ნ. ბენიძე

შესავალი ალგორითმების თეორიაში

ლექციათა კურსი

2022

I ლექცია: ალგორითმის ინტუიციური განმარტება. ალგორითმის დროითი შეფასება.

შესავალი

დღევანდელი სამყარო წარმოუდგენელია ელექტრული გამომთვლელი მანქანების - კომპიუტერების გარეშე. კომპიუტერებმა უკვე დიდი ხანია დატოვეს სამეცნიერო-კვლევითი ლაბორატორიები/დაწესებულებები და ყოველდღიურობის განუყოფელ ნაწილად იქცნენ,
თუმცა კომპიუტერულმა ტექნიკამ გარკვეული ევოლუცია განიცადა ვიდრე დღევანდელ დღემდე მოვიდოდა.

კომპიუტერების კლასიფიკაცია თაობების მიხედვით - თაობების მიხედვით დაყოფა ძირითადად ელექტრონული რეალიზაციის მიხედვით ხდებოდა.
I თაობა- მე-20 საუკუნის 40-იანი წლები 1939-1951. I თაობის მანქანები ელექტრონულ ლამპებზე იყო აგებული და სწორედ ეს ფაქტი განაპირობებდა მათ შთამბეჭდავ მასშტაბებს.
II თაობა- მე-20 საუკუნის 50-იანი წლები და 60-იანი წლების დასაწყისი. ამ პერიოდში გამოიგონეს ნახევარგამტარები და II თაობის მანქანების ელექტრონულ საფუძველს წარმოადგენდა ნახევარგამტარები.
III თაობა- მე-20 საუკუნის 60-იან წლებში ელექტრონიკაში ინტეგრალური სქემების ერა დადგა. III თაობის მანქანები ინტეგრალურ სქემებზე იყო აგებული.
IV თაობა - 1969 წელს კომპანია Intel-ის თანამშრომელმა თედ ჰოფმა ერთ კრისტალზე ცენტრალური პროცესორის შექმნის იდეა გაახმოვანა - ეს ნიშნავდა, რომ ბევრი ინტეგრალური მიკროსქემების ნაცვლად შეექმნათ ერთი მთავარი ინტეგრალური სქემა, რომელიც შეასრულებდა მანქანურ კოდში ჩაწერილ ყველა არითმეტიკულ, ლოგიკურ და მართვის გადაცემის ოპერაციებს. ამ მოწყობილობას მიკროპროცესორი ეწოდა, 1971 წელს კომპანია Intel-მა გამოუშვა პირველი მიკროპროცესორი „Intel 4004“. მიკროპროცესორების შექმნამ საფუძველი დაუდო მიკროკომპიუტერების ერას - ეს იყო პატარა, არც თუ ძალიან ძვირი კომპიუტერები და მათი შეძენა უკვე შეეძლოთ პატარა კომპანიებს და ცალკეულ ფიზიკურ პირებს.

1977 წელს კორპორაცია Apple Computer იწყებს პირველი მასობრივი პერსონალური კომპიუტერების წარმოებას.

1981წლის აგვისტოში ფირმა IBM-მა შექმნა კომპიუტერული სისტემა IBM PC.

1984 წელს ბაზარზე გამოჩნდა Apple Macintosh, რომელიც პირველი მასობრივი წარმოების პერსონალური კომპიუტერები იყო გრაფიკული ინტერფეისით.
V თაობა - ალბათ იქნება შემდგომი ნაბიჯი კომპიუტერული ტექნოლოგიების განვითარების გზაზე, ეს იქნება მაღალტექნოლოგიური სისტემა, დიდი ალბათობით ხელოვნური ინტელექტით.

<u>რამდენიმე საჭირო ცნება</u>

გამოთვლითი სისტემა არის პროგრამული უზრუნველყოფისა და ტექნიკური უზრუნველყოფის ერთობლიობა.

პროგრამული უზრუნველყოფა - ოპერაციული სისტემა, პროგრამა რომელიც განკარგავს და მომხმარებლის სამსახურში აყენებს კომპიუტერის ყველა რესურსს: ოპერატიულ და გარე მეხსიერებას, პროცესორის დროს, ნებისმიერ მოწყობილობას: კლავიატურას, ეკრანს, მაუსს და ა.შ.

ტექნიკური უზრუნველყოფა - ეკრანი, კლავიატურა, ე.წ. „ქეისი"(მთელი თავისი შიგთავსით: ოპერატიული მეხსიერება, გარე მეხსიერება, პროცესორი და სხვა), მაუსი და სხვა დამხმარე მოწყობილობები.

რამდენიმე სიტყვა IBM PC-ზე:

IBM PC 286 - ოპერატიული მეხსიერება 640 კილობაიტი; ოპერაციული სისტემა ტექსტურ რეჟიმში მომუშავე DOS(Disc Operation System);

IBM PC 286 AT - ოპერატიული მეხსიერება 1მეგაბაიტი; ოპერაციული სისტემა ტექსტურ რეჟიმში მომუშავე DOS(Disc Operation System);

IBM PC 386 - ოპერატიული მეხსიერება 4მეგაბაიტი; ოპერაციული სისტემა DOS-ი ინტეგრირებული გრაფიკული ოპერაციული გარემო Windows 3.0/3.1

IBM PC 486 - ოპერატიული მეხსიერება 8მეგაბაიტი; ოპერაციული სისტემა Windows 95.

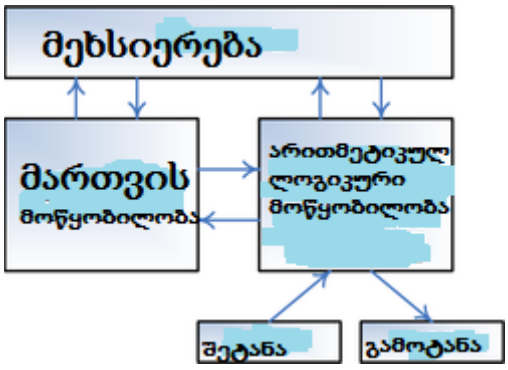
IBM PC 586 და Pentium-ის ოჯახის პერსონალური კომპიუტერები სწრაფად მზარდი ოპერატიული მეხსიერებით - 16, 32, 64, 128მბ და ა. შ. 2, 4 გბაიტამდე.

ფონ ნეიმანის პრიციპები

ზემოთ განხილული ოთხივე თაობის მანქანები წარმოადგენს ე. წ. ფონ-ნეიმანის მანქანებს.

ჯონ ფონ ნეიმანი - უნგრული წარმოშობის ამერიკელი მათემატიკოსი. მან 1944 წელს საფუძველი დაუდო გამოთვლითი მანქანების არქიტექტურას, როცა შეუერთდა პირველი თაობის ელექტრული გამოთვლითი მანქანის ENIAC-ის შექმნის პროექტს.

ფონ ნეიმანის მანქანების ფუნქციონალური სქემა შემდეგია:



კომპიუტერის დანიშნულებაა ინფორმაციის შენახვა და დამუშავება. არსებობს ტექსტური, გრაფიკული, ხმოვანი, ვიდეო და სხვა სახის ინფორმაცია.
ფონ ნეიმანის პრინციპები:

1. კომპიუტერის ფუნქციონალურ სქემაში ფიგურირებს ოპერატიული მეხსიერება*, პროცესორი და შეტანა/გამოტანის მოწყობილობები. დროის მოცემულ მომენტში პროცესორი მხოლოდ იმ ინფორმაციას ამუშავებს რომელიც ოპერატიულ მეხსიერებაშია მოთავსებული
2. ინფორმაცია და ბრძანებები ინახება კომპიუტერის მეხსიერებაში. კომპიუტერის მეხსიერება არის გადამისამართებული ბაიტების ერთობლიობა.
   მეხსიერების ერთეულს ეწოდება ბაიტი. ბაიტების გადამისამართება იწყება აბსოლუტური ნულიდან, ყოველ ბაიტს გააჩნია ერთადერთი უნიკალური მისამართი. ფიზიკურ დონეზე ინფორმაციისა და ბრძანებების ერთმანეთისაგან გარჩევა შეუძლებელია.
   არსებობს მეხსიერების უფრო მსხვილი ერთეულები:
   1კილობაიტი=1024ბაიტი=2^{10}ბაიტი;
   1მეგაბაიტი=1024კილობაიტი=2^{20}ბაიტი;
   1 გიგაბაიტი=1024მეგაბაიტი=2^{30} ბაიტი;
   1ტერაბაიტი=1024გიგაბაიტი=
3. ინფორმაცია კომპიუტერში წარმოდგენილია ორობით თვლის სისტემაში; ორობითი თვლის სისტემის საფუძველს წარმოადგენს 0 და 1; **ინფორმაციის ერთეულს ეწოდება ბიტი;** ბიტის მნიშვნელობაა 0 ან 1; 1 ბაიტში ეტევა მხოლოდ 8 ბიტი ინფორმაცია **1 ბაიტი = 8 ბიტი**
4. კომპიუტერის დახმარებით ამოვხსნათ ამოცანა ნიშნავს, შევადგინოთ ამოცანის შესაბამისი ალგორითმი, რომელიც საწყისს მონაცემებს საშედეგო მონაცემებად გარდაქმნის და მიგვიყვანს დასმული ამოცანის გადაჭრამდე.

ალგორითმის განმარტება

ალგორითმი არის გარკვეული წესების მიხედვით შედგენილი მოქმედებათა(ნაბიჯების, ბრძანებების) სასრული მიმდევრობა. ამ მოქმედებების(ნაბიჯების, ბრძანებების) გარკვეული მიმდევრობით შესრულება უზრუნველყოფს დასმული ამოცანის ამოხსნას, ანუ საწყისი მონაცემების საშედეგო მონაცემებად გარდაქმნას.

ალგორითმის ზემოთ მოყვანილი განმარტება ინტუიტურია, დღემდე არ არსებობს ალგორითმის მკაცრად ფორმულირებული მათემატიკური განმარტება. ინტუიტურს უწოდებენ ცოდნას, რომელიც დაგროვილია გარკვეული პერიოდის მანძილზე მიღებული

გამოცდილების საფუძველზე, თუმცა ეს ცოდნა ჯერ კიდევ არ არის მკაცრად ჩამოყალიბებული, მას არა აქვს მიღებული საბოლოო სახე და ახალი გამოცდილების შეძენასთან ერთად შეიძლება შეიცვალოს და დაზუსტდეს.

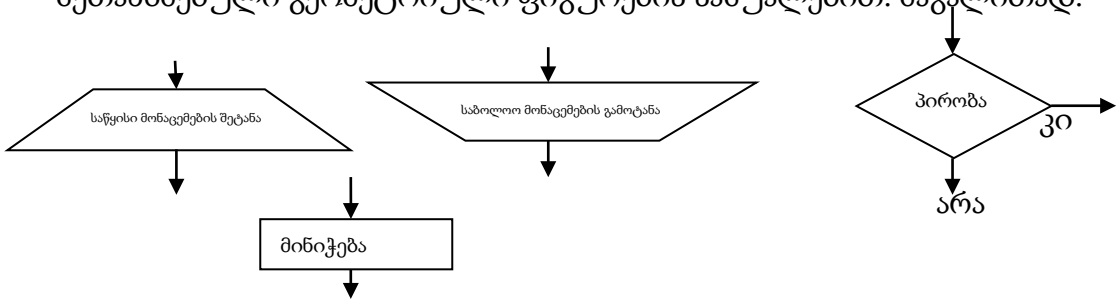
დღეს აღიარებული ვერსიის მიხედვით სიტყვა ალგორითმის წარმოშობა უკავშირდება შუა აზიის ცნობილი მეცნიერის, აბუ ჯაფარ მუჰამედ იბნ მუსა ალ-ხორეზმი ალ-მაჯუსის (783-850) სახელს. ხორეზმი მდებარეობს დღევანდელი უზბეკეთის ტერიტორიაზე, ხოლო ალ-ხორეზმი სიტყვა-სიტყვით ნიშნავს ხორეზმში მცხოვრებს. შუა საუკუნეებში ევროპელები თვლის ინდურ ათობით პოზიციურ სისტემას და თვლის მეთოდებს ამ სისტემაში ალ-ხორეზმის არაბული ტრაქტატის ლათინური თარგმანების მიხედვით გაეცვნენ. ალ-ხორეზმის არითმეტიკული ტრაქტატის ლათინური თარგმანი იწყებოდა სიტყვებით "Dixit algorizmi …" – " და თქვა ალხორიზმმა...". ამგვარად, თავიდან სიტყვა ალგორითმი გამოიყენებოდა ათობითი პოზიციური სისტემის და არითმეტიკული ოპერაციების ალგორითმის, ანუ წესების აღწერისთვის, ხოლო შემდეგ კი, ნებისმიერი ალგორითმისთვის. დღევანდელ ინგლისურ ენაში სიტყვა algorism ნიშნავს სწორედ როგორც არაბულ (ათობით) პოზიციურ სისტემას, ასევე ალგორითმს. უფრო გავრცელებულია სინონიმი სიტყვა algorithm, რომელიც ითარგმნება როგორც მეთოდი, წესი.

ალგორითმის თვისებები, ალგორითმი უნდა იყოს:

- **დისკრეტული** - ალგორითმი წარმოადგენს ცალკეულ ნაბიჯების ერთობლიობას, რომლებიც მიმდევრობით (ან პარალელურად) სრულდება. ალგორითმის თითოეული ნაბიჯი წარმოადგენს დასრულებულ მოქმედებას. მომდევნო ნაბიჯის შესრულება შესაძლებელია მხოლოდ წინა ნაბიჯის შესრულების შემდეგ.
- **დროში სასრული** – ალგორითმი არ შეიძლება იყოს უსასრულო, ალგორითმი სასრული რაოდენობის ნაბიჯების შემდეგ აუცილებლად უნდა დასრულდეს.
- **თვალსაჩინო** – ალგორითმის ყოველი ნაბიჯი უნდა იყოს ზუსტად განსაზღვრული.
- **ზოგადი** - უნიკალური ალგორითმები არ არსებობს, თუმცა გარკვეული კლასის ამოცანებისათვის ალგორითმი უნდა იყოს ზოგადი.
- **კორექტული** - ალგორითმს ვუწოდებთ კორექტულს, თუ ალგორითმის შესრულების შემდეგ შესაძლებელია გარკვეული შედეგის მიღება. თუნდაც იმ ფაქტის დადგენა, რომ ამოცანას ამონახსნი არა აქვს, წარმოადგენს ალგორითმის შესრულების ერთერთ შესაძლო შედეგს. ამბობენ, რომ კორექტული ალგორითმის დახმარებით შესაძლებელია დასმული ამოცანის ამოხსნა.

ალგორითმის ჩაწერის ხერხები:

1. ალგორითმის ჩაწერა რომელიმე ბუნებრივ ენაზე, ყველა ნაბიჯის გაწერით, ანუ **სიტყვიერი ალგორითმი.**
2. ალგორითმის ჩაწერა გრაფიკულად **ბლოკ-სქემის** საშუალებით, ანუ მიღებული, შეთანხმებული გეომეტრიული ფიგურების საშუალებით. მაგალითად:



3. ალგორითმის ჩაწერა **ფსევდოკოდის** საშუალებით. **ფსევდოკოდი** არის არაფორმალური ენა, რომელიც გამოიყენება ალგორითმების დამუშავებისა და ჩაწერისათვის. ფსევდოკოდი ხშირად წარმოადგენს გამარტივებულ დაპროგრამების ენას. ჩვეულებრივ ალგორითმის აღწერისა და დამუშავებისთვის არ არის საჭირო ყველა იმ წვრილმანის გათვალისწინება, რომლებიც აუცილებელია შესაბამისი პროგრამის მუშაობისთვის. ალგორითმიკის თეორიაში ცნობილია, რომ ნებისმიერი პროგრამა შეიძლება დავწეროთ მხოლოდ სამი ე.წ. მმართველი სტრუქტურის გამოყენებით: მიმდევრობითი, პირობითი/ამორჩევის და განმეორებადი (ციკლური) სტრუქტურების საშუალებით. ამიტომ ფსევდოკოდის დასაწერად სავსებით საკმარისია, თუ გამოვიყენებთ ამ მმართველ სტრუქტურებს.
4. ალგორითმის ჩაწერა დაპროგრამების ნებისმიერი ალგორითმული ენის დახმარებით.

ბუნებრივია, რომ კომპიუტერის დახმარებით მხოლოდ ისეთი ალგორითმის რეალიზება და მოსალოდნელი შედეგების მიღებაა შესაძლებელი, რომლებიც ჩაწერილია რომელიმე დაპროგრამების ენის დახმარებით.

არსებობს სამი სახის ალგორითმი:

1. **წრფივი ალგორითმი,** როცა ალგორითმის ყველა ნაბიჯი ერთმანეთის მიმდევრობით სრულდება.
2. **განშტოებადი ალგორითმი,** როცა რაიმე პირობის მიხედვით საბოლოო შედეგის მიღებამდე ალგორითმში რამდენიმე „გზა, ან ტრაექტორია“ გამოიკვეთება.
3. **განმეორებადი, ან ციკლური ალგორითმი,** როცა ალგორითმის რომელიმე ფრაგმენტი ერთხელ და უფრო მეტად მეორდება, მაგრამ არა უსასრულოდ.

ნებისმიერი, თუნდაც მარტივი ამოცანის და პრობლემის გადაჭრის დროს ადამიანი, ინტუიციურად, შეიძლება ამას ვერც კი ამჩნევდეს, ადგენს ალგორითმს. მარტივი ამოცანის

გადაჭრის შესაბამისი ალგორითმი მარტივია, რთული ამოცანის ამოხსნის ალგორითმი კი რთულია.
ალგორითმების თეორიის განვითარების შედეგად აღმოჩნდა,რომ არსებობს ამოცანები, რომელთათვისაც შეუძლებელია ამოხსნის ზოგადი ალგორითმის შედგენა. მათ **ალგორითმულად ამოუხსნად ამოცანებს** უწოდებენ.

კონკრეტული ამოცანის შესაბამისი ალგორითმის შედგენის დროს საჭიროა იმ საგნობრივი არის შესწავლა, რომელშიც განიხილება ამოცანა. მაგალითად, გენეტიკური ამოცანის ამოხსნის დროს საგნობრივი არეა ბიოლოგია, კერძოდ გენეტიკა; ფიზიკური ამოცანის ამოხსნის დროს საგნობრივი არეა ფიზიკა; სარეცხი მანქანის ან ავტომობილის შესაბამისი პროგრამული უზრუნველყოფის შექმნის დროს საგნობრივი არეა ფიზიკა და მათემატიკა;
რადგან ტექნოლოგიური რევოლუცია ყოფიერების ნებისმიერ სფეროს შეეხო, ამიტომ რეალური ამოცანების შესაბამისი საგნობრივი არეების სიმრავლე არც თუ ისე მცირეა.

განვიხილოთ ერთი სახალისო ამოცანა და მისი შესაბამისი სიტყვიერი ალგორითმი.
<u>ამოცანა 1:</u> გადაიყვანეთ ნავით მდინარის ერთი ნაპირიდან მეორე ნაპირზე მგელი, თხა და თივის ზვინი, ისე რომ მგელმა თხა არ შეჭამოს, თხამ კი თივა.
<u>ამოხსნა:</u>
ნაბიჯი 1. მეორე ნაპირზე გადაიყვანე თხა;
ნაბიჯი 2. პირველ ნაპირზე(უკან) დაბრუნდი მარტო;
ნაბიჯი 3. მეორე ნაპირზე გადაიტანე თივა;
ნაბიჯი 4. პირველ ნაპირზე(უკან) გადაიყვანე თხა;
ნაბიჯი 5.მეორე ნაპირზე გადაიყვანე მგელი;
ნაბიჯი 6. პირველ ნაპირზე დაბრუნდი მარტო;
ნაბიჯი 7. მეორე ნაპირზე გადაიყვანე თხა;

ზემოთ მოყვანილი ალგორითმი წრფივი ალგორითმის მაგალითია: თანმიმდევრობით სრულდება ყველა ნაბიჯი 1ნაბიჯიდან დაწყებული მე-7 ნაბიჯის ჩათვლით.
დავაკვირდეთ ალგორითმს, თუ მე-3 და მე-5 ნაბიჯებს გადავანაცვლებთ ალგორითმი მაინც კორექტული იქნება და სწორად იმუშავებს. მაგრამ ახლა ვთქვათ გადავანაცვლოთ მე-4 და მე-5 ნაბიჯები, რა მოხდება? თხა შეჭამს თივას, ხოლო მგელი შეჭამს თხას.

გავაკეთოთ დასკვნა: ალგორითმის კორექტულად მუშაობისათვის აუცილებლად საჭიროა დავიცვათ ნაბიჯების შესრულების თანმიმდევრობა.
<u>ამოცანა 2</u>: მოცემული a, b, c კოეფიციენტებისათვის იპოვეთ კვადრატული განტოლების ამონახსნები.
ამოხსნა: საწყისი მონაცემებია: a, b, c
საშედეგო მონაცემები: ორი ამონახსნი X_1, X_2 ან
ერთი ამონახსნი $X_1 = X_2$ ან
საერთოდ არა აქვს ამონახსნი
სიტყვიერი ალგორითმი: შესაბამისი პროგრამა C-ზე

1. დასაწყისი
2. შევიტანოთ a, b, c;
3. D=b*b-4*a*c;
4. თუ D>0 მაშინ 4.1 {X_1=(-b+√D)/(2*a);
   X_2=(-b-√D)/(2*a);
   გამოვიტანოთ X_1, X_2 ;}
   თუ არა 4.2 თუ D==0 მაშინ 4.2.1 {X=-b/(2*a);
   გამოვიტანოთ X;}
   თუ არა
   4.3 გამოვიტანოთ „არა აქვს ამონახსნი“
5. დასასრული

```
#include <iostream>
#include <math>
using namespace std;
main()
{  float a,b,c,D,X1,X2,X;
cin>>a>>b>>c;
D=b*b-4*a*c;
if( D>0)  { X1=(-b+sqrt(D))/(2*a);
            X2=(-b-sqrt(D))/(2*a);
            cout<<”X1=”<<X1<<”X2=”<<X2;}
    else if(D==0) { X=-b/(2*a);

            cout<<”X=”<<X;}
           else cout<<”ara aqvs amonaxsni”;  }
```

კვადრატული განტოლების ამოხსნის ალგორითმი განშტოებადი ალგორითმის მაგალითია, მე-4 ნაბიჯიდან დაწყებული მოქმედება სამ ალტერნატიულ გზად იყოფა იმის მიხედვით თუ რა ყოფაქცევისაა კვადრატული განტოლების დისკრიმინანტი, დროის მოცემულ მომენტში მხოლოდ ერთ გზაზე მოძრაობაა შესაძლებელი.
მაგალითად: თუ D>0, მაშინ ალგორითმის შესრულების რეალური სურათია: 1; 2; 3; 4; 4.1; 5 თუ D==0, მაშინ შესრულდება ნაბიჯები: 1; 2; 3; 4; 4.2; 4.2.1; 5
თუ D<0, მაშინ შესრულდება ნაბიჯები: 1; 2; 3; 4; 4.1; 4.2; 4.3; 5

<u>ამოცანა 3</u>: იპოვეთ ორი მთელი რიცხვის უდიდესი საერთო გამყოფი.
ამოხსნა: სანამ უშუალოდ ამოცანის შესაბამისი ალგორითმზე გადავალთ, ცოტა რამ ისტორიიდან.
ჩვენს წელთაღრიცხვამდე III საუკუნეში ბერძენმა მათემატიკოსმა ევკლიდემ თავის ფუნდამენტურ ნაშრომში “საწყისები“ ჩამოაყალიბა ორი ნატურალური რიცხვის უდიდესი საერთო გამყოფის პოვნის ალგორითმი, რომელსაც საფუძვლად გეომეტრიული მოსაზრებები დაედო. თუმცა, ძველი ბერძენი მათემატიკოსებისათვის ეს ალგორითმი ევკლიდემდე იყო ცნობილი “ურთიერთგამოკლების” წესის სახელწოდებით და ის არისტოტელეს ერთ-ერთ ნაშრომშია აღწერილი.

ვთქვათ, მოცემული გვაქვს ორი რიცხვი

a	b	
48	27	48 != 27 48>27
48-27	27	
21	27	21 != 27 21<27
21	27-21	
21	6	21 != 6 21>6
21-6	6	
15	6	15 != 6 15>6
15-6	6	
9	6	9 != 6 9>6
9-6	6	
3	6	3 != 6 3<6
3	6-3	
3	3	3 == 3

პროცესი შეწყდა, 48 და 27-ის უდიდესი საერთო გამყოფია 3

ალგორითმის ყოველ ნაბიჯზე უფრო დიდ რიცხვს აკლდება უფრო ნაკლები რიცხვი და შედეგი იწერება დიდი რიცხვის ადგილზე, ევკლიდეს ალგორითმი გრძელდება მანამ სანამ ერთმანეთის ტოლ მნიშვნელობებს არ მივიღებთ.

სანამ ევკლიდეს ალგორითმის სიტყვიერ ანალოგს ავღწერდეთ, ჩვენი ამოცანა შევცვალოთ:
<u>ვიპოვოთ ორი ნატურალური რიცხვის უდიდესი საერთო გამყოფი ევკლიდეს ალგორითმის დახმარებით.</u>

სიტყვიერი ალგორითმი

1. დასაწყისი
2. შევიტანოთ a,b;
3. სანამ a !=b
   თუ a>b, მაშინ a= a – b;
   თუ არა b = b - a;
4. გამოვიტანოთ a ან b;
5. დასასრული

შესაბამისი პროგრამა C-ზე

```
#include <iostream>
#include <math>
        using namespace std;
        main()
        {  float a,b;
        cin>>a>>b;
        while( a != b )
            if( a>b)  a-=b;
              else    b-=a;

    cout<<a<<"-ს და "<<b<<"-ს უსგ= "<<a;}                    }
```

ევკლიდეს ალგორითმი არის ციკლური ალგორითმის მაგალითი. ფრაგმენტი

თუ a>b, მაშინ a= a – b;
თუ არა b = b - a;

მეორდება მანამ სანამ a !=b, როგორც კი a ==b, პროცესი წყდება.

ახლა ვნახოთ რა მოხდება, თუ a და b იქნება არა ნატურალური რიცხვთა სიმრავლიდან, არამედ მთელი რიცხვები.

a	b	
48	-27	48 !=- 27 48>-27
48-(-27)	-27	
75	-27	75 != -27 75>-27
75-(-27)	-27	
102	-27	102 != -27 102>-27
102-(-27)	-27	
129	-27	129 != -27 129>-27

ზემოთ მოყვანილი ილუსტრაციიდან ჩანს, რომ ალგორითმის მსვლელობისას მოყვანილი ორი რიცხვიდან ერთი გამუდმებით იზრდება, მეორე კი უცვლელი რჩება და ალგორითმი არასდროს არ შეწყდება, რადგან ეს ორი რიცხვი ერთმანეთს კი არ უახლოვდება, არამედ უფრო შორდება ერთმანეთს. ე.ი. ევკლიდეს ალგორითმი მთელ რიცხვთა სიმრავლეზე არ მუშაობს.

<u>დავალება</u>: ევკლიდეს ალგორითმი მოარგეთ სიტუაციას, როცა a და b ეკუთვნის მთელ რიცხვთა სიმრავლეს.

ალგორითმის ანალიზი და სირთულე

ვთქვათ ერთი და იგივე ამოცანისათვის ორმა სხვადასხვა პროგრამისტმა ორი სხვადასხვა ალგორითმი მოიფიქრა, როგორ უნდა შევაფასოთ, რომელი ალგორითმი იმუშავებს უკეთესად და ოპტიმალურად. სწორედ ასეთი სიტუაციებისათვის და არა მარტო, საჭიროა ალგორითმების ანალიზი და მათი სირთულის შეფასება.

ამისათვის კი საჭიროა

- **ორი ალგორითმის შედარება;**
- **ალგორითმის ყოფაქცევის წინასწარი შეფასება;**

ძირითადად ალგორითმის ორი ტიპის სირთულეს განიხილავენ:

სირთულე დროის მიხედვით: ამოცანის ამოხსნისას ალგორთმის შემადგენელი ოპერაციები იყოფა ე.წ. ელემენტარულ ბიჯებად. ბლოკ-სქემით ჩაწერილი ალგორითმისთვის ეს იქნება შესასრულებელი და პირობითი ბლოკები. შესაძლებელია მათ მივაწეროთ განსხვავებული ბიჯების რაოდენობა, ანუ, როგორც ამბობენ განსხვავებული წონები. ფსევდოკოდისა და დაპროგრამების ენებისთვის ეს შეიძლება იყოს ის ოპერაციები, რომლებისგანაც შედგება შესაბამისი პროგრამის სტრიქონები. აქაც ბუნებრივია

განსხვავებულ ოპერაციებს მივაწეროთ განსხვავებული ბიჯების რაოდენობა ე.ი. წონები. ანუ რამდენ ბიჯს "იწონის" თითოეული ოპერაცია. ანალიზისთვის მთავარია მხოლოდ, რომ ეს ბიჯები ერთნაირად ითვლებოდეს ყოველ სიტუაციაში. ჩვეულებრივ, შეფასებისას იგულისხმება, რომ კომპიუტერზე არითმეტიკული ოპერაციები ერთმანეთის მიმდევრობით სრულდება.

პირველი მიდგომით, ბუნებრივია, სწორედ ელემენტარული ბიჯების სრულ რაოდენობას ვუწოდოთ ალგორითმის სირთულე დროის მიხედვით. მაგრამ, რადგან ალგორითმი დამოკიდებულია საწყის მონაცემებზე, ამიტომ საინტერესოა დამოკიდებულება საწყისს მონაცემებსა და ბიჯების სრულ რაოდენობას შორის. მაგალითად, თუ გვაქვს n მონაცემი, მაშინ ბიჯების სრული რაოდენობა იქნება $T(n)$ ე.ი. **ამოცანის დროითი სირთულე**. ზოგჯერ საწყისი მონაცემების ზომას (რაოდენობრივ ზომას) უწოდებს **ამოცანის ზომას**, სიდიდეს. როდესაც აკვირდებიან ამოცანის n ზომის ზრდის მიხედვით $T(n)$ ამოცანის დროითი სირთულის ყოფაქცევას, მაშინ ამბობენ, რომ აკვირდებიან **ასიმპტოტურ დროით სირთულეს**.

შესაძლებელია გვაინტერესებდეს არა ალგორითმის ყველა ოპერაციის შესაბამისი ელემენტარული ბიჯების რაოდენობა, არამედ მხოლოდ რომელიღაც ამორჩეული ოპერაციების შესაბამისი ელემენტარული ბიჯების რაოდენობა. ზუსტად თუ ვიტყვით, გვაინტერესებს ამორჩეული ოპერაციების შესაბამისი ელემენტარული ბიჯების რაოდენობის დამოკიდებულება შემავალ მონაცემებზე. ამ დროს ვამბობთ, რომ ვსაუბრობთ ალგორითმის ანალიზზე ამ ამორჩეული ოპერაციების მიმართ.

სირთულე მეხსიერების მიხედვით: ამოცანის შესაბამისი პროგრამის რეალიზაციისას, ბუნებრივია ხდება გარკვეული რაოდენობის სხვადასხვა რესურსების გამოყენება. ყველაზე ძვირფასი რესურსია ოპერატიული მეხსიერება და პროცესორის რეალური დრო. დროით სირთულეზე ზემოთ უკვე ვილაპარაკეთ, ახლა განვიხილოთ მეხსიერებასთან დაკავშირებული პრობლემები. თუ განვიხილავთ, მაგალითად, სიმარტივისთვის, ერთი რომელიმე ტიპის მეხსიერების დაკავებულ რაოდენობას, მაშინ წინა შემთხვევის ანალოგიურად შეიძლება შემოვიტანოთ მოცულობითი სირთულე ანუ სხვა ტერმინოლოგიით **სირთულე მეხსიერების მიხედვით** – ესაა დამოკიდებულება ამოცანის n ზომასა და $T(n)$ მეხსიერების დაკავებულ რაოდენობას შორის. ანალოგიურადვე განიმარტება **ასიმპტოტური სირთულე მეხსიერების მიხედვით**.

ვთქვათ, შევარჩიეთ კრიტერიუმი ალგორითმის შესაფასებლად. გარკვეულობისთვის ვიგულისხმოთ, რომ ესაა დროითი სირთულე. ახლა, შემავალ მონაცემებზე დამოკიდებულებით ერთი და იგივე ალგორითმმა შეიძლება ამოცანის ამოხსნას განსხვავებული ბიჯების რაოდენობა მოახმაროს. ამიტომ, ალგორითმის შეფასებისას უნდა გათვალისწინებულ იქნას სხვადასხვა შემავალ მონაცემებთან დაკავშირებული ბიჯების რაოდენობები. ასეთი მიდგომით, ალგორითმი ამომწურავად ხასიათდება სამი მახასიათებლით: **დროითი სირთულე საუკეთესო შემთხვევაში, დროითი სირთულე უარეს შემთხვევაში, დროითი სირთულე საშუალო შემთხვევაში.**

საუკეთესო შემთხვევა – ეს ნიშნავს შემავალი მონაცემების მიხედვით დროითი სირთულის მინიმალურ მნიშვნელობას ანუ ელემენტარული ბიჯების ყველაზე ნაკლებ

რაოდენობას. ანალოგიურად, **უარესი შემთხვევა** იქნება ელემენტარული ბიჯების მაქსიმალური რაოდენობა შემავალი მონაცემების მიხედვით. ზოგიერთი ავტორი მხოლოდ ამ უკანასკნელ მაჩვენებელს უწოდებს ალგორითმის სირთულეს.

ანალიზისთვის ძალიან მნიშვნელოვანია ე.წ. **საშუალო შემთხვევა** (ყველაზე მოსალოდნელი შემთხვევა)**.** ანუ გვაინტერესებს ალგორითმს (ან ალგორითმის გარკვეულ ოპერაციას) რამდენი ელემენტარული ბიჯი სჭირდება საშუალოდ, შემავალი მონაცემების მიხედვით. მარტივ შემთხვევაში ამის გამოთვლა ხდება შემდეგნაირად: უნდა განვიხილოთ შემავალი მონაცემების ყველა შესაძლო ვარიანტი (ანუ განვიხილოთ ყველა დასაშვები ვარიანტის სიმრავლე). ჯერ ეს სიმრავლე უნდა დავყოთ ისეთ თანაუკვეთ ქვესიმრავლეებად, რომ თითოეული ქვესიმრავლის ელემენტებზე ალგორითმს ჰქონდეს მუდმივი სირთულე. დავუშვათ, სულ გამოგვივიდა k ცალი ქვესიმრავლე და i-ურ ქვესიმრავლეში, რომელიც შეიცავს m_i ელემენტს ($i = 1, \ldots, k$) სირთულეა x_i. თუ მთელ სიმრავლეში m ცალი ელემენტია, მაშინ i-ური ქვესიმრავლის წილი არის $\frac{m_i}{m} = p_i$. საშუალოს გამოსათვლელად უნდა გამოვიყენოთ ფორმულა

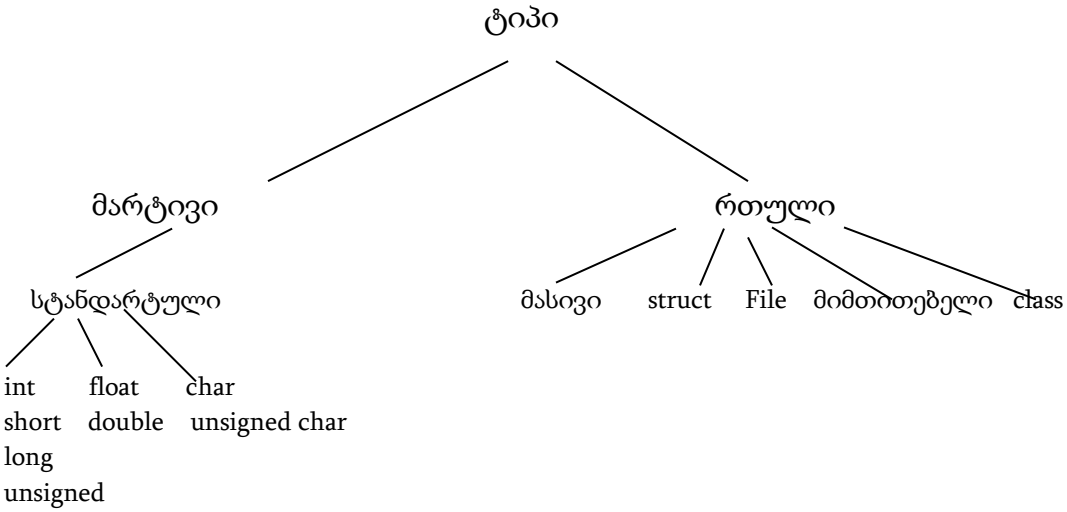
$$\sum_{i=1}^{k} p_i x_i = \sum_{i=1}^{k} \frac{m_i}{m} x_i = \frac{1}{m} \sum_{i=1}^{k} m_i x_i \ .$$

ყოველივე ზემოთ თქმული შევაჯამოთ:
თუ დროის მოცემულ მომენტში უფრო ფასეულია პროცესორის რეალური დრო, მაშინ უპირატესობას ანიჭებთ იმ ალგორითმს, რომელის ასიმტოტური სირთულე დროის მიხედვით უკეთესია.
თუ დროის მოცემულ მომენტში უფრო ფასეულია ოპერატიული მეხსიერება, მაშინ გადაწყვეტილებას მიიღებთ იმ ალგორითმის სასარგებლოდ, რომელის ასიმტოტური სირთულე მეხსიერების მიხედვით უკეთესია.
ბუნებრივია იდეალურია ის სიტუაცია, თუ თქვენი ალგორითმის როგორც ასიმტოტური სირთულე დროის მიხედვით და ასიმტოტური სირთულე მეხსიერების მიხედვით ოპტიმალურია.

დამატებითი მასალა, რომელიც უნდა იცოდეთ, ან უახლოეს დროში უნდა ისწავლოთ

მასივი

C/C++ – ის ტიპების კლასიფიკაცია შესაძლებელია ასე:



ტიპები იმიტომ არსებობენ დაპროგრამების ენებში, რომ მათი დახმარებით შესაძლებელი იყოს ცვლადების აღწერა.

თუ ცვლადი არის მარტივი ტიპის, მაშინ მის უკან დროის მოცემულ მომენტში მოიაზრება მხოლოდ ერთი მნიშვნელობა. ხოლო თუ ცვლადი არის რთული ტიპის, ეს ნიშნავს რომ საქმე გვაქვს არა ერთ მნიშვნელობასთან, არამედ მნიშვნელობათა ჯგუფთან.

რთული ტიპის პირველი წარმომადგენელი არის მასივი.

მასივი არის ერთი და იგივე ტიპის ელემენტთა ერთობლიობა. იმისათვის რომ მასივის ტიპის ცვლადთან დავიწყოთ ურთიერთობა, საჭიროა ავღწეროთ მასივის ტიპის ეს ცვლადი. როგორ მოვახერხოთ ეს?

გარდა იმისა რომ მასივში ყველა ელემენტი უნდა იყოს ერთი და იგივე ტიპის, აუცილებლად აღწერის დროს უნდა ვიცოდეთ რამდენი ელემენტია ამ მასივში. ანუ ავღწეროთ მასივი ნიშნავს: განვმარტოთ რა ჰქვია მასივს, რამდენი ელემენტია ამ მასივში და თითოეული ელემენტი რა ტიპისაა.

მასივის აღწერა ხდება შემდეგნაირად:

ტიპი ცვ.სახელი[მასივის განზომილება];

ქართულად გაფორმებული ადგილები უნდა შეივსოს შესამაბისი ინფორმაციით. ცვლადის სახელი როგორ უნდა შევარჩიოთ უკვე ცნობილია, ტიპი არის თქვენს მიერ ცნობილი ნებისმიერი, გარდა ფაილური და class ტიპისა. რაც შეეხება ელემენტების რაოდენობას: მასივში ელემენტები გადანომრილია, ანუ ყველა ელემენტს აქვს თავისი რიგითი ნომერი, ანუ **ინდექსი.** მასივში ელემენტების რაოდენობა განისაზღვრება მისი

განზომილებით.

მაგ:

```
int x[34];
float y[50];
char z[100];
```

მაგალითში აღწერილი ყველა ცვლადი მასივის ტიპისაა.x მასივში არის 34 მთელი ტიპის ელემენტები, y მასივში არის 50 რეალური ტიპის ელემენტები, z მასივში არის 100 სიმბოლური ტიპის ელემენტები.

ყურადღება მიაქციეთ იმას რომ მასივში ელემენტების გადანომვრა აუცილებელად იწყება 0-დან.

მასივის თითოეულ ელემენტს გააჩნია საკუთარი სახელი, ამასთან დაკავშირებით შემოდის ახალი ცნება: **ინდექსირებული ცვლადი.** ინდექსირებული ცვლადი გამოიყენება მხოლოდ მასივთან მიმართებაში და იგი ფორმირდება ასე: მასივის ზოგადი სახელი და ელემენტის რიგითი ნომერი.

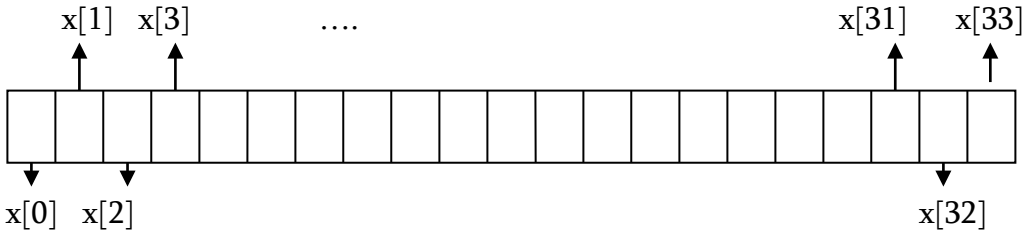
მაგ:

```
X[5]=67;   ეს ჩანაწერი ნიშნავს X მასივის მხოლოდ მე-5 ელემენტის მნიშვნელობა არის 67
D[3]=68.2;
```

არასწორია ჩანაწერი: X=5 , ან D=68.2. კრებითი,მრავლობითი ურთიერთობა მასივთან არ შეიძლება. უნდა დააკონკრეტოთ მასივის რომელ ელემენტთან გაქვთ საქმე.

საზოგადოდ მასივის ელემენტი ჩაიწერება ასე X[I], სადაც X არის მასივის სახელი, ხოლო I არის ამ მასივის ინდექსი და მისი მნიშვნელობა იცვლება აუცილებლად 0-დან 34-მდე.

სქემატურად მასივი შეიძლება ასე გამოვსახოთ:



თუ x[5]=-67; -67 ჩაიწერება მხოლოდ მისთვის განკუთვნილ უჯრაში, თუ x[33]=0, მაშინ 0 ჩაიწერება x[33]-ის შესაბამის უჯრაში. x მასივის თითოელი ელემენტი იკავებს sizeof(int) რაოდენობა ბაიტს, ხოლო მთლიანად x მასივი დაიკავებს 34*sizeof(int) ბაიტს.

წავიკითხოთ მასივი,ნიშნავს, რომ უნდა წავიკითხოთ ამ მასივის ყველა ელემენტების მნიშვნელობები. ეს ნიშნავს, რომ უნდა გავაფორმოთ ციკლი:

```
   .   .   .
for(i=0;i<34;i++)
 cin>>x[i];          // ან   scanf("%d",&x[i]);
    .   .   .
```

ასევე დავბეჭდოთ მასივი ნიშნავს დავბეჭდოთ ამ მასივის ყველა ელემენტი:

```
 .   .   .

 for(i=0;i<34;i++)
 cout<<x[i];          // ან   printff("%d",&x[i]);
    .  .  .
```

მასივის ყოველ ელემენტს, გარდა პირველისა და ბოლო ელემენტისა ჰყავს ორი მეზობელი: მარჯვენა მეზობელი და მარცხენა მეზობელი x[i] - ის მარჯვენა მეზობელია x[i+1], ხოლო მარცხენა მეზობელია x[i-1].

მიაქციეთ ყურადღება, რომ x[i+1] და x[i]+1 სხვადასხვა შინაარსის ჩანაწერია.

იფიქრეთ სახლში რატომ?

ამოცანა: მოცემული გაქვთ 20 ელემენტიანი მთელი ტიპის მასივი. დაითვალეთ მხოლოდ უარყოფითი ელემენტების ჯამი

```
#include <iostream>
using namespace std;
main()
{int A[50],S=0;
 int i;
    for(i=0; i<50; i++)
    cin>>A[i];                              //    ან  scanf("%d",&A[i]);

    for(i=0; i<50; i++)
    if(A[i]<0) S+=A[i];

cout<<"uarkofiti el. jami="<<S;          //     ან   printf("uarkofiti el. jami=%s",S);
 }
```

ფუნქციათა რაობა, მათი დანიშნულება. პრინციპები: პროგრამირება ზემოდან ქვემოთ და პროგრამირება ქვემოდან ზემოთ.

საზოგადოდ, C/C++ დაწერილი პროგრამა წარმოადგენს ფუნქციათა ერთობლიობას. ამ ფუნქციებიდან მხოლოდ ერთია ძირითადი ფუნქცია, ხოლო დანარჩენი ფუნქციები, რომლებიც გამოიძახება ძირითადი ფუნქციიდან არის ე.წ. ქვეფუნქციები. ძირითადი ფუნქცია არის main() ფუნქცია, რომელსაც იძახებს ოპერაციული სიტემა, ამიტომაც არის ის ძირითადი. C/C++ ფუნქციათა იერარქიის სათავეში დგას main ფუნქცია, ყველა დანარჩენი ფუნქცია გამოიძახება ან ძირითადი ფუნქციიდან, ან რომელიმე მათგანი გამოიძახებს სხვას.

საზოგადოდ რა საჭიროა ფუნქციები?

ყველაზე მარტივი არგუმენტი ფუნქციების არსებობის სასარგებლოდ შეიძლება გამოდგეს ის, რომ მომხმარებლის პროგრამაში განმეორებადი ფრაგმენტები შეიძლება

აღიწეროს ერთხელ, როგორც ფუნქცია, მაგრამ გამოძახებული იყოს იმდენჯერ რამდენჯერაც საჭიროა.

მაგრამ არსებობს რა თქმა უნდა გაცილებით სერიოზული არგუმენტი: სერიოზული პროგრამული პროექტების შექმნის დროს ხდება ე.წ. საქმის დანაწილება: როგორც წესი ასეთ პროექტებზე მუშაობს ჯგუფი პროგრამისტებისა, ყოველი ადამიანი აკეთებს მასზე დაკისრებულ საქმეს, ანუ წერს საკუთარ ფუნქციას. საბოლოო პროექტი, როგორც წესი წარმოადგენს ფუნქციების ერთობლიობას, რომელთა გაერთიანება ხდება ძირითად ფუნქციაში (მათი გამოძახების გზით).

ფუნქციებთან დაკავშირებით ორი პრინციპი: პროგრამირება ზემოდან ქვემოთ და პროგრამირება ქვემოდან ზემოთ.

პირველი პრინციპი მდგომარეობს შემდეგში: ჯერ იწერება ძირითადი ფუნქცია, ხოლო მხოლოდ ამის შემდეგ იწერება ის ფუნქციები, რომელთა გამოძახებაც ხდება ძირითადი ფუნქციიდან.

მეორე პრინციპი მდგომარეობს შემდეგში: ჯერ იწერება ფუნქციები, ხოლო ამის შემდეგ იკვეთება ძირითადი ფუნქციის კონტურები, ანუ სულ ბოლოს იწერება ძირითადი ფუნქცია, საიდანაც ხდება უკვე გამზადებული ფუნქციების გამოძახება.

ფუნქცია აღიწერება შემდეგნაირად:

```
დასაბრუნებელი_ტიპი ფუნქციის_სახელი(ფორმალური პარამეტრების სია)
{
        ფუნქციის ტანი;
                }
```

ფუნქციის დასაბრუნებელი ტიპი არის ნებისმიერი სტანდარტული ტიპი. ფუნქციის სახელი არის მომხმარებლის მიერ შერჩეული იდენტიფიკატორი.

ფუნქციის ფორმალური პარამეტრები არის ისეთი პარამეტრები, რომლისთვისაც აღიწერება ფუნქცია.

ფუნქციის ტანის გაფორმება ხდება იმავე წესებით, როგორც ეს ხდება main ფუნქციის შემთხვევაში. ანუ ფუნქციაში ხდება ლოკალური ცვლადების აღწერა,

არსებობს ოპერატორების მიმდევრობა, რომლებიც განმარტავს კონკრეტული ფუნქციის შინაარსს და ბოლოს ფუნქციის ტანში შეიძლება ფიგურირებდეს ოპერატორი return, ან შეიძლება ეს ოპერატორი არ იყოს.

ფუნქციებს, რომელთა ტანში არ არის ოპერატორი return, ეწოდება void ფუნქციები. ხოლო ფუნქციებს, რომელთა ტანში ფიგურირებს ოპერატორი return, ეწოდება არა void ფუნქციები, ასეთ ფუნქციებს არა აქვთ კონკრეტული დასაბრუნებელი ტიპი.

void და არა void ფუნქციები ძირითადი ფუნქციიდან გამოძახების თვალსაზრისითაც განსხვავდებიან: მათი გამოძახება main ფუნქციიდან ხდება სხვადასხვანაირად.

ფუნქციის აღწერა უნდა უსწრებდეს ძირითად ფუნქციის აღწერას, საიდანაც მოხდება ამ ფუნქციის გამოძახება.

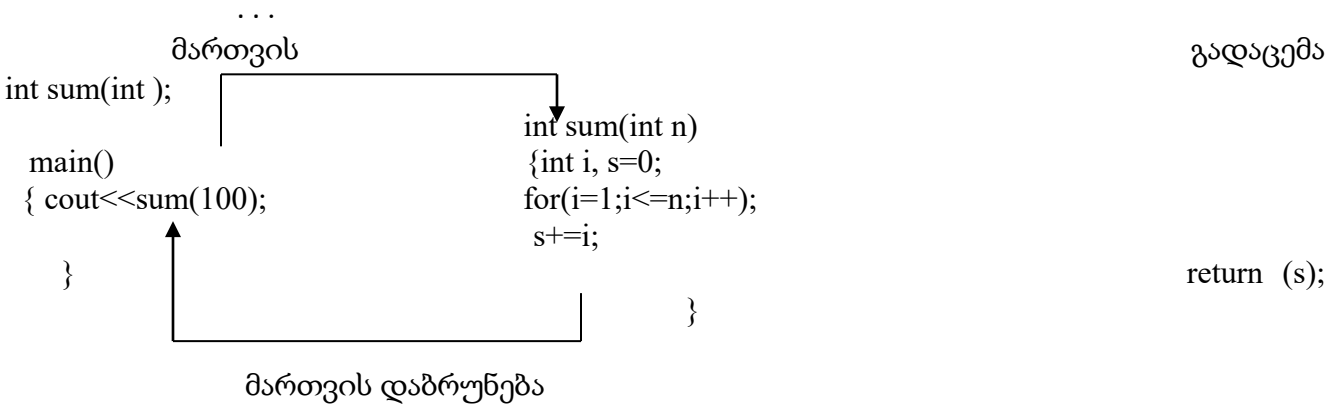
magaliTi 1:

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

    int sum(int n)                                          // sum ფუნქციის სახელი
    {int i;                          //ფუნქცია ითვლის 1-დან n-მდე რიცხვების ჯამს
    int s=0;                         // i და s არის ფუნქციის ლოკალური ცვლადები
    for( i=1;i<=n;i++)
    s+=i;
    return (s); }

      main()
    {cout<<sum(100)<<"\n";            // printf("%d\n", sum(100));
     cout<<sum(57)<<endl;             //  printf("%d\n", sum(57));

                                }
```

• ფორმალური და ფაქტიური პარამეტრების ცნება. ლოკალური ცვლადები. მართვის გადაცემა ძირითადი ფუნქციიდან და მართვის დაბრუნება.

ფაქტიური პარამეტრები არის ის პარამეტრები, რომლისთვისაც ხდება ფუნქციის გამოძახება, მოყვანილ მაგალითში ფაქტური პარამეტრიებია: 100 და 57. ფუნქციის პირველი გამოძახებისას n ფორმალური პარამეტრის მნიშვნელობა იქნება 100, ხოლო მეორე გამოძახებისას კი 57. ანუ ფუნქციის გამოძახებისას ფორმალური პარამეტრი იცვლება შესაბამისი ფაქტიური პარამეტრით და ფუნქცია მუშაობს ფაქტიური პარამეტრისათვის.

ამბობენ, რომ ფუნქციის გამოძახებისას ხდება ამ ფუნქციისათვის მართვის გადაცემა, ხოლო ფუნქციის მუშაობის დასრულების შემდეგ ხდება მართვის დაბრუნება გამომძახებელი ფუნქციისათვის.



ახლა განვიხილოთ void ფუნქციის მაგალითი.

მაგალითი 2: დავალაგოთ ნამდვილი ტიპის სამი ცვლადი კლებადობით. ორი რიცხვის შედარებისათვის და თუ საჭიროა მათთვის მნიშვნელობების შესაცვლელად გავაფორმოთ

ფუნქცია

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

void Shecvla(float& x, float &y)    //ფუნქციია სათაური x და y ფორმალური პარამეტრი
{float z;                                   //   z  ფუნქციის ლოკალური ცვლადი
  if(x<y){d=x;
              x=y;
               y=d;}
}

main()
{float a,b,c;
  cin>>a>>b>>c;
 Shecvla(a,b);                          //ფუნქციის პირველი გამოძახება, a და b ფაქტიური პარ.
 Shecvla(b,c);                                ////ფუნქციის მეორე გამოძახება, b და c ფაქტიური პარ. Shecvla(a,b);
////ფუნქციის მესამე გამოძახება, a და b ფაქტიური პარ.
 cout<<a<<" "<<b<<" "<<c;        }
```

void ფუნქციის გამოძახება ძირითადი ფუნქციიდან ხდება როგორც დამოუკიდებელი ოპერატორი: საჭიროა მიუთითოთ ფუნქციის სახელი (Shecvla) და ფრჩხილებში შესაბამისი ფაქტიური პარამეტრები (მაგალითად a და b), ვისთვისაც რეალურად უნდა იმუშაოს ფუნქციამ ამ გამოძახების დროს.

•ფორმალური პარამეტრების აღწერა და ფაქტიური და ფორმალური პარამეტრების შესაბამისობის წესი.

ფორმალური პარამეტრებია აღწერა ხდება შემდეგნაირად: ყოველ ფორმალურ პარამეტრს ექსკლუზიურად უნდა ჰქონდეს თავისი ტიპი, სახელი. ფორმალური პარამეტრების ჩამოთვლა ხდება მძიმით.

არსებობს ორი სახის ფორმალური პარამეტრი: მნიშვნელობითი ფორმალური პარამეტრი და მისამართული ფორმალური პარამეტრები.

მნიშვნელობითი ფორმალური პარამეტრი აღიწერება შემდეგნაირად:

ტიპი ფორმალური_პარ_სახელი

ხოლო მისამართული ფორმალური პარამეტრი აღიწერება შემდეგნაირად:

ტიპი& ფორმალური_პარ_სახელი

პირველ მაგალითში n არის მნიშვნელობითი ფორმალური პარამეტრი, ხოლო მეორე მაგალითში x და y არის მისამართული ფორმალური პარამეტრი.

ის ფაქტი, თუ როგორ არის აღწერილი ფორმალური პარამეტრი, მნიშვნელოვან როლს თამაშობს ფორმალური და ფაქტიური პარამეტრეების შესაბამისობისას. განვიხილოთ ახლა ეს საკითხი.

ფორმალური და ფაქტიური პარამეტრების შესაბამისობის საკითხი დღის წესრიგში დგება,

როცა ხდება ფუნქციის გამოძახება.

ფორმალური და ფაქტიური პარამეტრების შესაბამისობა გულისხმობს შემდეგი სამი წესის დაცვას:

1**. რაოდენობა:** ფორმალური და ფაქტიური პარამეტრების რაოდენობა აუცილებლად უნდა ემთხვეოდეს ერთმანეთს.

2. **პოზიციურობა**: ფორმალური და ფაქტიური პარამეტრების შესაბამისობის დროს უნდა იყოს დაცული ე.წ პოზიციურობა: პირველ ფორმალურ პარამეტრს აუცილებლად შეესაბამება პირველი ფაქტიური პარამეტრი, მეორე ფორმალურ პარამეტრს აუცილებლად შეესაბამება მეორე ფაქტიური პარამეტრი და ა.შ.

3. **ტიპების შესაბამისობა:** შესაბამისი ფორმალურ - ფაქტიურ პარამეტრების ტიპები აუცილებლად უნდა ემთხვეოდნენ ერთმანეთს, ანუ აბსოლუტურად იდენტურები უნდა იყვნენ.

როცა საჭიროა ძირითად ფუნქციას დაუბრუნდეს ქვეფუნქციიდან შეცვლილი მნიშვნელობები ფორმალური პარამეტრების გავლით, ამ შემთხვევაში ფორმალური პარამეტრები აუცილებლად უნდა იყოს აღწერილი, როგორც მისამართული ფორმალური პარამეტრები. მაგრამ თუ ძირითად ფუნქციაში არავითარი დატვირთვა არა აქვს ფორმალური პარამეტრის ცვლილებას, მაშინ შეიძლება ფორმალური პარამეტრი აღწერილი იყოს როგორც მნიშვნელობითი ფორმალური პარამეტრები.

თუ მაგ.2-ში ფორმალურ პარამეტრებს არ აღვწერდით, როგორც მისამართულ ფორმალურ პარამეტრებს, მაშინ x და y-ს შესაბამისი ფაქტიურ პარამეტრებში არ დაფიქსირდებოდა ის ცვლილება რაც განიცადეს x და y ფორმალურმა პარამეტრებმა.

როცა ფუნქციაში ფიგურირებს ოპერატორი return, მაშინ ამბობენ, რომ ეს ფუნქცია ცხადად აბრუნებს მნიშვნელობას გამომძახებელ ფუნქციაში return ოპერატორის საშუალებით. return ოპერატორში მითითებული გამოსახულების მნიშვნელობა აუცილებლად უნდა ემთხვეოდეს ფუნქციის დასაბრუნებელ ტიპს.

მაგ.1-ში return ოპერატორში მითითებული s გამოსახულების ტიპი ემთხვევა ფუნქციის დასაბრუნებელ ტიპს. შესაბამისად ასეთი ფუნქციების გამოძახება main ფუნქციაში შეიძლება განხორციელდეს ყველა იმ ადგილას, სადაც შეიძლება იმ ტიპის ცვლადის გამოყენება, რა ტიპისაცაა ეს ფუნქცია. ამ კონკრეტულ მაგალითში ფუნქციის გამოძახება ხდება cout "ოპერატორში" . შეიძლება კონკრეტული ტიპის ფუნქციის გამოძახება მოხდეს მინიჭების ოპარატორის, ან პირობითი გადასვლის (if ოპარატორი) ოპერატორის მეშვეობით. ამ დროს ფუნქციის გამოძახების ადგილზე დაბრუნდება return ოპერატორში მითითებული გამოსახულების მნიშვნელობა.

როცა ფუნქციის ფორმალურ პარამეტს წარმოადგენს მასივი, ამ შემთხვევაში, მისი აღწერისას უნდა დავიცვათ შემდეგი წესი. ამ დროს ფორმალური პარამეტრი აღიწერება ასე:

ტიპი ფორმალური-პარ_სახელი[ ]

ან

ტიპი ფორმალური-პარ_სახელი[მასივის განზომილება]

ან

ტიპი [ ]ფორმალური-პარ_სახელი

მეორე ვარიანტი ნაკლებ მოქნილია, ამ შემთხვევაში ფუნქცია, მხოლოდ კონკრეტული განზომილების მასივებისათვის იმუშავებს, ანუ ის დაკარგავს ზოგადობის თვისებას. აქედან გამომდინარე არჩევანი უნდა გაკეთდეს პირველი, ამ მესამე ვარიანტის სასრგებლოდ, მაგრამ მაშინ ფუნქციას დამატებით უნდა გადავცეთ ინფორმაცია მასივის განზომილების შესახებ.

მაგალითი 3: მოც. ნამდვილი ტიპის 23 ელემენტიანი მასივი, იპოვეთ ამ მასივში მინიმალური დადებით ელემენტებს შორის.

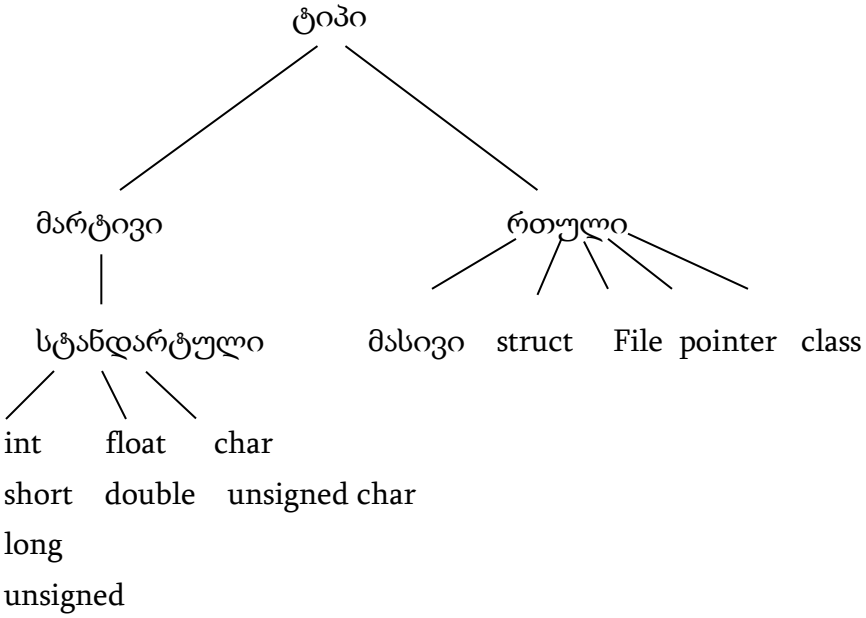
```
#include <iostream>
using namespace std;

float min_positiv(float x[], int n)
{int i,p=1;
  float min;                          მასივის შესაბამისი ფორმალური პარამეტრი
  for(i=0;i<n&&p;i++)
   if(x[i]>0){min=x[i];
              p=0;}
  if(p) return -1;
        else { for(;i<n;i++)
               if(x[i]>0&&min>x[i]) min=x[i];
                return min;  }
}
main()
{float A[23],k;
  for(int i=0;i<23;i++)
  cin>>A[i];

if((k=min_positiv(A,23))= = -1)  cout<<"masivSi dadebiTi elementebi ar aris";
                                 else cout<<"minimaluri dadebiT elementebs Soris="<<k;
}
```

ყურადღება მიაქციეთ ფუნქციის გამოძახებას, მასივის გადაცემისას საჭიროა მხოლოდ მასივის სახელის მითითება და რადგან ამ ფუნქციას მეორე პარამეტრიც აქვს, უნდა გადასცეთ ამ მასივის კონკრეტული განზომილება(რადგან შესაბამის ფუნქციას, რომლის გამოძახებას ახლა ვაკვირდებით აქვს ორი ფორმალური პარამეტრი: პირველი float მასივის შესაბამისი, მეორე პარამეტრი კი პასუხისმგებელია მასივის განზომილებაზე. გამოძახებისას x ფორმალური პარამეტრი შეიცვლება AA-თი, ხოლო Nn ფორმალური პარამეტრის მნიშვნელობა იქნება 23). თუ ამავე ფუნქციას გამოვიძახებთ მაგალითად 89 ელემენტიანი მასივისათვის, მაშინ ფუნქციის გამოძახებას ექნება სახე: min_positiv(A,89)

ლექცია II
მონაცემთა მარტივი და რთული სტრუქტურების ზოგადი მიმოხილვა. ტიპის ცნება. ორობით კოდში წარმოდგენილი ინფორმაციის ინტერპრეტაცია ტიპის მიხედვით.

მთელი რიცხვების ორობით კოდში გადაყვანის ალგორითმები. პირდაპირი, შებრუნებული და დამატებითი კოდის ცნება.

ნებისმიერი დაპროგრამების ენის ქვაკუთხედს წარმოადგენს ტიპების კონცეპცია. საზოგადოდ, ტიპების კლასიფიკაცია, C/C++ – ის მაგალითზე, შეიძლება გრაფიკულად ასე გამოვსახოთ:



ტიპები იმიტომ არსებობენ დაპროგრამების ენებში, რომ მათი დახმარებით შესაძლებელი იყოს ცვლადების აღწერა. ტიპი განსაზღვრავს თუ რამდენ ბაიტს იკავებს მოცემული ცვლადი მეხსიერებაში და როგორ უნდა იყოს აღქმული ამ მეხსიერებაში ჩაწერილი ორობითი ინფორმაცია.

მთელი ტიპების ოჯახი		
დასახელება	სიგრძე	დიაპაზონი
int	sizeof(int)	$-2^{8*\,sizeof(int)-1}$ - $2^{8*\,sizeof(int)-1}-1$
long	2*sizeof(int)	$-2^{8*2*\,sizeof(int)-1}$ - $2^{8*2*\,sizeof(int)-1}-1$
short	sizeof(int) ან sizeof(int)/2	
usigned	sizeof(int)	0 - $2^{8*\,sizeof(int)}-1$
usigned short	sizeof(int) ან sizeof(int)/2	
usigned long	2* sizeof(int)	0 - $2^{8*2*\,sizeof(int)}-1$

ნამდვილი ტიპების ოჯახი		
დასახელება	სიგრძე	დაპაზონი
float	32	3.4e-038 – 3.4e+038
double	64	1.7e-308 – 1.7e+308
long double დამოკიდებულია ტრანსლიატორის კონკრეტულ რეალიზაციაზე		

სიმბოლური ტიპი		
დასახელება	სიგრძე	დიაპაზონი
char	8	-2^7 - 2^7-1
usigned char	8	0 - 2^8-1

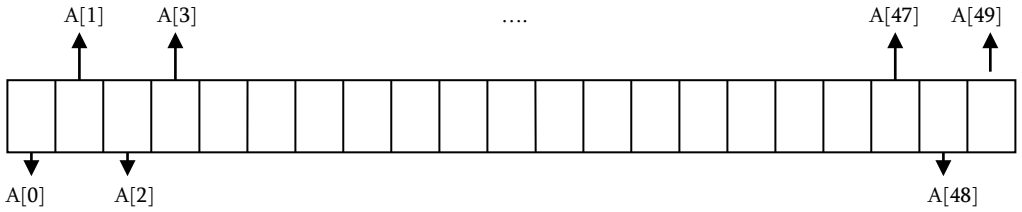
ლოგიკური ტიპი		
დასახელება	სიგრძე	დიაპაზონი
bool	8	true/false

თუ ცვლადი არის მარტივი ტიპის, მაშინ მის უკან დროის მოცემულ მომენტში მოიაზრება მხოლოდ ერთი მნიშვნელობა. ხოლო თუ ცვლადი არის რთული ტიპის, ეს ნიშნავს რომ საქმე გვაქვს არა ერთ მნიშვნელობასთან, არამედ მნიშვნელობათა ჯგუფთან.
მაგალითად:

int X; რადგან თქვენი პროგრამის რეალიზაცია შეიძლება მხოლოდ კომპიუტერის დახმარებით, ამიტომ x ცვლადთან გაიგივებულია sizeof(int) რაოდენობა ბაიტები და ამ ცვლადის მნიშვნელობები ჩაიწერება მაინც და მაინც ამ ბაიტებში და არა სადმე სხვა ადგილას.თუ X=9, მაშინ

X | 9(ორობით კოდში)

int A[50]; ეს აღწერა კი ნიშნავს, რომ A ცვლადის უკან მოიაზრება 50 მთელი ტიპის მონაცემი და თითოეულ ამ მონაცემთან შესაძლებელია ინდექსირებული ცვლადის A[i] (i=0, … 33) დახმარებით მუშაობა. მასივი A მეხსიერებაში იკავებს 50* sizeof(int) ბაიტს. სქემატურად ეს ასე შეიძლება გამოისახოს:



ფონ ნეიმანის 2-3 პრინციპის თანახმად ინფორმაცია და ბრძანებები ინახება კომპიუტერის მეხსიერებაში; ინფორმაცია და ბრძანებები მეხსიერებაში წარმოდგენილია ორობით თვლის სისტემაში;
ფიზიკურ დონეზე შეუძლებელია იმის გარკვევა, თუ 0-ებისა და 1-ების მიმდევრობა კონკრეტულად რა ინფორმაციას, ან რა ბრძანებას ნიშნავს. ტიპის მიხედვით 0-ებისა და 1-ების კონკრეტული მიმდევრობა ერთ შემთხვევაში შეიძლება სხვა რაიმე იყოს, მეორე შემთხვევაში კი სხვა რამ.
მაგალითად: ავიღოთ 00110000. თუ 0-ებისა და 1-ების ამ მიმდევრობას short int ტიპის ცვლადის მნიშვნელობად განვიხილავთ ეს იქნება მთელი რიცხვი 48.
მაგრამ თუ იგივე 0-ებისა და 1-ების მიმდევრობას შევუსაბამებთ char ტიპის ცვლადს, მაშინ ეს იქნება კონსტანტური სიმბოლო '0'.
რაც უფრო რთულია ინფორმაცია, მით უფრო რთულია შესაბამისი ორობითი კოდიდან ორიგინალი ინფორმაციის აღდგენა და პირიქით რთული სახის ინფორმაციის(გრაფიკული

გამოსახულების, ხმოვანი ინფორმაციის და ა. შ.) ორობით კოდში გადატანა. ამიტომ გრაფიკული რედაქტორების გარეშე შეუძლებელია გრაფიკული ინფორმაციის ბუნებრივი ინტერპრეტაციით აღქმა, მუსიკალური რედაქტორების გარეშე ხმოვანი ინფორმაციის მოსმენა, ტექსტური რედაქტორების გარეშე ტექსტური ინფორმაციის მომზადება და ა.შ. ყველა ეს ჩამოთვლილი რედაქტორები ძალიან „ჭკვიანი“ პროგრამებია, რომლებიც 0-ებისა და 1-ების ოკეანედან აღადგენენ ადამიანისათვის ბუნებრივ ინფორმაციას.

ნებისმიერი ინფორმაცია, რომელიც კომპიუტერში ინახება და რისი დამუშავებაც კომპიუტერის საშუალებით ხდება აუცილებლად წარმოდგენილია ორობით კოდში.

<u>საჭირო ინფორმაცია:</u> ინფორმაციის დამუშავება არის მუსიკის მოსმენა, გრაფიკული ინფორმაციის დათვალიერება, თამაში, ინტერნეტში ინფორმაციის მოძიება, skype-ით და Facebook-ით ჭორაობა, კონკრეტული ამოცანების ამოხსნა შესაბამისი პროგრამების დახმარებით, ექსელის ცხრილების შექმნა და ა.შ ნებისმიერი მოქმედება რასაც კომპიუტერის დახმარებით აკეთებთ ინფორმაციის დამუშავებაა.

ყველაზე რთული ინფორმაციაა გრაფიკული/ხმოვანი ინფორმაცია და ყველაზე მარტივი ინფორმაციაა დადებითი მთელი რიცხვები და სიმბოლური ინფორმაცია. რაც უფრო რთულია ინფორმაცია, მით უფრო რთულია ალგორითმები, რომელთა საშუალებით ეს ინფორმაცია ორობით კოდში ჩაიწერება და პირიქით, ორობითი კოდიდან ორიგინალი ინფორმაციის აღდგენა.

განვიხილოთ მთელი დადებითი რიცხვის ორობით კოდში გადაყვანის და ორობითი კოდიდან მთელი რიცხვის აღდგენის ალგორითმი.

<u>**მთელი დადებითი რიცხვის ორობით კოდში გადაყვანის ალგორითმი**</u>

<u>მოცემული მთელი რიცხვი გავყოთ 2-ზე(რადგან 2-ობით თვლის სისტემაში გადაგვყავს), დავიმახსოვროთ განაყოფის ნაშთი(რომელიც ან 0-ია, ან 1). დავაკვირდეთ განაყოფის მთელ ნაწილს, თუ !=0, პროცესი გავაგრძელოთ და ახლა ის გავყოთ 2-ზე და ა.შ., სანამ განაყოფის მთელ ნაწილში არ მივიღებთ 0-ს. ამის შემდეგ ამოვწეროთ ყველა ნაშთი მარჯვნიდან მარცხნივ თანმიმდევრობით, მიღებული 0-ებისა და 1-ების მიმდევრობა იქნება მოცემული მთელი რიცხვის შესაბამისი ორობითი კოდი.</u>

მაგალითი 1: $(28)_{10}=(\ ?\ )_2$

28:2=14 :2=7:2=3:2=1:2=0

0 0 1 1 1

ე.ი. $(28)_{10}=(\ 011100\ )_2$ (1)

ბაიტში ბიტები გადანომრილია მარჯვნიდან მარცხნივ, 0-დან 7-ის ჩათვლით

<u>ახლა ორობითი კოდიდან აღვადგინოთ ისევ ათობითი რიცხვი.</u>

ორობით რიცხვში ციფრები გადავნომროთ მარჯვნიდან მარცხნივ დაწყებული 0-დან.

$0^5 1^4 1^3 1^2 0^1 0^0$

ახლა ამოვწეროთ 2-ის ხარისხების ჯამი: 2-ის ხარისხი არის ორობითი ციფრის რიგითი ნომერი, ხოლო კოეფიციენტი არის თვითონ ეს ორობითი ციფრი, ე.ი.:

$$(0^5 1^4 1^3 1^2 0^1 0^0)_2 = 0*2^0 + 0*2^1 + 1*2^2 + 1*2^3 + 1*2^4 + 0*2^5 = 0+0+4+8+16+0=(28)_{10} \qquad (2)$$

დავუბრუნდეთ ისე (1) ჩანაწერს. გავიხსენოთ, რომ მეხსიერების უმცირესი ერთეულია 1 ბაიტი, 1 ბაიტში ეტევა 8 ბიტი ინფორმაცია, ჩვენს მაგალითში კი განსაზღვრულია მხოლოდ 6 ბიტი, მე-7 და მე-8 ბიტი რა ინფორმაციით უნდა შევავსოთ? 0-ით, თუ 1-ით?
(2) ჩანაწერი გვკარნახობს, რომ ე.წ. „ზედმეტ“ ბიტებში უნდა ჩავწეროთ 0-ები, მაშინ შედეგი არ დამახინჯდება:

$$(0^7 0^6 0^5 1^4 1^3 1^2 0^1 0^0)_2 = 0*2^0 + 0*2^1 + 1*2^2 + 1*2^3 + 1*2^4 + 0*2^5 + 0*2^6 + 0*2^7 = 0+0+4+8+16+0+0+0=(28)_{10}$$

ეს კი ნიშნავს, რომ $(28)_{10}=(\underbrace{00011100}_{\text{8 ბიტი}})_2$ და არა ~~$(28)_{10}=(011100)_2$~~ (3)

<u>დადებითი რიცხვის შესაბამის ორობით კოდს პირდაპირი კოდი ეწოდება.</u>
დავიმახსოვროთ, თუ დადებითი რიცხვის ორობით კოდში გადაყვანისას განისაზღვრება 8 ბიტზე ნაკლები ბიტები, მაშინ დარჩენილი ბიტები უნდა შეავსოთ 0-ებით.
1 ბაიტში ეტევა მთელი რიცხვი -2^7 - 2^7 - 1(-128 - +127) დიაპაზონში, ამიტომ უფრო დიდი რიცხვებისათვის საჭიროა უფრო მეტი მეხსიერება, ვთქვათ 2ბაიტი, ანუ 16 ბიტი. მაგრამ თუ რიცხვი 16 ბიტში არ ეტევა, მაგალითად 2^{17}, მაშინ რიცხვს უნდა დავუთმოთ 4 ბაიტი, ანუ 32 ბიტი და ა.შ.
ე.ი. მთელი რიცხვი შეიძლება იყოს ან 8 ბიტიანი, ან 16 ბიტიანი, ან 32 ბიტიანი, ან 64 ბიტიანი და ა. შ. არავითარ შემთხვევაში 5ბიტიანი, 13 ბიტიანი და ა.შ.

სიტყვიერი ალგორითმი
1.დასაწყისი
2. შევიტანოთ X;
3.სანამ X != 0
{ a = X % 2;
დავბეჭდოთ a;
X = X / 2;}
4. დასასრული

შესაბამისი პროგრამა

```
#include <iostream>
using namespace std;
main()
 { int a, X;
   cin>>X;
    while(X != 0)
      { a  = X % 2;
        cout<<a;
        X = X / 2;}
 }
```

გავიხსენოთ:
/ - მთელად გაყოფის ოპერაცია, მიიღება განაყოფის მთელი ნაწილი
10 / 3 = 3
% - ნაშთით გაყოფის ოპერაცია, მიიღება განაყოფის ნაშთი
10 % 4 = 1

<u>შენიშვნა:</u> აკრიფეთ მოცემული ფრაგმენტი და ნახეთ შედეგი. ის შედეგი დადგა რასაც ელოდით? რატომ?

უარყოფითი რიცხვის ორობით კოდში გადაყვანის ალგორითმი

იმისათვის, რომ უარყოფითი რიცხვი ორობით კოდში გადავიყვანოთ, უნდა მოვიქცეთ ასე:

1. ავიღოთ შესაბამისი დადებითი რიცხვი და გადავიყვანოთ ორობით კოდში(აუცილებლად უნდა იყოს ან 8 ბიტიანი, ან 16 ბიტიანი, ან 32 ბიტიანი)
2. შემდეგ გავაკეთოთ ე.წ. ინვერსია, ანუ ყველა 1-იანი შევცვალოთ 0-ით, ხოლო ყველა 0 კი 1-ით. ასე მივიღებთ ე.წ. შებრუნებულ კოდს
3. შებრუნებულ კოდს დავუმატოთ 1 ორობით კოდში. რასაც მივიღეთ ეს იქნება უარყოფითი რიცხვის შესაბამისი ორობითი კოდი. **უარყოფითი რიცხვის შესაბამისი ორობით კოდს დამატებითი კოდი ეწოდება.**

მაგალითი 2:

$(-28)_{10}=(\ ?\ )_2$

ჯერ გადავიყვანთ შესაბამისი დადებითი რიცხვი ორობით კოდში:

```
                    00011100        პირდაპირი კოდი
შემდეგი ეტაპი  ინვერსია
                    11100011        შებრუნებული კოდი
                  +
                    00000001
                   ---------
                    11100100        დამატებითი კოდი
```

ორობითი არითმეტიკა
0+0=0
0+1=1
1+0=1
1+1=10

ე.ი. $(-28)_{10}=(\ 11100100)_2$ (4)

შევამოწმოთ ეს მართლაც ასეა თუ არა. ცხადია, რომ ათობით თვლის სისტემაში 28+(-28)=0, ბუნებრივია, რომ ორობით თვლის სისტემაშიც $(28)_2+(-28)_2=0$.

შევამოწმოთ:

```
                    00011100    ⟶    28
                  + 11100100    ⟶    -28
                   ---------
CF⟵  1              00000000
```

ალბათ ყურადღება მიაქციეთ, რომ უკიდურესი მარცხენა 1-იანისათვის ადგილი არ არის, მაგრამ ამასთანავე მიღებული შედეგი არ დამახინჯდა, ამ მოვლენას პროგრამირებაში ეწოდება თანრიგის გადატანა.

ამრიგად, $(28)_2+(-28)_2=0$, ე.ი. უარყოფითი რიცხვის ორობით კოდში გადაყვანის ალგორითმი სწორად მუშაობს.

თუ დავაკვირდებით (3) და (4)-ს შევნიშნავთ, რომ დადებითი რიცხვის შესაბამისს ორობით კოდში უფროსს(მე-7) ბიტში წერია 0, ხოლო უარყოფითი რიცხვი უფროსს ბიტში კი 1, ეს შემთხვევითი არაა.

ორობით კოდში უფროსს ბიტს(მე-7, ან მე-15, ან 31-ე ბიტს) ე.წ. ნიშან-თვისების ბიტი /თანრიგი ეწოდება, თუ ამ ბიტში წერია 1, ეს ნიშნავს რომ რიცხვი დადებითია, თუ 0, მაშინ რიცხვი უარყოფითია.

მაგალითი 3: $(126)_2 + (7)_2 = (01111110)_2 +(00000111)_2$

+ 01111110
00000111
10000001

დააკვირდით შესაკრებების ნიშანს, ორივე შესაკრები დადებითია, აი შედეგი კი უარყოფითია, რაც რა თქმა უნდა სწორი შედეგი არ არის. მაგრამ თუ ამ შეკრების შედეგის შესანახად არა 8 ბიტს, არამედ 16 ბიტს გამოვყოფთ, მაშინ სწორ პასუხს მივიღებთ.

+ 00000000 01111110
00000000 00000111
00000000 10000001

ახლა ყველაფერი რიგზეა, შევკრიბეთ ორი დადებითი რიცხვი და მივიღეთ 16 ბიტიანი დადებითი რიცხვი 133, 16 ბიტიან რიცხვში ნიშან-თვისების ველი ახლა მე-15 ბიტია.

მოვლენას, როცა ორი დადებითი რიცხვის შეკრების / გამრავლების / გაყოფის დროს მიიღება რატომღაც უარყოფითი რიცხვი, ან უარყოფითი რიცხვების შეკრების დროს მიიღება დადებითი რიცხვი, ან უარყოფითი რიცხვების გამრავლება / გაყოფის დროს მიიღება უარყოფითი რიცხვი **გადავსება** ეწოდება. გადავსება საკმაოდ სერიოზული შეცდომაა, რადგან შედეგი არასწორია. გადავსება ნიშნავს, რომ ოპერაციის შედეგად მიღებული შედეგი არ ეტევა მისთვის გამოყოფილ მეხსიერებაში და საჭირო რიცხვს უფრო მეტი მეხსიერება(ბაიტები) დავუთმოთ/გამოვუყოთ.

და ბოლოს კიდევ ერთი
შენიშვნა: ზემოთ ვთქვით, რომ ყველაზე რთული გრაფიკული და ხმოვანი ინფორმაციის ორობით კოდში(და პირიქით) გადაყვანის ალგორითმებია. რა თქმა უნდა ამ ალგორითმებს ამ ლექციაში არ განვიხილავთ, უბრალოდ ავღნიშნოთ ის ფაქტი, თუ რა პრინციპით წარმოიდგინება გრაფიკული და ხმოვანი ინფორმაცია.
გრაფიკული ინფორმაციის უკან განიხილება ე.წ. ფიქსელების მატრიცა და მატრიცის ყოველი ელემენტი(ანუ ფიქსელი) ხასიათდება მხოლოდ ფერით. ფიქსელების ფერების ურთიერთშენაცვლებით მიიღება გრაფიკული ნახატის ილუზია ეკრანზე.

ხმოვანი ინფორმაციაც განიხილება როგორც ფიქსელების მატრიცა და მატრიცის ყოველი ელემენტი(ანუ ფიქსელი) ამ შემთხვევაში ხასიათდება ბგერის ამპლიტუდით და სიხშირით. გაითვალისწინეთ, რომ კომპიუტერული ხმა დისკრეტულია, ანუ წყვეტილია. ხმაში დისკრეტულობას ადამიანის სმენის აპარატი(ყური) ვერ აღიქვამს და ადამიანის ყურისათვის ხმა თითქოს უწყვეტია.
დავალება: აიღეთ 5 დადებითი და 5 უარყოფითი რიცხვი და გადაიყვანეთ ორობით კოდში. დარჭმუნდით, რომ თქვენს მიერ მიღებული ორობითი კოდი სწორია.
შეკრიბეთ და გამოაკელით ეს რიცხვები ერთმანეთს ორობით კოდში.

ლაბორატორიული

I და II ლექციის ტექსტში მოყვანილი ყველა პროგრამა აკრიფეთ, გამართეთ, თუ რაიმე სინტაქსური შეცდომაა გაპარული, გაასწორეთ და პასუხი მიიღეთ.

ამოცანა 1: მოცემული გაქვთ სამი რიცხვი, დაალაგეთ ზრდადობით ეს რიცხვები ზრდადობით.
ამოხსნა: ვთქვათ მოცემული გვაქვს სამი რიცხვი a, b, c(შეგახსენებთ , რომ თქვენი ალგორითმი უნდა იყოს ზოგადი, ამიტომ იგი უნდა მუშაობდეს ნებისმიერი a, b, c-თვის). ჩვენი ალგორითმის მუშაობის შემდეგ a, b, c რიცხვები ზრდადობით უნდა იყოს დალაგებული, ე.ი. a<b<c. ამ ფაქტის გათვალიწინებით, უნდა დავიწყოთ იმის გარკვევა რომელიმე ორი რიცხვი ამ სამიდან ხომ არ არღვევს ზრდადობის პირობას.

a	b	c	
77	19	-7	77>19, ე.ი. a>b ანუ a და b არღვევს ზრდადობის პირობას, ამიტომ a <-> b
19	77	-7	უკვე a<b, ახლა შევადაროთ b და c, რადგან b>c ამიტომ b <-> c
19	-7	77	მართალია b<c, მაგრამ ისევ a>b, ამიტომ a <-> b
-7	19	77	

სიტყვიერი ალგორთმი

1. დასაწყისი
2. შევიტანოთ a, b, c
3. თუ a>b, მაშინ {k=a; a=b; b=k;}
4. თუ b>c, მაშინ {k=b; b=c; c=k;}
5. თუ a>b, მაშინ {k=a; a=b; b=k;}
6. გამოვიტანოთ a, b, c
7. დასასრული

პროგრამა

```
#include <iostream>
using namespace std;
main()
{float a, b, c , k;   // a, b, c საწყისი მონაცემებია
                      // k  კი დამხმარე ცვლადია
cin>>a>>b>>c;
if (a>b)   {k=a; a=b; b=k;}
if (b>c)   {k=b; b=c; c=k;}
if (a>b)   {k=a; a=b; b=k;}              // a<b<c
cout<<a<<" "<<b<<" "<<c;
}
```

დავალება 1: დაალაგეთ სამი რიცხვი კლებადობით.

ამოცანა 2: მოც. გაქვთ სამი რიცხვი, რომლებიც სამკუთხედის გვერდების სიგრძეების როლში განიხილეთ. გაარკვიეთ ამ გვერდებიდან აიგება თუ არა სამკუთხედი და თუ აიგება რა ტიპისაა ეს სამკუთხედი.
ამოხსნა: ვთქვათ ეს სამი რიცხვია a, b, c. გავიხსენოთ სამკუთხედის აგების პირობა: ნებისმიერი ორი გვერდის ჯამი მესამე გვერდზე მეტი უნდა იყოს, ე.ი. თუ a+b > c და a+c > b და b+c > a, მაშინ სამკუთხედი აიგება, თუ არა არ აიგება.
თუ რა ტიპისაა სამკუთხედი, ამის გასარკვევად გამოვიყენოთ პითაგორას თეორემა: თუ a*a+b*b ==c*c ან a*a+c*c== b*b ან b*b+c*c == a*a, მაშინ სამკუთხედი მართკუთხაა, წინააღმდეგ შემთხვევაში კი ან მახვილკუთხაა, ან ბლაგვკუთხა.

სიტყვიერი ალგორთმი

1. დასაწყისი

2. შევიტანოთ a, b, c
3. თუ a+b>c && a+c>b && b+c>a მაშინ

3.1{სამკუთხედი აიგება;

3.1.1 თუ a*a+b*b ==c*c || a*a+c*c== b*b || b*b+c*c == a*a Δ მართკუთხაა ;
წინააღმდეგ შემთხვევაში

3.1.2 თუ a*a+b*b >c*c || a*a+c*c> b*b || b*b+c*c > a*a Δ ბლაგვკუთხაა;

3.1.3 წინააღმდეგ შემთხვევაში Δ მახვილკუთხაა;

3.2 წინააღმდეგ შემთხვევაში სამკუთხედი არ აიგება;

4. დასასრული

პროგრამა

```
#include <iostream>
using namespace std;
main()
{float a, b, c;
cin>>a>>b>>c;
if (a+b>c && a+c>b && b+c>a) {cout<<“samkuTxedi aigeba da igi ”;
                                if (a*a+b*b ==c*c || a*a+c*c== b*b || b*b+c*c ==
a*a)cout<<“marTkuTxaa ”;
                                  else
                                  if (a*a+b*b >c*c || a*a+c*c> b*b || b*b+c*c > a*a)cout<<“blagvkuTxaa;
                                     else       cout<<“maxvilkuTxaa”;    }
    else cout<<“samkuTxedi ar aigeba”;}
```

იგივე ამოცანის მეორე ალგორითმი.
თუ მოცემულ a, b, c დავალაგებთ ზრდადობით, მაშინ ალგორითმის მსვლელობა უფრო გამარტივდება.
a+b>c && a+c>b && b+c>a მაგივრად საკმარისია დავსვათ ერთი კითხვა a+b>c(იფიქრეთ რატომ)
a*a+b*b ==c*c || a*a+c*c== b*b || b*b+c*c == a*a მაგივრად მხოლოდ a*a+b*b ==c*c და ა.შ.

სიტყვიერი ალგორითმი

1. დასაწყისი
2. შევიტანოთ a, b, c
3. დავალაგოთ a, b, c ზრდადობით //იხილეთ ამოცანა 1
4. თუ a+b>c მაშინ 4.1{სამკუთხედი აიგება;

4.1.1 თუ a*a+b*b ==c*c Δ მართკუთხაა ;
წინააღმდეგ შემთხვევაში

4.1.2 თუ a*a+b*b >c*c Δ ბლაგვკუთხაა;

4.1.3 წინააღმდეგ შემთხვევაში Δ მახვილკუთხაა;

4.2 წინააღმდეგ შემთხვევაში სამკუთხედი არ აიგება;

5. დასასრული

შესაბამისი პროგრამა:

```
#include <iostream>
using namespace std;
main()
{float a, b, c,k;
cin>>a>>b>>c;
// დავალაგოთ a, b, c ზრდადობით
if (a>b)   {k=a; a=b; b=k;}
if (b>c)   {k=b; b=c; c=k;}
if (a>b)   {k=a; a=b; b=k;}
//          a<b<c
if (a+b>c ) {cout<<“samkuTxedi aigeba da igi ”;
                                    if (a*a+b*b ==c*c )cout<<“marTkuTxaa ”;
                                      else
                                      if (a*a+b*b >c*c)cout<<“blagvkuTxaa;
                                          else       cout<<“maxvilkuTxaa”;    }
    else cout<<“samkuTxedi ar aigeba”;}
```

დავალება 2: შეადარეთ ეს ორი ალგორითმი, დაითვალეთ ბიჯების რაოდენობა, რომელი ალგორითმი უფრო სწრაფად იმუშავებს?

დავალება 3: გაარკვიეთ, საჭადრაკო დაფაზე (x, y)უჯრაზე მდგომი ცხენი ერთ სვლაში მოკლავს, თუ არა (x_1, y_1) უჯრაზე მდგომ ფიგურას.

დავალება 4: მოცემული გაქვთ, რომ x, y და z მთელი რიცხვებია. იპოვეთ შემდეგი გამოსახულების მნიშვნელობა:

$$\frac{\max^2(x,y,z) - 2^x * \min(x,y,z)}{\min(x,z)*\max(y,z)}$$

ამოცანა 3: (მთელი რიცხვის დაშლა ციფრებად დაშლა)მოცემული გაქვთ მთელი რიცხვი, იპოვეთ ამ რიცხვის ციფრთა ჯამი.

ამოხსნა: ამოცანის ალგორითმი განვიხილოთ რაიმე კონკრეტულ რიცხვზე და შემდეგ განვაზოგადოთ. ვთქვათ საწყისი მონაცემია x და მისი მნიშვნელობაა 379. ცხადია, რომ გამოსახულების x % 10 მნიშვნელობა იქნება 9, ანუ ერთეულების თანრიგი. ახლა x-ის მნიშვნელობა შევცვალოთ შემდეგი წესით x / 10, თუ მიაქცევთ ყურადღებას x-ის ახალი მნიშვნელობა იქნება 37, ანუ ერთეულების თანრიგი აღარ არის.

X	მორიგი ციფრი
379	9
37	7

3 0	3

ცხადია, ალგორითმი მუშაობას დაასრულებს, როცა x-ის ახალი მნიშვნელობა გახდება 0-ის ტოლი და ალგორითმი მუშაობს მანამ სანამ x !=0-გან, ე.ი. გვაქვს ციკლური ალგორითმი

სიტყვიერი ალგორითმი	შესაბამისი პროგრამა

1. დასაწყისი
2. შეიტანეთ x;
3. s=0;
4. სანამ x !=0
{k = x % 10;
s = s + k;
x = x / 10;}
5. გამოიტანეთ s;
6. დასასრული
	#include <iostream>
	using namespace std;
	main()
	{int x,s,k;
	cin>>x;
	s=0;
	while (x !=0)
	{k = x % 10; //მორიგი ციფრი
	s = s + k;
	x = x / 10;} /* x შეიცვალა, ანუ ბოლო ციფრი ჩამოსცილდა */
	cout<<"cifrTa jami="<<s;
	}

დავალება 5: ა) მოცემული გაქვთ მთელი რიცხვი, იპოვეთ ამ რიცხვის მხოლოდ ლუწი(კენტი) ციფრთა ნამრავლი.

ბ) გაარკვიეთ მოცემული მთელი რიცხვის ციფრთა ჯამი ბოლო ციფრის გარეშე კენტია თუ ლუწი.

III – IV ლექცია
ევკლიდეს ალგორითმისა და ფიბონაჩის მიმდევრობის აგების ალგორითმის გარჩევა. მთელი რიცხვის მარტივობა/ შედგენილობის გასარკვევი ალგორითმი.

ევკლიდეს ალგორითმი. ამ ალგორითმზე უკვე ვილაპარაკეთ, მაგრამ ისევ გავიმეორებ. ჩვენს წელთაღრიცხვამდე III საუკუნეში ბერძენმა მათემატიკოსმა ევკლიდემ თავის ფუნდამენტურ ნაშრომში "საწყისები" ჩამოაყალიბა ორი ნატურალური რიცხვის უდიდესი საერთო გამყოფის პოვნის ალგორითმი, რომელსაც საფუძვლად გეომეტრიული მოსაზრებები
დაედო. თუმცა, ძველი ბერძენი მათემატიკოსებისათვის ეს ალგორითმი ევკლიდემდე იყო ცნობილი "ურთიერთგამოკლების" წესის სახელწოდებით და ის არისტოტელეს ერთ-ერთ ნაშრომშია აღწერილი და ეს ალგორითმი მხოლოდ ნატურალურ რიცხვთა სიმრავლეზე იყო განმარტებული.

ვთქვათ, მოცემული გვაქვს ორი ნატურალური რიცხვი

a	b	
48	27	48 != 27 48>27
48-27	27	
21	27	21 != 27 21<27
21	27-21	
21	6	21 != 6 21>6
21-6	6	
15	6	15 != 6 15>6
15-6	6	
9	6	9 != 6 9>6
9-6	6	
3	6	3 != 6 3<6
3	6-3	
3	3	3 == 3

პროცესი შეწყდა, 48 და 27-ის უდიდესი საერთო გამყოფია 3

ალგორითმის ყოველ ნაბიჯზე უფრო დიდ რიცხვს აკლდება უფრო ნაკლები რიცხვი და შედეგი იწერება დიდი რიცხვის ადგილზე, ევკლიდეს ალგორითმი გრძელდება მანამ სანამ ერთმანეთის ტოლ მნიშვნელობებს არ მივიღებთ.

სიტყვიერი ალგორითმი

6. დასაწყისი
7. შევიტანოთ a,b;

8. სანამ a !=b
   თუ a>b, მაშინ a= a – b;
   თუ არა b = b - a;
9. გამოვიტანოთ a ან b;
10. დასასრული

ახლა ვნახოთ რა მოხდება, თუ a და b იქნება არა ნატურალური რიცხვთა სიმრავლიდან, არამედ მთელ რიცხვთა სიმრავლიდან.

a	b	
48	-27	48 !=- 27 48>-27
48-(-27)	-27	
75	-27	75 != -27 75>-27
75-(-27)	-27	
102	-27	102 != -27 102>-27
102-(-27)	-27	
129	-27	129 != -27 129>-27

ზემოთ მოყვანილი ილუსტრაციიდან ჩანს, რომ ალგორითმის მსვლელობისას მოყვანილი ორი რიცხვიდან ერთი გამუდმებით იზრდება, მეორე კი უცვლელი რჩება და ალგორითმი არასდროს არ შეწყდება(ჩაიციკლება), რადგან ეს ორი რიცხვი ერთმანეთს კი არ უახლოვდება, არამედ უფრო შორდება ერთმანეთს. ე.ი. ევკლიდისეული ალგორითმი მთელ რიცხვთა სიმრავლეზე არ მუშაობს.

დავუბრუნდეთ ისევ ჩვენ ნატურალურ რიცხვთა წყვილს:
48 და 27-ის საერთო გამყოფებია: 1, -1, 3, -3; უდიდესი საერთო გამყოფია 3
-48 და 27-ის საერთო გამყოფებია: 1, -1, 3, -3; უდიდესი საერთო გამყოფია 3
48 და -27-ის საერთო გამყოფებია: 1, -1, 3, -3; უდიდესი საერთო გამყოფია 3
-48 და -27-ის საერთო გამყოფებია: 1, -1, 3, -3; უდიდესი საერთო გამყოფია 3

ახლა გავაკეთოთ დასკვნა, თუ a და b მთელი რიცხვებია, მაშინ მათი უდიდესი საერთო გამყოფი აუცილებლად დაემთხვევა |a| და |b| უდიდესს საერთო გამყოფს. ამრიგად თუ ევკლიდეს ალგორითმში, პროცესს ჩავატარებთ |a| და |b|-თვის, შევძლებთ ვიპოვოთ a და b ნებისმიერი მთელი რიცხვების უდიდესს საერთო გამყოფს.

ბოლოს შევცვალოთ ევკლიდეს ალგორითმი და ის მოვარგოთ მთელ რიცხვთა სიმრავლეს.

სიტყვიერი ალგორითმი	შესაბამისი პროგრამა C-ზე	პროგრამის სხვა ვერსია
1. დასაწყისი	#include <iostream>	#include <iostream>
2. შევიტანოთ a,b;	#include <math>	#include <math>
3. a=abs(a); b=abs(b);	**using namespace** std;	**using namespace** std;
		main()

4. სანამ a !=b
   თუ a>b, მაშინ a= a – b;
   თუ არა b = b - a;
5. გამოვიტანოთ a ან b;
6. დასასრული

```
main()
{  int a,b,c,d;
   cin>>a>>b;
c=abs(a);
d=abs(b);
   while( c != d )
     if( c>d)  c-=d;
        else    d-=c;
cout<<a<<"-ს  და  "<<b<<"-ს
უსგ= "<<d;}
```

```
{  int a,b,c,d;
   cin>>a>>b;
for(c=abs(a), d=abs(b); c != d;)
     if( c>d)  c-=d;
        else    d-=c;
cout<<a<<"-ს  და  "<<b<<"-ს
უსგ= "<<d;
}
```

შენიშვნა 1: ჩვენ ევკლიდეს ალგორითმი შევცვალეთ და იგი გადავიტანეთ მთელ რიცხვთა სიმრავლეზე, ანუ ახლა ევკლიდეს ალგორითმის საგნობრივი არეა არა მხოლოდ ნატურალურ რიცხვთა სიმრავლე, არამედ მთელ რიცხვთა სიმრავლე. ე.ი. ალგორითმის მუშაობის არეალი გავაფართოვეთ. რაც უფრო დიდია ამა თუ იმ ალგორითმის საგნობრივი არე, მით უფრო ზოგადია ალგორითმი.

შენიშვნა 2: ზემოთ ხედავთ, რომ სიტყვიერი ალგორითმი ერთია, ხოლო შესაბამისი პროგრამული რეალიზაცია 2, შეიძლება იყოს უფრო მეტიც; ეს დამოკიდებულია იმაზე თუ როგორ ფლობს პროგრამისტი დაპროგრამების ამა თუ იმ ენას, როგორც ინსტრუმენტს. უნდა გვახსოვდეს, რომ მხოლოდ პროგრამულად ჩაწერილი ალგორითმის რეალიზაციაა შესაძლებელი კომპიუტერის დახმარებით და მოსალოდნელი შედეგების მიღება.

საზოგადოდ, ამოცანის ამოხსნა კომპიუტერის დახმარებით, მოიცავს შემდეგ ეტაპებს:

1. ამოცანის დასმა, ფორმულირება;
2. შესაბამისი საგნობრივი არის განსაზღვრა და შესწავლა, თუ არ იცნობთ ამ საგნობრივ არეს;
3. შესაბამისი მათემატიკური მოდელის შედგენა;
4. ალგორითმის ფორმულირება / ჩამოყალიბება;
5. ალგორითმის ჩაწერა რომელიმე ალგორითმულ ენაზე, რომელსაც უკეთესად ფლობთ(კიდევ ერთხელ დაპროგრამების ენები ალგორითმის ჩაწერის მხოლოდ ინსტრუმენტია);
6. და ბოლოს პროგრამის სახით ჩაწერილი ალგორითმის რეალიზაცია, კომპიუტერის დახმარებით.
   6.1 გამართვა
   6.2 პასუხის მიღება

დავალება1: ევკლიდეს ალგორითმის რეალიზაციის ზემოთ მოყვანილ ორივე პროგრამაში დაითვალეთ ოპერაციათა რაოდენობა და შეაფასეთ ორივე პროგრამის დროითი სირთულე.

დავალება2: ევკლიდეს ალგორითმის დახმარებით იპოვეთ სამი დადებითი რიცხვის უდიდესი საერთო გამყოფი.

ახლა განვიხილოთ ევკლიდეს ალგორითმის სხვა ვერსია, რომელიც ურთიერთგაყოფის

ალგორითმის სახელით არის ცნობილი.

a	b
48	27
48 % 27	27
21	27
21	27 % 21
21	6
21 % 6	6
3	6
3	6 % 3
3	0

თუ a>b, მაშინ a = a % b=48 % 27 = 21 !=0

თუ a<b, მაშინ b= b % a = 27 % 21 =6 !=0

21>6

21 % 6 = 3 !=0

6 > 3

6 % 3 = 0

პროცესი შეწყდა, 48 და 27-ის უდიდესი საერთო გამყოფია 3

სიტყვიერი ალგორითმი

1. დასაწყისი
2. შევიტანოთ a,b;
3. სანამ a !=0 და b !=0
   თუ a>b, მაშინ a= a %b;
   თუ არა b = b % a;
4. თუ a!=0 გამოვიტანოთ a;
   თუ არა გამოვიტანოთ b;
7. დასასრული

შესაბამისი პროგრამა C-ზე

```
#include <iostream>
#include <math>
using namespace std;
main()
{  int a,b,c,d;
  cin>>a>>b;
c=abs(a);
d=abs(b);
  while( c != 0 && d!=0 )
    if( c>d)  c=c % d;
       else    d=d % c;
if(c!=0) cout<<c;
      else cout<<d; }
```

პროგრამის სხვა ვერსია

```
#include <iostream>
#include <math>
using namespace std;
main()
{  int a,b,c,d;
  cin>>a>>b;
for(c=abs(a),  d=abs(b);  c  !=0  &&
d=0;)
    if( c>d)  c=c % d;
       else    d=d % c;
if(c!=0) cout<<a<<"-ს და "<<b<<"-ს
                        უსგ= "<<c;
else   cout<<a<<"-ს   და   "<<b<<"-ს
                       უსგ= "<<d;
}
```

კითხვა: რომელი ალგორითმი იმუშავებს უფრო სწრაფად - ევკლიდეს ურთიერთგამოკლების ალგორითმი, თუ ევკლიდეს ურთიერთგაყოფის ალგორითმი?

ფიბონაჩის მიმდევრობა, ფიბონაჩის რიცხვები

ფიბონაჩის მიმდევრობა ანუ ფიბონაჩის რიცხვები ეწოდება ნატურალურ რიცხვთა შემდეგ მიმდევრობას

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, … (1)

ფორმალურად ფიბონაჩის მიმდევრობა შეიძლება ჩავწეროთ შემდეგი რეკურენტული ფორმულის დახმარებით:

$$F_n = F_{n-1} + F_{n-2};\quad n=2,3,\ldots;\ F_0 = F_1 = 1;$$

ანუ ფიბონაჩის მიმდევრობაში, მიმდევრობის ყოველი მორიგი წევრი წინა ორი წევრის ჯამის ტოლია.

რიცხვთა ასეთ მიმდევრობას ეწოდება ფიბონაჩის მიმდევრობა, შუასაუკუნეების (XIII) მათემატიკოსის ლეონარდო ფიბონაჩის პატივსაცემად. თუმცა მე-13 საუკუნემდე ამ რიცხვებს იცნობდნენ ძველ ინდოეთში და მას ლექსთწყობაში იყენებდნენ.

ფიბონაჩიმ ეს რიცხვები შემოიტანა თავის ნაშრომში „«Liber Abaci“(აბაკის წიგნი) 1202 წელს. ის ცდილობდა ამ რიცხვების დახმარებით აღეწერა კურდღლების პოპულაციის ზრდის კანონზომიერება.

ფიბონაჩის რიცხვები აღწერს სხვადასხვა მოვლენებს, ხელოვნებაში, მუსიკაში, ბუნებაში. მაგალითად, ნაძვის გირჩებზე და ანანასზე სპირალების რაოდენობა ხშირად აღიწერება ფიბონაჩის მიმდევრობის რიცხვებით, მზესუმზირები ფიბონაჩის მიმდევრობის ყველაზე თვალსაჩინო მაგალითია. მზესუმზირის მარცვლები სპირალზე სწორედ ფიბონაჩის მიმდევრობის კანონზომიერებით არის განლაგებული.

მათემატიკოსები განიხილავენ ფიბონაჩის მიმდევრობის ასეთ ვარიანტსაც:

0,1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, … (2)

$$F_n = F_{n-1} + F_{n-2};\quad n=2,3,\ldots;\ F_0 = 0;\ F_1 = 1;$$

თუ შევადარებთ (1) და (2) მიმდევრობას, ვნახავთ რომ ეს მიმდევრობები ბევრად არ განსხვავდება ერთმანეთისაგან: (1) მიმდევრობა, მხოლოდ ერთი ნაბიჯით უსწრებს (2) მიმდევრობას.

<u>ამოცანა 1:</u> ავაგოთ ფიბონაჩის მიმდევრობის პირველი n წევრი.

ცხადია, ნავიგაცია უნდა დავიწყოთ $F_0 = F_1 = 1$-დან და ვიპოვოთ $F_2 = F_0 + F_1 = 2$;

							F_0	F_1	F_2
1	1	2	3	5	8	13	1	1	2
I ნაბიჯი F_0	F_1	F_2					1	2	3
II ნაბიჯი	F_0	F_1	F_2				2	3	5
III ნაბიჯი		F_0	F_1	F_2			3	5	8
IV ნაბიჯი			F_0	F_1	F_2		5	8	13
V ნაბიჯი F_2				F_0	F_1				

1. დასაწყისი
2. შევიტანოთ n;
3. k=2; // მთვლელი, მიუთითებს ფიბონაჩის მერამდენე რიცხვი ვიპოვეთ
4. $F_0 = F_1 = 1$;
5. სანამ (k<=n)
   { $F_2 = F_0 + F_1$;
   K++;
   გამოვიტანოთ F_2;
   $F_0 = F_1$;
   $F_1 = F_2$; }
6. დასასრული

ვთქვათ, n=100. დააკვირდით ალგორითმს, იმის მაგივრად, რომ 100 ნაბიჯი გაგვეწერა, გავაციკლეთ პროცესი:

{ $F_2 = F_0 + F_1$;
K++;
გამოვიტანოთ F_2;
$F_0 = F_1$;
$F_1 = F_2$; }

და იმის ხარჯზე, რომ ციკლის ყოველ ნაბიჯზე F_0 და F_1-ის მნიშვნელობები იცვლება, F_2-ის მნიშვნელობა იქნება ფიბონაჩის მიმდევრობის მორიგი ელემენტი.

I ვარიანტი

```
#include <iostream>
using namespace std;
main()
{  int  F0, F1, F2, n,k;
cin>>n;
  F0 = F1 =1;
  k=2;
  cout<<F0<<"  "<<F1;
  while(k<=n)
    {F2 = F0 +F1;
     k++;
     cout<<"  "<<F2;
     F0=F1;
     F1=F2;  }
  }
```

II ვარიანტი

```
#include <iostream>
using namespace std;
main()
{  int  F0, F1, F2, n,k;
  cin>>n;
  cout<<"1  1 ";
for(F0 = F1 =1, k=2; k<=n; F0=F1, F1=F2, k++)
     {F2 = F0 +F1;
      cout<<"  "<<F2;  }
  }
```

დავალება 3: შეაფასეთ ამ ორი პროგრამის დროითი სირთულე, რომელი უფრო სწრაფად იმუშავებს?

დავალება 4*: მოცემული გაქვთ ნატურალური რიცხვი. გაარკვიეთ, არის თუ არა ის ფიბონაჩის რიცხვი.

დავალება 5: მოცემული გაქვთ მთელი ტიპის მასივი, გაარკვიეთ ეს მასივი წარმოადგენს თუ არა ფიბონაჩის მიმდევრობას.

დავალება 6: შეგაქვთ n. იპოვეთ 1-დან n-მდე მხოლოდ კენტი(ლუწი) ფიბანაჩის რიცხვების საშუალო არითმეტიკული.

მარტივი და შედგენილი რიცხვები

მარტივია ისეთი მთელი რიცხვი, რომელსაც **მხოლოდ** ორი გამყოფი აქვს. აქ სიტყვა მხოლოდ მნიშვნელოვანია, იმიტომ რომ ყველა მთელ რიცხვს აუცილებლად აქვს ორი გამყოფი: 1 და თავისი თავი. ხოლო ის მთელი რიცხვები, რომელთაც ამ ორი გამყოფის გარდა სხვა გამყოფი არ გააჩნიათ მარტივი რიცხვებია, ყველა დანარჩენი რიცხვები შედგენილი რიცხვებია.

ამრიგად იმისათვის, რომ გავარკვიოთ მთელი რიცხვის მარტივობა/ შედგენილობის საკითხი, საჭიროა ამ რიცხვის გამყოფების რაოდენობის დათვლა. თუ ეს გამყოფების რაოდენობა ზუსტად 2-ის ტოლია, მაშინ რიცხვი მარტივია, ხოლო თუ გამყოფების რაოდენობა 2-ზე მეტია, მაშინ რიცხვი შედგენილია.

ეს არის მოსაზრება, რომელიც მარტივი რიცხვის განმარტებიდან გამომდინარეობს. პირველ ჯერზე სწორედ ამ მოსაზრებაზე დაყრდნობით ჩამოვაყალიბოთ შესაბამისი ალგორითმი.

სანამ უშუალოდ ალგორითმზე გადავალთ, გავარკვიოთ რიგი წვრილმანი პრობლემები.

ვთქვათ მოცემული გვაქვს რაღაც მთელი რიცხვი X, სად უნდა ვეძებოთ ამ რიცხვის გამყოფები? ალბათ 1-დან X-ის ჩათვლით, მაგრამ რადგან ცნობილი ფაქტია, რომ ყველა მთელი რიცხვი იყოფა 1-ზე და თავის თავზე, ამიტომ 1 და X შეიძლება თამაშგარე მდგომარეობაში გავიტანოთ, ე.ი. X-ის გამყოფები მოთავსებულია 2-დან X-მდე, თუ კიდევ გავაგრძელებთ ფიქრს, დავადგენთ, რომ X-ის გამყოფები მოთავსებულია 2-დან X/2-ის ჩათლით.

ალგორითი

1. დასაწყისი
2. შევიტანოთ X;
3. Count=2; //Count გამყოფების მთვლელი
4. K =2; // X-ის სავარაუდო გამყოფი
5. სანამ K<= X/2 მანამ
   {თუ X % K ==0, მაშინ Count++;

```
#include <iostream>
using namespace std;
main()
{  int Count, X, k;
   cin>>X;
   Count = 2;
   K=2;
   while(K<=X/2)
      {if(X % K ==0) Count++;
```

ან

```
#include <iostream>
using namespace std;
main()
{  int Count, X, k;
   cin>>X;
for(K=2,Count =2; K<=X/2; K++)
      if(X % K ==0) Count++;
```

K++;}
6. თუ Count==2, მაშინ X მარტივია,
   თუ არა X შედგენილია
7. დასასრული

```
      K++;}
if(Count==2)
        cout<<X<<" -   martivia";
 else  cout<< X<<" - Sedgenilia";
                }
```

```
if(Count==2)
      cout<<X<<" -   martivia";
    else  cout<< X<<" - Sedgenilia"; }
}
```

მოდით, ახლა შევცვალოთ ალგორითმი: X მთელი რიცხვი შედგენილია, თუ მას 2-დან X/2-ის ჩათლით შუალედში ერთი მაინც გამყოფი აქვს და X არის მარტივი თუ ამ შუალედში არ მოიძებნა X-ის არც ერთი გამყოფი.

ალგორითი
1. დასაწყისი
2. შევიტანოთ X;
3. Count=0;
4. K =2; // X-ის სავარაუდო გამყოფი
5. სანამ (K<= X/2) && (Count==0) მანამ
   {თუ X % K ==0, მაშინ Count=1;
   K++;}
6. თუ Count==0, მაშინ X მარტივია,
   თუ არა X შედგენილია

დასასრული

```
#include <iostream>
using namespace std;
main()
{  int Count, X, k;
   cin>>X;
   Count = 2;
   K=2;
   while(K<=X/2 && Count==0)
      {if(X % K ==0) Count=1;
       K++;}
 if(Count==0)    cout<<X<<"  -
                          martivia";
      else       cout<<  X<<"  -
                  Sedgenilia"; }
```

ან

```
#include <iostream>
using namespace std;
main()
{  int  X, k; bool t=false;
   cin>>X;
   K=2;
   while(K<=X/2 && t==false)
      {if(X % K ==0) t=true;
       K++;}
 if(t==false)    cout<<X<<"  -
                          martivia";
      else  cout<< X<<" -
Sedgenilia"; }
```

დავალება: შეადარეთ ზემოთ მოყვანილი ორი ალგორითმი, რომელი იმუშავებს უფრო სწრაფად? X=100 000-თვის დაითვალეთ ოპერაციათა რაოდენობა, ორივე ალგორითმისათვის და შეაფასეთ დროითი სირთულე.

დავალება: მოცემული გაქვთ მთელი რიცხვი. გაარკვიეთ

- ეს რიცხვი პალინდრომია თუ არა(პალინდრომია რიცხვი, რომელიც ერთნაირად იკითხება მარცხნიდან მარჯვნივ და მარჯვნიდან მარცხნივ. მაგალითად, ასეთია: 1221, 777, 35953).
- ეს რიცხვი სრულყოფილი რიცხვია თუ არა (რიცხვი სრულყოფილია, თუ იგი ემთხვევა თავისი გამყოფების(გარდა საკუთარი თავისა) ჯამს; მაგ. 6=1+2+3 – 6 სრულყოფილი რიცხვია).

- ეს რიცხვი მერსენის რიცხვია თუ არა(რიცხვს ეწოდება მერსენის რიცხვი, თუ იგი წარმოიდგინება როგორც $2^n - 1$; მაგ. $127=2^7 - 1$).

დავალება:მოცემული გაქვთ ორი მთელი რიცხვი, გაარკვიეთ ისინი ერთმანეთის მეგობრები არიან თუ არა(ორ რიცხვს ერთმანეთის მეგობრები ეწოდება, თუ პირველი რიცხვის გამყოფების ჯამი ემთხვევა მეორე რიცხვის გამყოფების ჯამს. მაგ., 220 და 284 ერთმანეთის მეგობარი რიცხვებია).

დავალება: მოცემული გაქვთ მთელი რიცხვი. გაარკვიეთ, ჰყავს თუ არა მას მეგობარი და თუ ჰყავს, დაბეჭდეთ მისი მეგობარი.

დავალება: დაბეჭდეთ ყველა ოთხნიშნა რიცხვი, რომელთა ათობით ჩანაწერში ერთნაირი ციფრები არ გხვდებათ(არ გამოიყენოთ ოპერაციები / და %).

დავალება: მოცემული გაქვთ ნატურალური რიცხვი n. იპოვეთ კვადრატული ფესვი n-დან (გამოიყენეთ ის ფაქტი, რომ კენტი რიცხვების ჯამი გვაძლევს რიცხვის სრულ კვადრატს; მაგ. $1+3+5=3^2$, $1+3+5+7+9+11+13+15+17+19=10^2$)

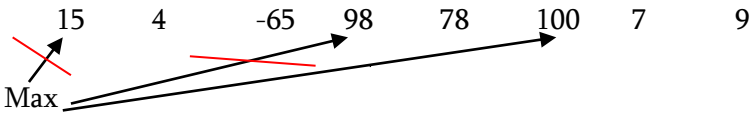
ლექცია V. რიცხვით მიმდევრობაში უდიდესი და უმცირესი ელემენტის პოვნის ალგორითმი. რამდენიმე მაქსიმალურისა და რამდენიმე მინიმალური ელემენტის პოვნის ალგორითმი

რიცხვით მიმდევრობაში უდიდესი და უმცირესი ელემენტის პოვნის ალგორითმი

რიცხვით მიმდევრობაში(მასივში) უდიდესი(მაქსიმალური) და უმცირესი(მინიმალური) ელემენტის პოვნის ალგორითმები ძალიან მარტივ ალგორითმების კლასს მიეკუთვნება. მასივში მაქსიმალური(მინიმალური) ელემენტის პოვნა:

- დავუშვათ მასივის პირველი ელემენტია ამ მასივში მაქსიმალური(მინიმალური) ელემენტი;
- ჩვენს მიერ არჩეული მაქსიმალური(მინიმალური) ელემენტი შევადაროთ მასივის ყველა დანარჩენ ელემენტებს, თუ მასივის რომელიმე ელემენტი ჩვენს მიერ არჩეულ მაქსიმალურ(მინიმალურ) ელემენტზე მეტია(ნაკლებია), მაშინ მაქსიმალურ (მინიმალურ) ელემენტად ის ჩავთვალოთ;
- როცა მასივის ყველა ელემენტს დავათვალიერებთ, ჩვენს ხელთ მაქსიმალური (მინიმალური) ელემენტი იქნება.

მასივში მაქსიმალური(მინიმალური) ელემენტის პოვნის ზოგად სულიკვეთებას ზემოთ მოყვანილი ნაბიჯების მიმდევრობა ზუსტად ასახავს.
განვიხილოთ მაგალითი:



ნახაზიდან ჩანს, რომ Max ცვლადის მნიშვნელობა იცვლება, როგორც კი მასზე მეტ მნიშვნელობას მივაგნებთ მასივში, როცა მასივის ბოლომდე გადათვალიერებას დავასრულებთ Max ცვლადის მნიშვნელობა იქნება მასივის ყველაზე დიდი ელემენტის მნიშვნელობა. ყურადღება მიაქციეთ იმ ფაქტს, რომ მასივის დანარჩენ ელემენტებს ვადარებთ არა A[0]-ს, არამედ ცვლადს Max, რომლის მნიშვნელობა პირველ ნაბიჯზე ემთხვევა A[0]-ს. მე-3 ნაბიჯზე Max-ის მნიშვნელობა იცვლება A[2]-ით, მე-6 ნაბიჯზე კი Max-ის მნიშვნელობა იცვლება A[5]-ით. ამის შემდეგ Max-ის მნიშვნელობა არ იცვლება და როგორც კი მასივის ყველა ელემენტს გადავათვალიერებთ, Max-ის მნიშვნელობა იქნება ყველაზე დიდი მნიშვნელობა მასივში.

მაქსიმალური ელ. პოვნის ალგორითმი	მინიმალური ელ. პოვნის ალგორითმი
1. დასაწყისი	1. დასაწყისი
2. A[n] მასივის შეტანა	2. A[n] მასივის შეტანა

3. Max = A[0]; i=1;
4. სანამ (i<n)
   თუ (Max < A[i]) Max = A[i];
5. გამოვიტანოთ Max;
6. დასასრული

შესაბამისი პროგრამა

```
#include <iostream>
using namespace std;
main()
{  float A[15], Max;
  for(int i=0; i<10; i++)   //მასივის წაკითხვა
  cin>>A[i];

  Max=A[0];
for(int i=1; i<10; i++)
  if(Max < A[i]) Max = A[i];

cout<<"Max="<<Max;
}
```

3. Min = A[0]; i=1;
4. სანამ (i<n)
   თუ (Min < A[i]) Min = A[i];
5. გამოვიტანოთ Min;
6. დასასრული

შესაბამისი პროგრამა

```
#include <iostream>
using namespace std;
main()
{  float A[15], Min;
  for(int i=0; i<10; i++)   //მასივის წაკითხვა
  cin>>A[i];

  Min=A[0];
for(int i=1; i<10; i++)
  if(Min < A[i]) Min = A[i];

cout<<"Min="<<Min;
}
```

ახლა ვიპოვოთ მასივში მაქსიმალური(მინიმალური) ელემენტის ინდექსი(რიგითი ნომერი), ანუ რომელ ადგილზე დგას მასივში ეს მაქსიმალური(მინიმალური) ელემენტი. როცა Max=A[0], ცხადია ჩვენს მიერ არჩეული მაქსიმალური ელემენტის ინდექსი არის 0, ე.ი. index_Max = 0; ხოლო როცა Max = A[i], მაშინ მაქსიმალური ელემენტის ინდექსი იქნება index_Max = i;

მაქსიმალური ელ. ინდექსის პოვნის ალგორითმი

1. დასაწყისი
2. A[n] მასივის შეტანა
3. Max = A[0]; index_Max=0;
4. i=1;
5. სანამ (i<n)
   {თუ (Max < A[i]) {Max = A[i];
   index_Max=i;}
   i++;}
6. გამოვიტანოთ index_Max;
7. დასასრული

მინიმალური ელ. ინდექსის პოვნის ალგორითმი

1. დასაწყისი
2. A[n] მასივის შეტანა
3. Min = A[0]; index_Min=0;
4. i=1;
5. სანამ (i<n)
   { თუ (Min < A[i]) {Min = A[i];
   index_Min=i;}
   i++; }
6. გამოვიტანოთ index_Min;
7. დასასრული

შესაბამისი პროგრამა

```
#include <iostream>
using namespace std;
main()
{  float A[15], Max,index_Max;
   for(int i=0; i<10; i++)   //მასივის წაკითხვა
   cin>>A[i];

   Max=A[0]; index_Max=0;
for(int i=1; i<10; i++)
   if(Max < A[i]) {Max = A[i]; index_Max=i;}

cout<<"index_Max="<<index_Max;
}
```

შესაბამისი პროგრამა

```
#include <iostream>
using namespace std;
main()
{  float A[15], Min;
   for(int i=0; i<10; i++)   //მასივის წაკითხვა
   cin>>A[i];

   Min=A[0];   index_Min=0;
for(int i=1; i<10; i++)
   if(Min < A[i]) {Min = A[i]; index_Min=i;}

cout<<"index_Min="<<index_Min;
}
```

ახლა ცოტათი გავართულოთ ჩვენი ალგორითმი და ვიპოვოთ მასივში მაქსიმალური მასივის მხოლოდ უარყოფით ელემენტებს შორის.

მაგალითი:

15 4 -65 -12 98 -1 100 -7 9

ამ მასივში ყველა ელემენტებს შორის მაქსიმალური 98-ია, მაგრამ მხოლოდ უარყოფით ელემენტებს შორის მაქსიმალურია -1.

ამ შემთხვევაში არ შეიძლება, რომ Max = A[0], იმიტომ რომ ის დადებითი რიცხვია და ვერც ერთი უარყოფითი რიცხვი ვერ იქნება მასზე ნაკლები.

როგორ უნდა მოვიქცეთ? საჭიროა

- მასივში მივაგნოთ პირველ უარყოფით რიცხვს და ის ჩავთვალოთ პირველ შესაძლო მაქსიმალურ მნიშვნელობად.
- ამის შემდეგ, ყოველი მორიგი, მხოლოდ უარყოფითი ელემენტი შევადაროთ ჩვენს მიერ არჩეულ მაქსიმალურ ელემენტს და მივიღოთ შესაბამისი გადაწყვეტილება.

სიტყვიერი ალგორითმი

1. დასაწყისი
2. A[n] მასივის შეტანა
3. მოვძებნოთ მასივში პირველი უარყოფითი ელემენტი
   3.1. i=0;
   3.2. სანამ (i<n და A[i]>=0) i++;
4. თუ (i<n) { 4.1 Max = A[i];
   4.2 სანამ (i<n)
   {თუ (A[i]<0 და Max < A[i]) Max = A[i];
   i++;}
   4.3 გამოვიტანოთ Max; }

თუ არა გამოვიტანოთ მასივში უარყოფითი ელემენტები არ არის;

5. დასასრული

```
#include <iostream>
using namespace std;
main()
{  float A[15], Max;
    int i;
   for( i=0; i<10; i++)   //მასივის წაკითხვა
   cin>>A[i];
  // ვიპოვოთ პირველი უარყოფითი ელემენტი
for( i=0; i<10 && A[i]>=0; i++);

if(i<10)    // ე.ი. მივაგენით პირველ უარყოფით ელემენტს
             {Max=A[i];
              for(; i<10; i++)
              if(A[i]<0 && Max < A[i]) Max = A[i];
              cout<<"Max_negative="<<Max; }
     else     cout<<"masivSi uarkofiti elementebi ar aris";
}
```

დავალება 1: მოცემული გაქვთ მასივი. იპოვეთ მასივის მხოლოდ დადებით ელემენტებს შორის მინიმალური.

დავალება 2: მოცემული გაქვთ მასივი. იპოვეთ მასივის მაქსიმალურ და მინიმალურ ელემენტებს შორის მოთავსებული დანარჩენი ელემენტების ჯამი.

დავალება 3: მოცემული გაქვთ მთელი ტიპის მასივი. გაარკვიეთ ამ მასივის მაქსიმალური და მინიმალური ელემენტები ურთიერთმარტივი რიცხვებია თუ არა.(რიცხვრბს ურთიერთმარტივი ეწოდება თუ მათი უსგ=1, მაგ. უსგ(10,15)=1, ე.ი. 10 და 15 ურთიერთმარტივი რიცხვებია)

დავალება 4: მოცემული გაქვთ მასივი. ამ მასივის მაქსიმალურ და მინიმალურ ელემენტებს გაუცვალეთ ადგილები და შეცვლილი მასივი დაბეჭდეთ.

დავალება 5:.მოცემული გაქვთ 30 ელემენტიანი მთელი ტიპის მასივი. გაარკვიეთ ეს მასივი წარმოადგენს თუ არა მუდმივ მიმდევრობას.

დავალება 6: მოცემული გაქვთ 25 ელემენტიანი მთელი ტიპის მასივი. გაარკვიეთ ამ მასივის ყველა ელემენტი არის თუ არა 3-ის ჯერადი.

დავალება 7: მოცემული გაქვთ 20 ელემენტიანი მთელი ტიპის მასივი. დაითვალეთ მხოლოდ ლუწი ელემენტების საშუალო არითმეტიკული.

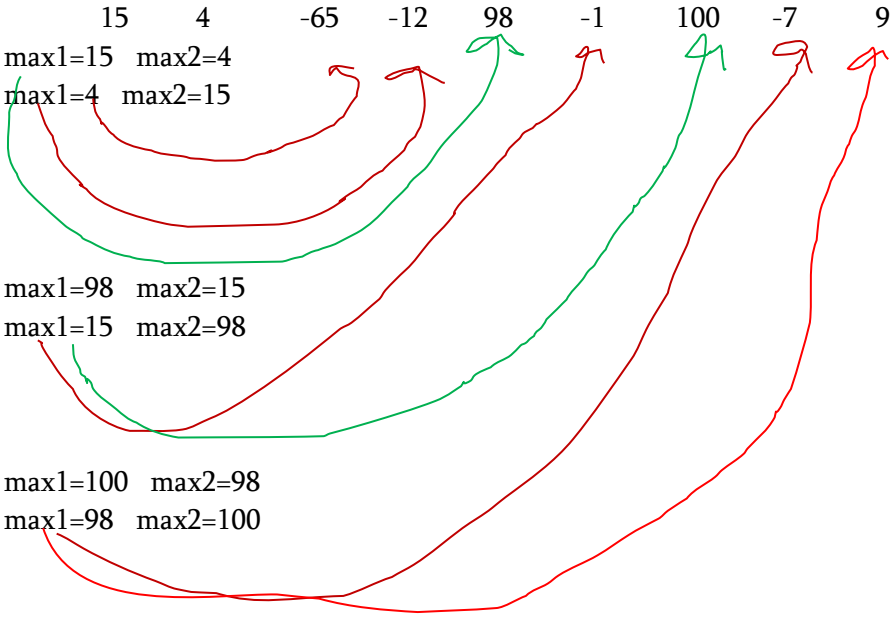
დავალება 8: მოცემული გაქვთ 200ელემენტიანი მთელი ტიპის მასივი. გაარკვიეთ ამ მასივის მინიმალური ელემენტი შედგენილი რიცხვია თუ არა.

რიცხვით მიმდევრობაში(მასივში) რამდენიმე მაქსიმალური და რამდენიმე მინიმალური ელემენტის პოვნის ალგორითმი

ვთქვათ მოცემული გვაქვს მასივი. ვიპოვოთ ამ მასივში ორი მაქსიმალური(მინიმალური) ელემენტი და თან მასივი მხოლოდ ერთხელ გადაათვალიერეთ.
შესაბამისი ალგორითმის ზოგადი სქემა:

- დავუშვათ, რომ მასივის პირველი ორი ელემენტი არის ამ მასივის ორი ყველაზე დიდი ელემენტი - max1=A[0]; max2=A[1];
- max1 და max2 ელემენტები დავალაგოთ ზრდადობით ან კლებადობით. მაგალითად ზრდადობით, ე.ი. max1 < max2
- დავიწყოთ მასივის დათვალიერება მე-3 ელემენტიდან(i=2,3,4,…). მასივის მორიგი A[i] ელემენტი შევადაროთ max1(ანუ უფრო მცირე ელემენტს); თუ A[i]> max1, მაშინ max1= A[i]; და max1 და max2 ელემენტები ისევ დავალაგოთ ზრდადობით.
- როცა მასივის ყველა ელემენტებს გადავათვალიერებთ max1 და max2-ის მნიშვნელობები იქნება მასივის ყველაზე დიდი ორი ელემენტის მნიშვნელობა.

მაგალითი:



სიტყვიერი ალგორითმი

1. დასაწყისი
2. A[n] მასივის შეტანა
3. max1=A[0]; max2=A[1];
4. თუ max1 > max2, მაშინ {t=max1;
   max1=max2; max2=t;}
   // max1 და max2 ელემენტები დალაგებულია ზრდადობით, ე.ი. max1 < max2
5. i = 2;
6. სანამ (i<n)
   {თუ (A[i]> max1) {max1= A[i]; // და max1 და max2 ისევ დავალაგოთ ზრდადობით
   6.1 თუ max1 > max2, მაშინ {t=max1;

```
                                        max1=max2; max2=t;}
                                    }
            i++;}
```

7. გამოვიტანოთ max1 და max2 ;
8. დასასრული

```
#include <iostream>
using namespace std;
main()
{  float A[15], max1, max2,t;
   int i;
  for( i=0; i<10; i++)   //მასივის წაკითხვა
  cin>>A[i];
     max1=A[0]; max2=A[1];
     if (max1 > max2)      {t=max1;
                            max1=max2;
                            max2=t;}    //     max1, max2 ზრდადობით არის დალაგებული

for( i=2; i<10; i++)
if (A[i]> max1)  {max1= A[i];   // და max1 და  max2 ისევ დავალაგოთ ზრდადობით
                  if (max1 > max2)      {t=max1;
                                         max1=max2;
                                         max2=t;}


 cout<<"Max1="<<max1<<"  Max2="<<max2;
    }
```

დავალება 1: მოცემული გაქვთ მასივი. იპოვეთ ამ მასივში ორი უმცირესი ელემენტი.
დავალება 2: მოცემული გაქვთ მასივი. იპოვეთ ამ მასივში სამი უმცირესი(უდიდესი) ელემენტი.
დავალება 3: დაბეჭდეთ ყველა სამნიშნა რიცხვი, რომელთა ჯამი ეკრანიდან წაკითხული n-ის ტოლია(არ გამოიყენოთ გაყოფისა და გამრავლების ოპერაციები)

VI-VII ლექცია

1. რიცხვითი მიმდევრობის ბუნების გარკვევა გარკვეული კრიტერიუმის მიხედვით

განვიხილოთ უმარტივესი ალგორითმი: ვთქვათ გავარკვიოთ რიცხვითი მიმდევრობა ზრდადობით(კლებადობით) არის თუ არა დალაგებული. მასივი ზრდადობით არის დალაგებული ნიშნავს რომ ამ მასივის ნებისმიერი მეზობელი ელემენტისათვის სრულდება ზრდადობის პირობა: A[i]< A[i+1], ან A[i-1]< A[i];. თუ მასივის ერთი მაინც მეზობელი წყვილი არ აკმაყოფილება ზრდადობის პირობას, ეს ნიშნავს რომ მასივი არ არის დალაგებული ზრდადობით.

ზემოთ თქმულიდან გამომდინარე ამ პაწია ამოცანის გადასაჭრელად ორი გზა არსებობს:

- მასივში ვეძებოთ ყველა მეზობელი წყვილი, რომელიც ზრდადობის პირობას აკმაყოფილებს - A[i]< A[i+1]; თუ ასეთი წყვილების რაოდენობა n-1-ია, ეს ნიშნავს რომ მასივი ზრდადობით არის დალაგებული; თუ არა და მასივი არა არის ზრდადობით დალაგებული;
- მასივში ვეძებოთ ერთი მაინც მეზობელი წყვილი, რომელიც ზრდადობის პირობას არღვევს - A[i]> A[i+1]; თუ მივაგენით ასეთ „დამნაშავე - მავნე“ წყვილს, ეს ნიშნავს, რომ მასივი არა არის ზრდადობით დალაგებული; ხოლო თუ ვერ მივაგენით ასეთ „დამნაშავე - მავნე“ წყვილს, ეს ნიშნავს, რომ მასივი ზრდადობით დალაგებული;

სიტყვიერი ალგორითმი:

7. დასაწყისი
8. A[n] მასივის წაკითხვა
9. Z=true;
10. For(i=0; i<n-1; i++)
    if (A[i]> A[i+1]) Z=false;
11. თუ Z==true მასივი ზრდადობით არის დალაგებული;
    თუ არა მასივი ზრდადობით არ არის დალაგებული;
12. დასასრული

სიტყვიერი ალგორითმის აღწერიდან ჩანს, რომ n ელემენტიანი მასივის ბუნების გასარკვევად n-1 ნაბიჯია საჭირო, ეს ნიშნავს, რომ ეს ამოცანა n რიგის ამოცანაა.

```
#include <iostream>
using namespace std;
main()
{  float A[15];
   bool Z=true;
   for(int i=0; i<15; i++)   //მასივის წაკითხვა
   cin>>A[i];
```

```
#include <iostream>
using namespace std;
//    ფუნქციის აღწერა
bool great(float X[], int n)
{   for(int i=0; i< n-1; i++)
   if (X[i]> X[i+1])  return false;

    return true;
```

```
  for(int i=0; i<14; i++)
  if (A[i]> A[i+1])  Z=false;

 if(Z==true)   cout<<"zrdadia  ";
          else   cout<<"ar aris zrdadi  ";
}
```

```
   }
main()
{  float A[15];
   for(int i=0; i<15; i++)   //მასივის წაკითხვა
   cin>>A[i];


  if(great(A,15)==true)   cout<<"zrdadia  ";
           else   cout<<"ar aris zrdadi  ";
}
```

დავალება:
ამოცანა 1: მოცემული გაქვთ მასივი. გაარკვიეთ ეს მასივი კლებად მიმდევრობას წარმოადგენს თუ არა
ამოცანა 2: მოცემული გაქვთ მასივი. გაარკვიეთ ეს მასივი მუდმივ მიმდევრობას წარმოადგენს თუ არა

2. მასივის დალაგების (სორტირების) ალგორითმები

მასივის დალაგების(სორტირების) უამრავი ალგორითმი არსებობს, როგორც იტერაციული, ასევე რეკურსიული. დავალაგოთ მასივი ნიშნავს - მასივის ყველა ელემენტს მოვუძებნოთ თავისი ადგილი შერჩეული კრიტერიუმის მიხედვით; თუ ამას მოვახერხებთ ეს ნიშნავს რომ მასივი დალაგებულია შესაბამისი კრიტერიუმის მიხედვით.

განვიხილოთ მხოლოდ ორი ყველაზე პოპულარული იტერაციული ალგორითმი.

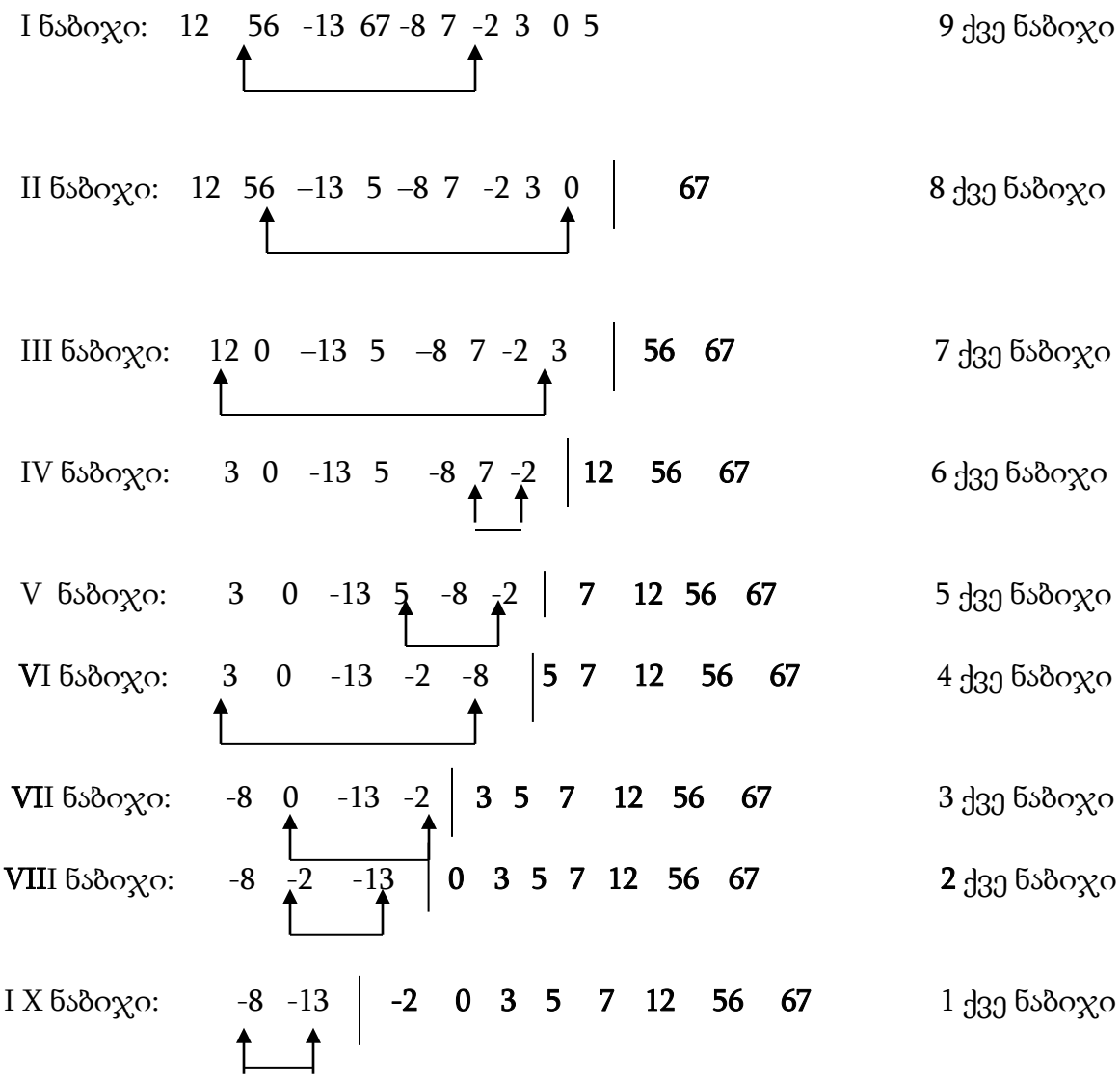
2.1 მასივის დალაგება ზრდადობით მაქსიმალური ელემენტის ძიების გზით

ალგორითმის ზოგადი სქემა ასეთია:

- ვიპოვოთ მასივში მაქსიმალური ელემენტი და ამ მაქსიმალურ ელემენტს და ბოლო ელემენტს გავუცვალოთ ადგილები. ანუ ყველაზე მაქსიმალური ელემენტი დაიჭერს თავის ადგილს.
- ამის შემდეგ მოვძებნოთ მასივში ისევ მაქსიმალური ელემენტი, მხოლოდ ბოლო ელემეტი აღარ ჩავთვალოთ საძიებო არეალში. და ამ ახალ მაქსიმალურ ელემენტსა და ბოლოს წინა ელემენტს გავუცვალოთ ადგილები.
- შემდეგ ნაბიჯზე მაქსიმალური ელემენტის ძიების არეალი ისევ შევამციროთ და ვიპოვოთ მორიგი მაქსიმალური ელემენტი და გავუცვალოთ ადგილები ამ მაქსიმალურ და ბოლოს წინის წინა ელემენტს.
- და ა. შ. სანამ მასივის ყველა ელემენტი არ დაიკავებს საკუთარ ადგილს. როცა ეს მოხდება, ალგორითმის დასრულების შემდეგ მასივი ზრდადობით იქნება დალაგებული.

სქემატურად ეს ალგორითმი შეიძლება ასე წარმოვადგინოთ:

ვთქვათ მოცემულია 10 ელემენტიანი A მასივი:



IX ნაბიჯის შემდეგ მასივი დალაგდა ზრდადობით:

-13 -8 -2 0 3 5 7 12 56 67

როგორც მაგალითიდან ჩანს 10 ელემენტიანი მასივის დასალაგებლად ზრდადობით დაგვჭირდა 9 ნაბიჯი; თუმცა თითოეული ნაბიჯი მოიცავს გარკვეულ ქვენაბიჯებს:

I ნაბიჯი - 9 ქვე ნაბიჯი
II ნაბიჯი - 8 ქვე ნაბიჯი
III ნაბიჯი - 7 ქვე ნაბიჯი
IV ნაბიჯი - 6 ქვე ნაბიჯი
V ნაბიჯი - 5 ქვე ნაბიჯი
VI ნაბიჯი - 4 ქვე ნაბიჯი
VII ნაბიჯი - 3 ქვე ნაბიჯი

ქვენაბიჯების რაოდენობა მცირდება

VIII ნაბიჯი - 2 ქვე ნაბიჯი

IX ნაბიჯი - 1 ქვე ნაბიჯი

ე.ი.ზოგადად n ელემენტიანი მასივის დასალაგებლად საჭიროა n-1 დიდი ნაბიჯი(გარე ციკლი), ხოლო ყოველი ნაბიჯი შესაბამისად მოიცავს n-1, n-2, n-3,... 1 ქვენაბიჯს(შიდა ციკლი). აქედან გამომდინარე სორტირების ალგორითმი არის n^2 რიგის ამოცანა.

ფსევდოკოდის სახით ჩაწერილი ალგორითმი:

13. დასაწყისი
14. A[n] მასივის წაკითხვა
15. For(i=0; i<n-1; i++) //გარე ციკლი
    { Max = A[0]; k=0;
      For(j=1; j<n; j++) // შიდა ციკლი
      if (Max < A[j]) {Max = A[j]; k=j;} // end j
      /* გავუცვალოთ ადგილები მაქსიმალურ - A[k] და A[n-1](შემდეგ ნაბიჯებზე A[n-2], A[n-3], . . . A[1]) ელემნტებს */
    temp= A[k];
    A[k] = A[n-1];
    A[n-1]m = temp;
    } //end i
16. გამოვიტანოთ A[n];
17. დასასრული

```
#include <iostream>
using namespace std;
main()
{  float A[15], Max;
   int k;
   for(int i=0; i<15; i++)   //მასივის წაკითხვა
   cin>>A[i];

   for(int i=0; i<14; i++)          //გარე ციკლი
           { Max = A[0]; k=0;
             for(int j=1; j<15-i; j++)    // შიდა ციკ.
             if ( Max < A[j])  {Max = A[j];
                                k=j;}       // end j
/*   გავუცვალოთ ადგილები მაქსიმალურ - A[k] და
A[14](შემდეგ ნაბიჯებზე    A[13], A[12], . . . A[1])
ელემნტებს      */
           float temp= A[k];
              A[k] = A[14-i];
              A[14-i] = temp;
            }                          //end i
```

```
#include <iostream>
using namespace std;
//   ფუნქციის აღწერა
void sort_great(float X[], int n)
{   int k;
    float Max;
 for(int i=0; i<n-1; i++)
           { Max = X[0]; k=0;
             for(int j=1; j<n-i; j++)
             if ( Max < X[j])  {Max = X[j];
                                k=j;}       // end j
         float temp= X[k];
              X[k] = X[n-i-1];
              X[n-i-1] = temp;
            }                          //end i

}

main()
```

```
//   მასივი დალაგდა ზრდადობით
   for(int i=0; i<15; i++)   //მასივის დაბეჭდვა
   cout<<A[i]<<"  ";
}
```

```
{  float A[15], Max;
   int k;
   for(int i=0; i<15; i++)   //მასივის წაკითხვა
   cin>>A[i];

sort_great(A, 15);      //ფუნქციის გამოძახება

for(int i=0; i<15; i++)   //მასივის დაბეჭდვა
    cout<<A[i]<<"  ";
}
```

დავალება:
ამოცანა 3: დაალაგეთ მასივი კლებადობით მაქსიმუმის ძიების გზით.
ამოცანა 4: დაალაგეთ მასივი ზრდადობით მინიმუმის ძიების გზით.
ამოცანა 5: დაალაგეთ მასივი კლებადობით მინიმუმის ძიების გზით.

2.2 მასივის სორტირება მეზობელი ელემენტების შედარების (ბუშტულების) მეთოდით(bubble sort).

ალგორითმი მდგომარეობს შემდეგში: ვთქვათ მასივს ვალაგებთ ზრდადობით;
ვადარებთ მასივის მეზობელ ელემენტებს ერთმანეთს, ვთქვათ X[i] და X[i+1](ან X[i-1] და X[i]).
თუ X[i] > X[i+1] - ე. ი. ირღვევა ზრდადობის პირობა, მაშინ X[i] და X[i+1] ელემენტებს უნდა გაუცვალოთ მნიშვნელობები; წინააღმდეგ შემთხვევაში X[i] და X[i+1] წყვილს უცვლელად ვტოვებთ.
ალგორითმის პირველი გავლის შემდეგ ამ მასივის ყველაზე მაქსიმალური ელემენტი დაიკავებს მასივში ბოლო ელემენტის ადგილს, ანუ მასივის ბოლო ელემენტი უკვე თავის ადგი;ლე იქნება.
ამის შემდეგ პროცესს დავიწყებთ ისევ თავიდან. მეორე გავლის შემდეგ მორიგი მაქსიმალური ელემენტი დაიჭერს ბოლოდან მეორე ადგილს და ა. შ.
ალგორითმის მუშაობა გაგრძელდება, მანამ სანამ მასივი არ დალაგდება ზრდადობით (a[0]<a[1]<a[2]<a[3]...), ანუ ყველა ელემენტი არ იპოვის თავის ადგილს.

სქემატურად ეს ყველაფერი ასე შეიძლება გამოვსახოთ, ვთქვათ მოცემულია შემდეგი მასივი:

12 56 -13 67 -8 7 -2 3 0 2

I ნაბიჯი: 12 –13 56 -8 7 -2 3 0 2 **67** — 9 ქვე ნაბიჯი

II ნაბიჯი: –13 12 -8 7 -2 3 0 2 **56** **67** — 8 ქვე ნაბიჯი

III ნაბიჯი: –13 -8 7 -2 3 0 2 **12 56 67** 7 ქვე ნაბიჯი

IV ნაბიჯი: –13 -8 -2 3 0 2 **7 12 56 67** 6 ქვე ნაბიჯი

V ნაბიჯი: –13 -8 -2 0 2 **3 7 12 56 67** 5 ქვე ნაბიჯი

VI ნაბიჯი: –13 -8 -2 0 2 **3 7 12 56 67** 4 ქვე ნაბიჯი

```
for(int k=0; k<n-1;k++)   //გარე ციკლი
      for(int i=0; i<n-k-1; i++)        //shida cikli                n-k-1   n-1-1=n-2   n-2-1=n-3
      if(a[i]<a[i+1]) {int t=a[i];
                       a[i]=a[i+1];
                        a[i+1]=t;}
```

ფსევდოკოდის სახით ჩაწერილი ალგორითმი:

1. დასაწყისი
2. A[n] მასივის წაკითხვა
3. For(i=0; i<n-1; i++) //გარე ციკლი
   For(k=0; k<n-i-1; k++) // შიდა ციკლი

```
if ( A[k] > A[k+1])  {temp = A[k];          // ირღვევა ზრდადობის პირობა
                      A[k]=A[k+1];          //  A[k] ⇔ A[k+1];
                       A[k+1]=temp;}
   //    k_ end
   //    i_end
```

4. გამოვიტანოთ A[n];
5. დასასრული

```
#include <iostream>
using namespace std;
main()
{  float A[15];

  for(int i=0; i<15; i++)   //მასივის წაკითხვა
   cin>>A[i];

   for(int i=0; i<14; i++)          //გარე ციკლი
```

```
#include <iostream>
using namespace std;
//    ფუნქციის აღწერა
void sort_great(float X[], int n)
{
    for(int i=0; i<n-1; i++)          //გარე ციკლი
   for(int k=0; k<n-i-1; j++)     // შიდა ციკ.
      if ( X[k] > X[k+1])  {float temp= X[k];
                            X[k] = X[k+1];
```

```
   for(int k=0; k<15-i-1; j++)     // შიდა ციკ.
      if ( A[k] > A[k+1])  {float temp= A[k];
                            A[k] = A[k+1];
                           A[k+1] = temp; }
          //end k
          // end i
//    მასივი დალაგდა ზრდადობით
    for(int i=0; i<15; i++)   //მასივის დაბეჭდვა
    cout<<A[i]<<"  ";
}
```

```
                            X[k+1] = temp; }
          //end k
          // end i
}

main()
{  float A[15];
   for(int i=0; i<15; i++)   //მასივის წაკითხვა
   cin>>A[i];

sort_great(A, 15);       //ფუნქციის გამოძახება

for(int i=0; i<15; i++)   //მასივის დაბეჭდვა
    cout<<A[i]<<"  ";
}
```

დავალებები:

1.მოცემული გაქვთ 30 ელემენტიანი მთელი ტიპის მასივი. გაარკვიეთ ეს მასივი წარმოადგენს თუ არა მუდმივ მიმდევრობას.

2. მოცემული გაქვთ 25 ელემენტიანი მთელი ტიპის მასივი. გაარკვიეთ ამ მასივის ყველა ელემენტი არის თუ არა 3-ის ჯერადი.

3. ამოცანა: მოცემული გაქვთ 20 ელემენტიანი მთელი ტიპის მასივი. დაითვალეთ მხოლოდ ლუწი ელემენტების ჯამი

4. ამოცანა: მოცემული გაქვთ 120 ელემენტიანი მთელი ტიპის მასივი. გაარკვიეთ ეს მასივი წარმოადგენს თუ არა ფიბონაჩის მიმდევრობას.

5. მოცემული გაქვთ 200ელემენტიანი მთელი ტიპის მასივი. გაარკვიეთ ამ მასივის მინიმალური ელემენტი შედგენილი რიცხვია თუ არა.

6. ამოცანა: მოცემული გაქვთ 50 ელემენტიანი ნამდვილი ტიპის მასივი. იპოვეთ ამ მასივში მხოლოდ დადებით ელემენტებს შორის მინიმალური.

VIII ლექცია
გაუმჯობესებული Bubble sort და შერწმით სორტირება

Bubble sort ალგორითმს ამორჩევით სორტირებასთან შედარებით უპირატესობა აქვს: თუ რომელიმე ნაბიჯზე მასივი უკვე დალაგდა ზრდადობით, მაშინ შესაძლებელია ალგორითმმა მუშაობა შეწყვიტოს, ანუ სრულებით არ არის სავალდებულო რომ Bubble sort ალგორითმში გარე ციკლი გარანტირებულად n-1-ჯერ სრულდებოდეს; მაშინ როცა ამორჩევით სორტირების დროს გარე ციკლი აუცილებლად n-1-ჯერ მუშაობს, მიუხედავად იმისა მასივი შეიძლება უკვე დალაგებულიც იყოს.

როგორ ვისარგებლოთ Bubble sort-ის ამ უპირატესობით? რას ნიშნავს მასივი უკვე დალაგდა?

ეს ნიშნავს რომ შიდა ციკლში არც ერთ მეზობელ ელემენტებს შორის ურთიერთშენაცვლება არ მოხდა, ანუ მასივის არც ერთი მეზობელი ელემენტი არ არღვევს სორტირების კრიტერიუმს, რაც თავის თავად ნიშნავს რომ მასივი უკვე დალაგდა.

თუ შიდა ციკლში მოხდა ერთი მაინც გადანაცვლება(t=true), ეს ნიშნავს რომ გარე ციკლი უნდა გაგრძელდეს, მაგრამ თუ არც ერთი გადანაცვლება არ მოხდება მაშინ **t=false და გარე ციკლმა უნდა შეწყვიტოს მუშაობა; ე.ი. გარე ციკლი მუშაობა დამოკიდებულია t ცვლადის მნიშვნელობაზე, რომელიც თვალყურს ადევნებს მოხდა თუ არა გადანაცვლება შიდა ციკლში.**

```
bool t=false;
  for(int k=0; k<n-i-1; j++)     // შიდა ციკ.
     if ( A[k] > A[k+1])  {t=true;
                           float temp= A[k];
                           A[k] = A[k+1];
                          A[k+1] = temp; }
```

12 56 -13 67 -8 7 -2 3 0 2

I ნაბიჯი: 12 –13 56 -8 7 -2 3 0 2 **67** 9 ქვე ნაბიჯი

II ნაბიჯი: –13 12 -8 7 -2 3 0 2 **56 67** 8 ქვე ნაბიჯი

III ნაბიჯი: –13 -8 7 -2 3 0 2 **12 56 67** 7 ქვე ნაბიჯი

IV ნაბიჯი: –13 -8 -2 3 0 2 **7 12 56 67** 6 ქვე ნაბიჯი

V ნაბიჯი: –13 -8 -2 0 2 **3 7 12 56 67** 5 ქვე ნაბიჯი

VI ნაბიჯი: –13 -8 -2 0 2 3 7 12 56 67 4 ქვე ნაბიჯი

Bubble sort მუშაობას შეწყვეტს, რადგან მასივი უკვე დალაგდა.

```
#include <iostream>
using namespace std;
main()
{  float A[15];
   bool t=true;

   for(int i=0; i<15; i++)   //მასივის წაკითხვა
   cin>>A[i];

for(int i=0;  t==true; i++)          //გარე ციკლი
   { t=false;
     for(int k=0; k<15-i-1; k++)     // შიდა ციკ.
       if ( A[k] > A[k+1])  {t=true;
                             float temp= A[k];
                             A[k] = A[k+1];
                             A[k+1] = temp; }
          }                  //end k
          // end i
//    მასივი დალაგდა ზრდადობით
    for(int i=0; i<15; i++)   //მასივის დაბეჭდვა
    cout<<A[i]<<"  ";
}
```

```
#include <iostream>
using namespace std;
//    ფუნქციის აღწერა
void sort_great(float X[], int n)
{   bool sort=true;
  for(int i=0; sort==true; i++)    //გარე ციკლი
   {sort=false;
   for(int k=0; k<n-i-1; k++)     // შიდა ციკ.
      if ( A[k] > A[k+1])  {sort=true;
                             float temp= X[k];
                             X [k] = X[k+1];
                             X [k+1] = temp; }
                    }         //end k
          // end i
}

main()
{  float A[15];
   for(int i=0; i<15; i++)   //მასივის წაკითხვა
   cin>>A[i];

sort_great(A, 15);       //ფუნქციის გამოძახება

for(int i=0; i<15; i++)   //მასივის დაბეჭდვა
    cout<<A[i]<<"  ";
}
```

დავალებები:

1.მოცემული გაქვთ 30 ელემენტიანი მთელი ტიპის მასივი. გაარკვიეთ ეს მასივი წარმოადგენს თუ არა მუდმივ მიმდევრობას.

2. მოცემული გაქვთ 25 ელემენტიანი მთელი ტიპის მასივი. დაითვალეთ ამ მასივის იმ ელემენტების საშუალო არიოთმეტიკული, რომლებიც 3-ის ჯერადია.

3. ამოცანა: მოცემული გაქვთ 20 ელემენტიანი მთელი ტიპის მასივი. დაითვალეთ მხოლოდ იმ ელემენტების ჯამი, რომელთა ათობით წარმოდგენაში ყველა ციფრი ერთი და იგივეა.

4. ამოცანა: მოცემული გაქვთ 120 ელემენტიანი მთელი ტიპის მასივი. გაარკვიეთ ეს მასივი წარმოადგენს თუ არა ფიბონაჩის მიმდევრობას.

5. მოცემული გაქვთ 200ელემენტიანი მთელი ტიპის მასივი. გაარკვიეთ ამ მასივის მინიმალური ელემენტი შედგენილი რიცხვია თუ არა.

6. ამოცანა: მოცემული გაქვთ 50 ელემენტიანი ნამდვილი ტიპის მასივი. იპოვეთ ამ მასივში მხოლოდ დადებით ელემენტებს შორის მინიმალური.

შერწყმით სორტირება

ვთქვათ მოცემული გაქვთ ზრდადობით დალაგებული ორი მასივი(A[n] და B[m]). მათგან ააწყვეთ ასევე ზრდადობით დალაგებული მესამე მასივი C[n+m]) (მასივები დაათვალიერეთ მხოლოდ ერთხელ).

ამ ამოცანაში შეზღუდვა - მასივები დაათვალიერეთ მხოლოდ ერთხელ - მნიშვნელოვანია.

ეს ნიშნავს, რომ საშედეგო მასივში მექანიკურად არ უნდა გადაწეროთ საწყისი მასივების ელემენტები და შემდეგ დაალაგოთ ის ზრდადობით. ამ შემთხვევაში თქვენი (n+m)რიგის ამოცანა $(n+m)^2$ რიგის ამოცანად გადაიქცევა..

არამედ საშედეგო მასივში საწყისი მასივიდან ელემენტები ზრდადობის პირობის გათვალიწინებით უნდა ჩაალაგოთ.(ამ შემთხვევაში თქვენი ამოცანა მხოლოდ და მხოლოდ (n+m) რიგის ამოცანა იქნება)

მაგალითად, ალგორითმი განვიხილოთ შემდეგ კონკრეტულ მასივებზე.

A: -8 3 12 33 100 109 index i =2
ზრდადობით დალაგებული
B: 0 2 3 78 90 102 190 200 201 300 index j=3
ზრდადობით დალაგებული

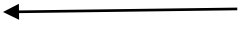
C: -8 0 2 3 3 12 33 78 90 100 102 109 190 200 201 300 Index k
არაკლებადობით დალაგებული
C[0] = A[0] და B[0] შორის უფრო მცირე
C[1] = A[1] და B[0] შორის უფრო მცირე
C[2] = A[1] და B[1] შორის უფრო მცირე
C[3] = A[1] და B[2] შორის უფრო მცირე
და ა.შ.
თუ რომელიმე ნაბიჯზე აღმოჩნდა, რომ პირველ ან მეორე მასივში უკვე ყველა ელემენტი დავათვალიერეთ, მაშინ საშედეგო მასივში უბრალოდ უნდა გადაწეროთ ელემენტები იმ მასივიდან, რომელშიც ჯერ კიდევ არის ელემენტები.
ამ ალგორითმს შერწყმით სპორტირების ალგორითმი ეწოდება.

```
#include <iostream>
using namespace std;
#define n 5
```

```
#define m 7
void merge_sort(int x[], int n1, int y[], int m1, int z[])
{int i, j, k;
for(i=j=k=0;   i<n1 && j<m1 ;      k++  )
       if(x[i] < y[j])    {z[k]=x[i];  i++;}
            else     {z[k]=y[j];  j++; }
if(i<n1)   for(;   i<n1;  i++,k++)
                    z[k] = x[i];
          else for(;   j<m1;  j++,k++)
                    z[k] = y[j];
}
main()
{int A[n], B[m], C[n+m];
for(int i=0; i<n; i++)
cin>>A[i];
for(int i=0; i<m; i++)
cin>>B[i];

merge_sort(A, n, B, m, C);
for(int i=0; i<n+m; i++)
cout<<C[i]<<"     ";
}
a[n],b[m],c[n+m]
int i=j=k=0;
while (i<n && j<m)
if(a[i]<b[j]) {c[k] = a[i]; i++; k++;}
      else { c[k]=b[j];  j++; k++}
  if(i<n) for(;  i<n; i++,k++)  c[k]=a[i];
    else   for(;  j<m; j++,k++)  c[k]=b[j];
for(i=0;  i<n+m; i++)
cout<<c[k]<<" ";}
```

A: -8 3 12 33 100 109 index i=0 i++ ზრდადობით დალაგებული

B: 200 150 78 5 -99 -1000 index j=5 j-- კლებადობით დალაგებული



C: -1000 -99 -8 3 5 12 33 78 100 109 150 200 ზრდადობით

index k =0 k++

C[0] = A[0] და B[m-1] შორის უმცირესი
C[1] = A[0] და B[m-2] შორის უმცირესი
C[2] = A[1] და B[m-2] შორის უმცირესი და ა.შ.

შესაბამისი პროგრამა

```
#include <iostream>
using namespace std;
#define n 5
#define m 7
void merge_sort(int x[], int n1, int y[], int m1, int z[])
{int i, j, k;
for(i=k=0, j=m-1;  i<n1  && j >=0  ;     k++  )
       if(x[i] < y[j])    {z[k]=x[i];  i++;}
           else     {z[k]=y[j];  j--; }
if(i<n1)   for(;   i<n1;  i++,k++)
                   z[k] = x[i];
          else for(;   j>=0;  j--,k++)
                   z[k] = y[j];
}

main()
{int A[n], B[m], C[n+m];
for(int i=0; i<n; i++)
cin>>A[i];
for(int i=0; i<m; i++)
cin>>B[i];

merge_sort(A, n, B, m, C);
for(int i=0; i<n+m; i++)
cout<<C[i]<<"     ";
}
```

დავალება:

1. A –ზრდადობით, B- ზრდადობით; C- კლებადობით
2. A –კლებადობით, B- კლებადობით; C- კლებადობით
3. A –კლებადობით, B- კლებადობით; C- ზრდადობით
4. A –კლებადობით, B- ზრდადობით; C- კლებადობით
5. A –კლებადობით, B- ზრდადობით; C- ზრდადობით
6. A –ზრდადობით, B- კლებადობით; C- კლებადობით
7. A –ზრდადობით, B- კლებადობით; C- ზრდადობით

ლექცია XI - X

რეკურსიული ალგორითმებისა და ფუნქციების ცნება. პირდაპირი და ირიბი რეკურსია. რეკურსიის სიღრმე. რეკურსიის დასრულების პირობა.

რეკურსიული ფუნქცია ეწოდება ისეთ ფუნქციას, რომელიც საკუთარ თავს იძახებს, მაგრამ არა იმ პარამეტრით, რომლითაც თავის დროზე თავად იყო გამოძახებული.
რა თქმა უნდა ფუნქციის რეკურსიული გამოძახება უსასრულოდ არ შეიძლება გაგრძელდეს, ამიტომ აუცილებელია რეკურსიის დასრულების პირობის განსაზღვრა.
არსებობს ორი სახის რეკურსია: **პირდაპირი და ირიბი.**
პირდაპირი რეკურსიის დროს პირობითად A ფუნქცია იძახებს ისევ A ფუნქციას, ხოლო ირიბი რეკურსიის დროს A ფუნქცია იძახებს B ფუნქციას და B ფუნქცია თავის მხრივ ისევ A ფუნქციას იძახებს.

პირდაპირი რეკურსიის მაგალითი:

```
int k=0;
void direct_rec_func(int x)
{
    . . .
    if (rekursiis_dasrulebis piroba) return;
   direct_rec_func(45);                    //რეკურსიული ფუნქციის გამოძახება
   cout<<++k;
}
```

ირიბი რეკურსიის გამოძახების მაგალითი:

```
void other_func(int);
void indirect_rec_func(int x)
{ . . .
  if(rec_das_pir) return;
   other_func(x);
    cout<<x;
. . .
```

```
void other_func(int x)
{ . . .
      indirect_rec_func(x);
      cout<<x;
       . . .
```

} }

რეკურსიის არსი გავარკვიოთ მარტივ მაგალითზე დაყრდნობით, მოცემული n რიცხვის ფაქტორიალის დათვლის მაგალითზე.

რიცხვის ფაქტორიალის განმარტება შეიძლება როგორც იტერაციულად, ასევე რეკურსიულად.

მაგალითად, თუ ფაქტორიალს განმარტავთ ასე: n! = 1 *2 *3 *4 * . . .(n-1)*n, მაშინ ეს იქნება რიცხვის ფაქტორიალის ცნების იტერაციულად განსაზღვრის მცდელობა;

მაგრამ თუ ფაქტორიალს განვმარტავთ ასე - n! = (n-1)!*n, მაშინ ეს უკვე ფაქტორიალის განმარტების რეკურსიული ფორმულირება იქნება.

F(n)=n!

n! = (n-1)!*n

(n-1)! = (n-2)!*(n-1)

(n-2)! = (n-1)!*(n-2)

(n-3)! = (n-2)!*(n-3)

. . .
4!=3!*4
3!=2!*3

2!=1!*2
1!=1

რეკურსია თავისი შინაარსით ჰგავს ბურღის პრინციპს: ჯერ ჩავდივართ რეკურსის სიღრმეში, სანამ არ მივაღწევთ რეკურსიის დასრულების პირობას; ხოლო ამის შემდეგ იწყება ზესვლა რეკურსიის გზაზე.

n! -ის პოვნის რეკურსიული ფუნქცია ძალიან მარტივი, ლამაზი და თვალსაჩინო ფუნქციაა:

int faqtorial(int n)

{ if(n == 1) return 1;

return (faqtorial(n-1) * n); }

როცა n=5, ფაქტორიალის დათვლის პროცესი შემდეგია

5! = 4! * 5

4! = 3! * 4

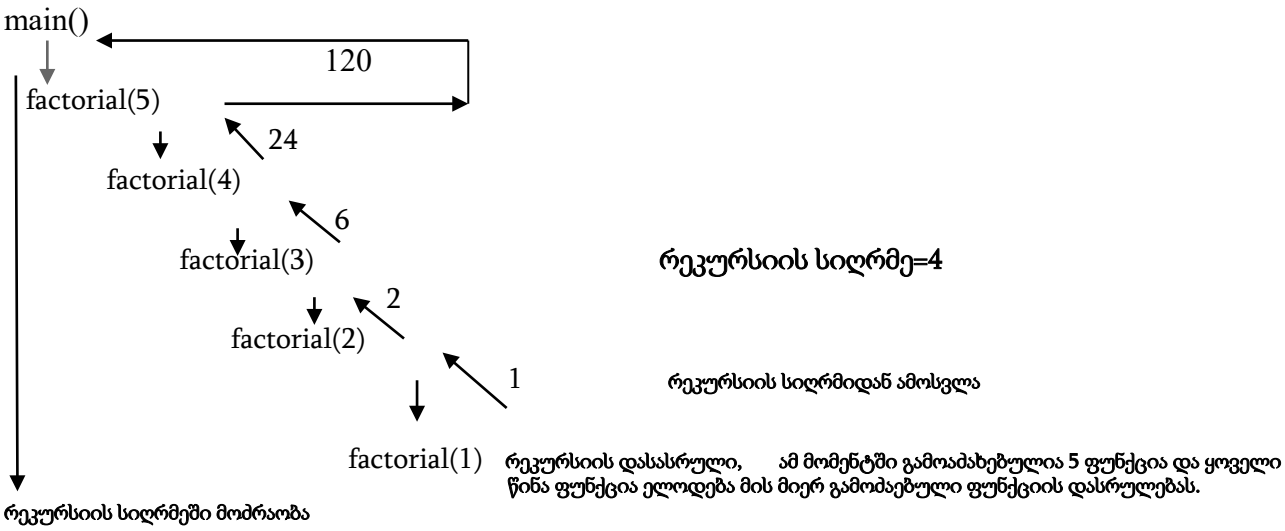
3! = 2! * 3

2! = 1! * 2

1! = 1

რეკურსია დასრულდა, ამის შემდეგ ვიცით რა 1!, შეგვიძლია გავიგოთ 2!, ვიცით რა 2!, შეგვიძლია გავიგოთ 3! და ა.შ., ვიცით რა (n-1)!, შეგვიძლია გავიგოთ n!; ე.ი. ვიწყებთ რეკურსიის სიღრმიდან ამოსვლას და გზად ყველა გამოძახებული ფუნქცია დაასრულებს მუშაობას და გამომძახებელ ფუნქციას დაუბრუნებს მართვას.

ეს პროცესი სქემატურად შეიძლება შემდეგნაირად გამოვსახოთ



იგივე ამოცანა შეიძლება გადავჭრათ არარეკურსიული ფუნქციის დახმარებით, გამოვიყენებთ რა ფაქტორიალის იტერაციულ განმარტებას:

n! =1*2*3*…*(n-1)*n

შესაბამისს ფუნქციას აქვს სახე:

```
int fact(n)
{ int s=1;
  for(int i=1; i<=n; i++)
      s*=i;
 return(s);
}
```

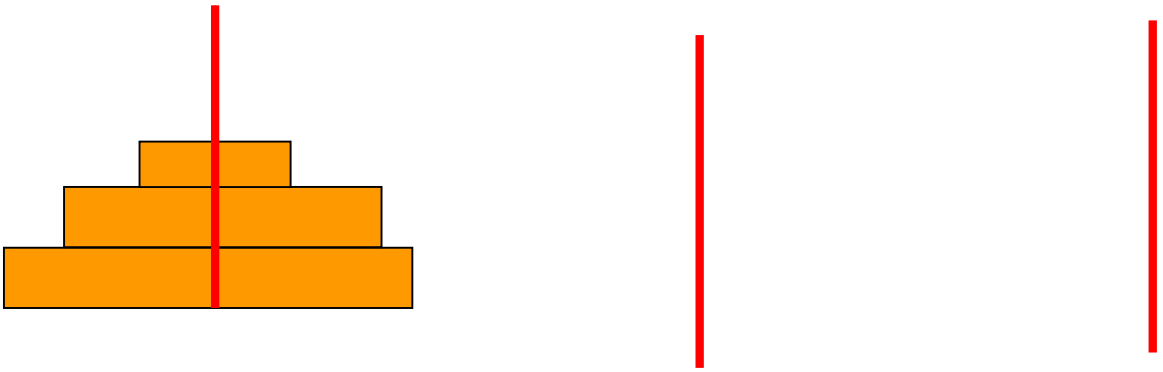
ამ ორი მაგალითის შედარებით შეიძლება შემდეგი დასკვნის გაკეთება: თუ ალგორითმი არის იტერაციული, მაშინ შესაბამისი ფუნქცია იქნება არარეკურსიული ფუნქცია, ხოლო თუ ალგორითმი არის რეკურსიული, მაშინ შესაბამისი ფუნქციაც რეკურსიულია.

ჰანოის კოშკის ალგორითმი.

ეს თავსატეხი მოიფიქრა ფრანგმა მათემატიკოსმა ედუარდ ლიუკმა 1883 წ. და ეს ლეგენდაც მისი შეთხზულია: ღმერთმა ბრაჰმამ დასაჯა ქალაქ ბენარესის დიდებული ტაძრის ბუდისტი ბერები - განრისხებულმა ბრაჰმამ აღმართა სამი ალმასის ღერძი და ერთ-ერთ ღერძზე ოქროს 64 სხვადასხვა ზომის მრგვალი ფირფიტა პირამიდისებურად ააწყო. ეს ფირფიტები ბერებს უნდა გადაეწყოთ პირველი ღერძიდან მეორე ღერძზე, ისე რომ პირამიდისებური კონსტრუქცია შენარჩუნებული ყოფილიყო. იყო მხოლოდ ორი შეზღუდვა: დიდი ზომის ფირფიტაზე უნდა დაედოთ მხოლოდ მცირე ზომის ფირფიტა და ყოველ გადატანაზე უნდა გადაეტანათ მხოლოდ ერთი ფირფიტა (ბერებს დამატებით შეეძლოთ გამოეყენებინათ მესამე ღერძი).

ლეგენდის თანახმად, როგორც კი ბერები ფირფიტების გადატანას დაასრულებდნენ, დადგებოდა ქვეყნიერების დასასრული. ფირფიტების გადატანათა რაოდენობა გამოითვლება ფორმულით $2^{64} - 1$; ბერებს დღე და ღამე შეუსვენებლივ რომ ეშრომათ და ერთი ფირფიტის გადასატანად მხოლოდ 1 წამი დაეხარჯათ, ყველა ფირფიტის გადასატანად 585 მლრდ წელი დასჭირდებოდათ.

ალგორითმი განვიხილით სამი ფირფიტის მაგალითზე



იმისათვის რომ მოვახერხოთ სამი ფირფიტის გადატანა I ღერძიდან IIღერძზე, საჭიროა III ღერძზე გადავიტანოთ ორი ფირფიტა , II ღერძზე გადავიტანოთ ყველაზე დიდი ფირფიტა და ბოლოს III ღერძიდან II –ზეგადავიტანოთ ორი მცირე ფირფიტა.
თუ განვაზოგადებთ სამი ფირფიტის შემთხვევას n ფირფიტისათვის,
ალგორითმი იქნება შემდეგი:
იმისათვის რომ n ცალი ფირფიტა I ღერძიდან II ღერძზე გადავიტანოთ, საჭიროა I-დან n -1 ფირფიტა გადავიტანოთ III ღერძზე ამის შემდეგ I ღერძზე დარჩენილი ყველაზე დიდი ფირფიტა გადავიტანოთ II-ზე და ბოლოს III-დან II-ზე გადავიტანოთ n -1 ფირფიტა.
თავის მხრივ n -1 ფირფიტის გადატანა II -> III-ზე გულისხმობს n -2
ფირფიტა გადავიტანოთ II -> I-ზე, II -> III გადავიტანოთ დარჩენილი ერთი

ფირფიტა და ბოლოს n -2 ფირფიტა II -> III და ა.შ.

ალგორითმი ცალსახად რეკურსიული ალგორითმია, არ არსებობს მისი შესაბამისი იტერაციული ალტერნატივა. თუ რიცხვის ფაქტორიალის დათვლის დროს გვაქვს არჩევანი, რომელი ალგორითმი ავირჩიოთ იტერაციული, თუ რეკურსიული, ამ შემთხვევაში ასეთი არჩევანი აღარ გვაქვს.

```
void hanoi (int n, int I,int II )
{if (n= =1) {cout<<“gadavitanot firfita I-dan II-ze”;
        return;}
  hanoi(n-1,I, 6-(I+II));
  hanoi(1, I,II);
  hanoi(n-1,6-(I+II),II);
}
```

მიუხედავად იმისა, რომ რეკურსია ცუდად განშლადია, ამ ალგორითმის გადაწყვეტის ერთადერთი გზა რეკურსიული ფუნქციის აღწერაა, უბრალო იტერაციით აქ ფონს ვერ გავალთ.

ჰანოის კოშკის ალგორითმი ერთადერთი არ არის ცუდად განშლად რეკურსიულ ალგორითმებს შორის. მსგავს რეკურსიულ ალგორითმს მიეკუთვნება ფიბონაჩის მიმდევრობის აგების რეკურსიული ალგორითმი. თუმცა აქ არჩევანის შანსი გვაქვს: რეკურსიასა და იტერაციულ ფუნქციას შორის.

ფიბონაჩის მიმდევრობის აგების რეკურსიული ალგორითმი

უკვე ვიცით, რომ ფიბონაჩის მიმდევრობა არის ისეთი რიცხვითი მიმდევრობა, რომლის პირველი ორი წევრი 1-ის ტოლია(ან ალტერნატიული ინტერპრეტაციით 0 და 1), ხოლო ყოველი მომდევნო წევრი არის წინა ორი წევრის ჯამი. რეკურენტული ფორმულების დახმარებით ფიბონაჩის მიმდევრობა შეიძლება აღვწეროთ ასე:

F(1)=1 F(2)=1 ან F(1)=0 F(2)=1

F(n)=F(n-1)+F(n-2), n=3,4… F(n)=F(n-1)+F(n-2), n=3,4…

ამ ფორმულების თანახმად

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233…. . .

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233…. . .

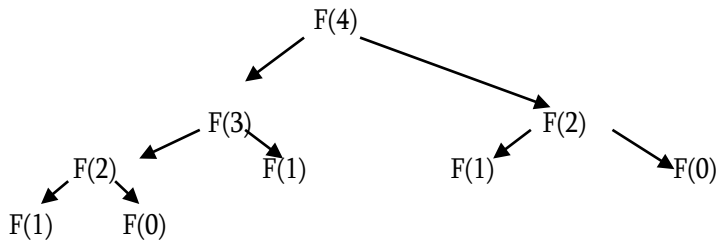
ზემოთ მოყვანილი ორივე რიცხვითი მიმდევრობები ფიბონაჩის მიმდევრობას

წარმოადგენს.

ფიბონაჩის მიმდევრობის აგება შეიძლება როგორც ჩვეულებრივი იტერაციული ფუნქციის დახმარებით, ასევე რეკურსიული ფუნქციის მეშვეობით. მოვიყვანოთ ორივე ფუნქციის შესაბამისი კოდი და ვნახოთ რომელი მათგანის მხარესაა უპირატესობა.

```
// იტერაციული ფუნქცია
int fibonachi_itaration(int  n)
{int F1,F2,F3;                          1    1    2         3       5
F1=F2=1;                                F1   F2   F3
for (int i=3;i<=n;i++)                       F1    F2      F3
{F3=F1+F2;                                   F1    F2      F3
F1=F2;
F2=F3;}
return F3;}

//რეკურსიული ფუნქცია
int  fibonachi_rec(int  n)
{
if   (n= =0 || n= =1) return 1;                 //რეკურსიის დასრულების პირობა
   else   { int k=fibonachi(n-1);
                int k1=fibonachi(n-2);
                  return k+k1;}
}
```

ამ რეკურსიული ფუნქციის მუშაობის შედეგი n=4-სთვის შემდეგია:



თუ დააკვირდებით სურათს F(2)-ის მნიშვნელობის დათვლა n=4-თვის 2-ჯერ ხდება, იმის გამო რომ ეს ერთი და იგივე მნიშვნელობა რეკურსიის სხვადასხვა დონეზე, სხვადასხვა ფუნქციების მიერ იყო ნაპოვნი. დამოუკიდებლად ნახეთ რა სურათს მიიღებთ, როცა n=6 და რამდენჯერ დაგჭირდებათ F(2) ფუნქციის გამოძახება და მარტო ამ ფუნქციის.

როცა რეკურსიის გაშლის დროს ერთი და იგივე პარამეტრიანი ფუნქციის გამოძახება ბევრჯერ ხდება ალგორითმის თავისებურებიდან გამომდინარე, ამ დროს ამბობენ, რომ რეკურსიული ფუნქცია ცუდად იშლება.

დასკვნა

ნებისმიერი იტერაციული ფუნქციის აღწერა შესაძლებელია რეკურსიული ფუნქციის დახმარებით. თუმცა პირუკუ დებულებას ადგილი არა აქვს. ბუნებრივია როცა არჩევანი არის იტერაციულ ფუნქციასა და ცუდად განშლად რეკურსიულ ფუნქციას შორის, არჩევანი იტერაციული ფუნქციის სასარგებლოდ უნდა გააკეთოთ.

მაგრამ არსებობს კლასი ამოცანებისა, რომელთა გადაწყვეტა რეკურსიული ფუნქციების გარეშე შეუძლებელია. ასეთ ალგორითმთა კლასს მიეკუთვნება ზემოთ განხილული ჰანოის კოშკის მაგალითი, დინამიურ მონაცემთა სტრუქტურებთან სამუშაო ალგორითმები, სინტაქსური გარჩევის ალგორითმები (მაგრამ არა ყველა, ბუნებრივია) და ა. შ.

ნაკლი: რეკურსიული ფუნქციები, როგორც წესი, შედარებით ნელა მუშაობს,რადგან ერთი ფუნქციის მაგივრად რამდენიმე ფუნქციაა გამოძახებული(დამოკიდებულია რეკურსიის სიღრმეზე). თუ ღრმა რეკურსიასთან გვაქვს საქმე, შესაძლებელია მეხსიერებასთანაც გაჩნდეს პრობლემა, ე.წ. სტეკის გადავსების გამო.

დავალება 1: რეკურსიული ფუნქციით იპოვეთ მთელი რიცხვის
ციფრთა ჯამი.

მაგალითი:

jami(1543)= jami(154)+3=13

Jami(154) = jami(15)+4=10

Jami(15)=jami(1)+5=6

Jami(1)=jami(0)+1=1

Jami(0)=0

დააკვირდით რომელი ცვლადის მიმართ მიდის რეკურსია, როდის უნდა დამთავრდეს რეკურსია. პროცესი ზოგადად აღიწერება შემდეგი რეკურენ-ტული ფორმულით:

Jami(x)= jami(x/10)+x%10

დავალება2: შეგაქვთ X და n. რეკურსიული ფუნქციის დახმარებითიპოვეთ X^n

მითითება: ისარგებლეთ იმ ფაქტით, რომ:

$X^n = X^{n-1} * X$

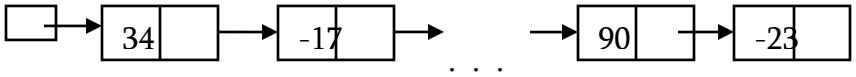
იფიქრეთ როდის უნდა დასრულდეს რეკურსია.

დავალება 3: ჩაწერეთ რეკურსიულად ევკლიდეს ალგორითმი

ლექცია XII - XIV
მონაცემთა დინამიური სტრუქტურები

საცნობარო მასალა. მონაცემთა დინამიური სტრუქტურებია: ცალადბმული სია, ორადბმული სია, რიგი, სტეკი, ორობითი ხე.

ცალადბმული სია. ცალადბმული სიის ყოველი რგოლი (ჩანაწერი) შეიცავს ორ მაინც ველს და ერთი ველი აუცილებლად არის მიმთითებელი სიის შემდეგ რგოლზე (ანუ ამ ველის მნიშვნელობა არის სიის მორიგი რგოლის მისამართი).



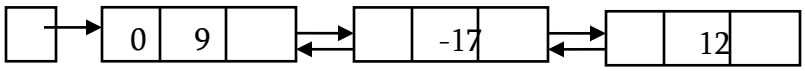
სიის ბოლო რგოლი არაფერზე არ მიუთითებს, ამიტომ მისი მნიშვნელობა არის 0.

ცალაბმული სიის შესაბამისი ტიპის აღწერას აქვს შემდეგი სახე:

```
struct List {
// ერთი, ან რამდენიმე ველის აღწერა;
    List* next; };
```

ცალადბმულ სიაზე განსაზღვრულია შემდეგი მოქმედებები: სიის აგება. სიიდან ელემენტის ამოგდება, სიაში ელემენტის დამატება, სიის დალაგება.

ორადბმული სია. ორადბმული სიის ყოველი რგოლი (ჩანაწერი) შეიცავს სამ მაინც ველს და აქედან ორი ველი არის მიმთითებელი: ერთი სიის შემდეგ რგოლზე, ხოლო მეორე მითითებს წინა რგოლზე.



ორადბმული სიის შესაბამისი ტიპის აღწერას აქვს შემდეგი სახე:

```
struct DList {
// ერთი, ან რამდენიმე ველის აღწერა;
    DList* next;
    DList* prev; };
```

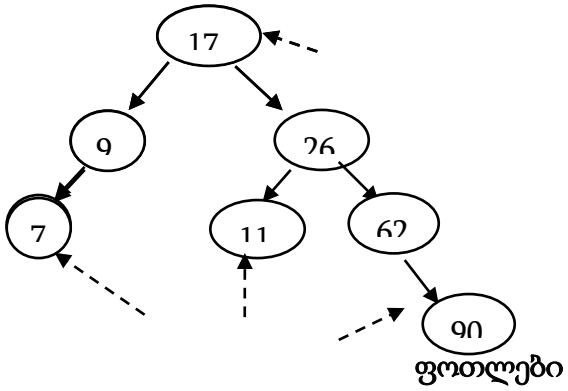
ორადბმულ სიაზე განსაზღვრულია შემდეგი მოქმედებები: სიის აგება, სიიდან ელემენტის ამოგდება, სიაში ელემენტის დამატება, სიის დალაგება.

რიგი. რიგი არის ისეთი ცალადბმული სია, რომელშიც ელემენტი ემატება მხოლოდ

ბოლოდან, ხოლო ელემენტის ამოგდება ხდება სიის თავიდან. რიგი აგებულია FIFO-ს პრინციპით (First In First Out - პირველი მოხვედი, პირველი წახვალ). რიგზე განმარტებულია მხოლოდ ორი მოქმედება: ელემენტის დამატება რიგის ბოლოში და ელემენტის ამოგდება რიგის თავიდან.

სტეკი. სტეკი, ისეთი ცალადბმული სიაა, რომელშიც ელემენტი ემატება სიის თავში და ელემენტის ამოგდება შეიძლება მხოლოდ სიის დასაწყისიდან. სტეკი აგებულია LIFO-ს პრინციპით (Last In First Out - ბოლო მოხვედი, პირველი წახვალ). სტეკზე განმარტებულია მხოლოდ ორი მოქმედება: ელემენტის დამატება სტეკის თავში (Push) და პირველი ელემენტის ამოგდება სტეკიდან (Pop).

ორობითი ხე. ორობითი ხე არის რგოლების, ფოთლებისა და მათი დამაკავშირებელი წიბოების ერთობლიობა. ორობითი ხის ყოველი რგოლი მიუთითებს ხის მარჯვენა და მარცხენა ქვეხეზე. ეს ნიშნავს რომ ორიბითი ხის ყოველ რგოლზე მიუთითებს ერთი მიმთითებელი, ხოლო თვითონ რგოლი შეიცავს ორ მიმთითებელს მარჯვენა და მარცხენა ქვეხეზე. ორობით ხეში არსებობს მხოლოდ ერთი რგოლი, რომელზეც არაფერი არ მიუთითებს, ამ რგოლს ხის თავი, ან ფესვი ეწოდება. ორობითი ხის ისეთ რგოლებს, რომლებიც „არაფერზე“ არ მიუთითებენ ორობითი ხის ფოთლები ეწოდება.



ორობითი ხის თავი(ფესვი)

ორობით ხეში ელემენტები გარკვეული წესით არიან მოწესრიგებული, ყოველი რგოლის მარჯვნივ არის მასზე მეტი ქვერგოლები, ხოლო მარცხნივ - მასზე ნაკლები.

ორობითი ხის შესაბამისი ტიპის აღწერას აქვს სახე:

```
struct BTree {
// ერთი, ან რამდენიმე ველის აღწერა;
BTree* left;
    BTree* Right; };
```

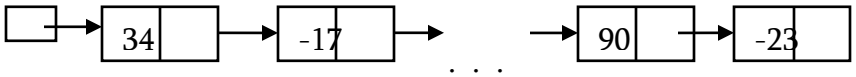
ორობით ხეზე განსაზღვრულია შემდეგი მოქმედებები: ორობითი ხის აგება, ხიდან ელემენტის ამოგდება, ხეში ელემენტის დამატება, ხეში ელემენტის ძიება.

ორობით ხეში ელემენტის დამატება, ან ამოგდება ისე უნდა მოხდეს, რომ ორობითი ხის პრინციპი არ დაირღვეს (ყოველი რგოლის მარჯვნივ მასზე მეტი, მარცხნივ მასზე ნაკლები).

ცალადბმული სია

ჩვენ განვიხილავთ მხოლოდ ცალადბმულ სიას. ცალადბმული სია არის მონაცემთა ისეთი სტრუქტურა, რომლის ყოველი ელემენტი შეიცავს მომდევნო ელემენტის მისამართის შესახებ ინფორმაციას. ტექნიკურად, ეს ხერხდება ყოველი ელემენტისათვის ერთი ველის დამატებით, რომელშიც ჩაიწერება მომდევნო ელემენტის მისამართი. ეს სიტუაცია იმის ანალოგიურია, როდესაც რიგში მდგომები თავიანთ წინ მდგომს იმახსოვრებენ და ამის წყალობით თანმიმდევრობა შენარჩუნდება, თუნდაც ეს ადამიანები იდგნენ ჯგუფად და არა ერთ მწკრივში. ეს ანალოგია მართლაც მნიშვნელოვანია, რადგან, მასივებისაგან განსხვავებით, ბმული სიის ელემენტები მეხსიერებაში ერთმანეთის მიმდევრობით არ განლაგდება, რასაც აქვს თავისი დადებითი და უარყოფითი მხარე. დადებითი ისაა, რომ ადვილია სიაში ელემენტის ჩამატება და სიიდან ელემენტის ამოგდება და შესაძლებელია პრაქტიკულად ნებისმიერი რაოდენობის ელემენტების დამატება, რის საშუალებებიც არ გააჩნია მასივის სახით ორგანიზებულ მონაცემებს.

ცალადბმული სიის ყოველი რგოლი (ჩანაწერი) შეიცავს ორ მაინც ველს და ერთი ველი აუცილებლად არის მიმთითებელი სიის შემდეგ რგოლზე (ანუ ამ ველის მნიშვნელობა არის სიის მორიგი რგოლის მისამართი).



სიის ბოლო რგოლი არაფერზე არ მიუთითებს, ამიტომ მისი მნიშვნელობა არის 0.

მოკლედ, ცალადმული სიის შესაბამისი ტიპი აღიწერება სტრუქტურის დახმარებით და ეს არის სამომხმარებლო ტიპი, რომელსაც თვითონ პროგრამისტი ქმნის:

```
struct List {
// ერთი, ან რამდენიმე ველის აღწერა;
    List* next; };
```

ჩვენ განვიხილავთ მთელი რიცხვების შესაბამისს ცალადბმულ სიას.

```
struct List_int {
    int info;
    List* next; };
```

ცალადბმულ სიაში მხოლოდ ერთი მიმართულებით არის შესაძლებელი გადაადგილება

მარცხნიდან მარჯვნივ. სიის პირველ ელემენტზე წვდომა აუცილებლად უნდა გვქონდეს წინააღმდეგ შემთხვევაში სიაში ნავიგაციის დაწყება შეუძლებელია. რადგან სიის ბოლო ელემენტი არაფერზე არ მიუთითებს, ამიტომ სიის ბოლო რგოლის ე.წ. სამისამართო ველის მნიშვნელობა 0-ის ტოლია. დინამიური მასივისაგან განსხვავებით არასდროს არ არის ცნობილი სიაში რამდენი ელემენტი უნდა იყოს, ან არის.

ცალადბმულ სიაზე განსაზღვრულია შემდეგი მოქმედებები: სიის აგება, სიიდან ელემენტის ამოგდება, სიის დალაგება, დალაგებულ სიაში ელემენტის თავის ადგილზე ჩამატება, სიის დაბეჭდვა, სიაში ელემენტის ძიება, სიის პირველი ელემენტის ამოგდება, სიის თავში ახალი ელემენტის ჩამატება, სიის ბოლოს ელემენტის ამოგდება, სიის ბოლოში ახალი ელემენტის დამატება.

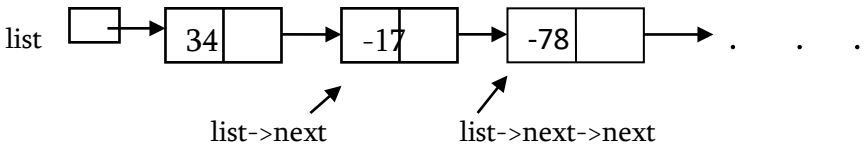
სიის აგება

დავიწყოთ სიის აგების ფუნქციის აღწერა, იმიტომ რომ თუ სია აგებული არ არის რაიმე სხვა მოქმედების განხორციელებას აზრი არა აქვს.

სია ჯერ კიდევ არ არსებობს, მისი პირველი ელემენტის რეალიზაციისათვის საჭიროა მოვითხოვოთ მეხსიერება - list= new List_int;

ხოლო სიის მორიგი ელემენტის „რეალიზაციისათვის“, საჭიროა შემდეგი მოქმედების ჩატარება:

list->next= new List_int; და ა.შ. სანამ სია არ აიგება.



რადგან სიაში ინდექსირების ოპერაცია არ არსებობს, ამიტომ სიის ერთი ელემენტიდან შემდეგ ელემენტზე გადასვლა შეიძლება ასეთი მინიჭების ოპერატორით: list = list->next;

სიაში ნავიგაციის განსახორციელებლად სიის თავი უმნიშვნელოვანესი ინფორმაციაა, ამიტომ ყველა ფუნქციაში შემოგვაქვს დამატებითი p=list ცვლადი და მისი საშუალებით გადავდივართ სიის ერთი ელემენტიდან მორიგ ელემენტზე.

სწორედ ამ მინიჭების დამსახურებაა სიის აგების ციკლის განხორციელება.

სიის აგების ფუნქციას აქვს შემდეგი სახე, იგულისხმება, რომ ფაილში მოცემული მონაცემების მიხედვით ვაგებთ სიას:

```
void Build(List_int* &list, ifstream &F)
{ list=new List_int;              // სიის პირველი ელ. მისამართი -  სიის თავი
  List_int *p=0, *pr=0;
  p=list;                         // დამატებითი ცვლადი, სიის თავის ასლი
  while(!F.eof())
  {F>>p->info;
   p->next=new List_int;
   pr=p;
   p=p->next;}
         delete p;
         pr->next=0;  }
        main()
        {ifstream F;
         F.open("f.txt", ios::in);
    if (!F.is_open())  {cout<<"error"; exit(0); }
        List_int* list=0;
          Build(list,F);
        }
```

სიის დაბეჭდვა

ყველაზე მარტივი სამუშაო ფუნქციაა: უნდა გავიაროთ უკვე შექმნილ სიაზე - სიის პირველი ელემენტიდან სიის ბოლო ელემენტამდე და დავბეჭდოთ ყველა რგოლის საინფორმაციო ველი, ანუ ე.წ. გასაღები

```
void Print_List(List_int* &list)     // დამხმარე, სამუშაო ფუნქცია
          {List_int* temp=list;
              while(temp != 0)     // იფიქრეთ რატომ
                 {cout<<temp->info<<" ";
                  temp=temp->next;}
            cout<<endl;
             }
```

ძებნა სიაში. მორიგი მარტივი ფუნქცია, მან უნდა დააბრუნოს იმ პირველი რგოლის მისამართი რომლის გასაღები ტოლია საძიებელი ელემენტის, თუ საძიებელი ელემენტი სიაში არ არის, მაშინ მეთოდმა უნდა დააბრუნოს null.

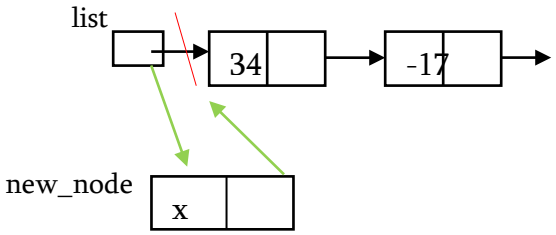
```
        List_int*  Search(List_int* &list, int  x)
             { List_int *temp=list;
              while(temp !=0)
                 {if (temp->info ==x) return temp;
                  temp=temp->next;}
```

```
return 0;
    }
```

ელემენტის დამატება სიაში

ეს მოქმედება სამ სხვადასხვა შემთხვევას მოიცავს - 1. ელემენტის დამატება სიის თავში, ამ დროს სიის მისამართი იცვლება; 2. ელემენტის დამატება სიის ბოლოში; 3. ელემენტის დამატება სორტირებულ(დალაგებულ) სიაში, ისე რომ ელემენტის ჩამატების შემდეგ სია ისევ დალაგებული იყოს. განვიხილოთ თითოეული შემთხვევა.

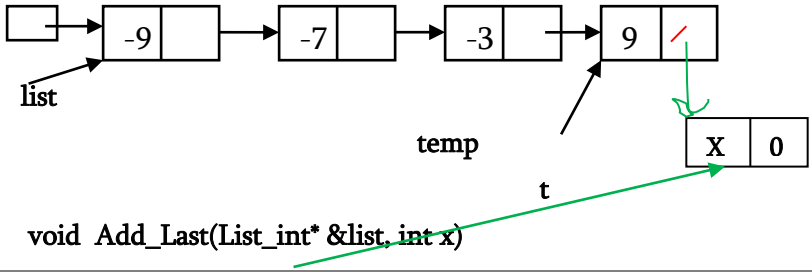
ახალი რგოლის ჩასმა სიის თავში



ყურადღება მიაქციეთ იმ ფაქტს, რომ სიის მისამართი აუცილებლად იცვლება და სიაში ნავიგაცია დაიწყება ჩამატებული ელემენტიდან.

```
void  Add_head(List_int* &list, int  x)
        {
      List_int* new_node=new List_int;
           new_node->info=x;
       new_node->next=list;        //იცვლება სიის მისამართი
       list= new_node;
           }
```

სიის ბოლოში ახალი ელემენტის დამატება



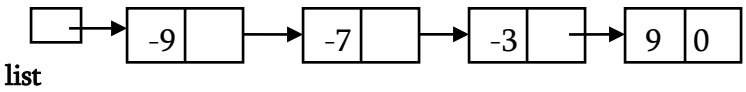

```
void  Add_Last(List_int* &list, int x)
```

```
 { List_int *t=new List_int;   // ჩასამატებელი რგოლი
List_int *temp=list;
    t->info=x;
        t->next=0;
  while(temp->next != 0)      //  იფიქრეთ რატომ
      temp=temp->next;
   temp->next=t; }
```

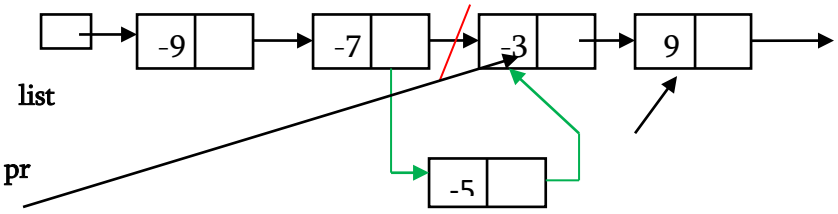
ელემენტის დამატება დალაგებულ სიაში

შედარებით რთული ფუნქციაა, მაგრამ სანამ უშუალოდ ამ საკითხის გარჩევაზე გადავალთ, შევთანხმდეთ, რომ ელემენტს არ დავამატებთ თავში, ან ბოლოში და შემდეგ არ დავალაგებთ სიას, რადგან სორტირებულ სიაში ელემენტის ჩამატება n რიგის ამოცანაა(n რგოლების რაოდენობაა სიაში), ხოლო სიის დალაგება n^2 რიგის ამოცანაა.

მოკლედ მოცემული გვაქვს უკვე სორტირებული სია და მასში უნდა ჩავამატოთ ახალი ელემენტი იქ სადაც მისი ადგილია. ვთქვათ, ჩვენი სია ასეთია:



და გვინდა ამ სიაში ჩავამატოთ ახალი ელემენტი, ვთქვათ -5. ამისათვის უნდა მოვიქცეთ ასე:



შეიძლება, ბედის ირონიით, ჩასამატებელი ელემენტი ემატებოდეს სიის თავში, ან ბოლოში, იმიტომ რომ სწორედ იქ არის მისი ადგილი. თუ ჩასამატებელი ელემენტი უნდა ჩავსვათ სიის თავში, მაშინ გამოვიძაებთ უკვე აღწერილ ფუნქციას **Add_head(list, x).**

```
void  Add_Sort(List_int* &list, int  x)
    { List_int *t=list;
        List_int *pr=0;
          while(t!=0 && x>=t->info)
                    { pr=t;                    //რატომ?
```

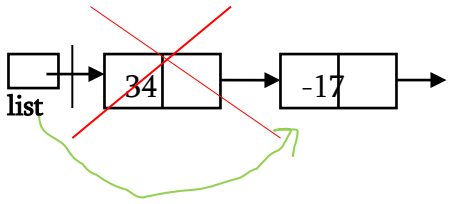
```
                t=t->next;   }
        if(t==this) Add_head(list, x);   // ელემენტი ემატება სიის თავში
              else if(t==null)  {pr->next= new List_int;    pr->next->info=x;}    // სიის ბოლოში ემატება
             else  {pr->next=new List_int;    //ემატება  pr და t რგოლებს შორის
                    pr->next->next=t;
                    pr->next->info=x;}
         }
```

ელემენტის ამოგდება სიიდან

ისევე, როგორც ელემენტის ჩამატება სიაში, აქაც სამი სხვადასხვა შემთხვევა გვაქვს - 1. პირველი ელემენტის ამოგდებასიიდან და ამ დროს სიის მისამართი იცვლება; 2. სიიდან ბოლო ელემენტის ამოგდება; 3. ისეთი ელემენტის ამოგდება რომელსაც გააჩნია როგორც მარჯვენა და ასევე მარცხენა მეზობელი. განვიხილოთ თითოეული შემთხვევა.

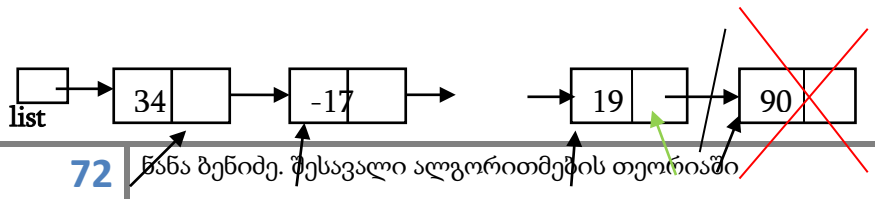
სიიდან პირველი ელემენტის ამოგდება

თუ პირველი ელემენტი უნდა ამოვაგდოთ სიიდან ეს ნიშნავს, რომ შეიცვლება სიის „თავი“, სწორედ ეს არის ყველაზე რთული ამ ფუნქციაში.



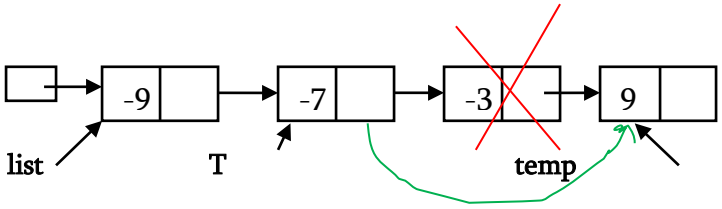

```
void  Remove_first(List_int* &list)
     {List_int *p=list;
      list=list->next;
       delete p;                //  პირველი ელემენტი ამოვაგდეთ
                }
```

სიიდან ბოლო ელემენტის ამოგდება



```
        T          temp                  T        0  temp
void  Remove_Last(List_int* &list)
   { List_int *temp=list;
           List_int *T=list;
            while(temp->next != 0)     //  temp!=null     temp.next.next!=null
              {   T=temp;                //      ოფიქრეთ რატომ
                 temp=temp->next; }
            if(T==temp)   Remove_first(list);     //სიაში ერთი ელემენტია
                 else
                { T->next=0;
                  delete temp; }
                 }
```

სიიდან ნებისმიერი ელემენტის ამოგდება



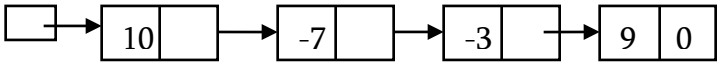
ეს ფუნქციაც საკმაოდ მარტივი ფუნქციაა, მხოლოდ უნდა ვაკონტროლთ ის ფაქტი ამოსაგდები ელემენტი პირველი ელემენტია თუ არა.

```
void remove(List_int* &list, int x)
{    List_int *temp=list;
     List_int *T=temp;
        while(temp!=0 && temp->info!=x)
          {T=temp;
             temp=temp->next;}
        if(temp==0)    cout<<"not found";
            else   { if(list->info!=x) {T->next=temp->next;
                                  delete temp; }
                   else {list=list->next;
                         T=0;
                         delete temp;} //1-ლი ელ. ამოგდება
              }
```

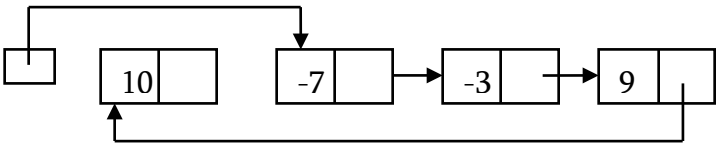
სიის დალაგება(სორტირება)

სორტირების უკვე ცნობილი ალგორითმებიდან(ამორჩევით სორტირება, მეზობელი ელემენტების შედარების გზით სორტირება, რეკურსიული სწრაფი სორტირება) გამოდგება მხოლოდ მეზობელი ელემენტების შედარების გზით სორტირება, ანუ bubl sort-ის გაუმჯობესებული ვერსია, რადგან როგორც ვთქვით სიაში რამდენი ელემენტია, ეს არ არის წინასწარ ცნობილი.(მხოლოდ ეს არ არის მთავარი მიზეზი, ამიტომ იფიქრეთ რატომ?)

დავალაგოთ სია ზრდადობით(კლებადობით) ნიშნავს, რომ სიის ყოველი რგოლის გასაღები, მორიგი რგოლის გასაღებზე ნაკლები(მეტი) უნდა იყოს, მაგალითად თუ საწყისი სია ასე გამოიყურება:



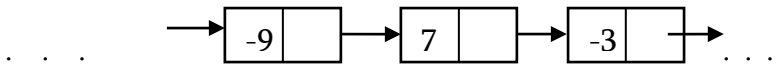
დალაგების შემდეგ მისი სახე ასეთი უნდა იყოს:



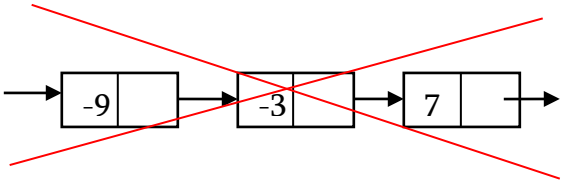
ნახ 1.

თუ ნახაზს დააკვირდებით, სიის რგოლები განლაგება მეხსიერებაში უცვლელი დარჩა, მხოლოდ კავშირი გაწყდა 10-სა და -7-ს შორის, სამაგიეროდ გაჩნდა ახალი კავშირი 9-სა და 10-ს შორის. ე.ი. თუ წინა რგოლის გასაღები მორიგი რგოლის გასაღებზე ნაკლები არ არის, მაშინ უნდა გადავანიჭოთ **რგოლების მისამართები და არვითარ შემთხვევაში არ უნდა გადავანიჭოთ მათი საინფორმაციო ველების მნიშვნელობები.**

რას ნიშნავს გადავანიჭოთ რგოლების მისამართები? ვთქვათ:

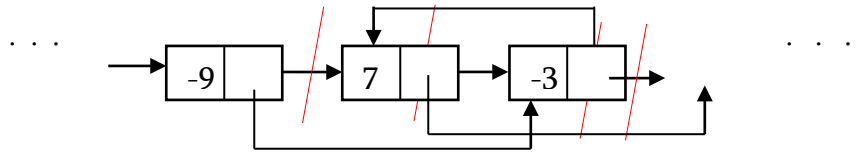


მე-2 და მე-3 რგოლი აშკარად არ დგას თავის ადგილზე, სიტუაცია რომ გამოვასწოროთ



ასე არ უნდა მოვიქცეთ

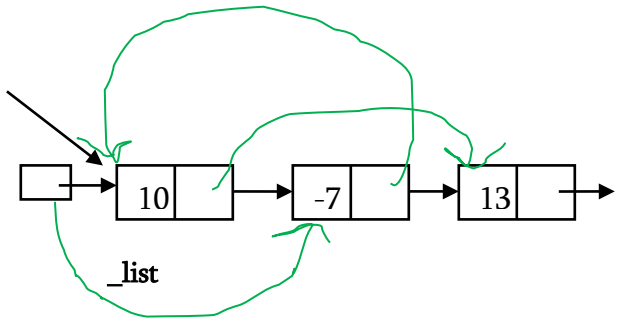
უნდა მოვიქცეთ ასე:

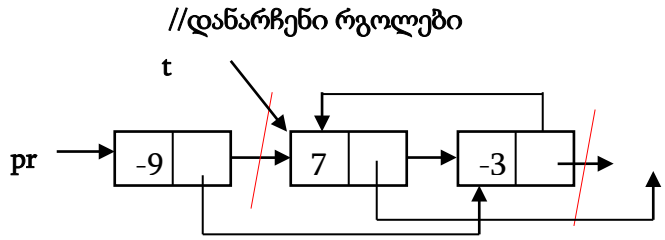


უნდა გადავანიჭოთ მისამართები: რგოლი გასაღებით -9 უნდა მიუთითებდეს რგოლზე გასაღებით -3, ეს უკანასკნელი კი უნდა მიუთითებდეს რგოლზე გასაღებით 7, რომელიც თავის მხრივ მიუთითებს იმ რგოლზე რომელზეც მიუთითებდა -3. ყურადღება მიაქციეთ: ყველა რგოლი მეხსიერებაში უცვლელად რჩება, მხოლოდ მათი წინა და მომდევნო რგოლებზე მიმთითებლები იცვლება.

თუ ნახ. 1-ს დააკვირდებით შენიშნავთ, რომ სიის თავი შეიცვალა, ანუ სიაში ნავიგაცია უკვე იწყება არა 10-დან, არამედ -7-დან, ე.ი. სიის სორტირებისას უნდა ვაკონტროლოთ ის ფაქტი, ხომ არ შეიცვალა სიის სასტარტო რგოლი. თუმცა როცა სიის პირველი ელემენტი იცვლება, ან ბოლო ელემენტი იცვლება მაშინ ფრაგმენტი ცოტა სხვანაირია, რადგან სიის პირველ და ბოლო ელემენტს მხოლოდ ერთი მეზობელი ჰყავს, ხოლო სიის ყველა დანარჩენ ელემენტებს ორი მეზობელი ჰყავს.

```
void  Sort_List(List_int* &list)
{List_int *t=list;
 List_int *_list=list;
 List_int pr=0;
 bool is_sorted=true;
     while(is_sorted)
     {is_sorted=false; // პირველი ორი რგოლის დამუშავება
                                                  t


                                                 _list
     if(t->info>t->next->info)  {_list=t->next;
                                 t->next=_list->next;
                                 _list->next=t;
                                  is_sorted=true;
                                    pr=_list;}
           else {pr=t; t=t->next;}
```






```
while(t->next != 0)
 if(t->info>t->next->info) { is_sorted=true;
                                pr->next=t->next;
                                t->next=t->next->next;   //?
                                 pr->next->next=t;
                                 pr=pr->next;}
                    else {pr=t; t=t->next;}   // end inside loop

          t=_list; }    //end outside loop

list = _list;} // end sort
```

დასკვნა: საბოლოოდ შევაჯამოთ ყველაფერი, მაშინ ცალადმული სიის შესაბამისი ბიბლიოთეკა მიიღებს შემდეგ სახეს:(ეს ბიბლიოთეკა ჩავწეროთ სამომხმარებლო “list.h” ფაილში)

```
void Build(List_int* &list, ifstream &F)
{
  list=new List_int;
  List_int *p=0, *pr=0;
  p=list;
  while(!F.eof())
  {
   F>>p->info;
   p->next=new List_int;
   pr=p;
   p=p->next;}
    delete p;
    pr->next=0;
  }

  void Print_List(List_int* &list)
         {List_int* temp=list;
             while(temp != 0)
                 {cout<<temp->info<<" ";
```

```
                temp=temp->next;}
                cout<<endl;
                                                    }

  List_int*  Search(List_int* &list, int  x)
    {List_int *temp=list;
            while(temp !=0)
                {if (temp->info ==x) return temp;
                 temp=temp->next;}
               return 0;   }
void  Add_head(List_int* &list, int  x)
    {
          List_int* new_node=new List_int;
               new_node->info=x;
            new_node->next=list;
            list= new_node;
                 }

 void  Add_Last(List_int* &list, int x)
    { List_int *t=new List_int;
              t->info=x;
                          t->next=0;
          List_int *temp=list;
            while(temp->next != 0)
               temp=temp->next;
             temp->next=t; }

 void  Add_Sort(List_int* &list, int  x)
    { List_int *t=list;
          List_int *pr=0;
             while(t!=0 && x>=t->info)
                        { pr=t;
                          t=t->next;   }
           if(t==list) Add_head(list, x);
                  else if(t==0)  {pr->next= new List_int;    pr->next->info=x;}
                 else  {pr->next=new List_int;
                        pr->next->next=t;
                        pr->next->info=x;}
            }


  void  Remove_first(List_int* &list)
    {List_int *p=list;
     list=list->next;
      delete p;
```

```
                }

  void  Remove_Last(List_int* &list)
   { List_int *temp=list;
           List_int *T=list;
            while(temp->next != 0)     //  temp!=null     temp.next.next!=null
              {   T=temp;
                 temp=temp->next; }
            if(T==temp)   Remove_first(list);
                 else
                { T->next=0;
                  delete temp; }
                 }

void remove(List_int* &list, int x)
{  List_int *temp=list;
  List_int *T=temp;
      while(temp!=0 && temp->info!=x)
        {T=temp;
           temp=temp->next;}
      if(temp==0)    cout<<"not found";
          else   if(list->info!=x) {T->next=temp->next;
                                    delete temp; }
                 else {list=list->next;
                       T=0;
                       delete temp;}
           }

 void  Sort_List(List_int* &list)
 {List_int *t=list;
   List_int *_list=list;
   List_int *pr=0;
   bool is_sorted=true;
       char c;
    while(is_sorted)
      {is_sorted=false;
       if(t->info>t->next->info)  {_list=t->next;
                                   t->next=_list->next;
                                   _list->next=t;
                                    is_sorted=true;
                                     pr=_list;}
           else {pr=t; t=t->next;}
        while(t->next != 0)
        if(t->info>t->next->info) { is_sorted=true;
                                    pr->next=t->next;
```

```
                        t->next=t->next->next;
                                pr->next->next=t;
                              pr=pr->next; }
                    else {pr=t; t=t->next;}   // end inside loop

              t=_list; }    //end outside loop
          list = _list;} // end sort
```

ხოლო list.cpp ფაილს ექნება სახე:

```
#include <fstream>
#include <iostream>
#include "list.h"
using namespace std;

struct List_int {
     int info;
     List_int* next; };

    main()
    {ifstream F;
     F.open("f.txt", ios::in);
     if (!F.is_open()){cout<<"error"; exit(0);
           }
    List_int* list=0;
      Build(list,F);
      Print_List(list);

     /* if(Search(list,-22)!=0) cout<<"is found\n";
                  else cout<<"not found\n";

      Add_head(list, -900);
      Print_List(list);

       Add_Last(list, -900);
      Print_List(list);

          Add_Sort(list, 90);
      Print_List(list);

          Remove_first(list);
          Print_List(list);

                  Remove_Last(list);
          Print_List(list);
```

```
    remove(list, 4);
Print_List(list);    */

    Sort_List(list);
     Print_List(list);}
```

ჩვენ ავღწერეთ მთელი რიცხვების ცალადბმული სია და მისი საბაზო ფუნქციების ბიბლიოთეკა, ბუნებრივია, ამ ბიბლიოთეკას უამრავი რაღაც შეიძლება დაემატოს, რაც დამოკიდებულია კონკრეტულ ამოცანაზე. თუმცა თუ თქვენ ფლობთ საბაზისო უნარებს მონაცემთა დინამიურ სტრუქტურებთან სამუშაოდ, სხვა დამატებითი ფუნქციების აღწერა არ არის რთული.

დავალება: გააკეთეთ ყველა ფუნქციის ტესტირება, თუ რაიმე შეცდომას, ან უზუსტობას აღმოაჩენთ შეასწორეთ და ლაბორატორიულზე განვიხილოთ.