# თემა 1. საწყისი ცნებები და ცნობები

განხილული საკითხები:

- მონაცემთა სტრუქტურები, მონაცემთა აბსტრაქტული ტიპი
- STL ბიბლიოთეკაში რეალიზებული მონაცემთა სტრუქტურები >>>
- კონტეინერების კლასიფიკაციის საფუძვლები >>>
- ასიმპტოტური აღნიშვნები, სისწრაფესთან დაკავშირებული განმარტებები >>>
- ნამდვილი რიცხვის (ქვედა და ზედა) მთელი მიახლოებები >>>
- ლოგარითმული ფუნქციის ერთი მნიშვნელოვანი გამოყენება >>>
- ლიტერატურა >>>

## მონაცემთა სტრუქტურები, მონაცემთა აბსტრაქტული ტიპი

კომპიუტერულ მეცნიერებაში, **მონაცემთა სტრუქტურები** გამოიყენება კომპიუტერში მონაცემთა შენახვისთვის და მათზე გარკვეული მოქმედებების განხორციელებისთვის.

ერთი და იმავე გამოყენებითი ამოცანის გადაჭრა ხშირად შესაძლებელია რამდენიმე განსხვავებული მონაცემთა სტრუქტურის გამოყენებით. ასეთ შემთხვევაში, მონაცემთა სტრუქტურის შერჩევისთვის განმსაზღვრელი მნიშვნელობა აქვს თუ რომელი სტრუქტურის ოპერაციები უფრო სწრაფად იმუშავებს მოცემულ ამოცანაში.

თანამედროვე მიდგომით, მონაცემთა სტრუქტურების დიზაინისთვის (აგებისთვის) ძირითად კონცეფციას წარმოადგენს **მონაცემთა აბსტრაგირება**, ანუ **მონაცემთა აბსტრაქტული ტიპი** (ADT). მონაცემთა აბსტრაქტული ტიპი გვთავაზობს ობიექტების სიმრავლეს და ამ ობიექტებზე განსაზღვრული ოპერაციების სიმრავლეს ([1], გვ.127). ტერმინი აბსტრაქტული ნიშნავს, რომ მომხმარებლისთვის ცნობილია მხოლოდ ADT-ს ინტერფეისი (ანუ ღია წვდომის მეთოდების აღწერილობები), მაგრამ ოპერაციების იმპლემენტაცია მისთვის დახურულია), რასაც C++ ენის კლასის ცნება ადვილად უზრუნველყოფს. ასეთი მიდგომა აზღვევს პროგრამისტებს, რომ მათ მიერ შექმნილ კოდში არ გამოჩნდეს იმპლემენტაციის დეტალებზე დამოკიდებული წვრილმანები, რაც თავის მხრივ უზრუნველყოფს რომ მონაცემთა აბსტრაგირების საფუძველზე შექმნილი პროგრამული უზრუნველყოფა იქნება მოქნილი, ადვილად დაექვემდებარება მოდიფიკაციას და გათანამედროვეობას პროგრამული პროდუქტის შექმნიდან მისი მთელი არსებობის დროის განმავლობაში. მარტივ მაგალითად შეგვიძლია მოვიყვანოთ C++ ენის პრიმიტიული ტიპები, რომლებიც აქ C ენისგან გადმოვიდა. მაგალითად **int**. ამ ტიპის შესახებ ვიცით თუ რა ოპერაციებია მასზე დასაშვები, მაგრამ ჩვენთვის არაა საინტერესო თუ როგორაა ეს ოპერაციები რეალიზებული. დროთა განმავლობაში **int** ტიპის შინაგანი წარმოდგენა და იმპლემენტაცია იცვლება, მაგრამ მისი ინტერფეისი უცვლელია, რაც ძალიან კომფორტულია პროგრამისტებისთვის. მაგალითად, რამდენიმე წლის წინ **int** ის ერთი ობიექტი (ცვლადი) იკავებდა ორ ბაიტს, ამჟამად ძირითადად გავრცელებულია 4 ბაიტიანი წრმოდგენა, უკვე ხდება გადასვლა 8 ბაიტიანზე. ცხადია, წარმოდგენის შეცვლა იწვევს იმპლემენტაციის შეცვლას. მაგრამ ჩვენ ვერ ვგრძნობთ ამას, - მხოლოდ ინტერფეისის გამოყენებით დაწერილი პროგრამა იმუშავებს განსხვავებული წარმოდგენების მქონე კომპიუტერებზე.

აქვე შევნიშნოთ, რომ ხშირად აუცილებელია რომ მონაცემთა ტიპს მკაცრად განსაზღვრული დიაპაზონი ჰქონდეს, რაც არ იცვლება ენის სტანდარტების მიხედვით. ასეთი მთელი ტიპების ნიმუშებია `int8_t, int16_t, int32_t, int64_t` და მათი უნიშნო სახესხვაობები `uint8_t, ...`

მონაცემთა სტრუქტურების იმპლემენტაცია დამოკიდებულია იმ ენის შესაძლებლობებზე, რომლითაც ვაპროგრამებთ. ობიექტზე ორიენტირებულ ენებში (C++, C#, Java და სხვა),

აბსტრაგირებისთვის გამოიყენება კლასი. ამ ენებს აქვთ დამატებითი შესაძლებლობები, რაც გამოიხატება ობიექტზე ორიენტირებულ პროგრამირების სტილთან ერთად მის მიერ განზოგადებული დაპროგრამების (generic programming) სტილის მხარდაჭერაში.

თანამედროვე ენები გვთავაზობენ აგრეთვე ერთდროულად (concurrent) მიმდენარე პროცესების დამუშავების შესაძლებლობას მრავალი დინების (thread) გამოყენებით. ძალიან აქტურალურია ისეთი მონაცემთა სტრუქტურების დამუშავება, ან არსებული იმპლემენტაციების გამოყენება, რომლებიც მრავალდინებიან გარემოში კორექტულად ფუნქციონირებენ და, უფრო მეტიც, მრავალი დინების არსებობას იყენებენ შედეგის უფრო სწრაფად მისაღწევად.

ჩვენს კურსში ვიმუშავებთ STL ბიბლიოთეკის კონტეინერებთან, ხშირ შემთხვევაში ავხსნით იმ ალგორითმებს, რაც საფუძვლად უდევს მათ მეთოდებს (კონტეინერში ძებნა, შემოვლა და სხვა).

მოკლედ მიმოვიხილოთ რამდენიმე საკითხი, რაც დაგვჭირდება შემდეგ ლექციებში.

## <<< STL ბიბლიოთეკაში რეალიზებული მონაცემთა სტრუქტურები

ამ ბიბლიოთეკის აღწერისთვის ჩვენ ვეყრდნობით [1]-ს და [2]-ს. STL ბიბლიოთეკის მონაცემთა სტრუქტურები დაყოფილია რამდენიმე მონათესავე ჯგუფად, ე.წ. განზოგადებათა ოჯახებად (families of abstractions). თითოეულ ასეთ ოჯახში მეთოდების უმეტესობა არის საზიარო. თუ ოჯახიდან ჩვენ გავარჩევთ ერთ მონაცემთა სტრუქტურას, იგივე ოჯახის დანარჩენი სტრუქტურების შესაძლებლობების გასაცნობად საკმარისია ჩამოვთვალოთ თუ რა განასხვავებს მას უკვე განხილული წარმომადგენლისაგან.

STL ბიბლიოთეკიდან ჩვენ განვიხილავთ მონაცემთა სტრუქტურების რამდენიმე ოჯახს: მიმდევრობის (შესანახ) კონტეინერებს (ვექტორი, დეკი, სია), დახარისხებულ ასოციაციურ კონტეინერებს (ასახვა, მულტი ასახვა, სიმრავლე, მულტი სიმრავლე), დაუხარისხებელ ასოციაციურ კონტეინერებს (დაუხარიხებელი ასახვა/მულტი ასახვა/სიმრავლე/მულტი სიმრავლე), და კონტეინერთა ადაპტერებს (სტეკი, დეკი, პრიორიტეტების რიგი).

კონტეინერი, როგორც სახელწოდებიდან ჩანს, არის ობიექტი, რომელიც ინახავს ერთი და იმავე ტიპის ობიექტების კრებულს. დაპროგრამების თანამედროვე ენებში კონტეინერების მდიდარი არჩევანი არსებობს. კონკრეტულ მაოცანაში მათი შერჩევა დამოკიდებულია იმაზე, თუ რა ოპერაციების განხორციელება იგეგმება მონაცემებზე. სხვადასხვა კონტეინერს თავ-თავისი უპირატესობები აქვს.

თუ შენახვა ხდება მხოლოდ ქრონოლოგიური რიგის (მონაცემების შემოსვლის დროის) გათვალისწინებით, მაშინ გვაქვს მიმდევრობის კონტეინერი. ვექტორის კლასი არის ამ ტიპის ერთ-ერთი წარმომადგენელი. მაგალითად, თუ შექვმნით ვექტორს განაცხადით:

```
vector<int> v = { 44, 11, 24, -54, 22 };
```

და დავბეჭდავთ მას:

```
for (auto m: v )
      cout << m << '\t' ;
cout << endl;
```

შედეგად მივიღებთ იგივე მიმდევრობას: `44, 11, 24, -54, 22`.

ასოცირებული კონტეინრები ახალი ელემენტის შესანახი ადგილი ამ ელემენტის გარკვეულ მახასიათებელთან ასოცირდება, რაც აჩქარებს შენახვასაც, მოძებნასაც და მერე წაშლასაც.

დახარისხებული ასოცირებული კონტეინერი მონაცემების შენახვის პროცესში მონაცემებს ადარებს ერთმანეთს და ახარისხებს მათ რაღაც პარამეტრის (გასაღების) საფუძველზე. მაგალითად, თუ შექვმნით დახარიხებულ სიმრავლეს განაცხადით:

```
set<int> v = { 44, 11, 24, -54, 22 };
```

და დავბეჭდავთ (იგივე მეთოდით), მივიღებთ დახარისხებულ მიმდევრობას:

```
-54      11      22      24      44
```

დაუხარისხებული ასოცირებული კონტეინერი მონაცემების შენახვის პროცესში ითვალისწინებს მონაცემების სიდიდეს, თუმცა მათ არ ადარებს ერთმანეთს. თუ შევქმნით დაუხარიხებულ სიმრავლეს განაცხადით:

```
unordered_set<int> w = { 44, 11, 24, -54, 22, 43, 12, -32, 32, 123};
```

დავბეჭდვის შემდეგ მივიღებთ:

```
12      44      -32     43      11      32      24      -54     22      123
```

თუ რა პრინციპის მიხედვით ხდება ელემენტების სიდიდის გათვალისწინება მათი შენახვის პროცესში, ვნახავთ ჰეშ-ცხრილების შესწავლის დროს.

რაც შეეხება კონტეინერების ადაპტერებს, მათ ეს სახელი ჰქვიათ იმ უბრალო მიზეზის გამო, რომ მათი ინტერფეისი ახდენს ერთ-ერთი არსებული კონტეინერის ადაპტირებას, სპეციფიკური ოპერაციების განხორციელების მიზნით (მაგალითად, სტეკი შეიძლება წარმოადგენდეს ადაბტირებულ ვექტორს. აქ ადაპტირება ნიშნავს, რომ ვექტორის მეთოდების ნაწილი გაუქმებულია, ნაწილი გადაკეთებული).

ჩვენს კურში შევისწავლით აგრეთვე ორობით გროვას და მასთან დაკავშირებულ ალგორითმებს. ეს მონაცემთა სტრუქტურა გამოიყენება ვექტორის ადაპტირებისთვის პრიორიტეტების რიგის შექმნის მიზნით.

## <<< კონტეინერების კლასიფიკაციის საფუძვლები

როგორც აღვნიშნეთ, კონტეინერი არის ობიექტი, რომელიც ინახავს სხვა (ერთი და იმავე ტიპის) ობიექტების კრებულს. STL ბიბლიოთეკა მხოლოდ ერთ-ერთი საშულება არის კონტეინერების შექმნისა და გამოყენებისთვის. კონტეინერები არსებობენ სხვა თანამედროვე ენებში, და კონტეინერები არსებობენ აბსტრაქტულად.

კონტეინერები ერთმანეთისგან განსხვავდებიან ელემეტებზე წვდომის მიხედვით, და ელემენტების შენახვის პრინციპით.

მაგალითად, მიმდევრობის კონტეინერები, ვექტორი, სია და დეკი, ელემენტებს ინახავენ მათი შემოსვლის რიგის მიხედვით, ან მითითებული ადგილის მიხედვით, ელემენტის რიცხვითი მახასიათებლების გაუთვალისწინებლად. მაგალითად, მთელი რიცხვის შენახვის დროს არავითარი მნიშვნელობა არა აქვს ეს რიცხვი დიდია თუ მცირე. ამავე დროს, ეს სამი კონტეინერი ერთმანეთისგან განსხვავდება მათ ელემენტებზე წვდომის თვალსაზრისით: ვექტორისა და დეკის ნებისმიერ ელემენტს შეგვიძლია მივწვდეთ (ანუ ვნახოთ მისი მნიშვნელობა და აუცილებლობის შემთხვევაში შევცვალოთ კიდეც) დროის ფიქსირებულ შუალედში (როგორც ამბობენ, $O(1)$ დროში). ამათგან განსხვავებით, სიაში ელემენტზე წვდომის დრო დამოკიდებულია იმაზე, თუ რამდენად ახლოსაა ან შორს ეს ელემენტი მოთავსებული სიის თავიდან. როდესაც ელემენტზე წვდომა ხორციელდება ფიქსირებულ დროში, ამბობენ რომ გვაქვს **სწრაფი წვდომა** (random access). სიის შესახებ ვამბობთ, რომ მას აქვს წვდომა წრფივ დროში (ე.ი. წვდომის საშუალო დრო წრფივადაა დამოკიდებული სიაში ელემენტების რაოდენობაზე).

კონტეინერის ელემენტებზე წვდომის დრო დამოკიდებულია მეხსიერებაში კონტეინერის განთავსების მეთოდზე. ვექტორის და დეკის ობიექტის შესანახად, დროის ყოველ მომენტში, მეხსიერებაში გამოყოფილია მონაკვეთი, რომელიც მთლიანად ეთმობა ამ ობიექტს (თანამედროვე იმპლემენტაციები უფრო ცბიერია და შესაძლოა რამდენიმე მონაკვეთს იყენებდეს ერთად). სიის ობიექტის შესანახად, ერთი მთლიანი ფრაგმენტი არ არის გამოყოფილი. სიაში შესანახ ყოველ ინფორმაციას ეწებება მომდენო ინფორმაციის მისამართი და ასე გაბნეულად ხდება მათი განთავსება მეხსიერებაში. ორივე მეთოდს აქვს თავისი დადებითი და უარყოფითი მხარეები. ვექტორის და დეკის შემთხვევაში, უპირატესობას წარმოადგენს სწრაფი წვდომა ელემენტებზე, ხოლო მინუსს წარმოადგენს ის ფაქტი, რომ ელემენტის ჩამატება და წაშლა, თუ ის ბოლოში არაა, ხდება წრფივ (და არა ფიქსირებულ) დროში. სიის უპირატესობები არის: მას არ სჭირდება ერთიანი მონაკვეთი მეხსიერებაში, ამიტომ სიაში შეგვიძლია შევინახოთ უფრო მეტი მონაცემი, ვიდრე წინა შემთხვევაში (შეიძლება მეხსიერებაში ბევრი ადგილი იყოს, მაგრამ არ იყოს ერთი მთლიანად თავისუფალი დიდი მონაკვეთი), მითითებულ მისამართზე ელემენტის ჩამატება და წაშლა ხორციელდება ფიქსირებულ დროში. მინუსი ისაა, რომ საჭირო თვისების მქონე ელემენტის მოძებნა, ან ელემენტის მნიშვნელობის ამოღება ხდება წრფივ (და არა ფიქსირებულ) დროში.

ასოცირებულ კონტეინერებში, შესანახი ელემენტის „ადგილის“ განსაზღვრა კონტეინერში ხდება ამ ელემენტის გარკვეული მახასიათებლების გათვალისწინებით, რაც დამატებით შესაძლებლობას ქმნის ზოგიერთი ოპერაციის დაჩქარებითვის.

## <<< ასიმპტოტური აღნიშვნები, სისწრაფესთან დაკავშირებული განმარტებები

აღნიშვნები და ძირითადი ფაქტები (გარდა ბოლო პუნქტისა) აღებული გვაქვს [2]-იდან.

როდესაც ვლაპარაკობთ რაიმე მოცემული ალგორითმის სისწრაფეზე, ჩვენ მხედველობაში გვაქვს თუ როგორ არის დამოკიდებული ალგორითმის შესრულების დრო (computing time) ალგორითმების არგუმენტების ცვლილებაზე. ცხადია, რომ **რთულ** და **დიდ** ამოცანებს მეტი დრო დასჭირდებათ შესრულებისთვის. მაგრამ იმისათვის, რომ რაოდენობრივ მახასიათებლებში მოვაქციოთ ალგორითმის სისწრაფე და არ შევიზღუდოთ მხოლოდ სტატისტიკით, თუ რა პირობებში რა მოხდა, უნდა შეგვეძლოს მოცემული ამოცანის **ზომის** განსაზღვრა.

ამოცანის ზომის განსაზღვრისთვის არგუმენტები უნდა გავყოთ ორად: რომელთა ცვლილება გავლენას არ ახდენს შესრულების დროზე, მათი გათვალისწინება საჭირო არაა. ამოცანის ზომას განსაზღვრავს დანარჩენების ერთობლიობა. მაგალითად, განვიხილოთ რიცხვის არაუარყოფით ხარისხში ახარისხების საკითხი და ვისარგებლოთ $f(a,n) \equiv a^n$ აღნიშვნით. რომელი ალგორითმიც არ უნდა განვიხილოთ, ყოველთვის ცხადია რომ $a$ რიცხვის ცვლილება გავლენას არ ახდენს ჩასატარებელი ოპერაციების რაოდენობაზე, სულ ერთია ეს რიცხვი დიდია თუ მცირე, სისწრაფეზე იგი გავლენას არ ახდენს. ამიტომ, ამოცანის ზომად უნდა მივიღოთ მეორე არგუმენტი $n$, რომლის ცვლილება არსებით გავლენას ახდენს ოპერაციების რაოდენობაზე. ზოგიერთ მარტივ ამოცანაში მხოლოდ ერთი არგუმენტია და ის ააღწერს ზომას, ხშირად რამდენიმე არგუმენტი ერთდროულად ახდენს გავლენას სისწრაფეზე.

თუ ამოცანის ზომას აღწერს არაუარყოფითი მთელი $n$ რიცხვი, მაშინ, როგორც წესი $T(n)$-ით აღინიშნება ის მაქსიმალური დრო, რომელიც ალგორითმის შესრულებას სჭირდება $n$ ზომის არგუმენტისთვის.

$n$ რიცხვის დიდი მნიშვნელიბებისთვის $T(n)$-ის შესაფასებლად, ზუსტი ფორმულების დაწერის ნაცვლად უნდა გამოვიყენოთ გაცილებით მარტივი მეთოდი: შევარჩიოთ საკმაოდ მარტივი სახის ფუქცია, რომელიც ზემოდან შემოსაზღვრავს $T(n)$-ს. მაგალითად, ჩვენ შესაძლოა გვქონდეს:

$$T(n) \le c \cdot n \qquad (1)$$

რომელიმე $c$ მუდმივისა და ყველა $n$-ისთვის გარდა მისი რამდენიმე მნიშვნელობისა (ანუ ყველა საკმაოდ დიდი $n$-ისთვის. ეკვივალენტურ ტერმინებს ცალკე დავუთმობთ დროს ამავე პარაგრაფში). ამ შემთხვევაში ვამბობთ, რომ "დრო უარეს შემთხვევაში იზრდება წრფივად ამოცანის ზომის ცვლილებასთან ერთად". $c$ მუდმივი შეიძლება სხვადასხვა მნიშნელობებს ღებულობდეს, როდესაც ალგორითმი ეშვება სხვადასხვა სიმძლავრის კომპიუტერზე. ამიტომ საჭიროა კიდევ უფრო გავამარტივოთ $T(n)$-ის შეფასების მეთოდიკა ისე, რომ მუდმივები ცხადად არ მონაწილეობდნენ შეფასებაში.

ეს კეთდება ე.წ. $O$-დიდი აღნიშვნების გამოყენებით. მაგალითად, (1) შეფასება ჩაიწერება ასე:

$$T(n) = O(n).$$

იმისათვის, რომ მკაცრად ჩამოვაყალიბოთ განმარტებები, განვიხილოთ მხოლოდ მთელი არაუარყოფითი არგუმენტის ასიმპტოტურად არაუარყოფითი (asymptotically nonnegative) ფუნქციები, ანუ ფუნქციები, რომლებმაც მხოლოდ რამდენიმე (ანუ სასრულ რაოდენობა) წერტილში შეიძლება მიიღონ უარყოფითი მნიშვნელობა.

ყურადღება მივაქციოთ, რომ თუ $f(n)$ არის მთელი არაუარყოფითი არგუმენტის ნამდვილი ფუნქცია, მაშინ შემდეგი წინადადებები არის ეკვივალენტური:

1. $f(n) < 0$ რამდენიმე წერტილში;
2. $f(n) < 0$ სასრულ რაოდენობა წერტილებში;
3. $f(n) \ge 0$ რამდენიმე წერტილის გარდა;
4. $f(n) \ge 0$ დაწყებული რაღაც $n_0$-იდან;
5. არსებობს $n_0 \in \mathbb{N}$, ისეთი რომ $f(n) \ge 0$ როცა $n \ge n_0$.

**განმარტება**. ვამბობთ, რომ $f(n) = O(g(n))$, თუ რომელიმე $c > 0$ მუდმივისთვის სრულდება $0 \le f(n) \le cg(n)$ გარდა $n$-ის რამდენიმე მნიშვნელობისა. □

სხვა სიტყვებით, $f(n) = O(g(n))$ გამომდინარეობს შემდეგი სამი წინადადებიდანაც:

- სრულდება $0 \le f(n) \le cg(n)$ გარდა $n$-ის სასრული რაოდენობის მნიშვნელობებისა;
- სრულდება $0 \le f(n) \le cg(n)$ დაწყებული რაღაც $n_0$-იდან;
- არსებობს $n_0 \in \mathbb{N}$, ისეთი რომ სრულდება $0 \le f(n) \le cg(n)$ როცა $n \ge n_0$.

$f(n) = O(g(n))$ მოკლედ გამოითქმის ასე: ეფ ენი უდრის ო-დიდ ჟე ენს, ხოლო შინაარსის გათვალისწინებით ვამბობთ რომ $f(n)$ ასიმპტოტურად ისევე სწრაფად იზრდება როგორც $g(n)$ ან უფრო ნელა, ან რომ $f(n)$ -ის ზრდის რიგი არ აღემატება $g(n)$-ისას.

მაგალითად, თუ $f(n) = 3n^4$ და $g(n) = 2n^4 + n^3 - 23n - 111$, მაშინ $f(n) = O(g(n))$. იგივე იქნება, თუ $f(n)$ იქნება მეოთხეზე დაბალი ხარისხის ნებისმიერი პოლინომი დადებითი კოეფიციენტით უმაღლესი ხარისხის მქონე წევრთან.

**შენიშვნა**. რადგან ფიქსირებული $g(n)$ ფუნქციისთვის $f(n) = O(g(n))$ სრულდება უამრავი $f(n)$-ისთვის, ამიტომ $O(g(n))$ სინამდვილეში განსაზღვრავს სიმრავლეს ანუ კლასს. ამიტომ,

ამჟამად ბევრი ავტორი ტრადიციული $f(n)=O(g(n))$ აღნიშვნის ნაცვლად წერს $f(n)\in O(g(n))$, ან უბრალოდ $f\in O(g(n))$.

**განმარტება**. ვამბობთ, რომ $f(n)=\Omega(g(n))$, როდესაც $g(n)=O(f(n))$. □

$f(n)=\Omega(g(n))$ მოკლედ გამოითქმის ასე: ეფ ენი უდრის სიგმა-დიდ ჟე ენს, ხოლო შინაარსის გათვალისწინებით ვამბობთ რომ $f(n)$ ასიმპტოტურად ისევე სწრაფად იზრდება როგორც $g(n)$ ან უფრო სწრაფად, ან რომ $f(n)$ -ის ზრდის რიგი არ ჩამორჩება $g(n)$-ისას.

$f(n)=\Omega(g(n))$-ის მნიშვნელობით ხშირად ხმარობენ ჩანაწერს $f(n)\in\Omega(g(n))$, რადგან $\Omega(g(n))$ წარმოადგენს ფუნქციების კლასს.

**განმარტება**. როდესაც $g(n)=O(f(n))$ და $f(n)=O(g(n))$ ერთდროულად სრულდება, ვამბობთ, რომ $f(n)=\Theta(g(n))$. □

$f(n)=\Theta(g(n))$ მოკლედ გამოითქმის ასე: ეფ ენი უდრის თეტა-დიდ ჟე ენს, ხოლო შინაარსის გათვალისწინებით ვამბობთ რომ $f(n)$ ასიმპტოტურად ისევე სწრაფად იზრდება როგორც $g(n)$, ან რომ მათ ზრდის ერთნაირი რიგი აქვთ. აქაც, ხშირად გამოიყენება ჩანაწერი: $f(n)\in\Theta(g(n))$

შემდეგ ცხრილში მოყვანილია ასიმპტოტური ზრდის რამდენიმე რამდენიმე ძალიან გავრცელებული კლასი:

| აღნიშვნა | სახელი |
|---|---|
| $O(1)$ | მუდმივი |
| $O(\log\log n)$ | ლოგ-ლოგარითმული |
| $O(\log n)$ | ლოგარითმული |
| $O(n)$ | წრფივი |
| $O(n\log n)=O(\log n!)$ | ლოგწრფივი, კვაზიწრფივი |
| $O\left(n^2\right)$ | კვადრატული |
| $O\left(n^3\right)$ | კუბური |
| $O\left(c^n\right), c>1$ | ექსპონენციალური ან გეომეტრიული |
| $O(n!)$ | ფაქტორიალური ან კომბინატორული |

ასიმპტოტური შეფასებები ძალიან მარტივი შესამოწმებელია პოლინომებისთვის. მაგალითად, თუ არაუარყოფითი $f(n)$ ფუნქცია არგუმენტის რამდენიმე მნიშვნელობის გარდა ყველგან ნაკლებია ან ტოლი ვიდრე პოლინომი $a_0+a_1\cdot n+\cdots+a_k\cdot n^k$ და $a_k>0$, მაშინ $f(n)\in O\left(n^k\right)$. თუ იგივე პირობებში $f(n)$ მეტია ან ტოლი ამ პოლინომზე, მაშინ $f(n)\in\Omega\left(n^k\right)$.

## <<< ნამდვილი რიცხვის (ქვედა და ზედა) მთელი მიახლოებები

ყოველი ნამდვილი $x$ რიცხვისთვის $\lfloor x\rfloor$-ით აღინიშნება $x$-ის მთელი ნაწილი, ანუ მისი ქვედა მიახლოება. ესაა უდიდესი მთელი რიცხვი, რომელიც არ აღემატება $x$-ს. $\lceil x\rceil$-ით აღინიშნება უმცირესი მთელი რიცხვი, რომელიც არაა ნაკლები $x$-ზე. ცხადია, ყოველთვის სრულდება

$$x-1<\lfloor x\rfloor\le x\le\lceil x\rceil<x+1,$$

ისევე როგორც $\lceil n/2\rceil+\lfloor n/2\rfloor=n$ ყოველი მთელი n-ისთვის. დასასრულ, ყოველი ნამდვილი x-ისა და ყოველი ნატურალური $a,b$ რიცხვებისთვის გვაქვს:

$$\lceil\lceil x/a\rceil/b\rceil=\lceil x/ab\rceil,$$
$$\lfloor\lfloor x/a\rfloor/b\rfloor=\lfloor x/ab\rfloor.$$

დაწვრილებით იხ. [3].

## <<< ლოგარითმული ფუნქციის ერთი მნიშვნელოვანი გამოყენება

კომპიუტერულ მეცნიერებაში ტერმინი ლოგარითმი გულისხმობის პრინციპით (by default) დამაგრებულია ლოგარითმზე ორის ფუძით.

ლოგარითმის მთელი ნაწილი გვიჩვენებს, თუ რამდენჯერ შეიძლება მოხდეს ნატურალური რიცხვის განახევრება (მთელად გაყოფით ორზე), ვიდრე ეს განაყოფი ერთის ტოლი არ გახდება.

დავამტკიცოთ ეს ფაქტი.

ნებისმიერად ავიღოთ ნატურალური $n>1$ რიცხვი. არსებობს ნატურალური $k$ რიცხვი, ისეთი რომ

$$2^k\le n<2^{k+1}. \qquad (2)$$

ერთი მხრივ, (2) -იდან გვაქვს $k\le\log n<k+1$ ანუ $k=\lfloor\log n\rfloor$, ხოლო მეორე მხრივ

$$1\le\frac{n}{2^k}<2,\ \text{ანუ}\ 1=\left\lfloor\frac{n}{2^k}\right\rfloor=\left\lfloor\frac{n}{2^{\lfloor\log n\rfloor}}\right\rfloor.$$

რადგან $\left\lfloor\frac{n}{2^k}\right\rfloor=\lfloor\lfloor\lfloor\lfloor n/2\rfloor/2\rfloor/\cdots/2\rfloor/2\rfloor$ (მარჯვენა მხარეში $k$ -ჯერ ხდება მთელად გაყოფა ორზე), ამიტომ შეგვიძლია ვთქვათ, რომ ნებისმიერი ნატურალური $n$ რიცხვი უნდა (მთელად) განახევრდეს ზუსტად $\lfloor\log n\rfloor$ -ჯერ, რათა განაყოფში მივიღოთ ერთიანი.

## <<< ლიტერატურა


1. D. Musser, G. Derge, A. Saini. STL Tutorial and Reference Guide, Second Edition, Addison-Wesley, 2006.
2. N. Josuttis . The C++ Standard Library - A Tutorial and Reference, 2nd Edition, Addison-Wesley Professional 2012
3. T. Cormen, C. Leiserson, R. Rivest, C. Stein. Introduction to Algorithms, Third Edition. The MIT Press, 2009
4. S. Meyers. Effective Modern C++. O'reilly, 2014.