Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ КОМП’ЮТЕРНИХ НАУК ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Катедра «Комп’ютерна інженерія та програмування»

ЗВІТ

про виконання лабораторної роботи №2

з навчальної дисципліни «Алгоритми та структури даних»

Варіант 5

Виконав студент:

Омельніцький Андрій Миколайович
Група: КН-1023б

Перевірив:

Старший викладач

Бульба Сергій Сергійович

Харків-2024

Зміст

1 Мета роботи

Набути навичок та практичного досвіду у розробці рекурсивних програм.

Теми для попередньої роботи:

- ітераційні алгоритми;
- рекурсивні алгоритми.

2 Завдання

2.1 Пункти завдання

- Розробити рекурсивний та ітераційний алгоритми розв'язання індивідуального завдання.
- Визначити та порівняти час виконання відповідних функцій, зробити висновки.

2.2 Завдання за варіантом (5)

Розробити програму, що для заданого n будує трикутник Паскаля
Коефіцієнти, що утворюють трикутник Паскаля, визначаються так:

$$C(n,0) = 1;$$
$$C(n,n) = 1; (n > 0)$$
$$C(n,k) = C(n–1,k–1) + C(n–1,k); (n > 0, m > 0)$$

3 Хід виконання

Для виконання завдання було обрано мову Rust. Увесь код також додатково був розміщений в GitHub репозитарії: https://github.com/blackgolyb/algos-labs.

3.1 Програмна реалізація рекурсивного алгоритму

Якщо робити наївну реалізацію цього алгоритму, тоді ми отримаємо дуже погані результати по часовій складності алгоритма. А саме $O(2^n)$ для функції підрахунку біноміального коєфіцієнта та $O(n^2)$ для функції побудови самого трикутника, а отже фінальна складність алгоритму ставновить $O(2^n * n^2)$.

Але якщо дотати кешування результатів функції біноміального коєфіцієнта, то в нас вийде прийти до складності $O(n^2)$. Підвищення продуктивності відбувається за рахунок того, що кожна унікальна пара n та k обчислюється лише один раз для функції binome, проте вона все ж не буде дуже ефективна через накладні витрати на постійну взаємодію з хеш-таблицею.

Код програми алгоримта:

```
use cache_macro::cache;
use lru_cache::LruCache;

#[cache(LruCache : LruCache::new(1000))]
fn binome(n: u64, k: u64) -> u64 {
    if n == 0 || k == 0 || k == n {
        return 1;
    }
    binome(n - 1, k - 1) + binome(n - 1, k)
}

pub fn pascale_triangle(n: u64) {
    fn draw(n: u64, line: u64, i: u64) {
        if line == n {
            return;
        }

        let t = binome(line, i);
        print!("{} ", t);

        if i == line {
            println!();
            draw(n, line + 1, 0);
        } else {
            draw(n, line, i + 1);
        }
    }
    draw(n, 0, 0);
}
```

3.2 Програмна реалізація ітераційного алгоритму

Переписавши минулу рекурсивну версію під ітераційну, не змінуючи сильно підхід, можна отримати такий результат. Для побудови усього трикутника все також потрібно $O(n^2)$, але для обрахування біноміального коефіцієнта треба всього $O(N)$ в гіршому випадку. Тому загальна складність буде мати приблизно $O(n^3)$.
Код програми алгоримта:

```
fn binome(n: u64, k: u64) -> u64 {
    let mut res = 1;
    for i in 0..k {
        res = res * (n - i) / (i + 1);
    }
    res
}

pub fn pascale_triangle(n: u64) {
    let mut t: u64;
    for line in 0..n {
        for i in 0..=line {
            t = binome(line, i);
            print!("{} ", t);
        }
        println!();
    }
}
```

3.3 Програмна реалізація ефективного ітераційного алгоритму

Але можна ще краще, тому ми можемо відійти від минулого підходу та спробувати рахувати значення в трикунику, спираючись на попередні значення.

Таким чином ми зможемо досягти $O(n^2)$ в цілому, але виростуть витрати по пам'яті, а саме вони будуть становити $2n$ або ж $O(n)$. Тому треба обирати що нам важливіше: витрати за пам'ятю чи по часу.
Код програми алгоримта:

```
pub fn pascale_triangle(n: usize) {
    let mut line = vec![1; n];
    let mut buffer = vec![1; n];

    if n == 0 {
        return;
    }

    println!("{}", buffer[0]);

    for i in 1..n {
        print!("{}␣", buffer[0]);
        for j in 1..i {
            buffer[j] = line[j - 1] + line[j];
            print!("{}␣", buffer[j]);
        }
        println!("{}", buffer[i]);
        (line, buffer) = (buffer, line);
    }
}
```

3.4 Приклад роботи програми

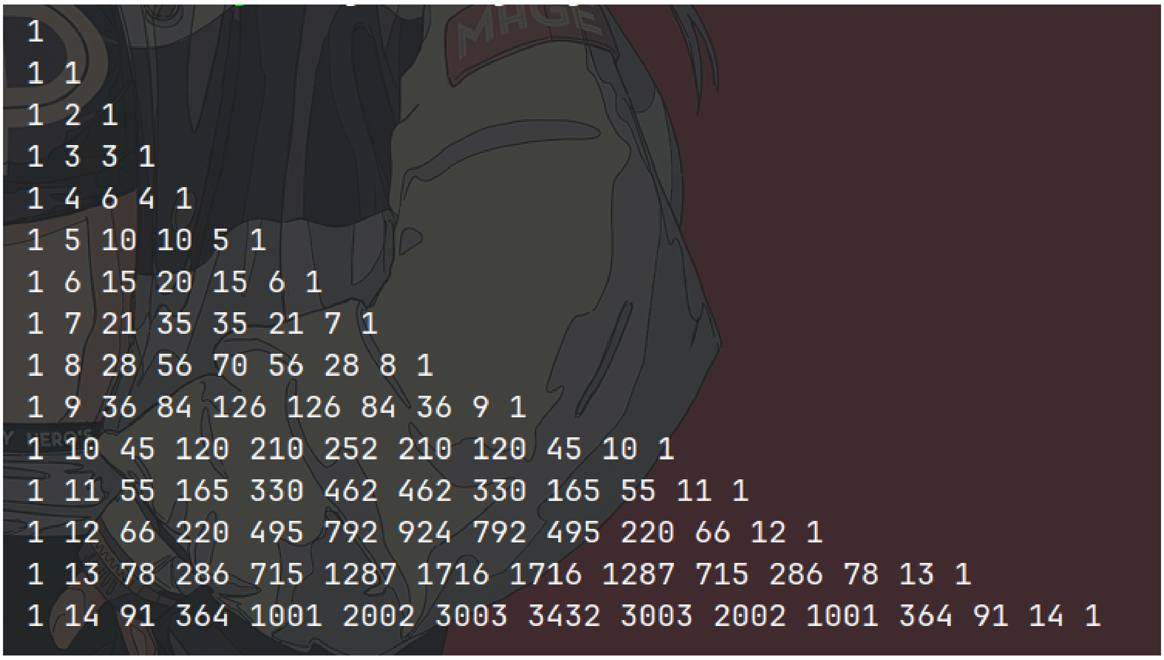

Рис. 1. Трикутник паскаля для $n = 15$

3.5 Порівняння усіх алгоритмів

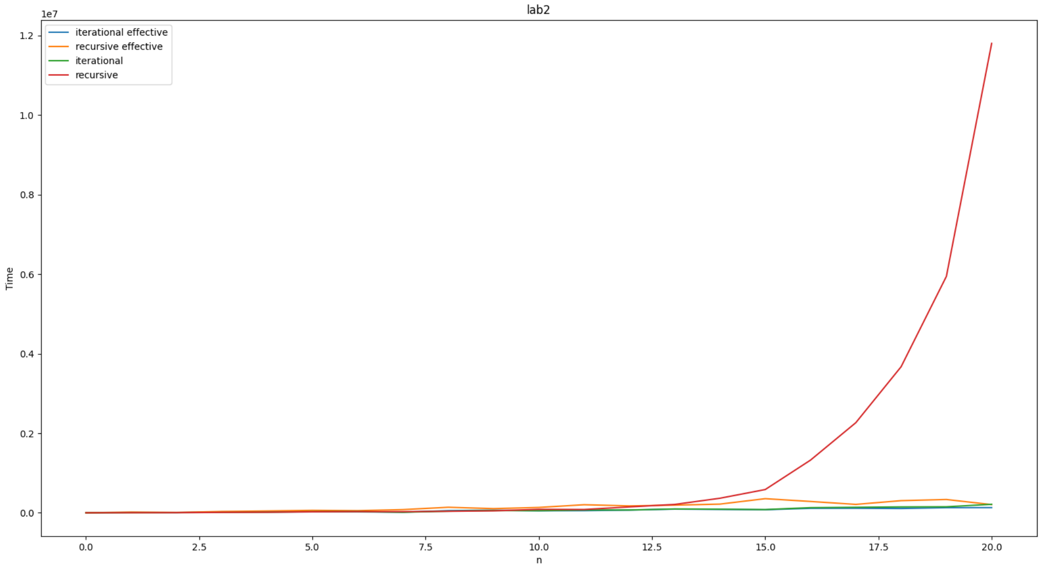

Рис. 2. Порівняння усіх алгоритмів при $n = 20$

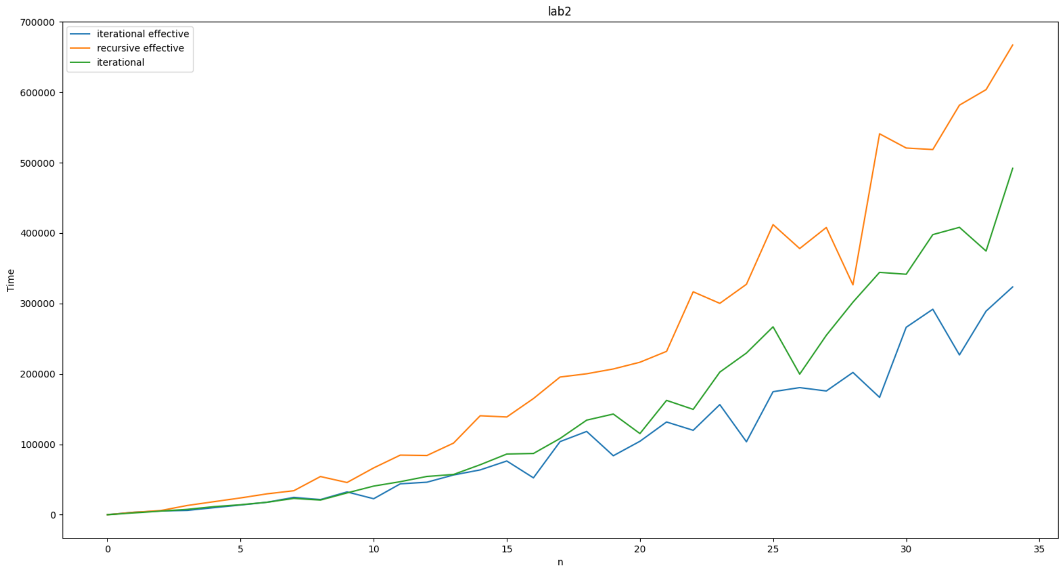

Рис. 3. Порівняння усіх алгоритмів окрім не ефективного рекурсивного при $n = 34$

4 Висновки

В ході виконання лабораторної робити було порівняно складність роботи ітераційного та рекурсивного варіанту алгоримту, а також знайдено найоптимальніший алгоритм з розібраних, їм виявився ефективний варіант ітераційного алгоритму, проте він потребує $O(n)$ додаткового місця. Тому якщо потрібно $O(1)$ місця, то буде краще скористатися наівною реалізацією ітераційного алгоритму.