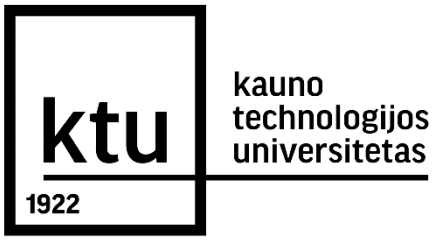
KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS



Modulio P170B400 „Algoritmų sudarymas ir analizė“

Laboratorinio darbo aprašas (ataskaita)
Trečias laboratorinis darbas

Dėstytojas
lekt. MAKACKAS Dalius

Studentas
Matas Palujanskas IFF-1/8

KAUNAS 2023

TURINYS

1. Pirma užduoties dalis

1 užduoties dalis (4 balai):

- Pateikite rekursinį uždavinio sprendimo algoritmą (rekursinis sąryšis su paaiškinimais), bei realizuokite programinį kodą sprendžiantį nurodytą uždavinį (rekursinis sprendimas netaikant dinaminio programavimo).
- Pritaikykite dinaminio programavimo metodologiją pateiktam uždaviniui (pateikti paaiškinimą), bei realizuokite programinį kodą sprendžiantį nurodytą uždavinį (taikant dinaminį programavimą).
- Atlikite realizuotų programinių kodų analizę ir apskaičiuokite įverčius „iš viršaus“ ir „iš apačios“. Atlikite našumo analizę (skaičiuojant programos vykdymo laiką arba veiksmų skaičių) ir patikrinkite, ar apskaičiuotas metodo asimptotinis sudėtingumas atitinka eksperimentinius rezultatus.

Užduoties sąlyga:

Kiekvienam medžio (ne binarinis) mazgui priskirtas teigiamas sveikas skaičius. Reikia rasti mazgus, kad bendra priskirtų skaičių suma būtų didžiausia, bet kurioje nebūtų imami mazgai, kurių tėvinis elementas jau yra įtrauktas į sumą.

2. Pirmo uždavinio sprendimas

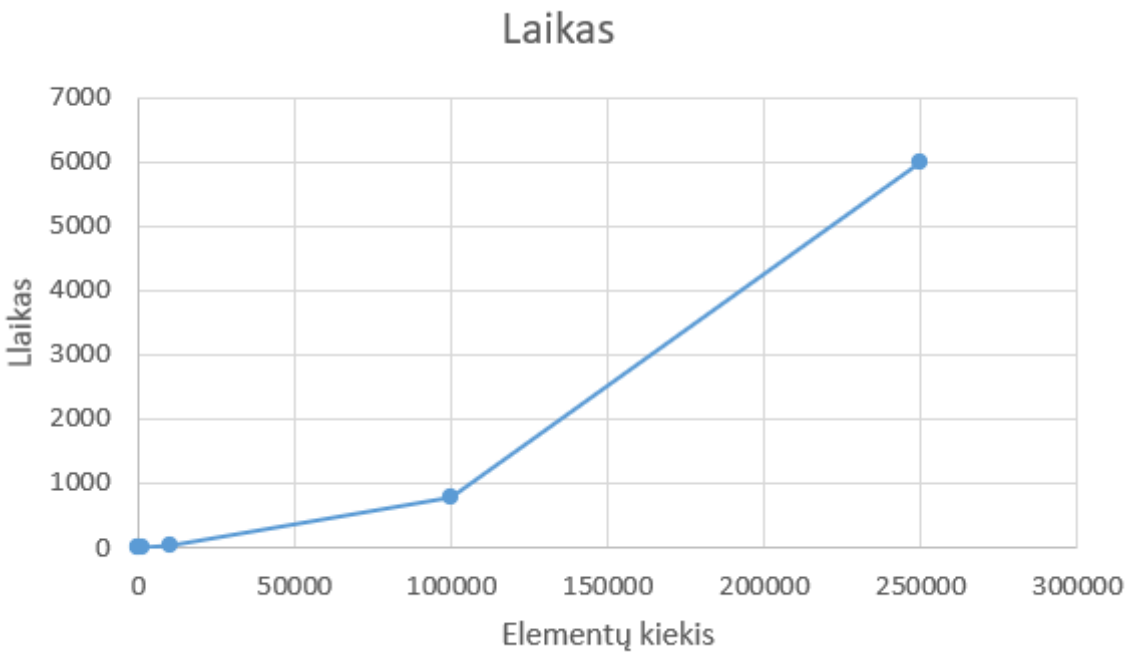
2.1. Rekursinis sprendimas

Kodas	Kaina	Kiekis
`public static int FindMaxSumRecursion(Node node)`		
`{`		
`int included = node.Value;`	c_1	1
`foreach (var child in node.Children)`	c_2	n
`{`		
`foreach (var grandchild in child.Children)`	c_3	n^2
`{`		
`included += FindMaxSumRecursion(grandchild);`	$T(n-1)$	n^2
`}`		
`}`		
`int excluded = 0;`	c_4	1
`foreach (var child in node.Children)`	c_5	n
`{`		
`excluded += FindMaxSumRecursion(child);`	$T(n-1)$	n
`}`		
`return Math.Max(included, excluded);`	c_6	1
`}`		

$$T(n) = \begin{cases} \\ (c_1 + c_2 * n) + c_3 * n^2 + T(n-1) * n^2 + c_4 + c_5 * n + T(n-1) * n + c_6 \end{cases}$$

$$T(n) = \begin{cases} O(1), geriausias \\ O(n^2), blogiausias \end{cases}$$

2.2. Veikimo grafikas



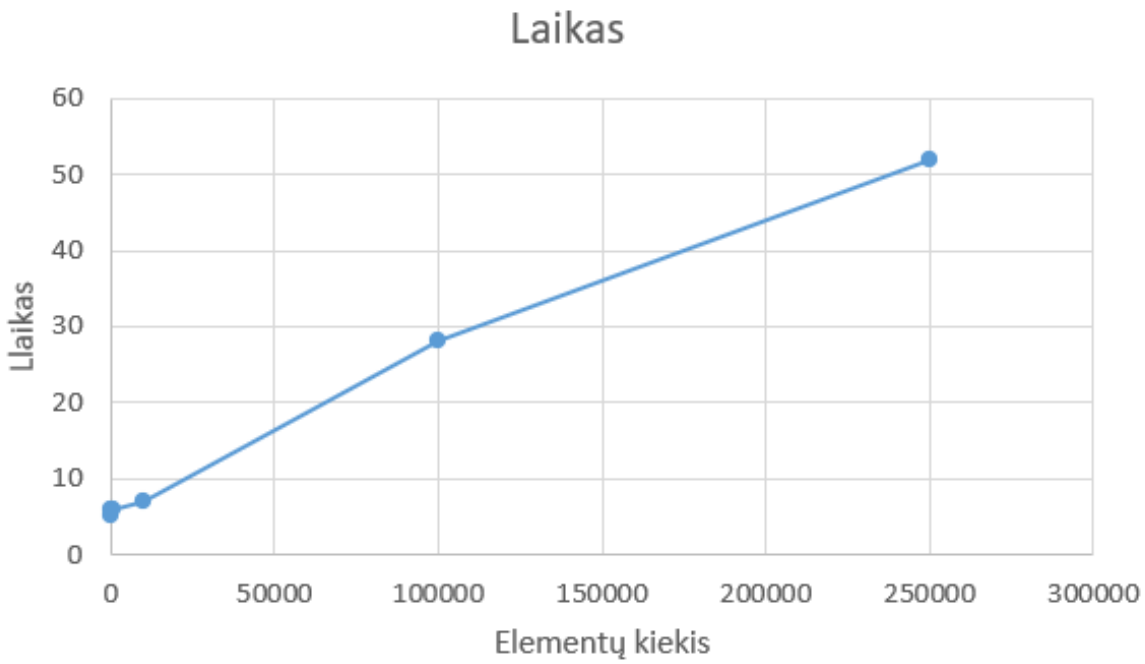
2.3. Sprendimas naudojant dinaminį programavimą

Kodas	Kaina	Kiekis
`public static int FindMaxSumDynamic(Node node)`		
`{`		
`    if (node == null) return 0;`	c_1	1
`    Dictionary<Node, int> memo = new Dictionary<Node, int>();`	c_2	1
`    Stack<Node> stack = new Stack<Node>();`	c_3	1
`    stack.Push(node);`	c_4	1
`    while (stack.Count > 0)`	c_5	n
`    {`		
`        Node current = stack.Peek();`	c_6	1
`        if (current.Children.Count == 0)`	c_7	1
`        {`		
`            memo[current] = current.Value;`	c_8	1
`            stack.Pop();`		
`        }`		
`        else`		
`        {`		
`            bool allChildrenVisited = true;`	c_9	1
`            int sum = current.Value;`	c_{10}	1
`            foreach (Node child in current.Children)`	c_{11}	n^2
`            {`		
`                if (!memo.ContainsKey(child))`	c_{12}	1
`                {`		
`                    stack.Push(child);`	c_{13}	1
`                    allChildrenVisited = false;`	c_{14}	1
`                }`		
`                else`		
`                {`		
`                    sum += memo[child];`	c_{14}	
`                }`		
`                foreach (Node grandchild in child.Children)`	c_{15}	n^3
`                {`		
`                    if (memo.ContainsKey(grandchild))`	c_{16}	1
`                    {`		
`                        sum -= memo[grandchild];`	c_{17}	1
`                    }`		
`                }`		
`            }`		
`            if (allChildrenVisited)`	c_{18}	1
`            {`		
`                int exclude = 0;`	c_{19}	1
`                if (current.Children.Count > 0)`	c_{20}	1
`                {`		
`                    exclude = memo[current.Children[0]];`	c_{21}	1
`                }`		
`                memo[current] = Math.Max(sum, exclude);`	c_{22}	1
`                stack.Pop();`	c_{23}	1
`            }`		
`        }`		
`    }`		
`    return memo[node];`	c_{24}	1
`}`		

$$T(n) = (c_1 + c_2 + c_3) + c_4 + c_5 * n + c_6 + c_7 + c_8 + c_9 + c_{10} + c_{11} * n^2 + c_{12} + c_{13} + c_{14} + c_{15} * n^3 + c_{16} + c_{17} + c_{18} + c_{19} + c_{20} + c_{21} + c_{22} + c_{23} + c_{24}$$

$$T(n) = \begin{cases} O(n), geriausias \\ O(n^3), blogiausias \end{cases}$$

2.4. Veikimo grafikas



3. Kodas

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Diagnostics.Metrics;
using System.Drawing;

namespace PirmaUzduotis
{
	class Node
	{
		public int Value { get; set; }
		public List<Node> Children { get; set; }

		public Node(int value, List<Node> children)
		{
			Value = value;
			Children = children;
		}
		public static Node CreateRandomTree(int size)
		{
			if (size < 1)
			{
		   return null;
			}

			Random random = new Random();
			int value = random.Next(100);
			List<Node> children = new List<Node>();
			for (int i = 0; i < size; i++)
			{
				int childSize = random.Next(0, size - i);
				Node node = CreateRandomTree(childSize);
				if (node != null)
				{
					children.Add(node);
					i += childSize;
				}
			}

			return new Node(value, children);
		}
	}

	class TaskUtils
	{
		public static int FindMaxSumRecursion(Node node)
		{
			int included = node.Value;
			foreach (var child in node.Children)
			{
				foreach (var grandchild in child.Children)
				{
					included += FindMaxSumRecursion(grandchild);
				}
			}
			int excluded = 0;
			foreach (var child in node.Children)
			{
				excluded += FindMaxSumRecursion(child);
			}
			return Math.Max(included, excluded);
		}

		public static int FindMaxSumDynamic(Node node)
		{
			if (node == null) return 0;
```

```
        Dictionary<Node, int> memo = new Dictionary<Node, int>();
        Stack<Node> stack = new Stack<Node>();
        stack.Push(node);

        while (stack.Count > 0)
        {
            Node current = stack.Peek();

            if (current.Children.Count == 0)
            {
                memo[current] = current.Value;
                stack.Pop();
            }
            else
            {
                bool allChildrenVisited = true;
                int sum = current.Value;
                foreach (Node child in current.Children)
                {
                    if (!memo.ContainsKey(child))
                    {
                        stack.Push(child);
                        allChildrenVisited = false;
                    }
                    else
                    {
                        sum += memo[child];
                    }
                    foreach (Node grandchild in child.Children)
                    {
                        if (memo.ContainsKey(grandchild))
                        {
                            sum -= memo[grandchild];
                        }
                    }
                }
                if (allChildrenVisited)
                {
                    int exclude = 0;
                    if (current.Children.Count > 0)
                    {
                        exclude = memo[current.Children[0]];
                    }
                    memo[current] = Math.Max(sum, exclude);
                    stack.Pop();
                }
            }
        }

        return memo[node];
    }

    public static void PrintTreeHorizontally(Node root)
    {
        if (root == null)
        {
            return;
        }

        Queue<Node> queue = new Queue<Node>();
        queue.Enqueue(root);

        while (queue.Count > 0)
        {
            int levelSize = queue.Count;

            for (int i = 0; i < levelSize; i++)
```

```
                        {
                                Node node = queue.Dequeue();
                                Console.Write(node.Value.ToString().PadLeft(3));

                                if (i < levelSize - 1)
                                {
                                        Console.Write("─┬─");
                                }

                                foreach (Node child in node.Children)
                                {
                                        queue.Enqueue(child);
                                }
                        }

                        Console.WriteLine();
                }
        }
}

class Program
{
        static void Main(string[] args)
        {
                var stopWatch = new Stopwatch();
                stopWatch.Start();
                int n = 250000;
                Node tree = Node.CreateRandomTree(n);
                //TaskUtils.PrintTree(tree);
                //Create a sample tree
                Node root = new Node(10, new List<Node>());
                Node node1 = new Node(20, new List<Node>());
                Node node2 = new Node(30, new List<Node>());
                Node node3 = new Node(40, new List<Node>());
                Node node4 = new Node(50, new List<Node>());
                Node node5 = new Node(60, new List<Node>());
                Node node6 = new Node(70, new List<Node>());
                Node node7 = new Node(80, new List<Node>());

                root.Children.Add(node1);
                root.Children.Add(node2);
                node1.Children.Add(node3);
                node1.Children.Add(node4);
                node2.Children.Add(node5);
                node2.Children.Add(node6);
                node5.Children.Add(node7);
                //TaskUtils.PrintTreeHorizontally(root);


                // Find the maximum sum using recursion
                int maxSumRecursion = TaskUtils.FindMaxSumRecursion(tree);
                Console.WriteLine("Maximum sum using recursion: " + maxSumRecursion);
                stopWatch.Stop();
                Console.WriteLine("Time in milliseconds for FIRST({1}): {0} ms",
stopWatch.ElapsedMilliseconds.ToString(), n);

                stopWatch.Start();
                Node node10 = new Node(100, new List<Node>());
                Node node20 = new Node(50, new List<Node>());
                Node node30 = new Node(75, new List<Node>());
                Node node40 = new Node(25, new List<Node>());
                Node node50 = new Node(175, new List<Node>());
                Node node60 = new Node(125, new List<Node>());
                Node node70 = new Node(50, new List<Node>());
                Node node80 = new Node(100, new List<Node>());
                Node node90 = new Node(50, new List<Node>());
                Node node100 = new Node(200, new List<Node>());
                Node node110 = new Node(150, new List<Node>());
```

```
                    node10.Children.Add(node20);
                    node10.Children.Add(node30);
                    node20.Children.Add(node40);
                    node20.Children.Add(node50);
                    node30.Children.Add(node60);
                    node30.Children.Add(node70);
                    node40.Children.Add(node80);
                    node50.Children.Add(node90);
                    node50.Children.Add(node100);
                    node60.Children.Add(node110);

                    int maxSumDynamic = TaskUtils.FindMaxSumDynamic(tree);
                    Console.WriteLine("Maximum sum using dynamic: " + maxSumDynamic);
                    stopWatch.Stop();
                    Console.WriteLine("Time in milliseconds for SECOND({1}): {0} ms",
stopWatch.ElapsedMilliseconds.ToString(), n);


            }

        }
}
```

4. Pirmos užduoties analizė

5. Užduotis Nr. 2

2-3 užduotys (6 balai):

- Atlikite pateiktų procedūrų lygiagretinimą.
- Įvertinkite teorinį nelygiagretintų ir lygiagretintų procedūrų sudėtingumą.
- Atlikite realizuotų programinių kodų analizę ir apskaičiuokite įverčius „iš viršaus“ ir „iš apačios“. Atlikite našumo analizę (skaičiuojant programos vykdymo laiką arba veiksmų skaičių) ir patikrinkite, ar apskaičiuotas metodo asimptotinis sudėtingumas atitinka eksperimentinius rezultatus.
- 2 uždavinys - 3 balai / 3 uždavinys - 3 balai

6. Pirmas metodas

6.1. Pirmas pradinis metodas

```
public static long methodToAnalysis (int [] arr)
    {
        long n = arr.Length;
        long k = n;
        for (int i = 0; i < n; i++)
        {
            if (arr[i] / 7 == 0)
            {
                k -= 2;
            }
            else
            {
                k += 3;
            }
        }
        if (arr[0] > 0) {
            for (int i = 0; i < n*n; i++)
            {
                if (arr[0] > 0)
                {
                    k += 3;
                }
        }
        return k;
    }
```

6.2. Pirmas lygiagretintas metodas

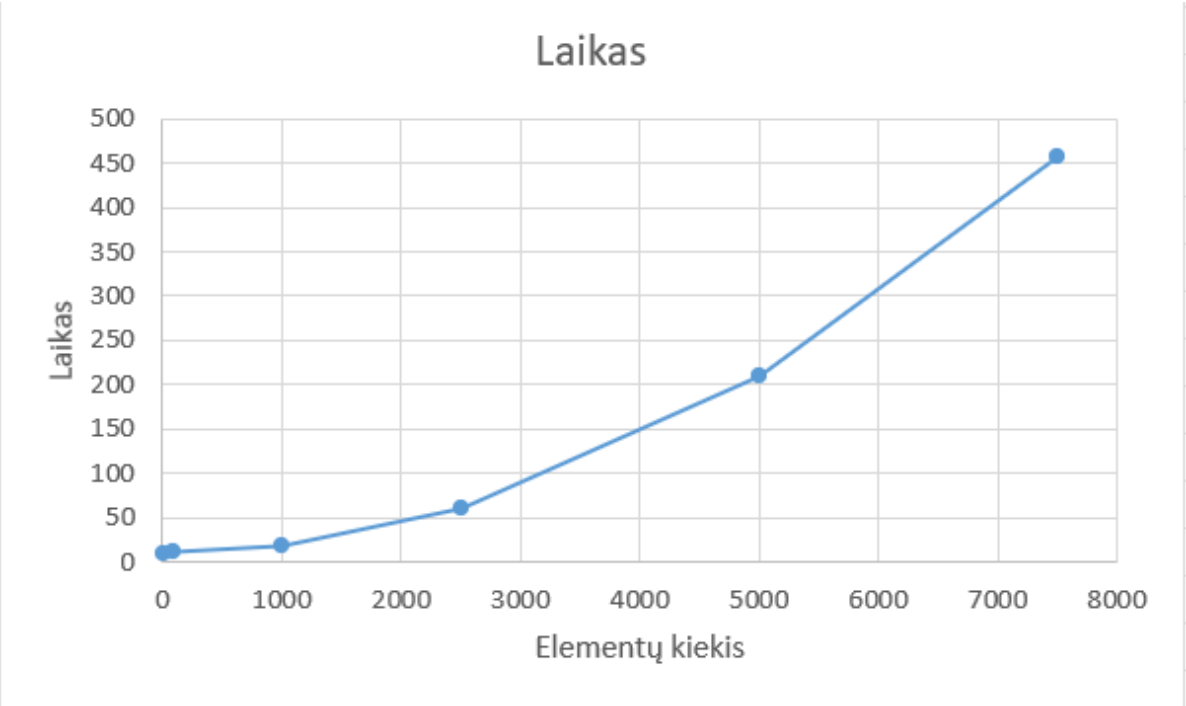
Kodas	Kaina	Kiekis
`public static long FIRST(int[] arr)`		
`{`		
`long n = arr.Length;`	c_1	1
`long k = n;`	c_2	1
`Parallel.For(0, arr.Length, i =>`	c_3	n
`{`		
`if (arr[i] / 7 == 0)`	c_4	1
`{`		
`k -= 2;`	c_5	1
`}`		
`else`		
`{`		
`k += 3;`	c_6	1
`}`		
`});`		
`if (arr[0] > 0)`	c_7	1
`{`		
`Parallel.For(0, n * n, i =>`	c_8	n
`{`		
`if (arr[0] > 0)`	c_9	1
`{`		
`k += 3;`	c_{10}	1
`}`		
`});`		
`}`		
`return k;`	c_{11}	1
`}`		

$$T(n) = \begin{cases} \\ (c_1 + c_2) + c_3 * n + c_4 + c_5 + c_6 + c_7 + c_8 * n + c_9 + c_{10} + c_{11} \end{cases}$$

$$T(n) = \begin{cases} O(1), geriausias \\ O(n), blogiausias \end{cases}$$

$$\textbf{Lygiagretinimo koeficientas} = \frac{\boldsymbol{O(n)}}{\boldsymbol{O(n)}} = \mathbf{1}$$

6.3. Veikimo grafikas



Atvaizduotas laiko kitimas nuo elementų kiekio.

7. Antras metodas

7.1. Antras pradinis metodas

```
public static long methodToAnalysis (int n, int[] arr)
    {
        long k = 0;
        for (int i = 1; i < n; i++)
        {
            k += k;
            k += FF10(i, arr);
            k += FF10(i/i, arr);
        }

        return k;
    }

    public static long FF10(int n, int[] arr)
    {
        if (n > 1 && arr.Length > n )
        {
            return FF10(n - 2, arr) + FF10 (n / n, arr);
        }
        return n;
    }
```

7.2. Antras lygiagretintas metodas

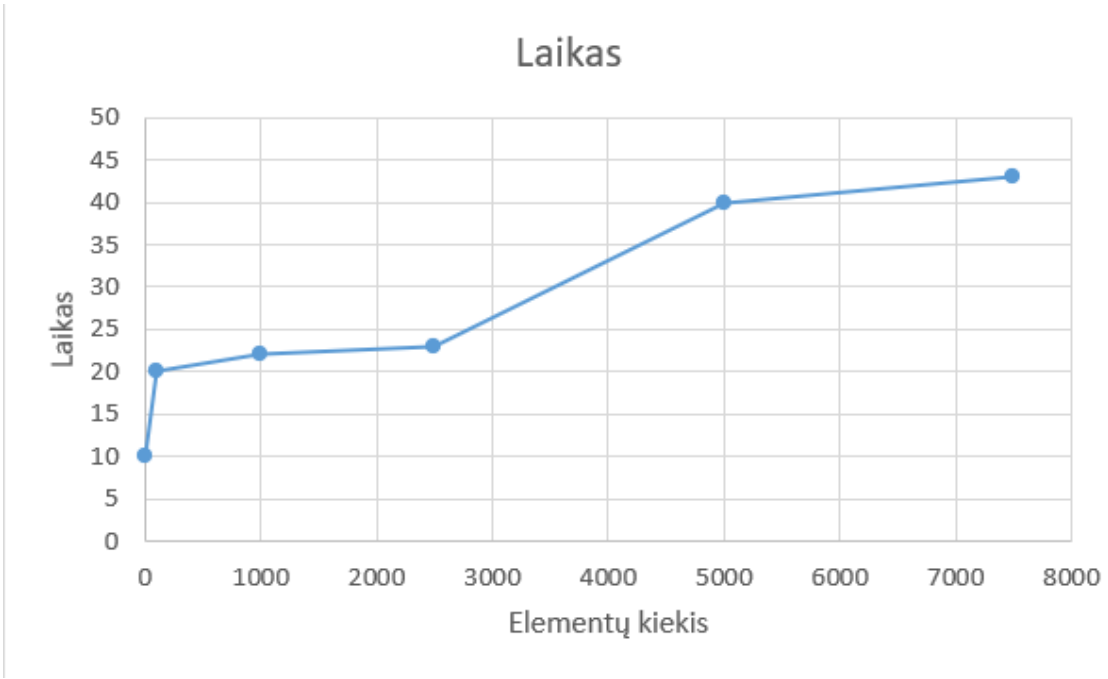
Kodas	Kaina	Kiekis
`public static long SecondMethodToAnalysisParallel(int n, int[] arr)`		
`{`		
`long[] localKArray = new long[n - 1];`	c_1	1
`object lockObj = new object();`	c_2	1
`Parallel.For(0, n, i =>`	c_3	n
`{`		
`long localK = 0;`	c_4	1
`localK += FF10(i, arr);`	T(i)	n
`localK += FF10(i / i, arr);`	T(i/i)	n
`localKArray[i - 1] = localK;`	c_5	1
`});`		
`long k = 0;`	c_6	1
`lock (lockObj)`	c_7	1
`{`		
`for (int i = 0; i < localKArray.Length; i++)`	c_8	n
`{`		
`k += k;`	c_9	1
`k += localKArray[i];`	c_{10}	1
`}`		
`}`		
`return k;`	c_{11}	1
`}`		
`public static long FF10(int n, int[] arr)`		
`{`		
`if (n > 1 && arr.Length > n)`	c_{12}	1
`{`		
`return FF10(n - 2, arr) + FF10(n / n, arr);`	T(n-2)+T(n/n)	
`}`		
`return n;`	c_{13}	1
`}`		

$$\boldsymbol{T(n)} = (c_1 + c_2) + c_3 * n + c_4 + 2\big(T(n-1)\big) * n + c_5 + c_6 + c_7 + c_8 * n + c_9 + c_{10} + c_{11} + c_{12} + T(n-2) + T\left(\frac{n}{n}\right) + c_{13}$$

$$T(n) = \begin{cases} O(1), geriausias \\ O(n), blogiausias \end{cases}$$

$$\textbf{Lygiagretinimo koeficientas} = \frac{\boldsymbol{O(n)}}{\boldsymbol{O(n)}} = \mathbf{1}$$

7.3. Veikimo grafikas



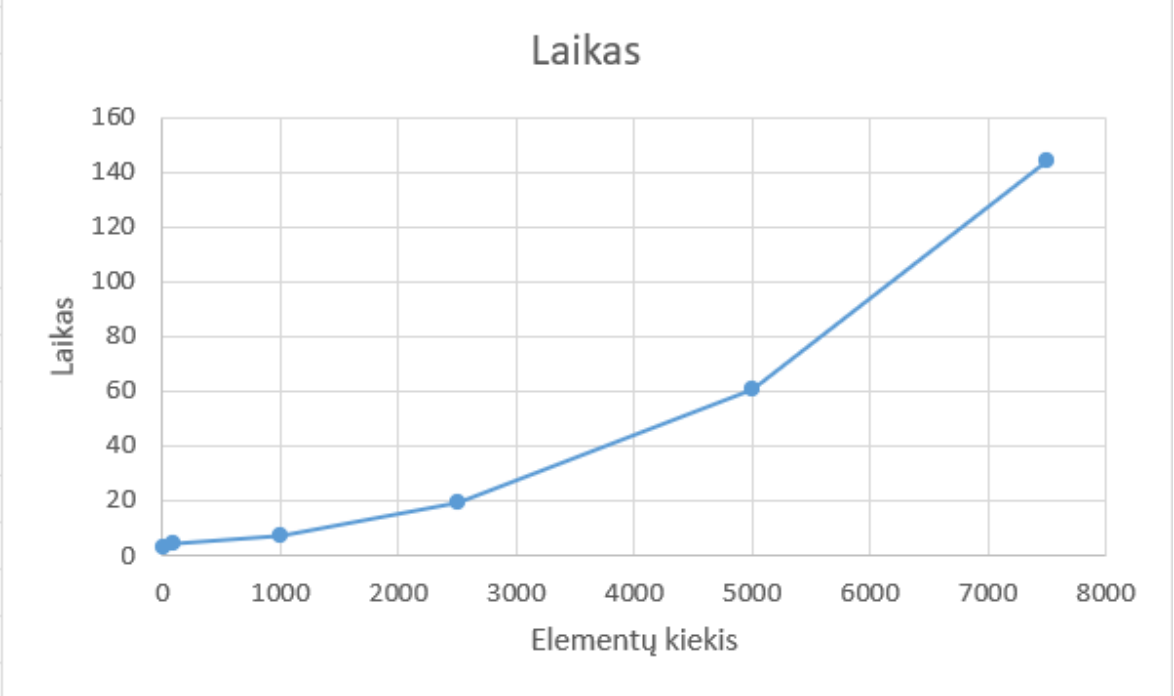
8. 2-3 užduočių kodas

```
using System.Diagnostics;

namespace Lygiagretinimas
{
	class Program
	{
		public static long FIRST(int[] arr)
		{
			long n = arr.Length;
			long k = n;

			Parallel.For(0, arr.Length, i =>
			{
				if (arr[i] / 7 == 0)
				{
					k -= 2;
				}
				else
				{
					k += 3;
				}
			});
			if (arr[0] > 0)
			{
				Parallel.For(0, n * n, i =>
				{
					if (arr[0] > 0)
					{
						k += 3;
					}
				});
			}
			return k;
		}

		static object monitor = new object();
		static int threadCount = 0;
		public static long SECOND(int n, int[] arr)
		{
			long k = 0;
			Task[] tasks = new Task[2];
			Int64 x=0;
			Int64 y=0;

			Parallel.For(1, arr.Length, i =>
			{
				k += k;
				k += SECONDFF10(i, arr);
				k += SECONDFF10(i / i, arr);
				Task<Int64> task1 = Task<Int64>.Factory.StartNew((local_n) => {
lock (monitor) threadCount++; return SECONDFF10((int)i, arr); }, n);
				Task<Int64> task2 = Task<Int64>.Factory.StartNew((local_n) => {
lock (monitor) threadCount++; return SECONDFF10((int)i / i, arr); }, n);
				Task.WaitAll(task1);
				Task.WaitAll(task2);

				//k += task1.Result;
				//k += task2.Result;
			});
			//return x+y;
			return k;
		}

		public static long SECONDFF10(int n, int[] arr)
		{
```

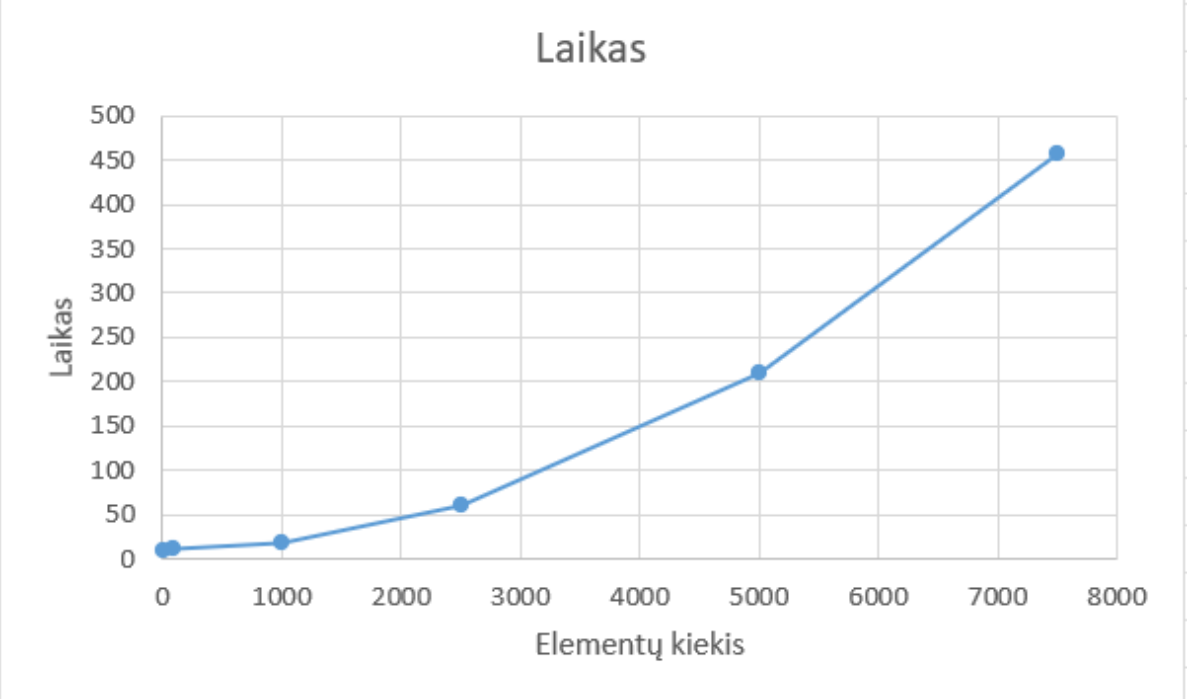
```
                    long k = 0;
                    Int64 x = 0;
                    Int64 y = 0;
                    if (n > 1 && arr.Length > n)
                    {
                          Task<Int64> task1 = Task<Int64>.Factory.StartNew((local_n) => {
lock (monitor) threadCount++; return SECONDFF10((int)n-2, arr); }, n);
                          Task<Int64> task2 = Task<Int64>.Factory.StartNew((local_n) => {
lock (monitor) threadCount++; return SECONDFF10((int)n / n, arr); }, n);
                          Task.WaitAll(task1);
                          Task.WaitAll(task2);

                          k += task1.Result;
                          k += task2.Result;
                          return k;
                          //return SECONDFF10(n - 2, arr) + SECONDFF10(n / n, arr);
                    }
                    return n;
              }
              public static int[] GenerateRandomArray(int length, int minValue, int
maxValue)
              {
                    Random random = new Random();
                    int[] arr = new int[length];
                    for (int i = 0; i < length; i++)
                    {
                          arr[i] = random.Next(minValue, maxValue);
                    }
                    return arr;
              }


              static void Main(string[] args)
              {
                    var stopWatch = new Stopwatch();
                    int[] values = GenerateRandomArray(100, 1, 100);
                    int n = 100;
                    stopWatch.Start();
                    Console.WriteLine("First result: " + FIRST(values));
                    stopWatch.Stop();
                    Console.WriteLine("Time in milliseconds for FIRST({1}): {0} ms",
stopWatch.ElapsedMilliseconds.ToString(), n);

                    stopWatch.Reset();
                    stopWatch.Start();
                    Console.WriteLine("Second result: " + SECOND(n, values));
                    stopWatch.Stop();
                    Console.WriteLine("Time in milliseconds for SECOND({1}): {0} ms",
stopWatch.ElapsedMilliseconds.ToString(), n);
              }
      }
}
```

9. Rezultatai

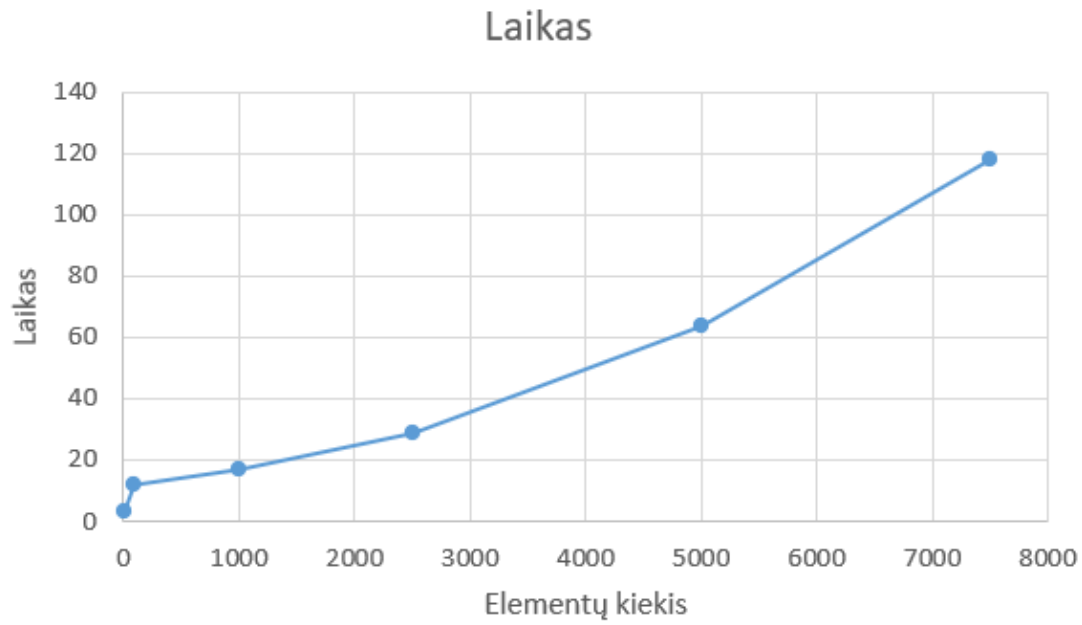
Pirmas prieš išlygiagretinimą:



Pirmas išlygiagretintas:



Antras prieš išlygiagretinimą:



Antras išlygiagretintas:

IŠVADA:
Apskaičiuotas darbo laikas šiek tiek skyrėsi pradinių ir lygiagretintų metodų. Pirmas metodas greičiau dirbo prieš lygiagretinimą, antrasis po lygiagretinimo. Apibendrinant pradinių ir lygiagretintų metodų darbo laikas buvo panašus.