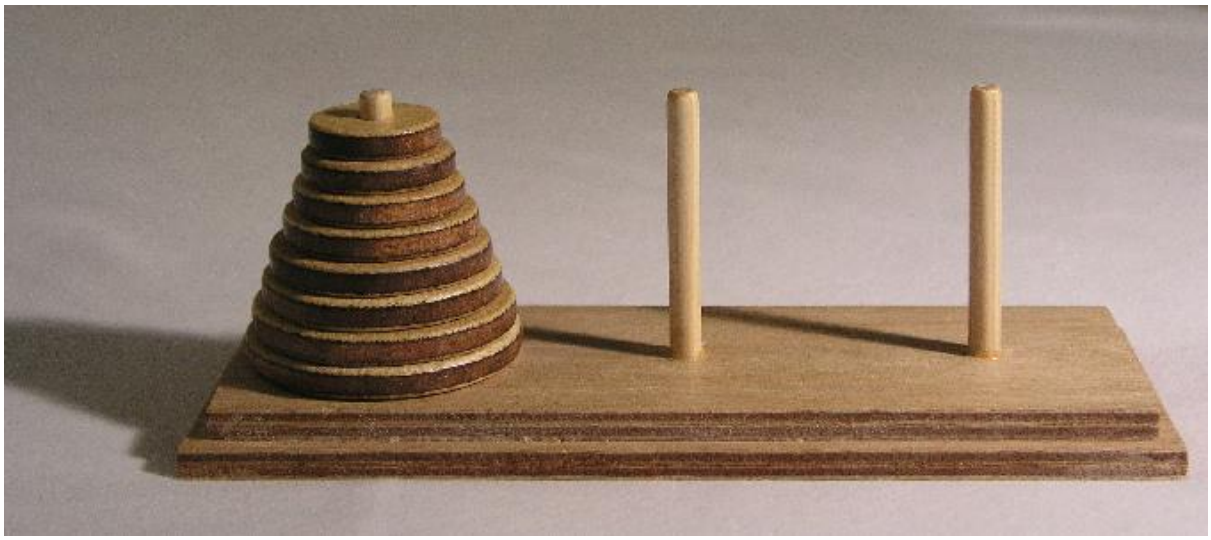
ჰანოის კოშკები და რეკურსია

 რეკურსიული ფუნქცია ეწოდება ისეთ ფუნქციას რომელიც იძახებს საკუთარ თავს, მისი მთავარი იარაღია ამოცანის მარტივ ნაწილებად დაშლა ანუ დეკომპოზიცია. რეკურსია მოქმედებს პრინციპით "დაყავი და იბატონე". რეკურსიული ფუნქციის დასრულება ხდება ამოცანის საბაზო ანუ ყველაზე მარტივ დონემდე დაყვანით. ამოცანის რეკურსიულად გადაწყვეტა მოითხოვს გაცილებით დიდ რესურსებს კომპიუტერისგან, ვიდრე ამას საჭიროებს ტრადიციული(იტერაციული) მიდგომა, თუმცა უნდა აღინიშნოს რომ ზოგიერთი ამოცანის გადაწყვეტისას რეკურსიული მიდგომა გაცილებით უკეთესია , ვიდრე იტერაციული. ასეთი ამოცანების რიგს მიეკუთვნება ამოცანა ჰანოის კოშკების შესახებ, რომლის იტერაციული გადაწყვეტა ძალიან დიდ სირთულეს წარმოადგენს, როდესაც რეკურსიით ყველაფერი მარტივად და გასაგებად წყდება.

არსებობს ლეგენდა, რომ შორეული აღმოსავლეთის ერთ-ერთ მონასტერში ბერები ცდილობდნენ გადაეტანათ ფირფიტები ერთი ღერძიდან მეორეზე, ფირფიტები ღერძზე ჩამოცმული იყო ისე, რომ მათი ზომა მცირდებოდა ქვემოდან ზემოთ. ფირფიტები გადატანა ხდებოდა შემდეგი წესებით:

1. ერთ გადატანაზე შეიძლება გადაიტანო მხოლოდ ერთი ფირფიტა
2. დიდი ზომის ფირფიტა არასდროს არ უნდა აღმოჩნდეს პატარა ფირფიტის ზემოდან
3. შეგვიძლია გამოვიყენოთ მხოლოდ ერთი დამხმარე ღერძი



ბერები ცდილობდნენ ეს პროცესი შეესრულებინათ 64 ფირფიტისთვის :) ლეგენდა ასევე ამბობს, რომ როდესაც ისინი ამ პროცესს დაასრულებენ (ამ ყველაფერს ისინი კომპიუტერის გარეშე აკეთებენ) ქვეყნიერების დასასრული დადგება ..

ეხლა, რაც შეეხება ამოცანის ალგორითმს, იგი ამგვარად გამოიყურება:

დავარქვათ პირველ ღერძს-first დამხმარეს-temp და იმ ღერძს, რომელზეც უნდა გადავიტანოთ-last; ვთქვათ სულ გვაქვს N რაოდენობის ფირფიტა. მაშინ N ცალის ფირფიტის გადატანის ოპერაციას დავარქვათ Hanoy (N).თუ ჩვენ ვიცით N-1 რაოდენობის ფირფიტა როგორ გადავიტანოთ first-დან temp-ზე, მაშინ დარჩენილ 1 ფირფიტას გადავიტანთ first-დან last-ზე და საბოლოოდ temp-დან N-1 ფირფიტას გადავიტანთ last-ზე;

თუ ამ ალგორითმს დავწერთ რეკურსიული ფუნქციით მაშინ მისი ფსევდოკოდი გამოიყურება ასე მარტივად:

```
Hanoy(n, first, last, temp)
{
    if( n == 1 )
    {
        cout<<first<<” -> “<<last<<endl;
return;
//იბეჭდება,საიდან სად უნდა გადავიტანოთ ფირფიტა. მაგალითად:1 -> 3
    }
        Hanoy(disc – 1, first, temp, last);
        Hanoy(  1, first,last,temp);
        Hanoy(n– 1, temp, last, first);
}
```

ეხლა კი წარმოგიდგენთ ჩემი პროგრამის შიგთავსს

```
#include <iostream>     // input/output ოპერაციებისთვის
#include <ctime>      // დროის ათვლისთვის
#include <vector>     //ვექტორის კლასით აღვწერ ფირფიტების სიგრძის
პარამეტრები
using namespace std;  // კლავიატურიდან ფირფიტების რაოდენობის
შესატანად
```

გამოვიყენე 2 სახის სტრუქტურა:

// ღერძების დასახელებების უცვლელი სახით შესანახად. მისი იმპლემენტაცია თავად გავაკეთე ფუნქცია დრაივერში სახელებით ‘F’, ‘T’, ‘L’.

```
struct hanoy_name   {
      char first, last, temp;
};
```

ეს სტრუქტურა გამოვიყენე ფირფიტების რაოდენობისა და ღერძების სახელების დასამახსოვრებლად, ასევე ამ ღერძებზე არსებული ფირფიტების სიგრძეების შემცველი ვექტორების შესანახად;

```
struct hanoy
{
      int disc, first, last, temp;
      vector <int> first1, temp1, last1;
};
```

და 4 ფუნქცია, რომელთაგან ერთი არის main() ანუ ფუნქცია დრაივერი:

//ეს ფუნქცია არგუმენტად იღებს იმ სტრუქტურას რომელიც შეიცავს ღერძების სახელებსა და მათზე არსებული ფირფიტების ზომებს და ბეჭდავს ეკრანზე ამ ინფორმაციას... მას არ აქვს დასაბრუნებელი მნიშვნელობა რადგან `void` ტიპისაა. დასაწყისში იგი ავსებს ორგანზუმილებიან `char` ტიპის მასივს `matrix[10][80];` რომელიც მორგებულია ჩვეულებრივი კონსოლის აპლიკაციის ზომებზე და დასაწყისშივე ავსებს მას ‘.’ სიმბოლოებით, შემდგომში კი იგი იღებს გადაცემული არგუმენტიდან ყველა ინფორმაციას და ციკლების მეშვეობით ამ მასივის ყველა ‘.’ სიმბოლოს გადააწერს ‘X’- სიმბოლოს; საბოლოო ჯამში კი მიიღება ჩვენი ამოცანის სიმულაცია, რომელიც ასახავს ფირფიტების გადატანას ღერძებზე...

```
void room(hanoy h)
{
      system("cls");
      system("color 1C");
      char matrix[10][80];
      for (int i = 0; i < 10; i++)
      for (int j = 0; j < 80; j++)
            matrix[i][j] = '.';


      for (unsigned int i = 0; i < h.first1.size(); i++)
      {
            int counter = 0, left_index = 9, right_index = 10;
            while (counter < h.first1[i])
            {
                  for (int j = left_index; j <= right_index; j++)
                        matrix[9 - i][j] = 'X';
```

```
                counter++;
                left_index--;
                right_index++;
            }
        }
        for (unsigned int i = 0; i < h.temp1.size(); i++)
        {
            int counter = 0, left_index = 39, right_index = 40;
            while (counter < h.temp1[i])
            {
                for (int j = left_index; j <= right_index; j++)
                    matrix[9 - i][j] = 'X';
                counter++;
                left_index--;
                right_index++;
            }
        }
        for (unsigned int i = 0; i < h.last1.size(); i++)
        {
            int counter = 0, left_index = 69, right_index = 70;
            while (counter < h.last1[i])
            {
                for (int j = left_index; j <= right_index; j++)
                    matrix[9 - i][j] = 'X';
                counter++;
                left_index--;
                right_index++;
            }
        }
        for (int i = 0; i < 10; i++)
        {
            for (int j = 0; j < 80; j++)
            {
                cout << matrix[i][j];
            }
            cout << endl;
        }
        cout << "        FIRST                        TEMPORARY
LAST        " << endl;
}
```

// ეს ფუნქცია არის სტრუქტურის ტიპის, ანუ იგი აბრუნებს “hanoy” სტრუქტურის ტიპის ობიექტს. არგუმენტებად იღებს ამავე ტიპის სტრუქტურას, რასაც უცვლის მნიშვნელობას. მეხსიერების დაზოგვის მიზნით იგი reference-ითაა გადაცემული. ასევე char ტიპის სიმბოლოებს, რომლებიც, ამ შემთხვევაში არის ღერძების დასახელებები.. ბოლო არგუმენტი არის მეორე სტრუქტურა, რომელიც შეიცავს პოლუსების თავდაპირველ სახელებს. ამ ფუნქციის მიზანია ამოიცნოს ის ღერძები, რომლებზეც ფირფიტები გადაიტანება. ვინაიდან რეკურსიულ ფუნქციაში ხდება

სახელების მონაცემების მრავალჯერადი გადარქმევა და პროგრამაში გამოყენებული first, temp , last ცვლადები იცვლიან მნიშვნელობებს. საბოლოოდ, როდესაც ამოიცნობს კონკრეტულ ღერძებს , ფუნქცია ამოშლის ბოლო ფირფიტის სიგრძის მონაცემს იმ ღერძის ვექტორიდან საიდანაც ხდება ფირფიტის გადატანა და ჩასვამს იმ ღერძის ვექტორში სადაც გადააქვს ფირფიტა..

```
hanoy cal(hanoy &h, char f, char l, char t, hanoy_name &name)
{
	int index;
	if (f == name.first)
	{
		h.first--;
		index = h.first1[h.first1.size() - 1];
		h.first1.erase(h.first1.end() - 1);
	}
	else if (f == name.last)
	{
		h.last--;
		index = h.last1[h.last1.size() - 1];
		h.last1.erase(h.last1.end() - 1);
	}
	else
	{
		h.temp--;
		index = h.temp1[h.temp1.size() - 1];
		h.temp1.erase(h.temp1.end() - 1);
	}


	if (l == name.first)
	{
		h.first++;
		h.first1.push_back(index);
	}
	else if (l == name.last)
	{
		h.last++;
		h.last1.push_back(index);
	}
	else
	{
		h.temp++;
		h.temp1.push_back(index);
	}

	return h;
}
```

// ამ ფუნქციას არ აქვს დასაბრუნებელი მნიშვნელობა ამიტომ არის void ტიპის. იგი არგუმენტად იღებს ფირფიტების საწყის რაოდენობას, ღერძების სახელებს, ორივე სახის სტრუქტურას და unsigned int ტიპის ცვლადს, რაც შეიცავს

ინფორმაციას კომპიუტერის მიმდინარე დროის შესახებ და რომლის იმპლემენტაციაც ხდება main-ში წინასწარ. სწორედ ctime ბიბლიოთეკის მეშვეობით ვიგებთ კომპიუტერის კონკრეტულ დროს და ციკლი ტრიალებს მანამ სანამ 2 წამი არ გავა, ან სანამ ეს ცვლადი 2-ით არ მოიმატებს. დანარჩენი კი უკვე ყველაფერი ნათელია იგი მოქმედებს როგორც ზემოთ მოყვანილი ფსევდოკოდი.. ყველა ბიჯზე room ფუნქცია იძახებს cal ფუნქციას რომელიც აწვდის მას უკვე მზა ინფორმაციას შეცვლილი ცვლადებით დ თავად კი ეკრანზე ბეჭდავს ამ ყველაფერს. იმ შემთხვევაში თუ ფირფიტების რაოდენობა იქნება 0-ზე მეტი ან 11-ზე ნაკლები, პროგრამა მუშაობს, სხვა შემთხვევაში ბეჭდავს რომ 10-ზე მეტი არ შეიძლება..

```
void tower(int n, char f, char l, char t, hanoy &h, hanoy_name
&name, unsigned int &now)
{
	if (n <= 10)
	{

		if (n == 1)
		{
			while (1)
			{
				if ((time(0) - now) >= 2)
				{
					now = time(0);
					room(cal(h, f, l, t, name));
					break;
				}
			}
		}
		else
		{
			tower(n - 1, f, t, l, h, name, now);
			tower(1, f, l, t, h, name, now);
			tower(n - 1, t, l, f, h, name, now);
		}
	}
	else cout << " 10 ze meti koshki ar sheidzleba" << endl;
}
```

/// რაც შეეხება ფუნქცია დრაივერს მასში ხდება ყველა ცვლადის იმპლემენტაცია და ყველაფრის მართვა...

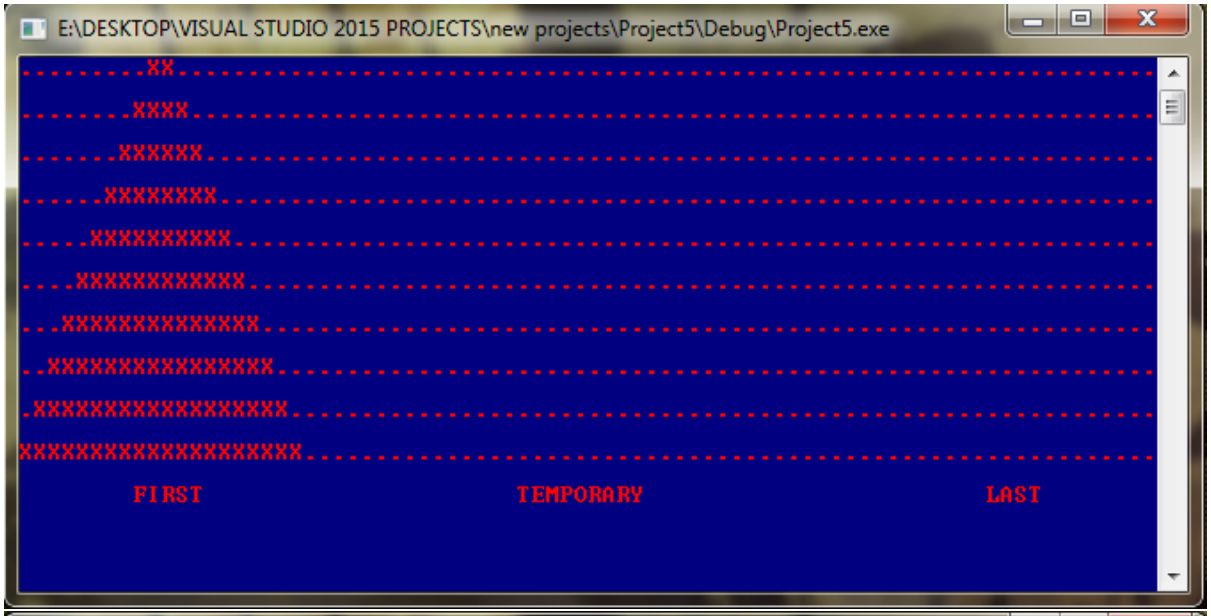
```
int main()
{
	system("color 0c");
	int disc;
	char first = 'F', last = 'L', temp = 'T';


	cout <<
"////////////////////////////////////////////////////////////////
/////////////";
	cout << "          HELLO!!! THE NAME OF THE PROGRAM IS 'HANOY
TOWER'                     ";
	cout << "        I AM THE EDITOR OF THE PROGRAM-ANZOR
GOZALISHVILI                          ";
	cout << "           PLEA/////SE!!!FOLLOW THE INSTRUQTIONS...!!
";
	cout <<
"////////////////////////////////////////////////////////////////
////////////\n";
	cout << "print the number of the towers n=";
	cin >> disc;

	hanoy_name name;
	name.first = first;
	name.last = last;
	name.temp = temp;

	hanoy h;
	h.disc = disc;
	h.first = disc;
	h.last = 0;
	h.temp = 0;
	for (int i = disc; i > 0; i--){
		h.first1.push_back(i);
	}
	unsigned int now = time(0);
	long double count = pow(2.0, disc) - 1;
	room(h);

	tower(disc, first, last, temp, h, name, now);

	cout << "THE NUMBER OF THE MOVES:" << count << endl;
	system("pause");
}
```

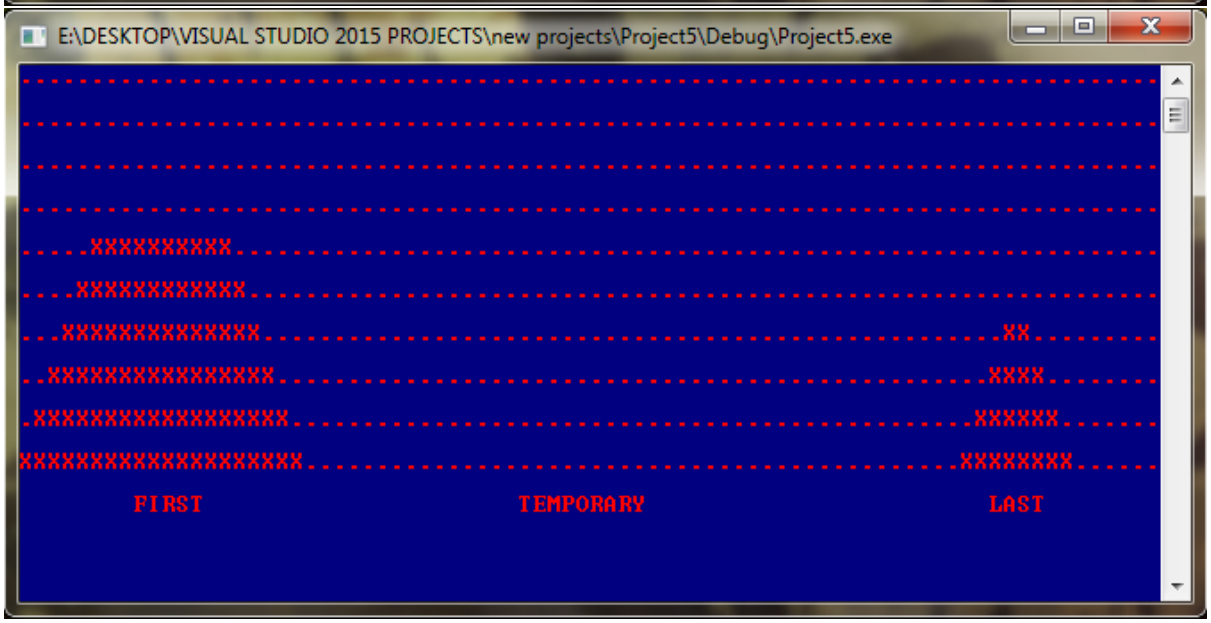
კიდევ ერთი დეტალია აქ რომელსაც უნდა მივაქციოთ ყურადღება, ესაა ბიჯების რაოდენობა. როგორც ხედავთ აქ ძალიან მარტივადაა დათვლილი, მიუხედავად

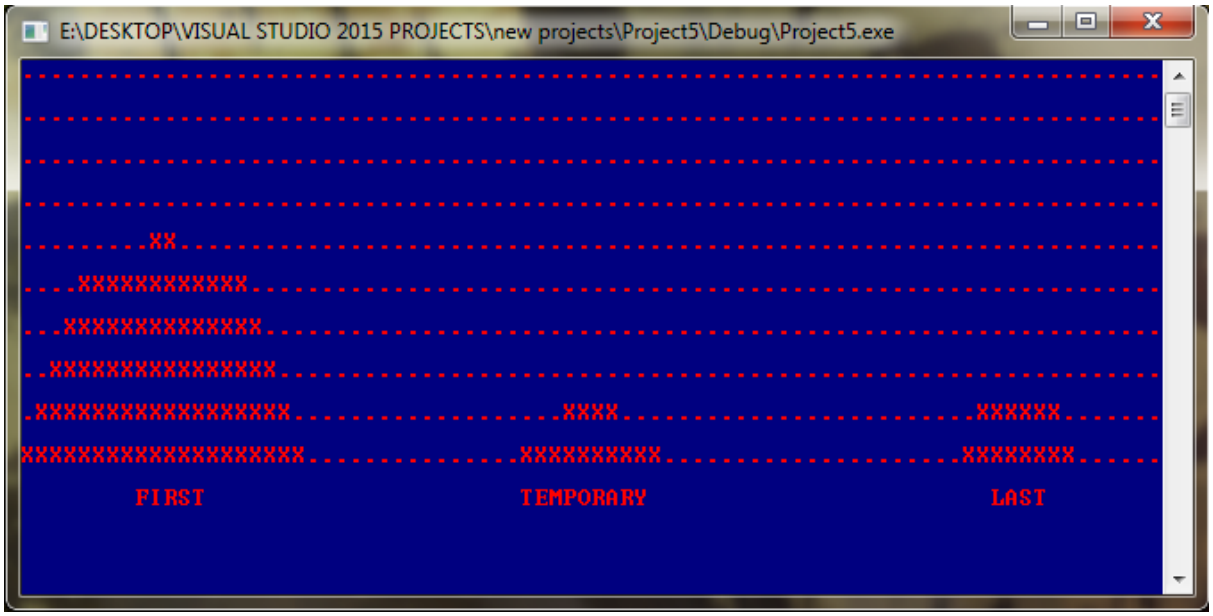
იმისა რომ შეიძლებოდა ერთი დამატებითი ცვლადის შემოტანა და მისი ყოველ ბიჯზე ერთით გაზრდა.. მაგრამ თუ გავხსნით ჰანოის ალგორითმის რეკურსიას მივიღებთ T(n)=2^n-1 . ეს გამოდის შემდეგიდან : T(n)=T(n-1)+T(1)+T(n) სადაც T(1)=1;

F(n)=f(n-1)+F(1)+F(n-1)=2F(n-1)+1=2(2F(n-2)+1)+1=2^2F(n-2)+2^1+2^0= …….. .= 2^(n-1)+ …+ 2^1+2^0=2^n-1

ანუ რეკურსიის გახსნით მივიღეთ მისი ბიჯების რაოდენობა რომელიც არის T(n)=2^n-1


E:\DESKTOP\VISUAL STUDIO 2015 PROJECTS\new projects\Project5\Debug\Project5.exe
FIRST
TEMPORARY
LAST


E:\DESKTOP\VISUAL STUDIO 2015 PROJECTS\new projects\Project5\Debug\Project5.exe
FIRST
TEMPORARY
LAST


E:\DESKTOP\VISUAL STUDIO 2015 PROJECTS\new projects\Project5\Debug\Project5.exe
FIRST
TEMPORARY
LAST