სარჩევი

1 დალაგებისა და ძებნის ალგორითმები და მათი შეფასება

წინა სემესტრის შესავალ კურსში ჩვენ განვიხილეთ „მიმართებები და დალაგება": თუ მოცემულ სიმრავლეზე განსაზღვრული იყო დალაგება (სრული, ანტისიმეტრიული და ტრანზიტული მიმართება), ამ სიმრავლეზე განსაზღვრული სიტყვებიც შეიძლებოდა გარკვეული წესით დაგველაგებინა. დალაგებულ სიმრავლეებზე ბევრი ამოცანის სწრაფად გადაჭრა შეიძლება, მათ შორის ძებნისაც: მილიონიანი ქალაქის ტელეფონის წიგნში მოცემული სახელისა და გვარის პიროვნების ნომრის პოვნა, როგორც წესი, სულ რამოდენიმე წამს გრძელდება, დაულაგებელ სიმრავლეში ძებნას კი ყველა თავს აარიდებდა. დალაგების სხვა გამოყენება სტატისტიკური მონაცემების გადამუშავებაშია: დავუშვათ, გვაინტერესებს, საქართველოში 25 - 30 წლის ასაკის რამდენ მოქალაქეს აქვს მიღებული განათლება თბილისის სახელმწიფო უნივერსიტეტში? ამისათვის საქართველოს მოქალაქეთა ბაზა უნდა დავახარისხოთ ჯერ ასაკის მიხედვით, შემდეგ კი - განათლების მიღების ინსტიტუტის მიხედვით. შედეგად მიღებული ორი ცხრილიდან ადვილად შეიძლება სტატისტიკის დადგენა.
თუ საჭიროა დიდ მონაცემთა ბაზაში გამეორებების აღმოფხვრა, აქაც შედეგს ბაზის დალაგებით ეფექტურად მივიღებთ.
დიდი ბაზის გადამუშავების ერთ-ერთი პირველი მაგალითია ამერიკის შეერთებული შტატების მოსახლეობის 1880 წლის აღწერა. 1500 ადამიანი შვიდი წლის განმავლობაში ალაგებდა ბაზას საჭირო მონაცემების მიხედვით. იმ დროისათვის უცნობმა ინჟინერმა ჰერმან ჰოლერითმა (Herman Hollerith) დაახლოებით ათი წელი მოანდომა დამახარისხებელი მანქანების შექმნას, რომლის საშუალებითაც 1890 წლის აღწერა (მეტი მოსახლეობითა და დასამუშავებელი მონაცემით) ხუთასმა თანამშრომელმა ორ წელიწადზე ნაკლებ დროში დაასრულა. ჰოლერითის ფირმა, რომელიც 1924 წლიდან International Business Machines (IBM) Corporation სახელითაა ცნობილი, შემდგომშიც დიდ როლს თამაშობდა მეცნიერებასა და ტექნიკაში. ყველაფერი კი დალაგების ალგორითმებითა და მათი იმპლემენტაციით დაიწყო.

როგორც ვიცით, ერთსა და იმავე სიმრავლეზე შესაძლებელია სხვადასხვა დალაგების განსაზღვრა (აქ და შემდგომში - თუ სხვაგვარად არ იქნა აღნიშნული - განვიხილავთ სრულ და რეფლექსურ დალაგებას $\leq$).

<u>სავარჯიშო 1.1:</u> $\mathbb{N}$ ნატურალურ რიცხვთა სიმრავლეზე მოიყვანეთ ხუთი სხვადასხვა დალაგების მაგალითი.

<u>სავარჯიშო 1.2:</u> მოცემულია ორი სიმრავლე A დალაგებით $\leq_A$ და B დალაგებით $\leq_B$. როგორ შეიძლება განვსაზღვროთ დალაგება $A \times B$ სიმრავლეზე?

<u>სავარჯიშო 1.3:</u> განსაზღვრეთ ისეთი სრული დალაგება $\leq$ კომპლექსურ რიცხვთა $\mathbb{C} = \mathbb{R} \times \mathbb{R}$ სიმრავლეზე, რომ $a \leq b \Rightarrow |a| \leq |b|$ (თუ $c = (c_1, c_2) \in \mathbb{C}, |c| = \sqrt{c_1^2 + c_2^2}$).

1.1 ძებნა და ჩასმა დალაგებულ მიმდევრობებში

განვიხილოთ შემდეგი ***ძებნის ამოცანა:*** მოცემულია დალაგებუი მიმდევრობა $A = (a_1, a_2, ..., a_n), a_1 \le a_2 \le ... \le a_n$ და ელემენტი c. დაადგინეთ, ჭეშმარიტია თუ არა $c \in A$. სხვა სიტყვებით რომ ვთქვათ, უნდა დავადგინოთ, არის თუ არა c ელემენტი A სიაში.
ცხადია, რომ ჩვენ შეგვიძლია A მიმდევრობის ყველა ელემენტის c რიცხვთან შედარება და თუ $\exists i, a_i = c$, პასუხი იქნება „კი", წინააღმდეგ შემთხვევაში - „არა".

<u>სავარჯიშო 1.4:</u> დაწერეთ რეკურსიული ალგორითმი, რომელიც ზემოთ აღწერილი მეთოდით დაადგენს, შედის თუ არა c ელემენტი A მიმდევრობაში. დაამტკიცეთ, რომ თუ $|A| = n$, ყველაზე ცუდ შემთხვევაში $O(n)$ ბიჯი იქნება საჭირო. რამდენი ბიჯი დაჭირდება ალგორითმს ყველაზე კარგ შემთხვევაში?

ამ მეთოდში ჩვენ არ ვითვალისწინებთ იმ ფაქტს, რომ A სიმრავლე დალაგებულია. არა და, დალაგებულ მიმდევრობაში, თუ ავიღებთ A სიმრავლის ნებისმიერ ელემენტს a_i და $c > a_i$, მაშინ ცხადია, რომ ძებნა მიმდევრობის a_i ელემენტიდან მარცხენა ნაწილში საჭირო არაა. ამ იდეაზეა აგებული შემდეგი ალგორითმი: A მიმდევრობის შუა ელემენტს ვუწოდოთ a (თუ შუა ელემენტი არ არსებობს, ვიღებთ სიის მარჯვენა ნახევრის მინიმალურ ელემენტს). თუ $a = c$, მაშინ ვადგენთ, რომ $c \in A$ და ალგორითმს ვასრულებთ. თუ $c < a$, მაშინ ძებნა A სიის მარცხენა ნახევარში უნდა გავაგრძელოთ, წინააღმდეგ შემთხვევაში - მარჯვენაში. ამ პროცედურას ვიმეორებთ მანამ, სანამ საძიებელი სიმრავლე ცარიელი არ იქნება (ანუ საძიებელი ელემენტი არ მოიძებნება).

<u>მაგალითი 1.1:</u>

საწყისი მონაცემი: $A = (-12, -8, 1, 2, 3, 4, 5, 8, 9, 11, 12, 13, 17)$, $c = 1$.

$$\begin{array}{llllll}
c < a? & 1 = 5?\ \text{არა};\ 1 < 5?\ \text{კი} & & 1 = 2?\ \text{არა};\ 1 < 2?\ \text{კი} & & 1 = -8?\ \text{არა};\ 1 < -8?\ \text{არა} \\
A & (\underbrace{-12, -8, 1, 2, 3, 4}, [5], 8, 9, 11, 12, 13, 17) & \rightarrow & (\underbrace{-12, -8, 1}, [2], 3, 4) & \rightarrow & (-12, [-8], \underbrace{1}) \\
 & & & & & \\
 & 1 = 1?\ \ \text{კი} & & & & \\
\rightarrow & ([1]) \rightarrow\ \text{პასუხი: არსებობს} & & & &
\end{array}$$

<u>მაგალითი 1.2:</u>

საწყისი მონაცემი: $A = (-12, -8, 1, 2, 3, 4, 5, 8, 9, 11, 12, 13, 17)$, $c = 15$.

$$\begin{array}{llllll}
c < a? & 15 = 5?\ \text{არა};\ 15 < 5?\ \text{არა} & & 15 = 12?\ \text{არა};\ 15 < 12?\ \text{არა} & & 15 = 17?\ \text{არა};\ 15 < 17?\ \text{კი} \\
A & (-12, -8, 1, 2, 3, 4, [5], \underbrace{8, 9, 11, 12, 13, 17}) & \rightarrow & (8, 9, 11, [12], \underbrace{13, 17}) & \rightarrow & (\underbrace{13}, [17]) \\
 & & & & & \\
 & 15 = 13?\ \ \text{არა} & & & & \\
\rightarrow & ([13]) & \rightarrow & () & \rightarrow & \text{პასუხი: არ არსებობს}
\end{array}$$

იტერაციულად ეს ალგორითმი შემდეგნაირად შეიძლება ჩაიწეროს:

მოცემულობა: რაციონალური რიცხვებისგან შემდგარი, დალაგებული სასრული მიმდევრობა A და რაციონალური რიცხვი c;
შედეგი: პასუხი შეკითხვაზე $c \in A$?

```
Find( A, c )
სანამ A მიმდევრობა ცარიელი არაა, გაიმეორე შემდეგი ოპერაციები:
{
  a = A მიმდევრობის შუა ელემენტი;
  if( a = c ) {return(„კი")}
     else if( a < c ) A = A სიმრავლის მარჯვენა ნახევარი;
            else A = A სიმრავლის მარცხენა ნახევარი
}
{return(„არა")}
```

სავარჯიშო 1.5: იგივე ამოცანა ჩაწერეთ რეკურსიულად და დაამტკიცეთ მისი სისწორე.

სავარჯიშო 1.6: დაამტკიცეთ, რომ ზემოთ მოყვანილი ალგორითმის დროის ზედა ზღვარია $O(\log n)$, სადაც $n = |A|$ (შედარებისა და სიმრავლის მარცხენა ან მარჯვენა ნაწილის აღების ოპერაციები თითო ბიჯად ჩათვალეთ). რა არის ძებნის ამოცანის დროის ქვედა ზღვარი?

იგივე პრინციპზეა აგებული დალაგებულ სიმრავლეში ***ელემენტის ჩამატების ამოცანა:*** მოცემულ დალაგებულ მიმდევრობაში A უნდა ჩავამატოთ ახალი ელემენტი c ისე, რომ მიღებული მიმდევრობაც დალაგებული იყოს. განსხვავება ისაა, რომ თუ პოვნის ამოცანაში ვეძებდით ელემენტს $a_i = c$, აქ ვეძებთ ორ ერთმანეთის მიყოლებით მყოფ ელემენტს a_i, a_{i+1} ისეთს, რომ $a_i \le c \le a_{i+1}$ და c ამ ორ ელემენტს შორის უნდა ჩავსვათ (ცხადია, თუ $c < a_1$ან $c > a_n$, ახალი ელემენტი სიის თავში ან შესაბამისად ბოლოში უნდა ჩაისვას). თვალსაჩინოებისათვის მოვიყვანოთ შემდეგი მაგალითი:

მაგალითი 1.3:

საწყისი მონაცემი: $A = (6, 8, 9, 11, 12, 13, 17)$, $c = 7$.

$$\begin{array}{llllll} a_{i-1} \le c \le a_i? & 7 \le 6?\ \text{არა;} & & 7 \le 8?\ \text{კი;} & & \\ A & ([6], 8, 9, 11, 12, 13, 17) & \rightarrow & (6, [8], 9, 11, 12, 13, 17) & \rightarrow & \text{პასუხი: } A = (6, [7], 8, 9, 11, 12, 13, 17). \end{array}$$

მაგალითი 1.4:

საწყისი მონაცემი: $A = (6, 8, 9, 11, 12, 13, 17)$, $c = 20$.

$$\begin{array}{llllllll} a_{i-1} \le c \le a_i? & 20 \le 6?\ \text{არა;} & & 20 \le 8?\ \text{არა;} & & 20 \le 9?\ \text{არა;} & \\ & ([6], 8, 9, 11, 12, 13, 17) & \rightarrow & (6, [8], 9, 11, 12, 13, 17) & \rightarrow & (6, 8, [9], 11, 12, 13, 17) & \rightarrow \\ a_{i-1} \le c \le a_i? & 20 \le 11?\ \text{არა;} & & 20 \le 12?\ \text{არა;} & & 20 \le 13?\ \text{არა;} & \\ & (6, 8, 9, [11], 12, 13, 17) & \rightarrow & (6, 8, 9, 11, [12], 13, 17) & \rightarrow & (6, 8, 9, 11, 12, [13], 17) & \rightarrow \\ a_{i-1} \le c \le a_i? & 20 \le 17?\ \text{არა;} & & & & & \\ & (6, 8, 9, 11, 12, 13, [17]) & \rightarrow & \text{პასუხი: } A = (6, 8, 9, 11, 12, 13, 17, [20]) & & & \end{array}$$

როგორც ვხედავთ, იმის მიხედვით, თუ როგორი მონაცემი გვექნება, ბიჯების რაოდენობა შეიძლება არ იყოს დამოკიდებული მონაცემთა სიგრძეზე (როგორც პირველ მაგალითში), ან იყოს დაახლოებით იმდენივე, რამდენიც მონაცემთა სიგრძეა (მეორე მაგალითი). აქედან გამომდინარე, ამ მეთოდით ჩასმის ბიჯების რაოდენობის ქვედა და ზედა ზღვარი იქნება $\Omega(1)$ და $O(n)$ (სხვა სიტყვებით რომ ვთქვათ, მონაცემებისგან დამოუკიდებლად, დაგვჭირდება მინიმუმ ორი ბიჯი - შედარება და ჩასმა - ან მაქსიმუმ $n+1$ ბიჯი - ყოველ ელემნტთან შედარება და ბოლოს ჩასმა).

თუ გავითვალისწინებთ იმ ფაქტს, რომ A მიმდევრობა დალაგებულია, მაშინ შეგვიძლია იგივე მეთოდი გამოვიყენოთ, როგორც ელემენტის ძებნის ამოცანაში: ვეძებთ ისეთ ელემენტს a_i, რომლისთვისაც სრულდება პირობა $a_{i-1} \le c \le a_i$ ან $a_i \le c \le a_{i+1}$. თუ $c \le a_1$ ან $c \ge a_n$, ახალი ელემენტი ემატება შესაბამისად თავში ან ბოლოში. თვალსაჩინოებისათვის განვიხილოთ შემდეგი მაგალითი:

საწყისი მონაცემი: $A = (6, 8, 9, 11, 12, 13, 17)$, $c = 20$.

$$\begin{array}{lllll} a_{i-1} \le c \le a_i? & 20 \le 11?\ \text{არა;} & & 20 \le 13?\ \text{არა;} & \\ & (\underbrace{6, 8, 9, [11], 12, 13, 17}) & \rightarrow & (6, 8, 9, 11, \underbrace{12, [13], 17}) & \rightarrow \\ & 20 \le 17?\ \text{არა;} & & & \\ & (6, 8, 9, 11, 12, 13, \underbrace{[17]}) & \rightarrow & \text{პასუხი: } A = (6, 8, 9, 11, 12, 13, 17, [20]) & \end{array}$$

ზოგადად ეს მეთოდი შემდეგი ალგორითმით შეიძლება ჩაიწეროს:

მოცემულობა: რაციონალური რიცხვებისგან შემდგარი, დალაგებული სასრული არაცარიელი მიმდევრობა $A = (a_1, a_2, ..., a_n)$ და რაციონალური რიცხვი c;

შედეგი: $A \cup \{c\}$ სიმრავლის ელემენტებისგან შემდგარი ***დალაგებული*** მიმდევრობა.

InsertFast($a_1, a_2, ..., a_n, c$ *)*
if($c \leq a_1$)
  $\{A = (c, a_1, a_2, ..., a_n)$; *return(A)* } /* თუ შესაძლებელია, პირველ ელემენტად ჩავსვათ */

if($c \geq a_n$)
  $\{A = (a_1, a_2, ..., a_n, c)$; *return(A)* } /* თუ შესაძლებელია, ბოლო ელემენტად ჩავსვათ */

$min = 1$;
$max = n$; /* დავადგინოთ საძიებელი ნაწილის საზღვრები */
do
{
$i = \lceil \frac{min+max}{2} \rceil$; /* საძიებელი ნაწილის შუა ელემენტის ინდექსი */

if($a_{i-1} \leq c \leq a_i$*)*
  $\{A = (a_1, ..., a_{i-1}, c, a_i, ..., a_n)$; *return(A)* }

if($a_i \leq c \leq a_{i+1}$) /* თუ შესაძლებელია, ახალი ელემენტი საჭირო ადგილზე ჩავსვათ */
  $\{A = (a_1, ..., a_i, c, a_{i+1}, ..., a_n)$; *return(A)* }

if($c < a_i$)
  $max = i$; /* საძიებელი ნაწილის ახალი საზღვრები: მარცხენა ნახევარი */
  else $min = i$; /* საძიებელი ნაწილის ახალი საზღვრები: მარჯვენა ნახევარი */
}

<u>სავარჯიშო 1.7:</u> დაამტკიცეთ, რომ ზემოთ მოყვანილი ალგორითმი ყოველთვის შეჩერდება.

<u>სავარჯიშო 1.8:</u> დაამტკიცეთ ზემოთ მოყვანილი ალგორითმის სისწორე.

<u>სავარჯიშო 1.9:</u> დაამტკიცეთ, რომ ზემოთ მოყვანილი ალგორითმის დროის ქვედა ზღვარია $\Omega(1)$. რაზეა დამოკიდებული მისი ზედა ზღვარი?

აღსანიშნავია, რომ ეს ალგორითმი პირველზე (მიყოლებით ძებნა და მერე ჩასმა) გაცილებით უფრო სწრაფი შეიძლება იყოს (მაგალითად, $2^{10} = 1024$ ელემენტიან სიაში ჩამატებას მიყოლებითი ალგორითმი დაახლოებით 2000 ბიჯს ***შეიძლება*** ანდომებდეს, ხოლო *InsertFast* ალგორითმი (მონაცემთა სათანადო სტრუქტურის შერჩევისას) დაახლოებით 70 ბიჯში ამთავრებდეს მუშაობას.
ეს იმას არ უნდა ნიშნავდეს, რომ მიმდევრობითი ალგორითმი ***ყოველთვის*** უფრო ნელი იქნება - გარკვეული მონაცემებისთვის იგი შეიძლება უფრო სწრაფიც იყოს, მაგრამ ყველაზე ცუდ შემთხვევაში *InsertFast* ალგორითმი გაცილებით უფრო სწრაფია.

<u>სავარჯიშო 1.10:</u> დაამტკიცეთ, რომ $A = (1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16)$ და $c = 3$ მონაცემისათვის მიმდევრობით ჩასმის ალგორითმი უფრო სწრაფია, ვიდრე *InsertFast*.

იმისათვის, რომ მიმდევრობაში ელემენტი ჩავამატოთ, საჭიროა ალგორითმი, რომელიც მონაცემთა სტრუქტურაზეა დამოკიდებული. თუ გვაქვს ელემენტების ვექტორი, საჭირო ადგილზე ჩასამატებლად მის მარჯვნივ მყოფი ელემენტები უნდა გადავანაცვლოთ თითო პოზიციით მარჯვნივ, რომ ადგილი „განთავისუფლდეს":

ცვლადი	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$
მნიშვნ.	1	2	4	5	6	

$\longrightarrow$

ცვლადი	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$
მნიშვნ.	1	2	4	5		6

$\longrightarrow$

ცვლადი	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$
მნიშვნ.	1	2	4		5	6

$\longrightarrow$

ცვლადი	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$
მნიშვნ.	1	2		4	5	6

$\longrightarrow$

ცვლადი	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$
მნიშვნ.	1		2	4	5	6

$\longrightarrow$

ცვლადი	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$
მნიშვნ.		1	2	4	5	6

ყველაზე ცუდ შემთხვევაში, თუ მოცემული გვაქვს n ელემენტიანი ვექტორი და გვჭირდება პირველ პოზიციაზე ჩამატება, მოგვიწევს n გადანაცვლების ოპერაციის ჩატარება. აქედან გამომდინარე, ვექტორში ჩამატების ოპერაცია ყველაზე კარგ შემთხვევაში $\Omega(1)$, ყველაზე ცუდ შემთხვევაში კი $O(n)$ ოპერაციის ჩატარებას მოითხოვს (მისი მუშაობის დრო წრფივია).

სავარჯიშო 1.11: ზემოთ აღწერილი მეთოდის დახმარებით დაწერეთ ალგორითმი *Insert(A,k,c)*, რომელიც $A = (a_1,a_2,...,a_n)$ მიმდევრობის k პოზიციაზე c ელემენტს ჩაამატებს და პასუხად $(a_1,...,a_{k-1},c,a_k,...,a_n)$ მიმდევრობას მოგვცემს.

სავარჯიშო 1.12: დაწერეთ ალგორითმი *InsertList(A,k,c)*, რომელიც A ბმული სიის k პოზიციაზე c ელემენტს ჩაამატებს.

1.2 დალაგების მარტივი ალგორითმები

დალაგება ამორჩევით: თუ მოცემულია დასალაგებელი მიმდევრობა A, შეგვიძლია შემდეგნაირად მოვიქცეთ:

SelectSort(მონაცემი: რაციონალური რიცხვებისგან შემდგარი სასრული მიმდევრობა A *)*

- სანამ A მიმდევრობა ცარიელი არაა, გაიმეორე შემდეგი ოპერაციები:

  A მიმდევრობაში იპოვნე მინიმალური ელემენტი, ამოშალე იქიდან და ჩართე B მიმდევრობაში

$$\begin{array}{lllllllllll} A & (5,1,13,-8,17) & \to & (5,1,13,17) & \to & (5,13,17) & \to & (13,17) & \to & (17) & \to & () \\ B & () & & (-8) & & (-8,1) & & (-8,1,5) & & (-8,1,5,13) & & (-8,1,5,13,17) \end{array}$$

სავარჯიშო 1.13: იგივე ალგორითმი ჩაწერეთ რეკურსიულად.

სავარჯიშო 1.14: დაამტკიცეთ, რომ ამ ალგორითმის დასრულების შემდეგ B მიმდევრობაში დალაგებული A სიმრავლე ჩაიწერება.

ზოგადად, ალგორითმის ბიჯების რაოდენობა დამოკიდებულია მისი ***მონაცემების სიგრძეზე***. ჩვენს მაგალითში კი მონაცემის სიგრძე დამოკიდებულია A მიმდევრობაში შემავალ ელემენტთა რაოდენობაზე და თითოეული რიცხვის სიგრძეზე. თუ $|A| = n$ და A მიმდევრობის თითოეული ელემენტი k ბიტისაგან შედგება, მონაცემთა სიგრძე იქნება $n \cdot k$.
ახლა კი გამოვითვალოთ, თუ რამდენ ბიჯს მოანდომებს *SelectSort* ალგორითმი მონაცემად მიღებული მიმდევრობის დალაგებას. ამისათვის უნდა გვქონდეს სიმრავლეში მინიმალური ელემენტის პოვნის ალგორითმი, რომელიც შემდეგნაირად შეიძლება ჩაიწეროს:

Min(მონაცემი: რაციონალური რიცხვებისგან შემდგარი სასრული მიმდევრობა $A = (a_1,a_2,...,a_n)$ *)*
 $min = a_1$;
 for($i = 2; i \leq n; i{+}{+}$ *)*
  {
  if($a_i < min$*)*
  $min = a_i$;
  }

სავარჯიშო 1.15: დაამტკიცეთ ზემოთ მოყვანილი ალგორითმის სისწორე.

Min ალგორითმის ბიჯების რაოდენობა შემდეგნაირად გამოითვლება:

min ცვლადისათვის A მიმდევრობის პირველი ელემენტის მინიჭება: 1 ბიჯი;
for ციკლი n-ჯერ მეორდება; მასში ხდება ერთი შედარება $a_i < min$ და თუ ეს ჭეშმარიტია, ერთი მინიჭება $min = a_i$.
აქედან გამომდინარე, ყველაზე ცუდ შემთხვევაში *for* ციკლში გვექნება თითო შედარება და თითო მინიჭება, სულ 2 ოპერაცია. ყველაზე კარგ შემთხვევაში კი მხოლოდ ერთი შემოწმება და არც ერთი მინიჭება.
ესე იგი, სულ ბიჯების რაოდენობა იქნება:

საუკეთესო შემთხვევაში: $n+1$;
უარეს შემთხვევაში: $2n+1$.

აღსანიშნავია, რომ ბიჯების გამოთვლის დროს ჩვენ არ გავითვალისწინეთ ციკლის მთვლელის შედარებისა და გაზრდის ოპერაციები, ციკლიდან გამოსვლის ოპერაცია და სხვა ტექნიკური დეტალები, რომლებიც ჩვეულებრივ ბიჯების გამოთვლისას არ ითვლება ხოლმე. სწორედ ასეთი დეტალების უგულებელყოფის მიზნითაა შემოღებული ზედა და ქვედა ზღვრის შეფასება O და Ω აღნიშვნით.

ამ ალგორითმის ქვედა და ზედა ზღვარი შეიძლება გამოისახოს როგორც $\Omega(n+1)=\Omega(n)$ და $O(2n+1)=O(n)$. აქედან გამომდინარე, შეიძლება შეფასდეს ზუსტი ზღვარი $\Theta(n)$.
ყოველივე თქმულიდან ვიღებთ: $T(Min)\in\Theta(n)$.
თუ ალგორითმის გამოთვლის დროის ზედა ზღვარია $O(n)$, მაშინ იტყვიან, რომ მისი დროის ზრდის რიგი არის წრფივი.

SelectSort ალგორითმის მსვლელობაში საჭიროა აგრეთვე ელემენტის ამოშლა. თუ A მიმდევრობა წარმოდგენილი იქნება როგორც ბმული სია, ამის მოხერხება 5 ბიჯში შეიძლება (იხ. წინა კურსის მასალაში ბმული სიების ოპერაციები).
აქედან გამომდინარე, A სიიდან x პოზიციაზე მდგომი ელემენტის ამოშლის $Erase(A,x)$ ალგორითმის მოქმედების დრო იქნება $T(Erase(A,x))\in\Theta(5)=\Theta(1)$.

ადვილი დასანახია, რომ *SelectSort* ალგორითმის მუშაობის დროს ჯერ უდა შემოწმდეს, ცარიელია თუ არა A სიმრავლე (ერთი ბიჯი), შემდეგ (თუ A ცარიელი არაა) მასში მოიძებნოს მინიმალური ელემენტი ($c_1\cdot n$ ბიჯი), ბოლოს ეს ელემენტი ამოიშალოს და ჩაიწეროს B მიმდევრობაში (ორივე ოპერაცია ჯამში c_2 ბიჯი):

$$T(SelectSort(A))=T(SelectSort(A-\{A \text{ სიმრავლის მინიმალური ელემენტი}\}))+1+c_1\cdot n+c_2,$$

თუ ჩავთვლით, რომ $|A|=n$. რეკურსიის გახსნის შედეგად (იმის გათვალისწინებით, რომ
$|A-\{A \text{ სიმრავლის მინიმალური ელემენტი}\}|=n-1$, მივიღებთ:

$$T(SelectSort(A))=T(SelectSort(\emptyset))+c_1(n+(n-1)+(n-2)+\cdots+1)+(c_2+1)n=1+c_1\frac{n(n+1)}{2}+(c_2+1)n\in O(n^2).$$

ამ შემთხვევაში იტყვიან, რომ *SelectSort* ალგორითმის დროის ზრდის ზედა ზღვარია (მუშაობის დროა) $O(n^2)$, ან მისი დროის ზრდის ზედა ზღვარი (მუშაობის დრო) კვადრატულია.

<u>სავარჯიშო 1.16:</u> გამოითვალეთ, რა იქნება პროგრამის დროის ზედა ზღვარი, თუ ბმული სიის ნაცვლად ავიღებთ ვექტორს და ელემენტის ამოშლის შემდეგ ცარიელი ადგილის „ამოვსება" (მარცხენა ან მარჯვენა ელემენტების თითო პოზიციით გადაწევა) დაგვჭირდება.

<u>სავარჯიშო 1.17:</u> დაწერეთ პროგრამა, რომელიც ამორჩევით დალაგების ალგორითმის მიხედვით A მიმდევრობას დაალაგებს ისე, რომ ***არ*** გამოიყენებს მეორე მიმდევრობას: ყოველ ჯერზე არჩეული მინიმალური ელემენტი ისევე A მიმდევრობაში ჩაწეროს. ამ პროგრამის დროის ზედა ზღვარიც კვადრატული უნდა იყოს.

<u>სავარჯიშო 1.18:</u> რა განსხვავება იქნება გამოთვლის დროში, თუ წინა სავარჯიშოში მოყვანილი ამოცანისათვის ალგორითმს ჯერ ბმული სიის, შემდეგ კი ვექტორის გამოყენებით დავწერთ? შეიცვლება თუ არა დროის ზრდის რიგი? შეიცვლება თუ არა რეალური გამოთვლის დრო?

1.2.1 დალაგება ჩადგმით:

მოცემული A მიმდევრობის დალაგება შემდეგი მეთოდითაც შეიძლება:

- სანამ A მიმდევრობა ცარიელი არაა, გაიმეორე შემდეგი ოპერაციები:

  აირჩიე A მიმდევრობის პირველი ელემენტი, ამოშალე იქიდან და ჩართე B მიმდევრობაში (რომელიც დალაგებულია) საჭირო ადგილზე ისე, რომ მიღებული მიმდევრობა დალაგებული იყოს.

ეს მეთოდი ფართოდ გამოიყენება პრაქტიკაში: მაგალითად, კარტის თამაშის დროს რიგ-რიგობით აღებულ კარტს „ვახარისხებთ" - ახალს უკვე დალაგებულ მიმდევრობაში საჭირო ადგილზე ვსვამთ.

A	$(5,13,8,1,7)$	$\rightarrow$	$(13,8,1,7)$	$\rightarrow$	$(8,1,7)$	$\rightarrow$	$(1,7)$	$\rightarrow$	(7)	$\rightarrow$	$()$
B	$()$		$([5])$		$(5,[13])$		$(5,[8],13)$		$([1],5,8,13)$		$(1,5,[7],8,13)$

აღსანიშნავია, რომ დალაგების ზემოთ აღნიშნული მეთოდები რაღაცა თვალსაზრისით ერთმანეთის „შებრუნებულია": პირველ მეთოდში ჯერ A მიმდევრობის საჭირო ადგილიდან ელემენტს ვარჩევთ და მერე მას B მიმდევრობის თავში ვსვამთ. მეორე მეთოდში კი ჯერ A მიმდევრობის თავიდან პირველ ელემენტს ვიღებთ და მას ვსვამთ B მიმდევრობაში საჭირო ადგილზე.

რადგან ჩასმით დალაგების ალგორითმში A სიმრავლიდან ვიღებთ ერთ ელემენტს და მას B სიმრავლეში ვსვამთ ისე, რომ ეს უკანასკნელი დალაგებული იყოს, შეგვიძლია გამოვიყენოთ *InsertFast* ალგორითმიც:

მოცემულობა: რაციონალური რიცხვებისგან შემდგარი სასრული მიმდევრობა $A = (a_1, a_2, ..., a_n)$;

შედეგი: A მიმდევრობის ელემენტებისგან შემდგარი დალაგებული მიმდევრობა B.

InsertionSort(A)
$B = \emptyset$;
while(A სიმრავლე ცარიელი არაა **)**
{
$a = A$ მიმდევრობის პირველი ელემენტი;
InsertFast(B, a) /* გამოვიყენოთ *InsertFast* ალგორითმი B სიმრავლეში ახალი ელემენტის ჩასამატებლად */
$A = A \backslash \{a\}$ /* A მიმდევრობიდან ამოვაგდეთ პირველი ელემენტი */
}

InsertionSort ალგორითმის ბიჯების რაოდენობა შემდეგნაირად შეიძლება დავითვალოთ:
while ციკლი n-ჯერ მოერდება; მასში კი შემდეგი ოპერაციები სრულდება:

- A მიმდევრობის პირველი ელემენტის გამოყოფა (1 ბიჯი);
- B მიმდევრობაში a ელემენტის საჭირო ადგილზე ჩამატება *InsertionSort* ალგორითმის გამოყენებით . $T(\mathit{InsertFast}(B,a)) = O(\log|B|) = O(\log n)$ ბიჯი;
- A მიმდევრობიდან პირველი ელემენტის ამოგდება (1 ბიჯი).

აქედან გამომდინარე, სულ გვექნება

$$T(InsertionSort(A)) \leq (c\log n+2)+(c\log(n-1)+2)+\cdots+(c\log 1+2) = 2n+c(\log n+\log(n-1)+\cdots+\log 2+\log 1)$$

და ვიღებთ:

$$T(InsertionSort(A)) \in O(\log n + \log(n-1) + \cdots + \log 2 + \log 1) = O(n \log n).$$

<u>სავარჯიშო 1.19:</u> დაამტკიცეთ ტოლობა $O(\log n + \log(n-1) + \cdots + \log 2 + \log 1) = O(n\log n)$.

<u>სავარჯიშო 1.20:</u> ზემოთ მოყვანილი ალგორითმის რომელი ნაწილები განსაზღვრავენ მუშაობის დროის ფუნქციის ზედა ზღვარს (სხვა სიტყვებით რომ ვთქვათ, რომელი ბიჯების უგულებელყოფა შეიძლება O აღნიშნვაში)?

<u>სავარჯიშო 1.21:</u> მონაცემთა რა სტრუქტურა უნდა ავირჩიოთ, რომ ალგორითმის ბიჯების ზედა ზღვარი იყოს $O(n\log n)$? რა შეიძლება მოხდეს სხვა სტრუქტურის არჩევის შემდეგ?

1.2.2 სწრაფი დალაგება

თუ მოცემულია დასალაგებელი მიმდევრობა $A = (a_n, ..., a_{n/2+1} a_{n/2}, , ..., a_1)$, დალაგების პროცედურის დაჩქარება შეიძლება მონაცემთა ორ ტოლ ნაწილად დაყოფით, მათი ცალ-ცალკე დალაგებით და დალაგებული ქვემიმდევრობების ერთმანეთში ისე შერწყმით, რომ მიღებული მიმდევრობა დალაგებული იყოს. ყოველივე ეს ერთ მაგალითზე განვიხილოთ:

მოცემულია დასალაგებელი მიმდევრობა $A = (3, 7, 1, 15, 12, 2, 13, 6)$. მისი მონაცემები დავყოთ ორ ტოლ ნაწილად: $A = (A_2, A_1)$, სადაც $A_2 = (3, 7, 1, 15)$ და $A_1 = (12, 2, 13, 6)$. თითოეული ქვემიმდევრობის დალაგების შედეგად ვიღებთ: $Sort(A_2) = (1, 3, 7, 15)$ და $Sort(A_1) = (2, 6, 12, 13)$. ცხადია, რომ A მიმდევრობის მინიმალური ელემენტი ან $Sort(A_1)$, ან $Sort(A_2)$ მიმდევრობის მინიმალური (მარცხენა) ელემენტი იქნება. თუ ამ ელემენტს შესაბამისი მიმდევრობიდან ამოვშლით, დარჩენილი მიმდევრობებიდან მინიმალური ელემენტი A მიმდევრობის მეორე ელემენტი იქნება. ამ პროცესს ვაგრძელებთ მანამ, სანამ ერთ-ერთი მიმდევრობა არ დაცარიელდება, რის შემდეგაც არაცარიელ მიმდევრობას პასუხს მივაწერთ:

A_1	$([1], 3, 14, 15)$	$(3, 14, 15)$	$([3], 14, 15)$	$(14, 15)$	$(14, 15)$	$(14, 15)$	$(14, 15)$
A_2	$(2, 6, 12, 13)$	$([2], 6, 12, 13)$	$(6, 12, 13)$	$([6], 12, 13)$	$([12], 13)$	$([13])$	$(\ )$
B	$(\ )$	(1)	$(1, 2)$	$(1, 2, 3)$	$(1, 2, 3, 6)$	$(1, 2, 3, 6, 12)$	$(1, 2, 3, 6, 12, 13)$

საბოლოო პასუხი: $(B, A_1) = (1, 2, 3, 6, 12, 13, 14, 15)$

ყოველივე ეს შემდეგი ალგორითმით შეიძლება ჩაიწეროს:

საწყისი მონაცემი: ორ (თითქმის) ტოლ ნაწილად დაყოფილი მიმდევრობა $A = (A_2, A_1) = (a_n, ..., a_{\frac{n}{2}+1}, a_{\frac{n}{2}}, ..., a_1)$, $a_i \le a_j$, $a_{i+\frac{n}{2}} \le a_{j+\frac{n}{2}}$, $1 \le i < j \le \frac{n}{2}$.

$MergeSort(A)$
if(A მიმდევრობა ერთ ელემენტიანია *)* $return(A)$ /* პირდაპირ ერთი ელემენტი დააბრუნე */
$B_1 = MergeSort(A_1)$;
$B_2 = MergeSort(A_2)$; /* დაალაგე მიმდევრობის ორივე ნაწილი */
$Merge(B_1, B_2)$; /* შეურიე დალაგებული ნახევრები ისე, რომ მიღებული მიმდევრობა დალაგებული იყოს */

ზემოთ მოყვანილ ფსევდო კოდში გამოყენებულია ქვეპროგრამა $Merge$, რომელიც შემდეგნაირად შეიძლება ჩაიწეროს:

საწყისი მონაცემი: ორი დალაგებული მიმდევრობა $A = (a_n, ..., a_1)$ და $B = (b_m, ..., b_1)$.

$Merge(A, B)$
$C = (\ )$;
do
{
  if(A მიმდევრობა ცარიელია *)* $return(\,(C, B)\,)$;
  if(B მიმდევრობა ცარიელია *)* $return(\,(C, A)\,)$;
  if(A მიმდევრობის მინიმალური ელემენტი < B მიმდევრობის მინიმალური ელემენტი *)*
    {
      $C = (\,C, A$ მიმდევრობის მინიმალური ელემენტი$)$;
      A მიმდევრობიდან ამოაგდე მინიმალური ელემენტი;
    }
  else
    {
      $C = (\,C, B$ მიმდევრობის მინიმალური ელემენტი$)$;
      B მიმდევრობიდან ამოაგდე მინიმალური ელემენტი;
    }
}

<u>სავარჯიშო 1.22:</u> ინდუქციის გამოყენებით დაამტკიცეთ მოყვანილი ალგორითმების სისწორე.

სავარჯიშო 1.23: დაამტკიცეთ, რომ თუ $|A|+|B|=n$, მაშინ $T(Merge(A,B)) \in O(n)$ და იყენებს არა უმეტეს $n-1$ ელემენტების შედარებას.

$MergeSort$ ალგორითმის დროის ზრდის რიგის შეფასება შემდეგნაირად შეიძლება:

თეორემა 1.1: $MergeSort$ ალგორითმი იყენებს მაქსიმუმ $\lceil n\log n\rceil$ შედარების ოპერაციას და მისი ბიჯების მაქსიმალური რაოდენობა ეკუთვნის $O(n\log n)$ სიმრავლეს.

დამტკიცება: თუ $MergeSort$ ალგორითმი n სიგრძის მიმდევრობას ალაგებს, მის მიერ გამოყენებულ შედარების ოპერაციათა მაქსიმალური რაოდენობა აღვნიშნოთ როგორც $C(n)$. რა თქმა უნდა, $C(1)=0$ და ალგორითმის ანალიზითა და $Merge$ ფუნქციის შედარების ოპერაციათა რაოდენობის გამოთვლით ვიღებთ:

$$C(n)=C(\lfloor n/2\rfloor)+C(\lceil n/2\rceil)+(n-1)=2C(\lceil n/2\rceil)+(n-1).$$

რეკურსიის გახსნის შედეგად ვიღებთ:

$$C(n)=2C(\lceil n/2\rceil)+(n-1)=2^{\lceil\log n\rceil}+(n+n/2+n/4+\cdots1)-\log n=n+n(1/2+1/4+\cdots+1/2^{\log n})-\log n\le\lceil n\log n\rceil.$$

სავარჯიშო 1.24: მათემატიკური ინდუქციის გამოყენებით დაამტკიცეთ, რომ თუ $n>1$,
$n+n(1/2+1/4+\cdots+1/2^{\log n})-\log n\le\lceil n\log n\rceil$.

სავარჯიშო 1.25: დაამტკიცეთ, რომ $MergeSort$ ალგორითმის დროის ზრდის რიგი იქნება $O(n\log n)$ (ჯერ დაამტკიცეთ, რომ ამ ალგორითმის მუშაობის დრო დიდად არ აღემატება შედარების ოპერაციათა რიცხვს და აქედან გამოიტანეთ დასკვნა).

სავარჯიშო 1.26: დაწერეთ პროგრამა, რომელიც $MergeSort$ ალგორითმის ბმულ სიებზე რეალიზაცია იქნება.

სავარჯიშო 1.27: დაწერეთ ალგორითმი, რომლის საშუალებითაც n ელემენტიან დალაგებულ მიმდევრობაში k ელემენტს $O(k\log k+n)$ დროში ჩავსვამთ.

ახლა კი განვიხილოთ მეთოდი, რომელშიც დალაგების „რთული ნაწილი" რეკურსიულ გამოძახებამდე ხდება:

მოცემულია დასალაგებელი მიმდევრობა $A=(a_n,...,a_1)$. თავიდან ვირჩევთ მიმდევრობის ერთ-ერთ (მაგალითად, პირველ) ელემენტს $a=a_1$ და გამოვყოფთ სამ ნაწილს: $C_1=\{a_i|a_i<a\}, C_2=\{a_i|a_i=a\}, C_3=\{a_i|a_i>a\}$. სიტყვებით რომ ვთქვათ, პირველი სიმრავლე შედგება A მიმდევრობის ყველა იმ ელემენტისაგან, რომელიც არჩეულ ელემენტზე ნაკლებია, მეორე - ისეთებისგან, რომელიც არჩეული ელემენტის ტოლია და მესამე - ისეთებისაგან, რომლებიც არჩეულ ელემენტზე მეტია. ცხადია, რომ თუ შემდგომ ეტაპზე პირველ და მესამე სიმრავლეს ცალ-ცალკე დავალაგებთ, მაშინ დალაგებული A სიმრავლე იქნება:

$$Sort(A)=(Sort(C_1),C_2,Sort(C_3)).$$

დალაგების ამ მეთოდს $QuickSort$ ეწოდება, რომელიც ფართოდ გამოიყენება პრაქტიკაში, იმის და მიუხედავად, რომ მისი მაქსიმალური ბიჯების ზრდის რიგი კვადრატული შეიძლება იყოს გარკვეული მონაცემებისათვის: $T(QuickSort(A))\in O(n^2)$, სადაც $|A|=n$.

საწყისი მონაცემი: $A=(a_n,...,a_1)$ რიცხვთა მიმდევრობა.

$Quicksort(A)$
if($|A|=1$*)* $return(A)$;
ნებისმიერი მეთოდით აირჩიე ერთი ელემენტი $a\in A$;
ააგე სიმრავლეები $C_1=\{a_i|a_i<a\}, C_2=\{a_i|a_i=a\}, C_3=\{a_i|a_i>a\}$;
$return((QuickSort(C_1),C_2,QuickSort(C_3))$;

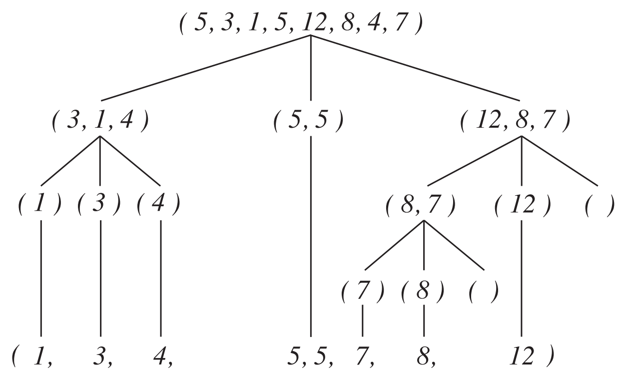
აღსანიშნავია, რომ საწყისი a ელემენტის არჩევა შეიძლება ნებისმიერი მეთოდით: ან შემთხვევით, ან ფიქსირებული (მაგალითად, პირველი, ან ბოლო, ან სხვა) ელემენტის, რაც მუდმივ დროში შეიძლება. ამას გარდა, C_1, C_2

და C_3 სიმრავლეების აგება წრფივ დროში შეიძლება, ისევე, როგორც საბოლოო პასუხის გამოტანა იმ პირობით, თუ ეს ქვესიმრავლეები დალაგებულია.
აქედან გამომდინარე, ვიღებთ ბიჯების რაოდენობის შეფასების შემდეგ რეკურსიულ ფორმულას:

$$T(QuickSort(A)) = O(1) + O(n) + T(QuickSort(C_1)) + T(QuickSort(C_3)).$$

სამაგიეროდ უმეტეს შემთხვევაში ეს ალგორითმი $O(n \log n)$ დროში ალაგებს მონაცემებს, ზუსტად კი მისი ბიჯე-ბის რაოდენობა უმეტეს შემთხვევაში გვექნება $T(Quicksort(A)) < 2n \lg n$, სადაც $|A| = n$.

მაგალითისათვის განვიხილოთ $A = (5, 3, 1, 5, 12, 8, 4, 7)$ (ნახ. 1). პირველ რიგში ვიღებთ პირველ ელემენტს $a = 5$ და ვაგებთ $C_1 = \{3, 1, 4\}$; $C_2 = \{5, 5\}$, $C_3 = \{12, 8, 7\}$ სიმრავლეებს. C_2 სიმრავლის ელემენტები პირდაპირ უნდა გავიდეს პასუხში, ხოლო C_1 და C_3 იგივე პრინციპით უნდა დაიყოს (ნახაზის მეორე სტრიქონი).
ერთ ელემენტიანი ქვესიმრავლეები პირდაპირ უნდა გამოვიტანოთ პასუხში, ხოლო სულ ცოტა ორ ელემენტიანი (როგორიცაა $\{8, 7\}$) იგივე პრინციპით უნდა დაიშალოს. აღსანიშნავია ის ფაქტი, რომ ცარიელი სიმრავლეების პასუხში არ უნდა გამოვიტანოთ.



ნახ. 1: *QuickSort* ალგორითმის გამოთვლის სქემა

<u>სავარჯიშო 1.28:</u> დაამტკიცეთ, რომ არსებობს ისეთი საწყისი მიმდევრობა A, რომლის დალაგებასაც *QuickSort* ალგორითმი კვადრატულ დროს მოანდომებს.

<u>სავარჯიშო 1.29:</u> დაწერეთ ალგორითმი, რომელიც ***მიმდევრობით*** მოსული $(a_1, a_2, ..., a_n)$ ელემენტების სიიდან k-ურ ელემენტს ამოარჩევს (მინიმალური ელემენტი პირველია, მეორე იქნება ის ელემენტი, რომელიც მინიმალურზე ნაკლები ან ტოლია და ა.შ.). ამ ალგორითმის მეხსიერების ხარჯვის ზედა ზღვარი უნდა იყოს $O(k)$ (მეხსიერების ხარჯვის ფუნქციის ზედა ზღვარი ბიჯების რაოდენობის ზედა ზღვრის ანალოგიურად გამოითვლება).

1.3 დალაგების ამოცანის ქვედა ზღვარი

აქამდე ალგორითმების ანალიზის დროს ჩვენ მათ ზედა და ქვედა ზღვარს ვითვლიდით. თუ რაიმე ალგორითმი გარკვეულ ამოცანას ჭრის (მაგ. მონაცემთა მიმდევრობის დალაგებას) მისი ზედა ზღვარი გვეუბნება იმას, თუ რამდენად სწრაფად შეიძლება ამ ამოცანის გადაჭრა. იმის დადგენა, თუ რამდენ დროს მოანდომებს ***ყველაზე სწრაფი ალგორითმი*** მოცემული ამოცანის გადაჭრას, საკმაოდ ძნელია და მას ამოცანის ქვედა ზღვრის დადგენა ეწოდება (არ აგერიოთ ***ალგორითმის*** ქვედა ზღვარში, რომელიც გვიჩვენებს, სულ ცოტა რამდენი ბიჯი ჭირდება ამ კონკრეტულ ალგორითმს ყველაზე კარგ შემთხვევაში). მოცემული ამოცანის ქვედა ზღვარი გვეუბნება, რომ ვერავინ დაწერს ისეთ ალგორითმს, რომლის მუშაობის მაქსიმალური დრო ამ ქვედა ზღვარზე სწრაფი იქნება. არაა გასაკვირი, რომ იმის დადგენა, რომ რაღაცის გაკეთებას ვერავინ შეძლებს, საკმაოდ რთული პროცესია და არ არსებობს ერთი მეთოდი, რომლითაც ამას ყველა ამოცანისათვის გავაკეთებდით: ამოცანათა სხვადასხვა ჯგუფს სხვადასხვანაირი მიდგომა ჭირდება.
აქ ჩვენ დავამტკიცებთ, რომ შედარების ოპერაციებზე დაფუძნებული ძებნის ამოცანის ქვედა ზღვარია $\Omega(n \log n)$. სხვა სიტყვებით რომ ვთქვათ, ვერავინ დაწერს ისეთ ალგორითმს, რომელიც შედარების ოპერაციებზე იქნება

დაფუძნებული (როგორებსაც ჩვენ აქამდე განვიხილავდით) და რომლის ბიჯების ზედა ზღვრის (მაქსიმალური რაოდენობის) ზრდის რიგი იქნება უკეთესი, ვიდრე $O(n \log n)$.

ამისათვის შემოვიღოთ შემდეგი

განმარტება 1.1: მოცემული $A = (a_1, a_2, ..., a_n)$ მიმდევრობის ***პერმუტაცია*** მისი ელემენტების გადანაცვლებას ეწოდება (ლათინური სიტყვიდან "permutare" - გაცვლა).

მაგალითად, $A = (a_1, a_2, a_3, a_4, a_5)$ მიმდევრობის ***ერთ-ერთი*** პერმუტაციის შედეგია $(a_5, a_1, a_2, a_4, a_3)$, ან $(a_1, a_2, a_3, a_5, a_4)$, ან $(a_3, a_2, a_5, a_1, a_4)$. თვით ეს მიმდევრობაც $(a_1, a_2, a_3, a_4, a_5)$ A მიმდევრობის პერმუტაციის შედეგია, რომელიც ყველა ელემენტს თავის ადგილზე ტოვებს.

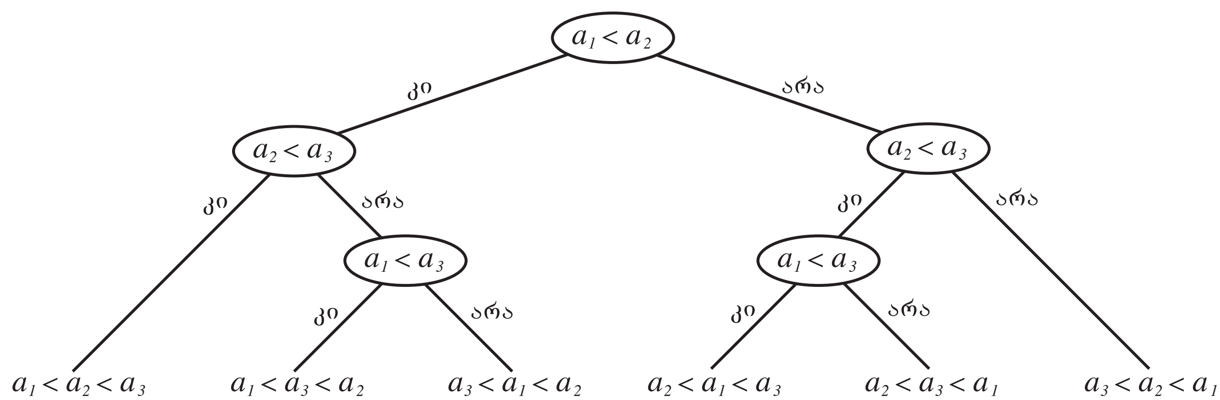
პერმუტაციათა აღწერის სხვადასხვა მეთოდი არსებობს, მაგრამ ხშირად მათ რიცხვთა მიმდევრობით გამოსახავენ ხოლმე, რომელიც გვიჩვენებს, თუ რომელ პოზიციაზე უნდა გადავიდეს საწყისი მიმდევრობის ესა თუ ის ელემენტი. მაგალითად, $\sigma = (2, 3, 5, 4, 1)$ პერმუტაციით A მიმდევრობა გადავა $(a_5, a_1, a_2, a_4, a_3)$ მიმდევრობაში: მისი პირველი ელემენტი გადავა მეორე ადგილზე, მეორე - მესამეზე, მესამე - მეხუთეზე, მეოთხე ისევე მეოთხეზე დარჩება და მეხუთე გადავა პირველ ადგილზე. ანალოგიურად, $\rho = (4, 2, 1, 5, 3)$ პერმუტაციით A მიმდევრობის ელემენტები გადავა $(a_3, a_2, a_5, a_1, a_4)$ მიმდევრობაში, ხოლო $(1, 2, 3, 4, 5)$ კი საწყის მიმდევრობას უცვლელს დატოვებს.

სავარჯიშო 1.30: მათემატიკური ინდუქციის გამოყენებით დაამტკიცეთ, რომ n ელემენტიანი მიმდევრობის $n!$ სხვადასხვა პერმუტაცია არსებობს.

სავარჯიშო 1.31: რომელი პერმუტაციებით მიიღება საწყისი (a, b, c, d, e, f, g, h) მიმდევრობიდან *(ა)* (a, d, h, b, c, f, e, g), *(ბ)* (d, a, h, b, g, f, e, e) მიმდევრობები?

სავარჯიშო 1.32: რა მიმდევრობებში გადაიყვანს საწყისი (a, b, c, d, e, f, g, h) მიმდევრობიდან *(ა)* $(1, 2, 4, 3, 6, 5, 7, 8)$, *(ბ)* $(8, 7, 6, 5, 4, 3, 2, 1)$ პერმუტაცია?

აღსანიშნავია, რომ დალაგებაც საწყისი მიმდევრობის პერმუტაციაა. ფაქტიურად, დალაგების ალგორითმის ამოცანაა, რაც შეიძლება სწრაფად გააანალიზოს შემოსული მონაცემი და შესაბამისი პერმუტაციით დალაგებულ მიმდევრობაში გადაიყვანოს.
ხშირად დალაგების ალგორითმისათვის მონაცემთა თანმიმდევრობის შესწავლის ერთად-ერთი საშუალება მხოლოდ მისი ელემენტების შედარებაა. მაგალითად, თუ სამ ელემენტიან მიმდევრობაში დავასკვნით, რომ $a_2 < a_1$ და $a_1 < a_3$, მაშინ დალაგების პერმუტაცია იქნება $(2, 1, 3)$ და დალაგებული სიმრავლე გამოვა (a_2, a_1, a_3). ამ შემთხვევაში იტყვიან, რომ დალაგების ეს ალგორითმი შედარების ოპერაციებზეა აგებული. აქამდე განხილული ყველა ალგორითმი ასეთი იყო.
ზემოთ თქმულიდან გამომდინარეობს, რომ ჩვენ შეგვიძლია პერმუტაციების გამოთვლის ხის აგება. იმავე სამ ელემენტიანი მიმდევრობის მაგალითზე შეგვიძლია ავაგოთ გამოთვლის ხე, რომელიც მოყვანილია ნახაზში 2.



ნახ. 2: სამ ელემენტიანი პერმუტაციის ხე

ასეთ სტრუქტურას „გადაწყვეტილების ორობითი ხე" ეწოდება. „ორობითი" იმიტომ, რომ ყოველ კვანძს (ფოთლების გარდა) ზუსტად ორი შვილი ყავს, ხოლო „გადაწყვეტილების" იმიტომ, რომ გამოთვლის დროს რაღაცა შეკითხვას პასუხი უნდა გავცეთ (გადაწყვეტილება მივიღოთ) და შემდეგ შესაბამის გზას გავყვეთ. რადგან ამ კონკრეტულ მაგალითში დასმულ შეკიტხვაზე ($a_i < a_j$?) ორი სხვადასხვა პასუხია შესაძლებელი, ეს სქემა ორობით ხეში კარგად ჯდება.

ამ პრინციპით ნებისმიერი $A = (a_1, ..., a_n)$ მიმდევრობის ორობითი გადაწყვეტილების ხის აგება შეიძლება, რომელსაც ფოთლებში A მიმდევრობის ყველა პერმუტაცია ექნება. ცხადია, რომ ნებისმიერ ალგორითმს, რომელიც შედარებებზეა აგებული, დალაგების დროს ასეთი ხის ზემოდან ქვემოთ გასვლა მოუწევს და პასუხი (დალაგება) ის შესაბამისი პერმუტაცია იქნება, რომელსაც ხის ფოთოლში მივაღწევთ.

მტკიცებათა სიმარტივისათვის დავუშვათ, რომ დასალაგებელი მიმდევრობის ყველა ელემენტი ერთმანეთისაგან განსხვავებულია (თუ ორი ან რამოდენიმე ელემენტი ერთმანეთის ტოლია, ამ შემთხვევისათვისაც შეიძლება ანალოგიური თეორემების დამტკიცება).

მნიშვნელოვანია შემდეგი

ლემა 1.1: თუ μ და σ ერთი და იგივე მიმდევრობის სხვადასხვა პერმუტაციაა, მაშინ შესაბამის ორობით გადაწყვეტილების ხეს ორი სხვადასხვა ფოთოლი $\ell_\mu \neq \ell\sigma$ ექნება, რომელიც ამ პერმუტაციებს შეესაბამება.

სავარჯიშო 1.33: საწინააღმდეგოს დაშვებით დაამტკიცეთ ზემოთ მოყვანილი ლემა.

ეს თითქოს და ელემენტარული ლება გადამწყვეტია დალაგების ალგორითმის ქვედა ზღვრის გამოთვლაში, რადგან აქედან გამომდინარეობს, რომ ყოველი გადაწყვეტილების ორობითი ხე, რომელიც n მონაცემს ზრდადობის მიხედვით ალაგებს, აუცილებლად უნდა შეიცავდეს $n!$ ფოთოლს.

რადგან T სიღრმის ორობით ხეს მაქსიმუმ 2^T ფოთოლი შეიძლება ქონდეს, ვიღებთ:

$$2^T \geq n! \text{ და, აქედან გამომდინარე, } T \geq \log n!.$$

სავარჯიშო 1.34: მათემატიკურ ინდუქციაზე დაყრდნობით დაამტკიცეთ, რომ T სიღრმის ორობით ხეს მაქსიმუმ 2^T ფოთოლი შეიძლება ქონდეს.

ე.წ. სტირლინგის ფორმულის თანახმად, რომელიც ფართოდ გამოიყენება კომბინატორიკაში და ჩვენ აქ დაუმტკიცებლად მივიღებთ, გვაქვს:

$$T \geq \underbrace{\log n! \geq \log\left(\frac{n}{e}\right)^n}_{\text{სტირლინგის ფორმულა}} = n\log n - n\log e,$$

სადაც e ე.წ. ნატურალური ლოგარითმის ფუძე (ან, როგორც მას სხვანაირადაც უწოდებენ ეილერის რიცხვია) - მუდმივა $e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = \sum_{n=0}^{\infty}\frac{1}{n!} = 2,718281828459045235...$

შენიშვნა: თავისი სრული სახით სტირლინგის ფორმულა შემდეგნაირად გამოისახება:

$$\log n! \sim \log\left(\frac{n}{e}\right)^n \cdot \sqrt{2\pi n},$$

რაც მარცხენა და მარჯვენა ნაწილში მოცემული ფუნქციების „მსგავსებას" ნიშნავს:

$$\lim_{n\to\infty}\frac{\log n!}{\log\left(\frac{n}{e}\right)^n \cdot \sqrt{2\pi n}} = 1.$$

ეს ფორმულა იმითიცაა საინტერესო, რომ მასში მეცნიერების ორი უმნიშვნელოვანესი მუდმივა - π და e ერთად ფიგურირებს.

აქედან გამომდინარე, ზემოთ მოყვანილი გადაწყვეტილების ორობითი ხე, რომელიც საჭირო პერმუტაციამდე მიგვიყვანს, ***დაახლოებით*** $n\log n$ სიღრმისაა და ჩვენ დავამტკიცეთ

თეორემა 1.2: შედარების ოპერაციებზე აგებული დახარისხების ალგორითმის ქვედა ზღვარია $\Omega(n \log n)$. უფრო ზუსტად მისი გამოთვლა შეიძლება ფორმულით $n \log n - O(n)$.

სავარჯიშო 1.35: აჩვენეთ, რომ შედარების ოპერაციებზე აგებული ნებისმიერი ალგორითმი, რომელიც დაულაგებელი n ელემენტიანი სიის მინიმალურ ელემენტს იპოვნის, სულ ცოტა $n-1$ შედარებას მოითხოვს.

სავარჯიშო 1.36: აჩვენეთ, რომ შედარების ოპერაციებზე აგებული ნებისმიერი ალგორითმი, რომელიც დაულაგებელი n ელემენტიანი სიიდან მინიმალურ და მის შემდგომ (ანუ ორ უმცირეს) ელემენტს იპოვნის, სულ ცოტა $n-1+\log n$ შედარებას მოითხოვს.
მოიყვანეთ ასეთი (ოპტიმალური) ალგორითმის მაგალითი.

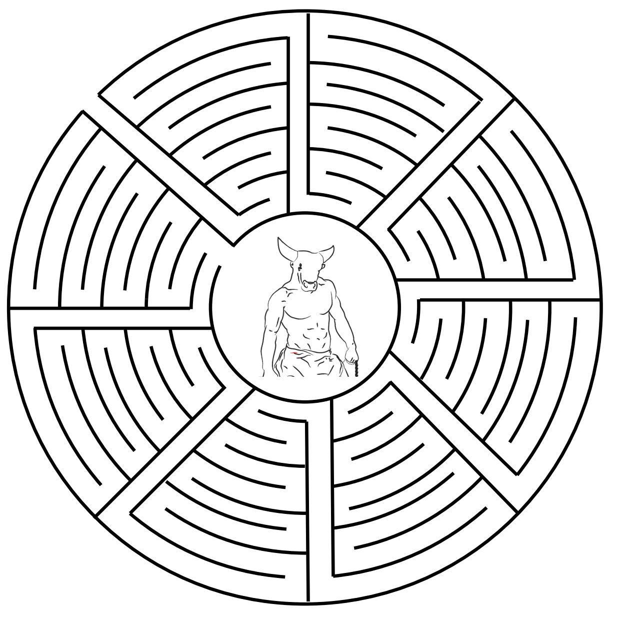
2 ალგორითმები გრაფებზე

როგორც წინა სემესტრის შესავალ კურსში აღვნიშნეთ, გრაფებით *ძალიან ბევრი* პრობლემის აღწერა და გადაჭრა შეიძლება. თუ მოცემულია რაიმე ამოცანა A, მისი მონაცემების გარდაქმნა შეიძლება ისეთ გრაფად, რომელზედაც რაღაცა სხვა ამოცანის ამოხსნით ამ საწყისი პრობლემის პასუხის დადგენა იქნება შესაძლებელი.
მაგალითად, მანქანებში ჩადგმული ნავიგაციის სისტემები, რომლელთა მეშვეობითაც ქალაქის ერთი ადგილიდან მეორეზე მისვლის უმოკლეს გზას ვგებულობთ, გრაფებზე ორ წვეროს შორის უმოკლესი გზის პოვნაზე დაიყვანება: თუ ქუჩების გადაკვეთას აღვნიშნავთ როგორც გრაფის წვეროებს, ხოლო წიბოებით კი თვითონ ქუჩებს, გამოგვივა შეწონილი გრაფი, რომელშიც ქუჩების სიგრძე წიბოს წონის ტოლი იქნება. ცხადია, რომ გრაფში უმოკლესი გზის პოვნა ქალაქში უმოკლესი გზის პოვნის ტოლფასი იქნება.
პირველ რიგში, აუცილებელია გადასაჭრელი ამოცანის მკაფიოდ და *სწორად* გადატანა გრაფებზე, რის შემდეგაც მის ამოსახსნელად რამოდენიმე ფუნდამენტური ალგორითმის ცოდნაა საჭირო, მათ შორის (მაგრამ არა მხოლოდ) უმცირესი დამფარავი ხის, მოცემული ორი წვეროს შორის უმოკლესი გზის, გრაფის პლანარულად (ბრტყლად) სიბრტყეზე დახაზვის ამოცანები. თუ ადამიანი რამოდენიმე ძირითადი ამოცანის გადაჭრის ხერხს დაეუფლება, უფრო რთული ამოცანების გადაჭრა, როგორც წესი, ამ ძირითადი ამოცანების თანმიმდევრულად გადაჭრის საშუალებითაც შეიძლება.
ამოცანათა გადაჭრის ძირითად მეთოდებს შორის (რომელიც ფართოდ გამოიყენება გრაფებზე ალგორითმებში) შეიძლება მოვიყვანოთ ე.წ. „სიგანეში ძებნის" და „სიღრმეში ძებნის" ალგორითმები, რომელთა საშუალებითაც სწრაფად შეგვიძლია შემოვიაროთ გრაფის ყველა წვერო (ყოველგვარი შეზღუდვის გარეშე).

ალგორითმებისა და მათი გადაჭრის მეთოდების შესწავლა უმჯობესია პრაქტიკული პრობლემების განხილვით დავიწყოთ.

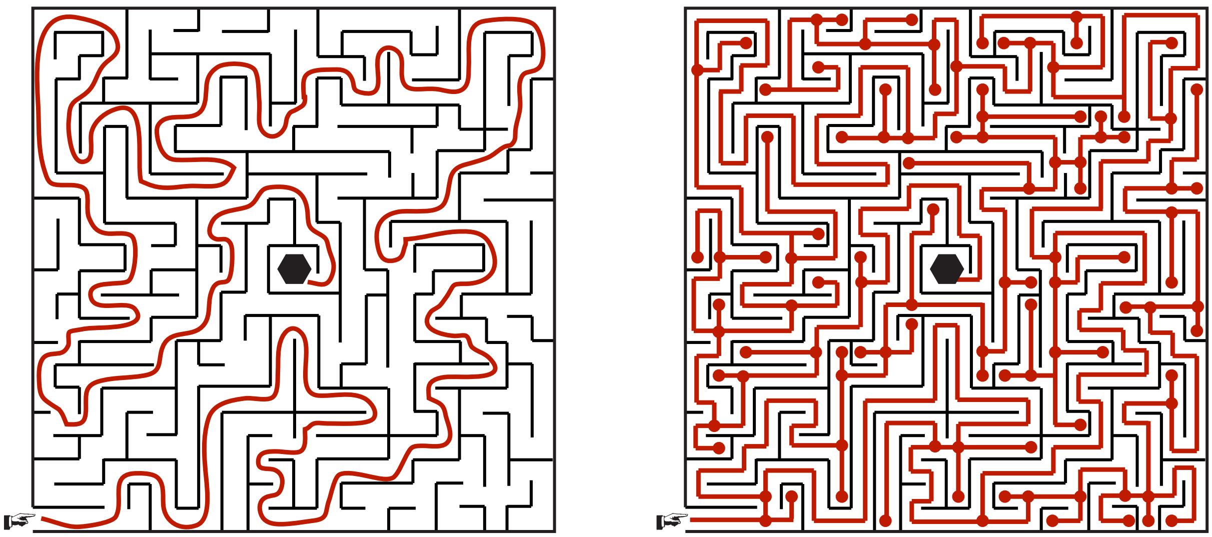
2.1 ბერძნული მითი მინოტავრის შესახებ

კუნძულ კრეტაზე ლაბირინთში დამწყვდეული იყო ადამიანის ტანისა და ხარის თავის მქონე ურჩხული მინოტავრი, რომლისთვის ათენელებს ხალხის მსხვერპლი უნდა შეეგზავნათ. ამ საშინელებისაგან თავის დახსნის მიზნით ბერძენი გმირი თესევსი ლაბირინთში შევიდა და მინოტავრი განგმირა. რადგან ლაბირინთში გზის გაკვლევა საკმაოდ რთული საქმეა, მან მეფის ქალიშვილის - არიადნას მიერ მიცემული ძაფი გამოიყენა, რითაც ადვილად გამოაგნო გარეთ (აქედან მომდინარეობს გამოთქმა „მსჯელობის ძაფი დაკარგა", „ძაფი გაექცა" და სხვა).



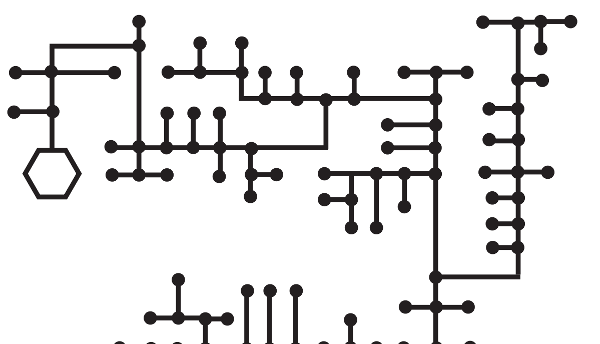
ნახ. 3: მსოფლიოში ყველაზე განთქმული ლაბირინთი

ეს მითი - ფილოსოფიური, ისტორიული, კულტურული და მრავალი სხვა მნიშვნელობის გარდა - ასევე დიდ როლს თამაშობს ინფორმატიკაშიც, რადგან უცნობ გარემოში მოძრაობის ამოცანას უკავშირდება, ხოლო ძაფის ან სხვა საშუალებებით განვლილი გზის მონიშნვა და უკან დაბრუნება ფართოდ გამოიყენება ალგორითმებთან დაკავშირებული პრობლემების გადასაჭრელად.
ყოველ ლაბირინთს შეგვიძლია შევუსაბამოთ რაღაცა გრაფი. ნახ. 4-ში ნაჩვენებია შედარებით რთული ლაბირინთი, რომელშიც შესაბამისი გრაფია ჩახაზული.



ნახ. 4: შედარებით რთული ლაბირინთი შესაბამისი გრაფით

იგივე გრაფი შეგვიძლია სხვანაირადაც დავხატოთ, რომ აღსაქმელად უფრო ადვილი იყოს (ნახ. 5). ცხადია, რომ თუ გვექნება ალგორითმი, რომლის საშუალებითაც გრაფის ყველა წვეროს შემოვლას შევძლებთ, ამით ლაბირინთში შესვლის ან გამოსვლის ალგორითმსაც ავაგებთ.



ნახ. 5: ლაბირინთის ექვივალენტური გრაფი

აღსანიშნავია, რომ ზედა ორ ნახაზში მოყვანილი გრაფი ერთმანეთის ექვივალენტურია (ერთის მეორეში გადაყვანა შეიძლება ისე, რომ გრაფის სტრუქტურა არ შეიცვალოს, აქ მხოლოდ დახატვის წესია სხვადასხვა), ამიტომ თუ ლაბირინთში შესვლისას გზის პირველ გასაყართან უნდა გავუხვიოთ მარცხნივ, მეორე გრაფში უნდა ვიაროთ პირდაპირ. მაგრამ ამას რაიმე პრინციპული მნიშვნელობა არ აქვს: ერთ გრაფზე მოძებნილი ამონახსნი ადვილად შეიძლება გადავიტანოთ მეორეზე.

ლაბირინთებში გზის გაკვლევის გარდა, გრაფში წვეროების შემოვლის ამოცანას მრავალი გამოყენება შეიძლება მოვუძებნოთ, მაგალითად, გრაფის წვეროების შიგთავსის ამობეჭდვაში, გრაფთა კოპირებასა ან სხვადასხვა ფორმატებში ჩაწერაში, წვეროთა ან წიბოთა რაოდენობის დათვლაში, გრაფის ბმული კომპონენტების პოვნაში, ორ წვეროს შორის გზის პოვნაში, გრაფში ციკლების აღმოჩენაში და ბევრ სხვა ამოცანაში.

თუ მოვახერხებთ იმას, რომ გრაფის შემოვლისას ყოველ წიბოზე გავივლით ***ზუსტად ორჯერ*** („იქით-აქეთ"), გვექნება იმის გარანტია, რომ არ გავიჭედებით და ყველა წვეროსაც გავივლით.

სავარჯიშო 2.1: ფორმალურად დაამტკიცეთ, რომ თუ გრაფის ყველა წიბოს შემოვუვლით ზუსტად ორჯერ, მის ყველა წვეროში ერთხელ მაინც შევალთ.

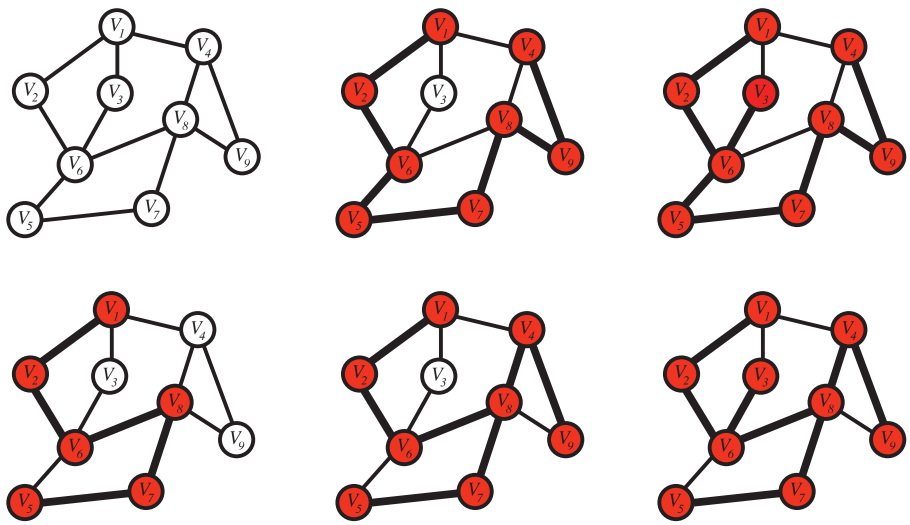
გრაფის შემოვლის ძირითადი პრინციპია წვეროებისა თუ წიბოების *სტატუსის აღნიშვა*: რაიმე მეთოდით იმის აღწერა, გავლილი გვაქვს თუ არა რაღაცა გზა და ღირს თუ არა მისი თავიდან გავლა. ზღაპრებსა და მითებში განვლილი გზების დასანიშნად ძაფებს, პურის ნამცეცებს, ხორბალს, ქვებსა და სხვა ყოველდღიურ ნივთებს იყენებენ, მაგრამ ამოცანისა და მისი ამოხსნის აღწერის დროს უმჯობესია მათემატიკური ობიექტებით ოპერირება. ამიტომ ყოველ წვეროს თითო ცვლადი შევუსაბამოთ, რომელიც მის „აქტუალურ მდგომარეობას" აღწერს. ეს მდგომარეობები შეიძლება იყოს:

- **აღმოუჩენელი** - თავდაპირველად ყველა წვერო „აღმოუჩენელ" მდგომარეობაშია, მასთან დაკავშირებული არც ერთი წიბო ჯერ გავლილი არაა;
- **აღმოჩენილი** - წვეროსთან მიერთებული გზები ნაწილობრივაა გავლილი, დარჩენილია შესასწავლი წიბოები;
- **შესწავლილი** - ამ წვეროსთან დაკავშირებული ყველა წიბო უკვე შემოვლილია.

ცხადია, რომ ყველა წვეროს სტატუსი თავიდან აღმოუჩენელია, შემდეგ ხდება ნაწილობრივ შესწავლილი და ბოლოს - შესწავლილი. ყველა წვეროს აქტუალური სტატუსის შესარჩევად საჭიროა შესაბამისი მონაცემთა სტრუქტურა. გრაფის შემოვლის დაწყებამდე ყველა წვერო აღმოუჩენელ მდგომარეობაშია, შემდეგ პირველი წვერო, რომლიდანაც ვიწყებთ შემოვლას, ხდება აღმოჩენილი. მასთან დაკავშირებული წიბოს გავლით გადავდივართ წვეროში, რომლის სტატუსსაც ვცვლით, ვხდით აღმოჩენილად და ვუმატებთ „დასამუშავებელ წვეროთა" სიას. თუ რომელიმე წიბოს შესწავლილ წვეროსთან მივყავართ, მას უგულებელვყოფთ, რადგან იქ მიბმული ქვეგრაფი სრულადაა შესწავლილი და ამით ახალ ინფორმაციას ვერ მივიღებთ.
ამ ძირითადი პრინციპების გათვალისწინებით შეგვიძლია გრაფის შემოვლის ალგორითმები ავაგოთ, რომლის ორ მაგალითს ახლა მოვიყვანთ.

2.1.1 სიღრმეში ძებნა

ლაბირინთში მოძრაობის ყველაზე ბუნებრივი მეთოდია სიარული გზის პირველ გასაყარამდე, ნებისმიერი მიმართულების არჩევა და შემდგომი სიარული შემდეგ გასაყარამდე, შემდეგ ისევე რაიმე მიმართულების არჩევა, სიარული და ა.შ. მანამ, სანამ ან ჩიხში არ მოვექცევით, ან არ მივალთ გზის ისეთ გასაყართან, რომლის ყველა მიმართულება გავლილი გვაქვს. თუ წინ მოძრაობის საშუალება აღარაა, გზას უკან გავივლით გზის უახლოეს გასაყარამდე და იგივე მეთოდს გავიმეორებთ. თუ ლაბირინთის შესასვლელთან დავბრუნდებით და იქიდან ყველა გზა გავლილი გვექნება, შეგვიძლია დარწმუნებით ვთქვათ, რომ მთელი ლაბირინთი შემოვლილია.
სხვა სიტყვებით რომ ვთქვათ, შესასვლელიდან უნდა ავირჩიოთ რაიმე გზა და მას რაც შეიძლება შორს მივყვეთ მანამ, სანამ ამის საშუალება გვაქვს (გამოვყოთ მაქსიმალური გზა). შემდეგ ამ გზას გავყვეთ უკან და პირველივე შესაძლებლობისას სხვა მაქსიმალური გზა ვიპოვნოთ. ამის მაგალითი ნაჩვენებია ნახაზში 6.



ნახ. 6: სიღრმეში ძებნის ორი მაგალითი

თუ შესასვლელს წვერო V_1 აღნიშნავს, მისგან *ნებისმიერ* წიბოს ვირჩევთ (რომელიც ჯერ არ გაგვივლია) და მის გასწვრივ ვაგრძელებთ მოძრაობას. ნახაზის ზემოთა ნაწილში ნაჩვენებია გზა ($V_1V_2V_6V_5V_7V_8V_9V_4$). რა თქმა უნდა, სრულიად შემთხვევით ჩვენ შეგვეძლო აგვერჩია აგრეთვე ნახაზის ქვედა ნაწილში მოყვანილი გზა ($V_1V_2V_6V_8V_7V_5$), რაც, ცხადია, შემოვლის ალტერნატიულ ვარიანტს მოგვცემს.
შემოვლის ზედა ვარიანტში V_4 წვეროდან ვეღარსად გადავალთ, რადგან ერთად-ერთი ვარიანტი V_1 იქნებოდა, რომელიც უკვე გავლილია. ამიტომ უკან უნდა დავბრუნდეთ მის წინა წვეროში (ამ შემთხვევაში V_9) და ვნახოთ, შეგვიძლია თუ არა იქიდან რაიმე გზის გავლა. რადგან ეს ასე არაა, ისევე ერთი წვეროთი უკან დაბრუნება გვიწევს, შემდეგ ისევ ერთით და ასე მანამ, სანამ არ მივალთ V_6 წვერომდე, რომლიდანაც კიდევ დარჩენილია გზა V_3-ში. რადგან აქედან გზის გაგრძელება შეუძლებელია, ისევე უკან ვბრუნდებით, ვამოწმებთ წინა წვეროს, ვრწმუნდებით, რომ წინ ვერ მივდივართ, ისევ ავდივართ ზემოთ და ამ პროცესს ვაგრძელებთ მანამ, სანამ ისევე საწყის წვერომდე არ მივალთ. რადგან გზის გაგრძელება იქიდანაც აღარაა შესაძლებელი, ალგორითმი დასრულებულია.
ნახაზის ქვემოთ მოყვანილ ვარიანტში, როდესაც საწყისი გზა იყო ($V_1V_2V_6V_8V_7V_5$), V_5 წვეროდან ვბრუნდებით V_7 წვეროში და, რადგან იქიდან ახალ წვეროს ვერ ვუერთდებით, V_8-ში. იქიდან უკვე არსებობს ახალი გზის გავლის ორი ვარიანტი. *შემთხვევით* ვირჩევთ ერთ-ერთს: V_4 და იქიდან V_9, თუმცა პირიქითაც შეგვეძლო. რადგან აქედან უკვე ახალი გზა აღარაა, ვბრუნდებით უკან პირველ წვერომდე, საიდანაც ახალი გზის პოვნაა შესაძლებელი. ამ შემთხვევაში ესაა V_6, საიდანაც გადავალთ წვეროში V_3. უკვე ნაცნობი პრინციპით ვბრუნდებით უკან V_1 წვერომდე და, რადგან იქიდან ახალი გზა არ არსებობს, ალგორითმს ვასრულებთ.

ამ მეთოდს „სიღრმეში ძებნას" უწოდებენ, რადგან მისი ძირითადი პრინციპია რაც შეიძლება გრძელი გზების გამოყოფა, ანუ „სიღრმეში ჩასვლა".

ცხადია, რომ სიღრმეში ძებნის დროს განვლილი წიბოებით ხე შეიქმნება (არ გვექნება ციკლები). რადგან ყველა წვერო გავლილი გვექნება, ეს იქნება მოცემული გრაფის „დამფარავი ხე", ანუ ისეთი, რომელიც ყველა წვეროს

მოიცავს (ფარავს).

<u>სავარჯიშო 2.2:</u> დაამტკიცეთ, რომ სიღრმეში ძებნის დროს ციკლები არ შეიქმნება.

<u>სავარჯიშო 2.3:</u> დაამტკიცეთ სიღრმეში ძებნის მეთოდის სისწორე (რომ შედეგად ყველა წვეროს ერთხელ მაინც გავივლით).

ამ თავის დასაწყისში აღწერილი პრინციპების თანახმად, სიღრმეში ძებნის კონკრეტული ალგორითმი უნდა ადგენდეს „დასამუშავებელი წვეროების" სიას და მის მიხედვით მოქმედებდეს. ჩვენს მაგალითებში ეს პროცედურა შემდეგნაირი იქნება:

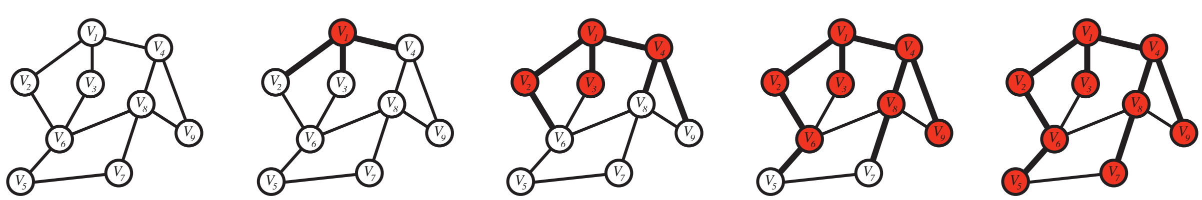
საწყისი მოცემულობა: $R = (\quad)$ ცარიელი სია (ამ სიაში წვეროები იქნება ჩამოწერილი იმ თანმიმდევრობით, რომლითაც შემოვივლით გრაფს);
გრაფი G წვეროთა სიმრავლით $E(G)$ და წიბოთა სიმრავლით $V(G)$ და ამ გრაფის რომელიმე წვერო u (გრაფის ყველა წვერო თავდაპირველად მონიშნულია როგორც „აღმოუჩენელი").

DFS(G,u)
მონიშნე u წვერო როგორც „აღმოჩენილი";
მიუმატე u წვერო R სიას; /* ახალი წვერო მიემატა გზის სიას */
for$(\ \forall v,\ (u,v) \in E(G)\ )$ /* განვიხილავთ u წვეროსთან მიბმულ ყველა წვეროს */
{
if(v წვერო „აღმოუჩენელია")
{
DFS(G,v)
}
მონიშნე u წვერო როგორც „შესწავლილი"; /* რადგან *for* ციკლიდან გამოვედით, u წვეროს ყველა წიბო გავიარეთ */ }

<u>სავარჯიშო 2.4:</u> დაამტკიცეთ ამ ალგორითმის სისწორე და შეაფასეთ მისი ბიჯების რაოდენობის ზედა ზღვარი.

2.1.2 სიგანეში ძებნა

ბუნებრივი მოვლენების, კერძოდ კი ტალღების გავრცელების პრინციპზეა აგებული ე.წ. „სიგანეში ძებნის" (Breadth-First Search, BFS) მეთოდი. წყალში ჩაგდებული ქვის მიერ გამოწვეული ტალღები, როგორც ვიცით, ცენტრიდან ვრცელდება, პირველ რიგში უშუალო სიახლოვეში მყოფ არეს მოიცავს და შემდეგ გადადის უფრო შორეულ სივრცეზე (სიგანეში თანაბრად ვრცელდება). ჩვენს მიერ ნახსენებ მეთოდშიც საწყისი წვეროდან ჯერ *ყველა* მის უშუალო *აღმოუჩენელ* მეზობელს ჩავინიშნავთ, შემდეგ მათზე რიგ-რიგობით გადავალთ და იგივე პროცედურას გავიმეორებთ.



ნახ. 7: სიგანეში ძებნის სქემა

თავიდან საწყის წვეროს ვწერთ „დასამუშავებელ სიაში", შემდეგ ვინიშნავთ მის ყველა *აღმოუჩენელ* უშუალო მეზობელს და საწყის წვეროს სიიდან ვშლით (სამაგიეროდ ვინიშნავთ შესწავლილ წვეროთა სიაში, რომელშიც ბოლოს თანმიმდევრულად ყველა ის წვერო იქნება ჩანიშნული, რომელიც გავიარეთ). რა თქმა უნდა, ყოველი

შესწავლილი წვეროსათვის უნდა ჩავინიშნოთ ასევე, თუ რომელი წვეროდან გადმოვედით მასზე, რომ მერე უკან წასვლა შევძლოთ.
ამ პროცესს ვიმეორებთ მანამ, სანამ გრაფის ყველა წვერო არ იქნება შესწავლილი.

სავარჯიშო 2.5: დაწერეთ სიგანეში ძებნის ალგორითმი, დაამტკიცეთ მისი სისწორე და გამოითვალეთ ბიჯების რაოდენობის ზედა ზღვარი.

2.1.3 გრაფის შემოვლის ალგორითმების გამოყენება

ბმული კომპონენტები

როგორც ვიცით, არაა აუცილებელი, რომ გრაფში ყველა ნაწილი ბმული იყოს, ანუ შეიძლება არსებობდეს ორი წვერო, რომელთა შორის შემაერთებელი გზა არ მოიძებნება. ასეთი ცალკეული კომპონენტების პოვნა ფუნდამენტური ამოცანაა გრაფთა თეორიაში: როდესაც რაიმე ამოცანა დაყოფილ გრაფებზე უნდა ამოიხსნას, უმეტეს შემთხვევაში უნდა დავადგინოთ დამოუკიდებელი ნაწილები და ისინი ცალ-ცალკე დავამუშაოთ.
მაგალითად, თუ მოცემულია რაიმე სიმრავლე, მასზე მოცემული ექვივალენტობის მიმართება, როგორც ვიცით, ამ სიმრავლეს ექვივალენტურობის კლასებად ყოფს და თუ ამ მიმართებას გრაფის სახით გამოვხატავთ, თითო კლასი ამ გრაფის ბმულობის კომპონენტი იქნება.

როგორც სიღრმეში, ასევე სიგანეში ძებნის ალგორითმების გამოყენებით ადვილად შეიძლება გრაფის ბმულობის კომპონენტების გამოყოფა: ავირჩევთ ნებისმიერ წვეროს და ამ რომელიმე ალგორითმით შემოვივლით დანარჩენ შესაძლო წვეროებს, რომლებსაც ერთი და იგივე ნიშანს დავადებთ (მაგალითად, მთვლელის რიცხვი). თუ გრაფში სხვა წვეროებიც დარჩა, მთვლელს ერთით გავზრდით და იგივე პროცედურას გავიმეორებთ მანამ, სანამ გრაფის ყველა წვეროს რაიმე ნიშანი არ დაედება.

სავარჯიშო 2.6: დაწერეთ ბმულობის კომპონენტების გამოყოფის ალგორითმი, დაამტკიცეთ მისი სისწორე და გამოითვალეთ ბიჯების ზედა ზღვარი.

ხეებისა და ციკლების პოვნა

ხეები - აციკლური (უციკლო) გრაფები - საინტერესო გრაფთა ყველაზე მარტივ კლასს ქმნიან. სიღრმეში ძებნის მეთოდი პირდაპირ გვაძლევს იმის პასუხს, არის თუ არა მოცემული გრაფი ხე: თუ ძებნის პროცესში განვლილ წიბოს აღვნიშნავთ როგორც „ხის წიბოს", ხოლო ისეთ წიბოს, რომელსაც არ გადავივლით (მაგალითად ისეთს, რომელსაც ერთი წვეროდან უკვე შესწავლილ წვეროში მივყავართ) აღვნიშნავთ, როგორც „დამატებითს", მოცემული გრაფი იქნება ხე მაშინ და მხოლოდ მაშინ, თუ სიღრმეში ძებნის შემდეგ დამატებითი წიბოები არ აქვს. რადგან ნებისმიერი ხისათვის $|E| = |V| - 1$, ამ ალგორითმის დროის ზედა ზღვარი იქნება $O(|V|)$.
თუ გრაფი შეიცავს ციკლს, მისი აღმოჩენა შეიძლება პირველივე დამატებითი წიბოს პოვნით: თუ აღმოჩნდა დამატებითი წიბო (u, v), მაშინ უკვე შექმნილ ხეში უნდა არსებობდეს გზა u წვეროდან v წვეროში და იგი (u, v) წიბოსთან ერთად მოგვცემს ციკლს.

სავარჯიშო 2.7: დაამტკიცეთ, რომ ნებისმიერი დამატებითი (u, v) წიბოს წვერეობს შორის არსებობს შემაერთებელი გზა.

სავარჯიშო 2.8: სიღრმეში ძებნის გამოყენებით დაწერეთ ალგორითმი, რომელიც მოცემული გრაფისათვის დაადგენს, არის თუ არა იგი ხე და თუ არა, ციკლებსაც იპოვნის.

სავარჯიშო 2.9: განიხილეთ წინა ამოცანაში დაწერილი ალგორითმი. შეიძლება თუ არა მისი საშუალებით ყველა ციკლის აღმოჩენა?

სავარჯიშო 2.10: შეიძლება თუ არა იგივე ამოცანის სიგანეში ძებნის მეთოდით გადაჭრა?

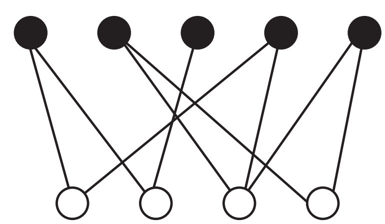
ორად შეღებილი გრაფები

გრაფის შეღებვის ზოგად ამოცანაში გრაფის წვეროები ისე უნდა შეიღებოს, რომ წიბოთი შეღებილ ორ წვეროს სხვადასხვა ფერი ქონდეს. ცხადია, თუ ყველა წვეროს სხვადასხვა ფრად შევღებავთ, ეს ამოცანა გადაიჭრება,

მაგრამ საინტერესოა გრაფის რაც შეიძლება ცოტა ფრად შეღებვის ამოცანა - მოკლედ: გრაფის შეღებვის ამოცანა - რომელიც ფართოდ გამოიყენება ალგორითმების თეორიასა თუ პრაქტიკაში:

- ეფექტური ცხრილების შედგენაში - მაგალითად, მრავალბირთვიან პროცესორებში ამოცანის ნაწილების პარალელურად დამუშავების მიზნით - რა ნაწილი რის შემდეგ უნდა დამუშავდეს;
- რადიოტალღების სიხშირეების ეფექტურ დადგენაში - მაგალითად, თუ ორი მომხმარებელი რადიოგადამცემს ხმარობს, ახლოს მდგომებს სხვადასხვა სიხშირეები უნდა ქონდეთ, შორს მდგომებს შეიძლება ერთი და იგივე;
- რეგისტრების ეფექტურად დანაწილების ამოცანა - პროცესორის რეგისტრების გამოყენებით მონაცემების დამუშავება გაცილებით უფრო სწრაფად შეიძლება, ვიდრე RAM მეხსიერებიდან. მაგრამ რადგან ხშირად ცვლადთა რაოდენობა რეგისტრების რაოდენობას აჭარბებს, საჭიროა იმის დადგენა, რა დროს რომელი ცვლადი ჩაიწეროს რეგისტრებში;
- სახეთა ამოცნობაში - მაგალითად, მოცემული სურათით კატალოგში ადამიანის მოძებნა;
- არქეოლოგიური ან ბიოლოგიური მასალის ანალიზი - როგორც ბიოლოგიაში, ასევე არქეოლოგიაში მონაცემები ხის სახით შეგვიძლია შევინახოთ: ერთი სახეობა ან კულტურა მეორედან მომდინარეობს, ერთი სახეობა ან კულტურა სხვა რამოდენიმეს წარმოშობს.

ზოგადად, გრაფის მინიმალურად შეღებვის ამოცანა (ან მისი ***ქრომატული რიცხვის*** დადგენის ამოცანა, როგორც მას უწოდებენ ხოლმე), ძნელი გადასაჭრელია. მისი ერთ-ერთი მნიშვნელოვანი ქვეამოცანაა, შეიძლება თუ არა მოცემული გრაფის ორ ფრად შეღებვა, ან, სხვა სიტყვებით რომ ვთქვათ, ისეთ ორ ნაწილად დაყოფა, რომელშიც წვეროები ერთმანეთისგან იზოლირებულნი არიან. ასეთი გრაფის მაგალითია მოყვანილი ნახაზში 8.



ნახ. 8: ორად შეღებილი გრაფის მაგალითი

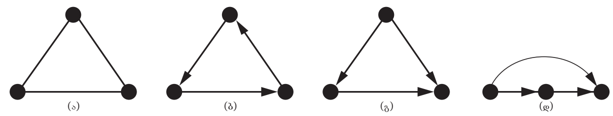
ორად შეღებილ გრაფს ზოგჯერ *ორად დაყოფილსაც* უწოდებენ. იმის დასადგენად, შეიძლება თუ არა მოცემული გრაფის ორად დაყოფა, შემდეგი სტრატეგიით შეიძლება მოქმედება:
ვირჩევთ ერთ-ერთ წვეროს, რომელსაც ვღებავთ რომელიმე ფრად (დავუშვათ, თეთრად). სიღრმეში ან სიგანეში ძებნით ახლად აღმოჩენილ წვეროს ვღებავთ მისი მშობლის (იმ წვეროსი, რომელთანაც არის დაკავშირებული) განსხვავებული ფერით (თეთრი ან შავი). შემდეგ *ყოველი აღმოუჩენელი* წვეროსათვის ვამოწმებთ, არის თუ არა იგი მიერთებული ორ ერთსა და იმავე ფრად შეღებილ წვეროსთან და თუ ასეა, ეს იმას უნდა ნიშნავდეს, რომ გრაფი არ შეიღებება (არ დაიყოფა) ორად. თუ ალგორითმი ამ სახის კონფლიქტის გარეშე დასრულდება, ეს იმას უნდა ნიშნავდეს, რომ გრაფი ორად შეიღება.

<u>სავარჯიშო 2.11:</u> დაამტკიცეთ ამ მეთოდის სისწორე. დამოკიდებულია თუ არა ეს მეთოდი იმაზე, თუ რომელ საწყის წვეროს ავიღებთ?

<u>სავარჯიშო 2.12:</u> ამ მეთოდზე დაყრდნობით დაწერეთ ალგორითმი, რომელიც დაადგენს, შეიძლება თუ არა გრაფის ორად შეღებვა.

ტოპოლოგიური დალაგება

განვიხილოთ ნახ. 9-ში მოყვანილი სამი გრაფის მაგალითი.



ნახ. 9: არამიმართული ციკლური, მიმართული ციკლური და მიმართული აციკლური გრაფები

პირველი გრაფი არაა მიმართული და შეიცავს ციკლს; მეორე მიმართულია და ციკლს შეიცავს, ხოლო მესამე კი მიმართულია, მაგრამ ციკლს არ შეიცავს (აციკლურია), რადგან ვერ მოვძებნით ისეთ *დაშვებულ* გზას, რომელიც გაუყვება ისრებს და რომელიმე წვეროდან ისევ იგივე წვეროში დაგვაბრუნებდა.
ასეთ სტრუქტურას *აციკლური მიმართული გრაფი* ეწოდება და მათი დახაზვა შეიძლება ისე, რომ ყველა წიბო *მარცხნიდან მარჯვნივ* იყოს მიმართული (მაგალითად, ნახ. 9(დ)), რასაც ამ გრაფის ***ტოპოლოგიურ დალაგებას*** უწოდებენ.

აღსანიშნავია, რომ მხოლოდ აციკლურ მიმართულ გრაფებს შეიძლება მოვუძებნოთ ტოპოლოგიური დალაგება, რადგან ნებისმიერი ციკლი რომელიმე კვანძიდან *უკან* (ანუ მარჯვნიდან მარცხნივ) დაგვაბრუნებდა, თანაც აციკლურ მიმართულ გრაფებს *ყოველთვის* მოვუძებნით ერთ ტოპოლოგიურ დალაგებას მაინც.

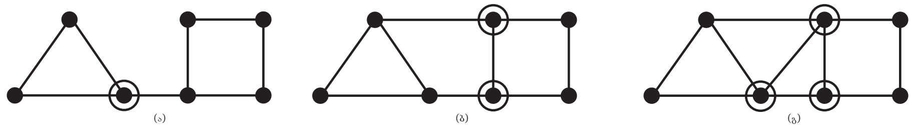
ტოპოლოგიურ დალაგებას დიდი მნიშვნელობა აქვს მთელ რიგ პრაქტიკულ ამოცანებში, მაგალითად ზემოთ ნახსენებ ცხრილის შედგენაში.

შიღრმეში ძებნის ალგორითმით ადვილად შეგვიძლია მიმართული გრაფის აციკლურობის დადგენა (რაც დამოკიდებულია იმაზე, შეგვხვდება თუ არა მუშაობის პროცესში ზემოთ ნახსენები „დამატებითი წიბოები”). თუ გრაფი აციკლურია, მაშინ ალგორითმის მსვლელობისას შექმნილი R მიმდევრობა ტოპოლოგიური დალაგების თანმიმდევრობას გვაძლევს.

სავარჯიშო 2.13: დაამტკიცეთ, რომ სიღრმეში ძებნის პროცესში შექმნილი R მიმდევრობა ტოპოლოგიური დალაგების თანმიმდევრობას გვაძლევს.

საარტიკულაციო კვანძები

ხშირად საჭიროა იმის დადგენა, თუ მინიმუმ რამდენი კვანზის ამოგდებაა საჭირო ბმული გრაფიდან იმისათვის, რომ იგი ნაწილებად დაიშალოს. ქვემოთ მოყვანილ ნახაზში ნაჩვენებია სამი გრაფი, რომელთა დაშლა *მინიმუმ* ერთი (ა), ორი (ბ) და სამი (გ) კვანძის ამოგდებით შეიძლება. ამ შემთხვევაში იტყვიან, რომ გრაფია *ერთად* ბმული, ან *ორად* ბმული, *სამად* ბმული და, ზოგადად, n-ად ბმული.
იტყვიან ასევე, რომ გრაფის ბმულობის კოეფიციენტია n.



ნახ. 10: ერთად, ორად და სამად ბმული გრაფი

თუ გრაფის ბმულობის კოეფიციენტია 1, მაშინ იტყვიან, რომ მას აქვს ე.წ. ***საარტიკულაციო კვანძი***.

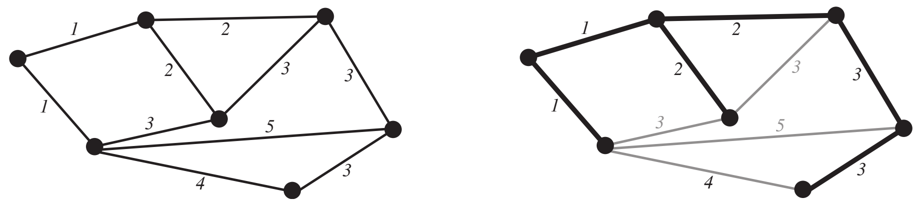
საარტიკულაციო კვანძების ძებნა ძალიან მნიშვნელოვანია მაგალითად საკომუნიკაციო ქსელების სტაბილურობის დადგენაში. ზოგადად, რაც უფრო მაღალია ქსელის შესაბამისი გრაფის ბმულობის კოეფიციენტი, მით უფრო სტაბილურია იგი.
ადვილი შესამჩნევია, რომ საარტიკულაციო კვანძების ძებნა გრაფიდან რიგ-რიგობით წვეროების ამოგდებითა და დარჩენილი სტრუქტურის ბმულობაზე შემოწმებით შეიძლება.

სავარჯიშო 2.14: დაწერეთ ალგორითმი, რომლითაც გრაფის საარტიკულაციო კვანძების არსებობას დავადგენთ. დაამტკიცეთ მისი სისწორე და დაითვალეთ ბიჯების ზედა ზღვარი.

2.2 უმცირესი დამფარავი ხე

თუ მოცემულია შეწონილი გრაფი, ძალიან მნიშვნელოვანია ე.წ. „მინიმალური დამფარავი ხის" გამოყოფა: ისეთი ხისა, რომელიც გრაფის ყველა წვეროს მოიცავს და რომლის წიბოთა წონების ჯამი მინიმალურია ყველა შესაძლო დამფარავი ხის წიბოთა წონების ჯამს შორის.

ქვემოთ მოყვანილ ნახაზში ნაჩვენებია ასეთი დამფარავი ხის მაგალითი.



ნახ. 11: მოცემული გრაფის მინიმალური დამფარავი ხე

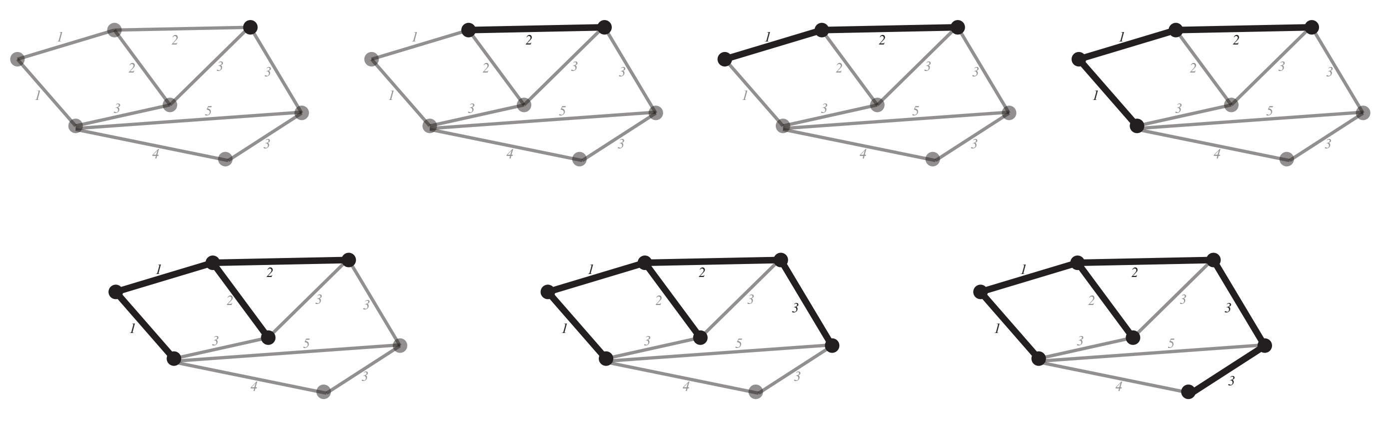
ეს ამოცანა ხშირად წამოიჭრება ხოლმე რამოდენიმე პუნქტს შორის ოპტიმალური ქსელის (მაგ. სატელეფონო, საკაბელო, საგზაო ან სხვა) შედგენის დროს.
გარდა ამისა, მოცემულ გრაფში მინიმალური დამფარავი ხის გამოყოფის შემდეგ ბევრი რთული ამოცანის მიახლოებითი ამოხსნის პოვნა შეიძლება.

ცხადია, რომ ბევ შემთხვევაში ერთ გრაფს რამოდენიმე მინიმალური დამფარავი ხე შეიძლება ქონდეს. თუ გრაფი შეწონილი არაა, მაშინ მიიჩნევენ, რომ მისი წიბოები ერთი და იგივე წონისაა (მაგ. 1).

მინიმალური დამფარავი ხის საპოვნელად ორი „ხარბი" ალგორითმი გამოიყენება, რომელსაც ახლა განვიხილავთ (ხარბი ეწოდება ისეთ ალგორითმს, რომელიც ყოველ ჯერზე შექმნილ სიტუაციაში საუკეტესო არჩევანს ეძებს, გლობალური სურათის გათვალისწინების გარეშე).

2.2.1 პრიმის ალგორითმი

ამ ალგორითმის პრინციპი საკმაოდ მარტივია: ხის აგებას ვიწყებთ ნებისმიერი წვეროდან და ვეძებთ მინიმალური წონის მქონე წიბოს, რომელიც უკვე შედგენილი ხის წვეროსა და დარჩენილ წვეროებს შორის (რომლებსაც ჩვენ *„გარე წვეროებს"* ვუწოდებთ) არსებობს. ასეთ წიბოს და მასთან მიერთებულ გარე წვეროს ხეს ვუმატებთ (თანაც ახალ წვეროს გარე წვეროთა სიმრავლიდან ვშლით). ამ პროცესს ვაგრძელებთ მანამ, სანამ გარე წვეროთა სიმრავლე არ იქნება ცარიელი.



ნახ. 12: მინიმალური დამფარავი ხის აგების პროცედურა

შესაბამისი ფსევდო კოდი შემდეგნაირად შეიძლება ჩაიწეროს:

საწყისი მონაცემი: შეწონილი გრაფი $G = (E, V)$ წონათა ფუნქციით $d : E \to \mathbb{Q}$;
მოსალოდნელი შედეგი: G გრაფის მინიმალური დამფარავი ხე.

$Prim(G, d)$
$TV = \emptyset$; /* ხის წვეროთა სიმრავლე ჯერ ცარიელია */
$TE = \emptyset$; /* ხის წიბოთა სიმრავლე ჯერ ცარიელია */
$Out = V$ /* თავდაპირველად ყველა წვერო გარეა (ხეში არ შედის) */
აირჩიე ნებისმიერი წვერო $v \in V$;
$TV = TV \cup \{v\}$ /* საწყისი წვერო ხის ნაწილი ხდება */
$Out = Out - \{v\}$; /* გარე წვეროთა სიმრავლე ერთით შემცირდა */
while($Out \neq \emptyset$)
{
მოძებნე ისეთი $w_1 \in TV$, $w_2 \in Out$, /* მოძებნე მინიმალური წონის მქონე ისეთი წიბო,
რომ $d(w_1, w_2) = \min\{d(x, y) | x \in TV, y \in Out\}$ რომლის ერთი წვერო აგებულ ხეშია, მეორე კი არა */
$TV = TV \cup \{w_2\}$; /* ხის წიბოთა სიმრავლეს ახალ წვეროს ვუმატებთ */
$Out = Out - \{w_2\}$; /* იგივე წვეროს ვაკლებთ გარე წვეროთა სიმრავლეს */
$TE = TE \cup \{(w_1, w_2)\}$; /* ხის წიბოთა სიმრავლეს შესაბამისი წიბო ემატება */
}
return$(T = (TV, TE))$;

<u>სავარჯიშო 2.15:</u> საწინააღმდეგოს დაშვებით დაამტკიცეთ ამ ალგორითმის სისწორე.

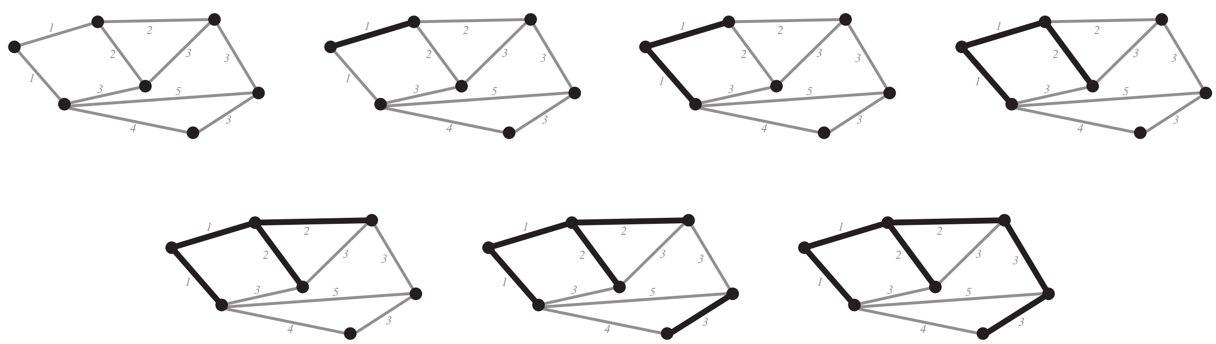
ბიჯების რაოდენობის შესაფასებლად შემდეგნაირად შეიძლება ვიმსჯელოთ: *while* ციკლის ყოველ ბიჯზე (რომელთა რაოდენობა წიბოთა რაოდენობის ტოლია) უნდა გადავათვალიეროთ ყველა ის წიბო, რომელიც გარე წვეროს ხის წვეროსთან აერთებს და ასეთებს შორის მინიმალურ წონიანი ავირჩიოთ.
მონაცემთა სტრუქტურის სწორად არჩევის შემთხვევაში პრიმის ალგორითმის რეალიზაცია $O(|V|^2)$ დროშია შესაძლებელი.
რადგან ზედა ზღვრის შეფასებაში წიბოების რაოდენობა არ ფიგურირებს, ეს ალგორითმი კარგად უნდა მუშაობდეს ე.წ. „მჭიდრო" გრაფებთან: თუ წიბოების რაოდენობა წვეროების რაოდენობასთან შედარებით დიდია, ეს მაინც ვერ მოახდენს გავლენას ალგორითმის მუშაობის რდოზე.

2.2.2 კრასკალის ალგორითმი

ახლა განვიხილოთ კრასკალის ალგორითმი, რომელიც უფრო სწრაფად ე.წ. „გაუხშოებულ" გრაფებზე მუშაობს: ისეთებზე, რომელთა წიბოთა რიცხვი წვეროების რიცხვთან შედარებით დაბალია.
თავიდან $G = (V, E)$ ბმული გრაფის ყველა წვერო განვიხილოთ როგორც ცალკე აღებული ერთ ელემენტიანი ხე (რაც იმას ნიშნავს, რომ ერთდროულად $|V|$ ცალ ხეს ვაგებთ). ყოველ ბიჯზე ვარჩევთ მინიმალური წონის ისეთ (u, v) წიბოს, რომ u და v წვეროები ერთსა და იმავე უკვე შექმნილ ხეს **არ** ეკუთვნოდეს (თუ ორივე წვერო ერთ ხეშია, მაშინ ამ წიბოს ვაგდებთ). თუ $u \in T_1$ და $v \in T_2$, $T_1 \neq T_2$, ამ ორ ხეს (u, v) წიბოთი ერთმანეთს ვაბამთ, რის შედეგადაც ერთი ხით ნაკლებს ვიღებთ. ამ პროცესს ვაგძელებთ მნამ, სანამ არ შეიქმნება ერთი ბმული კომპონენტი (ხე).

<u>სავარჯიშო 2.16:</u> დაამტკიცეთ, რომ ამ პროცედურის ჩატარების შედეგად აუცილებლად მივიღებთ ხეს (არ გვექნება ციკლები).

ყოველივე ზემოთ ნათქვამი განვიხილოთ მაგალითზე (ნახ. 13).



ნახ. 13: კრასკალის ალგორითმის მოქმედების მაგალითი

ყოველივე ეს შემდეგი ფსევდო კოდით შეიძლება ჩაიწეროს:

<u>*საწყისი მონაცემი:*</u> ბმული შეწონილი გრაფი $G = (V, E)$ წონის ფუნქციით $d : E \to \mathbb{Q}$.

Kruskal(G, d)
G გრაფის n წიბოსაგან შეადგინე ხე T /* თავდაპირველად T ხე n ცალი იზოლირებული წვეროსაგან შედგება */

შეადგინე G გრაფის ყველა შესაძლო წიბოთა სია /* ალგორითმის დასწრაფების მიზნით სია შეიძლება დავალაგოთ */

for($i = 1, i < |V|, i++$)
{
წიბოთა სიიდან ამოარჩიე მინიმალური წონის წიბო (u, v);
if(u და v სხვადასხვა ბმულ კომპონენტს ეკუთვნის)
T ხეში ჩაუმატე წიბო (u, v);
(u, v) წიბო ამოშალე წიბოთა სიიდან
}
return(T)

შენიშვნა: რადგან ყოველ ხეს წვეროებზე ერთით ნაკლები წიბო აქვს, *for* ციკლს შესაბამისად ვატრიალებთ.

<u>სავარჯიშო 2.17:</u> რა ცვლადები და ოპერაციები უნდა დავამატოთ ზედა ალგორითმს, რომ ადვილად დავადგინოთ, T ხეში u და v წვეროები ერთ ბმულ კომპონენტს ეკუთვნის, თუ არა?

<u>სავარჯიშო 2.18:</u> დაამტკიცეთ, რომ კრასკალის მეთოდის ბიჯების ზედა ზღვარია $O(|V| \cdot |E|)$.

<u>სავარჯიშო 2.19:</u> რომელ შემთხვევებში ჯობია პრიმის ალგორითმის გამოყენება და რომელში - კრასკალის? პასუხი დაასაბუთეთ.

2.3 უმოკლესი გზა გრაფში

თუ მოცემულია შეწონილი გრაფი, ხშირად საჭიროა ხოლმე მის ორ წვეროს შორის უმოკლესი გზის დადგენა (ორ წვეროს შემაერთებელ ყველა შესაძლო გზას შორის ისეთის არჩევა, რომლის წიბოთა წონების ჯამი მინიმალურია).

ამ ამოცანის გადაჭრაზე ზალიან ბევრი სხვა ამოცანაა დამოკიდებული, მათ შორის:

- სატრანსპორტო ქსელებში ორ პუნქტს შორის უმოკლესი გზის პოვნა: თუ გრაფს განვიხილავთ როგორც ქალაქებს (წვეროები) და მათ შემაერთებელ გზებს (წიბოები), ან ქალაქში ქუჩებს (წიბოები) და მათ გადაკვეთებს (წვეროები), ერთი პუნქტიდან მეორეში გადასვლისათვის უმცირესი გზის გამოთვლა ამ ამოცანის გადაჭრით შეიძლება;

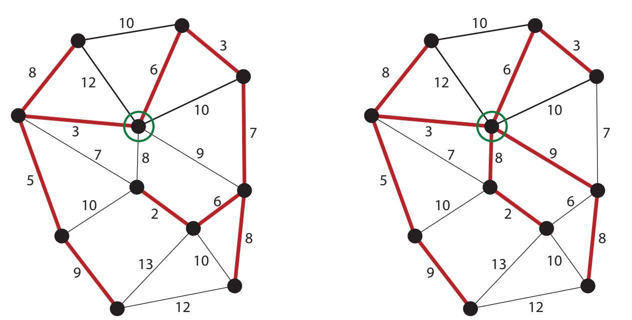
- ნალაპარაკევი ტექსტის ამოცნობის ერთ-ერთი უმთავრესი ამოცანა ერთნაირი ჟღერადობის სიტყვების (ომოფონების) განსხვავებაა. ასეთ სიტყვებზეა აგებული აკაკი წერეთელის ცნობილი ლექსი „აღმართ-აღმართ”:

  აღმართ-აღმართ მივდიოდი ***მე ნელა***,
  სერზედ შევდექ, ჭმუნვის ალი ***მენელა***;
  მზემან სხივი მომაფინა ***მაშინა***,
  სიცოცხლე ვგრძენ, სიკვდილმა ***ვერ მაშინა***.

  თუ ენის სიტყვებს აღვნიშნავთ, როგორც გრაფის წვეროებს და „მსგავს” სიტყვებს წიბოებით შევაერთებთ (თანაც მსგავსების კოეფიციენტს წიბოს წონად მივუწერთ - რაც უფრო მსგავსია ორი სიტყვა, უფრო ნაკლებს), წვეროებს შორის უმოკლესი გზის პოვნა წინადადების აზრის დადგენაში დაგვეხმარება.

- გრაფთა განლაგებაში: ხშირად საჭიროა ხოლმე გრაფის „ცენტრის” დადგენა და ისე განლაგება, რომ იგი მის შუაგულში მოექცეს. ასეთი შეიძლება იყოს წვერო, რომლის მაქსიმალური დაშორება ყველა სხვა წვეროსთან ყველაზე დაბალია. ცხადია, რომ ამის დასადგენად საჭიროა ნებისმიერ ორ წვეროს შორის მანძილის ცოდნა.

აღსანიშნავია, რომ უმცირესი დამფარავი ხე ყოველთვის უმცირეს მანძილს არ მოგვცემს, როგორც ეს შემდგომ ნახაზშია ნაჩვენები.



(ა) (ბ)

ნახ. 14: მინიმალური დამფარავი ხე (ა) და შემოხაზული წვეროდან უმოკლესი მანძილის ხე (ბ)

სავარჯიშო 2.20: მოიყვანეთ სხვა ხეების მაგალითი, რომლებშიც უმცირესი დამფარავი ხე არ მოგვცემს უმოკლეს მანძილებს.

$u, v \in V$ წვეროებსშორის უმოკლესი მანძილი აღვნიშნოთ როგორც $d(u, v)$ (თუ ეს წვეროები ერთმანეთთან დაკავშირებული არაა, მივიჩნიოთ $d(u, v) = \infty$), ხოლო გრაფის $e \in E$ წიბოს წონა იყოს მოცემული ფუნქციით $w(e)$. გრაფის რაიმე s წვეროდან ნებისმიერ სხვა t წვერომდე უმოკლესი გზის დასადგენად ამოცანას *„ბოლოდან”* შევხედოთ: თუ დადგენილი გვაქვს უმოკლესი მანძილი s წვეროსა და *ნებისმიერ* სხვა წვეროს შორის, უნდა ავარჩიოთ ისეთი x წვერო, რომლისთვისაც $d(s, x) + w(x, t)$ იქნება მინიმალური და s, t წვეროებს შორის მინიმალურ გზად მივიღებთ მინიმალურ გზას s, x წვეროებს შეორის და (x, t) წიბოს.

სხვა სიტყვებით რომ ვთქვათ, უნდა ავირჩიოთ t წვეროს ისეთი x მეზობელი, რომ x წვეროს გავლით გზა s წვეროდან t წვერომდე მინიმალური იყოს (ამის დადგენა შესაძლებელია, თუ s და x წვეროებს შორის უმოკლესი გზა წინასწარ გვექნება დადგენილი).
თავდაპირველად უნდა განვსაზღვროთ, თუ რისი ტოლია s წვეროს მანძილი თავის თავთან. ცხადია, რომ ***თუ გრაფში არ გვაქვს უარყოფითი წონის მქონე წიბოები,*** $d(s,s) = 0$. ამას გარდა, $\forall x \in V, x \neq s, d(s,x) = \infty$. ასევე უნდა განისაზღვროს ყველა წვეროს „წინამორბედი" - ის წვერო, რომელიც უმოკლეს გზაში ამ წვეროს უკავშირდება: $p(s) = s, \forall x \in V, x \neq s, p(x) = NULL$.

აქედან გამომდინარე, უმოკლესი გზის პოვნის ალგორითმის ფსევდოკოდი შემდეგნაირად შეიძლება ჩაიწეროს:

საწყისი მონაცემი: ბმული შეწონილი გრაფი $G = (V, E)$ წონის ფუნქციით $d : E \to \mathbb{Q}$ და ერთ-ერთი წვერო s.

```
ShortPathDijkstra(G, d, s)
for( ∀v ∈ G)                          /* თავდაპირველად ყველა წვეროს დაშორება
   {                                     საწყისთან არის უსასრულობა და
    d(v) = ∞;                            წინამორბედიც არ არსებობს */
    p(v) = NULL;
   }
d(s) = 0;                             /* მხოლოდ s წვეროსთვის განისაზღვრება
p(s) = s;                                თავის თავთან მანძილი და წინამორბედი */
Q = V;                                /* დასამუშავებელ წვეროთა სია თავიდან
                                         ყველა წვეროს მოიცავს */
while( Q ≠ ∅ )                        /* სანამ კიდევ გვაქვს დასამუშავებელი
   {                                     წვეროები */
    u = უმცირესი d(u) მანძილის მქონე წვერო;
    if( d(u) = ∞ )   break;           /* თუ უმცირესი მანძილი უსასრულობაა,
                                         ყველა დანარჩენი წვერო საწყისისგან
                                         იზოლირებულია */

    ამოაგდე u დასამუშავებელი წვეროების Q სიმრავლიდან;
    for( ∀v ∈ Q, (u, v) ∈ E)
    {
    if( d(v) > d(u) + w(u, v))
      {
       d(v) = d(u) + w(u, v);
       p(v) = u;
      {
    }
```

ეს ალგორითმი პირველად ჰოლანდიელმა მეცნიერმა ედსგერ დეიქსტრამ გამოაქვეყნა და მის სახელს ატარებს. უხეშად რომ ვთქვათ, ეს იგივე ხარბ პრინციპს იყენებს, როგორც პრიმის ალგორითმი იმ განსხვავებით, რომ პრიმის მეთოდი ყველა წვეროს შემაერთებელი დამფარავი ხის, დეიქსტრას ალგორითმი კი ორ წვეროს შორის უმოკლეს მანძილის მქონე ხის შექმნას ცდილობს.

სავარჯიშო 2.21: დაამტკიცეთ, რომ დეიქსტრას ალგორითმით მართლაც შესაძლებელია მოცემული წვეროდან ყველა სხვა წვერომდე უმოკლესი გზის პოვნა.

სავარჯიშო 2.22: დაამტკიცეთ, რომ დეიქსტრას ალგორითმის დროის ზედა ზღვარია $O(|V|^2)$.

სავარჯიშო 2.23: გააანალიზეთ დეიქსტრას ალგორითმი უარყოფით წონიან გრაფებზე. რატომ ვერ მოგვცემს იგი სწორ პასუხს?

3 ამოცანათა გრაფებზე გადატანის მაგალითები

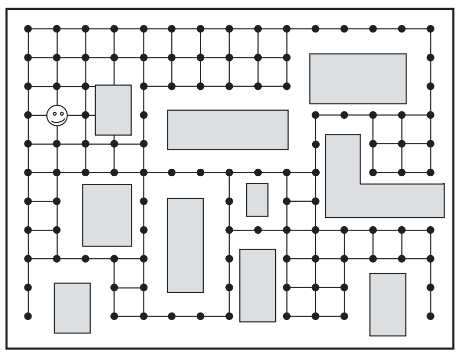
ამოცანათა სწორად ჩამოყალიბება და გრაფებზე გადატანა (მოდელირება) უმნიშვნელოვანეს როლს თამაშობს მათ გადაწყვეტაში - სწორად დასმული ამოცანა ნახევარი ამოხსნის ტოლფასია. აქამდე გრაფებთან დაკავშირებულ ძირითად ამოცანებსა და განსაზღვრებებს განვიხილავდით, ახლა კი გადავიდეთ ამოცანათა მოდელირების მაგალითებზე და მოვიყვანოთ ის უმნიშვნელოვანესი ამოცანები, რომელთა გადაჭრაც ხშირ შემთხვევაში მრავალი სხვა პრაქტიკული ამოცანის დაძლევაში მოგვეხმარება.

ქვემოთ მოყვანილი მაგალითებით ნათლად უნდა ჩანდეს, თუ როგორ შეიძლება ამა თუ იმ ამოცანის გრაფების მეშვეობით გადაჭრა. ხშირ შემთხვევაში გამოვიყენებთ იმ ფაქტს, რომ ამოცანის მონაცემთა ელემენტების სიმრავლეზე *მიმართებების* განმარტება შეიძლება (იმის გათვალისწინებით, თუ რა დამოკიდებულებაა ამ ელემენტებს შორის), ხოლო მიმართებების გამოსახვა გრაფთა მეშვეობით ადვილად შეიძლება.

შენიშვნა: ამოცანის პირობის წაკითხვის შემდეგ, სანამ კითხვას გააგრძელებთ, თქვენ თვითონ დაფიქრდით საკითხზე, თუ როგორ შეიძლება მისი გადატანა გრაფებზე.

ამოცანა: მოძრაობა ვიდეო თამაშებში. წარმოიდგინეთ, რომ გვინდა ვამოძრაოთ ვიდეო თამაშის გმირი ოთახში, რომელშიც საგნებია განთავსებული. როგორ შეიძლება ოპტიმალური გზის გამოანგარიშება?

ამ ამოცანის ამოხსნისას ბუნებრივად გრაფებში უმოკლესი გზის პოვნის ასოციაცია ჩნდება. მაგრამ როგორი უნდა იყოს გრაფი? პირველი, რაც თავში შეიძლება მოგვივიდეს, შემდეგი იდეაა: ოთახის სურათს დავადოთ ბადე, შემდეგ ამოვყაროთ ის წერტილები (მათთან მიერთებულ წიბოებთან ერთად), რომლებიც საგნების ნახატებს ემთხვევა. მივიღებთ გრაფს, რომლის წიბოებს შორის მანძილი ერთის ტოლად შეიძლება მივიჩნიოთ (ანუ გრაფი არ იქნება შეწონილი).



ნახ. 15: მოძრაობის ბადე

რა თქმა უნდა, შეგვეძლო სხვა გრაფის შექმნაც, რომელიც უფრო ეფექტურად გადაჭრიდა ამ ამოცანას (მაგალითად, არაა აუცილებელი სულ მართი კუთხე გვქონდეს - ზოგჯერ შეიძლება ზემოთ მოყვანილი ბადის დიაგონალებზე გასვლა, ან ისეთ ადგილებში, სადაც ბადის ორი პარალელური ხაზი გადის ერთის აღება და ა.შ. ამ შემთხვევებში გრაფები უკვე შეწონილი უნდა იყოს). ამას გარდა, არსებობს გეომეტრიული ალგორითმები, რომლებიც ანალოგიური ამოცანებისათვის უფრო ეფექტურად გადაჭრიდა უმოკლესი გზის საკითხს, მაგრამ ამ სახის ალგორითმის იმპლემენტაცია გაცილებით უფრო მარტივია და შედეგიც არ იქნება საგრძნობლად უარესი.

ამოცანა: გენეტური კოდის აგება. ამ ამოცანაში მოცემული გვაქვს დნმ კოდის ნაწილები $\Phi = (\phi_1, \phi_2, ..., \phi_n)$, რომლებიც ექსპერიმენტების შედეგად იქნა მიღებული. ყოველი $f \in \Phi$ ფრაგმენტისათვის ვიპოვნით ისეთ ელემენტებს, რომლებიც უნდა განთავსდნენ f ფრაგმენტის მარჯვნივ (ან, შესაბამისად, მარცხნივ). როგორც წესი, იარსებებს აგრეთვე ისეთი (ერთი ან რამოდენიმე) ელემენტი, რომლის განთავსებაც ორივე მხარეს შეიძლება. აღსანიშნავია, რომ განლაგების წესი ტრანზიტულია: თუ f_1 ფრაგმენტი f_2 ფრაგმენტის მარცხნივ და ეს კი თავის თავად f_3 ფრაგმენტის მარცხნივ უნდა განთავსდეს, მაშინ f_1 უნდა აღმოჩნდეს f_3 ნაწილის მარცხნივ.

გენეტური კოდის აგების ამოცანა იმაში მდგომარეობს, რომ ვიპოვნოთ Φ მიმდევრობის *ყველა* ელემენტისაგან შემდგარი ისეთი მიმდევრობა, რომელიც ზედა პირობებს აკმაყოფილებს. სხვა სიტყვებით რომ ვთქვათ, უნდა ვიპოვნოთ Φ მიმდევრობის ისეთი პერმუტაცია $(\phi_{i_1}, ..., \phi_{i_n})$, რომ თუ $k < l$, მაშინ ზემოთ მოყვანილი წესების თანახმად ϕ_{i_k} ფრაგმენტი *უნდა* იდგეს ϕ_{i_l} ფრაგმენტის მარცხნივ (შესაბამისად, ϕ_{i_l} ფრაგმენტი *უნდა* იდგეს ϕ_{i_k} ფრაგმენტის *მარჯვნივ*).

ამ ამოცანის გადაჭრა შემდეგნაირად შეიძლება: შევადგინოთ მიმართება

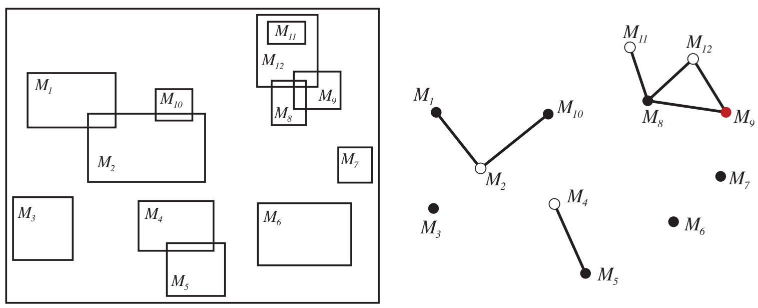
$$R = \{(\phi_i,\phi_j)|\phi_i \text{ ფრაგმენტი უნდა განთავსდეს } \phi_j \text{ ფრაგმენტის } \textit{მარცხნივ}\}.$$

ცხადია, რომ ეს მიმართება შექმნის (ერთ ან რამოდენიმე) მიმართულ აციკლურ გრაფს, რომლის ტოპოლოგიური დალაგება საძიებო მიმდევრობას მოგვცემს.

სავარჯიშო 3.1: დაამტკიცეთ, რომ ზემოთ მოყვანილი წესით მიმართულ *აციკლურ* გრაფს მივიღებთ.

სავარჯიშო 3.2: დაამტკიცეთ, რომ ამ აციკლური გრაფების ტოპოლოგიური დალაგება მართლაც გენეტური კოდის დასაშვებ მიმდევრობას მოგვცემს (Φ მიმდევრობის ისეთ პერმუტაციას $(\phi_{i_1},...,\phi_{i_n})$, რომ თუ $k < l$, მაშინ ϕ_{i_k} ფრაგმენტი *შეიძლება* იდგეს ϕ_{i_l} ფრაგმენტის მარცხნივ).

ამოცანა: ობიექტების დაჯგუფება. გრაფიკულ ამოცანებში ხშირად საჭიროა ხოლმე გრაფიკული ობიექტების (მაგ. მართკუთხედების) ისეთი დაჯგუფება, რომ ერთ ჯგუფში მყოფი ობიექტები ერთმანეთს არ კვეთდნენ (სხვადასხვა ჯგუფში მოხვედრილი ობიექტები ერთმანეთს შეიძლება კვეთდნენ).



ნახ. 16: გეომეტრიული ობიექტები და მათი შესაბამისი (სამად შეღებილი) გრაფი

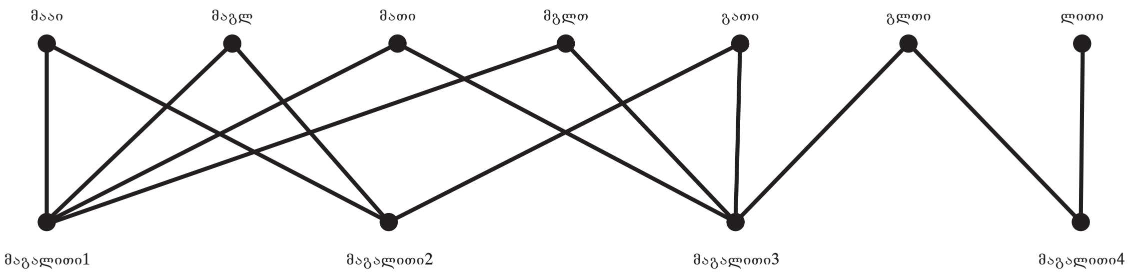
ამ ამოცანის გადასაჭრელად ყოველ ობიექტს გრაფის წვერო შევუსაბამოთ. თუ ორი ობიექტი ერთმანეთს კვეთს, მაშინ შესაბამისი წვეროები წიბოთი შევაერთოთ. ცხადია, გრაფიკულ ობიექტთა ყოველი გაერთიანება ასე შექმნილი გრაფის დამოუკიდებელ წვეროთა სიმრავლეა.
გრაფის წვეროების შეღებვის ამოცანა, რომელიც ზემოთ გვქონდა აღწერილი, სწორედ ასეთ იზოლირებულ წერტილთა სიმრავლეს (და, შესაბამისად, არაგადამკვეთ ობიექტთა გაერთიანებას) მოგვცემს. გრაფის შეღებვაში გამოყენებული ფერების რაოდენობის მინიმიზაცია კი მინიმალური რაოდენობის გაერთიანებებს მოგვცემს.

ზემოთ მოყვანილი ნახაზის მაგალითში ობიექტები სამ კლასად შეიძლება დავაჯგუფოთ:

$$K_1 = \{M_1, M_3, M_5, M_6, M_7, M_8, M_{10}\},\ K_2 = \{M_2, M_4, M_{11}, M_{12}\},\ K_3 = \{M_9\}.$$

ამოცანა: სიტყვათა შემოკლება. ზოგჯერ საჭიროა ხოლმე დიდ მონაცემთა ბაზაში მონაცემთა სახელების სიგრძის შემცირება. მაგალითად, თუ გვაქვს გრძელ სიტყვათა სიმრავლე (დავუშვათ, რამოდენიმე ასეული 32 სიმბოლოსაგან შემდგარი), მათი სიგრძის შემცირება შეიძლება (მაგალითად 8 სიმბოლოიანზე) ისე, რომ სხვადასხვა სიტყვა ისევე სხვადასხვა დარჩეს. ამ შემთხვევაში პირველი 8 ასოს აღება არ გამოდგება, რადგან „მაგალითი1" და „მაგალითი2" განსხვავებული აღარ იქნება. რა წესით უნდა შევამოკლოთ სიტყვები ისე, რომ შედეგები ერთმანეთს არ დაემთხვეს?
ყოველ შესამოკლებელ სიტყვას w_i შევუსაბამოთ გრაფის ერთი წვერო v_i. შემდეგ ყოველი w_i სიტყვისათვის შევქმნათ შესაძლო შემოკლებული სიტყვები $w'_{i,j}$ და შევუსაბამოთ წვეროები $v'_{i,j}$. შემდეგ გავავლოთ წიბოები $v_i, v'_{i,j}$ ყველა შესაძლო i და j პარამეტრებისათვის (ყოველ სიტყვასა და მის შესაძლებელ შემოკლებას შორის).

ცხადია, რომ თუ ამ გრაფში ავიღებთ ისეთ წიბოებს, რომლებიც ერთმანეთთან არ იქნება დაკავშირებული (იზოლირებულ წიბოებს), ცალსახა შემოკლებების სიის შედგენას შევძლებთ. ასე, ნახ. 17-ში მოყვანილ მაგალითში შეიძლება ავიღოთ წიბოები {(მაგალითი1, მაგლ), (მაგალითი2, მააი), (მაგალითი3, გათი), (მაგალითი4, ლითი)}, რაც ერთ-ერთ შესაძლებელ შემოკლების სიას მოგვცემს.



ნახ. 17: შემოკლების გრაფი

ამოცანა: გაყალბების აღმოჩენა. ამ ამოცანაში საჭიროა გამყალბებელთა დოკუმენტების დადგენა. ხშირად ისე ხდება ხოლმე, რომ გამყალბებლები მათ მიერ შექმნილ საბუთებს გზავნიან (ბანკებში, საგადასახადო სისტემებში გადასახადების ასანაზღაურებლად, ბენზოგასამართ სადგურებში ან სხვა დაწესებულებებში გაყალბებული ვაუჩერებით საქონლის მისაღებად და ა.შ.), რომლებიც იდენტური არაა, მაგრამ გარკვეული თვალსაზრისით *ერთმანეთის მსგავსია*. როგორ შეიძლება მათი აღმოჩენა?

პირველ რიგში დასადგენია, როგორი საბუთები ითვლება *მსგავსად* (სხვადასხვა ამოცანისათვის ეს სხვადასხვა შეიძლება იყოს, როგორც მაგ. მსგავსი ნომრები, ფორმები, მისამართები, სახელები, გვარები და ა.შ.).

ყოველ საბუთს გრაფის ერთი წვერო შევუსაბამოთ და ორ წვეროს შორის წიბო გავავლოთ, თუ შესაბამისი საბუთები მსგავსია. ცხადია, რომ თუ ასეთ გრაფში ვიპოვნით წვეროთა სიმრავლეს, რომელიც *ბევრი* წიბოთი იქნება შეერთებული (მაგალითად ყველა ყველასთან - სრული ქვეგრაფი), შესაბამისი საბუთების უფრო დეტალური შესწავლა შეიძლება ღირდეს. ამ ამოცანის გადასაწყვეტად გრაფში მაქსიმალური სრული ქვეგრაფის ამორჩევის ალგორითმები შეიძლება გამოგვადგეს.

4 არითმეტიკული ალგორითმები და მათი გამოყენება

რადგან ჩვენ ორობით სისტემაში მოქმედებას ვაპირებთ, აუცილებელია შესაბამისი ოპერაციების განსაზღვრაც. თუ ჩვენ ათობით არითმეტიკაში (შესაბამისად ალგებრაში) მიმატების, გამრავლების, გაყოფის ოპერაციები გვაქვს შემოღებული, ანალოგიური ოპერაციები უნდა შემოვიღოთ ორობით ალგებრაშიც, ანუ **ბულის ალგებრაში**, როგორც ამას მისი ფუძემდებლის, ინგლისელი მათემატიკოსის ჯორჯ ბულის (George Boole) პატივსაცემად უწოდებენ.

4.1 ბულის ალგებრის ელემენტები

ბულის ლოგიკა და, აქედან გამომდინარე, ბულის ალგებრა $\mathbb{B} = \{0, 1\}$ ორობით ანბანზეა განსაზღვრული. ზოგადად რომ ვთქვათ, ეს კლასიკური ლოგიკის მათემატიკურ ენაზე გადატანის ერთ-ერთი (ყველაზე გავრცელებული) მაგალითია. ლოგიკაში გვაქვს ჭეშმარიტი და მცდარი გამონათქვამები: ყოველი გამონათქვამი (მაგალითად, „$3 + 4 = 7$", „$12 - 3 = 1$", „ხვალ მზე ამოვა", „გუშინ წვიმდა" და ა.შ.) ***ან*** ჭეშმარიტია, ***ან*** მცდარი - სხვა რამ შეუძლებელია.
ბულის ძირითადი იდეა გამონათქვამების ***მათემატიკურ ცვლადებზე გადატანა*** იყო: ყოველი გამონათქვამი X ან ჭეშმარიტია (მაშინ $X = 1$), ან მცდარი ($X = 0$). ასევე შესაძლებელია გამონათქვამების კომბინირებაც, მაგალითად: „გუშინ მზე ამოვიდა ***და ამავდროულად*** წვიმდა", ან „$2 + 3 = 5$ და ამავდროულად $2 - 7 = 1$".
ამ მაგალითებში, თუ $X =$„გუშინ მზე ამოვიდა", $Y =$„წვიმდა", მაშინ სრული გამონათქვამი $Z =$„გუშინ მზე ამოვიდა ***და ამავდროულად*** წვიმდა" მათემატიკურად შემდეგნაირად ჩაიწერება: $Z = X\&Y$. საბოლოო ჯამში, თუ გუშინ მართლა ამოვიდა მზე ($X = 1$) და ამ დროს მართლაც წვიმდა ($Y = 1$), მაშინ $Z = X\&Y = 1$ ჭეშმარიტი იქნება.
მეორეს მხრივ, თუ $X' =$„$2 + 3 = 5$" (ჭეშმარიტია) და $Y' =$„$2 - 7 = 1$" (მცდარია), ცხადია, რომ $X' = 1$ და $Y' = 0$. აქედან გამომდინარე, $X'\&Y' = 0$ და გამონათქვამი $Z =$„$2 + 3 = 5$ და ამავდროულად $2 - 7 = 1$" მცდარია.
ანალოგიურად შეიძლება შემდეგი გამონათქვამების შედგენა: „ხვალ იწვიმებს ***ან*** ხვალ ქარი იქნება"; „$3+7 = 11$ ***ან*** $2 - 5 = -3$". ცხადია, რომ ასეთი გამონათქვამები ჭეშმარიტია, თუ ერთი მაინც ჭეშმარიტია. მათემატიკურად ეს შემდეგნაირად შეიძლება ჩამოყალიბდეს: $X =$„ხვალ იწვიმებს", $Y =$„ხვალ ქარი იქნება", $Z = X \vee Y$=„ხვალ იწვიმებს ***ან*** ხვალ ქარი იქნება"; $X' =$„$3 + 7 = 11$", $Y' =$„$2 - 5 = -3$", $Z' = X' \vee Y' = 1$: აქ ***ან*** ერთი უნდა შესრულებულიყო, ***ან*** მეორე.
მესამე მნიშვნელოვანი ოპერაცია ***უარყოფაა***: გამონათქვამის შებრუნებულის აღება.
მაგალითად, „ხვალ იწვიმებს"$\longrightarrow$„ხვალ *არ* იწვიმებს; „$3 + 7 = 13 \longrightarrow 3 + 7 \neq 13$" და ა.შ. X გამონათქვამის უარყოფა მათემატიკურად შემდეგნაირად ჩაიწერება: $\neg X$.
ბულის ალგებრის ეს სამი ოპერაცია ქართულ ენაზე შემდეგნაირად შეიძლება ჩამოვაყალიბოთ: გამონათქვამი $Z = X\&Y$ ჭეშმარიტია, თუ X ***და*** Y გამონათქვამი ორივე ჭეშმარიტია; $Z = X \vee Y$ ჭეშმარიტია, თუ X ***ან*** Y ჭეშმარიტია; $Z = \neg X$ ჭეშმარიტია, თუ X მცდარია.
ეს ყველაფერი მათემატიკერ ენაზე შემდეგნაირად ჩამოყალიბდება: ორი X, Y გამონათქვამის ***კონიუნქცია*** $X\&Y$ ორ ცვლადიან ფუნქციას $f : \mathbb{B}^2 \to \mathbb{B}$ ეწოდება, რომლის მნიშვნელობაა 1, თუ ორივე ცვლადის მნიშვნელობაა 1; ორი X', Y' გამონათქვამის ***დიზიუნქცია*** $X \vee Y$ ორ ცვლადიან ფუნქციას $g : \mathbb{B}^2 \to \mathbb{B}$ ეწოდება, რომლის მნიშვნელობაა 1, თუ ერთ-ერთი ცვლადის მნიშვნელობაა 1; Z გამონათქვამის ***უარყოფა*** $\neg Z$ ერთ ცვლადიან ფუნქციას $h : \mathbb{B} \to \mathbb{B}$ ეწოდება, რომლის მნიშვნელობაა 1, თუ Z ცვლადის მნიშვნელობაა 0.
ყოველივე ეს ცხრილის სახითაც შეიძლება გამოვსახოთ:

X	Y	$X\&Y$	$X \vee Y$	$\neg X$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

ამ ცხრილში მოცემულია აღსაწერი ფუნქციების მნიშვნელობები ცვლადების (ამ შემთხვევაში X და Y) ყველა შესაძლო კომბინაციისათვის.
შენიშვნა: კონიუნქცია და უარყოფა სხვადასხვანაირად აღიწერება ხოლმე. სიმარტივისთვის შეგვიძლია დავწეროთ: $X\&Y = X \cdot Y = XY$, $\neg X = \overline{X}$.

ბულის ალგებრაში უსასრულოდ ბევრი ფუნქციის მოყვანა შეიძლება, მაგრამ მთავარი ისაა, რომ ყველა ეს ფუნქცია ზემოთ მოყვანილი დიზიუნქციის, კონიუნქციისა და უარყოფის საშუალებით გამოისახება.
ფუნქციების მაგალითად შეგვიძლია მოვიყვანოთ:

$$\begin{aligned} f(x_1,x_2,x_3) &= \overline{x_1}x_2 \vee x_3, \\ g(x_1,x_2,x_3,x_4) &= (x_1 \vee x_2\overline{x_3} \vee \overline{x_4}, x_1x_2), \\ h(x_1,x_2) &= (x_1x_2, x_1 \vee x_2, x_1\overline{x_2} \vee \overline{x_1}x_2) \end{aligned}$$

ამ მაგალითში f ფუნქცია სამ ცვლადიანია (თითოეული ცვლადი $\mathbb{B}$ სიმრავლიდან) და ერთ ელემენტს გვაძლევს პასუხად (იგივე $\mathbb{B}$ სიმრავლიდან). ამ შემთხვევაში იტყვიან, რომ ეს ფუნქცია $\mathbb{B}^3$ სიმრავლეს ასახავს $\mathbb{B}$ სიმრავლეში: $f:\mathbb{B}^3 \to \mathbb{B}$.
g ფუნქცია 4 ცვლადს ასახავს ორ პარამეტრიან პასუხში, ესე იგი $g:\mathbb{B}^4 \to \mathbb{B}^2$, ხოლო $h\mathbb{B}^2 \to \mathbb{B}^3$.
ზოგადად, თუ რამიე ფუნქცია ϕ n ცვლადიანია, ხოლო ეს ცვლადები მნიშვნელობას რაიმე A სიმრავლიდან შეიძლება იღებდნენ და მისი პასიხი m პარამეტრიანია და ამ პასუხის ელემენტები C სიმრავლიდან შეიძლება იყოს, იტყვიან, რომ ეს ფუნქცია A^n სიმრავლეს (ანუ A სიმრავლის ელემენტებისაგან შემდგარ n სიგრძის სიტყვას - ვექტორს) ასახავს C^m სიმრავლეში (ანუ C სიმრავლის ელემენტებისაგან შემდგარ m სიგრძის სიტყვაში - ვექტორში).
მათემატიკურად ეს შემდეგნაირად ჩაიწერება: $\phi: A^n \to C^m$.
ამ თავში ჩვენ $f:\mathbb{B}^n \to \mathbb{B}^m$ ფუნქციებს განვიხილავთ. ასეთ ფუნქციებს დისკრეტულსაც, ანუ ყველგან წყვეტილს უწოდებენ. ანალოგიურად, ასეთ ფუნქციებზე შედგენილ მათემატიკას დისკრეტული მათემატიკა ეწოდება.

განვიხილოთ დისკრეტული ფუნქცია $f(x_1,x_2,x_3,x_4) = x_1\overline{x_2}x_3 \vee x_4 \vee \overline{x_2x_3} \vee x_1x_2x_3x_4$. ამ ფუნქციის გამოსათვლელად საჭიროა შემდეგი ბიჯები:

1. $z_1 = x_2x_3$;
2. $z_2 = x_1z_1 = x_1x_2x_3$;
3. $z_3 = z_2x_4 = x_1x_2x_3x_4$;
4. $z_4 = \overline{z_1} = \overline{x_2x_3}$;
5. $z_5 = \overline{x_2}$;
6. $z_6 = x_1z_5 = x_1\overline{x_2}$;
7. $z_7 = z_6x_3 = x_1\overline{x_2}x_3$;
8. $z_8 = z_7 \vee x_4 = x_1\overline{x_2}x_3 \vee x_4$;
9. $z_9 = z_8 \vee z_4 = x_1\overline{x_2}x_3 \vee x_4 \vee \overline{x_2x_3}$;
10. $z_{10} = z_9 \vee z_3 = x_1\overline{x_2}x_3 \vee x_4 \vee \overline{x_2x_3} \vee x_1x_2x_3x_4$.

ლოგიკურად ისმის ორი შეკითხვა: რამდენი ოპერაციის ჩატარება გვიხდება ამ გამოსახულების გამოსათვლელად? რამდენი ბიჯია (დროა) საჭირო ამ გამოსახულების გამოსათვლელად? ამ შეკითხვებზე პასუხის გაცემა შემდეგნაირად შეიძლება:
ოპერაციათა რაოდენობის დასათვლელად საკმარისია ლოგიკური ოპერაციების (კონიუნქცია, დიზიუნქცია, უარყოფა) დათვლა: $C(f(x_1,x_2,x_3,x_4)) = 10$.
რაც შეეხება დროს (ბიჯების რაოდენობას) $T(f(x_1,x_2,x_3,x_4))$, ზემოთ მოყვანილ გამოთვლის მეთოდში ეს ოპერაციათა რაოდენობას დაემთხვა, რადგან ჩვენ ყველა ოპერაციას რიგ-რიგობით ვატარებდით.
ამ მაგალითში გასათვალისწინებელია ის ფაქტი, რომ რამოდენიმე ოპერაცია ***ერთდროულად*** შეიძლება ჩატარდეს: მაგალითად, შესაძლებელია (x_1x_3), $\overline{x_2}$ და $(x_2x_3$ გამოსახულებების გამოთვლა, რადგან ისინი ერთმანეთზე დამოკიდებული არაა, განსხვავებით, მაგალითად, $y_1 = x_1x_2$ და $y_2 = x_1x_2x_3$ გამოსახულებელისაგან, რომელთა გამოთვლა ერთდროულად არ შეიძლება: ერთი მეორეზეა დამოკიდებული.

აქედან გამომდინარე, შეგვიძლია შემდეგი „პარალელური” ალგორითმის შემოთავაზება:

1. $z_1 = x_2x_3$ და ***ამავდროულად*** $z_2 = x_1x_4$ და ***ამავდროულად*** $z_5 = \overline{x_2}$ და ***ამავდროულად*** $z_6 = x_1x_3$;
2. $z_3 = z_1z_2 = x_1x_2x_3x_4$ და ***ამავდროულად*** $z_4 = \overline{z_1} = \overline{x_2x_3}$ და ***ამავდროულად*** $z_7 = z_5z_6 = x_1\overline{x_2}x_3$;
3. $z_8 = z_7 \vee x_4 = x_1\overline{x_2}x_3 \vee x_4$ და ***ამავდროულად*** $z_9 = z_4 \vee z_3 = \overline{x_2x_3} \vee x_1x_2x_3x_4$;
4. $z_{10} = z_8 \vee z_9 = x_1\overline{x_2}x_3 \vee x_4 \vee \overline{x_2x_3} \vee x_1x_2x_3x_4$.

აქ მნიშვნელოვანია ის ფაქტი, რომ გარკვეული ოპერაციები ***ერთდროულად*** სრულდება, რის ხარჯზეც ფუნქციის ***სიღრმე*** (ანუ გამოთვლის ბიჯების რაოდენობა) მცირდება.

აქედან გამომდინარე ვიღებთ შემდეგ განსაზღვრებას:

განმარტება 4.1: $f : \mathbb{B}^n \to \mathbb{B}^m$ ბულის ფუნქციის ოპერაციათა რაოდენობა $C(f(x_1, ..., x_n)$ მასში შემავალი კონიუნქციის, დიზიუნქციისა და უარყოფების რაოდენობათა ჯამის ტოლია; იგივე ფუნქციის სიღრმე $C(f(x_1, ..., x_n)$ მისი რეალიზაციისათვის პარალელურად ჩატარებულ ოპერაციათა ბიჯების რაოდენობის ტოლია.

აღსანიშნავია, რომ თუ ფუნქცია მრავალგანზომილებიანია, როგორც, მაგალითად, $f(x_1, x_2) = (\overline{x_1} \vee \overline{x_2}, x_2)$, მისი ელემენტების რაოდენობის გამოსათვლელად უნდა დავითვალოთ ყველა პასუხისთვის (ამ შემტხვევაში $C(\overline{x_1} \vee \overline{x_2}) = 3$, $C(x_2) = 0$ და დავითვალოთ მათი ჯამი: $C(f(x_1, x_2)) = 3 + 0 = 3$, ხოლო სიღრმის დასათვლელად უნდა გამოვიანგარიშოთ თითოეულის სიღრმე და ავიღოთ მათი მაქსიმუმი (ფუნქიის ყოველი მნიშვნელობის გამოთვლა შეიძლება ერთდროულად): $T(\overline{x_1} \vee \overline{x_2}) = 2$, $T(x_2) = 0$, $T(\overline{x_1} \vee \overline{x_2}, x_2) = \max\{T(\overline{x_1} \vee \overline{x_2}), T(x_2)\} = \max\{2, 0\} = 2$.

სავარჯიშო 4.1: განიხილეთ ფუნქციები $f(x_1, x_2, x_3) = \overline{x_1}x_2 \vee x_3$, $g(x_1, x_2, x_3, x_4) = (x_1 \vee x_2\overline{x_3} \vee \overline{x_4}, x_1x_2)$ და $h(x_1, x_2) = (x_1x_2, x_1 \vee x_2, x_1\overline{x_2} \vee \overline{x_1}x_2)$. დაითვალეთ მათი ოპერაციათა რაოდენობა და სიღრმე.

იმ ამოცანებში, რომელთა განხილვას ჩვენ ვაპირებთ, მნიშვნელოვან როლს ორის მოდულით მიმატება ასრულებს. ეს განპირობებულია იმით, რომ კომპიუტერული სისტემები ორობით ანბანზეა აგებული.
ორობითი მიმატება (როგორც მას სხვანაირად უწოდებენ) განსაზღვრულია შემდეგნაირად:
ფუნქცია $f(x, y) = x \oplus y = 1$ მაშინ და მხოლოდ მაშინ, თუ მისი ცვლადებიდან ***ზუსტად ერთი*** ტოლია ერთის: $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$.
როგორ შეიძლება ამ ფუნქციის კონიუნქიის, დიზიუნქციისა და უარყოფის მეშვეობით ჩაწერა? პირველ რიგში ქართულ ენაზე ჩამოვაყალიბოთ, თუ ცვლადების რა მნიშვნელობებისათვის ხდება ფუნქცია 1:
$f(x, y) = x \oplus y$ ფუნქცია ერთის ტოლი ხდება მაშინ და მხოლოდ მაშინ, თუ $x = 1$ და $y = 0$ ***ან*** $x = 0$ და $y = 1$.
ლოგიკურად, თუ ცვლადი არის 1, მისი უარყოფა უნდა იყოს 0. ამიტომაც ვიღებთ შემდეგ გამონათქვამს:$f(x, y) = x \oplus y$ ფუნქცია ერთის ტოლი ხდება მაშინ და მხოლოდ მაშინ, თუ $x = 1$ და $\neg y = 1$ ***ან*** $\neg x = 1$ და $y = 1$. ეს შემდეგ ფუნქციას განაპირობებს: $f(x, y) = x \oplus y = \neg x \cdot y \vee \neg x \cdot y$. აქ „$x = 0$" (ანალოგიურად „$y = 0$") გამონათქვამების „$\neg x = 1$" („$\neg y = 1$") გამონათქვამებად შეცვლა იმიტომ დაგვჭირდა, რომ $x \cdot y$ გამოსახულება გამოსულიყო 1, თუ ***ზუსტად*** ერთი ცვლადია 1.

ზოგადად, თუ რაიმე უცნობი ფუნქცია მოცემულია ცხრილის სახით, მისი ფორმულებით ჩაწერა შემდეგნაირად შეიძლება:
მოცემულია ფუნქცია $f(x_1, x_2, ..., x_n)$;

- გამოყავით ცვლადების ის კომბინაციები, რომლისთვისაც ფუნქცია ხდება 1 (ზედა შემთხვევაში ესაა $x = 0, y = 1$ და $x = 1, y = 0$);
- თითოეული ასეთი კომბინაციისათვის შეადგინეთ კონიუნქციებისაგან შემდგარი გამოსახულება. თუ $c_i = 0$, აიღეთ $\neg c_i$, წინააღმდეგ შემთხვევაში თვითონ c_i (ზედა შემთხვევაში ესაა $\neg x \cdot y$ და $x \cdot \neg y$);
- ეს გამოსახულებები შეაერთეთ დიზიუნქციებით (ზედა შემთხვევაში ვიღებთ $\neg x \cdot y \vee x \cdot \neg y$).

ასეთი სახით ჩაწერილ ფუნქციებს, სადაც კონიუნქციებით შეკრული ცვლადებით (ან მათი უარყოფებით) გამოსახულებები შეერთებულია დიზიუნქციებით, დიზიუნქციური ნორმალური ფორმა ეწოდება.

სავარჯიშო 4.2: ცრილით მოცემული ფუნქციები ჩაწერეთ დიზიუნქციური ნორმალური ფორმით:

x_0	x_1	x_2	f_1	f_2	f_3	f_4
0	0	0	1	0	1	1
0	0	1	0	1	0	0
0	1	0	0	1	1	0
0	1	1	1	0	1	1
1	0	0	1	1	0	0
1	0	1	1	1	1	1
1	1	0	0	0	1	0
1	1	1	0	0	0	0

აღსანიშნავია, რომ (ჩვეულებრივი ალგებრის მსგავსად) ჭეშმარიტია შემდეგი ტოლობები:

$$x(y \vee z) = x \cdot y \vee x \cdot z; \;\; x \vee 0 = x; \;\; x \vee 1 = 1; \;\; x \cdot 0 = 0; \;\; x \vee 0 = x.$$

სავარჯიშო 4.3: დაამტკიცეთ ზემოთ მოყვანილი ტოლობების ჭეშმარიტება (მოიყვანეთ თითოეული ფუნქციის ცხრილი და შეადარეთ მათი მნიშვნელობები).

სავარჯიშო 4.4: დაამტკიცეთ: $x \vee x \cdot y = x; \;\; \neg x \vee x \cdot y = \neg x \vee y.$

როგორც სიმრავლეთა თეორიაში, ასევე ბულის ლოგიკაშიც მნიშვნელოვანია ე.წ. დე მორგანის კანონები:

$$x \vee y = \overline{\overline{x} \cdot \overline{y}}; \;\; x \cdot y = \overline{\overline{x} \vee \overline{y}}.$$

სავარჯიშო 4.5: დაამტკიცეთ დე მორგანის კანონში მოყვანილი ფორმულები.

სავარჯიშო 4.6: რისი ტოლია $\neg(\neg x)$?

დე მორგანის კანონებზე დაყრდნობით დისიუნქციური ნორმალური ფორმის გადაყვანა შეიძლება ე.წ. კონიუნქციურ ნორმალურ ფორმაში - ისეთ გამოსახულებაში, რომელიც შედგება ცვლადების დიზიუნქციური გაერთიანებებით და ამ გამოსახულებათა კონიუნქციებით გაერთიანებებისაგან. კონიუნქციური ნორმალური ფორმით ჩაწერილი ფუნქციების მაგალითებია $(x_2 \vee \neg x_3)(x_1 \vee x_2 \vee x_3)(x_1 \vee x_2 \vee \neg x_3)$ და $(x_1 \vee x_3)(x_1 \vee x_2)(x_1 \vee \neg x_2 \vee x_3)$, მაგრამ არა $(x_2 \vee \neg x_3)(x_1 \vee x_2 \vee x_3)(x_1 \vee x_2 \vee \neg x_3) \vee x_1$.

თუ მოცემული გვაქვს რაიმე ფუნქცია, ზოგჯერ მისი ოპერაციებისა და ბიჯების რაოდენობის შემცირება შეიძლება ზემოთ მოყვანილი ტოლობების მეშვეობით: $(x_1 \vee x_2\overline{x_3} \vee \overline{x_4}x_1x_2) = x_1(1 \vee \overline{x_4}x_2) \vee x_2\overline{x_3} = x_1 \vee x_2\overline{x_3}$.
იგივე ფუნქციის კონიუნქციური ნორმალური ფორმით ჩაწერა შემდეგნაირად შეიძლება:
$x_1 \vee x_2\overline{x_3} = \neg(\overline{x_1} \cdot \overline{(x_2\overline{x_3})}) = \neg(\overline{x_1} \cdot (\overline{x_2} \vee x_3))$.

სავარჯიშო 4.7: შემდეგი ფუნქციები ჩაწერეთ დიზიუნქციური ნორმალური ფორმით:

$$\begin{aligned} f(x_1, x_2, x_3) &= (x_2 \vee \neg x_3)(x_1 \vee x_2 \vee x_3)(x_1 \vee x_2 \vee \neg x_3); \\ g(x_1, x_2, x_3) &= (x_1 \vee x_3)(x_1 \vee x_2)(x_1 \vee \neg x_2 \vee x_3); \\ h(x_1, x_2, x_3) &= (x_1 \vee \neg x_2)(x_1 \vee x_2)(x_1 \vee x_2 \vee x_3). \end{aligned}$$

4.2 n ბიტიანი რიცხვების მიმატება

მიმატების ოპერაცია იმდენად ხშირია ჩვენს ყოველდღიურ ცხოვრებაში, რომ ბევრ ადამიანს, ალბათ, არც კი მოსვლია თავში აზრად ის ფაქტი, რომ ეს პროცესი არც თუ ისე მარტივია. მაგალითად, ყველა სწრაფად გამოგვითვლის $5 + 3 = 8$, მაგრამ $3434164136861 + 3289747301047 = 6723911437908$ არც თუ ისე მცირე დროსა და ყურადღებას მოითხოვს. ზოგადად, რაც უფრო გრძელია შესაკრები რიცხვები, მით უფრო დიდ დროს ვანდომებთ გამოთვლას.
მიუხედავად იმისა, რომ შეკრების ყველაზე მარტივი ალგორითმი - ქვეშ მიწერით მიმატება - საყოველთაოდ ცნობილია, ჩვენ მაინც შევეცდებით მის განხილვასა და გაანალიზებას და ამას როგორც ათობით, ასევე ორობითი რიცხვების მაგალითზე გავაკეთებთ.

4.2.1 ქვეშ მიწერით მიმატების მეთოდი

საყოველთაოდ ცნობილი მეთოდის გარჩევა მარტივი მაგალითით დავიწყოთ:

$$+\begin{array}{r} 427 \\ 613 \\ \hline 0 \end{array}\ 1 \qquad +\begin{array}{r} 427 \\ 613 \\ \hline 40 \end{array}\ 0 \qquad +\begin{array}{r} 427 \\ 613 \\ \hline 040 \end{array}\ 1 \qquad +\begin{array}{r} 427 \\ 613 \\ \hline 1040 \end{array}$$

ზოგადად, თუ მოცემულია ორი n ციფრიანი რიცხვი $a_{n-1}...a_0$ და $b_{n-1}...b_0$, მისი ჯამი $d_n...d_0$ შემდეგი ალგორითმით შეიძლება გამოვიანგარიშოთ:

$c_0 = 0;$
for($i = 0, i < n, i++$*)*
{
  $d_i = a_i + b_i + c_i$ *mod* $10;$
  if($a_i + b_i + c_i > 9$ *)*
    $c_{i+1} = 1;$
    else $c_{i+1} = 0;$
}
$d_n = c_n;$

იმის დასამტკიცებლად, რომ მოყვანილი ალგორითმი მართლაც სწორ შედეგს მოგვცემს, საჭიროა შემდეგი მათემატიკური ფორმულა: $(x_nx_{n-1}...x_0) = 10^n \cdot x_n + 10^{n-1} \cdot x_{n-1} + ... + 10^0 \cdot x_0$.

<u>სავარჯიშო 4.8:</u> დაამტკიცეთ ტოლობა $(x_nx_{n-1}...x_0) = 10^n \cdot x_n + 10^{n-1} \cdot x_{n-1} + ... + 10^0 \cdot x_0$.

<u>სავარჯიშო 4.9:</u> წინა სავარჯიშოს შედეგის გამოყენებით დაამტკიცეთ ზემოთ მოყვანილი ალგორითმის სისწორე.

იგივე მეთოდით ორობითი როცხვების შეკრებაც შეგვიძლია:
მოცემულია ორი n ბიტიანი ორობითი რიცხვი $a = (a_{n-1}...a_0)_2$ და $b = (b_{n-1}...b_0)_2$. გამოიანგარიშეთ მისი ჯამი $(d_n...d_0)_2$:

$$\begin{array}{ccccc} + & a_{n-1} & ... & a_1 & a_0 \\ & b_{n-1} & ... & b_1 & b_0 \\ \hline d_n & d_{n-1} & ... & d_1 & d_0 \end{array}$$

$c_0 = 0;$
for($i = 0, i < n, i++$*)*
{
  $z_i = a_i + b_i + c_i$ *mod* $2;$
  if($a_i + b_i + c_i \geq 2$ *)*
    $c_{i+1} = 1;$
    else $c_{i+1} = 0;$
}
$d_n = c_n;$

<u>სავარჯიშო 4.10:</u> დაამტკიცეთ ტოლობა $(x_nx_{n-1}...x_0)_2 = 2^n \cdot x_n + 2^{n-1} \cdot x_{n-1} + ... + 2^0 \cdot x_0$.

<u>სავარჯიშო 4.11:</u> წინა სავარჯიშოს შედეგის გამოყენებით დაამტკიცეთ ზემოთ მოყვანილი ალგორითმის სისწორე.

აღსანიშნავია, რომ $c_{i+1} = 1$ მაშინ და მხოლოდ მაშინ, თუ a_i, b_i და c_i ცვლადებს შორის ორი ან სამი ერთის ტოლია. ამის განსაზღვრა შემდეგნაირად შეიძლება: თუ $a_i = b_i = 1$, მაშინ პირობა სრულდება. თუ ამ ორი ცვლადიდან ზუსტად ერთია ერთის ტოლი, მაშინ ამავდროულად მესამე ცვლადიც (ანუ c_i) უნდა იყოს 1. იმის დადგენა, არის თუ არა ორი ცვლადიდან ზუსტად ერთი ერთიანის ტოლი, შეიძლება ორის მოდულით მიმატებით: $a_i \oplus b_i$. აქედან გამომდინარე, იმის დადგენა, გვხვდება თუ არა ორი ან სამი ერთიანი სამ ცვლადში, შემდეგი ფორმულით შეიძლება: $c_{i+1} = a_ib_i \vee (a_i \oplus b_i)c_i$ (აღსანიშნავია, რომ $a_ib_i = 1$ მაშინ და მხოლოდ მაშინ, თუ ორივე ცვლადი არის 1).

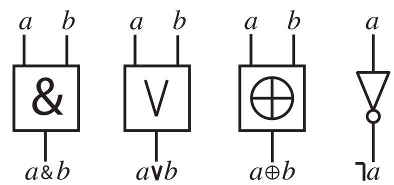
აქედან გამომდინარე z_i და c_i $(i = 1, ..., n-1)$ ცვლადების გამოსათვლელად გვაქვს შემდეგი ფორმულები:

$d_i = a_i \oplus b_i \oplus c_i,$
$c_{i+1} = a_i b_i \vee (x_i \oplus y_i) c_i,$
$d_n = c_n.$

მაგალითი:

	7	6	5	4	3	2	1	0
a	1	1	0	0	1	0	0	1
b	1	1	1	1	0	0	1	0
d	1	0	1	1	1	0	1	1
c	1	1	0	0	0	0	0	0

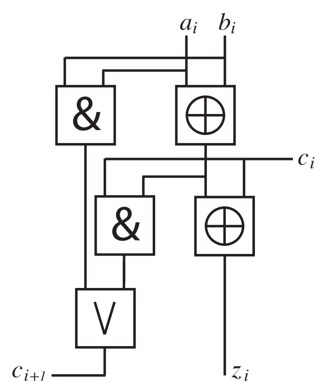
ზემოთ მოყვანილი ბულის ალგებრის ფორმულები გრაფიკულადაც შეიძლება გამოვსახოთ:



ნახ. 18: ლოგიკური ოპერაციების გრაფიკული გამოსახვა

კონიუნქციის, დიზიუნქციისა და უარყოფის ოპერაციებს ბულის ალგებრის ელემენტარულ ოპერაციებსაც უწოდებენ.

აქედან გამომდინარე, შეგვიძლია შევადგინოთ z_i და c_i ცვლადების გამოსათვლელი სქემა:

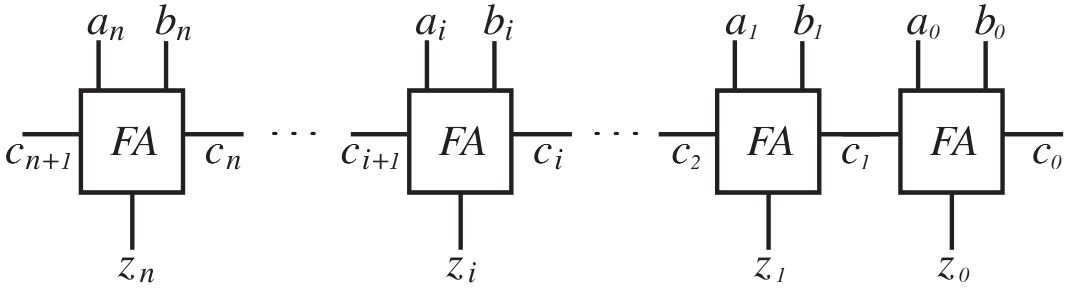


ნახ. 19: z_i და c_{i+1} ცვლადების გამოსათვლელი სქემა

აღსანიშნავია, რომ $\oplus$ ოპერაცია იმდენად ხშირად გამოიყენება, რომ მისი სქემა ერთი სიმბოლოთია აღნიშნული, თუმცა იგი რამოდენიმე ოპერაციის შესრულებას მოითხოვს.

სავარჯიშო 4.12: გამოიანგარიშეთ, რისი ტოლია $T(\oplus)$ და $C(\oplus)$.

თუ ჩვენ ამ სქემას აღვნიშნავთ როგორც FA (ინგლისური Full Adder, ანუ სრული შემკრები), მაშინ ორი n ბიტიანი რიცხვის შეკრებისათვის საჭირო სქემა (რომელსაც ვუწოდებთ CRA_n) შემდეგნაირი იქნება:



ნახ. 20: ორი n ბიტიანი რიცხვის შეკრებისათვის საჭირო სქემა CRA_n

სავარჯიშო 4.13: გამოიანგარიშეთ, რისი ტოლია $T(FA)$ (ანუ იმ ბიჯების რაოდენობა, რაც საჭიროა FA სქემის ყველა შედეგის გამოსაანგარიშებლად) და $C(FA)$ (ანუ FA სქემაში არსებული ელემენტების რაოდენობა), თუ $T(\&) = T(\vee) = T(\neg) = 1$, და $C(\&) = C(\vee) = C(\neg) = 1$. აქვე გამოიყენეთ წინა სავარჯიშოში გამოთვლილი $T(\oplus)$ და $C(\oplus)$.

შენიშვნა: ხშირად იღებენ $T(\neg) = 0$ და $C(\neg) = 0$, ანუ სქემებში უარყოფის ელემენტებს უგულებელყოფენ იმის გამო, რომ მათი რეალიზაცია სხვა ელემენტების რეალიზაციასთან შედარებით საკმაოდ მცირეა და, ამავე დროს, უარყოფებს ხშირად იყენებენ დამხმარე ელემენტებად (სხვადასხვა ტექნიკური მიზეზებით ორ ერთმანეთზე მიყოლებულ უარყოფას სვამენ ხოლმე). ამას გარდა, ტექნიკურად შესაძლებელია ელემენტების სქემის ისეთი რეალიზაცია, რომ $T(\neg(ab)) = T(ab)$, $T(\neg(a \vee b)) = T(a \vee b)$, $T(\neg ab) = T(ab)$, $T(\neg a \vee b) = T(\neg a \vee b)$.

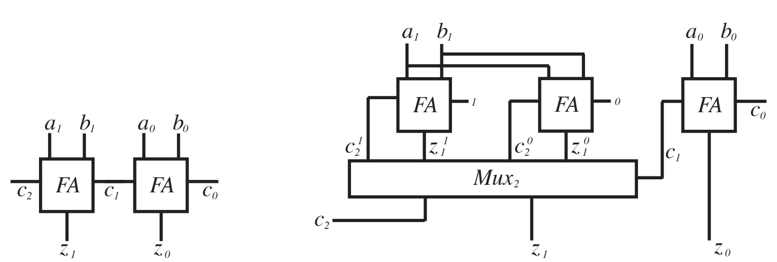
სავარჯიშო 4.14: გამოიანგარიშეთ, რისი ტოლია $T(\oplus)$, $C(\oplus)$ და, აქედან გამომდინარე, $T(FA)$ იმის გათვალისწინებით, რომ $T(\neg) = C(\neg) = 0$.

ადვილი დასანახია, რომ z_1 ცვლადი არ გამოითვლება, სანამ არ იქნება გამოთვლილი c_1 და, ზოგადად, z_i ცვლადის გამოთვლა არ შეიძლება, სანამ არ იქნება გამოთვლილი c_{i-1}. აქედან გამომდინარე, $T(CRA_n) = 4n$, $C(CRA_n) = 9n$ (აქ და შემდგომში დავუშვებთ, რომ $T(\neg) = C(\neg) = 0$).

სავარჯიშო 4.15: დაამტკიცეთ $T(CRA_n) = 4n$ და $C(CRA_n) = 9n$ ტოლობები.

სავარჯიშო 4.16: დახაზეთ CRA_1, CRA_2, CRA_3 და CRA_4 სქემები.

ბუნებრივია შემდეგი შეკითხვა: შესაძლებელია თუ არა ბიჯების რაოდენობისა და ელემენტების რიცხვის შემცირება?
თუ დავაკვირდებით ნახ. 21 *(ა)*-ში პირველ ორ FA ელემენტს, დავინახავთ შემდეგ მნიშვნელოვან ფაქტს: მარცხენა FA ელემენტი, რომელიც z_1 და c_1 ცვლადებს ითვლის, „ელოდება" c_0 ცვლადის გამოთვლას.



(ა) *(ბ)*

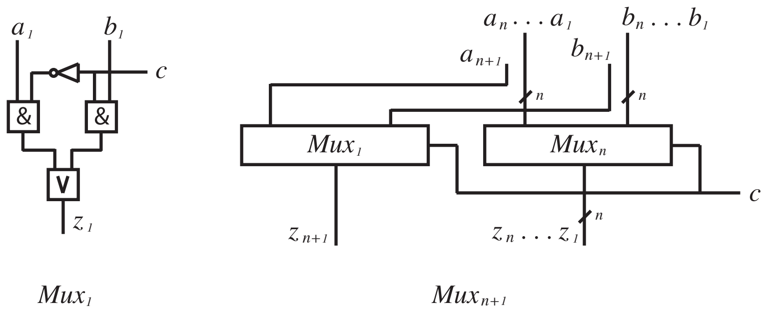
ნახ. 21: ორი n ბიტიანი რიცხვის მიმატების პარალელური სქემა

იმის გამო, რომ მას შეიძლება მიეწოდოს მხოლოდ $c_1 = 0$ ან $c_1 = 1$, ჩვენ შეგვიძლია ***ერთდროულად*** გამოვითვალოთ z_1 და c_2 იმ შემთხვევისათვის, როდესაც $c_1 = 0$ (z_1^0, c_2^0) და იმ შემთხვევისათვის, როდესაც $c_1 = 1$ (z_1^1, c_2^1).შემდეგ, როდესაც c_1 გამოთვლილი იქნება, შეიძლება ამ ორი საშუალედო შედეგიდან ერთ-ერთის არჩევა (ნახ. 21 *(ბ)*).

ამ ნახაზში გვხვდება ახალი ელემენტი Mux_2, რომლის მუშაობის შედეგები შემდეგი ცხრილით შეიძლება გამოისახოს:

$$z_1 = \begin{cases} z_1^0, & \text{თუ } c_1 = 0, \\ z_1^1, & \text{თუ } c_1 = 1 \end{cases} \qquad c_2 = \begin{cases} c_2^0, & \text{თუ } c_1 = 0, \\ c_2^1, & \text{თუ } c_1 = 1. \end{cases}$$

ზოგადად, Mux_n შემდეგნაირად შეიძლება აღიწეროს (ნახ. 22) :

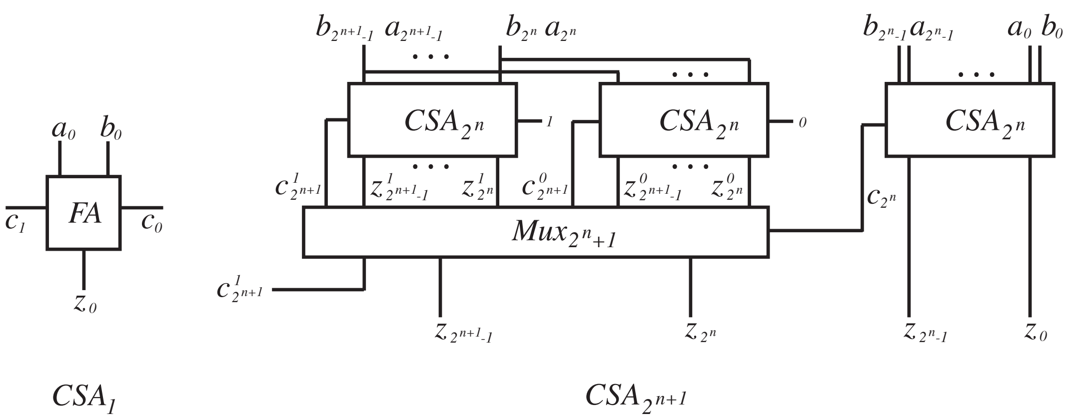


ნახ. 22: Mux_n – ორი n ბიტიანი რიცხვის ამორჩევის სქემა

$$z_i = \begin{cases} b_i, & \text{თუ } c = 0, \\ a_i, & \text{თუ } c = 1. \end{cases} \qquad i = \overline{1; n+1}.$$

ნახ. 21 -ში მოყვანილ სქემას უწოდებენ CSA_2 (Carry Select Adder – Carry bit დახსომებულ ბიტს ეწოდება, Select - შერჩევა, Adder -შემკრები). იგი ორ 2 ბიტიან რიცხვს $a = (a_1, a_0)$ და $b = (b_1, b_0)$ შეკრებს და 3 ბიტიან რიცხვს $z = (c_2, z_1, z_0)$ მოგვცემს პასუხად.

ზოგადად, $CSA_{2^{n+1}}$, რომელიც ორ 2^{n+1} ბიტიან რიცხვს $A = (a_{2^n-1}, ..., a_0)$ და $B = (b_{2^n-1}, ..., b_0)$ შეკრებს და $2^{n+1} + 1$ ბიტიან რიცხვს $z = (c_{2^n}, z_{2^n-1}, ..., z_0)$ მოგვცემს პასუხად, შემდეგნაირად აღიწერება (ნახ. 23):



ნახ. 23: $CSA_{2^{n+1}}$ – ორი 2^{n+1} ბიტიანი რიცხვის შეკრების პარალელური სქემა

CSA_1, ანუ ორი ერთბიტიანი რიცხვის შემკრები არის ზემოთ განხილული სქემა FA.
ძირითადი იდეა „დაყავი და იბატონე" პარადიგმაზეა აგებული: მონაცემები ორ ნაწილად იყოფა —

$$A_1 = (a_{2^{n+1}-1}...a_{2^n}) \quad A_0 = (a_{2^n-1}...a_0)$$
$$B_1 = (b_{2^{n+1}-1}...b_{2^n}) \quad B_0 = (b_{2^n-1}...b_0)$$

შემდეგ გამოითვლება $Z_0 = A_0 + B_0$ და ***ამავდროულად*** $Z_1^0 = A_1 + B_1 + 0$ და $Z_1^1 = A_1 + B_1 + 1$. ამის შემდეგ, c_{2^n} სიგნალის მეშვეობით, ამოირჩევა

$$Z_1 = \begin{cases} Z_1^0, & \text{თუ } c_{2^n} = 0, \\ Z_1^1, & \text{თუ } c_{2^n} = 1 \end{cases} \qquad c_{2^{n+1}} = \begin{cases} c_{2^{n+1}}^0, & \text{თუ } c_{2^n} = 0, \\ c_{2^{n+1}}^1, & \text{თუ } c_{2^n} = 1 \end{cases}$$

აქ $Z_0 = (z_{2^n-1}, ..., z_0)$ და $Z_1 = (z_{2^{n+1}-1}, ..., z_{2^n})$.

ადვილი დასამტკიცებელია ამ სქემის სისწორე მათემატიკურ ინდუქციაზე დაყრდნობით:

- ინდუქციის შემოწმება: თუ $n = 0$, ცხადია, რომ $CSA_1 = FA$ და იგი ორ ერთ ბიტიან რიცხვს სწორად შეკრებს;
- ინდუქციის დაშვება: დავუშვათ, CSA_{2^n} სწორად შეკრებს ორ 2^n ბიტიან რიცხვს;
- ინდუქციის ბიჯი: დავამტკიცოთ, რომ $CSA_{2^{n+1}}$ სწორად შეკრებს ორ 2^{n+1} ბიტიან რიცხვს.

სავარჯიშო 4.17: დაამტკიცეთ, რომ თუ CSA_{2^n} სწორად შეკრებს ორ 2^n ბიტიან რიცხვს, მაშინ $CSA_{2^{n+1}}$ სწორად შეკრებს ორ 2^{n+1} ბიტიან რიცხვს.

რაც შეეხება ამ ალგორითმის ბიჯების რაოდენობას $T(CSA_{2^{n+1}})$, მისი გამოთვლა შემდეგნაირად შეიძლება: უპირველესად ყოვლისა, უნდა გამოვითვალოთ ცვლადები Z_0, Z_1^0 და Z_1^1, რაც ***ერთდროულად*** შეიძლება მოხდეს $T(CSA_{2^n})$ ბიჯში. ამის შემდეგ უნდა ავირჩიოთ Z_1^0 და Z_1^1 ცვლადებიდან ერთ-ერთი Mux_{2^n+1} სქემის საშუალებით, რაც $T(Mux_{2^n+1}) = 2$ ბიჯშია შესაძლებელი.
აქედან გამომდინარე, $T(CSA_{2^{n+1}}) = T(CSA_{2^n}) + T(Mux_{2^n+1}) = T(CSA_{2^n}) + 2$. ამ რეკურსიული ფორმულის გახსნის შემდეგ მივიღებთ:

$$T(CSA_{2^{n+1}}) = O(\log n).$$

სავარჯიშო 4.18: დაამტკიცეთ ტოლობა $T(Mux_{2^n+1}) = 2$ (გამოიყენეთ ნახ. 22-ში მოყვანილი რეკურსიული სქემა).

$C(CSA_{2^{n+1}})$ ოპერაციათა რაოდენობის გამოსათვლელად გამოვიყენოთ ფორმულა:

$$C(CSA_{2^{n+1}}) = 3 \cdot C(CSA_{2^n}) + C(Mux_{2^n+1}).$$

სავარჯიშო 4.19: დაამტკიცეთ, რომ $C(CSA_{2^{n+1}})$ მართლაც ამ რეკურსიული ფორმულით გამოითვლება და გამოითვალეთ მისი მნიშვნელობა.

როგორც ვხედავთ, პარალელური ალგორითმებით შეიძლება შეკრების ამოცანის სწრაფად გადაჭრა: ორი n ბიტიანი რიცხვისათვის არა $O(n)$, არამედ $O(\log n)$ ბიჯია საჭირო, სამაგიეროდ იზრდება ელემენტების რაოდენობა. ეს გასაკვირი არ არის: მეტ ოპერაციას ვატარებთ, ოღონდ ერთდროულად და ამის ხარჯზე ვიგებთ დროს.
აქვე უნდა აღინიშნოს, რომ არსებობს ორი n ბიტიანი რიცხვის მიმატების პარალელური ალგორითმი, რომლის დროის ზედა ზღვარია $O(\log n)$ და ელემენტების რაოდენობის ზედა ზღვარია $O(n)$ (სხვა სიტყვებით რომ ვთქვათ, შესაძლებელია ლოგარითმულ დროში გამოთვლა ისე, რომ ელემენტების რაოდენობა ძალიან არ გაიზარდოს), მაგრამ მათი განხილვა ჩვენი კურსის პროგრამას ცდება.

4.3 n ბიტიანი რიცხვების გამოკლება

წინა პარაგრაფში განხილული ალგორითმებით ფიქსირებული $n \in \mathbb{N}$ ბიტიანი სისტემების აგება შეიძლება. როდესაც აწყობილია სისტემა ფიქსირებული n ბიტიანი ორობითი რიცხვების დასამუშავებლად, მაქსიმალური რიცხვი, რაც შეიძლება წარმოვადგინოთ, იქნება $2^n - 1$: $(11...1)_2$. მასზე ერთით მეტი რიცხვი უკვე $n+1$ ბიტიანი იქნება: $(100...0)$, სადაც მარჯვენა n ბიტი ნულის ტოლია, ანუ თუ ჩვენ მხოლოდ n ბიტიან რიცხვებს განვიხილავთ და უფრო მაღალ ბიტებს უბრალოდ ვაგდებთ, ნებისმიერ არითმეტიკულ ოპერაციას 2^n მოდულით არითმეტიკაში ვატარებთ, რაც იმას ნიშნავს, რომ ნებისმიერი $0 \leq x < 2^n$ რიცხვისათვის შეგვიძლია გამოვიანგარიშოთ ისეთი შესაბამისი y, რომ $x + y = 0 \ mod \ 2^n$, ან, სხვა სიტყვებით რომ ვთქვათ, x რიცხვის შებრუნებული (უარყოფითი) მოდულით 2^n.
ბუნებრივია შეკითხვა: როგორ შეიძლება გამოვიანგარიშოთ მოცემული x რიცხვის შებრუნებული y? თუ განვიხილავთ რიცხვს $0 \ mod \ 2^n = 2^n = (11...1)_2 + 1_2 \ mod \ 2^n$, შეიძლება დავასკვნათ, რომ x რიცხვის შებრუნებულის გამოსათვლელად უნდა გამოვიანგარიშოთ ისეთი z რიცხვი, რომ $x + z = (11...1)_2$ და შემდეგ $y = z + 1$.

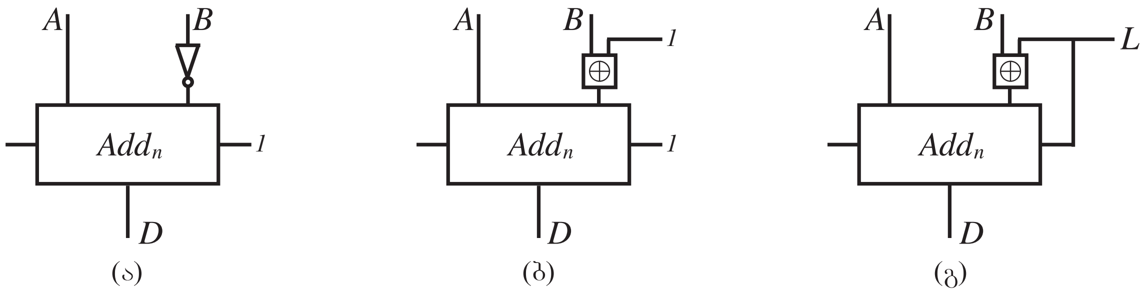
სავარჯიშო 4.20: დაამტკიცეთ, რომ თუ მოცემული n ბიტიანი x რიცხვისთვის მოვძებნით ისეთ z რიცხვს, რომ $x + z = (11...1)_2$, მაშინ x რიცხვის შებრუნებული (მოდულით 2^n) იქნება $y = z + 1$.

ცხადია, თუ ავიღებთ $x = (x_{n-1}x_{n-2}...x_0)_2$, მაშინ $z = \overline{x} = (\overline{x}_{n-1}\overline{x}_{n-2}...\overline{x}_0)_2$.

სავარჯიშო 4.21: დაამტკიცეთ, რომ მოცემული $x = (x_{n-1}x_{n-2}...x_0)_2$ და $z = (\overline{x}_{n-1}\overline{x}_{n-2}...\overline{x}_0)_2$ რიცხვებისათვის ჭეშმარიტია ტოლობა $x + z = 0 \ mod \ 2^n$.

აქედან გამომდინარე, $x = (x_{n-1}x_{n-2}...x_0)_2$ რიცხვის შებრუნებულია $y = \overline{x}+1 \ mod \ 2^n = (\overline{x}_{n-1}\overline{x}_{n-2}...\overline{x}_0)_2+1 \ mod \ 2^n$.

ყოველივე ზემოთ თქმულიდან შეიძლება დავასკვნათ, რომ მოცემული $a = (a_{n-1}a_{n-2}...a_0)_2$ და $b = (b_{n-1}b_{n-2}...b_0)_2$ რიცხვის სხვაობის გამოსათვლელად უნდა გამოვითვალოთ შემდეგი ჯამი: $d = a - b \ mod \ 2^n = a + \overline{b} + 1 \ mod \ 2^n$. ეს კი ნახ. 24(ა)-ში ნაჩვენებ სქემას მოგვცემს.



ნახ. 24: ორი n ბიტიანი რიცხვის გამოკლების სქემა (ა), (ბ) და მიმატება-გამოკლების კომბინირებული სქემა (გ)

აქ Add_n ორი n ბიტიანი რიცხვის მიმატების სქემაა, რომლის კონკრეტული რეალიზაცია ამ შემთხვევაში მნიშვნელოვანი არაა.

სავარჯიშო 4.22: დაამტკიცეთ, რომ ნახ. 24(ა)-ში ნაჩვენებ სქემა მართლაც A და B რიცხვების სხვაობას გამოითვლის.

რადგან $\neg x = x \oplus 1$, 24(ა) და 24(ბ) ნახაზებში ნაჩვენები სქემები ერთსა და იგივე შედეგს იძლევა. აქედან გამომდინარე, შესაძლებელია 24(გ) ნახაზში ნაჩვენები სქემით შეკრებისა და გამოკლების ერთიანი სქემის შექმნა: თუ მაკონტროლებელი სიგნალი $L = 1$, შესრულდება გამოკლება, ხოლო თუ $L = 0$ — მიმატება.

სავარჯიშო 4.23: დაამტკიცეთ, რომ $\neg x = x \oplus 1$ და $x = x \oplus 0$.

სავარჯიშო 4.24: დაამტკიცეთ, რომ თუ 24(გ) ნახაზში ნაჩვენებ სქემაში მაკონტროლებელი სიგნალი $L = 1$, შესრულდება გამოკლება, ხოლო თუ $L = 0$ — მიმატება.

აღსანიშნავია, რომ რეალურ სისტემებში შედეგი $D = (d_n, d_{n-1}...d_0)_2$ (და საერთოდ რიცხვები) $n + 1$ ბიტიანი სიტყვებია, რომლებშიც უფროსი ბიტი d_n ნიშანს გვიჩვენებს: თუ $d_n = 1$, D რიცხვი უარყოფითია, წინააღმდეგ შემთხვევაში კი დადებითი. მაგრამ ჩვენ 2^n მოდულით არითმეტიკული ოპერაციებით შემოვიფარგლებით (ნიშნის გარეშე), რითიც უფრო ადვილია ძირითადი პრინციპების გაგება და შემდგომ სხვადასხვა პრაქტიკული რეალიზაციის ათვისება.

4.4 n ბიტიანი რიცხვების გამრავლება

„ქვეშ მიწერით გამრავლება" პირველი ალგორითმია, რომელსაც სკოლაში ვსწავლობთ:

$$
\begin{array}{r} \times \quad 427 \\ 613 \\ \hline 1281 \end{array}
\qquad
\begin{array}{r} \times \quad 427 \\ 613 \\ \hline 1281 \\ 427 \end{array}
\qquad
\begin{array}{r} \times \quad 427 \\ 613 \\ \hline 1281 \\ 427 \\ 2562 \end{array}
\qquad
\begin{array}{r} \times \quad 427 \\ 613 \\ \hline 1281 \\ 427 \\ 2562 \\ \hline 261751 \end{array}
$$

$A = (a_{n-1}...a_0)$ და $B = (b_{n-1}...b_0)$ რიცხვების გადასამრავლებლად გამოიანგარიშე $C_k = 10^k \cdot A \cdot b_k$ და მათი ჯამი $C = C_0 + C_1 + \cdots + C_{n-1}$. აღსანიშნავია, რომ $A \cdot b_k$ რიცხვის გამოსათვლელად ცალკე ალგორითმია საჭირო, ხოლო $10^k x$ მოცემული x რიცხვის k პოზიციით მარცხნივ „ჩაჩოჩებას" ნიშნავს (ან მარჯვნივ შესაბამისი რაოდენობის ნულების მიწერას).

სავარჯიშო 4.25: დაამტკიცეთ, რომ ზემოთ მოყვანილი ქვეშ მიწერით გამრავლების მეთოდი მართლაც სწორ პასუხს იძლევა.
მინიშნება: $B = (b_{n-1}...b_0)$ რიცხვი წარმოადგინეთ შემდეგი ჯამის სახით: $B = 10^{n-1} \cdot b_{n-1} + \cdots + 10^0 \cdot b_0$.

ანალოგიურად შეგვიძლია გამოვიანგარიშოთ ორობითში წარმოდგენილი რიცხვების ნამრავლიც:

$$
\begin{array}{r} \times \quad 10110 \\ 10011 \\ \hline 10110 \\ 10110 \\ 00000 \\ 00000 \\ 10110 \\ \hline 110100010 \end{array}
$$

ცხადია, რომ $A = (a_{n-1}...a_0)$ და $B = (b_{n-1}...b_0)$ ორობითი რიცხვის გამრავლების შემთხვევაში შემდეგნაირად უნდა მოვიქცეთ: გამოვიანგარიშოთ $C_k = 2^k \cdot A \cdot b_k = 2^k \cdot A \,\&\, b_k$ და მათი ჯამი $C = C_0 + C_1 + \cdots + C_{n-1}$.

სავარჯიშო 4.26: ათობითი რიცხვების ალგორითმის ანალოგიურად დაამტკიცეთ ამ მეთოდის სისწორე.

სავარჯიშო 4.27: დაამტკიცეთ, რომ ქვეშ მიწერით გამრავლების მეთოდის ოპერაციათა რაოდენობის ზედა ზღვარია $O(n^2)$. რა არის მისი ბიჯების რაოდენობის ზედა ზღვარი? შეიძლება თუ არა ამ მეთოდის პარალელიზაცია?

სავარჯიშო 4.28: განიხილეთ ათობითში ჩაწერილი n ბიტიანი რიცხვების ქვეშ მიწერით გამრავლების ალგორითმი $MultDec_n$ და ანალოგიური ალგორითმი $MultBin_n$, რომელიც ორობითში ჩაწერილ n ბიტიან რიცხვებს ამრავლებს. რა განსხვავებაა $C(MultDec_n)$ და $C(MultBin_n)$ ზედა ზღვრებს შორის? $T(MultDec_n)$ და $T(MultBin_n)$ ზედა ზღვრებს შორის? პასუხი დაამტკიცეთ.

4.4.1 გამრავლების პარალელური მეთოდი: ვოლესის ხე (Wallace Tree)

1964 წელს ავსტრალიელმა მეცნიერმა კრის ვოლესმა (Chris Wallace) გამრავლების პარალელიზაციის იდეა წამოაყენა, რომელსაც ჩვენს მაგალითზე განვიხილავთ.

C_i ცვლადები გამოვიანგარიშოთ როგორც ქვეშ მიწერით მეთოდში:

$$
\begin{array}{rr}
 & \times \quad 10110 \\
 & 10011 \\ \hline
C_0 & 10110 \\
C_1 & 10110 \\
C_2 & 00000 \\
C_3 & 00000 \\
C_4 & 10110 \\ \hline
C & 110100010 \\
\text{დახსომებული} & 001111100
\end{array}
\qquad
\begin{array}{rr}
 & \times \quad a_4\; a_3\; a_2\; a_1\; a_0 \\
 & b_4\; b_3\; b_2\; b_1\; b_0 \\ \hline
C_0 & c_{0,4}c_{0,3}c_{0,2}c_{0,1}c_{0,0} \\
C_1 & c_{1,4}c_{1,3}c_{1,2}c_{1,1}c_{1,0}\phantom{c_{0,0}} \\
C_2 & c_{2,4}c_{2,3}c_{2,2}c_{2,1}c_{2,0}\phantom{c_{0,0}c_{0,0}} \\
C_3 & c_{3,4}c_{3,3}c_{3,2}c_{3,1}c_{3,0}\phantom{c_{0,0}c_{0,0}c_{0,0}} \\
C_4 & c_{4,4}c_{4,3}c_{4,2}c_{4,1}c_{4,0}\phantom{c_{0,0}c_{0,0}c_{0,0}c_{0,0}} \\ \hline
C & c_9\;\; c_8\;\; c_7\;\; c_6\;\; c_5\;\; c_4\;\; c_3\;\; c_2\;\; c_1\;\; c_0
\end{array}
$$

ცვლადებისათვის შემოვიტანოთ აღნიშვნა $C_i = (c_{i,4}c_{i,3}c_{i,2}c_{i,1}c_{i,0})_2$ და ყოველ ასეთ $c_{i,j}$ ცვლადს ვუწოდოთ 2^{i+j} რიგის. აქედან გამომდინარე, გვექნება 1 ცალი $2^0 = 1$ რიგის, 2 ცალი 2^1 რიგის, სამი 2^2 რიგის, ოთხი 2^3 რიგის, ხუთი 2^4 რიგის, ისევე ოთხი 2^5 რიგის, სამი 2^6 რიგის, ორი 2^7 რიგის და ერთი 2^8 რიგის ცვლადი.

$c_{0,0}$ ცვლადი პირდაპირ უნდა გადავიდეს, როგორც საბოლოო პასუხის ყველაზე დაბალი ბიტი: $c_0 = c_{0,0}$ (2^0 რიგის ცვლადი მხოლოდ ერთია, ასე რომ, მას არაფერი ემატება).

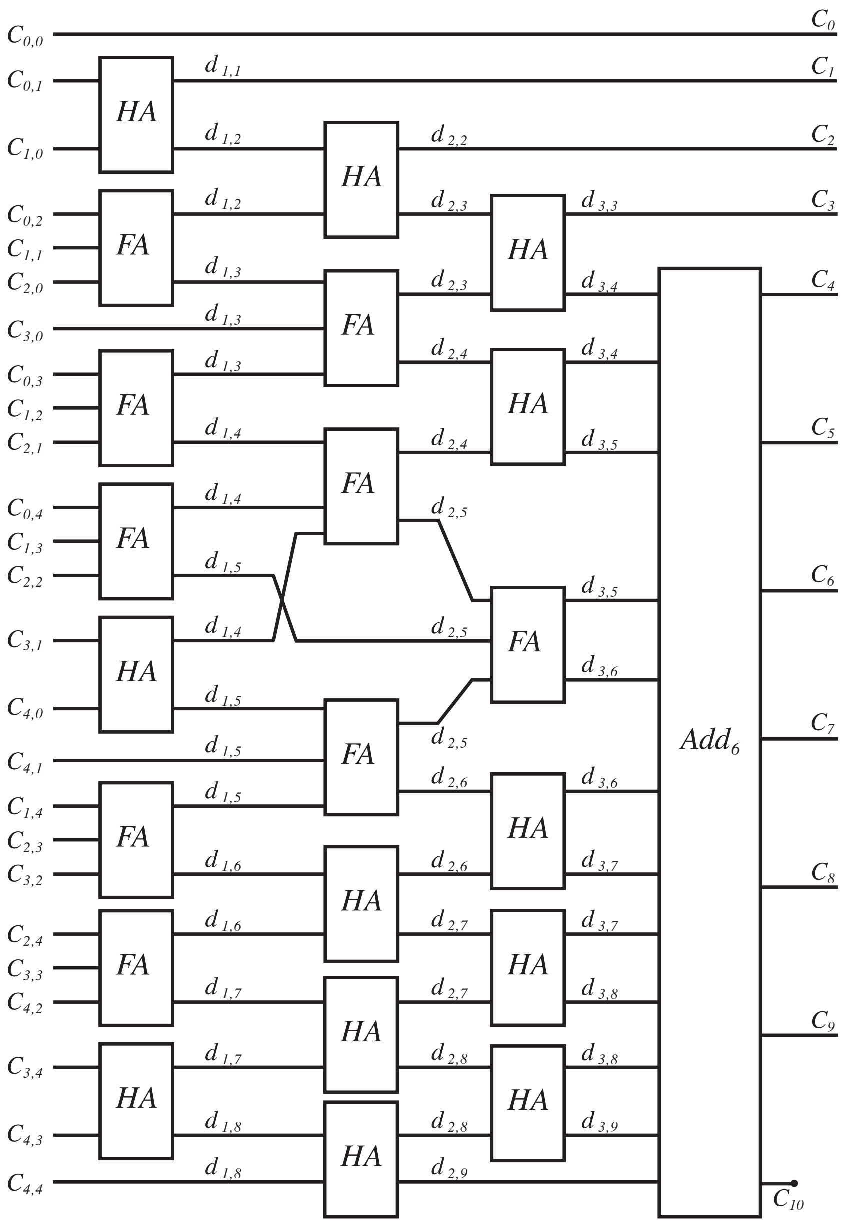
2^1 რიგის ცვლადები უნდა შევკრიბოთ, შედეგად ვიღებთ ერთ 2^1 რიგის პასუხს (მათ ორობით ჯამს) და ერთ 2^2 რიგის პასუხს (დახსომებულ ბიტს, რომელიც შემდეგში უფრო მაღალი ბიტების ჯამს დაემატება).

ანალოგიურად ვაჯამებთ 2^2 რიგის სამ ცვლადს, პასუხად ვიღებთ ერთ 2^2 რიგის და ერთ 2^3 რიგის ბიტს.

ამ წესით ვაჯამებთ 2^i რიგის ცვლადებს (ორს ან სამს ერთად) და შედეგად ვიღებთ 2^i და 2^{i+1} რიგის ბიტებს, რომლებიც შემდეგ ბიჯში უნდა დავაჯგუფოთ და იგივე წესით ავჯამოთ.

ამ პროცესს ვიმეორებთ მან ამ, სანამ არ მივიღებთ ყოველი რიგში ორ ან ერთ ბიტს. ბოლოს ჩვეულებრივი შემკრებით ვაჯამებთ იმ ნაწილს, რომელიც ყოველი რიგის ორ-ორი ბიტისაგან შედგება.

ყოველივე ეს სქემატურად ნაჩვენებია ნახაზში 25.



ნახ. 25: ორი 5 ბიტიანი რიცხვის გამრავლების ვოლესის ხის შემკრები ნაწილი

აღსანიშნავია, რომ *HA* ორი ბიტის შემკრები სქემაა, რომელიც a და b ერთ ბიტიანი რიცხვების ორობით ჯამს და დახსომებულ ბიტს გამოითვლის. ფაქტიურად ეს იგივე *FA* სქემაა, სადაც $C = 0$.

სავარჯიშო 4.29: FA სქემის გამოყენებით დახაზეთ HA სქემა.

ზემოთ მოყვანილ მეთოდს ვოლესის ხე ეწოდება, რადგან მის სქემას ხის სტრუქტურა აქვს: ყოველ შრეში შესაკრებთა რაოდენობა იკლებს. უხეშად რომ დავითვალოთ, სამი შესაკრები ორზე დადის.
მისი ძირითადი იდეაც ესაა: HA სქემის გამოყენებით სამი შესაკრები a, b, c ორ ისეთ შესაკრებზე x, y დავიყვანოთ, რომ $a + b + c = x + 2y$.

სავარჯიშო 4.30: მაქსიმუმ რამდენ ბიტიანი რიცხვი შეიძლება მივიღოთ ორი n ბიტიანი რიცხვის გამრავლების შედეგად?

4.4.2 კარაცუბა-ოფმანის გამრავლების მეთოდი

1960ან წლებში მოსკოვში მომუშავე მათემატიკოსებმა ანატოლი კარაცუბამ და იური ოფმანმა გამრავლების მეთოდი შეიმუშავეს, რომელმაც ოპერაციათა რაოდენობის ზედა ზღვარი $O(n^2)$-ზე უკეთესია, რითაც მნიშვნელოვანი ნაბიჯი გადადგეს ეფექტური ალგორითმების შემუშავების თვალსაზრისით.

ძირითადი იდეა მარტივია: თუ მოცემულია ორი n ბიტიანი რიცხვი $A = (a_{n-1}...a_0)_2$, $B = (b_{n-1}...b_0)_2$ და ვეძებთ $C = (c_{2n-1}...c_0)_2 = A \cdot B$, ჯერ მონაცემები ორ ტოლ ნაწილა დავყოთ: $A = (A_1A_0)_2$, $B = (B_1B_0)_2$, სადაც $A_1 = (a_{n-1}...a_{\frac{n}{2}})_2$, $A_0 = (a_{\frac{n}{2}-1}...a_0)_2$, $B_1 = (b_{n-1}...b_{\frac{n}{2}})_2$, $B_0 = (b_{\frac{n}{2}-1}...b_0)_2$.
მივიღებთ $A = 2^{\frac{n}{2}}A_1 + A_0$, $B = 2^{\frac{n}{2}}B_1 + B_0$ და

$$\begin{aligned} A \cdot B &= (2^{\frac{n}{2}}A_1 + A_0)(2^{\frac{n}{2}}B_1 + B_0) = \\ &= 2^n \cdot A_1B_1 + 2^{\frac{n}{2}}(A_0B_1 + A_1B_0) + A_0B_0. \end{aligned}$$

როგორც ვხედავთ, ჩასატარებელია 4 გამრავლება, ოღონდ უკვე $\frac{n}{2}$ ბიტიანი რიცხვების. თუ გამრავლების ალგორითმს აღვნიშნავთ როგორც $Mult_n$ და მას რეკურსიულად გამოვიყენებთ, მივიღებთ ელემენტების რაოდენობის შემდეგ შეფასებას:

$$C(Mult_n) = 4 \cdot C(Mult_{\frac{n}{2}}) + 3 \cdot C(Add_{\frac{n}{2}}) \in O(4^{\log n}) = O(n^{\log 4}) = O(n^2)$$

და, როგორც ვხედავთ, ელემენტების რაოდენობას *ვერ* ვამცირებთ. ეს კი იმითაა გამოწვეული, რომ ზემოთ მოყვანილ გამოსახულებაში 4 გამრავლება გვხვდება. თუ რამენაირად მათ შემცირებას მოვახერხებთ, ელემენტების რაოდენობის ზედა ზღვარს კვადრატულზე უკეთესს გავხდით.

სავარჯიშო 4.31: დაამტკიცეთ, რომ $C(Mult_n) = 4 \cdot C(Mult_{\frac{n}{2}}) + 3 \cdot C(Add_{\frac{n}{2}}) \in O(4^{\log n})$.
კარაცუბამ და ოფმანმა შემდეგი ძირითადი იდეა წამოაყენეს: ზედა გამრავლების ფორმულაში $A_1 \cdot B_1$ და $A_0 \cdot B_0$ ნამრავლს გვერდს ვერ ავუვლით. ამიტომ ფრჩხილებში მოცემულ გამოსახულება უნდა შევცვალოთ ისეთით, რომელიც ერთ ახალ გამრავლებასა და $A_1 \cdot B_1$ და $A_0 \cdot B_0$ ელემენტებს შეიცავს.
მისი გამოთვლა შემდეგნაირად შეიძლება:
$A_0B_1 + A_1B_0 = X - A_1 \cdot B_1 - A_0 \cdot B_0$. აქედან გამომდინარე,

$$X = A_0 \cdot B_1 + A_1 \cdot B_0 + A_1 \cdot B_1 + A_0 \cdot B_0 = A_0(B_0 + B_1) + A_1(B_0 + B_1) = (A_1 + A_0) \cdot (B_1 + B_0).$$

საბოლოოდ ვიღებთ ნამრავლის ფორმულას:

$$A \cdot B = 2^n \cdot A_1 \cdot B_1 + 2^{\frac{n}{2}}((A_1 + A_0) \cdot (B_1 + B_0) - A_1 \cdot B_1 - A_0 \cdot B_0) + A_0 \cdot B_0.$$

ერთი შეხედვით გამოსახულება უფრო გართულდა, მაგრამ იმის გამო, რომ $A_1 \cdot B_1$ და $A_0 \cdot B_0$ ერთხელ უკვე გამოვითვალეთ, ფრჩხილებში მყოფ გამოსახულებაში მისი ახლად გამოთვლა საჭირო აღარაა, რადგან აქ მისი ადრე გამოთვლილი მნიშვნელობის გამოყენებაა შესაძლებელი.

საბოლოდ ვიღებთ ელემენტთა რაოდენობის შემდეგ შეფასებას:

$$C(Mult_n) = 3 \cdot C(Mult_{\frac{n}{2}}) + 4 \cdot C(Add_{\frac{n}{2}}) + 2 \cdot C(Sub_{\frac{n}{2}})$$

რადგან $C(Add_k) = C(Sub_k) = const \cdot k$ (როგორც ვნახეთ, ეს ორი ოპერაცია ერთი და იგივე სქემით შეგვიძლია ჩავატაროთ), ვიღებთ:

$$C(Mult_n) = 3 \cdot C(Mult_{\frac{n}{2}}) + 6 \cdot C(Add_{\frac{n}{2}}) = 3 \cdot C(Mult_{\frac{n}{2}}) + const \cdot n \in O(3^{\log n}) = O(n^{\log 3}).$$

ამით დამტკიცდა, რომ შესაძლებელია გამრავლების ოპერაციის კვადრატულზე უკეთეს ელემენტების რაოდენობით ჩატარება, რაც თავის დროზე ძალიან მნიშვნელოვანი შედეგი იყო.

<u>სავარჯიშო 4.32:</u> დაამტკიცეთ, რომ $3 \cdot C(Mult_{\frac{n}{2}}) + const \cdot n \in O(n^{\log 3})$.