისადმი, ისინი ბრუნვას წყვეტენ. შემდეგში იგივე წესი გავრცელდება მათ მეზობლებზე და ა.შ. ბრუნვათა რაოდენობის გასარკვევად საჭიროა ციკლი პირობით, რომელიც გაარკვევს მორიგ ბიჯზე მოხდა რაიმე ცვლილება რიგში თუ არა.

1.59. ვირუსები

(სტუდენტთა და მოსწავლეთა გუნდური პირველობა, ყაზანი, 2000 წ.)

n×n (n≤500) ზომის არეზე მოთავსებულია m(1≤m≤10) ვირუსი. ყოველ სვლაზე ვირუსი ასნებოვნებს მის 4 მეზობელ უჯრედს. დაწერეთ პროგრამა, რომელიც გაარკვევს სვლების იმ უმცირეს რაოდენობას, რომელიც საჭიროა მოცემული არის მთლიანად დასნებოვნებისთვის

შემავალი მონაცემები: პირველ სტრიქონში მოცემულია ორი მთელი რიცხვი – n და m. მეორიდან m+1 სტრიქონამდე მოცემულია ორი მთელი რიცხვი – ვირუსის კოორდინატები.

გამომავალი მონაცემები: ერთადერთ სტრიქონში უნდა გამოიტანოთ ერთი მთელი რიცხვი – სვლების უმცირესი რაოდენობა, რომლითაც შეიძლება არე მთლიანად დაიფაროს.

მითითება. თუკი შევეცდებით, რომ უბრალოდ აღვწეროთ პროცესი და გამოვითვალოთ ყოველ ახალ სვლაზე ვირუსების მიერ დასნებოვანებული მეზობელ უჯრედები, მოგვიწევს 500×500 ზომის ორი მასივით მუშაობა (ერთი მასივით პროცესის აღწერა ალგორითმს გაართულებს) და უარეს შემთხვევაში შეიძლება 500-მდე სვლის გამოთვლამ მოგვიწიოს (თუკი ვირუსები არის ერთ-ერთი კიდისაკენ იქნებიან განლაგებული). ასეთი მიდგომა არაეკონომიურია როგორც მეხსიერების, ასევე დროის თვალსაზრისითაც. თუ წარმოვიდგენთ, რომ გვაქვს მხოლოდ ერთი ვირუსი (i,j) კოორდინატებით, ცხადია, რომ ნებისმიერი (x,y) კოორდინატის მქონე უჯრედი დასნებოვნდება ABS(x-i)+ABS(y-j) სვლაში. მაგალითად 7×7 ზომის არისათვის გვაქვს:

6543456
5432345
4321234
321*123
4321234
5432345
6543456

სადაც ვარსკვლავით აღნიშნულია ვირუსი, ხოლო რიცხვები მიუთითებენ, თუ მერამდენე სვლაზე დასნებოვნდა უჯრედი. რამდენიმე ვირუსის შემთხვევაში კონკრეტული უჯრედისათვის სათითაოდ გამოვთვლით ზემოთ მოყვანილ ფორმულას და უჯრედში ჩავწერთ უმცირეს შედეგს მათ შორის. ასეთი წესით შევსებული მასივის უდიდესი ელემენტი წარმოადგენს ჩვენი ამოცანის ამონახსნს.

1.60. დომინო

(ჩიგირინსკების ოლიმპიადა, 2003-04 წლები)

დომინოს ჩვეულებრივი კომპლექტი შეიცავს 28 ქვას. ცხადია, რომ თუკი ქვებზე ქულების რაოდენობა შეიცვლებოდა არა 0-დან 6-მდე, არამედ 0-დან K-მდე, ქვების რაოდენობაც სხვაგვარი იქნებოდა. დაწერეთ პროგრამა, რომელიც მოცემული K-სათვის გაარკვევს კომპლექტში ქვების რაოდენობას.

შემავალი მონაცემების ფორმატი (ფაილი DOMINO.DAT):

ერთადერთ სტრიქონში ნატურალური K (K≤100000000) რიცხვის სახით მითითებულია ქვებზე ქულათა რაოდენობის მარჯვენა საზღვარი.

გამომავალი მონაცემების ფორმატი (ფაილი DOMINO.SOL):

გამომავალი ფაილი უნდა შეიცავდეს ერთადერთ რიცხვს – კომპლექტში ქვების რაოდენობას.

შემავალი მონაცემების მაგალითი :	გამომავალი მონაცემი ნაჩვენები მაგალითისათვის:
11	78

მითითება. პასუხი შეგვიძლია პირდაპირ ფორმულით გამოვითვალოთ: (n+1)(n+2)/2, მაგრამ K-სთვის მითითებული სასაზღვრო მნიშვნელობების გამო საჭირო იქნება დიდი რიცხვების არითმეტიკის გამოყენება (იხ. ამოცანა 1.35).*

1.61. გვირაბი

(ჩიგირინსკების ოლიმპიადა, 2003-04 წლები)

ორი მეგობარი – ივანე და აბრამი სამუშაოდ საზღვარგარეთ გაემგზავრნენ. ივანე მოეწყო საფრანგეთის საგზაო პოლიციაში, ხოლო აბრამი ინგლისის ანალოგიურ სამსახურში. მათ მუშაობა მოუწიათ ლა-მანშის გვირაბის შესასვლელ პოსტებთან, ოღონდ ივანე საფრანგეთის მხრიდან იდგა გვირაბთან, ხოლო აბრამი – ინგლისის მხრიდან. ისინი იწერდნენ მათი სამუშაო ცვლის განმავლობაში გვირაბით მოსარგებლე მანქანების ნომრებს, ხოლო აღრიცხულ ინფორმაციას დღის ბოლოს აბარებდნენ პოლიციის სერჟანტს.

ამ ინფორმაციის საფუძველზე პოლიციას შეუძლია უშეცდომოდ განსაზღვროს, თუ რომელმა მანქანებმა გადაასწრეს სხვებს გვირაბის შიგნით და ამით დაარღვიეს მოძრაობის წესები, რადგან გვირაბში გადასწრება აკრძალულია.

დაწერეთ პროგრამა, რომელიც განსაზღვრავს იმ მანქანების რაოდენობას, რომელთა მძღოლებმაც დაარღვიეს საგზაო მოძრაობის წესები გვრაბში ყოფნისას.

შემავალი მონაცემების ფორმატი (ფაილი TUNNEL.IN):

ფაილი შეიცავს 2N+1 სტრიქონს, რომელთაგან პირველში წერია რიცხვი N ($1<N\leq1000$) – გვირაბში გამავალი მანქანების რაოდენობა. შემდეგ N სტრიქონში მოცემულია მანქანების ნომრების სია იმ მიმდევრობით, რომლითაც ისინი შევიდნენ გვირაბში, ხოლო მომდევნო N სტრიქონში მოცემულია მანქანების ნომრების სია იმ თანმიმდევრობით, რომლითაც ისინი გამოვიდნენ გვირაბიდან.

თითოეული მანქანის ნომერი შეიცავს არაუმეტეს 8 სიმბოლოსი და წარმოადგენს ლათინური ანბანის მთავრული სიმბოლოებისა (A-Z) და ათობით სისტემაში შემავალი ციფრების (0-9) კომბინაციას. ნომრების არ შეიცავენ ჰარებს.

გამომავალი მონაცემების ფორმატი (ფაილი TUNNEL.OUT):

გამომავალი ფაილი უნდა შეიცავდეს ერთადერთ რიცხვს – იმ მანქანების რაოდენობას, რომელთა მძღოლები შეიძლება მხილებულნი იქნან საგზაო მოძრაობის წესების დარღვევაში.

შემავალი მონაცემების მაგალითი :	გამომავალი მონაცემი ნაჩვენები მაგალითისათვის:
4 ZG431SN ZG5080K ST123D ZG206A ZG206A ZG431SN ZG5080K ST123D	1

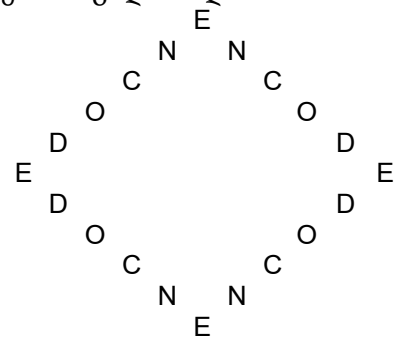
მითითება. გვირაბში შემავალი მანქანების რიგი გადავნომროთ ზრდადობით (1,2,3,4,...) და დავადგინოთ გვირაბიდან გამომავალი მანქანების რიგი ამ ნომრების შესაბამისად. პირობაში მოყვანილი მაგალითისათვის გვექნება: 4,1,2,3. ამის შემდეგ მიღებულ მასივში დავითვალოთ ისეთი რიცხვების რაოდენობა, რომლებიც აღემატებიან მათგან მარჯვნივ განლაგებული რიცხვებიდან თუნდაც ერთ რიცხვს. მიღებული რაოდენობა იქნება ამოცანის პასუხი.

შევნიშნოთ, რომ თითოეული მანქანისათვის გვირაბიდან გამომავალ მანქანათა რიგში მისი ნომრის მოსაძებნად საჭიროა არაუმეტეს N ოპერაციისა. ყველა მანქანისათვის ოპერაციათა რაოდენობა არ იქნება N^2-ზე მეტი. მაქსიმუმ ამდენივე ოპერაციაა საჭირო იმის გასარკვევად, დგას თუ არა მოცემული რიცხვის მარჯვნივ მასზე ნაკლები რიცხვი. მაშასადამე ყველაზე უარეს შემთხვევაშიც კი ოპერაციების რაოდენობა 2 მილიონს არ გადააჭარბებს.

1.62. ალმასისმაგვარი სიტყვები

(USACO, 2001 წლის ზამთარი, ”ნარინჯისფერი” დივიზიონი)

სიტყვას, რომელიც იწყება და მთავრდება ერთი და იგივე სიმბოლოთი უწოდებენ ალმასისმაგვარს. მაგალითად ალმასისმაგვარია სიტყვა ”ENCODE”. ასეთი სიტყვებისათვის გამოიყენება ალმასისმაგვარი ჩაწერა. მაგალითად:



დაწერეთ პროგრამა, რომელიც მოცემულ ალმასისმაგვარ სიტყვას ალმასისმაგვარად ჩაწერს.

შემავალი მონაცემები: მთავრული სიმბოლოებით ჩაწერილი სიტყვა 3-დან 13 სიმბოლომდე, რომლის პირველი და ბოლო სიმბოლოები ემთხვევა.

გამომავალი მონაცემები: მოცემული სიტყვის ალმასისმაგვარი ჩანაწერი (თუ სიტყვა შედგება N სიმბოლოსაგან, გამომავალი მონაცემი იქნება 2N-1 სტრიქონი).

მითითება. ამოცანის ამოხსნისას მთავარია ზუსტად განისაზღვროს ჰარების რაოდენობა ყოველ სტრიქონში, ამასთან ჰარები პირობითად ორ ნაწილად შეგვიძლია დავყოთ – სიმბოლოებამდე ჩასაწერი ჰარები და სიმბოლოთა შორის ჩასაწერი ჰარები. პირველი ტიპის ჰარების რაოდენობა იცვლება (N-1)-დან 0-მდე და შემდეგ 0-დან (N-1)-მდე. მეორე ტიპის ჰარები პირველ და ბოლო სტრიქონებში საერთოდ არ გვხვდება, ხოლო მეორიდან მე-N-ე სტრიქონამდე იზრდება 1-დან (2N-3)-მდე ბიჯით 2, ხოლო შემდეგ იმავე ბიჯით მცირდება 1-მდე.

1.63. პარკსიდის სამკუთხედი

(USACO, 2001 წელი, ნოემბერი, “ნარინჯისფერი” დივიზიონი)

პარკსიდის სამკუთხედი გენერირდება ორი რიცხვისაგან: სტრიქონების რაოდენობა R ($1 \le R \le 39$) და საწყისი რიცხვი S($1 \le S \le 9$). საწყისი რიცხვი განსაზღვრავს სამკუთხედის ზედა მარცხენა კუთხეს. ქვემოთ მოყვანილი მაგალითის მიხედვით განსაზღვრეთ პარკსიდის სამკუთხედის შედგენის წესი და დაწერეთ პროგრამა, რომელიც წაიკითხავს R და S რიცხვებს შემავალი მონაცემების ფაილიდან და შეადგენს პარკსიდის სამკუთხედს ამ რიცხვების მნიშვნელობათა მიხედვით.

შემავალი მონაცემები: ერთადერთ სტრიქონში მოცემულია ორი მთელი რიცხვი R და S.

შემავალი მონაცემების მაგალითი (ფაილი parktri.in):

6 1

გამომავალი მონაცემები:

პარკსიდის სამკუთხედი, სტრიქონის ბოლოში ჰარის გარეშე და მაგალითში ნაჩვენები ინტერვალის დაცვით.

გამომავალი მონაცემების მაგალითი (ფაილი parktri.out):

1 2 4 7 2 7

3 5 8 3 8

6 9 4 9

1 5 1

6 2

3

მითითება.

```
READLN (m,n);
a0=n; a1=m; hor=1; ver=1; j=1;
WHILE n>=0 do {
    WRITE(a1);
    i=1; hor=1; a2=a1;
    WHILE i<=n do {
        a1=(a1+hor) MOD 9;
        WRITE (' ',a1);
        hor=hor+1; i=i+1; }
    WRITELN;
    IF n=a0 THEN ver=ver+1;
    a1=(a2+ver) MOD 9;
    ver=ver+1;
    n=n-1  }
```

1.64. დროის გადამყვანი

(USACO, 2002 წელი, ნოემბერი , "ნარინჯისფერი" დივიზიონი)

ფერმერმა ჯონიმ (ფჯ) უნდა იმოგზაუროს აშშ-ს ტერიტორიაზე სამი თვის განმავლობაში. მას სჭირდება თქვენი დახმარება განსხვავებულ სასაათო სარტყლებში გასარკვევად.

ფერმერი ჯონი იმოგზაურებს მხოლოდ კონტინენტურ აშშ-ში, სადაც მოქმედებს ოთხი სასაათო სარტყელი: Eastern (აღმოსავლეთის), Central (ცენტრალური), Mountain (მთიანი) and Pacific (წყნარი ოკეანის). ფჯ კარგად ვერ ერკვევა სასაათო სარტყლებში, მაგრამ მას აქვს ექვივალენტობის ცხრილი:

Pacific	5:00 AM
Mountain	6:00 AM
Central	7:00 AM
Eastern	8:00 AM

საიდანაც ჩანს, რომ როცა წყნარი ოკეანის სარტყელში დღის 5 საათია, მთიან სარტყელში ამ დროს დღის 6, ხოლოს ცენტრალურში დღის 7 საათია და ა.შ. ყურადღება მიაქციეთ 11:59 AM-დან ერთი წუთის შემდეგ დგება 12:00 PM, ხოლო 11:59 PM-დან ერთი წუთის შემდეგ 12:00 AM.

შემავალი მონაცემების ფორმატი(ფაილი tconvert.in): 1 სტრიქონი: დრო, რომელიც გადაყვანილი უნდა იქნას ზემოთ მოყვანილი წესით; 2 სტრიქონი: სასაათო სარტყლის დასახელება, რომლითაც ნაჩვენებია 1 სტრიქონში ნახსენები დრო; 3 სტრიქონი: სასაათო სარტყლის დასახელება, რომელშიც უნდა იქნას გადაყვანილი დრო.

გამომავალი მონაცემების ფორმატი(ფაილი tconvert.out): ერთადერთ სტრიქონში ნაჩვენები უნდა იყოს გადაყვანილი დრო სს:წწ ფორმატით, შესაბამისი AM ან PM-ის მითითებით. დროის გადაყვანის დროს თარიღის ცვლილების შემთხვევაში დამატებით რაიმეს მითითება საჭირო არაა.

შემავალი მონაცემების მაგალითი :	გამომავალი მონაცემი ნაჩვენები მაგალითისათვის:
10:15 PM Pacific Central	12:15 AM

1.65. კროსვორდი

(USACO, 2002 წელი, ნოემბერი, “ნარინჯისფერი” დივიზიონი)

ძროხები ხსნიან N x N (3 < = N < = 100) ზომის კროსვორდს, მაგრამ წინასწარ აინტერესებთ რამდენად რთულია ეს კროსვორდი. დაეხმარეთ მათ ამის გარკვევაში. კროსვორდი არის ასეთი ტიპის:

```
- - - - #
- - # # -
- - - - -
- # # - -
# - - - -
```

სადაც ‘-‘ სიმბოლო წარმოადგენს შესავსებ ანუ ცარიელ უჯრედებს, ხოლო ‘#’ სიმბოლო – შეუვსებად ანუ შავ უჯრედებს. კროსვორდში ჩასაწერი ნებისმიერი სიტყვა ორი სიმბოლოსაგან მაინც უნდა შედგებოდეს. მოცემულ მაგალითში არის 4 შვეული და 5 თარაზული სიტყვა. თქვენი ამოცანაა გაარკვიოთ რამდენი სიტყვაა კროსვორდში შვეულად და თარაზულად.

შემავალი მონაცემების ფორმატი:

1 სტრიქონი: ერთადერთი მთელი რიცხვი N ;

2..N+1 სტრიქონები: N ცალი სიმბოლო (‘-‘ ან ‘#’)

გამომავალი მონაცემების ფორმატი: ჰარით გაყოფილი ორი მთელი რიცხვი: თარაზული სიტყვების რაოდენობა; შვეული სიტყვების რაოდენობა

შემავალი მონაცემების მაგალითი(ფაილი cowcross.in): :	გამომავალი მონაცემი ნაჩვენები მაგალითისათვის:
- - - - # - - # # - - - - - - - # # - - # - - - -	5 4

მითითება. შემავალი მონაცემების წაკითხვისას სიმბოლური მასივი გადავწეროთ რიცხვით მასივში ისე, რომ ‘-‘ სიმბოლო შეიცვალოს 1-ით, ხოლო ‘#’ სიმბოლო – 0-ით. ამის შემდეგ ავჯამოთ 0-ების მიერ ჯგუფებად დაყოფილი 1-იანების თითოეული ჯგუფი ცალკე სტრიქონებისა და ცალკე სვეტების მიხედვით. თუკი კონკრეტული ჯამი 1-ზე მეტია, მაშინ შესაბამის ადგილას შესაძლებელია სიტყვის ჩაწერა.

```
READ(n, a)  – სადაც a უკვე  რიცხვითი მასივია
FOR i=1 TO n {
j=1;
        WHILE j<n {
        k=0;
                WHILE a[i,j]=1 {
                k=k+1; j=j+1; }
        IF k>1 THEN k1=k1+1;
        j=j+1; }          }
FOR j=1 TO n {
i=1;
        WHILE i<n {
        k=0;
                WHILE a[i,j]=1 {
                k=k+1; i=i+1; }
      IF k>1 THEN k2=k2+1;
        i=i+1;   ]        }
WRITE (k1,k2);
```

1.66. თავსატეხი სიტყვები

(USACO, 2002 წელი, დეკემბერი, “ნარინჯისფერი” დივიზიონი)

ძროხებს ხელში ჩაუვარდათ თავსატეხების ცნობილი კრებული – “სიტყვების ძებნა”, რომელშიც მოცემულია ქვემოთ ნაჩვენები ტიპის ნაირგვარი სიმბოლური მოზაიკები:

```
H A F B T H E W
J B S Y O Y J S
L G B E S T Y U
T S C O L T E D
J T F S D O W L
W T S E T N O C
```

მოცემულ ცხრილში უნდა იპოვოთ სიტყვები: “BEST” და “CONTEST”. პირველი მათგანი იკითხება მესამე სტრიქონის მესამე პოზიციიდან მარცხნიდან მარჯვნივ, ხოლო მეორე – ბოლო სტრიქონის ბოლო პოზიციიდან მარჯვნიდან მარცხნივ.

სიტყვები შეიძლება დაიწყოს ნებისმიერი სიმბოლოდან და წაკითხულ იქნას ზემოთ, ქვემოთ, მარჯვნივ, მარცხნივ ან დიაგონალურად – ზემოთ-მარჯვნივ, ზემოთ-მარცხნივ, ქვემოთ-მარჯვნივ ან ქვემოთ-მარცხნივ. თქვენი ამოცანაა, დაეხმაროთ ძროხებს სიტყვების მოძებნაში. მოცემულია ცხრილი, რომლის სიგანე N სიმბოლო, ხოლო სიმაღლე M სიმბოლოა ($3 \le N \le 80$, $3 \le M \le 50$) და სიტყვა, რომელიც ჩამალულია ცხრილში. დაწერეთ პროგრამა, რომელიც მოძებნის ამ სიტყვას.

შემავალი მონაცემების ფორმატი: პირველ სტრიქონში – ჰარით გაყოფილი ორი მთელი რიცხვი: N და M; მეორე სტრიქონში – საძებნი სიტყვა; 3..M+2 სტრიქონებში – თითოეულ სტრიქონში ზუსტად N მთავრული სიმბოლო

შემავალი მონაცემების მაგალითი (ფაილი words.in):

```
8 6
CONTEST
H A F B T H E W
J B S Y O Y J S
L G B E S T Y U
T S C O L T E D
J T F S D O W L
W T S E T N O C
```

გამომავალი მონაცემების ფორმატი: პირველ სტრიქონში ჰარით გაყოფილი ორი მთელი რიცხვი – შესაბამისად იმ სტრიქონისა და იმ სვეტის ნომერი, სადაც განლაგებულია საძებნი სიტყვის პირველი სიმბოლო; მეორე სტრიქონში – მიმართულება, რომელშიც ჩაწერილია სიტყვა პირველი სიმბოლოს მიმართ. მიმართულება ვუჩვენოთ კომპასის მსგავსად. თუ სიტყვის პირველი სიმბოლოა ‘x’, მაშინ მიმართულებებია:

```
NW   N   NE
W    x   E
SW   S   SE
```

გამომავალი მონაცემების მაგალითი (ფაილი words.out):

```
6 8
W
```

მითითება. ამოცანის ამოსახსნელად რაიმე განსაკუთრებული ალგორითმი არაა საჭირო. პირველ რიგში, ცხრილში უნდა მოინახოს საძებნი სიტყვის პირველი სიმბოლო, შემდეგ უნდა შემოწმდეს მის გარშემო მოთავსებული სიმბოლოებიდან რომელიმე ემთხვევა თუ არა საძებნი სიტყვის მეორე სიმბოლოს. პირველი ორი სიმბოლოს დამთხვევა უკვე განსაზღვრავს სიტყვის მიმართულებას და ციკლით შემოწმდება დარჩენილი სიმბოლოები.

1.67. სტენოგრაფია

(USACO, 2002 წელი, ნოემბერი, “ნარინჯისფერი” დივიზიონი)

ძროხებს კარგად არ ესმით სიტყვა “სტენოგრაფიის” მნიშვნელობა. ისინი ფიქრობენ, რომ ეს ნიშნავს ასეთი წესით მუშაობას: დაბალი რეგისტრის [‘a’,..,’z’] სიმბოლოებისაგან შედგენილ 250 სიმბოლოზე ნაკლები სიგრძის სიტყვაში უნდა მოიძებნოს ყველაზე ხშირად გამოყენებული

სიმბოლო (თუკი ასეთი რამდენიმეა, ირჩევენ ლათინურ ალფავიტში უფრო წინ მდგომს) და წაიშალოს. დაიბეჭდოს მიღებული სტიქონი და განმეორდეს იგივე მოქმედება მანამ, სანამ არ დარჩება უკანასკნელი სიმბოლო (ცარიელი სტრიქონი არ უნდა დაიბეჭდოს).

შემავალი მონაცემების ფორმატი: ერთადერთ სტრიქონში მოცემული ერთადერთი სიტყვა.

შემავალი მონაცემების მაგალითი (ფაილი shorthand.in):

maryhadalittlelambitsfleecewaswhite

გამომავალი მონაცემების ფორმატი:

თავდაპირველი სიტყვის შემოკლებით მიღებული თითოეული სიტყვა (სტრიქონში – თითო)

გამომავალი მონაცემების მაგალითი (ფაილი shorthand.out):

```
mryhdlittlelmbitsfleecewswhite
mryhdlittllmbitsflcwswhit
mryhdittmbitsfcwswhit
mryhdimbisfcwswhi
mryhdmbsfcwswh
mrydmbsfcwsw
rydbsfcwsw
rydbfcww
rydbfc
rydfc
rydf
ryf
ry
y
```

მითითება. დაბალი რეგისტრის ['a',..,'z'] სიმბოლოები განთავსებულია კოდებზე 97-დან 122-მდე. წავიკითხოთ მოცემული სიტყვის თითოეული სიმბოლო და მისი კოდის შესაბამის ინდექსზე რაიმე რიცხვითი მასივის ელემენტი ვზარდოთ ერთით. მივიღებთ თითოეული სიმბოლოს რაოდენობას სიტყვაში. რიცხვითი მასივის შევსების შემდეგ ვპოულობთ 0-ზე მეტი ელემენტების რაოდენობას და წაშლას ვახორციელებთ ციკლურად მიღებულ რიცხვზე ერთით ნაკლებ მნიშვნელობამდე, რათა ბოლოს უკანასკნელი სიმბოლოც არ წავშალოთ და ცარიელი სტრიქონი არ დავბეჭდოთ.

1.68. ფიბონაჩის რიცხვები

(USACO, 2002 წელი, დეკემბერი, "ნარინჯისფერი" დივიზიონი)

ძროხებმა, მართალია მოგვიანებით, მაგრამ მაინც აღმოაჩინეს ფიბონაჩის რიცხვები 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ..., სადაც მიმდევრობა იწყება რიცხვებით 1, 1, ხოლო ყოველი მომდევნო წევრი წინა ორის ჯამის ტოლია. სამწუხაროდ, ძროხებმა არ იციან სიმბოლო "ჰარი" და მათ მიმდევრობა ჩაწერეს გამყოფი ნიშნებისა და ჰარების გარეშე 11235813213455.... თუმცა წარმოიშვა საინტერესო ამოცანა. მიმდევრობის მოცემული საწყისი ორი წევრისათვის (თითოეული მათგანი დადებითია და 100-ზე ნაკლები) გამოვთვალოთ k-ური ციფრი. შემავალი მონაცემები ისეთია, რომ მიმდევრობაში მიღებული ყოველი რიცხვი გარანტირებულად თავსდება 32-ბიტიან მთელ რიცხვში.

პროგრამის სახელია cowfib.

შემავალი მონაცემების ფორმატი: ერთადერთი სტრიქონი სამი მთელი რიცხვით – ძროხული მიმდევრობის პირველი რიცხვი; ძროხული მიმდევრობის მეორე რიცხვი; k

შემავალი მონაცემების მაგალითი (ფაილი cowfib.in):

1 1 10

გამომავალი მონაცემების ფორმატი:

ერთადერთი სტრიქონი ერთი ციფრით, რომელიც წარმოადგენს k-ურ ციფრს მოცემულ მიმდევრობაში.

გამომავალი მონაცემების მაგალითი (ფაილი cowfib.out):

1

მითითება. ამოცანის ამოსახსნელად საჭიროა ე.წ. "დიდი რიცხვების არითმეტიკის" გამოყენება (იხ. ამოცანა 1.29). ერთ მასივში საჭიროა ახალი რიცხვების გენერაცია, ხოლო მეორეში – მიღებული რიცხვის მიერთება მანამდე არსებული მიმდევრობისათვის. როცა მიმდევრობის სიგრძე k-ს გადაამეტებს, დავბეჭდოთ k-ური ელემენტის მნიშვნელობა.

1.69. ვერტიკალური ჰისტოგრამა

(USACO, 2003 წელი, თებერვალი, "ნარინჯისფერი" დივიზიონი)

დაწერეთ პროგრამა, რომელიც შემავალი ფაილიდან წაიკითხავს ზედა რეგისტრის (ანუ მთავრული სიმბოლოების) ოთხსტრიქონიან ტექსტს (თითოეულ სტრიქონში 72 სიმბოლოზე მეტი არ გვხვდება) და შეადგენს ჰისტოგრამას, რომელიც გვაჩვენებს თუ რამდენჯერ გვხვდება ტექსტში თითოეული სიმბოლო (არ იგულისხმება ჰარი, ციფრები და პუნქტუაცია). გამოტანის ფორმატი ზუსტად უნდა ემთხვეოდეს ქვემოთ ნაჩვენებს.

შემავალი მონაცემები: ზედა რეგისტრის ტექსტის 4 სტრიქონი (სტრიქონში არა უმეტეს 72 სიმბოლო).

შემავალი მონაცემების მაგალითი (ფაილი vhist.in):

```
THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
THIS IS AN EXAMPLE TO TEST FOR YOUR
HISTOGRAM PROGRAM.
HELLO!
```

გამომავალი მონაცემები:

ვარსკვლავებითა და ჰარებით შედგენილი რამდენიმე სტრიქონი, რომელსაც თან ახლავს ზედა რეგისტრის ანბანის ჰარებით გაყოფილი სიმბოლოები. სტრიქონების თავსა და ბოლოში ჰარები არ დაბეჭდოთ.

გამომავალი მონაცემების მაგალითი (ფაილი vhist.out):

```
                            *
                            *
          *                 *
          *                 *       *   *
          *                 *       *   *
*         *       *         *       *   *
*         *       * *   * * *       * * *
*         *     * * *   * * *   * * * * *
*       * * * * * * *     * * * * * * *       * *
* * * * * * * * * * * * * * * * * * * * * * * * * *
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

1.70. ძროხების ტრანსმიგრაცია

(USA OPEN, 2003 წელი, აპრილი, "ნარინჯისფერი" დივიზიონი)

ფერმერ ჯონის ძროხებს სურთ მოხვდნენ მთვარეზე, მაგრამ სამწუხაროდ, პრობლემების მთელი რიგი ხელს უშლის ამ სურვილის განხორციელებას.

ადგილობრივმა ჯადოქარმა დაამზადა P (1≤P≤150000) მიქსტურა, რათა დაეხმაროს ძროხებს მთვარეზე გადასვლაში. მიქსტურები შეიძლება გამოყენებულ იქნან მხოლოდ მათი ნომრების შესაბამისი თანმიმდევრობით, თუმცა შესაძლებელია ზოგი მათგანის გამოტოვებაც. ყოველ მიქსტურას აქვს თავისი ძალა (1≤ ძალა ≤500), რომელიც მოქმედებს მთვარეზე გადასვლის უნარზე: კენტ ბიჯზე მიღებული მიქსტურა აძლიერებს, ხოლო ლუწ ბიჯზე მიღებული მიქსტურა ამცირებს ამ უნარს. მიქსტურების მიღებამდე ძროხათა მთვარეზე მოხვედრის უნარი 0-ის ტოლია.

ყოველი მიქსტურა შეიძლება მიღებულ იქნას მხოლოდ ერთხელ. ყოველი მიქსტურის მიღება ითვლება ერთ ბიჯად. ორ მეზობელ ბიჯს შორის (ასევე პირველ ბიჯამდე და ბოლო ბიჯის შემდეგაც) შეიძლება გამოტოვებულ იქნას ნებისმიერი რაოდენობის მიქსტურა.

დაწერეთ პროგრამა, რომელიც გაარკვევს თუ რომელი მიქსტურები უნდა მიიღონ ძროხებმა, რათა მათ შეიძინონ მთვარეზე გადასვლის მაქსიმალური უნარი.

შემავალი მონაცემების ფორმატი: 1 სტრიქონი: ერთადერთი მთელი რიცხვი P ;2..P+1 სტრიქონები: ერთადერთი მთელი რიცხვი, რომელიც განსაზღვრავს მიქსტურის ძალას, მე-2 სტრიქონში მოცემულია 1-ლი მიქსტურის ძალა, მე-3 სტრიქონში – მე-2 მიქსტურის და ა.შ.

გამომავალი მონაცემების ფორმატი:

ერთადერთი მთელი რიცხვი, რომელიც განსაზღვრავს მთვარეზე გადასვლის მაქსიმალურ უნარს

შემავალი მონაცემების მაგალითი (ფაილი jumpcow.in):

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

გამომავალი მონაცემების მაგალითი (ფაილი jumpcow.out):

```
17
```

განმარტება. მე-2, მე-5 და მე-7 ნომერი მიქსტურების (მნიშვნელობები შესაბამისად 2, 4 და 5) გამოტოვება გვაძლევს შემდეგ თანმიმდევრობას +7-1+8-3+6=17, რომელიც მაქსიმალურია ყველა შესაძლო კომბინაციათა შორის.

მითითება. რიცხვთა მოცემული მიმდევრობიდან გამოვყოთ ზრდადი ქვემიმდევრობები. მათში მომგებიანია ლუწ სვლაზე ავიღოთ პირველი წევრი, კენტ სვლაზე კი – ბოლო. ზრდად ქვემიმდევრობებს შორის მოქცეული არაზრდადი ქვემიმდევრობები შეგვიძლია უგულვებელვყოთ (ანუ გამოვტოვოთ). ცალკე უნდა განვიხილოთ საწყისი მიმდევრობის დასაწყისი და ბოლო. თუ მიმდევრობა ზრდადი ქვემიმდევრობით იწყება – პირველ სვლაზე ვიღებთ მის ბოლო წევრს, ხოლო თუ არაზრდადი ქვემიმდევრობით იწყება – ვიღებთ პირველ წევრს. პროცესს ვასრულებთ კენტი სვლით.

1.71. დაკარგული ძროხები

(USA OPEN, 2003 წელი, აპრილი, "ნარინჯისფერი" დივიზიონი)

N (1≤N≤150000) ძროხას მინიჭებული აქვს უნიკალური ნომერი 1-დან N-მდე. ერთხელ, შუადღისას ისინი სტუმრობდნენ მეზობელ საძოვარზე, სადაც მათ ზომაზე ბევრად მეტი ლუდი მიირთვეს და ვახშმის წინ ვეღარ მოახერხეს თავიანთი ნომრების ზრდადობის შესაბამისად დამდგარიყვნენ რიგში.

ფერმერმა ჯონმა სამწუხაროდ არ იცის, თუ როგორ დაალაგოს ისინი. ამასთან, მან განსაზღვრა არა თითოეული ძროხის ნომერი, არამედ ფრიად უცნაური სტატისტიკა: ყოველი ძროხისათვის მან დაადგინა, თუ რამდენი ნაკლები ნომრის მქონე ძროხა დგას მის წინ.

დაწერეთ პროგრამა, რომელიც ძროხების ზუსტ თანმიმდევრობას.

შემავალი მონაცემების ფორმატი:

1 სტრიქონი: ერთადერთი მთელი რიცხვი N ;

2..N სტრიქონები: ერთადერთი მთელი რიცხვი, რომელიც შესაბამის ადგილზე მყოფი ძროხისათვის განსაზღვრავს თუ რამდენი მასზე ნაკლები ნომრის მქონე ძროხა დგას მის წინ. პირველ ადგილას მდგარი ძროხისათვის ეს რიცხვი ყოველთვის 0-ს უდრის, ამიტომ ის ამ ჩამონათვალში არ შედის და მე-2 სტრიქონში მოცემულია მე-2 ძროხის წინ ნაკლები ნომრის მქონე ძროხების რაოდენობა, მე-3 სტრიქონში – მე-3 ადგილზე მყოფი ძროხის და ა.შ.

შემავალი მონაცემების მაგალითი (ფაილი lost.in):

5
1
2
1
0

შემავალი მონაცემების განმარტება: სულ 5 ძროხაა. მეორე ადგილზე მყოფი ძროხის წინ დგას ერთი ძროხა, რომელსაც მასზე უფრო ნაკლები ნომერი აქვს. მე-3 ადგილზე მყოფი ძროხის წინ დგას 2 უფრო ნაკლები ნომრის მქონე ძროხა. მე-4 ადგილზე მყოფი ძროხის წინ დგას 1 უფრო ნაკლები ნომრის მქონე ძროხა. მე-5 ადგილზე მყოფი ძროხის წინ არცერთ ძროხას არა აქვს მასზე ნაკლები ნომერი.

გამომავალი მონაცემების ფორმატი:

1-დან N-მდე სტრიქონში გამოდის თითო მთელი რიცხვი, რომელიც განსაზღვრავს შესაბამის ადგილზე მდგომი ძროხის ნომერს.

გამომავალი მონაცემების მაგალითი (ფაილი lost.out):

2
4
5
3
1

მითითება. თუკი რიცხვთა მოცემულ მიმდევრობაში რიცხვის სიდიდე ემთხვევა მის ნომერს, შეგვიძლია ვთქვათ, რომ შესაბამის ძროხას ყველა მის წინ მდგომზე მეტი ნომერი აქვს. ასეთ რიცხვებს შორის ყველაზე მარჯვენა კი მიუთითებს რიგში მყოფი ძროხებიდან ყველაზე დიდი ნომრის მქონეს. მისი განსაზღვრის შემდეგ ის შეგვიძლია ამოვიღოთ რიგიდან. მის მარჯვნივ მყოფებს ინდექსი 1-ით შევუმციროთ და პროცესი გავიმეოროთ. მიმდევრობაში ყოველთვის მოიძებნება ერთი მაინც რიცხვი, რომელიც საკუთარი ინდექსის ტოლია, რადგან ნებისმიერ მომენტში ერთ-ერთ ძროხას ყოველთვის ექნება სხვებზე მეტი ნომერი.

1.72. საძოვარი მინდვრები

(USACO, 2002 წელი, ზამთრის პირველობა, “მწვანე” დივიზიონი)

ფერმერ ჯონს აქვს გრძელი ღობე, დამზადებული რელსებისგან, რომლებიც ერთმანეთთან შეერთებულია ღობის წვეროებში. ყოველი N (1≤N≤3 000) ღობის წვერო მონიშნულია თითო რიცხვით –1000-დან 1000-ის ჩათვლით. შესაძლებელია, რომ რამდენიმე წვეროს შეესაბამებოდეს ერთნაირი რიცხვი.

ფერმერის ძროხებმა მოიფიქრეს შემდეგი თამაში. ძროხა, რომელიც მოძებნის ”ღობეების საუკეთესო ჯამს”, იგებს საპრიზო ნაყინს. თამაშის მოსაგებად ძროხამ უნდა მოძებნოს ღობის თანმიმდევრული წვეროების უგრძელესი სიმრავლე, რომ წვეროების შესაბამისი რიცხვების ჯამის აბსოლუტური სიდიდე აღმოჩნდეს უმცირესი. დავეხმაროთ ძროხას მოსაგები ჯამის განსაზღვრაში.

შემავალი ფაილის ფორმატი: სტრიქონი 1 – მხოლოდ ერთი მთელი რიცხვი N; 2–N+1 სტრიქონებიდან ყოველი შეიცავს წვეროს შესაბამის რიცხვს: სტრიქონი 2 შეიცავს მიმდევრობაში პირველი წვეროს შესაბამის რიცხვს და ა.შ.

შემავალი ფაილის მაგალითი (ფაილი pasture.in):

6
5
10
-5
-6
2
4

გამომავალი ფაილის ფორმატი:

ერთი სტრიქონი, რომელიც შეიცავს სამ რიცხვს: იმ წვეროს ნომერი, რომელიც ჯამში იკრიფება პირველი; იმ წვეროს ნომერი, რომელიც ჯამში იკრიფება ბოლო; მოძებნილი ჯამის აბსოლუტური სიდიდე.

ერთნაირი ჯამის შემთხვევაში აირჩიეთ უგრძელესი მიმდევრობა, ხოლო ერთნაირი ჯამისა და მიმდევრობის სიგრძის შემთხვევაში აირჩიეთ იმ წვეროების მიმდევრობა, რომელიც იწყება უმცირესი ნომრის წვეროთი.

გამომავალი ფაილის მაგალითი (ფაილი pasture.out):

4 6 0

მითითება. ამოცანა მარტივად იხსნება ორი ერთმანეთში ჩადგმული ციკლით. გარე ციკლი აფიქსირებს განსახილველი ჯამის მარცხენა ბოლოს, ხოლო შიგა ციკლი – მარჯვენა ბოლოს. ამ ორი ციკლის მეშვეობით განხილული იქნება ყველა შესაძლო ჯამი. ამასთან მარცხენა ბოლოს დაფიქსირებისას აჯამვას ვიწყებთ 0-დან და მის მარჯვნივ მდებარე ყველა რიცხვს სათითაოდ ვუმატებთ რაიმე ცვლადს. პარალელურად ყოველ ბიჯზე ვამოწმებთ ხომ არ გახდა ეს ცვლადი მინიმუმზე ნაკლები (მინიმუმს თავდაპირველად დიდ რიცხვს მივანიჭებთ). ერთნაირი ჯამების შემთხვევისათვის დამატებითი შემოწმებაა საჭირო.

```
READ (n, A)
k=maxlongint
FOR i0=1 TO n {
  s=0
  FOR i=i0 TO n {
     s=s+A[i]
     IF ( (k>abs(s)) OR ( (k=abs(s)) AND ((i-i0)>(j-l)) )) THEN {  k=abs(s); l=i0; j=i;   }
  }          }
WRITE (l, j, k);
```

1.73. წილადებიანი გამოსახულება

(ბალტიისპირეთის ქვეყნების ოლიმპიადა ინფორმატიკაში, 2000 წელი)

წილადებიანი გამოსახულება ეწოდება შემდეგი ფორმის არითმეტიკულ გამოსახულებას: $x_1/x_2/x_3\ldots/x_k$, ნებისმიერი i ($1 \le i \le k$) რიცხვისათვის x_i დადებითი მთელი რიცხვია. გაყოფა სრულდება მარცხნიდან მარჯვნივ. მაგალითად 1/2/1/2 გამოსახულების მნიშვნელობა 1/4-ის ტოლია. ჩვენ შეგვიძლია დავსვათ ფრჩხილები ამ გამოსახულებაში, რათა შევცვალოთ მისი მნიშვნელობა. მაგალითად (1/2)/(1/2) გამოსახულების მნიშვნელობა 1-ის ტოლია.

მოცემული გვაქვს E წილადებიანი გამოსახულება. უნდა დავადგინოთ, შესაძლებელია თუ არა ფრჩხილების საშუალებით ისეთი E′ გამოსახულების მიღება, რომლის მნიშვნელობაც იქნება მთელი რიცხვი.

დაწერეთ პროგრამა, რომელიც რამდენიმე მონაცემთა ბაზიდან თითოეულისათვის:

- წაიკითხავს E გამოსახულებას ფაილიდან DIV.IN;
- შეამოწმებს არის თუ არა შესაძლებელი E-ში ფრჩხილების ისეთნაირად დასმა, რომ მიღებული E′ გამოსახულების მნიშვნელობა მთელი იყოს;
- დაბეჭდავს შედეგს DIV.OUT ფაილში.

შემავალი მონაცემების ფორმატი:

DIV.IN ფაილის პირველი სტრიქონი შეიცავს ერთადერთი დადებით მთელ d ($d \le 5$) რიცხვს, რომელიც აღნიშნავს სხვადასხვა მონაცემთა ბაზების რაოდენობას. შემდეგ მოდის მონაცემთა თითოეული ბაზა, რომლის პირველი სტრიქონი შეიცავს მთელ n-ს ($2 \le n \le 10000$), რომელიც წარმოადგენს მთელი რიცხვების რაოდენობას ამ გამოსახულებაში. მომდევნო n სტრიქონი შეიცავს თითო მთელს, რომელიც არ აღემატება 1000000000-ს. i-ურ სტრიქონში მდგომი რიცხვი შეესაბამება წილადებიან გამოსახულებაში i-ურ ადგილზე მდგომ რიცხვს.

გამომავალი მონაცემების ფორმატი:

ყოველი i-სათვის (1≤i≤d) პროგრამამ DIV.OUT ფაილის i-ურ სტრიქონში უნდა დაბეჭდოს ერთი სიტყვა YES ან NO. თუ i-ური გამოსახულება შეიძლება ისე გარდავქმნათ, რომ მივიღოთ მთელი რიცხვი, მაშინ დაიბეჭდება YES, წინააღმდეგ შემთხვევაში NO.

DIV.IN	DIV.OUT
2 4 1 2 1 2 3 1 2 3	YES NO

მითითება. ნებისმიერი სიგრძის წილადებიან გამოსახულებაში შეგვიძლია ფრჩხილები ისე განვალაგოთ, რომ ყველა რიცხვი, გარდა რიგით მეორისა, წილადის მრიცხველში მოექცეს. აქედან გამომდინარე, ამოცანა მარტივდება – თუ კონკრეტულ მონაცემთა ბაზაში შემავალი ყველა რიცხვის ნამრავლი რიგით მეორის გარდა, უნაშთოდ იყოფა რიგით მეორე ელემენტზე (მისი მოხვედრა წილადის მრიცხველში გამორიცხულია), მაშინ ამოცანის პასუხია YES, წინააღმდეგ შემთხვევაში – NO.

2. მონაცემთა სტრუქტურები

პროგრამირებაში ხშირადა გვაქვს საქმე სიმრავლეებთან, რომლებიც იცვლებიან ალგორითმის მუშაობის პროცესში. ამგვარ სიმრავლეებს **დინამიურ** (dynamic) **სიმრავლეებს** ეძახიან. ჩვეულებისამებრ, დინამიური სიმრავლის ელემენტს წარმოადგენს ჩანაწერი, რომელიც შეიცავს სხვადასხვა მინდვრებს. ხშირად ამ მინდვრებიდან ერთ-ერთი განიხილება როგორც **გასაღები** (key), რომელიც განკუთვნილია ელემენტის ცალსახად განსაზღვრისათვის (იდენტიფიკაციისათვის), ხოლო დანარჩენი მინდვრები განიხილებიან, როგორც **დამატებითი მონაცემები** (satellite data), რომლებიც ინახებიან გასაღებთან ერთად. სიმრავლის ელემენტი იძებნება გასაღების მიხედვით, ხოლო მოძებნის შემდეგ შესაძლებელია დამატებითი მონაცემების წაკითხვა ან შეცვლა. ხშირად ყველა გასაღები ერთმანეთისაგან განსხვავდება და მათთან ერთად ელემენტში ინახება არა მარტო დამატებითი ინფორმაცია, არამედ სამუშაო სახის ინფორმაციაც (მაგ.: მიმთითებელი სხვა ელემენტზე).

ხშირ შემთხვევაში გასაღებთა სიმრავლეზე არსებობს ბუნებრივი წრფივი თანმიმდევრობა. ასეთ დროს შესაძლოა განისაზღვროს სიმრავლის უმცირესი ელემენტი ან მოცემული ელემენტის უშუალო მეზობელი. შეგვიძლია ჩავთვალოთ, რომ სიმრავლის ელემენტები ინახება მეხსიერების რაღაც არეში და ყოველი ელემენტისათვის არსებობს მიმთითებელი, რომელიც საშუალებას გვაძლევს მივმართოთ ამ ელემენტს. მიმთითებლის როლში, როგორც წესი, გამოდის მეხსიერების მისამართი, ხოლო თუ პროგრამულ ენას ასეთი საშუალება არა აქვს, მიმთითებლის როლში გამოდის მასივის ინდექსი.

მოქმედებები სიმრავლეებზე იყოფიან **მიმართვებად** (queris), რომლებიც არ ცვლიან სიმრავლეებს და სიმრავლეში **ცვლილების შემტან** (modifying operations) ოპერაციებად. ჩამოვთვალოთ ტიპიური ოპერაციები სიმრავლეებზე.

SEARCH(S,k) (ძებნა) – მიმართვა, რომელიც მოცემული S სიმრავლისა და k გასაღებისათვის პასუხად აბრუნებს მიმთითებელს S სიმრავლეში k გასაღების მქონე ელემენტზე. თუკი ასეთი ელემენტი S სიმრავლეში არ არსებობს, პასუხად ბრუნდება nil.

INSERT(S,x) – S სიმრავლეს უმატებს ელემენტს, რომლის მიმთითებელია x (იგულისხმება, რომ დამატების მომენტისათვის ყველა მინდორი ჩანაწერში, რომელსაც მიუთითებს x, შევსებულია).

DELETE(S,x) – შლის S სიმრავლიდან ელემენტს, რომელზეც მიუთითებს x (ყურადღება მიაქციეთ იმას, რომ x მიმთითებელია და არა გასაღები).

MINIMUM(S) – პასუხად აბრუნებს მიმთითებელს S სიმრავლის უმცირეს გასაღებზე (ითვლება, რომ გასაღებები წრფივად დალაგებულია)..

MAXIMUM(S) – პასუხად აბრუნებს მიმთითებელს S სიმრავლის უდიდეს გასაღებზე

SUCCESSOR(S,x) (მომდევნო) – პასუხად აბრუნებს მიმთითებელს S სიმრავლის იმ ელემენტზე, რომელიც უშუალოდ x ელემენტის შემდეგ დგას (გასაღებების წრფივი დალაგების შემთხვევაში). თუ x უდიდესი ელემენტია – ბრუნდება nil.

PREDECESSOR(S,x) (წინა) – პასუხად აბრუნებს მიმთითებელს S-ის იმ ელემენტზე, რომელიც უშუალოდ x ელემენტის წინ დგას (გასაღებების წრფივი დალაგების შემთხვევაში). თუ x უმცირესი ელემენტია – ბრუნდება nil.

ოპერაციათა ღირებულება დამოკიდებულია იმ სიმრავლეთა ზომებზე, რომელთა მიმართაც ისინი გამოიყენებიან.

2.1. წრფივი სიები

დინამიური სიმრავლეების ყველაზე მარტივი სახეა წრფივი სია.

წრფივი სია წარმოადგენს n≥0 ელემენტის X[1], X[2],…,X[n] მიმდევრობას, რომლის მნიშვნელოვანი თავისებურებაა შემადგენელ ელემენტთა ისეთი ურთიერთდამოკიდებულება, თითქოს ისინი ერთ ხაზზე იყვნენ განლაგებული. სხვაგვარად რომ ვთქვათ, ასეთ სტრუქტურაში დაცული უნდა იქნას შემდეგი პირობა: თუ n>0 და X[1] არის პირველი ელემენტი, ხოლო X[n] – ბოლო, მაშინ k-ური ელემენტი X[k] მოთავსებულია X[k-1] ელემენტის შემდეგ და X[k+1] ელემენტის წინ ნებისმიერი 1<k<n-სათვის.

წრფივ სიებზე შეიძლება შესრულდეს შემდეგი ოპერაციები:

1) მოვახდინოთ მიმართვა სიის k-ურ ელემენტზე მისი მინდვრების მნიშვნელობის შემოწმების და(ან) შეცვლის მიზნით; 2) ახალი ელემენტის ჩამატება უშუალოდ k-ური ელემენტის წინ ან შემდეგ; 3) k-ური ელემენტის წაშლა; 4) ერთ წრფივ სიაში ორი (ან მეტი) წრფივი სიის გაერთიანება; 5) წრფივი სიის გაყოფა ორ (ან მეტ) წრფივ სიად; 6) წრფივი სიის ასლის შექმნა; 7) წრფივ სიაში ელემენტების რაოდენობის განსაზღვრა; 8) ელემენტების დალაგება ამ ელემენტებით განსაზღვრული მინდვრების მნიშვნელობების ზრდადობით (ან კლებადობით); 9) ელემენტის მოძებნა რომელიმე მინდვრის მოცემული მნიშვნელობით.

ერთ პროგრამულ კოდში ერთდროულად ცხრავე ოპერაციის გამოყენება იშვიათად ხდება, ამიტომ წრფივი სიების წარმოდგენა შეიძლება მრავალნაირად განხორციელდეს იმის მიხედვით, თუ რა ტიპის ოპერაციებია ჩასატარებელი სიაზე. აქვე შევნიშნოთ, რომ წრფივი სიის ისეთი წარმოდგენა, რომელიც ერთნაირად ეფექტურს გახდიდა ყველა ოპერაციის ჩატარებას, პრაქტიკულად შეუძლებელია.

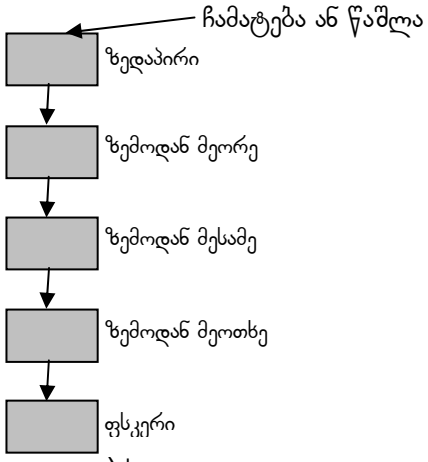
წრფივმა სიებმა, რომლებშიც ჩასმის, წაშლის და მნიშვნელობებზე მიმართვის ოპერაციები ძირითადად ხდება პირველ ან ბოლო ელემენტზე, მიიღეს შემდეგი სპეციალური სახელები:

სტეკი – ესაა წრფივი სია, რომელშიც ჩასმის, წაშლის და მნიშვნელობებზე მიმართვის ოპერაციები ხდება მხოლოდ სიის ერთ ბოლოზე;

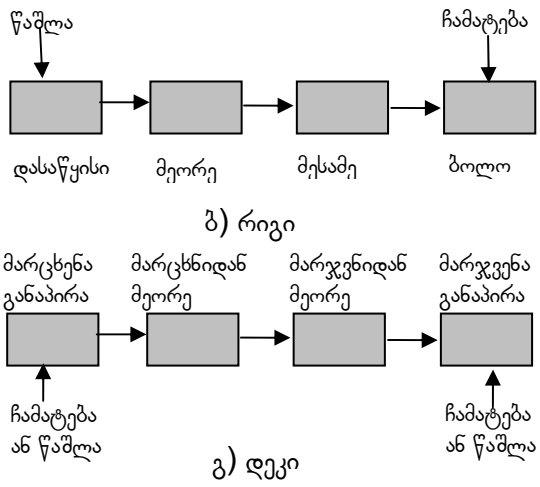
რიგი ანუ ცალმხრივი რიგი – ესაა წრფივი სია, რომელშიც ჩასმის ყველა ოპერაცია ხორციელდება ერთ ბოლოზე , ხოლო წაშლის (და როგორც წესი, მნიშვნელობებზე მიმართვის) ოპერაცია – მეორე ბოლოზე;

დეკი ანუ ორმხრივი რიგი – ესაა წრფივი სია, რომელშიც ჩასმის, წაშლის და მნიშვნელობებზე მიმართვის ოპერაციები ხდება სიის ორივე ბოლოზე;

სტეკში წაშლის ოპერაციის შესრულებისას პირველ ყოვლისა წაიშლება სიის ყველაზე “უმცროსი” ობიექტი, ანუ ობიექტი, რომელიც ყველაზე ბოლოს იქნა ჩასმული სტეკში. რიგში პირიქით ხდება: ყველაზე ადრე წაიშლება “უფროსი” ობიექტი, ანუ ის, რომელიც ყველაზე ადრე ჩაისვა რიგში. ბევრი მკვლევარი ერთდროულად მიხვდა მონაცემთა ასეთი სტრუქტურების დიდ მნიშვნელობას, ამიტომ “სტეკს” და “რიგს” ლიტერატურაში მრავალი შესატყვისი გააჩნიათ. მაგალითად სტეკს ხშირად უწოდებენ მჭიდურ სიას (push-down list), რევერსიულ საცავს (reversion storages), მჭიდს (cellars), ჩალაგებულ საცავს (nesting stores), ფლუგერულ სიას (yo-yo lists). რიგს ხშირად უწოდებენ ციკლურ საცავს (circular stores). სტეკის მოქმედების პრინციპს უწოდებენ “ბოლოს მოვიდა – პირველი წავიდა” (last-in, first-out, შემოკლებით – LIFO), ხოლო რიგის მოქმედების პრინციპია “პირველად მოვიდა – პირველი წავიდა” (first-in, first-out – FIFO).



ა) სტეკი

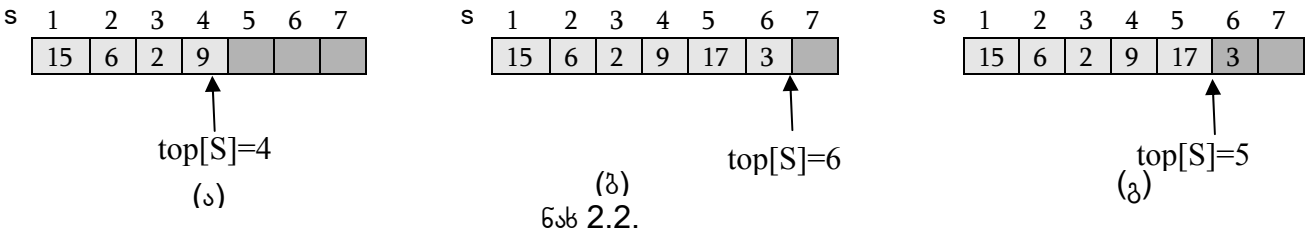


ბ) რიგი

გ) დეკი

ნახ. 2.1.

მონაცემთა აღწერილ სტრუქტურებთან მიმართებაში გამოიყენება სპეციალური ტერმინოლოგია. იტყვიან, რომ ელემენტს ათავსებენ სტეკის ზედაპირზე ან ხსნიან სტეკის ზედაპირიდან. მიმართვა იმ ელემენტზე, რომელიც მოთავსებულია სტეკის ფსკერზე, გარკვეულწილად გართულებულია, რადგან მას ვერ მივმართავთ მანამ, სანამ ყველა სხვა ელემენტი სტეკიდან არ წაიშლება (ანუ მოიხსნება სტეკის ზედაპირიდან). ხშირად იყენებენ გამოთქმას, რომ ელემენტი ჩაიტვირთა (push) სტეკში ან ამოიტვირთა სტეკიდან (pop). რიგთან მიმართებაში გამოიყენება ტერმინი “დასაწყისი” (front) და “ბოლო” (rear). დეკთან დაკავშირებით კი განასხვავებენ მარცხენა და მარჯვენა ბოლოებს.



ნახ 2.2.

ნახ. 2.2-ზე ნაჩვენებია არაუმეტეს n ელემენტის მოცულობის მქონე სტეკის რეალიზაცია S[1..n] მასივის ბაზაზე. მასივთან ერთად ჩვენ ვინახავთ რიცხვს top[S], რომელიც წარმოადგენს სტეკისათვის უკანასკნელად დამატებული ელემენტის ინდექსს. სტეკი შედგება ელემენტებისაგან S[1..top[S]], სადაც S[1] სტეკის ფსკერია, ხოლო S[top[S]] – სტეკის ზედაპირი. ნახ. 2.2-ზე ღია რუხი ფერის მქონე უჯრედებში ჩაწერილია სტეკის ელემენტები. (ა)-ს მიხედვით S სტეკი შეიცავს 4 ელემენტს და მისი ზედა ელემენტია რიცხვი 9. (ბ)-ზე მოცემულია იგივე სტეკი PUSH(S,17) და PUSH(S,3) ოპერაციების შემდეგ. (გ)-ზე სტეკი მოცემულია მას შემდეგ, რაც POP(S) ოპერაციამ დააბრუნა ზედა ელემენტის მნიშვნელობა – 3. მართალია, რიცხვი 3 მასივში ჯერ კიდევ არის, მაგრამ სტეკში ის უკვე აღარ იმყოფება და სტეკის ზედაპირზეა რიცხვი 17.

თუკი top[S]=0, მაშინ სტეკი **ცარიელია** (is empty). თუკი top[S]=n, მაშინ სტეკში ელემენტის დამატების მცდელობისას ხდება **გადავსება** (overflow), რადგან სტეკის ზომა ჩვენს მაგალითში განსაზღვრულია n-ით. სიმეტრიული სიტუაცია იქმნება ცარიელი სტეკიდან ელემენტის ამოღების მცდელობისას. ინგლისურად ამ შემთხვევას უწოდებენ underflow, ქართულად შეიძლება ვუწოდოთ “გადაცარიელება”.

ოპერაციები სტეკზე (სიცარიელის შემოწმება, ელემენტის დამატება, ელემენტის ამოღება) ასე ჩაიწერება:

```
STACK-EMPTY(S)
1 if top[S]=0 then { return TRUE }
2                  else { return FALSE }
```

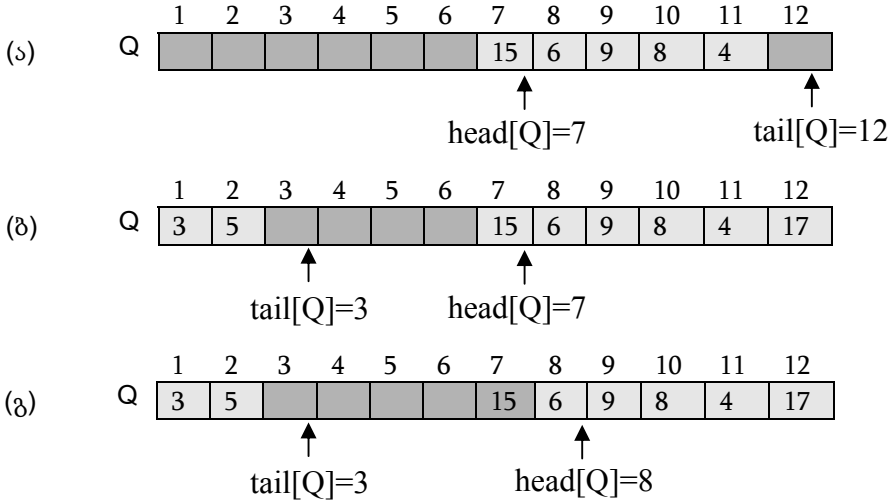
```
PUSH(S,x)
1 top[s]=top[S]+1
2 S[top[S]]=x

POP(S)
1 if STACK-EMPTY(S) then { error “underflow” }
2                        else { top[s]=top[S]-1
3                               return S[top[S]+1]
```

სტეკთან დაკავშირებული სამივე ოპერაცია სრულდება O(1) დროში.

რიგისათვის ელემენტის დამატების ოპერაციას დავარქვათ ENQUEUE, ხოლო რიგიდან ელემენტის წაშლის ოპერაციას – DEQUEUE. სტეკის მსგავსად წასაშლელი ელემენტი რიგშიც ცალსახად განსაზღვრულია და ამიტომ ის კი არ გადაეცემა DEQUEUE პროცედურას, არამედ ეს პროცედურა აბრუნებს მის მნიშვნელობას.

რიგს გააჩნია დასაწყისი ანუ თავი (head) და ბოლო (tail). რიგისათვის დამატებული ელემენტი აღმოჩნდება მის ბოლოში (როგორც უკანასკნელად მოსული მყიდველი ნამდვილ რიგში), ხოლო წასაშლელი ელემენტი იმყოფება რიგის თავში (როგორც ყველაზე ადრე მოსული მყიდველი).



ნახ. 2.3.

ნახ. 2.3-ზე ნაჩვენებია არაუმეტეს n-1 ელემენტის შემცველი რიგის რეალიზაცია Q[1..n] მასივის ბაზაზე. ჩვენ ვინახავთ რიცხვებს: head[Q] – რიგის დასაწყისის ინდექსი და tail[Q] – თავისუფალი უჯრედის ინდექსი, რომელშიც უნდა მოთავსდეს რიგისათვის დამატებული მომდევნო ელემენტი. ნახ 2.3-ზე ღია რუხი ფერის მქონე უჯრედები შეიცავენ რიგის ელემენტებს. (ა)-ზე მოცემულია 5-ელემენტიანი რიგი Q[7]-დან Q[11]-მდე. (ბ)-ზე მოცემულია იგივე რიგი ENQUEUE(Q,17), ENQUEUE(Q,3) და ENQUEUE(Q,5) ოპერაციების შემდეგ. შევნიშნოთ, რომ Q მასივი განიხილება, როგორც წრიული და n-ური ელემენტის შემდეგ მოდის პირველი ელემენტი. (გ)-ზე რიგი მოცემულია DEQUEUE(Q) ოპერაციის შემდეგ, რომელიც აბრუნებს მნიშვნელობას 15 და რიგის ახალ დასაწყისად ხდის რიცხვს 6.

თუ head[Q]=tail[Q], მაშინ რიგი ცარიელია, ხოლო თავდაპირველად ყოველთვის head[Q]=tail[Q]=1. ცარიელი რიგიდან ელემენტის წაშლის მცდელობა იწვევს underflow (გადაცარიელება) შეცდომას, ხოლო თუ რიგი მთლიანადაა შევსებული, მისთვის ელემენტის დამატების მცდელობა იწვევს overflow (გადავსება) შეცდომას. ქვემოთ მოყვანილ პროცედურებში ეს შეცდომები იგნორირებულია.

```
ENQUEUE(Q,x)
1 Q[tail[Q]]=x
2 if tail[Q]=length[Q] then   { tail[Q]=1 }
3                      else   { tail[Q]=tail[Q]+1 }
```

```
DEQUEUE(Q)
1 x=Q[head[Q]]
2 if head[Q]=length[Q]  then   { head[Q]=1 }
3                              else   { head[Q]=head[Q]+1 }
4 return x
```

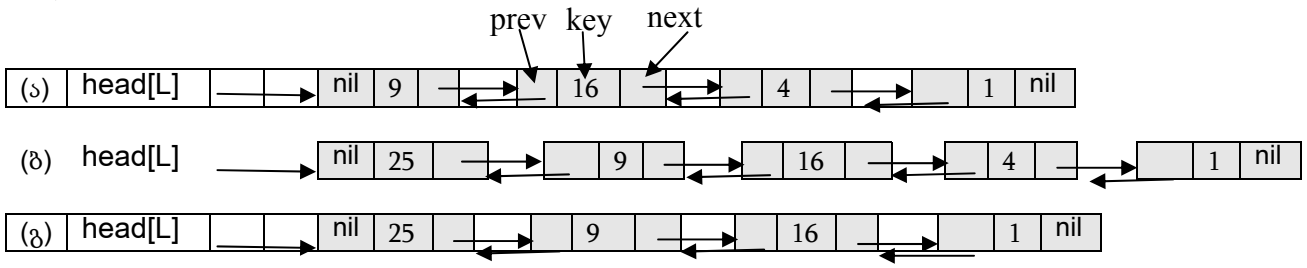
ორივე პროცედურის მუშაობის დროა O(1).

2.2. ბმული სიები

ბმულ სიაში (linked list) ან უბრალოდ სიაში ელემენტები წრფივადაა დალაგებული, მაგრამ დალაგება განსაზღვრულია არა ნომრებით (როგორც მასივში), არამედ მიმთითებლებით, რომლებიც შედიან სიის ელემენტებში. სიები წარმოადგენენ დინამიური სიმრავლეების შენახვის მოხერხებულ საშუალებას, რომლებზეც შესაძლებელია ამ თავის შესავალში მოცემული ყველა ოპერაციის რეალიზაცია.

თუკი ყოველი რიგში მდგომი დაიმახსოვრებს მის შემდეგ მდგომს, მივიღებთ ცალმხრივად ბმულ სიას (თუნდაც ისინი უწესრიგოდ იყვნენ გარშემო გაბნეულნი), ხოლო თუ თითოეული დაიმახსოვრებს მის წინ მდგომსაც, მივიღებთ ორმხრივად ბმულ სიას.

სხვაგვარად რომ ვთქვათ, **ორმხრივად ბმული სია** (double linked list) წარმოადგენს ჩანაწერს, რომელიც შეიცავს სამ მინდორს: key (გასაღები) და ორი მიმთითებელი – next (მომდევნო) და prev (წინა). ამას გარდა სიის ელემენტი შეიძლება შეიცავდეს დამატებით მონაცემებს. თუკი x სიის ელემენტია, მაშინ next[x] მიუთითებს სიის მომდევნო ელემენტზე, ხოლო prev[x] სიის წინა ელემენტზე. თუკი prev[x]=nil, მაშინ x ელემენტს არა ჰყავს წინა ელემენტი და ის წარმოადგენს სიის დასაწყისს ანუ თავს (head). თუ next[x]=nil, მაშინ x არის სიის უკანასკნელი ელემენტი და ის წარმოადგენს სიის ბოლოს (tail). ნახ. 2.4-ზე მოცემულია ორმხრივად ბმული სია. (ა)-ზე სია შეიცავს ოთხ რიცხვს 1, 4, 9, 16. სიის თითოეული ელემენტი შეიცავს სამ მინდორს: გასაღებს, მიმთითებელს წინა ელემენტზე და მიმთითებელს მომდევნო ელემენტზე.



ნახ. 2.4.

ვიდრე დავიწყებდეთ სიაში მოძრაობას მიმთითებლების საშუალებით, უნდა ვიცოდეთ სიის ერთი მაინც ელემენტი. ჩვენ ვთვლით, რომ L სიისათვის ცნობილია მიმთითებელი მის დასაწყისზე head[L]. თუკი head[L]=nil, მაშინ სია ცარიელია.

სხვადასხვა შემთხვევაში გამოიყენება სხვადასხვა სახის სია. **ცალმხრივად ბმულ** (singly linked) სიაში არა გვაქვს მინდორი prev. **დალაგებულ** (sorted) სიაში ელემენტები განლაგებულია გასაღებთა ზრდადობის მიხედვით და განსხვავებით **დაულაგებელი** (unsorted) სიისაგან სიის დასაწყისს ყოველთვის უმცირესი მნიშვნელობის გასაღები აქვს, ხოლო სიის ბოლოს – უდიდესი. წრიულ სიაში (circular list) სიის დასაწყისის prev მინდორი მიუთითებს სიის ბოლო ელემენტზე, ხოლო სიის ბოლოს next მინდორი მიუთითებს სიის დასაწყისზე.

თუკი სიის ტიპი ხაზგასმული არ არის, ტერმინი "სია" გულისხმობს ორმხრივად ბმულ დაულაგებელ სიას.

პროცედურა LIST-SEARCH(L,k) წრფივი ძებნით პოულობს პირველ ელემენტს L სიაში, რომლის გასაღებიც არის k. ის პაუხად აბრუნებს მიმთითებელს ამ ელემენტზე ან nil-ს – თუ ელემენტი ასეთი გასაღებით სიაში არ არსებობს. მაგალითად, თუკი L არის 2.4(ა)-ზე გამოსახული სია, მაშინ LIST-SEARCH(L,4) პასუხად დააბრუნებს მიმთითებელს მესამე ელემენტზე, ხოლო LIST-SEARCH(L,7) – nil-ს.

LIST-SEARCH(L,k)

1 x=head[L]

2 while x≠NIL and key[x]≠k do

3 { x=next[x] }

4 return x

n-ელემენტიან სიაში ძებნა უარეს შემთხვევაში მოითხოვს Θ (n) ოპერაციას.

პროცედურა LIST-INSERT ამატებს x ელემენტს L სიაში და ამასთან ათავსებს მას სიის თავში.

LIST-INSERT(L,x)

1 next[x]=head[L]

2 if head[L]≠NIL then { prev[head[L]]=x }

3 head[L]=x

4 prev[x]=nil

პროცედურა LIST-INSERT(L,x) სრულდება O(1) დროში სიის ზომის მიუხედავად. მისი მაგალითი მოყვანილია ნახ.2.4 (ბ)-ზე, სადაც ელემენტი გასაღებით 25 თავსდება სიის თავში.

პროცედურა LIST-DELETE შლის x ელემენტს L სიიდან და მიმთითებელს გადაამისამართებს ამ ელემენტის გვერდის ავლით. თუკი მოცემულია x ელემენტის გასაღები, მაშინ წაშლის წინ საჭიროა მოიძებნოს მისი მიმთითებელი LIST-SEARCH პროცედურის საშუალებით.

LIST-DELETE (L,x)

1 if prev[x]≠nil then { next[prev]]=next[x] }

2 else { head[L]=next[x] }

3 if next[x]]≠nil then {prev[next[x]]=prev[x]

სიიდან ელემენტის ამოშლა ნაჩვენებია ნახ.2.4 (გ)-ზე. უშუალოდ წაშლის ოპერაციას სჭირდება O(1) დრო, მაგრამ რადგან წასაშლელი ელემენტი მოცემული გასაღების მეშვეობითაა მოსაძებნი, ამიტომ პროცედურა მთლიად მოითხოვს Θ (n) დროს.

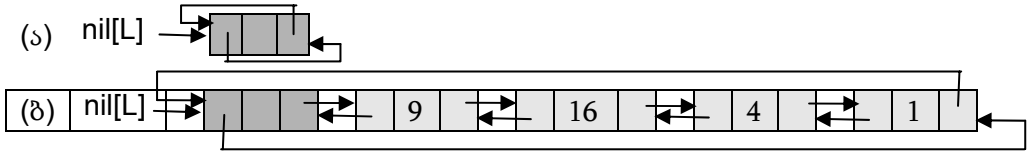
თუკი არ გავითვალისწინებთ განსაკუთრებულ სიტუაციებს სიის ბოლოში, LIST-DELETE(L,x) პროცედურა მნიშვნელოვნად გამარტივდება:

LIST-DELETE′ (L,x)

1 next[prev[x]]=next[x]

2 prev[next[x]]=prev[x]

ასეთი გამარტივება მართებული იქნება თუკი L სიას დავუმატებთ **ფიქტიურ** (sentinel) nil[L] ელემენტს, რომელსაც next და prev მინდვრები სხვა ელემენტების ანალოგიურად ექნება. ეს ელემენტი დაგვიცავს სიის გარეთ გასვლისაგან. გარდავქმნად სია წრიულ სიად: next[nil[L]] და prev[nil[L]] მინდვრებში მივუთითოთ შესაბამისად სიის დასაწყისი და სიის ბოლო, ხოლო სიის დასაწყისის prev მინდორსა და სიის ბოლოს next მინდორში შევიტანოთ მიმთითებლები nil[L] ელემენტზე. რადგან next[nil[L]] მიუთითებს სიის თავზე, head[L] ატრიბუტი ზედმეტი ხდება. ცარიელი L სია ამჟამად იქნება წრე, რომელშიც ერთადერთი nil[L] ელემენტია. ასეთი სია მოცემულია ნახ. 2.5 (ა)-ზე.



ნახ. 2.5.

LIST-SEARCH პროცედურაში საჭიროა nil შეიცვალოს nil[L]-ით, ხოლო head[L] – next[nil[L]]-ით.

LIST-SEARCH′ (L,k)

1 x= next[nil[L]]

2 while x≠nil[L] and key[x]≠k do

3 { x=next[x] }

4 return x

სიისათვის ელემენტის დამატება ასე ხორციელდება:

LIST-INSERT′ (L,x)

1 next[x]= next[nil[L]]

2 prev[next[nil[L]]]=x

3 next[nil[L]]=x

4 prev[x]=nil[L]

ფიქტიური ელემენტის გამოყენება ალგორითმის მუშაობის სიჩქარეს ვერ აუმჯობესებს, სამაგიეროდ მნიშვნელოვნად ამარტივებს პროგრამას.

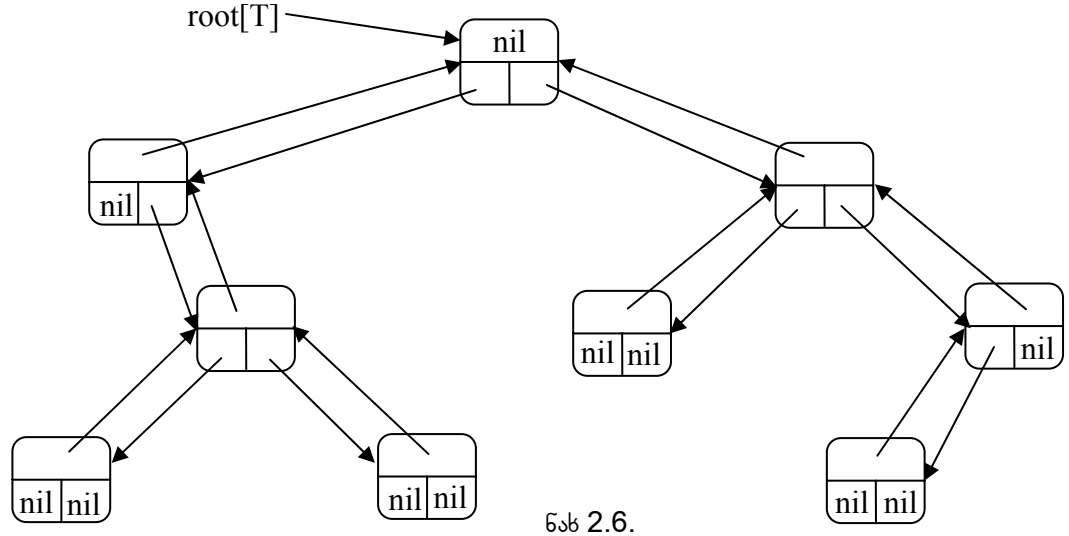
პროგრამირების ზოგიერთ ენაში (მაგ. ფორტრანი) არაა მიმთითებლების ან რამდენიმემინდვრიანი ელემენტების გამოყენების საშუალება. ასეთ შემთხვევაში იყენებენ რამდენიმე მასივს (თითოს – ჩანაწერის ყოველი მინდვრისათვის). ზემოთ მოყვანილი მაგალითისათვის უნდა შეიქმნას key, next და prev მასივები. შესაძლებელია ამ მონაცემების რეალიზაცია ერთ მასივშიც (თუკი მონაცემები მოგვცემენ საშუალებას), სადაც ერთი ელემენტის განსხვავებული მინდვრები ერთმანეთის გვერდით განლაგდებიან.

2.3. სათავის მქონე ხეების წარმოდგენა

წინა თავში აღწერილი სიის წარმოდგენის მეთოდები შეგვიძლია გამოვიყენოთ მონაცემთა სხვა სტრუქტურების მიმართ, რომლებიც შედგენილი იქნებიან ერთგვაროვანი ელემენტებით. ახლა ჩვენ განვიხილავთ მიმთითებლების გამოყენებას ხეების წარმოდგენის დროს.

ხის ყოველი წვერო იქნება რამდენიმე მინდვრის მქონე ჩანაწერი. ერთ-ერთი მინდორი სიების მსგავსად წარმოადგენს გასაღებს, ხოლო სხვა მინდვრები განკუთვნილია დამატებითი ინფორმაციის შესანახად. მათ შორის მიმთითებლები სხვა წვეროებზე. მინდვრების კონკრეტული მოწყობა დამოკიდებულია ხის ტიპზე.

ორობითი T ხის წარმოდგენისას ვიყენებთ p, left და right მინდვრებს, რომლებშიც შესაბამისად ინახება მიმთითებლები x წვეროს მშობელზე, მარცხენა შვილსა და მარჯვენა შვილზე. თუ p[x]=nil, მაშინ x ხის ძირია, თუკი x-ს არა ჰყავს მარცხენა ან მარჯვენა შვილი, შესაბამისად left[x] ან right[x] უდრის nil-ს. T ხესთან დაკავშირებულია ატრიბუტი root[T] – მიმთითებელი მის ძირზე. თუ root[T]=nil, მაშინ ხე ცარიელია. ნახ. 2.6-ზე ნაჩვენებია ორობითი ხე, სადაც თითოეულ x წვეროს აქვს მინდვრები p[x] (ზედა ნაწილი), left[x] (ქვედა მარცხენა ნაწილი) და right[x] (ქვედა მარჯვენა ნაწილი). გასაღებები სქემაზე ნაჩვენები არ არის.

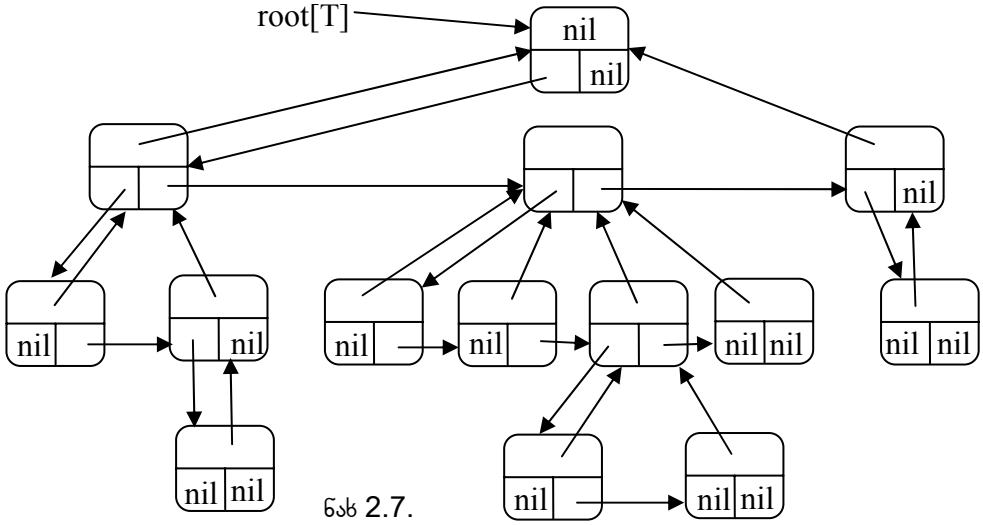


ნახ 2.6.

თუკი წინასწარ ცნობილია, რომ თითოეული წვეროს შვილების რაოდენობა ზემოდან შემოსაზღვრულია k კონსტანტით, მაშინ ასეთი ხის რეალიზაცია შესაძლებელია ორობითი ხის ანალოგიურად: left და right მინდვრების ნაცვლად მიმთითებლებს შვილებზე მოვათავსებთ მინდვრებში $child_1$, $child_2$,...,$child_k$. თუკი შვილების რაოდენობა წინასწარ უცნობია, მაშინ ასე

ვერ გავაკეთებთ – არ გვეცოდინება, თუ რამდენი მინდორი (ან მასივი – მასივების საშუალებით წარმოდგენისას) უნდა გამოვყოთ.

პრობლემა შეიძლება წარმოიშვას იმ შემთხვევაშიც, თუკი შვილების რაოდენობა შემოსაზღვრულია წინასწარ ცნობილი k რიცხვით, მაგრამ წვეროების უმეტესობას შვილები k-ზე ბევრად ნაკლები ჰყავს. ამ დროს ხდება მეხსიერების არაეკონომიური გამოყენება.



ნახ 2.7.

ამ პრობლემის გადაწყვეტა მარტივად ხდება – ნებისმიერი ხე შეიძლება გარდავქმნათ ორობითად. ყოველ წვეროს ეყოლება არაუმეტეს ორი შვილისა: მარცხენა შვილი ძველებურად დარჩება, ხოლო მარჯვენა შვილად გახდება წვერო, რომელიც წარმოადგენდა მარჯვენა მეზობელს, ანუ იმავე მშობლის უშუალოდ მომდევნო შვილს. ამ გარდაქმნის შემდეგ ორობითი ხე შეგვიძლია შევინახოთ ზემოთ აღწერილი ხერხით.

თავისუფალი განტოტების მქონე ხის ასეთი სქემით შენახვას უწოდებენ "მარცხენა შვილი – მარჯვენა მეზობელი" (left-child, right-sibling representation). ეს სქემა მოცემულია ნახ. 2.7-ზე. ყოველი წვეროსათვის ინახება p მიმთითებელი მშობელზე და root[T] ატრიბუტი წარმოადგენს მიმთითებელს ხის ძირზე. p-ს გარდა ყოველ წვეროში ინახება კიდევ ორი მიმთითებელი: left-child[x], რომელიც მიუთითებს x წვეროს ყველაზე მარცხენა შვილს და right-sibling[x], რომელიც მიუთითებს x წვეროს უახლოეს მარჯვენა მეზობელს ანუ ასაკით მომდევნო დედმამიშვილს.

x წვეროს არა ჰყავს შვილი მაშინ და მხოლოდ მაშინ, როცა left-child[x]=nil. თუ x წვერო თავისი მშობლის განაპირა მარჯვენა შვილია, მაშინ right-sibling[x]=nil.

ხის წარმოდგენა ზოგჯერ შეიძლება სხვა ხერხითაც მოხდეს. მაგალითად, გროვის რეალიზაციისას საჭირო არ არის მშობლებსა და შვილებზე მიმთითებლების შენახვა, რადგან მათი ნომრები გამოითვლება 2-ზე გაყოფითა და გამრავლებით. ზოგიერთ შემთხვევაში გვხვდებიან ხეები, რომლებზეც საჭიროა მოძრაობა ფოთლებიდან ძირისაკენ – ამ დროს შეგვიძლია არ შევინახოთ მიმთითებლები შვილებსა და მეზობლებზე. ასე, რომ ხეთა წარმოდგენის არჩევანი დამოკიდებულია ამოცანის სპეციფიურობაზე.

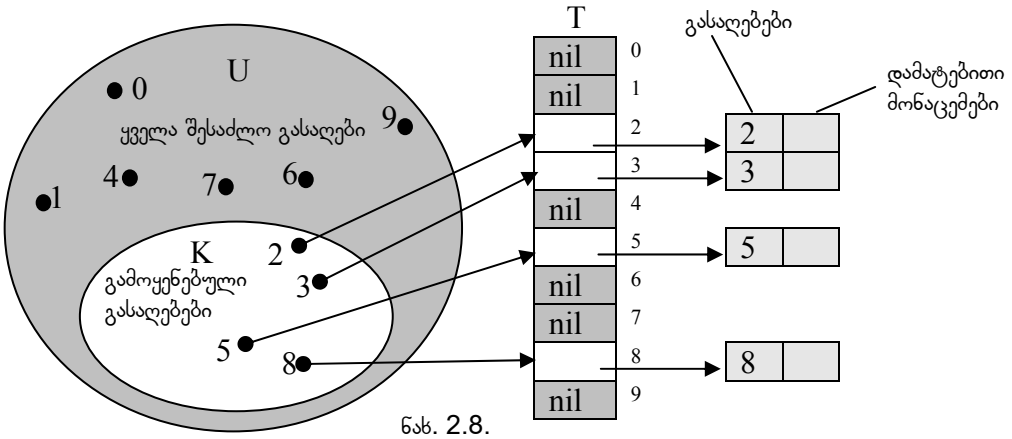
2.4. ჰეშირება. ჰეშ-ცხრილები.

პრაქტიკაში ხშირად გამოიყენება დინამიური სიმრავლეები, რომლებზეც საჭიროა შესრულდეს მხოლოდ ლექსიკონური ოპერაციები (ელემენტის დამატება, ძებნა და წაშლა). ამ შემთხვევაში ხშირად იყენებენ ე.წ. ჰეშირებას, ხოლო მონაცემთა შესაბამის სტრუქტურას უწოდებენ "ჰეშ-ცხრილს". უარეს შემთხვევაში ძებნას "ჰეშ-ცხრილში" სჭირდება იმდენივე დრო, რამდენიც სიაში ძებნას – $\Theta(n)$, მაგრამ პრაქტიკაში ჰეშირება ბევრად უფრო ეფექტურია, ხოლო გარკვეული პირობების შესრულებისას ჰეშ-ცხრილში ელემენტის ძებნის დროს მათემატიკური ლოდინია $O(1)$.

ჰეშ-ცხრილი შეიძლება განვიხილოთ როგორც ჩვეულებრივი მასივის განზოგადება. თუკი ჩვენ გაგვაჩნია საკმარისი მეხსიერება მასივისათვის, რომლის ელემენტების რაოდენობა ყველა შესაძლო გასაღების რაოდენობის ტოლია, მაშინ თითოეული შესაძლო გასაღებისათვის

შეიძლება გამოიყოს უჯრედი ამ მასივში და ამით გვექნება საშუალება თითოეულ ჩანაწერამდე მივაღწიოთ O(1) დროში. მაგრამ თუ ჩანაწერთა რეალური რაოდენობა გაცილებით ნაკლებია ყველა შესაძლო გასაღების რაოდენობაზე, მაშინ მიზანშეწონილია გამოვიყენოთ ჰეშირება: გასაღების მიხედვით გამოვთვალოთ ჩანაწერის პოზიცია მასივში.

პირდაპირი დამისამართება. პირდაპირი დამისამართება გამოიყენება მაშინ, როცა შესაძლო გასაღებთა რაოდენობა მცირეა. ვთქვათ, შესაძლო გასაღებებს წარმოადგენენ რიცხვები U={0,1,…,m-1} სიმრავლიდან (m მცირე რიცხვია). დავუშვათ ასევე, რომ ყველა ელემენტის გასაღები განსხვავებულია. სიმრავლის შესანახად გამოვიყენოთ T[0..m-1] მასივი, რომელსაც ეწოდება **პირდაპირი დამისამართების ცხრილი** (direct-address table). ყოველი **პოზიცია**, ან **უჯრედი** (position, slot) შეესაბამება გარკვეულ გასაღებს U სიმრავლიდან.



ნახ. 2.8.

ნახ. 2.8-ზე გამოსახულია პირდაპირი დამისამართების T ცხრილი. U={0,1,…9} წარმოადგენს ყველა შესაძლო გასაღების სიმრავლეს. თითოეულ გასაღებს ამ სიმრავლიდან T ცხრილში შეესაბამება საკუთარი უჯრედი. ცხრილის 2, 3, 5 და 8 ნომრის შესაბამის პოზიციებში (ფაქტობრივად გამოყენებული გასაღებები) ჩაწერილია მიმთითებლები სიმრავლის ელემენტებზე, ხოლო ცხრილის გამოუყენებელ პოზიციებში (მუქი ფერითაა მოცემული) ჩაწერილია nil.

აღვნიშნოთ T[k]-თი k გასაღების მქონე ელემენტზე მიმთითებლის ჩაწერის ადგილი. თუ k გასაღების მქონე ელემენტი ცხრილში არა გვაქვს, მაში T[k]=nil. ლექსიკონური ოპერაციების რეალიზება ასე მოხდება:

```
DIRECT-ADDRESS-SEARCH(T,k)
return T[k]

DIRECT-ADDRESS-INSERT(T,x)
return T[key[x]]=x

DIRECT-ADDRESS-DELETE(T,x)
return T[key[x]]=nil
```

თითოეული ამ ოპერაციიდან საჭიროებს O(1) დროს.

ზოგჯერ შესაძლებელია მეხსიერების ეკონომია T ცხრილში არა მიმთითებლების, არამედ თავად ელემენტების ჩაწერით. შესაძლოა არ გამოვიყენოთ გასაღების მინდორიც და გასაღების როლი შეასრულოს მასივის ინდექსმა. ასეთ შემთხვევაში (როცა მიმთითებლებს და გასაღებებს არ ვიყენებთ), უნდა გვქონდეს საშუალება აღვნიშნოთ, რომ კონკრეტული პოზიცია თავისუფალია.

ჰეშ-ცხრილები. პირდაპირ დამისამართებას აშკარა ნაკლი გააჩნია: თუკი ყველა შესაძლო გასაღებთა U სიმრავლე დიდია, |U| ზომის მქონე T მასივის შენახვა მეხსიერებაში არაპრაქტიკულია, ხოლო ზოგჯერ – შეუძლებელიც. გარდა ამისა, თუკი ცხრილში რეალურად არსებული ჩანაწერები მცირეა |U|-სთან შედარებით, მაშინ მეხსიერების დიდი ნაწილი უსარგებლოდ იხარჯება.

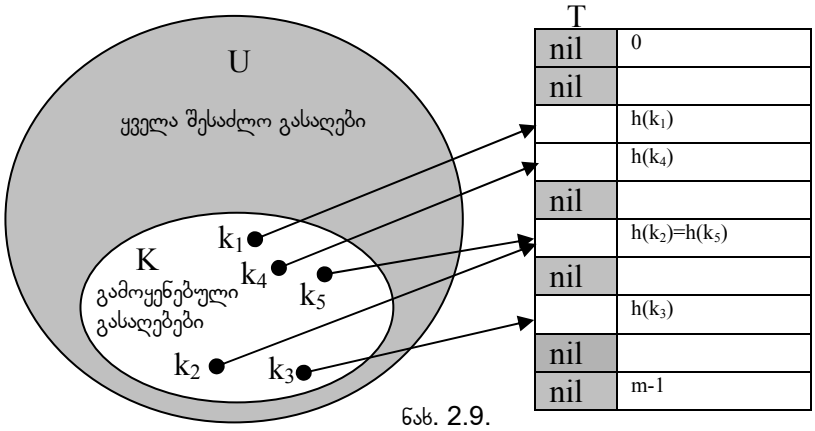
თუ ცხრილში არსებული ჩანაწერები მნიშვნელოვნად მცირეა ყველა შესაძლო გასაღებთან შედარებით, მაშინ ჰეშ-ცხრილი იკავებს გაცილებით მცირე ადგილს, ვიდრე პირდაპირი

დამისამართების ცხრილი. სახელდობრ, ჰეშ-ცხრილი იკავებს $\Theta(|K|)$ მოცულობის მეხსიერებას, სადაც K – ჩანაწერთა სიმრავლეა. ამასთან ძებნის დროდ ჰეშ-ცხრილში რჩება ისევ O(1), იმ განსხვავებით, რომ ამჯერად ეს საშუალო შეფასებაა (ისიც გარკვეული წინაპირობებით) და არა შეფასება უარეს შემთხვევაში.

თუკი პირდაპირი დამისამართების დროს k გასაღების მქონე ელემენტისათვის გამოიყოფოდა k ნომრის მქონე პოზიცია, ჰეშირების დროს ეს ელემენტი ჩაიწერება h(k) ნომრის მქონე პოზიციაზე T[0..m-1] **ჰეშ-ცხრილში** (hash table), სადაც

$h:U\rightarrow\{0,1,\ldots,m-1\}$

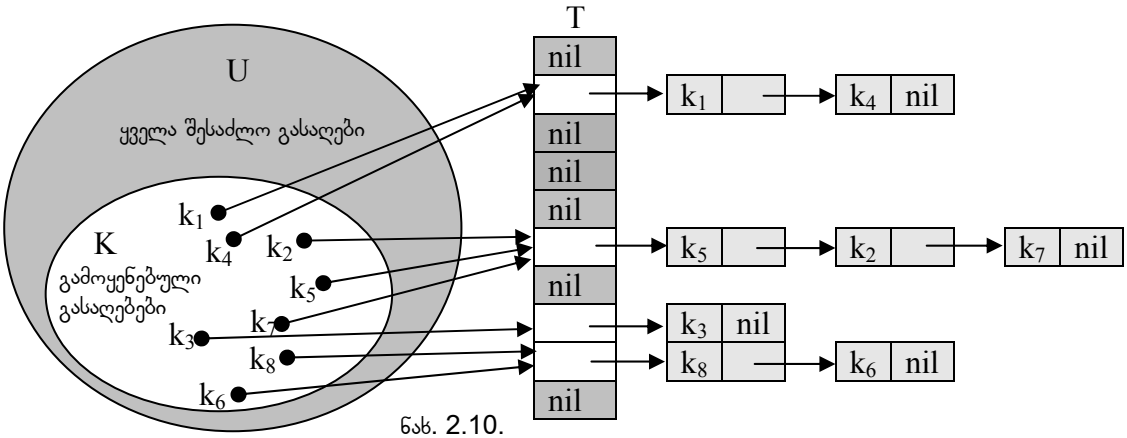
ჰეშ-ფუნქციად (hash function) წოდებული ვექტორული ფუნქციაა. h(k) რიცხვს უწოდებენ k გასაღების **ჰეშ-მნიშვნელობას** (hash value). ჰეშირების იდეა ნაჩვენებია ნახ.2.9-ზე, სადაც ვიყენებთ არა $|U|$, არამედ m სიგრძის მასივს და ამით ეკონომიას ვუკეთებთ მეხსიერებას.



ნახ. 2.9.

ჰეშირების პრობლემა იმაში მდგომარეობს, რომ ორი განსხვავებული გასაღების ჰეშ-მნიშვნელობა შეიძლება დაემთხვეს. ასეთ შემთხვევაში იტყვიან, რომ მოხდა **კოლიზია** ანუ **შეჯახება** (collision). საბედნიეროდ ეს პრობლემა მოგვარებადია და ჰეშ-ფუნქციის გამოყენება შესაძლოა კოლიზიის დროსაც.

სასურველია ჰეშ-ფუნქცია ისე შეირჩეს, რომ კოლიზია თავიდან ავიცილოთ, მაგრამ თუ $|U|>m$, მაშინ აუცილებლად იარსებებენ განსხვავებული გასაღებები, რომელთაც ერთნაირი ჰეშ-მნიშვნელობები ექნებათ. ზემოთ თქმულიდან გამომდინარე, კოლიზიების დასამუშავებლად ყოველთვის მზად უნდა ვიყოთ.



ნახ. 2.10.

ჰეშ-ფუნქციის შერჩევისას ყოველთვის არ ვიცით, თუ რომელი გასაღებები იქნება შენახული. გარკვეული აზრით აჯობებს, რომ ჰეშ-ფუნქცია იყოს შემთხვევითი, რათა კარგად "გააბნიოს" გასაღებები უჯრედებში. იგულისხმება, რომ შემთხვევითი ჰეშ-ფუნქციაა იმდენად იქნება დეტერმინირებული, რომ განმეორებითი გამოძახებების დროსაც ერთ და იგივე არგუმენტისათვის ერთნაირ ჰეშ-მნიშვნელობას მიიღებს.

კოლიზიის პრობლემის გადაჭრის ერთ-ერთი გზაა **ელემენტთა გადაბმა** (chaining). ვუწოდოთ ამ მეთოდს ჰეშირება ჯაჭვებით. მისი არსი იმაში მდგომარეობს, ერთნაირი ჰეშ-მნიშვნელობის მქონე ელემენტებისაგან ყალიბდება ბმული სია (იხ. ნახ. 2.10). j-ურ პოზიციაში

ინახება მიმთითებელი იმ ელემენტთა სიაზე, რომელთა გასაღებების ჰეშ-მნიშვნელობა j-ს ტოლია. თუკი ასეთი ელემენტები არ არსებობენ, j-ურ პოზიციაში იწერება nil.

დამატების, ძებნისა და წაშლის ოპერაციები იოლად რეალიზდება:

CHAINED-HASH-INSERT(T,x)

დავუმატოთ x ელემენტი T[h(key[x])] სიის დასაწყისში

CHAINED-HASH-SEARCH(T,k)

მოვძებნოთ k გასაღების მქონე ელემენტი T[h(k)] სიაში

CHAINED-HASH-DELETE(T,x)

წავშალოთ x ელემენტი T[h(key[x])] სიიდან

დამატების ოპერაცია უარეს შემთხვევაში მუშაობს O(1) დროში. ძებნის ოპერაციის მუშაობის დრო სიის სიგრძის პროპორციულია. ელემენტის წაშლა შეიძლება განხორციელდეს O(1) დროში იმ პირობით, რომ სიები ორმხრივად ბმულია. თუკი სიები ცალმხრივადაა ბმული, მაშინ x ელემენტის წასაშლელად საჭიროა წინასწარ ვიპოვოთ მის წინ მდგომი ელემენტი, რასაც ძებნის ოპერაციის ჩატარება სჭირდება და ამ შემთხვევაში ძებნისა და წაშლის ოპერაციებს თითქმის ერთნაირი დრო დასჭირდებათ სამუშაოდ.

შევაფასოთ ოპერაციათა მუშაობის დრო ჯაჭვებით ჰეშირებისას. ვთქვათ T არის m პოზიციის მქონე ჰეშ-ცხრილი, რომელშიც შეტანილია n ელემენტი. ცხრილის **შევსების კოეფიციენტი** (load factor) ეწოდება $\alpha=n/m$ რიცხვს (ეს რიცხვი შეიძლება იყოს 1-ზე მეტიც და 1-ზე ნაკლებიც). გამოვსახოთ ოპერაციათა ღირებულება α-ს საშუალებით.

უარეს შემთხვევაში ყველა n გასაღების ჰეშ-მნიშვნელობა შეიძლება ერთმანეთს დაემთხვეს, მაშინ ცხრილი წარმოდგენილი იქნება n სიგრძის მქონე ერთი სიით და ელემენტის ძებნაზე დაიხარჯება იგივე $\Theta(n)$ დრო, რაც დაიხარჯებოდა სიაში ძებნისას. ამას დაემატება კიდევ ჰეშ-ფუნქციის გამოთვლის დრო. ცხადია, რომ ასეთ შემთხვევაში ჰეშირებას აზრი არა აქვს.

ძებნის საშუალო ღირებულება დამოკიდებულია იმაზე, თუ რამდენად თანაბრად გაანაწილებს ჰეშ-ფუნქცია ჰეშ-მნიშვნელობებს ცხრილის პოზიციებში. თანაბრად განაწილების მიღწევის საშუალებებს ქვემოთ შევეხებით, ამჯერად პირობითად დავუშვათ, რომ თითოეულ ელემენტს თანაბარი ალბათობით შეუძლია მოხვდეს ცხრილის ნებისმიერ m პოზიციაში და არაა დამოკიდებული იმაზე, თუ რა პოზიციაში მოხვდა სხვა ელემენტი. ამ დაშვებას ვუწოდოთ **თანაბარი ჰეშირების** (simple uniform hashing) ჰიპოთეზა.

ჩავთვალოთ, რომ მოცემული k გასაღებისათვის h(k) ჰეშ-მნიშვნელობის გამოთვლა, სიაში ბიჯი და გასაღებთა შედარება საჭიროებენ ფიქსირებულ დროს. მაშინ k გასაღების მქონე ელემენტის ძებნის დრო წრფივადაა დამოკიდებული T[h(k)] სიაში ელემენტების რაოდენობაზე. განვასხვავოთ ორი შემთხვევა: ა) ძებნა წარუმატებლად დასრულდა და k გასაღების მქონე ელემენტი ვერ ვიპოვეთ; ბ) საძებნი ელემენტი ნაპოვნია. ადგილი აქვს შემდეგ თეორემებს:

თეორემა 2.1. ვთქვათ T ჯაჭვების შემცველი ჰეშ-ცხრილია, რომლის შევსების კოეფიციენტია α. დავუშვათ, რომ ჰეშირება თანაბარია. მაშინ ცხრილში არარსებული ელემენტის ძებნისას საშუალოდ განხილული იქნება ცხრილის α ელემენტი, ხოლო ასეთი ძებნის საშუალო დრო (ჰეშ-ფუნქციის გამოთვლის დროის ჩათვლით) იქნება $\Theta(1+\alpha)$.

თეორემა 2.2. თანაბარი ჰეშირებისას ჯაჭვების შემცველ ჰეშ-ცხრილში, წარმატებული ძებნის საშუალო დროა $\Theta(1+\alpha)$.,სადაც α შევსების კოეფიციენტია.

<u>ჰეშ-ფუნქციები.</u> განვიხილოთ ჰეშ-ფუნქციის აგების სამი მეთოდი: ნაშთიანი გაყოფა, გამრავლება და უნივერსალური ჰეშირება. პრაქტიკაში მათი გამოყენება დამოკიდებულია ამოცანის სპეციფიკაზე. ზოგადად, კი ჰეშ-ფუნქციისაგან მოითხოვება, რომ იგი უზრუნველყოფდეს თანაბარ ჰეშირებას. ჩვეულებისამებრ, გულისხმობენ, რომ ჰეშ-ფუნქციების განსაზღვრის არე მთელი არაუარყოფითი რიცხვებია. თუკი გასაღებები არ წარმოადგენენ ნატურალურ რიცხვებს, მათ გარდაქმნიან ასეთ სახემდე (თუმცა შეიძლება დიდი რიცხვები მივიღოთ). მაგალითად, სიმბოლოთა მიმდევრობა შეიძლება ინტერპრეტირებულ იქნას

როგორც რიცხვები, რომლებიც ჩაწერილი არიან შესაბამისი ფუძის მქონე თვლის სისტემაში. იდენტიფიკატორი pt წარმოადგენს რიცხვთა წყვილს (112,116) (ასეთია p და t სიმბოლოების ASCII კოდები), ან ფუძედ 128-ის მქონე თვლის სისტემაში ჩაწერილ რიცხვს – 112×128+116=14452. შემდგომში ჩვენ ყოველთვის ვიგულისხმებთ, რომ გასაღებები მთელ არაუარყოფით რიცხვებს წარმოადგენენ.

ჰეშ-ფუნქციის აგება **ნაშთიანი გაყოფის მეთოდით** (division method) მდგომარეობს იმაში, რომ k სიდიდის მქონე გასაღებს ეთანადება k-ს m-ზე გაყოფის ნაშთი, სადაც m წარმოადგენს შესაძლო ჰეშ-მნიშვნელობების რაოდენობას:

h(k)= k mod m

მაგალითად, თუკი ჰეშ-ცხრილის ზომა m=12-ის ტოლია და გასაღები უდრის 100-ს, მაშინ შესაბამისი ჰეშ-მნიშვნელობა იქნება 4.

m-ის გარკვეულ მნიშვნელობებს თავი უნდა ავარიდოთ. მაგალითად თუ $m=2^p$, მაშინ h(k) წარმოადგენს k რიცხვის p უმცროს ბიტებს. თუკი დარწმუნებული არა ვართ, რომ გასაღებთა უმცროსი ბიტები ერთნაირი სიხშირით შეგვხვდება, m რიცხვის როლში ორის ხარისხებს არ ირჩევენ. ასევე არასასურველია m-ის ადგილას 10-ის ხარისხების არჩევა, თუკი გასაღებები ათობით რიცხვებს წარმოადგენენ – ასეთ შემთხვევაში აღმოჩნდება, რომ გასაღებთა ციფრების ნაწილი მთლიანად განსაზღვრავს ჰეშ-მნიშვნელობებს. თუკი გასაღებები წარმოადგენენ რიცხვებს 2^p ფუძის მქონე თვლის სისტემაში, მაშინ არაეფექტურია $m=2^p-1$ მნიშვნელობის აღება, რადგან ამ შემთხვევაში ერთნაირ ჰეშ-მნიშვნელობებს მიიღებენ გასაღებები, რომლებიც განსხვავდებიან მხოლოდ თვლის ამ სისტემაში არსებული ციფრების გადანაცვლებით.

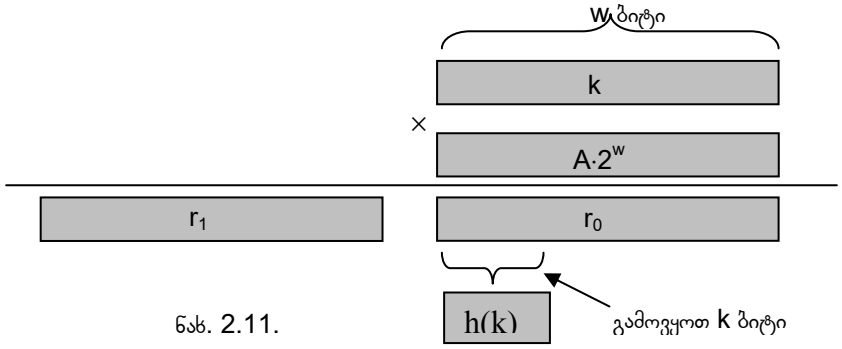
კარგ შედეგებს იძლევა m-ის როლში ისეთი მარტივი რიცხვის აღება, რომელიც შორს დგას 2-ის ხარისხებისაგან. მაგალითად, თუ ჰეშ-ცხრილში შესატანია 2000-მდე ჩანაწერი და დაახლოებით სამი ვარიანტის გადარჩევა პრობლემას არ ჰქმნის ელემენტთა ძებნისას, m-ის მნიშვნელობად შეგვიძლია ავიღოთ 701. რიცხვი 701 მარტივია, 701≈2000/3 და ორის ხარისხებისგანაც შორს დგას. ყოველივე ამის გამო მიზანშეწონილია ავირჩიოთ ჰეშ-ფუნქცია h(k)=k mod 701.

ჰეშ-ფუნქციის აგება **გამრავლების მეთოდით** (multiplication method) შემდეგში – ვთქვათ ჰეშ-მნიშვნელობების რაოდენობა m-ის ტოლია. დავაფიქსიროთ A მუდმივა 0<A<1 შუალედში და განვსაზღვროთ:

$$h(k)=\lfloor m\times(kA \bmod 1)\rfloor$$

სადაც (kA mod 1) წარმოადგენს kA-ს წილად ნაწილს.

გამრავლების მეთოდის ღირსება იმაში მდგომარეობს, რომ ჰეშ-ფუნქციის ეფექტურობა ნაკლებადაა დამოკიდებული m-ის არჩევაზე. ჩვეულებისამებრ m-ის როლში ირჩევენ ორის ხარისხს, რადგან კომპიუტერთა უმეტესობაში ასეთ m-ზე გამრავლება რეალიზდება როგორც სიტყვის წანაცვლება. მაგალითად, თუ სიტყვის სიგრძე თქვენს კომპიუტერში w ბიტის ტოლია, k გასაღები თავსდება ერთ სიტყვაში და $m=2^p$, მაშინ ჰეშ-ფუნქციის გამოთვლა ასე შეიძლება ჩატარდეს: k გავამრავლოთ w-ბიტიან მთელ $A\cdot 2^w$ რიცხვზე. მივიღებთ 2w ბიტი სიგრძის მქონე სიტყვას. r_0-ით აღვნიშნოთ უმცროსი w თანრიგით განსაზღვრული რიცხვი. r_0-ის უფროსი p ბიტი წარმოადგენს საძებნ ჰეშ-მნიშვნელობას (იხ. ნახ. 2.11.).



ნახ. 2.11.

გამრავლების მეთოდი მუშაობს A მუდმივის ნებისმიერი მნიშვნელობისათვის, მაგრამ ზოგიერთმა მნიშვნელობამ შეიძლება უკეთესი შედეგი მოგვცეს. დონალდ კნუტის აზრით

$$A \approx \left(\sqrt{5}-1\right)/2 = 0.6180339887 \qquad (2.1)$$

მნიშვნელობა საკმაოდ ეფექტურია.

მოვიყვანოთ მაგალითი: ვთქვათ, k=123456, m=10000 და A განსაზღვრულია (2.1) ფორმულით, მაშინ

h(k) = ⌊10000·(123456·0.61803… mod 1⌋ = ⌊10000·(76300.0041151… mod 1⌋ =

= ⌊10000·0.0041151…⌋=)=⌊41.151…⌋=41.

თუკი ჰეშ-ფუნქცია ცნობილია, მაშინ, რა თქმა უნდა, ყოველთვის შესაძლებელია ისეთი მონაცემების შერჩევა, რომ ყველა n გასაღები შეესაბამებოდეს ჰეშ-ცხრილის ერთ პოზიციას, რის გამოც ძებნის დრო გახდება Θ(n)-ის ტოლი. ასეთი გზით ნებისმიერი ფიქსირებული ჰეშ-ფუნქციის დისკრედიტირება შეიძლება. ერთადერთი გამოსავალი ამ სიტუაციიდან არის ის, რომ ჰეშ-ფუნქცია შევარჩიოთ შემთხვევითად, იმის მიუხედავად თუ რა სახის მონაცემებზეა ჰეშირება ჩასატარებელი. ასეთ მეთოდს უწოდებენ **უნივერსალურ ჰეშირებას** (universal lashing). უნივერსალური ჰეშირების ძირითადი იდეაა, შევარჩიოთ ჰეშ-ფუნქცია შემთხვევითად რაღაც სიმრავლიდან პროგრამის შესრულების დროს. ცხადია, რომ პროგრამის იმავე მონაცემებით ხელმეორედ გაშვებისას ჰეშირება სხვაგვარად მოხდება. შემთხვევითი ფუნქციის გამოყენება გარანტიას იძლევა, რომ შეუძლებელია ისეთი საწყისი მონაცემების მოფიქრება, რომ ალგორითმმა ყოველთვის ნელა იმუშაოს.

ვთქვათ H წარმოადგენს იმ ფუნქციათა სასრულ ოჯახს, რომლებიც მოცემულ U სიმრავლეს (ყველა შესაძლო გასაღებთა სიმრავლე) ასახავს {0,1,...,m-1) სიმრავლეზე (ჰეშ-მნიშვნელობათა სიმრავლე). ოჯახს ეწოდება უნივერსალური (universal), თუკი ნებისმიერ ორი x,y∈U გასაღებისათვის იმ h∈H ფუნქციათა რიცხვი, რომლებისთვისაც h(x)=h(y), ტოლია |H|/m-ის. სხვა სიტყვებით რომ ვთქვათ, ჰეშ-ფუნქციის შემთხვევითად არჩევისას კოლიზიის ალბათობა ორი მოცემული გასაღებისათვის ტოლია შემთხვევითად არჩეული ორი ჰეშ-მნიშვნელობის დამთხვევის ალბათობისა (რაც თავის მხრივ უდრის 1/m).

ღია დამისამართება. ჯაჭვური ჰეშირებისაგან განსხვავებით **ღია დამისამართების** (open addressing) დროს სიები არ გამოიყენება და ყველა ჩანაწერი ინახება თავად ჰეშ-ცხრილში. ჰეშ-ცხრილის ყოველი უჯრედი შეიცავს ან დინამიური სიმრავლის ელემენტს ან nil-ს. ძებნა მდგომარეობს იმაში, ჩვენ გარკვეული წესით ვამოწმებთ ცხრილის ელემენტებს, ვიდრე საჭიროს არ ვიპოვით ან არ დავრწმუნდებით მის არარსებობაში. ამასთან შესანახ ელემენტთა რაოდენობა არ უნდა იყოს ცხრილის ზომაზე მეტი ანუ შევსების კოეფიციენტი 1-ზე მეტი არ უნდა იყოს.

ჯაჭვური ჰეშირების დროსაც შესაძლებელია ჰეშ-ცხრილის თავისუფალი ადგილების გამოყენება სიების შესანახად, მაგრამ ღია დამისამართების დროს მიმთითებლები საერთოდ არ გამოიყენებიან – განსახილველ უჯრედთა მიმდევრობა გამოითვლება. მიმთითებლების გამოუყენებლობით ეკონომირებული მეხსიერების ხარჯზე შეგვიძლია გავზარდოთ პოზიციათა რაოდენობა ცხრილში, რაც ამცირებს კოლიზიებს და აჩქარებს ძებნას.

ახალი ელემენტის ჩამატებისას ღია დამისამართების ცხრილში, რომელიც გადანომრილია 0-დან (m-1)-მდე, ჩვენ განვიხილავთ მას თავისუფალი ადგილის პოვნამდე. თუკი ყოველ ჯერზე მოგვიწევს თავიდან ბოლომდე მთელი ცხრილის შემოწმება, დაიხარჯება Θ(n) დრო, მაგრამ მეთოდის არსი იმაშია, რომ ცხრილის განხილვის რიგი დამოკიდებულია გასაღებზე. კერძოდ ძებნა შეიძლება დაიწყოს ცხრილის ნებისმიერი ადგილიდან გასაღების მნიშვნელობის მიხედვით და შეწყდეს რამდენიმე შემოწმების შემდეგ. სხვა სიტყვებით რომ ვთქვათ, ჰეშ-ფუნქციას ემატება მეორე არგუმენტი – ცდის ნომერი (გადანომვრა ნულიდან იწყება). ასე რომ ჰეშ-ფუნქციას აქვს სახე:

$$h: U \times \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots,m-1)$$

სადაც U გასაღებთა სიმრავლეა. **გამოყენებულ ადგილთა მიმდევრობას** ანუ **ცდათა მიმდევრობას**(probe sequence) მოცემული k გასაღებისათვის აქვს სახე:

$$\langle h(k,0),\ h(k,1),\ldots,h(k,m-1)\rangle;$$

h ფუნქცია ისეთი უნდა იყოს, რომ ამ მიმდევრობაში 0-დან (m-1)-მდე ყველა რიცხვი ზუსტად ერთხელ შეგვხვდეს (ყოველი გასაღებისათვის ცხრილის ნებისმიერი პოზიცია მისაწვდომი უნდა იყოს). ქვემოთ მოყვანილია ღია დამისამართების ცხრილში ელემენტის დამატების პროცედურა. მასში იგულისხმება, რომ ჩანაწერებს გასაღების გარდა არა აქვთ დამატებითი ინფორმაცია. თუ ცხრილის უჯრედი ცარიელია, მასში ჩაწერილია nil.

```
HASH-INSERT(T,k)
1 i=0
2 repeat j=h(k,i) {
3         if T[j]=nil then {  T[j]=k;  return j    }
4                   else {  i=i+1   }
5 until i=m     }
6 error "ჰეშ-ცხრილის გადავსება"
```

ღია დამისამართების ცხრილში k გასაღების მქონე ელემენტის ძებნისას ცხრილის უჯრედები განიხილება იმავე მიმდევრობით, როგორც იმავე გასაღების მქონე ელემენტის დამატებისას. თუკი ამ განხილვისას ჩვენ წავაწყდით უჯრედს, რომელშიც nil წერია, შეგვიძლია დავასკვნათ, რომ საძებნი ელემენტი ცხრილში არაა – წინააღმდეგ შემთხვევაში ის შეტანილი იქნებოდა ამ უჯრედში. აქვე განსაკუთრებით უნდა გავუსვათ ხაზი იმ გარემოებას, რომ არანაირი ელემენტი ცხრილიდან არ იშლება.

ქვემოთ მოყვანილია ღია დამისამართების ცხრილში ელემენტის ძებნის პროცედურა. თუ k გასაღების მქონე ელემენტი ჩაწერილია T ცხრილის j-ურ პოზიციაში, პროცედურა პასუხად აბრუნებს j-ს, წინააღმდეგ შემთხვევაში – nil-ს.

```
HASH-SEARCH(T,k)
1 i=0
2 repeat j=h(k,i) {
3         if T[j]=k then {  return j    }
4         i=i+1
5 until (T[j]=nil) OR (i=m)     }
6 return nil
```

ღია დამისამართების ცხრილიდან ელემენტის წაშლა მარტივი საქმე არ არის. თუკი წასაშლელი ელემენტის ნაცვლად ცხრილში უბრალოდ nil-ს ჩავწერთ, მაშინ ვეღარ ვიპოვით ელემენტებს, რომელთა ცხრილში დამატებისას ეს ადგილი შევსებული იყო. გამოსავალი ამ სიტუაციიდან შეიძლება ასე ვიპოვოთ: წაშლილი ელემენტის ადგილას ჩავწეროთ არა nil, არამედ რაიმე სპეციალური მნიშვნელობა – deleted ("წაშლილია"). ელემენტის დამატების დროს ეს უჯრა განვიხილოთ როგორც თავისუფალი, ხოლო ძებნის დროს – როგორც დაკავებული და გავაგრძელოთ ძებნა. ამ მიდგომის ნაკლი ისაა, რომ ძებნის დრო შეიძლება დიდი იყოს შევსების დაბალი კოეფიციენტის დროსაც. ამიტომ როცა საჭიროა ჰეშ-ცხრილიდან ჩანაწერების წაშლა, უპირატესობას ჯაჭვებით ჰეშირებას ანიჭებენ.

ღია დამისამართების ცხრილში დაკავებული ადგილების გამოთვლისათვის გამოიყენებენ სამ მეთოდს: წრფივი, კვადრატული და ორმაგი ჰეშირება.

ვთქვათ h': U $\rightarrow$ {0,1,…,m-1} ჩვეულებრივი ჰეშ-ფუნქციაა. **ცდათა წრფივი მიმდევრობის** (linear probing) განმსაზღვრელი ფუნქცია მოიცემა ფორმულით:

$$h(k,i)=(h'(k)+i) \bmod m.$$

სხვა სიტყვებით რომ ვთქვათ, k გასაღების დამუშავება იწყება $T[h'(k)]$ უჯრედიდან, ხოლო შემდეგ ცხრილის უჯრედები განიხილება თანმიმდევრობით: $T[h'(k)+1]$, $T[h'(k)+2]$, ... ($T[m-1]$-ის შემდეგ განიხილება $T[0]$). ცდათა თანმიმდევრობა მთლიანად განისაზღვრება პირველი უჯრედით, ამიტომ რეალურად განიხილება მხოლოდ m განსხვავებული მიმდევრობა.

ცდათა წრფივი მიმდევრობით ღია დამისამართების რეალიზაცია იოლია, მაგრამ ამ მეთოდს აქვს ერთი ნაკლი: მან შეიძლება წარმოქმნას **კლასტერები** (primary clustering – პირველადი კლასტერი), ანუ მიყოლებით განლაგებული დაკავებული უჯრედების გრძელი მიმდევრობები, ეს კი ართულებს ძებნას. თუ m უჯრედისაგან შემდგარ ცხრილში ლუწნომრიანი უჯრედები დაკავებულია, ხოლო კენტნომრიანი – თავისუფალი, მაშინ ცდათა

საშუალო რაოდენობა ცხრილში არარსებული ელემენტის ძებნისას არის 1,5. თუკი იგივე დაკავებული m/2 უჯრედი განლაგებულია მიყოლებით, მაშინ ცდათა საშუალო რიცხვი დაახლოებით უდრის m/8=n/4 (n – დაკავებული უჯრედების რაოდენობა ცხრილში). კლასტერების წარმოქმნის ტენდენცია მარტივად აიხსნება: თუკი n დაკავებული უჯრედი განლაგებულია მიყოლებით, იმის ალბათობა, რომ ცხრილში მორიგი ჩამატებისას დაკავებული უჯრედი მოთავსებული იქნება უშუალოდ ამ n უჯრედის შემდეგ, წარმოადგენს (n+1)/m, ხოლო გამოყენების ალბათობა ისეთი თავისუფალი უჯრედისა, რომელიც ასევე თავისუფალი უჯრედის შემდეგაა მოთავსებული წარმოადგენს 1/m. ზემოთ თქმულიდან ნათლად ჩანს, რომ ცდათა წრფივი მიმდევრობა შორს დგას თანაბარი ჰეშირებისაგან.

ცდათა კვადრატული მიმდევრობის (quadratic probing) განმსაზღვრელი ფუნქცია მოიცემა ფორმულით:

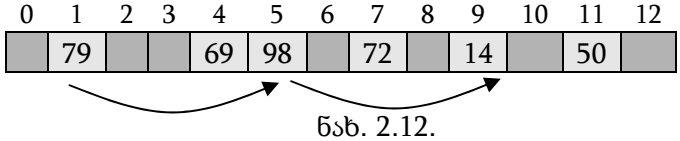
$$h(k,i)=(h'(k)+c_1 i+c_2 i^2) \bmod m.$$

სადაც h' – ჩვეულებრივი ჰეშ-ფუნქციაა, ხოლო c_1 და $c_2 \neq 0$ – მუდმივები. წრფივი მეთოდის მსგავსად ცდები იწყება $T[h'(k)]$ უჯრედიდან, მაგრამ არ ხდება უჯრედების განხილვა მიყოლებით – განსახილველი უჯრედის ნომერი კვადრატულადაა დამოკიდებული ცდის ნომერზე. ეს მეთოდი გაცილებით უკეთესად მუშაობს ვიდრე წრფივი მეთოდი, მაგრამ თუ გვინდა, რომ ჰეშ-ცხრილში ყველა უჯრედი იქნას განხილული, c_1 და c_2-ის შერჩევა ნებისმიერად არ შეიძლება. წრფივი მეთოდის მსგავსად ცდათა მიმდევრობა აქაც განისაზღვრება მისი პირველი წევრით. კლასტერების წარმოქმნის საშიშროება აღარ არის, თუმცა ანალოგიური ეფექტი შეიძლება უფრო მსუბუქი ფორმით გამოვლინდეს **მეორადი კლასტერების** (secondary clastering) წარმოქმნაში.

ღია დამისამართების ერთ-ერთ საუკეთესო მეთოდს წარმოადგენს **ორმაგი ჰეშირება** (double hashing). ორმაგი ჰეშირებისას წარმოქმნილი ინდექსთა გადანაცვლებები ფლობენ თანაბარი ჰეშირების ბევრ თვისებას. ორმაგი ჰეშირებისას h ფუნქციას აქვს სახე

$$h(k,i)=(h_1(k)+i\cdot h_2(k)) \bmod m,$$

სადაც h_1 და h_2 – ჩვეულებრივი ჰეშ-ფუნქციებია. სხვაგვარად რომ ვთქვათ, k გასაღებთან მუშაობისას ცდათა მიმდევრობა წარმოადგენს არითმეტიკულ პროგრესიას (მოდულით m), რომლის პირველი წევრია $h_1(k)$ და ბიჯი – $h_2(k)$. ორმაგი ჰეშირების მაგალითი ნაჩვენებია ნახ. 2.12-ზე, სადაც m=13, $h_1(k)=k \bmod 13$ და $h_2(k)=1+(k \bmod 11)$. ვთქვათ k=14, მაშინ ცდათა მიმდევრობა იქნება: 1 და 5 დაკავებულია, 9 თავისუფალია, ამიტომ 14 ჩაიწერება მე-9 უჯრედში.



ნახ. 2.12.

დაკავებულ ადგილთა მიმდევრობამ ცხრილი მთლიანად რომ დაფაროს $h_2(k)$-ს მნიშვნელობა და m ურთიერთმარტივები უნდა იყვნენ (თუკი $h_2(k)$-სა და m-ის უდიდესი საერთო გამყოფია d, მაშინ $h_2(k)$ ბიჯის მქონე არითმეტიკული პროგრესია მოდულით m დაიკავებს ცხრილის 1/d ნაწილს). ამ პირობის შესრულება მარტივად შეიძლება – m-ის როლში ავიღოთ 2-ის ხარისხი, ხოლო h_2 ფუნქცია ისე შევარჩიოთ, რომ მან მხოლოდ კენტი მნიშვნელობები მიიღოს. მეორე ვარიანტში შეიძლება m ავიღოთ მარტივი რიცხვი, ხოლო h_2-ის მნიშვნელობა – m-ზე ნაკლები მთელი დადებითი რიცხვები. მაგალითად, მარტივი m-სათვის ავიღოთ

$$h_1(k)=k \bmod m$$
$$h_2(k)=1+(k \bmod m')$$

სადაც m მცირედით ნაკლები m'-ზე (ვთქვათ $m'=m-1$ ან $m'=m-2$). თუ m=701, $m'=700$ და k=123456, მაშინ $h_1(k)=80$ და $h_2(k)=257$. მაშასადამე, ცდათა მიმდევრობა დაიწყება მე-80 პოზიციიდან და გაგრძელდება ბიჯით 257, ვიდრე ცხრილი მთლიანად არ იქნება განხილული.

განსხვავებით წრფივი და კვადრატული მეთოდებისაგან ორმაგი ჰეშირებისას შეიძლება მივიღოთ არა m, არამედ $\Theta(m^2)$ გადანაცვლება (თუკი h_1 და h_2 სწორად იქნება შერჩეული),

რადგან ყოველ ($h_1(k)$, $h_2(k)$) წყვილს შეესაბამება ცდათა საკუთარი მიმდევრობა. ყოველივე ამის გამო ორმაგი ჰეშირების ეფექტურობა უახლოვდება თანაბარ ჰეშირებას.

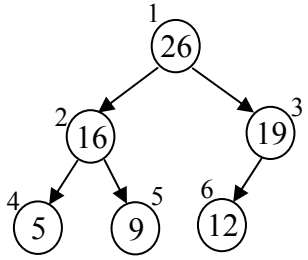
გროვა

გროვას (Heap) უწოდებენ ორობით ხეს, რომლის ნებისმიერ კვანძში შვილის მნიშვნელობა არ აღემატება მშობლის მნიშვნელობას, ანუ $x \le parent(x)$. ამ თვისებას **გროვის ძირითად თვისებას** უწოდებენ. გროვა საშუალებას იძლევა დინამიურ სიმრავლეზე ჩატარდეს შემდეგი სახის ოპერაციები: INSERT (დამატება), DELETE (წაშლა), MINIMUM (მინიმუმი), MAXIMUM (მაქსიმუმი).

გროვის განსაზღვრებიდან გამომდინარეობს შემდეგი თვისება:

გროვის არცერთი ელემენტი არ არის ხის ძირზე მეტი.

გროვის ძირითადი თვისების ანალოგიური შეზღუდვა შეგვიძლია ნებისმიერი ტიპის ხეზე განვახორციელოთ, მაგრამ განსაკუთრებით მოსახერხებელია სრული ორობითი ხის (complete binary tree) გამოყენება, რადგან იგი იოლად რეალიზდება ერთგანზომილებიანი მასივის საშუალებითაც, სადაც კავშირები ელემენტებს შორის მიმთითებლების ნაცვლად ხორციელდება მათი ინდექსებით. მასივში i-ურ ინდექსზე მყოფი ელემენტის შვილებია 2i და 2i+1 ინდექსებზე მყოფი ელემენტები, ხოლო იმავე ელემენტის მშობლის ინდექსი იქნება $\lfloor i / 2 \rfloor$. ცხადია, რომ ასეთ შემთხვევაში მასივში ხის ძირის ინდექსი იქნება 1, ძირის მარცხენა შვილის ინდექსი – 2, ძირის მარჯვენა შვილის ინდექსი – 3 და ა. შ. ქვემოთ ნახაზზე ნაჩვენებია ხის ელემენტებისა და მასივის ინდექსების ურთიერთმიმართება

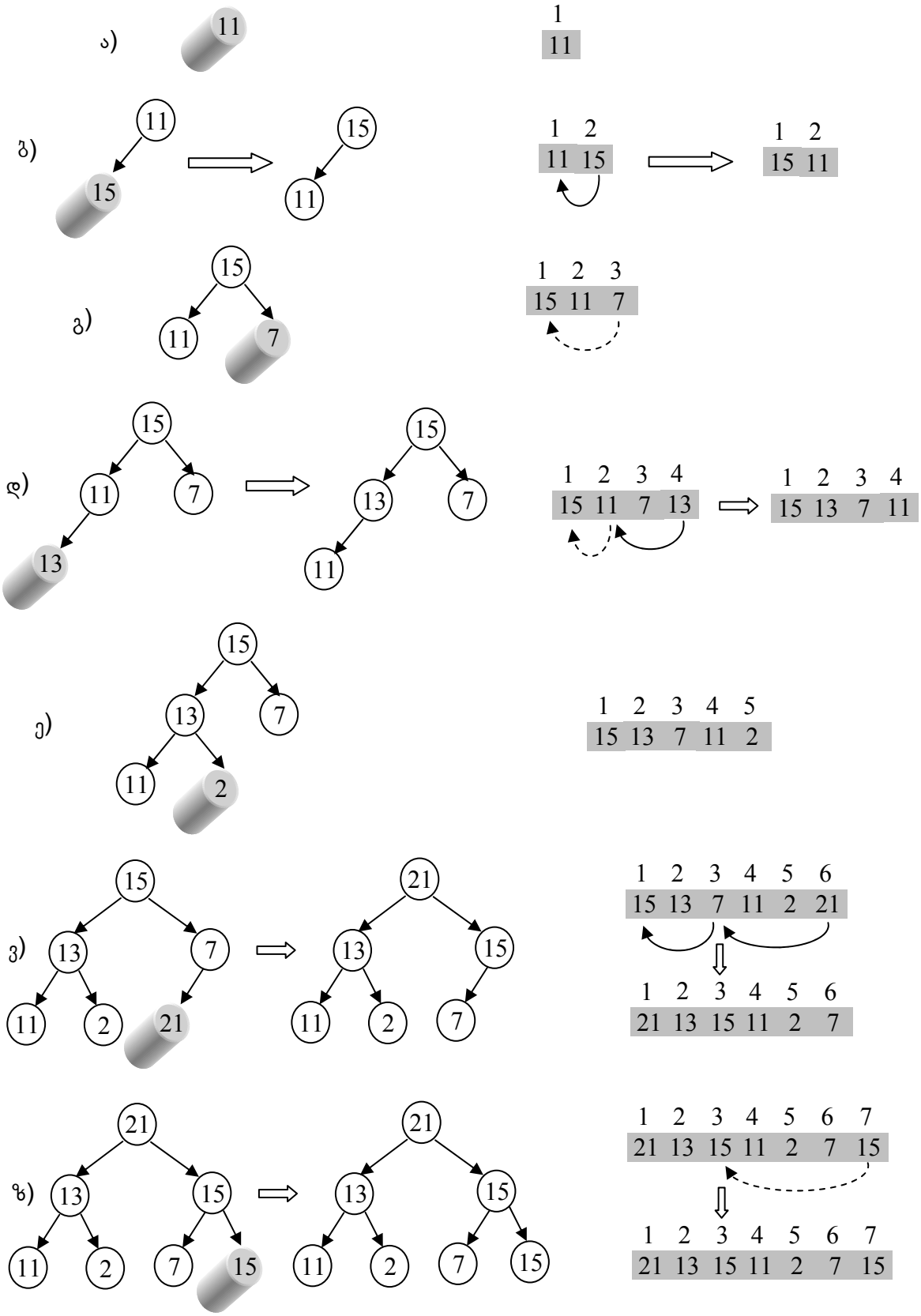


1	2	3	4	5	6
26	16	19	5	9	12

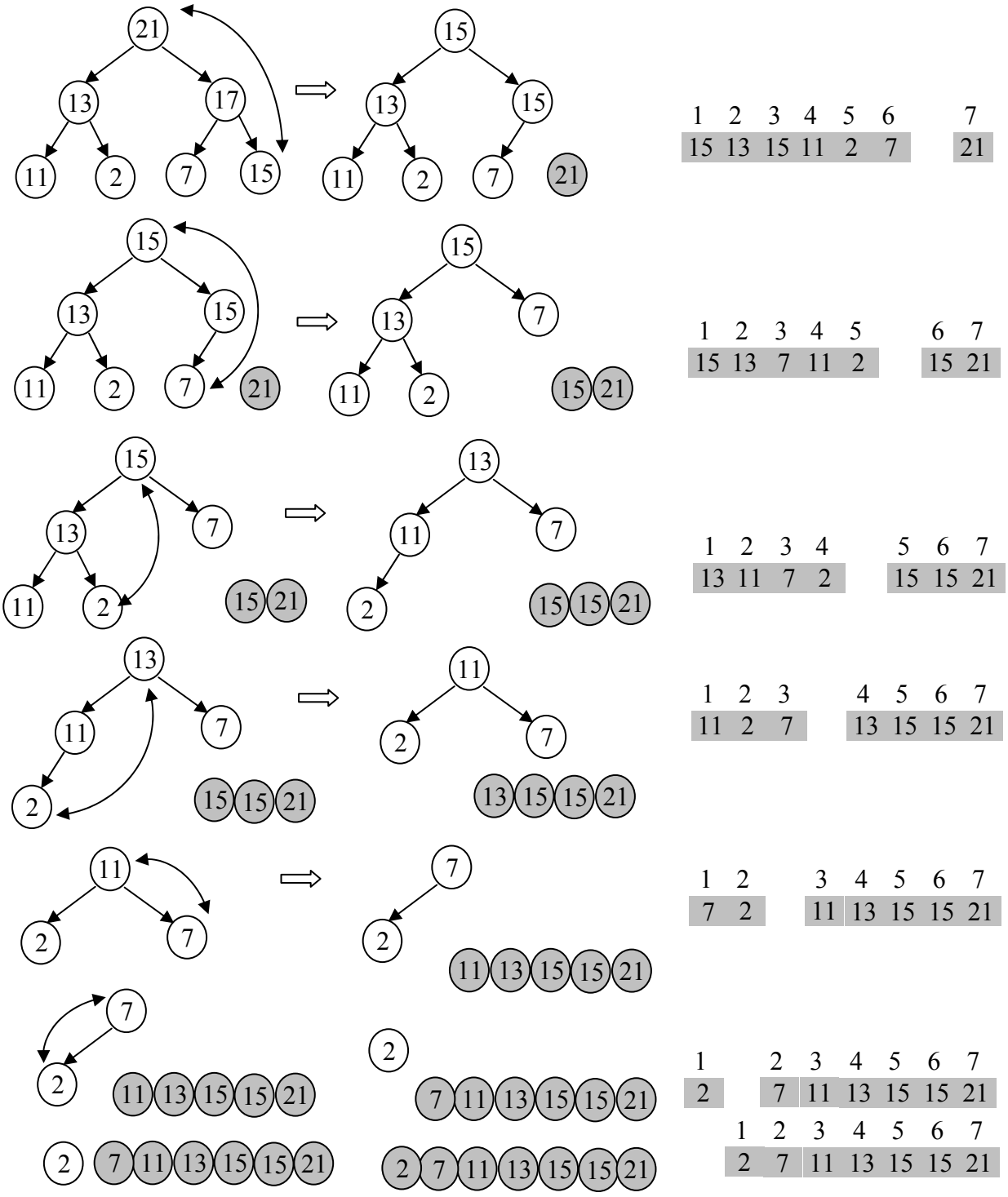
უმეტეს შემთხვევაში თავდაპირველად საჭიროა საწყისი სიმრავლიდან გროვის აგება. ეს პროცესი შეიძლება განხორციელდეს როგორც საწყისი სიმრავლის ელემენტების თანმიმდევრული განხილვით და გროვის თანდათანობით აგებით, ასევე საწყისი მასივი ერთბაშად მოთავსებით მასივში და შემდეგ ელემენტებისათვის ადგილების გაცვლით. პირველი მეთოდის გამოყენებისას ვასრულებთ შემდეგ ბიჯებს: ა) პირველი ელემენტი განიხილება, როგორც გროვის ძირი; ბ) დაწყებული მეორე ელემენტიდან, ახალი ელემენტის დამატება ხდება ყველაზე მცირე ნომრის მქონე თავისუფალი ელემენტის მარცხენა, ხოლო თუ მარცხენა უკვე არსებობს, მარჯვენა შვილად; გ) თუკი ელემენტის დამატებისას დაირღვა გროვის ძირითადი თვისება, აღვადგინოთ იგი ელემენტებისათვის ადგილების გაცვლით. მეორე მეთოდის გამოყენებისას თანმიმდევრულად ვამოწმებთ არღვევენ თუ არა ელემენტები გროვის ძირითად თვისებას და თუკი არღვევენ, ადგილების გაცვლით აღვადგენთ მას კონკრეტული კვანძისათვის.

ქვემოთ ნახაზზე მოცემულია გროვის აგების პროცესი. მონაცემები შემოდის შემდეგი თანმიმდევრობით: 11, 15, 7, 13, 2, 21, 15. ახალი ელემენტის დამატება გროვისათვის გამოსახულია სივრცული ფიგურით. თუკი ახალი ელემენტი არღვევს გროვის ძირითად თვისებას, ანუ მისი მნიშვნელობა მეტია მშობლის მნიშვნელობაზე, ხდება ადგილების გაცვლა ახალ ელემენტსა და მის მშობელს შორის მანამ, ვიდრე გროვის ძირითადი თვისება არ აღდგება. ნახაზზე ნაჩვენებია ამ პროცესის მიმდინარეობა, როგორც ხის, ასევე მასივის

სახითაც. მასივის ელემენტებს შორის მომხდარი შედარებები მითითებულია ისრების სახით: შედარებები, რომლებსაც მოჰყვა ელემენტებს შორის ადგილის გაცვლა, აღნიშნულია უწყვეტი ისრებით, ხოლო შედარებები, რომლებსაც არ მოჰყვა ადგილის გაცვლა – წყვეტილი ისრებით. პროცესის დასრულების შემდეგ მასივს ექნება სახე: 21, 13, 15, 11, 2, 7, 15. ასე დალაგებული მასივი უკვე გროვას წარმოადგენს.



სორტირება გროვის (heapsort) საშუალებით გარკვეულწილად წააგავს სორტირებას ამორჩევით (selectionsort), სადაც ყოვრლ იტერაციაზე ჩვენ ვეძებთ მასივის მინიმალურ ელემენტს და ვუცვლით მას ადგილს პირველ ელემენტთან. გროვით სორტირება პირიქით იქცევა: მაქსიმალურ ელემენტს (გროვის ძირს) უცვლის ადგილს ბოლო ელემენტთან და განსახილველ მასივს ერთით ამცირებს მარჯვენა ბოლოდან. თუმცა განსხვავება ამ ორ მეთოდს შორის მსგავსებაზე მეტია – თუ ამორჩევით სორტირებაში მინიმუმის ყოველი მოძებნისას ალგორითმა მთლიანად თავიდან უნდა გაიაროს მასივის დაულაგებელი ნაწილი, გროვით სორტირებაში მაქსიმუმის ამორჩევის შემდეგ მომდევნო მაქსიმუმი გაცილებით სწრაფად იპოვება გროვის ძირითადი თვისების აღდგენის შედეგად.



```
void Heapify(int A[],int i,int HeapSize)
{
  int left=2*i, right=2*i+1;
  int largest;
  if ((left <= HeapSize) &&
      (A[left] > A[i]))
      largest = left;
  else
      largest = i;

  if ((right <= HeapSize) &&
      (A[right] > A[largest]))
      largest = right;

  if (largest != i) {
    Swap(&A[i],&A[largest]);
    Heapify(A,largest,HeapSize);
  }   }

void HeapSort(int A[], int n)
{
     int i, HeapSize = n;
     for (i= HeapSize/2; i >= 1; i--)
         Heapify(A,i,HeapSize);

     for (i=n; i>=2; i--) {
          Swap(&A[i],&A[1]);
          HeapSize--;
          Heapify(A,1,HeapSize);
     }   }
```

რეკურსიის გარეშე

```
#include <iostream.h>
#include <stdlib.h>
void siftDown( int k, int a[], int N) {
   int v, j;
   v = a[k];
   while( k <= N/2) {
      j = 2 * k;
      if(j < N && a[j] < a[j+1]) ++j;
      if( v >= a[j]) break;
      a[k] = a[j]; k = j;
   }
   a[k] = v;
}

void heapSort(int a[], int N)
{
   int k, v;
      for(k = N/2; k >= 1; --k)
      siftDown( k, a, N);

   for(k = N; k > 1; --k) {
      v = a[1];
      a[1] = a[N--];
      siftDown( 1, a, N);
      a[k] = v;
}   }


void main()
{
   int a[21];
int i;
   // insert 20 random number between 0 and 19 into array
   for(i = 1; i <= 20; ++i)
      a[i] = rand() % 100;

cout << "\nUnsorted array is:\n";
for(i = 1; i <= 20; ++i)
       cout << " " << a[i];
```

```
heapSort( a, 20);
cout << "\n\nSorted array is:\n";
for(i = 1; i <= 20; ++i)
       cout << " " << a[i];
       cout << "\n\n";
}
```

2.6. ძებნის ორობითი ხეები

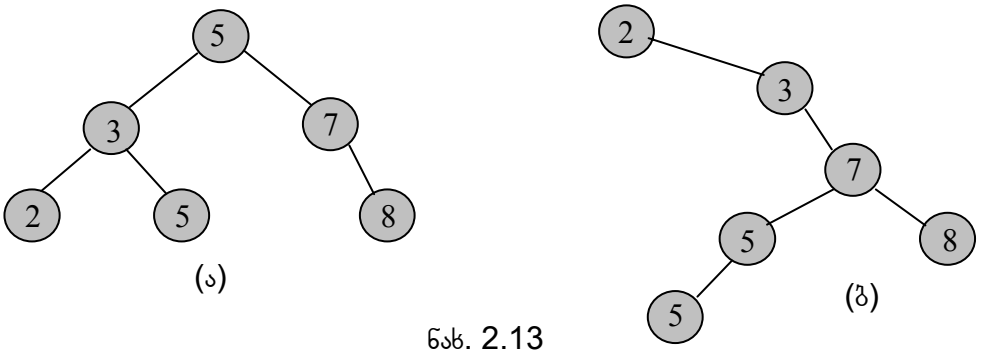
ძებნის ორობითი ხეები საშუალებას იძლევიან დინამიურ სიმრავლეებზე ჩატარდეს შემდეგი სახის ოპერაციები: SEARCH (ძებნა), INSERT(დამატება), DELETE (წაშლა), MINIMUM (მინიმუმი), MAXIMUM (მაქსიმუმი), SUCCESSOR (მომდევნო), PREDECESSOR (წინა). ამგვარად ძებნის ხე შეიძლება გამოყენებულ იქნას როგორც ლექსიკონი და როგორც პრიორიტეტებიანი რიგი.

ძირითადი ოპერაციების შესრულების დრო ხის სიმაღლის პროპორციულია. თუკი ორობითი ხე "მჭიდროდაა შევსებული" ანუ მისი ყველა დონე შეიცავს წვეროთა მაქსიმალურ შესაძლო რაოდენობას, მაშინ მისი სიმაღლე (და შესაბამისად ოპერაციების შესრულების დროც) წვეროთა რაოდენობის ლოგარითმის პროპორციულია. თუკი პირიქით – ხე წარმოადგენს n წვეროს წრფივ ჯაჭვს, მაშინ შესრულების დრო იზრდება $\Theta(n)$-მდე.

ძებნის ორობითი ხის (binary search tree) თითოეულ წვეროს შეიძლება გააჩნდეს (ან არ გააჩნდეს) მარცხენა და მარჯვენა შვილი; ძირის გარდა ყველა წვეროს გააჩნია მშობელი. მიმთითებლების გამოყენებით წარმოდგენისასხის თითოეული წვეროსათვის key გასაღებისა და დამატებითი მონაცემების გარდა ვინახავთ აგრეთვე მიმთითებლებს left, right და p (მარცხენა შვილი, მარჯვენა შვილი, მშობელი). თუკი შვილი (ან მშობელი – ძირისათვის) არ არსებობს, შესაბამისი მინდორი შეიცავს nil-ს.

გასაღებები ძებნის ორობით ხეში ინახება **დალაგებულობის** ანუ **ძებნის ორობითი ხის თვისების** (binary-search-tree property) მიხედვით:

ვთქვათ, x ძებნის ორობითი ხის ნებისმიერი წვეროა. თუ y წვერო იმყოფება x წვეროს მარჯვენა ქვეხეში, მაშინ $key[y] \le key[x]$. თუ y წვერო იმყოფება x წვეროს მარცხენა ქვეხეში, მაშინ $key[y] \ge key[x]$.



ნახ. 2.13

ნახ. 2.13-ზე ორობითი ძებნის ხის მაგალითები. სხვადასხვა ორობითი ხე შეიძლება წარმოადგენდეს ერთ და იმავე სიმრავლეს. ოპერაციების შესრულების დრო ხის სიმაღლის პროპორციულია. (ა) ნახაზზე მოცემული ხის სიმაღლეა 2 და ის შეიცავს 6 წვეროს, (ბ) ნახაზზე მოცემულია იმავე გასაღებების შემცველი ხე, რომლის სიმაღლეა 4. (ა)-ზე ნაჩვენები ხის ძირი შეიცავს გასაღებს 5, მარცხენა ქვეხეში მყოფი გასაღებები 2, 3 და 5 არ აღემატებიან მის მნიშვნელობას, ხოლო მარჯვენა ქვეხეში მყოფი გასაღებები – 7 და 8, არ არიან მასზე ნაკლები. იგივე სრულდება ნებისმიერი სხვა წვეროსათვის.

დალაგებულობის თვისება საშუალებას გვაძლევს მარტივი რეკურსიული ალგორითმის დახმარებით დავბეჭდოთ ყველა გასაღების მნიშვნელობა არაკლებადი თანმიმდევრობით. ალგორითმს ჰქვია inorder tree walk. ეს ალგორითმი ქვეხის ძირის გასაღებს ბეჭდავს მისი მარცხენა ქვეხის ყველა გასაღების შემდეგ და მარჯვენა ქვეხის ყველა გასაღებზე ადრე. აქვე

შევნიშნოთ, რომ თანმიმდევრობას, რომელშიც ძირი წინ უსწრებს ორივე ქვეხეს, უწოდებენ preorder, ხოლო თანმიმდევრობას, რომელშიც ძირი ქვეხეების მერე დგას – postorder.

INORDER-TREE-WALK(root[T]) მიმდევრობით ბეჭდავს root[T] ძირის მქონე T ხის ყველა გასაღებს:

```
INORDER-TREE-WALK(x)
1 if x≠nil then   {
2         INORDER-TREE-WALK(left[x])
3         write key[x]
4         INORDER-TREE-WALK(right[x])
```

ნახ. 2.13-ზე მოცემული ორივე ხისათვის პროცედურა დაბეჭდავს 2, 3, 5, 5, 7, 8. მუშაობის დრო n წვეროსათვის არის Θ(n). ყოველი წვეროს დამუშავება ხდება მხოლოდ ერთხელ.

ძებნის პროცედურისათვის შემავალი მონაცემებია საძებნი გასაღები key და იმ ქვეხის ძირის x მიმთითებელი, რომელშიც ძებნა უნდა განხორციელდეს. გამომავალ მონაცემს წარმოადგენს მიმთითებელი k გასაღების მქონე წვეროზე (თუკი ასეთი არსებობს) ან სპეციალური მნიშვნელობა nil (წვეროს არარსებობის შემთხვევაში).

```
TREE-SEARCH(x,k)
1 if x=nil OR k=key[x] then {     return x     }
2 if k<key[x] then  { return TREE-SEARCH(left[x],k)     }
3                   else  { return TREE-SEARCH(right[x],k)   }
```

ძებნის პროცესში ჩვენ ვმოძრაობთ ძირიდან და k გასაღებს ვადარებთ მიმდინარე x წვეროში მოთავსებულ გასაღებთან. თუ ისინი ტოლია, ძებნა მთავრდება. თუ k<key[x], ძებნა გრძელდება მარცხენა ქვეხეში (k გასაღები, დალაგებულობის პრინციპიდან გამომდინარე, მხოლოდ იქ შეიძლება იყოს). k>key[x], მაშინ ძებნა გრძელდება მარჯვენა ქვეხეში. ძებნის ხის სიგრძე არ აღემატება ხის სიმაღლეს, ამიტომ ძებნის დროა O(h), სადაც h – ხის სიმაღლეა.

მოვიყვანოთ იმავე პროცედურის იტერაციული ვერსია, რომელიც, როგორც წესი, უფრო ეფექტურია:

```
ITERATIVE-TREE-SEARCH(x,k)
1 while x≠NIL OR k≠key[x] do {
2                   if k<key[x] then  {  x=left[x]  }
3                                 else  {  x=right[x]    }          }
4 return x
```

ძებნის ხეში მინიმალური გასაღების მოსაძებნად უნდა ვიმოძრაოთ ძირიდან left მიმთითებლების მიმართულებით, ვიდრე არ შეგვხვდება nil. პროცედურის გამომავალი მონაცემია მიმთითებელი x ძირის მქონე ქვეხის მინიმალურ ელემენტზე:

```
TREE-MINIMUM(x)
1 while left[x]≠nil do   {
2         x=left[x]  }
3 return x
```

დალაგებულობის პრინციპი უზრუნველყოფს პროცედურის სწორად მუშაობას. თუკი x წვეროს არ გააჩნია მარცხენა შვილი, მაშინ x ძირის მქონე ქვეხის მინიმალური ელემენტი თავად x-ია, რადგან ნებისმიერი გასაღები მარჯვენა ქვეხეში არაა key[x]-ზე ნაკლები (იხ. ნახ. 2.13(ბ)). თუკი x-ის მარცხენა ქვეხე ცარიელი არ არის, მაშინ მინიმალური ელემენტი სწორედ ამ ქვეხეში იქნება. ალგორითმი TREE-MAXIMUM(x) სიმეტრიულია:

```
TREE- MAXIMUM (x)
1 while right[x]≠nil do   {
2         x=right[x]  }
3 return x
```

ორივე ალგორითმი საჭიროებს O(h) დროს (მოძრაობა მხოლოდ ქვემოთ ხდება), სადაც h – ხის სიმაღლეა.

დალაგებულობის თვისება საშუალებას იძლევა ვიპოვოთ მოცემული ელემენტის მომდევნო ელემენტი (თუ ყველა გასაღები განსხვავებულია, გამოსავალ მონაცემს წარმოადგენს სიდიდით მომდევნო გასაღები), ან nil, თუ x ელემენტი ხეში უკანასკნელია:

```
TREE-SUCCESSOR (x)
1 if right[x]≠NIL then {  return TREE-MINIMUM(right[x])   }
2 y=p[x]
3 while y≠NIL AND x=right[y] do {
4                   x=y
5                   y=p[y]
6 return y
```

TREE-SUCCESSOR პროცედურა ცალ-ცალკე განიხილავს ორ შემთხვევას. თუ x წვეროს მარჯვენა ქვეხე ცარიელი არ არის, მაშინ x-ის მომდევნო ელემენტი არის ამ ქვეხის მინიმალური ელემენტი და ტოლია TREE-MINIMUM(right[x])-ის. თუკი მარჯვენა ქვეხე ცარიელია, მაშინ ჩვენ ვმოძრაობთ x-დან ზემოთ, ვიდრე არ ვიპოვით წვეროს, რომელიც თავისი მშობლის მარცხენა შვილია (2-6 სტრიქონები). სწორედ ეს მშობელი (თუკი ის არსებობს) წარმოადგენს საძებნ ელემენტს.

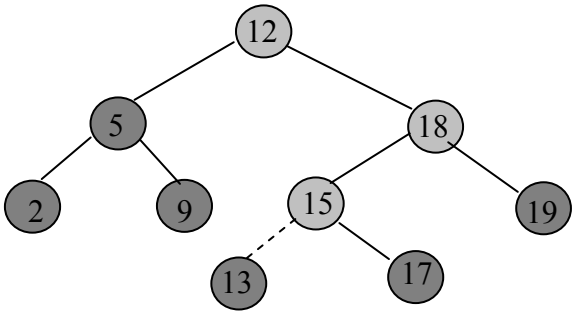
TREE-SUCCESSOR პროცედურის მუშაობის დრო h სიმაღლის მქონე ხეზე O(h)-ის ტოლია, რადგან მოძრაობა ხდება ან მხოლოდ ზემოთ, ან მხოლოდ ქვემოთ. პროცედურა TREE-PREDECESSOR სიმეტრიულია.

ზემოთ თქმულიდან გამომდინარე ადგილი აქვს თეორემას:

თეორემა 2.3. h სიმაღლის მქონე ხეში SEARCH, MINIMUM, MAXIMUM, SUCCESSOR და PREDECESSOR ოპერაციები სრულდება O(h) დროში.

პროცედურა TREE-INSERT ამატებს მოცემულ ელემენტს T ხის შესაბამის ადგილას ისე, რომ დალაგებულობის თვისება შენარჩუნებულ იქნას. პროცედურის პარამეტრს წარმოადგენს მიმთითებელი ახალ წვეროზე z, რომელშიც მოთავსებულია მნიშვნელობები: key[z], left[z]=nil და right[z]=nil. მუშაობის დროს პროცედურა ცვლის T ხეს და z წვეროს ზოგიერთ მინდორს, რის შემდეგაც ახალი წვერო აღმოჩნდება ჩასმული ხის შესაბამის ადგილას.

```
TREE-INSERT(T,z)
1 y=nil
2 x=root[T]
3 while x≠nil do  {
4          y=x
5          if key[z]<key[x] then  {  x=left[x]   }
6                                 else  {  x=right[x]  }      }
7 p[z]=y
8 if y=nil then {  root[T]=z  }
9          else {  if key[z]<key[y]  then {    left[y]=z   }
10                                        else  {   right[y]=z  }     }
```

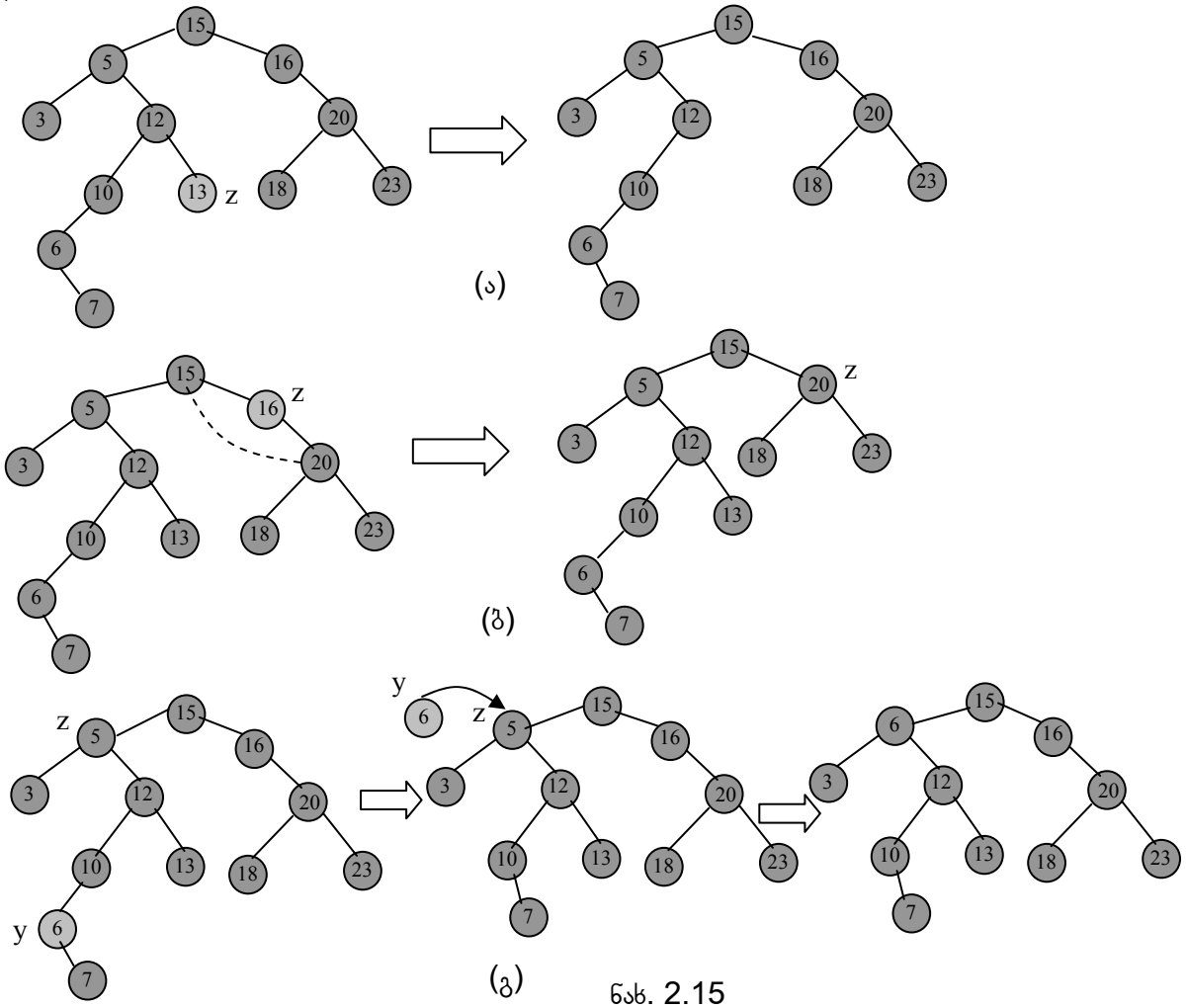


ნახ. 2.14

ნახ. 2.14-ზე ნაჩვენებია თუ როგორ მუშაობს პროცედურა TREE-INSERT. იგი მოძრაობს ხის ძირიდან ქვემოთ, ამასთან y წვეროში ინახება მიმთითებელი x წვეროს მშობელზე (ციკლი

3-6 სტრიქონებში). key[z]-ისა და key[x]-ის შედარებით პროცედურა წყვეტს მარჯვნივ წავიდეს თუ მარცხნივ. პროცესი მთავრდება, როცა x ხდება nil. სწორედ ამ nil-ის ადგილზე უნდა მოთავსდეს z, რაც ხორციელდება 3-10 სტრიქონებში. ნახაზზე ღია ფერის წვეროებით აღნიშნულია გზა ფუძიდან ახალი ელემენტის პოზიციამდე. წყვეტილი ხაზით კი ახალი ელემენტი შეერთებულია ძველთან.

სხვა ოპერაციების მსგავსად h სიმაღლის მქონე ხისათვის ამ პროცედურის მუშაობის დროა O(h).



ნახ. 2.15

წაშლის პროცედურის შემავალი მონაცემია მიმთითებელი წასაშლელ წვეროზე. წაშლისას შესაძლებელია სამი შემთხვევა, რომლებიც ნაჩვენებია ნახ. 2.15-ზე: ა) z-ს არ გააჩნია შვილები, ამ დროს z-ის წასაშლელად საკმარისია მისი მშობლის შესაბამის მინდორში მოვათავსოთ nil; ბ) z-ს გააჩნია ერთი შვილი, ამ შემთხვევაში z "ამოიშლება" მისი მშობლისა და შვილის პირდაპირი შეერთებით; გ) z-ს გააჩნია ორი შვილი – აქ წვეროს წაშლამდე საჭიროა გარკვეული მოსამზადებელი სამუშაოების ჩატარება: უნდა მოვძებნოთ გასაღების სიდიდის მიხედვით მომდევნო y ელემენტი. მას არ ეყოლება მარცხენა შვილი (ძებნის ორობით ხეში მტკიცდება შემდეგი თვისება: თუ ელემენტს გააჩნია ორი შვილი, მაშინ გასაღების სიდიდის მიხედვით მომდევნო ელემენტს არ გააჩნია მარცხენა შვილი, ხოლო გასაღების სიდიდის მიხედვით წინას – მარჯვენა შვილი). ამის შემდეგ y წვეროს გასაღები და დამატებითი მონაცემები z წვეროს შესაბამის მინდვრებში, ხოლო თავად y წვერო წავშალოთ ზემოთ ნახსენები (ბ) ხერხით.

დაახლოებით ასეთი ალგორითმით მუშაობს პროცედურა TREE-DELETE (იგი ზემოთ ჩამოთვლილ სამ შემთხვევას ოდნავ სხვაგვარი მიმდევრობით განიხილავს):

```
TREE-DELETE(T,z)
1 if left[z]=nil OR right[z]=nil then {  y=z  }
2                                 else {  y=TREE-SUCCESSOR(z)  }
3 if left[y]≠nil then {  x=left[y]   }
```

```
4                 else {   x=right[y]   }
5 if x≠nil then p[x]=p[y]
6 if p[y]=nil then {   root[T]=x  }
7               else {   if y=left[p[y]] then {  left[p[y]]=x  }
8                                        else  {  right[p[y]]=x  }        }
9 if y≠z then  { key[z]=key[y]
10                    დამატებითი ინფორმაციის კოპირება y-დან z-ში      }
11 return y
```

1-2 სტრიქონებში განისაზღვრება y წვერო, რომელიც წასაშლელია ხიდან. ეს არის z წვერო – თუკი მას ერთზე მეტი შვილი არ გააჩნია, ან უშუალოდ მისი მომდევნო ელემენტი – თუკი z-ს გააჩნია ორი შვილი. 3-4 სტრიქონებში x ცვლადი ხდება y წვეროს შვილის მიმთითებელი ან ხდება nil, თუკი y-ს შვილი არ გააჩნია. 5-8 სტრიქონებში ხდება y წვეროს წაშლა (იცვლება მიმთითებლები p[y] და x წვეროებში), ამასთან ცალკე განიხილება შემთხვევები, როცა x=nil ან y წარმოადგენს ხის ძირს. 9-10 სტრიქონებში ხდება წაშლილი y წვეროს შედარება z-თან და თუ ისინი განსხვავდებიან, y წვეროს მიმთითებელი და დამატებითი მონაცემები ჩაიწერება z-ში, რადგან ჩვენ უნდა წაგვეშალა z, და არა y. ბოლოს პროცედურა აბრუნებს y მიმთითებლის მნიშვნელობას (რაც საშუალებას მისცემს გამომძახებელ პროცედურას, გაათავისუფლოს y წვეროსათვის გამოყოფილი მეხსიერება).

h სიმაღლის მქონე ხისათვის ამ პროცედურის მუშაობის დროა O(h) და ადგილი აქვს თეორემას:

თეორემა 2.4. h სიმაღლის მქონე ხეში INSERT და DELETE ოპერაციები სრულდება O(h) დროში.

2.7. წითელ-შავი (ჟღალი) ხეები

წინა თავში ჩვენ ვნახეთ, რომ ძირითადი ოპერაციები h სიმაღლის მქონე ძებნის ორობით ხეზე შეიძლება შესრულდეს O(h) დროში. ასეთი ხეების გამოყენება ეფექტურია, თუკი მათი სიმაღლე მცირეა, წინააღმდეგ შემთხვევაში ისინი სიებზე მეტად არ არიან ეფექტურნი. წითელ-შავი ხეები ძებნის "დაბალანსებული" ხეების ერთ-ერთი ტიპია და ბალანსირების სპეციალური ოპერაციები უზრუნველყოფენ, რომ ხის სიმაღლე O(log *n*)-ს არ აღემატება.

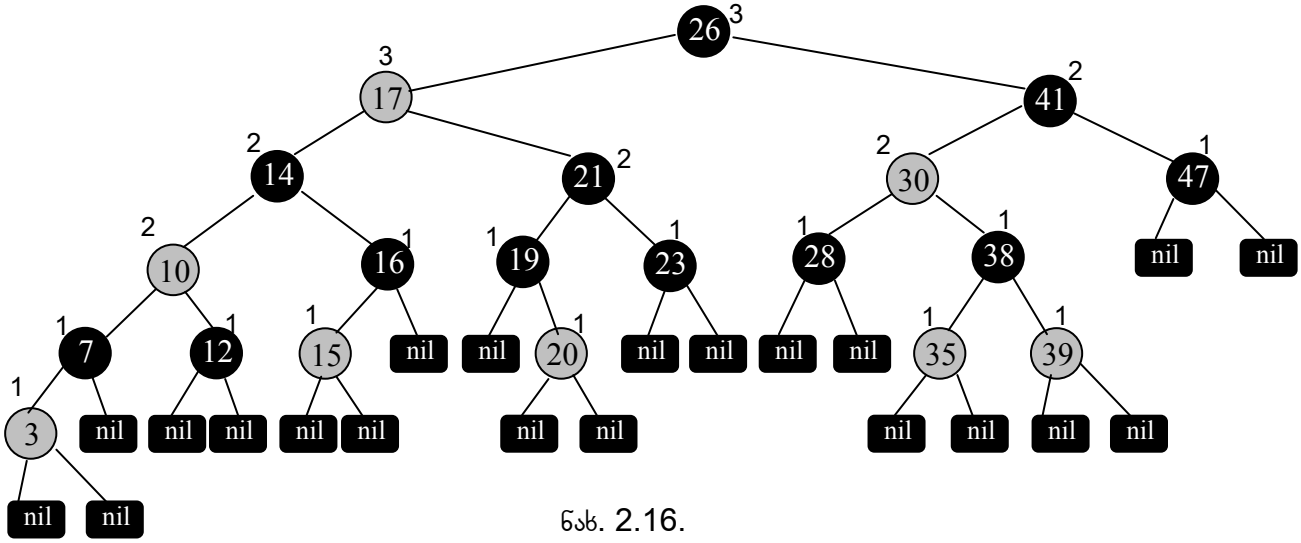
წითელ-შავი ანუ ჟღალი ხე (red-black tree, სიმარტივისათვის ვუწოდოთ "ჟღალი", რომელიც საბას განმარტებით ნიშნავს "მოწითალო შავს") წარმოადგენს ძებნის ორობით ხეს, რომლის წვეროები გაყოფილია წითლებად (red) და შავებად (black). ყოველი წვერო დამატებით ინახავს კიდევ ერთ ბიტს – ფერს. ამასთან, უნდა შესრულდეს გარკვეული მოთხოვნები, რომლებიც უზრუნველყოფენ, რომ ნებისმიერი ორი ფოთლის სიღრმე ერთმანეთს ორჯერ მეტად არ აღემატება. ამის გამო ხეს ეწოდება **ბალანსირებული** (balanced).

ჟღალი ხის ნებისმიერ წვეროს გააჩნია მინდვრები: color (ფერი), key (გასაღები), left (მარცხენა), right (მარჯვენა) და p (მშობელი). თუკი რომელიმე წვეროს არ გააჩნია მშობელი ან შვილი, შესაბამისი მინდორი უდრის nil-ს. მეტი მოხერხებულობისათვის ჩავთვალოთ, რომ left და right მინდვრების nil მნიშვნელობები წარმოადგენენ მიმართვას ხის დამატებით, ფიქტიურ ფოთლებზე. ხის ასეთი შევსებით გასაღების შემცველი ყველა წვერო გახდება შინაგანი წვერო, რადგან თითოეულ მათგანს უკვე გააჩნია ორ-ორი შვილი.

ძებნის ორობით ხეს ეწოდება ჟღალი (წითელ-შავი) ხე, თუკი ის აკმაყოფილებს შემდეგ თვისებებს (ვუწოდოთ მათ **RB-თვისებები**, ინგლისურად red-back properties):

1) ყოველი წვერო ან წითელია, ან შავი;
2) ყოველი ფოთოლი (nil) შავია;
3) თუკი წვერო წითელია, მისი ორი შვილი შავია;
4) ძირიდან ფოთლებისაკენ მიმავალი ყველა გზა შეიცავს შავი წვეროების თანაბარ რაოდენობას.

მეოთხე თვისებიდან გამომდინარეობს, რომ ნებისმიერი x წვეროდან ფოთლებისაკენ მიმავალი თითოეული გზა შეიცავს შავ წვეროების ერთნაირ რაოდენობას, რომელსაც x წვეროს შავ სიმაღლეს უწოდებენ და აღნიშნავენ bh(x)–ით. ხის შავ სიმაღლედ ითვლება მისი ძირის შავი სიმაღლე.



ნახ. 2.16.

ჟღალი ხის მაგალითი მოცემულია ნახ. 2.16-ზე. შავი წვეროები აღნიშნულია უფრო მუქად, ხოლო წითელი წვეროები – შედარებით ღია რუხი ფერით. ყოველი წვერო ან წითელია, ან – შავი. ყველა nil-ფოთოლი შავია. თითოეული წვეროსათვის მისგან ფოთლებისაკენ მიმავალი გზა შეიცავს ერთნაირი რაოდენობის შავ წვეროებს. ყოველ წვეროს გვერდით მიწერილი აქვს მისი შავი სიმაღლე. ფოთლების შავი სიმაღლე 0-ის ტოლია

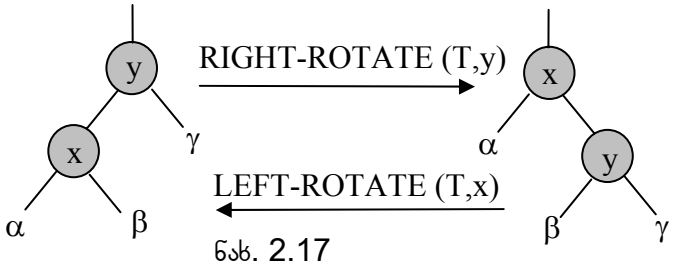
ჟღალი ხეებისათვის მტკიცდება შედეგი ლემა:

ლემა 2.5. n შინაგანი წვეროს მქონე ჟღალი ხის სიმაღლეა არაუმეტეს 2log(n+1).

ლემიდან გამომდინარეობს, რომ n წვეროს მქონე ჟღალ ხეში SEARCH, MINIMUM, MAXIMUM, SUCCESSOR და PREDECESSOR ოპერაციები სრულდება O(log n) დროში, რადგან h სიმაღლის მქონე ხეში ეს ოპერაციები სრულდება O(h) დროში, ხოლო ჟღალი ხის სიმაღლეა O(log n).

უფრო რთულადაა საქმე წინა თავში აღწერილ TREE-INSERT და TREE-DELETE პროცედურებთან, რადგან მუშაობის დროს მათ შესაძლოა დაარღვიონ RB-თვისებები, ამიტომ ისინი საჭიროებენ მოდიფიცირებას – პროცედურის დასრულების შემდეგ უნდა აღვადგინოთ RB-თვისებები და შევცვალოთ ხის სტრუქტურა.

სტრუქტურის შეცვლა ხორციელდება **ბრუნვის** (rotation) საშუალებით, რომელიც წარმოადგენს ლოკალურ ოპერაციას (იცვლება მხოლოდ რამდენიმე მიმთითებელი) და ინარჩუნებს დალაგებულობის თვისებას.

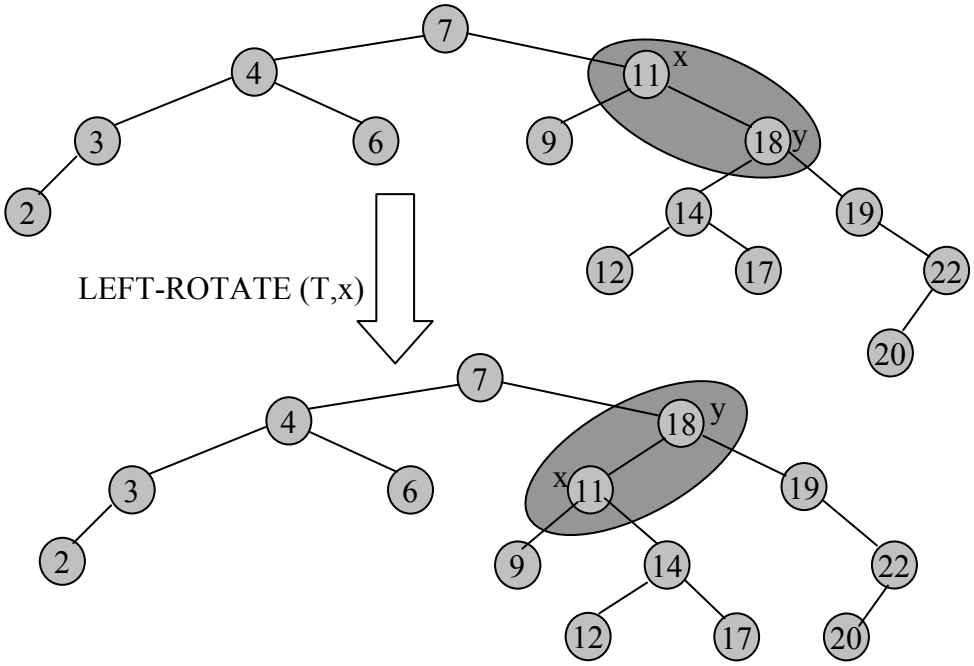


ნახ. 2.17

ნახ.2.17-ზე ნაჩვენებია ურთიერთსაპირისპირო ბრუნვები: მარცხენა და მარჯვენა. მარცხენა ბრუნვა შესაძლებელია ნებისმიერ x წვეროში, რომლის მარჯვენა შვილიც (დავარქვათ მას y) არ წარმოადგენს nil-ფოთოლს. ბრუნვის შემდეგ y მოექცევა ზემოთ, x გახდება y-ის მარცხენა შვილი, ხოლო y-ის ყოფილი მარცხენა შვილი – x-ის მარჯვენა შვილი. α, β და γ სიმბოლოები აღნიშნავენ შესაბამის ქვეხეებს x და y წვეროები შეიძლება იმყოფებოდნენ ხის ნებისმიერ ადგილას. LEFT-ROTATE პროცედურაში იგულისხმება, რომ right[x]≠nil.

```
LEFT-ROTATE(T,x)
1 y=right[x]
2 right[x]=left[y]
3 if left[y]≠nil then {  p[left[y]]=x  }
4 p[y]=p[x]
5 if p[x]=nil then {  root[T]=y  }
6               else {  if x=left[p[x]] then {  left[p[x]]=y  }
7                                       else {  right[p[x]]=y }   }
8 left[y]=x
9 p[x]=y
```

პროცედურა LEFT-ROTATE-ს მოქმედება ნაჩვენებია ნახ.2.18-ზე. პროცედურა RIGHT-ROTATE ანალოგიურია. ორივე მუშაობს O(1) დროში და ცვლიან მხოლოდ მიმთითებლებს.



ნახ. 2.18

ჟღალი ხისათვის წვეროს დამატება ხდება O(log n) დროში. თავდაპირველად ვიყენებთ ძებნის ორობითი ხისათვის დაწერილ TREE-INSERT პროცედურას და ახალ წვეროს ვაძლევთ წითელ ფერს. ამის შემდეგ უნდა აღვადგინოთ RB-თვისებები, რისთვისაც საჭიროა ზოგიერთი წვეროსათვის ფერის შეცვლა და ბრუნვის პროცედურა. განიხილება რამდენიმე სიტუაცია.

```
RB-INSERT(T,x)
1 TREE-INSERT(T,x)
2 color[x]=red
3 while x≠root[T] AND color[p[x]]=red do {
4 if p[x]=left[p[p[x]]] then {  y=right[p[p[x]]]
5                   if color[y]=red then  { color[p[x]]=black;  color[y]=black
6                                                 color[p[p[x]]]=red;    x=p[p[x]]   }
7                                       else  {  if x=right[p[x]] then { x=p[x]
8                                                                                 LEFT-ROTATE(T,x)  }
9                                                 color[p[x]]=black;  color[p[p[x]]]=red;
10                                                RIGHT-ROTATE(T,p[p[x]])  }
11                                  else { ÉÂÉÅÄ ÔÄØÓÔÉ left↔right ÝÅËÉËÄÁÉÈ}     }
12 color[root[T]]=black
```

2.1. გამჭვირვალე სინჯარები

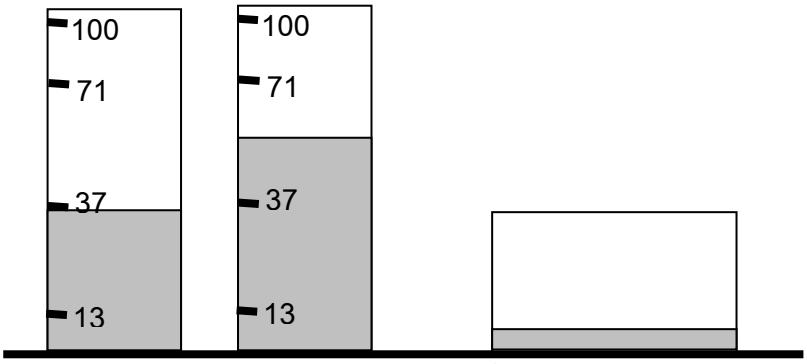
(საქართველოს მოსწავლეთა რესპუბლიკური ოლიმპიადა, 1996-97 წელი)

გვაქვს სამი სინჯარა – თითოეული 100 ერთეული მოცულობის. ორ მათგანს ერთნაირად აქვს ოთხ-ოთხი დანაყოფი. ყოველ დანაყოფს გვერდით მიწერილი აქვს რიცხვი, რომელიც მიუთითებს ფსკერიდან ამ დანაყოფამდე მოცულობის მნიშვნელობას. მესამე სინჯარას დანაყოფები არ გააჩნია. ერთი დანაყოფებიანი სინჯარა სავსეა (100 ერთეული მოცულობის სითხით), ხოლო დანარჩენი ორი – ცარიელი.

დაწერეთ პროგრამა, რომელიც დაადგენს შესაძლებელია თუ არა ერთი ერთეული მოცულობის სითხის მიღება ცარიელ უდანაყოფო სინჯარაში. თუ ეს შესაძლებელია, დაბეჭდავს ამ მიზნის მისაღწევად საჭირო გადასხმების უმცირეს რაოდენობას. ყოველი გადასხმის შემდეგ ერთი გამოყენებული სინჯარა მაინც უნდა შეიცავდეს სითხეს რომელიმე დანაყოფამდე ან ცარიელი იყოს.

შესატანი მონაცემები შეიცავს ოთხ მთელ დადებით რიცხვს, რომელთაგან თითოეული მოთავსებულია [1;100] შუალედში და წარმოადგენს პირველი და მეორე სინჯარის დანაყოფების შესაბამის მოცულობათა მნიშვნელობებს.

გამოსატანი მონაცემების პირველ სტრიქონში უნდა ჩაიწეროს “YES”, თუ შესაძლებელია უდანაყოფო სინჯარაში ერთი ერთეული სითხის მიღება, ხოლო მეორე სტრიქონში – გადასხმების მინიმალური რაოდენობის გამომხატველი რიცხვი. თუ მიზნის მიღწევა შეუძლებელია, მაშინ პირველ სტრიქონში ჩაიწერება “NO”.



ნახ. 5.1

შესატანი მონაცემების მაგალითი:	გამოსატანი მონაცემი მოყვანილი მაგალითისათვის:
37 13 71 100	YES 8

მითითება. მოცემული სამი სინჯარა შეიძლება განვიხილოთ როგორც სისტემა, რომლის მდგომარეობის აღწერაც სამი რიცხვითაა შესაძლებელი. საწყისი მდგომარეობა შეიძლება აღვწეროთ როგორც – (100,0,0). ერთი მდგომარეობიდან მეორეში სისტემა შეიძლება გადავიდეს გადასხმის საშუალებით. მაგალითად, მოყვანილი მაგალითისათვის საწყისი მდგომარეობიდან შეგვიძლია მივიღოთ (63,37,0), (71,29,0), (13,0,87) და ა.შ. X მდგომარეობას, რომელიც მიიღება Y მდგომარეობიდან, ვუწოდოთ Y-ის შვილი. მდგომარეობას, რომლის დროსაც მესამე სინჯარა შეიცავს 1 ლიტრ სითხეს, ვუწოდოთ საბოლოო. ჩვენი ამოცანაა, დავადგინოთ, შესაძლებელია საწყისი (100,0,0) მდგომარეობიდან საბოლოო მდგომარეობის მიღება. ამოცანის ამოსახსნელად შემოვიღოთ მდგომარეობათა ორი სია და ვიმოქმედოთ შემდეგნაირად:

1. *ორივე სიაში ჩავწეროთ საწყისი მდგომარეობა (100,0,0);*
2. *თუკი მეორე სია ცარიელია, გამოვიტანოთ პასუხი “NO”;*
3. *მეორე სიიდან რიგითობის მიხედვით ამოვშალოთ Y მდგომარეობა;*
4. *Y-ის ყველა შვილი, რომელიც არ შედის პირველ სიაში, შევიტანოთ ორივე სიაში; თუკი რომელიმე შვილი წარმოადგენს საბოლოო მდგომარეობას – პროცესი შევწყვიტოთ.*
5. *გადავიდეთ მეორე პუნქტზე.*

ცხადია, რომ ეს ალგორითმი განიხილავს გადასხმათა ყველა ვარიანტს. პირველ სიაში ჩაიწერება უკვე განხილული მდგომარეობები, რათა არ მოხდეს მათი განმეორება, ხოლო მეორე

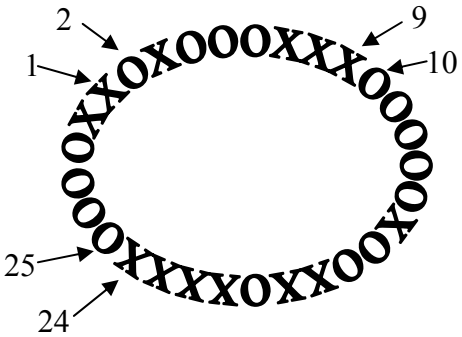
სიაში იქნება ის მდგომარეობები, რომელთა შვილები ჯერ კიდევ შესამოწმებელია. თუკი მეხსიერების შეზღუდვა არ გვიშლის ხელს პირველი სია შეიძლება რეალიზებულ იქნას, როგორც სამგანზომილებიანი ლოგიკური ტიპის მასივი. თუკი (x1,x2,x3) მდგომარეობა განხილულია, მაშინ [x1,x2,x3] ელემენტის მნიშვნელობა იქნება TRUE და შემოწმებას მხოლოდ ერთი ოპერაცია დასჭირდება.

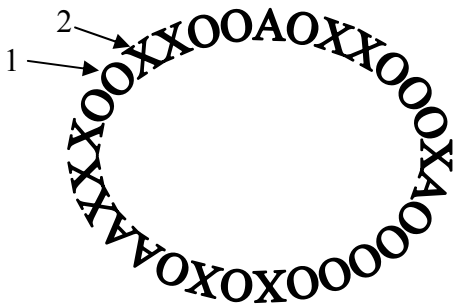
მეორე სიის რეალიზაციისათვის უნდა გამოვიყენოთ რიგი. ყოველი ახალი ელემენტი სიას ბოლოში ანუ მარცხნივ დაემატება, ხოლო მათი განხილვა მოხდება თანმიმდევრობით სიის დასაწყისიდან, ანუ მარჯვნიდან. ელემენტი უნდა შეიცავდეს ინფორმაციას სისტემის მდგომარეობის შესახებ და თუ რომელ ბიჯზე მოხდა ამ მდგომარეობის მიღება. სწორედ ეს უკანასკნელი წარმოადგენს პასუხს საბოლოო მდგომარეობის მიღების შემთხვევაში.

2.2. მძივები

(მოსწავლეთა საერთაშორისო ოლიმპიადა, 1993 წ. არგენტინა)

გვაქვს N(N≤100) ცალი მარცვლისაგან შედგენილი მძივი. მარცვლებიდან ზოგი წითელია, ზოგი ლურჯი, ხოლო დანარჩენი თეთრი. ნახ. 2.19-ზე მოცემულია მძივების მაგალითები N=29-თვის(1-ით და 2-ით აღნიშნულია მძივში პირველი და მეორე პოზიციები),





ნახ. 2.19.

სადაც სიმბოლო O-წითელი მარცვალია; X-ლურჯი მარცვალია A-თეთრი მარცვალია

მძივის კონფიგურაცია მოიცემა მარცვლების ფერების ჩამოთვლით პირველი მარცვლიდან დაწყებული("b"-ლურჯი, "r"-წითელი, "w"-თეთრი). მაგალითად, პირველი მძივი კონფიგურაცია მოიცემა შემდეგი მიმდევრობით:

brbrrrbbbrrrrrbrrbbrbbbbbrrrrb

გავწყვიტოთ მძივი და ერთი ბოლოდან მოვხსნათ მიმდევრობით განლაგებული ერთნაირი ფერის ყველა მარცვალი მანამ, სანამ სხვა ფერის მარცვალი არ შეგვხვდება. ანალოგიურად მოვხსნათ მარცვლები მძივის მეორე ბოლოდანაც (სხვადასხვა ბოლოდან მოხსნილი მარცვლები შეიძლება სხვადასხვა ფერის იყოს). ვიპოვოთ მძივში გაწყვეტის ადგილი, რომლის შემთხვევაშიც ორივე ბოლოდან მოხსნილი მარცვლების რაოდენობათა ჯამი მაქსიმალური იქნება. მაგალითად პირველ მძივში გაწყვეტის ადგილები შეიძლება იყოს მე-9 და მე-10 ან 24-ე და 25-ე მარცვლებს შორის. ორივე შემთხვევაში მოხსნილი მარცვლების რაოდენობა 8-ის ტოლია.

თითოეული ბოლოდან მარცვლების მოხსნის შემთხვევაში თეთრი მარცვალი შეიძლება ჩაითვალოს წითელ ან ლურჯ მარცვლად სიტუაციის მიხედვით, ანუ იგი შეიძლება მოიხსნას როგორც წითელ, ისე ლურჯ მარცვლებთან ერთად.

დავწეროთ პროგრამა, რომელიც:

1.შეიტანს მონაცემებს შესატან მონაცემთა NECKLACE.DAT ფაილიდან, რომლის ყოველი სტრიქონი წარმოადგენს ფერების მიმდევრობის სახით მოცემულ მძივის კონფიგურაციას და ჩაწერს შესატან მონაცემებს გამოსატან მონაცემთა NECKLACE.SOL ფაილში.

2.თითოეული კონფიგურაციისთვის დაადგენს მარცვლების იმ მაქსიმალურ M რაოდენობას, რომელთა მოხსნაც შეიძლება გაწყვეტის შემდეგ და მიუთითებს ერთ-ერთი ოპტიმალური წყვეტის წერტილის მდებარეობას.

3.გამოსატან მონაცემთა NECKLACE.SOL ფაილში გამოიტანს ჯერ M-ის მნიშვნელობას, ხოლო შემდეგ წყვეტის წერტილის მდებარეობას. სხვადასხვა კონფიგურაციისათვის პასუხები ერთი ცარიელი სტრიქონით გამოიყოფა.

მაგალითი:

ფაილი NECKLACE.DAT	ფაილი NECKLACE.SOL
brbrrrbbbrrrrrbrrbbrbbbbbrrrrb	brbrrrbbbrrrrrbrrbbrbbbbbrrrrb 8 between 9and 10
bbwbrrrwbrbrrrrrb	bbwbrrrwbrbrrrrrb 10 between 16 and 17

მითითება. შემავალი მონაცემები წარმოვადგინოთ როგორც წრიული ბმული სია, ანუ შემოვიღოთ მასივი, რომელშიც მითითებული იქნება ყოველი ელემენტის წინა და მომდევნო მეზობლების ნომრები. ამასთან პირველი ელემენტის წინა ელემენტად ჩავთვალოთ ბოლო სიმბოლო, ხოლო ბოლოს ელემენტის მომდევნოდ – პირველი სიმბოლო. ამის შემდეგ განვიხილოთ წყვეტის ყველა შესაძლო წერტილი და თითოეული მათგანიდან ბმულ სიაში ვიმოძრაოთ ორივე მხარეს ვიდრე მეზობლებად ორი განსხვავებული ფერის მარცვალი არ შეგვხვდება (ბუნებრივია, თეთრ ფერზე ამოცანაში მოცემული პირობის დაცვით). მიმდინარე წყვეტის წერტილისათვის ორივე მიმართულებაში მიღებულ რიცხვებს შევკრებთ და ვიპოვით ასეთ რიცხვებს შორის მაქსიმუმს.

2.3 ინტერვალები

(პოლონეთის მოსწავლეთა ოლიმპიადა ინფორმატიკაში, პირველი ეტაპი, 2000-01 წლები)

მოცემულია n დახურული ინტერვალი $[a_i; b_i]$, სადაც i=1,2,...,n. ამ ინტერვალების ჯამი განისაზღვრება, როგორც წყვილ-წყვილად არაგადამკვეთი ინტერვალების ჯამი. ამოცანა იმაში მდგომარეობს, რომ ვიპოვოთ ჯამის ასეთი წარმოდგენა ინტერვალთა მინიმალური რიცხვით. ინტერვალები გამომავალ ფაილში ჩაწერილი უნდა იყოს ზრდადობით. ვიტყვით, რომ [a; b] და [c; d] ინტერვალები დალაგებულნი არიან ზრდადობით, მაშინ და მხოლოდ მაშინ, როცა $a \le b < c \le d$.

დაწერეთ პროგრამა, რომელიც

- PRZ.IN ტექსტური ფაილიდან კითხულობს ინტერვალთა რიგის აღწერას;
- გამოითვლის წყვილ-წყვილად არაგადამკვეთ ინტერვალებს, რომლებიც აკმაყოფილებენ ზემოთ აღწერილ პირობებს;
- გამოთვლილ ინტერვალებს ზრდადობის მიხედვით ჩაწერს PRZ.OUT ფაილში.

შემავალი მონაცემები: PRZ.IN ტექსტური ფაილის პირველ სტრიქონში მოცემულია მთელი რიცხვი n ($3 \le n \le 50000$) – ინტერვალების რაოდენობა. (i+1)-ე სტრიქონში ($1 \le i \le n$) მოცემულია $[a_i; b_i]$ ინტერვალის აღწერა ჰარით გაყოფილი ორი მთელი a_i და b_i რიცხვების სახით, რომლებიც შესაბამისად წარმოადგენენ ინტერვალის დასაწყისს და ბოლოს, $1 \le a_i \le b_i \le 1000000$.

გამომავალი მონაცემები: PRZ.OUT ტექსტური ფაილი უნდა შეიცავდეს წყვილ-წყვილად არაგადამკვეთი ინტერვალების აღწერას. ყოველ სტრიქონში ჩაწერილი უნდა იყოს ერთი ინტერვალის აღწერა, რომელიც წარმოადგენს ჰარით გაყოფილ ორ რიცხვს – ინტერვალის დასაწყისს და ბოლოს შესაბამისად. ინტერვალები გამომავალ ფაილში ჩაწერილი იქნებიან ზრდადობით.

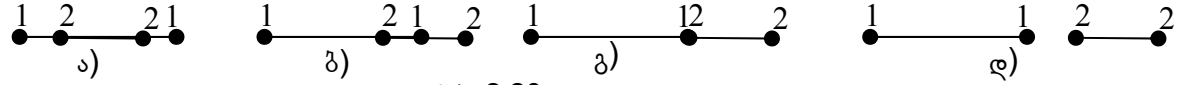
მაგალითი:

PRZ.IN	PRZ.OUT
5 5 6 1 4 10 10 6 9 8 10	1 4 5 10

მითითება. რადგან ინტერვალთა ბოლოების მნიშვნელობები არ აღემატება 1000000-ს, შემოვიღოთ 1000000-ელემენტიანი მასივი და შემომავალი ფაილიდან მონაცემების კითხვისას ინტერვალის დასაწყისის შესაბამის ინდექსზე ჩავწეროთ მისი ბოლოს მნიშვნელობა. თუკი ორი ან მეტი ინტერვალის დასაწყისი ემთხვევა ერთმანეთს, მაშინ ჩავწეროთ მათი ბოლოებიდან უდიდესის მნიშვნელობა. ამოცანაში მოცემული მაგალითისათვის ამ მასივს ექნება სახე:

ინდექსი	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
მნიშვნელობა	*4*	*0*	*0*	*0*	*6*	*9*	*0*	*10*	*0*	*10*

ეს მასივი ერთი გავლით (ე.ი. მილიონი ოპერაციით) შეგვიძლია გადავაქციოთ ბმულ სიად, სადაც უკვე აღარ გვექნება 0-ის ტოლი მნიშვნელობების ელემენტები და მაქსიმუმ 50000 ოპერაციით შეგვეძლება მივიღოთ არაგადამფარავი ინტერვალების მინიმალური სიმრავლე. ეს მარტივად განხორციელდება მეზობელი ინტერვალების შედარებით. ორი მეზობელი ინტერვალისათვის შეიძლება განვიხილოთ ურთიერთგანლაგების 4 შემთხვევა (იხ. ნახ. 2.20):



ნახ. 2.20

ვთქვათ, შემოწმებისას მიმდინარე ინტერვალი არის 1, ხოლო რომელთანაც ვადარებთ – 2. (ა) შემთხვევაში მე-2 ინტერვალი ამოვარდება ბმული სიიდან ყოველგვარი ცვლილების გარეშე და 1 ინტერვალი დარჩება

მიმდინარე ინტერვალად. ასევე ანულირდება მე-2 ინტერვალი (ბ) და (გ) შემთხვევებშიც, ოღონდ ამ დროს პირველი ინტერვალის შესაბამის ელემენტში ჩაიწერება მეორე ინტერვალის შესაბამისი ელემენტის მნიშვნელობა (სხვანაირად რომ ვთქვათ, ინტერვალები გაერთიანდებიან) და აქაც 1 ინტერვალი რჩება მიმდინარე ინტერვალად. (დ) შემთხვევისას ბმული სიაში არანაირი ცვლილება არ მოხდება და მიმდინარე ინტერვალი ხდება 2.

პროცესის დასრულების შემდეგ მიღებული ბმული სია ამოცანის პასუხს წარმოადგენს.

2.4. არითმეტიკული პროგრესია
(USACO, "მწვანე დივიზიონი", 2003-04 წ. იანვარი)

ფერმერმა ჯონმა ძროხებს ჩიპები ჩაუნერგა. შედეგად ის სრულად ფლობს ინფორმაციას საძოვრიდან მათი დაბრუნების შესახებ. ფერმერი გაოცდა, როდესაც შეამჩნია, რომ ძროხათა გარკვეული ჯგუფების წყობა მათი ნომრების მიხედვით ხშირად შეადგენს არითმეტიკულ პროგრესიას. მაგალითად, ერთხელ მან შენიშნა ძროხების შემდეგი წყობა მათი ნომრების მიხედვით: 1 4 3 5 7. ამ მიმდევრობის ქვემიმდევრობა: 1 3 5 7, არითმეტიკულ პროგრესიას წარმოადგენს.

ფერმერს სურს ამ დამთხვევათა შესწავლა. მოცემულია სია, რომელიც შედგება N (1≤N≤2000) ნომრისაგან. ყოველი ძროხის ნომერი არაუარყოფითი, მილიარდზე ნაკლები მთელი რიცხვია. სიაში მოძებნეთ ის უდიდესი მიმდევრობა, რომელიც არითმეტიკულ პროგრესიას წარმოადგენს. პროგრესია შეიძლება იყოს ზრდადი ან კლებადი.

შემავალი მონაცემები: პირველი სტრიქონი შეიცავს ერთ მთელ რიცხვს N. 2..N+1 სტრიქონებში თანმიმდევრობით მოცემულია ძროხების სერიული ნომრები (მეორე სტრიქონში – პირველი ძროხის, მესამეში – მეორე ძროხის და ა.შ.)

გამომავალი მონაცემები: ერთადერთ სტრიქონში მოცემულია უგრძელესი ქვემიმდევრობის წევრთა რაოდენობა.

შემავალი ფაილის მაგალითი (arithprg.in):
```
5
1
4
3
5
7
```

გამომავალი ფაილის მაგალითი (arithprg.out):
```
4
```

მითითება. არითმეტიკული პროგრესიის განსაზღვრისათვის საკმარისია ვიცოდეთ მისი პირველი ორი წევრი, რომელთა როლშიც შესაძლოა მოგვევლინოს საწყისი მიმდევრობის (შევინახოთ a მასივში) ნებისმიერი ორი წევრი. ამიტომ ამოცანის ამოსახსნელად საჭიროა პრაქტიკულად ყველა შესაძლო წყვილის განხილვა. წყვილების საერთო რაოდენობა N^2-ის რიგისაა, ამასთან თითოეული წყვილისათვის პროგრესიის მომდევნო წევრების საპოვნელად თითქმის მთელი მიმდევრობის გადარჩევაა საჭირო, რაც N>1000-სათვის პრაქტიკულად შეუძლებელია.

პროგრესიის წევრთა პოვნა იოლი იქნებოდა, თუკი რაიმე სხვა მასივში (დავარქვათ მას b) მათ შევინახავდით საკუთარი მნიშვნელობების შესაბამის ინდექსებზე. ასეთი შენახვა მოსახერხებელია იმის გამო, რომ ამოცანის პირობის თანახმად თითოეულ ძროხას უნიკალური ნომერი გააჩნია და b მასივში მათი მნიშვნელობები ერთმანეთს არ დაემთხვევა. მაშინ ერთი შემოწმებით შეგვიძლია დავადგინოთ პროგრესიის მომდევნო წევრი არსებობს თუ არა. ამოცანაში მოყვანილი მაგალითისათვის ასეთ მასივს ექნება სახე:

ელემენტის ინდექსი ახალ b მასივში	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
არის თუ არა ელემენტი a მასივში (1 – არის, 0 – არ არის)	*1*	*0*	*1*	*1*	*1*	*0*	*1*	*0*	*0*	*0*
ელემენტის ინდექსი საწყის მიმდევრობაში (a მასივში)	*1*	*0*	*3*	*2*	*4*	*0*	*5*	*0*	*0*	*0*

პირველი განსახილველი წყვილი a მასივიდან იქნება (1,4), ანუ a_1=1 და a_2=4. ამ დროს პროგრესიის სხვაობა d=3, ამიტომ მომდევნო წევრი უნდა იყოს $a_3=a_2+d=4+3=7$. ვამოწმებთ b მასივის მე-7 ელემენტს: მისი მნიშვნელობა 1-ის ტოლია, ამასთან მისი ინდექსი a მასივში აღმოჩნდება a_2 ელემენტის ინდექსს, მაშასადამე პროგრესიის მომდევნო ელემენტი არსებობს. ამის შემდეგ განვიხილავთ $a_4=a_3+d=7+3=10$, ვამოწმებთ b მასივის მე-10 ელემენტს – ივი 0-ის ტოლია და ძებნას ვწყვეტთ. მივიღეთ 3 წევრისაგან შედგენილი პროგრესია 1, 4, 7. ამის შემდეგ განვიხილავთ მომდევნო წყვილს a მასივიდან: a_1=1 და a_2=3. ამ წყვილისათვის ანალოგიური მოქმედებებით მიიღება 4 წევრისაგან შედგენილი პროგრესია 1, 3, 5, 7 და ა.შ.

ამ ალგორითმის განხორციელებას ხელს უშლის ის გარემოება, რომ a მასივის ელემენტთა მნიშვნელობების დიაპაზონი ძალზე დიდია: 1-დან მილიარდამდე. ცხადია, რომ b მასივს ასეთი ზომებით ვერ აღვწერთ. თუმცა a მასივის ელემენტთა რაოდენობა 2000-ს არ აღემატება და ამიტომ აღნიშნული პრობლემა იოლად გადაიჭრება ჰეშირების საშუალებით. ჰეშირებისათვის მივმართოთ ნაშთიანი გაყოფის მეთოდს (იხილე თავი 2.4.), ამასთან ჰეშ-ფუნქცია ისე უნდა შეირჩეს, რომ b მასივში კოლიზია არ მივიღოთ. კოლიზიის თავიდან ასაცილებლად გამყოფები ერთი ციკლით გადავარჩიოთ და თუ b მასივის შევსებისას 1-იანის ჩაწერა ორჯერ მოგვიწია რომელიმე ინდექსზე – ჰეშირება შევწყვიტოთ, b მასივი გავანულოთ და აღნიშნულ ციკლში გამყოფის ახალ მნიშვნელობაზე გადავიდეთ. ასეთი ალგორითმით სხვადასხვა საწყისი მონაცემისათვის სხვადასხვა ჰეშ-ფუნქცია შეირჩევა. გამყოფად a მასივის ელემენტთა რაოდენობასთან შედარებით დიდი რიცხვის არჩევა (ვთქვათ, 150000) სწრაფად აჯამებს b მასივს კოლიზიის გარეშე. ჰეშირების შემდეგ შესაძლებელი ხდება ზემოთ აღწერილი ალგორითმით პროგრესიების გადარჩევა. აქვე შევნიშნოთ, რომ ცხრილში ნაჩვენები მონაცემების გარდა საჭიროა b მასივში შევინახოთ a მასივის ელემენტთა საწყისი მნიშვნელობებიც. მაგალითად, თუ a_{71}=873678323; a_{232}=186544938, a_{698}=439567352 და ჰეშ-ფუნქციის მნიშვნელობაა 150001, მაშინ შესაბამის მონაცემებს b მასივში ექნება სახე:

ელემენტის ინდექსი ახალ b მასივში	...	64422	...	72499	...	93695	...
არის თუ არა ელემენტი a მასივში	...	1	...	1	...	1	...
ელემენტის ინდექსი საწყის მიმდევრობაში	...	698	...	71	...	232	...
ელემენტის მნიშვნელობა		439567352		873678323		186544938	

განმარტებისათვის: 439567352 mod 150001=64422, 873678323 mod 150001=72499 და 186544938 mod 150001=93695.

3. გადარჩევა და რეკურსია

3.1. გადარჩევა

მრავალ გამოყენებით ამოცანაში ოპტიმალური ამონახსნი მოსაძებნია ვარიანტთა ძალიან დიდ, თუმცა სასრულ რაოდენობაში. პროცესს, რომლის დროსაც ამონახსნის პოვნისათვის საჭიროა ყველა შესაძლო ვარიანტის განხილვა და ერთმანეთთან შედარება, **გადარჩევას** უწოდებენ. ამ მეთოდით ხდება რიგი კომბინატორული ამოცანების ამოხსნა, რომლებიც ეხებიან მიმდევრობებს, გადანაცვლებებს, ქვესიმრავლეებს და ა.შ.

გადარჩევის სქემის აგება ძირითადად დამოკიდებულია ორ მომენტზე: ა) უნდა განისაზღვროს **გადასარჩევ ელემენტთა რიგი**, ანუ ცალსახად იქნას განსაზღვრული, თუ რომელი მათგანია პირველი და რომელი – ბოლო; ბ) ნებისმიერი ელემენტისათვის ცალსახად იქნას განსაზღვრული უშუალოდ მისი მომდევნო ელემენტი, ანუ მოცემული x_1 ელემენტისათვის ყოველთვის ავაგოთ ისეთი x_2, რომ $x_1<x_2$ და x_1-სა და x_2-ს შორის სხვა ელემენტი არ იარსებებს.

გადარჩევის პროცესში ხშირად გვიწევს ისეთი კომბინატორული ობიექტების გამოყენება, როგორებიცაა **მიმდევრობა** და **გადანაცვლება**. განვიხილოთ ზოგადი სახის მქონე ასეთი ამოცანა: 1,2,...,m რიცხვებისაგან შევადგინოთ n სიგრძის მქონე ყველანაირი მიმდევრობა. გადასარჩევ ელემენტთა რიგში პირველი ელემენტად უმჯობესია ჩავთვალოთ n ცალი 1-ისაგან შემდგარი მიმდევრობა, ხოლო ბოლო ელემენტად – n ცალი m-ისაგან შემდგარი მიმდევრობა (კონკრეტულ ამოცანაში რიგი შესაძლოა პირუკუც განვიხილოთ). i-ური მიმდევრობიდან მისი მომდევნო (i+1)-ს მისაღებად i-ური მიმდევრობის ბოლო წევრი უნდა გავზარდოთ 1-ით, ხოლო თუ მისი გაზრდა არ ხერხდება, ანუ თუ ეს წევრი m-ის ტოლია, მაშინ 1-ით უნდა გაიზარდოს მის მარცხნივ მდებარე m-ზე ნაკლები წევრებიდან უახლოესი, ხოლო ამ უკანასკნელის მარჯვნივ მდებარე ყველა წევრი 1-ის ტოლი უნდა გახდეს. პროცესი წყდება, თუკი არცერთი წევრის გაზრდა არ ხერხდება. მაგალითად, თუ m=7, n=5 და პირველ მიმდევრობად ჩავთვალეთ (1,1,1,1,1), მაშინ მეოთხე მიმდევრობა იქნება (1,1,1,1,4), ბოლო – (7,7,7,7,7), ხოლო (3,2,4,7,7) მიმდევრობის მომდევნო მიმდევრობა იქნება (3,2,5,1,1). სულ მიმდევრობების რაოდენობა იქნება m^n (ჩვენს მაგალითში 75). მოვიყვანოთ ამ ალგორითმის რეალიზაცია. X მასივი შევავსოთ 1-იანებით, ყოველი ახალი მიმდევრობის ფორმირება მოვახდინოთ X-ში და დავბეჭდოთ:

```
READ (m,n);
FOR i=1 TO n { X[i]=1   }
j=n;      WRITE (X);
WHILE j>0 {
     IF X[j]<m THEN { X[j]=X[j]+1;      j=n;      WRITE (X)   }
               ELSE  { X[j]=1;       j=j-1 }
 }
```

გადანაცვლების შემთხვევაში საკითხი ასე დაისმება: 1,2,...,m რიცხვებისაგან შევადგინოთ m სიგრძის მქონე ყველანაირი გადანაცვლება ისე, რომ ყოველ მათგანში 1,2,...,m რიცხვებისაგან

თითოეული მხოლოდ ერთხელ შედიოდეს. ბუნებრივია, აქაც შეგვიძლია გადანაცვლებათა რიგი დავადგინოთ. მაგალითად, m=5-სათვის პირველი ელემენტი იქნება (1,2,3,4,5), ბოლო – (5,4,3,2,1), ხოლო i-ური გადანაცვლებიდან მისი მომდევნო (i+1)-ს მისაღებად ასე ვიმოქმედოთ: მოვძებნოთ უდიდესი ისეთ წევრებს შორის, რომელთა მარჯვნივ მდებარეობს მასზე მეტი თუნდაც ერთი წევრი. ანუ მოვძებნოთ ისეთი i, რომლისთვისაც X[i] ნაკლები იქნება X[i+1]-დან X[m]-მდე ერთ-ერთზე მაინც (ასეთი არ მოიძებნება მხოლოდ უკანასკნელ გადანაცვლებაში). ამის შემდეგ მოძებნილი წევრის მარჯვნივ მოთავსებულ X[i+1],…,X[m] წევრებს შორის მოვძებნოთ უმცირესი, რომელიც X[i]-ზე მეტია და მას და X[i]-ს გავუცვალოთ ადგილები. დასასრულს კი i+1,…,m ნომრების მქონე წევრები ზრდადობით უნდა დავალაგოთ, რათა გადანაცვლება უმცირესი იყოს. ამ საკითხის მოგვარებას ის აიოლებს, რომ X[i+1],…,X[m] წევრები კლებადობით არიან დალაგებულნი.

გადანაცვლებათა საერთო რაოდენობაა m-ის ფაქტორიალი: m!=m*(m-1)*...*2*1. მოვიყვანოთ ამ ალგორითმის რეალიზაცია. მასში გამოყენებულია პროცედურა SWAP, რომელიც ადგილს უცვლის მასივის 2 ელემენტს. გადანაცვლების ფორმირება ხდება X მასივში, რომელშიც თავდაპირველად ჩაიწერება გადანაცვლება (1,2,...,m).

```
READ (m);
FOR k=1 TO m {   X[k]=k  }
i=m-1;
WHILE i>0 {
    WRITE (X)
    i=m-1;
    WHILE (X[i]>X[i+1]) {  i=i-1    }
    IF i>0 THEN {
        j=i+1;
        WHILE (j<m) AND (X[j+1]>X[i])  {  j=j+1    }
        SWAP(X[i],X[j]);
        FOR j=i+1 TO (m+i) DIV 2 { SWAP(X[j],X[m-j+i+1])   }
                }                }
```

მიმდევრობებისა და გადანაცვლებების გარდა კომბინატორიკაში განიხილავენ ე.წ. "დაყოფის" ამოცანებს, სადაც მოცემული ნატურალური რიცხვი N-სათვის უნდა ვიპოვოთ მისი ყველა განსხვავებული დაყოფა (შესაკრებებად დაშლა) ნატურალურ რიცხვებად. ამასთან განსხვავებულად არ ითვლებიან ის დაყოფები, რომელთაც მხოლოდ შესაკრებთა რიგი განასხვავებს საწყის დაყოფად შეგვიძლია განვიხილოთ ყველა 1-იანი, ხოლო უკანასკნელ დაყოფად – თავად რიცხვი N.

```
READ(N)
L=N
FOR k=1 to L {   X[k]=1      }
WRITE (X)
REPEAT
        i=L-1;    s=X[L];
        WHILE (i>1) and (X[i-1]≤X[i]) {
                s=s+X[i];     i=i-1   }
        X[i]=X[i]+1;      L=i+s-1
        FOR j=i+1 TO  L  { X[j]=1    }
        WRITE (X)
UNTIL L=1
```

პრაქტიკაში ხშირად გვიხდება სხვადასხვა კომბინატორული ფორმულის გამოყენებაც. მაგ. პასუხის გასაცემად შეკითხვაზე: "რამდენი განსხვავებული ხუთეული შეგვიძლია შევადგინოთ 20 რიცხვისაგან?" დაგვჭირდება ფორმულა

$$C_n^m = \frac{n!}{m!*(n-m)!} \qquad C_{20}^5 = \frac{20!}{5!*15!} = \frac{20*19*18*17*16}{5*4*3*2*1} = 15504$$

განვიხილოთ ასეთი ამოცანა:

3.1. მხიარული ვაჭრობა

(USACO, 2002 წელი, დეკემბერი, "ნარინჯისფერი" დივიზიონი)

ფერმერმა ჯონმა საკუთარ ნახირს საშობაო საჩუქრები უნდა უყიდოს. ჯონს აქვს 10 დოლარი (ეს თანხა ასე აღინიშნება – US$10), ხოლო სათამაშოები, რომლებიც მან უნდა შეიძინოს ღირს შესაბამისად US$5, US$2, US$1, US$1/2 (ნახევარი დოლარი) და US$1/4 (მეოთხედი დოლარი). ჯონის ხელთ არსებული თანხა ეყოფა ორ $5-დოლარიან სათამაშოს, ან 20 $1/4-დოლარიან, 2 $2-დოლარიან და 1 $1-დოლარიან სათამაშოს ერთად. ფერმერ ჯონს სურს გაიგოს $10 დოლარად რამდენი განსხვავებული ვარიანტით შეიძლება სათამაშოების ყიდვა (ფული მთლიანად უნდა დაიხარჯოს).

დაწერეთ პროგრამა, რომელიც გამოითვლის იმ განსხვავებული ვარიანტების რაოდენობას, რომლითაც ფერმერ ჯონს შეუძლია სათამაშოების შეძენა ზემოთ მითითებულ ფასებში, თუკი ის მთლიანად დახარჯავს N (N მთელია, 1<=N<=150) დოლარს.

შემავალი მონაცემების ფორმატი: ერთადერთი სტრიქონი ერთი რიცხვით: N.

გამომავალი მონაცემების ფორმატი: ერთადერთი სტრიქონი ერთი რიცხვით, რომელიც წარმოადგენს იმ განსხვავებული ვარიანტების რაოდენობას, რომლითაც შეიძლება დახარჯული იქნას N დოლარი ზემოთ მითითებული ფასების მქონე სათამაშოების შესაძენად.

შემავალი მონაცემების მაგალითი (ფაილი shop.in):	გამომავალი მონაცემების მაგალითი (ფაილი shop.out):
10	343

*მითითება. ასეთ ამოცანებს სხვაგვარად "გადახურდავების" ამოცანებსაც უწოდებენ. თავისი არსით ისინი დაყოფის ამოცანებს წარმოადგენენ, თუმცა ამჯერად მისი ამოხსნისათვის გამოვიყენოთ უფრო მარტივი ალგორითმი და უბრალოდ შესაბამისი რაოდენობის ერთმანეთში ჩადგმული ციკლებით გადავარჩიოთ ყველა შესაძლო ვარიანტი, რომელიც ჯამში ნაკლებია ან ტოლი $N დოლარზე. გადასარჩევ ელემენტთა რიგის შესაქმნელად ასე მოვიქცეთ: განვსაზღვროთ თითოეული კუპიურის მაქსიმალური შესაძლო რაოდენობა $N დოლარის მისაღებად: $5 – N div 5, $2 – N div 2, 1$ – N, $1/2 – 2*N, $1/4 – 4*N. თავდაპირველად ყველა კუპიურის რაოდენობა 0-ის ტოლად ჩავთვალოთ და ყველაზე მცირე კუპიურის ($1/4 დოლარი) რაოდენობა ვზარდოთ ერთით მანამ, სანამ არ მივიღებთ $N დოლარს. შემდეგ რაიმე მთვლელი, რომლის საბოლოო მნიშვნელობაც პასუხი იქნება, გავზარდოთ ერთით, სიდიდით მომდევნო კუპიურის ($1/2 დოლარი) რაოდენობაც გავზარდოთ ერთით და $1/4 დოლარის რაოდენობათა ზრდა ისევ დავიწყოთ ნულიდან და ა.შ. მათემატიკურად რომ ვთქვათ, თუ განვიხილავთ კუპიურების რაოდენობათა ხუთეულებს (x_5, x_2, x_1, $x_{0.5}$, $x_{0.25}$), $10 დოლარისათვის პირველი ხუთეული იქნება (0,0,0,0,0), მეორე – (0,0,0,0,1), ორმოცდამეერთე – (0,0,0,0,40), ორმოცდამეორე – (0,0,0,1,0) და ა.შ. ხუთეულში თითოეული წევრის მნიშვნელობა ვერ გადაამეტებს შესაბამისი კუპიურის მაქსიმალურ რაოდენობას. ამასთან ყოველ ბიჯზე შემოწმდება კუპიურების საერთო ჯამის 10-თან ტოლობა. ასეთი წესით ხუთეულების თანმიმდევრობა ცალსახად განისაზღვრება. მაგალითად (1,1,2,1,2)-ის შემდეგ მოდის (1,1,2,2,0), რადგან (1,1,2,1,3)-ის საერთო ჯამი 10-ს აღემატება.*

*$10-სათვის განსახილველ ვარიანტთა საერთო რაოდენობა არ აღემატება 2*5*10*20*40=80000-ს, ხოლო 150-სათვის (N-ის მაქსიმალური მნიშვნელობა ამოცანაში) 202 მილიონზე მეტია. ამ უკანასკნელი რიცხვიდან გამომდინარე აუცილებელია გადარჩევის შემცირება, რისთვისაც თითოეულ ციკლში ხდება გარკვეული შემოწმებები და ვარიანტთა საკმაოდ დიდი რაოდენობა იგნორირდება.*

```
readln (k);
p1=k div 5; p2=k div 2;    n=0;
for i1=0 to p1    {
   n1=i1*5;
   for i2=0 to p2    {
      n2=n1+i2*2;
      if n2>k then break;
```

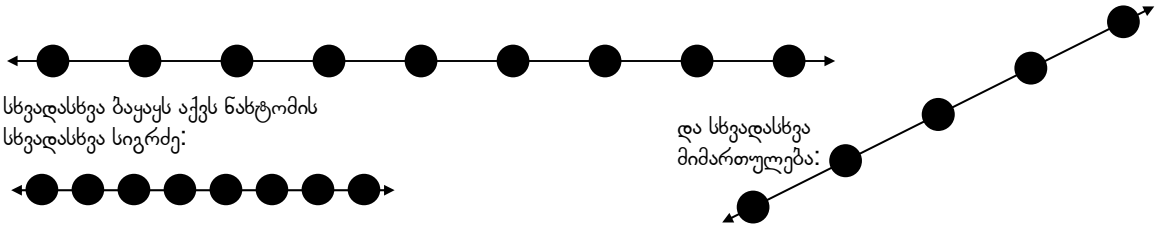
```
    for i3=0 to k     {
        n3=n2+i3;
        if n3>k then break;
       for i4=0 to k*2     {
          if n3+i4*0.5<=k then n=n+1 else break;
          }        }         }          }
write (n);
```

განვიხილოთ კიდევ ერთი ამოცანა გადასარჩევი ვარიანტების შემცირებაზე.

3.2. მავნე ბაყაყი

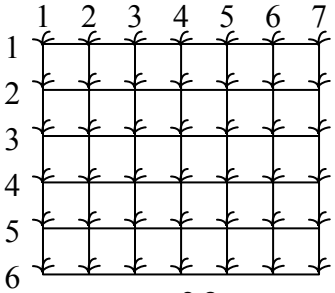
(მოსწავლეთა საერთაშორისო ოლიმპიადა, 2002 წელი, სამხრეთ კორეა)

პატარა მავნე ბაყაყები _ სახელად "ჩენგაიგურები", ლეგენდარულები გახდნენ მთელს კორეაში. ისინი ღამ-ღამობით ხტიან ბრინჯის ყანებში და თელავენ ბრინჯის ნარგავებს. თუკი დილით დააკვირდებით ბაყაყების მიერ დაზიანებულ ნარგავებს, შეამჩნევთ, რომ ბაყაყი ყოველთვის მოძრაობს სწორ ხაზზე და მისი ყოველი ნახტომის სიგრძე ერთნაირია:

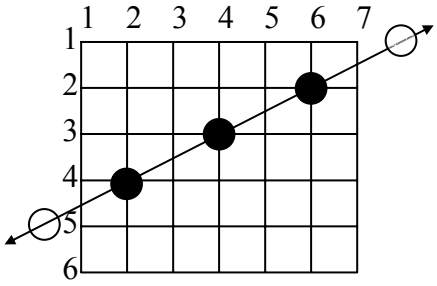


ნახ. 3.1

ბრინჯის ნარგავები ყანაში განლაგებულია პერპენდიკულარული ხაზების გასწვრივ ერთმანეთისაგან თანაბარ მანძილზე ისე, როგორც ეს ნახაზი 3.2-ზეა ნაჩვენები. მავნე ბაყაყები მოძრაობას იწყებენ ყანის გარეთ მისი ერთ-ერთი მხრიდან და ამთავრებენ ასევე ყანის გარეთ სხვა მხარეს, როგორც ეს ნაჩვენებია ნახაზი 3.3-ზე.

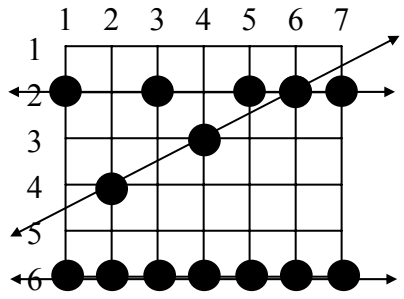


ნახ. 3.2

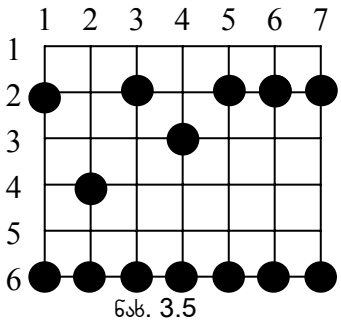


ნახ. 3.3

ყანაში შეიძლება ხტოდეს ბევრი ბაყაყი ერთი ნარგავიდან მეორისაკენ. ყოველ ნახტომზე ბაყაყი თელავს მორიგ ნარგავს, როგორც ეს ნახ. 3.4-ზეა ნაჩვენები. შევნიშნოთ, რომ ღამის განმავლობაში ერთი და იგივე მცენარე შეიძლება ერთზე მეტმა ბაყაყმა გათელოს. რასაკვირველია, თქვენ ვერ დაინახავთ სწორ ხაზებს, რომლის გასწვრივაც ხტოდნენ ბაყაყები და ასევე ვერ დაინახავთ მოძრაობის რაიმე კვალს ყანის საზღვრებს გარეთ. ნახ. 3.4-ზე ნაჩვენები მაგალითისათვის თქვენ დაინახავთ დაზიანებული ყანის ისეთ სურათს, რომელიც გამოსახულია ნახ. 3.5-ზე.



ნახ. 3.4



ნახ. 3.5

ნახ. 3.5-ის მიხედვით თქვენ შეგიძლიათ აღადგინოთ ყველა შესაძლო გზა, რომლითაც ბაყაყებს შეეძლოთ გადაადგილებულიყვნენ ბრინჯის ყანაში. ამასთან, თქვენ გაინტერესებთ მხოლოდ ის ბაყაყები, რომლებმაც ყანის გადავლის დროს დააზიანეს არანაკლებ სამი ნარგავისა. ასეთ გზას *ბაყაყის გზა* ვუწოდოთ.

შეიძლება შევნიშნოთ, რომ სამი გზა, რომლებიც ნაჩვენებია ნახ. 3.4-ზე, *ბაყაყის გზებს* წარმოადგენენ (გაითვალისწინეთ, რომ იგივე ნახაზზე არსებობს *ბაყაყის გზების* სხვა ვარიანტებიც). ვერტიკალური გზა, რომელიც გადის პირველი სვეტის გასწვრივ, შესაძლოა ყოფილიყო *ბაყაყის გზა* სიგრძით 4, მაგრამ ამ შემთხვევაში დაზიანდა ბრინჯის მხოლოდ ორი ნარგავი, ამიტომ ასეთი გზა ჩვენ არ გვაინტერესებს. განვიხილოთ დიაგონალური გზა, რომელიც გადის მე-2 სტრიქონისა და მე-3 სვეტის, მე-3 სტრიქონისა და მე-4 სვეტის და მე-6 სტრიქონისა და მე-7 სვეტის გადაკვეთებზე. ამ გზაზე იმყოფება სამი დაზიანებული ნარგავი, მაგრამ გზა არათანაბარია და შეიცავს სხვადასხვა სიგრძის ინტერვალებს. ამიტომ, ეს გზაც არ წარმოადგენს *ბაყაყის გზას*. აგრეთვე შევნიშნოთ, რომ ბაყაყის მოძრაობის ხაზზე შეიძლება შეგვხვდეს სხვა ბაყაყის მიერ დაზიანებული ნარგავები (მაგალითად, ნახ. 3.5-ზე (2,6) წერტილში არსებული ნარგავი ჰორიზონტალურ გზაზე მეორე სტრიქონის გასწვრივ). ყანაში შეიძლება არსებობდეს გადათელილი ნარგავები, რომლებიც არცერთ *ბაყაყის გზას* არ ეკუთვნის.

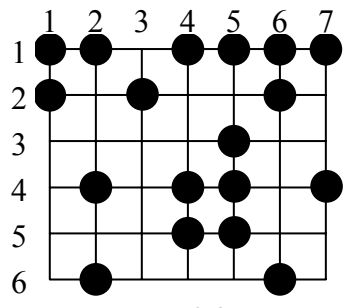
დაწერეთ პროგრამა, რომელიც განსაზღვრავს ყველა შესაძლო *ბაყაყის გზას* და იპოვის მათ შორის ისეთს, რომელიც გათელილი ნარგავების ყველაზე დიდ რაოდენობას შეიცავს. ნახ. 3.5-ზე ასეთი გზა მდებარეობს მე-6 სტრიქონის გასწვრივ და პასუხი არის რიცხვი 7.

შესატანი მონაცემები: თქვენმა პროგრამამ უნდა განახორციელოს სტანდარტული წაკითხვა. პირველი სტრიქონი შეიცავს ორ მთელ R და C რიცხვს ($1 \le R,C \le 5000$), რომლებიც შეესაბამება ბრინჯის ყანაში სტრიქონებისა და სვეტების რაოდენობას. მეორე სტრიქონში ჩაწერილია ერთი მთელი N რიცხვი ($3 \le N \le 5000$) – გათელილი ნარგავების რაოდენობა. ყოველი მომდევნო N სტრიქონიდან თითოეულში მოცემულია თითო ჰარით გამოყოფილი ორი მთელი რიცხვი, რომლებიც გათელილი ნარგავის პოზიციას განსაზღვრავენ ($1 \le$ *სტრიქონის ნომერი* $\le R$), ($1 \le$ *სვეტის ნომერი* $\le C$). ყოველი დაზიანებული ნარგავი ფაილში მხოლოდ ერთხელაა აღწერილი.

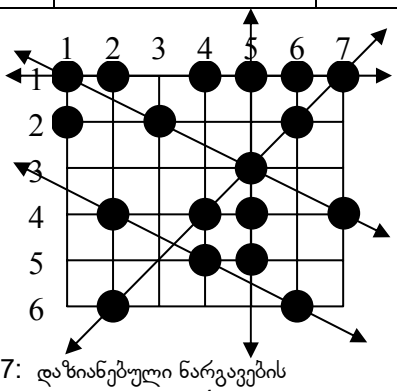
გამოსატანი მონაცემები: თქვენმა პროგრამამ უნდა განახორციელოს სტანდარტული გამოტანა. გამოტანა ხდება 1 სტრიქონში, რომელიც ერთადერთი რიცხვისაგან შედგება და განსაზღვრავს გათელილი ნარგავების რაოდენობას იმ *ბაყაყის გზაზე*, რომელმაც ყანას ყველაზე მეტი ზიანი მიაყენა. თუკი ასეთი გზა არ არსებობს, უნდა გამოიტანოთ 0.

შესატანი და გამოსატანი მონაცემების მაგალითები:

მაგალითი 1 (ნახ. 3.5-ისათვის)			მაგალითი 2 (ნახ. 3.6-ისათვის)	
შეტანა	გამოტანა		შეტანა	გამოტანა
6 7 14 2 1 6 6 4 2 2 5 2 6 2 7 3 4 6 1 6 2 2 3 6 3 6 4 6 5 6 7	7		6 7 18 1 1 6 2 3 5 1 5 4 7 1 2 1 4 1 6 1 7 2 1 2 3 2 6 4 2 4 4 4 5 5 4 5 5 6 6	4



ნახ. 3.6.



ნახ. 3.7: დაზიანებული ნარგავების მაქსიმალური რაოდენობაა 4.

შეფასება

თუ თქვენს პროგრამას ტესტის გავლისათვის განსაზღვრულ დროში გამოაქვს სწორი პასუხი, მაშინ მიიღებთ ქულათა მაქსიმალურ რაოდენობას, წინააღმდეგ შემთხვევაში კი – 0 ქულას.

მითითება. თავდაპირველად შევნიშნოთ, რომ ბაყაყს გზა ცალსახად განისაზღვრება მისი ორი მეზობელი წერტილით (აქაც და შემდგომშიც ტერმინ "წერტილის" ქვეშ "ბუჩქის" ადგილმდებარეობა იგულისხმება). მართლაც თუ ჩვენ გვაქვს ორი მეზობელი წერტილი, მაშინ შეგვიძლია ცალსახად აღვადგინოთ წინა და მომდევნო წერტილები, ან დავასკვნათ, რომ წინა ან მომდევნო წერტილი არ არსებობს და შესაბამისად ჩვენ ვიმყოფებით გზის დასაწყისში ან ბოლოში. გზის აღსადგენად საჭიროა მისი გაგრძელება იმავე სიგძის მონაკვეთით, რა სიგრძისაც არის მოცემული წერტილების შემაერთებელი მონაკვეთი, ხოლო ამ ოპერაციის განმეორებით შესაძლებელია მთელი გზის აღდგენა.

ზემოთ მოყვანილი დებულებებიდან გამომდინარეობს შემდეგი მარტივი ალგორითმი: განვიხილოთ მოცემული წერტილების ყველა შესაძლო წყვილი, ჩავთვალოთ რომ ისინი რომელიღაც გზაზე მეზობელი წერტილები არიან და აღვადგინოთ შესაბამისი გზა ყოველი წყვილისათვის. პარალელურად შევამოწმოთ, გათელილია თუ არა ბრინჯის ნარგავი ამ გზის ყოველ წერტილში და არის თუ არა გზის სიგრძე (ანუ გათელილი ნარგავების რაოდენობა) სამზე არანაკლები. ცხადია, რომ წერტილთა ყველა წყვილს შორის აუცილებლად განხილული იქნება ყველაზე გრძელი გზაც, ამიტომ ნაპოვნ გზათა შორის მაქსიმალური სიგრძის პოვნა მოგვცემს სწორ პასუხს ჩვენს ამოცანაზე.

დიდი მნიშვნელობა აქვს როგორ შევამოწმებთ, გათელილია მორიგი ნარგავი თუ არა. თუკი ამის გასაგებად ჩვენ გადავამოწმეთ გათელილი ნარგავების მთელი სია – პროგრამა ძალიან ნელა იმუშავებს. ამიტომ საჭიროა შემოვიღოთ $R \times C$ ზომის მასივი (თითო ელემენტი თითო ნარგავისათვის), რომლის ყველა უჯრედს თავდაპირველად შევასებთ მნიშვნელობით – "არაა გათელილი". შემდეგ შესატანი მონაცემების ერთი გავლით მასივში მოვნიშნოთ გათელილი ნარგავები. ახლა ნარგავის მდგომარეობის დასადგენად მასივის მხოლოდ ერთი ელემენტის შემოწმებაა საჭირო და პროგრამის სიჩქარე მკვეთრად მოიმატებს. თუმცა მხოლოდ ეს გაუმჯობესება არ იძლევა ოპტიმალურ ამოხსნას, რადგან ზოგიერთ ბაყაყს გზას რამდენჯერმე ვამოწმებთ (როდესაც ამ გზის შემადგენელი წერტილების ყველა წყვილს ვიხილავთ). შესრულებული ოპერაციების რიცხვი N^3-ის პროპორციულია, რადგან წერტილთა წყვილების რაოდენობა N^2-ის რიგისაა, ხოლო თითოეული წყვილისათვის ჩვენ გვიწევს უარეს შემთხვევაში N რიგის ნარგავის გადარჩევა. ასეთი ალგორითმით დაწერილი პროგრამა ვერაგროვებდა ქულათა 40%-ზე მეტს, რადგან ტესტების დიდი ნაწილისათვის ამეტებდა სამუშაო დროის ლიმიტს.

ალგორითმის გასაუმჯობესებლად შევნიშნოთ, რომ წერტილთა ყველა წყვილის შემოწმება საჭირო არ არის. საკმარისია შევამოწმოთ მხოლოდ ის წერტილები, რომელთათვისაც მათი შემაერთებელი მონაკვეთი მარცხნივ გავრძელებისას გადის მინდვრის საზღვრებს გარეთ. სხვაგვარად რომ ვთქვათ, უნდა შევამოწმოთ მხოლოდ ის წერტილები, რომლებიც წარმოადგენენ საძებნი გზის უკიდურეს მარცხენა წერტილებს. ცხადია, სწორ პასუხს ასეთი გადარჩევითაც მივიღებთ, რადგან როდესმე შევამოწმებთ უკიდურეს მარცხენა წერტილთა წყვილს მაქსიმალურ გზაზეც. ამგვარ ალგორითმში ცალკეა შესამოწმებელი ვერტიკალური გზები, რადგან მათში უნდა შემოწმდეს არა ორი მარცხენა, არამედ – ორი ზედა წერტილი.

აღწერილი ალგორითმი აგროვებს ქულათა 100%-ს. ბოლოს კი აღვნიშნოთ, რომ ამოცანა შეიძლება ასევე ამოიხსნას გრაფთა თეორიის ან დინამიური პროგრამირების საშუალებითაც.

განვიხილოთ კიდევ ერთი ამოცანა, რომლის ამოხსნასაც გადარჩევის არატრივიალური სქემა სჭირდება.

3.3. ავტობუსების გაჩერებები

(მოსწავლეთა საერთაშორისო ოლიმპიადა, 2002 წელი, სამხრეთ კორეა)

ქალაქ იონგ-ინში იგეგმება საავტობუსო ქსელის მშენებლობა ავტობუსების N რაოდენობის გაჩერებით. ყოველი გაჩერება მდებარეობს გზაჯვარედინზე. იონგ-ინის რუკა წარმოადგენს თანაბარი ზომის კვადრატული კვარტლებისგან შედგენილ ბადეს. N რაოდენობის გაჩერებიდან უნდა მოხდეს ორი გადასაჯდომი H_1 და H_2 პუნქტის შერჩევა, რომლებიც ერთმანეთს ავტობუსის პირდაპირი მარშრუტით დაუკავშირდება, ხოლო დანარჩენი $N-2$ რაოდენობის გაჩერება უშუალოდ უნდა დაუკავშირდეს ამ ორი პუნქტიდან მხოლოდ ერთ-ერთს და არ უნდა დაუკავშირდეს არცერთს დანარჩენი გაჩერებებიდან.

მანძილი ნებისმიერ ორ გაჩერებას შორის განისაზღვრება როგორც უმოკლესი გზა ერთიდან მეორეში მოსახვედრად ქალაქის ქუჩების გავლით. ეს ნიშნავს, რომ თუ გაჩერება წარმოდგენილია (x, y) კოორდინატთა წყვილით, მაშინ (x_1, y_1) და (x_2, y_2) კოორდინატების მქონე ორ გაჩერებას შორის მანძილი $|x_1-x_2|+|y_1-y_2|$ იქნება. თუ A და B გაჩერებები ერთმანეთს უკავშირდება ერთი და იგივე გადასაჯდომი H_1 პუნქტით, მაშინ A-დან B-ში მისასვლელი გზის სიგრძე წარმოადგენს A-დან H_1-მდე და H_1-დან B-მდე მანძილების ჯამს. თუ A და B გაჩერებები ერთმანეთს უკავშირდება სხვადასხვა გადასაჯდომი პუნქტებით (მაგალითად, A დაკავშირებულია H_1-თან და B დაკავშირებულია H_2-თან), მაშინ A-დან B-ში მისასვლელი გზის სიგრძე წარმოადგენს A-დან H_1-მდე, H_1-დან H_2-მდე და H_2-დან B-მდე მანძილების ჯამს.

იონგ-ინის დამპროექტებლებს სურთ დარწმუნებულნი იყვნენ, რომ ყოველ მოქალაქეს შეეძლება მოხვდეს ქალაქის ნებისმიერ წერტილში რაც შეიძლება სწრაფად. ამიტომ მათ სურთ, გადასაჯდომ პუნქტებად შეარჩიონ ისეთი ორი გაჩერება, რომ მიღებულ საავტობუსო ქსელში ყველაზე გრძელი გზა ნებისმიერ ორ გაჩერებას შორის რაც შეიძლება

მოკლე იყოს (გადასაჯდომ პუნქტებად ნებისმიერი სხვა წყვილის არჩევისას მიღებულ ასევე ყველაზე გრძელ გზებს შორის).

გადასაჯდომი პუნქტების არჩევის *P* ვარიანტი *Q* ვარიანტზე უკეთესია, თუ *P* ვარიანტში ნებისმიერ ორ გაჩერებას შორის ყველაზე გრძელი გზის სიგრძე ნაკლებია *Q* ვარიანტში ნებისმიერ ორ გაჩერებას შორის ყველაზე გრძელი გზის სიგრძეზე.

შესატანი მონაცემები

თქვენმა პროგრამამ უნდა განახორციელოს სტანდარტული წაკითხვა. პირველი სტრიქონი შეიცავს ერთ მთელ დადებით N რიცხვს (2≤*N*≤500) – გაჩერებათა რაოდენობას. ყოველი მომდევნო *N* რაოდენობის სტრიქონიდან თითოეულში მოცემულია თითო ჰარით გამოყოფილი ორი მთელი დადებითი *x* და *y* რიცხვი (1≤x,y≤5000) – ავტობუსების გაჩერებათა კოორდინატები. არცერთი ორი გაჩერება არ შეიძლება იყოს წარმოდგენილი კოორდინატთა ერთი და იგივე წყვილით.

გამოსატანი მონაცემები

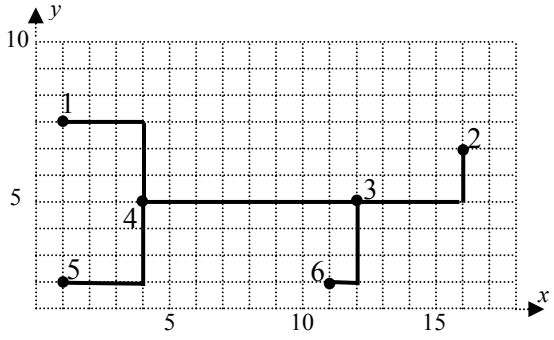
თქვენმა პროგრამამ უნდა განახორციელოს სტანდარტული გამოტანა. გამოტანა ხდება 1 სტრიქონში, რომელიც ერთადერთი რიცხვისაგან – შესატანი მონაცემებისათვის ყველაზე გრძელ გზათა შორის უმოკლესის სიგრძისაგან შედგება.

შესატანი და გამოსატანი მონაცემების მაგალითები

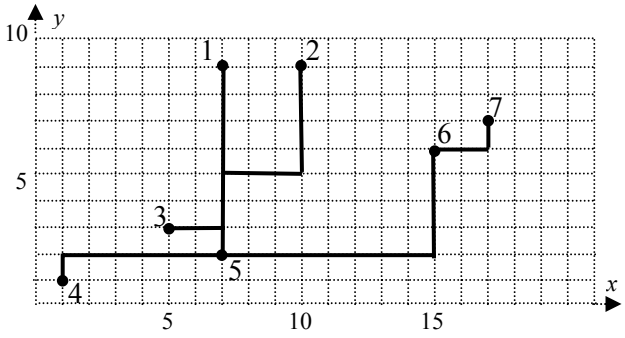
მაგალითი 1 (ნახაზი 4-ისათვის)			მაგალითი 2 (ნახაზი 5-ისათვის)	
შეტანა	გამოტანა		შეტანა	გამოტანა
6 1 7 16 6 12 4 4 4 1 1 11 1	20		7 7 9 10 9 5 3 1 1 7 2 15 6 17 7	25

ქვემოთ მოცემული ნახაზებზე ნაჩვენებია საავტობუსო ქსელები მაგალითებში მოცემული შესატანი მონაცემებისათვის. თუ 1-ელ მაგალითში მე-3 და მე-4 გაჩერებებს ავირჩევთ, როგორც გადასაჯდომ პუნქტებს, მაშინ ყველაზე გრძელი გზა იქნება ან მე-2 და მე-5, ან მე-2 და 1-ელ გაჩერებებს შორის. გადასაჯდომი პუნქტების არჩევის უკეთესი ვარიანტი არ არის, ამიტომ პასუხი იქნება 20.

თუ მე-2 მაგალითში გადასაჯდომ პუნქტებად ავირჩევთ მე-5 და მე-6 გაჩერებებს, მაშინ ყველაზე გრძელი გზა იქნება მე-2 და მე-7 გაჩერებებს შორის. უკეთესი ვარიანტი არა გვაქვს, ამიტომ პასუხი იქნება 25.



საავტობუსო ქსელი 1-ლი მაგალითისათვის



საავტობუსო ქსელი მე-2 მაგალითისათვის

შეფასება

თუ თქვენს პროგრამას ტესტის გავლისათვის განსაზღვრულ დროში გამოაქვს სწორი პასუხი, მაშინ მიიღებთ ქულათა მაქსიმალურ რაოდენობას, წინააღმდეგ შემთხვევაში კი – 0 ქულას.

მითითება. იმის გამო, რომ გაჩერებათა რაოდენობა არცთუ ისე დიდია (2≤N≤500), შეგვიძლია ვიფიქროთ ვარიანტთა სრულ გადარჩევაზე – გადასაჯდომი პუნქტების როლში განვიხილოთ გაჩერებათა ყველა წყვილი და თითოეული წყვილისათვის მოვძებნოთ დანარჩენი გაჩერებების ისეთი შეერთება, რომ ყველაზე გრძელი გზის სიგრძე იყოს მინიმალური. ამ მინიმუმებს შორის მინიმალური იქნება ამოცანის ამოხსნა, მაგრამ აქ დგება შემდეგი საკითხი: როგორ ვიპოვოთ ეფექტურად თითოეული წყვილისათვის მინიმალური შეერთება?

ვთქვათ ოპტიმალური შეერთება უკვე ნაპოვნია. ჩამოვაყალიბოთ მისი ზოგიერთი თვისება.

***ლემა 1**. ვთქვათ d ყველაზე გრძელი გზის სიგრძეა, მაშინ ერთი მაინც გადასაჯდომი პუნქტისათვის მასთან შეერთებული ყველა გაჩერება დაშორებული იქნება მისგან არაუმეტეს d/2-სა. დავარქვათ ასეთ გადასაჯდომ პუნქტს კარგი, ხოლო მეორეს (იმ შემთხვევაშიც კი თუ ეს თვისება მისთვისაც სრულდება) – ცუდი.*

მართლაც, თუ ორივე გადასაჯდომ პუნქტზე მიერთებულია გაჩერება, რომელიც მისგან d/2-ზე მეტი მანძილითაა დაშორებული, მაშინ მანძილი ამ ორ გაჩერებას შორის d-ზე მეტი იქნება.

ლემა 2. *განვიხილოთ კარგი გადასაჯდომი პუნქტი. ვთქვათ, მასთან მიერთებული უმორქი გაჩერება მისგან დაცილებულია r მანძილით. მაშინ ყველა გაჩერება, რომელიც დაშორებულია კარგი გადასაჯდომი პუნქტიდან არაუმეტეს r მანძილისა შეგვიძლია შევაერთოთ ამ გადასაჯდომ პუნქტთან და ამით ყველაზე გრძელი გზის სიგრძე არ გაიზრდება, ე.ი. ქს შეერთებაც მოგვცემს ოპტიმალურ პასუხს.*

ლემა იოლად მტკიცდება საწინააღმდეგოს დაშვების მეთოდით.

შედეგი. *თუკი ჩვენ დავფიქსირებთ კარგ გადასაჯდომ პუნქტს, ცუდ გადასაჯდომ პუნქტს და კარგ გადასაჯდომ პუნქტთან მიერთებულ უმორქს გაჩერებას, ჩვენ იოლად გამოვთვლით უგრძელესი გზის მინიმალურ დასაშვებ სიგრძეს ამ შემთხვევისათვის. ყველა გაჩერებას, რომლებიც მდებარეობენ ჩვენს მიერ დაფიქსირებულ უმორქს გაჩერებაზე ახლოს კარგი გადასაჯდომი პუნქტიდან, ვაერთებთ კარგ გადასაჯდომ პუნქტთან, ხოლო დანარჩენებს – ცუდთან.*

ცხადია, რომ უგრძელესი გზა წარმოადგენს მაქსიმუმს შემდეგ სამ სიდიდეს შორის: ა) მანძილი კარგი გადასაჯდომი პუნქტის უმორქი გაჩერებიდან ამავე პუნქტის სიდიდით მეორე უმორქ გაჩერებამდე; ბ) ანალოგიური სიდიდე ცუდი გაჩერებისათვის; გ) მანძილი კარგი გადასაჯდომი პუნქტის უმორქი გაჩერებიდან ცუდი გადასაჯდომი პუნქტის უმორქ გაჩერებამდე ამ გადასაჯდომი პუნქტების გავლით.

ამრიგად, თუ გადავარჩევთ კარგ გადასაჯდომი პუნქტის, ცუდ გადასაჯდომი პუნქტის და კარგ გადასაჯდომ პუნქტთან მიერთებულ უმორქი გაჩერების ყველა შესაძლო სამეულს – მიღებული შედეგებიდან მინიმალური იქნება ამოცანის პასუხი. თუმცა ასეთი ალგორითმის სირთულე წარმოადგენს $O(N^4)$-ს და არ აკმაყოფილებს დროის ლიმიტს. ამგვარი ამოხსნა იძლეოდა ქულების 30%-ს, ხოლო სრული ქულების მისაღებად საჭირო იყო ალგორითმის სირთულე დაგვეყვანა $O(N^3)$-მდე. კუბური სირთულის მიღება შეიძლებოდა ქვემოთ მოყვანილი მომენტების გათვალისწინებით.

თავდაპირველად გამოვთვალოთ N ცალი W_i $(i = 1, 2, \ldots N)$ მასივი, სადაც W_i წარმოადგენს მანძილთა მასივს i-ური გაჩერებიდან ყველა დანარჩენამდე. მასივი დალაგებული უნდა იყოს ზრდადობით, ამასთან ჩვენ გვჭირდება არა მარტო მანძილების, არამედ გაჩერებათა ნომრების შენახვაც. ყოველივე ამას სჭირდება $O(N^2 \log N)$ დრო.

ჩავთვალოთ, რომ k-ური გაჩერება წარმოადგენს კარგ გადასაჯდომ პუნქტს. გადავარჩიოთ ყველა შესაძლო მანძილი, რომელზეც შეიძლება იმყოფებოდეს მასთან შეერთებული უმორქი გაჩერება. ამისათვის გადავარჩიოთ ყველა დანარჩენი გაჩერება და თითოეულისათვის ჩავთვალოთ, რომ მოცემული გაჩერება წარმოადგენს უმორქს. გადარჩევა დაწარმოოთ კარგი გადასაჯდომი პუნქტიდან მანძილის ზრდის მიხედვით, ანუ W_k მასივის საშუალებით. თავიდან ჩავთვალოთ, რომ არა გვაქვს კარგ გადასაჯდომ პუნქტთან მიერთებული გაჩერება. შემდეგ შევაერთოთ მასტან უახლოესი, შემდეგ მომდევნო და ა.შ. შევნიშნოთ, რომ მე-2 ლემის მიხედვით, თუკი რომელიმე გაჩერება მივაერთეთ კარგ გადასაჯდომ პუნქტთან, შემდეგში შეგვიძლია ჩავთვალოთ, რომ ქს გაჩერება მუდამ მასთანაა მიერთებული.

დავუშვათ, რომ ცუდ გადასაჯდომ პუნქტს წარმოადგენს გაჩერება h ნომრით. მაშინ უგრძელესი გზის სიგრძის ნებისმიერ მომენტში განსაზღვრისათვის უნდა დავმახსოვროთ h გაჩერებიდან ყველაზე მეტად დაშორებული ორი გაჩერება. ამისათვის შემოვიღოთ ორი ცვლადი (მიმთითებელი), რომელთაც მიენიჭებათ W_h მასივიდან შესაბამისი ორი ელემენტის ინდექსი. თავდაპირველად, როცა k-სთან არცერთი გაჩერება არაა მიერთებული, ისინი მიუთითებენ h-დან ყველაზე მეტად დაშორებულ ორ გაჩერებას, ანუ W_h-ის ორ უკანასკნელ ელემენტს, რომლებიც k-ს არ ემთხვევიან. შემდგომში, როცა ჩვენ დავიწყებთ k-სთან მისი უახლოესი გაჩერებების მიერთებას, შეიძლება აღმოჩნდეს, რომ ჩვენ k-ს მივუერთეთ ამ ორი ცვლადში მითითებული გაჩერებებიდან ერთ-ერთი. მაშინ შესაბამის მიმთითებელს გადავწევთ W_h მასივში მარცხნივ k-სთან მიუერთებელ მომდევნო გაჩერებამდე (ყურადღება უნდა მივაქციოთ იმასაც, რომ მიმთითებლები ერთმანეთს არ დაემთხვნენ. ამ შემთხვევაშიც ერთ-ერთი მათგანი უფრო მარცხნივ უნდა გადავწიოთ). საბოლოოდ ორივე მიმთიტებელი 0-ის ტოლი აღმოჩნდება, ანუ ყველა გაჩერება მიერთებული იქნება k-სთან.

ასეთი გადარჩევის პროცესში ჩვენ განვსაზღვრავთ k და h გაჩერებების ოპტიმალურ მიერთებას ყველა დანარჩენთან. შევნიშნოთ, რომ მიმთითებლების გადაადგილების ჯამური სიგრძე 2N-ის ტოლია. ამრიგად უგრძელესი გზის მინიმალური შესაძლო სიგრძე იმ შემთხვევაში, როცა k კარგი გადასაჯდომი პუნქტია, ხოლო h – ცუდი, შეიძლება $O(h)$ დროში. k და h-ის ყველა შესაძლო N^2 წყვილის პასუხებს შორის მინიმალური ამოცანის ამონახსნია და მის მოძებნას დასჭირდება $O(N^3)$ დრო.

3.2. რეკურსია

“ადამიანები, რომლებიც პროგრამისტებს მოსაწყენ ადამიანებად თვლიან, პროგრამისტებს უფრო მეტად მიაჩნიათ მოსაწყენ ადამიანებად, ვიდრე ადამიანებს, რომლებიც პროგრამისტებს მოსაწყენ ადამიანებად თვლიან, მიაჩნიათ მოსაწყენ ადამიანებად პროგრამისტები, რომლებსაც მოსაწყენ ადამიანებად მიაჩნიათ ისინი, ვინც მათ მოსაწყენ ადამიანებად თვლის” – ასე ამბობდა ზარატუსტრა. თუ თქვენ პირველივე წაკითხვით ყველაფერს მიხვდით, მაშინ რეკურსიას იოლად აითვისებთ, ხოლო თუ რამდენიმე წაკითხვა დაგჭირდათ – მეტი მობილიზება დაგჭირდებათ. თვალსაჩინოებისათვის შეგიძლიათ

მოძებნოთ ორი დიდი სარკე, დააყენოთ ერთმანეთის პირისპირ და თქვენ შუაში ჩადგეთ. ორივე სარკეში მიიღება არა მარტო თქვენი გამოსახულებები, არამედ გამოსახულებათა გამოსახულებები, იმათი გამოსახულებები და ასე დაუსრულებლად. შეიძლება ითქვას, რომ ეს გამოსახულებები რეკურსიულია.

მათემატიკური ტერმინოლოგიით თუ ვიტყვით – ***რეკურსიული*** ეწოდება ობიექტს, რომელიც ნაწილობრივ შედგება ან განისაზღვრება საკუთარი თავის მეშვეობით. მოვიყვანოთ რეკურსიული (მათემატიკაში ხშირად იტყვიან – ***რეკურენტული***) დამოკიდებულების ორი მაგალითი:

1. n! ფაქტორიალი (განისაზღვრება არაუარყოფითი მთელი რიცხვებისათვის):

ა) 0!=1;

ბ) თუ n>0, n!=n*(n-1)!

2. ფიბონაჩის მიმდევრობა:

ა) FIB(1)=1, FIB(2)=1;

ბ) თუ n>2, FIB(n)=FIB(n-1)+FIB(n-2)

რეკურსიული განსაზღვრების მთავარი ღირსება იმაში მდგომარეობს, რომ სასრული გამოსახულებით შეიძლება განისაზღვროს ობიექტების უსასრულო რაოდენობა. ანალოგიურად, პროგრამირებაში სასრული პროგრამული კოდით შესაძლოა აღვწეროთ უსასრულო გამოთვლითი პროცესი. რეკურსიული პროგრამის შესადგენად საკმარისია ვიცოდეთ პროცედურის ან ქვეპროგრამის ცნება, რადგან ისინი საშუალებას გვაძლევენ ოპერატორთა ნებისმიერ ჯგუფს მივანიჭოთ რაიმე სახელი, რომლითაც ისინი შემდეგ გამოიძახებიან. თუკი რომელიღაც P პროცედურა შეიცავს საკუთარი თავის ცხად გამოძახებას, იტყვიან რომ საქმე გვაქვს **პირდაპირ რეკურსიასთან**, ხოლო თუ P პროცედურა იძახებს Q პროცედურას, რომელიც თავის მხრივ შეიცავს P-ს გამოძახებას, მაშინ P-ს **ირიბად რეკურსიულს** უწოდებენ.

P პროცედურის ყოველი რეკურსიული გამოძახებისას მანქანის მეხსიერებაში იქმნება პროცედურის "ასლი", რომელსაც გამომძახებელი პროცედურისაგან გადაეცემა პარამეტრების მნიშვნელობები მოცემულ მომენტში. გადაცემული პარამეტრების მნიშვნელობები ინახება სტეკში, რომლის ზომაც სასრულია და რომლის **გადავსების (overflow)** საშიშროებაც არსებობს. სტეკი რომ არ გადაივსოს, პარამეტრების რომელიღაც მნიშვნელობებისათვის რეკურსიული გამოძახება აღარ უნდა განხორციელდეს. ვთქვათ, ეს მოხდა პროცედურის P_k-ური ასლისათვის, რომლის გამოძახებაც განხორციელებული იყო P_{k-1} ასლის i-ური სტრიქონიდან. მაშინ P_k ასლი დაასრულებს მუშაობას, მისთვის გადაცემული პარამეტრები ამოიტვირთება სტეკიდან და მართვა გადაეცემა P_{k-1} ასლის (i+1)-ე სტრიქონს.

პროგრამული თვალსაზრისით რეკურსიული შეიძლება იყოს როგორც პროცედურა, ასევე ფუნქციაც. რეკურსიული ფუნქციის მაგალითად შეიძლება მოვიყვანოთ ფაქტორიალის გამოთვლა

```
function Factorial (n)
{   if n=0 then Factorial=1 else Factorial=n*Factorial(n-1)   }
```

ტიპიურ ამოცანას, რომელიც რეკურსიის საშუალებით იხსნება, წარმოადგენს ამოცანა

3.4. ჰანოის კოშკი

მოცემულია სამი ღერო, რომელთაგანაც პირველზე ჩამოცმული N ცალი განსხვავებული დიამეტრის მქონე რგოლი ჰქმნის პირამიდას (ნებისმიერი მეტი დიამეტრის მქონე რგოლი უფრო ქვემოთაა განლაგებული, ვიდრე ნებისმიერი ნაკლები დიამეტრის მქონე რგოლი). საჭიროა ამ პირამიდის გადატანა პირველი ღეროდან მეორეზე ისე, რომ ერთ სვლაზე ღეროდან ღეროზე გადავანაცვლოთ მხოლოდ ერთი რგოლი და ამასთან პირამიდის პრინციპი არასდროს არ უნდა დაირღვეს, ანუ მეტი დიამეტრის რგოლი არ უნდა მოექცეს ნაკლები დიამეტრის რგოლზე ზემოთ.

ღეროები აღვნიშნოთ A, B და C სიმბოლოებით, ხოლო რგოლები გადავნომროთ ზრდადობით 1-დან N-მდე. თუკი ჩვენ მოვახერხებთ ზედა N-1 რგოლის გადატანას A-დან C ღეროზე, მაშინ მე-N-ე რგოლი გათავისუფლდება და შევძლებთ მის გადატანას B ღეროზე.

ამგვარად ჩვენ მივიღებთ ზუსტად იგივე ამოცანას, ოღონდ უფრო მცირე – N-1 რგოლისათვის. თუ ამ პროცესს N-ჯერ გავიმეორებთ, ამოცანა დაიყვანება ყველაზე პატარა რგოლის გადანაცვლებაზე, რომელიც პრობლემას არ წარმოადგენს. ალგორითმის რეალიზაციისთვის საჭიროა რეკურსიული ქვეპროგრამა (პროცედურა), რომელიც გამოიძახება ძირითადი პროგრამიდან.

რეკურსიული ქვეპროგრამა (პროცედურა)	ძირითადი პროგრამა

Move(M, A, B)
{ IF M=1 THEN { WRITE ('გაკეთდა სვლა ',A,'→',B) }
else {
C=6-A-B /* 6 ღეროთა ნომრების ჯამია – 6=1+2+3 */
Move(M-1,A,C)
Move(1,A,B)
Move(M-1,C,B) } }
	program Hanoi
	{
	READ (N)
	Move (N,1,2)
	}

ამრიგად, შეგვიძლია დავასკვნათ, რომ რეკურსიული ალგორითმების ძირითადი იდეა იმაში მდგომარეობს, რომ ამოცანა დავიყვანოთ ზუსტად იგივე ამოცანაზე, ოღონდ შემცირებული პარამეტრით. ამასთან პარამეტრის რომელიღაც მინიმალური მნიშვნელობა უნდა გამოითვლებოდეს რეკურსიული გამოძახების გარეშე. წინააღმდეგ შემთხვევაში პროგრამა “ჩაიციკლება”, რადგან რეკურსიული გამოძახებები უსასრულოდ გაგრძელდება. ზოგ რეკურსიულ ამოცანაში პარამეტრი შეიძლება გაიზარდოს, მაშინ არარეკურსიული ამოხსნა უნდა მოხდეს რომელიღაც მაქსიმალური მნიშვნელობისათვის.

ვცადოთ 3.1 თავში განხილული ამოცანების რეკურსიული ამოხსნა. 1,2,...,m რიცხვებისაგან შევადგინოთ n სიგრძის მქონე ყველანაირი მიმდევრობა. მიმდევრობა ფორმირდება X მასივში. რეკურსიული პროცედურა მუშაობს k პარამეტრით. თუ k<n, მაშინ ამოცანა გამოიძახება (k+1)-სათვის, ხოლო თუ k=n-სათვის გვაქვს ტრივიალური ამონახსნი – მიმდევრობა ფორმირებულია და საჭიროა მისი დაბეჭდვა.

რეკურსიული ქვეპროგრამა (პროცედურა)	ძირითადი პროგრამა

Generate (k)
{
IF k=N THEN { WRITE (X) }
ELSE {
FOR j=1 TO m { X[k+1]=j; Generate(k+1) } }
}
	program mimdevroba;
	{
	READ (m,n)
	Generate (0)
	}

მეორე ამოცანა, რომელიც 3.1 თავში განვიხილეთ, ეხებოდა ყველანაირი გადანაცვლების პოვნას 1,2,...,n რიცხვებისათვის. მიმდევრობა აქაც ფორმირდება X მასივში. k პარამეტრი აქაც იზრდება n-მდე და ბოლო გამოძახება ხორციელდება k=n-სათვის. Generate პროცედურიდან გამოსვლის შემდეგ X მასივს ექნება იგივე მნიშვნელობები, რაც ამავე პროცედურაში შესვლის წინ ჰქონდა. აქვე შევნიშნოთ, რომ გადანაცვლებათა გამოტანის თანმიმდევრობა არ იქნება დალაგებული ლექსიკოგრაფიულად, როგორც ეს არარეკურსიულ ალგორითმში იყო.

რეკურსიული ქვეპროგრამა (პროცედურა)	ძირითადი პროგრამა

Generate (k)
{
IF k=n THEN
WRITE (X)
ELSE
FOR j=k+1 to n {
Swap(X[k+1],X[j])
Generate(k+1)
Swap(X[k+1],X[j]) }
}
	program gadanacvleba
	{
	READ (N);
	FOR i=1 TO N { X[i]=i; }
	Generate(0)
	}

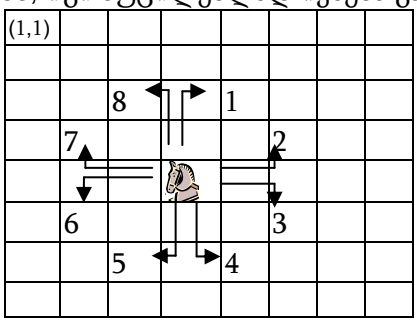
	დამხმარე ქვეპროგრამა (პროცედურა)
	Swap(a, b) { c=a; a=b; b=c }

რეკურსიული პროცესის უფრო ნათლად წარმოჩენისათვის განვიხილოთ ამოცანა:

3.5. მხედრით დაფის შემოვლა

ჭადრაკის დაფაზე მოთავსებულია მხედარი. დავწეროთ პროგრამა, რომელიც გამოითვლის სვლების იმ მიმდევრობას, რომლითაც მხედარს შეუძლია შემოიაროს მთელი დაფა ისე, რომ ყოველ უჯრაზე მხოლოდ ერთხელ იმყოფებოდეს.

კონკრეტული უჯრიდან მხედარს შეუძლია მაქსიმუმ რვა განსხვავებულ უჯრაზე გადავიდეს. გადავნომროთ მხედრის სვლები საათის ისრის მოძრაობის მიმართულებით. მოვათავსოთ მხედარი რომელიმე საწყის უჯრაზე – ვთქვათ (1,1)-ზე და ავირჩიოთ ასეთი სტრატეგია: განვიხილოთ სვლები მოცემული უჯრიდან ზრდადი თანმიმდევრობით და გავაკეთოთ პირველივე შესაძლებელი სვლა. სვლა შესაძლებლად ითვლება, თუკი იგი არ გადის დაფის საზღვრებიდან და თუ უჯრაზე, საითაც სვლის გაკეთებას ვაპირებთ, ჯერ მხედარი არ ყოფილა. თუკი რომელიმე უჯრიდან სვლის გაკეთება შეუძლებელია, მაშინ გავაუქმოთ ბოლო სვლა, დავბრუნდეთ წინა უჯრაზე და განვიხილოთ მომდევნო სვლა საათის ისრის მოძრაობის მიმართულებით. ასეთ მეთოდს უწოდებენ **გადარჩევას უკან დაბრუნებით** და ცხადია, რომ თუკი ამოხსნა საერთოდ არსებობს, იგი აუცილებლად იქნება განხილული.



(1,1) უჯრიდან მხედარს შეუძლია გააკეთოს მხოლოდ მე-3 და მე-4 სვლები. მე-3 სვლის შემთხვევაში ის გადავა (2,3) უჯრაზე, საიდანაც უკვე შეუძლია გააკეთოს სვლები მეორიდან მეექვსის ჩათვლით და ა.შ.

1			34	3	36	19	22
		2	37	20	23	4	17
		33		35	18	21	10
		38		24	11	16	5
	32			39	26	9	12
			25		15	6	27
31			40	29	8	13	42
		30		14	41	28	7

1			34	3	36	19	22
	53	2	37	20	23	4	17
		33	46	35	18	21	10
52	47	38		24	11	16	5
	32	45	48	39	26	9	12
	51		25	42	15	6	27
31	44	49	40	29	8	13	
50		30	43	14	41	28	7

მხედარი 42-ე სვლის ჩათვლით რვა შესაძლოდან ერთ-ერთს ყოველთვის გააკეთებს, ხოლო 43-ე სვლის გასაკეთებლად უჯრას ვეღარ იპოვის (პირველი 5 სვლა დაფის გარეთ გადის, ხოლო ბოლო სამი სვლიდან მიღწევად უჯრებზე უკვე ნამყოფია მე-9, მე-15 და 41-ე სვლებზე). სწორედ აქ ხორციელდება უკან დაბრუნება – 42-ე სვლა უქმდება და განიხილება 41-ე სვლაზე მიღწეული უჯრიდან 42-ე სვლის გაკეთების სხვა შესაძლებლობა რიგის მიხედვით. ასეთი სვლა არსებობს, ამიტომ მხედარი გადადის შესაბამის უჯრაზე (თუ სვლა ვერ მოინახებოდა, 40-ე სვლაც გაუქმდებოდა). შემდეგი მობრუნება განხორციელდება 53-ე სვლაზე და ა.შ.

ალგორითმის რეალიზაციისათვის საჭიროა გარკვეული მოსამზადებელი სამუშაოები. პირველ ყოვლისა, უნდა აღწეროთ სვლების მასივი ზომით [8,2]. დავარქვათ ამ მასივს c. მისი მნიშვნელობები იქნება:

c= ((-2,1),(-1,2),(1,2),(2,1),(2,-1),(1,-2),(-1,-2),(-2,-1))

სამუშაო არე (დავარქვათ მას d მასივი), ანუ ჭადრაკის დაფა, აღვწეროთ არა 8×8, არამედ – 12×12. უფრო ზუსტად რომ ვთქვათ, თითოეული ინდექსი ვცვალოთ -1-დან 10-მდე, ისე როგორც ეს ნახაზზეა მოცემული. d მასივი ჯერ მთლიანად შევავსოთ -1-ებით (ზოგადად რომ ვთქვათ – nil-ით), ხოლო შემდეგ მის შიგნით 0-ებით შევავსოთ ჭადრაკის დაფის ტოლი არე ინდექსებით 1-დან 8-მდე. სამუშაო არის ასეთი აღწერა თავიდან აგვაცილებს ყოველი ახალი სვლის დროს იმის შემოწმებას, ხომ არ გასცდა მხედარი საჭადრაკო დაფას, ანუ ხომ არ გადააჭარბა რომელიმე ინდექსმა 8-ს ან ხომ არ გახდა რომელიმე ინდექსი 1-ზე ნაკლები. მუშაობის პროცესში 0-ის ადგილას ჩაიწერება იმ სვლის ნომერი, რომლითაც მხედარი ამ

უჯრაზე მოვა და ამგვარი მიდგომით ჩვენ დაგვჭირდება მხოლოდ იმის შემოწმება, არის თუ არა მოცემული უჯრა 0-ის ტოლი. თუკი უჯრა 0-ის ტოლია, მაშინ მხედარი გადადის ამ უჯრაზე, ხოლო ყველა სხვა შემთხვევაში ახალი უჯრაა მოსაძებნი. აქვე შევნიშნოთ, რომ სამუშაო არის ასეთი აღწერა რიგ სხვა ამოცანებშიც შეიძლება გამოგვადგეს.

	-1	0	1	2	3	4	5	6	7	8	9	10
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	-1	-1	0	0	0	0	0	0	0	0	-1	-1
2	-1	-1	0	0	0	0	0	0	0	0	-1	-1
3	-1	-1	0	0	0	0	0	0	0	0	-1	-1
4	-1	-1	0	0	0	0	0	0	0	0	-1	-1
5	-1	-1	0	0	0	0	0	0	0	0	-1	-1
6	-1	-1	0	0	0	0	0	0	0	0	-1	-1
7	-1	-1	0	0	0	0	0	0	0	0	-1	-1
8	-1	-1	0	0	0	0	0	0	0	0	-1	-1
9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

რეკურსიული პროცედურის გამოძახებამდე საჭიროა განვსაზღვროთ საწყისი უჯრა, საიდანაც მხედარმა მოძრაობა უნდა დაიწყოს. ვთქვათ, ეს უჯრა იყოს (1,1) და მასში ჩავწეროთ 1-იანი. რეკურსიულ პროცედურას გადაეცემა სამი პარამეტრი: სტრიქონის ნომერი i და სვეტის ნომერი j იმ უჯრისა, რომელშიც მოცემულ მომენტში იმყოფება მხედარი და სვლის ნომერი n. თუკი სვლის ნომერმა გადააჭარბა 64-ს, მხედარი უკვე ყოფილა ყველა უჯრაზე და ვბეჭდავთ d მასივს. შევნიშნოთ, რომ k ცვლადი რეკურსიულ პროცედურაში უნდა აღიწეროს.

რეკურსიული ქვეპროგრამა (პროცედურა)	ძირითადი პროგრამა

mxedari (i, j, n)
{
IF n=65 THEN WRITE (d)
k=0
WHILE k<=8 {
k=k+1
IF k>=9 THEN BREAK
IF (d[i+c[k,1],j+c[k,2]]=0) THEN
d[i+c[k,1],j+c[k,2]]=n ELSE CONTINUE
mxedari(i+c[k,1], j+c[k,2], n+1) }
d[i,j]=0; n=n-1; }
	FOR i=-1 TO 10 {
	FOR j=-1 TO 10 { d[i,j]=-1 }
	}
	FOR i=1 TO 8 {
	FOR j=1 TO 8 { d[i,j]=0 }
	}
	i=1; j=1; n=1; d[1,1]=1;
	mxedari (i, j, n+1);

დასასრულს მოვიყვანოთ ორი ამოხსნა იმ მრავალთაგან, რომელიც ამ ამოცანას გააჩნია:

1	38	55	34	3	36	19	22
54	47	2	37	20	23	4	17
39	56	33	46	35	18	21	10
48	53	40	57	24	11	16	5
59	32	45	52	41	26	9	12
44	49	58	25	62	15	6	27
31	60	51	42	29	8	13	64
50	43	30	61	14	63	28	7

1	52	35	48	3	50	19	22
34	47	2	51	20	23	4	17
53	40	33	36	49	18	21	10
46	59	54	41	24	11	16	5
39	32	45	58	37	26	9	12
60	55	38	25	42	15	6	27
31	44	57	62	29	8	13	64
56	61	30	43	14	63	28	7

ახლა განვიხილოთ კიდევ ერთი კლასიკური ამოცანა რეკურსიის თემაზე:

3.6. ლაზიერები

განალაგეთ N ლაზიერი ჭადრაკის N×N დაფაზე ისე, რომ არცერთ მათგანს არ შეეძლოს სხვა ლაზიერის მოკვლა (მეტი სიზუსტისათვის შევნიშნოთ, რომ ლაზიერს შეუძლია ფიგურის მოკვლა იმ შემთხვევაში, თუკი ისინი მდებარეობენ ერთ ჰორიზონტალზე, ერთ ვერტიკალზე ან ერთ დიაგონალზე).

ამოცანის ამოსახსნელად საჭიროა გადარჩევა უკან დაბრუნებით. სრული გადარჩევის შემთხვევაში განსახილველ ვარიანტთა რაოდენობა N^N იქნება, რაც საკმაოდ დიდი რიცხვია თუნდაც N=8-სათვის. უკან დაბრუნება მნიშვნელოვნად ამცირებს ამ რიცხვს. ცხადია, რომ ამონახსნში ყოველი ლაზიერი განლაგებული იქნება განსხვავებულ ვერტიკალზე (და ჰორიზონტალზეც). ავირჩიოთ რაიმე მიმართულება ლაზიერების დასმისათვის: ვთქვათ ზემოდან ქვემოთ და მარცხნიდან მარჯვნივ. პირველი ლაზიერი დავსვათ (1,1) უჯრაზე და დავიწყოთ თავისუფალი უჯრის ძებნა მეორე სვეტში. პირველივე თავისუფალ უჯრაზე დავსვათ მეორე ლაზიერი და დავიწყოთ თავისუფალი უჯრის ძებნა მესამე სვეტში და ა.შ. საზოგადოდ თუკი 1-დან i-ური სვეტის ჩათვლით ლაზიერები დავსვით, (i+1)-ე სვეტში ვიწყებთ თავისუფალი უჯრის ძებნას (ანუ ისეთი უჯრისას, რომელსაც არცერთი უკვე დასმული ლაზიერი არ ემუქრება). თუკი ასეთი უჯრა მოინახა, ვსვამთ (i+1)-ე ლაზიერს და ვამოწმებთ, ხომ არ მივაღწიეთ N-ურ სვეტს (მაშინ ამონახსნი უკვე მიგვიღია), ხოლო თუკი (i+1)-ე სვეტში თავისუფალი უჯრა არ არის, მაშინ დაფიდან ვიღებთ i-ურ ლაზიერს (ანუ ვახორციელებთ უკან დაბრუნებას) და ვცდილობთ მოვძებნოთ მისთვის თავისუფალი უჯრა იმ უჯრის ქვემოთ, სადაც ის აქამდე იყო მოთავსებული. თუ ასეთი უჯრა არ მოიძებნა, დაფიდან ვიღებთ (i-1)-ე ლაზიერს და ა.შ.

ზემოთ თქმულიდან ჩანს, რომ თავისუფალი უჯრის ძებნა სვეტებში ერთნაირი პრინციპით ხორციელდება, ამიტომ შეგვიძლია გამოვიყენოთ რეკურსია. ყველა ამონახსნის მისაღებად საჭიროა უკან დაბრუნება ამონახსნის პოვნის შემდეგაც მოვახდინოთ. პირველი ორი უკან დაბრუნება კი ნაჩვენებია ნახაზზე, სადაც 0-ები აღნიშნავენ ჭადრაკის დაფის ცარიელ (და არა თავისუფალ) უჯრედებს, ხოლო 1 აღნიშნავს დასმულ ლაზიერს.

	1	2	3	4	5	6	7	8
1	1	0	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0
3	0	1	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0
5	0	0	1	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

	1	2	3	4	5	6	7	8
1	1	0	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0
3	0	1	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	1	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	1	0	0	0

	1	2	3	4	5	6	7	8
1	1	0	0	0	0	0	0	0
2	0	0	0	0	1	0	0	0
3	0	1	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0
5	0	0	1	0	0	0	0	0
6	0	0	0	0	0	0	1	0
7	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0

ალგორითმის რეალიზაციისათვის გამოვიყენოთ ლოგიკური ტიპის ფუნქცია (კოდში მისი სახელია svla), რომელიც გამოთვლის, შეიძლება თუ არა მოცემულ უჯრაზე მორიგი ლაზიერის დასმა. თუ უჯრა თავისუფალია, ფუნქცია დააბრუნებს TRUE მნიშვნელობას, წინააღმდეგ შემთხვევაში – FALSE-ს. ერთ-ერთ პრობლემას ამ ამოცანის ამოხსნისას წარმოადგენს იმის გარკვევა, დგანან თუ არა ლაზიერები ერთ დიაგონალზე. ამ საკითხის გადაწყვეტა მოცემულია წიგნის პირველ თავში (ამოცანა 1.38). პასუხის ფორმირება მოვახდინოთ 1..N ზომის X მასივში, რომელიც ერთგანზომილებიანი იქნება. მისი ინდექსი აღნიშნავს სვეტის ნომერს, ხოლო შესაბამისი ელემენტის მნიშვნელობა – სტრიქონის ნომერს. raod ცვლადში ხდება ამონახსნების რაოდენობის დათვლა. N ძირითად პროგრამაში აღწერილია, როგორც მუდმივა, რადგან შემდგომში მისი საშუალებით უნდა მოხდეს X მასივის ზომის განსაზღვრა.

რეკურსიული პროცედურა	ძირითადი პროგრამა

solve (k)
{
FOR y=1 to N {
IF svla(X, k, y) THEN
{ X[k]=y
IF k=N THEN {
FOR i=1 TO N { WRITE(X) }
raod=raod+1 }
solve(k+1)
} }
}
	program Queens;
	const N=8;
	{ raod=0
	solve(1);
	WRITE(raod); }

	დამხმარე ფუნქცია (ლოგიკური ტიპის)
	function svla(X, k, y) { z=1 WHILE (z<k) and (y<>X[z]) and (abs(k-z)<>abs(y-X[z])) { z=z+1 } svla=(z=k) }

დასასრულს, მოვიყვანოთ ყველა ამონახსნი N=8-სათვის, რომელთა საერთო რაოდენობა 92-ია. ამონახსნებს ახასიათებთ გარკვეული სიმეტრიულობა და დაფის ბრუნვით ერთი ამონახსნიდან შეიძლება სხვა ამონახსნი მივიღოთ.

15863724	31758246	36824175	46152837	51863724	57263184	63724815	73825164
16837425	35281746	37285146	46827135	52468317	57413862	63728514	74258136
17468253	35286471	37286415	46831752	52473861	58413627	63741825	74286135
17582463	35714286	38471625	47185263	52617483	58417263	64158273	75316824
24683175	35841726	41582736	47382516	52814736	61528374	64285713	82417536
25713864	36258174	41586372	47526138	53168247	62713584	64713528	82531746
25741863	36271485	42586137	47531682	53172864	62714853	64718253	83162574
26174835	36275184	42736815	48136275	53847162	63175824	68241753	84136275
26831475	36418572	42736851	48157263	57138642	63184275	71386425	
27368514	36428571	42751863	48531726	57142863	63185247	72418536	
27581463	36814752	42857136	51468273	57248136	63571428	72631485	
28613574	36815724	42861357	51842736	57263148	63581427	73168524	

3.7. ლათინური კვადრატი

(საქართველოს მოსწავლეთა X ოლიმპიადა, ზონური ტური)

განსაზღვრება: N-ური რიგის ლათინური კვადრატი ეწოდება N×N განზომილების ცხრილს, რომელიც შედგენილია N რაოდენობის განსხვავებული ელემენტისაგან (მოცემულ ამოცანაში დავუშვათ, რომ ეს ელემენტები ნატურალური რიცხვებია 1-დან N-მდე). ამასთან, ცხრილის ყოველი უჯრა შევსებულია და თითოეული ელემენტი მხოლოდ ერთხელ გვხვდება თითოეულ სტრიქონსა და თითოეულ სვეტში.

მოცემულობა: მოცემულია N×N განზომილების ცხრილი (1< N<7), რომლის პირველი სტრიქონი შევსებულია ზრდადობით დალაგებული რიცხვებით: 1,2,..., N.

დავალება: დაწერეთ პროგრამა, რომელიც: ა) მოცემული N-სათვის ცხრილის დანარჩენ სტრიქონებს შეავსებს ისე, რომ მივიღოთ ლათინური კვადრატი (A ქვეამოცანა); ბ) გამოთვლის, მოცემული N-სათვის რამდენი ერთმანეთისაგან განსხვავებული ლათინური კვადრატის მიღება შეიძლება (B ქვეამოცანა).

შესატანი მონაცემები:

N

გამოსტანი მონაცემები:

ერთ-ერთი N-ური რიგის ლათინური კვადრატი და ყველა შესაძლებელი N-ური რიგის ლათინური რიგის კვადრატის რაოდენობა.

შეზღუდვები: დროის ლიმიტი 1 ტესტისათვის – 30 წამი (Pentium 1, 133 მეგაჰერცი).

მითითება. B ქვეამოცანა ფაქტობრივად მოიცავს A-ს, ამიტომ ვიმსჯელოთ მხოლოდ მის შესახებ. ყველა შესაძლო ვარიანტთა რაოდენობის გამოთვლა, ერთი შეხედვით კომბინატორულადაც შეიძლება. მიუხედავად იმისა, რომ ამ ხერხით ბევრმა სცადა პასუხების

მიღება, მათ მცდელობებს შედეგი არ მოჰყოლია. დროითი შეზღუდვა ტესტებისათვის პრობლემურია მხოლოდ N=6-სათვის. უფრო მცირე N-ებისათვის პასუხი არაოპტიმალური ალგორითმითაც შეიძლება მივიღოთ. ასეთად შეიძლება ჩაითვალოს ალგორითმი, რომელიც მხოლოდ N×N ზომის რიცხვით მასივთან იმუშავებს და ცხრილში ყოველი ახალი რიცხვის ჩასმის დროს გადაამოწმებს, ხომ არ ემთხვევა ახალი რიცხვი რომელიმე უკვე ჩასმულ რიცხვს შესაბამის სტრიქონსა და სვეტში. მაგალითისათვის მოვიყვანოთ ლოგიკური ტიპის ფუნქცია, რომელიც ამ დებულებას ორი ციკლის საშუალებით ამოწმებს i-ური სტრიქონისა და j-ური სვეტის გადაკვეთაზე დასმული რიცხვისათვის.

```
function f (i,j) – ლოგიკური ტიპის
{  f=true
FOR  j1=1  TO  j-1  {
IF  a[i,j]=a[i,j1]  THEN  { f=false }   }
FOR i1=1 TO i-1 {
IF  a[i,j]=a[i1,j]  THEN  { f=false }   }
}
```

ეს ფუნქცია დაახლოებით 5-ჯერ ზრდის პროგრამის მუშაობის ხანგრძლივობას და მისი გამოყენების შემთხვევაში N=6-სათვის დროითი შეზღუდვის გადალახვა თითქმის შეუძლებელია. უმჯობესია შემოვიღოთ N×N ზომის ორი ლოგიკური ტიპის მასივი, რომელთაგან ერთში შევინახოთ ინფორმაცია კონკრეტული რიცხვის გამოყენების შესახებ ამა თუ იმ სტრიქონში (დავარქვათ მას st), ხოლო მეორეში – ანალოგიური ინფორმაცია სვეტების მიხედვით (მისი სახელი იყოს sv). მაგალითად, თუ ლათინური კვადრატის მეორე სტრიქონისა და მეოთხე სვეტის გადაკვეთაზე ჩაწერილია რიცხვი 5, მაშინ st[2,5]=false და sv[4,5]=false, ხოლო თუკი გამოთვლის პროცესში იმავე უჯრაში 5-ს წავშლით, მაშინ ეს ელემენტები მიიღებენ true მნიშვნელობას. ასეთი რეალიზაციის შემთხვევაში დამხმარე ფუნქცია საჭირო აღარ არის და ორი მიმართვის საშუალებით შეგვიძლია იმის გარკვევა, კონკრეტული რიცხვის ჩაწერა განსახილველ უჯრაში შესაძლებელია თუ არა.

რეკურსიული ქვეპროგრამა (პროცედურა)	ძირითადი პროგრამა

solve(i1,j1,k: integer);
{
for k=1 to n {
a[i1,j1]=k
if ((st[i1,k]) and (sv[j1,k])) then {
if ((i1=n) and (j1=n)) then { rez=rez+1; break }
st[i1,k]=false; sv[j1,k]=false;
if j1<n then s(i1,j1+1,1) else s(i1+1,1,1);
a[i1,j1]=0; st[i1,k]=true; sv[j1,k]=true;
}
else { a[i1,j1]=0
}
}
}
	program Latin;
	{ read(n); rez=0
	init
	solve(2,1,0)
	write (rez)
	}
	დამხმარე ქვეპროგრამა(პროცედურა) – ინიციალიზაცია
	Init ( )
	{ for i=1 to n {
	for j=1 to n {
	sv[i,j]=true; st[i,j]=true } }
	for j=1 to n {
	a[1,j]=j; st[1,j]=false; sv[j,j]=false }
	}

თავდაპირველად, დამხმარე ქვეპროგრამის საშუალებით ხდება საწყისი მონაცემების ინიციალიზაცია. sv და st მასივები შეივსება true მნიშვნელობებით, ხოლო იმის გამო რომ ლათინური კვადრატის პირველი სტრიქონი განსაზღვრულია, შესაბამის უჯრებში ჩაიწერება false. შესაბამისად შეივსება ძირითადი a მასივიც. რეკურსიული ქვეპროგრამა solve ძირითადი პროგრამიდან გამოიძახება a[2,1] უჯრისათვის. თითოეული უჯრისათვის მნიშვნელობის შერჩევა მოხდება k ცვლადის საშუალებით, რომელიც ციკლის ცვლადს წარმოადგენს. თუკი მოცემული უჯრისათვის რიცხვი შეირჩა, მაშინ მოხდება რეკურსიული გამოძახება მომდევნო უჯრისათვის (უჯრების თანმიმდევრობა განიხილება მარცხნიდან მარჯვნივ და ზემოდან

ქვემოთ), თუკი რიცხვის შერჩევა შეუძლებელია, მაშინ მიმდინარე რეკურსიული ქვეპროგრამა მთავრდება და ვბრუნდებით წინა რეკურსიულ გამოძახებაზე. a მასივის მთლიანად შევსების შემთხვევაში შედეგის მთვლელს (ცვლადი rez) ვზრდით 1-ით და ასევე ვბრუნდებით წინა რეკურსიულ გამოძახებაზე. ბოლოს მოვიყვანოთ ტესტების პასუხები:

ტესტი – N=	*3*	*4*	*5*	*6*
ტესტის პასუხი	*2*	*24*	*1344*	*1128960*

რეკურსიის ტექნიკის დაუფლება ზოგჯერ იწვევს ზედმეტ მიჯაჭვულობას ამ მეთოდისადმი იმ მარტივი მიზეზის გამო, რომ მისი გამართვა ხშირად საკმაოდ იოლია და პროგრამული კოდიც მცირე გამოდის, მაგრამ რეკურსიას გააჩნია გარკვეული ნაკლოვანებაც, რომლის წარმოსაჩენად გავარჩიოთ ამოცანა ამ თავის დასაწყისში ნახსენები ფიბონაჩის მიმდევრობის შესახებ:

გამოვთვალოთ N-ური (N<100) წევრი ფიბონაჩის მიმდევრობაში, რომლის პირველი ორი წევრი 1-ის ტოლია, ხოლო ყველა დანარჩენი წევრი წარმოადგენს წინა ორის ჯამს – 1, 1, 2, 3, 5, 8, 11,...

პირველი, რაც შეიძლება მოგვაფიქრდეს, რეკურსიული ფუნქციის დაწერაა:

```
function FIB(N)
{     IF (N=1) OR (N=2) THEN FIB=1 ELSE FIB=FIB(N-1)+FIB(N-2)        }
```

საკმარისია ამ ფუნქციას მივაწოდოთ 70-ზე დიდი მნიშვნელობა, რომ ის გაურკვეველი ვადით ჩაფიქრდება. თუკი იმავე ამოცანას რაიმე მასივის გამოყენებით რეკურსიის გარეშე ამოვხსნიდით, პროგრამას გამოთვლის სიჩქარის პრობლემა არ ექნებოდა N=1000-თვისაც (მხოლოდ დიდი რიცხვების არითმეტიკა გახდება საჭირო):

```
F[1]=1;         F[2]= 1
FOR   i= 3   TO   N     {  F[i]= F[i-1] + F[i-2]         }
```

რატომ ვერ უმკლავდება რეკურსია ასეთ პრიმიტიულ ამოცანას, რომლის ამოსახსნელად ერთი ციკლიც საკმარისია? საქმე იმაშია, რომ FIB[70]-ის გამოთვლისთვის რეკურსიულმა ფუნქციამ უნდა გამოთვალოს FIB[69] და FIB[68]. მიუხედავად იმისა, რომ FIB[68] ფუნქციას უკვე გამოთვლილი ჰქონდა FIB[69]-ის გამოთვლის დროს, ის ამ მნიშვნელობას თავიდან ითვლის და ასე იქცევა მიმდევრობის ყველა სხვა წევრის მიმართაც. სწორედ ეს გახლავთ რეკურსიის ნელი მუშაობის მიზეზი – ის უამრავჯერ ითვლის ფუნქციის მნიშვნელობას ერთი და იგივე არგუმენტისათვის. მართალია, მასივის შემოღება და მიღებული შედეგების მასში შენახვა მკვეთრად გააუმჯობესებს რეკურსიის მუშაობის სიჩქარესაც:

```
Function FIB(X)
{  IF F[X] = 0 THEN    {
       IF (X=1) OR (X=2) THEN F[X] = 1
                         ELSE F[X] = FIB(x-1) + FIB(x-2)          }
  FIB = F[X]
}
```

მაგრამ ზემოთ მოყვანილი არარეკურსიული ალგორითმი ამ ალგორითმზეც უფრო სწრაფია, ამიტომ დასკვნის სახით შეიძლება ითქვას, რომ რეკურსიის გამოყენების დროს სიფრთხილეა საჭირო და აუცილებლად უნდა შეფასდეს მისი მუშაობის დრო მაქსიმალური ტესტებისათვის.

4. სორტირება და ძებნა

4.1.სორტირება პირდაპირი ჩასმით

სორტირების ეს მეთოდი ძალიან წააგავს ბანქოს თამაშის დროს მიღებული დასტის დალაგებას. მასივის (ან ბანქოს) ელემენტები წარმოსახვით იყოფიან ორ ნაწილად: უკვე დალაგებულ მიმდევრობად $a_1...a_{i-1}$ და საწყის მიმდევრობად $a_i...a_n$. ყოველ ბიჯზე, დაწყებული i=2-დან, i იზრდება ერთით, საწყისის მიმდევრობიდან ამოიღება i-ური ელემენტი და ჩაისმება დალაგებულ მიმდევრობაში შესაბამის ადგილზე. 4.1. ცხრილში მოცემულია პირდაპირი ჩასმით სორტირების პროცესი 8 შემთხვევითად აღებული რიცხვისათვის.

საწყისი მიმდევრობა	44	55	12	42	94	18	06	67
i=2	44	55	12	42	94	18	06	67
i=3	12	44	55	42	94	18	06	67
i=4	12	42	44	55	94	18	06	67
i=5	12	42	44	55	94	18	06	67
i=6	12	18	42	44	55	94	06	67
i=7	06	12	18	42	44	55	94	67
i=8	06	12	18	42	44	55	67	94

ცხრილი 4.1.

შესაბამისი ადგილის მოსაძებნად i-ური ელემენტი რიგ-რიგობით შეედარება მის მარცხნივ მყოფ ელემენტებს დაწყებული უშუალოდ მისი მარცხენა მეზობლიდან, ვიდრე: ა) ნაპოვნი არ იქნება მასზე ნაკლები მნიშვნელობის მქონე ელემენტი ან ბ) მიღწეული არ იქნება დალაგებული მიმდევრობის მარცხენა საზღვარი. ორი დამამთავრებელი პირობის მქონე ასეთი განმეორებადი პროცესის ტიპიური შემთხვევა საშუალებას გვაძლევს გამოვიყენოთ ბარიერის (sentinel) კარგად ცნობილი ხერხი. ბარიერად შეგვიძლია დავსვათ a_0, რომელიც ტოლი იქნება საწყისი მიმდევრობიდან დალაგებულ მიმდევრობაში ჩასასმელი ელემენტის მნიშვნელობისა.

მოვიყვანოთ ამ ალგორითმის სრული პროგრამული კოდი:

```
SraightInsertion (a,n);
FOR i=2 TO n {
        X=a[i];    a[0]=x;   j=I;
        WHILE x<a[j-1]  {
                A[j]=a[j-1]; j=j-1     }
        A[j]=x  }
```

ცხადია, რომ ყველაზე მცირე რაოდენობის შედარებებს ეს ალგორითმი მაშინ გააკეთებს, როცა მასივი უკვე დალაგებულია. ამ შემთხვევაში ყოველი ელემენტი შეედარება მხოლოდ მის მარცხივ მდებარე ელემენტს და რადგან ის ნაკლები იქნება – დარჩება საკუთარ ადგილზე. სულ შესრულდება n-1 შედარება. ყველაზე მეტი შედარება შესრულდება, თუკი საწყისი მასივი შებრუნებული მიმდევრობითაა დალაგებული. მაშინ ყოველი i-ური ელემენტის ჩასმას საკუთარ ადგილზე i შედარება დასჭირდება ბარიერთან შედარების ჩათვლით.

პირდაპირი ჩასმის ალგორითმი შეიძლება მცირედით გაუმჯობესდეს, თუკი საწყისი მიმდევრობიდან აღებული ელემენტისათვის დალაგებულ მიმდევრობაში ადგილის ძებნისას გამოვიყენებთ არა თითოეულ მარცხნივ მდებარე ელემენტთან შედარებას, არამედ ორობითი ძებნის ალგორითმს. რომელიც განხილულია ამ თავის დასასრულს. ორობითი ძებნის გამოყენების საშუალებას გვაძლევს ის ფაქტი, რომ აღებული ელემენტის მარცხნივ მდებარე ელემენტები უკვე სორტირებულია.

4.2. სორტირება პირდაპირი ამორჩევით

ეს მეთოდი დაფუძნებულია შემდეგ პრინციპებზე:

ა) მასივიდან ამოირჩევა უმცირესი მნიშვნელობის მქონე ელემენტი;

ბ) ამორჩეული ელემენტი ადგილს უცვლის მასივის პირველ ელემენტს;

გ) პირველ ორ პუნქტში განხორციელებული პროცესი მეორდება n-1 ელემენტისათვის (2-დან n-მდე), შემდეგ n-2 ელემენტისათვის (3-დან n-მდე) და ა.შ. სანამ არ დარჩება ერთი – ყველაზე დიდი მნიშვნელობის მქონე ელემენტი.

4.2. ცხრილში მოცემულია ალგორითმის მუშაობის პროცესი:

საწყისი მიმდევრობა	44 55 12 42 94 18 06 67
i=2	06 55 12 42 94 18 44 67
i=3	06 12 55 42 94 18 44 67
i=4	06 12 18 42 94 55 44 67
i=5	06 12 18 42 94 55 44 67
i=6	06 12 18 42 44 55 94 67
i=7	06 12 18 42 44 55 94 67
i=8	06 12 18 42 44 55 67 94

ცხრილი 4.2.

პირდაპირი ამორჩევის მეთოდი მნიშვნელოვნად განსხვავდება პირდაპირი ჩასმის მეთოდისაგან. პირდაპირი ჩასმის მეთოდში განიხილებოდა საწყისი მიმდევრობის მხოლოდ ერთი ელემენტი და უკვე დალაგებული მიმდევრობის ყველა ელემენტი, რათა აღებული ელემენტისათვის გვეპოვნა ჩასმის წერტილი. პირდაპირი ამორჩევის მეთოდში კი განიხილება საწყისი მიმდევრობის ყველა ელემენტი უმცირესი მნიშვნელობის მქონე ელემენტის საპოვნელად და მოძებნილი ელემენტი ჩაისმება დალაგებულ მიმდევრობაში.

ქვემოთ მოყვანილია პირდაპირი ამორჩევის ალგორითმის სრული პროგრამული კოდი:

```
StraightSelection (a, n)
FOR i=1 TO n-1 {
        K=i; x=a[i]
        FOR j=i+1 TO n {
                IF a[j]<x THEN  {  k=j;   x=a[k]  }     }
        A[k]=a[i]; a[i]=x     }
```

პირდაპირი ჩასმის ალგორითმის მუშაობის საშუალო დროა n*(ln(n)+g), სადაც n მასივში ელემენტების რაოდენობაა, ხოლო g=0.577216... – ეილერის მუდმივა.

4.3. სორტირება პირდაპირი გაცვლით (ბუშტულა)

ამ მეთოდის ძირითადი პრინციპია ყოველი ორი მეზობელი ელემენტისათვის ადგილის გაცვლა მანამ, სანამ მთელი მასივი არ დალაგდება. თუკი საწყის მასივს წარმოვიდგენთ არა პორიზონტალურად, არამედ ვერტიკალურად, მასივის ელემენტები შეიძლება შევადაროთ წყალში მოთავსებულ ჰაერის ბუშტულებს, რომელთა წონები შეესაბამებიან მასივის ელემენტების მნიშვნელობებს. ამ დროს შეიძლება ითქვას, რომ მასივის ყოველი გავლისას ბუშტულები მოძრაობენ ქვემოდან ზემოთ საკუთარი წონის შესაბამის დონემდე. სორტირების ამ მეთოდს უმეტესობა იცნობს ბუშტულების სორტირების (BubbleSort, пузырковая сортировка) სახელწოდებით. 4.3. ცხრილში ნაჩვენებია ამ მეთოდის მუშაობის პროცესი:

i=	1	2	3	4	5	6	7	8
	44	06	06	06	06	06	06	06
	55	44	12	12	12	12	12	12
	12	55	44	18	18	18	18	18
	42	12	55	44	42	42	42	42
	94	42	18	55	44	44	44	44
	18	94	42	42	55	55	55	55
	06	18	94	67	67	67	67	67
	67	67	67	94	94	94	94	94

ცხრილი 4.3.

ქვემოთ მოყვანილია ალგორითმის პროგრამული კოდი:

```
BubbleSort (a,n)
FOR i=2 To n {
        FOR j=n TO i STEP -1   {
                IF a[j-1]>a[j] THEN   { x=a[j-1]; a[j-1]=a[j];  a[j]=x }      }        }
```

ცხადია, რომ ალგორითმი შორს დგას სრულყოფილებისაგან. 4.3. ცხრილიდან ჩანს, რომ უკანასკნელი სამი ბიჯი ელემენტების განლაგებაზე გავლენას ვერ ახდენს. ამიტომ თუკი დავიმახსოვრებთ იმ ფაქტს – იყო თუ არა ცვლილება უკანასკნელი ბიჯის დროს, ცვლილების არარსებობის შემთხვევაში პროცესი შეგვიძლია შევწყვიტოთ. ალგორითმის კიდევ მეტად გაუმჯობესება შეიძლება, თუკი დავიმახსოვრებთ არამარტო გაცვლის ფაქტის არსებობას, არამედ უკანასკნელი გაცვლის ადგილსაც (ანუ შესაბამისი ელემენტის ინდექსს). ცხადია, რომ ამ ინდექსის ზემოთ მოთავსებული ელემენტები უკვე დალაგებულია და შემოწმებას აღარ საჭიროებენ. მცირედი დაკვირვებაა საჭირო იმის შესამჩნევად, რომ მასივის "მძიმე ბოლოში" არასწორად მოთავსებული "მსუბუქი" ბუშტულა ციკლის ერთი გავლით მოძებნის საჭირო ადგილს, ხოლო "მსუბუქ ბოლოში" განლაგებული "მძიმე" ბუშტულა საკუთარ ადგილამდე თითო-თითო პოზიციაზე გადაადგილებით მივა. მაგალითად მასივი

12 18 42 44 55 67 94 06

დალაგდება ციკლის ერთი გავლით, ხოლო მასივს

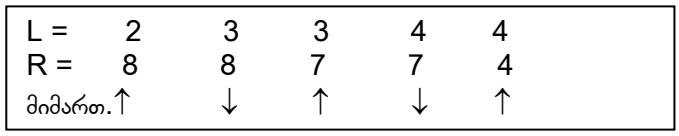
94 06 12 18 42 44 55 67

დასჭირდება ციკლის შვიდჯერ გავლა. ასეთი ასიმეტრიულობა თავისთავად გვკარნახობს შემდეგ გაუმჯობესებას – შევცვალოთ ციკლის გავლის მიმართულებები. ასეთ ალგორითმს უწოდებენ შეიკერულ სორტირებას (ShakerSort, "შეიკერ"-ს უწოდებენ ერთმანეთზე პირით დადებულ ორ ჭიქას, რომელთაც ანჯღრევენ ვერტიკალურად კოქტეილის მომზადებისათვის). მოვიყვანოთ პროგრამის კოდი:

```
ShakerSort (a,n)
L=2; R=n; k=n;
REPEAT {
        FOR j=R TO L STEP -1  {
                IF a[j-1]>a[j] THEN  { x=a[j-1]; a[j-1]=a[j]; a[j]=x; k=j  }       }
        L=k+1;
        FOR j=L TO R STEP  +1  {
                IF a[j-1]>a[j] THEN  { x=a[j-1]; a[j-1]=a[j]; a[j]=x; k=j  }       }
```

```
        R=k-1;
  UNTIL L>R }
```

ამ ალგორითმის მუშაობის პროცესი მოცემულია 4.4. ცხრილში:



44	06	06	06	06
55	44	44	12	12
12	55	12	44	18
42	12	42	18	42
94	42	55	55	44
18	94	18	42	55
06	18	67	67	67
67	67	94	94	94

ცხრილი 4.4.

ბუშტულების სორტირება ყველაზე ნელია სორტირების მეთოდებს შორის და ის უფრო მარტივი ალგორითმისა და ეგზოტიკური სახელწოდების გამო არის ცნობილი, ხოლო შეიკერული სორტირება წარამტებით გამოიყენება, თუკი წინასწარ ცნობილია, რომ მასივი თითქმის დალაგებულია. საშუალოდ ორივე მათგანი მოითხოვს n^2 ოპერაციას.

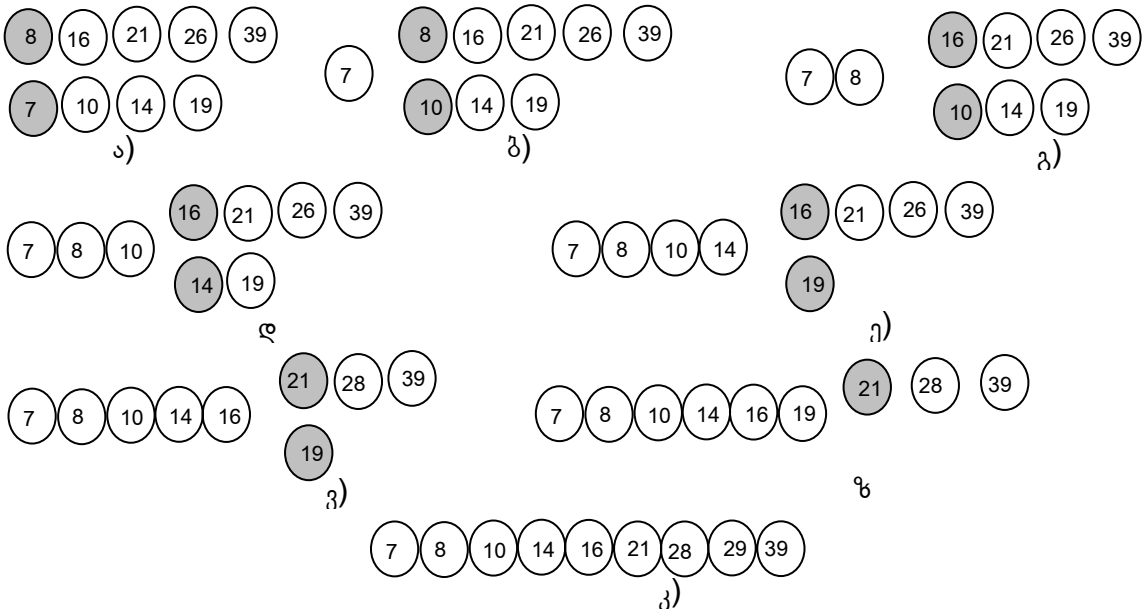
4.4. სორტირება შერწყმის მეთოდით (merge sort)

სორტირების ეს მეთოდი 1945 წელს შეიმუშავა ჯონ ფონ ნეიმანმა. ჩვენ განვიხილავთ ამ მეთოდის რეკურსიულ რეალიზაციას. რეკურსიის საშუალებით კარგად ხერხდება ამოცანის დაყოფა იმავე შინაარსის ქვეამოცანებად. ალგორითმის მუშაობა პირობითად შეიძლება გავანაწილოთ 3 ეტაპად:

1. ხდება საწყისი მასივის დაყოფა ორ თანაბარი ზომის ქვემასივად;
2. პირველ ეტაპზე მიღებული მასივები ცალ-ცალკე სორტირდება;
3. ხდება დასორტირებული მასივების გაერთიანება (შერწყმა).

რეკურსიული ბიჯია გადასვლა მთლიანი ჯგუფიდან ორ ქვეჯგუფზე. მუშაობის პროცესში მიღებული ქვემასივების დაყოფა გრძელდება მანამ, ვიდრე თითო ელემენტიან მასივებამდე არ დავალთ. ესაა რეკურსიის ბაზა, იგი ემყარება იმ მოსაზრებას, რომ ერთი ელემენტისგან შედგენილი ჯგუფი ფაქტიურად დალაგებულია. შემდეგ ვიწყებთ მეზობელი სორტირებული ქვემასივების გაერთიანებას, ვიდრე არ მივიღებთ გადახარისხებულ საწყის მასივს. გაერთიანების პროცესსი არსებითია შევინარჩუნოთ დალაგება ზრდადობის მიხედვით. ჩვენ აღვწერთ ორი დახარისხებული მასივისგან ახალი დახარისხებული მასივის მიღების ერთ ბუნებრივ მეთოდს, რომელიც ჯერ გრაფიკულად წარმოვიდგინოთ შემდეგი ნახაზის საშუალებით (რუხი ფერით ის ელემენტებია აღნიშნული,

რომელთა შედარებაც ხდება მოცემულ ეტაპზე):



ამგვარად ალგორითმის მთავარ ნაწილს წარმოადგენს რიცხვთა ორი დალაგებული ჯგუფის შერწყმა ერთ ჯგუფად. განვიხილოთ ორი რიცხვითი მასივი $a_0 \le a_1 \le \cdots \le a_n$ და $b_0 \le b_1 \le \cdots \le b_m$. ყველაზე ადვილია ისინი შევრწყათ მესამე ($n + m$) განზომილების მასივის გამოყენებით. შესაბამისი პროგრამის კოდი მოყვანილია, მაგალითად, [4]-ში 332გვ.-ზე (გარკვეული გამუჯობესებებით [6]-ში 164გვ.-ზე და [2]-ში, 2001წ.-ის გამოცემა 30გვ.). შერწყმა ხდება რიგრიგობით a და b მასივების ელემენტების შედარებით და შესაბამის ადგილზე c მასივში მინიმუმის გადატანით. ცხადია, უნდა ვიზრუნოთ იმ შემთხვევაზე როდესაც ერთერთი a და b–ს შორის ამოიწურება და დარჩენილი ნაწილი მეორესი უნდა პირდაპირ გადავიტანოთ c–ში. ამ იდეის მოყვანილ რეალიზაციას საკმაოდ ბევრი ნაკლი აქვს, ალგორითმული თვალსაზრისით როგორებიცაა: დიდი დამატებითი მეხსიერების გამოყენება, ციკლში მოტრიალე ზედმეტი შემოწმებები და ა.შ. , მაგრამ აქვს ღირსებაც – ის ადვილად აღსაქმელია.

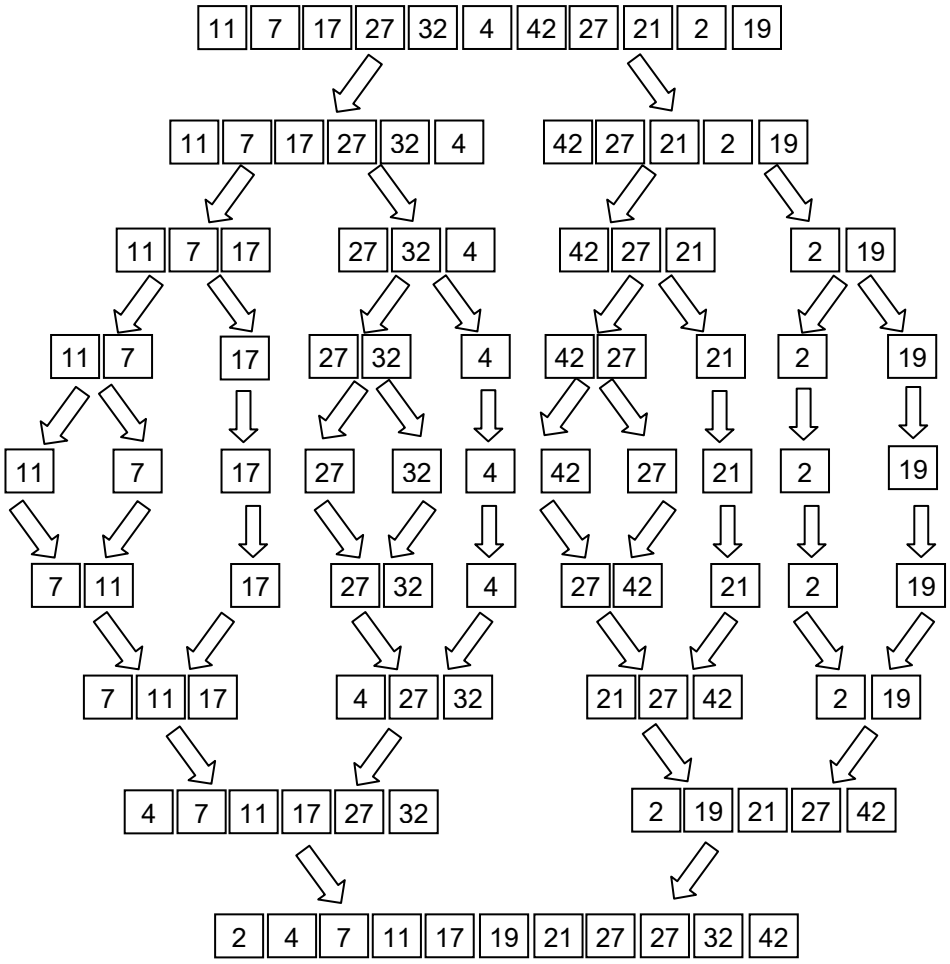
არსებობს შერწყმის ისეთი რეალიზაციები, რომლებიც იმავე მასივებში ათავსებენ დასხარიხებელ ელემენტებს, თუმცა მათი პროგრამული გამართვა მაღალ ოსტატობას მოითხოვს. განსაკუთრებით ეფექტურია ბმული სიების გამოყენება. შეიძლება ითქვას, რომ ამ შემთხვევაში ორმხრივ საინტერესო ეფექტებს აქვს ადგილი. ბმული სიების გამოყენებით შერწყმით სორტირებაში დამატებითი მეხსიერების პრობლემა იხსნება, ხოლო თავის მხრივ სორტირების ყველა მეთოდთაგან სწორედ შერწყმით სორტირებაა რეკომენდირებული ბმული სიების დასალაგებლად.

MERGE-SORT(*a*,*p*,*r*) - ის ფსევდოკოდს აქვს სახე (იხ. [2], 26-27 გვ.):

```
MERGE-SORT(a, p, r)
if (  p < r ) {
    q = (p + r)/2    //  იგულისხმება მთელად გაყოფა
   MERGE-SORT(a,p,q)
   MERGE-SORT(a, q+1, r)
   MERGE(a, p, q, r)
}
```

მთელი მასივის დალაგებისთვის საკმარისია გამოვიძახოთ ფუნქცია MERGE-SORT(*a*,0,n-1).

განვიხილოთ ალგორითმის მუშაობა კონკრეტულ მაგალითზე:



ალგორითმის ანალიზისთვის ჯერ განვიხილოთ შერწყმა. საუკეთესო შემთხვევა გვექნება, როდესაც ერთი მასივის, მაგალითად a მასივის, ყველა ელემენტი ნაკლებია მეორე, b მასივის, ელემენტებზე. მაშინ შედარება შესრულდება n–ჯერ. თუ a და b მასივების ელემენტები განლაგებულნი არიან მიმდევრობით $a_0 \le b_0 \le a_1 \le b_1 \ldots$ მაშინ ეს იქნება უარესი შემთხვევა და შედარება შესრულდება $n + m - 1$ –ჯერ.

ახლა გადავიდეთ რეკურსიის სირთულის ანალიზზე. რეკურენტული ფორმულა დავწეროთ ცალცალკე უარესი და უკეთესი შემთხვევებისთვის. როდესაც გვაქვს ერთელემენტიანი მასივი, მაშინ შედარებათა რაოდენობა ნულია. საუკეთესო შემთხვევისთვის გამოდის ფომულა

$$T(n) = \begin{cases} 0 & \text{როდესაც } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \frac{n}{2} & \text{როდესაც } n > 1 \end{cases}$$

და უარესი შემთხვევისთვის

$$T(n) = \begin{cases} 0 & \text{როდესაც } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + n - 1 & \text{როდესაც } n > 1 \end{cases}$$

საუკეთესო შემთხვევისთვის პირდაპირ მუშაობს ძირითადი რეკურენტული ლემა (შეგვიძლია უშუალოდ მათემატიკური ინდუქციის გამოყენებაც) და სირთულე არის $O(n \log_2 n)$.

უარესი შემთხვევისთვის პასუხს გვაძლევს ძირითადი რეკურენტული ლემის ანალოგიური მსჯელობები (ცხადია, შეგვიძლია უშუალოდ მათემატიკური ინდუქციის გამოყენებაც), რომლებიც განვიხილოთ ცალკე. სიმარტივისთვის ავიღოთ შემთხვევა, როდესაც $n = 2^k = 2^{\log_2 n}$

მაშინ

$$\boldsymbol{T(n)} = T\left(2^k\right) = 2\cdot T\left(\tfrac{2^k}{2}\right) + 2^k - 1 =$$

$$= 2\cdot T\left(2^{k-1}\right) + 2^k - 1 = 2\cdot\left(2\cdot T\left(2^{k-2}\right) + 2^{k-1} - 1\right) + 2^k - 1 = 2^2\cdot T\left(2^{k-2}\right) + 2^k - 2 + 2^k - 1$$

$$= \ldots = 2^k\cdot T(1) + 2^k\cdot k - \sum_{i=0}^{k-1} 2^i = 2^k\cdot k - \sum_{i=0}^{k-1} 2^i = n\cdot\log_2 n - \sum_{i=0}^{\log_2 n-1} 2^i = n\cdot\log_2 n - n + 1.$$

ამგვარად უარესი შემთხვევაც იგივე $O(n\log_2 n)$ რიგისაა რაც საუკეთესო, და ბუნებრივია ამავე რიგის იქნება საშუალო შემთხვევაც.

4.6. ჩქარი სორტირება (QUICKSORT)

ამ მეთოდს ასევე უწოდებენ სორტირებას გაყოფის საშუალებით. მისი ავტორი გახლავთ ჰოარი. ალგორითმის იდეა იმაში მდგომარეობს თავდაპირველად ხდება გადანაცვლებები დიდ მანძილებზე. ვთქვათ, ჩვენ გვაქვს n ელემენტი, რომელიც დალაგებულია შებრუნებული მიმდევრობით. მათი სორტირება შესაძლებელია n/2 რაოდენობის გაცვლით. ჯერ გავუცვლით ადგილებს პირველ და ბოლო ელემენტებს, შემდეგ – მეორეს და ბოლოსწინას და ა.შ. თანმიმდევრულად ვიმოძრავებთ ორივე ბოლოდან. მართალია, ასეთი ხერხის გამოყენებისათვის მასივი თავიდან უსათუოდ შებრუნებული უნდა იყოს, მაგრამ ამ მაგალითიდან საყურადღებო დასკვნის გამოტანა შეიძლება.

გამოვიყენოთ ასეთი ალგორითმი: შემთხვევითად ავარჩიოთ რომელიმე ელემენტი (ვთქვათ, x) და განვიხილოთ მის მარცხნივ მდებარე ელემეტები მანამ, ვიდრე არ ვიპოვით $a_i>x$ ელემენტს. შემდეგ განვიხილოთ x-ის მარჯვნივ მდებარე ელემენტები, სანამ არ ვიპოვით $a_j<x$. ახლა გავუცვალოთ ადგილი ამ ორ ელემენტს და გავგრძელოთ ეს პროცესი, ვიდრე ორივე მხრიდან არ მივაღწევთ x ელემენტამდე. შედეგად ჩვენ მივიღებთ მასივს, რომელიც გაყოფილი იქნება ორ ნაწილად და მარცხენა ნაწილში მოთავსებული ნებისმიერი ელემენტი მარჯვენა ნაწილში მყოფ ნებისმიერ ელემენტზე ნაკლები იქნება. მოვიყვანოთ ამ ალგორითმის პროგრამული კოდი:

```
 PARTITION (A, n)
i=1;    j=n;
x შევარჩიოთ შემთხვევითად
REPEAT   {
          WHILE A[i]<x   { i=i+1   }
          WHILE x < A[j]  { j=j-1   }
          IF i <= j   THEN   {  w=a[i]; a[i]=a[j]; a[j]=w;  i=i+1;  j=j-1   }
UNTIL   i>j   }
```

თუკი მასივში, რომლის ელემენტებია

44 55 12 42 94 18 06 67

x ელემენტად ავიღებთ 42-ს, მაშინ მოხდება ორი გადანაცვლება – 18↔44 და 06↔55 და მასივი მიიღებს სახეს:

18 06 12 |42| 55 44 94 67

ცხადია, რომ ჩვენ ჯერ კიდევ არ მიგვიღწევია სასურველი მიზნისათვის და მასივი ჯერ სორტირებული არ არის, მაგრამ ამ მიზნის მისაღწევად საკმარისია რეკურსიულად გავიმეოროთ იგივე პროცესი მიღებული ნაწილების მიმართ მანამ, სანამ გაყოფის შედეგად მიღებული ნაწილი ერთელემენტიანი არ გახდება. ალგორითმი აღწერილია ქვემოთ მოყვანილი პროგრამაში, რომელსაც საწყის მონაცემებად გადაეცემა – L=1 და R=n, სადაც n ელემენტების რაოდენობაა საწყის მასივში:

```
QUICKSORT (L,R)
i=L;     j=R;
x=a[(L+R)  DIV 2]
REPEAT   {
          WHILE A[i]<x   { i=i+1   }
          WHILE x < A[j]  { j=j-1   }
          IF i <= j   THEN   {  w=a[i]; a[i]=a[j]; a[j]=w;  i=i+1;  j=j-1   }
UNTIL   i>j   }
IF L<j   THEN     QUICKSORT (L,j)
IF i<R  THEN     QUICKSORT (i,R)
```

როგორც ყველა სხვა შემთხვევაში, არსებობს ალგორითმის რეალიზაციის არარეკურსიული გზაც, თუმცა ის შედარებით მოუქნელია და საჭიროებს დამატებით ოპერაციებს გაყოფის ადგილების დასამახსოვრებლად.

QUICKSORT-ის მუშაობის საშუალო დროა $n\log n$, იგი ნაკლებად ეფექტურია მცირე ზომის მასივებისათვის, ხოლო მისთვის ყველაზე ცუდი შემთხვევაა, როცა საწყისი მასივი უკვე დალაგებულია. ასეთ შემთხვევაში მისი მუშაობის დრო n^2-მდე იზრდება.

ერთ-ერთ უმთავრეს მომენტს ალგორითმში წარმოადგენს x ელემენტის არჩევა. ჯერჯერობით არ არსებობს ამ პრობლემის გადაჭრის თეორიულად დასაბუთებული გზა. თავად ალგორითმის ავტორი – ჰოარი თვლიდა, რომ უმჯობესია x ელემენტი შემთხვევითად ავარჩიოთ. თუმცა არსებობს განსხვავებული მოსაზრებებიც და მათ შორის ყველაზე პოპულარულია x ელემენტად მასივის მედიანის არჩევა (მასივის მედიანას უწოდებენ ისეთ ელემენტს, რომლის მნიშვნელობაც მეტია მასივის ელემენტების მნიშვნელობათა ნახევარზე და ნაკლებია მეორე ნახევრის მნიშვნელობებზე), თუმცა მისი მოძებნა მასივში ასევე დამატებით სირთულეს წარმოადგენს.

4.7. სორტირება გადათვლით

სორტირების ეს მეთოდის გამოყენება მხოლოდ კონკრეტულ შემთხვევებშია მიზანშეწონილი, როდესაც დასალაგებელი მასივის ელემენტების რიცხვითი მნიშვნელობები არის მთელი და მოთავსებულია შეზღუდულ დიაპაზონში. ამის ერთ –ერთი მიზეზია მეთოდია დროითი სირთულე, რომელიც პროპორციულია მასივის ელემენტების მნიშვნელობათა დიაპაზონისა და მასივის ელემენტების რაოდენობის ჯამისა, ხოლო მეორე მიზეზი მეხსიერების პრობლემაა, რადგან საჭიროა იმ ზომის მეორე მასივის აღწერა, რომელიც დასალაგებელი მასივის უდიდესი ელემენტის მნიშვნელობას აღემატება.

გადათვლით სორტირებაში შემავალი მასივის ელემენტები განიხილება როგორც სხვა (დამხმარე) მასივის ინდექსები. ალგორითმის იდეა შემდეგია: თუ მასივის ნებისმიერი x ელემენტისათვის წინასწარ დავთვლით, თუ ამ მასივის რამდენი ელემეტია x-ზე ნაკლები (ვთქვათ m), მაშინ ის შეგვიძლია გამომავალ მასივში პირადაპირ ჩავწეროთ (m+1)–ე ადგილზე, ანუ ინდექსით m. თუ შემავალ მასივში გვხვდება ერთმანეთის ტოლი რიცხვები, მაშინ დამატებით უნდა ვიზრუნოთ, რომ ტოლი რიცხვები ერთმანეთის მეზობლად და ძველი რიგის შენარჩუნებით განლაგდეს.

განვიხილოთ პროგრამული კოდი რომელიც სხვა მეთოდებთან შედარებით მარტივი აღსაქმელია. შემომავალი მასივი არის a, c არის დამხმარე მასივი, ხოლო ხოლო b არის გამომავალი მასივი.

ინიციალიზაციის შემდეგ (სტრიქონი 1), ჯერ b–ში თავსდება a მასივის იმ ელემენტების რაოდენობა, რომლებიც i ინდექსის ტოლია (სტრიქონები 4 და 5), შემდეგ კი იწერება იმ ელემენტების რაოდენობა, რომლებიც არ აღემატებიან i ინდექსს (სტრიქონები 6 და 7). ბოლოს, ყოველი ელემენტი თავსდება b მასივში მის ადგილზე (სტრიქონები 8 და 9). ეს ადგილი განისაზღვრება შემდეგნაირად. თუ შემომავალ მასივში ყველა ელემენტი განსხვავებულია, მაშინ დალაგებულ ანუ გამომავალ მასივში მოთავსდება c[a[i]]-1 ინდექსით, რადგან ზუსტად ამდენი ელემენტი არის x-ზე ნაკლები; თუ შემავალი მასივი შეიცავს ერთმანეთის ტოლ ელემენტებსაც, მაშინ პირველი ჩაწერის დროს მოთავსდება c[a[i]]-1 ინდექსით, მაგრამ ამავე დროს უნდა შემცირდეს c[a[i]] 1-ით.

მიუხედავად თავისი სპეციფიკისა, ამ მეთოდს ორი უპირატესობა აქვს ბევრ სხვა მეთოდებთან შედარებით: მისი შესრულების დრო, როგორც ჩანს ცხრილიდან, არის სიდიდის პროპორციული და ეს არის მდგრადი მეთოდი შემდეგი თვალსაზრისით: შემომავალ მასივში ერთმანეთის ტოლი ელემენტები გამომავალ (უკვე დალაგებულ) მასივში ინარჩუნებენ თავიანთ რიგს ერთმანეთის მიმართ.

#	კოდის სტრიქონი	ფასი	განმეორებათა რიცხვი
1	ცვლადების აღწერა: `int i,n,k; int a[n], b[n];` a მასივის და n სიდიდის შეყვანა, k სიდიდის განსაზღვრა	c_1	1 - ჯერ
2	`for (i=0; i<k; i++){`	c_2	(k+1) - ჯერ
3	c[i]=0;}	c_3	(k) - ჯერ
4	`for (i=0; i<n; i++) {`	c_4	(n +1) - ჯერ
5	c[a[i]]++; }	c_5	(n) - ჯერ
6	`for (i=1; i<k; i++) {`	c_6	(k) - ჯერ
7	c[i]+=c[i-1]; }	c_7	(k-1) - ჯერ
8	`for (i=n-1; i>=0; i--) {`	c_8	(n +1) - ჯერ
9	b[c[a[i]]-1]=a[i]; c[a[i]]--; }	c_9	(n) - ჯერ
10	b მასივის გამოტანა	c_{10}	1 - ჯერ

განვიხილოთ მაგალითი.

0	1	2	3	4	5	6	7	8
5	2	7	4	4	2	1	9	2
შემომავალი მასივი a								

0	1	2	3	4	5	6	7	8	9
0	1	3	0	2	1	0	1	0	1

დამხმარე მასივი c, 4-5 სტრიქონების შესრულების შემდეგ. c არის 10 ელემენტიანი მასივი, რადგან a მასივის უდიდესი ელემენტი არის 9 – ის ტოლი. ამ ეტაპზე, c მასივში აღრიცხულია a მასივის ელემენტების რაოდენობები ცალ – ცალკე.

0	1	2	3	4	5	6	7	8	9
0	1	4	4	6	7	7	8	8	9

დამხმარე მასივი c, 6-7 სტრიქონების შესრულების შემდეგ. ამ მომენტისათვის, c მასივი უკვე შეიცავს ინფორმაციას იმის შესახებ, თუ რამდენ ელემენტს არ აღემატება a მასივის თითოეული წევრი.

0	1	2	3	4	5	6	7	8
0	0	0	2	0	0	0	0	0
0	0	0	2	0	0	0	0	9
1	0	0	2	0	0	0	0	9
1	0	2	2	0	0	0	0	9
1	0	2	2	0	4	0	0	9
1	0	2	2	4	4	0	0	9
1	0	2	2	4	4	0	7	9
1	2	2	2	4	4	0	7	9
1	2	2	2	4	4	5	7	9
დამხმარე მასივი b, ივსება პროგრამის 8-9 სტრიქონების ციკლის მიხედვით. ცხრილში ნაჩვენებია მასივის შევსების თანმიმდევრობა ბიჯების მიხედვით. რუხი ფერით მითითებულია ის ელემენტები, რომლებიც უკვე განთავსდნენ კუთვნილ ადგილას გამომავალ მასივში.								

4.8. წრფივი ძებნა. ორობითი ძებნა

ძებნის ამოცანა ერთ-ერთი ყველაზე გავრცელებული ამოცანაა პროგრამირებაში და მისთვის ალგორითმის შერჩევა ხშირადაა დამოკიდებული მონაცემთა სტრუქტურაზე. ჩვენ განვიხილოთ ტიპიური შემთხვევა, როცა მონაცემთა არე ფიქსირებულია. ვთქვათ, მოცემულია n-ელემენტიანი მასივი და საჭიროა მასში მოიძებნოს ელემენტი, რომლის მნიშვნელობა x-ის ტოლია. თუკი რაიმე დამატებითი ინფორმაცია საძებნ ელემენტზე ან საწყის მასივზე არ არსებობს, მაშინ ჩვენ უბრალოდ თანმიმდევრობით უნდა შევადაროთ მასივის ყველა ელემენტი x-ის მნიშვნელობას, ვიდრე არ ვიპოვით საძებნ ელემენტს ან არ დავრწმუნდებით, რომ ასეთი ელემენტი მასივში არ არის. ძებნის ასეთ მეთოდს უწოდებენ **წრფივ ძებნას** და მისი დამთავრების პირობებია: ა) ელემენტი მოძებნილია, ანუ $a_i=x$; ბ) განხილულია მასივის ყველა ელემენტი და დამთხვევა არ აღმოჩნდა.

```
LINEARFIND (A, n, x)
i=0
WHILE (i<N) AND (A[i]≠x)   {  i=i+1   }
```

შევნიშნოთ, რომ n-ელემენტიანი მასივი აქ ინდექსირებულია 0-დან (n-1)-მდე და ამ ალგორითმის ოდნავ დაჩქარება შესაძლებელია იმის ხარჯზე, რომ მასივს ბოლო ელემენტად დავუმატოთ საძებნი x-ის მნიშვნელობა და იმის შემოწმება, მივაღწიეთ თუ არა მასივის ბოლოს, საჭირო აღარ არის. ასეთ ხერხს უწოდებენ **ბარიერის მეთოდს** და თუ ამოცანის პასუხად პროგრამამ n გამოიტანა, ცხადია, რომ საძებნი ელემენტის მასივში არ არსებობს. ბარიერის გამოყენების დროს მასივი აღიწერება 0-დან n-მდე. წრფივ ძებნას ბარიერის გამოყენებით ექნება სახე:

```
A[n]=x;     i=0
WHILE A[i]≠x   {  i=i+1   }
```

წრფივი ძებნის კიდევ მეტად დაჩქარება შეუძლებელია. თუმცა ხშირ შემთხვევაში მასივი, რომელშიც ხორციელდება ძებნა დალაგებულია და ეს ძებნის პროცესის მნიშვნელოვნად დაჩქარების საშუალებას იძლევა. ალგორითმის იდეა ასეთია: ავიღოთ მასივის რომელიმე a_i ელემენტი და შევადაროთ x-ს. თუ x მეტია, მაშინ ძებნა უნდა გავაგრძელოთ i-ის მარცხნივ, ხოლო თუ x ნაკლებია – i-ის მარჯვნივ (ტოლობის შემთხვევაში ძებნა შეწყდება). თუკი a_i ელემენტად ავიღებთ მასივის ზუსტად შუა ადგილზე მყოფ ელემენტს, საძებნი არე პირველი შედარების შემდეგ 2-ჯერ შემცირდება. შემდეგ იმავე დარჩენილი არისათვის გამოვიყენებთ იმავე პრინციპს მანამ, სანამ ახალ საზღვრად აღებული

რომელიმე ელემენტი x-ს არ დაემთხვევა, ან არ მივიღებთ x-საგან განსხვავებულ ორ მეზობელ ელემენტს (ამ შემთხვევაში x მასივში არ მოიძებნა). ძებნის ასეთ მეთოდს უწოდებენ **ორობით (ბინარულ) ძებნას**.

```
BINARYFIND (A, n, x)
L=0;   R=n
WHILE L<R   {
        m=(L+R) DIV 2
        IF A[m]<x THEN L=m+1 ELSE R=m        }
```

საძებნი არის საწყისი საზღვრებია 0 (მარცხენა) და n (მარჯვენა). ყოველ ბიჯზე ხდება ერთ-ერთი საზღვრის შეცვლა საძებნი არის შუა ელემენტით, ვიდრე არ იქნება ნაპოვნი x, ან საზღვრები ერთმანეთს არ დაემთხვევიან.

ალგორითმის სიჩქარე ძალზე მაღალია. მაგალითად, მილიარდი ელემენტის შემცველი მასივის შესამოწმებლად სულ 30 იტერაციაა საჭირო.

4.1. Sorta სორტირება

(USACO, 2000-01 წელი, შემოდგომა, "ნარინჯისფერი" დივიზიონი)

დაწერეთ პროგრამა, რომელიც კითხულობს ორ დადებით მთელ რიცხვს შემავალი ფაილიდან და მათ მიერ განსაზღვრული დიაპაზონიდან ყველა მთელ რიცხვს თვით ამ რიცხვების ჩათვლით დაალაგებს ციფრთა რევერსიის (შებრუნების) მიხედვით. გამოტანისას თითო რიცხვი თითო სტრიქონში უნდა დაიბეჭდოს, ამასთან პირველად უნდა დაიბეჭდოს ყველაზე მცირე რიცხვი, ხოლო ბოლოს – ყველაზე დიდი. თუკი შებრუნების შემდეგ ორმა რიცხვმა მიიღო ერთნაირი მნიშვნელობა, ჯერ დაიბეჭდოს მათგან უმცირესი.

შემავალი მონაცემები: ერთადერთ სტრიქონში შემოდის ორი მთელი რიცხვი, რომლებიც წარმოადგენენ დასალაგებელი მიმდევრობის დასაწყისს და ბოლოს (1≤პირველი რიცხვი≤მეორე რიცხვი≤999999999). დიაპაზონში იქნება არაუმეტეს 100 რიცხვისა.

მაგალითი. ვთქვათ Sorta-სორტირება მოსახდენია 19..21 დიაპაზონზე. {19,20,21} რიცხვების შებრუნებით მივიღებთ {91,2,12}, რომლებიც დალაგდებიან ასე {2,12,91}, ამიტომ გამოსატან ფაილში უნდა მივიღოთ {20,21,19}.

sort.in	sort.out
22 39	30 31 22 32 23 33 24 34 25 35 26 36 27 37 28 38 29 39

მითითება. 100-ელემენტიანი ორგანზომილებიანი მასივის ერთ სტრიქონში ჩავწეროთ რიცხვები შემომავალი დიაპაზონიდან, მეორე სტრიქონში ჩავწეროთ შესაბამისი ელემენტის შებრუნებული მნიშვნელობა. მასივი დავალაგოთ მეორე სტრიქონის მიხედვით და დავბეჭდოთ პირველი სტრიქონი. შეიძლება გამოვიყენოთ ბმული სიებიც.

4.2. არჩევნები

(ლ. ვოლკოვი)

კარიბის აუზის ერთ-ერთ კუნძულ-სახელმწიფოში ყველა გადაწყვეტილება ტრადიციულად მიიღებოდა ხმების უმრავლესობით მოქალაქეთა საერთო კრებაზე. მოქალაქეთა რიცხვი თავდაპირველად დიდი არ იყო, მაგრამ დროთა განმავლობაში ის საგრძნობლად გაიზარდა და საერთო კრებების მოწვევაც გართულდა. ამით ისარგებლა ერთ-ერთმა ადგილობრივმა პარტიამ, რომელსაც სურდა, რომ კანონიერი გზით მოსულიყო ხელისუფლებაში და მოახერხა საარჩევნო სისტემაში რეფორმის გატარება.

რეფორმის არსი შემდგომში მდგომარეობდა: კუნძულის ყველა ამომრჩეველი იყოფოდა k ჯგუფად, ამასთან რიცხობრივი თანაბრობა ამ ჯგუფებს შორის აუცილებელი არ იყო. ნებისმიერი საკითხის შესახებ კენჭისყრა ტარდებოდა თითოეულ ჯგუფში ცალ-ცალკე და თუ ჯგუფის ნახევარზე მეტი ხმას მისცემდა საკითხის დადებითად გადაწყვეტას, ითვლებოდა, რომ ჯგუფი მთლიანად თანახმაა საკითხის დადებითად გადაწყვეტაზე, წინააღმდეგ შემთხვევაში კი ითვლებოდა, რომ ჯგუფი ხმას აძლევს საკითხის უარყოფითად გადაწყვეტას. ჯგუფებში კენჭისყრის ჩატარების შემდეგ

განსაზღვრავდნენ თუ რამდენი ჯგუფი იყო საკითხის დადებითად გადაწყვეტის მომხრე და თუ ასეთი ჯგუფები ნახევარზე მეტი აღმოჩნდებოდა, საკითხი ითვლებოდა დადებითად გადაწყვეტილად.

ეს სისტემა თავდაპირველად სიხარულით იქნა მიღებული კუნძულის ამომრჩეველთა მიერ, როცა აღტაცებამ გაიარა, ცხადი გახდა ახალი სისტემის ნაკლოვანებები. აღმოჩნდა, რომ პარტია, რომელიც ამ სისტემას ნერგავდა, ზეგავლენას ახდენდა ამომრჩეველთა ჯგუფების ფორმირებაზე. ამის საშუალებით ისინი ახერხებდნენ მიეღოთ გარკვეული გადაწყვეტილებები ისე, რომ რეალურად არ ფლობდნენ ხმათა უმრავლესობას.

მაგალითისათვის ვთქვათ, კუნძულზე შეიქმნა ამომრჩეველთა სამი ჯგუფი შესაბამისად 5, 5 და 7 ადამიანის ოდენობით. მაშინ გადაწყვეტილების მისაღებად საკმარისია პარტიას ჰყავდეს 3-3 მხარდამჭერი პირველ ორ ჯგუფში, ანუ 6 ხმით მიიღებს გადაწყვეტილებას, რომლისთვისაც საერთო კრებაზე 9 ხმა იყო საჭირო.

დაწერეთ პროგრამა, რომელიც ამომრჩეველთა ჯგუფების მოცემული დაყოფისათვის განსაზღვრავს პარტიის მომხრეთა მინიმალურ რაოდენობას, რომელიც ჯგუფებში მათი გარკვეული განაწილების შემთხვევაში საკმარისია ნებისმიერი გადაწყვეტილების მისაღებად.

შემომავალი მონაცემები: პირველ სტრიქონში მოცემულია ნატურალური რიცხვი k<101 – ამომრჩეველთა ჯგუფების რაოდენობა. მეორე სტრიქონში მოცემულია k ცალი ჰარით გაყოფილი რიცხვი, რომლებიც აღწერენ ჯგუფებში ამომრჩეველთა რაოდენობას. ცნება "ხმათა უმრავლესობის" გასამარტივებლად ჩავთვალოთ, რომ როგორც ჯგუფების რაოდენობა, ასევე ამომრჩეველთა რაოდენობა ჯგუფებში კენტი რიცხვია. კუნძულის მოსახლეობა არ აღემატება 10001 ადამიანს.

გამომავალი მონაცემები: ერთადერთი ნატურალური რიცხვი, რომელიც აღნიშნავს გადაწყვეტილების მიღებისათვის პარტიის მომხრეთა მინიმალურ რაოდენობას.

შემავალი მონაცემების მაგალითი :	გამომავალი მონაცემი ნაჩვენები მაგალითისათვის:
3 5 7 5	6

მითითება. კონკრეტულ ჯგუფში ხმების უმრავლესობის მოსაპოვებლად საჭიროა ხმების ნახევარზე მეტი. პარტიისთვის მომგებიანი იქნება უფრო მცირერიცხოვანი ჯგუფების დაკომპლექტება. ამიტომ რაოდენობის ზრდადობის მიხედვით დავალაგოთ ამომრჩეველთა ჯგუფები (დალაგების მეთოდს მნიშვნელობა არა აქვს). ავიღოთ მათგან პირველიდან k div 2+1 ჯგუფის ჩათვლით და თითოეული მათგანისათვის ასევე გამოვთვალოთ ნახევარს პლიუს ერთი. ამ რიცხვების ჯამი იქნება ამოცანის პასუხი.

4.3. ქოლგები
(USACO, 2003-04 წელი, დეკემბერი, "მწვანე" დივიზიონი)

კოკისპირულად წვიმს. ძროხები, რომელთაც სიმშრალეში სურთ ბალახობა, შეიკრიბნენ N (1≤N≤25000) ქოლგის ქვეშ. საძოვარი წარმოადგენს ერთიან ბალახოვან მონაკვეთს. ყოველი ქოლგა მთლიანად ფარავს საძოვრის გარკვეულ ფრაგმენტს, თუმცა ძროხებმა ქოლგები იმგვარად განალაგეს, რომ მათ მიერ დაფარული ფრაგმენტები შეიძლება რამდენჯერმე გადაიფარონ.

თქვენი ამოცანაა მოძებნოთ ისეთი ქოლგების უდიდესი რაოდენობა, რომლებიც ერთი ქოლგით არიან დაფარულები.

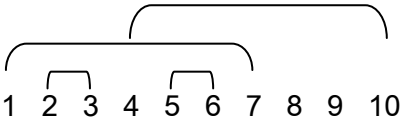
ამოცანის სახელია: cover.

შემავალი მონაცემების ფორმატი(ფაილი cover.in): 1 სტრიქონი: მოცემულია მთელი N რიცხვი; 2..N+1 სტრიქონებში: მთელი A და B (1≤A<B≤2 000 000 000) რიცხვები, რომლებიც წარმოადგენენ ქოლგით დაფარულ საძოვრის ინტერვალებს. ქოლგათა არცერთ წყვილს არ გააჩნია საერთო სასაზღვრო წერტილი.

გამომავალი მონაცემების ფორმატი(ფაილი cover.out): ერთადერთ სტრიქონში ნაჩვენები უნდა იყოს ერთადერთი მთელი რიცხვი – იმ ქოლგების მაქსიმალური რაოდენობა, რომლებიც ერთი ქოლგით არიან დაფარულები.

შემავალი მონაცემების მაგალითი :	გამომავალი მონაცემი ნაჩვენები მაგალითისათვის:
4 1 7 2 3 5 6 4 10	2

განმარტება: პირველი ქოლგა ფარავს მეორე და მესამე ქოლგებს (იხ. ნახ. 4.4).



ნახ. 4.4.

მითითება. გადაგნომროთ ქოლგები და მოვახდინოთ მათი სორტირება (უმჯობესია QUICKSORT-ით) ცალკე მარცხენა და ცალკე მარჯვენა ბოლოების მიხედვით. თუკი ქოლგა მარცხენა ბოლოთა მიხედვით მოხვდა k1 ადგილზე, ხოლო მარჯვენა ბოლოების მიხედვით – k2 ადგილზე და ამასთან k1<k2, მაშინ მას გადაუფარავს k2-k1 ქოლგა. იმ პირობას, რომ k1<k2, პრინციპული მნიშვნელობა აქვს, რადგან თუ ქოლგის მარჯვენა

კოორდინატის მარცხნივ იმყოფება გარკვეული რაოდენობის ქოლგათა მარჯვენა კოორდინატები, ცხადია, რომ ამავე ქოლგების მარცხენა კოორდინატებიც მისგან მარცხნივ იქნება.

4.4. წრე მართკუთხედებისაგან

(რუსეთის მოსწავლეთა ოლიმპიადა, დასკვნითი ტური, 1998-99 წელი)

სიბრტყე დაყოფილია ერთნაირი M×N ზომის მართკუთხედებად, რომელთა გვერდებიც კოორდინატთა ღერძების პარალელურია და წვეროებად, რომლებიც განლაგებულნი არიან (M×i, N×j) წერტილებში, სადაც i და j წარმოადგენენ ყველანაირ მთელ რიცხვებს. ვთქვათ, ამ სიბრტყეზე მოცემული P(x,y) წერტილი მთელი კოორდინატებით. მანძილი P წერტილიდან რომელიმე მართკუთხედამდე ვუწოდოთ უმცირეს მანძილს P-დან ამ მართკუთხედის წერტილებამდე (მართკუთხედის საზღვრების ჩათვლით). მაგალითად, მანძილი წერტილიდან იმ მართკუთხედამდე, რომელშიც თავად მდებარეობს, 0-ის ტოლია.

დაწერეთ პროგრამა, რომელიც გამოიტანს P-დან არაუმეტეს L მანძილით დაშორებულ მართკუთხედებს. მართკუთხედები ჩამოთვლილნი უნდა იქნან P-დან დაშორების მიხედვით, არაკლებადი თანმიმდევრობით.

შემავალი მონაცემები: INPUT.TXT ტექსტური ფაილში მოცემულია რიცხვები M, N, L, x и y (0<M≤10, 0<N≤10, 0≤L≤300, –30000<x,y<30000), რომლებიც გაყოფილნი არიან ჰარებით ან სტრიქონის გადამყვანით.

გამომავალი მონაცემები: OUTPUT.TXT ტექსტური ფაილი უნდა შეიცავდეს საძებნი მართკუთხედების ქვედა მარცხენა კუთხის კოორდინატებს ზემოთ აღწერილი ირობების დაცვით. P-დან თანაბარი მანძილით დაშორებული მართკუთხედების გამოტანის რიგს მნიშვნელობა არა აქვს.

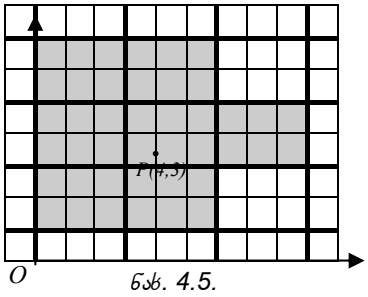
ქვემოთ მოყვანილია შემავალი და გამომავალი მონაცემების მაგალითები ნახ. 4.5-ის შესაბამისად.

შემავალი ფაილის მაგალითი:

```
3 2 2
4 3
```

გამომავალი ფაილის მაგალითი:

```
3 2
3 0
0 2
3 4
0 0
0 4
6 2
```



ნახ. 4.5.

მითითება. ამოცანის ამოხსნისას შეზღუდვა მეხსიერებაზე წარმოადგენდა 640კბ-ს, ხოლო დრო თითოეული ტესტისათვის წარმოადგენდა 10წმ-ს PentiumII-ზე. ამიტომ თუკი რადიუსის მიკნით მოხვედრილი მართკუთხედებს უბრალოდ დავალაგებდით L (x,y) წერტილიდან დაშორების მიხედვით და შემდეგ შევეცდებოდით მათ გამოტანას საჭირო თანმიმდევრობით, დიდი L-ებისათვის მეხსიერების პრობლემა შეგვექმნებოდა. მაგალითად, თუ L=300 და M=N=1, მაშინ მართკუთხედების რაოდენობა დაახლოებით ტოლია $\pi L^2 \approx 3.14\times300^2>3\times300^2 =3\times90\times10^3 = 270\times10^3$. თუ გავითვალისწინებთ, რომ ყოველი მართკუთხედისათვის უნდა შევინახოთ ორი მთელი რიცხვი (ქვედა მარჯვენა კუთხის კოორდინატები), რომელთაგან თითოეული მეხსიერების 2 ბაიტს მოითხოვს, ცხადია, რომ სა ჭირო მეხსიერების მოცულობა 1000კბ-ს აჭარბებს (4×270კბ). მაშასადამე, ასეთი მიდგომა გაუმართლებელია.

მეხსიერების პრობლემა შეიძლება მოგვარდეს, თუ მართკუთხედებს გავყოფთ s ჯგუფად, რომლებიც კონცენტრულ რგოლებად იქნებიან განლაგებულნი ერთმანეთის მიმართ შიგა kd და გარე (k+1)×d რადიუსებით, სადაც d – რგოლის სიგანეა, s=1+L div d, ხოლო k=0,...,s-1. ჯგუფები დამუშავდება თანმიმდევრულად k-ს ზრდის მიხედვით, თითოეული ჯგუფის შიგნით მოხდება სორტირება და შემდეგ მათი გამოტანა შესაბამისი მიმდევრობით. ასეთი მიდგომის დროს წარმოიქმნება სწრაფქმედების პრობლემა, რადგან ცალკეული ჯგუფების სორტირება დიდ დროს მოითხოვს.

*ამოცანის ამოხსნის კიდევ ერთი გზაა, საჭირო მართკუთხედების თანმიმდევრული გენერაცია. მართკუთხედების ყოველი ჰორიზონტალური ზოლისათვის შესაძლებელია შევინახოთ მასში მოთავსებული უკვე გამოტანილი მართკუთხედებისაგან შედგენილი მონაკვეთის მარცხენა და მარჯვენა ბოლოების კოორდინატები (თითოეული ზოლის შიგნით უკვე გამოტანილი მართკუთხედები აუცილებლად მიყოლებით იქნებიან განლაგებული და მათ შორის წყვეტა შეუძლებელია) . მეხსიერების დანახარჯი შეადგენს O(L)-ს (საჭიროა შევინახოთ L ზოლი (x,y) წერტილის ზემოთ და L ზოლი (x,y) წერტილის ქვემოთ). თუმცა შეიძლება ისევ წარმოიქმნას სწრაფქმედების პრობლემა. რადგან მომდევნო გამოსატანი მართკუთხედის ძებნა ყველა ზოლში გვიწევს, დროის ხარჯი პროპორციული იქნება გამოსატანი მართკუთხედების რაოდენობისა და მომდევნო მართკუთხედის ძებნაზე დახარჯული დროის ნამრავლისა. მართკუთხედების რაოდენობა დაახლოებით πL^2-ის ტოლია, ხოლო თანმიმდევრულად ძებნის შემთხვევაში, რომლის დროსაც ყველა ზოლი უნდა გავიაროთ, საჭიროა O(L) ოპერაცია. მათი ნამრავლი გვაძლევს $O(L^3)$-ს, რაც ასევე ვერ აკმაყოფილებს დროით შეზღუდვებს. ძებნის დრო შეიძლება მნიშვნელოვნად შევამცირდეს, თუ მონაცემთა ორგანიზაციისათვის გამოვიყენებთ **გროვას**.*

როგორც ზემოთ ვთქვით, ყოველი ზოლისათვის ვინახავთ უკვე გამოტანილი მართკუთხედების მონაკვეთს. ჩავთვალოთ, რომ (N×i, M×j) ქვედა მარცხენა კუთხის მქონე მართკუთხედის ნომერია (i, j). ვთქვათ, y

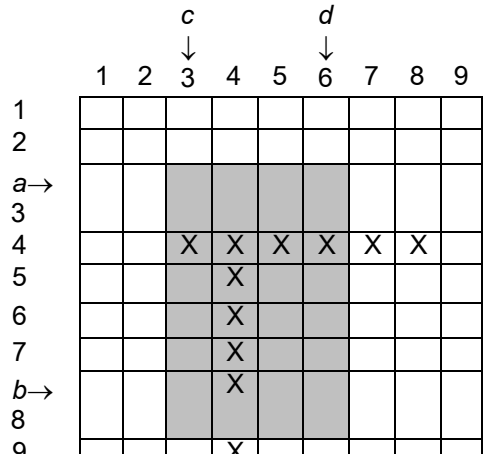
კოორდინატის მქონე ზოლში უკვე გამოტანილია მართკუთხედების l+1, ..., r – 1. მომდევნო მართკუთხედი, რომელიც უნდა გამოვიტანოთ y ზოლში, იქნება ან (l,y), ან (r,y). y ზოლიდან P(x,y) წერტილამდე მიმდინარე მანძილი ვუწოდოთ (l,y) და (r,y) მართკუთხედებიდან P(x,y) წერტილამდე მანძილებს შორის მინიმალურს.

ზოლები შევინახოთ გროვად, P(x,y) წერტილიდან მათი დამორების მიხედვით. მორიგ გამოსატან მართკუთხედს წარმოადგენს იმ ზოლის მარცხენა ან მარჯვენა მართკუთხედი, რომელიც მოცემულ მომენტში გროვის სათავეშია მოთავსებული. ამის შემდეგ გამოტანილი მართკუთხედებისაგან შედგენილი მონაკვეთი იზრდება და თუ მანძილი ზოლიდან P(x,y) წერტილამდე გაიზარდა, მაშინ მასივი აღარ წარმოადგენს გროვას და უნდა აღვადგინოთ გროვის ძირითადი თვისება. ზოლის "ჩაძირვას" შესაბამის ადგილამდე სჭირდება log L დრო. ამრიგად, ამოცანის ამოხსნის ალგორითმი გროვის საშუალებით მოითხოვს O(L) მეხსიერებას და O(L² log L) დროს.

4.4. ორი ბილიკი

(მოსწავლეთა საერთაშორისო ოლიმპიადა – IOI, 2002 წელი, სამხრეთ კორეა)

ბილიკი წარმოადგენს ბადის უჯრედთა ვერტიკალურ ან ჰორიზონტალურ მიმდევრობას, რომელიც 2 ან მეტი მიმდევრობით განლაგებული უჯრედისაგან შედგება. ერთი ვერტიკალური და ერთი ჰორიზონტალური ბილიკი განლაგებულია $N \times N$ ზომის ბადეზე. ნახ. 4.6-ზე ეს ორი ბილიკი მონიშნულია X სიმბოლოებით. ბილიკები შეიძლება იყვნენ ერთნაირი ან სხვადასხვა სიგრძის, მათ შეიძლება ჰქონდეთ ან არ ჰქონდეთ საერთო უჯრედი. თუკი, ნახ. 4.6-ზე გამოსახული შემთხვევის მსგავსად, წერტილი (4,4) შეიძლება განვიხილოთ, როგორც მხოლოდ ჰორიზონტალურის, ასევე როგორც ორივე ბილიკის წერტილი (ვერტიკალურისათვის ის იქნება განაპირა, ხოლო ჰორიზონტალურისათვის – შიგა წერტილი), ყოველთვის ჩაითვლება, რომ ის ორივე ბილიკს ეკუთვნის. აქედან გამომდინარე, ვერტიკალური ბილიკის განაპირა ზედა წერტილი იქნება (4,4) და არავითარ შემთხვევაში (5,4).



ბილიკების განლაგება თავდაპირველად ცნობილი არაა. თქვენი ამოცანაა – დაწეროთ პროგრამა, რომელიც გაარკვევს მათ მდებარეობას. დავარქვათ ჰორიზონტალურ ბილიკს *"ბილიკი1"* და ვერტიკალურს – *"ბილიკი2"*. ბადის ყოველი წერტილი წარმოდგენილია წერტილთა (r,c) წყვილით, რომლებიც შესაბამისად სტრიქონისა და სვეტის ნომრებს აღნიშნავენ. ბადის ზედა მარცხენა უჯრის კოორდინატებია (1,1). ყოველი ბილიკი მოიცემა უჯრედთა წყვილით $<(r_1, c_1), (r_2, c_2)>$. ნახაზი 1-ზე *ბილიკი1* მოიცემა როგორც <(4,3),(4,8)>, ხოლო *ბილიკი2* – როგორც <(4,4),(9,4)>.

ამ ამოცანის ამოხსნისას უნდა გამოიყენოთ ბიბლიოთეკა შესატანი მონაცემების მისაღებად, ამონახსნის მოსაძებნად და პასუხის გამოსატანად. კვადრატული ბადის N ზომის მისაღებად გამოიყენება ფუნქცია gridsize, რომელიც თქვენ უნდა გამოიძახოთ მუშაობის დაწყებისას. ბილიკების მდებარეობის განსაზღვრისათვის თქვენ შეგიძლიათ გამოიყენოთ მხოლოდ ფუნქცია rect(a,b,c,d), რომელიც შემავალ მონაცემებად ღებულობს მართკუთხა არის კოორდინატებს და აბრუნებს რიცხვ 1-ს, თუკი მითითებულ არეში გვხვდება ერთი მაინც უჯრედი, რომელიც რომელიმე ბილიკს ეკუთვნის და აბრუნებს 0-ს – წინააღმდეგ შემთხვევაში. მართკუთხა არე მოიცემა $[a,b]\times[c,d]$ სახით, სადაც $a \le b$ და $c \le d$. ამასთან საჭიროა, ყურადღება მიექცეს არგუმენტების რიგითობას. ნახაზი 1-ზე შესამოწმებელი არე შეფერადებულია რუხ ფერად. თქვენ უნდა დაწეროთ პროგრამა, რომელიც rect ფუნქციის შეზღუდული რაოდენობის გამოძახებით განსაზღვრავს ბილიკების ზუსტ მდებარეობას.

თქვენმა პროგრამამ უნდა გამოიტანოს პასუხი ბიბლიოთეკური ფუნქციის report (r1, c1, r2, c2, p1, q1, p2, q2)-ის გამოძახებით, სადაც ბილიკი1 მოიცემა წყვილით <(r1, c1),(r2, c2)>, ხოლო ბილიკი2 – <(p1, q1),(p2, q2)>. report ფუნქციის გამოძახება გამოიწვევს თქვენი პროგრამის მუშაობის დამთავრებას. მიაქციეთ ყურადღება, რომ ბილიკი1 ჰორიზონტალურია, ხოლო ბილიკი2 – ვერტიკალური და (r1, c1) ბილიკი1-ის მარცხენა ბოლოს კოორდინატებია, ხოლო (p1, q1) – ბილიკი2-ის ზედა ბოლოს კოორდინატები. ამრიგად r1= r2, c1 < c2, p1 < p2 და q1= q2. თუკი report ფუნქციის პარამეტრები არ აკმაყოფილებენ ამ შეზღუდვებს, მაშინ სტანდარტული გამოტანა გამოიტანს შეტყობინებას შეცდომის შესახებ.

შეზღუდვები:შესატანი მონაცემები ხელმისაწვდომია მხოლოდ ბიბლიოთეკური ფუნქციების gridsize და rect გამოძახების საშუალებით. N ბადის ზომა აკმაყოფილებს პირობას $5 \le N \le 10000$.

rect ფუნქციის გამოძახებათა რაოდენობა ყოველი ტესტისათვის არ უნდა აღემატებოდეს 400-ს. თუკი თქვენი პროგრამა შეასრულებს rect ფუნქციის გამოძახებას 400-ზე მეტჯერ, პროგრამის შესრულება შეწყდება. თქვენმა პროგრამამ უნდა შეასრულოს rect ფუნქციის 1-ზე მეტი გამოძახება და report ფუნქციის მხოლოდ 1 გამოძახება.

თუკი rect ფუნქციის გამოძახება მოხდება არასწორი პარამეტრებით, პროგრამის მუშაობა შეწყდება. თქვენმა პროგრამამ არ უნდა წაიკითხოს ან შექმნას რაიმე ფაილი, ასევე არ უნდა გამოიყენოს სტანდარტული შეტანა-გამოტანა.

ბიბლიოთეკა. თქვენს განკარგულებაშია ბიბლიოთეკა, რომელიც შეიცავს:

FreePascal Library (prectlib.ppu, prectlib.o)
function gridsize: LongInt;
function rect(a,b,c,d : LongInt) : LongInt;
procedure report(r1, c1, r2, c2, p1, q1, p2, q2 : LongInt);

ინსტრუქციები: თქვენი ფაილის rods.pas-ის კომპილაციისას გამოიყენეთ ოპერატორი uses prectlib; და ბრძანების სტრიქონი fpc –So –O2 –XS rods.pas

პროგრამა prodstool.pas წარმოადგენს მოცემული FreePascal ბიბლიოთეკის გამოყენების მაგალითს.

ექსპერიმენტირება: ბიბლიოთეკაზე ექსპერიმენტების ჩასატარებლად უნდა შექმნათ ტექსტური ფაილი rods.in. ფაილი უნდა შედგებოდეს სამი სტრიქონისაგან. პირველ სტრიქონში ჩაწერილი უნდა იყოს ბადის ზომა N. მეორე სტრიქონი უნდა შეიცავდეს ბილიკი1-ის კოორდინატებს: r1, c1, r2, c2; სადაც (r1, c1) ბილიკი1-ის მარცხენა ბოლოს კოორდინატებია. მესამე სტრიქონი უნდა შეიცავდეს ბილიკი2-ის კოორდინატებს: p1, q1, p2, q2; სადაც (p1, q1) ბილიკი2-ის ზედა ბოლოს კოორდინატებია.

თქვენი პროგრამის ამოშავების შემდეგ ფუნქცია report-ის გამოძახებით შეიქმნება გამოსატანი ფაილი rods.out. ეს ფაილი შეიცავს rect ფუნქციის გამოძახების რაოდენობას და ბილიკების ბოლოების კოორდინატებს, რომელსაც თქვენ გადასცემთ ფუნქცია report-ს. თუკი მუშაობის პროცესში მოხდა რაიმე შეცდომები ბიბლიოთეკასთან მიმართვისას, მაშინ შესაბამისი შეტყობინება შეცდომის შესახებ მოთავსებული იქნება ფაილში rods.out.

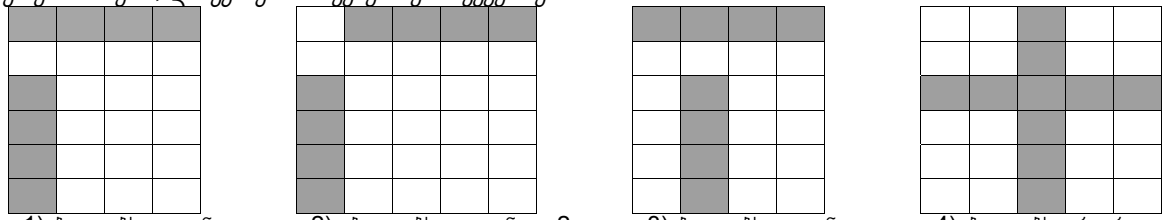
დიალოგი თქვენს პროგრამასა და ბიბლიოთეკას შორის ჩაიწერება ფაილში rods.log. ამ ფაილში ჩაიწერება rect ფუნქციის გამოძახების თანმიმდევრობა ფორმატით: “k : rect(a,b,c,d) = ans”, ეს ნიშნავს, რომ rect(a,b,c,d) ფუნქციის k-ურმა გამოძახებამ დააბრუნა რიცხვი ans.

შესატანი მონაცემების მაგალითი (rods.in)	გამოსატანი მონაცემების მაგალითი (rods.out)
9 4 3 4 8 4 4 9 4	20 4 3 4 8 4 4 9 4

შეფასება: თუკი თქვენი პროგრამა არღვევს რომელიმე ზემოთ დასახელებულ შეზღუდვას ან პასუხი არაკორექტულია – მიიღებთ 0 ქულას. თუ თქვენი პროგრამის პასუხი სწორია, მაშინ თქვენი შეფასება დამოკიდებულია rect ფუნქციის გამოძახებათა რაოდენობაზე ყოველი ტესტის დროს. თუ გამოძახებათა რიცხვი არ აღემატება 100-ს, მიიღებთ 5 ქულას, თუ 101-დან 200-მდეა – მიიღებთ 3 ქულას, ხოლო თუ 201-დან 400-მდე – 1 ქულას.

მითითება. ამოცანის ამოხსნა ძირითადად ორობით ძებნას ეყრდნობა. ორობითი ძებნის საშუალებით ჯერ მინიმუმამდე შევამციროთ საძებნი არე, ანუ კვადრატული ბადის ოთხივე მხრიდან მოვძებნოთ მაქსიმალური ზომის ისეთი მართკუთხა არეები, რომლებშიც ბილიკების არცერთი უჯრედი არ შედის. ამას მივაღწევთ ორობითი ძებნის 4-ჯერ გაშვებით. ორობით ძებნას N ზომის მასივის დასამუშავებლად სჭირდება $\lceil \log_2 N \rceil$ დრო, მაშასადამე ჩვენ მთლიანად დაგვჭირდება $4\lceil \log_2 N \rceil$ შეკითხვა. N-ის მაქსიმალური მნიშვნელობისათვის ეს წარმოადგენს 56 შეკითხვას, რადგან $\lceil \log_2 10000 \rceil = 14$.

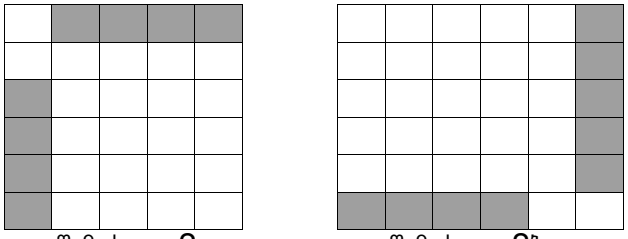
ამგვარად, ჩვენ მივიღეთ მინიმალური ზომის მართკუთხა არე, რომლის შიგნითაც ორივე ბილიკია მოთავსებული. მართალია, ჩვენ ამისათვის ერთი შეხედვით ძალიან ბევრი – 56 შეკითხვა დაჭირდეთ, მაგრამ როგორც ქვემოთ ვნახავთ, ინფორმაცია არც ისე მწირია. თუკი არ ჩავთვლით მობრუნებებსა და ღერძების მიმართ სიმეტრიებს, ორი ბილიკისათვის ურთიერთგანლაგების პრინციპულად განსხვავებული მხოლოდ 4 ვარიანტი არსებობს, რომლებიც ნაჩვენებია ნახ. 4.7-ზე. ის შემთხვევა, როცა ბილიკებს საერთო წერტილი აქვთ და არ ჰკვეთენ ერთმანეთს, დაიყვანება ნაჩვენებ შემთხვევებზე.



1) ბილიკებს ეკუთვნით 3 კუთხე
2) ბილიკებს ეკუთვნით 2 მოპირდაპირე კუთხე
3) ბილიკებს ეკუთვნით 2 მოსაზღვრე კუთხე
4) ბილიკებს არცერთი კუთხე არ ეკუთვნით

ნახ.4.7

იმის გასარკვევად, თუ ამ ოთხიდან რომელ შემთხვევასთან გვაქვს საქმე, საკმარისია შევამოწმოთ მიღებული მინიმალური მართკუთხედის ოთხივე კუთხე (ამისათვის საჭიროა შეკითხვა 1×1 ზომის მარკუთხედზე, რომელიც 1 უჯრედს შეიცავს). იმისდა მიხედვით, ამ ოთხიდან რომელი უჯრედები აღმოჩნდებიან დაფარული ბილიკით, შევძლებთ ვიმსჯელოთ თუ ძირითადი შემთხვევის რომელ მობრუნებასთან გვაქვს საქმე. ერთადერთ გამონაკლისს წარმოადგენს მეორე შემთხვევა, სადაც ბილიკებით დაფარულ ერთ და იგივე წერტილებს სრულიად განსხვავებული სიტუაციები შეესაბამებათ (იხ. ნახ. 4.8). თუმცა ამ ორი შემთხვევის ერთმანეთისაგან განსასხვავებლად საკმარისია ერთი უჯრედის შემოწმება ბილიკით დაფარული კუთხის მეზობლად.



შემთხვევა 2ა შემთხვევა 2ბ

ნახ. 4.8.

ამ შემთხვევების კლასიფიკაციის შემდეგ ჩვენ გვრჩება ბილიკების ზუსტი კოორდინატების დადგენა, ამასთან ჩვენ უკვე აუცილებლად ვიცით ერთი ბილიკის x-კოორდინატი და მეორის – y-კოორდინატი. სრული ინფორმაციის მისაღებად კვლავ გამოვიყენოთ ორობითი ძებნა, ოღონდ უკვე მინიმალური მართკუთხედის შიგნით. თითოეული შემთხვევისათვის ორობითი ძებნის გამოყენება ორჯერ მოგვიწევს. სულ ჯამში დაგვჭირდება $4 \cdot \lceil \log N \rceil + 4 + 1 + 2 \cdot \lceil \log N \rceil = 6 \cdot \lceil \log N \rceil + 5$ *შეკითხვის დასმა, რაც მაქსიმალური N=10000-სათვის არ აღემატება 89-ს.*

5. რიცხვთა თეორია

5.1. საწყისი დებულებები

იტყვიან, რომ **d ჰყოფს a-ს** (ან "a იყოფა d-ზე", ან "a ჯერადია d-სი" – d divides a), თუკი მოიძებნება ისეთი მთელი k ($k\in Z$), რომლისთვისაც a=kd. ეს ფაქტი მათემატიკურად ასე ჩაიწერება: $d|a$. 0 ნებისმიერი რიცხვის ჯერადია. ნებისმიერი რიცხვი საკუთარი თავის ჯერადია. თუ $a>0$ და $d|a$, მაშინ $|d|\le|a|$. ჩვეულებისამებრ, რიცხვის გამყოფებში გულისხმობენ მხოლოდ დადებით გამყოფებს.

თუ რიცხვი იყოფა მხოლოდ 1-ზე და საკუთარ თავზე, მაშინ მას **მარტივ რიცხვს** (prime) უწოდებენ, ხოლო თუ რიცხვი მეტია ერთზე და ის მარტივი არაა – **შედგენილ რიცხვს** (composite). 0 და 1 არ მიეკუთვნებიან არც მარტივ და არც შედგენილ რიცხვებს. მარტივ რიცხვთა სიმრავლე უსასრულოა: 2,3,5,7,11,13,17,19,23,29,31,37,…

ნებისმიერი მთელი n რიცხვისათვის მთელ რიცხვთა სიმრავლე იყოფა n კლასად n-ზე გაყოფით მიღებული ნაშთების მიხედვით: kn, kn+1,… . ადგილი აქვს თეორემას:

თეორემა 5.1 (ნაშთიანი გაყოფის შესახებ). ნებისმიერი მთელი a რიცხვისა და მთელი დადებითი n რიცხვისათვის არსებობს მთელ რიცხვთა ერთადერთი წყვილი q და r, რომელთათვისაც $0\le r<n$ და a=qn+r.

ამ შემთხვევაში $q=\lfloor a/n\rfloor$-ს უწოდებენ **მთელ** (quotent) ნაწილს, ხოლო r=a mod n-ს – **ნაშთს** (remainder, residue). ცხადია, რომ $n|a$ მაშინ და მხოლოდ მაშინ, როცა a mod n=0. ასევე ცხადია, რომ $a=\lfloor a/n\rfloor n+(a \bmod n)$ და $a \bmod n=a-\lfloor a/n\rfloor n$.

იტყვიან, რომ **a სადარია b-სი მოდულით n** (a is equivalent to b, modulo n – ჩაიწერება $a\equiv b \pmod n$), თუ $(a \bmod n)=(b \bmod n)$.

მოცემული n რიცხვისათვის მთელ რიცხვთა სიმრავლე იყოფა n-ის მოდულით **ექვივალენტობის n ცალ კლასად** (eqivalence classes modulo n). რიცხვი a შედის კლასში $[a]_n=\{a+kn: k\in Z\}$.

a და b რიცხვების ყველა **საერთო გამყოფს** (common divisors) შორის შეგვიძლია ამოვარჩიოთ **უდიდესი საერთო გამყოფი** (greatest common divisor), რომელსაც აღნიშნავენ ასე gcd(a,b) ან ÖÓÂ(a,b). ის განსაზღვრულია, თუ a და b რიცხვებისაგან ერთ-ერთი განსხვავებულია 0-საგან. პირობითად, შეგვიძლია ჩავთვალოთ, რომ ÖÓÂ(0,0)=0.

საერთო გამყოფებისა და უდიდესი საერთო გამყოფის თვისებები:

თუ $d|a$ და $d|b$, მაშინ $d|(a+b)$ და $d|(a-b)$;

თუ $d|a$ და $d|b$, მაშინ $d|(ax+by)$ ნებისმიერი მთელი x და y-სათვის;

თუ $b|a$ და $a|b$, მაშინ $a=\pm b$;

ÖÓÂ(a,b)=ÖÓÂ(b,a);

ÖÓÂ(a,b)=ÖÓÂ(-a,b);

ÖÓÂ(a,b)=ÖÓÂ(|b|,|a|)

ÖÓÂ(a,0)=|a|

ÖÓÂ(a,ka)=|a|, ნებისმიერი მთელი k-სათვის.

თეორემა 5.2. თუ a და b მთელი რიცხვები ერთდროულად არ უდრიან 0-ს, მაშინ მათი უდიდესი გამყოფი წარმოადგენს უმცირეს დადებით ელემენტს a და b რიცხვების წრფივ კომბინაციათა სიმრავლეში $\{ax+by: x,y\in Z\}$.

შედეგი 1. ორი მთელი რიცხვის უდიდესი საერთო გამყოფი მათი ნებისმიერი საერთო გამყოფის ჯერადია.

შედეგი 2. ნებისმიერი მთელი a და b რიცხვებისათვის და არაუარყოფითი n-სათვის სრულდება ტოლობა ÖÓÂ(an,bn)=n·ÖÓÂ(a,b).

შედეგი 3. ნებისმიერი მთელი დადებითი n, a და b-სათვის თუ n|ab და ÖÓÂ(a,n)=1, მაშინ n|b.

a და b რიცხვებს უწოდებენ **ურთიერთმარტივებს** (are relatively prime), თუ ÖÓÂ(a,b)=1.

თეორემა 5.3. თუ a, b და p მთელი რიცხვებისათვის ÖÓÂ(a,p)=1 და ÖÓÂ(b,p)=1, მაშინ ÖÓÂ(ab,p)=1.

$n_1, n_2, \ldots, n_k$ მთელ რიცხვებს უწოდებენ **წყვილ-წყვილად ურთიერთმარტივს** (parwise relatively prime), თუ ნებისმიერი ორი მათგანი ურთიერთმარტივია.

თეორემა 5.4. თუ მარტივი რიცხვი p ჰყოფს a და b მთელი რიცხვების ნამრავლს, მაშინ p|a ან p|b.

თეორემა 5.5. (დაშლის არსებობისა და ერთადერთობის თეორემა). ნებისმიერი შედგენილი a რიცხვი წარმოდგება ერთადერთი გზით შემდეგნაირად: $a=p_1^{e1}p_2^{e2}\cdot\ldots\cdot p_r^{er}$, სადაც $p_1<p_2<\ldots<p_r$ მარტივი რიცხვებია, ხოლო e_i – დადებითი მთელი რიცხვები.

თეორემა 5.6 (ÖÓÂ-ის რეკურენტული ფორმულა). თუ a მთელი არაუარყოფითი რიცხვია, ხოლო b – მთელი დადებითი, მაშინ ÖÓÂ(a,b)=ÖÓÂ(b, a mod b).

5.2. ÄÅÊËÉÃÄÓ ÀËÂÏÒÉÈÉ.

უდიდესი საერთო გამყოფის პოვნის ქვემოთ მოყვანილი ალგორითმი აღწერილია ევკლიდეს "საწყისებში" (300 წელი ჩვენს წელთაღრიცხვამდე), თუმცა შესაძლოა იგი უფრო ადრეც იყო ცნობილი. რეკურსიული პროცედურის შემავალი მონაცემებია არაუარყოფითი a და b რიცხვები.

```
EUCLID(a,b)
1 if b=0 {
2   then return a
3   else return EUCLID(b, a mod b)  }
```

მაგალითად EUCLID(30,21)=EUCLID(21,9)=EUCLID(9,3)=EUCLID(3,0)=0. შეგვიძლია ჩავთვალოთ, რომ შემავალი a და b რიცხვებისათვის a>b≥0, რადგან თუ b>a≥0, მაშინ პროცედურა EUCLID(a,b) პირველი გამოძახებისას უბრალოდ ადგილს შეუცვლის არგუმენტებს, ხოლო თუ a=b>0, მაშინ პროცედურა პირველივე გამოძახებით დაასრულებს მუშაობას, რადგან a mod a=0. პროცედურის მუშაობის ხანგრძლივობა დამოკიდებულია რეკურსიის სიღრმეზე და მის შესაფასებლად გამოდგება ფიბონაჩის რიცხვები (რეკურენტული ფორმულა $F_i=F_{i-1}+F_{i-2}$, სადაც $i\geq 2$, $F_0=0$, $F_1=1$).

ლემა. ვთქვათ a>b≥0. თუ პროცედურა EUCLID(a,b) გამოიძახებს საკუთარ თავს k-ჯერ (k≥1), მაშინ $a\geq F_{k+2}$ და $b\geq F_{k+1}$.

თეორემა 5.7 (ლამეს თეორემა). ვთქვათ k მთელი დადებითი რიცხვია. თუ a>b≥0 და $b<F_{k+1}$, მაშინ პროცედურა EUCLID(a,b) ასრულებს k-ზე ნაკლებ რეკურსიულ გამოძახებას.

ევკლიდეს გაფართოებული ალგორითმი. ევკლიდეს ალგორითმის მცირეოდენი შევსებით შესაძლებელი ხდება x და y კოეფიციენტების მიღება, რომელთათვისაც d=ÖÓÂ(a,b)=ax+by

```
EXTENDED-EUCLID(a,b)
1 if b=0
2   then {  return (a,1,0)  }
3 (d′,x′,y′)=EXTENDED-EUCLID(b, a mod b)
4 (d,x,y)=(d′,y′,x′-⌊a/b⌋y′)
5 return (d,x,y)
```

პროცედურის მუშაობის შედეგები ნაჩვენებია ცხრილში:

a	b	⌊a/b⌋	d	x	y
99	78	1	3	-11	14
78	21	3	3	3	-11
21	15	1	3	-2	3
15	6	2	3	1	-2
6	3	2	3	0	1
3	0	–	3	1	0

ცხრილი 5.1.

5.3. ÌÏÃÖËÀÖÖËÉ ÀÒÉÈÌÄÔÉÊÀ

არითმეტიკული ოპერაციები – შეკრება, გამოკლება, გამრავლება n-ის მოდულით 0,1,2,…, n-1 რიცხვებზე სრულდება ასე: თუ ოპერაციის შედეგი გადის მითითებული ინტერვალიდან, მაშინ ის შეიცვლება n-ზე გაყოფის ნაშთით. გაყოფის შემთხვევაში საქმე უფრო რთულადაა და დაგვჭირდება ზოგიერთი ცნება ჯგუფთა თეორიიდან.

S სიმრავლეს მასზე განსაზღვრული ბინარული ⊕ ოპერაციით ეწოდება **ჯგუფი** (group), თუ სრულდება შემდეგი თვისებები: ა) **ჩაკეტილობა** (closure): a⊕b∈S ნებისმიერი a,b∈S-თვის; ბ) **ნეიტრალური ელემენტის არსებობა** (identity): არსებობს e∈S ელემეტი, რომლისთვისაც e⊕a=a⊕e=a ნებისმიერი a∈S-თვის (ბუნებრივია, ასეთი ელემენტი ერთადერთია); გ) **ასოციატიურობა** (associativity): a⊕(b⊕c)=(a⊕b)⊕c ნებისმიერი a,b,c∈S-თვის დ) **შებრუნებული ელემენტის არსებობა** (inverse): ნებისმიერი a∈S-თვის მოიძებნება ერთადერთი ელემენტი b∈S, რომლისთვისაც a⊕b=b⊕a=e.

მაგალითად, მთელი რიცხვები შეკრების ოპერაციით ჰქმნიან ჯგუფს: ნეიტრალური ელემენტი 0-ია, ხოლო ნებისმიერი a რიცხვის შებრუნებული რიცხვია -a. ჯგუფს ეწოდება **აბელის** (abelian) ჯგუფი, თუ სრულდება კომუტატიურობის თვისება a⊕b=b⊕a ნებისმიერი a,b∈S-თვის. ჯგუფს ეწოდება **სასრული** (finite), თუ მასში ელემენტების რაოდენობა სასრულია.

შეკრება n-ის მოდულით და გამრავლება n-ის მოდულით სრულდება შემდეგნაირად:

$$[a]_n +_n [b]_n=[a+b]_n, \quad [a]_n \cdot_n [b]_n=[ab]_n$$

სხვაობათა ადიციური ჯგუფი მოდულით n (additive group modulo n) შეიცავს სხვაობებს $[0]_n$, $[1]_n,\ldots,[n-1]_n$. ისინი იკრიბებიან ზემოთ მოყვანილი წესით და აღინიშნებიან $(Z_n, +_n)$.

თეორემა 5.8. სისტემა $(Z_n, +_n)$ წარმოადგენს სასრულ აბელის ჯგუფს.

სხვაობათა მულტიპლიკაციური ჯგუფი მოდულით n (multiplicative group modulo n) აღინიშნება Z_n^*. მისი ნებისმიერი ელემენტი n-ის ურთიერთმარტივია.

თეორემა 5.9. სისტემა $(Z_n^*, \cdot_n)$ წარმოადგენს სასრულ აბელის ჯგუფს.

ელემენტთა რიცხვი Z_n^*-ში აღინიშნება φ(n)-ით. φ ფუნქციას უწოდებენ **ეილერის φ ფუნქციას**, რომლისთვისაც მტკიცდება შემდეგი ფორმულა:

$$\varphi(n) = n\left(1-\frac{1}{p_1}\right)\cdot\ldots\cdot\left(1-\frac{1}{p_s}\right)$$

სადაც $p_1,\ldots, p_s$ – n-ის ყველა მარტივი გამყოფია.

ვთქვათ, (S,⊕) ჯგუფია, ხოლო S′⊆S. თუ (S′,⊕) ასევე ჯგუფია, მაშინ (S′,⊕)-ს უწოდებენ (S,⊕)-ის **ქვეჯგუფს** (subgroup). მაგალითად, ლუწი რიცხვები შეკრების ოპერაციით წარმოადგენენ მთელი რიცხვების ჯგუფის ქვეჯგუფს.

თეორემა 5.10. თუ (S,⊕) სასრული ჯგუფია, S′ არის S-ის არაცარიელი ქვესიმრავლე და a⊕b∈S′ ნებისმიერი a,b∈S′-თვის, მაშინ (S′,⊕) წარმოადგენს (S,⊕)-ის ქვეჯგუფს.

თეორემა 5.11 (ლაგრანჟი). თუ (S′,⊕) არის (S,⊕) სასრული ჯგუფის ქვეჯგუფი, მაშინ |S′| ჰყოფს |S|-ს.

თუ S-ის S′ ქვეჯგუფი არ ემთხვევა მთელ ჯგუფს, მაშინ S′-ს S-ის **საკუთარი** (proper) ქვეჯგუფი ეწოდება.

შედეგი. თუ (S′,⊕) არის (S,⊕) სასრული ჯგუფის საკუთარი ქვეჯგუფი, მაშინ $|S'|\le|S|/2$.

ვთქვათ a წარმოადგენს S სასრული ჯგუფის ელემენტს. განვიხილოთ ელემენტთა ასეთი მიმდევრობა:

$$e, a, a\oplus a, a\oplus a\oplus a, \ldots$$

ხარისხების ანალოგიით დავწეროთ $a^{(0)}=e$, $a^{(1)}=a$, $a^{(2)}=a\oplus a$, $a^{(3)}=a\oplus a\oplus a$ და ა.შ. ცხადია, რომ $a^{(i)}\oplus a^{(j)}=a^{(l+j)}$ და კერძოდ $a^{(i)}\oplus a=a^{(l+1)}$. ასეთივე თანაფარდობა შეიძლება დაიწეროს "უარყოფითი ხარისხებისათვის": $a^{(i)}\oplus a^{-1}=a^{(l-1)}$.

თუ S ჯგუფი სასრულია, მაშინ e, a, a⊕a, a⊕a⊕a, … მიმდევრობა პერიოდული იქნება. ამასთან, წინაპერიოდი არ იარსებებს. მაშასადამე, მიმდევრობას ექნება სახე $e=a^{(0)}, a^{(1)}, a^{(2)}, \ldots, a^{(t-1)}, a^{(t)}=e, \ldots$ და შეიცავს t განსხვავებულ ელემენტს, სადაც t უმცირესი დადებითი რიცხვია, რომლისთვისაც $a^{(t)}=e$. ამ რიცხვს ეწოდება a ელემენტის რიგი და აღინიშნება ord(a). აღნიშნული t ელემენტი ჰქმნის ქვეჯგუფს. ამ ქვეჯგუფს უწოდებენ **a ელემენტის მიერ წარმოშობილს** (subgroup generated by a) და აღნიშნავენ – <a>. a-ს უწოდებენ ქვეჯგუფის **წარმომქმნელს** (generator). მაგალითად, სხვადასხვა ელემენტების მიერ წარმოქმნილი ქვეჯგუფები Z_6-ში: <0>={0}, <1>={0,1,2,3,4,5}, <2>={0,2,4}.

თეორემა 5.12. ვთქვათ (S,⊕) სასრული ჯგუფია. თუ a∈S, მაშინ a ელემენტის მიერ წარმოქმნილი ქვეჯგუფის ელემენტთა რაოდენობა ემთხვევა a-ს რიგს, ანუ |<a>|=ord(a).

შედეგი. მიმდევრობას $a^{(1)}$, $a^{(2)}$, … აქვს პერიოდი t=ord(a), ანუ $a^{(i)}=a^{(j)}$, მაშინ და მხოლოდ მაშინ, როცა $i\equiv j \pmod t$.

შედეგი. e ერთეულის მქონე (S,⊕) სასრულ ჯგუფში ნებისმიერი a∈S-სათვის სრულდება ტოლობა $a^{(|S|)}=e$.

5.4. წრფივი დიოფანტური განტოლებების ამოხსნა.

წრფივ დიოფანტურ განტოლებებს აქვთ სახე $ax \equiv b \pmod n$, სადაც a, b და n მთელი რიცხვებია. ცხადია, რომ განტოლების ამოსახსნელად აქ მთავარია მხოლოდ x-ის n-ზე განაყოფის ნაშთი. ამიტომ ამონახსნი წარმოადგენს Z_n ჯგუფის ელემენტს, ანუ იმ რიცხვთა კლასს, რომლებიც იძლევიან ერთნაირ ნაშთს n-ზე გაყოფისას. მაშასადამე, ამოცანა შეიძლება ასეც ჩამოყალიბდეს: გვაქვს ელემენტები $a,b \in Z_n$, ვეძებთ ყველა $x \in Z_n$, რომელთათვისაც $ax \equiv b \pmod n$.

განსაზღვრების თანახმად $\langle a \rangle = \{a^{(x)} : x>0\} = \{ax \bmod n : x>0\}$, ამიტომ განტოლებას ერთი მაინც ამონახსნი აქვს მაშინ და მხოლოდ მაშინ, როცა $b \in \langle a \rangle$. ლაგრანჟის თეორემის მიხედვით, $\langle a \rangle$-ში ელემენტების რაოდენობა წარმოადგენს n-ის გამყოფს.

თეორემა 5.13. ნებისმიერი მთელი დადებითი a და n-ისათვის $\langle a \rangle = \langle d \rangle = \{0.d.2d,\ldots,((n/d)-1)d\}$ და $|\langle a \rangle| = n/d$, სადაც $d = ÖÓÂ(a,n)$.

შედეგი. განტოლება $ax \equiv b \pmod n$ ამოიხსნება x-ის მიმართ მაშინ და მხოლოდ მაშინ, როცა $ÖÓÂ(a,n) | b$.

შედეგი. განტოლებას $ax \equiv b \pmod n$ აქვს $d = ÖÓÂ(a,n)$ განსხვავებული ამონახსნი Z_n-ში ან ამონახსნი საერთოდ არა აქვს.

თეორემა 5.14. ვთქვათ $d = ÖÓÂ(a,n) = ax' + ny'$, სადაც x' და y' მთელი რიცხვებია. თუ $d | b$, მაშინ რიცხვი $x_0 = x'(b/d) \bmod n$ არის $ax \equiv b \pmod n$ განტოლების ამონახსნი.

თეორემა 5.15. ვთქვათ $ax \equiv b \pmod n$ განტოლება ამოხსნადია და x_0 მისი ამონახსნია. მაშინ განტოლებას აქვს $d = ÖÓÂ(a,n)$ ცალი ამონახსნი Z_n-ში, რომლებიც მოიცემიან ფორმულით $x_i = x_0 + i(n/d)$, სადაც $I = 0,1,2,\ldots,n-1$.

ზემოთ თქმულიდან გამომდინარე შეგვიძლია დავწეროთ პროცედურა, რომელიც მთელი a, b და $n>0$ რიცხვებისათვის იძლევა $ax \equiv b \pmod n$ განტოლების ყველა ამონახსნს.

```
MODULAR-LINEAR-EQUATION-SOLVER(a,b,n)
1 (d, x', y')=EXTENDED-EUCLID(a,n)
2 IF d|b then {
3                 x0=x'(b/d) mod n
4         FOR I=0 to d-1 {
5                         PRINT (x0+I(n/d)) mod n     }
6         else PRINT "ამონახსნი არაა"       }
```

მაგალითად ავიღოთ განტოლება $14x \equiv 30 \pmod{100}$. აქ $a=14$, $b=30$, $n=100$. პირველ სტრიქონში EXTENDED-EUCLID პროცედურის გამოძახება მოგვცემს $(d,x,y)=(2,-7,1)$. რადგან $2|30$, მესამე სტრიქონში გამოითვლება $x_0 = (-7)(15) \bmod 100 = 95$ და 4-5 სტრიქონებში დაიბეჭდება 95 და 45.

პროცედურა MODULAR-LINEAR-EQUATION-SOLVER(a,b,n) ასრულებს $O(\log n)$ არითმეტიკულ ოპერაციას პირველ სტრიქონში და $O(ÖÓÂ(a,n))$ ოპერაციას – დანარჩენებში, მთლიანი დრო – $O(\log n + ÖÓÂ(a,n))$.

შედეგი. ვთქვათ $n>1$. თუ $ÖÓÂ(a,n)=1$, მაშინ $ax \equiv b \pmod n$ განტოლებას აქვს ერთადერთი ამონახსნი Z_n-ში.

$b=1$ შემთხვევაში x ელემენტის შებრუნებული მოდულით n, ე.ი. შებრუნებული Z_n^* ჯგუფში.

შედეგი. ვთქვათ $n>1$. თუ $ÖÓÂ(a,n)=1$, მაშინ $ax \equiv 1 \pmod n$ განტოლებას აქვს ერთადერთი ამონახსნი Z_n-ში. თუ $ÖÓÂ(a,n)>1$, მაშინ ამ განტოლებას ამონახსნი არა აქვს.

თეორემა 5.15 (ჩინური თეორემა ნაშთების შესახებ). ვთქვათ $n = n_1 \cdot n_2 \cdot \ldots \cdot n_k$, სადაც $n_1, n_2, \ldots, n_k$ წყვილ-წყვილად ურთიერთმარტივია. განვიხილოთ თანაფარდობა $a \leftrightarrow (a_1, a_2, \ldots, a_k)$, სადაც $a \in Z_n$, $a_i \in Z_n$ და $a_i = a \bmod n_I$ $(i=1,2,\ldots,k)$. ფორმულა $a \leftrightarrow (a_1, a_2, \ldots, a_k)$ განსაზღვრავს ურთიერთცალსახა დამოკიდებულებას Z_n-სა და $Z_{n1} \times Z_{n2}, \times \ldots \times Z_{nk}$ დეკარტულ ნამრავლს შორის. ამასთან, შეკრების, გამოკლების და გამრავლების ოპერაციებს Z_n-ში შეესაბამება ოპერაციები თითოეული კომპონენტის მიხედვით k-ელემენტიან კორტეჟებში, ანუ

თუ $a \leftrightarrow (a_1, a_2, \ldots, a_k)$ და $b \leftrightarrow (b_1, b_2, \ldots, b_k)$, მაშინ

$$(a+b) \bmod n \leftrightarrow ((a_1+b_1) \bmod n_1, \ldots, (a_k+b_k) \bmod n_k)$$

$$(a-b) \bmod n \leftrightarrow ((a_1-b_1) \bmod n_1, \ldots, (a_k-b_k) \bmod n_k)$$

$$(ab) \bmod n \leftrightarrow ((a_1 b_1) \bmod n_1, \ldots, (a_k b_k) \bmod n_k)$$

ხარისხის გამოთვლა კვადრატში ხელახალი აყვანით. მოდულით ხარისხში აყვანა მნიშვნელოვან როლს თამაშობს რიცხვის მარტივობის შემოწმებისას. ხარისხში ასაყვანად ხელახალი გადამრავლება არაა ყველაზე სწრაფი მეთოდი, უმჯობესია გამოვიყენოთ **კვადრატში ხელახალი აყვანის** (repeated squarinq) ალგორითმი.

ვთქვათ უნდა გამოვთვალოთ $a^b \bmod n$, სადაც a არის სხვაობა n-ის მოდულით, ხოლო b მთელი არაუარყოფითი რიცხვია, რომლის ორობითი ჩანაწერია – $(b_k, b_{k-1}, \ldots, b_1, b_0)$. გამოვითვლით $a^c \bmod n$ რომელიღაც c-სათვის, რომელიც შემდეგ გაიზრდება და გახდება b.

```
MODULAR_EXPONENTIATION(a,b,n)
1 c=0
2 d=1
3 (b_k, b_{k-1}, … , b_1, b_0) – b-ს ორობითი ჩანაწერია
4 FOR i=k DOWNTO 0    {
5     c=2c
6     d=(d*d) mod n
7     IF b_i=1 then {
8           c=c+1
9         d=(d*a) mod n   }      }
10 return d
```

მაგალითად, თუ a=7, b=560 და n=561, პროცედურა ასე იმუშავებს:

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

ცხრილი 5.2

პროცედურის მუშაობის დრო დამოკიდებულია შემავალ მონაცემებში ბიტების რაოდენობაზე. თუ შემავალ სამ მონაცემში ბიტების რაოდენობა არ აღემატება β-ს, მაშინ არითმეტიკული ოპერაციების რიცხვია $O(\beta)$, ხოლო ბიტური ოპერაციების $O(\beta^3)$.

5.5. რიცხვთა მარტივობის შემოწმება

შემოვიღოთ **მარტივი რიცხვების განაწილების ფუნქცია** (prime distribution function) $\pi(n)$. მისი მნიშვნელობა ტოლია იმ მარტივი რიცხვების რაოდენობისა, რომლებიც n-ს არ აღემატებიან. მაგალითად $\pi(10)=4$ (2, 3, 5, 7).

თეორემა (მარტივი რიცხვების განაწილების ასიმპტოტური კანონი).

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1$$

შეფარდება $n/\ln n$ მცირე n-ისათვისაც საკმაოდ კარგ მიახლოებას იძლევა. მაგალითად, $n=10^9$-თვის ცდომილება 6%-ს არ აღემატება: $\pi(10^9)=50\ 847\ 534$, ხოლო $10^9/\ln 10^9 \approx 48\ 254\ 942$.

ასიმპტოტური კანონი გვაძლევს იმ ალბათობის შეფასების საშუალებას, რომლითაც 1..n მონაკვეთზე შემთხვევითად არჩეული რიცხვი შესაძლოა იყოს მარტივი – $1/\ln n$. ეს ნიშნავს, რომ შემოწმებული უნდა იქნას შემთხვევითად აღებული $\ln n$ რიცხვი. მაგალითად, თუ საჭიროა ასნიშნა მარტივი რიცხვის პოვნა, უნდა შემოწმდეს დაახლოებით $\ln 10^{100} \approx 230$ შემთხვევითი რიცხვი 1-დან 10^{100}-მდე. ეს შეფასება შესაძლოა ორჯერ შევამციროთ, თუ მხედველობაში მივიღებთ, რომ მხოლოდ კენტი რიცხვების შემოწმებაა მართებული. აქვე შევნიშნავთ, რომ მითითებულ შუალედში შემთხვევითად აღებული რიცხვი დიდი ალბათობით 100-ნიშნა იქნება, რადგანაც ამ შუალედში 100-ნიშნა რიცხვების რაოდენობა 90%-ია, ხოლო 99 და მეტი ნიშნის მქონე – 99%.

ზემოთ თქმულიდან გამომდინარე, გასარკვევი რჩება მხოლოდ ის, თუ როგორ უნდა შევამოწმოთ მიღებული ასნიშნა რიცხვის მარტივობა. n-ის მარტივობის შესამოწმებლად ყველაზე უბრალო გზა არის მისი 2-დან $\sqrt{n}$-მდე ყველა **მარტივ რიცხვზე გაყოფადობის** (trial division) შემოწმება, მაგრამ ამისათვის საჭიროა $O(\sqrt{n})$ დრო, ე.ი. ეს მეთოდი მხოლოდ მცირე n-ებისთვის არის ვარგისი ან თუ წინასწარ ცნობილია, რომ გამყოფი წარმოადგენს მცირე რიცხვს. თუმცა ამ მეთოდს დადებითი მხარეც აქვს - n-ის არამარტივობის შემთხვევაში ნაპოვნი იქნება მისი გამყოფიც. რიგ მეთოდებში რიცხვის არამარტივობის დადგენა ხდება მისი გამყოფების პოვნის გარეშე.

<u>**ფსევდომარტივი რიცხვები**</u>. აღვნიშნოთ Z_n^+-ით n-ის მოდულით ყველა არანულოვანი ნაშთის სიმრავლე:

$$Z_n^+ = \{1,2,\ldots,n-1\}$$

თუ n მარტივია, მაშინ $Z_n^+ = Z_n^*$

რიცხვ n-ს ვუწოდოთ **ფსევდომარტივი a-ს ფუძით** (base-a pseudoprime), თუ სრულდება ფერმას მცირე თეორემა: $a^{n-1} \equiv 1 \pmod n$.

ნებისმიერი მარტივი რიცხვი ფსევდომარტივია ნებისმიერი $a \in Z_n^+$, ამიტომ თუკი ჩვენ მოვძებნეთ რომელიმე a ფუძე, რომლისთვისაც n არაა ფსევდომარტივი, შეგვიძლია ჩავთვალოთ, რომ n შედგენილია. როგორც ირკვევა, უმრავლეს შემთხვევაში საკმარისია შევამოწმოთ ფუძე a=2.

```
Pseudoprime(n)
1 IF MODULAR-EXPONENTIATION(2,n-1,n) ≠1 (mod n) THEN
2    return COMPOSITE
3    else PRIME
```

თუკი პროცედურა გამოიტანს COMPOSITE-ს, მაშინ n უსათუოდ შედგენილი რიცხვია, ხოლო PRIME-ს შემთხვევაში მარტივობა ჯერ კიდევ დასაზუსტებელია. თუმცა ცნობილია, რომ 1-დან 10000-მდე მხოლოდ 22 რიცხვია ისეთი, რომლისთვისაც ეს პროცედურა არასწორ პასუხს იძლევა (მაგ. 341, 561, 645, 1105). n რიცხვის ზრდასთან ერთად ასეთი რიცხვების პროცენტული რაოდენობა 0-საკენ მიისწრაფის. გამოკვლეულია, რომ 50-ნიშნა 2-ის ფუძით ფსევდომარტივ რიცხვთა შორის შედგენილი რიცხვები არ აღემატებიან 10^{-6} ნაწილს, ხოლო 100-ნიშნა რიცხვებში – 10^{-13}-ს. რა თქმა უნდა, ფერმას მცირე თეორემაში 2-ის გარდა სხვა ფუძეების შემოწმებაც გავაგრძელოთ, მაგრამ არსებობენ შედგენილი რიცხვები, რომლებიც ფსევდომარტივები არიან ნებისმიერი ფუძით. ასეთ რიცხვებს **კარმაიკლის რიცხვებს** (Carmichael numbers) უწოდებენ. კარმაიკლის რიცხვებიდან პირველი სამი რიცხვია 561, 1105 და 1729.

n რიცხვის მარტივობის დასაზუსტებლად გამოიყენება მილერ-რაბინის ალბათური ტესტი, რომელიც ამოწმებს ფერმას მცირე თეორემას a-ს რამდენიმე მნიშვნელობისათვის და თუ გამოთვლის პროცესში იპოვა არატრივიალური ფესვი 1 მოდულით n-დან, წყვეტს მუშაობას და იძლევა შეტყობინებას, რომ n შედგენილი რიცხვია. პროცედურა miller-rabin-ის საწყისი მონაცემებია n>2 – რიცხვი, რომლის მარტივობაც შესამოწმებელია და s – იტერაციათა რაოდენობა (რაც მეტია s, მით ნაკლებია შეცდომის შესაძლებლობა). პროცედურა miller-rabin იყენებს პროცედურას Witness(a,n), რომელიც ჭეშმარიტია მაშინ, როცა a წარმოადგენს n-ის არამარტივობის "მოწმეს".

```
WITNESS(a,n)
1 ვთქვათ (b_k, b_{k-1},…,b_0) n-1-ის ორობითი ჩანაწერია
2 d=1
3 FOR i=k downto 0 {
4       x=d
5       d=(d·d) mod n
6       IF d=1 and x≠1 and x≠n-1 then
7          return true
8       IF b_i=1 then
9          d=(d·a) mod n }
10 IF d≠1 then {
11    return true  }
12 return false
```

პროცედურა Witness(a,n) ღებულობს მნიშვნელობას true, თუკი მოიძებნება არატრივიალური ფესვი 1 მოდულით n-დან (სტრიქონი 6-7) ან თუ არ სრულდება ფერმას მცირე თეორემა (სტრიქონი 10-11). ორივე შემთხვევაში n შედგენილია. ახლა შეგვიძლია დავწეროთ მილერ-რაბინის ალბათური ალგორითმი:

```
MILLER-RABIN(n,s)
1 for I=1 to s {
2    a=random(1,n-1)
3       if WITNESS(a,n)=true then
4          return composite }
5 return prime
```

მილერ-რაბინის ტესტის შეცდომის ალბათობის შეფასებას იძლევა შემდეგი ორი თეორემა:

თეორემა. ვთქვათ n კენტი შედგენილი რიცხვია. Z_n^+ ელემენტთა შორის არანაკლებ (n-1)/2-ისა წარმოადგენს n-ის არამარტივობის მოწმეს.

თეორემა. ნებისმიერი კენტი n>2 და მთელი დადებითი s-ისათვის MILLER-RABIN(n,s) პროცედურის შეცდომის ალბათობა არ აღემატება 2^{-s}.

5.1. მარტივი რიცხვები ინტერვალიდან

(საქართველოს სტუდენტთა გუნდური პირველობა, 2003 წელი)

შესატანი ფაილის სახელი: input.txt

გამოსატანი ფაილის სახელი: output.txt

დროითი შეზღუდვა : 10 წამი ყოველ ტესტზე

როგორც ცნობილია, ნატურალურ რიცხვს ეწოდება მარტივი, თუ ის ნაშთის გარეშე არ იყოფა არც ერთ ნატურალურ რიცხვზე, გარდა 1-ისა და თავისი თავისა. მიღებულია, რომ 1 არ ჩაითვალოს მარტივ რიცხვად.

ნატურალურ რიცხვთა მოცემული ინტერვალისათვის დაითვალეთ მასში მარტივი რიცხვების რაოდენობა.

შესატანი მონაცემების ფორმატი:

შესატანი ფაილის პირველ სტრიქონში მოცემულია ნატურალური რიცხვი Min – ინტერვალის ქვედა საზღვარი. შესატანი ფაილის მეორე სტრიქონში მოცემულია მეორე ნატურალური რიცხვი Max – ინტერვალის ზედა საზღვარი.

შეზღუდვა: $1 \le \text{Min} \le \text{Max} \le 1500000$.

გამოსატანი მონაცემების ფორმატი:

გამოსატანი ფაილის ერთადერთ სტრიქონში უნდა ჩაიწეროს მთელი რიცხვი – მარტივი რიცხვების რაოდენობა ნატურალურ რიცხვთა მოცემულ დიაპაზონში.

შენიშვნა: შესატან და გამოსატან ფაილებში არ დაიშვება ზედმეტი სიმბოლოები (მათ შორის, შუალედები).

ნიმუში:

input.txt	output.txt
1 7	4
2 5	3
4 8	2
100 200	21
1000 2000	135
140000 150000	838
55550 55555	0

მითითება. გარეგნულად ამოცანა რთული არ არის, მაგრამ შემომავალი მონაცემების დიაპაზონი იმდენად დიდია, რომ ალგორითმის შედგენისას მცირედღნი არაეკონომიურობაც კი დროით შეზღუდვას არღვევს. გამოთვლის პროცესში აუცილებელია მარტივ რიცხვთა მიმდევრობის დამახსოვრება, რათა ყოველი ახალი რიცხვის მარტივობის დადგენისას გაყოფადობა მხოლოდ მარტივ რიცხვებზე შემოწმდეს. ამასთან, გაყოფადობის შემოწმება უნდა გაგრძელდეს შესამოწმებელი რიცხვის კვადრატულ ფესვამდე (ბუნებრივია, იგულისხმება კვადრატული ფესვის მთელი ნაწილი).

5.2. წყურვილის მოკვლა

(ჩიგირინსკების ოლიმპიადა, 2003-04 წელი)

ძლიერი ქარიშხლის გამო საზღვაო ხომალდმა კატასტროფა განიცადა უკაცრიელი კუნძულის მახლობლად. ეკიპაჟის წევრებმა ცურვით მოახერხეს კუნძულამდე მიღწევა და გადაწყვიტეს გემის ნარჩენებისაგან მოზრდილი ნავის გაკეთება. მეზღვაურებს მუშაობაში ხელს უშლით ის გარემოება, რომ კუნძულზე არ მოიპოვება სასმელი წყალი. საბედნიეროდ, ტალღებმა ნაპირზე გამორიყეს რამდენიმე კასრი რომი სამზარეულო ჭურჭელთან ერთად, მაგრამ კაპიტანს კარგად ესმის, რომ თუ ეკიპაჟი "წყურვილს მოიკლავს" ისე, როგორც მას საერთოდ სჩვევია, ნავის აგებაზე საუბარიც კი ზედმეტი იქნება. ამის გამო კაპიტანმა მიიღო ასეთი გადაწყვეტილება: ყველა მეზღვაურს დაურიგდა რომით პირთამდე სავსე D მოცულობის მქონე კათხა (სამზარეულო ჭურჭელში აღმოჩნდა სულ 4 ტიპის კათხა, შესაბამისად A, B, C და D მოცულობის მქონე, რომელთა მოცულობები გამოისახება მთელი რიცხვებით და არ აღემატება $2{,}1 \cdot 10^9$ ერთეულს), რის შემდეგაც ეკიპაჟმა დაიწყო ნავის აგება, ხოლო კაპიტანი კი დარჩა კასრებთან ცარიელი კათხებით. მეზღვაური, რომელსაც კიდევ ერთხელ სურს "წყურვილის მოკვლა", უნდა მივიდეს კაპიტანთან, დროებით გვერდზე გადადოს თავისი ცარიელი D კათხა, ხოლო დანარჩენი სამით შეუძლია:

- ჩაასხას კათხაში კასრიდან (ჩასხმის შემთხვევაში კათხა პირთამდე უნდა გაივსოს);
- გადაასხას ერთი კათხიდან მეორეში;
- ჩაასხას კათხიდან უკან კასრში;
- შეავსოს კათხა კასრიდან.

თუკი ამ ოპერაციების საშუალებით მეზღვაური A, B და C კათხებიდან ერთ-ერთში მოახერხებს ზუსტად იმდენი რომის მიღებას, რამდენსაც იტევს D კათხა, მაშინ მას უფლება ეძლევა გადაასხას რომი თავის D კათხაში და დალიოს, წინააღმდეგ შემთხვევაში ის "წყურვილის მოუკვლელად" უნდა მიუბრუნდეს სამუშაოს.

დაწერეთ პროგრამა, რომელიც გაარკვევს, მოახერხებს თუ არა მეზღვაური "წყურვილის მოკვლას" (განმეორებით), ანუ მიიღებს თუ არა D ერთეულ რომს.

შემავალი მონაცემების ფორმატი(ფაილი pour.in): ერთადერთ სტრიქონში მოცემულია ჰარით გაყოფილი A, B, C და D რიცხვები. კასრებში რომის რაოდენობა შეუზღუდავია.

გამომავალი მონაცემების ფორმატი(ფაილი pour.out): ერთადერთ სტრიქონში გამოტანილი უნდა იქნას სიტყვა "YES", თუკი მეზღვაური მოახერხებს წყურვილის მოკვლას, წინააღმდეგ შემთხვევაში – "NO".

შემავალი მონაცემების მაგალითი :	გამომავალი მონაცემი ნაჩვენები მაგალითისათვის:
1 2 3 4	YES

*მითითება. ცხადია, რომ როგორაც არ უნდა გადავასხათ სითხე ჭურჭლიდან ჭურჭელში, თითოეულ მათგანში ყოველთვის მივიღებთ A, B და C რიცხვების წრფივ კომბინაციას i*A+j*B+k*C, სადაც i, j, k მთელი რიცხვებია, ამიტომ თუკი D ნაკლებია A, B და C რიცხვებიდან ერთ-ერთზე მაინც (ამ პირობის გარეშე ვერცერთ ჭურჭელში ვერ მივიღებთ D მოცულობის სითხეს) და ამასთან, ევკლიდეს ალგორითმის მიხედვით, D იყოფა ÓÓÂ(A,B,C)-ზე, მაშინ პასუხი დადებითია, ხოლო წინააღმდეგ შემთხვევაში – უარყოფითი.*

6. დინამიური პროგრამირება და ხარბი ალგორითმები

6.1. დინამიური პროგრამირება

განვიხილოთ შემდეგი ამოცანა:

6.1. მაქსიმალური ჯამის შემცველი მართკუთხედი

ვთქვათ, მოცემულია n×n ზომის A მასივი, სადაც $2 \le n \le 1000$ და $-100 \le A[i,j] \le 100$. ვიპოვოთ მართკუთხედი, რომლის ზედა მარცხენა წვეროს წარმოადგენს A[1,1] ელემენტი და რომელშიც შემავალი ელემენტების მნიშვნელობათა ჯამი მაქსიმალურია. გამომავალ მონაცემებს წარმოადგენენ მაქსიმალური ჯამის შემცველი მართკუთხედის ქვედა მარჯვენა წვერო და ამ მართკუთხედში შემავალი ელემენტების ჯამი.

ამოცანის ამოსახსნელად, ბუნებრივია, უნდა განვიხილოთ (ან შევაფასოთ) საწყის მასივში შემავალი ყველანაირი მართკუთხედი და თითოეული მათგანისათვის განსაზღვრულ უნდა იქნას მასში შემავალი ელემენტების ჯამი. რადგან საძებნი მართკუთხედის ზედა მარცხენა ბოლო დაფიქსირებულია, ხოლო ქვედა მარჯვენა ბოლო პოტენციურად შესაძლოა იყოს ნებისმიერი ელემენტი, თვით A[1,1]-ის ჩათვლით (მაგალითად, თუკი მასივში მის გარდა ყველა წევრი უარყოფითია), ცხადია, რომ მოგიწევს n^2 მართკუთხედის განხილვა. n=1000-სათვის საჭიროა განვიხილოთ ზუსტად მილიონი მართკუთხედი და თუკი ჩვენ თითოეული მათგანისათვის უბრალოდ ავჯამავთ ყველა ელემენტს, მოგვიწევს დაახლოებით 2.5*1011 ოპერაციის ჩატარება, რაც ძალიან დიდი რიცხვია თანამედროვე პერსონალური კომპიუტერებისათვის. ამავე დროს ცხადია, რომ ამ მილიონი მართკუთხედის ელემენტთა ჯამების დადგენისას ჩვენ ხშირად გვიწევს ერთი და იმავე რიცხვების შეკრება და თუ მოვახერხებთ მცირე ზომის მართკუთხედებისათვის მიღებული შედეგების გამოყენებას მათი შემცველი მარკუთხედების წევრთა ჯამის გამოთვლისას – ოპერაციების რაოდენობა მნიშვნელოვნად შემცირდება.

A მასივი

-3	7	-2	4	5
-2	9	2	6	-10
5	-4	-6	0	-1
-3	8	-8	3	7
0	5	-9	-7	12

ნახ. 6.1.

მაგალითისათვის განვიხილოთ 5×5 ზომის A მასივი. შემოვიღოთ იმავე ზომის B მასივი, რომლის თითოეული B[i,j] ელემენტი ტოლი იქნება A[1,1]-ით და A[i,j]-ით განსაზღვრული მართკუთხედის ელემენტების ჯამისა (ჩვენი მთავარი ამოცანა სწორედ ამ მასივის ეფექტური გზით შევსებაა). თავდაპირველად შევავსოთ B მასივის პირველი სტრიქონი მარცხნიდან მარჯვნივ და პირველი სვეტი ზემოდან ქვემოთ. ცხადია, რომ B[1,1]=A[1,1], ხოლო სხვა ელემენტებისათვის მართებულია ფორმულა B[1,j]= B[1,j-1]+A[1,j] – პირველ სტრიქონში და B[i,1]= B[i-1,1]+A[i,1] – პირველ სვეტში.

B მასივი პირველი სტრიქონისა და პირველი სვეტის შევსების შემდეგ (nil – შეუვსებელი წევრი).

-3	4	2	6	11
-5	nil	nil	nil	nil
0	nil	nil	nil	nil
-3	nil	nil	nil	nil
-1	nil	nil	nil	nil

ნახ. 6.2.

დანარჩენი ელემენტები შევავსოთ შემდეგნაირად: ვიმოძრაოთ მარცხნიდან მარჯვნივ და ზემოდან ქვემოთ და B მასივის ყოველ უჯრედში ჩავწეროთ ამ უჯრედის მარცხნივ და მის ზემოთ მყოფი ელემენტებისა და A მასივის შესაბამისი ელემენტის ჯამს გამოკლებული B მასივში დიაგონალურად ზედა მარცხენა მეზობელ უჯრედში მოთავსებული ელემენტის მნიშვნელობა:

B[i,j]=A[i,j]+B[i-1,j]+b[i,j-1]-B[i-1,j-1]

ამ ფორმულის გეომეტრიული შინაარსი გამოსახულია ნახ. 6.3-ზე:

B[1,1]				
	B[i-1,j-1]	B[i-1,j]		
	B[i,j-1]	B[I,j]		
...				

ნახ. 6.3.

თუკი B[I,j] წევრის მნიშვნელობის მისაღებად A[i,j] ელემენტს დავუმატოთ {A[1,1],A[i-1,j]} და {A[1,1],A[i,j-1]} წყვილებით განსაზღვრული მართკუთხედების ელემენტები, რომელთა ჯამებიც ჩაწერილია შესაბამისად B[i-1,j]-სა და B[i,j-1]-ში. ცხადია, რომ ამ ჯამში ორჯერ შედის {A[1,1],A[i-1,j-1]} წყვილით განსაზღვრული ელემენტების ჯამი, რომელიც ჩაწერილია B[i-1,j-1] ელემენტში, ამიტომ საჭიროა მისი მნიშვნელობა ერთხელ გამოვაკლოთ. საბოლოოდ მივიღებთ:

B მასივის საბოლოო სახე

-3	4	2	6	11
-5	11	11	21	16
0	12	6	16	10
-3	17	3	16	17
-3	22	-1	7	20

ნახ. 6.4.

პასუხის მისაღებად საჭიროა ვიპოვოთ B მასივის მაქსიმალური ელემენტი და ამ ელემენტის ინდექსები. მაგალითში განხილული A მასივისთვის ყველაზე დიდი ჯამი მიიღება {A[1,1],A[5,2]} წყვილით განსაზღვრულ მართკუთხედში და მისი მნიშვნელობაა 22.

მოვიყვანოთ ზემოთ აღწერილი ალგორითმის პროგრამული კოდი:

```
READ (n,A)
B[1,1]=A[1,1];     max=-MAXINT;  imax=0; jmax=0
FOR i=2 to n { B[1,j]= B[1,j-1]+A[1,j]    }
FOR i=2 to n { B[i,1]= B[i-1,1]+A[i,1]    }
FOR i=2 to n {
        FOR j=2 to n { B[i,j]=A[i,j]+B[i-1,j]+b[i,j-1]-B[i-1,j-1]  }     }
FOR i=1 to n {
        FOR j=1 to n {
        IF B[i,j]>max then { max=B[i,j]; imax=i; jmax=j   }   }     }
WRITE (imax, jmax, max)
```

ამ ალგორითმით B მასივის თითოეული ელემენტის შევსებას ოთხ ოპერაციაზე მეტი არ სჭირდება. ყოველი ელემენტისათვის კიდევ თითო ოპერაცია ესაჭიროება მაქსიმალური ელემენტისა და მისი ინდექსების პოვნას. აქედან შეგვიძლია დავასკვნათ, რომ ეს ალგორითმი დაახლოებით 10000-ჯერ უფრო სწრაფად მუშაობს, ვიდრე თავდაპირველად ნახსენები

ალგორითმი, სადაც B მასივის თითოეული ელემენტის მისაღებად A მასივის შესაბამისი ელემენტები თავიდან უნდა აგვეჯამა.

ამოცანების ამოხსნის ასეთ მეთოდს, როდესაც ყოველ ბიჯზე თვლის დროს მიღებული საშუალედო შედეგების დამახსოვრება ხდება სპეციალურ ცხრილში და ეს შედეგები გამოიყენება მომდევნო ბიჯების გამოთვლისას, უწოდებენ **დინამიური პროგრამირების** მეთოდს.

დინამიური პროგრამირების ზოგადი პრინციპები პირველად აღწერა ამერიკელმა მათემატიკოსმა რიჩარდ ბელმანმა მეოცე საუკუნის 50-იან წლებში და ეს მეთოდი დღეისათვის წარმოადგენს ერთ-ერთ ყველაზე მძლავრ საშუალებას სხვადასხვა სახის ამოცანების ამოსახსნელად.

დინამიური პროგრამირება გამოიყენება ისეთი ამოცანებისათვის, სადაც საძებნი პასუხი შედგება ნაწილებისაგან, რომელთაგანაც თითოეული თავის მხრივ იძლევა რომელიღაც ქვეამოცანის ოპტიმალურ ამოხსნას. დინამიური პროგრამირება სასარგებლოა მაშინ, როცა მრავალგზის გვიხდება ერთი და იგივე ქვეამოცანების ამოხსნა და ასეთ დროს ძირითადი ტექნიკური საშუალებაა – დავიმახსოვროთ ამ ქვეამოცანის ამოხსნები იმ შემთხვევისათვის, როცა ისინი ისევ შეგვხვდებიან.

დინამიური პროგრამირების მეთოდი შეიძლება დავყოთ ოთხ ბიჯად:

1) აღვწეროთ ოპტიმალური ამოხსნის აგებულება;

2) ვიპოვოთ რეკურენტული თანაფარდობა ქვეამოცანებსა და ოპტიმალურ ამოხსნას შორის;

3) ქვემოდან ზემოთ მოძრაობისას გამოვთვალოთ პარამეტრის ოპტიმალური მნიშვნელობა ქვეამოცანებისათვის;

4) მიღებული ინფორმაციის საფუძველზე გამოვთვალოთ ოპტიმალური ამოხსნა.

სამუშაოს ძირითად ნაწილს წარმოადგენენ 1-3 ბიჯები. თუ ჩვენ გვაინტერესებს მხოლოდ პარამეტრის ოპტიმალური მნიშვნელობა, მაშინ მე-4 ბიჯი საჭირო არ არის, ხოლო თუ მე-4 ბიჯი აუცილებელია, მაშინ მე-3 ბიჯის შესრულების დროს მოგვიწევს დამატებითი ინფორმაციის მიღება და შენახვა.

ასეთი განზოგადების კონკრეტულ მაგალითზე ჩვენებისათვის განვიხილოთ მატრიცათა მიმდევრობის გადამრავლების ამოცანა. ორი A და B მატრიცა შეიძლება გადამრავლდეს მხოლოდ მაშინ, თუკი A მატრიცის სვეტების რაოდენობა ემთხვევა B მატრიცის სტრიქონების რაოდენობას. თუ A მატრიცის ზომებია $p\times q$ და B მატრიცის ზომებია $q\times r$, მაშინ მათი გადამრავლებით მიიღება $p\times r$ ზომის C მატრიცა და სტანდარტული ალგორითმით სრულდება pqr რაოდენობის ნამრავლის გამოთვლა. ვთქვათ, საჭიროა n ცალი მატრიცის $\langle A_1, A_2,\ldots,A_n\rangle$ გადამრავლება. ამ ამოცანის გადასაწყვეტად წინასწარ აუცილებელია ფრჩხილების სრული განთავსება, რათა განისაზღვროს გამრავლებათა თანმიმდევრობა. მაგალითად ოთხი $A_1A_2A_3A_4$ მატრიცის ნამრავლში ფრჩხილები შესაძლოა ხუთნაირად განთავსდეს:

$$(A_1(A_2(A_3A_4))),\ (A_1((A_2A_3)A_4)),\ ((A_1A_2)(A_3A_4)),\ ((A_1(A_2A_3))A_4),\ (((A_1A_2)A_3)A_4)$$

რადგან მატრიცების ნამრავლს გააჩნია ასოციაციურობის თვისება, საბოლოო შედეგი ყოველთვის ერთი და იგივეა, მაგრამ ჩატარებული ოპერაციების რაოდენობის მიხედვით ვარიანტები შეიძლება მკვეთრად განსხვავდებოდნენ. მაგალითად, სამი $\langle A_1, A_2, A_3\rangle$ მატრიცა შეიძლება ორნაირად გადავამრავლოთ: $((A_1A_2)A_3)$ და $(A_1(A_2A_3))$. ვთქვათ, მატრიცების ზომებია შესაბამისად 10×100, 100×5 და 5×50. $((A_1A_2)A_3)$ განლაგებით საჭიროა $10\cdot100\cdot5+10\cdot5\cdot50=7500$ ოპერაცია, ხოლო $(A_1(A_2A_3))$ განლაგებით – $100\cdot5\cdot50+10\cdot100\cdot50=75000$ ოპერაცია. მაშასადამე, პირველი გზით გამრავლება 10-ჯერ უფრო მომგებიანია.

საბოლოოდ, **6.2. მატრიცათა მიმდევრობის ნამრავლის ამოცანა** (matrix-chain multiplication problem) შეიძლება ასე ჩამოყალიბდეს: ვთქვათ, მოცემულია n ცალი მატრიცისაგან შემდგარი $\langle A_1, A_2,\ldots,A_n\rangle$ მიმდევრობა, რომელთა ზომებიც განსაზღვრულია. საჭიროა მოიძებნოს ფრჩხილების ისეთი სრული განლაგება, რომ მატრიცათა მიმდევრობის გადამრავლებისას შესრულდეს მინიმალური რაოდენობის გამრავლების ოპერაცია.

ამ ამოცანის გადასაწყვეტად სრული გადარჩევა არ გამოდგება, რადგან ვარიანტების რაოდენობა ექსპონენციალურადაა დამოკიდებული მატრიცების რაოდენობაზე. ამის დასამტკიცებლად მატრიცების მოცემული მიმდევრობა შეგვიძლია დავყოთ 3-3 წევრიან ჯგუფებად. თითოეულ ჯგუფში ნამრავლის გამოსათვლელად არსებობს ორი ვარიანტი. მაშასადამე 3n მატრიცისათვის იარსებებს არანაკლებ 2^n ვარიანტისა. ვცადოთ ამოცანის ამოხსნა დინამიური პროგრამირების მეთოდით:

ბიჯი 1. თავდაპირველად აღვწეროთ ოპტიმალურ ამოხსნათა აგებულება. აღვნიშნოთ $A_{i..j}$-ით მატრიცების ნამრავლი A_i მატრიციდან A_j მატრიცამდე. ცხადია, რომ იარსებებს ისეთი k ($1\le k<n$), რომ ყველა მატრიცის ნამრავლის გამოსათვლელად ჩვენ მოგვიწევს ჯერ $A_{1..k}$ და $A_{k..n}$ ნამრავლების გამოთვლა, ხოლო შემდეგ მიღებული შედეგების გადამრავლება. მაშინ ოპტიმალური განლაგების ღირებულება $A_{1..n}$-ისათვის ტოლი იქნება ოპტიმალური განთავსების ღირებულებების ჯამისა $A_{1..k}$-სა და $A_{k..n}$-სათვის პლიუს ამ ორი მატრიცის ნამრავლის ღირებულება.

რაც უფრო ნაკლები იქნება ნამრავლთა რაოდენობა $A_{1..k}$ და $A_{k..n}$ ნამრავლების გამოთვლისას, მით უფრო ნაკლები იქნება ნამრავლთა საერთო რაოდენობა. აქედან გამომდინარე, შეგვიძლია დავასკვნათ, რომ მატრიცათა მიმდევრობის გადამრავლების ოპტიმალური ამოხსნა შეიცავს ოპტიმალურ ამოხსნებს შემადგენელი ქვეამოცანებისათვის. ეს საშუალებას იძლევა გამოვიყენოთ დინამიური პროგრამირების მეთოდი.

ბიჯი 2. ახლა გამოვსახოთ ოპტიმალური ამოხსნის ღირებულება ქვეამოცანების ოპტიმალური ამოხსნებით. ასეთ ქვეამოცანებს წარმოადგენენ ფრჩხილთა ოპტიმალური განლგება $A_{i..j}$-ისათვის, სადაც $1\le i<j\le n$. აღვნიშნოთ m[i,j]-ით ნამრავლთა მინიმალური რაოდენობა, რომელიც საჭიროა $A_{i..j}$-ის გამოსათვლელად. შევნიშნოთ, რომ მთელი $A_{1..n}$ ნამრავლის ღირებულება იქნება m[1,n].

m[i,j] რიცხვი ასე გამოითვლება. თუ i=j, მაშინ m[i,i]=0, რადგან მიმდევრობა ერთი მატრიცისაგან შედგება და გამრავლება საჭირო არაა. თუ i<j, მაშინ ვისარგებლოთ პირველ ბიჯზე ჩამოყალიბებული ოპტიმალური ამოხსნის აგებულებით. ვთქვათ m[i,j]-ის გამოთვლის უკანასკნელ ბიჯზე ხდება $A_{i..k}$ და $A_{k+1..j}$ ნამრავლების გადამრავლება, სადაც $i\le k<j$. რადგან $A_{i..k}A_{k+1..j}$-ის გამოსათვლელად საჭიროა $p_{i-1}p_kp_j$ გამრავლების შესრულება, ცხადია, რომ

$$m[i,j]= m[i,k]+ m[k+1,j]+ p_{i-1}p_kp_j$$

ამ თანაფარდობაში k-მ შეიძლება მიიღოს მხოლოდ j-i განსხვავებული მნიშვნელობა. მათ შორის ერთ-ერთი ოპტიმალურია და მის საპოვნელად საჭიროა გადავარჩიოთ ეს მნიშვნელობები. მივიღეთ რეკურენტული ფორმულა:

$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{1\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & i<j\end{cases} \qquad (6.1)$$

m[i,j] რიცხვები ქვეამოცანების ოპტიმალური ამოხსნების ღირებულებებია. ოპტიმალური ამოხსნის მიღების უკეთ დასანახად შემოვიღოთ აღნიშვნა s[i,j]=k, რომლისთვისაც m[i,j]= m[i,k]+ m[k+1,j]+ $p_{i-1}p_kp_j$

ბიჯი 3. წინა ბიჯზე მიღებული რეკურენტული თანაფარდობის მიხედვით იოლად შეიძლება რეკურსიული ალგორითმის აგება, თუმცა მისი მუშაობის დრო, სრული გადარჩევის მსგავსად, ექსპონენციალურად იქნება დამოკიდებული n-ის მნიშვნელობაზე. დროში ნამდვილ მოგებას მხოლოდ მაშინ ვნახავთ, თუ გამოვიყენებთ იმ ფაქტს, რომ ქვეამოცანების რიცხვი მცირეა და არ აღემატება n^2-ს. რეკურსიული ალგორითმი კი ერთი და იგივე ქვეამოცანას მრავალჯერ ამოხსნის რეკურსიული ხის სხვადასხვა განშტოებაში. სწორედ ამითაა განპირობებული მისი მუშაობის ექსპონენციალური დრო, ხოლო ქვეამოცანების ასეთი "გადაფარვა" იმის ნიშანია, ამოცანა იხსნება დინამიური პროგრამირების მეთოდით.

რეკურსიის ნაცვლად ჩვენ გამოვთვლით ოპტიმალურ ღირებულებას "ქვემოდან ზემოთ". ქვემოთ მოყვანილ პროგრამულ კოდში შემავალ მონაცემია $p=\langle p_0, p_1,\dots,p_n\rangle$, სადაც length[p]=n+1. პროგრამა იყენებს დამხმარე მასივებს m[1..n,1..n] – m[I,j] ღირებულებების შესანახად და

s[1..n,1..n] – მასში აღინიშნება, თუ რომელი k-სათვის მიიღწევა ოპტიმალური ღირებულება m[I,j]-ის გამოთვლისას.

```
MATRIX-CHAIN-ORDER (p)
1 n=length[p]-1
2 FOR i=1 to n {
3            m[i,i]=0    }
4 FOR l=2 TO n {
5                FOR i=1 to n-l+1  {
6                        j=i+l-1
7                        m[i,j]= ∞
8                        FOR k=I TO j-1 {
9                                q=m[i,k]+m[k+1,j]+ p_{i-1}p_k p_j
10                               IF q<m[i,j] THEN {  m[i,j]=q;s[i,j]=k } } } }
11 return m,s
```

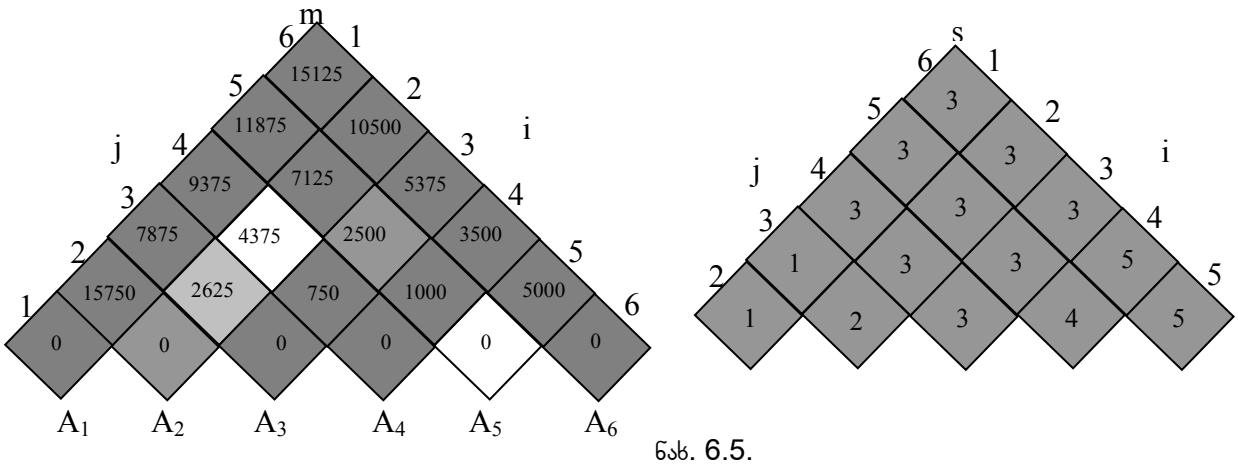
m მასივის შევსებისას ალგორითმი თანმიმდევრულად წყვეტს ამოცანას ფრჩხილების ოპტიმალური განთავსების შესახებ 1,2,...,n თანამამრავლისათვის. 6.1. ფორმულიდან ჩანს, რომ j-i+1 მატრიცის გადამრავლების ღირებულება – რიცხვი m[i,j], დამოკიდებულია მხოლოდ j-i+1 რიცხვზე ნაკლები მატრიცების გადამრავლების ღირებულებებზე. სახელდობრ, k=i,i+1,…,j-1 მნიშვნელობებისათვის ვღებულობთ, რომ $A_{i..k}$ არის k-i+1<j-i+1 მატრიცის ნამრავლი, ხოლო $A_{k+1..j}$ – j-k<j-i+1 მატრიცის ნამრავლი.

2-3 სტრიქონებში ალგორითმი ანულებს m მასივს. ციკლის პირველი გავლისას (4-10 სტრიქონები) გამოითვლის 2 სიგრძის მქონე ქვემიმდევრობების მინიმალურ ღირებულებებს. ციკლის მეორე გავლისას გამოითვლება მინიმალური ღირებულებები 3 სიგძის მქონე ქვემიმდევრობებისათვის და ა.შ. ყოველ ბიჯზე m[i,j]-ის მნიშვნელობის გამოთვლა დამოკიდებულია მხოლოდ მანამდე გამოთვლილ m[I,k]-სა და m[k+1,j]-ის მნიშვნელობებზე.

ნახ. 6.5-ზე ნაჩვენებია, თუ როგორ მიმდინარეობს გამოთვლები n=6-სათვის. რადგანაც ჩვენ განვსაზღვრავთ m[i,j]-ის მხოლოდ i≤j-სათვის, გამოიყენება მასივის მხოლოდ ის ნაწილი, რომელიც მთავარი დიაგონალის ზემოთაა მოთავსებული. მასივი შებრუნებულია და მთავარი დიაგონალი ჰორიზონტალურადაა. ქვემოთ მითითებულია მატრიცათა თანმიმდევრობა. m[i,j] რიცხვი – $A_iA_{i+1}\cdot\ldots\cdot A_j$ ნამრავლის მინიმალური ღირებულება – იმყოფება დიაგონალების გადაკვეთაზე, რომლებიც მიმართულნი არიან ზემოთ და მარჯვნივ A_i მატრიციდან და ზემოთ და მარცხნივ A_j მატრიციდან. ყოველ ჰორიზონტალურ რიგში თავმოყრილია ფიქსირებული სიგრძის მქონე ქვემიმდევრობების ნამრავლთა ღირებულებები. m[i,j] უჯრედის შესავსებად საჭიროა ვიცოდეთ $p_{i-1}p_kp_j$ ნამრავლი k=i,i+1,...,j-1-სათვის და m[i,j]-ის ქვედა-მარჯვენა და ქვედა-მარცხენა უჯრედების მნიშვნელობები.

ნახ. 6.5-ზე მოცემულ მატრიცათა ზომებია: A_1 – 30×35, A_2 – 35×15, A_3 – 15×5, A_4 – 5×10, A_5 – 10×20, A_6 – 20×25. m ცხრილში ნაჩვენებია მხოლოდ ის უჯრედები, რომლებიც არ მდებარეობენ მთავარი დიაგონალის ქვემოთ, ხოლო s ცხრილში – უჯრედები, რომლებიც მკაცრად ზემოთ მდებარეობენ. ყველა მატრიცის გადასამრავლებლად საჭირო ნამრავლთა მინიმალური ღირებულებაა m[1,6]=15125. ერთნაირი შეფერილობის მქონე უჯრედთა წყვილები ერთდროულად შედიან m[2,5]-ის გამოსათვლელი ფორმულის მარჯვენა ნაწილში:

$$m[2,5]=\min\begin{cases} m[2,2]+m[3,5]+p_1p_2p_5 = 0+2500+35\cdot15\cdot20 = 13000 \\ m[2,3]+m[4,5]+p_1p_3p_5 = 2625+1000+35\cdot5\cdot20 = 7125 \\ m[2,4]+m[5,5]+p_1p_4p_5 = 4375+0+35\cdot10\cdot20 = 11375 \end{cases}$$



ნახ. 6.5.

MATRIX-CHAIN-ORDER-ის მუშაობის დროა $O(n^3)$, რადგან ჩადგმული ციკლების რაოდენობა 3-ია და თითოეულის მთვლელი არ ღებულობს n-ზე მეტ მნიშვნელობას.

ბიჯი 4. ალგორითმი MATRIX-CHAIN-ORDER პოულობს ნამრავლთა მინიმალურ რაოდენობას, რომელიც საჭიროა მატრიცათა მიმდევრობის გადასამრავლებლად. ახლა მოვძებნოთ ფრჩხილების განლაგება, რომელიც მიგვიყვანს ნამრავლთა ასეთ რიცხვამდე. ამისათვის გამოვიყენოთ s[1..n,1..n] ცხრილი. s[1,n] უჯრედში ჩაწერილია უკანასკნელი გამრავლების ადგილი ფრჩხილების ოპტიმალური განლაგებისას. სხვაგვარად რომ ვთქვათ, $A_{1..n}$-ის ოპტიმალური გზით გამოთვლისას უკანასკნელად შესრულდა $A_{1..s[1,n]}$-ისა და $A_{s[1,n]+1..n}$-ის ნამრავლი. წინამორბედი გამრავლებები შეიძლება მოიძებნონ რეკურსიულად. ქვემოთ მოყვანილი რეკურსიული პროცედურა გამოითვლის $A_{i..j}$ ნამრავლს შემდეგი შემავალი მონაცემებით: მატრიცათა $A=\langle A_1,A_2,...,A_n\rangle$ მიმდევრობა, პროცედურა MATRIX-CHAIN-ORDER-ის მიერ გამოთვლილი s ცხრილი და i და j ინდექსები:

```
MATRIX-CHAIN-MULTIPLY(A,s,i,j)
1 if j>i then {   X= MATRIX-CHAIN-MULTIPLY(A,s,i,s[i,j])
2          Y= MATRIX-CHAIN-MULTIPLY(A,s[i,j]+1,j)
3          return MATRIX-MULTIPLY(X,Y) }
4else return Ai
```

ზემოთ მოყვანილი მაგალითისათვის პროცედურა განალაგებს ფრჩხილებს შემდეგნაირად:

$$((A_1(A_2A_3))((A_4A_5)A_6)).$$

როდისაა შესაძლებელი დინამიური პროგრამირების გამოყენება? შეიძლება მივუთითოთ ორი ნიშანი ამ ტიპის ამოცანებისათვის.

ოპტიმიზაციური ამოცანის დინამიური პროგრამირების მეთოდით ამოხსნისას ჯერ აუცილებელია ოპტიმალური ამოხსნის სტრუქტურის აღწერა. ვიტყვით, რომ ამოცანას **გააჩნია ოპტიმალურობის თვისება ქვეამოცანებისათვის** (has optimal substructure), თუ ამოცანის ოპტიმალური ამოხსნა შეიცავს ოპტიმალურ ამოხსნებს მისი ქვეამოცანებისათვის. თუკი ამოცანას ასეთი თვისება გააჩნია, მაშინ დინამიური პროგრამირება შეიძლება სასარგებლო აღმოჩნდეს მის ამოსახსნელად (ამ დროს შეიძლება ხარბმა ალგორითმმაც ივარგოს, იხ. 6.2).

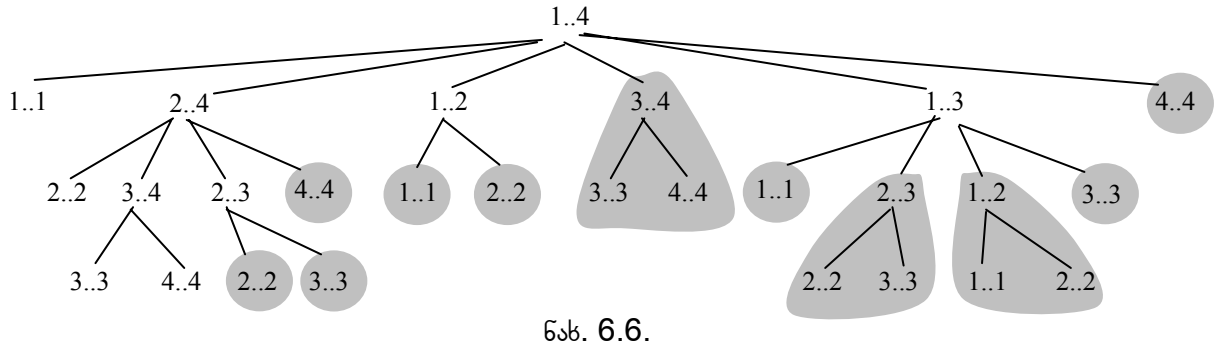
მატრიცების გადამრავლების ამოცანა ფლობს ოპტიმალურობის თვისებას ქვეამოცანებისათვის. ოპტიმალურ ნამრავლში განლაგებული ყოველი ფრჩხილი (ანუ ფრჩხილთა კონკრეტული წყვილი) მიუთითებს მასში შემავალი მატრიცების გამრავლების ოპტიმალურ გზას. იმაში დასარწმუნებლად, რომ ამოცანას გააჩნია ასეთი თვისება, უნდა ვაჩვენოთ, რომ ქვეამოცანის ამონახსნის გაუმჯობესებით გაუმჯობესდება საწყისი ამოცანის ამოხსნაც.

როცა ქვეამოცანების ოპტიმალურობის თვისება დადგინდება, როგორც წესი, ცხადი ხდება თუ ქვეამოცანების რა სიმრავლესთან ექნება საქმე ალგორითმს. მაგალითად, მატრიცების მიმდევრობის გადამრავლებისას ქვეამოცანებს წარმოადგენენ ამ მიმდევრობის ნაწილების გადამრავლების ამოცანები.

მეორე ნიშანი, რაც დამახასიათებელია დინამიური პროგრამირებით ამოხსნადი ამოცანებისათვის, არის სხვადასხვა ქვეამოცანების მცირე რიცხვი, რის გამოც ამოცანის რეკურსიული ამოხსნისას უამრავჯერ გვიხდება ერთი და იმავე ქვეამოცანის ამოხსნა. ამ შემთხვევაში იტყვიან, რომ ოპტიმიზაციურ ამოცანას გააჩნია **გადამფარავი ქვეამოცანები** (overlapping subproblems). ტიპიურ შემთხვევებში ქვეამოცანების რაოდენობა პოლინომიალურადაა დამოკიდებული საწყისი მონაცემების ზომაზე.

ამოცანებში, რომლებიც იხსნებიან პრინციპით "დაყავი და იბატონე" ასე არ ხდება. რეკურსიული ალგორითმი ყოველ ბიჯზე წარმოშობს ახალ ქვეამოცანებს. დინამიური პროგრამირების მეთოდი კი იყენებს გადამფარავ ქვეამოცანებს შემდეგნაირად: ყოველი ქვეამოცანა ამოიხსნება მხოლოდ ერთხელ და პასუხი შეიტანება სპეციალურ ცხრილში. როდესაც ეს ქვეამოცანა კვლავ შეგვხვდება, პროგრამა აღარ ხარჯავს დროს მის ხელმეორედ გამოთვლაზე და იღებს მზა პასუხს ცხრილიდან.

ნახ. 6.5-დან ჩანს, რომ მატრიცების გადამრავლების ამოცანაში თითოეული ქვეამოცანის ამონახსნი რამდენჯერმე გამოიყენება. მაგალითად, m[3,4] გამოიყენება ოთხჯერ: m[2,4], m[1,4], m[3,5] და m[3,6]-ს გამოთვლის დროს და ძალზე არაეფექტური იქნებოდა იგი ყოველთვის თავიდან გამოგვეთვალა.



ნახ. 6.6.

განვიხილოთ მატრიცათა მიმდევრობის გადამრავლების რეკურსიული (და ამის გამო არაეფექტური) ალგორითმი, რომელიც დაფუძნებულია (6.1) თანაფარდობებზე.

```
RECURSIVE-MATRIX-CHAIN (p,i,j)
1 if i=j then  { return 0 }
2 m[i,j]=∞
3 for k=i to j-1 {
4         q= RECURSIVE-MATRIX-CHAIN(p,i,k)+ RECURSIVE-MATRIX-CHAIN(p,i,j)+p_{i-1}p_kp_j
5         if q<m[i,j] then  {  m[i,j]=q  }         }
6 return m[i,j]
```

ნახ. 6.6-ზე მოცემულია რეკურსიის ხე RECURSIVE-MATRIX-CHAIN (p,1,4)-სათვის. ყოველ წვეროში ჩაწერილია i და j-ს მნიშვნელობები. რუხი ფერით აღნიშნულია ის წვეროები, რომელთა მნიშვნელობების გამოთვლა განმეორებით ხდება. ცხადია, რომ პროცედურის მუშაობის დრო T(n) ექსპონენციალურადაა დამოკიდებული n-ზე ($T(n)\geq 2^{n-1}$). ამის მიზეზია ერთი და იგივე ქვეამოცანების ყოველ ჯერზე თავიდან დათვლა, რისი თავიდან აცილება შეიძლება დინამიური პროგრამირებით.

განხილულ ამოცანაში დინამიური პროგრამირების მეთოდი მოქმედებდა "ქვემოდან ზემოთ" (ე.ი. ჯერ ხსნიდა ყველაზე მარტივ ქვეამოცანებს, შემდეგ კი მათი საშუალებით უფრო რთულებს და ა.შ.). იგივე მეთოდის (ქვეამოცანების ხელახალი გამოთვლის გამორიცხვა) გამოყენება შესაძლებელია "ზემოდან ქვემოთ" მომუშავე ალგორითმებისათვისაც. საამისოდ ყოველი ამოხსნილი ქვეამოცანის პასუხი უნდა დავიმახსოვროთ სპეციალურ ცხრილში. თავიდან ეს ცხრილი ცარიელია (ანუ შევსებულია nil-ით). ალგორითმის მუშაობის პროცესში პირველად შეხვედრილი ქვეამოცანის პასუხი გამოითვლება და შეიტანება ცხრილში. შემდგომში ამ ქვეამოცანის პასუხი აიღება ცხრილიდან. ჩვენს მაგალითში პასუხების ცხრილის შემოღება იოლია, რადგან ქვეამოცანები გადაინომრება (i,j) წყვილებით (უფრო რთულ

შემთხვევებში შეიძლება ჰეშირების გამოყენება). რეკურსიული ალგორითმების ასეთ გაუმჯობესებას ინგლისურად უწოდებენ memoization.

გამოვიყენოთ ეს გაუმჯობესება ალგორითმისათვის RECURSIVE-MATRIX-CHAIN.

```
MEMOIZED-MATRIX-CHAIN (p)
1 n=length[n]-1
2 for i=1 to n    {
3          for j=i to n     {
4                    m[i,j]=∞
5 return LOOKUP-CHAIN(p,1,n)

LOOKUP-CHAIN (p,i,j)
1 if m[i,j]< ∞ then return m[i,j]
2 if i=j     then {    m[i,j]=0   }
3          else {  for k=i to j-1      {
4          q= LOOKUP-CHAIN (p,i,k) + LOOKUP-CHAIN (p,k+1,j)+p_{i-1}p_kp_j
5          if q<m[i,j] then m[i,j]=q  }       }
6 return m[i,j]
```

თავდაპირველად m[i,j]= ∞ იმის მისათითებლად, რომ ცხრილში ეს უჯრედი შევსებული არ არის, შემდეგ შესაბამისი ქვეამოცანის პირველად ამოხსნისას უჯრედში ჩაიწერება პასუხი და თუ ამ ქვეამოცანის ამოხსნა კიდევ გახდა საჭირო, პასუხი უკვე პირდაპირ ცხრილიდან აიღება. 6.6 ნახაზზე ჩანს ის ეკონომია, რომელსაც ასეთი მიდგომა განაპირობებს. რუხად შეფერილი წვეროები შეესაბამება იმ შემთხვევებს, როცა განმეორებითი გამოთვლები საჭირო აღარ არის.

ალგორითმი MEMOIZED-MATRIX-CHAIN, ისევე როგორც ალგორითმი MATRIX-CHAIN-ORDER, საჭიროებს $O(n^3)$ დროს. ცხრილის თითოეული $\Theta(n^2)$ პოზიცია ერთხელ ინიციალიზირდება (MEMOIZED-MATRIX-CHAIN პროცედურის მე-4 სტრიქონი) და ერთხელ შეივსება – LOOKUP-CHAIN (p,i,j) პროცედურით კონკრეტული i და j-სათვის. LOOKUP-CHAIN (p,i,j)-ის გამოძახებები იყოფიან პირველ და განმეორებით გამოძახებებად. $\Theta(n^2)$ პირველი გამოძახებებიდან თითოეულს სჭირდება $O(n)$ დრო, ხოლო განმეორებით გამოძახებებს – $O(1)$ დრო. განმეორებითი გამოძახებების როადენობა არ აღემატება $O(n^3)$-ს. მაშასადამე, რეკურსიული ალგორითმი, რომელიც ითხოვდა ექსპონენციალურ $\Omega(2^n)$ დროს, გადაიქცა პოლინომიალურ ალგორითმად, რომელიც მუშაობს $O(n^3)$ დროში.

დასკვნის სახით შეგვიძლია ვთქვათ, რომ n მატრიცისაგან შედგენილი მიმდევრობის ოპტიმალური გდამრავლების ამოცანა შეიძლება ამოიხსნას $O(n^3)$ დროში ან "ზემოდან ქვემოთ" (რეკურსიული ალგორითმის პასუხების დამახსოვრებით) ან "ქვემოდან ზემოთ" (დინამიური პროგრამირება). ორივე ალგორითმი აგებულია ქვეამოცანათა გადაფარვაზე. ქვეამოცანათა რაოდენობა $\Theta(n^2)$-ია და ორივე ალგორითმი თითოეულ ქვეამოცანას მხოლოდ ერთხელ ხსნის.

საზოგადოდ, თუკი ყოველი ქვეამოცანა ერთხელ მაინც უნდა ამოიხსნას, მაშინ დინამიური პროგრამირება ("ქვემოდან ზემოთ") უფრო ეფექტურია ვიდრე რეკურსია პასუხების დამახსოვრებით, რადგან რეკურსიის რეალიზაცია (და ასევე პასუხის არსებობის შემოწმება ცხრილში) დამატებით დროს მოითხოვს. მაგრამ თუ ოპტიმუმის მოსაძებნად აუცილებელი არ არის ყველა ქვეამოცანის ამოხსნა, მაშინ უპირატესობა ენიჭება რეკურსიას პასუხების დამახსოვრებით ("ზემოდან ქვემოთ"), რადგან ის ხსნის მხოლოდ იმ ქვეამოცანებს, რომლებიც სჭირდება.

6.3 უდიდესი საერთო ქვემიმდევრობა. ვიტყვით, რომ მოცემული მიმდევრობიდან მივიღეთ ქვემიმდევრობა, თუ მიმდევრობის ზოგიერთი ელემენტი წავშალეთ, ხოლო დარჩენილ ელემენტებზე შენარჩუნებულია რიგი (თავად მიმდევრობაც ითვლება საკუთარ ქვემიმდევრობად). მათემატიკური ფორმულირებით: $Z=\langle z_1,z_2,\ldots z_k\rangle$ მიმდევრობას ეწოდება $X=\langle x_1,x_2,\ldots x_n\rangle$ მიმდევრობის **ქვემიმდევრობა** (subsequence), თუ არსებობს $\langle i_1,i_2,\ldots i_k\rangle$ ინდექსთა მკაცრად ზრდადი მიმდევრობა, რომლისთვისაც $z_j=x_i$, ყველა j=1,2,...,k-სათვის. მაგალითად,

Z=⟨B,C,D,B⟩ არის X=⟨A,B,C,B,D,A,B⟩ მიმდევრობის ქვემიმდევრობა. ინდექსთა შესაბამისი მიმდევრობაა ⟨2,3,5,7⟩. შევნიშნოთ, რომ მათემატიკისაგან განსხვავებით, ლაპარაკია მხოლოდ სასრულ მიმდევრობებზე.

ვიტყვით, რომ Z მიმდევრობა წარმოადგენს X და Y მიმდევრობების **საერთო ქვემიმდევრობას** (common subsequence), თუ Z წარმოადგენს როგორც X-ის, ასევე Y-ის ქვემიმდევრობას. მაგალითად, X=⟨A,B,C,B,D,A,B⟩, Y=⟨B,D,C,A,B,A⟩, Z= ⟨B,C,A⟩. ამ მაგალითში Z მიმდევრობა არ არის უდიდესი X-ისა და Y-ის საერთო ქვემიმდევრობათა შორის – ⟨B,C,B,A⟩ მიმდევრობა უფრო გრძელია. თავად ⟨B,C,B,A⟩ მიმდევრობა კი უდიდესს წარმოადგენს X-ისა და Y-ის საერთო ქვემიმდევრობათა შორის, რადგან 5 სიგრძის მქონე საერთო ქვემიმდევრობა არ არსებობს. უდიდესი საერთო ქვემიმდევრობა შეიძლება რამდენიმე იყოს, მაგ. ⟨B,D,A,B⟩ სხვა უდიდესი საერთო ქვემიმდევრობაა მოცემული X-ისა და Y-სათვის.

უდიდესი საერთო ქვემიმდევრობის (შემოკლებით უსქ, LCS=logest-common-subsequence) ამოცანა მდგომარეობს იმაში, რომ მოცემული X და Y მიმდევრობებისათვის ვიპოვოთ უდიდესი სიგრძის მქონე საერთო ქვემიმდევრობა. ეს ამოცანა იხსნება დინამიური პროგრამირების მეთოდით.

თუკი უსქ-ის ამოცანას ამოვხსნით სრული გადარჩევით, ალგორითმს დასჭირდება ექსპონენციალური დრო, რადგან m სიგრძის მიმდევრობა შეიცავს 2^m ქვემიმდევრობას (იმდენივეს, რამდენ ქვესიმრავლესაც შეიცავს {1,2,...,m} სიმრავლე). მაგრამ როგორც ქვემოთ მოყვანილი თეორემა გვაჩვენებს, უსქ-ის ამოცანას გააჩნია ოპტიმალურობის თვისება ქვეამოცანებისათვის. ქვეამოცანებად შეგვიძლია განვიხილოთ მოცემული ორი მიმდევრობის პრეფიქსების წყვილთა სიმრავლეები. ვთქვათ, $X=\langle x_1,x_2,\dots x_m\rangle$ რაღაც მიმდევრობაა. მისი i სიგრძის პრეფიქსი არის მიმდევრობა $X_i=\langle x_1,x_2,\dots x_i\rangle$, სადაც i მოთავსებულია 0-დან m-მდე. მაგალითად, თუ X=⟨A,B,C,B,D,A,B⟩, მაშინ $X_4=\langle A,B,C,B\rangle$, ხოლო X_0 ცარიელი ქვემიმდევრობაა

თეორემა 6.1 (უსქ-ის აგებულების შესახებ). *ვთქვათ $Z=\langle z_1,z_2,\dots z_k\rangle$ ერთ-ერთი უდიდესი საერთო ქვემიმდევრობაა $X=\langle x_1,x_2,\dots x_m\rangle$ და $Y=\langle y_1,y_2,\dots y_n\rangle$ მიმდევრობებისათვის. მაშინ*

1) თუ $x_m=y_n$, მაშინ $z_k= x_m=y_n$ და Z_{k-1} წარმოადგენს უსქ-ის X_{m-1}-ისა და Y_{n-1}-სათვის;

2) თუ $x_m\neq y_n$ და $z_k\neq x_m$ მაშინ Z წარმოადგენს უსქ-ის X_{m-1}-ისა და Y-სათვის;

3) თუ $x_m\neq y_n$ და $z_k\neq y_n$ მაშინ Z წარმოადგენს უსქ-ის X_m-ისა და Y_{n-1}-სათვის;

ამ თეორემიდან ჩანს, რომ ორი მიმდევრობის უსქ შეიცავს მათივე პრეფიქსების უსქ-ს. აქედან შეიძლება დავასკვნათ, რომ უსქ-ის ამოცანას გააჩნია ოპტიმალურობის თვისება ქვეამოცანებისათვის. როგორც ქვემოთ ვნახავთ, ადგილი აქვს ქვეამოცანების გადაფარვასაც.

6.1 თეორემა გვაჩვენებს, რომ უსქ-ის პოვნა $X=\langle x_1,x_2,\dots x_m\rangle$ და $Y=\langle y_1,y_2,\dots y_n\rangle$ მიმდევრობებისათვის დადის ან ერთი, ან ორი ქვეამოცანის ამოხსნაზე. თუ $x_m=y_n$, მაშინ საკმარისია ვიპოვოთ X_{m-1}-ისა და Y_{n-1}-ის უსქ და მას მივუწეროთ $x_m=y_n$. თუ $x_m\neq y_n$, მაშინ უნდა ამოიხსნას ორი ქვეამოცანა: ვიპოვოთ უსქ X_{m-1}-ისა და Y-სათვის, ვიპოვოთ ასევე უსქ X-ისა და Y_{n-1}-სათვის და მათ შორის უდიდესი იქნება X-ისა და Y-ის უსქ.

ზემოთ თქმულიდან ცხადი ხდება, რომ წარმოიქმნება ქვეამოცანების გადაფარვა, რადგან რომ ვიპოვოთ X-ისა და Y-ის უსქ, ჩვენ შეიძლება დაგვჭირდეს X_{m-1}-ისა და Y-ის, ასევე X-ისა და Y_{n-1}-ის უსქ-ების პოვნა, თითოეული ამ ამოცანიდან შეიცავს X_{m-1}-ისა და Y_{n-1}-ის უსქ-ის პოვნის ქვეამოცანას. ანალოგიური გადაფარვები შეგვხვდება სხვა შემთხვევებშიც.

ავაგოთ რეკურენტული თანაფარდობა ოპტიმალური მოხსნის ღირებულებისათვის. c[i,j]-ით აღვნიშნოთ უსქ-ის სიგრძე X_i და Y_i მიმდევრობებისათვის. თუ i ან j ტოლია 0-ის, მაშინ ორი მიმდევრობიდან ერთ-ერთი ცარიელია და c[i,j]=0. ზემოთ თქმული შეიძლება ასე ჩაიწეროს:

$$c[i,j]=\begin{cases}0, & \text{თუ } i=0 \text{ ან } j=0,\\ c[i-1,j-1], & \text{თუ } i,j>0 \text{ და } x_i=y_j,\\ \max(c[i,j-1],c[i-1,j]) & \text{თუ } i,j>0 \text{ და } x_i\neq y_j,\end{cases} \qquad (6.2)$$

(6.2) თანაფარდობიდან გამომდინარე, სირთულეს არ წარმოადგენს რეკურსიული ალგორითმის დაწერა, რომელიც ექსპონენციალურ დროში მოძებნიდა უსქ-ის სიგრძეს ორი

მოცემული მიმდევრობისათვის. მაგრამ რადგან განსხვავებული ქვეამოცანების რაოდენობა სულ $\Theta(mn)$, უმჯობესია დინამიური პროგრამირების გამოყენება.

LCS-LENGTH ალგორითმის შემავალი მონაცემებია $X=\langle x_1,x_2,\dots x_m\rangle$ და $Y=\langle y_1,y_2,\dots y_n\rangle$ მიმდევრობები. c[i,j] რიცხვები ჩაიწერება c[0..m,0..n] ცხრილში შემდეგნაირად: ჯერ შეივსება მარცხნიდან მარჯვნივ პირველი სტრიქონი, შემდეგ მეორე და ა.შ. გარდა ამისა ალგორითმი b[1..m,1..n] ცხრილში იმახსოვრებს c[i,j]-ის "წარმომავლობას". b[i,j] უჯრედში შეიტანება ისარი, რომელიც მიუთითებს შემდეგი სამი უჯრედიდან ერთ-ერთის კოორდინატებს: (i-1,j-1), (i-1,j) ან (i,j-1), იმისდა მიხედვით თუ რისი ტოლია c[i,j] შესაბამისად – c[i-1,j-1]+1, c[i-1,j] თუ c[i,j-1] (იხ. ნახ. 6.7). ალგორითმის მუშაობის შედეგებია c და b ცხრილები.

```
LCS-LENGTH (X,Y)
1 m=length[X]
2 n=length[Y]
3 for i=1 to m {   c[i,0]=0  }
4 for j=1 to n  {   c[0,j]=0  }
5 for i=1 to m {
6          for j=1 to n  {
7                    if  xi=yj  then  {       c[i,j]=c[i-1,j-1]+1
8                                                b[i,j]="↖"    }
9                              else  {   if  c[i-1,j]≥c[i,j-1]   then {  c[i,j]=c[i-1,j]
10                                                                                b[i,j]="↑"  }
11                                                                    else {  c[i,j]=c[i,j-1]
12                                                                                b[i,j]="←"  } } } }
13 return c,b
```

ნახ. 6.7-ზე ნაჩვენებია LCS-LENGTH ალგორითმის მუშაობა $X=\langle A,B,C,B,D,A,B\rangle$ და $Y=\langle B,D,C,A,B,A\rangle$ მიმდევრობებისათვის. b და c ცხრილები გაერთიანებულია და (i,j) კოორდინატების მქონე უჯრედში ჩაწერილია რიცხვი c[i,j] და ისარი b[i,j]. რიცხვი 4 ცხრილის ქვედა მარჯვენა უჯრედში წარმოადგენს უსქ-ის სიგრძეს. ამ პასუხის მიღების გზა ნახაზზე რუხი ფერითაა ნაჩვენები.

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	←1	↖ 1
2	B	0	↖ 1	←1	←1	↑ 1	↖ 2	←2
3	C	0	↑ 1	↑ 1	↖ 2	←2	↑ 2	2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	←3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

ნახ. 6.7.

LCS-LENGTH ალგორითმის მუშაობის დროა $O(mn)$. თითოეული უჯრედის შესავსებად საჭიროა $O(1)$ ბიჯი.

უსქ-ის სიგრძის პოვნის შემდეგ b ცხრილის საშუალებით შეგვიძლია დავადგინოთ თავად უსქ. ამისათვის საჭიროა b[m,n] უჯრედიდან დავბრუნდეთ უკან და ვიპოვოთ "↖" ისრები. ამის რელიზაციას ახდენს რეკურსიული პროცედურა PRINT-LCS:

```
PRINT-LCS (b,X,i,j)
```

```
1 if (i=0 OR j=0) then  {  return  }
2 if b[i,j]= “↖” then {   PRINT-LCS (b,X,i-1,j-1)
3                                print x_i      }
4 elseif b[i,j]= “↑” then {   PRINT-LCS (b,X,i-1,j)  }
5                          else {   PRINT-LCS (b,X,i,j-1)  }
```

მოყვანილი მაგალითისათვის პროცედურა დაბეჭდავს BCBA-ს. პროცედურის მუშაობის დროა O(m+n), რადგან ყოველ ბიჯზე ან m მცირდება და ან – n.

უსქ-ის ის ამოცანაზე მსჯელობის დასრულებისას შევნიშნოთ, რომ შესაძლებელია ალგორითმის გაუმჯობესებაც. მაგალითად, ჩვენს შემთხვევაში შეიძლებოდა b მასივი საერთოდ არ გამოგვეყენებინა და უსქ c მასივის გადამოწმებით დაგვედგინა. ნებისმიერი c[i,j] რიცხვისათვის მოგვიწევდა შეგვემოწმებინა c[i-1,j], c[i,j-1] და c[i-1,j-1] უჯრედები, რისთვისაც O(1) დროა საჭირო. მაშასადმე, უსქ-ის პოვნა იგივე O(m+n) დროში ერთი ცხრილითაც შეიძლებოდა.

განვიხილოთ ამოცანა, რომელიც იხსნება უსქ-ის პოვნის ალგორითმის გამოყენებით.

6.4. ვორდულატორი

(საქართველოს მოსწავლეთა ოლიმპიადის ზონური ტური, 1996 წელი)

ვორდულატორი არის გამარტივებული კომპიუტერი, რომელიც შედგება მონიტორისა და კლავიატურისაგან. მონიტორი შეიცავს ერთ ტექსტურ სტრიქონს და ერთ კურსორის სტრიქონს, რომელიც ზუსტად ტექსტური სტრიქონის ქვემოთაა განლაგებული. თითოეული სტრიქონის სიგრძე არის 40 პოზიცია.

ტექსტური სტრიქონი შეიძლება შეიცავდეს მხოლოდ დიდ ლათინურ ასოებს და ჰარებს. ტექსტური სტრიქონის ყოველი პოზიცია შეესაბამება კურსორის სტრიქონის ზუსტად ერთ პოზიციას და პირიქით. ყოველ მომენტში კურსორის სიმბოლო მდებარეობს კურსორის სტრიქონის ერთ პოზიციაში, ხოლო დანარჩენი 39 ცარიელია. თუ კურსორი არის პოზიციაში, რომელიც შეესაბამება ტექსტური სტრიქონის i-ურ პოზიციას, ამბობენ, რომ იგი არის i-ური სიმბოლოს ქვეშ. ტექსტი არ შეიძლება იყოს კურსორის სტრიქონში და კურსორი არ შეიძლება იყოს ტექსტურ სტრიქონში.

მონიტორის მაგალითი:

HIDROELECTROSTACIJ

სიტყვა " HIDROELECTROSTACIJ" არის მონიტორის ტექსტურ სტრიქონში, ხოლო კურსორი მოთავსებულია მეოთხე სიმბოლოს ქვემოთ.

ვორდულატორის კლავიატურა შედგება შემდეგი 29 ღილაკისაგან:

<	>	-	A	B	C	D	E	F	G	H	I	J	K	L
M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	

ტექსტურ და კურსორის სტრიქონებში ცვლილებები შეიძლება მოხდეს მხოლოდ ვორდულატორის კლავიატურის რომელიმე ღილაკზე დაჭერის საშუალებით. ღილაკების შესაბამისი მოქმედებები მოცემულია შემდეგი ცხრილით:

ღილაკი	ოპერაციის სახელი	ოპერაციის აღწერა
<	კურსორი მარცხნივ	თუ კურსორი არის პირველი სიმბოლოს ქვეშ, მაშინ არაფერი იცვლება. სხვა შემთხვევაში კურსორი გადაადგილდება ერთი პოზიციით მარცხნივ. ამ ოპერაციის დროს ტექსტური სტრიქონის შემადგენლობა არ იცვლება. ოპერაციის შესრულების დროა 1წმ.
>	კურსორი მარჯვნივ	თუ კურსორი არის მე-40-ე სიმბოლოს ქვეშ, მაშინ არაფერი იცვლება. სხვა შემთხვევაში კურსორი გადაადგილდება ერთი პოზიციით მარჯვნივ. ამ ოპერაციის დროს ტექსტური სტრიქონის შემადგენლობა არ იცვლება. ოპერაციის შესრულების დროა 1წმ.

-	ტექსტური სტრიქონის იმ სიმბოლოს წაშლა, რომლის ქვეშაც კურსორია მოთავსებული	თუ კურსორი მოთავსებულია არაა მე-40-ე სიმბოლოს ქვეშ, მაშინ ტექსტის ნაწილი, განლაგებული იმ სიმბოლოს მარჯვნივ, რომლის ქვეშაც კურსორია, გადაადგილდება ერთი პოზიციით მარცხნივ. მე-40-ე პოზიციაში გვექნება ცარიელი სიმბოლო. ოპერაციის დროს კურსორი ადგილზე რჩება. ოპერაციის შესრულების დროა 10წმ.
A..Z	შესაბამისი სიმბოლოს ჩამატება სტრიქონის იმ პოზიციაში, რომლის ქვეშაც კურსორია მოთავსებული	თუ კურსორი მოთავსებულია არაა მე-40-ე სიმბოლოს ქვეშ, მაშინ ტექსტის ნაწილი, დაწყებული იმ სიმბოლოთი, რომლის ქვეშაც დგას კურსორი, გადაინაცვლებს ერთი პოზიციით მარჯვნივ (თუ ტექსტში გვაქვს ყველა 40 სიმბოლო, მაშინ ბოლო სიმბოლო გაქრება). ტექსტური სტრიქონის იმ პოზიციაში, რომლის ქვემოთაც კურსორია მოთავსებული, ჩაიწერება კლავიატურაზე აკრეფილი სიმბოლო. ოპერაციის შესრულები დროს კურსორი ადგილზე რჩება. ოპერაციის შესრულების დროა 10წმ.

ვორდულატორის გამოყენების მაგალითი:

დავუშვათ მონიტორზე გვაქვს ასეთი მდგომარეობა: MAIJ

კლავიატურაზე “<” სიმბოლოს დაჭერით მონიტორზე გვექნება: MAIJ

კლავიატურაზე “>” სიმბოლოს დაჭერით მივიღებთ ისევ საწყის მდგომარეობას: MAIJ

კლავიატურაზე “-” სიმბოლოს დაჭერით მივიღებთ: MAI

კლავიატურაზე “G” სიმბოლოს დაჭერით მივიღებთ: MAGI

მარტივი გამოთვლები გვიჩვენებს, რომ ამ ოთხ ოპერაციას ვორდულატორი 22 წამს მოანდომებს.

NOR მდგომარეობიდან ოპერაციების: > > > U > T < < < - < - მიმდევრობის შესრულების შემდეგ ეკრანზე გვექნება: RUT. ამ ოპერაციების შესასრულებლად ვორდულატორს დასჭირდება 48 წამი

დაწერეთ პროგრამა, რომელიც ორი მოცემული სიტყვისათვის, რომლებიც შეიცავენ მხოლოდ დიდ ლათინურ ასოებს, განსაზღვრავს პირველი მათგანიდან მეორეს მისაღებად საჭირო უმცირეს შესაძლო დროს (წამებში) შესაბამის ოპერაციათა მიმდევრობის შესრულებით.

დამატებით ცნობილია, რომ ა) პირველი სიტყვა მოცემულია ვორდულატორის ტექსტურ სტრიქონში, იწყება პირველი პოზიციიდან და კურსორი მდებარეობს პირველი სიმბოლოს ქვეშ; ბ) თითოეული სიტყვის სიგრძე არაა 20 სიმბოლოზე მეტი; გ) ოპერაციათა მიმდევრობის დასრულების შემდეგ კურსორი უნდა იდგეს პირველი სიმბოლოს ქვეშ; დ) პროგრამის გამომავალ მონაცემს უნდა ჰქონდეს შემდეგი სახე: <პირველი სიტყვა> <მეორე სიტყვა> <გამოთვლილი უმცირესი დრო>

მაგალითები:

შესატანი მონაცემები	გამოსატანი მონაცემები	კომენტარი (მიმდევრობა)
ES TU	ES TU 40	- - U T
PAKA PIKA	PAKA PIKA 22	> - I <

მითითება. რადგან წაშლისა და ჩამატების ოპერაციები გაცილებით “ძვირია” მოძრაობის ოპერაციებთან შედარებით, ამიტომ უმჯობესი იქნება რაც შეიძლება ნაკლები სიმბოლო წავშალოთ პირველ სიტყვაში და რაც შეიძლება ნაკლები სიმბოლოს ჩამატება დაგვჭირდეს მეორე სიტყვიდან. ამისათვის შემომავალი მონაცემები განვიხილოთ როგორც მიმდევრობები და ვიპოვოთ მათი უდიდესი საერთო ქვემიმდევრობა. ყველა სიმბოლო პირველი სიტყვიდან, რომელიც ამ ქვემიმდევრობაში არ შევა, უნდა წაიშალოს, ხოლო ანალოგიური სიმბოლოები მეორე სიტყვიდან – უნდა ჩაემატოს.

უდიდესი საერთო ქვემიმდევრობის განსაზღვრის შემდეგ რჩება მხოლოდ კურსორის მოძრაობის ოპტიმიზაცია. ცხადია, რომ უმჯობესია პირველ რიგში კურსორი ვამოძრაოთ მარჯვნივ და თან წავშალოთ ყველა წასაშლელი სიმბოლო პირველი სიტყვიდან, რათა შემდეგ

კურსორს მათზე მოძრაობა არ მოუხდეს. როდესაც წასაშლელი სიმბოლოებიდან ყველაზე მარჯვენას წავშლით, უნდა განვსაზღვროთ გვჭირდება თუ არა კიდევ უფრო მარჯვნივ მოძრაობა სიმბოლოს ჩასამატებლად. საჭიროების შემთხვევაში კურსორს მივიყვანთ შესაბამის პოზიციამდე და შემდეგ დავიწყებთ სიმბოლოების ჩამატებას კურსორის მარჯვნიდან მარცხნივ მოძრაობისას.

განვიხილოთ ამოცანის ერთ-ერთი ტესტი: ABRAKADABRA და RABARBERAUSIS.

		A	B	R	A	K	A	D	A	B	R	A
		0	0	0	0	0	0	0	0	0	0	0
R	0	0	0	1	1	1	1	1	1	1	1	1
A	0	1	1	1	2	2	2	2	2	2	2	2
B	0	1	2	2	2	2	2	2	2	3	3	3
A	0	1	2	2	3	3	3	3	3	3	3	4
R	0	1	2	3	3	3	3	3	3	3	4	4
B	0	1	2	3	3	3	3	3	3	4	4	4
E	0	1	2	3	3	3	3	3	3	4	4	4
R	0	1	2	3	3	3	3	3	3	4	5	5
R	0	1	2	3	3	3	3	3	3	4	5	5
A	0	1	2	3	4	4	4	4	4	4	5	6
U	0	1	2	3	4	4	4	4	4	4	5	6
S	0	1	2	3	4	4	4	4	4	4	5	6
I	0	1	2	3	4	4	4	4	4	4	5	6
S	0	1	2	3	4	4	4	4	4	4	5	6

ნახ. 6.8

ამ ორი სიტყვის უდიდესი საერთო ქვემიმდევრობაა RAABRA (ნახ. 6.8-ზე ეს სიმბოლოები ორივე სიტყვაში რუხ ფონზეა ნაჩვენები). პირველ სიტყვაში წასაშლელია ხუთი სიმბოლო – 1, 2, 4, 5 და 7 პოზიციებზე და მეორე სიტყვიდან დასამატებელია 8 სიმბოლო – 3, 5, 7, 8, 12, 13, 14, 15. ეს ნიშნავს, რომ წაშლა-ჩამატებაზე დაგვეხარჯება 130 წამი. რადგან ჩამატება გვჭირდება პირველი სიტყვის ბოლოში, ამიტომ მოგვიწევს მისი ექვსივე სიმბოლოს გავლა ჯერ მარჯვნივ და მერე მარცხნივ. ამ ოპერაციებს დასჭირდება 12 წამი. მაშასადამე სულ საჭირო იქნება 130+12=142 წამი.

მოვიყვანოთ ოპერაციათა მიმდევრობა: - - > - - > - > > > > S I S U < < R E < R < B < <.

დასასრულ, შევნიშნოთ, რომ თუკი საწყის და საბოლოო სიტყვებს ერთნაირი სუფიქსები აქვთ, შეგვიძლია ისინი უგულვებელვყოთ და ნაკლები სიგრძის სიტყვები განვიხილოთ, თუმცა შემომავალი მონაცემები იმდენად დიდი არ არის, რომ ამან შესამჩნევი გავლენა მოახდინოს ალგორითმის მუშაობის სიჩქარეზე.

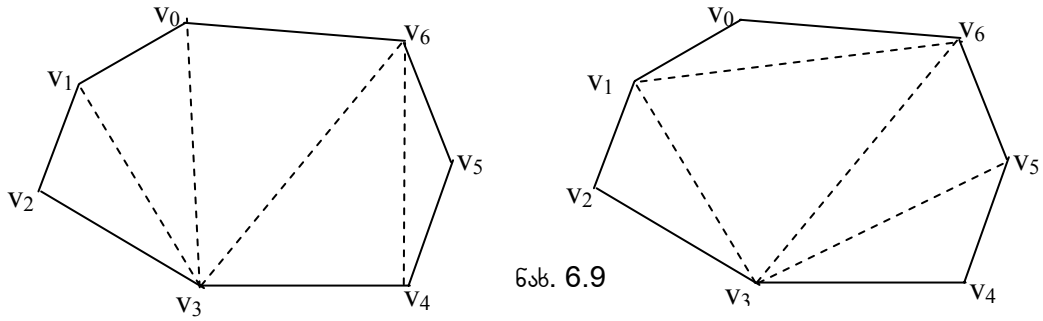
6.5. მრავალკუთხედის ოპტიმალური ტრიანგულაცია. მიუხედავად გეომეტრიული ფორმულირებისა, ეს ამოცანა ძალიან წააგავს მატრიცების გადამრავლების ამოცანას.

მრავალკუთხედი (polygon) – ესაა სიბრტყეზე მოთავსებული შეკრული ტეხილი, რომელიც შედგება მრავალკუთხედის **გვერდებად** (sides) წოდებული მონაკვეთებისაგან. წერტილს, რომელშიც ერთდება ორი მეზობელი გვერდი, უწოდებენ **წვეროს** (vertex). მრავალკუთხედს, რომელიც თავის თავს არ კვეთს, უწოდებენ **მარტივს** (simple). სიბრტყის წერტილებს, რომლებიც მდებარეობენ მარტივი მრავალკუთხედის შიგნით, უწოდებენ მრავალკუთხედის **შიგა არეს** (interior), მრავალკუთხედის გვერდების გაერთიანებას უწოდებენ **საზღვარს** (boundary), ხოლო სიბრტყის ყველა დანარჩენი წერტილების სიმრავლეს – **გარეთა არეს** (exterior). მარტივ მრავაკუთხედს უწოდებენ **ამოზნექილს** (convex), თუკი შიგა არეში ან საზღვარზე მდებარე ნებისმიერი ორი წერტილის შემაერთებელი მონაკვეთი მთლიანად მოთავსებულია მრავაკუთხედის შიგნით ან საზღვარზე.

ამოზნექილი მრავალკუთხედი შეგვიძლია აღვწეროთ მისი წვეროების ჩამოთვლით საათის ისრის საწინააღმდეგო მიმართულებით. $P=\langle v_0, v_1, \ldots v_{n-1}\rangle$ მრავალკუთხედს აქვს n გვერდი: v_0v_1, $v_1v_2, \ldots$, $v_{n-1}v_0$. თუ v_i და v_j წვეროები არ წარმოადგენენ მეზობელ წვეროებს მაშინ

v_iv_j მონაკვეთს უწოდებენ მრავალკუთხედის **დიაგონალს** (chord). v_iv_j დიაგონალი მრავალკუთხედს ჰყოფს ორად – $<v_i,v_{l+1},\dots v_j>$ და $<v_j,v_{j+1},\dots v_i>$. მრავალკუთხედის **ტრიანგულაცია** (triangulatuion) – ესაა დიაგონალთა ერთობლიობა, რომლებიც მრავალკუთხედს სამკუთხედებად ჰყოფს. ამ სამკუთხედების გვერდებს წარმოადგენენ საწყისი მრავალკუთხედის გვერდები და ტრიანგულაციის დიაგონალები.

ნახ. 6.9-ზე ნაჩვენებია მრავაკუთხედის ორგვარი ტრიანგულაცია. ტრიანგულაცია ასევე შეიძლება განისაზღვროს, როგორც იმ დიაგონალთა მაქსიმალური სიმრავლე, რომლებიც არ იკვეთებიან.



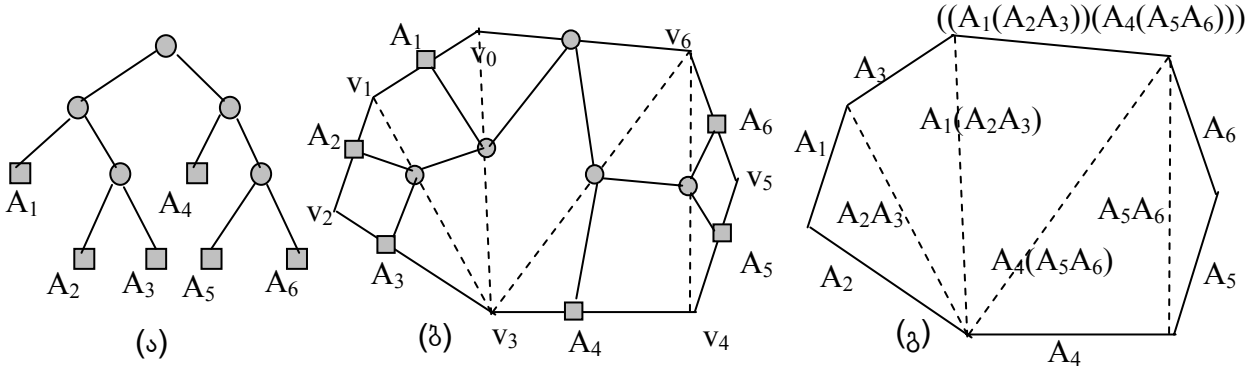
ნახ. 6.9

n-კუთხედის ნებისმიერი ტრიანგულაციისას საჭიროა სამკუთხედების ერთნაირი რაოდენობა. კერძოდ, ის იყოფა n-2 სამკუთხედად და ამისათვის გამოიყენება n-3 დიაგონალი. მრავაკუთხედის ყველა კუთხის ჯამი ტოლია 1800-ის ნამრავლისა სამკუთხედების რიცხვზე ტრიანგულაციაში.

ოპტიმალური ტრიანგულაციის ამოცანა (optimal triangulation problem) მდგომარეობს შემდეგში: მოცემულია ამოზნექილი მრავალკუთხედი $P=<v_0,v_1,\dots v_{n-1}>$ და წონითი ფუნქცია w, რომელიც განსაზღვრულია P წერტილებში წვეროების მქონე სამკუთხედების სიმრავლეზე. საჭიროა მოიძებნოს ტრიანგულაცია, რომლისათვისაც სამკუთხედის წონათა ჯამი უმცირესი იქნება.

წონითი ფუნქციის მაგალითია $w(\Delta v_iv_jv_k)=|v_iv_j|+|v_jv_k|+|v_kv_i|$, სადაც $|v_iv_j|$ აღნიშნავს ევკლიდურ მანძილს v_i და v_j-ის შორის. ამოცანის ამოხსნის ქვემოთ მოყვანილი ალგორითმი შეიძლება გამოყენებულ იქნას ნებისმიერი სახის წონითი ფუნქციისათვის.

არსებობს კავშირი მრავალკუთხედის ტრიანგულაციასა და ფრჩხილების განლაგების შორის. ეს კავშირი ნათლად გამოიხატება ხეების საშუალებით. ფრჩხილების სრულ განლაგებას შეესაბამება ე.წ. **განლაგების ხე** (parse tree). (ა) ნახაზზე გამოსახულია განლაგების ხე $((A_1(A_2A_3))(A_4(A_5A_6)))$ გამოსახულებისათვის. მის ფოთლებში დგანან თანამამრავლი მატრიცები, ხოლო წვეროებში მათი ნამრავლები. ამოზნექილი მრავალკუთხედის ტრიანგულაციაც ასევე შეიძლება წარმოვადგინოთ ხის სახით, სადაც ხის ფოთლები იქნებიან მრავალკუთხედის გვერდები გარდა v_0v_{n-1}-ისა. ეს უკანასკნელი იქნება ფუძე, ხოლო ტრიანგულაციის დიაგონალები – წვეროები.



ნახ. 6.10.

6.10(ბ) ნახაზზე ნაჩვენებია შესაბამისობა ტრიანგულაციასა და ორობით ხეს შორის და თუ როგორ უნდა აიგოს ეს ხე. ჯერ ვპოულობთ, თუ რომელ სამკუთხედში მოხვდა ხის ძირი – v_0v_{n-1}. ჩვენს შემთხვევაში ესაა $\Delta v_0v_3v_6$. ხის ძირის შვილებად ჩავთვალოთ ამ სამკუთხედის სხვა ორი

გვერდი. მრავალკუთხედის ტრიანგულაცია შედგება ამ სამკუთხედისა და დარჩენილი ნაწილების ($\langle v_0,v_1,v_2,v_3\rangle$ და $\langle v_3,v_4,v_5,v_6\rangle$) ტრიანგულაციებისაგან. ამასთან ხის ძირის შვილი დიაგონალები წარმოადგენენ ამ მრავალკუთხედის გვერდებს. გავიმეოროთ იგივე კონსტრუქცია მიღებული მრავალკუთხედებისათვის და ა.შ. ბოლოს ჩვენ მივიღებთ ორობით ხეს n-1 ფოთლით.თუკი ვიმოქმედებთ შებრუნებული თანმიმდევრობით, შეგვიძლია ორობითი ხიდან ავაგოთ ტრიანგულაცია. აგებული შესაბამისობა ორობით ხესა და ტრიანგულაციას შორის ურთიერთცალსახაა.

თუ გავიხსენებთ, რომ n მატრიცის ნამრავლში ფრჩხილების სრული განლაგება ურთიერთცალსახა დამოკიდებულებაშია n ფოთლის მქონე ორობით ხესთან, შეგვიძლია დავასკვნათ, რომ არსებობს ასევე ურთიერთცალსახა დამოკიდებულება n მატრიცის ნამრავლში ფრჩხილების სრულ განლაგებასა და (n+1)-კუთხედის ტრიანგულაციას შორის. ამასთან A_i მატრიცა $A_1A_2\cdot\ldots\cdot A_n$ ნამრავლში შეესაბამება $v_{i-1}v_i$ გვერდს, ხოლო $v_{i-1}v_j$ $(1\le i<j\le n)$ დიაგონალი შეესაბამება $A_{i..j}$ ნამრავლს.

ეს შესაბამისობა ხის გარეშეც შეიძლება აიხსნას. ტრიანგულირებული მრავალკუთხედის ყველა გვერდს ერთის გარდა მივუწეროთ თითო თანამამრავლი. შემდეგ სამკუთხედში, რომლის ორი გვერდი უკვე მონიშნულია, მესამე გვერდს მივუწეროთ ამ ორი გვერდის ნამრავლი. საბოლოოდ, თავდაპირველად მოუნიშნავ გვერდზე მივიღებთ ფრჩხილების სრულ განლაგებას (იხ. ნახ. 6.10(გ)).

შევნიშნოთ, რომ მატრიცების გადამრავლების ამოცანა ოპტიმალური ტრიანგულაციის ამოცანის კერძო შემთხვევაა. ვთქვათ, ჩვენ უნდა გამოვთვალოთ $A_1A_2\cdot\ldots\cdot A_n$, სადაც A_i წარმოადგენს $p_{i-1}\times p_i$ მატრიცას. განვიხილოთ $(n+1)$-კუთხედი $P=\langle v_0,v_1,\ldots,v_n\rangle$ და წონითი ფუნქცია $w(\Delta v_iv_jv_k)=p_ip_jp_k$, მაშინ ტრიანგულაციის ღირებულება გადამრავლებათა რიცხვის ტოლი იქნება ფრჩხილების შესაბამისი განლაგებისას.

მიუხედავად იმისა, რომ მატრიცების გადამრავლების ამოცანა ოპტიმალური ტრიანგულაციის ამოცანის კერძო შემთხვევაა, ალგორითმი MATRIX-CHAIN-ORDER იოლად შეიძლება გადავაკეთოთ ტრიანგულაციის ამოცანაზე. ამისათვის საკმარისია სათაურში p შევცვალოთ v-თი და მე-9 სტრიქონი შევცვალოთ ასე:

9 $q=m[i,k]+m[k+1,j]+ w(\Delta v_{i-1}v_kv_j)$

და ალგორითმის მუშაობის შედეგად m[1,n] გახდება ოპტიმალური ტრიანგულაციის წონის ტოლი.

ახლა განვიხილოთ რეკურენტული ფორმულა. ვთქვათ, m[i,j] არის $\langle v_{i-1},v_i,\ldots,v_j\rangle$ მრავალკუთხედის ოპტიმალური ტრიანგულაციის წონა, სადაც $1\le i<j\le n$. მთელი მრავაკუთხედის ოპტიმალური ტრიანგულაციის წონა ტოლია m[1,n]. ჩავთვალოთ, რომ $\langle v_{i-1},v_i\rangle$ "ორკუთხედების" წონა არის 0. მაშინ $m[i,i]=0$, ნებისმიერი $i=1,2,\ldots,n$-სათვის. თუ $j-i\ge 1$, მაშინ $\langle v_{i-1},v_i,\ldots,v_j\rangle$ მრავალკუთხედში გვაქვს არანაკლებ სამი წვეროსი და ყველა k-სათვის $i\le k\le j-1$ შუალედიდან უნდა ვიპოვოთ ასეთი ჯამის მინიმუმი: $\Delta v_{i-1}v_kv_j$-ის წონა პლუს $\langle v_{i-1},v_i,\ldots,v_k\rangle$-ს ოპტიმალური ტრიანგულაციის წონა პლუს $\langle v_k,v_{k+1},\ldots,v_j\rangle$-ს ოპტიმალური ტრიანგულაციის წონა. ამიტომ

$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\left\{m[i,k]+m[k+1,j]+w\left(\Delta v_{i-1}v_kv_j\right)\right\} & i<j\end{cases} \qquad (6.3)$$

ამ ფორმულის ერთადერთი განსხვავება (6.1) ფორმულიდან წონითი ფუნქციის უფრო ზოგადი სახეა, ამიტომ ალგორითმი MATRIX-CHAIN-ORDER მითითებული ცვლილებებით გამოითვლის ოპტიმალური ტრიანგულაციის წონას. ალგორითმის მუშაობის დროა $\Theta(n^3)$, ხოლო გამოყენებული მეხსიერება – $\Theta(n^2)$.

განვიხილოთ რამდენიმე ამოცანა დინამიური პროგრამირების მეთოდზე:

6.6. კიბე

დაწერეთ პროგრამა, რომელიც გაარკვევს რამდენნაირი განსხვავებული გზით შეიძლება კიბის მე-20 საფეხურზე ასვლა, თუკი ყოველ სვლაზე შესაძლებელია ერთი ან ორი საფეხურით გადაადგილება.

მითითება. ცხადია, რომ ნებისმიერ i-ურ საფეხურზე შესაძლებელია მოვხვდეთ მხოლოდ წინა ორი (i-1) და (i-2) საფეხურიდან. თუკი გვეცოდინება, რამდენი განსხვავებული გზით შეიძლება მოვხვდეთ ამ ორ საფეხურზე, მაშინ i-ურ საფეხურზე მისვლის ვარიანტების რაოდენობა მათი ჯამი იქნება. ასეთი დასკვნის საფუძველს გვაძლევს ის ფაქტი, რომ (i-2) საფეხურიდან – ორ საფეხურზე, ხოლო (i-1) საფეხურიდან ერთ საფეხურზე ანაცვლებით ავალთ i-ურ საფეხურზე. ის ვარიანტი, რომლითაც (i-2) საფეხურიდან (i-1)-ზე შეიძლება ასვლა, უკვე გათვალისწინებული იქნება (i-1)-ე საფეხურზე ასვლის ვარიანტთა რაოდენობაში. მაშასადამე, ადგილი აქვს რეკურენტულ ფორმულას:

$$RAOD(i)=RAOD(i-1)+RAOD(i-2)$$

რადგან პირველ საფეხურზე მხოლოდ ერთი გზით შეიძლება მოხვედრა, ხოლო მეორეზე – ორი გზით (პირდაპირ ორ საფეხურზე ანაცვლებით და პირველი საფეხურიდან ერთ საფეხურზე ანაცვლებით), შეგვიძლია განვსაზღვროთ, რომ RAOD(1)=1 და RAOD(2)=2. ყველა დანარჩენი წევრის გამოსათვლელად კი თანმიმდევრულად შევავსოთ ერთგანზომილებიანი 20-ელემენტიანი მასივი:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765	10946

ნახ. 6.11

შესაბამისი პროგრამული კოდის დაწერა დიდ სირთულეს არ წარმოადგენს:

```
RAOD[1]=1;   RAOD[2]=2;
For  i=3  to  20  {
        RAOD [i]= RAOD[i-1]+RAOD[i-2]    }
```

შევნიშნოთ, რომ მიიღება ფიბონაჩის მიმდევრობა, რომელსაც პირველი წევრი აკლია.

6.7. ზრდადი ქვემიმდევრობა

მოცემულია N-ელემენტიანი ნატურალური რიცხვების მიმდევრობა. იპოვეთ მასში მაქსიმალური სიგრძის მკაცრად ზრდადი ქვემიმდევრობა, რომლის ელემენტების ინდექსებიც ასევე ზრდადობით იქნებიან დალაგებული (აუცილებელი არაა, რომ ეს ელემენტები ერთმანეთის მიყოლებით იყვნენ განლაგებულნი).

შესატანი მონაცემები: პირველ სტრიქონში მოცემულია მიმდევრობის ელემენტთა რაოდენობა – N (2<N<1000). მეორე სტრიქონში მოცემულია ჰარებით დაშორებული N ცალი ნატურალური რიცხვი.

გამოსატანი მონაცემები: მაქსიმალური სიგრძის მქონე ქვემიმდევრობის სიგრძე.

შესატანი მონაცემების მაგალითი:
9
12 1 9 17 19 4 15 22 7

გამოსატანი მონაცემი მოყვანილი მაგალითისათვის:
5

მითითება. ცხადია, რომ მიმდევრობის ნებისმიერი წევრი მის მარცხნივ მყოფ უფრო მცირე წევრთან ერთად ჩვენთვის საინტერესო წყვილს წარმოადგენს, მაგრამ თუკი მარცხნივ ნაკლები მნიშვნელობის რამდენიმე წევრია, მაშინ მოგვიწევს მათ შორის ავარჩიოთ ისეთი, რომელიც უკვე არის აქამდე არსებული ყველაზე გრძელი ქვემიმდევრობის უკანასკნელი წევრი. შემოვიღოთ საწყისი A მასივის ტოლი B მასივი. მის ყოველ ელემენტში ჩავწეროთ იმ მაქსიმალური ქვემიმდევრობის სიგრძე, რომლის უკანასკნელი წევრია A მასივის შესაბამის ნომერზე მდგომი ელემენტი. ამოცანის ამონახსნი იქნება B მასივის უდიდესი ელემენტის მნიშვნელობა.

თავდაპირველად B მასივის ყველა ელემენტს 1 მივანიჭოთ, რადგან მიმდევრობის ყოველი წევრი თავად წარმოადგენს ერთელემენტიან ქვემიმდევრობას. შემდეგ A მასივის ყოველი i-ური ელემენტისათვის განვიხილოთ A მასივში მის მარცხნივ მდებარე ყველა ნაკლები მნიშვნელობის ელემენტი და მათ შორის ამოვარჩიოთ ის, რომლის შესაბამის ელემენტს B მასივში ყველაზე დიდი მნიშვნელობა აქვს. ამორჩეულ მნიშვნელობას დავუმატოთ 1 და ჩავწეროთ B მასივში i-ურ ადგილას. ამ წესით შევსებულ B მასივს ამოცანაში მოყვანილი მონაცემებისთვის ექნება სახე:

A მასივი

1	2	3	4	5	6	7	8	9
12	1	9	17	19	4	15	22	7

B მასივი

1	2	3	4	5	6	7	8	9
1	1	2	3	4	2	3	5	3

ნახ.6.12

მაგალითისათვის განვიხილოთ A მასივის მეშვიდე ელემენტი – 15. მის მარცხნივ მყოფ ელემენტებიდან ის მეტია პირველ, მეორე, მესამე და მეექვსე ელემენტებზე. ამ ოთხ ელემენტს შორის B მასივში ყველაზე მეტი მნიშვნელობა შეესაბამება მესამე და მეექვსე ელემენტებს – 2, ამიტომ B(7)-ში ვწერთ 1-ით მეტს – 3.

მოვიყვანოთ ამ ალგორითმის პროგრამული რეალიზაცია:

```
READ (n,A)
FOR  i=1 TO n {   B[i]= 1 }
FOR  i=2 TO n {
     FOR  j=1 TO i-1 {
       IF ((A[i]>A[j]) AND (B[i]<=B[j])) THEN B[i]=B[j]+1   }     }
max=1
For  i=2 TO N {
     IF B[i] > max  THEN max=B[i]     }
 WRITE (max);
```

6.8. სამკუთხედი

(მოსწავლეთა საერთაშორისო ოლიმპიადა, სტოკჰოლმი, 1994წ.)

ნახაზზე გამოსახულია რიცხვებისაგან აგებული სამკუთხედი. დაწერეთ პროგრამა, რომელიც იპოვის სამკუთხედის ზედა წვეროდან დაწყებულ და მის ფუძეზე დამთავრებულ გზების სიგრძეებს შორის მაქსიმალურს. გზის სიგრძედ ითვლება მასზე განლაგებული რიცხვების ჯამი.

```
                7
             3     8
          8     1     0
       2     7     4     4
    4     5     2     6     5
```

ნახ. 6.13

- ყოველი ბიჯი გზაზე კეთდება დიაგონალურად ქვემოთ და მარცხნივ ან დიაგონალურად ქვემოთ და მარჯვნივ.
- სტრიქონთა რაოდენობა სამკუთხედში მეტია 1-ზე და ნაკლებია ან ტოლია 100-ზე.
- სამკუთხედი შედგება მთელი რიცხვებისაგან 0-დან 99-მდე.

შესატანი მონაცემები: შესატან მონაცემთა INPUT.TXT ფაილის პირველ სტრიქონში სამკუთხედში სტრიქონთა N რაოდენობა იწერება, ხოლო მომდევნო N სტრიქონში – სამკუთხედში შემავალი რიცხვები.

გამოსატანი მონაცემები: გამოსატან მონაცემთა OUTPUT.TXT ფაილის ერთადერთ სტრიქონში იწერება მთელი რიცხვით გამოსახული მაქსიმალური სიგრძე.

მაგალითი:

INPUT.TXT ფაილი

```
5
7
3 8
8 1 0
2 7 4 4
4 5 2 6 5
```

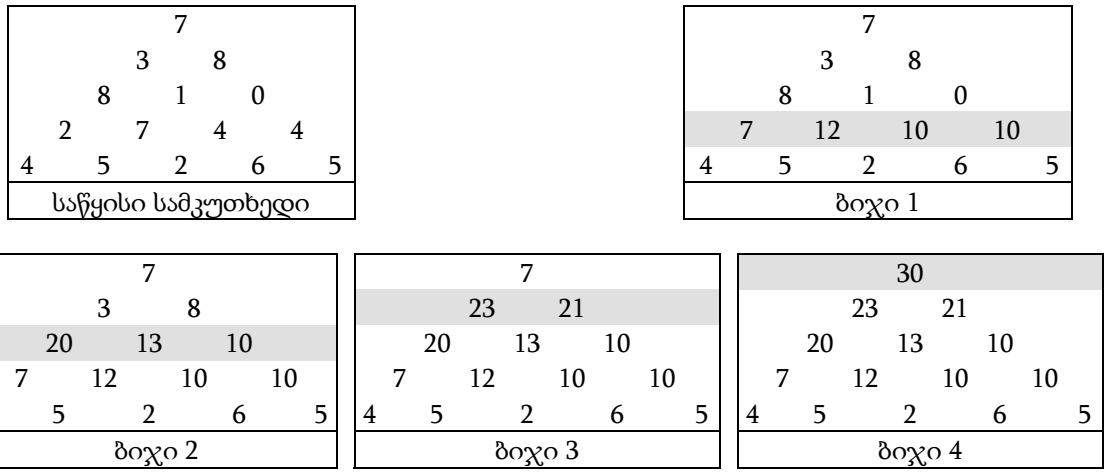
OUTPUT.TXT ფაილი

```
30
```

მითითება. დიდი მოცულობის შემავალი მონაცემებისათვის მათი რეალიზაცია შესაძლებელია ერთგანზომილებიან მასივშიც, მაგრამ ამოცანაში მოცემული პარამეტრებისათვის ორგანზომილებიანი მასივის (ზომით 100×100) გამოყენებაც შეიძლება. მართალია, ასეთ მასივში ელემენტების დიდი ნაწილი nil მნიშვნელობის იქნება (ჩვენი

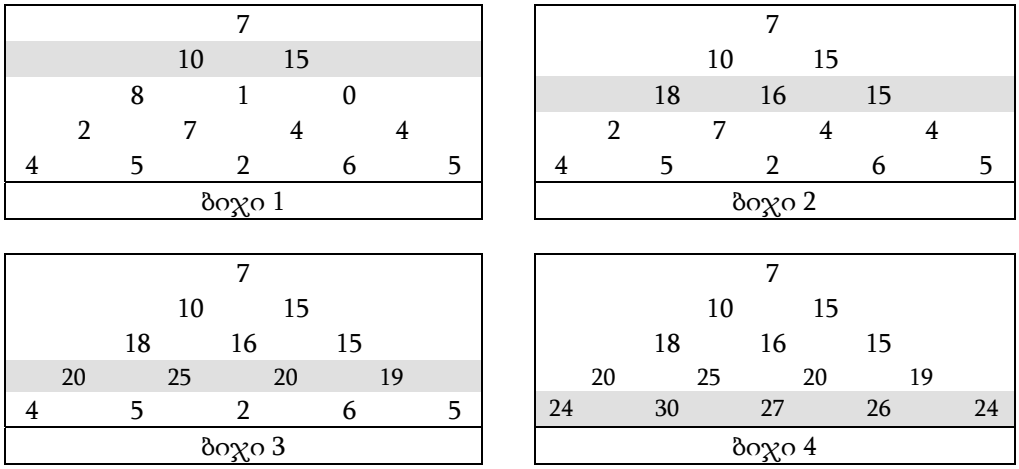
ამოცანისთვის გამოდგებოდა, ვთქვათ, -1) და მანქანურ მეხსიერებას უსარგებლოდ დაიკავებს, სამაგიეროდ თავიდან ავიცილებთ მეზობელი ელემენტების ფორმულებით გამოთვლას, რაც ერთგანზომილებიან მასივშია საჭირო. უშუალოდ სამკუთხა ფორმის მასივების აღწერის საშუალება რომელიმე პროგრამულ ენაზე, სამწუხაროდ, ჩემთვის ცნობილი არ არის.

მოცემული მაგალითისათვის ვთქვათ, ჩვენ უკვე შევარჩიეთ მაქსიმალური გზა პირველ ოთხ სტრიქონში და მხოლოდ მეხუთე სტრიქონში დაგვრჩა რიცხვი ასარჩევი. თუკი ჩვენ ვდგავართ მეოთხე სტრიქონის პირველ ელემენტზე (რიცხვი 2), მაშინ უნდა ავირჩიოთ 4-სა და 5-ს შორის უდიდესი და მისკენ გავაგრძელოთ მოძრაობა, თუკი ვდგავართ მეოთხე სტრიქონის მეორე ელემენტზე (რიცხვი 7), უნდა ავირჩიოთ უდიდესი 5-სა და 2-ს შორის და მასზე გადავიდეთ და ა.შ. ცხადია, რომ ბოლოსწინა სტრიქონიდან სვლის გაკეთებისას ნებისმიერ შემთხვევაში ორ ქვედა მეზობელს შორის უდიდესი უნდა ავირჩიოთ. აქედან გამომდინარეობს, რომ თუკი ბოლოსწინა სტრიქონის თითოეულ ელემენტს წინასწარ დავუმატებთ ორ ქვედა მეზობელთაგან უდიდესს და ბოლო სტრიქონს საერთოდ გავაუქმებთ – მივიღებთ იმავე მაქსიმალური სიგრძის მქონე სამკუთხედს, როგორიც თავდაპირველად გვქონდა მოცემული. თუკი ახალი სამკუთხედისთვისაც გავიმეორებთ იგივე ქმედებას, სამკუთხედის ზომა კიდევ ერთი სტრიქონით შემცირდება და ასე გავაგრძელოთ მანამ, სანამ არ დაგვრჩება მხოლოდ ზედა წვერო, რომელშიც უკანასკნელ ბიჯზე ჩაწერილი რიცხვი წარმოადგენს ჩვენი ამოცანის ამონახსნს.



ნახ. 6.14

ანალოგიურ შედეგს მივიღებდით, თუკი ვიმოძრავებდით სამკუთხედის ზედა წვეროდან ფუძისაკენ შემდეგი პრინციპით: თუ მეორე სტრიქონში მოთავსებულ ელემენტს ჰყავს მხოლოდ ერთი ზედა მეზობელი, ამ უკანასკნელის მნიშვნელობას ვუმატებთ მას, ხოლო თუ ორი ზედა მეზობელი ჰყავს – ვუმატებთ მათ შორის უდიდესს. როცა მეორე სტრიქონის ყველა ელემენტს ასე გარდავქმნით, ვაუქმებთ პირველ სტრიქონს და პროცესს ვიმეორებთ. ნახაზზე მოცემულია ამ ალგორითმის მუშაობა (საწყისი სამკუთხედი იგივეა):



ნახ. 6.15

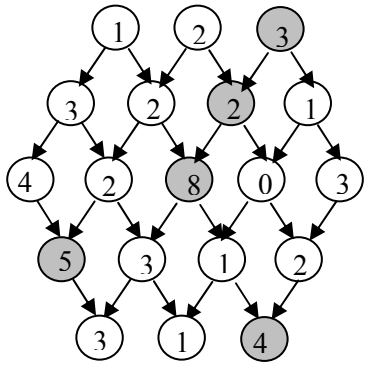
განსხვავებით პირველი ალგორითმისაგან (როცა ფუძიდან წვეროსაკენ ვმოძრაობდით) აქ პირდაპირ პასუხს ვერ ვღებულობთ და საჭიროა მაქსიმალური ელემენტის პოვნა უკანასკნელ სტრიქონში. ორივე ალგორითმში იკარგება მოძრაობის მარშრუტი და მის აღსადგენად საჭიროა შედეგამდე განვლილ გზაზე უკან დაბრუნება და ყოველ ბიჯზე იმის შემოწმება, თუ რომელი წევრიდან მოვედით მოცემულ ელემენტზე.

ანალოგიური ალგორითმით – ყოველ ბიჯზე მინიმალური ელემენტების შეკრებით, შესაძლებელია მინიმალური ჯამის პოვნაც.

ამოცანები იშვიათად არიან დასმული ისეთი სახით, რომ მათი ამოხსნა კონკრეტული მეთოდის სქემით პირდაპირ იყოს იყოს შესაძლებელი. ამოცანის პირობაში ესა თუ ის სქემა ზოგჯერ ნათლად იკითხება, ზოგჯერ მისი შემჩნევა კი საკმაოდ რთულია. ამის მაგალითად გამოდგება შემდეგი ორი ამოცანა, რომლებიც ზემოთ მოყვანილი "სამკუთხედის მეთოდით" იხსნება. პირველ მათგანში ("რიცხვითი ცხრილი") მსგავსება იოლი შესამჩნევია, ხოლო მეორეში ("ყვავილების მაღაზია") – შედარებით რთული.

6.9. რიცხვითი ცხრილი

(ბალტიისპირეთის ოლიმპიადა ინფორმატიკაში მოსწავლეთა შორის, 2000 წელი)



ნახ. 6.16

ნახაზზე ნაჩვენებია ექვსკუთხედის ფორმის რიცხვითი ცხრილი (გვერდის სიგრძე ამ შემთხვევაში 3-ის ტოლია). წრეს, რომელიშიც ჩაწერილია რიცხვი, ვუწოდოთ კვანძი. გზა იწყება ზედა რიგის რომელიმე კვანძიდან და მთავრდება ქვედა რიგის ერთ-ერთ კვანძში. ყოველი კვანძიდან გზა გრძელდება მხოლოდ დიაგონალური მიმართულებით ქვემოთ მარცხნივ ან მარჯვნივ. გზის შედგენისას თქვენ უფლება გაქვთ, მოახდინოთ არაუმეტეს ერთი შენაცვლება ცხრილის არაუმეტეს ერთ სტრიქონში (შენაცვლება ნიშნავს, რომ თქვენს მიერ არჩეულ ერთ რომელიმე რიგში შეგიძლიათ ამ რიგის უდიდესი რიცხვი გადაიტანოთ იგივე რიგის ნებისმიერ კვანძში).

დაწერეთ პროგრამა, რომელიც დათვლის ყველა შესაძლებელი გზების რიცხვების ჯამის უდიდეს მნიშვნელობას (მათ შორის იგულისხმება გზები, რომლებიც მიიღება შენაცვლების გამოყენებითაც).

შეზღუდვები: კვანძებში ჩაწერილია რიცხვები, რომლებიც მოთავსებულია 0 და 99 შორის. რიცხვითი ცხრილის გვერდის სიგრძე არის მთელი 1 და 99 შორის.

შემავალი მონაცემები: შემავალი HON.IN ფაილის პირველი სტრიქონში ჩაწერილია გვერდის სიგრძე. თუ გვერდის სიგრძეა n, მაშინ ცხრილი შედგება 2n-1 ჰორიზონტალური რიგისაგან, რომლებშიც მდგომი რიცხვები შესატან ფაილში აღწერილია პირველი სტრიქონის მომდევნო 2n-1 სტრიქონში.

გამომავალი მონაცემები: HON.OUT ფაილის ერთადერთ სტრიქონში დავბეჭდოთ ჯამის მაქსიმალური მნიშვნელობა.

შემავალი მონაცემები (HON.IN ფაილი)	გამომავალი მონაცემები (HON.OUT ფაილი)
3 1 2 3 3 2 2 1 4 2 8 0 3 5 3 1 2 5 1 4	22

ნახაზზე გამოსახული ცხრილის სწორი ამოხსნაა 3+2+8+5+4=22. ნახაზზე ეს კვანძები გამუქებულია. შევნიშნოთ, რომ მეოთხე რიგში უდიდესი რიცხვი 5 შეიძლება შევანაცვლოთ მესამე კვანძში მდგომ ერთიანთან.

მითითება. ამ ამოცანასა და "სამკუთხედს" შორის მსგავსება აშკარაა. ერთადერთ პრობლემას ჰქმნის ის დებულება, რომ ჩვენ შეგვიძლია სტრიქონის შიგნით ელემენტებისათვის ადგილის შეცვლა. ბუნებრივია, ამ პირობის გამოყენება ყოველთვის მიმართული უნდა იყოს საძებნი ჯამის გაზრდისაკენ, ამიტომ ვიპოვოთ თითოეული სტრიქონის მაქსიმალური ელემენტი და ვიმოქმედოთ შემდეგნაირად:

შევცვალოთ პირველი სტრიქონის ყველა ელემენტი ამავე სტრიქონის მაქსიმალური ელემენტით და "სამკუთხედის" ალგორითმით ვიპოვოთ მაქსიმალური ჯამი ექვსკუთხედში (განსხვავება გეომეტრიულ ფორმაში არაფერს ცვლის, რადგან მოძრაობის პრინციპი არ იცვლება). შემდეგ აღვადგინოთ პირველი სტრიქონი და მეორე სტრიქონის ყველა ელემენტი შევცვალოთ ამავე სტრიქონის მაქსიმუმით, ისევ ვიპოვოთ მაქსიმალური ჯამი ექვსკუთხედში, აღვადგინოთ მეორე სტრიქონი და ასე გავიმეოროთ სათითაოდ ყველა სტრიქონისათვის. სულ "სამკუთხედის მეთოდის" გამოყენება მოგვიწევს იმდენჯერ, რამდენი სტრიქონიცაა საწყის ექვსკუთხედში. მაქსიმუმი მიღებულ მაქსიმალურ ჯამებს შორის იქნება ამოცანის ამონახსნი. სტრიქონის ყველა წევრის შეცვლა მისივე მაქსიმალური ელემენტით ახდენს იმ პირობის რეალიზაციას, რომლითაც შეგვიძლია სტრიქონის შიგნით ელემენტებისათვის ადგილის გაცვლა და რადგან ძნელი შესაფასებელია, თუ რომელ სტრიქონში უნდა მოხდეს ეს პროცედურა, ვიხილავთ ყველა შესაძლო შემთხვევას.

6.10. ყვავილების მაღაზია

(11-ე საერთაშორისო ოლიმპიადა მოსწავლეთა შორის, ანტალია, 1999 წ.)

გაქვთ F რაოდენობის ყვავილების თაიგული, რომელთაგან თითოეული მხოლოდ ერთი სახეობის ყვავილებისაგან შედგება. ყვავილების მაღაზიის ფანჯრის რაფაზე ერთ მწკრივში განლაგებულია V ცალი ცარიელი ლარნაკი, რომლებიც გადანომრილია მარცხნიდან მარჯვნივ მიმდევრობით 1-დან V-მდე. თაიგულებიც ასევე გადანომრილია 1-დან F-მდე ურთიერთგანსხვავებული მთელი რიცხვებით.

თქვენ უნდა მოძებნოთ ყველა თაიგულის ლარნაკებში განლაგების საუკეთესო ვარიანტი.

თუ i<j, მაშინ არ შეიძლება i ნომრის მქონე თაიგულის ჩადება იმ ლარნაკში, რომელიც მდებარეობს უფრო მარჯვნივ, ვიდრე ის ლარნაკი, რომელშიც ჩადებთ j ნომრის მქონე თაიგულს.

მაგალითად: ვთქვათ, თქვენ გაქვთ მიხაკების (ნომერი 1), ვარდების (ნომერი 2) და ყაყაჩოების (ნომერი 3) თაიგულები. ისინი უნდა ჩაალაგოთ ლარნაკებში ისე, რომ დაიცვათ მათი ნომრების მიმდევრობა. ლარნაკი მიხაკების თაიგულით უნდა მდებარეობდეს უფრო მარცხნივ, ვიდრე ლარნაკი ვარდების თაიგულით, ხოლო ეს უკანასკნელი კი – უფრო მარცხნივ, ვიდრე ლარნაკი ყაყაჩოების თაიგულით. თუ ლარნაკების რაოდენობა მეტია თაიგულების რაოდენობაზე, მაშინ ზოგიერთი ლარნაკი ცარიელი დარჩება. თითოეულ ლარნაკში შეიძლება მხოლოდ ერთი თაიგულის ჩადება.

ყოველ ლარნაკს გააჩნია განსხვავებული მახასიათებელი. ამიტომ ლარნაკში რომელიმე თაიგულის ჩადება რაიმე მთელი რიცხვით აღიწერება (იხ. ცხრილი). თუ ლარნაკი ცარიელია, მას შეესაბამება რიცხვი 0.

		ლარნაკები				
		1	2	3	4	5
თაიგულები	1(მიხაკი)	7	23	-5	-24	16
	2(ვარდი)	5	21	-4	10	23
	3(ყაყაჩო)	-21	5	-4	-20	20

ნახ. 6.17

ამ ცხრილის მიხედვით მიხაკების მე-2 ლარნაკში ჩადება ყველაზე დიდი რიცხვით აღიწერება, ხოლო მე-4 ლარნაკში ჩადება – ყველაზე პატარა რიცხვით.

ლარნაკებში თაიგულების ჩალაგების ვარიანტი საუკეთესოდ ჩაითვლება, თუ ამ ჩალაგების აღმწერი რიცხვების ჯამი მაქსიმალური იქნება. თუ არსებობს რამდენიმე საუკეთესო ვარიანტი, უნდა გამოიტანოთ მხოლოდ ერთ-ერთი მათგანი.

შეზღუდვები:

- $1 \le F \le 100$, სადაც F – თაიგულების რაოდენობაა. თაიგულები გადანომრილია 1-დან F-მდე.
- $F \le V \le 100$, სადაც V არის ლარნაკების რაოდენობა.
- $-50 \le A_{ij} \le 50$, სადაც A_{ij} არის რიცხვი, რომლითაც აღიწერება i-ური თაიგულის j-ურ ლარნაკში ჩადება.

შესატანი მონაცემები (ფაილი flower.inp): პირველი სტრიქონი შეიცავ ორ რიცხვს: F და V. შემდეგი F რაოდენობის სტრიქონიდან თითოეული შეიცავს ლარნაკებში თაიგულის ჩადების აღმწერ V რაოდენობის მთელ რიცხვს.

გამოსატანი მონაცემები (ფაილი flower.out):პირველი სტრიქონი უნდა შეიცავდეს საუკეთესო ვარიანტში ლარნაკებში თაიგულების ჩალაგების აღმწერ რიცხვთა ჯამს. მეორე სტრიქონში წარმოდგენილი უნდა იყოს F რაოდენობის რიცხვი, სადაც k-ური რიცხვის მნიშვნელობა გვიჩვენებს იმ ლარნაკის ნომერს, რომელშიც k-ური თაიგულია ჩადებული.

მაგალითი:

flower.inp ფაილი	flower.out ფაილი
3 5	53
7 23 -5 -24 16	2 4 5
5 21 -4 10 23	
-21 5 -4 -20 20	

მითითება. ამოცანა იხსნება სამკუთხედის მეთოდის გამოყენებით. განსხვავება მხოლოდ ის არის, რომ ყურადღება უნდა მიექცეს ლარნაკებისა და თაიგულების რაოდენობას და ნომრებს. მაგალითად, თუ ვიმოძრავებთ ზემოდან ქვემოთ, რიცხვი 7 პირველი სტრიქონიდან მეორე სტრიქონში უნდა მიემატოს მხოლოდ 21-ს, -4-ს და 10-ს. მეორე სტრიქონის მეხუთე პოზიციაში მყოფ რიცხვ 23-სათვის მიმატება არ შეიძლება იმის გამო, რომ ერთი ადგილი მესამე თაიგულისთვისაც უნდა დარჩეს.

6.11. კედლის კარადა

(საქართველოს მოსწავლეთა პირველობა ინფორმატიკაში, ზონური ტური, 2002 წ.)

კედლის დიდ კარადაში, რომელიც შედგება ღია უჯრებისაგან, უჯრები განლაგებულია C რაოდენობის სვეტსა და R რაოდენობის სტრიქონში (სვეტები გადანომრილია მარცხნიდან მარჯვნივ, ხოლო სტრიქონები ქვემოდან ზემოთ). ზოგიერთ უჯრაში მოთავსებულია რაიმე ნივთი. ნებისმიერი უჯრიდან ნივთის ასაღებად საჭიროა კიბის გამოყენება, რომელიც შეიძლება საჭიროებისამებრ დავაგრძელოთ ან დავამოკლოთ. ამასთან, მოცემულ მომენტში (ყოველი გამოყენებისას) კიბე კარადის მხოლოდ ერთ სვეტზე შეგვიძლია მივაყუდოთ. ვთქვათ, კიბე მიყუდებულია i-ურ სვეტზე და მისი თავი ეყრდნობა j-ურ სტრიქონს. მაშინ, კიბეზე ასვლისას ჩვენ შეგვიძლია ნივთები ავიღოთ როგორც i-ური სვეტის 1-დან j-ურ სტრიქონამდე განლაგებული (j-ური სტრიქონის ჩათვლით) ნებისმიერი უჯრიდან, ასევე უშუალოდ მათ მარცხნივ და მარჯვნივ მდებარე მეზობელი უჯრებიდანაც. j-ს მნიშვნელობას ვუწოდოთ ასვლის სიმაღლე.

დაწერეთ პროგრამა, რომელიც კარადის უჯრებში ნივთების მოცემული განლაგებისათვის იპოვის ყველა ნივთის ასაღებად საჭირო ასვლათა სიმაღლეების მინიმალურ ჯამს.

შესატანი მონაცემები (SHEVLES.IN): პირველი სტრიქონი შეიცავს ორ მთელ C და R რიცხვს ($1 \le C \le 100$, $1 \le R \le 100$) – სვეტებისა და სტრიქონების რაოდენობას. მეორე სტრიქონში ჩაწერილია მთელი N რიცხვი ($1 \le N \le 100$) – კარადის უჯრებში განლაგებული ნივთების რაოდენობა. დანარჩენი N რაოდენობის სტრიქონიდან თითოეულში მოცემულია ორი მთელი A და B რიცხვი ($1 \le A \le C$, $1 \le B \le R$) – კარადის იმ უჯრების კოორდინატები, რომლებშიც ნივთებია განლაგებული. სტრიქონებში მონაცემები ერთმანეთისაგან თითო ჰარით არის გამოყოფილი.

გამოსატანი მონაცემები (ფაილი SHEVLES.OUT): ერთადერთ სტრიქონში უნდა ჩაიწეროს ერთი მთელი რიცხვი – ყველა ნივთის ასაღებად საჭირო ასვლათა სიმაღლეების მინიმალური ჯამი.

მაგალითები:

ფაილი SHEVLES.IN	ფაილი SHEVLES.IN	ფაილი SHEVLES.IN
5 5	6 20	10 10
3	4	5
2 3	5 6	9 1
3 4	1 1	7 6
4 4	6 1	5 8
	3 7	4 1

		3 2
ფაილი SHEVLES.OUT	ფაილი SHEVLES.OUT	ფაილი SHEVLES.OUT
4	9	11

მითითება. ამოხსნის იდეა მდგომარეობს იმაში, რომ თუ k სვეტისათვის ოპტიმალური სვლების მიმდევრობა ნაპოვნია, ახალი (k+1)-ე სვეტის დამატებისას საჭირო აღარ არის მთლიანად ყველა სვეტისათვის მინიმუმის თავიდან გადაანგარიშება – საკმარისია, მხოლოდ ბოლო სამი სვეტის განხილვა.

შემოვიღოთ ორი მასივი: sv *– რომელშიც სვეტის შესაბამის ინდექსზე ჩაწერილი იქნება ამ სვეტში მოთავსებული ნივთის სიმაღლე;* rez *– რომელშიც ჩაიწერება ამოცანის ამონახსნი შესაბამის ინდექსამდე. მუშაობისას დაგვჭირდება ფუნქცია MAX, რომელიც გამოითვლის მაქსიმუმის მნიშვნელობას მიწოდებულ არგუმენტთა შორის. თავდაპირველად შევავსებთ* rez *მასივის პირველ სამ წევრს:*

rez[1]=sv[1]; rez[2]=max(sv[1],sv[2]); rez[3]=max(sv[1],sv[2],sv[3]);

რაც ნიშნავს, რომ თუ 1 სვეტი გვაქვს, ავდივართ ამ სვეტში, თუ ორი სვეტი გვაქვს, ავდივართ ერთ-ერთში ამ ორს შორის მაქსიმალურ სიმაღლეზე და თუ სამი სვეტი გვაქვს ავალთ შუა სვეტში ამ სამთაგან მაქსიმალურის სიმაღლეზე. ამის შემდეგ ციკლში განვიხილავთ ყოველ ახალ სვეტს წინა სამთან კომბინაციაში:

```
FOR i=4 TO c    {
    rez[i]=+∞
    IF sv[i]+rez[i-1]<rez[i] THEN rez[i]=sv[i]+rez[i-1]
    IF max(sv[i],sv[i-1])+rez[i-2]<rez[i] THEN rez[i]=max(sv[i],sv[i-1])+rez[i-2]
    IF max(sv[i],sv[i-1],sv[i-2])+rez[i-3]<rez[i] THEN rez[i]=max(sv[i],sv[i-1],sv[i-2])+rez[i-3]   }
```

ციკლის დასრულების შემდეგ ამოცანის პასუხი მოთავსებული იქნება rez *მასივის n-ურ წევრში.*

6.12. პალინდრომი

(უკრაინის online-ოლიმპიადა, 2002-03 წლები)

პალინდრომი წარმოადგენს სიტყვას, რომელიც ორივე მხრიდან ერთნაირად იკითხება. დაწერეთ პროგრამა, რომელიც პალინდრომად გადააქცევს ნებისმიერ სიტყვას, მისგან მინიმალური რაოდენობის სიმბოლოთა ამოშლით. სიტყვად ჩაითვლება ლათინური ანბანის პატარა სიმბოლოების მიმდევრობა.

შესატანი მონაცემები: ლათინური ანბანის პატარა სიმბოლოების მიმდევრობა (არაუმეტეს 255-ისა).

გამოსატანი მონაცემები: ერთადერთი რიცხვი – იმ სიმბოლოთა მინიმალური რაოდენობა, რომელთა წაშლაც საჭიროა სიტყვის პალინდრომად გადაქცევისათვის.

მაგალითები:

qwerrewtq	qwert
1	4

მითითება: განვიხილოთ ყველა ქვესტრიქონი მათი სიგრძის ზრდადობის მიხედვით. თუ ქვესტრიქონი შედგება მხოლოდ ერთი სიმბოლოსაგან, მაშინ ის პალინდრომს წარმოადგენს და წასაშლელ სიმბოლოთა რაოდენობა 0-ის ტოლია.

განვიხილოთ S ქვესტრიქონი, რომლის სიგრძე 1-ზე მეტია. ვთქვათ, ამ ქვესტრიქონის სიგრძეა L და ვთქვათ ჩვენთვის ცნობილია წასაშლელ სიმბოლოთა მინიმალური რაოდენობა L-ზე ნაკლები ან ტოლი სიგრძის მქონე ყველანაირი ქვესტრიქონისათვის. თუკი S ქვესტრიქონს დავუმატებთ ერთ სიმბოლოს ბოლოში, მაშინ შეგვიძლია განვიხილოთ შემდეგი ვარიანტები: ა) ახალი სიმბოლო ემთხვევა პირველს – ამ შემთხვევაში მომგებიანია ეს სიმბოლოები ხელუხლებლად დავტოვოთ და განვიხილოთ ქვესტრიქონი 2-დან L-მდე, რომლისთვისაც დაშვების თანახმად პასუხი უკვე ცნობილია (ქვემოთ მოცემულ ნახაზზე ეს რიცხვი ჩაწერილია განსახილველი უჯრედის დიაგონალურად); ბ) ახალი სიმბოლო პირველს არ ემთხვევა – მაშინ პალინდრომის მისაღებად ამ ორი სიმბოლოსაგან ერთ-ერთი მაინც უნდა წაიშალოს. ნებისმიერი მათგანის წაშლისას დაგვრჩება ქვესტრიქონი (1-დან L-მდე ან 2-დან (L+1)-მდე),

რომლისთვისაც პასუხი უკვე ვიცით (ნახაზზე ეს რიცხვები ჩაწერილია მოცემული უჯრედის მარცხნივ ან ქვემოთ). მაშასადამე, ახალი ქვესტრიქონისათვის ხსენებული სამი ვარიანტიდან საუკეთესო უნდა შევარჩიოთ.

მოცემული სტრიქონის ყოველი ქვესტრიქონი ცალსახად განისაზღვრება მისი დასაწყისისა და ბოლოს ნომრებით, ამიტომ შეგვიძლია შევქმნათ n×n ზომის ორგანზომილებიანი მასივი, რომელშიც ჩავწერთ შედეგს შესაბამისი ქვესტრიქონისათვის.

განვიხილოთ მაგალითი სიტყვისათვის 'abracadabra'. 6.18ა ნახაზზე მოცემულია მასივის საწყისი მდგომარეობა. მასივის მარცხნივ მოთავსებული სვეტი განსაზღვრავს ქვესტრიქონების დასაწყისს, ხოლო მასივის ზემოთ მოთავსებული სტრიქონი – ბოლოს. შევსება ხდება მთავარი დიაგონალის პარალელურად, ხოლო შედეგი მიიღება ზედა მარჯვენა კუთხეში. შევსება იწყება 2 სიგრძის მქონე ქვესტრიქონებიდან. თუკი ასეთ ქვესტრიქონში სიმბოლოები ერთნაირია, შესაბამის უჯრაში იწერება 0, ხოლო თუ განსხვავებულია – 1. ჩვენს მაგალითში მეზობელი სიმბოლოების არცერთი წყვილი არ შედგება ერთნაირი სიმბოლოებისაგან, ამიტომ მათგან პალინდრომის მისაღებად ერთ-ერთი უნდა წაიშალოს, ე.ი. ყველგან ჩაიწერება 1-იანი. ამის შემდეგ განიხილება 3 სიგრძის მქონე ქვესტრიქონები. თუკი პირველი და მესამე სიმბოლოები ერთმანეთს ემთხვევიან, საქმე პალინდრომთან გვქონია, წაშლა საჭირო არ არის და შესაბამის უჯრედში 0-ს ვწერთ. ჩვენს მაგალითში ასეთი სამეული ორია: 'aca' 'და ada', ხოლო ყველა სხვა შემთხვევაში ქვესტრიქონებში შემავალი სამივე სიმბოლო განსხვავებულია და მათგან პალინდრომის მისაღებად მინიმუმ 2 სიმბოლოს წაშლაა საჭირო. ნახ. 6.18ბ-ზე ნაჩვენებია მასივის მდგომარეობა 1, 2 და 3 სიგრძის მქონე ქვესტრიქონების განხილვის შემდეგ.

		1	2	3	4	5	6	7	8	9	10	11
		a	**b**	**r**	**a**	**c**	**a**	**d**	**a**	**b**	**r**	**a**
1	**a**	0										
2	**b**	0	0									
3	**r**	0	0	0								
4	**a**	0	0	0	0							
5	**c**	0	0	0	0	0						
6	**a**	0	0	0	0	0	0					
7	**d**	0	0	0	0	0	0	0				
8	**a**	0	0	0	0	0	0	0	0			
9	**b**	0	0	0	0	0	0	0	0	0		
10	**r**	0	0	0	0	0	0	0	0	0	0	
11	**a**	0	0	0	0	0	0	0	0	0	0	0

(ა)

		1	2	3	4	5	6	7	8	9	10	11
		a	**b**	**r**	**a**	**c**	**a**	**d**	**a**	**b**	**r**	**a**
1	**a**	0	1	2								
2	**b**	0	0	1	2							
3	**r**	0	0	0	1	2						
4	**a**	0	0	0	0	1	0					
5	**c**	0	0	0	0	0	1	2				
6	**a**	0	0	0	0	0	0	1	0			
7	**d**	0	0	0	0	0	0	0	1	2		
8	**a**	0	0	0	0	0	0	0	0	1	2	
9	**b**	0	0	0	0	0	0	0	0	0	1	2
10	**r**	0	0	0	0	0	0	0	0	0	0	1
11	**a**	0	0	0	0	0	0	0	0	0	0	0

(ბ)

		1	2	3	4	5	6	7	8	9	10	11
		a	**b**	**r**	**a**	**c**	**a**	**d**	**a**	**b**	**r**	**a**
1	**a**	0	1	2	1	2	3	4	3	4	5	4
2	**b**	0	0	1	2	3	2	3	4	3	4	5
3	**r**	0	0	0	1	2	1	2	3	4	3	4
4	**a**	0	0	0	0	1	0	1	2	3	4	3
5	**c**	0	0	0	0	0	1	2	1	2	3	4
6	**a**	0	0	0	0	0	0	1	0	1	2	3
7	**d**	0	0	0	0	0	0	0	1	2	1	2
8	**a**	0	0	0	0	0	0	0	0	1	2	1
9	**b**	0	0	0	0	0	0	0	0	0	1	2
10	**r**	0	0	0	0	0	0	0	0	0	0	1
11	**a**	0	0	0	0	0	0	0	0	0	0	0

(გ)

ნახ. 6.18.

ნახ. 6.18გ-ზე მოცემულია მასივის საბოლოო მდგომარეობა, ხოლო პასუხს მოცემულია მაგალითისათვის წარმოადგენს 4, რომელიც წერია პირველი და ბოლო სიმბოლოების გადაკვეთაზე.

6.13. მრგვალი ფრჩხილები

(ბალტიისპირეთის ოლიმპიადა ინფორმატიკაში მოსწავლეთა შორის, 1999 წელი)

ვთქვათ, X არის მრგვალი ფრჩხილებისაგან კორექტულად შედგენილი გამოსახულება. X-ის ელემენტებს წარმოადგენენ მხოლოდ '(' და ')' სიმბოლოები. X კრებული ასე განისაზღვრება:

ცარიელი სტრიქონი ეკუთვნის X-ს.

თუ A ეკუთვნის X-ს, მაშინ (A) ეკუთვნის X-ს.

თუ A და B ეკუთვნიან X-ს, მაშინ კონკატენაცია AB ეკუთვნის X-ს.

მაგალითად, შემდეგი სტრიქონები წარმოადგენენ მრგვალი ფრჩხილებისაგან კორექტულად ფორმირებულ გამოსახულებებს (და ამის გამო ეკუთვნიან X კრებულს):

() (()) ()

((() (())))

შემდეგი სტრიქონები წარმოადგენენ მრგვალი ფრჩხილებისაგან არაკორექტულად ფორმირებულ გამოსახულებებს (და ამის გამო არ ეკუთვნიან X კრებულს):

((())) (()

()) (()

ვთქვათ, E არის მრგვალი ფრჩხილებისაგან კორექტულად შედგენილი გამოსახულება (შესაბამისად ეკუთვნის X კრებულს). E-ს სიგრძე ვუწოდოთ მასში შემავალი მრგვალი ფრჩხილების საერთო რაოდენობას.

E-ს სიღრმე D(E) განისაზღვრება შემდეგნაირად:

D(E)=0, თუ E ცარიელია.

D(E)=D(A)+1, თუ E=(A) და A ეკუთვნის X-ს.

D(E)=max(D(A),D(B)), თუ E=AB და A და B ეკუთვნიან X-ს.

მაგალითად გამოსახულება “() (()) ()”-ის სიგრძეა 8, ხოლო სიღრმე – 2.

n სიგრძისა და d სიღრმის მქონე რამდენი კორექტული გამოსახულება შეიძლება აიგოს მრგვალი ფრჩხილებისაგან მოცემული მთელი დადებითი n და d რიცხვებისათვის?

ამოცანა: დაწერეთ პროგრამა, რომელიც: წაიკითხავს ორ მთელ n და d რიცხვს PAR.IN ფაილიდან; გამოთვლის კორექტულად ფორმირებულ n სიგრძისა და d სიღრმის ყველა შესაძლო გამოსახულების რაოდენობას; ჩაწერს შედეგს ტექსტურ PAR.OUT ფაილში.

შემავალი მონაცემები: შემავალი PAR.IN ფაილი ერთადერთ სტრიქონი შეიცავს ჰარით გაყოფილ ორ მთელ n და d რიცხვს. 2<n<38 და 1<d<19.

გამომავალი მონაცემები: გამომავალი PAR.OUT ფაილის ერთადერთი სტრიქონი უნდა შეიცავდეს ერთ მთელ რიცხვს – კორექტულად ფორმირებულ n სიგრძისა და d სიღრმის ყველა შესაძლო გამოსახულების რაოდენობას.

მაგალითი

შემავალი მონაცემები (PAR.IN ფაილი)	გამომავალი მონაცემები (PAR.OUT ფაილი)
6 2	3

სულ არსებობს 6 სიგრძისა და 2 სიღრმის მქონე გამოსახულების 3 ვარიანტი:

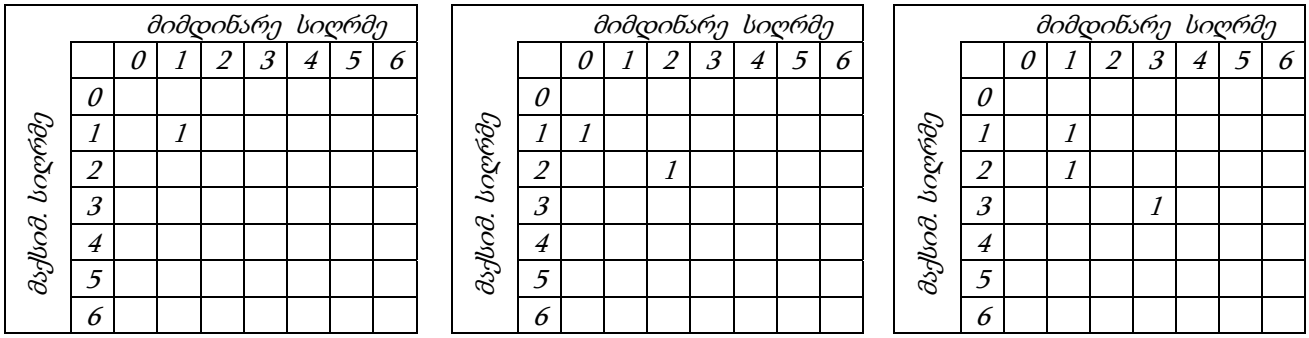
(()) () () (()) (() ())

მითითება. ამოცანის ამოხსნისას უნდა გამოვიყენოთ პირველ თავში ამოხსნილი ამოცანა ფრჩხილთა მიმდევრობის კორექტულობის შესახებ. მარცხენა ფრჩხილი აღვნიშნოთ 1-ით, მარჯვენა კი – -1-ით. ამოცანის პირობაში მოყვანილი სამი ამონახსნი ჩვენი აღნიშვნებით ასე ჩაიწერება:

{1,1,-1,-1,1,-1}, {1,-1,1,1,-1,-1}, {1,1,-1,1,-1,-1}

ცხადია, რომ კორექტული გამოსახულების საერთო ჯამი 0-ის ტოლი იქნება. შემოვიღოთ ასეთი განსაზღვრება: ა) n სიგრძის გამოსახულების მიღებამდე ყოველ კონკრეტულ ბიჯზე გამოსახულების საერთო ჯამს ვუწოდოთ მიმდინარე სიღრმე. მაგალითად პირველი ამონახსნის მიმდინარე სიღრმეები ასე იცვლება 1, 2, 1, 0, 1,0. ცხადია, რომ მიმდინარე სიღრმეებიდან უდიდესი წარმოადგენს გამოსახულების სიღრმეს. ახალი ფრჩხილის დამატებისას მიმდინარე სიღრმე არ უნდა გახდეს უარყოფითი,

შემოვიღოთ A და B მასივები [0..19,0..19] განზომილებით. ამ მასივების სვეტის ნომრები აღნიშნავენ მიღებული გამოსახულების მიმდინარე სიღრმეს, ხოლო სტრიქონის ნომრები – მიმდინარე სიღრმეებიდან უდიდესს. გამოსახულება ყოველთვის იწყება მარცხენა ფრჩხილით. რადგან “(“ გამოსახულების მიმდინარე და მაქსიმალური სიღრმეები წარმოადგენს 1-ს, მათ გადაკვეთაზე, ანუ A[1,1] ელემენტში ჩავწეროთ 1. n სიგრძის გამოსახულების ფორმირებამდე ყოველ ბიჯზე ჩვენ შეგვიძლია დავუმატოთ ან მარცხენა, ან მარჯვენა ფრჩხილი, ანუ მიმდინარე სიღრმეს დავუმატოთ 1 ან გამოვაკლოთ 1. მიმდინარე სიღრმის ცვლილების მიხედვით ჩანაწერები გადავანაცვლოთ სვეტებში, ხოლო თუ მიმდინარე სიღრმე გადააჭარბებს მაქსიმალურ სიღრმეს – ჩანაწერი გადავაადგილოთ სტრიქონებში. ამოცანაში ნაჩვენები მაგალითისათვის A მასივი ასე შეივსება:



ნახ. 6.19.

გამოთვლა შეწყდება მაშინ, როცა n მიაღწევს ამოცანის პირობით განსაზღვრულ მნიშვნელობას (ჩვენს შემთხვევაში 6-ს). პასუხი ჩაწერილი იქნება (d,0) ელემენტში (ჩვენი მაგალითისათვის ესაა ელემენტი (2,0), რომლის მნიშვნელობაა 3). შევნიშნოთ, რომ ამ ცხრილებში პრაქტიკულად მოცემულია ყველა პასუხი კონკრეტული n-სათვის. თუ n კენტია, კორექტული გამოსახულების შედგენა შეუძლებელია, ამიტომ კენტი n-ებისათვის 0-ვანი სვეტი ყოველთვის ცარიელია. მეტი თვალსაჩინოებისათვის ვაჩვენოთ, თუ რომელ გამოსახულებებს შეესაბამებიან ცხრილის ჩანაწერები თითოეულ ბიჯზე:

ბიჯი 1 – "("; ბიჯი 2 – "()" და"(("; ბიჯი 3 – "()(", "(()" და "(((". შევნიშნოთ, რომ ამავე ბიჯზე მიიღებოდა გამოსახულება "())", მაგრამ არაკორექტულობის გამო, მის შემდგომ განხილვას აზრი არა აქვს და ამიტომ ცხრილში ის და სხვა ანალოგიური გამოსახულებები არ აისახება; ბიჯი 4 – "()()", "()((", "(())", "(()(", "((()","((((". ამ ბიჯზე ორ განსხვავებულ გამოსახულებას (მეორეს და მეოთხეს), მიმდინარე და მაქსიმალური სიღრმეები ტოლი აქვთ, ამიტომ შესაბამის ელემენტში ჩაიწერება 2; ბიჯი 5 – "()()(", "()(()","()(((", "(())(", "(()()","(()((", "((())","((()(","((((()","(((((". აქ უკვე მეორე, მეოთხე და მეხუთე გამოსახულებებს აქვთ ერთნაირი მახასიათებლები და მომდევნო ბიჯზე სწორედ მათგან მიიღება ამოცანის პასუხი.

პროგრამის გამართვისას საჭიროა სულ ოთხი შემთხვევის გათვალისწინება ა) -1-ის დამატებისას ჯამი გახდა უარყოფითი (ეს შეიძლება მხოლოდ მარჯვენა ფრჩხილის დამატებისას მოხდეს) – ასეთ შემთხვევაში ცხრილში ცვლილება არ შეგვაქვს; ბ) -1-ის დამატებისას ჯამი დარჩა არაუარყოფითი, მაშინ მოცემული ელემენტის მნიშვნელობა დაემატება იმავე სტრიქონში ერთი ერთეულით მარცხნივ მყოფ ელემენტს, ხოლო მოცემული ელემენტი განულდება; გ) 1-ის დამატებისას მიმდინარე სიღრმემ ვერ გადააჭარბა მაქსიმალურ სიღრმეს, მაშინ მოცემული ელემენტის მნიშვნელობა დაემატება იმავე სტრიქონში ერთი ერთეულით მარჯვნივ მყოფ ელემენტს, ხოლო მოცემული ელემენტი განულდება; დ) 1-ის დამატებისას მიმდინარე სიღრმემ გადააჭარბა მაქსიმალურ სიღრმეს, მაშინ მოცემული ელემენტის მნიშვნელობა დაემატება მომდევნო (ქვედა) სტრიქონში ერთი ერთეულით მარჯვნივ მყოფ ელემენტს, ხოლო მოცემული ელემენტი განულდება.

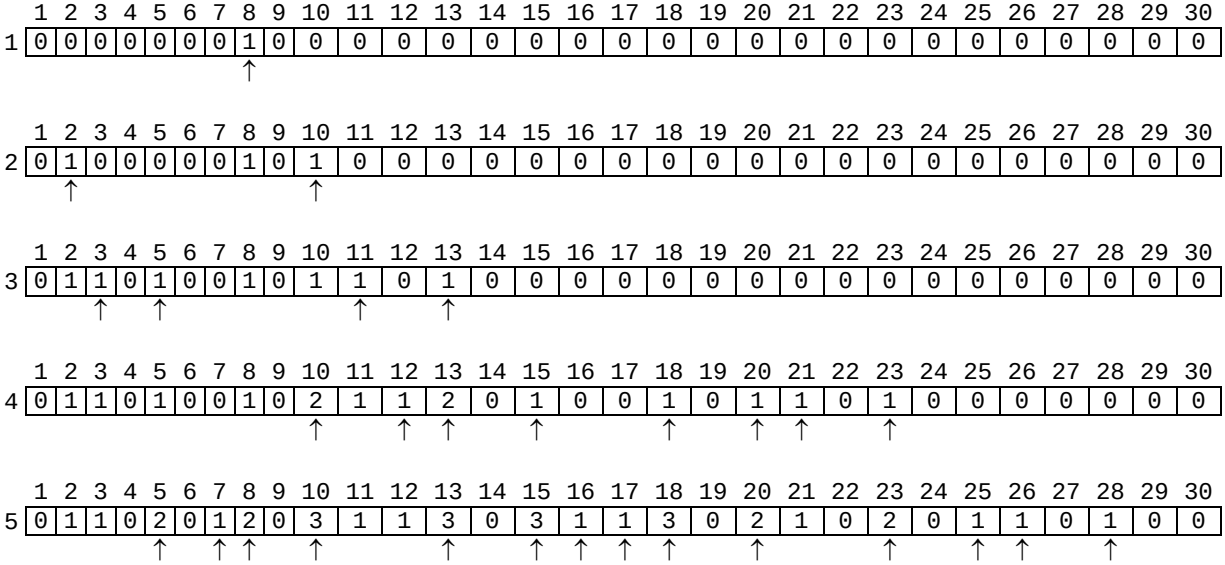
მიხედავად იმისა, რომკენტ და ლუწ ბიჯებზე ახალი მნიშვნელობები შესაბამისად განსხვავებულ სვეტებში ჩაიწერებიან, ალბათ მაინც უმჯობესია ორი მასივი გამოვიყენოთ. წავიკითხოთ A მასივი, ხოლო ახალი მნიშვნელობები ჩავწეროთ B მასივში. შემდეგ B გადავწეროთ A-ში და B გავანულოთ.

```
READ (n,d)
A[1,1]=1;
FOR i=1 to n-1  {
   FOR k=0 to d  {
        FOR j=0 to d {
            IF A[k,j]<>0 THEN {
                IF (j+1>k)  AND (j<=n-i) THEN B[k+1,k+1]=B[k+1,k+1]+A[k,j]
                IF (j+1<=k) AND (j<=n-i) THEN B[k,j+1]=B[k,j+1]+A[k,j]
                IF j-1>=0 THEN B[k,j-1]=B[k,j-1]+A[k,j]      }
}       }
A=B;    B=0
WRITE (A[d,0]);
```

ამოცანა 6.14: ვთქვათ, მოცემულია ხუთი ნატურალური რიცხვი: 8, 2, 3,10, 5. დავწეროთ პროგრამა, რომელიც გამოითვლის, თუ რამდენნაირი განსხვავებული რიცხვის მიღება შეიძლება მათი შეკრებით და რომელი რიცხვები მიიღება ყველაზე მეტი გზით. მოცემული რიცხვების მნიშვნელობებიც ითვლება განსხვავებულ ვარიანტებად.

ამოცანის ამოსახსნელად საჭიროა ერთგანზომილებიანი ისეთი სიდიდის მასივის შემოღება, რომელიც აღემატება მოცემული რიცხვების საერთო ჯამს. ჩვენს შემთხვევაში დავკმაყოფილდეთ 30-ელემენტიანი რიცხვითი A მასივით, რომლის ყველა ელემენტი თავიდან 0-ის ტოლია. ვიმოქმედოთ შემდეგნაირად: სათითაოდ ავიღოთ მოცემული რიცხვები და გადავამოწმოთ A მასივი ბოლოდან დასაწყისისაკენ. თუკი A მასივის რომელიმე ელემენტი 0-ისგან განსხვავებულია, მის ინდექსს დავუმატოთ აღებული რიცხვი და მიღებული შედეგის შესაბამის ინდექსზე მყოფი ელემენტი გავზარდოთ ნაპოვნი 0-საგან განსხვავებული ელემენტის მნიშვნელობით. მასივის სათავეში მისვლის შემდეგ კი თავად ამ რიცხვის შესაბამის ინდექსზეც მყოფი ელემენტი გავზარდოთ1-ით. შემდეგ ავიღოთ მომდევნო რიცხვი და ა.შ.

ნახაზზე მოცემულია A მასივის მდგომარეობა ბიჯების მიხედვით:



ნახ. 6.20.

ისრით მითითებულია, თუ რომელი ელემენტების მნიშვნელობები გაიზარდა შესაბამის ბიჯზე. საბოლოოდ, 0-საგან განსხვავებული ელემენტების რაოდენობა წარმოადგენს პასუხს კითხვაზე, თუ რამდენნაირი განსხვავებული ჯამის მიღებაა შესაძლებელი, ხოლო მასივში მაქსიმალური მნიშვნელობის ელემენტთა ინდექსები გვაძლევენ რიცხვებს, რომელთა მიღებაც ყველაზე მეტი ვარიანტით ხდება.

ალგორითმის სწრაფი მუშაობისათვის მიზანშეწონილია ყოველ ბიჯზე დავიმახსოვროთ 0-საგან განსხვავებული მაქსიმალური ელემენტი, ანუ მაქსიმალური ჯამი (ჩვენს ამოცანაში ბიჯების შესაბამისად 8,10,13,23,28). ეს საშუალებას მოგვცემს მომდევნო ბიჯზე სწორედ ამ ელემენტიდან დავიწყოთ 0-საგან განსხვავებული ელემენტების ძებნა და თავიდან ავიცილოთ მასივში მის მარჯვნივ მდგომი წევრების უსარგებლო შემოწმება.

ერთ-ერთ ყველაზე ცნობილ ამოცანას, რომელიც დინამიური პროგრამირების მეთოდით იხსნება, წარმოადგენს ე.წ. **ზურგჩანთის ამოცანა**. განასხვავებენ ზურგჩანთის **დისკრეტულ** და **უწყვეტ** ამოცანებს (მათი შედარება უფრო დეტალურად მოყვანილია 6.2 თავში). დინამიური მეთოდი გამოიყენება ზურგჩანთის დისკრეტული ამოცანის ამოხსნისას, რომელიც თავის მხრივ პირობის ჩამოყალიბების თვალსაზრისით, შეიძლება დავყოთ ორ ტიპად: წონიანი და უწონო ამოცანა. ამოხსნის მეთოდი ორივე შემთხვევაში ერთნაირია, ოღონდ წონიანი ამოცანის შემთხვევაში დამატებითი მასივია საჭირო. სიმარტივისათვის ჯერ განვიხილოთ უწონო ამოცანა.

ამოცანა 6.15 (ზურგჩანთის დისკრეტული ამოცანა). მოცემულია 5 საგანი, რომელთა ნაწილებად დაყოფა შეუძლებელია. ცნობილია თითოეული საგნის მასა – i-ურ საგანს M_i მასა შეესაბამება. ჩვენი ამოცანაა დავადგინოთ, შესაძლებელია თუ არა ამ საგნებიდან რამდენიმეს ამორჩევა ისე, რომ ამორჩეული საგნების ჯამური მასა ზუსტად 16კგ იყოს. თუკი საგანთა ასეთი სიმრავლე არსებობს, საჭიროა დავადგინოთ მასში შემავალი საგნების სია.

საგანთა მასები შევინახოთ M მასივში. T-თი აღვნიშნოთ ფუნქცია, რომლის მნიშვნელობა 1-ის ტოლია, თუკი საძებნი სიმრავლე მოიძებნება და 0-ის ტოლია – თუკი ასეთი სიმრავლე არ არსებობს. ამ ფუნქციის არგუმენტების იქნებიან საგნების რაოდენობა და სიმრავლის ჯამური მასა. ჩვენი ამოცანაა, ვიპოვოთ T(5,16)-ის მნიშვნელობა. განვსაზღვროთ T(i,j) ქვეამოცანები, სადაც i აღნიშნავს იმ საგნების რაოდენობას, საიდანაც უნდა გავაკეთოთ არჩევანი, ხოლო j შესაბამისი სიმრავლის საპოვნ ჯამურ მასას. T ფუნქციის საწყისი მნიშვნელობები იქნება:

T(0,j)=0, როცა j≥1 – საგნების გარეშე j>0 მასის მიღება შეუძლებელია;

T(i,0)=i, როცა i≥1 – ნულოვანი მასის მიღება ყოველთვის შესაძლებელია.

განვსაზღვროთ T(i,j) ფუნქციის მნიშვნელობები არგუმენტთა არანულოვანი მნიშვნელობებისათვის. ქვეამოცანის ამოხსნა შესაბამისი T(i,j) ფუნქციისათვის შეიძლება დავიყვანოთ ორი შემთხვევის განხილვაზე: ავიღოთ i-ური საგანი სიმრავლეში, თუ – არა. თუკი საგანს არ ვიღებთ, ამოცანის ამოხსნა i-ური საგნის შემთხვევაში დაიყვანება i-1 საგნისაგან შედგენილი სიმრავლის შესახებ ქვეამოცანის ამოხსნაზე, ანუ T(i,j)=T(i-1,j). ხოლო თუკი საგანს ვიღებთ, მაშინ ჯამური მასა მცირდება M[i] სიდიდით, ანუ T(i,j)= T(i-1,j-M[j]). აქვე შევნიშნოთ, რომ მეორე შემთხვევა შესაძლებელია მხოლოდ მაშინ, როცა i-ური საგნის მას არ აღემატება j-ს მნიშვნელობას. ამონახსნის მისაღებად საჭიროა ამ ორ შემთხვევას შორის საუკეთესოს ამორჩევა, ამიტომ რეკურენტული დამოკიდებულებები i≥1 და j≥1 შეიძლება ასე ჩაიწეროს:

T(i,j)=T(i-1,j), როცა j<M[i];

T(i,j)=max(T(i-1,j),T(i-1,j-M[i])), როცა j≥M[i].

ვთქვათ, საგნებს გააჩნიათ შემდეგი მასები: M[1]=4; M[2]=5; M[3]=3; M[4]=7; M[5]=6. მაშინ T ფუნქციის მნიშვნელობების მასივი (რომელიც ასევე T-თი აღვნიშნოთ), ასეთი იქნება:

i / j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0
3	1	0	0	1	1	1	0	1	1	1	0	0	1	0	0	0	0
4	1	0	0	1	1	1	0	1	1	1	1	1	1	0	1	1	1
5	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

ნახ. 6.21

ცხრილიდან ჩანს, რომ T(5,16)=1 და მაშასადამე, სიმრავლე არსებობს. სიმრავლეში შემავალი საგნების საპოვნელად ასე მოვიქცეთ: განვიხილოთ ელემენტები T(5,16) და T(4,16). რადგან მათი მნიშვნელობები ტოლია, ეს ნიშნავს, რომ 16კგ-ს მიღება შესაძლებელია პირველი ოთხი საგნის საშუალებით, ე.ი. მე-5 საგანი სიმრავლეში შეგვიძლია არ ჩავრთოთ. ახლა განვიხილოთ ელემენტები T(4,16) და T(3,16). მათი მნიშვნელობები ტოლი არაა, ამიტომ მე-4 საგანი სიმრავლეში აუცილებლად უნდა ჩავრთოთ, ხოლო დარჩენილი საჭირო მასა ტოლი იქნება – 16-M(4)=16-7=9. განვიხილოთ ელემენტები T(3,9) და T(2,9). მათი მნიშვნელობები

ტოლია, ამიტომ მე-3 საგანი სიმრავლეში არ უნდა ჩავრთოთ. შემდეგ განვიხილოთ T(2,9) და T(1,9). მათი მნიშვნელობები განსხვავებულია, ამიტომ მე-2 საგანი აუცილებლად უნდა ჩავრთოთ სიმრავლეში, ხოლო დარჩენილი საჭირო მასა ტოლი იქნება – 16-M(2)=9-5=4-ის ტოლი. ბოლოს განვიხილავთ T(1,4) და T(0,4). მათი მნიშვნელობები განსხვავებულია, ამიტომ პირველი საგანიც უნდა ჩავრთოთ სიმრავლეში, ხოლო დარჩენილი საჭირო მასა 0-ის ტოლი გახდება. ამრიგად, საძებნ სიმრავლეში შედიან 1, 2 და 4 საგნები. ქვემოთ კი მოყვანილია განხილული ალგორითმის რეალიზაცია:

```
T[0, 0]=1;
FOR j=1 TO 16 {  T[0, j]=0  }
FOR i=1 TO 5 {  T[i, 0]=0  }
FOR i=1 TO 5 {
   FOR j=1 TO 16 {
      IF j >= M[i] THEN {  T[i, j]=max(T[i - 1, j], T[i - 1, j - M[i]])  } ELSE  {  T[i, j]=T[i - 1, j]  }
   }     }
sum=16;
IF T[5, 16] = 1 THEN {
  FOR i= 5  TO  1 {
    IF T[i, sum] = T[i -1, sum] THEN {  WRITE (i, "–No")  }  ELSE {  WRITE (i, "–Yes");  sum=sum - M[i]  }
  }     }
ELSE {   WRITE ("No solution")   }
```

ზურგჩანთის დისკრეტული ამოცანის სხვადასხვა მოდიფიკაცია არსებობს, რომლებიც მოყვანილი ალგორითმის შესაბამისად გადაკეთებით ამოიხსნებიან, მაგრამ არსებობენ ამოცანებიც, რომლებიც გარეგნულად ჰგვანან ზურგჩანთის ამოცანას, სინამდვილეში კი მათი ამოხსნის ეფექტური ალგორითმი ნაპოვნი არ არის. მაგალითად, ასეთ ამოცანას წარმოადგენს ნატურალური რიცხვებისაგან შედგენილი სიმრავლის გაყოფა ორ ტოლ ნაწილად. ეს ამოცანა NP-სრულია (ტერმინი “NP-სისრულე” წარმოდგება ცნებისაგან “Non Polinomial”, ანუ ამოცანა, რომელიც პოლინომიალურ დროში არ იხსნება). იგი შეიძლება ამოვხსნათ, მხოლოდ ძალზე შეზღუდული სიდიდის მონაცემებისათვის.

6.16. განძის ძიება

(USACO, “მწვანე” დივიზიონი, 2000-01 წ.წ., ზამთრის პირველობა)

ფერმერმა ჯონმა (იგივე – ფჯ) იპოვა რუკა, რომელზეც აღნიშნულია მისი ფერმის ტერიტორიაზე ჩამარხული განძები. განძები ჩამოთვლილია მიმდევრობით 1..N ($1 \le N \le 80$). ყოველი განძი ჩამარხულია განსხვავებული მთელი (x,y) კოორდინატის მქონე ადგილზე და ჩამარხვის სიღრმე მოცემულია ასევე მთელი z რიცხვით, ხოლო განძის ღირებულება ასევე მთელი V მნიშვნელობით.

ფჯ ნებისმიერ ორ პუნქტს შორის მოძრაობს ე.წ. “მანჰეტენის” სტილით, რაც გამორიცხავს დიაგონალურ მოძრაობას. მაგალითად (1,2) კოორდინატების მქონე პუნქტიდან (5,7) კოორდინატების მქონე პუნქტამდე ფჯ-მ უნდა გაიაროს მანძილის ოთხი ერთეული პირველი კოორდინატის გასწვრივ და შემდეგ 5 ერთეული მეორე კოორდინატის გასწვრივ. მანძილის თითო ერთეულის გავლას ფჯ ანდომებს ერთ წუთს და ასევე ერთ წუთს ხარჯავს 0,5 ერთეულის ამოთხრაზე. თქვენ აგრეთვე გეძლევათ მთელი რიცხვით აღწერილი T დრო, რომელიც შეუძლია გამოიყენოს ფჯ-ს განძის ძიებაზე მთლიანად.

ფჯ იწყებს მოძრაობას (0,0) კოორდინატთა სათავიდან და განიხილავს განძებს თანმიმდევრობით 1-დან N-მდე. მან ყოველი კონკრეტული განძისათვის უნდა გადაწყვიტოს, გამოტოვოს ის თუ ამოთხაროს (რაც მოითხოვს გარკვეულ დროს, მაგრამ აძლევს მას რაღაც ღირებულების განძს, როდესაც განძი მთლიანად ამოთხრილია). თქვენი ამოცანაა განსაზღვროთ მაქსიმალური ღირებულება, რომელიც შეუძლია მოიპოვოს ფჯ-ს მოცემულ ვადაში. ფჯ უნდა დაბრუნდეს კოორდინატთა სათავეში მოცემული ვადის ამოწურვამდე.

შემავალი მონაცემები: პირველ სტრიქონში მოცემულია ორი მთელი რიცხვი: N (1≤N≤80) და T (1≤T≤1000000). 2-დან N+1 სტრიქონამდე თითოეულში მოცემულია ჰარით გამოყოფილი ოთხი მთელი რიცხვი, რომლებიც აღწერენ შესაბამის განძს: x (-100≤x≤100), y (-80≤y≤80), z (0<z≤25) და V (1≤V≤1000).

გამომავალი მონაცემები: ერთადერთი სტრიქონი ერთადერთი მთელი რიცხვით, რომელიც წარმოადგენს ფჯ-ს მიერ მოპოვებული მაქსიმალური ღირებულების განძს მოცემულ დროში.

შემავალი მონაცემების მაგალითი (treas.in)

```
3 20
2 3 2 5
-5 0 8 51
2 -2 1 14
```

გამომავალი მონაცემი მოყვანილი მაგალითისათვის (treas.out):

```
19
```

*მითითება. თავდაპირველად უნდა შევნიშნოთ, რომ პირობაში მითითებული T დრო აშკარად აღემატება იმ რეალურ დროს, რაც შეიძლება ფჯ-ს დაეხარჯოს. მაქსიმალური გადაადგილება x ღერძის გასწვრივ არ აღემატება 200 ერთეულს, ხოლო y ღერძის გასწვრივ 160 ერთეულს. იმ შემთხვევაშიც კი, თუ ვთქვათ, კენტნომრიანი განძები განლაგებულია საძებნი არის ერთ-ერთ კუთხეში, ხოლო ლუწნომრიანი განძები დიაგონალურად საპირისპირო კუთხეში, ყველა განძთან მისასვლელი დრო არ აღემატება (200+160)*80=28800 წუთს, ხოლო თუ ამას დავუმატებთ განძების ამოთხრის მაქსიმალურ დროს 80*25*2=4000 წუთს, ცხადი ხდება, რომ ყველაზე უარეს შემთხვევაშიც კი არ დაიხარჯება 33000 წუთზე მეტი. ეს გარემოება მნიშვნელოვანია იმდენად, რამდენადაც დინამიური სქემის შედგენისას მოგვიწევს სამუშაო მასივის შერჩევა დროისა და მიმდინარე განძის მიხედვით, ანუ 80×33000-ზე, ხოლო მასივი ზომით 80×1000000-ზე დიდ მანქანურ მეხსიერებას მოითხოვდა (ამოცანის ამოხსნისათვის განსაზღვრული პროგრამული უზრუნველყოფა იძლეოდა 16მბ-ის გამოყენების საშუალებას).*

ამოხსნის პირველ ეტაპზე საჭიროა შევადგინოთ მასივი (სახელად, ვთქვათ TM), რომელშიც მოცემული იქნება ნებისმიერ ორ A და B პუნქტს შორის გადასაადგილებლად საჭირო დრო. ამ დროში ჩართულ უნდა იქნას მეორე (ანუ B) პუნქტში მდებარე განძის ამოთხრისათვის საჭირო დროც, რადგან კონკრეტულ პუნქტში მისვლას მხოლოდ მაშინ აქვს აზრი, თუ ამ პუნქტში არსებულ განძს ამოვთხრით, წინააღმდეგ შემთხვევაში უმჯობესია პუნქტი საერთოდ გამოვტოვოთ. მაშასადამე: $TM(i,j)=abs(i_x-j_x)+abs(i_y-j_y)+2*z_j$, სადაც i_x და i_y არის i-ური წევრის, ხოლო j_x და j_y – j-ური წევრის კოორდინატები, z_j კი j-ური განძის ჩამარხვის სიღრმეა. მაგალითში ნაჩვენები მონაცემებისათვის TM მასივს ექნება სახე:

	1	2	3
0	9 (5+4)	21 (5+16)	6 (4+2)
1	nil	26(10+16)	7(5+2)
2	nil	nil	11(9+2)

ნახ. 6.25

სტრიქონი 0-ის გასწვრივ გვიჩვენებს (0,0) კოორდინატთა სათავიდან შესაბამის განძამდე მისვლის დროისა და ამ განძის ამოთხრის დროის ჯამს (ცალკეული მნიშვნელობები მითითებულია ფრჩხილებში და მათი შენახვა სამუშაო მასივში, რა თქმა უნდა, საჭირო არ არის). პირველ სვეტში მითითებული არაა მესამე განძის ნომერი, რადგან პირობის თანახმად განძებთან მისვლა მხოლოდ აღმავალი თანმიმდევრობით არის შესაძლებელი და ამიტომ ყველაზე მაღალი ნომრის განძიდან მხოლოდ კოორდინატთა სათავეში შეიძლება დაბრუნება. ცალკე მასივში შევინახოთ თითოეული განძიდან კოორდინატთა სათავეში დაბრუნების დრო (ამ მონაცემის შენახვა TM მასივშიც შეიძლება, მაგრამ ამას არსებითი მნიშვნელობა არა აქვს). დავარქვათ ამ მასივს TK:

განძის ნომერი	*1*	*2*	*3*
სათავეში დაბრუნების დრო	*5*	*5*	*4*

ნახ. 6.26

შეიძლება ითქვას, რომ აქამდე მოსამზადებელ სამუშაოებს ვატარებდით. ახლა შემოვიღოთ ზემოთ ნახსენები მასივი ზომით 80×33000 (დავარქვათ მას REZ მასივი). მისი ერთი მდგენელი იქნება დრო, მეორე – განძთა ნომრები, ხოლო გადაკვეთაზე ჩავწეროთ მოცემული მომენტისათვის მოგროვებული განძის ღირებულება. რადგან ჩვენი მაგალითისათვის დროის ლიმიტი 20 წუთია, განვიხილოთ მასივი ზომით 3×20. პირველ ბიჯზე მასივში ჩავწეროთ კოორდინატთა სათავიდან თითოეულ განძამდე მისვლის შემდეგ შექმნილი სიტუაცია:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1										5											
2																					
3							14														

ნახ. 6.27

*მეორე განძამდე მისვლასა და კოორდინატთა სათავეში დაბრუნებას ვერ ვასწრებთ, რადგან ამისათვის საჭიროა 5+2*8+5=26 წუთი. რადგან 26 მეტია ჩვენთვის მოცემულ დროის ლიმიტზე, ეს მნიშვნელობა REZ მასივში არ შეგვაქვს. აქვე შევნიშნოთ, რომ REZ მასივის ელემენტებს კოორდინატთა სათავეში დაბრუნების დრო არ ემატება, თუმცა ყოველი ახალი ელემენტი ჩაიწერება მხოლოდ მაშინ, თუ მიმდინარე წერტილიდან სათავეში დაბრუნებას ვასწრებთ.*

ახლა ვიმოძრაოთ REZ მასივში მარცხნიდან მარჯვნივ და ზემოდან ქვემოთ და nil-ისაგან განსხვავებული ყველა ელემენტისათვის (nil-ს ნახაზზე ცარიელი უჯრედები შეესაბამება) განვიხილოთ მასზე მაღალი ნომრის მქონე განძთან მისვლის შესაძლებლობა. თუკი დახარჯული დრო არ ამეტებს ლიმიტს, შესაბამისი ნომრისა და დროის მნიშვნელობის გადაკვეთაზე ჩავწეროთ დაგროვილი განძის ღირებულება და ა.შ. გასათვალისწინებელია სიტუაცია, როცა რომელიმე ელემენტში მნიშვნელობის ჩაწერისას მას უკვე ექნება nil-საგან განსხვავებული მნიშვნელობა. ამ შემთხვევაში უპირატესობა მიენიჭება უფრო დიდ რიცხვს, ანუ თუ რომელიმე წერტილში სხვადასხვა გზით მისვლისას ჩვენ ერთნაირი დრო დაგვეხარჯა, ვამჯობინებთ ვარიანტს, რომლითაც მეტ განძს დავაგროვებთ.

ჩვენი მაგალითისათვის nil-საგან განსხვავებული პირველი ელემენტია REZ(1,9)=5. აქედან მეორე განძთან მისვლას აზრი არა აქვს, ხოლო მესამე განძთან მისვლას და უკან დაბრუნებას ვასწრებთ. ამიტომ REZ მასივი მიიღებს სახეს:

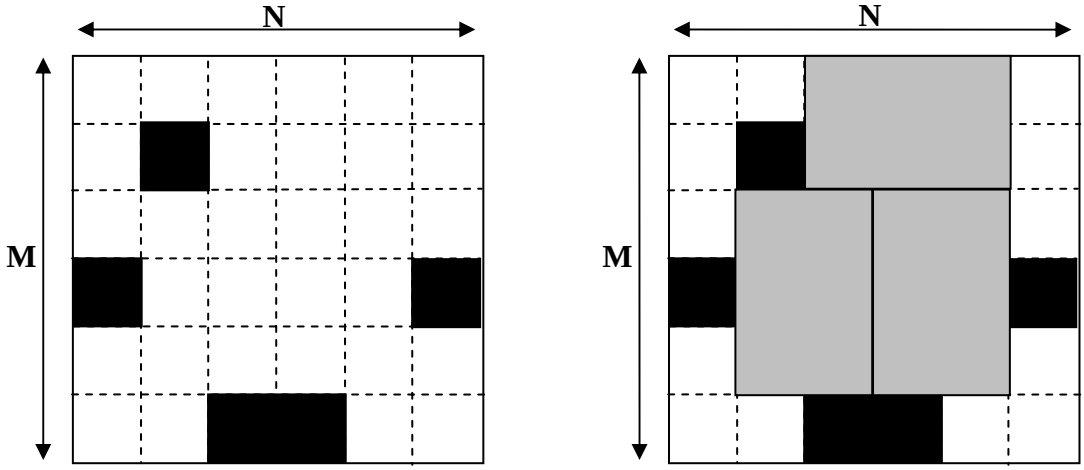
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1										5											
2																					
3							14										19				

ნახ. 6.28

რადგან მაგალითში სულ სამი განძია, მესამე სტრიქონის განხილვა საჭირო არაა (ყველაზე მაღალი ნომრის მქონე განძიდან მხოლოდ კოორდინატთა სათავეში თუ დავბრუნდებით) და პროცესი წყდება. ამის შემდეგ ვპოულობთ REZ მასივის მაქსიმუმს (მისი დამახსოვრება REZ მასივის შევსების დროსაც შეიძლება), რომელიც წარმოადგენს ჩვენი ამოცანის ამონახსნს.

ამოცანა 6.20. კომპანია Bugs Integrated
(ცენტრალური ევროპის ოლიმპიადა, 2002 წელი, სლოვაკეთი)

კომპანია Bugs Integrated წარმოადგენს მეხსიერების გაფართოებული ჩიპების მთავარ დამამზადებელს. კომპანიამ დაიწყო 6 ტერაბაიტი მოცულობის Q-RAM ჩიპების გამოშვება. ყოველი ჩიპი შედგება 6 კვადრატული მოდულისაგან, რომლებსაც გააჩნიათ 2×3 ზომის მართკუთხა ფორმა. Q-RAM ჩიპების ამოჭრა ხდება სილიციუმის მართკუთხა ფირფიტიდან, რომელიც დაყოფილია N×M კვადრატულ მოდულად. ფირფიტის თითოეული კვადრატი გულმოდგინედ მოწმდება და დაზიანებულთა მონიშვნა ხდება შავი ფერის მარკერით.



ნახ. 6.32.

შემოწმების შემდეგ სილიციუმის ფირფიტიდან ხდება მეხსიერების ჩიპების ამოჭრა. ყოველი ჩიპი წარმოადგენს 2×3 ან 3×2 ზომის მართკუთხედს. ჩიპი არ უნდა შეიცავდეს დაზიანებულ (მარკერით მონიშნულ) კვადრატს. ხშირად შეუძლებელია ფირფიტის ისე ამოჭრა, რომ ყველა კარგი კვადრატი გამოყენებულ იქნას. კომპანიას სურს, რომ რაც შეიძლება ნაკლები რაოდენობის კარგი კვადრატი დარჩეს გამოუყენებელი, ამიტომ მათ სურთ იცოდნენ, მოცემული ფირფიტიდან რა მაქსიმალური რაოდენობის ჩიპის ამოჭრაა შესაძლებელი.

ამოცანა: მოცემულია სილიციუმის რამდენიმე ფირფიტა და დაზიანებული კვადრატების სია თითოეულ ფირფიტაზე. დაწერეთ პროგრამა, რომელიც დაადგენს, თუ რა მაქსიმალური რაოდენობის ჩიპის ამოჭრაა შესაძლებელი თითოეული ფირფიტიდან.

შემავალი მონაცემები: პირველ სტრიქონში მოცემულია ერთადერთი მთელი რიცხვი D (1≤D≤5), რომელიც აღნიშნავს სილიციუმის ფირფიტების რაოდენობა. შემდეგ მოდის მონაცემთა D ცალი ბლოკი, რომელთაგან თითოეული აღწერს თითო ფირფიტას.

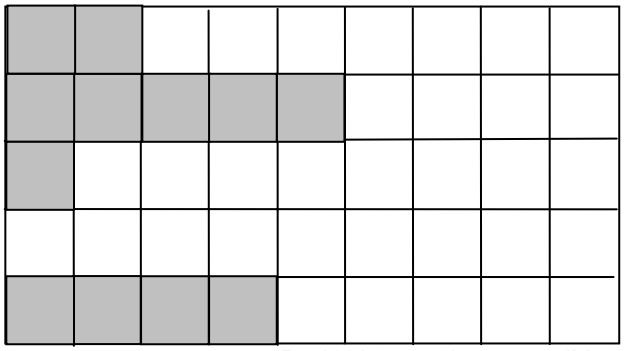
ყოველი ბლოკის პირველ სტრიქონში მოცემულია ჰარით გაყოფილი სამი მთელი რიცხვი – N (1≤N≤150), M (1≤M≤10) და K (1≤K≤MN). N წარმოადგენს ფირფიტის სიგრძეს, M – ფირფიტის სიმაღლეს, ხოლო K – დაზიანებული კვადრატების რაოდენობას ფირფიტაზე. მომდევნო K სტრიქონში მოცემულია დაზიანებული კვადრატების კოორდინატები ფირფიტაზე. კოორდინატები აღწერილია ორი მთელი x და y რიცხვით (1≤x≤N, 1≤y≤M). ზედა მარცხენა კვადრატის კოორდინატებია [1,1], ხოლო ქვედა მარჯვენასი – [N,M].

გამომავალი მონაცემები: ყოველი ფირფიტისათვის გამოაქვთ ერთი მთელი რიცხვი, რომელიც აღნიშნავს ფირფიტიდან ამოსაჭრელი მეხსიერების ჩიპების მაქსიმალურად შესაძლებელ რაოდენობას.

შემავალი მონაცემების მაგალითი	გამომავალი მონაცემი მოყვანილი მაგალითისათვის
2 6 6 5 1 4 4 6 2 2 3 6 6 4 6 5 4 3 3 6 1 6 2 6 4	3 4

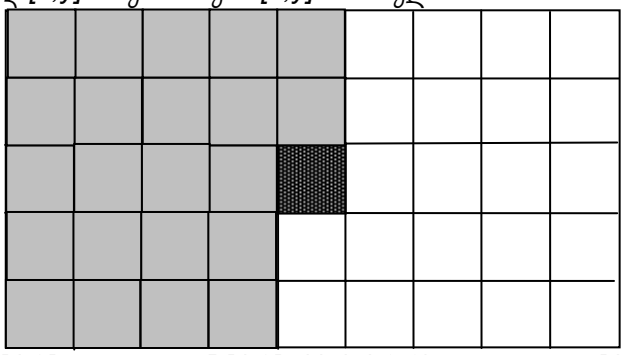
ავტორისეული ამოხსნა. ამოხსნა ეფუძნება დინამიურ პროგრამირებას და აქვს O ($MN3^M$) სირთულე დროის თვალსაზრისით. ყველა [x,y] კოორდინატი შემცირებულია 1-ით და მოთავსებულია საზღვრებში $0 \le x < N$, $0 \le y < M$.

განსაზღვრება 1: შემოვიღოთ $B = (b_0, b_1, \ldots, b_{M-1})$ რიცხვთა ვექტორი, რომელსაც ვუწოდოთ საზღვარი. S(B) საზღვრის სიმრავლე განვსაზღვროთ როგორც ყველა [x,y] უჯრედი, რომელიც აკმაყოფილებს $x \le b_y$ პირობას. სხვაგვარად რომ ვთქვათ, ესაა იმ უჯრედების სიმრავლე, რომელიც იმყოფება B საზღვრის მარცხნივ. ქვემოთ საზღვარი და მისი შესაბამისი სიმრავლე გაიგივებული იქნება.



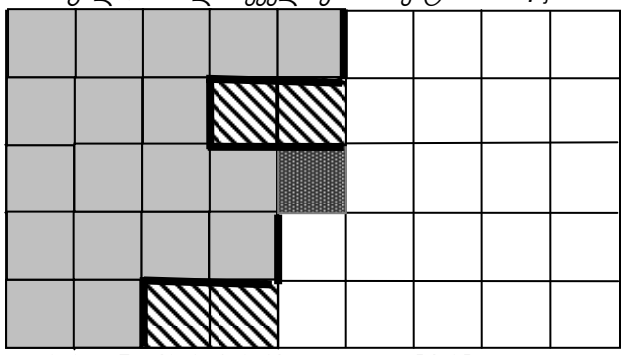
ნახ. 6.33. საზღვრის სიმრავლე B=(1,4,0,-7,3)-სათვის. (N=9, M=5).

განსაზღვრება 2: ყოველი [i,j]–სათვის (0≤i<N, 0≤j<M) განვსაზღვროთ [i,j]-საზღვარი, როგორც B[i, j] = (b_0, b_1, …, b_{M-1}), სადაც $b_0=b_1=\ldots=b_j=i$ და $b_{j+1}=\ldots=b_{M-1}=i-1$. დავარქვათ S(B[i,j]) სიმრავლეს [i,j]-ს ბაზისი. სხვა სიტყვებით S(B[i,j]) წარმოადგენს იმ უკრედთა ერთობლიობას, რომლებიც მდებარეობენ [x,y] უკრედის მარცხნივ ან იმავე სვეტში, ოღონდ [x,y]-ს ზემოთ თვით [x,y]-ის ჩათვლით.



ნახ. 6.34. A[4,2] საზღვარი – B[4,2]=(4,4,4,3,3) წარმოადგენს [4,2]-ის ბაზისს.

განსაზღვრება 3: ვთქვათ, გვაქვს რაიმე ბაზისი B[i,j] საზღვრით და ვთქვათ, გვაქვს P=(p_0,p_1,…,p_{M-1}) ვექტორი, სადაც $p_i\in\{0,1,2\}$. (b_0-p_0, b_1-p_1,…,b_{M-1}-p_{M-1}) ვექტორს ვუწოდოთ B-P. ამ ბაზისისათვის P-პროფილი განვსაზღვროთ როგორც S(B[i,j]-P) სიმრავლე. 0-ით აღვნიშნოთ პროფილი, რომლის ყველა კომპონენტი 0-ის ტოლია, ხოლო e_j–ით აღვნიშნოთ პროფილი, რომლის ყველა კომპონენტი 0-ია p_j=1-ის გარდა.



ნახ. 6.35. P=(0,2,1,0,2) პროფილი [4,2] ბაზისისათვის.

ჩვენ შეგვიძლია განვიხილოთ პროფილები, როგორც რიცხვები 0-დან 3^{M-1}-მდე 3-ის ფუძით. ასეთი სახით ჩვენ ზოგჯერ გამოვიყენებთ P-ს, როგორც მასივის ინდექსს. მუშაობისას ჩვენ ჯერ გადავიყვანთ P-ს 3-ის ფუძიდან 10-ის ფუძეზე, ხოლო შემდეგ გამოვიყენებთ მასივის ინდექსად. ჩვენ შეგვიძლია პლატა განვიხილოთ როგორც G კარგი უკრედების სიმრავლე. ჩვენი უპირველესი ამოცანა მდგომარეობს იმაში, რომ დავადგინოთ ჩიპების მაქსიმალური რაოდენობა, რომელიც შეიძლება ამოეჭრილი იქნას G-დან.

ამოხსნის მთავარი იდეა მდგომარეობს შემდეგში: ყოველი B[i,j] ბაზისისა და P პროფილისათვის გამოვთვალოთ A[0,j,P] – ჩიპების მაქსიმალური რაოდენობა, რომელიც შეიძლება ამოეჭრილ იქნან $G\cap S(B[i,j]-P)$ სიმრავლიდან. შევნიშნოთ, რომ $G\cap S(B[N-1,M-1])=P$, მაშასადამე A[N-1,M-1,0] რიცხვი წარმოადგენს ამოცანის პასუხს.

პლატის ნაწილი $G\cap S(B[0,j])$ ძალზე პატარაა საიმისოდ, რომ მისგან ჩიპები ამოიჭრას, ამიტომ დასაწყისში გვაქვს A[0,j,P] ნებისმიერი j-სა და ნებისმიერი P-სათვის.

ყველა სხვა B[i,j] ბაზისი დავამუშავოთ მარცხნიდან მარჯვნივ (i-ის ზრდადობით) და ზემოდან ქვემოთ (j-ის ზრდადობით). ყოველი ბაზისისათვის განვიხილოთ თითოეული P პროფილი. ფიქსირებული B[i,j] ბაზისისა და ფიქსირებული P პროფილისათვის არსებობს ორი შესაძლებლობა $p_j>0$ ან $p_j=0$.

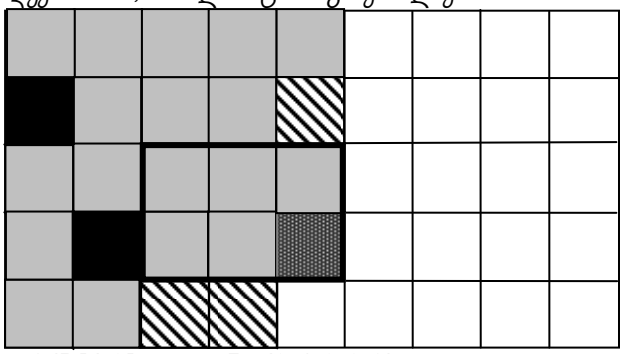
თუ $p_j>0$, მაშინ $G\cap S(B[i,j]-P)=G\cap S(B'-P')$, სადაც $B'=S(B[i',j'])$ წარმოადგენს მიმდევრობაში წინა ბაზისს, ხოლო $P'=P-e_j$. რადგანაც ეს ორი სიმრავლე ტოლია, ჩიპების მაქსიმალური რაოდენობა, რომელიც შეიძლება მათგან ამოიჭრას, ასევე თანაბარი იქნება, ამიტომ $A[i,j,P]=A[i',j',P']$.

თუ $p_j=0$, მაშინ არსებობს სამი შესაძლებლობა, რათა მივიღოთ ჩიპების მაქსიმუმი, რომლებიც შეიძლება ამოჭრილ იქნან $G\cap S(B'-P')$-დან:

- *არ ამოვჭრით ჩიპს, რომელსაც აქვს უფრო ქვედა მარჯვენა კუთხე, ვიდრე $[i,j]$.*
- *ამოვჭრით ჰორიზონტალურ (3x2) ჩიპს, რომელსაც აქვს უფრო ქვედა მარჯვენა კუთხე, ვიდრე $[i,j]$.*
- *ამოვჭრით ვერტიკალურ (2x3) ჩიპს, რომელსაც აქვს უფრო ქვედა მარჯვენა კუთხე, ვიდრე $[i,j]$.*

პირველი შემთხვევისათვის ჩიპების მაქსიმალური რაოდენობაა $A[i',j',P]$, სადაც $S(B[i',j'])$ წარმოადგენს წინა ბაზისს იმ მიმდევრობაში, რომლითაც ჩვენ მას განვიხილავთ.

მეორე შემთხვევაში ჩიპების მაქსიმალური რაოდენობა იქნება $A[i'',j'',P+2e_j+2e_{j-1}]$, სადაც $S(B[i'',j''])$ არის ორი ადგილით წინა ბაზისი იმ მიმდევრობაში, რომლითაც მას განვიხილავთ.



ნახ. 6.36. ჩვენ ვამუშავებთ $S(B[4,3]$ ბაზისს $P=(0,1,0,0,2)$ პროფილით. აქ შეგვიძლია ჰორიზონტალური ჩიპის ამოჭრა. ორი ადგილით წინა ბაზისი იყო $S(B[4,1])$, ახალი პროფილი იქნება $P''=(0,1,2,2,2)$. მიაქციეთ ყურადღება, რომ $S(B[4,1]-P'')$ ივიღეა რაც $S(B[4,3]-P)$ ჰორიზონტალური ჩიპის გარეშე.

მესამე შემთხვევაში ჩიპების მაქსიმალური რაოდენობა იქნება $A[i''',j''',P+e_j+e_{j-1}+e_{j-2}]$, სადაც $S(B[i''',j'''])$ არის სამი ადგილით წინა ბაზისი იმ მიმდევრობაში, რომლითაც მას განვიხილავთ.

ცხადია, რომ $A[i,j,P]$ იქნება მაქსიმუმი ამ სამი რიცხვისაგან. იმის შემოწმება, მოცემულ პოზიციაში ვერტიკალური ჩიპი უნდა ამოვჭრათ თუ ჰორიზონტალური, იოლია. ჰორიზონტალური ჩიპი უნდა ამოიჭრას მაშინ და მხოლოდ მაშინ, როცა ამ პოზიციებში არაა ცუდი უჯრედი და $i\geq 2$, $j\geq 1$ და $p_j=p_{j-1}=0$. ანალოგიურად ვერტიკალური ჩიპი უნდა ამოიჭრას, თუ შესაბამის პოზიციებში არა გვაქვს ცუდი უჯრედები, $i\geq 1$, $j\geq 2$ და $p_j=p_{j-1}=p_{j-2}=0$. ამ პირობების შემოწმება დროის თვალსაზრისით პრობლემას არ წარმოადგენს.

რთული არაა იმის შემჩნევა, რომ $A[i,j,P]$ მნიშვნელობების დამახსოვრება საჭიროა მხოლოდ ოთხი უკანასკნელი ბაზისისათვის (გამოსათვლელი და წინა სამი). თითოეული ბაზისისათვის დაგვჭირდება არაუმეტეს $3^M\leq 3^{10}<60000$ პროფილისა, რაც თავისუფლად მოთავსდება მეხსიერებაში.

$O(MN)$ ბაზისისათვის განსახილველია 3^M რაოდენობის პროფილი, ამიტომ სულ საჭიროა $O(MN3^M)$ დრო.

6.2. ხარბი ალგორითმები

რიგი ოპტიმიზაციური ამოცანები შეიძლება ამოიხსნან დინამიურ პროგრამირებაზე უფრო მარტივი და სწრაფი ალგორითმებით. ამის საშუალებას იძლევიან ე.წ. **ხარბი ალგორითმები** (greedy algorithms). ასეთი მეთოდის გამოყენებისას ყოველ ბიჯზე კეთდება ლოკალურად ოპტიმალური არჩევანი იმ იმედით, რომ საბოლოო შედეგიც ოპტიმალური იქნება. ეს რა თქმა უნდა, ყოველთვის ასე არ არის, მაგრამ ზოგიერთი ამოცანისათვის მართლაც იძლევა სწორ ამონახსნს.

განვიხილოთ **ამოცანა განაცხადების შერჩევაზე**: ვთქვათ, მოცემულია n განაცხადი ერთ და იმავე აუდიტორიაში მეცადინეობის ჩატარებაზე. ორი განსხვავებული მეცადინეობა არ შეიძლება დროში გადაიფაროს. ყოველ განაცხადში მითითებულია მეცადინეობის დაწყებისა და დამთავრების დრო (iური განაცხადისათვის შესაბამისად s_i და f_i). სხვადასხვა განაცხადები შეიძლება გადაიკვეთონ, მაგრამ ამ შემთხვევაში დაკმაყოფილდება მხოლოდ ერთი მათგანი. ჩვენ ვაიგივებთ თითოეულ განაცხადს $[s_i,f_i)$ შუალედთან, ასე რომ ერთი მეცადინეობის დამთავრების დრო შეიძლება დაემთხვეს მეორის დაწყებას. ასეთი სიტუაცია გადაკვეთად არ ითვლება.

ზოგადად i და j ნომრების მქონე განაცხადები თავსებადია (compatible), თუკი $[s_i, f_i)$ და $[s_j, f_j)$ ინტერვალები არ თანაიკვეთებიან (სხვა სიტყვებით – $f_i\leq s_j$ ან $f_j\leq s_i$). ამოცანა განაცხადების შერჩევაზე (activity-selection problem) მდგომარეობს იმაში, რომ ამოვარჩიოთ ერთმანეთთან თავსებადი მაქსიმალური რაოდენობის განაცხადი.

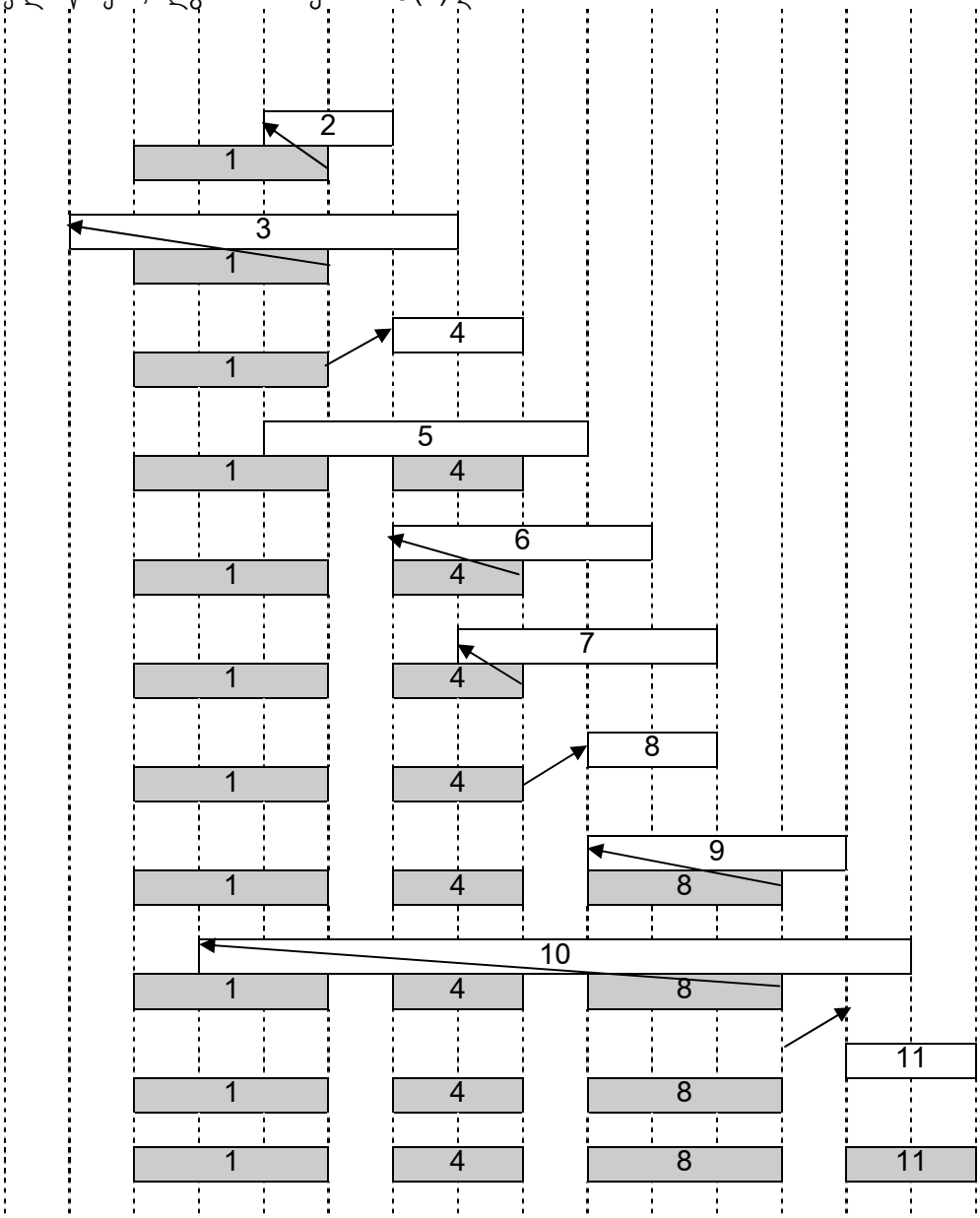
ხარბი ალგორითმი მუშაობს შემდეგნაირად: დავუშვათ განაცხადები დალაგებულია დამთავრების დროის ზრდადობის მიხედვით: $f_1\leq f_2\leq\ldots\leq f_n$. თუკი მონაცემები დალაგებული არ არის, მისი დალაგება შესაძლებელია O(n

logn) დროში. თუ განაცხადებს ერთნაირი დამთავრების დრო აქვთ, ისინი შეიძლება განლაგდნენ ნებისმიერად. თუ f-ს და s-ს განვიხილავთ როგორც შესაბამის მასივებს, ალგორითმს ექნება სახე:

```
GREEDY-ACTIVITY-SELECTOR (s,f)
1 n=length[s]
2 A={1}
3 j=1
4 for i=2 to n do {
5         if s_i ≥ f_j then  {  A=A∪⟨ i ⟩ ;   j=i  }        }
6 return A
```

ალგორითმის მუშაობა მოცემულია ნახ.6.37-ზე. A სიმრავლე შედგება ამორჩეული განაცხადების ნომრებისაგან, ხოლო j – უკანასკნელია ამ ნომრებს შორის, ამასთან $f_j=\max\{f_k:k\in A\}$. რადგან განაცხადები დალაგებულია მათი დამთავრების დროის მიხედვით, თავიდან A სიმრავლე შეიცავს ნომერ 1 განაცხადს და j=1 (2-3 სტრიქონები). შემდეგ 4-7 სტრიქონებში რიგის მიხედვით ვეძებთ განაცხადს, რომელიც არ იწყება j ნომრის მქონე განაცხადის დამთავრებაზე ადრე. თუკი ასეთი მოიძებნა, მას ჩავრთავთ A სიმრავლეში და j ცვლადს მივანიჭებთ მის ნომერს (5 სტრიქონი). თუკი სორტირების დროს არ გავითვალისწინებთ, ალგორითმი მუშაობს $\Theta(n)$ დროში.

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



ნახ. 6.37.

თეორემა 6.2. *ალგორითმი GREEDY-ACTIVITY-SELECTOR გვაძლევს თავსებადი განცხადების მაქსიმალურად შესაძლებელ რაოდენობას.*

დამტკიცება. როგორც აღვნიშნეთ, განაცხადები დალაგებულია დამთავრების დროის მიხედვით. ამის გამო ოპტიმალური ამონახსნი აუცილებლად შეიცავს პირველ განაცხადს. მართლაც დავუშვებთ საწინააღმდეგო და ვთქვათ, პირველი განაცხადი არ შედის ოპტიმალურ სიმრავლეში, მაშინ ჩვენ შეგვიძლია ამ სიმრავლიდან ყველაზე ადრე დამთავრებული განაცხადი შევცვალოთ პირველი განაცხადით. ამით ოპტიმალურობა არ დაირღვევა, რადგან არცერთი

განაცხადი არ მთავრდება პირველ განაცხადზე ადრე და მათ შორის ისიც, რომელიც შევცვალეთ. მაშასადამე, ამ ცვლილებით არ შეიცვლება განაცხადთა საერთო რაოდენობაც და თუკი მოცემული სიმრავლე ოპტიმალური იყო, ოპტიმალური იქნება ის სიმრავლეც, რომელიც პირველ განაცხადს შეიცავს.

ამის შემდეგ ჩვენ განვიხილავთ მხოლოდ იმ განაცხადებს, რომლებიც თავსებადია პირველ განაცხადთან (ყველა არათავსებადი შეგვიძლია გავაუქმოთ). ამრიგად, ჩვენ მივიღეთ იგივე ამოცანა, ოღონდ უფრო ნაკლები რაოდენობის განაცხადებისათვის. ინდუქციის მეთოდით ვასკვნით, რომ ყოველ ბიჯზე ხარბი ამორჩევის საშუალებით მივალთ ოპტიმალურ ამონახსნამდე. □

ზოგადი სქემა იმის გასაგებად, გვაძლევს თუ არა ხარბი ალგორითმი ოპტიმალურ ამოხსნას, არ არსებობს, თუმცა შეიძლება გამოიყოს ორი თავისებურება იმ ამოცანებისათვის, რომლებიც ხარბი ალგორითმით იხსნება. ესაა ხარბი ამორჩევის პრინციპი და ქვეამოცანების ოპტიმალურობის თვისება.

იტყვიან, რომ ოპტიმიზაციური ამოცანისათვის გამოყენებადია ხარბი ამორჩევის პრინციპი (greedy-shoice property), თუკი ლოკალურად ოპტიმალური (ხარბი) ამორჩევა იძლევა გლობალურ ოპტიმუმს. განსხვავება ხარბ ალგორითმებსა და დინამიურ პროგრამირებას შორის იმაში მდგომარეობს, რომ ხარბი ალგორითმი ყოველ ბიჯზე ირჩევს საუკეთესო ვარიანტს და ამის შემდეგ ცდილობს გააკეთოს იგივე დარჩენილ ვარიანტებში, ხოლო დინამიური პროგრამირების ალგორითმით წინასწარ ხდება შედეგების გამოთვლა ყველა ვარიანტისათვის.

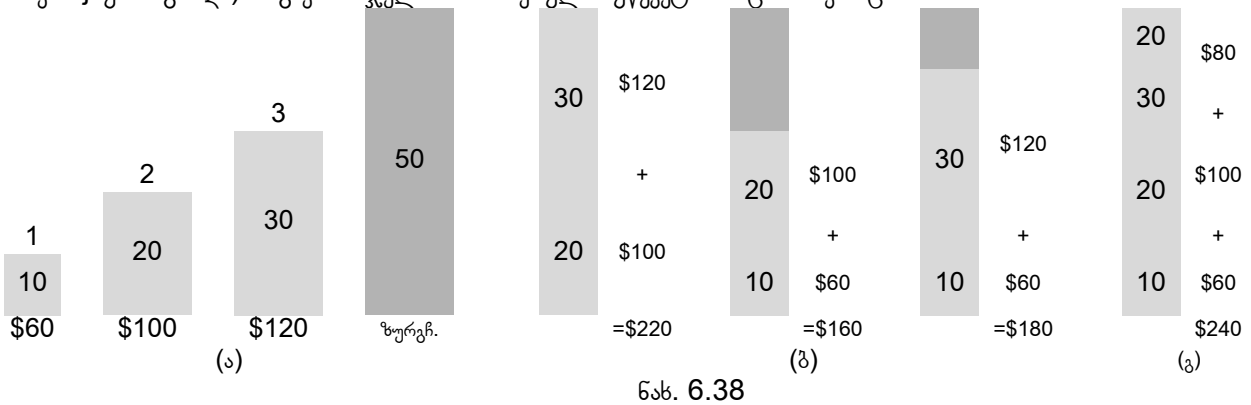
იმის დამტკიცება, რომ ხარბი ალგორითმი ოპტიმალურ ამონახსნს იძლევა, არ წარმოადგენს ტრივიალურ ამოცანას. ტიპიურ შემთხვევებში ასეთი მტკიცება ხდება 6.2 თეორემაში მოცემული სქემით. თავიდან ვამტკიცებთ, რომ პირველ ბიჯზე ხარბი ამორჩევა არ კეტავს გზას ოპტიმალური ამონახსნისაკენ – ნებისმიერი ამონახსნისათვის არსებობს სხვაც, რომელიც შეთანხმებულია ხარბ ამორჩევასთან და არაა პირველზე უარესი. ამის შემდეგ უნდა ვაჩვენოთ, რომ ქვეამოცანა, რომელიც წარმოიშვა პირველ ბიჯზე ხარბი ამორჩევის შემდეგ, საწყისის ანალოგიურია.

ხარბი ალგორითმებით ამოხსნად ამოცანებს უნდა ჰქონდეთ ქვეამოცანების ოპტიმალურობის (have optimal substructure) თვისება: მთელი ამოცანის ოპტიმალური ამონახსნი მოიცავს ოპტიმალურ ამონახსნებს ქვეამოცანებისათვის. ამ თვისებას ჩვენ უკვე გავეცანით დინამიური პროგრამირების განხილვის დროს. ორივე აღნიშნული მეთოდი – დინამიური პროგრამირებაც და ხარბი ალგორითმებიც ემყარება ქვეამოცანების ოპტიმალურობის თვისებას, ამიტომ ხშირად იქმნება ერთი მეთოდის ნაცვლად მეორის გამოყენების საფრთხე. თუკი ერთ შემთხვევაში – ხარბი ალგორითმის მაგიერ დინამიური პროგრამირების გამოყენებისას, მაინც შესაძლებელია სწორი პასუხის მიღება (თუმცა აუცილებლად წავაგებთ დროში), მეორე შემთხვევის დროს (დინამიურის ნაცვლად ხარბი ალგორითმის გამოყენებისას) პრაქტიკულად შეუძლებელია სწორი პასუხის მიღება. ამ ორი მეთოდის თავისებურებანი განვიხილოთ ცნობილი ოპტიმიზაციური ამოცანის ორ ვარიანტზე.

ზურგჩანთის დისკრეტული ამოცანა (0-1 knapsack problem): ვთქვათ ქურდი შეიპარა საწყობში, რომელშიც ინახება n ნივთი. i-ური ნივთი ღირს v_i დოლარი და იწონის w_i კილოგრამს (v_i და w_i – მთელი რიცხვებია). ქურდს სურს წაიღოს მაქსიმალური საფასურის საქონელი, ამასთან მაქსიმალური წონა, რომელიც მან ზურგჩანთით შეიძლება წაიღოს, W-ს ტოლია (W მთელი რიცხვია). რომელი ნივთები უნდა ჩაალაგოს ქურდმა ზურგჩანთაში?

ზურგჩანთის უწყვეტი ამოცანა (fractional knapsack problem): დისკრეტული ამოცანისაგან იმით განსხვავდება, რომ ქურდს შეუძლია დაანაწევროს მოპარული ნივთები და ისინი ზურგჩანთაში ჩაალაგოს ნაწილ-ნაწილ და არა მთლიანად (შეგვიძლია ვთქვათ, რომ დისკრეტულ ამოცანაში ქურდს საქმე აქვს ოქროს ზოდებთან, ხოლო უწყვეტ ამოცანაში – ოქროს ქვიშასთან).

ორივე ამოცანას ზურგჩანთის შესახებ გააჩნია ოპტიმალურობის თვისება ქვეამოცანებისათვის. მართლაც, დისკრეტული ამოცანის შემთხვევაში თუკი ოპტიმალურად გავსებული ზურგჩანთიდან ამოვიღებთ j ნომრის მქონე ნივთს, მივიღებთ ოპტიმალურ ამონახსნს $W-w_j$ მაქსიმალური წონის მქონე ზურგჩანთისა და n-1 ნივთისათვის (ყველა ნივთი j-ურის გარდა). მსგავსი მსჯელობა მართებულია უწყვეტი ამოცანისთვისაც.



ნახ. 6.38

მიუხედავად იმისა, რომ ამოცანები ზურგჩანთის შესახებ ძალიან წააგავს ერთმანეთს, ხარბი ალგორითმი პოულობს ოპტიმუმს უწყვეტი ამოცანისათვის და ვერ პოულობს – დისკრეტულისათვის. უწყვეტი ამოცანისათვის ალგორითმი ასე გამოიყურება. ყველა საქონლისათვის გამოვთვალოთ 1 კილოგრამის ფასი. თავდაპირველად ქურდი აიღებს ყველაზე ძვირფასი საქონლის მაქსიმუმს, თუკი ზურგჩანთაში ადგილი კიდევ დარჩა აიღებს ფასით მომდევნო საქონელს და ასე გააგრძელებს მანამ, ვიდრე ზურგჩანთა არ შეივსება. ამ ალგორითმის მუშაობის დროა O(n *log*n), რაც განპირობებულია იმით, რომ მონაცემებს წინასწარ სჭირდება სორტირება.

იმაში დასარწმუნებლად, რომ ანალოგიური ხარბი ალგორითმი არ სწორად მუშაობს დისკრეტული ამოცანისათვის, განვიხილოთ 6.38ა ნახაზზე მოცემული შემთხვევა. აქ მოცემულია 10, 20 და 30კგ წონის სამი ნივთი, რომელთა ღირებულება შესაბამისად არის 60$, 100$ და 120$. მასის ერთეულის ღირებულება იქნება 6$, 5$ და 4$. ხარბი სტრატეგიის თანახმად ქურდმა თავიდან პირველი ნივთი უნდა ჩადოს ზურგჩანთაში, რადგან ის ყველაზე ძვირფასია. მაგრამ ცხადია, რომ უფრო მომგებიანია 2-ე და 3-ე ნივთების არჩევა, რადგან ამ შემთხვევაში წაღებული ნივთების საერთო ღირებულება 220$ იქნება, მაშინ როცა ხარბი ალგორითმით აირჩეოდა 160$-ის საქონელი (ნახ. 6.38ბ).

უწყვეტი ამოცანისათვის ხარბი ალგორითმის მუშაობა ნაჩვენებია 6.38გ ნახაზზე, ხოლო დისკრეტული ამოცანის ამოხსნისათვის გამოიყენება დინამიური პროგრამირება.

თევზები

(ბალკანეთის ოლიმპიადა ინფორმატიკაში მოსწავლეთა შორის, 2000 წელი)

მეთევზემ დაიჭირა N რაოდენობის თევზი. თითოეული თევზის t_i წონა ნაპირზე განისაზღვრება. მეთევზე მუშაობს ერთი სათევზაო კომპანიისათვის და უფლება აქვს წაიღოს სახლში არაუმეტეს T წონის თევზი. მეთევზეს სურს სახლში წაიღოს მინიმალური რაოდენობის, მაგრამ მაქსიმალური L ($L \le T$) წონის თევზი.

ამოცანა: შეადგინეთ პროგრამა, რომელიც მოეხმარება მეთევზეს თევზის შერჩევაში.

შეზღუდვები:

$N \le 500000$ დადებითი მთელი რიცხვი აღნიშნავს თევზების რაოდენობას.

$T \le 7000$ დადებითი მთელი რიცხვი აღნიშნავს მაქსიმალურ წონას, რომელიც შეუძლია წაიღოს მეთევზემ.

$t_i \le 1000$ დადებითი მთელი რიცხვი აღნიშნავს i-ური თევზის წონას.

ტესტის გავლის დრო: 13 წამი.

შემავალი ფაილი FISH.IN. ფაილის პირველ სტრიქონში მოცემულია რიცხვები N და T, ხოლო მომდევნო N სტრიქონიდან თითოეულში – რიცხვი t_i.

გამომავალი ფაილი FISH.OUT. თუკი არ არსებობს ამონახსნი T რიცხვისათვის, მაშინ მოძებნილი უნდა იქნას მაქსიმალური $L \le T$. გამომავალი ფაილის პირველ სტრიქონში მოცემული უნდა იქნას შინ წასაღები თევზების K რაოდენობა, ხოლო მომდევნო K სტრიქონში შინ წასაღები თითოეული თევზის წონა.

მაგ.

FISH.IN	FISH.OUT
10 280 300 10 30 80 200 20 20 60 10 100	2 200 80

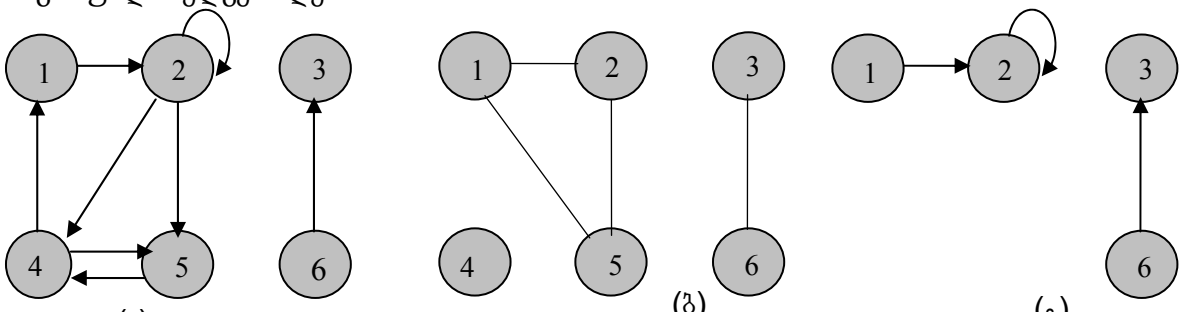
მითითება: შემომავალი მონაცემები ჯერ უნდა დავსორტიროთ კლებადობით. შემდეგ კი თანმიმდევრობით ავიღოთ თევზები დალაგებული სიმრავლის თავიდან მანამ, ვიდრე აღებული თევზების ჯამური მასა არ გადააჭარბებს T-ს.

7. გრაფები

7.1. ძირითადი ცნებები და განსაზღვრებები

ორიენტირებული გრაფი (directed graph) განისაზღვრება როგორც (V,E) წყვილი, სადაც V სასრული სიმრავლეა, ხოლო E წარმოადგენს V-ს ელემენტთა ბინარულ დამოკიდებულებას, ანუ V×V სიმრავლის ქვესიმრავლეა. ორიენტირებულ გრაფს ზოგჯერ **ორგრაფს** (digraph) უწოდებენ. V სიმრავლეს უწოდებენ გრაფის **წვეროთა სიმრავლეს** (vertex set), ხოლო E-ს **წიბოთა სიმრავლეს** (edge set). მათ ელემენტებს შესაბამისად ეწოდებათ **წვერო** (vertex, მრავლობითში vertices) და **წიბო** (edge). წიბოს, რომელიც წვეროს საკუთარ თავთან აერთებს, უწოდებენ **ციკლურ წიბოს**.

არაორიენტირებულ (undirected) გრაფში G=(V,E) წიბოთა E სიმრავლე შედგება წვეროთა დაულაგებელი (unordered) წყვილებისაგან. ასეთ გრაფში (u,v) და (v,u) ერთ და იგივე წიბოს აღნიშნავს, ხოლო ციკლური წიბო არ შეიძლება არსებობდეს, რადგან წიბო ორი განსხვავებული წვეროსაგან უნდა შედგებოდეს.



ნახ. 7.1.

ნახ. 7.1(ა)-ზე მოცემულია ორიენტირებული გრაფი 6 წვეროთი და 8 წიბოთი (V={1,2,3,4,5,6} და E={(1,2), (2,2), (2,4), (2,5), (4,1), (4,5), (5,4), (6,3)}), ნახ. 7.1(ბ)-ზე – არაორიენტირებული გრაფი 6 წვეროთი და 4 წიბოთი (V={1,2,3,4,5,6} და E={(1,2), (1,5), (2,5), (3,6)}). (u,v) წიბოს შესახებ ორიენტირებულ გრაფში იტყვიან, რომ იგი გამოდის u წვეროდან და შედის v წვეროში. ნახ. 7.1(ა)-ზე 2 წვეროდან გამოდის სამი წიბო – (2,2), (2,4), (2,5) და 2 წვეროში შედის ორი წიბო – (1,2), (2,2). არაორიენტირებულ გრაფში (u,v) წიბოს შესახებ იტყვიან, რომ იგი u და v წვეროების **ინციდენტურია** (incident).

თუ G გრაფში არსებობს (u,v) წიბო, იტყვიან, რომ v წვერო u **წვეროს მოსაზღვრეა** (is adjacent to u). არაორიენტირებულ გრაფებში მოსაზღვრეობა სიმეტრიულია, ხოლო ორიენტირებულ გრაფებში სიმეტრიულობა აუცილებელი არ არის. თუკი ორიენტირებულ გრაფში v წვერო u წვეროს მოსაზღვრეა, წერენ u→v.

არაორიენტირებულ გრაფში წვეროს **ხარისხს** (degree) უწოდებენ ამ წვეროსადმი ინციდენტური წიბოების რაოდენობას. მაგალითად ნახ. 7.1(ბ)-ზე 2 წვეროს ხარისხია 2. ორიენტირებულ გრაფში განასხვავებენ **შემავალ** (in-degree) და **გამომავალ** (out-degree)

ხარისხებს (შესაბამისად წვეროში შემავალი და გამომავალი წიბოების რაოდენობის მიხედვით) და მათ ჯამს უწოდებენ წვეროს ხარისხს. მაგალითად, ნახ. 7.1(ა)-ზე 2 წვეროს ხარისხია 5 (შემავალი ხარისხი – 2, გამომავალი ხარისხი – 3).

k **სიგრძის გზა** (path of lengt k) u წვეროდან v წვეროში განისაზღვრება როგორც წვეროთა $\langle v_0,v_1,v_2,...,v_k\rangle$ მიმდევრობა, სადაც $v_0=u$, $v_k=v$ და $(v_{i-1},v_i)\in E$ ნებისმიერი $i=1,2,\ldots,k$-სათვის. ამრიგად k სიგრძის გზა შედგება k წიბოსაგან. იგი **შეიცავს** (contains) $v_0,v_1,v_2,..,v_k$ წვეროებს და (v_0,v_1), (v_1,v_2),...,(v_{k-1},v_k) წიბოებს. v_0 წვეროს უწოდებენ გზის **დასაწყისს**, ხოლო v_k წვეროს – გზის **ბოლოს** და ამბობენ, რომ გზა მიდის v_0-დან v_k-საკენ. თუ მოცემული u და u′ წვეროებისთვის არსებობს p გზა u-დან u′-ში, მაშინ ამბობენ, რომ u′ **მიღწევადია** u-დან p გზით (u′ is reachable from u via p). გზას ეწოდება **მარტივი** (simple), თუკი ყველა წვერო მასში განსხვავებულია. ა) ნახაზზე 3 სიგრძის მქონე გზა $\langle 1,2,5,4\rangle$ მარტივია, ხოლო იმავე სიგრძის გზა $\langle 2,5,4,5\rangle$ – არაა მარტივი.

$p=\langle v_0,v_1,v_2,...,v_k\rangle$ გზის **ქვეგზა** (subpath) ეწოდება ამ გზიდან მიყოლებით აღებულ წვეროთა $\langle v_i,v_{i+1},...,v_j\rangle$ მიმდევრობას, რომლისთვისაც $0\le i\le j\le k$. ორიენტირებულ გრაფში **ციკლი** (cycle) ეწოდება გზას, რომელშიც საწყისი წვერო ემთხვევა ბოლო წვეროს და რომელიც ერთ წიბოს მაინც შეიცავს. ციკლს ეწოდება მარტივი, თუკი მასში არ მეორდება არცერთი წვერო პირველისა და ბოლოს გარდა. ციკლური წიბო წარმოადგენს ციკლს სიგრძით 1. აიგივებენ ციკლებს, რომლებიც განსხვავდებიან მხოლოდ წანაცვლებით ციკლის გასწვრივ. მაგ. ნახ. 7.1(ა)-ზე გზები $\langle 1,2,4,1\rangle$, $\langle 2,4,1,2\rangle$ და $\langle 4,1,2,4\rangle$ წარმოადგენენ ერთ და იგივე ციკლს. ამავე ნახაზზე ციკლი $\langle 2,2\rangle$ შექმნილია ერთი წიბოთ. ორიენტირებულ გრაფს უწოდებენ **მარტივს**, თუკი იგი არ შეიცავს ციკლურ წიბოებს.

არაორიენტირებულ გრაფში $\langle v_0,v_1,v_2,...,v_k\rangle$ გზას ეწოდება **მარტივი ციკლი**, თუ $k\ge 3$, $v_o=v_k$ და ყველა $v_0,v_1,v_2,...,v_k$ წვერო განსხვავებულია. ნახ. 7.1(ბ)-ზე მარტივი ციკლია $\langle 1,2,5,1\rangle$. გრაფს, რომელშიც არაა ციკლები, **აციკლური** (acyclic) ეწოდება.

არაორიენტირებულ გრაფს ეწოდება **ბმული** (connected), თუკი წვეროთა ნებისმიერი წყვილისათვის არსებობს გზა ერთიდან მეორეში. არაორიენტირებულ გრაფში გზის არსებობა ერთი წვეროდან მეორეში წარმოადგენს ექვივალენტურ შესაბამისობას წვეროთა სიმრავლეზე. ექვივალენტობის კლასებს ეწოდებათ გრაფის **ბმული კომპონენტები** (connected components). მაგ. ნახ. 7.1(ბ)-ზე სამი ბმული კომპონენტია: {1,2,5}, {3,6} და {4}. არაორიენტირებული გრაფი ბმულია მაშინ და მხოლოდ მაშინ, როცა ის შედგება ერთადერთი ბმული კომპონენტისაგან.

ორიენტირებულ გრაფს ეწოდება **ძლიერად ბმული** (strongly connected), თუკი მისი ნებისმიერი წვეროდან მიღწევადია (ორიენტირებული გზებით) ნებისმიერი სხვა წვერო. ნებისმიერი ორიენტირებული გრაფი შეიძლება დაიყოს **ძლიერად ბმულ კომპონენტებად** (strongly connected components). ნახ. 7.1(ა)-ზე არის სამი ასეთი კომპონენტი {1,2,4,5}, {3} და {6}. შევნიშნოთ, რომ 3 და 6 წვეროები ერთად არ ჰქმნიან ძლიერად ბმულ კომპონენტს, რადგან არ არსებობს გზა 6-დან 3-ში.

G=(V,E) და G′=(V′,E′) გრაფებს ეწოდებათ **იზომორფულები** (isomorphic), თუკი არსებობს ურთიერთცალსახა შესაბამისობა $f:V\to V'$ მათი წვეროების სიმრავლეებს შორის, რომლის დროსაც $(u,v)\in E$ მაშინ და მხოლოდ მაშინ, როცა $(f(u),f(v))\in E'$. შეიძლება ითქვას, რომ იზომორფული გრაფები ეს ერთი და იგივე გრაფია, სადაც წვეროები სხვადასხვაგვარადაა დასახელებული.

G′=(V′,E′) გრაფს ეწოდება G=(V,E) გრაფის **ქვეგრაფი** (subgraph), თუ $V'\subseteq V$ და $E'\subseteq E$. თუ G=(V,E) გრაფში ავირჩევთ V′ წვეროთა ნებისმიერ სიმრავლეს, მაშინ შეგვიძლია განვიხილოთ G-ს ქვეგრაფი, რომელიც შედგება ამ წვეროებისა და მათი შემაერთებელი წიბოებისაგან. ამ ქვეგრაფს უწოდებენ G გრაფის **შეზღუდვას** V′ წვეროთა სიმრავლეზე. ნახ. 7.1(ა)-ზე მოცემული გრაფის შეზღუდვა {1,2,3,6} წვეროთა სიმრავლეზე ნაჩვენებია ნახ. 7.1(გ)-ზე და შეიცავს სამ წიბოს (1,2), (2,2), (6,3).

ნებისმიერი არაორიენტირებული გრაფისათვის შეიძლება განვიხილოთ მისი **ორიენტირებული ვარიანტი** (directed version), თუკი ყოველ არაორიენტირებულ {u,v} წიბოს შევცვლით ორიენტირებული წიბოების (u,v) და (v,u) წყვილით, რომლებსაც ექნებათ

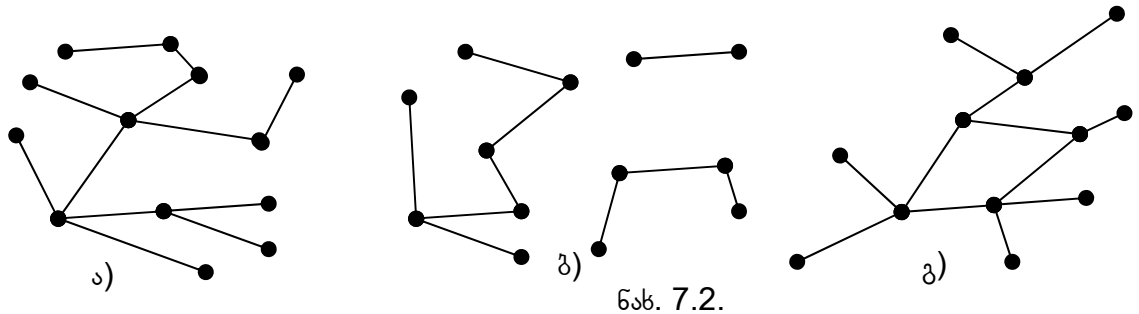
ურთიერთსაწინააღმდეგო მიმართულებები. მეორეს მხრივ, ნებისმიერი ორიენტირებული გრაფისათვის შეიძლება განვიხილოთ მისი **არაორიენტირებული ვარიანტი** (undirected version), თუკი წავშლით ციკლურ წიბოებს და (u,v) და (v,u) წიბოებს შევცვლით არაორიენტირებული {u,v}. ორიენტირებულ გრაფში u წვეროს მეზობელი (neighbor) ეწოდება ნებისმიერ წვეროს, რომელიც შეერთებულია მასთან ნებისმიერი მიმართულების წიბოთი, ე.ი. v წვერო არის u-ს მეზობელი თუ v არის u-ს მოსაზღვრე ან u არის v-ს მოსაზღვრე. არაორიენტირებულ გრაფში კი ცნებები "მეზობელი" და "მოსაზღვრე" სინონიმებია.

სრული (complete) გრაფი ეწოდება არაორიენტირებულ გრაფს, რომელიც შეიცავს ყველა შესაძლებელ წიბოს წვეროთა მოცემული სიმრავლისათვის, ე.ი. ნებისმიერი წვერო შეერთებულია ყველა დანარჩენთან. არაორიენტირებულ (V,E) გრაფს უწოდებენ **ორნაწილიანს** (bipartite), თუ V წვეროთა სიმრავლე შეიძლება გავყოთ ისეთ ორ V_1 და V_2 ნაწილად, რომ ნებისმიერი წიბოს ბოლოები სხვადასხვა ნაწილში აღმოჩნდეს. აციკლურ არაორიენტირებულ გრაფს უწოდებენ **ტყეს** (forest), ხოლო ბმულ აციკლურ არაორიენტირებულ გრაფს უწოდებენ **ხეს გამოკვეთილი ძირის გარეშე** ან **თავისუფალ ხეს** (free tree). ორიენტირებული აციკლური გრაფის (directed acyclic graph) აღსანიშნავად ზოგჯერ იყენებენ მის აბრევიატურას — dag.

განიხილავენ გრაფის განზოგადებულ ცნებებსაც. მაგალითად **მულტიგრაფი** (multigraph), რომელიც ჰგავს არაორიენტირებულ გრაფს, სადაც წვეროთა ერთი და იგივე წყვილი შეერთებულია მრავალი წიბოთი. **ჰიპერგრაფი** (hypergraph) შეიცავს ჰიპერწიბოებს, რომლებიც აერთებენ არა ორ, არამედ მრავალ წვეროს. ჩვეულებრივი გრაფების დამამუშავებელი ბევრი ალგორითმი მუშაობს ასეთ გრაფისმაგვარი სტრუქტურებისთვისაც.

7.2. ხეები

როგორც ზემოთ აღვნიშნეთ, ბმულ აციკლურ არაორიენტირებულ გრაფს უწოდებენ **ხეს გამოკვეთილი ძირის გარეშე** ან **თავისუფალ ხეს** (free tree), ხოლო აციკლურ არაორიენტირებულ გრაფს უწოდებენ **ტყეს** (forest). ტყე შედგება ხეებისაგან, რომლებიც მის ბმულ კომპონენტებს წარმოადგენენ. ხეებისათვის ვარგისი ბევრი ალგორითმი გამოსადეგია ტყეებისთვისაც.

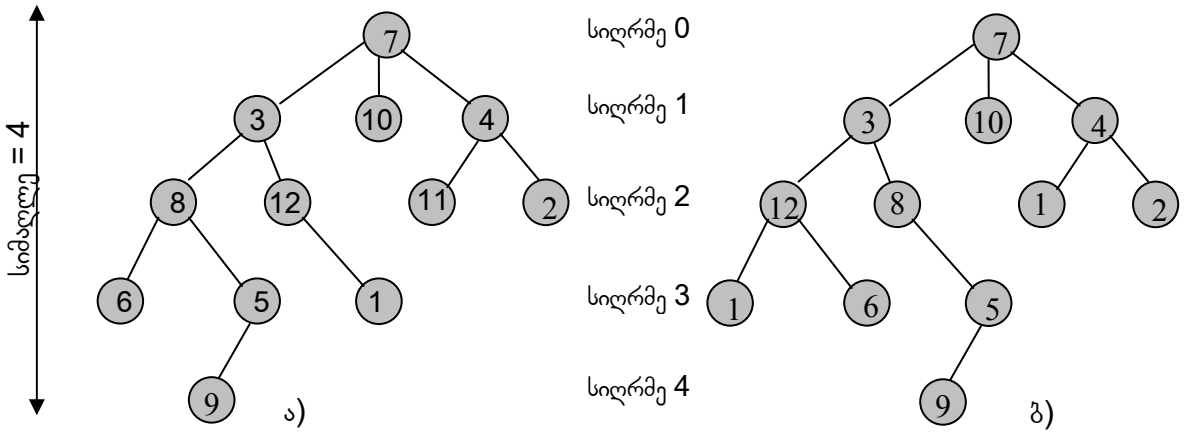


ნახ. 7.2.

ნახ.7.2(ა)-ზე გამოსახულია ხე, 7.2(ბ)-ზე – ტყე (ის არ წარმოადგენს ხეს იმის გამო, რომ ბმული არაა), ხოლო 7.2(გ) ნახაზზე მოცემული გრაფი არც ხეა და არც ტყე, რადგან იგი ციკლს შეიცავს.

თეორემა 7.1 (ხეთა თვისებები). *ვთქვათ, G=(V,E) არაორიენტირებული გრაფია. მაშინ ტოლფასია შემდეგი თვისებები: 1) G ხეა (გამოკვეთილი ძირის გარეშე); 2) G-ს ნებისმიერი ორი წვეროსათვის არსებობს მათი შემაერთებელი ერთადერთი მარტივი გზა; 3) G გრაფი ბმულია, მაგრამ კარგავს ბმულობას, თუ გამოვაკლებთ ნებისმიერ წიბოს; 4) G გრაფი ბმულია და /E/=/V/-1; 5) G გრაფი აციკლურია და /E/=/V/-1; 6) G გრაფი აციკლურია, მაგრამ ნებისმიერი წიბოს დამატებით მასში ჩნდება ციკლი.*

ხე ძირით (rooted tree) მიიღება მაშინ, როცა ბმულ აციკლურ არაორიენტირებულ გრაფში გამოყოფილია ერთ-ერთი წვერო, რომელსაც ძირს (root) უწოდებენ. ნახ.7.3-ზე ნაჩვენებია ძირის მქონე ხე 12 წვეროთი და ძირით 7.



ნახ. 7.3.

ვთქვათ x არის r ძირის მქონე ხის ნებისმიერი წვერო. არსებობს ერთადერთი გზა r-დან x-ში. ყველა წვეროს, რომელიც ამ გზაზე მდებარეობს, უწოდებენ x წვეროს **წინაპრებს** (ancestors). თუ y არის x-ის წინაპარი, მაშინ x-ს უწოდებენ y-ის **შთამომავალს**. ყოველი წვერო შეიძლება ჩაითვალოს საკუთარ წინაპრად ან შთამომავლად.

ყოველი x წვეროსათვის შეიძლება განვიხილოთ ხე, რომელიც x-ის ყველა შთამომავლისაგან შედგება და x ითვლება ამ ხის ძირად. მას უწოდებენ **ქვეხე x ძირით**. მაგალითად 7.3(ა) ნახაზზე ქვეხე ძირით 8 შეიცავს წვეროებს 8, 6, 5 და 9.

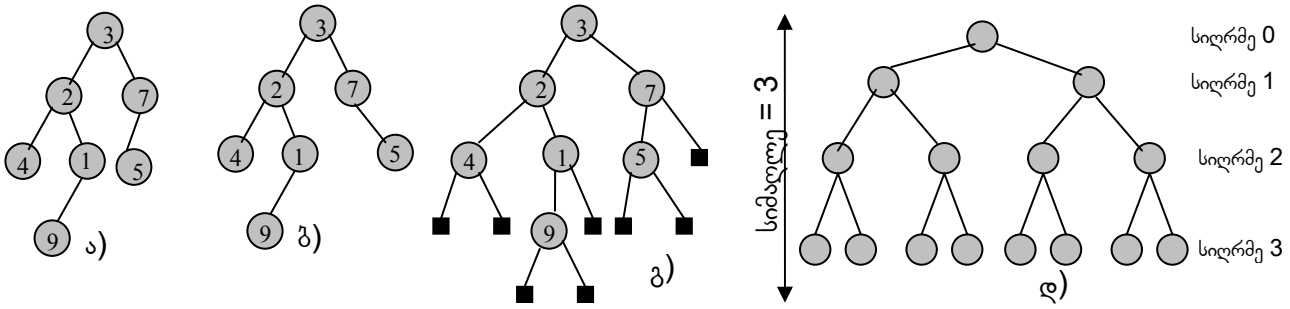
თუ (y,x) უკანასკნელი წიბოა ძირიდან x-საკენ მიმავალ გზაზე, მაშინ y-ს უწოდებენ x-ის **მშობელს** (parent), ხოლო x-ს უწოდებენ y-ის **შვილს** (child). ძირი ერთადერთი წვეროა, რომელსაც მშობელი არ გააჩნია. წვეროებს, რომელთაც საერთო მშობელი ჰყავს, უწოდებენ **დედმამიშვილებს** (siblings, რუსულ ლიტერატურაში — ძმები). ძირის მქონე ხის წვეროს, რომელსაც არა ჰყავს შვილები, უწოდებენ **ფოთოლს** (leaf, external node). წვეროებს, რომელთაც ჰყავთ შვილები, უწოდებენ **შინაგანს** (internal). ძირის მქონე ხის წვეროს შვილების რაოდენობას უწოდებენ მის **ხარისხს** (degree). შევნიშნოთ, რომ ძირის გარდა ყოველი წვეროსათვის ხარისხი იქნება ერთით ნაკლები, ვიდრე იმავე ხეში იმავე წვეროსათვის, თუკი ხეს განვიხილავთ როგორც არაორიენტირებულ გრაფს, რადგან ამ შემთხვევაში უნდა გავითვალისწინოთ ზემოთ მიმართული წიბოც. გზის სიგრძეს ძირიდან ნებისმიერ x წვერომდე უწოდებენ x წვეროს **სიღრმეს** (depth). ხის წვეროთა მაქსიმალურ სიღრმეს უწოდებენ ხის **სიმაღლეს** (height).

დალაგებული ხე (ordered tree) ეწოდება ძირის მქონე ხეს, რომელსაც დამატებითი სტრუქტურა გააჩნია: ყოველი წვეროსათვის მისი შვილების სიმრავლე დალაგებულია (ცნობილია თუ რომელია წვეროს პირველი შვილი, მეორე შვილი და ა.შ.). 7.3. ნახაზზე გამოსახული ორ ხეს ერთი და იგივე ძირი აქვს და ისინი განსხვავდებიან მხოლოდ შვილების დალაგებით.

ორობითი ხე (binary tree) ყველაზე მარტივი სახით განისაზღვრება რეკურსიულად, როგორც წვეროთა სასრული სიმრავლე, რომელიც: ან ცარიელია (არ შეიცავს წვეროებს), ან გაყოფილია სამ არაგადამკვეთ ნაწილად: წვერო რომელსაც ეწოდება **ძირი** (root), ორობითი ხე, რომელსაც ეწოდება ძირის **მარცხენა ქვეხე** (left subtree) და ორობითი ხე, რომელსაც ეწოდება ძირის **მარჯვენა ქვეხე** (right subtree)

ორობითი ხეს, რომელიც არ შეიცავს წვეროებს, ეწოდება **ცარიელი** (empty). ზოგჯერ მას აღნიშნავენ Nil-ით. თუ მარცხენა ქვეხე არაცარიელია, მაშინ მის ძირს უწოდებენ მთლიანი ხის ძირის **მარცხენა შვილს** (left child),შესაბამისად განისაზღვრება **მარჯვენა შვილიც** (right child). ორობითი ხის მაგალითი მოცემულია 7.4. ნახაზზე.

არასწორი იქნებოდა განგვესაზღვრა ორობითი ხე, როგორც დალაგებული ხე, რომელშიც თითოეული წვეროს ხარისხი არ აღემატება 2-ს. ამის მიზეზია ის, რომ ორობით ხეში მნიშვნელობა აქვს როგორია წვეროს ერთადერთი შვილი – მარცხენა თუ მარჯვენა, ხოლო დალაგებული ხისათვის ასეთი განსხვავება არ არსებობს. 7.4(ა) და 7.4(ბ) ნახაზებზე ნაჩვენები ორობითი ხეები განსხვავდებიან, რადგან ა)-ზე 5 არის 7-ის მარცხენა შვილი, ხოლო ბ)-ზე – მარჯვენა. როგორც დალაგებული ხეები ისინი ერთნაირები არიან.



ნახ. 7.4.

ხშირად ორობით ხეზე ცარიელ ადგილებს ავსებენ ფიქტიური ფოთლებით. ამის შემდეგ ყველა წვეროს ორ-ორი შვილი ჰყავს (ან თავდაპირველი, ან დამატებული). ეს გარდაქმნა ნაჩვენებია 7.4(გ) ნახაზზე.

ორობითი ხეები შეიძლება განვიხილოთ, როგორც k-ობითი ხეების კერძო შემთხვევა, როცა k=2. უფრო ზუსტად **პოზიციური ხე** (positional tree) განისაზღვრება როგორც ძირის მქონე ხე, რომელშიც ნებისმიერი წვეროს შვილები მონიშნულია სხვადასხვა მთელი დადებითი რიცხვებით, ანუ ნომრებით. ამასთან ყოველ წვეროს აქვს ვაკანსია შვილების ნომრებზე (1,2,3 და ა.შ.), რომელთაგან რაღაც სასრული რაოდენობა შევსებულია, ხოლო დანარჩენი თავისუფალი. შეგვიძლია დავასკვნათ, რომ k-ობითი ხე ეწოდება პოზიციურ ხეს, რომელსაც არ გააჩნია წვეროები k-ზე მეტი ნომრებით.

სრული k-ობითი ხე (complete k-ary tree) ეწოდება k-ობითი ხეს, რომელშიც ყველა ფოთოლს აქვს ერთნაირი სიღრმე და ყველა შინაგან წვეროს აქვს ხარისხი k. ასეთ შემთხვევაში ხის სტრუქტურა მთლიანად განისაზღვრება მისი სიმაღლით. 7.4(დ) ნახაზზე გამოსახულია სრული ორობითი ხე სიმაღლით 3. ძირი ყოველთვის არის 0 სიღრმის ერთადერთი წვერო, მისი k შვილი არის 1 სიღრმის მქონე წვეროები, რომელთა k^2 შვილი წარმოადგენენ 2 სიღრმის წვეროებს და ა.შ. h სიღრმეზე გვექნება k^h ფოთოლი. h სიმაღლის სრული k-ობითი ხის შინაგანი წვეროების რაოდენობა ტოლია

$$1 + k + k^2 + \ldots + k^{h-1} = \frac{k^h - 1}{k - 1}$$

კერძოდ, სრული ორობითი ხის შინაგანი წვეროების რაოდენობა ერთით ნაკლებია ფოთლების რაოდენობაზე.

7.3. გრაფთა წარმოდგენა

პროგრამირებაში არსებობს G=(V,E) გრაფის წარმოდგენის ორი სტანდარტული მეთოდი:

ა) **მოსაზღვრე წვეროების სიათა** (adjancency-list representation) ჩამონათვალით;

ბ) **მოსაზღვრეობის მატრიცით** (adjancency matrix).

პირველი მეთოდი უფრო მოსახერხებელია **ხალვათი** (sparse) გრაფებისათვის, სადაც $|E|$ მკვეთრად მცირეა $|V|^2$-თან შედარებით, ხოლო მეორე მეთოდი უფრო ეფექტურია **მკვრივი** (dense) გრაფებისათვის, სადაც $|E|$ უახლოვდება $|V|^2$-ს. მოსაზღვრეობის მატრიცის საშუალებით უფრო სწრაფად შეგვიძლია განვსაზღვროთ, არსებობს თუ არა წიბო ორ მოცემულ წვეროს შორის.

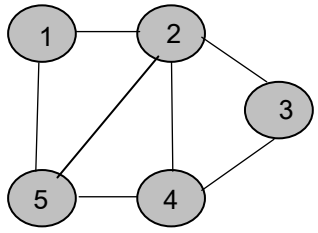
G=(V,E) გრაფის წარმოდგენა მოსაზღვრე წვეროების სიებით იყენებს $|V|$ ცალი სიისაგან შემდგარ Adj მასივს. თითოეული $u \in V$ წვეროსათვის Adj(u) შეიცავს u-ს მოსაზღვრე წვეროთა სიას ნებისმიერი მიმდევრობით.

7.5. ნახაზზე მოცემულია გრაფთა წარმოდგენის ორივე მეთოდი როგორც არაორიენტირებული, ასევე ორიენტირებული გრაფებისათვის.

მოსაზღვრე წვეროთა ყველა სიის სიგრძეთა ჯამი ორიენტირებული გრაფისათვის წიბოთა რაოდენობის ტოლია, ხოლო არაორიენტირებული გრაფისათვის — წიბოთა გაორმაგებული რაოდენობის ტოლი, რადგან (u,v) წიბო წარმოშობს ელემენტს როგორც u, ასევე v წვეროს მოსაზღვრე წვეროთა სიაში. ორივე შემთხვევაში მეხსიერების საჭირო მოცულობაა O(V+E). ამ

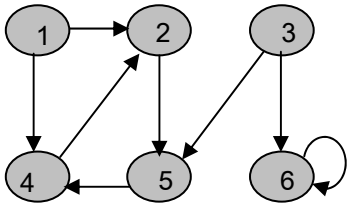
მეთოდით მოსახერხებელია **წონადი გრაფების** (weighted graphs) შენახვაც, სადაც ყოველ წიბოს მოწერილი აქვს რაღაც ნამდვილი **წონა** (weight), ანუ მოცემულია **წონითი ფუნქცია** (weight function) w: E→R. ამ შემთხვევაში მოსახერხებელია (u,v)∈E წიბოს w(u,v) წონის შენახვა v წვეროსთან ერთად u წვეროს მოსაზღვრე წვეროთა სიაში. თუმცა მეთოდს აქვს მნიშვნელოვანი ნაკლი: u-დან v-ში წიბოს არსებობის დასადგენად, საჭიროა გადავამოწმოთ მთლიანი სია Adj(u) მასში v-ს მოსაძებნად. ძებნას შესაძლოა თავი ავარიდოთ, თუ გამოვიყენებთ მოსაზღვრეობის მატრიცას, თუმცა ამ შემთხვევაში მეტი მანქანური მეხსიერებაა საჭირო.

მოსაზღვრეობის მატრიცის გამოყენებისას უნდა გადავნომროთ (V,E) გრაფის წვეროები 1,2,..., |V| რიცხვებით და განვიხილოთ |V|×|V| ზომის A=(a_{ij}) მატრიცა, სადაც a_{ij}=1, თუ (i,j) ∈E და a_{ij}=0, წინააღმდეგ შემთხვევაში. არაორიენტირებული გრაფისათვის მოსაზღვრეობის მატრიცა სიმეტრიულია მთავარი დიაგონალის მიმართ და ამიტომ საკმარისია შევინახოთ მხოლოდ მთავარი დიაგონალი და მის ზემოთ მყოფი ელემენტები, რაც თითქმის ორჯერ ამცირებს გამოყენებულ მეხსიერებას. წონადი გრაფების შენახვა პრობლემა არც ამ მეთოდისთვისაა — (u,v) წიბოს w(u,v) წონა მატრიცაში თავსდება u სტრიქონისა და v სვეტის გადაკვეთაზე, ხოლო წიბოს არარსებობის შემთხვევაში მატრიცის შესაბამისი ელემენტი მიიღებს ცარიელ მნიშვნელობას Nil (ზოგ ამოცანაში შეიძლება გამოვიყენოთ 0 ან ∞).



1	2	5		
2	1	5	3	4
3	2	4		
4	2	5	3	
5	4	1	2	

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



11	2	4
2	5	
3	6	5
4	2	
5	4	
66	6	

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

ნახ. 7.5.

თუკი მანქანური მეხსიერება საკმარისია, სჯობს გრაფი აღვწეროთ მოსაზღვრეობის მატრიცის საშუალებით, რადგან უმეტეს შემთხვევაში მასთან მუშაობა უფრო მოსახერხებელია. ამას გარდა, თუკი გრაფი წონადი არ არის, მოსაზღვრეობის მატრიცის ელემენტები შესაძლოა განიხილოთ როგორც ბიტები, რომლებიც შეგვიძლია გავაერთიანოთ მანქანურ სიტყვებში — ეს იძლევა მეხსიერების მნიშვნელოვან ეკონომიას.

7.4. განივად ძებნის ალგორითმი.

ვთქვათ მოცემულია G=(V,E) გრაფი და მისი s **საწყისი წვერო** (source vertex). **განივად ძებნის** (breadth-first search) ალგორითმი ადგენს უმოკლეს მანძილს s-დან ყველა მიღწევად წვერომდე (მანძილად აქ ითვლება უმოკლეს გზაზე წიბოთა რაოდენობა). ალგორითმის მუშაობის პროცესში მიიღება ე.წ. ძებნის ხე, რომელიც მოიცავს ძირიდან მიღწევად ყველა წვეროს. ალგორითმი გამოიყენება როგორც ორიენტირებული, ასევე არაორიენტირებული გრაფებისათვის.

მეტი თვალსაჩინოებისათვის ჩავთვალოთ, რომ ალგორითმის მუშაობის პროცესში გრაფის წვეროები შესაძლოა იყოს თეთრი, რუხი ან შავი ფერის. თავდაპირველად ყველა წვერო თეთრი ფერისაა, ხოლო ალგორითმის მუშაობის დროს თეთრი წვერო შეიძლება გადაიქცეს რუხად, ხოლო რუხი – შავად (მაგრამ არა პირიქით). რუხად იღებება ის წვერო, რომელიც ალგორითმმა აღმოაჩინა, მაგრამ ჯერ არაა შემოწმებული მისგან გამომავალი სხვა წიბოები, ხოლო შავად

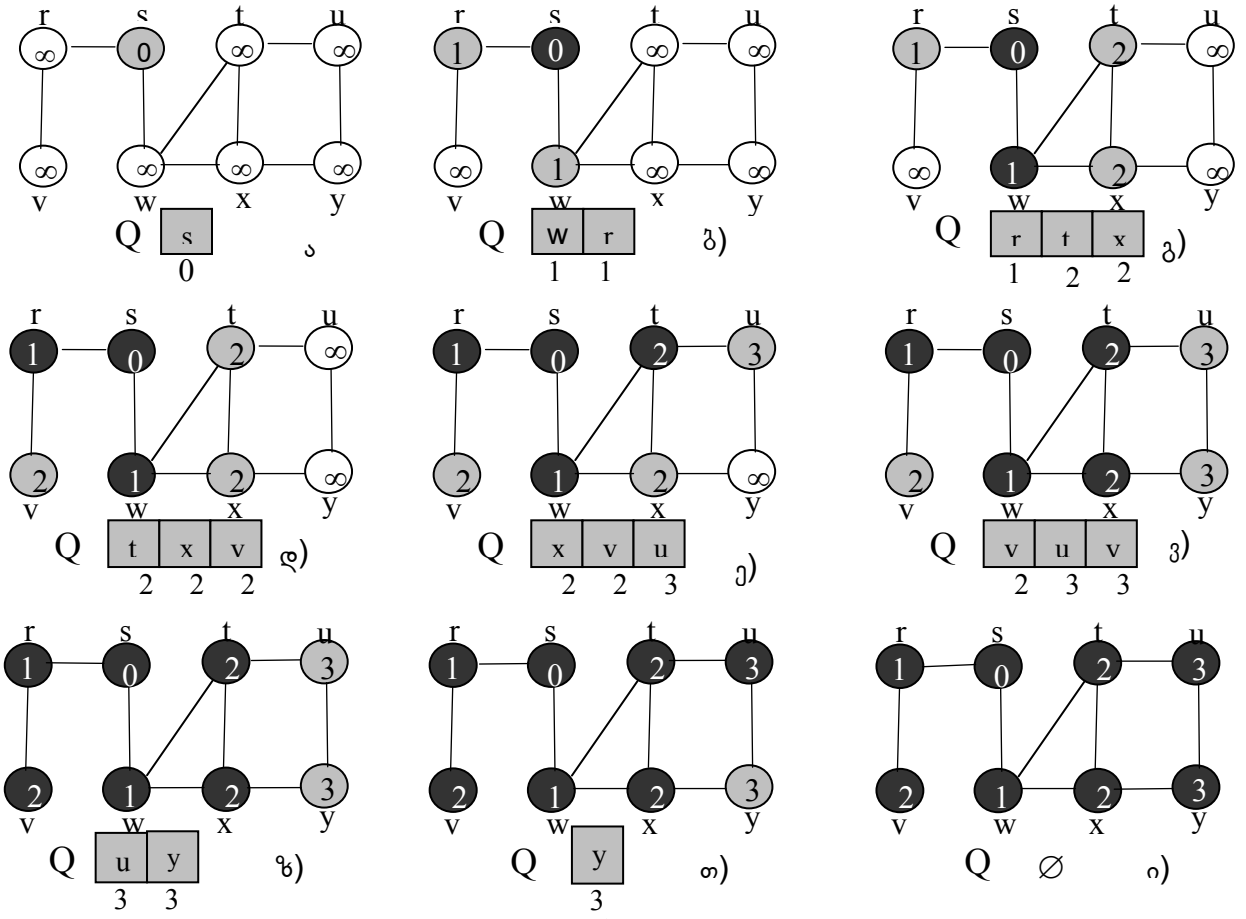
იღებება ის წვერო, რომლისგანაც გამომავალი ყველა წიბო უკვე შემოწმებულია, ანუ შეღებილია რუხად ან შავად.

თავიდან ძებნის ხე შედგება მხოლოდ ძირისაგან. როცა ალგორითმი იპოვის ახალ (თეთრი ფერის) v წვეროს, რომელიც ესაზღვრება უკვე ნაპოვნ u წვეროს, (u,v) წიბო ემატება ძებნის ხეს, v წვერო ხდება u წვეროს **შვილი** (child) და იღებება რუხად, u წვერო ხდება v წვეროს **მშობელი** (parent) და იღებება შავად, თუკი მისი ყველა შვილი ნაპოვნია. ყოველა წვეროს აღმოჩენა ხდება მხოლოდ ერთხელ, ამიტომ წვეროს არ შეიძლება ერთზე მეტი მშობელი ჰყავდეს. "წინაპრისა" და "შთამომავლის" ცნებებიც ამ შემთხვევაში ჩვეულებრივად განსაზღვრება (მაგ. ძირი ნებისმიერი სხვა წვეროს წინაპარია, წვეროდან ძირისკენ მოძრაობისას ამ წვეროს ყველა წინაპარი უნდა გავიაროთ და ა.შ.)

პროცედურა BFS (breadth-first search – განივად ძებნა) იყენებს გრაფის წარმოდგენას მოსაზღვრე წვეროთა სიებით. ყოველი u წვეროსათვის დამატებით ინახება მისი ფერი color[u] და მისი მშობელი π[u]. თუკი მშობელი ჯერ ნაპოვნი არ არის ან u=s, მაშინ π[u]=Nil. მანძილი s-დან u-მდე იწერება d[u] მინდორში. რუხი წვეროების შესანახად გამოიყენება რიგი Q.

```
BFS(G,s)
1  for ∀ u∈V[G]-⟨s⟩   {
2        color[u]=თეთრი
3        d[u]=∞
4        π[u]=Nil     }
5  color[s]=რუხი;   d[s]=0;   π[s]=Nil;   Q=⟨s⟩
6  while Q≠∅   {
7  u=head[Q]
8            for ∀ v∈Adj[u]   {
9                     if color[v]=თეთრი then  { color[v]=რუხი
10                                            d[v]=d[u]+1
11                                            π[v]=u
12                                            ENQUEUE(Q,v)   }
13                    DEQUEUE(Q)
14                    color[u]=შავი       }       }
```

1-4 სტრიქონებში ყველა წვეროსათვის ფერი ხდება თეთრი, მნიშვნელობები – უსასრულობა, ხოლო მშობლები – Nil. მე-5 სტრიქონში ხდება ძირის (s წვეროს) დამუშავება და Q რიგში იწერება მისი პირველი წევრი – s წვერო. პროგრამის ძირითადი ციკლი (6-14 სტრიქონები) სრულდება Q რიგის დაცარიელებამდე, ე.ი. სანამ არსებობენ რუხი ფერის წვეროები, ანუ წვეროები, რომლებიც აღმოჩენილია, მაგრამ რომელთა მოსაზღვრეობის სიები ჯერ განხილული არ არის. მე-7 სტრიქონში პირველი ასეთი წვერო თავსდება u-ში. 8-12 სტრიქონებში for ციკლი განიხილავს u-ს ყველა მეზობელ წვეროს. თუკი მათ შორის აღმოჩნდება თეთრი ფერის წვერო, ის იღებება რუხად, მის მშობლად ცხადდება u და მანძილად ძირამდე – d[u]+1, ხოლო თავად ეს წვერო იწერება Q რიგის ბოლოში. ამის შემდეგ u წვერო ამოიშლება Q რიგიდან და იღებება შავად (13-14 სტრიქონები).



ნახ. 7.6.

7.6. ნახაზზე მოცემულია BFS პროცედურის მუშაობა არაორიენტირებული გრაფისათვის.

განვსაზღვროთ პროცედურის მუშაობის დრო. ყოველი წვერო რიგში თავსდება მხოლოდ ერთხელ და ასევე ერთხელ ამოიღება რიგიდან, ამიტომ რიგთან დაკავშირებულ ოპერაციებზე დაიხარჯება O(V). მოსაზღვრე წვეროთა სია ასევე ერთხელ განიხილება, როცა შესაბამისი წვერო ამოიღება რიგიდან. მოსაზღვრე წვეროთა სიებში ელემენტთა ჯამური სიგრძე კი |E|-ს ტოლია (არაორიენტირებულ გრაფში 2|E|), ამიტომ ამ ოპერაციაზე დაიხარჯება O(E). ინიციალიზაციას სჭირდება O(V) დრო. მაშასადამე, ალგორითმის მუშაობის საერთო დრო იქნება O(V+E).

განივად ძებნა პოულობს მანძილებს ძირიდან გრაფის თითოეულ მიღწევად წვერომდე. მანძილში აქ იგულისხმება **უმოკლესი გზის სიგრძე** (shortest-path distance). გზის მინიმალური სიგრძე s ძირიდან v წვერომდე აღვნიშნოთ δ(s,v)-თი (გზის სიგრძე — მასში წიბოების რაოდენობა). δ(s,v) სიგრძის გზებს s ძირიდან v წვერომდე ეწოდებათ **უმოკლესი გზები** (shortest paths). ასეთი გზა შეიძლება რამდენიმე იყოს. ქვემოთ განხილული იქნება უმოკლესი გზების უფრო ზოგადი სახე, როცა საქმე გვაქვს წონად წიბოებთან და გზის სიგრძე წიბოების წონათა ჯამის ტოლია. ჩვენს შემთხვევაში კი წიბოთა წონები ერთეულის ტოლად ითვლება და გზის სიგრძე წიბოთა რაოდენობას უდრის. ამგვარად განსაზღვრული მანძილისათვის ადგილი აქვს თეორემას:

თეორემა 7.2. *s ძირის მქონე G=(V,E) გრაფისათვის პროცედურა BFS იპოვის (შეღებავს შავად) s-დან მიღწევად ყველა წვეროს და ყველა v∈V-სათვის შესრულდება ტოლობა d[v]=δ(s,v). ამას გარდა, s-დან მიღწევადი ნებისმიერი v≠s-თვის ერთ-ერთი უმოკლესი გზა s-დან v-ში შეგვიძლია მივიღოთ (π[v],v) წიბოს დამატებით ნებისმიერ უმოკლეს გზაზე s-დან π[v]-მდე.*

გამოვიყენოთ BFS პროცედურა s ძირის მქონე G=(V,E) გრაფისათვის და განვიხილოთ ქვეგრაფი, რომელიც მიიღება s-დან მიღწევადი ყველა წვეროსა და (π[v],v) წიბოებით.

ლემა 7.3. *ასეთი წესით აგებული G გრაფის G_π ქვეგრაფი წარმოადგენს ხეს, რომელშიც ყოველი v წვეროსათვის არსებობს ერთადერთი მარტივი გზა s-დან v-ში. ეს გზა არის უმოკლესი გზა s-დან v-ში G გრაფში.*

G_π ქვეგრაფს უწოდებენ **წინსვლის ქვეგრაფს** (predecessor subgraph), ან **განივად ძებნის ხეს** (breadt-first tree) მოცემული გრაფისათვის მოცემული ძირით. შევნიშნოთ, რომ აგებული ხე დამოკიდებულია იმაზე, თუ როგორი თანმიმდევრობით განიხილება წვეროები მოსაზღვრე წვეროთა სიებში.

თუკი π-ის მნიშვნელობები გამოთვლილია BFS პროცედურის მეშვეობით, მაშინ ძირიდან უმოკლესი მანძილების გამოთვლა იოლია — მათ ბეჭდავს პროცედურა PRINT-PATH:

```
PRINT-PATH(G,s,v)
1 IF v=s THEN
2  {  დავბეჭდოთ: s   }
3  ELSE {  IF π[v]=Nil THEN
4                { დავბეჭდოთ: "გზა" s "-დან" v "-ში არ არსებობს"  }
5                ELSE {  PRINT-PATH(G,s, π[v])
6                        დავბეჭდოთ v    }   }
```

პროცედურის მუშაობის დრო დასაბეჭდი გზის სიგრძის პროპორციულია, რადგან ყოველი რეკურსიული გამოძახება ერთი ერთეულით ამცირებს გზას ძირემდე.

7.5. სიღრმეში ძებნა

სიღრმეში ძებნა (depth-first search) ხორციელდება ასეთი სტრატეგიით: ვიაროთ გრაფში წინ ("სიღრმეში"), სანამ არსებობს გაუვლელი წიბო. თუკი გაუვლელი წიბო აღარ არსებობს, დავბრუნდეთ უკან და ვეძებოთ სხვა გზა. ასე გავაგრძელოთ მანამ, სანამ არ ამოიწურება ყველა წვერო, რომელიც მიღწევადია საწყისიდან. თუკი გაუვლელი წვერო მაინც დარჩა, ავიღოთ ერთ-ერთი მათგანი და გავიმეოროთ პროცესი, სანამ გამოვლენილი არ იქნება გრაფის ყველა წვერო.

განივად ძებნის მსგავსად, როცა პირველად გამოვლინდება u წვეროს მოსაზღვრე v წვერო, u-ს მნიშვნელობა თავსდება π[v]-ში მიიღება ხე (ან ხეები – თუკი ძებნა განმეორდება რამდენიმე წვეროდან. მიიღება წინსვლის ქვეგრაფი, რომელიც ასე განისაზღვრება: $G_\pi=(V,E_\pi)$, სადაც $E_\pi=\{(\pi[v],v):v\in V \text{ და } \pi[v]\neq Nil\}$. წინსვლის ქვეგრაფი წარმოადგენს სიღრმეში ძებნის ტყეს, რომელიც შედგება სიღრმეში ძებნის ხეებისაგან.

სიღრმეში ძებნის ალგორითმიც იყენებს წვეროების შეფერვას. თავიდან ყველა წვერო თეთრია. წვეროს პოვნის შემდეგ იგი ხდება რუხი ფერის. წვერო ხდება შავი, თუკი იგი მთლიანად დამუშავებულია, ანუ თუ მისი მოსაზღვრე წვეროების სია ბოლომდე განხილულია. ყოველი წვერო ხვდება სიღრმეში ძებნის მხოლოდ ერთ ხეში (ამიტომ ეს ხეები არ გადაიკვეთებიან). გარდა ამისა სიღრმეში ძებნა თითოეულ წვეროს მიუწერს დროის ჭდეებს. ყოველ წვეროს აქვს ორი ჭდე: d[u]-ში ჩაიწერება, თუ როდის იქნა ნაპოვნი ეს წვერო (და გახდა რუხი), ხოლო f[u]-ში — როდის დასრულდა წვეროს დამუშავება (და გახდა შავი).

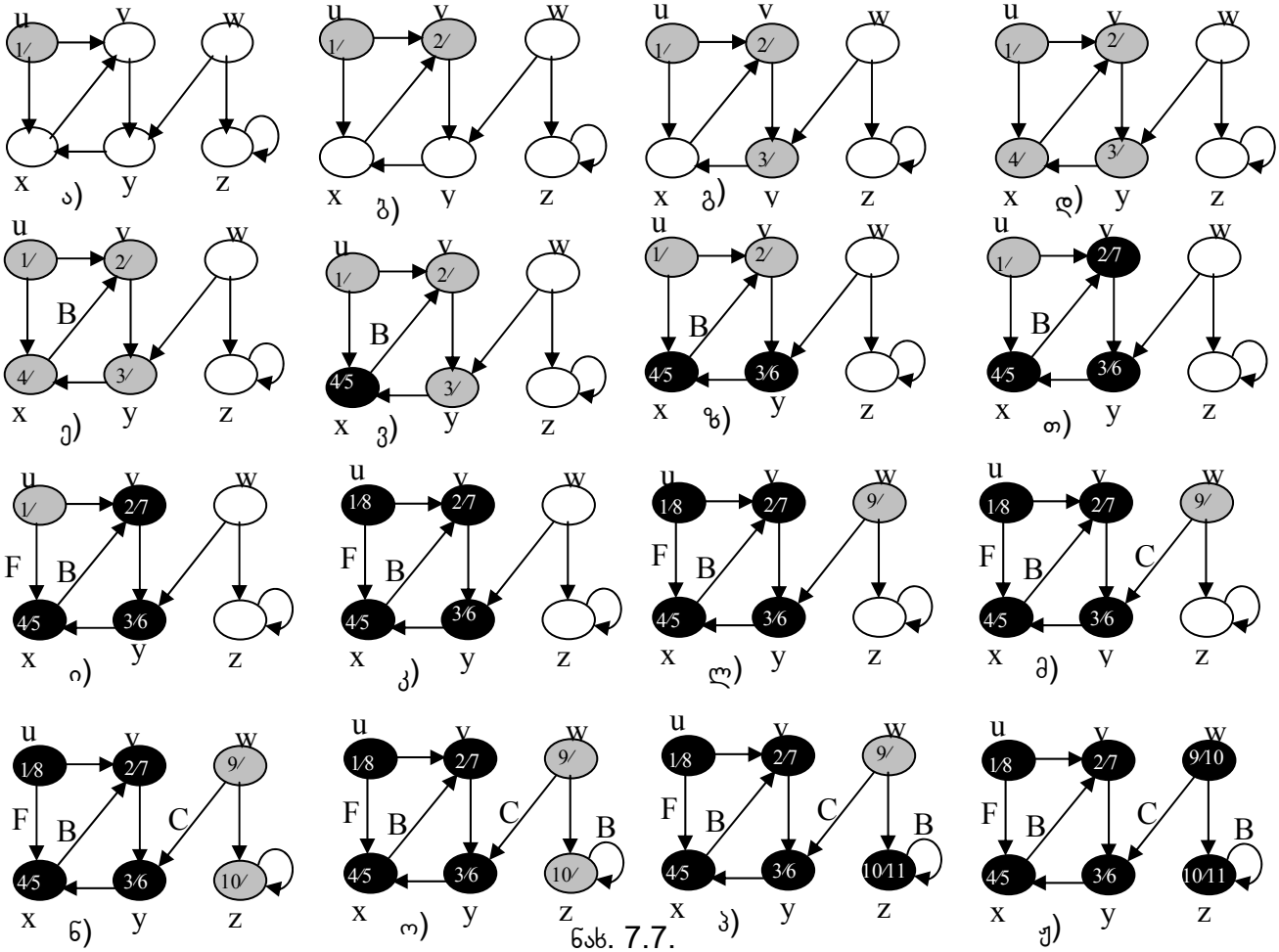
ქვემოთ მოყვანილ DFS (Depth-First Search) პროცედურაში d[u] და f[u] წარმოადგენენ მთელ რიცხვებს 1-დან $2|V|$-მდე. ნებისმიერი წვეროსათვის სრულდება უტოლობა d[u]<f[u]. u წვერო იქნება თეთრი d[u] მომენტამდე; d[u]-სა და f[u]-ს შორის იქნება რუხი, ხოლო f[u]-ის შემდეგ შავი.

საწყისი გრაფი შეიძლება იყოს ორიენტირებულიც და არაორიენტირებულიც. მიმდინარე დროის time ცვლადი გლობალურია და გამოიყენება ჭდეებისათვის.

```
DFS(G)
1 for ∀ u∈V[G] {
2          color[u]=ÈÄÈÒÉ
3          π[u]=Nil    }
4 time=0
5 for ∀ u∈V[G]  {
6          if color[u]=ÈÄÈÒÉ {
```

```
7                   then DFS-VISIT(u)  }   }

DFS-VISIT(u)
1 color[u]= რუხი
2 d[u]=time=time+1
3 for ∀ v∈Adj[u]  {
4          if  color[v]=თეთრი
5                   then  { π[v]=u
6                             DFS-VISIT(v)   }      }
7 color[v]= შავი
8 f[u]=time=time+1
```



ნახ. 7.7.

7.7. ნახაზზე მოცემულია DFS პროცედურის მუშაობა ორიენტირებული გრაფისათვის.

1-3 სტრიქონებში ყველა წვერო ხდება თეთრი, ხოლო π-ს მიენიჭება Nil. მე-4 სტრიქონში განისაზღვრება საწყისი (ნულოვანი) დრო. 5-7 სტრიქონებში გამოიძახება პროცედურა DFS-VISIT იმ წვეროებისათვის, რომლებიც თეთრი ფერის დარჩნენ გამოძახების მომენტისათვის (პროცედურის წინა გამოძახებამ ისინი შეიძლება შავი გახადოს). ეს წვეროები წარმოადგენენ სიღრმეში ძებნის ხეთა ძირებს.

DFS-VISIT(u) პროცედურის გამოძახებისას u წვერო თეთრია. პირველ სტრიქონში ის რუხი ხდება. მე-2 სტრიქონში მისი პოვნის დრო შეიტანება d[u]-ში (წინასწარ დროის მთვლელი 1-ით იზრდება). 3-6 სტრიქონებში განიხილება u-ს მოსაზღვრე წვეროები და DFS-VISIT პროცედურა გამოიძახება მათ შორის იმ წვეროებისათვის, რომლებიც თეთრი ფერის არიან გამოძახების მომენტში.

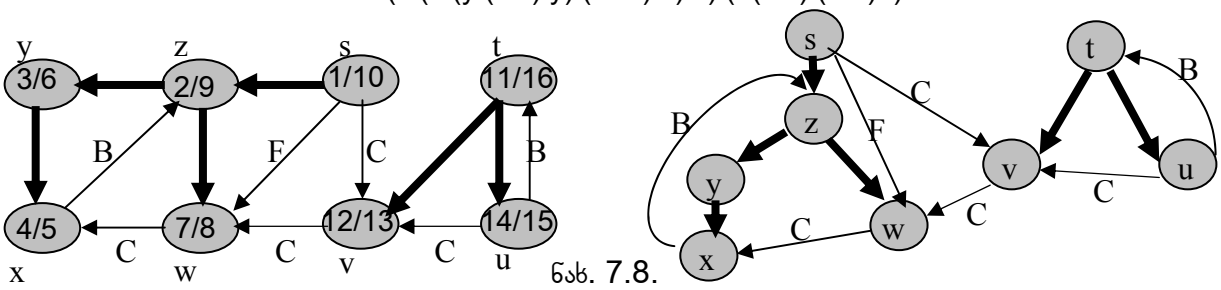
DFS პროცედურის მუშაობის დროა θ(V+E).

DFS ალგორითმის მუშაობა ორიენტირებული გრაფისათვის. უკუწიბოები აღნიშნულია B (back), ჯვარედინი – C (cross), პირდაპირი – F (forward).

შევნიშნოთ, რომ სიღრმეში ძებნის ხეებისაგან შედგენილი წინსვლის ქვეგრაფი ზუსტად შეესაბამება DFS-VISIT პროცედურის რეკურსიული გამოძახებების სტრუქტურას. სახელდობრ, u=π[v] მაშინ და მხოლოდ მაშინ, როცა მოხდა DFS-VISIT(v) გამოძახება u წვეროს მოსაზღვრე წვეროებისგანხილვის დროს.

სიღრმეში ძებნის მეორე მნიშვნელოვანი თვისება იმაში მდგომარეობს, რომ წვეროთა პოვნისა და დამუშავების დამთავრების დროები ჰქმნიან წესიერ ფრჩხილურ სტრუქტურას. აღვნიშნოთ u წვეროს პოვნა მარცხენა ფრჩხილითა და წვეროს სახელით – "(u", ხოლო მისი დამუშავების დამთავრება წვეროს სახელითა და მარჯვენა ფრჩხილით – "u)". მოვლენათა ჩამონათვალი იქნება ფრჩხილებისაგან წესიერად აგებული გამოსახულება. მაგალითად ნახაზზე გამოსახული გრაფისათვის:

(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)



ნახ. 7.8.

ადგილი აქვს მტკიცებულებებს:

თეორემა 7.4 (ფრჩხილური სტრუქტურის შესახებ). *სიღრმეში ძებნის დროს ორიენტირებულ ან არაორიენტირებულ G=(V,E) გრაფში ნებისმიერი ორი u და v წვეროსათვის სრულდება მხოლოდ ერთ-ერთი დებულება შემდეგი სამიდან: ა) [d[u], f[u]] და [d[v], f[v]] მონაკვეთები არ გადაიკვეთება; ბ) [d[u], f[u]] მონაკვეთი მთლიანად შედის [d[v], f[v]] მონაკვეთის შიგნით და u წვერო v-ს შთამომავალია სიღრმეში ძებნის ხეში; გ) [d[v], f[v]] მონაკვეთი მთლიანად შედის [d[u], f[u]] მონაკვეთის შიგნით და v წვერო u-ს შთამომავალია სიღრმეში ძებნის ხეში;*

შედეგი 7.5. *v წვერო u-ს შთამომავალია სიღრმეში ძებნის ტყეში ორიენტირებულ ან არაორიენტირებულ G=(V,E) გრაფში მაშინ და მხოლოდ მაშინ, როცა d[u]<d[v]<f[v]<f[u].*

თეორემა 7.6 (თეთრი გზის შესახებ). *v წვერო u-ს შთამომავალია სიღრმეში ძებნის ტყეში ორიენტირებულ ან არაორიენტირებულ G=(V,E) გრაფში მაშინ და მხოლოდ მაშინ, თუ d[u] დროის მომენტისათვის, როცა u წვერო ნაპოვნია, არსებობს გზა u-დან v-ში, რომელიც შედგება მხოლოდ თეთრი წვეროებისაგან.*

გრაფის წიბოები იყოფა რამდენიმე კატეგორიად იმის მიხედვით, თუ რა როლს თამაშობენ ისინი სიღრმეში ძებნის დროს. ეს კლასიფიკაცია სასარგებლოა სხვადასხვა ამოცანების განხილვისას. მაგალითად, ორიენტირებულ გრაფს არ გააჩნია ციკლები მაშინ და მხოლოდ მაშინ, როცა სიღრმეში ძებნის ალგორითმი ვერ პოულობს მასში "უკუწიბოებს".

ვთქვათ, ჩვენ ჩავატარეთ სიღრმეში ძებნა G გრაფში და მივიღეთ ტყე G_π.

1. ხის წიბოები (tree edges) — ესაა G_π გრაფის წიბოები. (u,v) წიბო იქნება ხის წიბო, თუ v წვერო ნაპოვნია ამ წიბოს დამუშავების დროს.

2. უკუწიბოები (back edges) — ესაა (u,v) წიბოები, რომელიც აერთებს u წვეროს მის v წინაპართან სიღრმეში ძების ხეზე. ორიენტირებული გრაფებისათვის დამახასიათებელი ციკლური წიბოები უკუწიბოებად ითვლებიან

3. პირდაპირი (წინმართი) წიბოები (forward edges) — აერთებენ წვეროს მის შთამომავალთან, მაგრამ არ შედიან სიღრმეში ძებნის ხეში.

4. ჯვარედინი წიბოები (cross edges) — გრაფის ყველა სხვა წიბო. მათ შეუძლიათ შეაერთონ სიღრმეში ძებნის ხის ორი ისეთი წვერო, რომელთა შორის არც ერთი არაა მეორის წინაპარი ან წვეროები, რომლებიც სხვადასხვა ხეებს ეკუთვნიან.

7.8(ბ) ნახაზზე გრაფი მოცემულია იმდაგვარად, რომ ხის წიბოები და პირდაპირი წიბოები მიემართებიან ქვემოთ, ხოლო უკუწიბოები — ზემოთ.

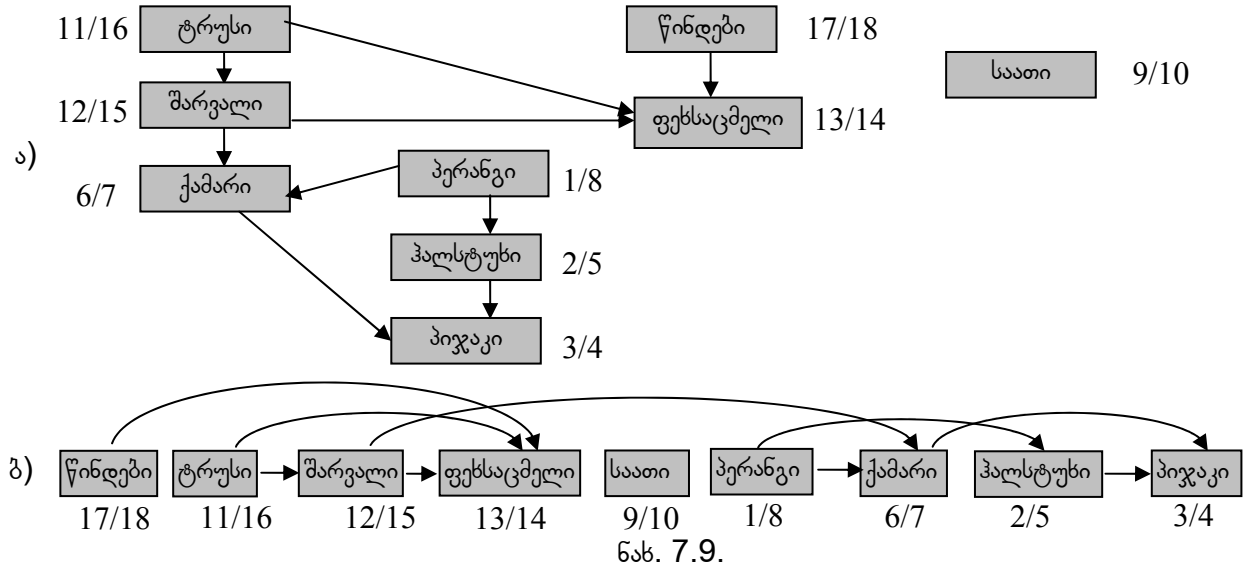
DFS ალგორითმით შესაძლებელია წიბოთა კლასიფიკაცია. (u,v) წიბოს ტიპი შეიძლება განისაზღვროს წვეროს ფერით პირველი დამუშავების დროს, ოღონდ პირდაპირ და ჯვარედინ წიბოებს შორის განსხვავების პოვნა არ ხერხდება: თეთრი ფერი ნიშნავს, რომ ეს ხის წიბოა, რუხი — უკუწიბო, შავი — პირდაპირი ან ჯვარედინი წიბო. ამ უკანასკნელთა ერთმანეთისაგან განსასხვავებლად შეგვიძლია გამოვიყენოთ d-ს მნიშვნელობა: თუ d[u]>d[v], მაშინ (u,v) წიბო პირდაპირია, ხოლო თუ d[u]<d[v] — ჯვარედინი.

არაორიენტირებული გრაფი საჭიროებს განსაკუთრებულ განხილვას, რადგან ერთი და იგივე წიბო (u,v)=(v,u) ორჯერ უნდა დამუშავდეს (თითოჯერ ორივე ბოლოდან) და შესაძლოა სხვადასხვა კატეგორიაში მოხვდეს. ამ შემთხვევაში ის უნდა მივაკუთვნოთ იმ კატეგორიას, რომელიც ჩვენს ჩამონათვალში უფრო წინ დგას. იმავე შედეგს მივიღებთ, თუკი ჩავთვლით, რომ წიბოს ტიპი განისაზღვრება მისი პირველი დამუშავებისას და არ იცვლება მეორე დამუშავებით. ირკვევა, რომ ასეთი შეთანხმებისას პირდაპირი და ჯვარედინი წიბოები არაორიენტირებულ გრაფში არ იქნება და ადგილი აქვს თეორემას.

თეორემა 7.7. *სიღრმეში ძებნისას არაორიენტირებულ G გრაფში ნებისმიერი წიბო ან ხის წიბოა, ან უკუწიბო.*

ტოპოლოგიური სორტირება. ვთქვათ, მოცემულია ორიენტირებული გრაფი ციკლების გარეშე (directed acyclic graph, ზოგჯერ აღნიშნავენ შემოკლებით — "dag"). ამ გრაფის **ტოპოლოგიური სორტირება** (topological sort) გულისხმობს, რომ უნდა მივუთითოთ წვეროების ისეთი წრფივი თანმიმდევრობა, რომ ნებისმიერი წიბო მიმართული იყოს ამ თანმიმდევრობაში ნაკლები ნომრის მქონე წვეროდან მეტი ნომრის მქონე წვეროსაკენ. ცხადია, რომ თუკი გრაფი შეიცავს ციკლს, ასეთი თანმიმდევრობა არ იარსებებს. ამოცანა შეგვიძლია სხვაგვარადაც დავსვათ: განვალაგოთ გრაფის წვეროები ჰორიზონტალურ წრფეზე ისე, რომ ყველა წიბო მიმართული იყოს მარცხნიდან მარჯვნივ (სიტყვა "სორტირება" აქ მისი პირდაპირი მნიშვნელობით არ იგულისხმება).

მაგალითისათვის განვიხილოთ ასეთი შემთხვევა: დაბნეული პროფესორისათვის დილემადაა ქცეული დილაობით ჩაცმა. მან ზოგიერთი რამის ჩაცმისას აუცილებლად უნდა დაიცვას მიმდევრობა (მაგ. წინდები და ფეხსაცმელი), ზოგ შემთხვევაში კი მიმდევრობას მნიშვნელობა არა აქვს (მაგ. წინდები და შარვალი). ნახაზზე ეს დამოკიდებულებანი მოცემულია ორიენტირებული გრაფის სახით. (u,v) წიბო აღნიშნავს, რომ u უნდა იქნას ჩაცმული v-ზე ადრე. ამ გრაფის ტოპოლოგიური სორტირების ერთ-ერთი ვარიანტი მოცემულია ნახაზ 7.9-ზე.



ნახ. 7.9.

წვეროების გვერდით მითითებულია მათი დამუშავების დაწყებისა და დამთავრების დროები. ნახაზის ბ) ნაწილში გრაფი ტოპოლოგიურად სორტირებულია. წვეროები დალაგებულია დამუშავების დამთავრების დროთა მიხედვით. ყველა წიბო მიმართულია მარცხნიდან მარჯვნივ.

შემდეგი ალგორითმი ტოპოლოგიურად ალაგებს ორიენტირებულ აციკლურ გრაფს.

TOPOLOGICAL-SORT(G)

1 გამოვიძახოთ DFS(G), ამასთან

2 წვეროს დამუშავების დამთავრებისას (DFS-VISIT, სტრ.8), დავამატოთ იგი სიის დასაწყისში

3 გამოვიტანოთ ამ წესით მიღებული წვეროების სია

ტოპოლოგიური სორტირება სრულდება $\theta(V+E)$ დროში, რადგან ამდენი დრო სჭირდება სიღრმეში ძებნას, ხოლო სიაში წვეროს ჩაწერას სჭირდება $O(1)$ დრო. ალგორითმის სისწორე მტკიცდება შემდეგი ლემის დახმარებით:

ლემა 7.8. *ორიენტირებული გრაფი არ შეიცავს ციკლებს მაშინ და მხოლოდ მაშინ, როცა სიღრმეში ძებნა ვერ პოულობს მასში უკუწიბოებს.*

სიღრმეში ძებნის ალგორითმის გამოყენების კლასიკური მაგალითია გრაფის ძლიერად ბმულ კომპონენტებად დაშლა. ორიენტირბულ გრაფებზე მომუშავე მრავალი ალგორითმი იწყება გრაფში ძლიერად ბმულ კომპონენტების მოძებნით. ამის შემდეგ ამოცანა იხსნება ცალკეული კომპონენტებისათვის, ხოლო შემდეგ ხდება კომბინირება ამ კომპონენტთა კავშირების შესაბამისად.

გავიხსენოთ, რომ $G=(V,E)$ ორიენტირებული გრაფის ძლიერად ბმულ კომპონენტი ეწოდება $U \subset V$ წვეროთა მაქსიმალურ სიმრავლეს, სადაც ნებისმიერი ორი u და v წვერო $(u,v \in U)$ ერთმანეთისაგან მიღწევადია. $G=(V,E)$ გრაფის ძლიერად ბმული კომპონენტების ძებნის ალგორითმი იყენებს "ტრანსპონირებულ" $G^T=(V,E^T)$ გრაფს, რომელიც მიიღება საწყისი გრაფიდან წიბოთა მიმართულებების შებრუნებით. ასეთის გრაფი შეიძლება აიგოს $O(V+E)$ დროში (ნაგულისხმევია, რომ ორივე გრაფი მოიცემა მოსაზღვრე წვეროთა სიების სახით). ცხადია, რომ G და G^T გრაფებს ერთი და იგივე ძლიერად ბმული კომპონენტები აქვთ. შემდეგი ალგორითმი პოულობს ძლიერად ბმულ კომპონენტებს $G=(V,E)$ ორიენტირებულ გრაფში სიღრმეში ძებნის ორჯერ გამოყენებით — G და G^T გრაფებისათვის. ალგორითმის მუშაობის დროა $O(V+E)$.

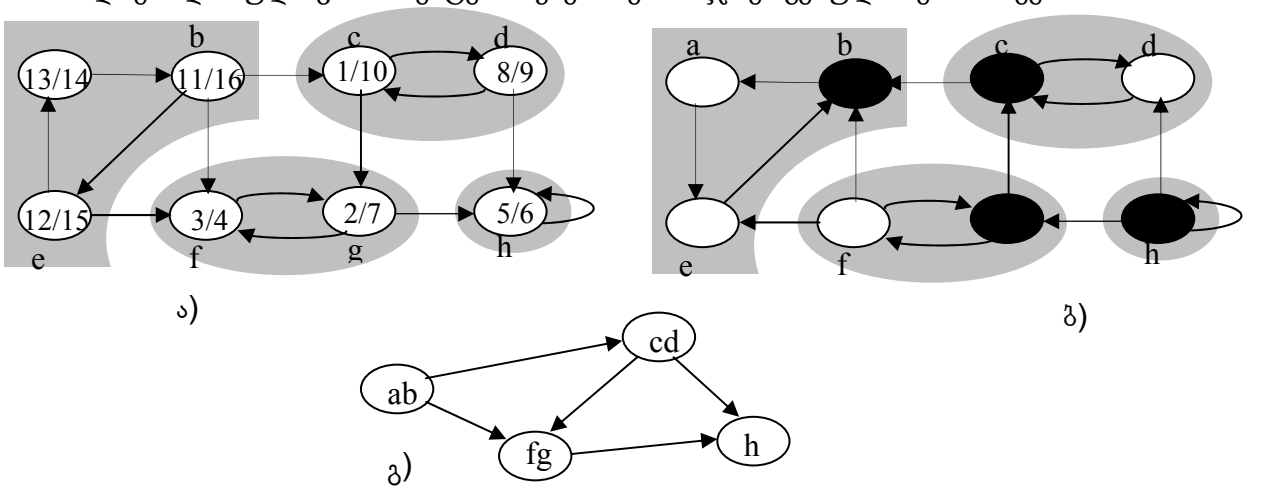
STRONGLY-CONNECTED-COMPONENTS(G)

1 DFS(G)-ს გამოყენებით ვიპოვოთ ყოველი u წვეროსთვის დამუშავების დამთავრების დრო $f[u]$

2 ავაგოთ G^T

3 გამოვიძახოთ DFS(G^T), ამასთან მის გარე ციკლში გადავარჩიოთ წვეროები $f[u]$ სიდიდის კლებადობის მიხედვით (რომელიც 1 სტრიქონში გამოითვალა)

4 ძლიერად ბმული კომპონენტები იქნება მე-3 ბიჯზე აგებული ძებნის ხეები



ნახ. 7.10.

ა) ნახაზზე რუხი ფონით აღნიშნულია G გრაფის ძლიერად ბმული კომპონენტები, ნაჩვენებია აგრეთვე სიღრმეში ძებნის ტყე და დროის ჭდეები G გრაფისათვის. ბ) ნახაზზე მოცემულია G^T გრაფის სიღრმეში ძებნის ხე, რომელიც პროცედურის მე-3 სტრიქონში გამოითვლება. b, c, g, h წვეროები წარმოადგენენ სიღრმეში ძებნის ხეთა ძირეებს G^T

გრაფისათვის და შეფერილი არიან შავად. გ) ნახაზზე მოცემულია აციკლური გრაფი, რომელიც მიიღება G გრაფის ძლიერად ბმული კომპონენტების წერტილებამდე შეკუმშვით.

ალგორითმის გასაანალიზებლად დაგვჭირდება ორი ასეთი დებულება:

ლემა 7.9. *თუ ორი წვერო ეკუთვნის ერთ ძლიერად ბმული კომპონენტს, მაშინ არანაირი გზა არ გამოდის ამ კომპონენტის საზღვრებიდან.*

თეორემა 7.10. *სიღრმეში ძებნისას ერთი ძლიერად ბმული კომპონენტის წვეროები ხვდებიან ერთ და იგივე ხეში.*

გრაფის ყოველი u წვეროსათვის განვსაზღვროთ მისი წინაპირველი (forefather) φ(u), როგორც u-დან მიღწევადი w წვეროებიდან ისეთი, რომლისთვისაც დამუშავება დამთავრდა ყველაზე გვიან: φ(u) წარმოადგენს ისეთ w წვეროს, რომლისთვისაც არსებობს გზა u-დან w-ში და f[w] მაქსიმალურია. აქვე შევნიშნოთ, რომ შესაძლოა φ(u)=u. რადგან ნებისმიერი წვერო მიღწევადია საკუთარი თავიდან — f[u]≤f[φ(u)] და φ(φ(u))= φ(u). ადილი აქვს შემდეგ დებულებებს:

თეორემა 7.11. ორიენტირებულ G=(V,E) გრაფში ნებისმიერი u∈V წვეროს წინაპირველი φ(u) წარმოადგენს მის წინაპარს სიღრმეში ძებნის ხეზე.

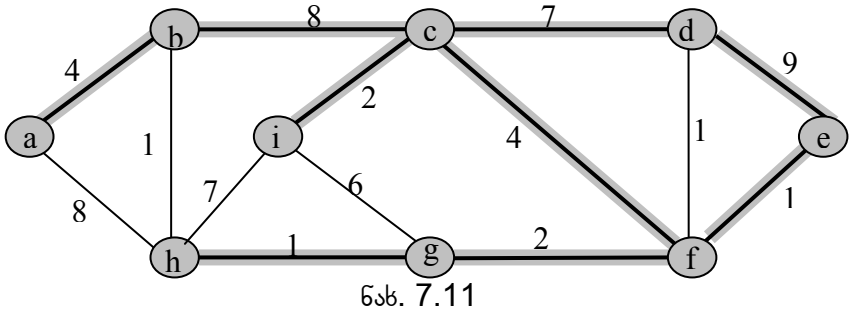
შედეგი 7.12. ორიენტირებულ G=(V,E) გრაფში სიღრმეში ნებისმიერად ძებნისას u და φ(u) ეკუთვნიან ერთ და იმავე ძლიერად ბმულ კომპონენტს ნებისმიერი u∈V წვეროსათვის.

თეორემა 7.13. ორიენტირებულ G=(V,E) გრაფში ორი წვერო ეკუთვნის ერთ და იმავე ძლიერად ბმულ კომპონენტს მაშინ და მხოლოდ მაშინ, როცა მათ ჰყავთ საერთო წინაპირველი სიღრმეში ძებნისას.

ამ დებულებებიდან გამომდინარეობს, ძლიერად ბმული კომპონენტების მოძებნა დადის გრაფის ყველა წვეროს წინაპირველების მოძებნის ამოცანაზე. სწორედ ამისათვის გამოიყენება სიღრმეში ძებნა STRONGLY-CONNECTED-COMPONENTS პროცედურის მესამე სტრიქონში.

7.6. მინიმალური დამფარავი ხეები

ვთქვათ, მოცემულია ბმული არაორიენტირებული G=(V,E) გრაფი. გრაფის ყოველი (u,v) წიბოსათვის მოცემულია w(u,v) არაუარყოფითი წონა. ვიპოვოთ ისეთი T⊂E ქვესიმრავლე, რომელიც მოიცავს ყველა წვეროს და რომლისთვისაც ჯამური წონა $w(T) = \sum_{(u,v)\in T} w(u,v)$ მინიმალურია. ასეთი ქვესიმრავლე შეგვიძლია ხედ ჩავთვალოთ. G გრაფის ბმულ ქვეგრაფს, რომელიც ხეს წარმოადგენს და შეიცავს მის ყველა წვეროს, ეწოდება ამ გრაფის **დამფარავი ხე** (spanning tree). ზოგჯერ იყენებენ ტერმინს "ხის ჩონჩხი". ჩვენ განვიხილავთ ამოცანას **მინიმალური დამფარავი ხის** (minimum-spanning-tree problem) შესახებ. აქ სიტყვა "მინიმალური" აღნიშნავს "მინიმალურ შესაძლო წონას". შევნიშნოთ, რომ თუკი განვიხილავთ მხოლოდ ხეებს, მაშინ წონათა არაუარყოფითობის პირობა შეგვიძლია უგულებელვყოთ, რადგან ყველა დამფარავ ხეში წიბოთა ერთნაირი რაოდენობაა და შეგვიძლია ერთი და იგივე სიდიდით გავზარდოთ ყველა წონა, რაც მათ დადებითად აქცევს.



ნახ. 7.11

7.11 ნახაზზე მოცემულია ბმული გრაფისა და მისი მინიმალური დამფარავი ხის მაგალითი, რომლის წიბოებიც გამოყოფილია. მინიმალური დამფარავი ხის ჯამური წონაა 37 და ასეთი ხე ერთი არაა. თუკი (b,c) წიბოს შევცვლით (a,h) წიბოთი, მივიღებთ სხვა დამფარავ ხეს იმავე ჯამური წონით.

ჩვენ განვიხილავთ მინიმალური დამფარავი ხის პოვნის ორ ხერხს: პრიმისა და კრასკალის ალგორითმებს. ორივე მათგანი რეალიზდება O(E*log*V) დროში, თუკი გამოვიყენებთ ორობით გროვებს, ხოლო ფიბონაჩის გროვების გამოყენების შემთხვევაში პრიმის ალგორითმის მუშაობის დრო მცირდება O(E+V*log*V). დროის ეკონომია მნიშვნელოვანია, თუ |V| ბევრად ნაკლებია |E|-ზე. ორივე ალგორითმი იყენებს "ხარბ" სტრატეგიას – ეძებს "ლოკალურად საუკეთესო" ვარიანტს მუშაობის ყოველ ბიჯზე.

ჩვენი ალგორითმების ზოგადი სქემა ასეთია: საძებნი დამფარავი ხე აიგება თანდათანობით – თავდაპირველად ცარიელ A სიმრავლეს ყოველ ბიჯზე ემატება თითო წიბო. A სიმრავლე ყოველთვის წარმოადგენს მინიმალური დამფარავი ხის ქვესიმრავლეს. მორიგ ბიჯზე დამატებული (u,v) წიბო იმგვარად ამოირჩევა, რომ არ დაირღვეს ეს თვისება, ე.ი. $A\cup\{(u,v)\}$ ასევე უნდა იყოს მინიმალური დამფარავი ხის ქვესიმრავლე. ასეთ წიბოს უწოდებენ **უსაფრთხო წიბოს** (safe edge) A-სათვის.

```
GNENERIC-MST(G,w)
1 A=∅
2 while სანამ A არ არის დამფარავი ხე {
3          მოვძებნოთ (u,v) უსაფრთხო წიბო A-სათვის
4                    A=A∪⟨ (u,v) ⟩  }
5 return A
```

ცხადია, რომ მთავარი პრობლემა მე-3 სტრიქონში უსაფრთხო წიბოს მოძებნაა, მაგრამ მანამდე შევნიშნოთ, რომ ასეთი წიბო გარანტირებულად არსებობს, რადგან თუ A არის მინიმალური დამფარავი ხის ქვესიმრავლე, მაშინ მინიმალური დამფარავი ხის ნებისმიერი წიბო, რომელიც არ შედის A-ში უსაფრთხოა. გარდა ამისა, ციკლის ნებისმიერი იტერაციისას შენარჩუნებულია თვისება — "A სიმრავლე წარმოადგენს რომელიღაც მინიმალური დამფარავი ხის ქვესიმრავლეს", ამიტომ მე-5 სტრიქონში ალგორითმი აუცილებლად დააბრუნებს მინიმალურ დამფარავ ხეს.

ვიდრე უსაფრთხო წიბოს მოძებნის ალგორითმებს განვიხილავდეთ, განვსაზღვროთ რამდენიმე ტერმინი.

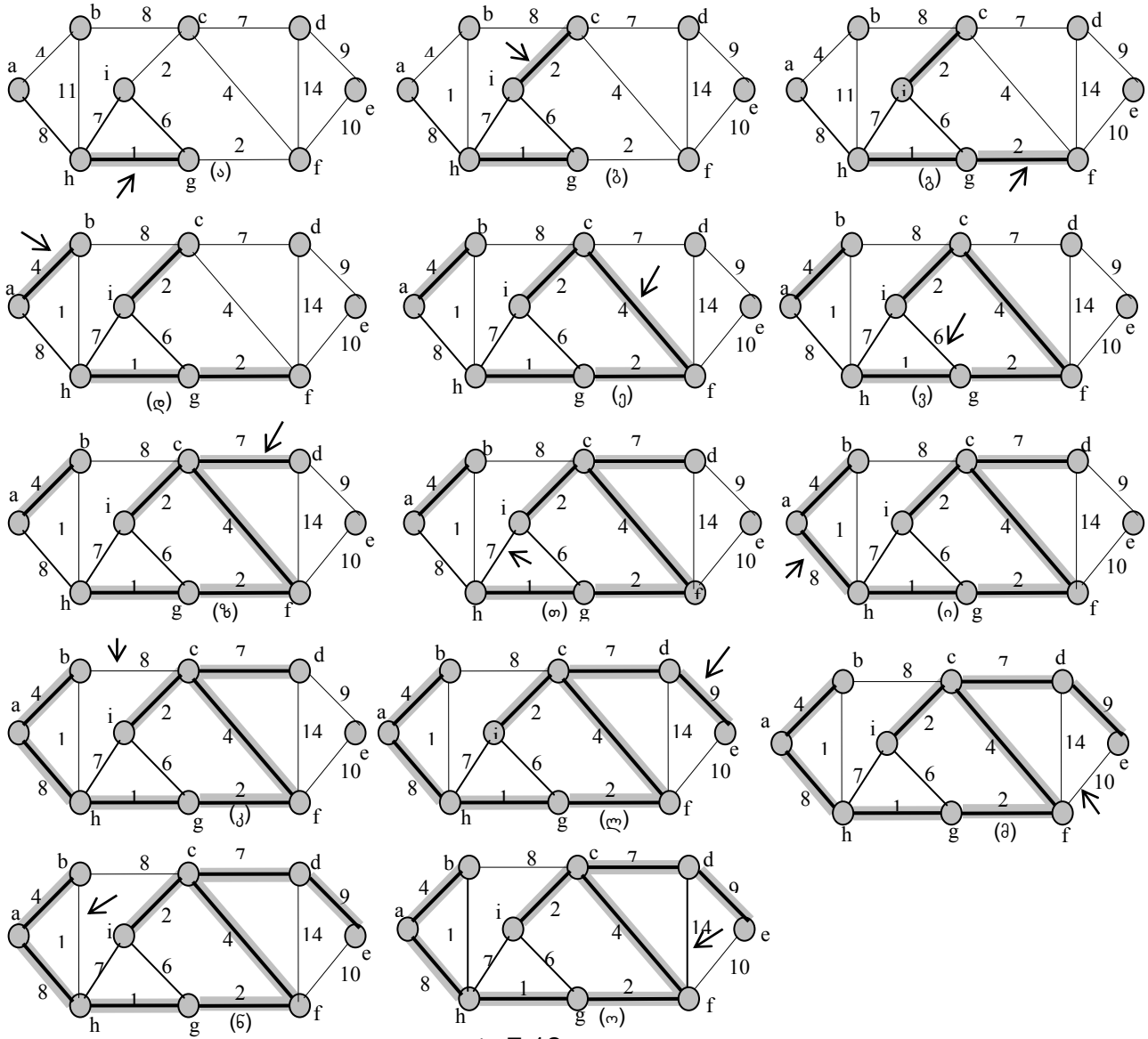
G=(V,E) არაორიენტირებული გრაფის (S,V\S) **ჭრილი** (cut) ეწოდება მისი წვეროთა სიმრავლის გაყოფას ორ ქვესიმრავლედ.

იტყვიან, რომ $(u,v)\in E$ წიბო **კვეთს** (crosses) (S,V\S) ჭრილს, თუ მისი ერთი ბოლო ეკუთვნის S-ს, ხოლო მეორე ბოლო — (V\S)-ს. ჭრილი **შეთანხმებულია წიბოთა A სიმრავლესთან** (respects the set A), თუ A-დან არცერთი წიბო არ კვეთს ამ ჭრილს. ჭრილის მიერ გადაკვეთილ წიბოთა სიმრავლეში გამოყოფენ უმცირესი წონის წიბოს, რომელსაც **მსუბუქს** (light edges) უწოდებენ.

თეორემა 7.14. *ვთქვათ G=(V,E) ბმული არაორიენტირებული გრაფია და მის წვეროთა სიმრავლეზე განსაზღვრულია ნამდვილი w ფუნქცია. ვთქვათ A წიბოთა სიმრავლეა, რომელიც წარმოადგენს G გრაფის რომელიმე მინიმალური დამფარავი ხის ქვესიმრავლეს. ვთქვათ (S,V\S) წარმოადგენს G გრაფის ისეთ ჭრილს, რომელიც შეთანხმებულია A-სთან, ხოლო (u,v) წიბო ამ ჭრილის მსუბუქი წიბოა. მაშინ (u,v) წიბო წარმოადგენს უსაფრთხო წიბოს A-სათვის.*

შედეგი 7.15. *ვთქვათ G=(V,E) ბმული არაორიენტირებული გრაფია და წვეროთა E სიმრავლეზე განსაზღვრულია წონითი w ფუნქცია. ვთქვათ A წიბოთა სიმრავლეა, რომელიც წარმოადგენს G გრაფის რომელიმე მინიმალური დამფარავი ხის ქვესიმრავლეს. განვიხილოთ ტყე $G_A=(V,A)$ და ვთქვათ C ხე მისი ერთერთი ბმული კომპონენტია. განვიხილოთ გრაფის ყველა წიბო, რომელიც აერთებს C-ს წვეროებს C-ს გარეთ მყოფ წვეროებთან და ავარჩიოთ მათგან ყველაზე ნაკლები წონის წიბო. მაშინ ეს წიბო უსაფრთხოა A-სთვის.*

კრასკალის ალგორითმი. კრასკალის ალგორითმის მუშაობის ნებისმიერ მომენტში A ამორჩეულ წიბოთა სიმრავლე (დამფარავი ხის ნაწილი) არ შეიცავს ციკლებს. ის აერთებს გრაფის წვეროებს რამდენიმე ბმულ კომპონენტად, რომელთაგან თითოეული წარმოადგენს ხეს. ყველა წიბოს შორის, რომელიც აერთებს წვეროებს სხვადასხვა კომპონენტებისგან, ვირჩევთ უმცირესი წონის წიბოს. ზემოთ მოყვანილი თეორემის (ან შედეგის) თანახმად, ის უსაფრთხოა. ნახ. 7.12.-ზე ნაჩვენებია თუ როგორ მუშაობს ალგორითმი.



ნახ. 7.12

ალგორითმი MST-KRUSKAL იყენებს სამ პროცედურას:

MAKE-SET(x) (“შევქმნათ სიმრავლე”) — პროცედურას გადაეცემა მიმთითებელი უკვე არსებულ x ობიექტზე. ის ჰქმნის ახალ სიმრავლეს, რომლის ერთადერთი ელემენტი თავდაპირველად მხოლოდ x-ია (ამიტომ x იქნება წარმომადგენელიც). რადგან სიმრავლეები არ უნდა გადაიკვეთონ, მოითხოვება, რომ x მიუთითებდეს ახალ ობიექტს, რომელსაც არ შეიცავს უკვე არსებული არცერთი სიმრავლე.

FIND-SET(x) (“მოვძებნოთ სიმრავლე”) — პროცედურა იძლევა იმ სიმრავლის წარმომადგენელს, რომელიც x-ის მიერ მითითებულ ელემენტს შეიცავს.

UNION(x,y) — გამოიყენება, როცა x და y ელემენტები ორი სხვადასხვა სიმრავლის წევრები არიან. ხდება ამ ორი სიმრავლის გაერთიანება და მისი ელემენტებიდან ამოირჩევა წარმომადგენელი. ძველი სიმრავლეები წაიშლება.

ზემოთ თქმულიდან გამომდინარე, გრაფის ორი u და v წვერო ეკუთვნის ერთ სიმრავლეს (ანუ კომპონენტს), როცა FIND-SET(u)=FIND-SET(v).

```
MST-KRUSKAL(G,w)
1 A=∅
2 for ∀ v∈V[G]  {
3              MAKE-SET(v)  }
4 დავალაგოთ E წიბოები წონების მიხედვით
5 for ყველა (u,v)∈E-თვის (წონის ზრდის მიხედვით)   {
6              if FIND-SET(u)≠FIND-SET(v)
```

```
7                then  {  A=A∪⟨(u,v)⟩
8                          UNION(u,v) }          }
9 return A
```

ალგორითმის 1-3 სტრიქონებში A სიმრავლე ცარიელია და გვაქვს |V| ხე, სადაც ყოველი მათგანი შეიცავს თითო წვეროს. მე-4 სტრიქონში წიბოები დალადგებიან წონათა არაკლებადობის მიხედვით. 5-8 სტრიქონებში ვამოწმებთ, ეკუთვნის თუ არა წიბოს წვეროები ერთ და იმავე ხეს. თუ ეკუთვნის, არ შეიძლება ამ წიბოს დამატება ტყისათვის (რათა ციკლი არ შეიქმნეს) და წიბო უკუიგდება, ხოლო თუ არ ეკუთვნის, მაშინ წიბო ემატება A-ს (მე-7 სტრიქონი) და ამ წიბოთი დაკავშირებული ორი ხე ერთიანდება.

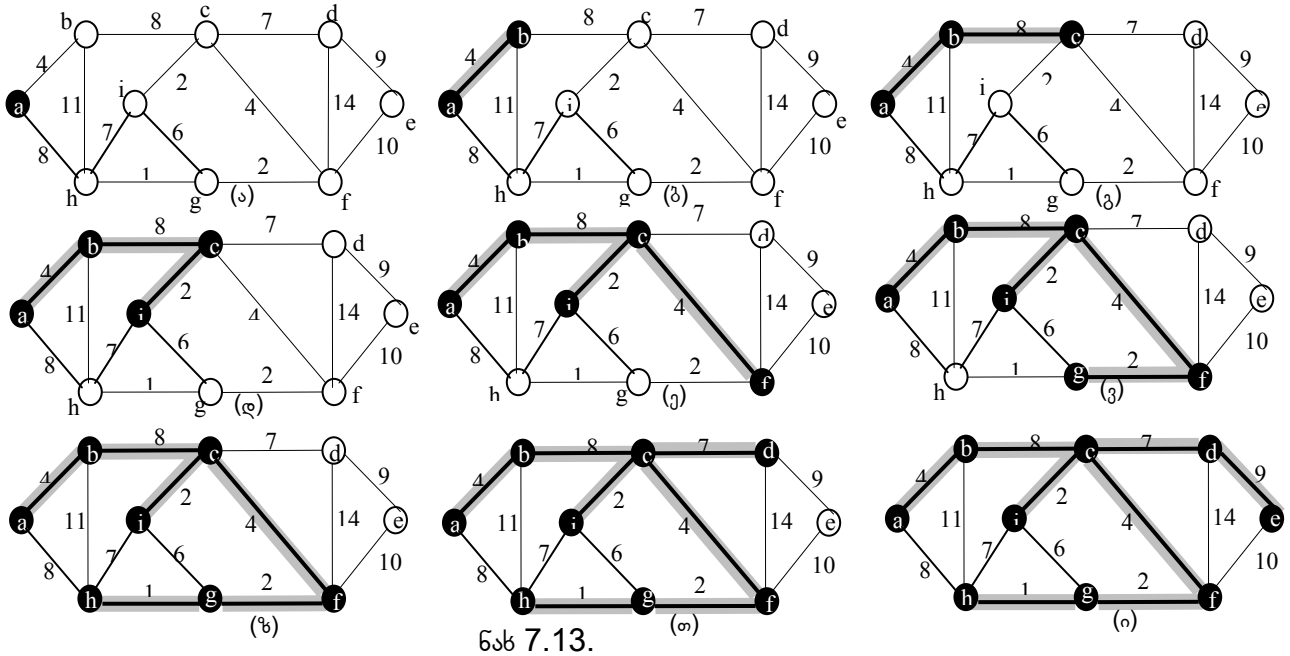
ალგორითმის მუშაობის დროა O(E*log*E) (დრო ძირითადად იხარჯება სორტირებაზე).

პრიმის ალგორითმი. პრიმის ალგორითმით მინიმალური დამფარავი ხის ფორმირება იწყება ნებისმიერი r წვეროდან. ყოველ ბიჯზე ემატება უმცირესი წონის წიბო იმ წიბოებს შორის, რომლებიც წვეროს აერთებენ ხის არაწევრ წვეროებთან. 7.15 შედეგის მიხედვით ასეთი წიბო უსაფრთხოა A-სათვის, ე.ი. მიიღება მინიმალური დამფარავი ხე.

პროცედურისათვის შემავალი მონაცემებია: ბმული G გრაფი, წიბოთა w წონები და r ძირი. რეალიზაციისას მნიშვნელოვანია სწრაფად ავარჩიოთ მსუბუქი წიბო. ალგორითმის მუშაობისას ყველა წვერო, რომელიც ჯერ არ გამხდარა ხის წევრი, ინახება პრიორიტეტებიან რიგში. v წვეროს პრიორიტეტი განისაზღვრება key[v] მნიშვნელობით, რომელიც იმ წიბოებისაგან მინიმალურ წონას, რომელიც აერთებს v-ს A ხესთან. თუ ასეთი წიბო არ არსებობს — key[v]=∞. π[v] მინდორი ხის წვეროებისთვის მიუთითებს მშობელს, ხოლო სხვა წვეროებისათვის ხის წვეროს, რომლისკენაც მივყავართ key[v] წონის წიბოს (თუ ასეთი წიბო რამდენიმეა, მაშინ მიეთითება ერთ-ერთი).

```
MST-PRIM(G,w,r)
1 Q=V[G]
2 for ∀ u∈Q   {
3          key[u]=∞   }
4 key[r]=0;  π[r]=Nil
5 while Q≠∅   {
6          u=EXTSCT-MIN(Q)
7                for ∀ v∈Adj[v]   {
8                      if v∈Q AND w(u,v)<key[v]
9                            then  {  π[v]=u
10                                key[v]=w(u,v)   }  }  }
```

1-4 სტრიქონებში ხდება მასივთა ინიციალიზაცია. 5-10 სტრიქონებში მოცემული ციკლის პირველი გავლისას ხე შედგება ერთადერთი r წვეროსაგან. ყველა დანარჩენი წვერო იმყოფება რიგში. key[v]-ს მნიშვნელობა მათთვის r-დან v-ში წიბოს სიგრძის ან უსასრულობის (თუკი წიბო არ არსებობს) ტოლია. ალგორითმის მუშაობა მოცემულია ნახ. 7.13-ზე.



ნახ 7.13.

ალგორითმის მუშაობის დრო დამოკიდებულია Q რიგის რეალიზაციაზე. თუკი გამოყენებულია ორობითი გროვა, მაშინ მუშაობის დრო კრასკალის ალგორითმის ანალოგიურია — O(E*log*V), ხოლო ფიბონაჩის გროვის გამოყენების შემთხვევაში შეფასება მცირდება O(E+V*log*V)-მდე.

7.7. უმოკლესი გზები ერთი წვეროდან

როგორ ვიპოვოთ საავტომობილო რუკაზე უმოკლესი მარშრუტი ორ ქალაქს შორის? შეიძლება განვიხილოთ ყველა შესაძლო მარშრუტი, დავითვალოთ თითოეულის სიგრძე და ამოვარჩიოთ უმცირესი. იმ შემთხვევაშიც კი, თუკი არ განვიხილავთ ციკლებს, ჩვენ მოგვიწევს შევამოწმოთ უამრავი უვარგისი ვარიანტი. ამ და მომდევნო თავებში განვიხილავთ ასეთი ამოცანების ამოხსნის ეფექტურ გზებს.

უმოკლესი გზის ამოცანაში (shortest-paths problem) ჩვენ მოცემული გვაქვს ორიენტირებული წონადი G=(V,E) გრაფი ნამდვილი წონადი w:E→R ფუნქციით. $p=\langle v_0,v_1,\ldots,v_k\rangle$ გზის **წონას** (weight) უწოდებენ ამ შემავალი ყველა წიბოს ჯამს

$$w(p)=\sum_{i=1}^{k} w(v_{i-1},v_i)$$

u-დან v-ში **უმოკლესი გზის წონა** (shortest-paths weight) განსაზღვრის თანახმად ტოლია

$$\delta(u,v)=\begin{cases}\min\{w(p)\}, & \text{თუ არსებობს გზა u - დან v - ში}\\ \infty, & \text{სხვა შემთხვევაში}\end{cases}$$

უმოკლესი გზა (shortest-paths) u-დან v-ში — ესაა ნებისმიერი p გზა u-დან v-ში, რომლისთვისაც w(p)=δ(u,v). წონებში შეიძლება ვიგულისხმოთ არა მარტო მანძილები, არამედ დრო, ღირებულება, ჯარიმა, ზარალი და ა.შ. განივად ძებნის ალგორითმი (იხ. თავი 7.4.) წონითი ფუნქციის გარეშე, შეგვიძლია განვიხილოთ, როგორც უმოკლესი გზების შესახებ ამოცანის ამოხსნის კერძო შემთხვევა, როცა თითოეული წიბოს წონა 1-ის ტოლია.

განიხილავენ უმოკლესი გზების ამოცანის სხვადასხვა ვარიანტს. ამ თავში ჩვენ განვიხილავთ ამოცანას **უმოკლესი გზების შესახებ ერთი წვეროდან** (single-source shortest-path problem): მოცემული გვაქვს ორიენტირებული წონადი G=(V,E) გრაფი და საწყისი წვერო s (source vertex). საჭიროა ვიპოვოთ უმოკლესი გზები s-დან ყველა v∈V წვეროში. ამ ამოცანის ამოხსნის ალგორითმი გამოიყენება სხვა ამოცანების ამოსახსნელადაც, კერძოდ:

უმოკლესი გზა ერთი წვეროსაკენ: მოცემულია საბოლოო t წვერო (destination vertex). საჭიროა მოვძებნოთ უმოკლესი გზები t-საკენ ყოველი v∈V წვეროდან. თუკი შევაბრუნებთ

მიმართულებას ყველა წიბოზე, ეს ამოცანა დაიყვანება ამოცანაზე უმოკლესი გზების შესახებ ერთი წვეროდან.

უმოკლესი გზა წვეროთა მოცემული წყვილისათვის: მოცემულია u და v წვეროები, მოვძებნოთ უმოკლესი გზა u-დან v-ში. რა თქმა უნდა, თუკი ჩვენ მოვძებნით ყველა უმოკლეს გზას u-დან, ამოცანა ამოიხსნება. უნდა აღინიშნოს, რომ უფრო სწრაფი მეთოდი (რომელიც გამოიყენებდა იმ ფაქტს, რომ უმოკლესი გზა მხოლოდ ორ წვეროს შორისაა მოსაძებნი) ჯერჯერობით ნაპოვნი არ არის.

უმოკლესი გზები წვეროთა ყველა წყვილისათვის: წვეროთა ყოველი u და v წყვილისათვის მოვძებნოთ უმოკლესი გზა u-დან v-ში. ამ ამოცანის ამოხსნა შეიძლება, თუკი რიგრიგობით ვიპოვით უმოკლეს გზა წვეროთა ყველა წყვილისათვის. თუმცა ეს არაა ოპტიმალური მეთოდი და უფრო ეფექტურ მიდგომას მომდევნო თავებში განვიხილავთ.

ზოგიერთ შემთხვევაში წიბოთა წონები შესაძლოა უარყოფითი იყოს. ამ დროს დიდი მნიშვნელობა აქვს არსებობს თუ არა უარყოფით წონათა ციკლი, რადგან თუკი s წვეროდან შესაძლებელია უარყოფით წონათა ციკლთან მისვლა, შემდეგ შეგვიძლია რამდენჯერაც გვინდა შემოვიაროთ ეს ციკლი და წონა განუწყვეტლივ შემცირდება. ამრიგად, ასეთ შემთხვევაში უმოკლესი გზა არ არსებობს და თვლიან, რომ იგი -∞-ის ტოლია.

თუკი უარყოფითწონიანი ციკლი არ არსებობს, მაშინ ნებისმიერი ციკლი შეგვიძლია უკუვაგდოთ, რათა თავიადან ავიცილოთ გზის გაზრდა. ციკლების გარეშე გზების რაოდენობა სასრულია, ამიტომ უმოკლესი გზის წონა კორექტულადაა განსაზღვრული. მრავალ ამოცანაში უარყოფითწონიანი ციკლი არ არსებობს, იმის გამო რომ ყველა წიბო არაუარყოფითია. ზოგიერთი ალგორითმი (მაგ. დეიქსტრას ალგორითმი) მხოლოდ ასეთი შემთხვევისათვის მუშაობს, ხოლო ზოგი (მაგ. ბელმან-ფორდის ალგორითმი) მუშაობს უარყოფითი წონებისთვისაც, თუკი უარყოფითწონიანი ციკლი არ გვხდება.

ზოგჯერ საჭირო ხდება არა მარტო უმოკლესი გზის წონის გამოთვლა, არამედ თავად ამ გზის დადგენა. ასეთ შემთხვევაში იყენებენ იმავე მეთოდს, რომლითაც პოულობდნენ გზას განივად ძებნის ხეებში.

რელაქსაცია. უმოკლესი გზის ყოველი ნაწილი თვითონ არის უმოკლესი გზა. ეს ნიშნავს, რომ უმოკლესი გზების ამოცანას გააჩნია ოპტიმალურობის თვისება ქვეამოცანებისათვის, რაც იმის მაუწყებელია, რომ ამოცანის ამოსახსნელად შესაძლოა გამოყენებულ იქნას დინამიური პროგრამირება ან ხარბი ალგორითმები. მართლაც დეიქსტრას ალგორითმი ხარბ ალგორითმს წარმოადგენს, ხოლო ფლოიდ-ვორშელის ალგორითმის დინამიური პროგრამირების მეთოდს იყენებს. შემდეგი დებულებები აკონკრეტებენ ოპტიმალურობას თვისებას ქვეამოცანებისათვის უმოკლესი გზების შესახებ ამოცანაში.

ლემა 7.16 (უმოკლესი გზის მონაკვეთები უმოკლესია). *ვთქვათ, მოცემული გვაქვს ორიენტირებული $G=(V,E)$ გრაფი წონადი $w:E\to R$ ფუნქციით. თუ $p=\langle v_1,v_2,\ldots,v_k\rangle$ უმოკლესი გზაა v_1-დან v_k-მდე და $1\le i\le j\le k$, მაშინ $p_{ij}=\langle v_i,v_{i+1},\ldots,v_j\rangle$ წარმოადგენს უმოკლეს გზას v_i-დან v_j-მდე.*

შედეგი 7.17. *ვთქვათ, მოცემული გვაქვს ორიენტირებული $G=(V,E)$ გრაფი წონადი $w:E\to R$ ფუნქციით. განვიხილოთ უმოკლესი p გზა s-დან v-ში. ვთქვათ, $u\to v$ ამ გზის უკანასკნელი წიბოა, მაშინ $\delta(s,v)=\delta(s,u)+w(u,v)$.*

ლემა 7.18. *ვთქვათ, მოცემული გვაქვს ორიენტირებული $G=(V,E)$ გრაფი წონადი $w:E\to R$ ფუნქციით და $s\in V$. მაშინ ნებისმიერი $(u,v)\in E$-სათვის გვაქვს $\delta(s,v)\le\delta(s,u)+w(u,v)$.*

რელაქსაციის მექანიზმი ასეთია: თითოეული $v\in V$-სათვის ვინახავთ რაღაც d[v] რიცხვს, რომელიც წარმოადგენს s-დან v-ში უმოკლესი გზის წონის ზედა შეფასებას ანუ უბრალოდ უმოკლესი გზის შეფასებას (shortest-path estimate). d და π მასივებს საწყისი მნიშვნელობები ენიჭებათ შემდეგი პროცედურით:

```
INITIALIZE-SINGLE-SOURCE(G,s)
1 for ∀ v∈V[G] {
2           d[v]= ∞
3           π[v]=Nil    }
4 d[s]=0
```

$(u,v)\in E$ წიბოს რელაქსაცია შემდეგში მდგომარეობს: $d[v]$ მცირდება $d[u]+w(u,v)$-მდე (თუკი ეს უკანასკნელი ნაკლებია). ამასთან, $d[v]$ რჩება ზედა შეფასებად 7.18. ლემის თანახმად. ამავდროულად იცვლება $\pi[v]$-ც, რომელიც მიუთითებს ზედა შეფასების მიღებისას გამოყენებულ გზას.

```
RELAX(u,v,w)
1 if d[v]>d[u]+w(u,v)
2         then {  d[v]=d[u]+w(u,v)
3                 π[v]-=u   }
```

ამ თავში აღწერილი ალგორითმები ინიციალიზაციისა და რელაქსაციის პროცედურებს ასეთივე მიმდევრობით გამოიყენებენ, თუმცა განსხვავდებიან რელაქსაციის პროცედურის გამოყენების ხარისხით. მაგ. დეიქსტრას ალგორითმი აციკლური გრაფებისათვის მხოლოდ ერთხელ ახდენს წიბოთა რელაქსაციას, ხოლო ბელმან-ფორდი ალგორითმი – რამდენჯერმე.

რელაქსაციას გააჩნია თვისებები, რომლებიც მოცემულია შემდეგ დებულებებში:

ლემა 7.19. *ვთქვათ, მოცემული გვაქვს ორიენტირებული $G=(V,E)$ გრაფი წონადი $w:E\to R$ ფუნქციით და ვთქვათ $(u,v)\in E$. მაშინ ამ წიბოს რელაქსაციის შემდეგ სრულდება უტოლობა $d[v]\le d[u]+w(u,v)$.*

ლემა 7.20. *ვთქვათ, მოცემული გვაქვს წონადი ორიენტირებული $G=(V,E)$ გრაფი წონადი w ფუნქციით. ვთქვათ, $s\in V$ — საწყისი წვეროა. მაშინ INITIALIZE-SINGLE-SOURCE(G,s) პროცედურის შესრულებისა და წიბოთა ნებისმიერი თანმიმდევრობით რელაქსაციის შემდეგ, თითოეული $v\in V$ წვეროსათვის სრულდება უტოლობა $d[v]\ge\delta(s,v)$. თუკი რომელიმე წვეროსათვის ეს უტოლობა გადაიქცევა ტოლობად, მაშინ $d[v]=\delta(s,v)$ ტოლობა ინარჩუნებს ჭეშმარიტობას შემდგომშიც.*

შედეგი 7.21. *ვთქვათ, მოცემული გვაქვს წონადი ორიენტირებული $G=(V,E)$ გრაფი წონადი w ფუნქციით. და s საწყისი წვეროთი. ვთქვათ, $v\in V$ წვერო მიუღწევადია s-დან. მაშინ INITIALIZE-SINGLE-SOURCE(G,s) პროცედურის შესრულებისა და წიბოთა ნებისმიერი თანმიმდევრობით რელაქსაციის შემდეგ, $d[v]$-ს მნიშვნელობად დარჩება უსასრულობა (და $\delta(s,v)$-ს ტოლი).*

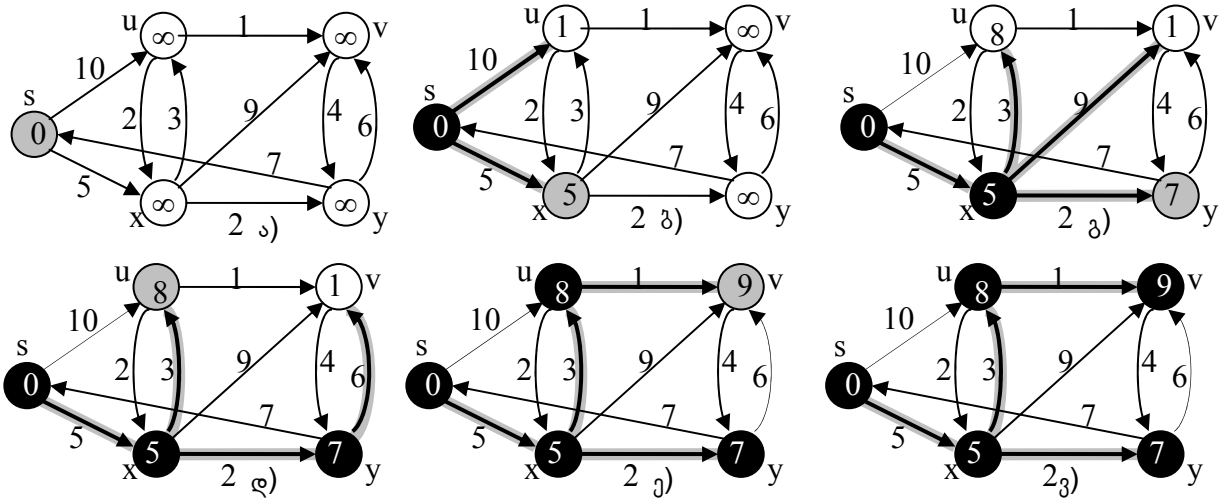
ლემა 7.22. *ვთქვათ, მოცემული გვაქვს წონადი ორიენტირებული $G=(V,E)$ გრაფი წონადი w ფუნქციით. და s საწყისი წვეროთი. ვთქვათ, არსებობს უმოკლესი გზა s-დან v-ში უკანასკნელი (u,v) წიბოთი.. ვთქვათ, შესრულდა INITIALIZE-SINGLE-SOURCE(G,s) პროცედურა, ხოლო შემდეგ რომელიღაც წიბოთა რელაქსაცია, მათ შორის (u,v) წიბოსიც. თუკი რაღაც მომენტში (u,v) წიბოს რელაქსაციამდე შესრულდა $d[u]=\delta(s,u)$ ტოლობა, მაშინ (u,v) წიბოს რელაქსაციის შემდეგ ნებისმიერ მომენტში შესრულდება ტოლობა $d[v]=\delta(s,v)$.*

<u>დეიქსტრას ალგორითმი.</u> დეიქსტრას ალგორითმი წყვეტს ამოცანას უმოკლესი გზების შესახებ ერთი წვეროდან წონადი ორიენტირებული $G=(V,E)$ გრაფისათვის საწყისი s წვეროთი. აუცილებელია, რომ ყველა წიბოს წონა იყოს არაუარყოფითი ($w(u,v)\ge 0$ ყოველი $(u,v)\in E$).

დეიქსტრას ალგორითმის მუშაობის დროს გამოიყენება $S\subseteq V$ სიმრავლე, რომელიც შედგება იმ v წვეროებისაგან, რომელთათვისაც $\delta(s,v)$ უკვე მოძებნილია (ე.ი. $d[v]=\delta(s,v)$). ალგორითმი ირჩევს უმცირესი $d[u]$-ს მქონე $u\in V\setminus S$ წვეროს, ამატებს u-ს S სიმრავლეში და ახდენს u-დან გამომავალი ყველა წიბოს რელაქსაციას, რის შემდეგ ციკლი მეორდება. წვეროები, რომლებიც S-ს არ მიეკუთვნებიან, ინახება Q რიგში პრიორიტეტებით, რომელიც განისაზღვრება d ფუნქციის მნიშვნელობებით. იგულისხმება, რომ გრაფი მოცემულია მოსაზღვრე წვეროთა სიების საშუალებით.

```
DIJKSTRA(G,w,s)
1 INITIALIZE-SINGLE-SOURCE(G,s)
2 S=∅
3 Q=V[G]
4 while Q≠∅ {
5         u=EXTRACT-MIN(Q)
```

```
6           S=S∪ ⟨u⟩
7           for ∀ v∈Adj[u]  {
8                   RELAX(u,v,w)    }       }
```



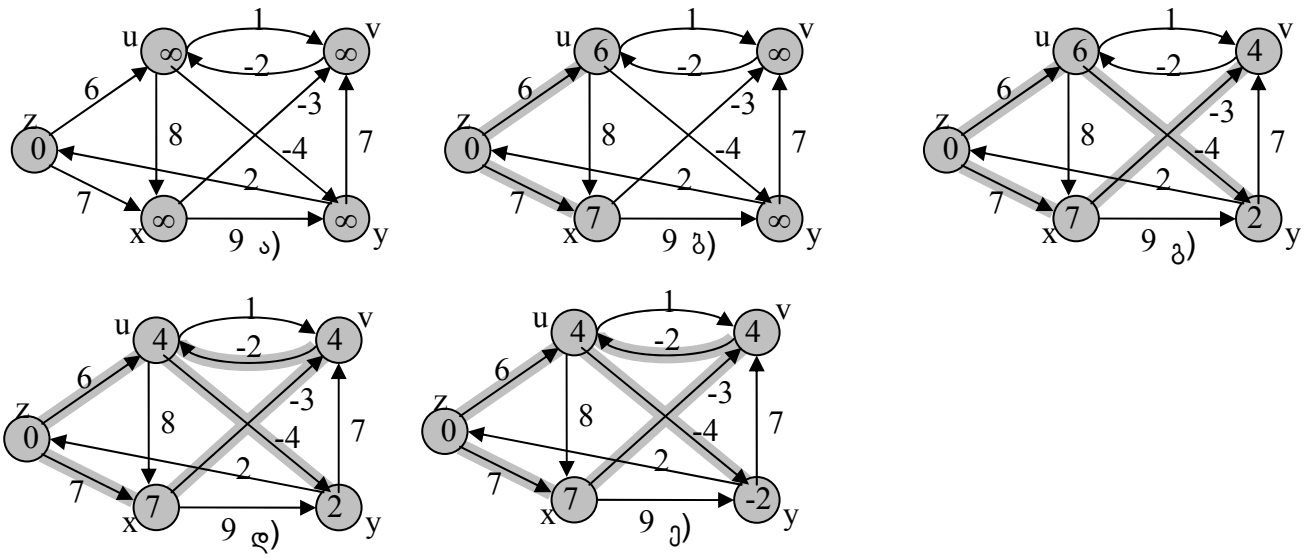
ნახ. 7.14.

დეიქსტრას ალგორითმის მუშაობის პროცესი აღწერილია ნახ. 7.14-ზე. საწყისი წერტილია s. წვეროებში ჩაწერილია უმოკლესი გზების შეფასებები მოცემული მომენტისათვის. შავი ფერით აღნიშნულია წვეროები, რომლებიც S სიმრავლეს ეკუთვნიან. სხვა წვეროები დგანან Q=V\S რიგში. რუხი ფერის წვეროები ციკლის მომდევნო იტერაციის დროს გამოდიან u წვეროს როლში. d და π თავიანთ საბოლოო მნიშვნელობებს ღებულობენ (ვ) ნახაზზე.

ალგორითმის მუშაობის 1 სტრიქონში ხდება d-ს და π-ს, მე-2 სტრიქონში – S-ის, ხოლო მე-3 სტრიქონში – Q-ს ინიციალიზაცია. დასაწყისში Q=V. 4-8 სტრიქონებში Q-დან ხდება უმცირესი d[u]-ს მქონე u წვეროს ამოღება და ის ემატება S სიმრავლეს (თავდაპირველად u=s). 7-8 სტრიქონებში ხდება u-დან გამოსული ყოველი (u,v) წიბოს რელაქსაცია. ამ დროს შეიძლება შეიცვალოს d[v] შეფასება და π[v] წინამორბედი. შევნიშნოთ, რომ ციკლის მუშაობის დროს Q რიგში ახალი წვეროები არ ემატება, ხოლო Q-დან ამოღებული ყოველი წვერო ემატება S სიმრავლეს მხოლოდ ერთხელ, ამიტომ while ციკლის იტერაციათა რაოდენობაა $|V|$.

დეიქსტრას ალგორითმი განეკუთნება ხარბ ალგორითმებს და მისი მუშაობის დროა O(E*log*V).

ბელმან-ფორდის ალგორითმი (Bellman-Ford algorithm) ხსნის უმოკლესი გზების ამოცანას ერთი წვეროდან იმ შემთხვევაში, როცა წიბოებს შესაძლოა ჰქონდეთ უარყოფითი წონები. ეს ალგორითმი იძლევა TRUE მნიშვნელობას, თუ გრაფში საწყისი წვეროდან არაა მიღწევადი უარყოფითწონიანი ციკლი და იძლევა მნიშვნელობას FALSE, თუკი ასეთი ციკლი საწყისი წვეროდან მიღწევადია. პირველ შემთხვევაში ალგორითმი პოულობს უმოკლეს გზებს და მათ წონებს, ხოლო მეორე შემთხვევაში – უმოკლესი გზა არ არსებობს. დეიქსტრას ალგორითმის მსგავსად, ბელმან-ფორდის ალგორითმი ახდენს წიბოების რელაქსაციას მანამ, სანამ d[v]-ს ყველა მნიშვნელობა δ(s,v)-ს არ გაუტოლდება.



ნახ. 7.15

```
BELLMAN-FORD(G,w,s)
1 INITIALIZE-SINGLE-SOURCE(G,s)
2 for i=1 to |V[G]|-1 {
3           for ∀ (u,v)∈E[G]  {
4                 RELAX(u,v,w)  }  }
5 for ∀ (u,v)∈E[G]  {
6           if d[v]>d[u]+w(u,v)
7                 then  {  return FALSE  }     }
8 return TRUE
```

ნახ. 7.15-ზე მოცემულია ბელმან-ფორდის ალგორითმის მუშაობის პროცესი. საწყისი წვეროა z. წვეროებში ნაჩვენებია უმოკლესი გზების შეფასებები. 1 სტრიქონში ხდება ინიციალიზაცია. შემდეგ ალგორითმი (|V|-1)-ჯერ იმეორებს ერთ და იმავე მოქმედებას: ახდენს გრაფის თითოეული წიბოს რელაქსაციას (2-4 სტრიქონები), ხოლო შემდეგ ამოწმებს, არსებობს თუ არა საწყისი წვეროდან მიღწევადი უარყოფითწონიანი ციკლი (5-8 სტრიქონები).

ბელმან-ფორდის ალგორითმის მუშაობის დროა O(V×E).

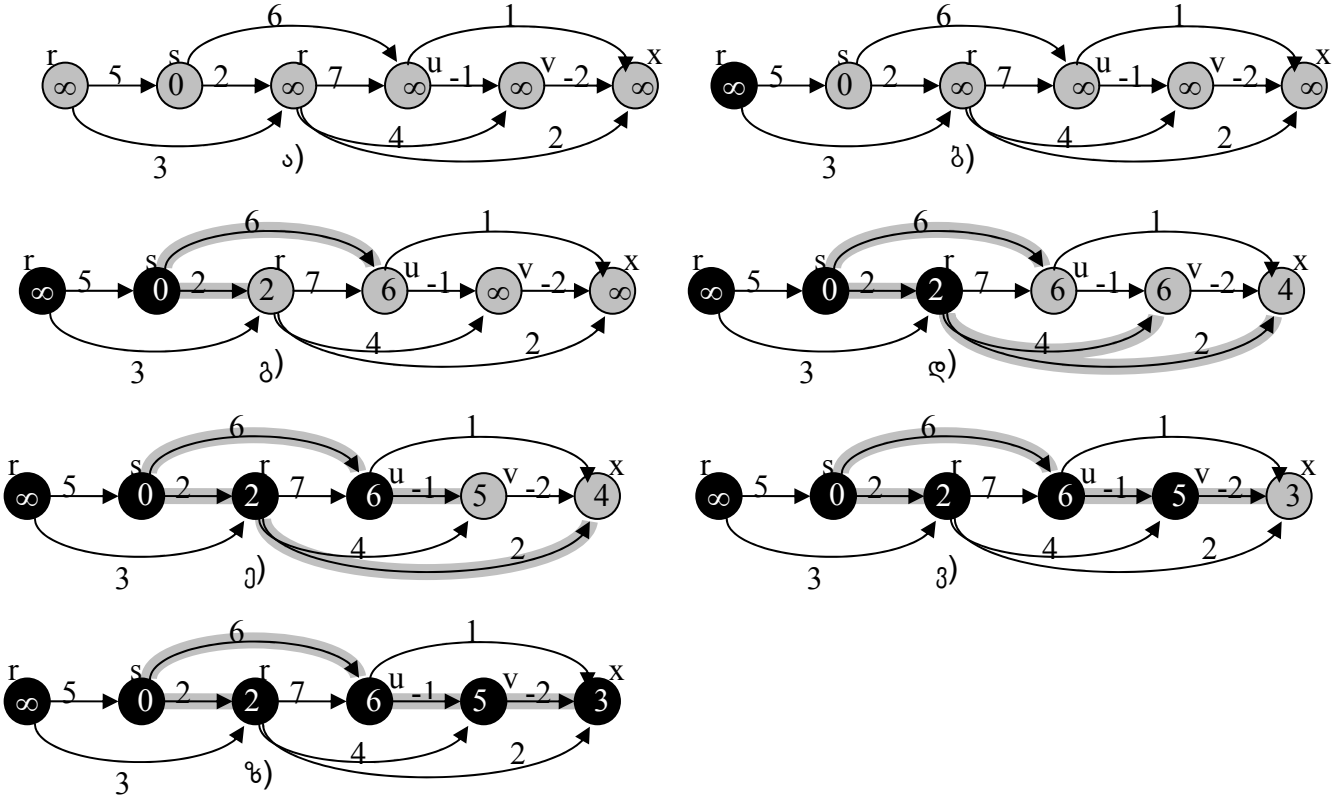
<u>უმოკლესი გზები აციკლურ ორიენტირებულ გრაფში.</u> აციკლურ ორიენტირებულ G=(V,E) გრაფში ერთი წვეროდან უმოკლესი გზების მოსაძებნად საჭიროა O(V+E) დრო, თუკი წიბოების რელაქსაციას ჩავატარებთ ტოპოლოგიურად სორტირებული წვეროების მიხედვით. შევნიშნოთ, რომ აციკლურ ორიენტირებულ გრაფში უმოკლესი გზები ყოველთვის განსაზღვრულია, რადაგან ციკლები (მათ შორის, უარყოფითწონიანიც) საერთოდ არ გვხვდება.

ტოპოლოგიური სორტირება (იხ. თავი 7.6.) იმგვარად განალაგებს წვეროებს წრფივი მიმდევრობით, რომ ყველა წიბო ერთნაირი მიმართულებისაა. ამის შემდეგ უნდა განვიხილოთ წვეროები ამ მიმდევრობით (წიბოს დასაწყისი ყოველთვის მის ბოლოზე ადრე იქნება განხილული) და ყოველი წვეროსათვის მოვახდინოთ მისგან გამომავალის ყველა წიბოს რელაქსაცია.

```
DAG-SHORTEST-PATHS(G,w,s)
1 ტოპოლოგიურად დავალაგოთ G-ს წვეროები
2 INITIALIZE-SINGLE-SOURCE(G,s)
3 for ყველა u წვეროსათვის მოცემული მიმდევრობით  {
4           for ყველა v∈Adj[u] წვეროსათვის  {
5                 RELAX(u,v,w)  }  }
```

ალგორითმის მუშაობის პროცესი ნაჩვენებია ნახ. 7.16-ზე. საწყისი წვეროა s.

ტოპოლოგიურ სორტირებაზე (1 სტრიქონი) იხარჯება θ(V+E) დრო, ხოლო ინიციალიზაციაზე (მე-2 სტრიქონი) — O(V). ციკლში (3-5 სტრიქონი) ყოველი წიბო ერთხელ დამუშავდება, ისე როგორც დეიქსტრას ალგორითმში, მაგრამ ამ უკანასკნელისაგან განსხვავებით, არ იქნება გამოყენებული პრიორიტეტებიანი რიგი. ასეთი დამუშავების ღირებულება O(1)-ია. ე.ი. ალგორითმი შესრულდება θ(V+E) დროში.



ნახ. 7.16.

აღწერილი ალგორითმი შესაძლებელია გამოვიყენოთ ე.წ. "კრიტიკული გზების" საპოვნელად. განვიხილოთ ამოცანა, სადაც ორიენტირებული, აციკლური გრაფის ყოველი წიბო წარმოადგენს რაღაც საქმიანობას, ხოლო წიბოს წონა — მის შესასრულებლად საჭირო დროს. თუკი გვაქვს წიბოები (u,v) და (v,x), მაშინ (u,v) წიბოს შესაბამისი სამუშაო უნდა შესრულდეს (v,x) წიბოს შესაბამისი სამუშაოს დაწყებამდე. **კრიტიკული გზა** (critical path) — ესაა უგრძელესი გზა გრაფში, რომლის წონა ტოლია ყველა სამუშაოს შესასრულებლად დახარჯული დროსი, თუკი მაქსიმალურადაა გამოყენებული ზოგიერთი სამუშაოს პარალელურად შესრულების შესაძლებლობა. კრიტიკული გზის საპოვნელად ყველა წონის ნიშანი უნდა შეიცვალოს საპირისპიროთი და შესრულდეს DAG-SHORTEST-PATHS ალგორითმი.

იენის ალგორითმი (ბელმან-ფორდის ალგორითმის მოდიფიკაცია). G გრაფის წვეროები გადავნომროთ ნებისმიერად და გრაფის წიბოთა E სიმრავლე გავყოთ ორ ნაწილად: E_f — წიბოები, რომლებიც მიმართულია ნაკლები ნომრის მქონე წვეროდან მეტი ნომრის მქონე წვეროსაკენ და E_b — წიბოები, რომლებიც მიმართულია მეტი ნომრის მქონე წვეროდან ნაკლები ნომრის მქონე წვეროსაკენ. ვთქვათ, $G_f=(V,E_f)$ და $G_b=(V,E_b)$, სადაც V გრაფის წვეროთა სიმრავლეა. ცხადია, რომ G_f და G_b გრაფები აციკლურია და ორივე მათგანი დალაგებულია ტოპოლოგიურად.

ბელმან-ფორდის ალგორითმის ციკლის ყოველ იტერაციაზე მოვახდინოთ წიბოთა რელაქსაცია შემდეგნაირად: ჯერ გადავარჩიოთ წვეროები ნომრების ზრდადობის მიხედვით და ყოველი წვეროსათვის მოხდეს მისგან გამომავალი E_f გრაფის ყველა წიბოს რელაქსაცია, შემდეგ წვეროები ნომრების კლებადობის მიხედვით და ყოველი წვეროსათვის მოხდეს მისგან გამომავალი E_b გრაფის ყველა წიბოს რელაქსაცია. ციკლის $[|V|/2]$ იტერაციის შემდეგ გრაფის ყველა წვეროსათვის შესრულებული იქნება $d[v]=\delta(s,v)$.

7.8. უმოკლესი გზები წვეროთა ყველა წყვილისათვის

განვიხილოთ G=(V,E) ბმული არაორიენტირებული გრაფი წონითი w ფუნქციით და ვთქვათ, ჩვენი ამოცანაა წვეროთა ყოველი u,v∈V წყვილისათვის ვიპოვოთ უმოკლესი გზა u-დან v-ში. ამ ამოცანის პასუხად შეიძლება ჩაითვალოს ცხრილი, რომელშიც u სტრიქონისა და v სვეტის გადაკვეთაზე მოცემულია უმოკლესი გზის წონა u-დან v-ში. რა თქმა უნდა, ამოცანა შეიძლება გადაწყდეს, თუკი |V|-ჯერ გამოვიყენებთ ერთი წვეროდან უმოკლესი გზების პოვნის ალგორითმს (სათითაოდ ყველა წვეროდან), მაგრამ დეიქსტრას ალგორითმის უბრალო რეალიზაციისას (პრიორიტეტებიანი რიგით), ალგორითმის მუშაობის დრო იქნება $O(V^3)$, ორობითი გროვების გამოყენებით — $O(VE\log V)$, ხოლო ფიბონაჩის გროვების შემთხვევაში — $O(V^2\log V+VE)$. თუკი გრაფი შეიცავს უარყოფითწონიან წიბოებს, მაშინ დეიქსტრას ალგორითმის გამოყენება დაუშვებელია, ხოლო უფრო ნელი ბელმან-ფორდის ალგორითმი დახარჯავს $O(V^3E)$ დროს (მკვრივი გრაფებისათვის — $O(V^4)$).

ფლოიდ-ვორშელის ალგორითმი. ეს ალგორითმი იყენებს დინამიური პროგრამირების მეთოდს. იგი მუშაობს უარყოფითწონებიანი წიბოების შემცველი გრაფებისთვისაც, თუკი არ გვხვდება უარყოფითწონიანი ციკლი.

ზემოთ განხილული ალგორითმები გამოარჩევდნენ გზის უკანასკნელ წიბოს, ფლოიდ-ვორშელის ალგორითმისათვის მნიშვნელოვანია, რომელი წვეროები წარმოადგენენ შუალედურ წვეროებს, ხოლო გზის მახასიათებლად გამოიყენება საშუალედო წვეროს მაქსიმალური ნომერი წვეროების ფიქსირებული ნუმერაციისას. $p=\langle v_1,v_2,\dots v_t\rangle$ მარტივი გზის საშუალედო (intermediate) წვერო ეწოდება ნებისმიერს $v_2,v_3,\dots v_{t-1}$ წვეროებიდან.

ჩავთვალოთ, რომ G გრაფის წვეროებს წარმოადგენენ რიცხვები 1,2,...,n. განვიხილოთ ნებისმიერი k≤n. მოცემული i,j∈V წვეროთა წყვილისათვის განვიხილოთ ყველა გზა i-დან j-ში, რომელთა წვეროები ეკუთვნიან {1,2,...,k} სიმრავლეს. ვთქვათ p — მინიმალური წონის გზაა ყველა ასეთ გზას შორის. ის იქნება მარტივი, რადგან გრაფში არაა უარყოფითწონიანი ციკლები. როგორ მოვძებნოთ ამ გზის წონა, თუკი ვიცით ყველა ასეთი გზის წონები ნაკლები k წვეროებისთვის?

p გზისათვის არსებობს ორნაირი შემთხვევა. თუ k წვერო არ წარმოადგენს საშუალედოს p გზისათვის, მაშინ p გზის ყველა საშუალედო წვერო მოთავსებულია {1,2,...,k-1} სიმრავლეში. ამ დროს, ცხადია p წარმოადგენს უმოკლეს გზას i-დან j-ში და მისი საშუალედო წვეროები ეკუთვნიან {1,2,...,k-1} სიმრავლეს. თუ k წვერო საშუალედოა p გზისათვის, მაშინ იგი ჰყოფს p-ს ორ p_1 და p_2 ნაწილებად. 7.1. ლემის თანახმად, p_1 არის უმოკლესი გზა i-დან k-მდე, ხოლო p_2 არის უმოკლესი გზა k-დან j-მდე.

ეს მსჯელობა საშუალებას გვაძლევს დავწეროთ რეკურენტული ფორმულა უმოკლესი გზების სიგრძეებისათვის. აღვნიშნოთ $d_{ij}^{(k)}$-თი უმოკლესი გზის წონა i წვეროდან j წვეროში საშუალედო წვეროებით {1,2,...,k} სიმრავლიდან. თუ k=0, მაშინ საშუალედო წვეროები საერთოდ არა გვაქვს და $d_{ij}^{(0)}=w_{ij}$. საზოგადოდ:

$$d_{ij}^{(k)}=\begin{cases} w_{ij}, & \text{თუ } k=0 \\ \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\right), & \text{თუ } k\geq 1 \end{cases}$$

მატრიცა $D^{(n)}=(d_{ij}^{(n)})$ შეიცავს საძებნ მნიშვნელობას, ანუ $d_{ij}^{(n)}=\delta(i,j)$.

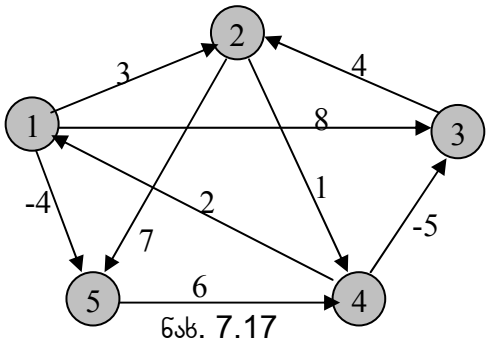
დავწეროთ პროცედურა, რომელიც გამოთვლის უმოკლესი გზების წონებს $d_{ij}^{(k)}$ (k=1,2,...n) მნიშვნელობების თანმიმდევრული პოვნით. მისთვის შემავალი მონაცემი იქნება n×n ზომის W მატრიცა, რომელშიც მოცემულია გრაფის წიბოთა წონები, გამომავალი მონაცემი კი იქნება უმოკლესი გზების წონათა $D^{(n)}$ მატრიცა.

```
FLOYD-WARSHALL(W)
1 n=rows[W]
2 D(n)=W
3 for k=1 to n  {
4         for i=1 to n   {
```

5 for j=1 to n {

6 $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$ } } }

7 return $D^{(n)}$



ნახ. 7.17

უმოკლესი გზების წონათა გარდა ხშირად საჭიროა თავად ეს გზაც. მისი გამოთვლა შეიძლება D მატრიცის გამოთვლის შემდეგაც $O(n^3)$ დროში, მაგრამ უფრო მოსახერხებელია გზები გამოვთვალოთ ფლოიდ-ვორშელის ალგორითმის პარალელურად. ამასთან, უნდა გამოვთვალოთ მატრიცები: $\Pi^{(0)}$, $\Pi^{(1)}$,..., $\Pi^{(n)}$, სადაც $\Pi=\Pi^{(n)}$, ხოლო $\pi_{ij}^{(k)}$ განისაზღვრება როგორც წვერო, რომელიც წინ უძღვის j წვეროს უმოკლეს გზაზე i-დან j-ში საშუალედო წვეროებით {1,2,....k} სიმრავლიდან:

$$\pi_{ij}^{(0)} = \begin{cases} Nil, & \text{თუ } i = j \text{ ან } w_{ij} = \infty \\ i, & \text{თუ } i \neq j \text{ და } w_{ij} < \infty \end{cases}$$

ვთქვათ $K \geq 1$. თუკი უმოკლესი გზა i-დან j-ში გადის k წვეროზე, მისი ბოლოსწინა წვერო იქნება იგივე წვერო, რომელიც იქნება ბოლოსწინა უმოკლეს გზაზე k-დან j-ში, ხოლო თუ გზა არ გადის k-ზე, მაშინ ის ემთხვევა უმოკლეს გზას i-დან j-ში. მაშასადამე:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)}, & \text{თუ } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)}, & \text{თუ } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

ამ ფორმულების დამატება FLOYD-WARSHALL პროცედურისათვის რთული არაა. ნახ. 7.18-ზე გამოსახულია D და Π მასივების შევსება ფლოიდ-ვორშელის ალგორითმის მიხედვით 7.17-ზე გამოსახული გრაფისათვის.

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} Nil & 1 & 1 & Nil & 1 \\ Nil & Nil & Nil & 2 & 2 \\ Nil & 3 & Nil & Nil & Nil \\ 4 & Nil & 4 & Nil & Nil \\ Nil & Nil & Nil & 5 & Nil \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} Nil & 1 & 1 & Nil & 1 \\ Nil & Nil & Nil & 2 & 2 \\ Nil & 3 & Nil & Nil & Nil \\ 4 & 1 & 4 & Nil & 1 \\ Nil & Nil & Nil & 5 & Nil \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} Nil & 1 & 1 & 2 & 1 \\ Nil & Nil & Nil & 2 & 2 \\ Nil & 3 & Nil & 2 & 2 \\ 4 & 1 & 4 & Nil & 1 \\ Nil & Nil & Nil & 5 & Nil \end{pmatrix}$$

$$D^{(3)}=\begin{pmatrix}0&3&8&4&-4\\ \infty&0&\infty&1&7\\ \infty&4&0&5&11\\ 2&-1&-5&0&-2\\ \infty&\infty&\infty&6&0\end{pmatrix}\qquad \Pi^{(3)}=\begin{pmatrix}Nil&1&1&2&1\\ Nil&Nil&Nil&2&2\\ Nil&3&Nil&2&2\\ 4&3&4&Nil&1\\ Nil&Nil&Nil&5&Nil\end{pmatrix}$$

$$D^{(4)}=\begin{pmatrix}0&3&-1&4&-4\\ 3&0&-4&1&-1\\ 7&4&0&5&3\\ 2&-1&-5&0&-2\\ 8&5&1&6&0\end{pmatrix}\qquad \Pi^{(4)}=\begin{pmatrix}Nil&1&4&2&1\\ 4&Nil&4&2&1\\ 4&3&Nil&2&1\\ 4&3&4&Nil&1\\ 4&3&4&5&Nil\end{pmatrix}$$

$$D^{(5)}=\begin{pmatrix}0&1&-3&2&-4\\ 3&0&-4&1&-1\\ 7&4&0&5&3\\ 2&-1&-5&0&-2\\ 8&5&1&6&0\end{pmatrix}\qquad \Pi^{(5)}=\begin{pmatrix}Nil&3&4&5&1\\ 4&Nil&4&2&1\\ 4&3&Nil&2&1\\ 4&3&4&Nil&1\\ 4&3&4&5&Nil\end{pmatrix}$$

ნახ. 7.18.

<u>ორიენტირებული გრაფის ტრანზიტიული ჩაკეტვის ამოცანა</u> მდგომარეობს შემდეგში: მოცემულია G=(V,E) გრაფი 1,2,...,n წვეროებით. საჭიროა ნებისმიერი i,j∈V წვეროებისათვის გაირკვეს, არსებობს თუ არა გრაფში გზა i წვეროდან j წვეროში. ორიენტირებული G გრაფის ტრანზიტიული ჩაკეტვა (transitive closure) ეწოდება $G^*=(V,E^*)$ გრაფს, სადაც $E^*=\{(i,j)$: G გრაფში არსებობს გზა i-დან j-ში$\}$.

გრაფის ტრანზიტიული ჩაკეტვა გამოითვლება $\theta(n^3)$ დროში ფლოიდ-ვორშელის ალგორითმის საშუალებით, სადაც გრაფის ყოველი წიბოს წონა უნდა ჩაითვალოს 1-ის ტოლად. თუკი არსებობს გზა i-დან j-ში, მაშინ d_{ij} იქნება n-ზე ნაკლები, ხოლო თუ ასეთი გზა არ არსებობს, მაშინ $d_{ij}=\infty$.

დროისა და მანქანური მეხსიერების ეკონომიის მიზნით, უმჯობესია ფლოიდ-ვორშელის ალგორითმში არითმეტიკული ოპერაციები — min და +, შევცვალოთ ლოგიკური ოპერაციებით AND (და) და OR (ან). უფრო ზუსტად, t_{ij}^k ჩავთვალოთ 1-ის ტოლად, თუ გრაფში არსებობს გზა i-დან j-ში და ჩავთვალოთ 0-ად, თუკი ასეთ გზა არ არსებობს. (i,j) წიბო ეკუთვნის G^* ტრანზიტიულ ჩაკეტვას, მაშინ და მხოლოდ მაშინ, როცა $t_{ij}^k=1$, ე.ი.

$$t_{ij}^{(0)}=\begin{cases}0, & \text{თუ } i\neq j \text{ და } (i,j)\notin E\\ 1, & \text{თუ } i=j \text{ ან } (i,j)\in E\end{cases}$$

ხოლო თუ $k\geq 1$

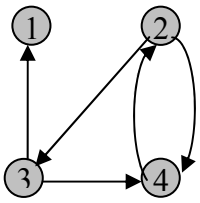
$$t_{ij}^{(k)}=t_{ij}^{(k-1)}\ OR\left(t_{ik}^{(k-1)}\ AND\ t_{kj}^{(k-1)}\right)$$

ამ თანაფარდობებზე დაყრდნობით ალგორითმი თანმიმდევრობით გამოითვლის $T^{(k)}=(t_{ij}^{(k)})$ მატრიცას k=1,2,...,n-სათვის:

```
TRANSITIVE-CLOSURE(G)
1 n=|V[G]|
2 for i=1 to n  {
3         for j=1 to n  {
4                 if i=j ან (i,j)∈E[G]
5                         then { t_ij^(0) = 1   }
```

```
6                    else { t_ij^(0) = 0 }      }      }
7 for k=1 to n  {
8          for i=1 to n  {
9                    for j=1 to n  {
10                        t_ij^(k) = t_ij^(k-1) OR (t_ik^(k-1) AND t_kj^(k-1))  }  }  }
11 return T^(n)
```

ნახ 7.19-ზე მოცემულია გრაფი და ალგორითმის მუშაობის პროცესი ამ გრაფისათვის:



$$T^{(0)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&0\\1&0&1&1\end{pmatrix}\qquad T^{(1)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&0\\1&0&1&1\end{pmatrix}$$

$$T^{(2)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&1\\1&0&1&1\end{pmatrix}\qquad T^{(3)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&1\\1&1&1&1\end{pmatrix}\qquad T^{(4)}=\begin{pmatrix}1&0&0&0\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{pmatrix}$$

ნახ. 7.19.

TRANSITIVE-CLOSURE-ს მუშაობის დრო ფლოიდ-ვორშელის ალგორითმის მსგავსად $\theta(n^3)$-ია, მაგრამ უფრო ეფექტურია რიგ შემთხვევებში, რადგან ლოგიკური ოპერაციები ზოგ კომპიუტერზე უფრო სწრაფად სრულდება, ვიდრე არითმეტიკული ოპერაციები მთელ რიცხვებზე, ხოლო ლოგიკურ ცვლადებს ნაკლები მეხსიერება უჭირავთ.

<u>ჯონსონის ალგორითმი ხალვათი გრაფებისათვის</u>. ჯონსონის ალგორითმი პოულობს უმოკლეს გზებს წვეროთა ყველა წყვილისათვის $O(V^2 \log V+VE)$ დროში და ამიტომ ხალვათი გრაფებისათვის უფრო ეფექტურია, ვიდრე ფლოიდ-ვორშელის ალგორითმი. ჯონსონის ალგორითმი ან იძლევა უმოკლესი გზების წონათა მატრიცას ან იტყობინება, რომ გრაფში არის უარყოფითწონიანი ციკლი. ეს ალგორითმი შეიცავს დეიქსტრასა და ბელმან-ფორდის ალგორითმების ზემოთ მოყვანილი პროცედურების გამოძახებას.

ჯონსონის ალგორითმი დაფუძნებული წონათა ცვლილების (reweighting) იდეაზე. თუ გრაფის ყველა წიბოს წონა არაუარყოფითია, მაშინ დეიქსტრას ალგორითმის გამოყენებით თითოეული წვეროსათვის შეგვიძლია ვიპოვოთ უმოკლესი გზები წვეროთა ყველა წყვილისათვის. თუკი გრაფში არის უარყოფითწონიანი წიბოები, შეგვიძლია ასეთი შემთხვევა დავიყვანოთ არაუარყოფითწონიანი წიბოების შემთხვევაზე w წონითი ფუნქციის შეცვლით ახალი $\hat{w}$ ფუნქციით. ამასთან უნდა შესრულდეს შემდეგი თვისებები: 1) უმოკლესი გზები არ იცვლება: წვეროთა ნებისმიერი $u,v\in V$ წყვილისათვის უმოკლესი გზა u-დან v-ში w წონითი ფუნქციის მიხედვით წარმოადგენს უმოკლეს გზას $\hat{w}$ წონითი ფუნქციის მიხედვითაც და პირიქით; 2) ყველა ახალი $\hat{w}(u,v)$ არაუარყოფითია.

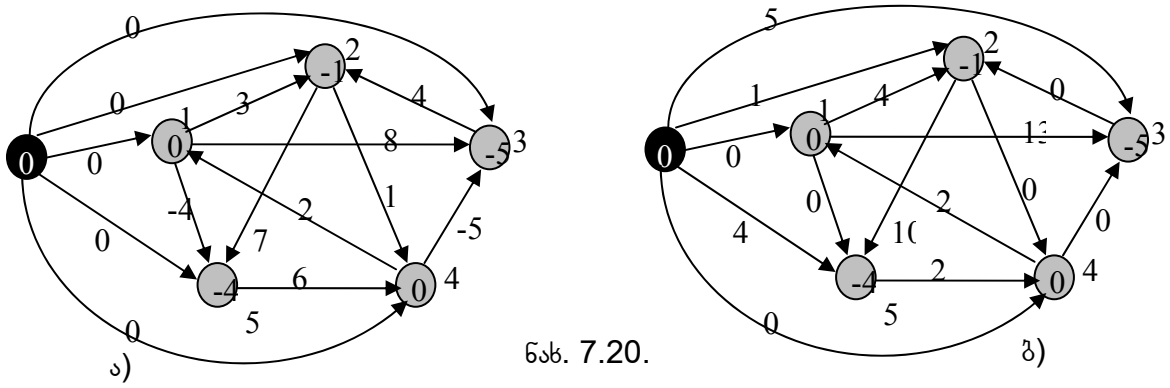
ლემა 7.23 (წონათა ცვლილება არ ცვლის უმოკლეს გზებს). ვთქვათ $G=(V,E)$ ბმული ორიენტირებული გრაფია წონითი $w:E\to R$ ფუნქციით. ვთქვათ $h:V\to R$ გრაფის წვეროებზე განსაზღვრული ფუნქციაა ნამდვილი მნიშვნელობებით. განვიხილოთ ახალი წონითი ფუნქცია $\hat{w}(u,v)=w(u,v)+h(u)-h(v)$, მაშინ:

ა) ნებისმიერი გზა წარმოადგენს უმოკლეს w-ს მიმართ მაშინ და მხოლოდ მაშინ, როცა ის იქნება უმოკლესი $\hat{w}$-ს მიმართ;

ბ) G გრაფი შეიცავს ციკლს უარყოფითი w წონით მაშინ და მხოლოდ მაშინ, როცა ის შეიცავს ციკლს უარყოფითი $\hat{w}$ წონით.

ახლა უნდა შევარჩიოთ h ფუნქცია ისე, რომ შეცვლილი $\hat{w}(u,v)$ წონები არაუარყოფითი იყოს. ამისათვის, ავაგოთ ახალი $G'=(V',E')$ გრაფი $G=(V,E)$ გრაფზე ერთი s წვეროს დამატებით, რომლიდანაც გრაფის ყველა სხვა წვეროსაკენ მიმართული იქნება ნულოვანი წონის წიბოები. ცხადია, რომ ახალ G' გრაფში უარყოფითწონიანი ციკლი იქნება მაშინ და მხოლოდ მაშინ, თუ ასეთი ციკლი G გრაფში არსებობდა.

h-ის განსაზღვრებიდან გამომდინარე ნებისმიერი $v\in V'$-სათვის $h(v)=\delta(s,v)$. 7.18 ლემის თანახმად, ნებისმიერი $(u,v)\in E'$ წიბოსათვის სრულდება უტოლობა $h(v)\le h(u)+w(u,v)$, რომელიც შეიძლება ასე გადავწეროთ: $w(u,v)+h(u)-h(v)\ge 0$. ეს კი ნიშნავს, რომ 7.23. ლემით განსაზღვრული ახალი წონითი ფუნქცია არაუარყოფითია.



ნახ. 7.20.

ნახ. 7.20-ზე გამოსახულია G გრაფის შესაბამისი G′ გრაფი. დამხმარე s წვერო აღნიშნულია შავად. ყოველი წვეროს შიგნით ჩაწერილია მნიშვნელობა h(v)=δ(s,v). (ბ) ნახაზზე ყოველ (u,v) წიბოს მიწერილი აქვს ახალი წონა — w(u,v)+h(u)-h(v).

```
JONSON(G)
1 შევქმნათ G′ გრაფი, რომლისთვისაც V[G′]= V[G]∪{s} და E[G′]= E[G]∪⟨(s,v):v∈V[G]⟩
2 if BELLMAN-FORD(G′,w,s)=FALSE
3        then { დავბეჭდოთ: არსებობს უარყოფითწონიანი ციკლი" }
4        else { for ∀ v∈E[G′] {
5                    h(v)=δ(s,v)   } (δ(s,v)-ის მნიშვნელობა გამოითვლება ბელმან-ფორდით)
6             for ∀ (u,v)∈E[G′] {
7                    ŵ(u,v)=w(u,v)+h(u)-h(v)   }
8             for ∀ u∈V[G′] {
9                    DIJKSTRA (G, ŵ, u)      (გამოვთვალოთ δ̂(u,v) ყოველი v∈ V[G]-სათვის)
10                   for ∀ v∈V[G] {
11                       d_uv= δ̂(u,v) )+h(u)-h(v)   }   }   }
12 return D
```

7.1. გზების მშენებლობა
(USACO, 2003 წელი, შემოდგომა, "ნარინჯისფერი" დივიზიონი)

ფერმერი ჯონი აშენებს P (1≤P≤1000) გზას, რათა დააკავშიროს N (1≤N≤1000) მინდორი. მას სურს გაიგოს, შეიძლება თუ არა ნებისმიერი მინდვრიდან ნებისმიერზე მოხვედრა. დაეხმარეთ მას ამის გარკვევაში და შეამოწმეთ დაკავშირებულია თუ არა ყველა მინდორი.

შემავალი მონაცემების ფორმატი:

1 სტრიქონი: ჰარით გაყოფილი ორი მთელი რიცხვი N და P;

2..P+1 სტრიქონები: ჰარით გაყოფილი ორი მთელი რიცხვი, რომლებიც მიუთითებენ გზის ბოლოებს. გზებზე მოძრაობა ორივე მიმართულებით შეიძლება. გზები ჩამოთვლილია ზრდადობით (1-დან N-საკენ).

შემავალი მონაცემების მაგალითი (ფაილი build.in):

```
4 5
1 2
1 4
2 3
2 4
3 4
```

გამომავალი მონაცემების ფორმატი: ერთადერთ სტრიქონში ნაჩვენები მთელი რიცხვი უნდა წარმოადგენდეს დაკავშირებული გზების უდიდეს რაოდენობას

გამომავალი მონაცემების მაგალითი (ფაილი build.out):

```
5
```

მითითება. ამოცანის ამოსახსნელად საჭიროა ტალღური მეთოდის გაშვება ნებისმიერი წვეროდან (ვთქვათ 1-დან) და იმ გზათა რაოდენობის დათვლა, რომლებზეც ტალღამ გაიარა. თუ რომელიმე გზა გაუვლელი დარჩა, მთვლელს ვანულებთ, შესაბამისი წვეროდან ვუშვებთ ახალ ტალღას, და ამ პროცესს გავაგრძელებთ მანამ, ვიდრე გაუვლელი გზები არ ამოიწურება. მიღებულ შედეგებს შორის უდიდესი წარმოადგენს ამოცანის პასუხს.

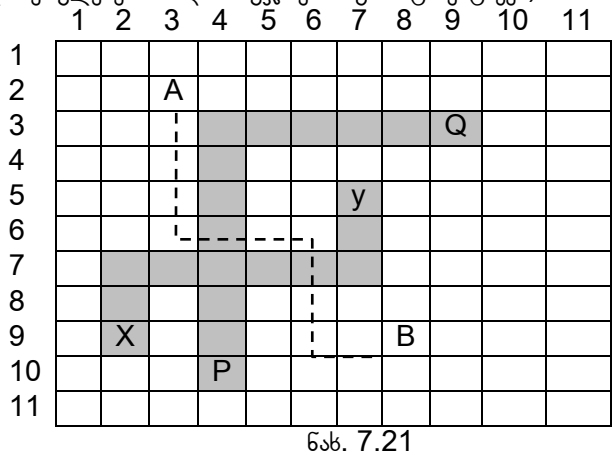
7.2. კონტურის განთავსება

(საქართველოს მოსწავლეთა ოლიმპიადა, ზონური ტური, 2000-01 წლები)

ჩვენი სურვილია განვათავსოთ კონტური n×n ზომის უჯრებიან ფურცელზე. კონტური – ეს არის საწყისი და საბოლოო წერტილების მქონე უჯრათა ისეთი მიმდევრობა, რომლის ყოველ უჯრას, დაწყებული მეორედან, მის წინა უჯრასთან საერთო გააჩნია. ქვემოთ მოცემულ ნახაზზე გვაქვს ორი კონტური, რომლებიც აერთებენ შესაბამისად P და Q, X და Y წერტილებს. თქვენ მოგეთხოვებათ განათავსოთ მოცემულ უჯრებიან ფურცელზე ახალი კონტური იქ უკვე არსებულ კონტურებთან (ბაზურ კონტურებთან) ერთად.

ნებადართულია ახალი კონტურით ბაზური კონტურების გადაკვეთა. ახალი კონტურის განთავსების ღირებულება შემდეგი წესით გამოითვლება: ცარიელ უჯრაზე გავლისას იგი 1-ით იზრდება, ხოლო ბაზური კონტურის უჯრაზე გავლისას კი K-თი (K≥2). ამოცანა მდგომარეობს მოცემული ბაზური კონტურებისათვის ახალი კონტურის ისეთი განთავსების პოვნაში, რომლის ღირებულება მინიმალური იქნება.

მაგალითად, როცა K=4, იმ კონტურის განთავსების ღირებულება, რომელიც A დაB წერტილებს აერთებს და ნახაზზე წყვეტილი წირით არის გამოსახული, 19-ის ტოლია. თუმცა შეიძლება სხვა განთავსების მოძებნაც, რომლის ღირებულება 16 იქნება (ეს ღირებულებები A და B უჯრების ფასსაც შეიცავენ).



ნახ. 7.21

შესატანი მონაცემები: შესატანი მონაცემების CIRCUIT.INP ფაილის პირველ სტრიქონში წერია n-ის მნიშვნელობა (2≤n≤50). მეორე სტრიქონში მოცემულია ოთხი მთელი რიცხვი, რომლებიც განსაზღვრავენ განსათავსებელი კონტურის საწყის და საბოლოო წერტილების კოორდინატებს (ეს წერტილები არ შეიძლება ერთმანეთს ემთხვეოდნენ). მესამე სტრიქონში ჩაწერილია K-ს მნიშვნელობა, ხოლო მეოთხეში – ბაზურ კონტურთა რაოდენობა.

მეხუთე სტრიქონიდან დაწყებული, თითოეულ მომდევნო სტრიქონში მოცემულია ინფორმაცია თითო ბაზური კონტურის შესახებ შემდეგი მიმდევრობით: საწყისი, მობრუნების და საბოლოო წერტილთა საერთო რაოდენობა (მობრუნების წერტილი ეწოდება წერტილს, რომელზედაც კონტური მიმართულებას იცვლის) და შესაბამისი რიგით ჩამოთვლილი მათი კოორდინატები.

ყოველ სტრიქონში მონაცემები ერთმანეთისაგან თითო ჰარით გამოიყოფა.

გამოსატანი მონაცემები: გამოსატან მონაცემების CIRCUIT.OUT ფაილის ერთადერთ სტრიქონში უნდა ჩაიწეროს ახალი კონტურის განთავსების მინიმალური ღირებულება.

CIRCUIT.INP	CIRCUIT.OUT
11	16
2 3 9 8	
4	
2	
3 3 9 3 4 10 4	
4 9 2 7 2 7 7 5 7	

მითითება. ეს ამოცანა იხსნება ტალღური მეთოდის გამოყენებით. უჯრედებიანი ფურცელი (ან ზოგადად, მსგავსი წესით უჯრედებად დაყოფილი არე) შეგვიძლია აღვწეროთ არაორიენტირებული გრაფის სახით, სადაც თითოეული უჯრედის შესაბამის წვეროს უკავშირდება მისი ოთხი მეზობელი უჯრედი. მეხსიერების თვალსაზრისით უფრო ეკონომიური გზაა თითოეული უჯრედისათვის მეზობლების გამოთვლა ტალღის ყოველ ბიჯზე. ტალღის გავრცელებას ვიწყებთ A წერტილიდან და ყველა უჯრედში ვწერთ A-დან ამ წერტილამდე მისვლის უმცირეს ღირებულებას. ტალღის დამთავრების შემდეგ B წერტილის შესაბამის უჯრაში ჩაწერილი რიცხვი იქნება ამოცანის პასუხი.

7.3. ნაკეთობები

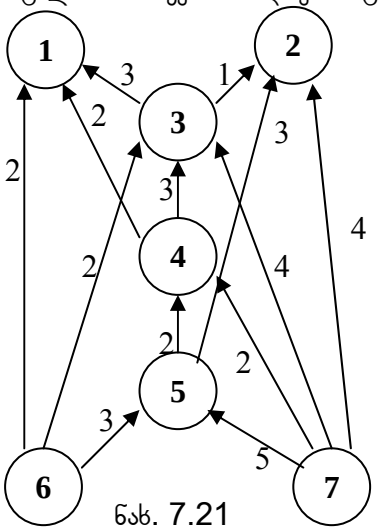
(საქართველოს სტუდენტთა პირადი პირველობა, 2002 წელი)

ნაკეთობის ქვეშ ვიგულისხმოთ მწვერვალი, რომელსაც არ აქვს გამომავალი რკალები. დეტალის ქვეშ ვიგულისხმოთ მწვერვალი, რომელსაც არ აქვს შემავალი რკალები. კვანძის ქვეშ ვიგულისხმოთ მწვერვალი, რომელსაც აქვს როგორც შემავალი, ასევე გამომავალი რკალები.

ნაკეთობის სტრუქტურა შეიძლება აღიწეროს უციკლო ორიენტირებული წონადი გრაფის საშუალებით, რომელიც გვიჩვენებს, თუ რომელი დეტალები/კვანძები და რა რაოდენობით გამოიყენება სხვა ნაკეთობების/კვანძების ასაწყობად. რკალი იწყება კვანძზე/დეტალზე და მთავრდება ნაკეთობაზე/კვანძზე, სადაც გამოიყენება პირველი. რკალის წონა მიუთითებს რაოდენობას. მაგალითად, ნახ. 1-ზე კვანძი №4 გამოიყენება №1 ნაკეთობაში რაოდენობით 2 ცალი, ხოლო №3 კვანძში - რაოდენობით 3 ცალი.

უნდა დაიწეროს პროგრამა, რომელიც ითვლის, თუ რა რაოდენობით გამოიყენება თითოეული დეტალი ან კვანძი ყველანაირი სახის თითო ცალი ნაკეთობის დასამზადებლად.

მაგალითად, ნახ. 1-ზე ნაჩვენები მონაცემებისათვის, უნდა მიეთითოს, თუ რა რაოდენობით გამოიყენება თითოეული კვანძი და თითოეული დეტალი ერთი ცალი №1 ნაკეთობის და ერთი ცალი №2 ნაკეთობის ასაწყობად.



ნახ. 7.21

№1 = 1

№2 = 1

№3 = 3·1+1·1 = 4

№4 = 2+3·3+3·1 = 14

№5 = 3+2·2+2·3·3+2·3·1 = 31

№6 = 2+2·3+2·1+3·3+3·2·2+3·2·3·3+3·2·3·1 = 103

№7 = 4+4·3+4·1+2·3·1+2·3·3+2·2+5·3+5·2·2+5·2·3·3+5·2·3·1 = 203

შესატანი ფაილის ფორმატი:

შესატანი ფაილის თითოეულ სტრიქონში მოცემულია ნატურალურ რიცხვთა სამეული თითო შუალედის გამოტოვებით. პირველი რიცხვი ასახავს ნაკეთობის/კვანძის ნომერს. მეორე რიცხვი ასახავს კვანძის/დეტალის ნომერს, რომელიც გამოყენებულია პირველში. მესამე რიცხვი ასახავს რაოდენობას. რიცხვთა სამეულები არ არის მოწესრიგებული, მაგრამ ყველა ერთად აღწერს ნაკეთობების მოცემულ სტრუქტურას.

ნაგულისხმევია, რომ ასაწყობი ერთეულების სახეობების საერთო რაოდენობა არ აღემატება 100-ს, ხოლო მათი ნომრები მოცემულია თანმიმდევრობით, დაწყებული 1-დან.

გამოსატანი ფაილის ფორმატი:

პროგრამის მუშაობის შედეგს წარმოადგენს ფაილი, რომელიც შეიცავს იმდენ სტრიქონს, რამდენი ასაწყობი ერთეულიც არის გამოაშკარავებული. თითოეულ სტრიქონში (სტრიქონის ნომერი უნდა ემთხვეოდეს ასაწყობი ერთეულის ნომერს) გამოიტანება ასაწყობი ერთეულების რაოდენობა (გარანტირებულია, რომ იგი არ აღემატება $2^{31}-1$-ს), რომელიც აუცილებელია თითოეული სახის ერთი ნაკეთობის ასაწყობად. სტრიქონებში, რომლებიც შეესაბამება ნაკეთობათა ნომრებს, გამოიტანება ერთიანები.

შესატანი ფაილის ნიმუში (ნახ. 7.21):	გამოსატანი ფაილის ნიმუში:
5 7 5 5 6 3 4 7 2 4 5 2 3 7 4 3 6 2 3 4 3 2 7 4 2 5 3 2 3 1 1 6 2 1 4 2 1 3 3	1 1 4 14 31 103 203

მითითება. ამოცანა იხსნება ტოპოლოგიური სორტირების გამოყენებით. ჯერ უნდა ვიპოვოთ დეტალები, ანუ გრაფის ის წვეროები, რომელთაც არა აქვთ გამომავალი რკალები, შემდეგ გადავიდეთ გრაფის იმ წვეროებზე, რომლებიც უკვე ნაპოვნებთან არიან დაკავშირებული. პარალელურად რაოდენობების მასივში უნდა მოვახდინოთ შესაბამისი გადაანგარიშებები. პროცესი შევწყვიტოთ მაშინ, როცა ყველა ნაკეთობა განხილული იქნება.

7.4. რძის საიდუმლო არხები
(USA OPEN, 2002 წელი, "მწვანე" დივიზიონი)

ფერმერ ჯონს სურს რაც შეიძლება მცირე დანახარჯებით ერთმანეთს დააკავშიროს მისი წყლის გამანაწილებელი სისტემა. მაგრამ მას არ უნდა გააცნოს გაყვანილობის მარშრუტი მის კონკურენტს, რომელიც ჩვეულებრივ ეკითხება ჯონს და ამიტომ პრობლემის ასაცილებლად ჯონი სათქმელად იყენებს სიაფით მეორე მარშრუტს. მოცემულია მილები რომლებითაც საჭიროა W (3≤W≤2000) რაოდენობით წყლის სადგურის დაკავშირება. ყოველ მილს გააჩნია ღირებულება, ხოლო მათი რაოდენობაა P (P≤20000). მოძებნეთ სიიაფის მიხედვით მეორე წყლის გამანაწილებელი სისტემა.

ცნობილია რომ ყოველ ორ სადგურს აერთებს მხოლოდ ერთი მილი. გარანტირებულია ყველაზე იაფი მხოლოდ ერთი მარშრუტის არსებობა. აგრეთვე გარანტირებილია წყლის განაწილების მინიმუმ ორი განსხვავებული ვარიანტის არსებობა. მილების ფასები წარმოადგენენ 16 ბიტიან რიცხვებს, ხოლო წყლის სადგურებს შეესაბამებიან მთელი რიცხვები დიაპაზონში 1..W.

შემავალი ფაილის ფორმატი:

სტრიქონი 1: ორი ჰარით დაშორებული მთელი W და P.

სტრიქონები 2..P+1: ყოველი სტრიქონი შეიცავს ჰარით დაშორებულ სამ მთელს შემდეგი თანმიმდევრობით - მილის ერთ ბოლოზე განლაგებულ სადგურის ნომერს, მილის მეორე ბოლოს განლაგებულ სადგურის ნომერს და მილის ღირებულებას.

შემავალი ფაილის მაგალითი (ფაილი secret.in):

```
5 7
1 2 3
2 3 4
1 4 7
2 4 11
2 5 9
5 4 5
3 5 8
```

გამომავალი ფაილის ფორმატი:

ერთი სტრიქონი ერთი მთელი რიცხვით, რომელიც აღნიშნავს სიიაფით მეორე გაყვანილობის მარშრუტის ჯამურ ღირებულებას.

გამომავალი ფაილის მაგალითი (ფაილი secret.out):

```
20
```

მითითება. ჯერ უნდა ვიპოვოთ ყველაზე იაფი მარშრუტი, ანუ ავაგოთ მინიმალური დამფარავი ხე. უმჯობესი იქნება კრასკალის ალგორითმი, სადაც მილები დასორტირებული იქნებიან ღირებულებების მიხედვით. მინიმალური დამფარავი ხის აგების შემდეგ მილების სიმრავლე გაიყოფა ორ ნაწილად: 1) მილები, რომლებიც შედიან მინიმალურ დამფარავ ხეში და 2) მილები, რომლებიც არ შედიან ამ ხეში. ცხადია, რომ სიიაფით მეორე მარშრუტი, ანუ სიდიდით მეორე დამფარავი ხე, იქნება ისეთი ხე, რომელშიც პირველი სიმრავლის რომელიმე წევრი, შეცვლილი იქნება მეორე სიმრავლის წევრით (ან წევრებით). ამიტომ პირველი სიმრავლის თითოეულ წევრს ცალ-ცალკე გავუკეთებთ იგნორირებას (თითქოს ასეთი მილი საერთოდ არ არსებობს) და ხელახლა გავუშვებთ კრასკალის ალგორითმს. მიღებულ პასუხებს შორის მინიმალური იქნება ამოცანის პასუხი. იმის გამო, რომ მონაცემები სორტირებულია, შესაძლებელია გადარჩევის შემცირება ჩაჭრების საშუალებით.

7.5. გზების შერჩევა

(მოსწავლეთა საერთაშორისო ოლიმპიადა, 2003 წელი, აშშ)

ფერმერ ჯონის ძროხებს სურთ თავისუფლად იარონ ფერმაში არსებულ N (1≤N≤200) რაოდენობის 1-დან N-მდე მიმდევრობით გადანომრილ ნაკვეთებზე იმ შემთხვევაშიც კი, თუ ნაკვეთები ერთმანეთისაგან ტყით არის გაყოფილი. ძროხები ნაკვეთთა წყვილებს შორის ირჩევენ გზათა სისტემას ისე, რომ მათ შესძლონ მოხვდნენ ნებისმიერი ნაკვეთიდან სხვა ნებისმიერ ნაკვეთში ამ გზებზე გავლით. ძროხებს გზაზე მოძრაობა შეუძლიათ ორივე მიმართულებით.

გზებად ძროხებს შეუძლით გამოიყენონ მხოლოდ ველური ცხოველების ბილიკები. ყოველ კვირაში მათ შეუძლიათ აირჩიონ ველურ ცხოველთა ზოგიერთი ან ყველა ის ბილიკი, რომელთა არსებობის შესახებ მათთვის ცნობილია. საინტერესოა, რომ ყოველი კვირის დასაწყისში ძროხები პოულობენ ველურ ცხოველთა ზუსტად ერთ ახალ ბილიკს. მათ უნდა გადაწყვიტონ, რომელ ბილიკთა სიმრავლე უნდა აირჩიონ გზათა იმ სისტემის როლში, რომელსაც მიმდინარე კვირაში გამოიყენებენ სასიარულოდ. ძროხებს შეუძლიათ აირჩიონ ველური ცხოველების ბილიკთა ნებისმიერი ქვესიმრავლე იმის მიუხედავად, თუ რომელი ბილიკები იქნა არჩეული წინა კვირაში.

ძროხებს ყოველთვის სურთ იმ გზების ჯამური სიგრძის მინიმიზაცია, რომელთა არჩევასაც ისინი აპირებენ.

ველურ ცხოველთა ბილიკები არ არის სწორხაზოვანი. ორი ბილიკი, რომლებიც ორ ერთი და იგივე ნაკვეთებს აერთებს, შეიძლება განსხვავებული სიგრძის იყოს. გარდა ამისა, ორი ბილიკის გადაკვეთის შემთხვევაში (რაც შეიძლება მხოლოდ ნაკვეთის გარეთ მოხდეს) ძროხებს არ შეუძლიათ მეორე ბილიკზე გადასვლა.

ყოველი კვირის დასაწყისში არსებობს ინფორმაცია ველურ ცხოველთა იმ ბილიკის შესახებ, რომელიც ძროხებმა აღმოაჩინეს. ამის საფუძველზე თქვენმა პროგრამამ უნდა გამოიტანოს ძროხების მიერ ამ კვირაში შერჩეული გზების მინიმალური ჯამური სიგრძე, ან მიუთითოს, რომ გზათა საძიებელი სისტემა არ არსებობს.

შესატანი მონაცემები: შესატან მონაცემთა ფაილის პირველ სტრიქონში ჩაწერილია ერთი ჰარით გამოყოფილი ორი მთელი N და W რიცხვი, სადაც W (1≤W≤6000) იმ კვირათა რაოდენობაა, რომლებიც პროგრამამ უნდა დაამუშაოს.

თითოეული კვირისათვის მოცემულია ერთ სტრიქონში ჩაწერილი ძროხების მიერ აღმოჩენილი ველურ ცხოველთა ბილიკი. ეს სტრიქონი შეიცავს თითო ჰარით გამოყოფილ სამ მთელ რიცხვს: ბილიკის ბოლოებს (შესაბამისი ორი ნაკვეთის ნომერს) და ამ ბილიკის სიგრძეს (1..10000). ველურ ცხოველთა არცერთი ბილიკის ორივე ბოლო არ მდებარეობს ერთი და იგივე ნაკვეთში.

გამოსატანი მონაცემები: როგორც კი თქვენი პროგრამისათვის ცნობილი გახდება ველურ ცხოველთა ახლად აღმოჩენილი ბილიკის შესახებ, მან უნდა გამოიტანოს ერთ სტრიქონში ჩაწერილი იმ გზათა მინიმალური ჯამური სიგრძე, რომლებიც უნდა იქნას არჩეული ყველა ნაკვეთის შესაერთებლად. თუ ყველა ნაკვეთის შეერთება შეუძლებელია ისე, რომ გამოყენებულ იქნას მხოლოდ ველურ ცხოველთა ბილიკები, მაშინ პროგრამამ უნდა გამოიტანოს "-1".

პროგრამამ მუშაობა უნდა დაამთავროს მაშინ, როცა გამოიტანს პასუხს უკანასკნელი კვირისათვის.

დიალოგის მაგალითი:

შეტანა	გამოტანა	განმარტება
4 6		
1 2 10		
	-1	მე-4 ნაკვეთი არ არის შეერთებული სხვა ნაკვეთებთან
1 3 8		
	-1	მე-4 ნაკვეთი არ არის შეერთებული სხვა ნაკვეთებთან
3 2 3		
	-1	მე-4 ნაკვეთი არ არის შეერთებული სხვა ნაკვეთებთან
1 4 3		
	14	უნდა აირჩეს 1 4 3, 1 3 8 და 3 2 3
1 3 6		
	12	უნდა აირჩეს 1 4 3, 1 3 6 და 3 2 3
2 1 2		
	8	უნდა აირჩეს 1 4 3, 2 1 2 და 3 2 3
	program exit	

შეზღუდვები: მუშაობის დრო – 1 წამი და მეხსიერება – 64მბ.

მითითება. თუკი გამოვიყენებთ პრიმის ან კრასკალის ალგორითმს და ყოველი ახალი ბილიკის დამატების შემდეგ თავიდან გამოვთვლით მინიმალურ დამფარავ ხეს, დროის ლიმიტში ჩაეტევა მხოლოდ ტესტების ნახევარი. ალგორითმის დასაჩქარებლად საჭიროა, რომ ახალი ბილიკის დამატების შემდეგ გამოყენებულ იქნას წინა ხის აგებისას მიღებული ინფორმაცია. ამ დებულების რეალიზაციის ერთ-ერთი გზაა, კრასკალის ალგორითმში ზრდადობით დალაგებულ ბილიკებს შორის მოვნიშნოთ ისინი, რომლებიც მინიმალურ დამფარავ ხეში შედიან და ახალი ბილიკის დამატებისას ვიანგარიშოთ არა თავიდან, არამედ იმ ადგილიდან, სადაც ახალი ბილიკი მოთავსდა. კრასკალის ალგორითმში ახალი წიბოს დამატება გავლენას ვერ მოახდენს მასზე მცირე ზომის წიბოების შერჩევაზე.

7.6. მოგზაურობა

(საქართველოს მოსწავლეთა ოლიმპიადა, რესპუბლიკური ტური, 2001-02 წელი)

მოცემულია N რაოდენობის ქალაქი, რომლებიც გადანომრილია 1-დან N-მდე მთელი რიცხვებით. ყოველი ორი ქალაქი ერთმანეთთან დაკავშირებულია არაუმეტეს 1 გზით (ყველა გზა ორმხრივია). მოცემულია აგრეთვე, ყოველი გზის გასავლელად საჭირო დრო საათებში.

გამოცხადდა, რომ ერთ-ერთ გზაზე დაიწყო რემონტი და იგი ბლოკირებულია მოძრაობისთვის, მაგრამ არ ითქვა, უშუალოდ რომელ გზაზე იყო საუბარი. მგზავრს სურს N-ური ქალაქიდან ჩავიდეს პირველ ქალაქში (ეს შესაძლებელია მიუხედავად იმისა, თუ რომელი გზაა ბლოკირებული, ანუ შესატანი მონაცემები უზრუნველყოფს ამოცანის ამოხსნის არსებობას) ისე, რომ არ იმოძრაოს ბლოკირებულ გზაზე და მისი მარშრუტი იყოს უმოკლესი. ამასთან მას აინტერესებს, თუ რა მაქსიმალური დრო დაეხარჯება ყველაზე უარეს შემთხვევაში (ანუ იმის მიხედვით, თუ რომელი გზა იქნება ბლოკირებული).

დაწერეთ პროგრამა, რომელიც დაეხმარება მგზავრს გამოთვალოს ის მაქსიმალური დრო, რომელიც შეიძლება დაეხარჯოს მას არაბლოკირებული გზებით პირველ ქალაქში უმოკლესი მარშრუტით ჩასასვლელად.

შესატანი მონაცემები: შესატან მონაცემთა JOURNEY.IN ფაილის პირველი სტრიქონი შეიცავს ორ მთელ N და M რიცხვს($1 \le N \le 1000$, $1 \le M \le N*(N-1)/2$) – ქალაქების რაოდენობას და ქალაქებს შორის გზების რაოდენობას. მომდევნო M რაოდენობის სტრიქონიდან თითოეულში ჩაწერილია 3 მთელი A, B და V რიცხვი ($1 \le A, B \le N$, $1 \le V \le 1000$), რომელთაგან პირველი და მეორე არის იმ ქალაქების ნომრები, რომელთა შორის გზა არსებობს, ხოლო მესამე – ამ გზის გასავლელად საჭირო დრო. სტრიქონებში მონაცემები ერთმანეთისაგან თითო ჰარითაა გამოყოფილი.

გამოსატანი მონაცემები: გამოსატან მონაცემთა JOURNEY.OUT ფაილის ერთადერთ სტრიქონში უნდა ჩაიწეროს დროის ის მაქსიმალური რაოდენობა, რომელიც შიძლება მგზავრს დასჭირდეს არაბლოკირებული გზებით პირველ ქალაქში უმოკლესი მარშრუტით ჩასასვლელად.

ფაილი JOURNEY.IN	ფაილი JOURNEY.IN	ფაილი JOURNEY.IN
5 6	6 7	5 7
1 2 4	1 2 1	1 2 8
1 3 3	2 3 4	1 4 10
2 3 1	3 4 4	2 3 9
2 4 4	4 6 4	2 4 10
2 5 7	1 5 5	2 5 1
4 5 1	2 5 2	3 4 7
	5 6 5	3 5 10
ფაილი JOURNEY.OUT	ფაილი JOURNEY.OUT	ფაილი JOURNEY.OUT
11	13	27

მითითება. ამოცანა უნდა ამოიხსნას დეიქსტრას მეთოდის გამოყენებით, ამასთან საჭირო იქნება მისი რამდენჯერმე გამოყენება. დეიქსტრას მეთოდი გვაძლევს არა მარტო უმცირესი მარშრუტის სიგრძეს, არამედ თავად მარშრუტსაც. ჯერ უნდა ავაგოთ უმცირესი მარშრუტი იმ შემთხვევისათვის, როცა არცერთი გზა არაა ბლოკირებული (ვუწოდოთ მას საწყისი მარშრუტი). თუკი ბლოკირებულია გზა, რომელიც საწყის მარშრუტში არ შედის, ცხადია, ხელახალი გამოთვლა საჭირო არ არის, ხოლო თუ ბლოკირებულია ამ მარშრუტში შემავალი გზა, მაშინ უნდა გამოვთვალოთ ახალი მინიმუმი (ანუ სხვა მარშრუტი, რომელიც ვერ იქნება საწყისზე ნაკლები) და ნაპოვნ მინიმუმებს შორის უდიდესი იქნება ჩვენი ამოცანის პასუხი. დეიქსტრას ალგორითმის გამოყენება დაგჭირდება იმდენჯერ, რამდენი გზაც შედის საწყის მარშრუტში.

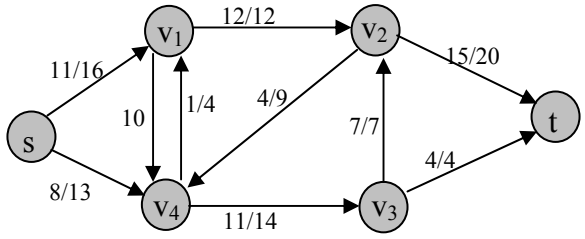
8. ქსელი და ნაკადი

8.1. ძირითადი ცნებები და განსაზღვრებები

ქსელი (flow network) ეწოდება ორიენტირებულ $G=(V,E)$ გრაფს, რომლის თითოეულ $(u,v)\in E$ წიბოს შეესაბამება $c(u,v)\geq 0$ რიცხვი, რომელსაც **გამტარუნარიანობას** (capacity) უწოდებენ. თუ $(u,v)\notin E$ ითვლება, რომ $c(u,v)=0$. გრაფში გამოყოფილია ორი წვერო: s **სათავე** (source) და t **ბოლო** (sink). სიმარტივისათვის ვთვლით, რომ გრაფში არაა "უსარგებლო" წვეროები, ანუ ნებისმიერი წვერო დევს სათავიდან ბოლოსაკენ მიმავალ გზაზე. ასეთ შემთხვევაში გრაფი ბმულია და $|E|\geq|V|-1$.

ვთქვათ, მოცემულია ქსელი $G=(V,E)$, რომლის გამტარუნარიანობა მოცემულია c ფუნქციით. ქსელს გააჩნია სათავე s და ბოლო t. **ნაკადი** (flow) G ქსელში ეწოდება $f:V\times V\to R$ ფუნქციას, რომელსაც გააჩნია შემდეგი სამი თვისება: ა) **გამტარუნარიანობის შეზღუდულობა** (capasity constraint)– $f(u,v)\leq c(u,v)$ ყველა $u,v\in V$-თვის (რომ ნაკადი ერთი წვეროდან მეორეში არ აღემატება წიბოს გამტარუნარიანობას); ბ) **ირიბი სიმეტრია** (skew symmetry) – $f(u,v)=-f(v,u)$ ყველა $u,v\in V$-თვის (უარყოფითი რიცხვები აღნიშნავენ ნაკადს საწინააღმდეგო მიმართულებით); გ**) ნაკადის შენახვა** (flow conservation) – $\sum_{u\in V} f(u,v)=0$ ნებისმიერი (V-{s,t})-თვის (ნებისმიერი წვეროსათვის, გარდა სათავისა და ბოლოსი, შემავალი და გამომავალი ნაკადები ტოლია).

$f(u,v)$ შეიძლება იყოს როგორც დადებითი, ასევე უარყოფითი. ის განსაზღვრავს თუ რა სიდიდის ნაკადი მიემართება u-დან v წვეროსაკენ. უარყოფითი მნიშვნელობა შეესაბამება მოძრაობას საპირისპირო მიმართულებით.



ნახ. 8.1.

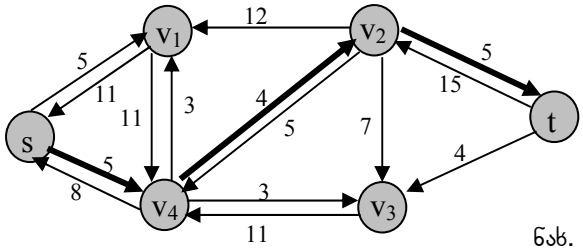
f ნაკადის **სიდიდე** (value) განისაზღვრება, როგორც ჯამი $\sum_{u\in V} f(s,v)$, ანუ როგორც s სათავიდან გამომავალი ნაკადების ჯამი და აღინიშნება $|f|$-ით. **მაქსიმალური ნაკადის ამოცანა** (maximum-flow problem) მდგომარეობს შემდეგში: მოცემული G ქსელისათვის ვიპოვოთ მაქსიმალური სიდიდის ნაკადი.

აქვე უნდა აღინიშნოს, რომ შემსვედრი ნაკადების შემთხვევაში შეგვიძლია განვიხილოთ მხოლოდ მათი სხვაობა უფრო დიდი ნაკადის მიმართულებით და თუკი გვაქვს ქსელი რამდენიმე სათავით ან ბოლოთი, ამოცანა დაიყვანება ერთი სათავის ან ერთი ბოლოს მქონე ქსელზე, შესაბამისად საერთო სათავის ან საერთო ბოლოს შემოღებით.

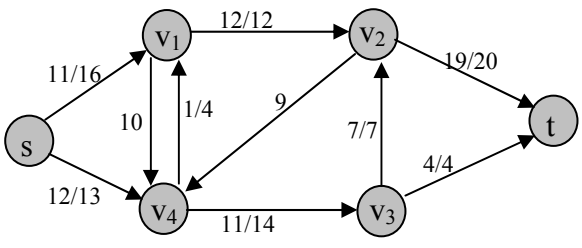
8.2. ფორდ-ფალკერსონის მეთოდი

ვთქვათ, მოცემულია ქსელი $G=(V,E)$, რომელსაც გააჩნია სათავე s და ბოლო t და ვთქვათ, f ნაკადია ამ ქსელში. წვეროთა ნებისმიერი წყვილისათვის განვიხილოთ **ნარჩენი გამტარუნარიანობა u-დან v-ში** (residual capacity of (u,v)), რომელიც ასე განისაზღვრება: $c_f(u,v)=c(u,v)-f(u,v)$. ეს სხვაობა განსაზღვრავს, თუ კიდევ რამხელა ნაკადი შეგვიძლია მივმართოთ u-დან v-ში. მაგალითად, თუ $c(u,v)=16$ და $f(u,v)=11$, მაშინ $c_f(u,v)=5$.

ქსელს $G_f=(V,E_f)$, სადაც $E_f=\{(u,v)\in V\times V: c_f(u,v)>0\}$, უწოდებენ f ნაკადის მიერ წარმოქმნილ G ქსელის **ნარჩენ ქსელს** (residual network), ხოლო მის წიბოებს უწოდებენ **ნარჩენ წიბოებს** (residual edges).



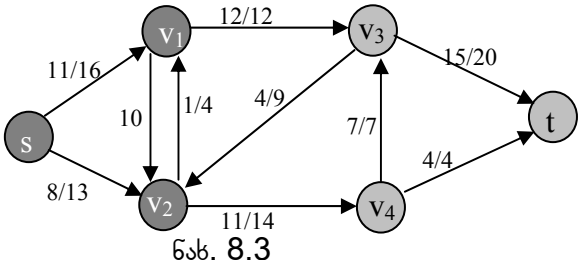
ნახ. 8.2



შევნიშნოთ, რომ აუცილებელი არაა ნარჩენი წიბო საწყის ქსელში არსებობდეს (მაგალითად, (v_1,s) და (v_2,v_3) წიბოები). ასეთი წიბო u-დან v-ში წარმოიქმნება, როცა $f(u,v)<0$, ანუ როცა არსებობს საწინააღმდეგო მიმართულების ნაკადი. მაშასადამე, თუ (u,v) წიბო ეკუთვნის ნარჩენ ქსელს, მაშინ (u,v) და (v,u)-დან ერთ-ერთი მაინც არსებობდა საწყის ქსელში.

ვთქვათ, f ნაკადია ქსელში G=(V,E). **შემავსებელი გზა** (augmenting path) ეწოდება უბრალო გზას s სათავიდან t ბოლომდე G_f ნარჩენ ქსელში. ნარჩენი ქსელის განსაზღვრებიდან გამომდინარეობს, რომ შემავსებელი გზის ყველა წიბოში გამტარუნარიანობის გადაუჭარბებლად შესაძლოა გაიაროს კიდევ რაღაც რაოდენობის ნაკადმა. იმ უდიდესი ნაკადის რაოდენობას, რომელიც შეიძლება გადაიგზავნოს შემავსებელი p გზით, ეწოდება **p გზის ნარჩენი გამტარუნარიანობა** (residual capacity of p).

G=(V,E) ქსელის **ჭრილი** (cut) ეწოდება V სიმრავლის გაყოფას ორ ნაწილად S და T=V\S, სადაც $s\in S$ და $t\in T$. **ჭრილის გამტარუნარიანობა** (capacity of the cut) ეწოდება ჭრილის მიერ გადაკვეთილი წიბოების გამტარუნარიანობათა ჯამს. მოცემული f ნაკადისათვის **(S,T) ჭრილის ნაკადი** (net flow across (S,T)) განისაზღვრება როგორც f(S,T) ჯამი ჭრილის მიერ გადაკვეთილი წიბოებისათვის. **მინიმალური ჭრილი** (minimum cut) ეწოდება მინიმალური გამტარუნარიანობის მქონე ჭრილს.



ნახ. 8.3

ნახ. 8.3-ზე ნაჩვენებია ჭრილი $(\{s,v_1,v_2\}, \{v_3,v_4,t\})$. ნაკადი ამ ჭრილში ტოლია $f(v_1,v_3)+f(v_2,v_3)+f(v_2,v_4)= 12+(-4)+11=19$, ხოლო გამტარუნარიანობა ტოლია $c(v_1,v_3)+c(v_2,v_4)=26$. როგორც ჩანს ჭრილის გამტარუნარიანობისაგან განსხვავებით, ჭრილის ნაკადმა შეიძლება უარყოფითი მნიშვნელობებიც მიიღოს.

მტკიცდება, რომ ყველა ჭრილში ნაკადის სიდიდე ერთნაირია და იგი ქსელში ნაკადის სიდიდის ტოლია. აქედან კი გამომდინარეობს, რომ ქსელში ნებისმიერი ნაკადის სიდიდე არ აღემატება ამ ქსელის ნებისმიერი ჭრილის გამტარუნარიანობას.

თეორემა 8.1. (მაქსიმალურ ნაკადსა და მინიმალურ ჭრილზე). ვთქვათ, f ნაკადია ქსელში G=(V,E). მაშინ ტოლფასია შემდეგი მტკიცებულებები: ა) f ნაკადი მაქსიმალური სიდიდისაა G ქსელში; ბ) G_f ნარჩენი ქსელი არ შეიცავს შემავსებელ გზებს; გ) რომელიღაც (S,T) ჭრილისათვის G ქსელში სრულდება ტოლობა $|f|=c(S,T)$.

მაქსიმალური ნაკადის მოსაძებნად გამოიყენება **ფორდ-ფალკერსონის მეთოდი**. საუბარია სწორედ მეთოდზე და არა ალგორითმზე, რადგან არსებობს რამდენიმე ალგორითმი, რომელიც რეალიზაციას უკეთებს ამ მეთოდს. ფორდ-ფალკერსონის მეთოდით მაქსიმალური ნაკადის ძებნა ხორციელდება ბიჯებად. თავიდან ნაკადი ნულოვანი სიდიდისაა. ყოველ ბიჯზე ხდება ნაკადის მნიშვნელობის ზრდა შემავსებელი გზების პოვნით. ეს ბიჯი მეორდება შემავსებელი გზების ამოწურვამდე. მიღებული ნაკადი თეორემის თანახმად მაქსიმალური იქნება.

```
FORD-FULKERSON(G,s,t)
1 FOR თითოეული (u,v)∈E[G]-თვის {
2       f[u,v]=0
3       f[v,u]=0  }
4 WHILE სანამ ნარჩენ Gf ქსელში არსებობს შემავსებელი p გზა  {
5           cf(p)=min{cf(u,v): (u,v) შედის p-ში}
6           FOR თითოეული (u,v)-სათვის p-დან    {
7                   f[u,v]=f[u,v]+cf(p)
8                   f[v,u]=-f[u,v]                  }          }
```

მოცემულ ალგორითმში f[v,u] მასივი გამოიყენება ნაკადის მიმდინარე მნიშვნელობების შესანახად. 4-8 სტრიქონებში ალგორითმი პოულობს შემავსებელ p გზას G_f ქსელში და ზრდის f ნაკადს. თუკი შემავსებელი გზა აღარ არის, ნაკადი მაქსიმალურია.

პროცედურის მუშაობის დრო დამოკიდებულია p გზის ძებნის სიჩქარეზე (მე-4 სტრიქონი). თუ ნაკადის სიდიდე გაიზრდება სულ უფრო და უფრო მცირე ბიჯებით, შესაძლოა პროცედურამ რეალურ დროში პასუხი ვერც იპოვოს. მაგრამ თუ p გზის საძებნად გამოვიყენებთ განივად ძებნას, შედეგი პოლინომიალურ დროში მიიღება. ფორდ-ფალკერსონის მეთოდის ამ რეალიზაციას უწოდებენ ედმონდს-კარპის ალგორითმს და მისი მუშაობის დრო არის $O(VE^2)$.

არსებობენ უფრო სწრაფი ალგორითმებიც. მათ შორისაა "წინარენაკადის გაშვების" ალგორითმი. მისი მუშაობის დროა $O(V^2E)$. ედმონდ-კარპის ალგორითმისაგან განსხვავებით ის არ იხილავს მთელს ნარჩენ ქსელს ყოველ ბიჯზე, არამედ მოქმედებს ერთი წვეროს გარშემო. ის არც ნაკადის შენახვის კანონის შესრულებას მოითხოვს და კმაყოფილდება წინარენაკადის თვისებების შესრულებით. **წინარენაკადი** (preflow) ეწოდება $f:V\times V\to R$ ფუნქციას, რომელიც ირიბად სიმეტრიულია, აკმაყოფილებს გამტარუნარიანობის შეზღუდვებს და შენახვის შესუსტებულ კანონს: $f(V,u)\geq 0$, ყველა წვეროსათვის სათავის გარდა. მაშასადამე, სათავის გარდა ყოველ u წვეროში არის რაღაც არაუარყოფითი სიჭარბე $e(u)=f(V,u)$. დადებითი სიჭარბის მქონე წვეროს (სათავისა და ბოლოს გარდა) ვუწოდოთ გადავსებული.

აღვწეროთ ორი ოპერაცია: "წინარენაკადის გაშვება" და "წვეროს ამაღლება".

ფორდ-ფალკერსონის მეთოდის გამოყენებისას ჩვენ ყოველ მომენტში საქმე გვაქვს მილებში არსებულ სითხის ნაკადთან სათავიდან ბოლოსაკენ. თითოეული ბიჯისას ჩვენ ვზრით ნაკადს შემავსებელი გზის მოძებნით. სითხე გზაში არსად არ იღვრება. წინარენაკადის გაშვების ალგორითმში ჭარბი სითხე ყოველ წვეროში იღვრება. გარდა ამისა, დიდ როლს თამაშობს მთელი მნიშვნელობის მქონე პარამეტრი, რომელსაც წვეროს სიმაღლეს ვუწოდებთ – იგი განსაზღვრავს, თუ საით უნდა გავუშვათ ჩვენ ჭარბი სითხე.

სათავის სიმაღლე ყოველთვის $|V|$-ს ტოლია, ხოლო ბოლოს სიმაღლე – 0. ყველა სხვა წვერო თავდაპირველად იმყოფება ნულოვან სიმაღლეზე, ხოლო შემდეგ თანდათანობით ზემოთ გადაადგილდებიან. დასაწყისში სათავიდან უშვებენ იმდენ სითხეს, რამდენის საშუალებასაც იძლევიან სათავიდან გამომავალი წიბოები თავიანთი გამტარუნარიანობით. სითხის წარმოქმნილი სიჭარბე თავდაპირველად უბრალოდ იღვრება, შემდეგ კი მიმართული იქნება სვლის გასაგრძელებლად..

ალგორითმის მუშაობის პროცესში განვიხილოთ რომელიმე u წვერო, რომელშიც შეიქმნა სითხის სიჭარბე, ხოლო მისგან გამომავალი დაუტვირთავი წიბოები (ანუ ისეთი წიბოები, რომელშიც შეიძლება გავუშვათ რაღაც რაოდენობის სითხე) მიემართებიან ან იმავე სიმაღლის ან უფრო მაღლა მდებარე წვეროებისაკენ. ასეთ შემთხვევაში სრულდება u წვეროს "ამაღლების" ოპერაცია, რის შემდეგაც წვერო u აღმოჩნდება ერთი ერთეულით მაღლა ყველაზე ქვემოთ მყოფ მეზობელ წვეროზე, რომლისკენაც მიემართება დაუტვირთავი წიბო. სხვაგვარად რომ ვთქვათ, ჩვენ ავწევთ u წვეროს ზუსტად იმდენზე, რომ წარმოიქმნას ქვემოთ მიმავალი დაუტვირთავი წიბო.

თითოეული წვეროსათვის ანალოგიური ქმედებით ჩვენ მივაღწევთ ქსელის ბოლოში მაქსიმალური სითხის ჩადინებას, თუმცა წინარენაკადი ჯერ კიდევ ვერ იქნება ნაკადი, რადგან ზოგიერთ წვეროში სითხე შეიძლება ისევ დაიღვაროს. ასეთი წვეროების კიდევ უფრო მაღლა აწევით (რის შედეგადაც ზოგიერთი შესაძლოა სათავეზე მაღლა აღმოჩნდეს) ჭარბი სითხე ისევ სათავეში დაბრუნდება (ანუ შემცირდება ნაკადი სათავიდან) და წინარენაკადი გადაიქცევა ნაკადად, რომელიც იქნება კიდევაც მაქსიმალური მოცემულ ქსელში.

მაშასადამე, წინარენაკადის გაშვების ალგორითმი იყენებს ორ ოპერაციას: "წინარენაკადის გაშვება" მეზობელ წვეროში და "წვეროს ამაღლება". მივცეთ მათ ზუსტი განმარტებები.

ვთქვათ, მოცემულია ქსელი $G=(V,E)$, რომელსაც გააჩნია სათავე s და ბოლო t, ხოლო f წინარენაკადია G-ში. ფუნქციას $h:V\to N$ უწოდებენ სიმაღლის ფუნქციას f წინარენაკადისათვის, თუ $h(s)=|V|$, $h(t)=0$ და $h(u)\le h(v)+1$ ნებისმიერი (u,v) წიბოსათვის ნარჩენი E_f ქსელიდან.

ლემა 8.2. ვთქვათ f წინარენაკადია $G=(V,E)$-ში და h სიმაღლის ფუნქციაა. მაშინ თუკი $u,v\in V$ წვეროებისთვის სრულდება უტოლობა $h(u)>h(v)+1$, მაშინ ნარჩენი ქსელი არ შეიცავს (u,v) წიბოს ("მკვეთრად ქვემოთ მიმავალ წიბოებში გადის მაქსიმალური ნაკადი").

პროცედურა PUSH(u,v) გამოიყენება, თუ წვერო u გადავსებულია ($e(u,v)>0$), (u,v) წიბო არაა დატვირთული ($c_f(u,v)>0$) და $h(u)=h(v)+1$. პროცედურის მიხედვით x წვეროს სიმაღლე ინახება h[x]-ში, ჭარბი ნაკადი e[x]-ში, ხოლო x-დან y-ში მიმავალი წინარენაკადი f[x,y]-ში.

PUSH(u,v)

1 მოცემულია: $e(u,v)>0$, $c_f(u,v)>0$ და $h(u)=h(v)+1$

2 საჭიროა: u-დან v-ში გავუშვათ $d_f(u,v)=\min(e[u], c_f(u,v))$ რაოდენობის ნაკადი.

3 $d_f(u,v)=\min(e[u], c_f(u,v))$

4 $f(u,v)=f[u,v]+d_f(u,v)$

5 $f(v,u)=-f(u,v)$

6 $e[u]=e[u]-d_f(u,v)$

7 $e[v]=e[v]-d_f(u,v)$

ბუნებრივია, რომ ჭარბი ნაკადი u წვეროში და (u,v) წიბოს ნარჩენი გამტარუნარიანობა დადებით სიდიდეებს წარმოადგენენ. ამიტომ შესაძლებელია u-დან v-ში გავუშვათ $\min(e[u], c_f(u,v))>0$ რაოდენობის ნაკადი ისე, რომ არ გადავაჭარბოთ გამტარუნარიანობას და არ გავხადოთ ნაკადის სიჭარბე უარყოფითი. პირობა $h(u)=h(v)+1$ იძლევა იმის გარანტიას, რომ დამატებითი ნაკადი გაშვებული იქნება მხოლოდ იმ წიბოებში, რომლებიც ერთი ერთეულით დაბლა მდებარეობენ. ლემის მიხედვით, სხვა – უფრო მკვეთრად ქვემოთ მიმავალი წიბოები უკვე ისედაც დატვირთული იქნებიან.

PUSH ოპერაციას უწოდებენ "ნაკადის გაშვებას" u-დან v-ში u წვეროსათვის. მას ეწოდება **დამტვირთავი** (saturating), თუ ოპერაციის შედეგად (u,v) წიბო **დატვირთული** (saturated) ხდება (ე.ი. $c_f(u,v)=0$ და წიბო ქრება ნარჩენი ქსელიდან), წინააღმდეგ შემთხვევაში ეწოდება – **არადამტვირთავი** (nonsaturating).

პროცედურა LIFT(u) ამაღლებს გადავსებულ u წვეროს იმ მაქსიმალურ სიმაღლემდე, რომელიც დასაშვებია სიმაღლის ფუნქციის განსაზღვრებით. წვეროს სიმაღლე უნდა აღემატებოდეს ნარჩენ ქსელში მეზობელი წვეროს სიმაღლეს არაუმეტეს 1 ერთეულით. თუკი არსებობს 1 ერთეულით ქვემოთ მყოფი მეზობელი, მაშინ შეიძლება ნაკადის გაშვება, მაგრამ არ შეიძლება წვეროს ამაღლება. თუკი არცერთი მეზობელი წვერო არაა ქვემოთ, მაშინ შეიძლება წვეროს ამაღლება და არ შეიძლება ნაკადის გაშვება.

LIFT(u)

1 მოცემულია: u გადავსებულია, ნებისმიერი $(u,v)\in E_f$-თვის სრულდება უტოლობა $h[u]\le h[v]$.

2 საჭიროა: გავზარდოთ h[u], u წვეროდან ნაკადის გაშვების მოსამზადებლად.

3 $h[u]=1+\min\{h[v]:(u,v)\in E_f\}$

შევნიშნოთ, რომ თუ u წვერო გადავსებულია, მაშინ E_f–ში მოიძებნება u-დან გამომავალი ერთი მაინც წიბო.
ალგორითმი იწყება პროცედურით INITIALIZE-PREFLOW, რომელიც იძლევა წინარენაკადს.

$$f\{u,v] = \begin{cases} c(u,v), & \text{როცა } u=s \\ -c(u,v), & \text{როცა } v=s \\ 0 & \text{სხვა შემთხვევებში} \end{cases}$$

```
INITIALIZE-PREFLOW(G,s)
1  FOR ყოველი წვეროსათვის u∈V[G]  {
2          h[u]=0
3          e[u]=0     }
4  FOR ყოველი წიბოსათვის (u,v)∈V[G]     {
5          f[u,v]=0
6          f[v,u]=0    }
7  h[s]=|V[G]|
8  FOR ყოველი წვეროსათვის u∈Adj[s]   {
9          f[s,u]=c(s,u)
10         f[u,s]=-c(s,u)
11         e[u]=c(s,u)    }
```

h მასივში ინახება სიმაღლეები, e მასივში – ჭარბი ნაკადი, c(u,v) მასივში – გამტარუნარიანობები. ნაკადი ჩაიწერება f მასივში.

წინარენაკადის გაშვების ალგორითმის ზოგადი სქემა:

```
GENERIC-PREFLOW-PUSH(G)
1 INITIALIZE-PREFLOW(G,s)
2 WHILE სანამ შესაძლებელია ამაღლების ან გაშვების ოპერაციები  {
3          შევასრულოთ ერთ-ერთი ოპერაცია.   }
```

მტკიცდება, რომ სანამ თუნდაც ერთი წვერო გადავსებულია, შესაძლებელია ან წვეროს ამაღლება ან წინარენაკადის გაშვება. ასევე მტკიცდება, რომ ალგორითმი სასრულია და მისი დამთავრების მომენტში წინარენაკადი მაქსიმალური ნაკადი ხდება.

8.3. ალგორითმი "ამაღლება-და-თავიდან"

ეს ალგორითმი ქსელის ყველა წვეროს ინახავს სიის სახით. ალგორითმი განიხილავს ამ სიას (განხილვა იწყება პირველი წევრიდან) და პოულობს მასში გადავსებულ წვეროს. შემდეგ ხდება ამ წვეროს "მომსახურება", ანუ მისთვის ტარდება "ამაღლების" და "ნაკადის გაშვების" ოპერაციები, ვიდრე ჭარბი ნაკადი ამ წვეროში 0 არ გახდება. თუკი ამისათვის წვეროს ამაღლება გახდა საჭირო, მაშინ მას გადაანაცვლებენ სიის თავში და სიის განხილვა თავიდან იწყება.

ალგორითმის ანალიზისას საჭიროა "დასაშვები წიბო"-ს ცნება. ის განისაზღვრება როგორც ნარჩენი ქსელის წიბო, რომელშიც შესაძლებელია ნაკადის გაშვება:

ვთქვათ, f წინარენაკადია G=(V,E)-ში და h სიმაღლის ფუნქციაა. ვუწოდოთ (u,v) წიბოს **დასაშვები** (admissible), თუკი ის შედის ნარჩენ ქსელში ($c_f(u,v)>0$) და h(u)=h(v)+1. დანარჩენ წიბოებს ვუწოდოთ **დაუშვებელი** (inadmissible). აღვნიშნოთ $E_{f,h}$-ით დასაშვებ წიბოთა სიმრავლე. $G_{f,h}=(V,E_{f,h})$ ქსელს უწოდებენ **დასაშვებ წიბოთა ქსელს** (admissible network),. იგი შედგება იმ წიბოებისაგან, რომლებშიც შესაძლებელია ნაკადის გაშვება. რადგან დასაშვები წიბოს გასწვრივ სიმაღლე მცირდება, ადგილი აქვს ლემას:

ლემა 8.3. ვთქვათ, f წინარენაკადია G=(V,E)-ში და h სიმაღლის ფუნქციაა. მაშინ დასაშვებ წიბოთა ქსელი $G_{f,h}=(V,E_{f,h})$ არ შეიცავს ციკლებს.

"წინარენაკადის გაშვებისა" და "წვეროს ამაღლების" ოპერაციების გავლენა ქსელზე აისახება ლემებში:

ლემა 8.4. ვთქვათ, f წინარენაკადია G=(V,E)-ში და h სიმაღლის ფუნქციაა. ვთქვათ (u,v) დასაშვები წიბოა და u წვერო გადავსებულია. მაშინ (u,v) წიბოში შესაძლებელია ნაკადის გაშვება. ამ ოპერაციის შესრულებით ახალი დასაშვები წიბოები არ წარმოიქმნება, ხოლო წიბო (u,v) შეიძლება გახდეს დაუშვებელი.

ლემა 8.5. ვთქვათ, f წინარენაკადია G=(V,E)-ში და h სიმაღლის ფუნქციაა. თუ u წვერო გადავსებულია და მისგან არ გამოდის დასაშვები წიბოები, მაშინ შესაძლებელია u წვეროს ამაღლება. ამაღლების შედეგად წარმოიქმნება u წვეროდან გამომავალი ერთი მაინც დასაშვები წიბო და არ იარსებებს u წვეროში შემავალი დასაშვები წიბო.

ალგორითმი "ამაღლება-და-თავიდან" G=(V,E) ქსელის წიბოთა შესანახად იყენებს სპეციალურ ხერხს. კერძოდ, თითოეული u∈V წვეროსათვის არსებობს მეზობელი წვეროების ცალმხრივად დაკავშირებული სია N(u). v წვერო მოთავსდება ამ სიაში, თუკი (u,v)∈E ან (v,u)∈E. მაშასადამე, N(u) სია შეიცავს ყველა იმ v წვეროს, რომლისთვისაც (u,v) შესაძლოა გამოჩნდეს ნარჩენ ქსელში. ამ სიის პირველ წევრს აღნიშნავენ head[N[u]]-თი, v წვეროს მომდევნო მეზობელს – next-neighbor[v]. თუკი v წვერო უკანასკნელია სიაში next-neighbor[v]=NIL.

მეზობელ წვეროთა სიაში რიგითობა შეიძლება ნებისმიერი იყოს. ის არ იცვლება ალგორითმის მუშაობის პროცესშიც. ყოველი u წვეროსათვის მიმთითებელი current[u] ინახება N[u] სიის მომდევნო ელემენტზე. დასაწყისში current[u] დაყენებულია head[N[u]]-ზე.

გადავსებული u წვეროს დამუშავება გამოიხატება მის განტვირთვაში (discharge), რომლის დროსაც მთელ ჭარბ ნაკადს გაუშვებენ მეზობელ წვეროებში დასაშვები წიბოების გავლით. ზოგჯერ ამისათვის საჭიროა ახალი დასაშვები წვეროების წარმოქმნა u წვეროს ამაღლებით.

```
DISCHARGE(u)
1 WHILE e[u]>0    {
2       v=current[u]
3       if v=NIL
4         then {  LIFT(u)
5                 current[u]=head[N[u]]
6       elseif  cf(u,v)>0 and h[u]=h[v]+1
7         then {  PUSH(u,v)  }
8       else {  current[u]=next-neighbor[v]  }  }
```

while ციკლის ყოველი იტერაცია ასრულებს ერთ-ერთ ოპერაციას სამი შესაძლებლობიდან:

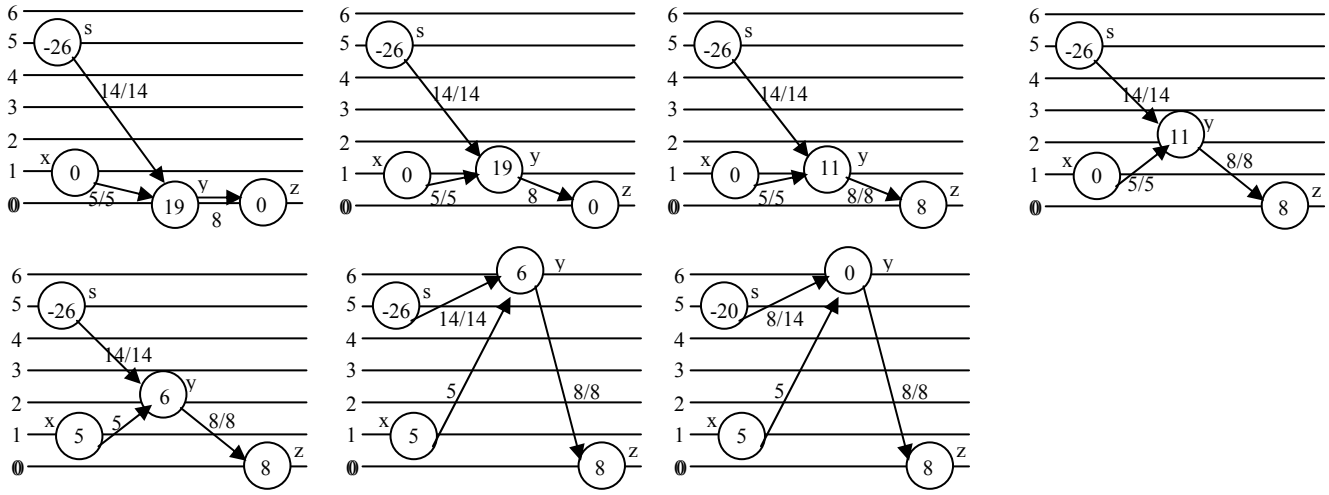
1. თუკი ჩვენ მივედით სიის ბოლოში (v=NIL), მაშინ ავამაღლებთ u წვეროს (მე-4 სტრიქონი) და გადავდივართ N[u] სიის დასაწყისში (მე-5 სტრიქონი).

2. თუკი ჩვენ არ მივსულვართ სიის ბოლომდე და (u,v) წიბო დასაშვებია (მოწმდება მე-6 სტრიქონში), მაშინ გავუშვებთ ნაკადს u-დან v-ში (მე-7 სტრიქონი).

3. თუკი ჩვენ არ მივსულვართ სიის ბოლომდე, მაგრამ (u,v) წიბო დაუშვებელია, მიმთითებელი current[u] გადაგვყავს ერთი პოზიციით წინ სიაში (მე-8 სტრიქონი).

შევნიშნოთ, რომ პროცედურის უკანასკნელი ოპერაცია შეიძლება მხოლოდ ნაკადის გაშვება იყოს, რადგან პროცედურა მხოლოდ მაშინ ჩერდება, როცა e[u] ჭარბი ნაკადი 0 ხდება, ხოლო არც წვეროს ამაღლება და არც მიმთითებლის გადანაცვლება ამ სიდიდეს არ ეხებიან.

შეგვიძლია ვთქვათ, რომ ალგორითმის შესრულებისას წვეროთა სია "კორექტულად დალაგებულია" შემდეგი აზრით: ნებისმიერი დასაშვები წიბოს დასაწყისი მის ბოლოზე წინ დგას სიაში.

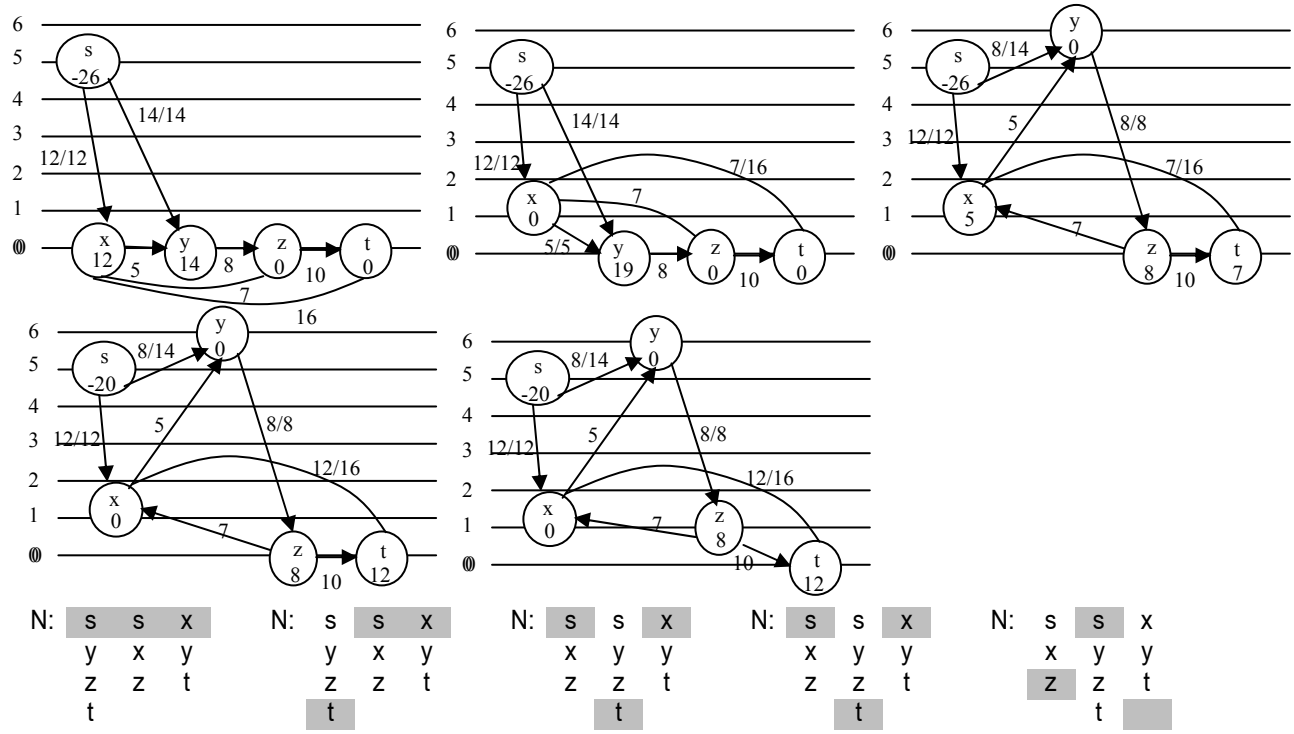


ნახ. 8.4

next[u]-თი აღვნიშნოთ წვერო, რომელიც სიაში u-ს შემდეგ მოდის. თუ u სიაში უკანასკნელია, მაშინ next[u]=0.

```
LIFT-TO-FRONT(G,s,t)
1 INITIALIZE-PREFLOW(G,s)
2 L=V[G]\{s,t} ნებისმიერი მიმდევრობით
3 for თითოეული წვეროსათვის u∈V[G]\{s,t} {
4       current[u]=head[N[u]]   }
5 u=head[L]
6 while u≠NIL {
7       old-heigt=h[u]
8      DISCHARGE(u)
9      if h[u]>old-height
10        then  { გადავანაცვლოთ u L სიის სათავეში  }
11     u=next[u]      }
```

ალგორითმის მუშაობის დროა $O(V^3)$.

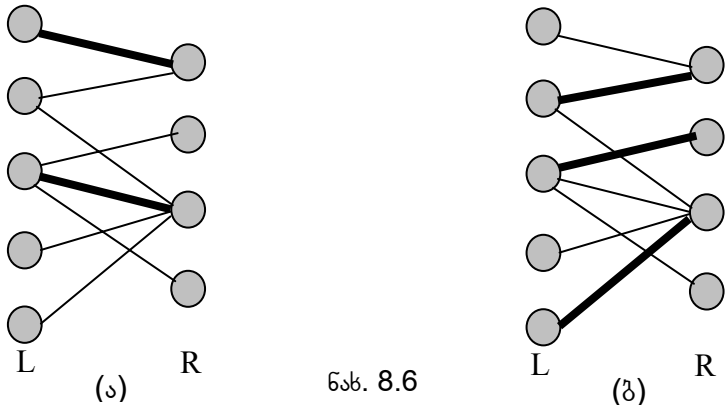


ნახ. 8.5

8.4. მაქსიმალური შეწყვილების ამოცანა ორნაწილიან გრაფში

ვთქვათ G=(V,E) – არაორიენტირებული გრაფია. **შეწყვილება** (matching) ვუწოდოთ წიბოთა $M \subseteq E$ სიმრავლეს, რომელთაც არ გააჩნიათ საერთო წვეროები. ვიტყვით, რომ $v \in V$ წვერო **შედის M შეწყვილებაში** (is matched by matching M), თუ M-ში მოიძებნება წიბო v წვეროთი, წინააღმდეგ შემთხვევაში v **თავისუფალია** (is unmatched). მაქსიმალური შეწყვილება (maximum matching) ეწოდება ისეთ M შეწყვილებას, რომელიც შეიცავს წიბოთა მაქსიმალურად შესაძლებელ რაოდენობას, ანუ $|m| \geq |M'|$ ნებისმიერი M' შეწყვილებისათვის. ჩვენ განვიხილავთ შეწყვილების ამოცანას მხოლოდ ორნაწილიან გრაფში, სადაც წვეროთა V სიმრავლე გაყოფილია ორ არაგადამფარავ L და R ქვესიმრავლედ და ნებისმიერი წიბო E სიმრავლიდან აერთებს L-ის რომელიმე წვეროს R-ის რომელიმე წვეროსთან.

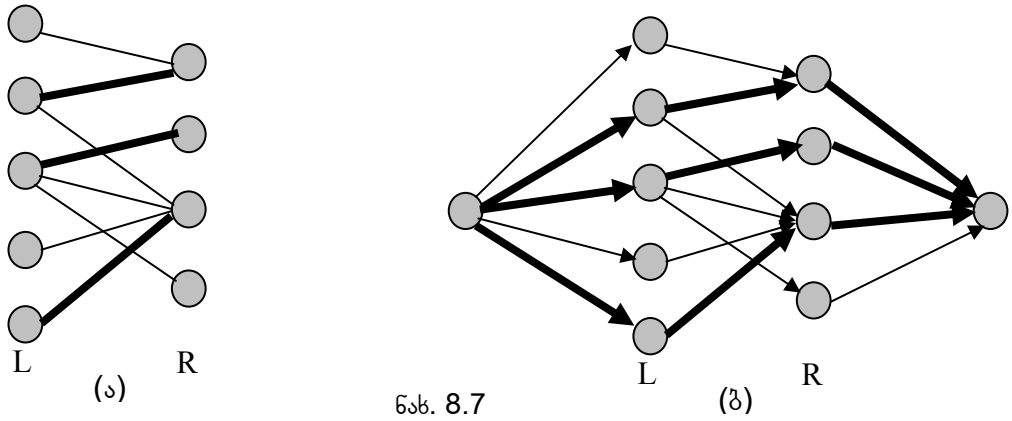
ნახ. 8.6-ზე გამოსახულია ორნაწილიანი გრაფი. 8.6ა-ზე მსხვილი ხაზებით ნაჩვენებია შეწყვილების ერთ-ერთი ვარიანტი, ხოლო 8.6ბ-ზე მაქსიმალური შეწყვილება, რომელიც 3-ის ტოლია.



ნახ. 8.6

ორნაწილიან გრაფში მაქსიმალური შეწყვილების ამოცანას "ქორწინების" ამოცანასაც უწოდებენ. გრაფის ერთი ნაწილი შეესაბამება საქმროებს, მეორე – საპატარძლოებს, ხოლო წიბოები მიუთითებენ, რომ ეს წყვილი თანახმაა შექმნას ოჯახი. ამოცანა კი მდგომარეობს იმაში, რომ რაც შეიძლება მეტი წყვილი დავაქორწინოთ.

ორნაწილიან გრაფში მაქსიმალური შეწყვილების საპოვნელად შესაძლებელია გამოყენებულ იქნას ფორდ-ფალკერსონის მეთოდი. ჯერ ჩამოვაყალიბოთ ასეთი განსაზღვრება: f ნაკადს G=(V,E) ქსელში ვუწოდოთ **მთელმნიშვნელიანი** (integer-valued), თუკი f(u,v)-ს ნებისმიერი მნიშვნელობა მთელია. არსებული ორნაწილიანი გრაფის ქსელად გარდასაქმნელად შემოვიღოთ ორი ახალი წვერო, რომლებიც იქნებიან სათავე (s) და ბოლო (t). სათავე შევაერთოდ L ნაწილის ყველა წვეროსთან, ბოლო კი – R ნაწილის ყველა წვეროსთან. ამოცანაში მოცემული და ჩვენს მიერ დამატებული თითოეული წიბოს გამტარუნარიანობა ჩავთვალოთ 1-ის ტოლად.

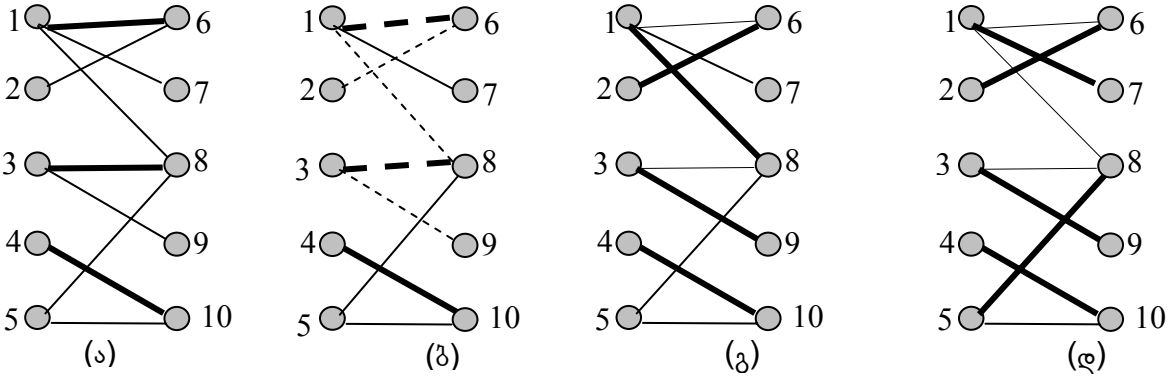


ნახ. 8.7

ნახ. 8.7-ზე ნაჩვენებია ორნაწილიანი გრაფი და მისი შესაბამისი ქსელი. მტკიცდება თეორემა, რომ ქსელში მაქსიმალური მთელმნიშვნელიანი ნაკადი ტოლია მაქსიმალური შეწყვილებისა ორნაწილიან გრაფში. მაქსიმალური ნაკადის პოვნის შემდეგ მსხვილი ისრებით აღნიშნულია ის წიბოები, რომლებშიც ნაკადი 1-ის ტოლია, ხოლო დანარჩენ წიბოებში კი – ნაკადი 0-ის ტოლია.

ზემოთ აღნიშნულიდან გამომდინარე, მაქსიმალური შეწყვილების საპოვნელად საკმარისია ფორდ-ფალკერსონის მეთოდის გამოყენება შესაბამის ქსელში, მაგრამ არსებობენ სხვა ალგორითმებიც ამ ამოცანის ამოსახსნელად. ერთ-ერთი მათგანია მონაცვლეობითი ჯაჭვების მეთოდი, რომლის ზოგადი ალგორითმია:

1) ავაგოთ ნებისმიერი სახის შეწყვილება. დროის ეკონომიისათვის უმჯობესია ხარბი ალგორითმი.
2) მოვძებნოთ ისეთი გზა ერთი ნაწილიდან მეორეში, რომელიც იწყება და მთავრდება თავისუფალი წვეროთი, ხოლო ამ გზაზე შეწყვილებაში შემავალი და თავისუფალი წიბოები მონაცვლეობენ.
3) ნაპოვნ გზაზე ყველა წიბოს შეფერილობა შევუცვალოთ საწინააღმდეგოთი: შეწყვილებაში შემავალი ყველა წიბო გავხადოთ თავისუფალი, ხოლო თავისუფალი წიბოები შეწყვილებაში შევიყვანოთ.
4) ვიდრე ასეთი გზა მოიძებნება, ყველაფერი გავიმეოროთ მეორე პუნქტიდან.



ნახ. 8.8

ალგორითმის იდეა იმაში მდგომარეობს, რომ რადგან მონაცვლეობითი ჯაჭვი იწყება და მთავრდება თავისუფალი წიბოთი, შეფერილობის შეცვლის შემდეგ შეწყვილებაში მყოფი წიბოები ყოველთვის 1-ით მეტი გახდება. ნახ. 8.8-ზე ნაჩვენებია ალგორითმის მუშაობის პრინციპი. (ა) ნახაზზე შეწყვილება აგებულია ხარბად: მარცხენა ნაწილის ყოველი წვერო უერთდება პირველივე (ზრდადობით) თავისუფალ წვეროს მარჯვენა ნაწილიდან. მიიღება სამი წყვილი: (1,6), (3,8) და (4,10), რომლებიც ნახაზზე შეფერადებულია (მსხვილი ხაზებითაა მოცემული). ამის შემდეგ ვეძებთ მონაცვლეობით ჯაჭვს. 1 წვეროდან ასეთი ჯაჭვი არ აიგება, 2 წვეროდან მოიძებნება ჯაჭვი: $2\to6\to1\to8\to3\to9$, რომელიც (ბ) ნაჩვენებია წყვეტილი ხაზებით. ამ ჯაჭვში სამი თავისუფალი და ორი გამოყენებული წიბოა. ხუთივე მათგანს შევუცვალოთ შეფერადება საპირისპიროთი. მივიღებთ ასეთ წყვილებს: (1,8), (2,6), (3,9) და (4,10), რაც ნაჩვენებია (გ) ნახაზზე. ამის შემდეგ მოიძებნება კიდევ ერთი მონაცვლეობითი ჯაჭვი $5\to8\to1\to7$, რომლის ინვერსიის შედეგად მიიღება საბოლოო პასუხი: (1,7), (2,6), (3,9), (4,10) და (5,8). მაქსიმალური შეწყვილება გამოსახულია (დ)-ზე.

უფრო დეტალურად განვიხილოთ მონაცვლეობითი ჯაჭვის აგების პროცესი. იგი წააგავს განივად ძებნას (იგივე ტალღურ მეთოდს). შესაბამისი გრაფის ნულოვან დონეს, საიდანაც ჯაჭვის უნდა დავიწყოთ, წარმოადგენს მარცხენა ნაწილის ყველა თავისუფალი წვერო. პირველ დონეზე და ზოგადად ყველა კენტნომრიან დონეზე გრაფს ემატება თავისუფალი წიბოები, რომლებიც გამოდიან წინა დონის წვეროებიდან, არ შედიან მიმდინარე შეწყვილებაში და უერთდებიან შეწყვილებაში შემავალ წვეროებს. ლუწნომრიან ბიჯზე გრაფს ემატება წიბოები, რომლებიც გამოყენებულია შეწყვილებაში და უერთდებიან თავისუფალ წვეროებს. ყველა ამ წიბოს ვიმახსოვრებთ და პროცესს ვაგრძელებთ მანამ, ვიდრე მოიძებნებიან ახალი წვეროები.

ალგორითმის კუბური სირთულისაა, რითაც მკვეთრად ჩამოუვარდება ფორდ-ფალკერსონის მეთოდს.

8.1. კუბიკები

(რუსეთის პირველი გუნდური ოლიმპიადა პროგრამირებაში მოსწავლეთა შორის, 2000 წ.)

მშობლებმა პეტიას აჩუქეს საბავშვო კუბიკები, რომლებზეც ანბანია გამოსახული. ყოველი კუბიკის ექვსიდან თითოეულ წიბოზე თითო სიმბოლო წერია. პეტიას სურს თავი მოიწონოს უფროსი დის წინაშე და კუბიკებისაგან მისი სახელის შედგენას ცდილობს. ეს არცთუ ისე იოლი საქმეა – სახელში შემავალი სხვადასხვა სიმბოლო შესაძლოა ერთ კუბიკზე აღმოჩნდეს და მაშინ პეტია ვერ შეძლებს ყველა სიმბოლოს გამოყენებას, თუმცა ერთი და იგივე სიმბოლო შესაძლოა სხვადასხვა კუბიკზეც აღმოჩნდეს. დაეხმარეთ პეტიას.

მოცემულია კუბიკების კრებული და დის სახელი. გაარკვიეთ შეიძლება თუ არა მოცემული კუბიკებით სახელის შედგენა და თუ შეიძლება, მაშინ გამოვიტანოთ კუბიკები შესაბამისი თანმიმდევრობით

შემავალი მონაცემები: პირველ სტრიქონში მოცემულია N (1≤N≤100) – კუბიკების რაოდენობა კრებულში. მეორე სტრიქონში მოცემულია პეტიას დის სახელი, რომელიც შედგენილია მხოლოდ მთავრული ლათინური სიმბოლოებისაგან და რომლის სიგრძეც არ აღემატება 100 სიმბოლოს. მომდევნო N სტრიქონი შეიცავს 6-6 სიმბოლოს (ასევე მთავრული სიმბოლოები), რომლებიც დაწერილია შესაბამის კუბიკზე.

გამომავალი მონაცემები: პირველ სტრიქონში გამოიტანეთ "YES", თუკი მოცემული კუბიკებით პეტიას დის სახელის შედგენა შესაძლებელია, წინააღმდეგ შემთხვევაში გამოიტანეთ – "NO" .

თუკი პასუხია "YES", მეორე სტრიქონში გამოიტანეთ M განსხვავებული რიცხვი 1..N დიაპაზონიდან, სადაც M სიმბოლოთა რაოდენობაა პეტიას დის სახელში. i-ური რიცხვი უნდა წარმოადგენდეს კუბიკის ნომერს, რომელიც უნდა დაიდოს i-ურ ადგილას პეტიას დის სახელის შედგენისას. კუბიკები გადანომრილია 1-დან N-მდე იმ მიმდევრობით, რომლითაც ისინი მოცემულია შემავალ ფაილში. თუკი ამონახსნი რამდენიმეა, გამოიტანეთ ერთ-ერთი. რიცხვები გაჰყავით ჰარებით.

მაგ.:

INPUT.TXT	OUTPUT.TXT	INPUT.TXT	OUTPUT.TXT
4 ANN ANNNNN BCDEFG HIJKLM NOPQRS	NO	5 HELEN ABCDEF GHIJKL MNOPQL STUVWN EIUOZK	YES 2 1 3 5 4

მითითება. ქ გახლავთ ორნაწილიან გრაფში მაქსიმალური შეწყვილების ამოცანა. თითოეული სიმბოლო შევაერთოთ იმ კუბიკთან, რომელშიც ქ სიმბოლო თუნდაც ერთხელ გვხვდება.

8.2. არჩევნები

(ამიერკავკასიის სტუდენტთა პირადი პირველობა, თბილისი, 2003 წელი)

როგორც ცნობილია, ნებისმიერ არჩევნებში არიან ცნობილი ("გავლენიანი") კანდიდატები, რომლებიც აგროვებენ ბევრ ხმას, მაგრამ არიან ისეთებიც, რომლებიც საერთოდ გაუგებარია, რისთვის მონაწილეობენ არჩევნებში. ჩავთვალოთ, რომ "გავლენიანი" კანდიდატი ყოველთვის იმარჯვებს, თუკი მის ოლქში არ არიან სხვა "გავლენიანი" კანდიდატები.

რა თქმა უნდა, კანდიდატი "გავლენიანია" არა ნებისმიერ ოლქში, არამედ მხოლოდ იმ ოლქში, სადაც მას კარგად იცნობენ. ახლა წარმოვიდგინოთ, რომ ყველა "გავლენიანი" კანდიდატი ჭკუაზე მოეგო და გადაწყვიტა ერთმანეთთან მოლაპარაკება, რათა ერთმანეთს ხელი არ შეუშალონ და ფული ტყუილად არ ხარჯონ. თითოეულმა მოიტანა ოლქების სია, რომელშიც ის საიმედოდ გადის სხვა "გავლენიანი" კონკურენტების არარსებობის შემთხვევაში. მათ ეს სიები ჩაგაბარეს თქვენ და დაგავალეს პროგრამის დაწერა, რომელიც განსაზღვრავს "გავლენიანი" დეპუტატების საუკეთესო განაწილებას ოლქების მიხედვით, ანუ ისეთ განაწილებას, რომლის დროსაც კანდიდატების უმრავლესობა გადის თავის "საიმედო" ოლქებში, ერთმანეთისათვის ხელშეუშლელად.

შესატანი ფაილის ფორმატი:

შესატანი ფაილის პირველ სტრიქონში მოცემულია რიცხვები N (1≤N≤250) და M (1≤M≤250) – "გავლენიანი" კანდიდატებისა და ოლქების რაოდენობები. კანდიდატები და ოლქები დანომრილია მთელი რიცხვებით, შესაბამისად 1-დან N-მდე და 1-დან M-მდე. შემდეგ მოდის N სტრიქონი, რომლებშიც მოცემულია საიმედო ოლქების სიები თითოეული "გავლენიანი" კანდიდატისათვის (სტრიქონი შეიძლება ცარიელიც იყოს, თუკი კანდიდატი სინამდვილეში არ არის "გავლენიანი" და შემთხვევით მოხვდა "ცუდ" კომპანიაში). ოლქების ნომრები სიაში გამოყოფილია შუალედებით.

გამოსატანი ფაილის ფორმატი:

პროგრამის მუშაობის შედეგს წარმოადგენს ფაილი, რომელიც შეიცავს N რიცხვს, თითოეულს ცალკე სტრიქონში. K ნომრის სტრიქონში მითითებული უნდა იყოს ოლქის ნომერი K ნომრის მქონე დეპუტატისათვის, რომელშიც მან უნდა იყაროს კენჭი. თუკი შემოთავაზებულ განაწილებაში მოცემულმა კანდიდატმა საერთოდ არ უნდა მიიღოს მონაწილეობა, სტრიქონში უნდა ეწეროს 0.

დროითი შეზღუდვა : 5 წამი ყოველ ტესტზე.
მაგალითი:

INPUT.TXT	OUTPUT.TXT
5 4 1 3 4 1 3 1 1 3	4 3 0 1 0

მითითება. წინა ამოცანის მსგავსად ეს გახლავთ ორნაწილიან გრაფში მაქსიმალური შეწყვილების ამოცანა. ორნაწილიანი გრაფის ერთი მხარე იქნება კანდიდატთა ნომრები, ხოლო მეორე მხარე – საარჩევნო ოლქების ნომრები.

8.3. დაფა

(საქართველოს მოსწავლეთა ოლიმპიადა, დასკვნითი ტური, 2002-03 წლები)

მოცემულია მართკუთხა დაფა, რომელიც შედგება N×M (N – სტრიქონების რაოდენობა, M – სვეტების რაოდენობა) უჯრისაგან. დაფიდან ზოგიერთი უჯრა ამოჭრილია (რის შედეგადაც იგი შეიძლება ბმული ფიგურა აღარ იყოს). საჭიროა დაფის დარჩენილი ნაწილის ისეთ მართკუთხედ დაფებად დაჭრა, რომლებიც ზუსტად ორ-ორი უჯრისაგან შედგება.

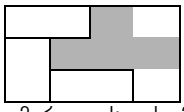
დაწერეთ პროგრამა, რომელიც დაადგენს ასეთი მართკუთხა დაფების მაქსიმალურ რაოდენობას.

შესატანი მონაცემები: შესატან მონაცემთა board.dat ფაილის პირველ სტრიქონში ჩაწერილია ორი მთელი N და M რიცხვი (2≤N,M≤50) – მოცემული დაფის ზომები. მეორე სტრიქონი შეიცავს ერთ მთელ რიცხვს – ამოჭრილ უჯრათა რაოდენობას (თუ არცერთი უჯრა არაა ამოჭრილი, მაშინ ეს რიცხვი 0-ის ტოლია). შემდეგ მოდის იმდენი სტრიქონი, რამდენი უჯრაცაა ამოჭრილი თითოეულ მათგანში მოცემულია ორი მთელი რიცხვი – ერთ-ერთი ამოჭრილი უჯრის კოორდინატები: პირველი მათგანი (რიცხვი 1-დან N-მდე) გვიჩვენებს სტრიქონის ნომერს, ხოლო მეორე (რიცხვი 1-დან M-მდე) – სვეტის ნომერს.

გამოსატანი მონაცემები: გამოსატან მონაცემთა board.rez ფაილის ერთადერთ სტრიქონში უნდა ჩაიწეროს იმ ორუჯრიანი მართკუთხედების მაქსიმალური რაოდენობა, რომელთა ამოჭრაც შეიძლება დაფის დარჩენილი ნაწილიდან.

მაგალითი 1:

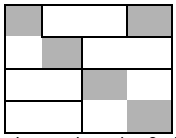
ფაილი board.dat
```
3 4
4
2 4
2 3
2 2
1 3
```
ფაილი board.rez
```
3
```
განმარტება



მართკუთხედის ამოჭრის ერთერთი ვარიანტი

მაგალითი 2:

ფაილი board.dat
```
4 4
5
1 1
3 3
2 2
4 4
1 4
```
ფაილი board.rez
```
4
```
განმარტება



მართკუთხედის ამოჭრის ერთერთი ვარიანტი

მითითება. ამოცანა დაიყვანება მაქსიმალური შეწყვილების პოვნაზე ორნაწილიან გრაფში. ამოცანაში მოცემული დაფა ჭადრაკის დაფის მსგავსად დავყოთ თეთრ და შავ უჯრებად, ანუ ისე, რომ ერთნაირი ფერის უჯრედები მეზობლად არ იყვნენ განლაგებული. თეთრი ფერის უჯრედები ჩავთვალოთ ორნაწილიანი გრაფის ერთ ნაწილად, ხოლო შავი ფერის უჯრედები – მეორე ნაწილად. შესაბამისად, ორივე სიმრავლეს მოვაკლოთ ის უჯრედები, რომლებიც ამოჭრილი უჯრედების კოორდინატებს ემთხვევიან. დარჩენილი უჯრედებიდან ავაგოთ გრაფი: წიბოებით შევაერთოთ ერთმანეთის მეზობლად მდებარე თეთრი და შავი უჯრედები (აქედან გამომდინარე, თითოეული წვერო შეიძლება დაუკავშირდეს მეორე სიმრავლის არაუმეტეს 4 წვერს). ამ სამუშაოების შემდეგ ნებისმიერი მეთოდით ვიპოვოთ მაქსიმალური შეწყვილება (ანუ ნაკადი) ორნაწილიან გრაფში.

9. გამოთვლითი გეომეტრია

9.1. მონაკვეთები

ორი განსხვავებული $p_1=(x_1,y_1)$ და $p_2=(x_2,y_2)$ წერტილების **ამოზნექილი კომბინაცია** (comvex combination) ეწოდება ნებისმიერ $p_3=(x_3,y_3)$ წერტილს, რომლისთვისაც $x_3=\alpha x_1+(1-\alpha)x_2$ და $y_3=\alpha y_1+(1-\alpha)y_2$ რომელიმე $0\le\alpha\le1$-სათვის. ასეთი სახით მოცემული p_3 წერტილი ეკუთვნის p_1 და p_2 წერტილების შემაერთებელ მონაკვეთს (შესაძლოა ემთხვეოდეს მის ერთ-ერთ ბოლოს). ამ თვისებიდან გამომდინარე $\overline{p_1p_2}$ **მონაკვეთი** (line segnent) შეიძლება ვუწოდოთ p_1 და p_2 წერტილების ყველა ამოზნექილი კომბინაციის სიმრავლეს. p_1 და p_2 წერტილებს უწოდებენ მონაკვეთის **ბოლოებს** (endpoints). თუ მნიშვნელოვანია ბოლოების თანმიმდევრობა, მაშინ საუბრობენ $\vec{P_1P_2}$ **ორიენტირებულ მონაკვეთზე** (directed segment). თუკი p_1 ემთხვევა (0,0) წერტილს, ანუ **კოორდინატთა სათავეს** (origin), მაშინ $\vec{P_1P_2}$ ორიენტირებულ მონაკვეთს უწოდებენ p_2 **ვექტორს** (vector).

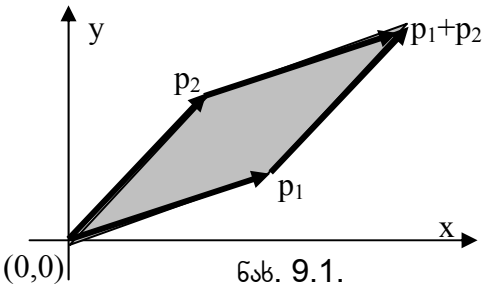
ჩვენ განვიხილავთ შემდეგ საკითხებს:

1. მოცემულია საერთო სათავის ორი ორიენტირებული მონაკვეთი. სათავის მიმართ რომელი მიმართულებით (საათის ისრის მოძრაობის თანხვდენილი თუ საწინააღმდეგო მიმართულებით) უნდა მოვაბრუნოთ ერთ-ერთი მონაკვეთი, რომ ის მეორისაკენ წავიდეს? (იგულისხმება ორი შესაძლო მობრუნებიდან უმცირესი).

2. მოცემულია ტეხილი $p_1p_2p_3$, რომელიც შედგება ორი $\overline{p_1p_2}$ და $\overline{p_2p_3}$ მონაკვეთისაგან. p_1-დან p_3-სკენ მოძრაობისას საით უნდა მოვუხვიოთ p_2 წერტილში — მარჯვნივ თუ მარცხნივ?

3. იკვეთებიან თუ არა $\overline{p_1p_2}$ და $\overline{p_3p_4}$ მონაკვეთები?

ამ საკითხთაგან თითოეული წყდება $O(1)$ დროში. ამოხსნის მეთოდებში გამოყენებული არ იქნება გაყოფის ოპერაცია და ტრიგონომეტრიული ფუნქციები, რადგან ისინი მგრძნობიარენი არიან დამრგვალების ცდომილებისადმი. მაგალითად, მესამე საკითხის გადაწყვეტისას თუკი ვეცდებით მოვძებნოთ მოცემული მონაკვეთების შემცველი წრფეების განტოლებები $y=mx+b$ სახით, ამ განტოლებებით ვიპოვოთ წრფეთა გადაკვეთის წერტილი და შემდეგ შევამოწმოთ, ეკუთვნის თუ არა ეს წერტილი მოცემულ მონაკვეთებს, დაგვჭირდება გაყოფის ოპერაცია და თითქმის პარალელური მონაკვეთების შემთხვევაში მოსალოდნელია გართულებები 0-თან ახლოს მდგომ რიცხვზე გაყოფის გამო.



ნახ. 9.1.

მთავარ საშუალებას ამ ამოცანების გადაწყვეტისას წარმოადგენს ვექტორული ნამრავლის ცნება. ვთქვათ, მოცემულია p_1 და p_2 ვექტორები. ჩვენ გვაინტერესებს მხოლოდ ერთ სიბრტყეში მდებარე ვექტორები, ამიტომ $p_1\times p_2$ **ვექტორული ნამრავლი** (cross product) შეგვიძლია გავიგოთ როგორც შესაბამისი პარალელოგრამის ფართობი ნიშნის გათვალისწინებით. პარალელოგრამის წვეროებია (0,0), p_1, p_2 და $p_1+p_2=(x_1+x_2, y_1+y_2)$. გამოთვლისათვის უფრო მოსახერხებელია ასეთი წარმოდგენა:

$$p_1 \times p_2 = \det\begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1y_2 - x_2y_1 = -p_2 \times p_1$$

თუკი $p_1\times p_2$ დადებითია, მაშინ (0,0)-ის მიმართ p_2-ის მობრუნება p_1-ზე დასამთხვევად ხდება საათის ისრის მოძრაობის მიმართულებით, ხოლო თუ უარყოფითია — საწინააღმდეგო მიმართულებით. ვთქვათ, p_0 წერტილი კოორდინატთა სათავეა, მაშინ მივიღებთ ორ ვექტორს $p_1-p_0=(x_1-x_0,\ y_1-y_0)$ და $p_2-p_0=(x_2-x_0,\ y_2-y_0)$. მათი ვექტორული ნამრავლი იქნება: $(p_1-p_0)\times(p_2-p_0)=(x_1-x_0)(y_2-y_0)-(x_2-x_0)(y_1-y_0)$. თუ ნამრავლი დადებითია, მაშინ $\vec{P_0P_1}$ ვექტორი $\vec{P_0P_2}$-ზე დასამთხვევად უნდა მოვაბრუნოთ საათის ისრის მოძრაობის ანუ "დადებითი" მიმართულებით, ხოლო თუ უარყოფითია — საწინააღმდეგო ანუ "უარყოფითი" მიმართულებით.

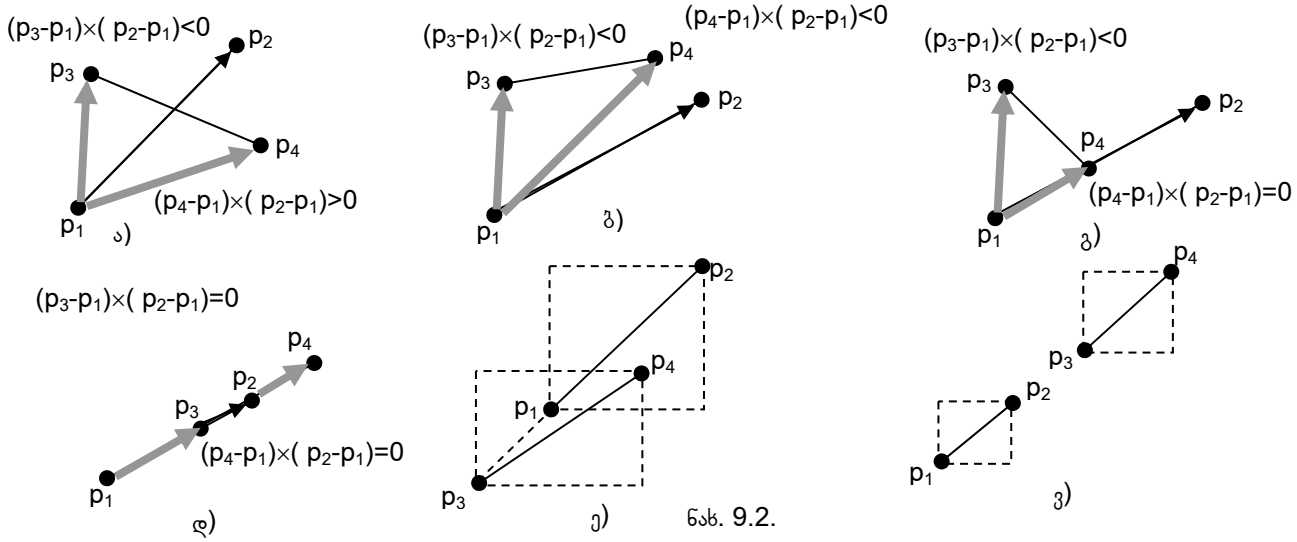
განვიხილოთ მეორე საკითხი – საით უნდა მოვუხვიოთ p_1 წერტილში $\overline{p_0p_1p_2}$ ტეხილზე მოძრაობისას? ამის გასარკვევად საკმარისია გავიგოთ თუ რა მიმართულებით უნდა მობრუნდეს $\vec{P_0P_1}$ ვექტორი, რომ ის $\vec{P_0P_2}$-ის მიმართულებას დაემთხვეს. საამისოდ უნდა გამოვთვალოთ ვექტორული ნამრავლი $(p_2-p_0)\times(p_1-p_0)$. თუ ნამრავლი უარყოფითია, მაშინ $\vec{P_0P_2}$ იმყოფება საათის ისრის საწინააღმდეგოდ $\vec{P_0P_1}$-დან და ე.ი. p_0 წერტილში ჩვენ ვუხვევთ მარცხნივ, ხოლო თუ ნამრავლი დადებითია, მაშინ ვუხვევთ მარჯვნივ. თუ ნამრავლი 0-ის ტოლია, მაშინ ვმოძრაობთ პირდაპირ ან მოვბრუნდებით უკან 180^0-ით – ამ შემთხვევაში p_0,p_1,p_0 წერტილები ერთ წრფეზე მდებარეობენ.

მესამე საკითხს – იკვეთებიან თუ არა მონაკვეთები, განვიხილავთ ორ ეტაპად. **საწყისი ტესტი** (quick rejection) მდგომარეობს შემდეგში: თუკი მონაკვეთების შემომსაზღვრელ მართკუთხედებს არ გააჩნიათ საერთო წერტილები, მაშინ მონაკვეთები არ იკვეთებიან. გეომეტრიული ფიგურის **შემომსაზღვრელ მართკუთხედებს** (bounding box) უწოდებენ იმ მართკუთხედებს შორის უმცირესს, რომელთა გვერდები კოორდინატთა ღერძების პარალელურია და რომლებიც შეიცავენ მოცემულ ფიგურას.

$\overline{p_1p_2}$ მონაკვეთისათვის ასეთი იქნება $(\hat{p}_1,\hat{p}_2)$ მართკუთხედი, ქვედა მარცხენა $\hat{p}_1=(\hat{x}_1,\hat{y}_1)$ და ზედა მარჯვენა $\hat{p}_2=(\hat{x}_2,\hat{y}_2)$ კუთხეებით, სადაც $\hat{x}_1=\min(x_1,x_2)$, $\hat{y}_1=\min(y_1,y_2)$, $\hat{x}_2=\max(x_1,x_2)$, $\hat{y}_2=\max(y_1,y_2)$. $(\hat{p}_1,\hat{p}_2)$ და $(\hat{p}_3,\hat{p}_4)$ მართკუთხედებს გააჩნიათ საერთო წერტილები მაშინ და მხოლოდ მაშინ, როცა $(\hat{x}_2\ge\hat{x}_3)\wedge(\hat{x}_4\ge\hat{x}_1)\wedge(\hat{y}_2\ge\hat{y}_3)\wedge(\hat{y}_4\ge\hat{y}_1)$. აქ პირველი ორი პირობა შეესაბამება x-პროექციის გადაკვეთას, მომდევნო ორი – y-პროექციის გადაკვეთას.

თუკი პირველ ეტაპზე არ ხერხდება იმის დადგენა, რომ მონაკვეთები არ იკვეთებიან, გადავდივართ მეორე ეტაპზე და ვამოწმებთ იკვეთება თუ არა თითოეული მონაკვეთი მეორე მონაკვეთის შემცველ წრფესთან. მონაკვეთი ჰკვეთს წრფეს (straddles a line), თუკი მისი ბოლოები მდებარეობენ წრფის სხვადასხვა მხარეს ან თუ ერთ-ერთი ბოლო მდებარეობს წრფეზე. ამ პირობის შემოწმება ხდება ვექტორული ნამრავლის საშუალებით. p_3 და p_4 წერტილები მდებარეობენ p_1p_2 წრფის სხვადასხვა მხარეს, თუ $\vec{P_1P_3}$ და $\vec{P_1P_4}$ ვექტორებს აქვთ სხვადასხვა ორიენტაცია $\vec{P_1P_2}$ ვექტორის მიმართ (ნახ. 9.2ა-ბ), ე.ი. თუ $(p_3-p_1)\times(p_2-p_1)$ და $(p_4-p_1)\times(p_2-p_1)$ ვექტორულ ნამრავლებს განსხვავებული ნიშანი აქვთ. თუკი ამ ვექტორულ ნამრავლთაგან ერთ-ერთი 0-ის ტოლია, მაშინ p_3 და p_4 წერტილებისაგან ერთ-ერთი ეკუთვნის p_1p_2 წრფეს. ორი ასეთი შემთხვევა ნაჩვენებია 9.2(გ-დ) ნახაზებზე. ორივე შემთხვევაში მონაკვეთები იკვეთებიან, თუმცა შესაძლებელია სხვა შემთხვევაც: შესაძლოა მონაკვეთები გადიოდნენ საწყის ტესტს და ვექტორული ნამრავლი 0-ის ტოლია, მაგრამ გადაკვეთა მაინც არ არსებობდეს (ნახ. 9.2(ე)), ამიტომ პირობა "მონაკვეთი ჰკვეთს წრფეს" უნდა შემოწმდეს ორივე მონაკვეთისათვის. შევნიშნოთ, რომ საწყისი ტესტის გამოტოვება არ შეიძლება. ნახ. 9.2(ვ)-ზე გამოსახულია შემთხვევა, როცა თითოეული მონაკვეთი ჰკვეთს მეორის შემცველ წრფეს (მდებარეობს მასზე – რაც ითვლება გადაკვეთად), მაგრამ თავად მონაკვეთები არ იკვეთებიან.

მაშასადამე, $\overline{p_1p_2}$ და $\overline{p_3p_4}$ მონაკვეთები იკვეთებიან მაშინ და მხოლოდ მაშინ, როცა ერთდროულად სრულდება შემდეგი სამი პირობა: ა) იკვეთებიან მათი შემომსაზღვრელი მართკუთხედები, ბ) $[(p_3-p_1)\times(p_2-p_1)]\cdot[(p_4-p_1)\times(p_2-p_1)]\le 0$, გ)) $[(p_1-p_3)\times(p_4-p_3)]\cdot[(p_2-p_3)\times(p_4-p_3)]\le 0$.

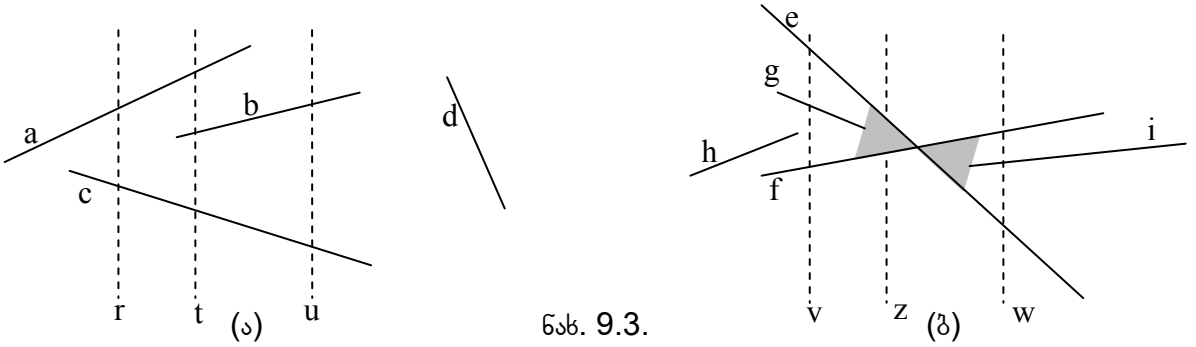


ნახ. 9.2.

განვიხილოთ ასეთი ამოცანა: ვთქვათ, მოცემულია n მონაკვეთი, საჭიროა გავარკვიოთ იკვეთება თუ არა რომელიმე ორი მათგანი. პრობლემის გადასაწყვეტად გამოიყენება "მოძრავი წრფის" მეთოდი, რომელიც ხშირად იხმარება გამოთვლით გეომეტრიაში. ალგორითმი მუშაობს O(n*log*n) დროში, სადაც n – მონაკვეთთა რაოდენობაა. ამასთან, მოწმდება მხოლოდ გადაკვეთის არსებობა, ხოლო თავად გადაკვეთა არ იძებნება.

მოძრავი წრფის მეთოდი (sweeping line method) მდგომარეობს იმაში, რომ წარმოსახვითი ვერტიკალური წრფე მოძრაობს მარცხნიდან მარჯვნივ განსახილველი გეომეტრიული ობიექტების გასწვრივ. მონაკვეთთა გადაკვეთის ამოცანაში ალგორითმი განიხილავს ყველა მონაკვეთის ბოლოებს მარცხნიდან მარჯვნივ და ყოველი გადანაცვლებისას ამოწმებს, ხომ არ მოხდა მონაკვეთთა გადაკვეთა ამ უბანზე. ალგორითმი უშვებს ორ გამამარტივებელ გარემოებას — განსახილველ მონაკვეთთა შორის არ არიან ვერტიკალურები და მონაკვეთთა არც ერთი სამეული არ გადის ერთ და იმავე წერტილში. თუმცა ამ სასაზღვრო პირობების გათვალისწინებაც არ მოითხოვს ალგორითმის რთულ მოდიფიცირებას.

რადგან განსახილველ მონაკვეთთა შორის არ იქნებიან ვერტიკალურები, ამიტომ მოძრავი ვერტიკალური წრფე გადაკვეთს თითოეულ მათგანს მაქსიმუმ ერთ წერტილში. ჩვენ ვალაგებთ გადაკვეთილ მონაკვეთებს გადაკვეთის წერტილთა ორდინატების მიხედვით. უფრო ზუსტად ორი s_1 და s_2 მონაკვეთი **შედარებადია x-ის მიმართ** (comparable at x), თუ x აბსცისის მქონე ვერტიკალური წრფე ჰკვეთს ორივე მონაკვეთს. ამასთან s_1 **მაღლაა** s_2-ზე x-ის მიმართ (s_1 is above s_2 at x, აღნიშვნა — $s_1 >_x s_2$), თუ s_1 და s_2 მონაკვეთები შედარებადია x-ის მიმართ და s_1-ის გადაკვეთის წერტილი ვერტიკალურ წრფესთან იმყოფება უფრო მაღლა, ვიდრე s_2-ის გადაკვეთის წერტილი ამავე წრფესთან. ნახ. 9.3(ა)-ზე $a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$ და $b >_u c$, ხოლო d მონაკვეთია არაა შედარებადი არცერთ სხვა მონაკვეთთან.



ნახ. 9.3.

ნებისმიერი ფიქსირებული x-სათვის თანაფარდობა "$>_x$" წარმოადგენს x-ზე გამავალი ვერტიკალური წრფის გადამკვეთ მონაკვეთთა სიმრავლის დალაგებას. სხვადასვა x-სათვის ეს დალაგება შესაძლოა სხვადასხა იყოს. მონაკვეთი ხვდება ამ სიმრავლეში, როცა ვერტიკალური წრფე გაივლის მის მარცხენა ბოლოზე, ხოლო ტოვებს სიმრავლეს, როცა გაივლის მის მარჯვენა ბოლოზე. როცა ვერტიკალური წრფე გაივლის ორი მონაკვეთის გადაკვეთის წერტილზე, ამ მონაკვეთთა დალაგების მიმდევრობა იცვლება საწინააღმდეგოთი. ნახ. 9.3(ბ)-ზე v და w წრფეები განლაგებულნი არიან e და f წრფეების გადაკვეთის წერტილის მარჯვნივ და მარცხნივ. ამასთან, $e >_v f$ და $f >_w e$. ჩვენი პირობის მიხედვით არცერთი სამი მონაკვეთი არ გადის ერთ წერტილში, ამიტომ გადამკვეთი მონაკვეთები არსებულ დალაგებაში უშუალოდ ერთმანეთის შემდეგ იქნებიან განლაგებული.

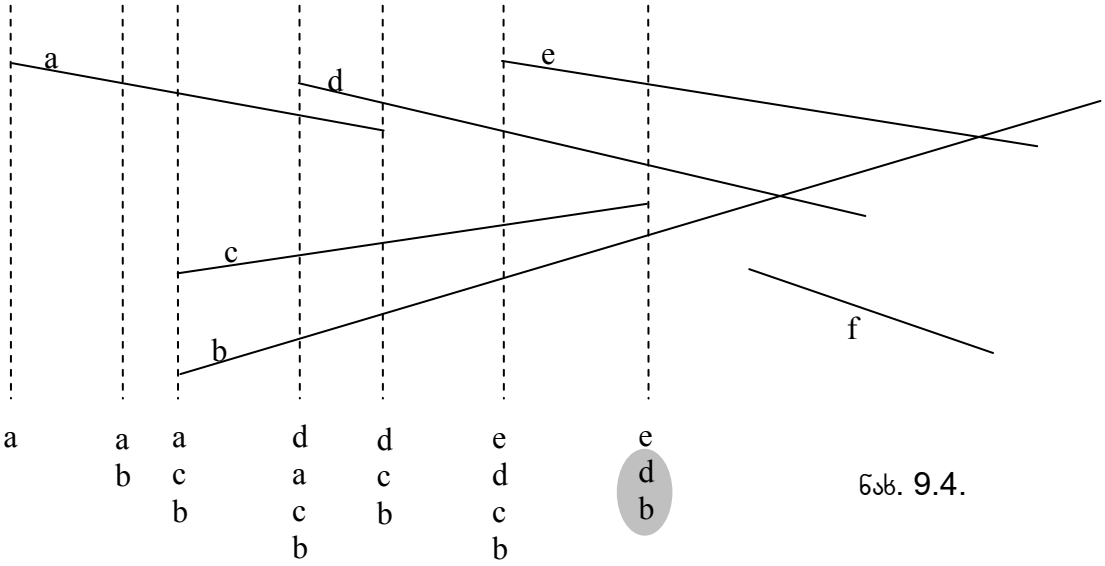
მოძრავი წრფის მეთოდის გამოყენებისას ჩვენ ვინახავთ შემდეგ ინფორმაციას: 1) **წრფის მდგომარეობა** (Sweep-line status), რომელიც მოცემულია იმ ობიექტთა დალაგებული სიმრავლით, რომელსაც ჰკვეთს მოძრავი წრფე მოცემულ მომენტში; 2) **განრიგი** (event-point shedule) — წარმოადგენს ზრდადობით დალაგებულ იმ წერტილთა მიმდევრობას, რომლებშიც წრფის მდგომარეობა შეიძლება შეიცვალოს. სხვაგვარად მათ **კრიტიკულ წერტილებს** (event-point) უწოდებენ.

ზოგიერთი ალგორითმისათვის კრიტიკული წერტილები შესაძლოა განისაზღვროს თანდათანობით ალგორითმის მუშაობის პროცესში, თუმცა ჩვენ განვიხილავთ ალგორითმს, რომლისთვისაც განრიგი წინასწარაა ცნობილი. კერძოდ, ნებისმიერი მონაკვეთის ბოლოს აბსცისა წარმოადგენს კრიტიკულ წერტილს. დავალაგოთ მონაკვეთთა ბოლოები მათი აბსცისების ზრდადობის მიხედვით. მონაკვეთი იწყებს ზემოქმედებას წრფის მდგომარეობაზე, როცა წრფე გაივლის მის მარცხენა ბოლოზე, ხოლო წყვეტს ზემოქმედებას, როცა გაივლის მის მარჯვენა ბოლოზე.

წრფის მდგომარეობა შეიძლება შევინახოთ როგორც მონაკვეთთა დალაგებული T სიმრავლე, რომლზეც სრულდება შემდეგი ოპერაციები: Insert(T,s) — დავუმატოთ s მონაკვეთი T-ს; Delete(T,s) — წავშალოთ s მონაკვეთი T-დან; Above(T,s) — მივუთითოთ მონაკვეთი, რომელიც განლაგებულია უშუალოდ s-ზე მაღლა T სიმრავლეში; Below(T,s) — მივუთითოთ მონაკვეთი, რომელიც განლაგებულია უშუალოდ s-ზე დაბლა T სიმრავლეში.

ავაგოთ ალგორითმი, რომელიც მოცემული n მონაკვეთისაგან შემდგარ S სიმრავლისათვის ამოწმებს არის თუ არა სიმრავლეში ორი მაინც გადამკვეთი წრფე.

```
ANY-SEGMENTS-INTERSECT(S)
1 T=∅
2 დავალაგოთ მონაკვეთთა ბოლოები აბსცისების ზრდადობის მიხედვით (ტოლი აბსცისები ლაგდება
ორდინატთა ზრდადობით). ვამოწმებთ და თუ რომელიმე ორი წერტილი ემთხვევა, ვაბრუნებთ TRUE-ს.
3 for ყოველი p წერტილისათვის მიღებული სიიდან  {
4     if (p – რომელიღაც s მონაკვეთის მარცხენა ბოლოა)
5          then {  INSERT(T,s)
6                  if (ABOVE(T,s) არსებობს და ჰკვეთს s-ს) ან (BELOW(T,s) არსებობს და ჰკვეთს s-ს)
7                  then {  return TRUE  }  }
8       if (p –  რომელიღაც s მონაკვეთის მარცხენა ბოლოა)
9          then {  if განსაზღვრულია ABOVE(T,s) და BELOW(T,s) და ისინი ჰკვეთენ ერთმანეთს
10                    then { return TRUE  }
11                 DELETE(T,s)    }  }
12 return FALSE
```



ნახ. 9.4.

ნახ. 9.4-ზე ნაჩვენებია ალგორითმის შესრულება. თავიდან T სიმრავლე ცარიელია (1-ლი სტრიქონი). მე-2 სტრიქონში მონაკვეთთა ბოლოების შესაბამისად აიგება განრიგი. მართალია კრიტიკულ წერტილებად ითვლებიან მონაკვეთთა გადაკვეთის წერტილებიც, მაგრამ პირველივე ასეთი წერტილის პოვნისას ალგორითმი ამთავრებს მუშაობას, ამიტომ ეს მომენტი შეგვიძლია უგულვებელვყოთ.

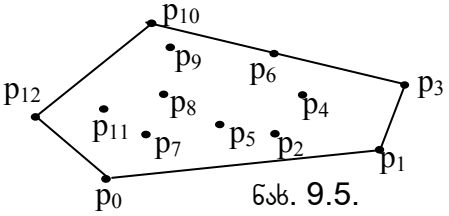
for ციკლის ყოველი იტერაცია 3-11 სტრიქონებში დაამუშავებს მხოლოდ ერთ კრიტიკულ წერტილს.. თუკი ეს წერტილი შეესაბამება რომელიღაც s მონაკვეთის მარცხენა ბოლოს. მაშინ მე-5 სტრიქონში ეს მონაკვეთი ემატება დალაგებულ სიმრავლეს, ხოლო 6-7 სტრიქონებში მოწმდება ხომ არ იკვეთება იგი რომელიმე მეზობელ მონაკვეთთან (თუ იკვეთება, მაშინ ალგორითმი ამთავრებს მუშაობას და გამოქვს – TRUE). შესაძლოა, კრიტიკული წერტილი შეესაბამებოდეს ერთდროულად რამდენიმე მონაკვეთის ბოლოებს, მაშინ ისინი ემტებიან ერთანეთის მიყოლებით.

თუკი დასამუშავებელი კრიტიკული წერტილი რომელიღაც s მონაკვეთის მარჯვენა ბოლოს შეესაბამება, მაშინ ის ამოიშლება T-დან (მე-11 სტრიქონი). წინასწარ მოწმდება, ხომ არ იკვეთებიან მონაკვეთები, რომელსაც ჰყოფდა წასაშლელი მონაკვეთი.

ასე გრძელდება ყველა მონაკვეთის ყველა ბოლოსათვის და თუ გადაკვეთა მაინც არ მოინახა, ალგორითმი აბრუნებს FALSE (მე-12 სტრიქონი). n ცალი მონაკვეთისათვის ალგორითმის მუშაობის დროა O(n *log*n).

9.2. ამოზნექილი გარსის აგება

წერტილთა Q სასრული სიმრავლის **ამოზნექილი გარსი** (convex hull) ეწოდება უმცირეს ამოზნექილ მრავალკუთხედს, რომელიც შეიცავს ყველა წერტილს Q-დან (წერტილთაგან ზოგიერთი იქნება მრავალკუთხედის შიგნით, ზოგიერთი — მის გვერდებზე, ხოლო ზოგიერთი ამ მრავალკუთხედის წვერო იქნება).



ნახ. 9.5.

Q სიმრავლის ამოზნექილი გარსი აღინიშნება CH(Q). თვალსაჩინოებისათვის შეიძლება ასე წარმოვიდგინოთ: Q-ს წერტილებში ჩაჭედებულია ლურსმნები, რომლებზეც შემორტყმულია რეზინა იმგვარად, რომ მოიცავს ყველა ლურსმანს. სწორედ ეს რეზინა იქნება ლურსმანთა სიმრავლის ამოზნექილი გარსი (იხ. ნახ. 9.5.).

ჩვენ განვიხილავთ n წერტილისათვის ამოზნექილი გარსის პოვნის ორ ალგორითმს: გრეჰემის ალგორითმს, რომელიც სრულდება O(n *log*n) დროში და ჯარვისის ალგორითმს, რომლის მუშაობის დროა O(nh), სადაც h ამოზნექილი გარსის წვეროების რაოდენობაა. ორივე ალგორითმი იყენებს **"მბრუნავი სხივის"** (rotational sweep) მეთოდს — პოლარულ კოორდინატთა სისტემაში წერტილები დამუშავდება პოლარულ კუთხეთა ზრდის მიხედვით. არსებობენ ამოზნექილი გარსის პოვნის სხვა მეთოდებიც, რომელიც მუშაობენ O(n *log*n) დროში: **წერტილების დამატების** მეთოდი (incremental method) განიხილავს წერტილებს მარცხნიდან მარჯვნივ აბსცისათა ზრდადობის მიხედვით. i-ურ ბიჯზე იგება ამოზნექილი გარსი $CH(p_1,p_2,...,p_i)$ პირველი i წერტილისათვის — პირველი i-1 წერტილისათვის უკვე ცნობილ ამოზნექილ გარსს ემატება p_i წერტილი.

მეთოდი **"დაყავი და იბატონე"** (divide-and-conquer method) $\theta(n)$ დროში ჰყოფს n წერტილისაგან შემდგარ სიმრავლეს ორ დაახლოებით ტოლ ქვესიმრავლედ, შემდეგ რეკურსიულად ეძებს ამოზნექილ გარსს თითოეული მათგანისათვის და ბოლოს აერთიანებს ორ გარსს O(n) დროში.

გამოხშირვისა და ძებნის მეთოდი (prune-and-search method) მედიანის ძებნის ალგორითმს დალაგებულ სიმრავლეში. ის პოულობს ამოზნექილი გარსის ზედა ჯაჭვს წერტილთა რაღაც ფიქსირებული ნაწილის უკუგდებით მანამ, სანამ არ დარჩება მხოლოდ ზედა ჯაჭვის წერტილები. ანალოგიურად მოიძებნება ქვედა ჯაჭვიც. ცნობილ მეთოდთაგან ეს მეთოდი ყველაზე სწრაფია: თუ ამოზნექილ გარსს აქვს h წვერო, ალგორითმის მუშაობის დროა O(n *log*h).

ამოზნექილი გარსის აგება გამოთვლით გეომეტრიაში ხშირად გამოიყენება როგორც საშუალედო საფეხური სხვადასხვა ამოცანების ამოხსნისას. მაგალითად, ორგანზომილებიანი ამოცანა უშორეს წერტილებზე (farthest-pair problem). მტკიცდება, რომ სიბრტყეზე მოცემული n წერტილისათვის ერთმანეთისაგან მაქსიმალურად დაშორებული ორი წერტილი ამოზნექილი გარსის წვეროებია.

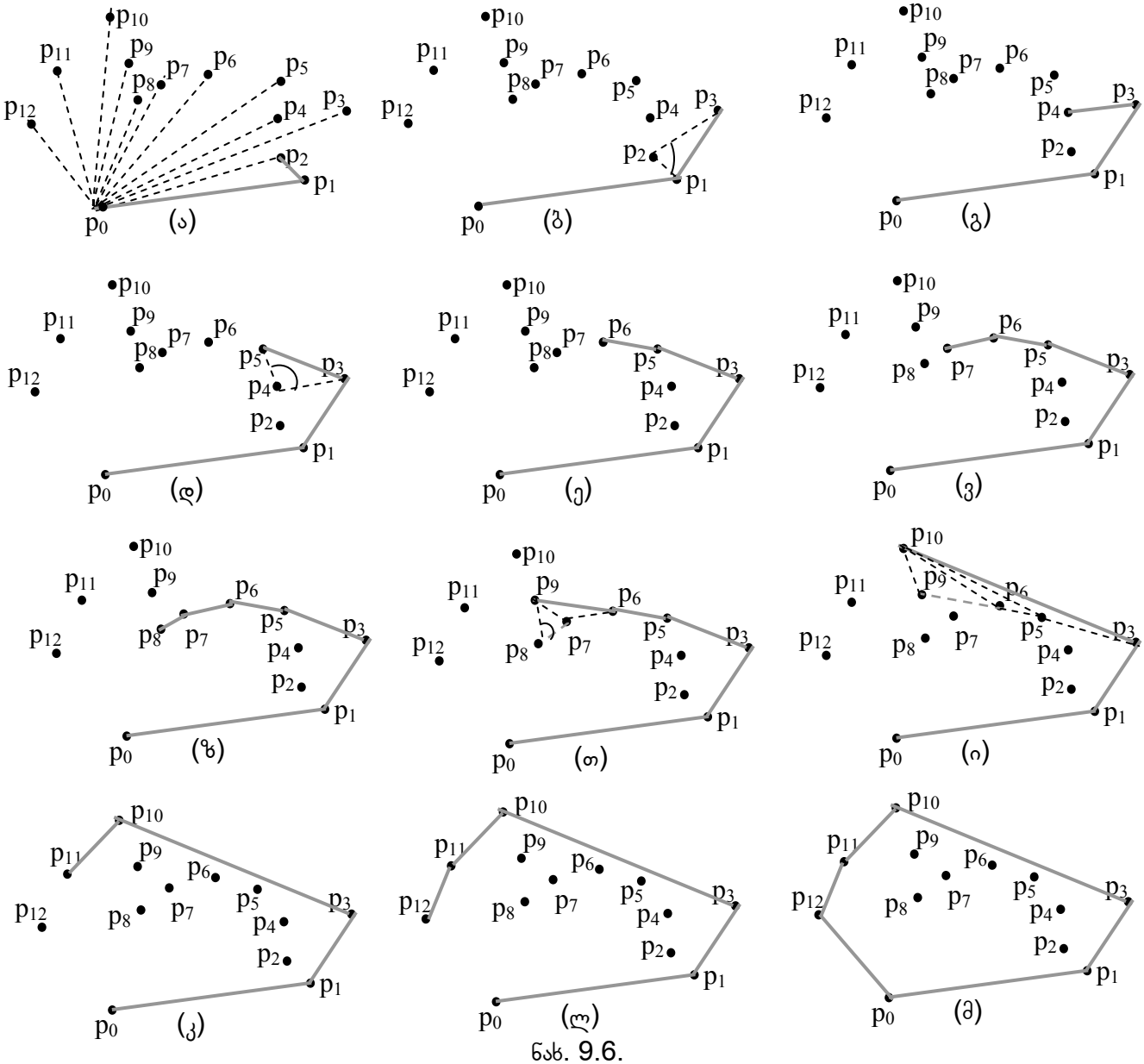
გრეჰემის ალგორითმი. გრეჰემის ალგორითმი იყენებს S სტეკს, რომელშიც ინახებიან ამოზნექილი გარსის კანდიდატი წერტილები. Q სიმრავლის ყოველი წერტილი რაღაც მომენტის განმავლობაში ინახება სტეკში. თუ იგი არ წარმოადგენს CH(Q) ამოზნექილი გარსის წვეროს, მაშინ დატოვებს სტეკს, ხოლო თუ წარმოადგენს — დარჩება. ალგორითმის მუშაობის დამთავრებისას S სტეკში იმყოფება CH(Q) ამოზნექილი გარსის ყველა წვერო, რომლებიც დალაგებულია საათის ისრის საწინააღმდეგო მიმართულებით შემოვლის რიგით.

GRAHAM-SCAN-ის საწყისი მონაცემებია Q სიმრავლე, რომელიც შედგება არანაკლებ სამი წერტილისაგან. პროცედურა იყენებს TOP(S) ფუნქციას, რომელიც აბრუნებს S სტეკის სათავეში მყოფ წერტილს სტეკის შიგთავსის შეუცვლელად NEXT-TO-TOP(S) ფუნქციას, რომელიც აბრუნებს სტეკის სათავის მომდევნო ელემენტს (ასევე სტეკის შიგთავსის შეუცვლელად).

```
GRAHAM-SCAN(Q)
1 ვთქვათ, p0 არის Q სიმრავლის წერტილი უმცირესი ორდინატით (თუ ასეთი რამდენიმეა — ყველაზე მარცხენა)
2 ვთქვათ, ⟨p1,p2,...,pm⟩ Q სიმრავლის დანარჩენი წერტილებია, რომლებიც დალაგებულნი არიან პოლარული
  კუთხის ზრდადობის მიხედვით p0-ის მიმართ (საათის ისრის საწინააღმდეგო მიმართულებით). თუკი რამდენიმე
  წერტილს ერთნაირი პოლარული კუთხე აქვს, ვტოვებთ p0-დან ყველაზე მეტად დაშორებულს.
3 top[S]=0
4 PUSH(S, p0)
5 PUSH(S, p1)
6 PUSH(S, p2)
7 for i=3 to m  {
8       while NEXT-TO-TOP(S)→ TOP(S)→pi ტეხილზე მოძრაობისას მივდივართ წინ ან მარჯვნივ {
9                   POP(S)  }
10            PUSH(S, pi)     }
11 return S
```

ნახ. 9.6-ზე მოცემულია ალგორითმის მუშაობა ბიჯების მიხედვით.

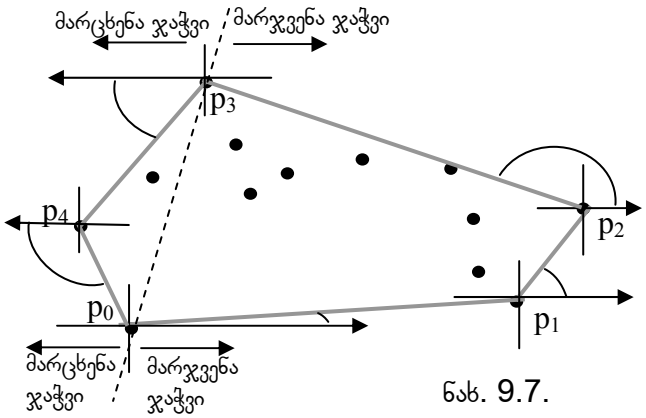
ალგორითმის 1-ელ სტრიქონში ვპოულობთ p_0 წერტილს. Q სიმრავლის ყველა სხვა წერტილის იმყოფება მის ზემოთ (ან იგივე დონეზე, ოღონდ მარჯვნივ). p_0 წერტილი აუცილებლად შედის CH(Q)-ში. მე-2 სტრიქონში Q-ს დანარჩენი წერტილები ლაგდებიან პოლარული კუთხეების ზრდადობით p_0 წერტილის მიმართ. ყოველი შედარება შეიძლება შესრულდეს O(1) დროში. ერთნაირი კუთხეების შემთხვევაში ვტოვებთ p_0-დან ყველაზე დაშორებულს (დანარჩენები არ მიეკუთვნებიან ამოზნექილ გარსს). Q სიმრავლის წერტილთა პოლარული კუთხეები მოთავსებული არიან $[0,\pi)$ შუალედის საზღვრებში. შევნიშნოთ, რომ p_1 და p_m წერტილები წარმოადგენენ CH(Q)-ს წვეროებს. 3-6 სტრიქონებით ჩვენ სტეკში ვათავსებთ სამ პირველ წერტილს — p_0, p_1 და p_2, რის შემდეგაც შეგიძლია ვთქვათ, რომ სტეკი შეიცავს ამოზნექილ გარსს $p_1,p_2,...,p_k$, k=2-სათვის. ეს თვისება შენარჩუნებული იქნება შემდგომშიც, ხოლო k გაიზრდება m-მდე. სტეკის ფსკერიდან მისი სათავისაკენ მოძრაობისას ჩვენ ყოველთვის მარცხნივ ვუხვევთ. სტეკში ახალი წერტილის დამატებისას ეს თვისება შეიძლება დაირღვეს, ამიტომ ჩვენ ვშლით სტეკიდან რამდენიმე ზედა წვეროს.



ნახ. 9.6.

<u>ჯარვისის ალგორითმი</u>. ჯარვისის ალგორითმი აგებს ამოზნექილ გარსს "შემოხვევის" (package wrapping, gift wrapping) საშუალებით. მისი მუშაობის დროა O(nh), სადაც h ამოზნექილი გარსის წვეროთა რაოდენობაა, ამიტომ ამოცანებში სადაც h არის o(*log*n), ჯარვისის მეთოდი ასიმპტოტურად სწრაფია გრეჰემის ალგორითმზე.

ალგორითმის იდეა ასეთია: თოკის ერთი ბოლო მივამაგროთ ყველაზე ქვედა p_0 ლურსმანს, რომლის ამორჩევა ხდება ზუსტად ისევე, როგორც გრეჰემის ალგორითმში. გავჭიმოთ თოკი ჰორიზონტალურად მარჯვნივ და ავწიოთ მისი მარჯვენა ბოლო მანამ, სანამ თოკი არ შეეხება რომელიღაც p_1 ლურსმანს. შემდეგ თოკს ვაბრუნებთ უკვე p_1-ის მიმართ, ვიდრე ის არ შეეხება p_2 ლურსმანს და ასე ვაგრძელებთ საწყის p_0 წერტილში დაბრუნებამდე.



ნახ. 9.7.

სხვაგვარად რომ ვთქვათ: ვიწყებთ p_0 წერტილიდან. მომდევნო p_1 წერტილს აქვს უმცირესი პოლარული კუთხე p_0-ის მიმართ Q სიმრავლის წერტილთა შორის (თუკი ასეთი რამდენიმეა, ვირჩევთ უშორესს). შემდეგ ვდგებით p_1 წერტილში და ვპოულობთ უმცირესი პოლარული კუთხის მქონე p_2 წერტილს p_1-ის მიმართ და ა.შ. რაღაც მომენტში მივაღწევთ ყველაზე ზედა წერტილს (უდიდესი ორდინატის მქონეს). ამ მომენტიდან პოლარული კუთხეები უნდა გამოვთვალოთ არა მარჯვნივ, არამედ მარცხნივ მიმართული სხივისადმი. ამგვარად ჯერ ვაგებთ **მარჯვენა ჯაჭვს** (right chain), ხოლო შემდეგ — **მარცხენა ჯაჭვს** (left chain). იხ ნახ. 9.7.

შესაძლებელია თავი ავარიდოთ მარჯვენა და მარცხენა ჯაჭვების გამოყოფას. ამისათვის საჭიროა შევინახოთ უკანასკნელად მოძებნილი გვერდის მიმართულება და ავარჩიოთ სხივის უახლოესი წერტილი დადებითი მიმართულებით. ამ შემთხვევაში უფრო რთულია კუთხეების შედარება. მარჯვენა და მარცხენა ჯაჭვების გამოყენებისას კი შეგვიძლია კუთხეების შევადაროთ 9.1. თავში აღწერილი ხერხით, როცა მათი ცხადად გამოთვლა საჭირო არ არის.

9.3. წერტილთა უახლოესი წყვილის მოძებნა

ვთქვათ, საჭიროა Q სიმრავლის $n\geq 2$ წერტილთა შორის ორი ერთმანეთთან ყველაზე ახლოს მდგომი წერტილის მოძებნა. წერტილათა შორის მანძილი განისაზღვრება ჩვეულებისამებრ: $p_1=(x_1,y_1)$ და $p_2=(x_2,y_2)$ წერტილებს შორის მანძილია $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$. კერძო შემთხვევაში ორი წერტილი შესაძლოა ერთმანეთს ემთხვეოდეს, მაშინ მათ შორის მანძილის 0-ის ტოლია.

თუკი წერტილთა წყვილებს სრულად გადავარჩევთ, მოგვიწევს $C_n^2=\theta(n^2)$ წყვილის განხილვა. ჩვენ განვიხილავთ ალგორითმს, რომელიც "დაყავი და იბატონე"-ს პრინციპით რეკურსიულად წყვეტს ამ ამოცანას $T(n)=2T(n/2)+O(n)$, ანუ $O(n\ log n)$ დროში.

თითოეული რეკურსიული გამოძახების შემავალი მონაცემები შედგება $P\subseteq Q$ ქვესიმრავლისა და ორი X და Y მასივისაგან. ეს მასივები შეიცავენ P ქვესიმრავლის წერტილებს, ოღონდ სხვადასხვანაირი დალაგებით: X მასივში წერტილები განლაგებულია აბსცისების ზრდის მიხედვით, ხოლო Y მასივში — ორდინატების ზრდის მიხედვით. შევნიშნოთ, რომ ყოველი რეკურსიული გამოძახებისას წერტილთა სორტირება მკვეთრად გაზრდიდა მუშაობის დროს $T(n)=2T(n/2)+O(n\ log n)$, ანუ $T(n)=O(n\ ln^2 n)$-მდე. ამ სირთულის თავიდან აცილება შეიძლება ე.წ. "წინასორტირებით", რომელიც მხოლოდ ერთხელ სრულდება რეკურსიული გამოძახებების დაწყებამდე და შემდეგ ხდება მხოლოდ მიღებული მასივის გაყოფა ორ შესაბამის ნაწილად.

მაშასადამე, ჩვენ მივიღეთ P, X და Y. თუ $|P|\leq 3$, გადავარჩევთ წერტილთა ყველა წყვილს და შევადარებთ მანძილებს, ხოლო თუ $|P|>3$, მაშინ ვიქცევით შემდეგნაირად:

ვპოულობთ ვერტიკალურ l წრფეს, რომელიც ჰყოფს P სიმრავლეს შუაზე P_L და P_R ქვესიმრავლეებად ($|P_L|=\lceil |P|/2\rceil$, $|P_R|=\lfloor |P|/2\rfloor$, თავად l წრფეზე მდებარე წერტილებიც იყოფა P_L-სა და P_R-ს შორის). X მასივი იყოფა X_L და X_R მასივებად, რომლებიც შეიცავენ P_L და P_R ქვესიმრავლეების წერტილებს დალაგების შენარჩუნებით. ანალოგიურად იყოფა Y მასივიც Y_L და Y_R მასივებად.

P სიმრავლის P_L და P_R ქვესიმრავლეებად გაყოფის შემდეგ ვასრულებთ ორ რეკურსიულ გამოძახებას ვპოულობთ უახლოესი წერტილების წყვილს P_L-ში (შემავალი მონაცემები P_L, X_L და X_R) და ასეთივე წყვილს P_R-ში (შემავალი მონაცემები P_R, Y_L და Y_R). აღვნიშნოთ მანძილები ნაპოვნ წყვილებს შორის შესაბამისად δ_L-ით და δ_R-ით. ვთქვათ $\delta=\min(\delta_L,\delta_R)$.

P სიმრავლისათვის უახლოესი წერტილების წყვილს წარმოადგენს ან ერთ-ერთი მოძებნილი წყვილი (δ მანძილზე დაშორებით) ან რომელიმე "საზღვრისპირა" წერტილთა წყვილი, რომელშიდაც ერთი წერტილი ეკუთვნის P_L-ს, ხოლო მეორე — P_R-ს (თუკი მათ შორის მანძილი ნაკლებია δ-ზე). ცხადია, რომ ასეთი საზღვრისპირა წყვილი წერტილები არ იქნებიან დაშორებულნი l წრფისაგან δ მანძილზე მეტად, ე.ი. იმყოფებიან 2δ სიგანის სასაზღვრო ზოლში l ვერტიკალური წრფიდან.ასეთი საზღვრისპირა წყვილის მოსაძებნად, თუკი ის არსებობს, საჭიროა გავაკეთოთ შემდეგი:

1) შევქმნათ Y′ მასივი, რომელშიც შევა ყველა ის წერტილი Y მასივიდან, რომელიც ხვდება სასაზღვრო ზოლში l წრფის ორივე მხრიდან. თანმიმდევრობა შენარჩუნებული უნდა იყოს: Y′ მასივი დალაგებულია წერტილთა ორდინატების ზრდადობით; 2) Y′ მასივის ყოველი p წერტილისათვის ვეძებთ Y′ მასივის ისეთ წერტილს, რომლის დაშორება p-დან არ აღემატება δ-ს. მტკიცდება, რომ საკმარისია განვიხილოთ p-ს მომდევნო მხოლოდ 7 წერტილი ორდინატების ზრდის მიხედვით. თითოეული მათგანიდან გამოითვლება მანძილები p წერტილამდე და იგივე გაკეთდება Y′ მასივის ყველა წერტილისათვის. ამგვარად ვიპოვით δ' — Y′ მასივის უახლოეს წერტილებს შორის მანძილს; 3) თუ $\delta'<\delta$, მაშინ უახლოეს წერტილთა წყვილი მთელი სიმრავლისათვის იმყოფება ვერტიკალური ზოლის შიგნით და ჩვენ ვაბრუნებთ ამ წყვილს და მათ შორის δ' მანძილს. წინააღმდეგ შემთხვევაში ვაბრუნებთ ერთ-ერთი რეკურსიული გამოძახებისას მიღებულ წყვილს და მანძილს δ.

9.1 კორსარი

(უკრაინის online-ოლიმპიადა, 2002-03 წლები)

საზღვაო მეკობრის მემკვიდრეებმა, თავიანთი წინაპრის – კორსარის სახლის რემონტის დროს იპოვეს დაუსახლებელი კუნძულის რუკა, რომელზეც რიცხვთა წყვილებით მითითებულია განძთა დეკარტული კოორდინატები. მემკვიდრეები სასწრაფოდ გაემგზავრნენ კუნძულზე და შემოიარეს ყველა ადგილი, სადაც კი განძი იყო დამალული. ისინი მოძრაობდნენ წრფივად – განძიდან განძისაკენ, რუკაზე მითითებული განძთა ნომრების მიხედვით. უკანასკნელი განძის პოვნის შემდეგ ისინი დაბრუნდნენ იმ ადგილას, სადაც პირველი განძი იპოვეს, ამასთან მათი განვლილი გზები არსად გადაკვეთილან. მემკიდრეებმა განძი მშვიდობიანად გაიყვეს, ოღონდ ვერ გაიხსენეს თუ როგორ მოძრაობდნენ – საათის ისრის თანხვედნილი თუ საწინააღმდეგო მიმართულებით. დაეხმარეთ მათ.

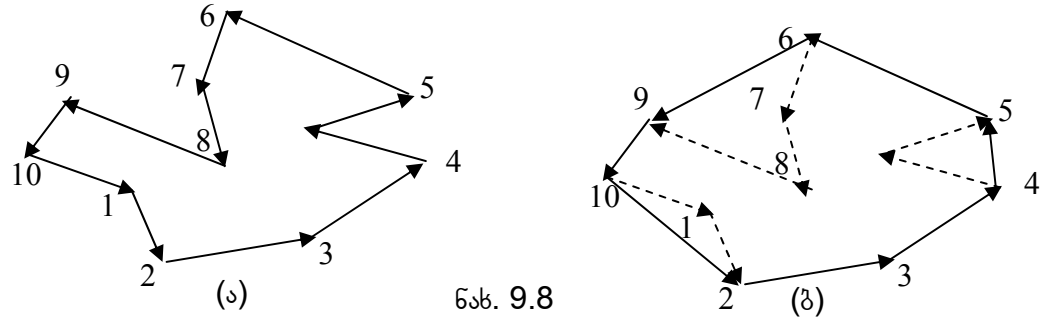
შემავალი მონაცემები: ჯერ შემოდის საკონტროლო მონაცემების რაოდენობა K (1≤K≤100), შემდეგ K-ჯერ შემოდის განძების რაოდენობა N (3≤N≤20) და ნამდვილი რიცხვების N წყვილი – მორიგი განძის კოორდინატები (პირველი რიცხვი აბსცისაა, მეორე – ორდინატი).

გამომავალი მონაცემები: თქვენ გამოგაქვთ ეკრანზე 0-ებისა და 1-ებისაგან შემდგარი K სიგრძის მიმდევრობა. თუ შემოვლა ხდება საათის ისრის მოძრაობის მიმართულებით, მაშინ მას შეესაბამება 1, ხოლო თუ საწინააღმდეგო მიმართულებით – 0.

მაგალითი:

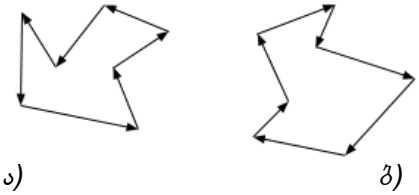
შეტანა	გამოტანა
3 3 0 0.2 1.2 1.2 2.7 0.8 4 4 4 8 4 8 8 4 8 3 10 10 10.6 15 5 15	100

მითითება. პირობიდან ჩანს, რომ მემკვიდრეთა მიერ განვლილი გზა წარმოადგენს ჩაკეტილ მრავალკუთხედს, რომლის გვერდებიც არ იკვეთებიან. ეს ფიგურა ამოზნექილი რომ იყოს, ამოცანის ამოხსნა ძალზე მარტივი იქნებოდა და მხოლოდ ორი მეზობელი გვერდის საშუალებით გავარკვევდით მოძრაობის მიმართულებას. რადგან მრავალკუთხედის გვერდები ერთმანეთს არ ჰკვეთენ, მოცემულ წერტილებზე ავაგოთ უმცირესი ამოზნექილი გარსი (ნახ. 9.8) და განვიხილოთ გარსზე მოხვედრილი წერტილები ზრდადობის მიხედვით. განძების რაოდენობა სამი მაინც არის, ამიტომ აგებულ ამოზნექილ გარსში ყოველთვის მოიძებნება ზრდადობით დალაგებული სამი წერტილი. 9.1 პარაგრაფის დასაწყისში დასმული საკითხებიდან მეორის გამოყენებით (მოხვევის მიმართულება ტეხილზე მოძრაობისას) შეგვიძლია გავარკვიოთ საით ვუხვევთ ორი მონაკვეთის შემაერთებელ წერტილში. თუ ვუხვევთ მარჯვნივ, გმოძრაობთ საათის ისრის მოძრაობის მიმართულებით, თუ ვუხვევთ მარცხნივ – საწინააღმდეგო მიმართულებით.



ნახ. 9.8

არსებობს ამოცანის ამოხსნის სხვა ალგორითმიც. მოძრაობის მიმართულება ასე განვსაზღვროთ: თუკი მოძრაობისას მრავალკუთხედის შიგა არე იმყოფება ჩვენგან მარჯვნივ, მაშინ მოძრაობა სრულდება საათის ისრის მოძრაობის მიმართულებით (ნახ. 9.9ბ), ხოლო თუ მარცხნივ – საწინააღმდეგო მიმართულებით (ნახ.9.9ა). ვთქვათ მრავალკუთხედის "ორიენტირებული: ფართობი.



ა) ბ)

ნახ. 9.9

მემკვიდრეთა ყოველი გადასვლა განვიხილოთ როგორც ვექტორი. ვთქვათ, ვექტორის დასაწყისის კოორდინატებია (x_a, y_a), ხოლო ბოლოს კოორდინატები – (x_b,y_b). განვიხილოთ ტრაპეცია, რომელიც შედგება ამ ვექტორის, აბსცისათა ღერძის და ვექტორის წვეროებიდან აბსცისათა ღერძისადმი დაშვებული ვერტიკალური მონაკვეთებით. მიღებული ტრაპეციის ორიენტირებული ფართობი ტოლია:

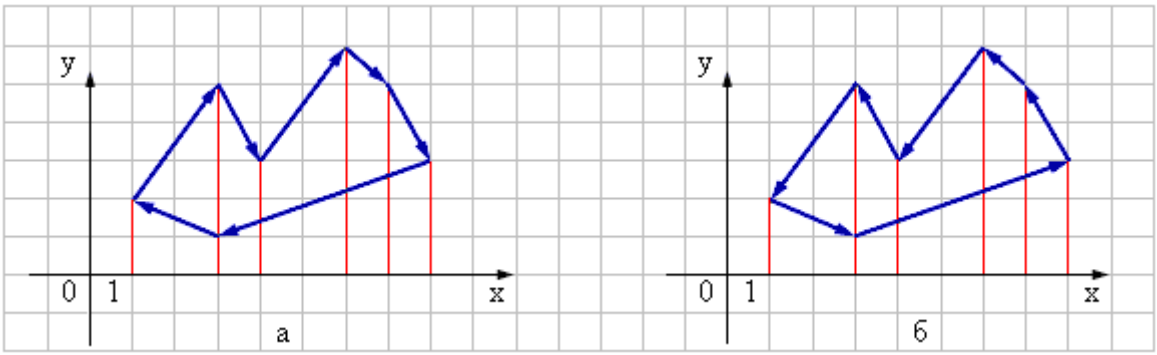
$$S = \frac{(x_b - x_a)(y_a + y_b)}{2}$$

მთელი მრავალკუთხედის ორიენტირებული ფართობი ტოლია მისი გვერდების მიერ ფორმირებული ტრაპეციების ორიენტირებული ფართობების ალგებრული ჯამისა. ნახ.9.10-ზე გამოსახული მრავალკუთხედების ორიენტირებული ფართობებია:

Sa=0,5·[(3-1)·(2+5)+(4-3)·(5+3)+(6-4)·(3+6)+(7-6)·(6+5)+(8-7)·(5+3)+(3-8)·(3+1)+(1+3)·(2+1)] =19

Sb=0,5·[(1-3)·(2+5)+(3-4)·(5+3)+(4-6)·(3+6)+(6-7)·(6+5)+(7-8)·(5+3)+(8-3)·(3+1)+(3-1)·(2+1)] = -19

თუკი ფართობი დადებითია, მაშინ მოძრაობა ხდება საათის ისრის მოძრაობის მიმართულებით, თუკი უარყოფითია – საწინააღმდეგო მიმართულებით. ამოცანაში დასმული პირობებისათვის მრავალკუთხედის ფართობი არ შეიძლება 0-ის ტოლი იყოს.



ნახ. 9.10

9.2. ნაღმები

(უკრაინის online-ოლიმპიადა, 2002-03 წლები)

გაეროს სამშვიდობო ჯარები პლანეტის ერთ-ერთ ცხელ წერტილში ახორციელებდნენ განაღმვას. მათ ჰქონდათ რუკა, რომელზეც ყოველი ნაღმი მოცემული იყო დეკარტულ კოორდინატებში მთელი რიცხვებით. გამნაღმველებმა შეამჩნიეს, რომ არცერთი სამი ნაღმი ერთ წრფეზე არ მდებარეობს. მათ გააბეს სპეციალური მავთული ნაღმიდან ნაღმამდე ისე, რომ წარმოიქმნა მინიმალური პერიმეტრის ამოზნექილი მრავალკუთხედი, ამასთან ყველა დანარჩენი ნაღმი აღმოჩნდა მრავალკუთხედის შიგნით. შეერთებული ნაღმების გაუვნებელყოფის შემდეგ მათ ისევ გააბეს მავთული იმავე პრინციპით და გააუვნებლეს მავთულით შეერთებული ნაღმები. ასე გაგრძელდა მანამ, ვიდრე მავთულის გაბმა შეუძლებელი არ გახდა. რამდენი ნაღმი დარჩა გასაუვნებელი და რამდენჯერ დასჭირდათ გამნაღმველებს მავთულის გაბმა.

შემავალი მონაცემები: პირველ სტრიქონში შემოდის ნაღმების რაოდენობა N (3≤N≤1000), მეორე სტრიქონში შემოდის N-ჯერ ორი მთელი რიცხვი x და y (-32000≤x≤32000, -32000≤y≤32000) – მორიგი ნაღმის კოორდინატები. მეორე სტრიქონში ყველა მონაცემი თითო ჰარითაა გამოყოფილი ერთმანეთისაგან.

გამომავალი მონაცემები: ერთადერთ სტრიქონში უნდა გამოიტანოთ დარჩენილი ნაღმების რაოდენობა და მავთულის გაბმის რაოდენობა.

შეტანა	გამოტანა
9 0 0 0 8 6 8 6 0 1 1 1 7 5 7 5 1 3 2	1 2

მითითება. ამოცანის ამოსახსნელად საჭიროა ამოზნექილი გარსის აგება ციკლში. ციკლის ყოველ ბიჯზე გარსში შემავალი წერტილები უნდა გაუქმდეს და ახალი გარსი აგებულ იქნას დარჩენილ წერტილებზე. პროცესი გაგრძელდება მანამ, სანამ არსებობს ორი მაინც ნაღმი. ამ ბოლო დებულებიდან ცხადია, რომ გამოსატან მონაცემთაგან პირველი (დარჩენილი ნაღმების რაოდენობა) 1 ან 0-ია.

9.3. ტყვეები ომიდან

(USACO, 2002-03 წელი, დეკემბერი, "მწვანე" დივიზიონი)

2002 წლის ხარების აჯანყების შემდეგ ძროხებს უამრავი ტყვე ადამიანის თვალყურის პრობლემა გაუჩნდათ. მათ გააჩნიათ საპყრობილე კოორდინატებით (P_x,P_y), სადაც -100000≤P_x, P_y≤100000. მის გარშემო ისინი აპირებენ მრავალი ღობის აშენებას რომ მაქსიმალურად გამორიცხონ გაპარვის შემთხვევები. ამის მიზნით ძროხებმა ციხის მახლობლად განალაგეს N (3≤N≤1000) პოსტი. ყოველი ღობე შეიცავს ციხეს და მის შიგნით განლაგებულ ღობეებს. ღობეები ერთმანეთს არ კვეთენ. ციხე და პოსტები წერტილების სახით რომ წარმოვიდგინოთ, მათგან არც ერთი სამეული არ მდებარეობს ერთ წრფეზე (არ არიან კოლინეარული). მოცემული პოსტების განლაგებების თანახმად ააგეთ მაქსიმალური რაოდენობით ერთმანეთში ჩადებული შეკრული ღობეები ისე რომ ღობეების მონაკვეთების

ბოლოები პოსტებს წარმოადგენდნენ. ღობეებს შორის და აგრეთვე ციხესა და ყველაზე შიდა ღობეს შორის მცველებს პატრულირება უნდა შეეძლოთ (წიბოს მეშვეობით არ შეაერთოთ ერთმანეთს ჩადებული ღობეები).

პირველი სტრიქონი შეიცავს სამ ჰარით დაშორებულ მთელს N, Px, და Py. სტრიქონები 2-დან (N+1)-ის ჩათვლით ორი მთელი Xi და Yi ($-100000 \le X_i, Y_i \le 100000$) – პოსტების კოორდინატები.

შემავალი ფაილის მაგალითი (ფაილი captives.in):

```
8 -1 0
2 2
2 -2
-2 2
-2 -2
0 10
8 0
-12 1
1 -5
```

გამომავალი ფაილის ფორმატი: ერთი სტრიქონი ერთი მთელი რიცხვით: ჩადებული ღობეების მაქსიმალური რაოდენობა.

გამომავალი ფაილის მაგალითი: (ფაილი captives.out):

```
2
```

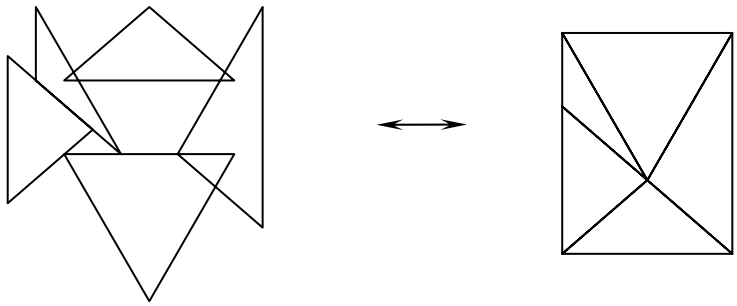
მითითება. ქ ამოცანა წინას ანალოგიურია, ოღონდ აქ არ მოითხოვება ამოზნექილი გარსების აგება ყველა მოცემულ წერტილზე. გადამწყვეტი მნიშვნელობა ენიჭება ციხის მდებარეობას, რომლის გარშემოც ამოზნექილი გარსები კონცენტრულად უნდა აიგოს (რომელიღაც მონაცემებისთვის აგება საერთოდ არ მოხერხდება). კონცენტრული გარსების აგება უნდა შევწყვიტოთ მაშინ, როცა მორიგ გარსში ციხის კოორდინატებიც მოხვდება.

9.4. მინა

(რუსეთის მოსწავლეთა ოლიმპიადა, დასკვნითი ტური, 2002-03 წლები)

მაგიდაზე განლაგებულია მინის რამდენიმე სამკუთხა ნატეხი. ცნობილია, რომ ა) ყველა ეს ნატეხი ეკუთვნის ერთ მართკუთხა მინას, რომელიც თავდაპირველად ისე იდო, რომ მისი გვერდები მაგიდის გვერდების პარალელური იყო; ბ) ნატეხები მიღებულია მინის ერთ-ერთ წერტილზე დარტყმით, ამასთან ეს წერტილი არ მდებარეობს მინის საზღვარზე; გ) დარტყმის მომენტში ყოველი ნატეხის ერთ-ერთი წვერო იმყოფებოდა დარტყმის წერტილში; დ) რომელიღაც ნატეხები შეიძლება გადაადგილეს; ე) მინა შეიძლება აღდგენილ იქნას ნატეხების პარალელური გადაადგილებით, ანუ არცერთი ნატეხი არ შემოუბრუნებიათ.

დაწერეთ პროგრამა, რომელიც ჩამოთვლილი დებულებების გათვალისწინებით მთლიანად აღადგენს მართკუთხა მინას ყველა არსებული ნატეხისაგან (ნახ. 9.11) ან გაარკვევს, რომ ამის გაკეთება შეუძლებელია.



ნახ. 9.11

შემავალი მონაცემები: პირველ სტრიქონში შემოდის ერთადერთი რიცხვი N ($1 \le N \le 2000$) – ნატეხების რაოდენობა. შემოვიღოთ კოორდინატთა სისტემა ისე, რომ კოორდინატთა ღერძები მაგიდის გვერდების პარალელური იყოს. მომდევნო N სტრიქონიდან თითოეულში ჩაწერილი იქნება რიცხვთა სამი წყვილი, რომლებიც აღწერენ ნატეხთა წვეროების მიმდინარე კოორდინატებს. ყველა კოორდინატი მთელი რიცხვია, რომელიც არ აღემატება მოდულით 10000-ს. ყველა ნატეხი არანულოვანი ფართობისაა.

გამომავალი მონაცემები: თუკი მინის აღდგენა მოცემული დებულებებით შესაძლებელია, მაშინ N სტრიქონში გამოიტანეთ ნატეხების წვეროთა კოორდინატები მინის აღდგენის შემდეგ. მინა იმგვარად უნდა აღდგეს, რომ მისი ყველა კუთხის კოორდინატები არაუარყოფითი იყოს, ხოლო ერთ-ერთი წვერო მოთავსებული იყოს კოორდინატთა სათავეში. ნატეხების ჩამოთვლის რიგი უნდა ემთხვეოდეს ნატეხების თანმიმდევრობას შემომავალ ფაილში.

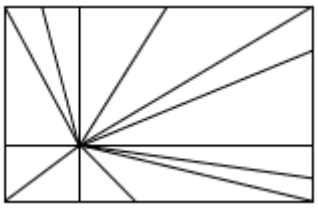
რამდენიმე ამოხსნის შემთხვევაში გამოიტანეთ ერთ-ერთი. თუკი მინის აღდგენა შეუძლებელია გამომავალ ფაილში გამოიტანეთ სიტყვა "NO".

შეზღუდვები: დრო – 2 წმ, მეხსიერება – 32 მბ.

მაგალითები:

glass.in	glass.out	glass.in	glass.out
5 0 0 3 3 0 9 1 3 7 3 4 -3 1 6 7 6 4 9 5 3 8 6 8 9 6 4 9 1 9 7	0 0 3 3 0 9 0 9 6 9 3 3 0 0 6 0 3 3 3 3 6 6 6 9 3 3 6 0 6 6	4 0 0 20 -20 20 20 0 0 -20 20 -20 -20 0 0 -20 20 20 20 -1 0 20 -20 -20 -20	NO

მითითება. თუკი ნატჯების შესახებ ჩამოყალიბებული დებულებები სრულდება, მაშინ ყოველ სამკუთხედში მოიძებნება ორი წვერო, რომელთა x ან y კოორდინატები ემთხვევა. და თუ წვეროთა ასეთი წყვილი ნატჯში მხოლოდ ერთია, მაშინ სამკუთხედის დარჩენილი მესამე წვერო აუცილებლად წარმოადგენს დარტყმის წერტილში მდებარე წვეროს. გამონაკლისს წარმოადგენენ მხოლოდ ის მართკუთხა სამკუთხედები, რომელთა კათეტებიც მაგიდის გვერდების პარალელურია. ასეთ ნატჯებს წვეროთა მითითებული წყვილი ორი აქვს, ამიტომ იმის გარკვევა, რომელი კუთხეა მოთავსებული დარტყმის წერტილში, არც ისე იოლია.

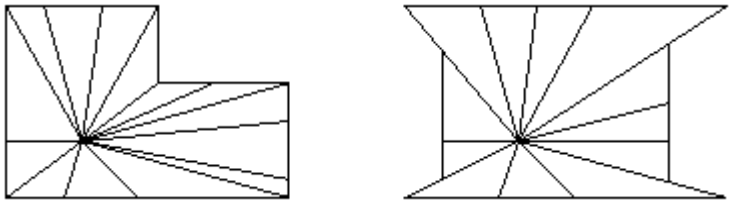

ნახ. 9.12

თუმცა შევნიშნოთ, რომ მინის გატჯვის შემდეგ თითოეულ გვერდზე შეიძლება წარმოიქმნას არაუმეტეს 2 მართკუთხა სამკუთხედი. მაშასადამე, მართკუთხა სამკუთხედების საერთო რაოდენობა არ შეიძლება იყოს 8-ზე მეტი (იხ. ნახ. 9.12). თითოეული მართკუთხა სამკუთხედისათვის, რომლის კათეტებიც მაგიდის გვერდების პარალელურია, დარტყმის წერტილში შეიძლება მოთავსდეს ორი წვეროდან ერთ-ერთი. მინის აღდგენისათვის საჭიროა ორივე ვარიანტის განხილვა. ასეთი შემთხვევების მაქსიმალური რაოდენობა არ აჭარბებს $2^8=256$-ს.

თუკი ყოველი ნატჯისათვის დარტყმის წერტილში მყოფი წვერო ფიქსირებულია, მაშინ მინის აღდგენა ვცადოთ შემდეგნაირად. შემოვიღოთ კოორდინატთა ახალი სისტემა ისე, რომ ღერძები დარჩნენ მაგიდის გვერდების პარალელურად, ხოლო კოორდინატთა სათავე გადავიტანოთ დარტყმის წერტილში. გადავნაცვლოთ ყველა ნატჯი იმგვარად, რომ i-ური ნატჯის წვეროების კოორდინატებს ჰქონდეთ შემდეგი მნიშვნელობები: (0,0), (x_{i1}-x_{i0}, y_{i1}-y_{i0}), (x_{i2}-x_{i0}, y_{i2}-y_{i0}), სადაც (x_{i0},y_{i0}) არის i-ური ნატჯის წვეროების კოორდინატები დარტყმის წერტილში, ხოლო (x_{i1},y_{i1}) და (x_{i2},y_{i2}) იმავე ნატჯის დანარჩენი ორი წვეროს კოორდინატები. ამასთან, $x_{i1}=x_{i2}$ ან $y_{i1}=y_{i2}$ თვისება კოორდინატთა ახალ სისტემაშიც სრულდება.

ფაქტობრივად პარალელური გადატანის საშუალებით ჩვენ ყველა ნატჯი გადავნაცვლეთ მის საწყის მდგომარეობაში. ახლა შესამოწმებელია, სრულდება თუ არა ამოცანის პირობაში მოცემული დებულებები.

კოორდინატთა ახალ სისტემაში თითოეული ნატჯის ტოლი კოორდინატების ნიშნით შეგვიძლია დავადგინოთ მართკუთხედის რომელ გვერდს – მარცხენას, მარჯვენას, ზედას თუ ქვედას ეკრდნობა ეს ნატჯი. ყოველი გვერდისათვის ცალ-ცალკე განვიხილოთ შესაბამისი სამკუთხა ნატჯების მონაკვეთები. ნებისმიერი ალგორითმით დავალაგოთ ჰორიზონტალური მონაკვეთები მარცხენა ბოლოების, ხოლო ვერტიკალურები – ქვედა ბოლოების ზრდადობით. ამის შემდეგ რთული არაა იმის შემოწმება, აკმაყოფილებენ თუ არა ეს მონაკვეთები მთლიანად მართკუთხედის შესაბამის გვერდს ან გადაფარავენ თუ არა ერთმანეთს. გარდა ამისა უნდა შევამოწმოთ მართკუთხედის თითოეული გვერდის საწყისი და საბოლოო კოორდინატები და თითოეული გვერდის შესაბამისი მონაკვეთები მდებარეობენ თუ არა ერთ წრფეზე. ასეთი შემთხვევების შესაბამისი ნახატები მოცემულია ნახ. 9.13.


ნახ. 9.13

თუკი ყველა შემოწმებამ წარმატებით ჩაიარა და არსებული ნატჯები ნამდვილად ადგენენ ერთიან მართკუთხედს, გვრჩება მხოლოდ ნატჯების წვეროთა კოორდინატების გამოტანა, რისთვისაც საჭირო გახდება ძველ კოორდინატთა სისტემაში დაბრუნება და კოორდინატთა შესაბამისად გადანგარიშება.

ალგორითმის მუშაობის დრო ფაქტობრივად ემთხვევა სორტირების გამოყენებული ალგორითმის დროს, რადგან ყველა დანარჩენი შემოწმება O(N) დროში შეიძლება განხორციელდეს.

9.5. მესერი

(მოსწავლეთა საერთაშორისო ოლიმპიადა, 2003 წელი, აშშ)

ფერმერი დონი ათვალიერებს მესერს, რომლითაც შემოღობილია მისი N მეტრი სიგრძის გვერდის მქონე კვადრატული, ბრტყელი ნაკვეთი ($2\le N\le 500000$). მესრის ერთი კუთხე მოთავსებულია საწყის (0,0) წერტილში, ხოლო მისი მოპირდაპირე კუთხე კი – (N,N) წერტილში. მესრის გვერდები X და Y ღერძების პარალელურია.

მესრის საყრდენი ბოძები დასობილია მის ოთხივე კუთხეში და ასევე ყოველ ერთ მეტრში თითოეული გვერდის გასწვრივ. ასე, რომ ბოძების საერთო რაოდენობა 4N-ის ტოლია. ბოძები ვერტიკალურია და ითვლება, რომ მათ

რადიუსი არ აქვს (იგი ნულის ტოლია). ფერმერ დონს სურს დაადგინოს, თუ რამდენი ბოძის დანახვას შესძლებს იგი, როცა განსაზღვრულ წერტილში იდგება მესრის შიგნით.

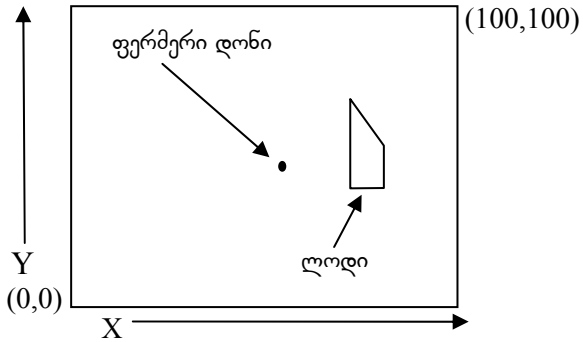
ფერმერ დონის ნაკვეთში მდებარეობს R რაოდენობის ($1 \le R \le 30000$) დიდი ლოდი, რომლებიც ხელს უშლის მას ზოგიერთი ბოძის დანახვაში. დონი საკმაოდ დაბალია და ამიტომ მას არ შეუძლია ლოდების უკან მდებარე ბოძების დანახვა. თითოეული ლოდის ფუძე არანულოვანი ფართობის მქონე ამოზნექილი მრავალკუთხედია, რომლის წვეროების კოორდინატები მთელ რიცხვებს წარმოადგენს. ყველა ლოდი ვერტიკალურად დგას. ლოდებს საერთო წერტილები არ გააჩნიათ არც ერთმანეთთან და არც მესერთან. წერტილი, სადაც დგას ფერმერი დონი, მდებარეობს ნაკვეთის შიგნით, მაგრამ არა მის საზღვარზე. იგი არ მდებარეობს ასევე ლოდების საზღვარზე ან მათ შიგნით.

მოცემულია ფერმერ დონის ნაკვეთის ზომა, მასში ლოდების მდებარეობები და ფორმები და პოზიცია, რომელშიც ფერმერი დონი დგას. დაადგინეთ იმ ბოძების რაოდენობა, რომელთა დანახვაც შეუძლია ფერმერ დონს. თუ ლოდის ფუძის წვერო, ფერმერი დონის პოზიცია და მესრის ბოძი ერთ წრფეზეა განლაგებული, მაშინ მას ამ ბოძის დანახვა არ შეუძლია.

შესატანი მონაცემები (ფაილი boundary.in): შესატან მონაცემთა ფაილის პირველი სტრიქონი შეიცავს ერთი ჰარით გამოყოფილ ორ მთელ N და R რიცხვს. მეორე სტრიქონში ჩაწერილია ერთი ჰარით გამოყოფილი ორი მთელი რიცხვი – იმ პოზიციის X და Y კოორდინატები მესრის შიგნით, რომელშიც ფერმერი დონი იმყოფება. ფაილის დანარჩენ სტრიქონებში აღწერილია ნაკვეთში მდებარე R რაოდენობის ლოდი: i-ური ლოდის აღწერა იწყება სტრიქონით, რომელიც შეიცავს ერთ მთელ $3 \le p_i \le 20$ რიცხვს – ლოდის ფუძის წვეროების რაოდენობას. მომდევნო p_i რაოდენობის სტრიქონიდან თითოეულში ჩაწერილია ერთი ჰარით გამოყოფილი ორი მთელი რიცხვი – ფუძის წვეროს X და Y კოორდინატები. ლოდის ფუძის წვეროები ურთიერთგანსხვავებულია და მოცემულია საათის ისრის მოძრაობის საწინააღმდეგო მიმართულებით.

შეტანის მაგალითი:

```
100 1
60 50
5
70 40
75 40
80 40
80 50
70 60
```



ნახ. 9.14.

შენიშვნა: ყურადღება მიაქციეთ იმას, რომ ნახაზზე მოცემული ლოდის ფუძე შეიცავს ერთ წრფეზე მდებარე სამ წვეროს: (70,40), (75,40) და (80,40).

გამოსატანი მონაცემები (ფაილი boundary.out): გამოსატან მონაცემთა ფაილის ერთადერთ სტრიქონში უნდა ჩაიწეროს ერთი მთელი რიცხვი – იმ ბოძების რაოდენობა, რომელთა დანახვაც შეუძლია ფერმერ დონს.

გამოტანის მაგალითი:

```
319
```

შეზღუდვები: პროგრამის მუშაობის დრო – 1წმ და გამოყენებული მეხსიერება – 64 მბ

*მითითება. ამოცანის ამოხსნისათვის შეიძლება რამდენიმე გზა ვცადოთ, მაგრამ ყველა შემთხვევაში გასათვალისწინებელია ის, რომ გეომეტრიული ამოცანების ამოხსნისას გამოთვლების მანქანურმა ცდომილებამ შეიძლება სერიოზული პრობლემები შეგვიქმნას. ყველაზე მარტივი ალგორითმი სრული გადარჩევაა. იმ წერტილიდან, სადაც ფერმერი დონი დგას გავუშვათ სხივი ბოძებისა და ლოდების ყველა წყვილისათვის და შევამოწმოთ ჩანს თუ არა შესაბამისი ბოძი. ამ ალგორითმის მუშაობის დროა O(N*R) და კარგი რეალიზაციის შემთხვევაში ქულების 72%-ს აგროვებს. იმავე O(N*R) სირთულისაა ალგორითმი, სადაც მქერი შეიძლება წარმოვადგინოთ ორგანზომილებიანი მასივის სახით, თითოეული ლოდისათვის განვსაზღვროთ დიაპაზონი, რომელსაც იგი ფარავს და მასივის შესაბამისი წევრები მოვნიშნოთ. ეს ალგორითმი აგროვებს ქულების 80%-ს.*

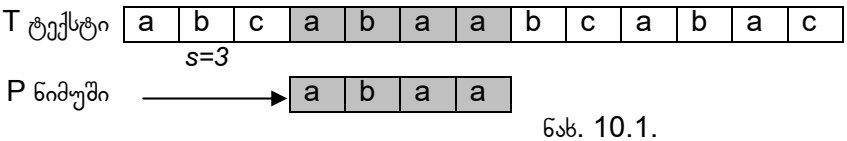
სრულყოფილად ამოხსნისათვის მქერი განვიხილოთ, როგორც წრიული სია. მოვახდინოთ ლოდთა საზღვრების (ანუ მარჯვენა და მარცხენა კიდური წერტილების) სორტირება ფერმერ დონის დგომის წერტილის მიმართ. თუკი დიაპაზონები ერთმანეთს თუნდაც ნაწილობრივ ფარავენ, ისინი გავაერთიანოთ. ბოლოს ერთი ციკლით, რომელიც მიღებული დიაპაზონების მიხედვით აიგება, გადავთვალოთ დიაპაზონებს გარეთ დარჩენილი ბოძების რაოდენობა.

ამ ალგორითმის მუშაობის დროა O(R log R + N log d), სადაც d ერთი ლოდის მიერ დაფარული ბოძების მაქსიმალური რაოდენობაა. ალგორითმი აგროვებს ქულების 100%-ს.

10. სტრიქონის ძებნა

ტექსტური რედაქტორების უმეტესობას შეუძლია მოცემული სიტყვის მოძებნა დოკუმენტში. ამოცანა ამ დავალების შესრულების სისწრაფეში მდგომარეობს. მეორე მაგალითად შეიძლება დასახელდეს ნუკლეოტიდთა მოცემული ჯაჭვის მოძებნა დნმ-ის მოლეკულაში. ზოგადად კი **ქვესტრიქონის ძებნის ამოცანა** (string-matching problem) მდგომარეობს შემდეგში: ვთქვათ, მოცემულია "ტექსტი" – n სიგრძის მქონე T[1..n] მასივი და "ნიმუში" – m სიგრძის მქონე P[1..m] მასივი, სადაც m≤n. ჩვენ ვთვლით, რომ P და T მასივების ელემენტები წარმოადგენენ რომელიღაც სასრული Σ ალფავიტის სიმბოლოებს (მაგალითად, Σ={0,1} ან Σ={a,b,…z}). Σ ალფავიტის სიმბოლოებისაგან შემდგარ მასივებს ხშირად უწოდებენ **სტრიქონებს** (strings) ან **სიტყვებს** ამ ალფავიტში.

ვიტყვით, რომ P ნიმუში **s წანაცვლებით შედის** (occurs with shift s) ან ტოლფასია, **შედის s+1 პოზიციიდან** (occurs beginning at position s+1) T ტექსტში, თუ 0≤s≤n-m და T[s+1..s+m]=P[1..m] (სხვაგვარად რომ ვთქვათ, T[s+j]=P[j] ნებისმიერი 0≤j≤m-სათვის). თუკი P შედის T ტექსტში s წანაცვლებით, მაშინ იტყვიან, რომ s **დასაშვები წანაცვლებაა** (valid shift), წინააღმდეგ შემთხვევაში s – **დაუშვებელი წანაცვლებაა** (invalid shift). ქვესტრიქონის ძებნის ამოცანა მდგომარეობს T ტექსტისა და P ნიმუშისათვის ყველა დასაშვები წანაცვლების მოძებნაში.



ნახ. 10.1.

ნახ.10.1-ზე გამოსახულია ქვესტრიქონის ძებნის მაგალითი. საჭიროა მოიძებნოს P=abaa ნიმუში T=abcabaabcabac ტექსტში. ნიმუში შედის ტექსტში მხოლოდ ერთხელ s=3 წანაცვლებით, ანუ 3 – დასაშვები წანაცვლებაა.

Σ*-ით აღინიშნება ყველა სასრული სტრიქონის სიმრავლე Σ ალფავიტში **ცარიელი სტრიქონის** (empty string) ჩათვლით, რომელსაც აქვს ნულოვანი სიგრძე და აღინიშნება ε-ით. x სტრიქონის სიგრძე აღინიშნება |x|-ით. x და y სტრიქონების **შეერთება** ანუ **კონკატენაცია** (concatenation) ეწოდება სტრიქონს, რომელშიც ჩაწერშუილია x და უალოდ მასზე მიბმულია y სტრიქონიც. x და y სტრიქონების კონკატენაცია აღინიშნება xy-ით. ცხადია, რომ |xy|=|x|+|y|.

ვიტყვით, რომ w სტრიქონი წარმოადგენს x სტრიქონის **პრეფიქსს** (prefix) ანუ **დასაწყისს**, თუ x=wy, რომელიღაც y∈Σ*. ვიტყვით, რომ w სტრიქონი წარმოადგენს x სტრიქონის **სუფიქსს** (suffix) ანუ **ბოლოს**, თუ x=yw, რომელიღაც y∈Σ*. ჩავწეროთ w⊏x, თუ w არის x-ის პრეფიქსი და w⊐x – თუ w არის x-ის სუფიქსი. მაგალითად ab⊏abdcca და cca⊐abdcca. შევნიშნოთ, რომ ამ აღნიშვნების გამოყენებისას სიფრთხილეა საჭირო, რადგან a⊏b და b⊐a სრულიად სხვადასხვა რამეს აღნიშნავენ.

ცარიელი სტრიქონი ნებისმიერი სტრიქონის პრეფიქსად და სუფიქსად ითვლება. თუ w წარმოადგენს x-ის პრეფიქსს ან სუფიქსს, მაშინ |w|≤|x|. ნებისმიერი x და y სტრიქონისა და ნებისმიერი a სიმბოლოსათვის თანაფარდობები x⊐y და xa⊐ya ტოლფასია. ⊏ და ⊐ თანაფარდობები ტრანზიტიულია. ადგილი აქვს ლემას:

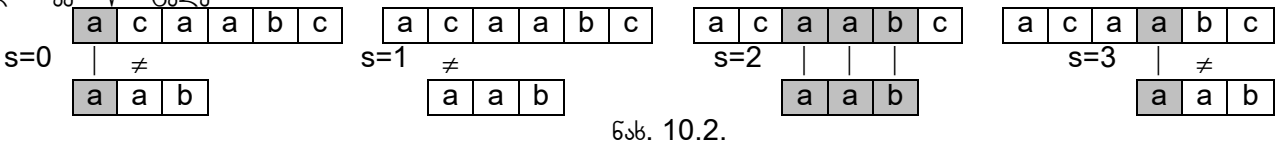
ლემა 10.1 (ორი სუფიქსის შესახებ). ვთქვათ, x, y და z სტრიქონებია, რომლებისთვისაც x⊐z და y⊐z, მაშინ x⊐y, თუ |x|≤|y|; y⊐x, თუ |x|≥|y| და x=y, თუ |x|=|y|;

10.1. უმარტივესი ალგორითმი

T ტექსტში P ნიმუშის ძებნის ყველაზე მარტივი გზაა s-ის თითოეული შესაძლო მნიშვნელობისათვის ყველა შესაძლო (n-m+1)-დან თანმიმდევრობით შევამოწმოთ ტოლობა P[1..m]=T[s+1..s+m]:

```
NAIVE-STRING-MATCHER (T,P)
1 n=LENGTH[T]
2 m=LENGTH[P]
3 for s=0 to n-m do {
4                   k=0
5                   for j=1 to m do {
6                   if P[j]≠T[s+j] then {  break  }
7                                       else {  k=k+1 }    }
8                   if k=m then {  print "ქვესტრიქონის შესვლის წანაცვლებაა" s }          }
```

შეიძლება ითქვას, რომ ჩვენ ვამოძრავებთ ნიმუშს ტექსტის გასწვრივ და ვამოწმებთ მის ყველა მდგომარეობას. ნახ. 10.2-ზე გამოსახულია P=aab ნიმუშის ძებნა T=acaabc ტექსტში. თითოეული წანაცვლებისათვის ის სიმბოლოები, რომელთა შედარებამაც დადებითი შედეგი მოგვცა, აღნიშნულია რუხი ფერით. s=2 ერთადერთი დასაშვები წანაცვლებაა.



ნახ. 10.2.

NAIVE-STRING-MATCHER პროცედურის მუშაობის დრო უარეს შემთხვევაში არის $\Theta((n-m+1)m)$. თუ $T=a^n$ (a სიმბოლო გამეორებულია n-ჯერ) და $P=a^m$, მაშინ n-m+1 შემოწმებიდან თითოეულში მოხდება m სიმბოლოს შედარება, რაც ჯამში მოგვცემს (n-m+1)m, რაც წარმოადგენს $\Theta(n^2)$-ს, როცა $m=\lfloor n/2 \rfloor$.

უმარტივესი ალგორითმის არაეფექტურობას განაპირობებს ის, რომ s წანაცვლების შემოწმების დროს T ტექსტის შესახებ მიღებული ინფორმაცია საერთოდ არ გამოიყენება მომდევნო წანაცვლებების შემოწმებისას.

10.2. რაბინ-კარპის ალგორითმი

რაბინმა და კარპმა შექმნეს ქვესტრიქონის ძებნის ალგორითმი, რომელიც პრაქტიკაში ეფექტურია და ამასთან შესაძლებელია მისი განზოგადება სხვა ანალოგიურ ამოცანებზე (მაგალითად, ნიმუშის ძებნა ორგანზომილებიან ბადეზე). მართალია, წინა ალგორითმის მსგავსად, უარეს შემთხვევაში რაბინ-კარპის ალგორითმის მუშაობის დროა $\Theta((n-m+1)m)$, მაგრამ საშუალოდ ის საკმაოდ სწრაფად მუშაობს.

დავუშვათ, რომ $\Sigma=\{0,1,2,...,9\}$ (საზოგადოდ შეიძლება ჩავთვალოთ, რომ ყოველი სიმბოლო Σ ალფავიტში d-ობითი ციფრია, სადაც $d=|\Sigma|$). მაშინ k სიმბოლოსაგან შედგენილი სტრიქონი შეგვიძლია განვიხილოთ k-ნიშნა რიცხვის ათობითი ჩანაწერი, ხოლო თავად სიმბოლოები – როგორც ციფრები.

თუ P[1..m] ნიმუშია, მაშინ p-თი აღვნიშნოთ რიცხვი, რომლის ათობით ჩანაწერსაც ის წარმოადგენს. ანალოგიურად, თუ T[1..n] ტექსტია, მაშინ s=0,1,…,n-m ყველა მნიშვნელობისათვის t_s-ით აღვნიშნოთ რომელიც წარმოადგენს T[s+1..s+m] სტრიქონის ათობით ჩანაწერს. ცხადია, რომ s დასაშვები წანაცვლებაა მაშინ და მხოლოდ მაშინ, თუ $t_s=p$. თუკი ჩვენ შევძლებთ, რომ p გამოვთვალოთ O(m) დროში, ხოლო ყველა t_i – O(n) დროში (ჯერჯერობით ნუ გავითვალისწინებთ გარემოებას, რომ გამოთვლები უნდა ჩატარდეს ძალიან დიდ რიცხვებზე), მაშინ ჩვენ შევძლებთ მოვძებნოთ ყველა დასაშვები წანაცვლება O(n) დროში, შევადარებთ რა p-ს რიგრიგობით ყველა t_s-თან.

p-ს გამოთვლა O(m) დროში შესაძლებელია **ჰორნერის სქემით** (Horner`s rule):

$$p=P[m]+10(P[m-1]+10(P[m-2]+...+10(P[2]+10P[1])..))$$

ასევე O(m) დროში შესაძლებელია t_0-ის გამოთვლაც.

შევნიშნოთ, რომ თუკი t_s ცნობილია, მაშინ t_{s+1}-ის გამოსათვლელად საჭიროა O(1) დრო. მართლაც

$$t_{s+1}=10(t_s-10^{m-1}T[s+1])+T[s+m+1] \qquad (10.1)$$

T[s+1..s+m] სტრიქონიდან T[s+2..s+m+1] სტრიქონის მისაღებად უნდა წავშალოთ T[s+1] (ანუ t_s-ს გამოვაკლოთ $10^{m-1}T[s+1]$) და მარჯვნიდან მივუწეროთ T[s+m+1] (ანუ მიღებულ სხვაობას გავამრავლებთ 10-ზე და მივუმატებთ T[s+m+1]-ს). გამოთვლათა ღირებულება (10.1) ფორმულაში შემოსაზღვრულია 10^{m-1} მუდმივით (რომლის გამოთვლა წინასწარაც შეიძლება). აქედან გამომდინარე, p და $t_0,t_1,\ldots,t_{n-m}$ შეიძლება მოიძებნონ O(n+m) დროში. ამავე დროში შეიძლება მოიძებნოს ყველა P[1..m] ნიმუში T[1..n] ტექსტში.

აქამდე ჩვენ არ ვითვალისწინებდით p და t_s რიცხვების ზომებს. ისინი კი იმდენად დიდები არიან, რომ მათზე ჩატარებული არითმეტიკული ოპერაციების შესრულების დროდ O(1)-ის ჩათვლა მართებული არ იქნება. ეს პრობლემა შეიძლება გადაწყდეს შემდეგნაირად: p და t_0 რიცხვების გამოთვლა და აგრეთვე (10.1) ფორმულით საჭირო გამოთვლები ჩატარდეს ფიქსირებული q რიცხვის მოდულით (იხ. თავი "რიცხვთა თეორია"). მაშინ ეს რიცხვები არ გადააჭარბებენ q-ს და შეგვიძლია ჩავთვალოთ, რომ p და t_s გამოითვლებიან O(n+m) დროში. q-ს როლში, როგორც წესი ირჩევენ ისეთ მარტივ რიცხვს, რომლისთვისაც 10q ეტევა მანქანურ სიტყვაში (საზოგადოდ, {0,1,2,...,d} ალფავიტისათვის ირჩევენ ისეთ მარტივ q რიცხვს, რომ dq მოთავსდეს მანქანურ სიტყვაში), ამიტომ ყველა

არითმეტიკული ოპერაცია ხორციელდება ერთი სიტყვის ფარგლებში. რეკურენტული დამოკიდებულება (10.1) ფორმულაში ღებულობს სახეს

$$t_{s+1}=(d(t_s - T[s+1]h)+T[s+m+1]) \bmod q \qquad (10.2)$$

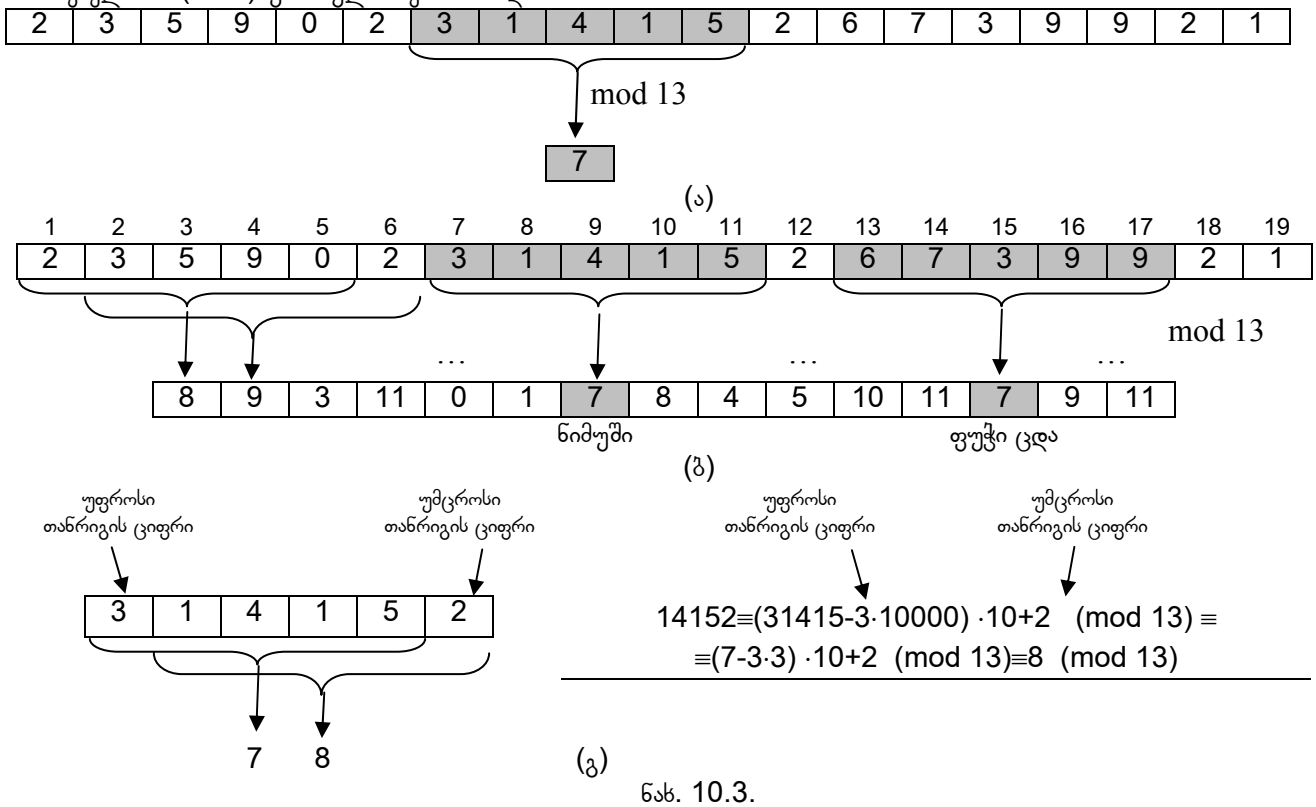
სადაც $h \equiv d^{m-1} \pmod q$.

გამოთვლებს q-ს მოდულით ერთი ნაკლი გააჩნია $t_s=p \pmod q$ ტოლობის შესრულება ჯერ კიდევ არ ნიშნავს, რომ $t_s=p$. ამიტომ თუ $t_s \neq p \pmod q$ ცალსახად ნიშნავს, რომ s წანაცვლება დაუშვებელია და მას გვერდი უნდა ავუაროთ, $t_s=p \pmod q$ ტოლობის შემთხვევაში ჯერ კიდევ უნდა შემოწმდეს ემთხვევა თუ არა ერთმანეთს P[1..m] და T[s+1..s+m] სტრიქონები. თუ ემთხვევა, ჩვენ ვიპოვეთ ნიმუში ტექსტში, ხოლო თუ არ ემთხვევა – მოხდა **ფუჭი ცდა** (spurios hit). თუკი მარტივი q რიცხვი საკმარისად დიდია, იმედი უნდა ვიქონიოთ, რომ ფუჭი ცდების ანალიზი დიდ დროს არ წაიღებს.

მოვიყვანოთ RABIN-KARP-MATCHER პროცედურის ტექსტი. მის შემავალ მონაცემებს წარმოადგენენ T ტექსტი, P ნიმუში, თვლის სისტემის ფუძე d (როგორც წესი, იღებენ $d=|\Sigma|$) და მარტივი რიცხვი q.

```
RABIN-KARP-MATCHER(T,P,d,q)
1 n=length[T]
2 m=length[P]
3 h=d^(m-1) mod q
4 p=0
5 t_0=0
6 for i=1 to m do  {
7                  p=(dp+P[i] mod q
8                  t_0=(dt_0+T[i] mod q    }
9 for s=0 to n-m do {
10                 if p=t_s then {
11                         if P[1..m]=T[s+1..s+m] then { print "ნიმუშის დასაშვები წანაცვლებაა"s }
12                 if s<n-m then {     t_{s+1}=(d(t_s – T[s+1]h)+T[s+m+1]) mod q     }        }
```

აღვწეროთ პროცედურის მუშაობა. ყველა სიმბოლო განიხილება, როგორც d-ობითი ციფრი. 1-5 სტრიქონებში ცვლადებს ენიჭებათ საწყისი მნიშვნელობები (h წარმოადგენს "უფროსი თანრიგის ერთეულს" d-ობით სისტემაში). 6-8 სტრიქონებში ჰორნერის სქემის მიხედვით გამოითვლება p და t_0-ის მნიშვნელობები. 9-12 სტრიქონებში გადაირჩევა s-ის ყველა შესაძლო მნიშვნელობა. თუკი მე-10 სტრიქონში აღმოჩნდება, რომ $p \equiv t_s$, მაშინ ერთმანეთს შედარდება if P[1..m] და T[s+1..s+m] სტრიქონები და მათი იგივეობის შემთხვევაში დაიბეჭდება შეტყობინება (სტრიქონი 11). $p \neq t_s$, მაშინ პროგრამა ამოწმებს უნდა შესრულდეს თუ არა ციკლი კიდევ და თუ უნდა შესრულდეს, ანახლებს t-ს მნიშვნელობას (10.2) ფორმულის შესაბამისად.



ნახ. 10.3.

ნახ. 10.3-ზე ნაჩვენებია რაბინ-კარპის ალგორითმის მუშაობა. $\Sigma=\{0,1,2,...,9\}$, q=13, P=31415. (ა) ნახაზზე მოცემულია T სტრიქონი, რომელზეც რუხი ფერით გამოყოფილია 5 სიგრძის მქონე ჩარჩო, რომლის მნიშვნელობა 13-

ის მოდულით უდრის 7-ს. (ბ) ნახაზზე მოცემულია 5 სიგრძის მქონე ყველა ქვესტრიქონის შესაბამისი რიცხვითი მნიშვნელობა. ჩვენ გვაინტერესებს მხოლოდ ის ქვესტრიქონები, რომელთა მნიშვნელობებია 7, რადგან $31415 \equiv 7 \pmod{13}$. ასეთი ქვესტრიქონი მხოლოდ ორია. ერთი მათგანი შეესაბამება ნიმუშს, ხოლო მეორე – ფუჭ ცდას. (გ)-ზე ნაჩვენებია რიცხვითი მნიშვნელობის ცვლილება ჩარჩოს გადაადგილებისას. წინა ჩარჩოში ეწერა 31415. უფროსი თანრიგის ციფრის წაშლითა და უმცროსი თანრიგის ციფრის მიწერით მივიღებთ 14152. იგივე გამოთვლები 13-ის მოდულით ღებულობენ მნიშვნელობა 8-ს.

უარეს შემთხვევაში ეს პროცედურა უმარტივესი ალგორითმის მსგავსად მუშაობს $\Theta((n-m+1)m)$ დროში, რადგან ყველა დასაშვები წანაცვლებისათვის ხდება სიმბოლოთა სათითაოდ შედარება, თუმცა ხშირად დასაშვებ წანაცვლებათა რაოდენობა მცირეა და მაშინ რაბინ-კარპის ალგორითმის მუშაობის დროა $O(m+n)$.

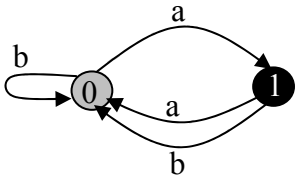
10.3. ქვესტრიქონის ძებნა სასრული ავტომატის საშუალებით

სასრული ავტომატი (finite automation) წარმოადგენს $M=(Q,q_0,A,\Sigma,\delta)$ ხუთეულს, სადაც Q – **მდგომარეობათა** (states) სასრული სიმრავლეა, $q_0 \in Q$ – **საწყისი მდგომარეობა** (start states), $A \subseteq Q$ – **დასაშვებ მდგომარეობათა** (accepting states) სასრული სიმრავლე, Σ – სასრული **შემავალი ალფავიტი** (input alphabet), δ – ფუნქცია $Q \times \Sigma$-დან Q-ში, რომელსაც ავტომატის **გადასვლის ფუნქციას** (transition fuction) უწოდებენ.

თავდაპირველად სასრული ავტომატი იმყოფება q_0 მდგომარეობაში. შემდეგ ის თანმიმდევრობით კითხულობს სიმბოლოებს შემომავალი სტრიქონიდან. როდესაც ავტომატი იმყოფება q მდგომარეობაში და კითხულობს a სიმბოლოს, იგი გადადის $\delta(q,a)$ მდგომარეობაში. თუკი ავტომატი იმყოფება $q \in A$ მდგომარეობაში, იტყვიან, რომ შემომავალი სტრიქონის წაკითხული ნაწილი **დასაშვებია** (accepts); თუკი $q \notin A$, მაშინ სტრიქონის წაკითხული ნაწილი **უარყოფილია** (is rejected).

მდგომარეობა	შემავალი მონაცემი a	b
0	1	0
1	0	0

(ა)



(ბ)

ნახ. 10.4.

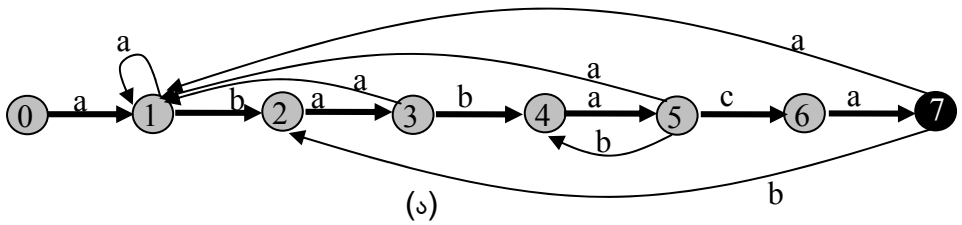
ნახ. 10.4-ზე მოცემულია სასრული ავტომატის მარტივი მაგალითი, რომელსაც აქვს ორი მდგომარეობა – $Q=\{0,1\}$. საწყისი მდგომარეობაა $q_0=0$ და შემავალი ალფავიტი $\Sigma=\{a,b\}$. (ა) ნახაზზე ცხრილში მოცემულია δ გადასვლის ფუნქციის მნიშვნელობები, ხოლო (ბ) ნახაზზე იგივე ფუნქცია დიაგრამის სახით. მდგომარეობა 1 (დიაგრამაზე შავადაა მოცემული) ერთადერთი დასაშვები მდგომარეობაა. გადასვლები ნაჩვენებია ისრებით. მაგალითად b სიმბოლოთი მონიშნული ისარი 1 მდგომარეობიდან 0 მდგომარეობაში ნიშნავს, რომ $\delta(1,b)=0$. ამ ავტომატისათვის დასაშვებია სტრიქონრბი, რომრლიც მთავრდება კენტი რაოდენობის a სიმბოლოთი (უფრო ზუსტად დასაშვებია ya^k ტიპის სტრიქონები, სადაც y ცარიელი სტრიქონია ან მთავრდება b-ზე, ხოლო k კენტია). მაგალითად $abaaa$ სტრიქონისათვის მდგომარეობათა თანმიმდევრობა საწყისის ჩათვლით იქნება $\langle 0,1,0,1,0,1 \rangle$ და ეს სტრიქონი დასაშვებია, ხოლო $abbaa$ სტრიქონისათვის მდგომარეობათა თანმიმდევრობა წარმოადგენს $\langle 0,1,0,0,1,0 \rangle$ და ეს სტრიქონი უარყოფილი იქნება.

M სასრულ ავტომატთან დაკავშირებულია **საბოლოო მდგომარეობის ფუნქცია** (final-state function) $\varphi: \Sigma^* \to Q$, რომელიც შემდეგნაირად განისაზღვრება: $\varphi(w)$ არის მდგომარეობა, რომელშიც მივა ავტომატი w სტრიქონის წაკითხვის შემდეგ. ავტომატისათვის w სტრიქონი დასაშვებია მაშინ და მხოლოდ მაშინ, როცა $\varphi(w) \in A$. φ ფუნქცია შეიძლება განვსაზღვროთ რეკურენტულად:

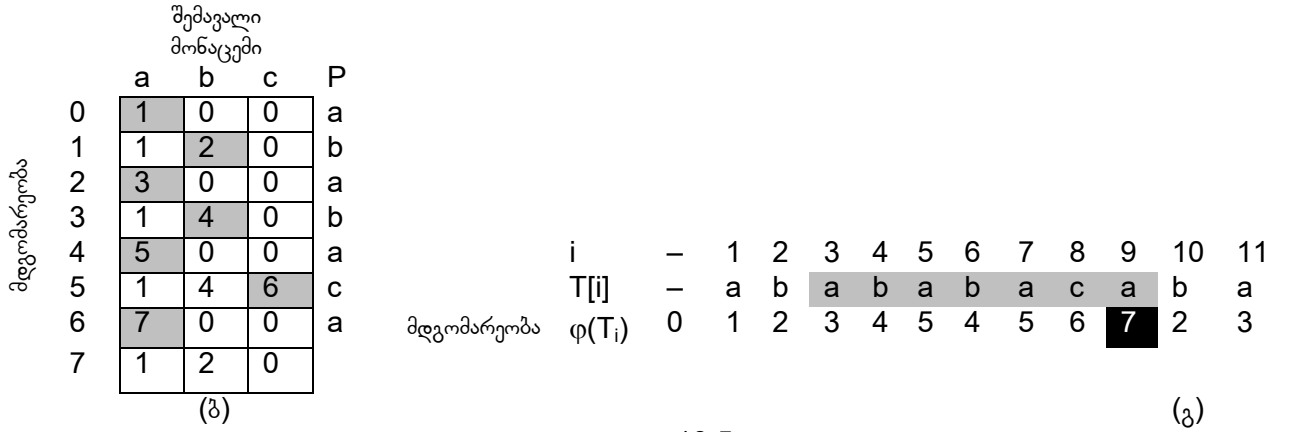
$$\varphi(\varepsilon)=q_0;$$

$$\varphi(wa)=\delta(\varphi(w),a)$$ ნებისმიერი $w \in \Sigma^*$ და $a \in \Sigma$-სათვის

ნებისმიერი P ნიმუშისათვის შეგვიძლია ავაგოთ ტექსტში ამ ნიმუშის მძებნელი სასრული ავტომატი. ნახ.10.5-ზე ნაჩვენებია ავტომატი, რომელიც ტექსტში ეძებს $P=ababaca$ ნიმუშს.



(ა)



ნახ. 10.5.

(ა)-ზე ნაჩვენებია გადასვლათა დიაგრამა P=abababaca ნიმუშისათვის. 0 – საწყისი მდგომარეობაა, 7 – ერთადერთი დასაშვები მნიშვნელობა. თუ a სიმბოლოთი მონიშნული ისარი მიდის i-დან j-ში, ეს ნიშნავს, რომ δ(a,i)=j. მარცხნიდან მარჯვნივ მიმართული მსხვილი ისრები შეესაბამებიან ნიმუშის ძებნის წარმატებულ ეტაპებს. მარჯვნიდან მარცხნივ მიმართული ისრები შეესაბამებიან წარუმატებელ ცდებს (ტექსტის უკანასკნელი j სიმბოლო დაემთხვა ნიმუშის პირველ j სიმბოლოს, მაგრამ მომდევნო სიმბოლო განსხვავებულია). მარჯვნიდან მარცხნივ მიმართული ისრებიდან ნახაზზე ყველა არაა ნაჩვენები. (ბ) ნახაზზე მოცემულია გადასვლათა ცხრილი, სადაც რუხი ფერის უჯრედები შეესაბამებიან წარმატებულ ცდებს (მსხვილ ისრებს დიაგრამაზე), ხოლო (გ) ნახაზზე ნაჩვენებია ავტომატის გამოყენება T=abababacaba ტექსტისათვის. მისი თითოეული სიმბოლოს ქვეშ ჩაწერილია ავტომატის მდგომარეობა შესაბამისი სიმბოლოს წაკითხვის შემდეგ. ნიმუში ტექსტში ნაპოვნია მხოლოდ ერთხელ – მესამე პოზიციიდან.

საზოგადოდ, P[1..m] ნიმუშის შესაბამისი ავტომატის აგებისას პირველი ბიჯია σ:Σ*→{0,1,...,m} დამხმარე ფუნქციის აგება, რომელსაც **სუფიქს-ფუნქციას** (suffix function) უწოდებენ. იგი x სტრიქონის შესაბამისად გამოითვლის x-ის მაქსიმალურ სუფიქსს, რომელიც P-ს პრეფიქსია:

$$\sigma(x)=\max\{k:P_k \sqsupset x\}$$

რადგან $P_0=\varepsilon$ წარმოადგენს ნებისმიერი სტრიქონის სუფიქსს, σ განსაზღვრულია მთელს Σ*-ზე. მაგ.: თუ P=ab, მაშინ σ(ε)=0, σ(ccaca)=1, σ(ccab)=2. თუკი P-ს სიგრძე ტოლია m-ის, მაშინ σ(x)=m, მაშინ და მხოლოდ მაშინ, როცა P წარმოადგენს x-ის სუფიქსს. თუ $x \sqsupset y$, მაშინ $\sigma(x) \le \sigma(y)$.

P[1..m] ნიმუშის შესაბამისი სასრული ავტომატი განვსაზღვროთ შემდეგნაირად:

- მდგომარეობათა სიმრავლეა – Q={0,1,…,m}. საწყისი მდგომარეობაა – $q_0=0$. ერთადერთი დასაშვები მნიშვნელობაა – m.
- გადასვლათა δ ფუნქცია განსაზღვრულია ფორმულით (q – მდგომარეობაა, $a \in \Sigma$ – სიმბოლო):

$$\delta(q,a)=\sigma(P_q a)$$

მაგალითად ნახ 10.5-ზე ნაჩვენებ ავტომატში δ(5,b)=4. თუ q=5, სტრიქონის წაკითხული ნაწილი მთავრდება ababa-თი და მორიგი შემომავალი სიმბოლოს წაკითხვისას წაკითხული ნაწილის უდიდესი სუფიქსი, რომელიც ამავე დროს P-ს პრეფიქსს წარმოადგენს, არის abab.

ჩავწეროთ პროგრამად სასრული ავტომატის მოქმედება, რომელიც ეძებს m სიგრძის მქონე P ქვესტრიქონს მოცემულ T ტექსტში (δ აღნიშნავს გადასვლათა ფუნქციას):

```
FINITE-AUTOMATION-MATCHER(T,δ,m)
1 n=length[T]
2 q=0
3 for i=1 to n do {
4                 q=δ(q, T[i])
5                 if q=m then  { s=i-m
6                                 print "ნიმუშის ტექსტში შესვლის წანაცვლებაა" s  }        }
```

რადგან პროგრამა დაამუშავებს T ტექსტის ყოველ სიმბოლოს, მისი მუშაობის დრო O(n) გამოდის, მაგრამ აქ უნდა გავითვალისწინოთ გადასვლათა δ ფუნქციის გამოთვლის დროც, რომლის პროგრამული კოდია:

```
COMPUTE-TRANSITION-FUNCTION (P,Σ)
1 m=length[P]
2 for q=0 to m do {
3         for ყველა a∈Σ სიმბოლოსათვის do {
4                 k=min(m+1, q+2)
5                 repeat {  k=k-1  }
6                 until Pk⊐Pqa
```

7 δ(q,a)=k } }

8 return δ

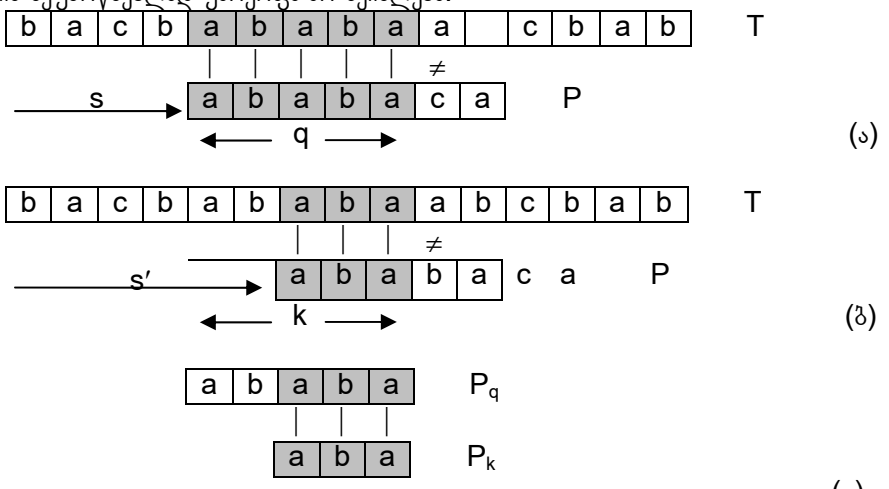
ეს პროცედურა გამოითვლის δ ფუნქციას სრული გადარჩევით. მე-2 და მე-3 სტრიქონებიდან დაწყებული ციკლები განიხილავენ (q,a) წყვილების ყველა შესაძლო ვარიანტს. ასევე ყველა ვარიანტი გადაირჩევა 4-7 სტრიქონებში თითოეული (q,a) წყვილისათვის შესაბამისი k-ს პოვნისათვის. გადარჩევა იწყება k-ს აპრიორულად ყველაზე დიდი მნიშვნელობიდან – min(m,q+1). ალგორითმის მუშაობის დროა $O(m^3|\Sigma|)$, თუმცა ის პრაქტიკაში უფრო სწრაფად მუშაობს და შეგვიძლია ვთქვათ, რომ m სიგრძის ნიმუშის ძებნის დრო n სიგრძის ტექსტში სასრული ავტომატის საშუალებით წარმოადგენს $O(n+ m|\Sigma|)$.

10.4. კნუტ-მორის-პრატის ალგორითმი

ქვესტრიქონის ძებნის ალგორითმი, რომელიც შემოგვთავაზეს კნუტმა, მორისმა და პრატმა, წრფივ დროში მუშაობს – $\Theta(m+n)$. ასეთი სისწრაფე განპირობებულია იმით, რომ წინასწარ გამოითვლება გადასვლათა ფუნქცია $\delta[0..m,1..|\Sigma|]$ და $|\Sigma|$-ჯერ მცირე მასივი – "პრეფიქს-ფუნქცია" $\pi[1..m]$, რომელიც გამოითვლება $O(m)$ დროში.

P ნიმუშთან ასოცირებული პრეფიქს-ფუნქცია ატარებს ინფორმაციას იმის შესახებ, თუ P სტრიქონში განმეორებით სად გვხვდება ამავე სტრიქონის სხვადასხვა პრეფიქსები. ამ ინფორმაციის გამოყენება საშუალებას გვაძლევს თავიდან ავიცილოთ უეჭველად დაუშვებელი წანაცვლებები (უმარტივესი ალგორითმის ტერმინებით) ან არ დაგვჭირდეს გადასვლათა ფუნქციის წინასწარი გამოთვლები (სასრული ავტომატების ტერმინებით).

პრეფიქს-ფუნქციის გამოყენებამდე მივყავართ შემდეგ გარემოებებს. ვთქვათ, უმარტივესი ალგორითმი T ტექსტში ეძებს P=ababaca ნიმუშს. დავუშვათ, რომელიღაც s წანაცვლებისათვის აღმოჩნდა, რომ ნიმუშის პირველია q სიმბოლო დაემთხვა ტექსტის სიმბოლოებს, ხოლო მომდევნო სიმბოლო განსხვავებულია (იხ. ნახ. 10.6, როცა q=5). შეგვიძლია ვთქვათ, რომ ჩვენ ვიცით ტექსტის q სიმბოლო T[s+1]-დან T[s+q]-მდე. ამ ინფორმაციის საფუძველზე შეგვიძლია დავასკვნათ, რომ ზოგიერთი წანაცვლება უეჭველად დაუშვებელია. ნახ. 10.6-დან ცხადად ჩანს, რომ დაუშვებელია წანაცვლება s+1, რადგან ამ წანაცვლების დროს ნიმუშის პირველი სიმბოლო a აღმოჩნდება ტექსტის (s+2)-ე სიმბოლოს ქვეშ, რომელიც ნიმუშის მეორე სიმბოლოს ემთხვევა – ეს სიმბოლოა b. ხოლო s+2 წანაცვლების დროს ნიმუშის პირველი სამი სიმბოლო დაემთხვევა ტექსტიდან ჩვენთვის ცნობილ ბოლო სამ სიმბოლოს, ამიტომ ამ წანაცვლების შეუმოწმებლად უარყოფა არ შეიძლება.



ნახ.10.6.

საზოგადოდ საკითხი შეიძლება ასე ჩამოვაყალიბოთ:

ვთქვათ, P[1..q]=T[s+1..s+q]. როგორია s′>s წანაცვლების უმცირესი მნიშვნელობა, რომლისთვისაც

$$P[1..k]=T[s'+1..s'+k] \quad (10.3), \quad \text{სადაც } s'+k=s+q$$

s′ წარმოადგენს წანაცვლების უმცირეს მნიშვნელობას, რომელიც აღემატება s-ს და ამასთან სრულდება ტოლობა T[s+1..s+q]=P[1..q]. ყველაზე კარგია, თუ s′=s+q. ამ შემთხვევაში ჩვენ შეგვიძლია არ განვიხილოთ ბიჯები s+1, s+2, ..., s+q-1. ნებისმიერ შემთხვევაში ახალი s′ წანაცვლების განხილვის დროს ჩვენ შეგვიძლია არ შევამოწმოთ ნიმუშის პირველი k სიმბოლო. (10.3) ფორმულიდან ჩანს, რომ ისინი უეჭველად ემთხვევიან ტექსტის შესაბამის სიმბოლოებს.

s′-ის საპოვნელად ჩვენ არ გვჭირდება რაიმეს ცოდნა ტექსტის შესახებ, საკმარისია ჩვენ ვიცოდეთ ნიმუში P და რიცხვი q. სახელდობრ, P_q სტრიქონის T[s′+1..s′+k] სუფიქსი. k რიცხვი (10.3) ფორმულაში წარმოადგენს ისეთ უდიდეს k<q რიცხვს, რომლისთვისაც P_k არის P_q-ის სუფიქსი. პრაქტიკულად მოსახერხებელია შევინახოთ ინფორმაცია სწორედ k რიცხვზე, რომელიც გვაძლევს იმ სიმბოლოთა რაოდენობას, რაც უეჭველად ემთხვევა ახალი s′ წანაცვლების შემოწმებისას. თავად s′-ის მნიშვნელობა გამოითვლება ფორმულით: s′=s+(q-k). ნახ.10.6(გ)-ზე ჩანს, რომ P

სტრიქონის უდიდესი პრეფიქსს, რომელიც ამავე დროს P_5-ის სუფიქსია (და განსხვავებულია P_5-საგან), აქვს სიგრძე 3. პრეფიქს-ფუნქციის ენაზე ეს ნიშნავს, რომ $\pi(5)=3$.

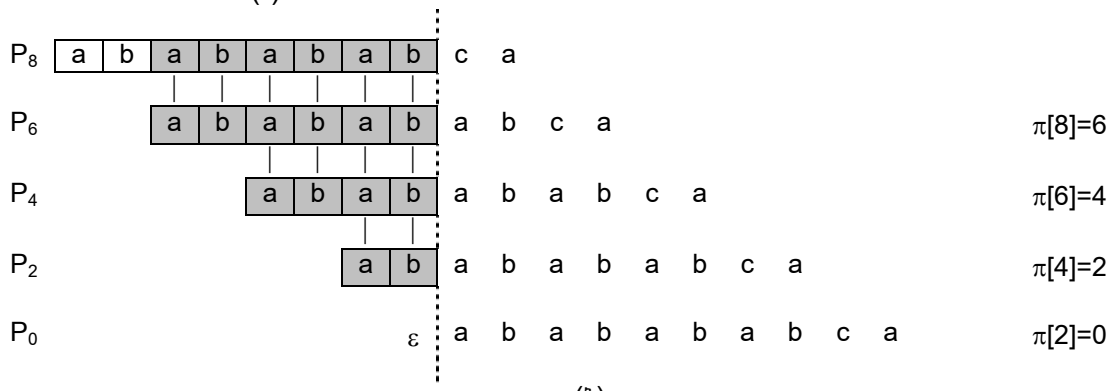
თუკი ფორმალურ სახეს მივცემთ, P[1..m] სტრიქონთან ასოცირებული **პრეფიქს-ფუნქცია** (prefix-function) ეწოდება ფუნქციას $\pi:\{1,2,...,m\}\to\{0,1,...,m-1\}$, რომელიც განსაზღვრულია შემდეგნაირად:

$$\pi[q]=\max\{k:k<q \text{ და } P_k \sqsupset P_q\}$$

სხვაგვარად რომ ვთქვათ $\pi[q]$ არის P-ს უდიდესი პრეფიქსი, რომელიც P_q-ს სუფიქსს წარმოადგენს. ნახ. 10.7(ა)-ზე მოყვანილია პრეფიქს-ფუნქცია ababababca სტრიქონისათვის.

i	1	2	3	4	5	6	7	8	9	10
P[i]	a	b	a	b	a	b	a	b	c	a
π[i]	0	0	1	2	3	4	5	6	0	1

(ა)



(ბ)

ნახ. 10.7.

კნუტ-მორის-პრატის ალგორითმი – KMP-MATCHER შეიძლება განვიხილოთ როგორც სასრული ავტომატის ალგორითმის (FINITE-AUTOMATION-MATCHER) გაუმჯობესება. მასში გამოყენებულია პროცედურა COMPUTE-PREFIX-FUNCTION, რომელიც გამოითვლის π პრეფიქს-ფუნქციას.

```
KMP-MATCHER (T,P)
1 n=lenght[T]
2 m=lenght[P]
3 π=COMPUTE-PREFIX-FUNCTION(P)
4 q=0
5 for i=1 to n do {
6       while q>0 and P[q+1]≠T[i] do {
7                       q=π [q]   }
8       if P[q+1]=T[i] then { q=q+1 }
9       if q=m then { print "ნიმუშის შესვლის წანაცვლებაა" i-m
10                      q=π[q] }          }
COMPUTE-PREFIX-FUNCTION (P)
1 m=lenght[P]
2 π[1]=0
3 k=0
4 for q=2 to m do {
5       while k>0 and P[k+1]≠P[q] do {
6               k=π[k]  }
7       if P[k+1]=P[q] then { k=k+1 }
8       π[q]=k
9 return π
```

KMP-MATCHER პროცედურის მუშაობის დროა O(m+n), რაც უკეთესია სასრული ავტომატის ალგორითმის მუშაობის დროზე.

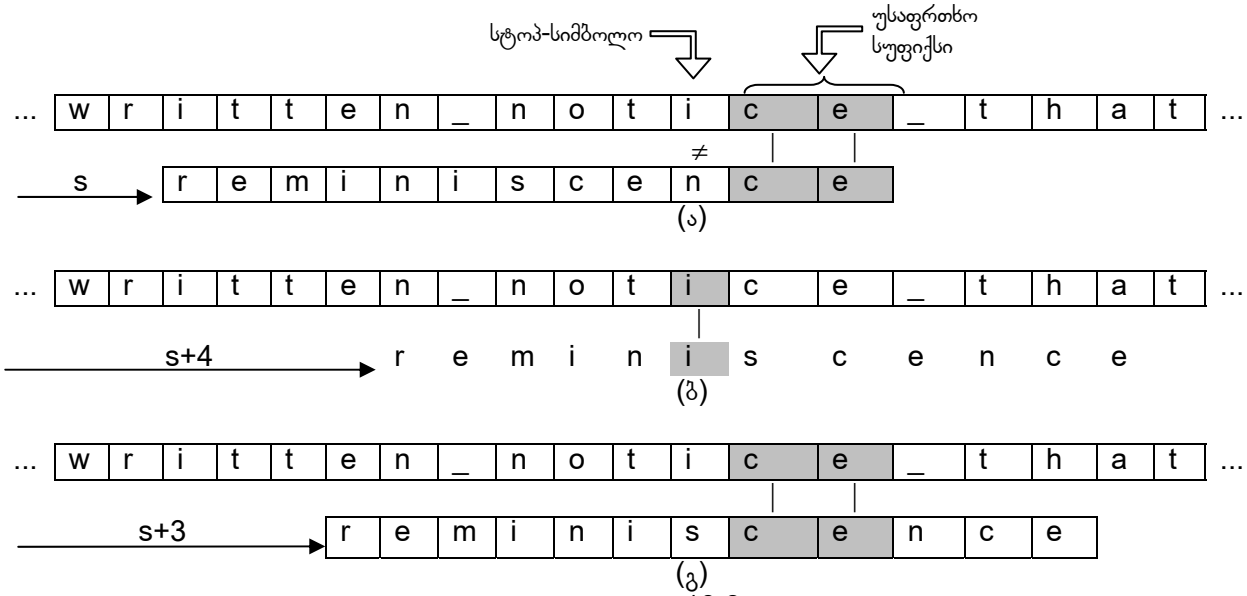
10.5. ბოიერ-მურის ალგორითმი

თუკი P ნიმუში გრძელია და Σ ალფავიტი საკმაოდ დიდი, მაშინ ქვესტრიქონის ძებნისათვის უფრო ეფექტურია ალგორითმი, რომელი შექმნეს ბოიერმა (Robert S. Boyer) და მურმა (J. Strother Moore):

```
BOYER-MOORE-MATCHER (T,P,Σ)
1 n=lenght[T]
```

```
2 m=lenght[P]
3 λ=COMPUTE-LAST-OCCURRENCE-FUNCTION (P,m,Σ)
4 γ=COMPUTE-GOOD-SUFFIX-FUNCTION (P,m)
5 S=0
6 while s≤n-m do {
7         j=m
8         while j>0 and P[j]=T[s+j] do {
9                 j=j-1      }
10                if j=0  then {  print "ნიმუშის შესვლის წანაცვლებაა" s
11                                s=s+γ[0]    }
12                        else {  s=s+max(γ[j], j-λ[T[s+j]])
```

თუ არ მივაქცევთ ყურადღებას λ-ს და γ-ს, მაშინ ეს ალგორითმი ძალიან წააგავს უმარტივეს ალგორითმს, მაგრამ ბოიერმა და მურმა უმარტივეს ალგორითმში შეიტანეს ორი გაუმჯობესება, რომლებსაც ეწოდება "სტოპ-სიმბოლოს ევრისტიკა" და "უსაფრთხო სუფიქსის ევრისტიკა". ეს ევრისტიკები საშუალებას გვაძლევენ არ განვიხილოთ ზოგიერთი წანაცვლებები (პრაქტიკაში საკმაოდ ბევრი). ორივე ევრისტიკა მოქმედებს დამოუკიდებლად და გამოიყენება ერთდროულად. თუ s წანაცვლების შემოწმებისას გაირკვევა, რომ ქვესტრიქონი T[s+1..s+m] არ ემთხვევა ნიმუშს, მაშინ ორივე ევრისტიკა იძლევა მნიშვნელობას, რომლითაც შეიძლება გავზარდოთ s ისე, რომ არ გამოვტოვოთ დასაშვები წანაცვლება (ეს მნიშვნელობებია: j-λ[T[s+j]] – სტოპ-სიმბოლოს ევრისტიკისათვის და γ[j] – უსაფრთხო სუფიქსის ევრისტიკისათვის). ბოიერ-მურის ირჩევს ამ ორი წანაცვლებიდან უდიდესს.



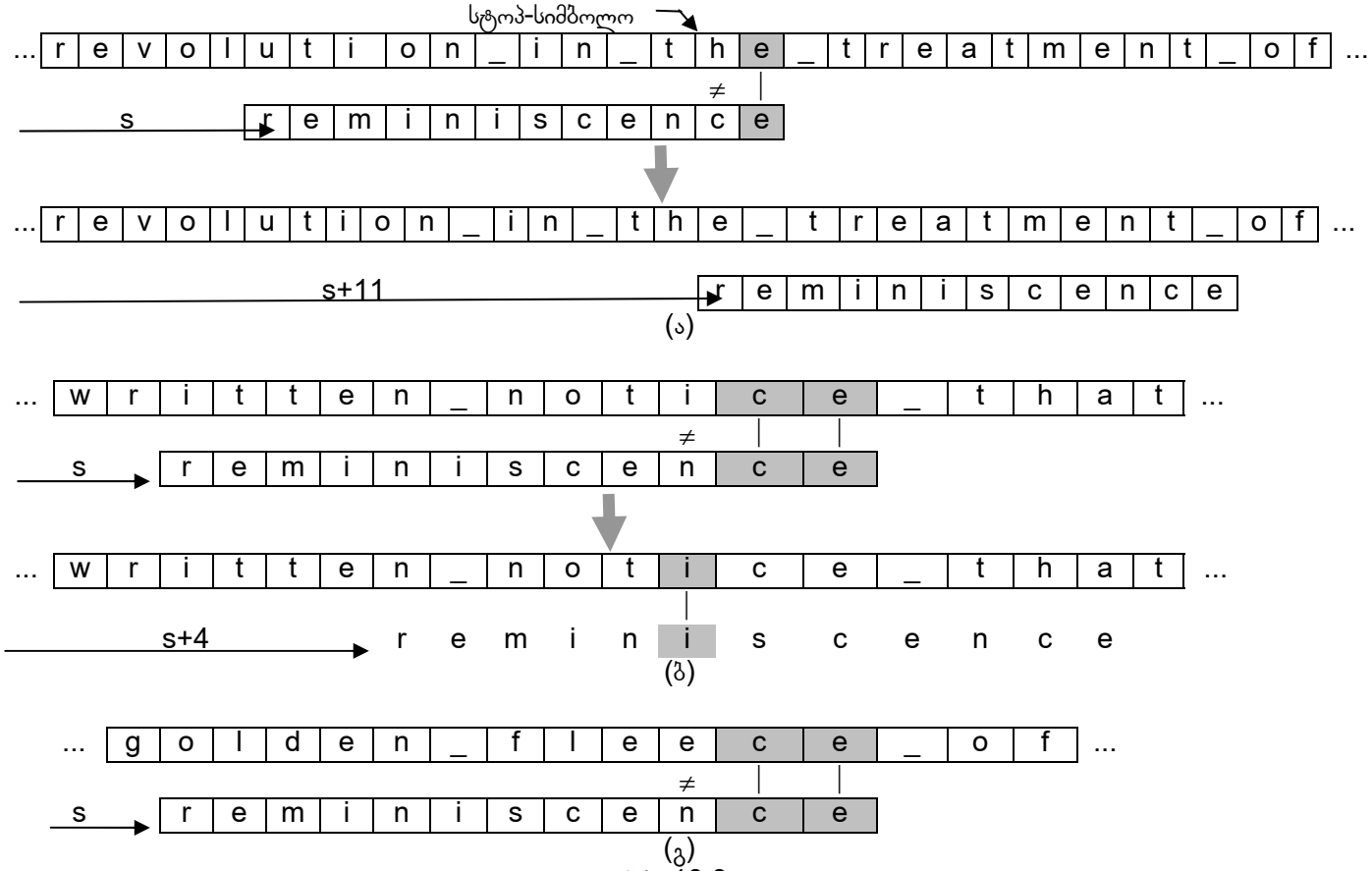
ნახ. 10.8.

ნახ. 10.8-ზე გამოსახულია ბოიერ-მურის ევრისტიკები. საძებნი ქვესტრიქონია reminiscence. (ა) ნახაზზე ნაჩვენებია სიტუაცია, როცა s წანაცვლების დროს ნიმუშის ტექსტთან შედარებისას დაემთხვა ორი განაპირა მარჯვენა სიმბოლო (ისინი წარმოადგენენ "უსაფრთხო სუფიქსს" ce), ხოლო მესამე სიმბოლო განსხვავებული აღმოჩნდა (ტექსტში ამ ადგილას დგას "სტოპ-სიმბოლო" – i და მასთან მისვლის შემდეგ შედარება წყდება). (ბ)-ზე მოცემულია სტოპ-სიმბოლოს ევრისტიკა, რომელიც გვთავაზობს ნიმუშის წანაცვლებას მარჯვნივ ისე, რომ სტოპ-სიმბოლო ტექსტში აღმოჩნდეს ამავე სიმბოლოს ყველაზე მარჯვენა პოზიციის პირდაპირ ნიმუშში. ჩვენ მაგალითზე ეს ნიშნავს წანაცვლებას 4 პოზიციით. თუკი ნიმუშში სტოპ-სიმბოლო საერთოდ არ აღმოჩნდება, მაშინ ნიმუში მთლიანად უნდა გადავანაცვლოთ ტექსტში სტოპ-სიმბოლოს იქით. თუკი სტოპ-სიმბოლო ნიმუშში უფრო მარჯვნივ აღმოჩნდება, ვიდრე სტოპ-სიმბოლო ტექსტში მაშინ ევრისტიკა არაფერს არ გვთავაზობს, რადგან მივიღებთ უარყოფით წანაცვლებას, რომელიც იგნორირებული იქნება ალგორითმის მიერ. (გ)-ზე უსაფრთხო სუფიქსის ევრისტიკა გვთავაზობს წავანაცვლოთ ნიმუში იმდენზე, რომ უსაფრთხო სუფიქსის უახლოესი განლაგება ნიმუშში (განიხილება მარჯვნიდან მარცხნივ) აღმოჩნდეს ტექსტში მყოფი უსაფრთხო სუფიქსის პირდაპირ. ჩვენს მაგალითში ეს ნიშნავს წანაცვლებას 3 პოზიციით. ბოიერ-მურის ალგორითმი ირჩევს ამ ორი წანაცვლებიდან უდიდესს – წანაცვლებას 4 პოზიციით.

ახლა უფრო დეტალურად განვიხილოთ ორივე ევრისტიკის გამოთვლის ალგორითმი, რომლებიც ძირითადი პროგრამის მე-3 და მე-4 სტრიქონებში უბრალოდ პროცედურათა სახელებითაა მოცემული.

სტოპ-სიმბოლოს ევრისტიკა (bad-character heuristic) მუშაობს შემდეგი პრინციპით. დავუშვათ, მარჯვნიდან მარცხნივ შედარებისას ჩვენ წავაწყდით პირველ განსხვავებულ სიმბოლოს: P[j]≠T[s+j], სადაც 1≤j≤m. ვთქვათ, k არის P ნიმუშში T[s+j] სიმბოლოს შესვლის ყველაზე მარჯვენა პოზიცია (თუკი სიმბოლო ნიმუშში საერთოდ არ გვხვდება, ითვლება რომ k=0). შეგვიძლია ვამტკიცოთ, რომ

s-ის გაზრდა (j-k)-თი არ გამოტოვებს არცერთ დასაშვებ სიმბოლოს. მართლაც, თუ k=0, მაშინ სტოპ-სიმბოლო T[s+j] საერთოდ არ გვხვდება ნიმუშში, ამიტომ შეგვიძლია ნიმუში გადავწიოთ j-k=j სიმბოლოთი მარჯვნივ (იხ. ნახ. 10.9(ა)); თუ 0<k<j, მაშინ ნიმუში შეგვიძლია გადავწიოთ j-k პოზიციით მარჯვნივ, რადგან უფრო ნაკლები წანაცვლებებისას ტექსტის სტოპ-სიმბოლო ნიმუშის არცერთ სიბოლოს არ დაემთხვევა (იხ. ნახ. 10.9(ბ)); ბოლოს, თუ k>j, მაშინ სტოპ-სიმბოლოს ევრისტიკა გვთავაზობს ნიმუში წავანაცვლოთ არა მარჯვნივ, არამედ – მარცხნივ, მაგრამ ბოიერ-მურის ალგორითმი ამ რეკომენდაციას იგნორირებას უკეთებს, რადგან უსაფრთხო სუფიქსის ევრისტიკა ყოველთვის გვთავაზობს ნულოვანზე მეტ წანაცვლებას მარჯვნივ.



ნახ. 10.9.

სტოპ-სიმბოლოს ევრისტიკის გამოსაყენებლად საჭიროა ყველა შესაძლო $a\in\Sigma$ სტოპ-სიმბოლოსათვის გამოვთვალოთ k-ს მნიშვნელობა. ამას აკეთებს მარტივი პროცედურა COMPUTE-LAST-OCCURRENCE-FUNTION, რომელიც თითოეული $a\in\Sigma$-სათვის გამოითვლის $\lambda[a]$-ს – a-ს ყველაზე მარჯვენა შესვლას P ნიმუშში, ან მიანიჭებს ნულს, თუ a არ შედის P-ში.

```
COMPUTE-LAST-OCCURRENCE-FUNTION (P, m, Σ)
1 for ყოველი a∈Σ სიმბოლოსათვის do {
2 λ[a]=0    }
3 for j=1 to m do   {
4 λ[P[j]]=j     }
5 return λ
```

პროცედურის მუშაობის დროა $O(|\Sigma|+m)$.

თუ Q და R სტრიქონებია, იტყვიან, რომ ისინი **შედარებადი** არიან (აღნიშვნა Q~R), თუკი ერთ-ერთი მათგანი წარმოადგენს მეორეს სუფიქსს. შედარებადობის დამოკიდებულება სიმეტრიულია: თუ Q~R, მაშინ ასევე R~Q. აგრეთვე ადგილი აქვს დებულებას:

თუ $Q\supset R$ და $S\supset R$, მაშინ Q~S (10.4)

უსაფრთხო სუფიქსის ევრისტიკა (good-suffix heuristic) მდგომარეობს შემდეგში: თუ $P[j]\neq T[s+j]$, სადაც $j<m$ და j რიცხვია უდიდესია ასეთი თვისებით, მაშინ ჩვენ შეგვიძლია გავზარდოთ წანაცვლება

$\gamma[j]=m-\max\{k: 0\le k<m$ და $P[j+1..m]\sim P_k\}$

სხვაგვარად რომ ვთქვათ, $\gamma[j]$ არის უმცირესი მანძილი, რომელზე ჩვენ შეგვიძლია გადავწიოთ ნიმუში ისე, რომ T[s+j+1..s+m] უსაფრთხო ნიმუშის არცერთი სიმბოლო არ აღმოჩნდეს მისგან განსხვავებული სიმბოლოს პირდაპირ.

რადგან P[j+1..m] სტრიქონი შედარებადია ცარიელ სტრიქონთან, ამიტომ $\gamma[j]$ კორექტულადაა განსაზღვრული ნებისმიერი j-სათვის. შევნიშნოთ აგრეთვე, რომ ნებისმიერი j-სათვის $\gamma[j]>0$, ამიტომ ბოიერ-მურის ალგორითმის ყოველი ბიჯი ნიმუშს წაანაცვლებს მარჯვნივ ერთი მაინც პოზიციით. γ-ს ვუწოდოთ **უსაფრთხო სუფიქსის ფუნქცია** (good-suffix function), ასოცირებული P სტრიქონთან.

γ-ის გამოსათვლელად შევნიშნოთ, რომ $P_{\pi[m]} \sqsupset P$, საიდანაც 10.4-ის თანახმად გვაქვს: $P[j+1..m] \sim P_{\pi[m]}$ ნებისმიერი j-სათვის. აქედან გამომდინარე, მაქსიმუმი $\gamma[j]$ სიდიდის განსაზღვრების მარჯვენა ნაწილში არაა ნაკლები $\pi[m]$-ზე. ასე, რომ $\gamma[j] \le m-\pi[m]$ ნებისმიერი j-სათვის. მაშინ γ-ის განსაზღვრება ასეც ჩაიწერება:

$$\gamma[j]=m-\max\{k: \pi[m]\le k<m \text{ და } P[j+1..m]\sim P_k\}$$

$P[j+1..m]\sim P_k$ პირობა შეიძლება შესრულდეს ორ შემთხვევაში: ან როცა $P[j+1..m] \sqsupset P_k$ ან $P_k \sqsupset P[j+1..m]$. მეორე შემთხვევაში გვექნება $P_k \sqsupset P_m$, საიდანაც $k\le\pi[m]$ და ამიტომ $k=\pi[m]$. ამიტომ γ-ის განსაზღვრება ასეთი სახითაც შეიძლება ჩავწეროთ:

$$\gamma[j]=m-\max(\{\pi[m]\}\cup \{k: \pi[m]\le k<m \text{ და } P[j+1..m]\sim P_k\})$$

სადაც ამ სიმრავლეებიდან მეორე შეიძლება აღმოჩნდეს ცარიელი. უფრო მარტივად რომ ვთქვათ, ჩვენ ვეძებთ P ნიმუშის P_k პრეფიქსს, რომელშიც P[j+1..m] სუფიქსს წარმოადგენს. ან სხვა სიტყვებით რომ ვთქვათ, ჩვენ ვეძებთ ნიმუშში მონაკვეთს, რომელიც ტოლია P[j+1..m] სუფიქსის და განლაგებულია უფრო მარცხნივ (k<m).

ჩვენ უნდა ვიპოვოთ ყველაზე მარჯვენა ასეთი მონაკვეთებიდან (k რიცხვები, რომლებიდანაც ვიღებთ მაქსიმალურს – ესაა ასეთი მონაკვეთების მარჯვენა საზღვრები). ამისათვის მოსახერხებელია განვიხილოთ P-ს შებრუნებული P' სტრიქონი და მისი შესაბამისი π' პრეფიქს-ფუნქცია.

ვთქვათ P სტრიქონის X მონაკვეთი მთავრდება k პოზიციაში (ანუ წარმოადგენს P_k სტრიქონის სუფიქსს) და ტოლია P[j+1..m]. ორივე თანაბარი მონაკვეთების სიგრძეები ტოლია m-j და ცხადია, რომ ეს მონაკვეთები (შებრუნებული სახით) იქნებიან სუფიქსი და პრეფიქსი P_l' მონაკვეთისათვის, სადაც $l=(m-k)+(m-j)$ (აქ (m-k) წარმოადგენს მანძილს X მონაკვეთის ბოლოდან ნიმუშის ბოლომდე, ხოლო (m-j) – მონაკვრთის სიგრძეა). ამიტომ $\pi'[l]\ge m-j$. დავამტკიცოთ, რომ სინამდვილეში ადგილი აქვს ტოლობას

$$\pi'[l]=m-j \qquad (10.5)$$

მართლაც, თუ X-ის საწყისი პოზიციიდან დაწყებული უფრო გრძელი X′ მონაკვეთი იქნებოდა ნიმუშის სუფიქსი, მაშინ რომელიღაც მისი სუფიქსი დაემთხვეოდა P[j+1..m]-ს და X არ იქნებოდა P სტრიქონის ყველაზე მარჯვენა მონაკვეთი, რომელიც P[j+1..m]-ს ემთხვევა.

10.5 ფორმულიდან $j= m-\pi'[l]$. გარდა ამისა, მისგან გამომდინარეობს, რომ $k=m-l+\pi'[l]$. ამიტომ $\gamma[j]$-ის გამოსათვლელი ფორმულა საბოლოო სახით შეიძლება ასე ჩაიწეროს:

$$\gamma[j]= m-\max(\{\pi[m]\}\cup \{ m-l+\pi'[l]: 1\le l\le m \text{ და } j= m-\pi'[l]\})=$$
$$=\min(\{m-\pi[m]\}\cup \{ l-\pi'[l]: 1\le l\le m \text{ და } j= m-\pi'[l]\}) \qquad (10.6)$$

ახლა ჩავწეროთ γ-ს გამოსათვლელი პროგრამული კოდი:

```
COMPUTE-GOOD-SUFFIX-FUNCTION (P,m)
1 π=COMPUTE-PREFIX-FUNCTION(P)
2 P′=P სტრიქონის შებრუნებულს
3 π′=COMPUTE-PREFIX-FUNCTION(P′)
4 for j=0 to m do {
5 γ[j]=m-π[m]      }
6 for l=1 to m do {
7 j=m-π′[l]
8 if γ[j]>l-π′[l]  then {  γ[j]=l-π′[l]  }      }
9 return γ
```

პროცედურის მუშაობის დროა O(m).

ბოიერ-მურის ალგორითმის მუშაობის დრო უარეს შემთხვევაში წარმოადგენს $O((n-m+1)m+|\Sigma|)$, თუმცა პრაქტიკაში ის ხშირადაა სხვა ალგორითმებზე უფრო ეფექტური.

10.1. ტექსტი
(კიროვის XV ღია ოლიმპიადა, 2003 წელი თებერვალი)

მოცემულია ტექსტი, რომელიც სედგება სიტყვების, სასვენი ნიშნებისა და სხვა სიმბოლოებისაგან. სიტყვად ტექსტში ითვლება ლათინური ანბანის მთავრული და მხედრული (პატარა) სიმბოლოებისაგან შედგენილი მიმდევრობა. საჭიროა შევატრიალოთ (დავწეროთ შებრუნებული მიმდევრობით) ტექსტის ყველა სიტყვა, ხოლო სასვენი ნიშნები და

ყველა სხვა სიმბოლო დავტოვოთ უცვლელად. სტრიქონი შედგება არაუმეტეს 255 სიმბოლოსაგან, სტრიქონების რაოდენობა ფაილში 1000-ზე მეტი არ არის.

მაგალითი

Input.txt	Output.txt
This is an example. Prim333primяяrt	sihT si na elpmaxe. mirP333mirpяятr

მითითება. ამოცანის ამოსახსნელად რაიმე სპეციალური ალგორითმი საჭირო არ არის – უბრალოდ უნდა წავიკითხოთ ლათინური ანბანის სიმბოლოთა მიმდევრობა და თან შევაბრუნოთ, ხოლო როცა განსხვავებული სიმბოლო შეგვხვდება, შებრუნებული მიმდევრობა დავბეჭდოთ და გავაგრძელოთ მოძრაობა ტექსტში.

```
s=' '
WHILE (ვიდრე შემომავალი ფაილი არ დასრულდება) {
        READ(c)
        IF c∈('A'..'Z') OR c∈('a'..'z') THEN  {  s=c+s  }
                                        ELSE  { WRITE(s); s=' ';  WRITE(c)  }        }
WRITE(s)
```

c ცვლადის საშუალებით თითო-თითო სიმბოლოდ წავიკითხოთ მთელი ფაილი. თუ სიმბოლო ლათინურია, იგი მარცხნიდან მივუმატოთ რაიმე s ცვლადს, ხოლო წინააღმდეგ შემთხვევაში დავბეჭდოთ s, რომელიც უკვე შებრუნებულია, დავბეჭდვის შემდეგ გავანულოთ და დავბეჭდოთ c სიმბოლოს ის მნიშვნელობაც, რომელიც არ წარმოადგენს ლათინური ანბანის სიმბოლოს. ციკლის დასრულების შემდეგ საჭიროა კიდევ ერთი ბეჭდვის განხორციელება, რადგან თუ ტექსტი ლათინური სიმბოლოებით მთავრდება, მისი დაბეჭდვა ციკლში ვერ მოხდებოდა, ხოლო თუ ბოლო სიმბოლო არაა ლათინური ანბანიდან – s ცარიელი სტრიქონი იქნება.

10.2. განმეორება

(პოლონეთის მოსწავლეთა ოლიმპიადა, 1999-2000 წელი)

მოცემულია ლათინური ალფავიტის პატარა სიმბოლოებისაგან ('a'..'z') შემდგარი სიტყვები. იპოვეთ სიმბოლოთა ყველაზე გრძელი მიმდევრობა, რომელიც საერთოა ყველა სიტყვისათვის.

დაწერეთ პროგრამა, რომელიც:

- წაიკითხავს სიტყვათა მიმდევრობას POW.IN ფაილიდან;
- იპოვის ამ სიტყვებისათვის საერთო ყველაზე გრძელ ფრაგმენტს;
- ჩაწერს შედეგს ტექსტურ POW.OUT ფაილში.

შემავალი მონაცემები

შემავალი POW.IN ფაილის პირველი სტრიქონი შეიცავს სიტყვათა რაოდენობის აღმნიშვნელ n რიცხვს, $1 \le n \le 5$. მომდევნო n სტრიქონიდან თითოეული შეიცავს ლათინური ალფავიტის პატარა სიმბოლოებისაგან ('a'..'z') შემდგარი სიტყვებს. თითოეული სიტყვის სიგრძე არის არანაკლებ 1 და არაუმეტეს 2000 სიმბოლოსი.

გამომავალი მონაცემები

გამომავალი POW.OUT ფაილის ერთადერთ სტრიქონი უნდა შეიცავდეს ერთ მთელ რიცხვს – მოცემული n რაოდენობის სიტყვებში უდიდესი საერთო ფრაგმენტის სიგრძე.

მაგალითი

შემავალი მონაცემები (POW.IN ფაილი)	გამომავალი მონაცემები (POW.OUT ფაილი)
3 abcb bca acbc	2

გამოყენებული ლიტერატურა

გ. მანდარია, ბ. შიოშვილი, ბ. პერტახია. ამოცანები ინფორმატიკაში. გამომცემლობა “ლამპარი”, თბილისი, 2000 წ.

მოსწავლეთა საერთაშორისო ოლიმპიადების მასალები (2003 წ. სამხრეთ კორეა, 2004 წ. აშშ).

Томас Кормен, Чарльз Лейзерсон, Рональд Ривест. Алгоритмы: построение и анализ. МЦНМО, Москва, 2001.

Дональд Эрвин Кнут. Искусство программирования. Издательский дом «Вильямс», Москва, 2000.

А. Шень. Программирование: теоремы и задачи.

А. Ахо, Дж. Хопкрофт, Дж. Ульман. Построение и анализ вычислительных алгоритмов. М.: Мир, 1979.

Ричард Беллман. Динамическое программирование. М.: ИЛ, 1960.

წიგნში გამოყენებულია აგრეთვე მასალები შემდეგი საიტებიდან:

ace.delos.com
olympiads.win.tue.nl/ioi/
www.informatics.ru
www.oi.edu.pl
olympiads.ru
g6prog.narod.ru
neerc.ifmo.ru

საგნობრივი საძიებელი

მოსწავლეთა საერთაშორისო ოლიმპიადები – IOI

თამაში (უნგრეთი, 1996) –16, მძივები (1993 წ. არგენტინა) – 46, მავნე ბაყაყი (სამხრეთ კორეა, 2002) – 52, ავტობუსების გაჩერებები (სამხრეთ კორეა, 2002) – 54, ორი ბილიკი (სამხრეთ კორეა, 2002) – 74, სამკუთხედი (სტოკჰოლმი, 1994წ.) – 96, ყვავილების მაღაზია (ანტალია, 1999 წ.) – 98, გზების შერჩევა (აშშ, 2003) –142, მესერი (აშშ, 2003) – 166.

საქართველოს მოსწავლეთა ოლიმპიადები:

გამჭვირვალე სინჯარები (რესპუბლ., 1997) – 45, ლათინური კვადრატი (ზონური, 1996) – 61, ვორდულატორი (ზონური, 1996) – 91, კედლის კარადა (ზონური, 2002) – 99, კონტურის განთავსება (ზონური ტური, 2001) – 139, მოგზაურობა (რესპუბლ. 2002) – 143, სიტყვებით თამაში (ზონური, 2002) – 144, დაფა (რესპუბლ., 2003) – 154.

USACO

ძროხათა სადგომი (2000, ზამთ. "მწვანე") – 10, ალმასისმაგვარი სიტყვები (2001 ზამთ. "ნარინჯ.") – 20, პარკსიდის სამკუთხედი (2001, ნოემბერი, "ნარინჯ.") – 21, დროის გადამყვანი (2002, ნოემბ., "ნარინჯ.") – 21, კროსვორდი (2002, ნოემბ., "ნარინჯ.") – 22, თავსატეხი სიტყვები (2002, დეკემბ., "ნარინჯ.") – 22, სტენოგრაფია (2002, ნოემბ., "ნარინჯ.") – 23, ფიბონაჩის რიცხვები (2002, დეკემბ., "ნარინჯ.") – 24, ვერტიკალური ჰისტოგრამა (2003, თებერ., "ნარინჯ.") – 25, ძროხების ტრანსმიგრაცია (2003, აპრ., "ნარინჯ.") – 25, დაკარგული ძროხები (2003, აპრ., "ნარინჯ.") – 25, საძოვარი მინდვრები (2002, ზამთ., "მწვანე") – 26, არითმეტიკული პროგრესია (2004, იანვ., "მწვანე") – 48, მხიარული ვაჭრობა (2002, დეკემბ., "ნარინჯ.") – 51, Sorta სორტირება (2000, შემოდგ., "ნარინჯ.") – 71, ქოლგები (2003, დეკემბ., "მწვანე") – 72, განძის ძიება ("მწვანე", 2000, ზამთ.) – 107, გზების მშენებლობა (2003, შემოდგ., "ნარინჯ.") –138, რძის საიდუმლო არხები (2002, "მწვანე") –141, ტყვეები ომიდან (2002, დეკემბ., "მწვანე") –164.

რუსეთის ოლიმპიადები:

კოდის კორექცია (უნივ. პირველობა, სამარა) –15, ლურსმნები (რაიონული, 1997) –15, ტექსტის შესწორება (უნივ. პირველობა, სამარა) – 17, ივანოვები და პეტროვები (დასკვნითი, 1990წ.) –18, სატვირთო მატარებელი (დასკვნითი, 1991) –18, ახალწვეულთა მწკრივი (დასკვნითი, 1991) – 18, ვირუსები (სტუდ. და მოსწ. გუნდური პირველობა, ყაზანი, 2000) – 19, წრე მართკუთხედებისაგან (დასკვნითი, 1998-99) – 72, ჩაინვორდი (დასკვნითი, 2002-03) – 143, კუბიკები (რუსეთის გუნდური ოლიმპიადა, მოსწ., 2000) – 153, მინა (დასკვნითი ტური, 2002-03) – 165.

ბალკანეთის ოლიმპიადა ინფორმატიკაში მოსწავლეთა შორის:

თევზის ბაზარი (2000) – 107, თევზები(2000) – 115.

ცენტრალური ევროპის ოლიმპიადები ინფორმატიკაში:

კომპანია Bugs Integrated (2002, სლოვაკეთი) –110.

ბალტიისპირეთის ქვეყნების ოლიმპიადა ინფორმატიკაში მოსწავლეთა შორის:

წილადებიანი გამოსახულება (2000) – 27, რიცხვითი ცხრილი (2000) – 98, მრგვალი ფრჩხილები (1999) – 101,

პოლონეთის მოსწავლეთა ოლიმპიადები ინფორმატიკაში:

ინტერვალები (პირველი ეტაპი, 2000-01) – 47, განმეორება (მესამე ეტაპი, 1999-2000) – 178.

სტუდენტური ოლიმპიადები:

პალინდრომი (ამიერკავკ. პირველობა, თბილისი, 2002) – 7, მარტივი რიცხვები ინტერვალიდან (საქ. პირველობა, თბილისი, 2003) – 82, ნაკეთობები (საქ. პირველობა, თბილისი, 2002) –140, არჩევნები (ამიერკავკ. პირველობა, თბილისი, 2003) – 153.

უკრაინის online-ოლიმპიადები:

კორსარი (2002-03) – 163, ნაღმები (2002-03) – 160, პალინდრომი (2002-03) – 100.

ჩიგირინსკების ოლიმპიადა, 2003-04 წლები

დომინო –19, გვირაბი –19, წყურვილის მოკვლა – 83, კვარცხლბეკი – 109.

სხვადასხვა:

რევერსი – 7, წილადის პერიოდი – 9, კალენდარი – 9, დიდი რიცხვები –12, სპირალი – 14, ჰანოის კოშკი – 57, მხედრით დაფის შემოვლა – 58, ლაზიერები – 60, არჩევნები – 71, მაქსიმალური ჯამის შემცველი მართკუთხედი – 84, მატრიცათა მიმდევრობის გადამრავლების – 86, მრავალკუთხედის ოპტიმალური ტრიანგულაცია – 93, კიბე – 95, ზრდადი ქვემიმდევრობა – 91, ზურგჩანთის ამოცანა –103.