Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

ЗВІТ

розрахунково-графічної роботи

з дисципліни «Архітектура програмного забезпечення»

Тема: «Архітектурні діаграми та бенчмарки»

Виконали: студенти групи ІП-93
Дмитрієв Дмитро
Заводовська Єлізавєта
Зегельман Марк
Марченко Максим

Перевірив:
Мазур Р. Ф.

Київ 2021

Мета: закріплення навичок ілюстрації організації програмних систем та оцінки часу виконання алгоритмів.

Завдання

- Для 2-гої роботи, підтвердьте лінійний час виконання вашої функції перетворення чи обчислення вхідного виразу.
- Для 3-тої роботи, побудуйте діаграму взаємодії компонентів у вашій імплементації.
- Для 4-ої роботи, побудуйте діаграму взаємодії для вашої реалізації (на ній, скоріш за все, мають опинитися компоненти парсера, черги команд, ядра цикла) та підтвердьте лінійний час роботи вашого парсера команд.

Лабораторна робота №2

Лінійний час виконання функції перетворення вхідного виразу (з постфіксного у інфіксний)

Для підтвердження лінійного часу виконання нашої функції нам знадобиться бенчмарк, який запускатиме її, передаючи в аргумент вираз різного розміру (довжина виразу постійно зростатиме).

Код нижче запускає функцію PostfixToInfix 18 разів, передаючи їй вхідний рядок різного розміру. Кожен запуск оформлено у вигляді окремого дочірнього бенчмарка, що дає можливість отримати середній час виконання функції для кожного з розмірів.

файл lab2/bench_test.go

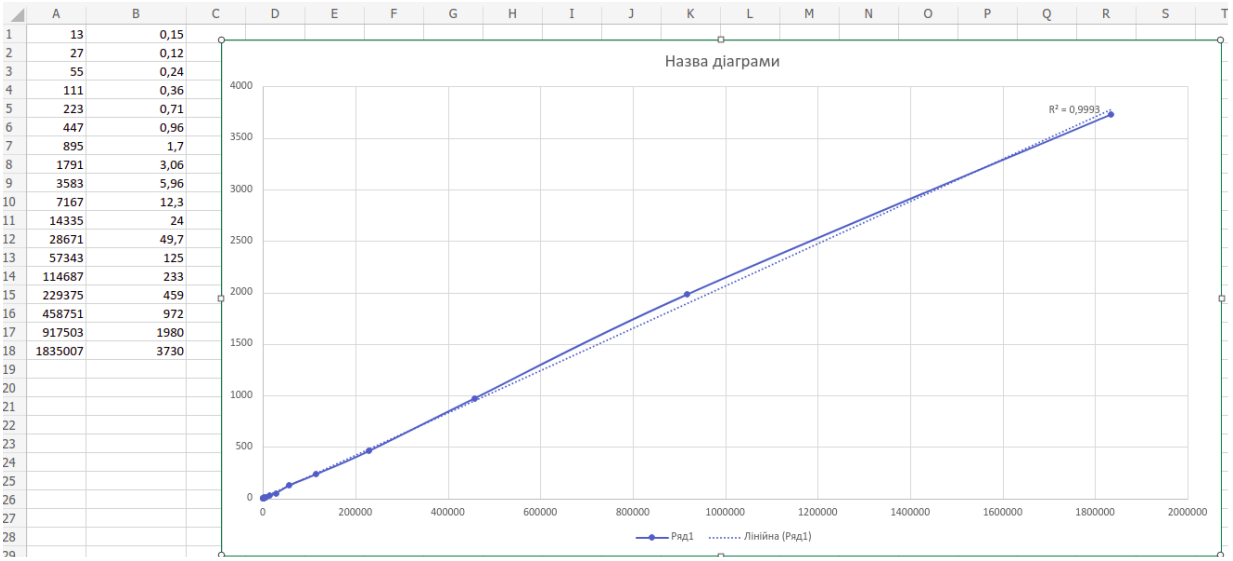
```
package lab2

import (
	"fmt"
	"testing"
)

var startStatement string = "1 2 +"
var cntRes string
var err error

func BenchmarkPostfixToInfix(b *testing.B) {
	const baseLen = 13
	iterationsNum := baseLen
	input := "1 34 + 53 + 13 + 43 454 / - 2 4 * 3 / 4 3 2 5 - * 6
/ + / +"

	for i := 0; i < 18; i++ {
		b.Run(fmt.Sprintf("length = %d", iterationsNum), func(b
*testing.B) {
			cntRes, err = PostfixToInfix(input)
		})

		iterationsNum = iterationsNum*2 + 1
		input = input + input + " +"
	}
}
```

Варто зазначити, що результат виконання та помилка функції зберігається у глобальних змінних пакет та err. Це зроблено, щоб уникнути оптимізації самого компілятора.

Запуск даного бенчмарка дає нам 18 точок для візуалізації:

```
kertnik@slave-node:~/Documents/go/lab2$ go test -bench . - cpu 1
goos: linux
goarch: amd64
pkg: github.com/zavad4/go/tree/main/lab2
BenchmarkPostfixToInfix/length_=_13-8          1000000000               0.000015 ns/op
BenchmarkPostfixToInfix/length_=_27-8          1000000000               0.000012 ns/op
BenchmarkPostfixToInfix/length_=_55-8          1000000000               0.000024 ns/op
BenchmarkPostfixToInfix/length_=_111-8         1000000000               0.000036 ns/op
BenchmarkPostfixToInfix/length_=_223-8         1000000000               0.000071 ns/op
BenchmarkPostfixToInfix/length_=_447-8         1000000000               0.000096 ns/op
BenchmarkPostfixToInfix/length_=_895-8         1000000000               0.000170 ns/op
BenchmarkPostfixToInfix/length_=_1791-8        1000000000               0.000306 ns/op
BenchmarkPostfixToInfix/length_=_3583-8        1000000000               0.000596 ns/op
BenchmarkPostfixToInfix/length_=_7167-8        1000000000               0.00123 ns/op
BenchmarkPostfixToInfix/length_=_14335-8       1000000000               0.00240 ns/op
BenchmarkPostfixToInfix/length_=_28671-8       1000000000               0.00497 ns/op
BenchmarkPostfixToInfix/length_=_57343-8       1000000000               0.0125 ns/op
BenchmarkPostfixToInfix/length_=_114687-8      1000000000               0.0233 ns/op
BenchmarkPostfixToInfix/length_=_229375-8      1000000000               0.0459 ns/op
BenchmarkPostfixToInfix/length_=_458751-8      1000000000               0.0972 ns/op
BenchmarkPostfixToInfix/length_=_917503-8      1000000000               0.198 ns/op
BenchmarkPostfixToInfix/length_=_1835007-8     1000000000               0.373 ns/op
PASS
ok      github.com/zavad4/go/tree/main/lab2     11.919s
kertnik@slave-node:~/Documents/go/lab2$ cd .
```

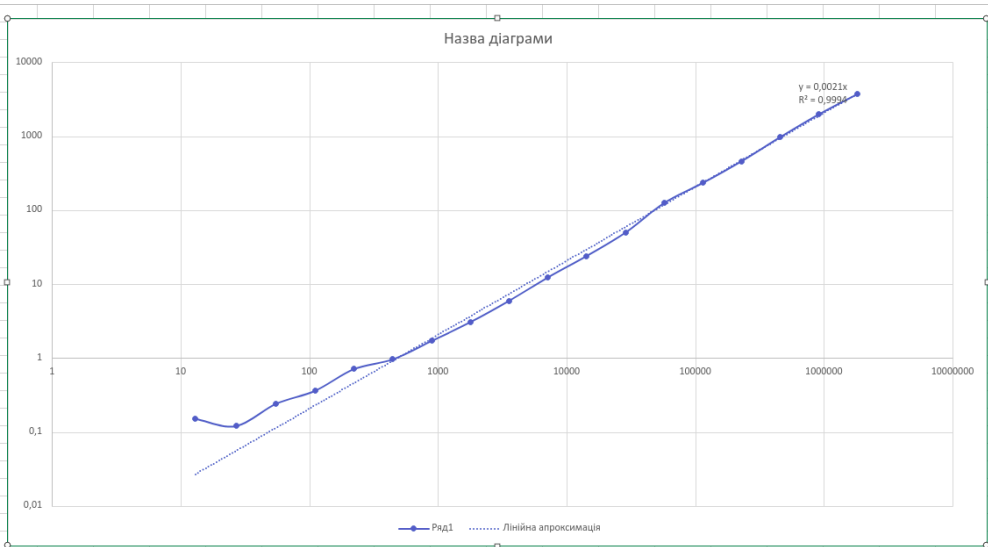
За ними будуємо наш графік:



13	0,15
27	0,12
55	0,24
111	0,36
223	0,71
447	0,96
895	1,7
1791	3,06
3583	5,96
7167	12,3
14335	24
28671	49,7
57343	125
114687	233
229375	459
458751	972
917503	1980
1835007	3730

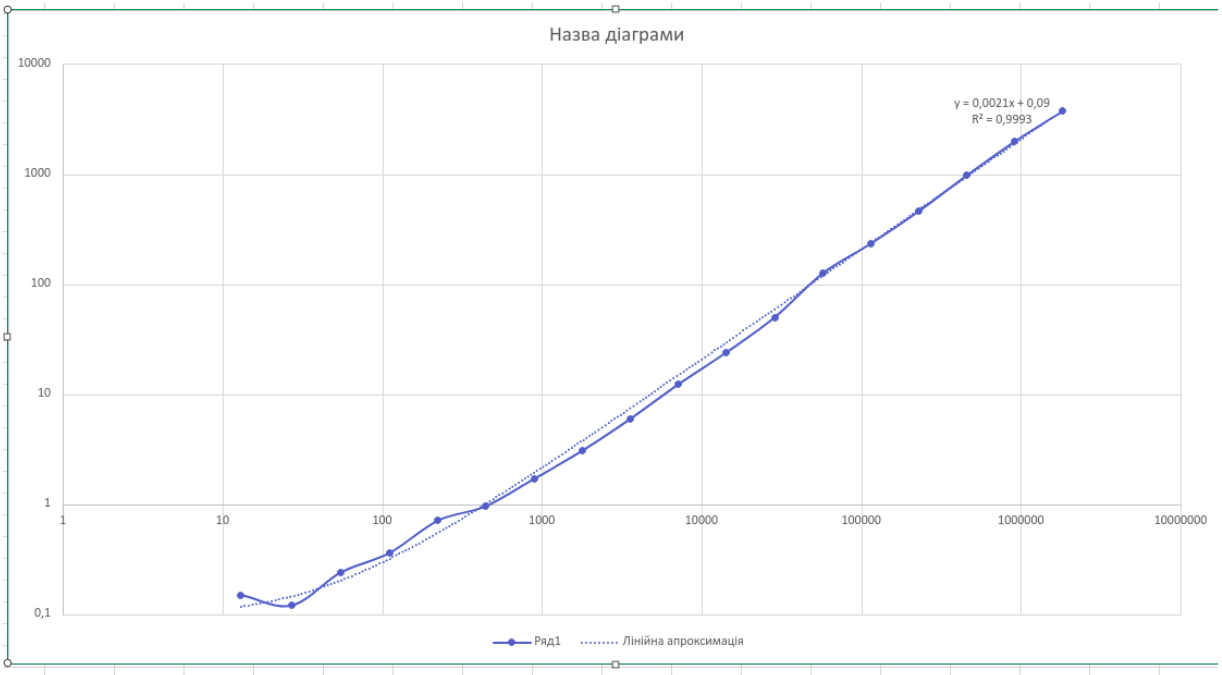
На зображенні суцільною лінією позначено емпіричний графік (створений за нашими точками даних), а штрихованою - лінійне наближення до нього. За графіком помітно, що лінії досить близькі одна до одної - ймовірність правильності наближення становить 0.9993, що є вагомим доказом лінійності алгоритму.

13	0,15
27	0,12
55	0,24
111	0,36
223	0,71
447	0,96
895	1,7
1791	3,06
3583	5,96
7167	12,3
14335	24
28671	49,7
57343	125
114687	233
229375	459
458751	972
917503	1980
1835007	3730



Перехід до логарифмічної шкали показує, що:

1) Для малого числа операцій вагомими є випадкові чинники - наприклад навантаження на процесор у конкретний момент часу. Саме вони є причиною нерівномірності зростання часу виконання алгоритму.
2) Також для малої кількості операцій значний вплив надають ті частини алгоритму, що є константними у часі для різних вхідних даних (наприклад перевірка, чи кількість аргументів більша 3; створення масиву операторів)



Врахуємо час виконання константих частин алгоритму на третьому графіку. Помітно, що для малих значень, лінійна апроксимація тепер є значно ближчою до емпіричної функції, що є вагомим доказом впливу цих

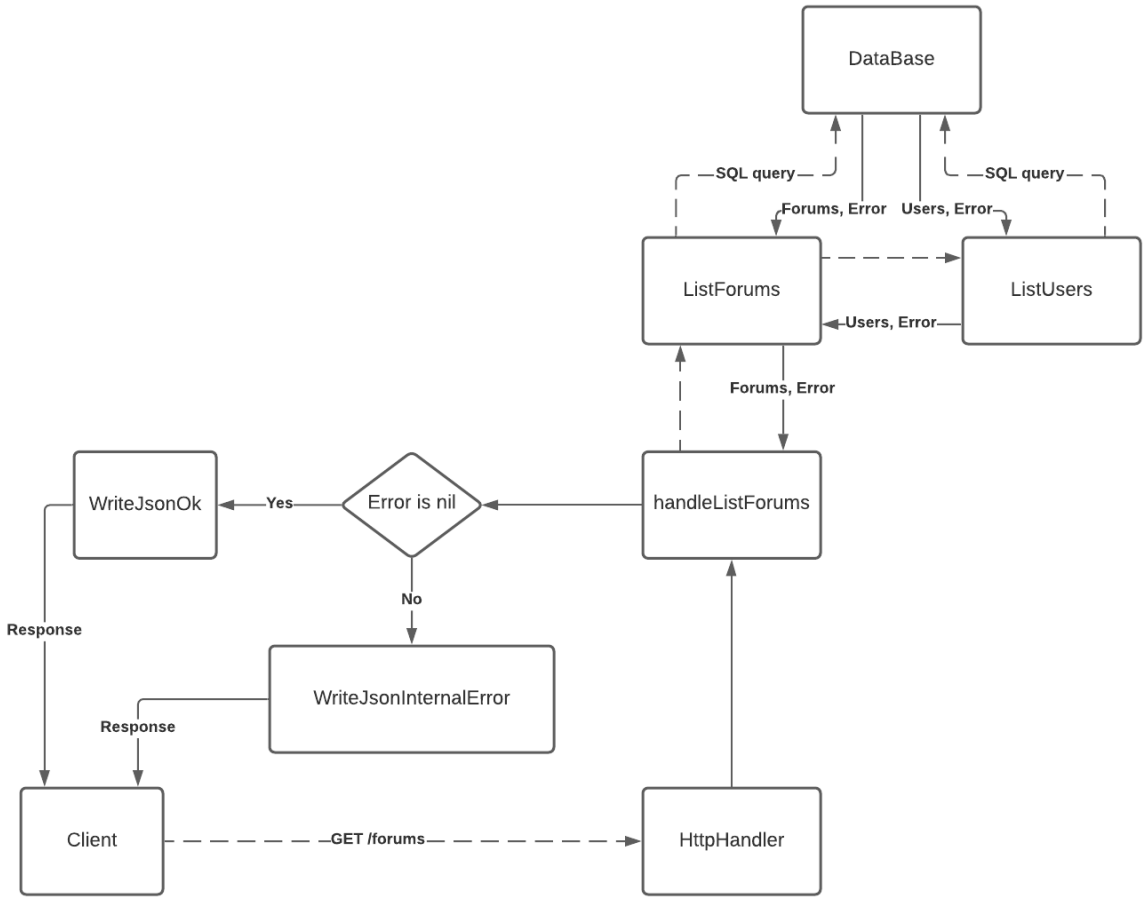
команд. Для нашого алгоритму цей час становить приблизно 0.009 пікосекунд (хоча варто зазначити, що оскільки саме для малої кількості операції на час виконання впливають випадкові фактори, цей час не є точним).

Лабораторна робота №3

Діаграма взаємодії компонентів

Для даної лабораторної роботи необхідно розглянути два випадки взаємодії компонентів: для варіанту перегляду форумів та для додавання користувача.

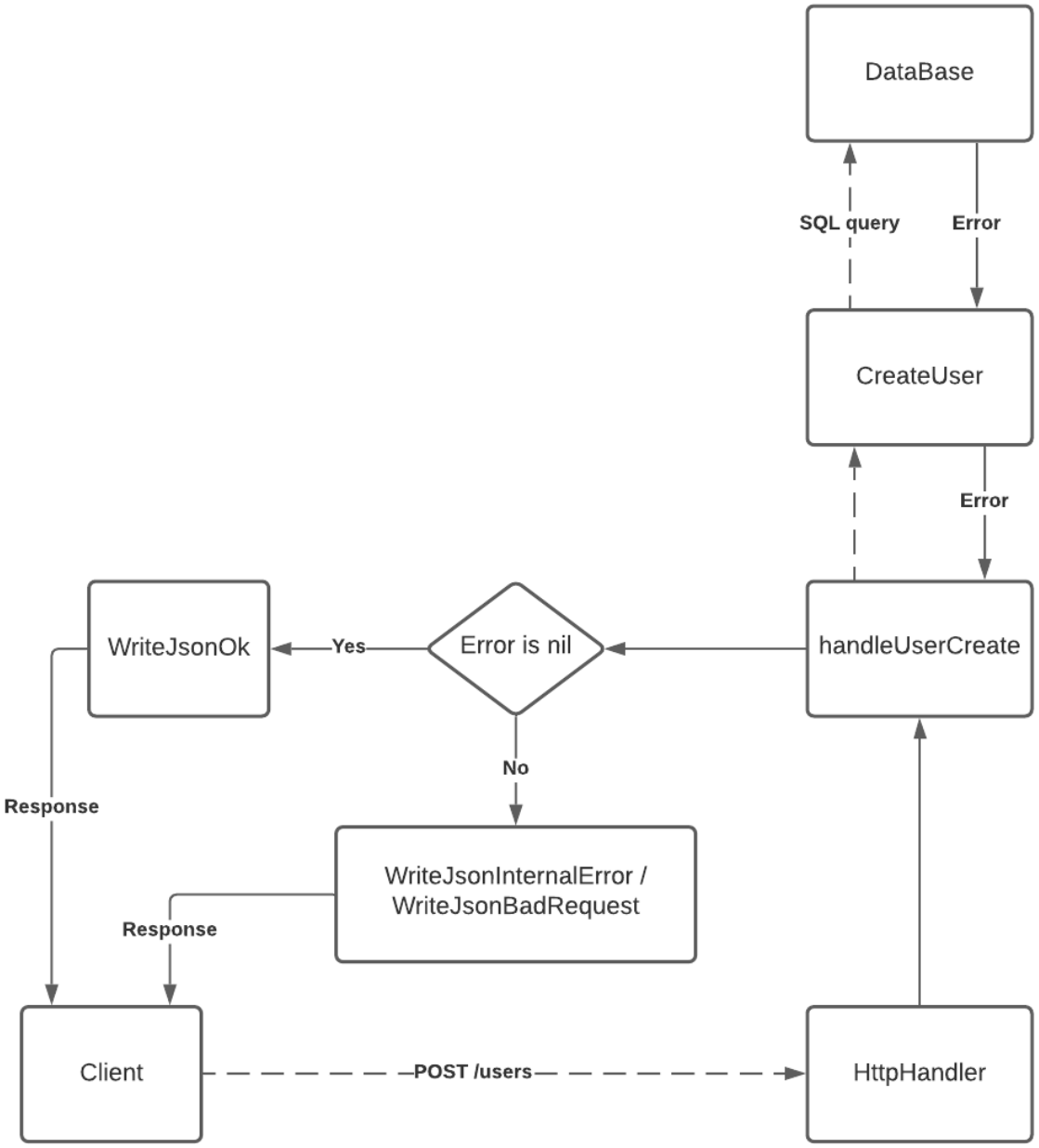
Перегляд форумів:



Після отримання від клієнта GET запиту по шляху /forums дія переходить на HttpHandler, який, в свою чергу, викликає handleListForums. Функція handleListForums відповідає саме за обробку запиту перегляду форумів. Для SQL запиту викликається ListForums. Оскільки в таблиці

форумів немає інформації про користувачів, бо інтереси користувачів не є константою, ListForums отримує список користувачів через ListUsers, та обробляє отримані дані. В результаті, handleListForums повертає клієнту результат запиту або помилку через WriteJsonOk та WriteJsonInternalError відповідно.

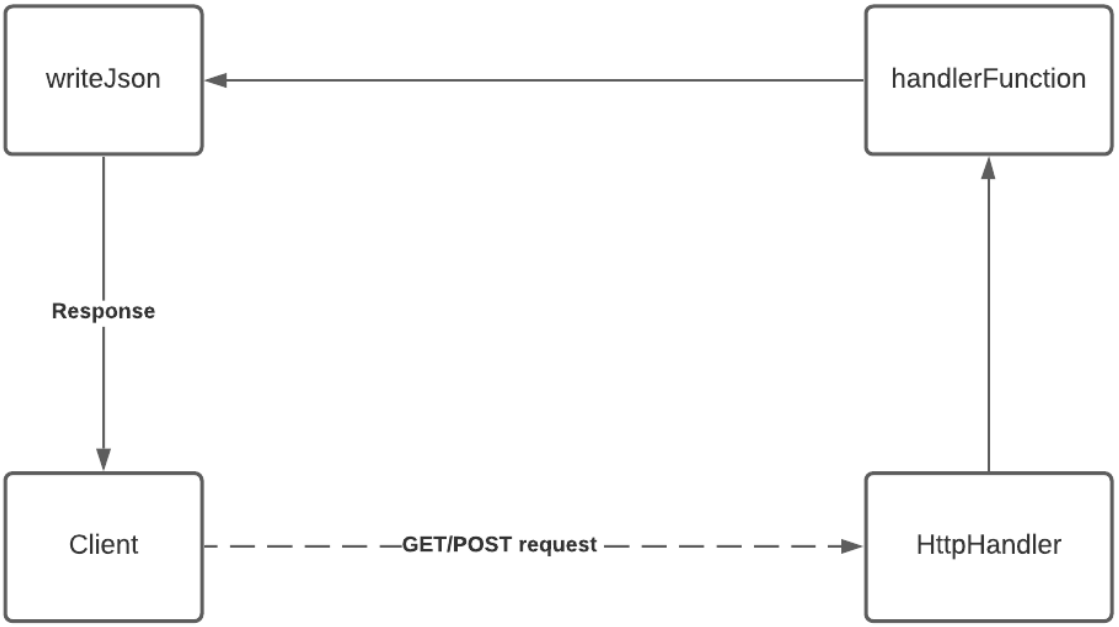
Додавання користувача:



Після отримання від клієнта POST запиту по шляху /users дія переходить на HttpHandler, який, в свою чергу, викликає handleUserCreate. Функція handleUserCreate відповідає саме за обробку запиту додавання

користувача. Для SQL запиту викликається CreateUser. В результаті, handleUserCreate повертає клієнту результат запиту або помилку через WriteJsonOk та WriteJsonInternalError/WriteJsonBadRequest відповідно.

Окрім цього, зробимо загальну діаграму:



За обробку клієнтських запитів відповідає HttpHandler. Залежно від запиту, викликається необхідний обробник (на діаграмі - handlerFunction), задачею якої є надсилання та обробка результату SQL запиту. Після чого, викликається writeJson, що надсилає клієнту результат його запиту.

Лабораторна робота №4

Тип 1: підтвердити лінійний час виконання вашого алгоритму

Для підтвердження лінійного часу виконання нашої функції нам знову знадобиться бенчмарк із 27 точками візуалізації:

файл lab4/commands/benchmark_test.go

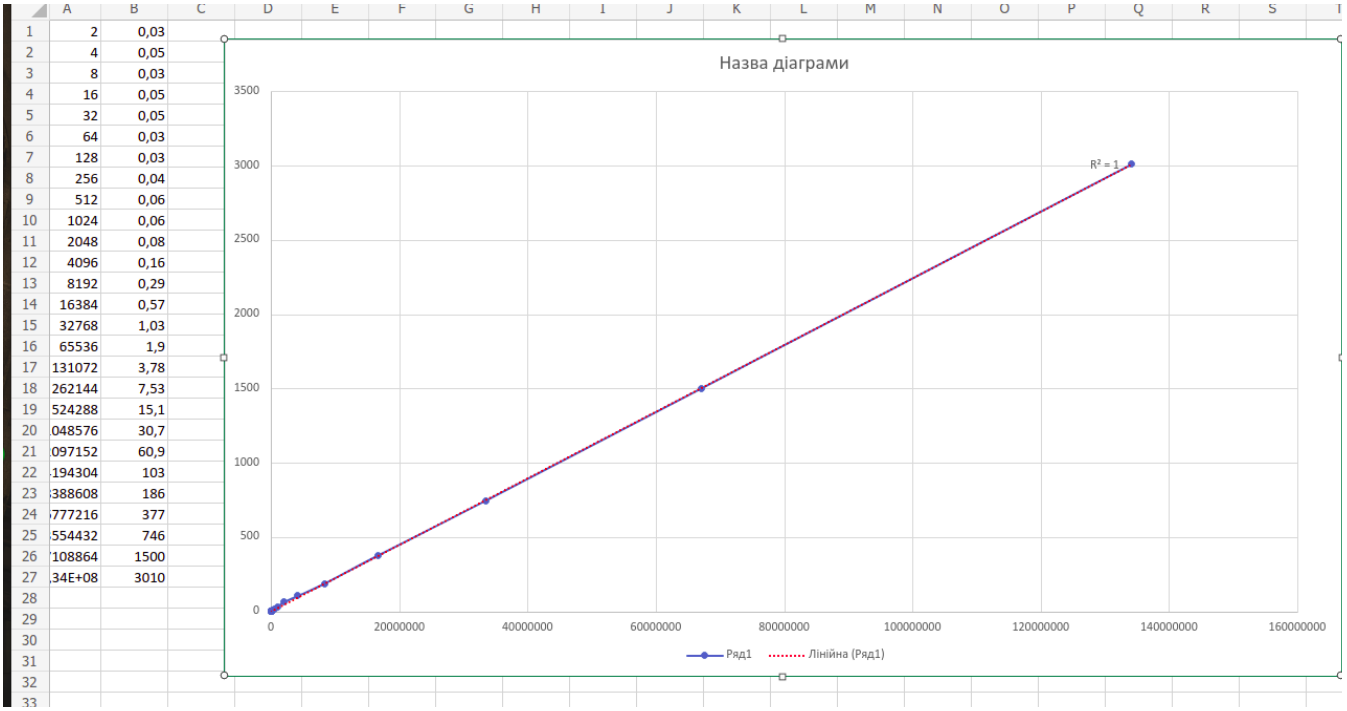
```
package commands

import (
	"fmt"
	"strings"
	"testing"
```

```
	"github.com/zavad4/go/lab4/engine"
)

var constInput = "printc"
var command engine.Command

func BenchmarkCount(b *testing.B) {
	length := 1
	for i := 0; i < 27; i++ {
		length = 2 * length
		input := constInput
		input += strings.Repeat("A", length)

		b.Run(fmt.Sprintf("len=%d", length), func(b
*testing.B) {
			command = Parse(input)
		})
	}
}
```

Його запуск дає нам наступні точки:

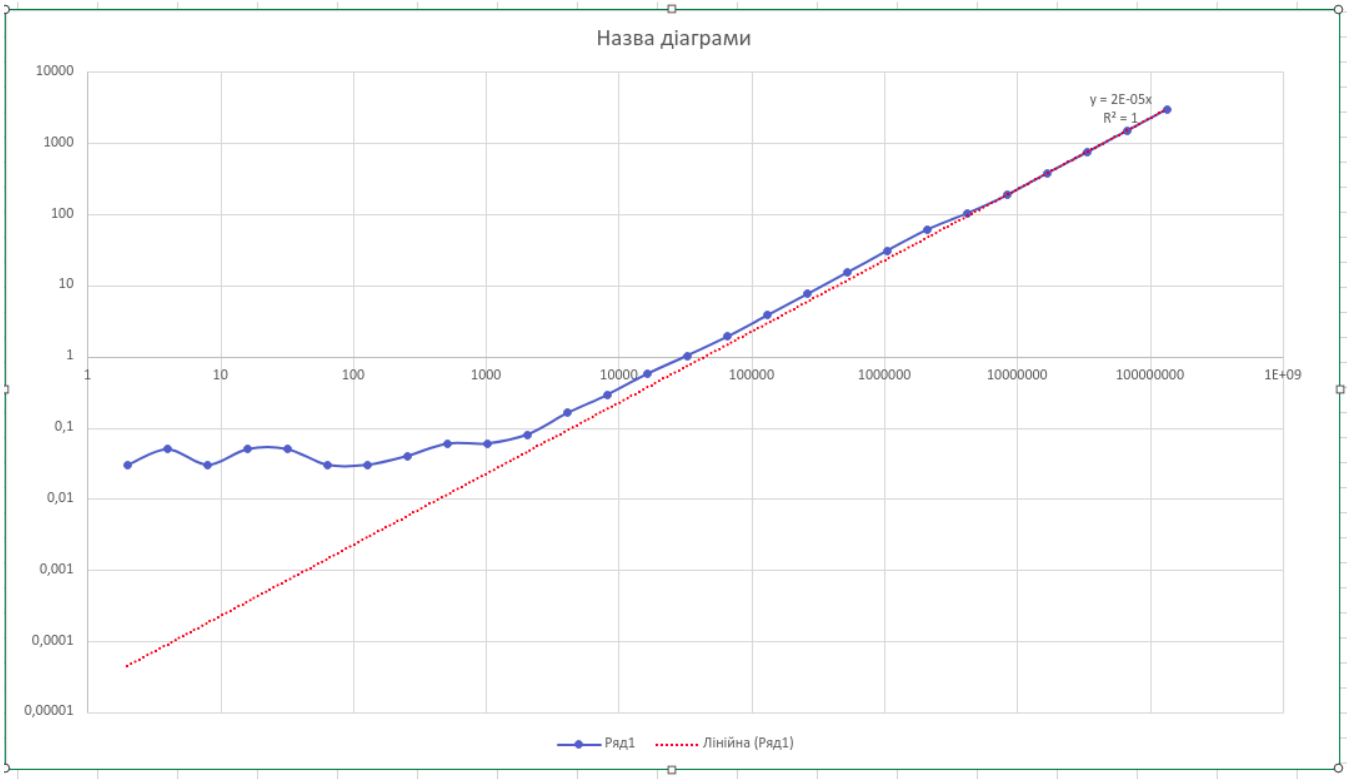
```
kertnik@slave-node:~/Documents/go/lab4/commands$ go test -bench . - cpu 1
goos: linux
goarch: amd64
pkg: github.com/zavad4/go/lab4/commands
BenchmarkCount/len=2-8          1000000000               0.000003 ns/op
BenchmarkCount/len=4-8          1000000000               0.000005 ns/op
BenchmarkCount/len=8-8          1000000000               0.000003 ns/op
BenchmarkCount/len=16-8         1000000000               0.000005 ns/op
BenchmarkCount/len=32-8         1000000000               0.000005 ns/op
BenchmarkCount/len=64-8         1000000000               0.000003 ns/op
BenchmarkCount/len=128-8        1000000000               0.000003 ns/op
BenchmarkCount/len=256-8        1000000000               0.000004 ns/op
BenchmarkCount/len=512-8        1000000000               0.000006 ns/op
BenchmarkCount/len=1024-8       1000000000               0.000006 ns/op
BenchmarkCount/len=2048-8       1000000000               0.000008 ns/op
BenchmarkCount/len=4096-8       1000000000               0.000016 ns/op
BenchmarkCount/len=8192-8       1000000000               0.000029 ns/op
BenchmarkCount/len=16384-8      1000000000               0.000057 ns/op
BenchmarkCount/len=32768-8      1000000000               0.000103 ns/op
BenchmarkCount/len=65536-8      1000000000               0.000190 ns/op
BenchmarkCount/len=131072-8     1000000000               0.000378 ns/op
BenchmarkCount/len=262144-8     1000000000               0.000753 ns/op
BenchmarkCount/len=524288-8     1000000000               0.00151 ns/op
BenchmarkCount/len=1048576-8    1000000000               0.00307 ns/op
BenchmarkCount/len=2097152-8    1000000000               0.00609 ns/op
BenchmarkCount/len=4194304-8    1000000000               0.0103 ns/op
BenchmarkCount/len=8388608-8    1000000000               0.0186 ns/op
BenchmarkCount/len=16777216-8   1000000000               0.0377 ns/op
BenchmarkCount/len=33554432-8   1000000000               0.0746 ns/op
BenchmarkCount/len=67108864-8   1000000000               0.150 ns/op
BenchmarkCount/len=134217728-8  1000000000               0.301 ns/op
PASS
ok      github.com/zavad4/go/lab4/commands      8.474s
```

Побудуємо графіки:



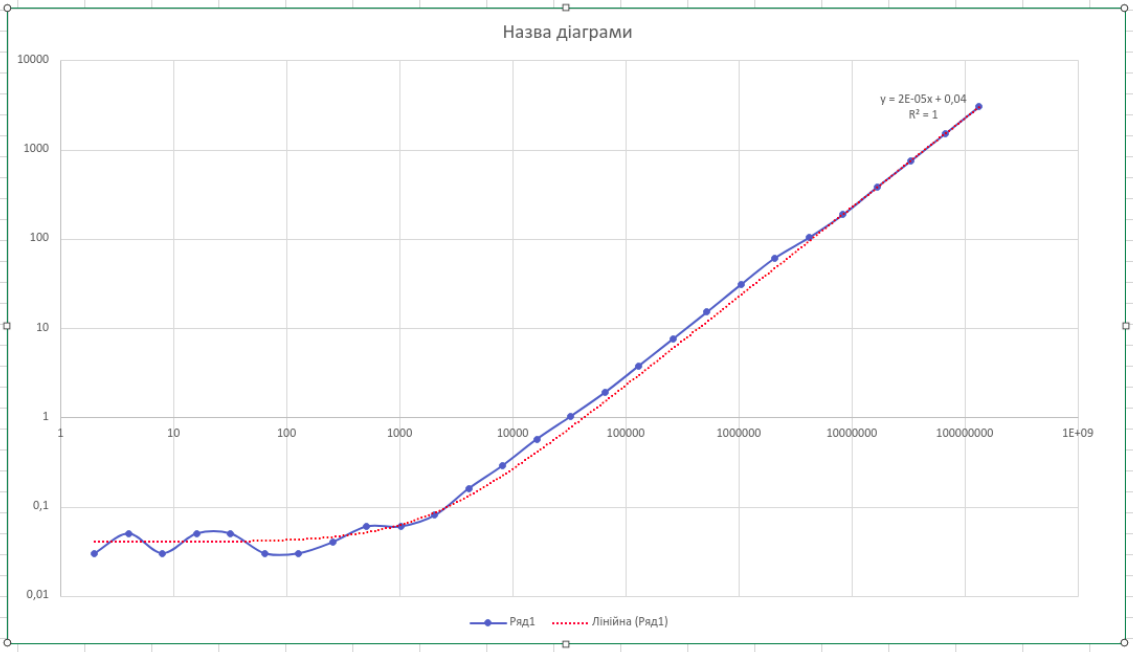
	A	B
1	2	0,03
2	4	0,05
3	8	0,03
4	16	0,05
5	32	0,05
6	64	0,03
7	128	0,03
8	256	0,04
9	512	0,06
10	1024	0,06
11	2048	0,08
12	4096	0,16
13	8192	0,29
14	16384	0,57
15	32768	1,03
16	65536	1,9
17	131072	3,78
18	262144	7,53
19	524288	15,1
20	048576	30,7
21	097152	60,9
22	194304	103
23	388608	186
24	777216	377
25	554432	746
26	108864	1500
27	34E+08	3010
28		
29		
30		
31		
32		
33		

На цьому зображені помітно як емпіричну функцію (суцільна лінія синього кольору), так і її лінійне наближення (штрихована лінія червоного кольору). Як і для другої лабораторної роботи, ці лінії дуже близькі одна до одної, а ймовірність правильності висновку про лінійність функції майже дорівнює одиниці.



Як і в другій лабораторній, при переході до логарифмічної шкали, бачимо відхилення від прямої для малих значень. Вони є вищими за неї (що пояснюється наявністю дій, що не залежать від довжини вхідних даних), та мають нерівномірне зростання (вплив випадкових процесів).

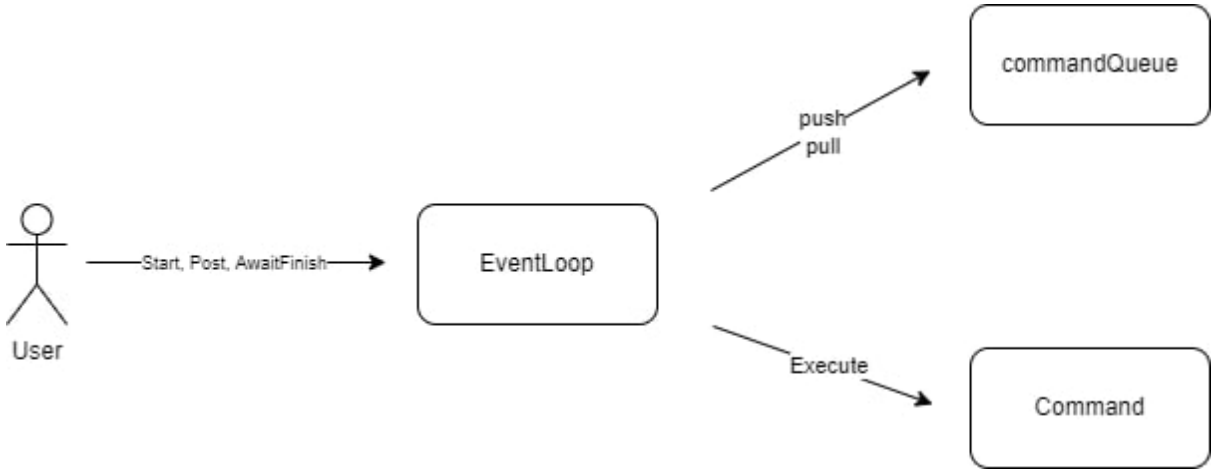
	A	B
1	2	0,03
2	4	0,05
3	8	0,03
4	16	0,05
5	32	0,05
6	64	0,03
7	128	0,03
8	256	0,04
9	512	0,06
10	1024	0,06
11	2048	0,08
12	4096	0,16
13	8192	0,29
14	16384	0,57
15	32768	1,03
16	65536	1,9
17	131072	3,78
18	262144	7,53
19	524288	15,1
20	048576	30,7
21	097152	60,9
22	194304	103
23	388608	186
24	777216	377
25	554432	746
26	108864	1500
27	34E+08	3010



Враховуючи виконання константних команд, отримуємо наступну апроксимацію. Час виконання таких дій приблизно рівний 0.004 пікосекунди.

Оскільки явних команд, що не залежать від довжини вхідних даних немає, на відміну від другої лабораторної, то їх час виконання зменшився.

Тип 2: побудувати діаграму взаємодії



Для 4 лабораторної роботи ми створили додаток з трьома основними компонентами:

1) Eventloop

Саме ця компонента взаємодіє з користувачем. Після вводу даних, програма створює екземпляр Eventloop’у із трьома заданими методами:

- Start - цей метод створює нову порожню чергу з команд
- Post - додає команду з вхідними даними у чергу
- AwaitFinish - очікує завершення роботи програми та віддає результат її виконання

2) Command

Ця компонента відповідає за виконання одного кроку Eventloop’у і має заданий метод Execute - отримання парсера та виконання певних дій над командою за його допомогою.

3) CommandQueue

Ця компонента є чергою команд на виконання. Як і звичайна черга, вона має заданими два основні методи:

- push - додати елемент до черги
- pull - дістати перший елемент з черги