

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА  
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА КІБЕРНЕТИКИ

**В. А. Колесников**

**Д. А. Ключин**

## **МЕТОДИЧНІ РЕКОМЕНДАЦІЇ**

ДО ЛАБОРАТОРНИХ РОБІТ З ДИСЦИПЛІНИ

“ОБ’ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ”

ДЛЯ СТУДЕНТІВ ФАКУЛЬТЕТУ КОМП'ЮТЕРНИХ НАУК ТА  
КІБЕРНЕТИКИ

Методичні рекомендації до лабораторних робіт з дисципліни “Об’єктно-орієнтоване програмування” для студентів факультету комп’ютерних наук та кібернетики / В. А. Колесников, Д. А. Ключин. — К.: Київський національний університет імені Тараса Шевченка, 2026. — XX с.

Укладачі:

В. А. Колесников, доктор філософії, асистент кафедри обчислювальної математики факультету комп’ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка.

Д. А. Ключин, доктор фізико-математичних наук, професор, професор кафедри обчислювальної математики факультету комп’ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка.

Рецензенти:

В. М. Терещенко, доктор фізико-математичних наук, завідувач кафедри математичної інформатики факультету комп’ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка.

С. С. Шкільняк, доктор фізико-математичних наук, професор кафедри теорії та технології програмування факультету комп’ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка.

Ухвалено на засіданні науково-методичної комісії факультету комп’ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка, протокол № XX від XXXXXX.

Рекомендовано до друку на засіданні вченої ради факультету комп’ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка, протокол № 10 від XXXXXX.

## ЗМІСТ

<b>Подяки</b>	<b>5</b>
<b>Вступ</b>	<b>6</b>
<b>Розділ 1. Лабораторна робота №1: Генерування псевдовипадкових чисел</b>	<b>8</b>
1.1. Умова . . . . .	9
1.2. Види розподілів . . . . .	10
1.3. Клас Rand та його нащадки . . . . .	13
1.4. Алгоритми генерації псевдовипадкових чисел . . . . .	18
1.5. Генерація вибірки та побудова гістограм . . . . .	20
<b>Розділ 2. Лабораторна робота №2: Арифметика довгих чисел</b>	<b>22</b>
2.1. Умова . . . . .	23
2.2. Представлення невід’ємних довгих цілих чисел за допомогою класу ULI . . . . .	23
2.3. Шкільні алгоритми . . . . .	27
2.4. Швидке множення: алгоритм Карацуби . . . . .	29
2.5. Швидке множення: алгоритм Тоома-Кука . . . . .	32
2.6. Швидке множення: алгоритм Шенхаге . . . . .	35
2.7. Швидке множення: алгоритм Штрасена . . . . .	38
2.8. Швидке перетворення Фур’є . . . . .	42
2.9. Швидке множення: алгоритм Штрасена: приклад . . . . .	44
2.10. Знаходження оберненого числа та швидке ділення . . . . .	50
2.11. Рекомендації до написання лабораторної роботи без використання бібліотеки STL . . . . .	52
2.12. Алгоритми перевірки на простоту . . . . .	53

<b>Розділ 3. Лабораторна робота №3: Алгоритми обчислювальної геометрії</b>	<b>58</b>
3.1. Умова . . . . .	59
3.2. Діаграма Вороного, алгоритм Форчуна та триангуляція Делоне	59
3.3. Алгоритм Форчуна: Загальний опис . . . . .	61
3.4. Алгоритм Форчуна: Подвійно зв'язаний список ребер . . . . .	70
3.5. Алгоритм Форчуна: Берегова лінія . . . . .	74
3.6. Алгоритм Форчуна: Черга подій . . . . .	80
3.7. Рекомендації до реалізації алгоритма Форчуна без використання бібліотеки STL . . . . .	81
3.8. Алгоритм Форчуна: приклад . . . . .	82
3.9. Побудова триангуляції Делоне за допомогою діаграми Вороного	103
3.10. Алгоритми побудови опуклої оболонки . . . . .	105
<b>Завдання</b>	<b>113</b>
<b>Список використаних джерел</b>	<b>116</b>

## ПОДЯКИ

Автори висловлюють вдячність асистентам кафедри обчислювальної математики факультету комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка Тимошенку Андрію Анатолійовичу та Оноцькому В'ячеславу Валерійовичу за сумлінне виконання обов'язків викладачів занять з дисципліни “Об'єктно-орієнтоване програмування” для студентів другого курсу напрямків “Прикладна математика” та “Системний аналіз”.

Також автори висловлюють окрему подяку Токарю Костянтину Сергійовичу за допомогу в формулюванні алгоритмів та оформленні коду і прикладів.

## ВСТУП

Ця методична розробка призначена для студентів факультету комп'ютерних наук та кібернетики, щоб допомогти їм у написанні лабораторних робіт з курсу “Об’єктно-орієнтоване програмування”. Даний курс є складовою освітньо-наукової програми рівня бакалавру для напрямків “Прикладна математика” та “Системний аналіз”. Представлені в даному документі лабораторні роботи пропонуються студентам до виконання вже упродовж багатьох років. Різноманітність алгоритмів, як по темам, так і по рівню складності для реалізації, дають можливість студентам перевірити свої навички програмування та розширити свої знання з відповідних напрямків. Лабораторні роботи “Генерування псевдовипадкових чисел”, “Арифметика довгих чисел” та “Алгоритми обчислювальної геометрії” містять набори ключових алгоритмів зі своїх напрямків, що дозволяє студенту ознайомитися з основними принципами, ідеями та проблемами під час роботи над програмами відповідної тематики. Частина алгоритмів є тривіальною, проте деякі з них є настільки складними, що процедура їхнього розбору може займати не одне лабораторне заняття. Метою написання даної методичної розробки є висвітлення ключових аспектів кожної з вищезгаданих лабораторних робіт, більш чітко формулювання вимог до задачі лабораторної та опис і пояснення важливих нюансів в алгоритмах, які неодноразово викликали питання з боку студентів.

Алгоритми вищезгаданих лабораторних робіт пропонується реалізувати мовою програмування C++, тому у тексті документу також будуть присутні блоки коду цією мовою, які можна використовувати в якості бази для написання власного коду. Дані фрагменти, хоч і будуть відображати деякі ключові аспекти тих чи інших алгоритмів, тим не менше, не будуть

повними. Це означає, що для того, щоб присутній у документі код компілювався, його необхідно буде доповнити власним кодом. Дана методична розробка не має на своїй меті представлення зразку реалізації всіх алгоритмів вищезгаданих лабораторних робіт. Присутній в документі код також буде слугувати зразком дотримання парадигми об'єктно-орієнтованого програмування. В деяких місцях буде пояснюватися, чому і як використовуються принципи даної парадигми: інкапсуляція, наслідування та поліморфізм.

Також присутні в методичній розробці міркування стосовно реалізації алгоритму та самого коду не будуть спрямовані на оптимізацію обчислень. Головною задачею під час розбору та реалізації алгоритмів, окрім коректності, буде отримання вірної асимптотичної оцінки їхньої роботи. Проте в деяких місцях, де оптимізацію обчислень можна зробити очевидно та просто, ідеї стосовно даної оптимізації будуть також висвітлені.

Значна частина алгоритмів вищезгаданих лабораторних робіт використовує контейнери даних, частина з яких вже реалізована в стандартній бібліотеці STL. І хоча автори задля більш глибокого розуміння роботи даних контейнерів наполегливо рекомендують спробувати реалізувати їх самостійно, основна частина рекомендацій буде стосуватися саме роботи з контейнерами з бібліотеки STL. Такі контейнери як `std::vector` або `std::map` мають достатньо повний функціонал для використання їх в представлених алгоритмах. Проте під час розбору алгоритмів 2 та 3 лабораторних робіт також будуть описані рекомендації стосовно використання самостійно реалізованих контейнерів, тому студент може сам обрати, як саме йому зручніше працювати зі структурами даних.

Автори сподіваються, що дана методична розробка допоможе студентам якісніше здавати лабораторні роботи та підвищить їхній рівень знань стосовно представлених в даних роботах алгоритмів.

## РОЗДІЛ 1

### ЛАБОРАТОРНА РОБОТА №1: ГЕНЕРУВАННЯ ПСЕВДОВИПАДКОВИХ ЧИСЕЛ

Випадкові величини є важливою частиною програмування. Вони активно використовуються під час тестування програм для генерації вхідних даних. Також вони є невід'ємною складовою програм для моделювання процесів, які мають стохастичну природу, таких як популяційні алгоритми та випадкові блукання. І, наостанок, випадкові величини є ключовою частиною так званих стохастичних алгоритмів, клас яких охоплює майже всі задачі, які можна розв'язати за допомогою програмування. Поняття випадкової величини є доволі складним поняттям теорії ймовірностей, але для подальшого розгляду достатньо розуміти випадкову величину як таку, кожен наступну реалізацію якої неможливо передбачити детермінованими алгоритмами.

Тут виникає певне протиріччя, оскільки щоб використовувати випадкові величини, тобто генерувати конкретні реалізації таких величин, ми маємо використовувати програмне забезпечення обчислювальної машини, де всі програми та операції, навпаки, мають бути передбачуваними та детермінованими. Звісно, можна спробувати вийти за межі програмного забезпечення та використовувати апаратне забезпечення, в якому за рахунок доволі складних фізичних процесів згідно з принципами квантової механіки може виникати справжня випадковість. Ця задача частково вирішена за допомогою `std::random_device`.

Проте іноді буває так, що немає можливості використовувати апаратне забезпечення або ж немає гарантій того, що воно буде працювати так, як користувачу цього б хотілося. В такому випадку вищезгадане протиріччя

розв'язується за допомогою так званих псевдовипадкових чисел. Ці числа поєднують в собі дві основні характеристики, які нам потрібні: вони генеруються за допомогою детермінованого алгоритму та послідовність таких чисел “виглядає” як послідовність реалізацій деякої випадкової величини. В рамках цієї методичної розробки ми не будемо поглиблюватися в тему того, як перевіряти послідовність чисел на випадковість, і будемо оцінювати випадковість “на око”. Для цього нам потрібно буде згенерувати достатню кількість псевдовипадкових чисел та поєднати їх в гістограму частот.

### 1.1. Умова

Написати програму, що реалізує десять методів генерації псевдовипадкових чисел.

1. Лінійний конгруентний метод.
2. Квадратичний конгруентний метод.
3. Метод чисел Фібоначчі.
4. Обернений конгруентний метод.
5. Метод об'єднання.
6. Правило 3-сігма.
7. Метод полярних координат.
8. Метод співвідношень.
9. Метод логарифму.
10. Метод Аренса.

Для кожного методу побудувати гістограму, яка ілюструє розподіл псевдовипадкових чисел.

## 1.2. Види розподілів

Не всі випадкові величини є однаковими. Результат підкидання монетки відрізняється від результату підкидання грального шестигранного кубика як мінімум кількістю можливих результатів. Проте, навіть якщо обмежитися гральним шестигранним кубиком, можна розглядати дві різні випадкові величини, якщо взяти два кубика з різними центрами мас. Частота випадання граней в такому випадку буде неоднаковою. Для того, щоб розрізнити такі випадки, в теорії ймовірностей вводиться поняття розподілу, яке повністю описує поведінку випадкової величини. Взагалі, існує незліченна кількість розподілів, проте в рамках даної лабораторної буде розглядатися лише 4 найбільш розповсюджені: рівномірний, нормальний (він же гаусівський) експоненційний та гама-розподіл. Реалізацією випадкової величини, яка має будь-який з цих розподілів, є дійсне число, проте у кожного з розподілів є своя область дійсних чисел, які потенційно можуть бути реалізацією відповідної випадкової величини.

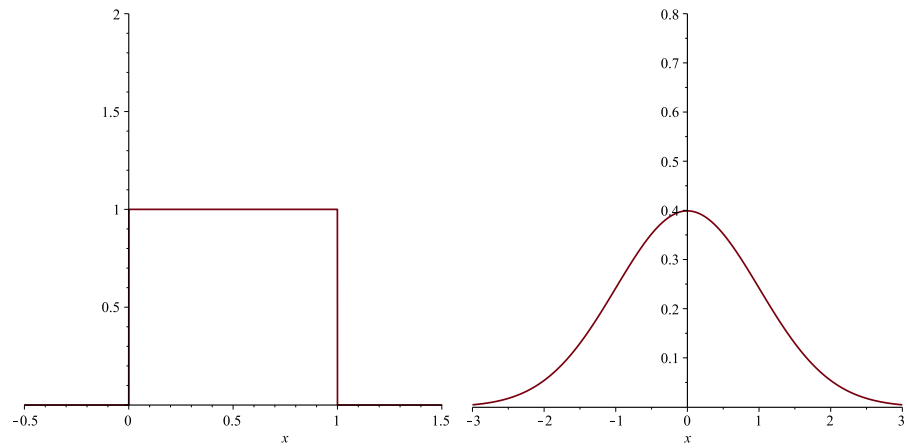
Рівномірний розподіл є найбільш простим та інтуїтивно зрозумілим. Суть цього розподілу полягає в тому, що при реалізації кожен раз вибирається “навмання” точка з проміжку  $[0, 1]$ . Це означає, що ймовірність обрати число, менше за 0.5, рівна ймовірності обрати число, більше за 0.5. В той же час ймовірність обрати число з першої третини проміжку така ж сама, як і ймовірність обрати число з другої або третьої третини, проте менша, ніж з другої половини проміжку. Узагальнюючи, можна сказати, що ймовірність вибрати число з проміжку  $[a, b]$  ( $0 < a < b < 1$ ) пропорційна лише довжині цього проміжку  $b - a$ . Зрозуміло, що такий розподіл можна “перенести” на будь-який скінченний проміжок  $[A, B]$  замість  $[0, 1]$ . І саме цей розподіл мається на увазі, коли кажуть “візьмемо випадкове дійсне число з проміжку  $[A, B]$ ”.

Нормальний, або ж гаусівський (але не “нормальний гаусівський”, це

тавтологія), розподіл доволі часто виникає під час збору даних до статистики. Більшість соціальних або біологічних показників підпорядковуються цьому розподілу. Зріст, вага, коефіцієнт інтелекту, температура тіла — всі ці показники, зібрані з великої сукупності людей, кожен по-своєму буде нагадувати певний нормальний розподіл. Характерною особливістю даного розподілу є явно виражене середнє значення та симетричність відносно цього значення. Експоненційний та гама-розподіл також мають свої важливі застосування, з якими можна ознайомитися в літературі з відповідної тематики.

Всі перелічені тут розподіли є неперервними. Точне визначення неперервного розподілу можна знайти в будь-якому підручнику з теорії ймовірностей, проте для наших цілей достатньо знати, що такі розподіли мають щільність, яка їх повністю характеризує. Щільність — це невід’ємна функція дійсної змінної, інтеграл якої по всій дійсній прямій дорівнює 1, така що чим більше значення щільності у точці  $x$ , тим вища ймовірність при реалізації випадкової величини з відповідним розподілом обрати число з околу  $x$ . На рис. 1.1. зображені функції щільностей для чотирьох вищезгаданих розподілів.

Поруч з неперервними розподілами існують дискретні розподіли. Вони характеризуються тим, що множина їхніх потенційних реалізацій не більш ніж зліченна, а кожна така реалізація має певну додатню ймовірність, так що загальна сума цих ймовірностей дорівнює 1. Неперервні розподіли в певному сенсі можуть наближатися дискретними розподілами, і ця ідея полягає в основі першої лабораторної роботи. Оскільки в програмуванні ми не можемо оперувати будь-якими дійсними числами, а лише певною множиною таких (лише деякими раціональними), то для генерації випадкової величини з неперервним розподілом буде використовуватися дискретна випадкова величина. Ця дискретна випадкова величина в свою чергу буде моделюватися за допомогою генератора псевдовипадкових чисел.



(а) Рівномірний.

(б) Стандартний нормальний.

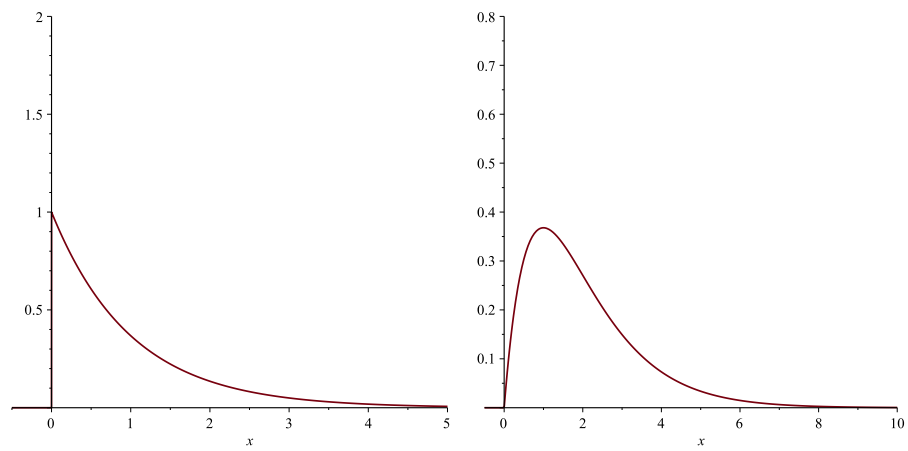
(в) Експоненційний ( $\lambda = 1$ ).(г) Гама-розподіл ( $a = 2$ ).

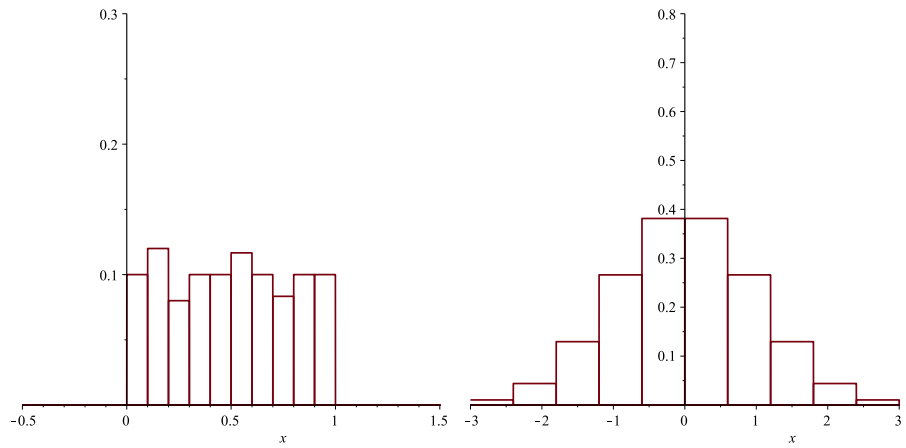
Рис. 1.1: Графіки функцій щільності для розподілів різних типів.

Таким чином, за допомогою вказаних в першій лабораторній роботі алгоритмів генерації псевдовипадкових чисел для кожного з вищенаведених розподілів буде генеруватися набір реалізацій. Після цього даний набір повинен бути об'єднаним в гістограму частот. Для цього для кожного розподілу спочатку визначається множина  $Dom_f = \{x \in \mathbb{R} \mid f(x) > 0\}$  ( $f$  — щільність), потім за необхідності вона обмежується скінченним проміжком, ділиться на наперед задану кількість однакових підпроміжків і для кожного такого підпроміжку рахується кількість реалізацій з набору, значення яких потрапило в даний підпроміжок. На основі результатів будуться прямокутники, висота яких пропорційна отриманим на попередньому кроці числам. Приклади коректних гістограм для чотирьох розподілів наведено на рис. 1.2. Варто зазначити, що силует гістограми нагадує силует функції щільності, що в межах поточної лабораторної роботи є критерієм коректно запрограмованого алгоритму.

### 1.3. Клас `Rand` та його нащадки

Перш ніж розглядати конкретні алгоритми генерації псевдовипадкових чисел, звернімося до стратегії їхньої реалізації в коді. Випадкову величину можна розглядати як певний об'єкт, який при зверненні до нього повертає псевдовипадкове число. Очевидним вибором для реалізації такого об'єкту є функтор — клас, що має перевантажений оператор `()`. Причому будь-яка випадкова величина, незалежно від типу розподілу або алгоритму генерації чисел, працюватиме за таким принципом. Це означає, що ми можемо спробувати поєднати всі випадкові величини за допомогою принципу наслідування.

Розглянемо абстрактний клас `Rand`, єдиним методом якого буде перевантажений оператор круглі дужки `()`. Оскільки даний клас узагальнює всі розподіли, то оператор `()` ми залишаємо абстрактним.



(а) Рівномірний.

(б) Стандартний нормальний.

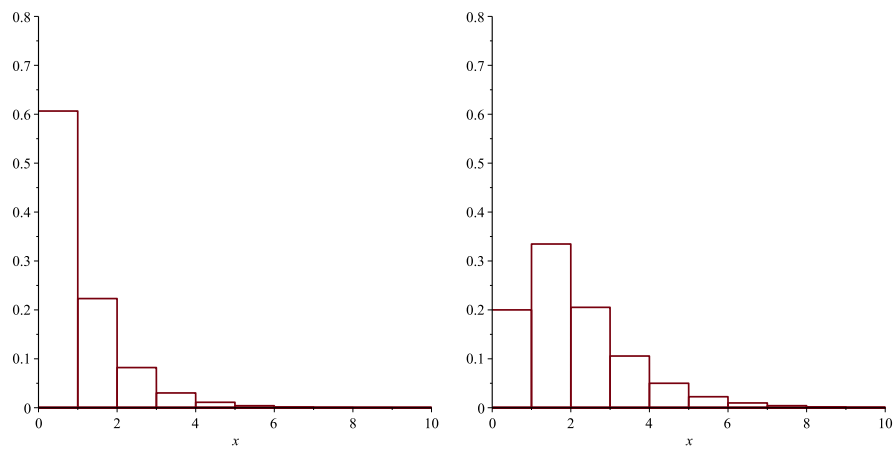
(в) Експоненційний ( $\lambda = 1$ ).(г) Гама-розподіл ( $a = 2$ ).

Рис. 1.2: Зразки гістограм для розподілів різних типів.

```

class Rand {
    public:
        virtual double operator()() = 0;

    private:
        double Dom_min;
        double Dom_max;
};

```

Тоді класи випадкових генераторів, що відповідають кожному з розподілів, можна зробити нащадками визначеного класу `Rand`. Для кожного з класів-нащадків оператор `()` вже повинен бути реалізованим. Оскільки ми розглядаємо лише дійснозначні випадкові величини, то в клас `Rand` можна винести також і границі множини  $Dom_f$ , бо ці характеристики присутні в усіх розподілах. Для доступу до них краще реалізувати відповідні геттери та сеттери.

```

class RandType1 : public Rand {
    public:
        double operator()() override { ... }
};

```

Тоді наш проект буде складатися з одного класу-предка та чотирьох класів-нащадків: по одному для кожного виду розподілу. Проте, як можна побачити з умови лабораторної, для одного типу розподілу може існувати декілька різних алгоритмів генерації цього розподілу. Тоді кожен такий алгоритм може бути реалізований як окремий клас, який може бути переданий як шаблонний параметр до відповідного класу розподілу.

```

class RandAlg {
    ...

    private:
        //parameters
};

```

```

template <class T>
class RandType1 : public Rand {
    public:
        double operator()() override { ... }
};

```

Проте в даному випадку виникає проблема, як реалізувати заборону підставляти у клас розподілу в якості шаблонного параметру клас алгоритму, який цьому розподілу не відповідає. При цьому також шаблонний клас розподілу стає просто обгорткою для класу алгоритму, не додаючи жодного власного функціоналу. До того ж, з використанням шаблонів код стає більш громіздким.

Для того, щоб одночасно вирішити ці проблеми, можна сприймати клас Rand як абстракцію, що містить саме **метод** генерації випадкової величини з будь-яким розподілом. Тоді кожен нащадок цього класу буде напряму відповідати конкретному алгоритму, а дані про розподіл, який таким чином генерується, можна за потреби зберігати в цьому класі, або ж, якщо такої потреби немає, обмежитися вибором інформативної назви для самого класу-нащадка.

Так, перші 4 алгоритми лабораторної роботи можуть бути названі наступним чином:

```

class RandUniLinear : public Rand {
    ...
};

class RandUniQuadr : public Rand {
    ...
};

class RandUniFib : public Rand {
    ...
};

```

```
class RandUniInverse : public Rand {
    ...
};
```

Якщо переходити ближче до самих алгоритмів, то можна помітити, що кожен з них використовує певний набір параметрів, які можуть обиратися користувачем. Також для кожного алгоритму можна написати конструктор за замовчуванням, де в якості параметрів алгоритму обираються обрані програмістом константи. Зрозуміло, що ці константи повинні бути такими, щоб алгоритм генерував якомога якіснішу послідовність псевдовипадкових чисел. Також не варто забувати, що разом з конструктором за замовчуванням варто ще реалізовувати конструктор копіювання та деструктор.

```
class RandUniLinear {
public:
    Rand();
    Rand(double a, ...);

    double operator()() override;
    ...
};
```

Також варто зазначити, що алгоритми для генерування нормального, експоненційного та гама-розподілу використовують в якості складової частини генератор рівномірного розподілу. Це означає, що насправді дані класи-нащадки можуть бути шаблонними, де параметр  $U$  відповідає за клас-алгоритм для генерації рівномірного розподілу.

```
template <class U>
class RandNormal : public Rand {
    ...
};
```

Те ж саме стосується і алгоритму 5 для генерації рівномірного розподі-

лу, тільки шаблонних параметрів в ньому має бути 2.

```
template <class U1, class U2>
class RandUniCombine : public Rand {
    ...
};
```

#### 1.4. Алгоритми генерації псевдовипадкових чисел

Для того, щоб згенерувати рівномірно розподілену випадкову величину, в перших чотирьох алгоритмах ми генеруємо деяку дискретну величину. Оскільки і комп'ютеру, і користувачу зручніше працювати з цілими числами, то дана дискретна випадкова величина буде з множини  $\{0, 1, 2, \dots, M - 1\}$ , де  $M$  — достатньо велике натуральне число. Тоді, згенерувавши одне ціле псевдовипадкове число  $x$ , за допомогою нормування  $x/M$  ми отримуємо дійсне число з проміжку  $[0, 1]$ . Зрозуміло, що чим більше  $M$ , тим більшу кількість дійсних чисел з проміжку  $[0, 1]$  ми можемо потенційно отримати таким способом.

Для генерації цілого числа  $x$  використовується рекурсивна формула, яка використовує значення, згенероване на попередньому кроці ( $x_{n+1} = g(x_n)$ ). Початкове значення  $x_0$  є одним з параметрів, що задається користувачем та може передаватися в конструктор класу відповідного алгоритму. Для алгоритму 1 функція  $g$  є лінійною, для алгоритму 2 — квадратичною. Алгоритм 3 використовує дві попередні згенеровані величини  $x_n$  та  $x_{n-1}$ .

$$x_{n+1} = ax_n + c \pmod{M} \text{ (лінійний),}$$

$$x_{n+1} = dx_n^2 + ax_n + c \pmod{M} \text{ (квадратичний),}$$

$$x_{n+1} = x_n + x_{n-1} \pmod{M} \text{ (Фібоначчі).}$$

Алгоритм 4 використовує обернене значення  $x_n^{-1}$  в полі цілих чисел за модулем  $p$ , де  $p$  – просте (в даному алгоритмі воно використовується замість  $M$ ). Це обернене значення характеризується формулою

$$x_n \cdot x_n^{-1} \equiv 1 \pmod{p}. \text{ (обернений)}$$

$x_n^{-1}$  також є цілим числом з проміжку  $[1, p - 1]$  та знаходиться за допомогою розширеного алгоритму Євкліда. Число  $M$  рекомендується брати якомога більшим, але при цьому щоб всі операції в формулах можна було виконувати в процесорі без переповнення.

Алгоритми 5-10 використовують генератори 1-4 для рівномірного розподілу та фактично є набором арифметичних дій та використанням аналітичних функцій над дійсними числами. Далі наведений опис цих алгоритмів.

5. Метод об'єднання.  $z_n = \{x_n - y_n\}$ , де  $x_n, y_n$  – реалізації рівномірно розподілених на  $[0, 1]$  псевдовипадкових величин, отримані з допомогою двох різних алгоритмів,  $\{\cdot\}$  – операція взяття дробової частини.

6. Правило 3-сігма.  $x_n = m + (\sum_{i=1}^{12} y_i - 6)\sigma$ , де  $m, \sigma \in \mathbb{R}, \sigma > 0$  – параметри нормального розподілу (матсподівання та середньоквадратичне відхилення), а в дужках знаходиться сума дванадцяти псевдовипадкових чисел, розподілених рівномірно на  $[0, 1]$ .  $Dom\_min$  та  $Dom\_max$  для цього розподілу можна вважати числа  $m - 3\sigma$  і  $m + 3\sigma$  відповідно.

7. Метод полярних координат видає одразу два числа згідно зі стандартним нормальним розподілом. Для цього спочатку генеруються два рівномірно розподілені на  $[-1, 1]$  числа  $v_1, v_2$ . Якщо  $S = v_1^2 + v_2^2 < 1$ , на вихід подається два числа  $v_1 \sqrt{\frac{-2 \ln S}{S}}, v_2 \sqrt{\frac{-2 \ln S}{S}}$ , якщо ні – генеруються нові числа  $v_1, v_2$  та процедура перевірки повторюється.

8. Метод співвідношень.

1. Згенерувати дві випадкові величини  $U \neq 0$  та  $V$ , рівномірно розподілені на інтервалі  $[0, 1]$ .
2. Визначити  $x = \sqrt{\frac{8}{e} \frac{V-0.5}{U}}$ .
3. Якщо  $x^2 \leq 5 - 4e^{\frac{1}{4}}U$ , повернути число  $x$ .
4. Якщо  $x^2 \geq \frac{4e^{-1.35}}{U} + 1.4$ , повернутися на крок 1.
5. Якщо  $x^2 \leq -4 \ln U$ , повернути число  $x$ , інакше — повернутися на крок 1.

#### 9. Метод логарифму.

$x = -\mu \ln U$ , де  $U$  — рівномірна розподілена величина на проміжку  $[0, 1]$ .

#### 10. Метод Аренса.

1. Обрахувати  $y = tg(\pi U)$ ,  $x = y\sqrt{2a-1} + a - 1$ , де  $U$  — рівномірна розподілена величина на проміжку  $[0, 1]$ .
2. Якщо  $x \leq 0$ , повернутися на крок 1.
3. Згенерувати число  $V$ , рівномірно розподілене на  $[0, 1]$ , та перевірити нерівність  $V > (1 + y^2)e^{(a-1) \ln \frac{x}{a-1} - y\sqrt{2a-1}}$ .
4. Якщо умова з попереднього кроку виконується, повернутися на крок 1, якщо ні — повернути число  $x$ .

В алгоритмах 6, 9 та 10 параметри  $m$ ,  $\sigma$ ,  $\mu$  та  $a$  відповідно є параметрами розподілів (не плутати з параметрами алгоритму) та вводяться користувачем.  $\sigma$  та  $\mu$  мають при цьому завжди бути додатними,  $a > 1$ .

### 1.5. Генерація вибірки та побудова гістограм

Оскільки ми поєднали всі алгоритми в ієрархію з єдиним абстрактним предком, то для побудови гістограми частот ми можемо написати одну функцію, яка буде приймати як параметр посилання на об'єкт класу `Rand`. Це узагальнення ми можемо зробити за допомогою поліморфізму.

```
void histogram(Rand* r, unsigned int N, unsigned int buckets);
```

Другим параметром даної функції є розмір вибірки  $N$ , на основі якої будується гістограма. Ця функція по суті викликає  $N$  разів оператор  $()$  для першого параметру та групує дані за принципом, описаним в пункті 1.2. Третій параметр відповідає за кількість колонок в гістограмі, тобто на скільки підпроміжків розбивається проміжок  $[Dom\_min, Dom\_max]$ . При цьому можуть використовуватися дані про межі випадкових чисел, які також зберігаються в класі `Rand` та його нащадках.

## РОЗДІЛ 2

### ЛАБОРАТОРНА РОБОТА №2: АРИФМЕТИКА ДОВГИХ ЧИСЕЛ

Арифметика довгих чисел виникає як ключова підзадача в сучасній криптографії. Фактично більша частина сучасної цифрової безпеки тримається на декількох алгоритмах, які використовують довгі числа та арифметичні операції над ними. Для швидкого шифрування та дешифрування повідомлень потрібні швидкі алгоритми арифметики довгих чисел. Основним об'єктом розгляду в даній лабораторній є алгоритми швидкого множення, адже для цієї арифметичної операції було вигадано достатньо ефективних алгоритмів та досягнуті визначні результати. Друга половина лабораторної присвячена перевірці чисел на простоту — іншій важливій задачі в криптографії.

Проте перш ніж переходити до алгоритмів, треба визначитися, як саме ми повинні працювати з довгими числами та реалізувати деякий набір стандартних операцій над ними. За допомогою інкапсуляції ми побудуємо клас невід'ємних довгих цілих чисел (**Unsigned Long Int**) та, використовуючи перевантаження операторів мови C++, отримаємо можливість працювати з довгими числами в тій самій нотації, як і зі змінними типу `unsigned int`. В межах цієї лабораторної роботи змінні типу `int` та `unsigned int` будуть називатися “звичайними” або “короткими” числами, щоб підкреслити відмінність від побудованих довгих чисел там, де це важливо.

## 2.1. Умова

Реалізувати клас невід’ємних довгих цілих чисел та реалізувати для нього стандартні арифметичні та логічні операції. Окрім цього, реалізувати наступні алгоритми:

1. Алгоритм швидкого множення Карацуби.
2. Алгоритм швидкого множення Тоома-Кука.
3. Алгоритм швидкого множення Шенхаге.
4. Алгоритм швидкого множення Штрасена.
5. Алгоритм Кука для знаходження оберненого значення.
6. Алгоритм швидкого ділення.
7. Алгоритм перевірки на простоту Люка-Лемера.
8. Алгоритм перевірки на простоту Рабіна-Міллера.
9. Алгоритм перевірки на простоту Соловея-Штрасена.
10. Алгоритм перевірки на простоту Фробеніуса.

## 2.2. Представлення невід’ємних довгих цілих чисел за допомогою класу `ULI`

Існує багато представлень натуральних чисел, проте більшість алгоритмів та операцій над ними сформульовані саме для позиційного запису числа. Тобто натуральне число записується у вигляді рядка, де кожна “цифра” може приймати значення від 0 до  $p$ , де  $p$  — основа числення, зазвичай просте, але в загальному випадку будь-яке натуральне число, не менше за 2. Ми будемо представляти числа саме в такому вигляді.

Всі комп’ютери представляють числа в позиційній системі з  $p = 2$  і деякі з алгоритмів з даної лабораторної працюють лише для таких чисел. Проте

всі алгоритми шкільної арифметики, а також перші 2 алгоритма швидкого множення і алгоритми перевірки на простоту коректно працюють і для довільних інших значень  $p$ . Тому в межах даної лабораторної ми будемо вважати  $p$  параметром числа і ті методи, які не обмежують значення  $p$  для свого застосування, повинні бути реалізовані для всіх можливих значень  $p$ .

Оскільки невід’ємне ціле число в позиційному записі — це просто набір цифр в певному порядку, то логічно цей набір зберігати за допомогою контейнера `std::vector` з бібліотеки STL. Тоді клас невід’ємних цілих чисел у якості полів буде мати вектор цифр та основу числення. Цифри в векторі зберігаються в порядку, зворотньому до запису числа.

```
class ULI {
    ...
private:
    vector<unsigned int> digits;
    unsigned int p;
};
```

number:	123456								
digits:	<table style="border-collapse: collapse; display: inline-table;"> <tr> <td style="border: 1px solid black; padding: 2px 5px;">6</td> <td style="border: 1px solid black; padding: 2px 5px;">5</td> <td style="border: 1px solid black; padding: 2px 5px;">4</td> <td style="border: 1px solid black; padding: 2px 5px;">3</td> <td style="border: 1px solid black; padding: 2px 5px;">2</td> <td style="border: 1px solid black; padding: 2px 5px;">1</td> <td style="border: 1px solid black; padding: 2px 5px;">0</td> <td style="border: 1px solid black; padding: 2px 5px;"> </td> </tr> </table>	6	5	4	3	2	1	0	
6	5	4	3	2	1	0			
index:	0 1 2 3 4 5 6								

Рис. 2.1: Представлення числа за допомогою масива цифр.

Серед конструкторів класу ULI можна реалізувати такий, що приймає звичайне число або рядок символів. Обов’язково в даному випадку передавати до конструктора також і основу числення окремим параметром. Всередині конструкторів обов’язково треба перевіряти, що основа числення не менша за 2, і що число записано коректно згідно з цим значенням

$p$ .

Задані таким чином довгі числа можна переводити з однієї системи числення в іншу. Для цього достатньо реалізувати метод класу `ULI`, що виконує дану процедуру за інтуїтивним алгоритмом.

```
class ULI {
public:
    void change_p(unsigned int new_p);
    ...
};
```

Тоді в залежності від обмежень на  $p$  в певних алгоритмах, можна переводити числа з однієї системи числення в іншу. Функція переведення має асимптотичну складність  $O(n)$ , тож вона не вплине на асимптотику жодного з представлених в даній лабораторній алгоритмів. Проте виникає певна проблема при перевантаженні бінарних операторів. Що, як ми спробуємо додати два числа, які записані з різними основами? Зрозуміло, що треба робити перетворення та зводити числа до однієї основи, проте до якої саме? До основи першого числа чи до основи другого, чи до якоїсь третьої основи? І з якою основою записувати результат? Звісно, ці питання можна вирішити, запровадивши деяке правило, наприклад, результат та обчислення мають основу першого аргумента. Проте цьому правилу можна знайти купу нічим не гірших альтернатив. До того ж, постійні перевірки та зведення до спільної основи на початку кожної функції нагадують той код, який хотілося б або якось узагальнити та автоматизувати, або якось прибрати.

Як альтернативу можна запропонувати перенести основу числення  $p$  в параметри шаблону класу `ULI`.

```
template <unsigned int p>
class ULI {
    ...
private:
    vector<unsigned int> digits;
```

```
};
```

Тоді ми можемо реалізуємо бінарні оператори тільки для класів `ULI` з однаковим параметром  $p$  за допомогою наступної сигнатури.

```
template <unsigned int p>
ULI<p> operator+(const ULI<p>& a, const ULI<p>& b);
```

Компілятор буде видавати помилку при спробі виклику функції з двома числами з різними основами числення. Така стратегія поведінки може бути доцільною, коли ми не ставимо ціллю одночасно оперувати з числами з різними основами числення і коли факт такого оперування свідчить про помилку в логіці програми. Так, наприклад, хоч алгоритм множення Карацуби і коректно працює при будь-якій основі числення, проте він не використовує числа з різною основою, а, отже, наявність таких чисел під час виконання методу є чимось підозрілим.

Серед усіх невід'ємних звичайних чисел  $p$  за основу не можна брати лише 0 та 1. Тоді для того, щоб уникнути створення таких чисел, можна явно вписати шаблони для  $p = 0$  та  $p = 1$  та зробити конструктори в них приватними. Тоді спроба ініціалізації змінної з неприйнятною основою числення призведе до помилки на етапі компіляції. Таким же чином за допомогою шаблону з  $p = 2$  можна вирішити питання більш ефективного зберігання чисел з основою числення 2. У такому випадку замість масиву цифр число можна зберігати у вигляді масиву бітів.

Проте в стратегії використання основи числення як шаблону є обмеження, яке полягає в тому, що всі основи  $p$ , які використовуються в програмі, повинні бути відомі на момент компіляції. Це означає, що з шаблонним класом реалізувати ввід довгого числа із заданною користувачем основою числення неможливо.

Для подальшого розгляду ми будемо вважати, що основа  $p$  реалізована як поле класу `ULI`, оскільки тоді код буде виглядати не таким громіздким. Проте для адаптації всього подальшого коду для шаблонного класу

достатньо до всіх сигнатур методів додавати шаблонні параметри.

### 2.3. Шкільні алгоритми

Перш ніж переходити до складних алгоритмів множення, треба реалізувати так звані “шкільні” алгоритми для деяких інших операторів, які застосовуються до коротких цілих чисел в мові C++. Причому ці оператори мають бути реалізовані з відповідною асимптотичною складністю. Перелік операцій над довгими числами та обмеження на час їхньої роботи наведений в таблиці 2.1. В таблиці та починаючи з цього місця в тексті великими літерами будуть позначатися довгі числа, маленькими — короткі.

$n$  в таблиці — це кількість цифр в числі (або максимальне з двох таких значень у випадку, коли операція застосовується для двох довгих чисел).

До таблиці треба додати декілька пояснень.

По-перше, операція доступу до цифри повинна давати користувачу змогу змінювати цю цифру за потреби. Складність цієї операції, на відміну від інших операцій, є константною, проте така складність може забезпечуватися лише в середньому по достатньо великій кількості викликів даної операції. Це тому що для доступу до великих розрядів може знадобитися розширення `std::vector`, яке є неконстантним по часу. Проте розширення вектору за “звичайної” роботи з цифрами потрібно далеко не завжди.

По-друге, оскільки ми працюємо з невід’ємними числами, то операція віднімання виконується за наступним правилом: якщо від довгого числа  $A$  віднімається довге число  $B$ , то якщо  $A \geq B$ , результатом є математичне значення  $A - B$ , інакше результатом є довге число 0. Те ж саме стосується операції декрементації.

Операції додавання, віднімання, множення та порозрядного зсуву доцільно зробити зовнішніми по відношенню до класу `ULI` дружніми функціями. Операції порозрядного зсуву повинні працювати за тими ж правилами.

Таблиця 2.1

## Операції над числами

Назва операції	Нотація	Складність
Присвоювання (копіювання)	$A = B$	$O(n)$
Доступ до цифри по розряду	$A[i]$	$O(1)$
Додавання	$A + B$	$O(n)$
Віднімання	$A - B$	$O(n)$
Множення (шкільне)	$A * B$	$O(n^2)$
Додавання (з присвоюванням)	$A+ = B$	$O(n)$
Віднімання (з присвоюванням)	$A- = B$	$O(n)$
Множення (шкільне) (з присвоюванням)	$A* = B$	$O(n^2)$
Інкрементація	$++ A$	$O(n)$
Декрементація	$-- A$	$O(n)$
Порозрядний зсув ліворуч	$A \ll k$	$O(n)$
Порозрядний зсув праворуч	$A \gg k$	$O(n)$
Порозрядний зсув ліворуч (з присвоюванням)	$A \ll = k$	$O(n)$
Порозрядний зсув праворуч (з присвоюванням)	$A \gg = k$	$O(n)$
Порівняння (дорівнює)	$A == B$	$O(n)$
Порівняння (не дорівнює)	$A != B$	$O(n)$
Порівняння (менше)	$A < B$	$O(n)$
Порівняння (менше або дорівнює)	$A < = B$	$O(n)$
Порівняння (більше)	$A > B$	$O(n)$
Порівняння (більше або дорівнює)	$A > = B$	$O(n)$
Перетворення в bool		$O(n)$

ми, що і звичайний нециклічний побітовий зсув для коротких чисел.

Всі операції порівняння можна виразити через комбінацію операцій порівнянь зі знаком  $<$ . Це означає, що лише реалізація цього оператора має спиратися на конкретні значення аргументів та використовувати доступ до їхніх цифр. Перетворення в `bool` повинне працювати за тою ж логікою, що і для коротких чисел: якщо число дорівнює 0, то результатом є `false`, інакше — `true`.

Також варто зазначити один важливий момент. Уявімо, що ми обчислюємо значення виразу  $(A - B) + C$ , причому числа  $A$  та  $B$  мають  $n^2$  цифр, а результат їхнього віднімання та число  $C$  мають по  $n$  цифр. Логічно було б припустити, що операція додавання повинна виконуватися за  $O(n)$  кроків. Проте у нашій поточної реалізації класу довгих чисел є одна неприємна особливість: `std::vector` не має гадки про те, що ми намагаємося зберігати і йому абсолютно не важливо, якою є кожна з цифр довгого числа. Це означає, що після віднімання двох чисел він може зберігати велику кількість нулів у вищих розрядах. В даному випадку це  $O(n^2)$  нулів, так що операція додавання буде виконуватися з часом  $O(n^2)$ . Для того, щоб уникнути такої ситуації, доцільно додати до класу нове поле `Length`, яке відслідковуватиме номер найбільшого розряду, де записана ненульова цифра. І після виконання кожної операції, яка потенційно може зменшити `Length` (це операції `[]`, `-`, `--`, `>>`, `- =`, `>>=`), варто видаляти з кінця `std::vector` непотрібні нулі. Асимптотично ця процедура не впливає на жодну з операцій `-`, `--`, `>>`, `- =`, `>>=`. А на складність операції `[]` вона не впливає у середньому.

## 2.4. Швидке множення: алгоритм Карацуби

Розібравшись зі шкільними алгоритмами, ми можемо переходити до методів “швидкого” множення. Першим з таких алгоритмів є алгоритм Ка-

рацуби, який множить два довгих числа довжиною в  $n$  цифр кожне за час  $O(n^{\log_2 3}) = O(n^{1.58496})$ , що є суттєво швидше за шкільне множення. Алгоритм Карацуби застосовний до чисел з будь-якою основою числення  $p$ .

Він є дуже простим та базується на розбитті обох параметрів  $A$  та  $B$  на вищі на нижчі розряди.

Припустимо, що

$$A = a_{2n-1}a_{2n-2}\dots a_1a_0,$$

$$B = b_{2n-1}b_{2n-2}\dots b_1b_0,$$

де  $a_i, b_i$  — цифри від 0 до  $p - 1$ . Визначимо числа

$$A_1 = a_{2n-1}a_{2n-2}\dots a_{n+1}a_n, \quad A_0 = a_{n-1}a_{n-2}\dots a_1a_0,$$

$$B_1 = b_{2n-1}b_{2n-2}\dots b_{n+1}b_n, \quad B_0 = b_{n-1}b_{n-2}\dots b_1b_0.$$

Кожне з цих чисел має по  $n$  цифр. Фактично, оригінальні  $A$  та  $B$  є простими конкатенаціями даних чисел ( $A = \overline{A_1A_0}, B = \overline{B_1B_0}$ ). Визначити програмно ці числа можна за допомогою наступних формул.

$$A_1 = A \gg n, \quad A_0 = A - (A_1 \ll n),$$

$$B_1 = B \gg n, \quad B_0 = B - (B_1 \ll n).$$

На основі цих чисел за допомогою рекурсивного виклику множення за алгоритмом Карацуби обчислюємо вирази.

$$C_2 = A_1 \cdot B_1, \quad C_0 = A_0 \cdot B_0,$$

$$C_1 = (A_0 + A_1) \cdot (B_0 + B_1) - C_0 - C_2.$$

Будуємо відповідь за формулою

$$C = (C_2 \ll (2n)) + (C_1 \ll n) + C_0, \quad (C = \overline{C_2 C_1 C_0}).$$

Нескладно побачити, що за наших обчислень насправді  $C_1 = A_0 B_1 + A_1 B_0$ . Якби наші числа склалися з двох цифр кожне, то під час шкільного множення ми мали б порахувати чотири добутки:  $A_0 B_0$ ,  $A_1 B_1$ ,  $A_0 B_1$  та  $A_1 B_0$ , тобто добуток нижчих розрядів, добуток вищих розрядів та два змішані добутки. Після цього змішані добутки просумовуюються та все це поєднується у відповідь за допомогою порозрядного зміщення отриманих величин за формулою, схожу на формулу для  $C$ .

Так само працює шкільне множення довгих чисел в рекурсивній формі. Нам треба 4 рази помножити вдвічі коротші числа для отримання результату. Алгоритм Карацуби дозволяє робити 3 рекурсивних виклики замість чотирьох, за рахунок чого він є асимптотично швидшим.

Оскільки алгоритм є рекурсивним, то для нього має бути визначеним дно рекурсії. В даному випадку це повинні бути настільки малі довжини чисел  $A$  та  $B$ , що рекурсію викликати або неможливо, або доцільніше використати шкільне множення.

Дно рекурсії студент може визначити самостійно, проте бажано, щоб це значення  $n$  було невеликим. Якщо числа зберігаються у двійковому записі ( $p = 2$ ), то доцільно дном рекурсії обирати такі довжини, що відповідні числа можна перемножити в процесорі без втрати розрядів.

Розглянемо процес виконання алгоритму Карацуби на прикладі множення двох чотирьохзначних десяткових чисел за умови, що дно рекурсії — це добуток двозначних чисел.

Нехай  $A = 1234$ ,  $B = 0321$ . Тоді  $n = 2$ ,  $A_1 = 12$ ,  $A_0 = 34$ ,  $B_1 = 03$ ,  $B_0 = 21$ . Рахуємо  $C_2 = 12 \cdot 3 = 36$ ,  $C_0 = 34 \cdot 21 = 714$ ,  $C_1 = (12 + 34) \cdot (3 + 21) - 36 - 714 = 354$ . Тоді  $C = 36 \cdot 10^4 + 354 \cdot 10^2 + 714 = 396114$ , що і є правильним

результатом.

Тут варто зазначити, що число  $B$  має насправді 3 цифри, проте, оскільки число  $A$  є довшим, то ми заповнили вищі розряди  $B$  нулями. Це ніяк не впливає на математичне значення числа і не впливає на операції, які використовувалися під час обчислення. Це є нормальною практикою в ситуації, коли числа мають різну довжину: доповнювати коротше нулями у вищих розрядах. Так само, якби наші числа були  $A = 234$ ,  $B = 321$ , для коректного ділення числа навпіл по розрядам, нам треба обидва числа згори доповнити одним нулем, щоб вийшли числа  $A = 0234$ ,  $B = 0321$ . При цьому спочатку треба “дотягнути” коротше число до довшого і після цього за потреби додавати до них обох нулі. Довжини блоків  $A_0$ ,  $B_0$ ,  $A_1$ ,  $B_1$  мають бути однаковими. Причому на рівні програми не треба додавати нулі до `std::vector` в числа. Це “дописування” нулів є лише уявним прийомом, який має ціллю пояснити процедуру коректного визначення довжини чисел для використання алгоритму Карацуби.

В наступних алгоритмах буде спостерігатися аналогічна ситуація з числами різної довжини або з такими, що не діляться на однакову кількість розрядів. Буде вважатися, що до запуску алгоритму числа зведені до одєї довжини.

## 2.5. Швидке множення: алгоритм Тоома-Кука

Метод Тоома-Кука схожий на метод Карацуби, тільки в ньому ділення чисел відбувається на три блоки  $A = \overline{A_2A_1A_0}$  та  $B = \overline{B_2B_1B_0}$ . Він також застосовний для чисел з будь-якою основою числення. Будемо вважати, що довжини чисел дорівнюють  $n$ , причому  $n$  ділиться на 3.

### 1. Визначити

$$A_2 = A \gg (2n/3),$$

$$A_1 = (A \gg (n/3)) - (A_2 \ll (n/3)),$$

$$A_0 = A - (A_2 \ll (2n/3) - (A_1 \ll (n/3))),$$

$$B_2 = B \gg (2n/3),$$

$$B_1 = (B \gg (n/3)) - (B_2 \ll (n/3)),$$

$$B_0 = B - (B_2 \ll (2n/3) - (B_1 \ll (n/3))).$$

2. Визначити рекурсивно 5 добутків

$$F_{0,0} = A_0 \cdot B_0,$$

$$F_{1,0} = (A_2 + A_1 + A_0) \cdot (B_2 + B_1 + B_0),$$

$$F_{2,0} = (4A_2 + 2A_1 + A_0) \cdot (4B_2 + 2B_1 + B_0),$$

$$F_{3,0} = (9A_2 + 3A_1 + A_0) \cdot (9B_2 + 3B_1 + B_0),$$

$$F_{4,0} = (16A_2 + 4A_1 + A_0) \cdot (16B_2 + 4B_1 + B_0).$$

Тут множення на короткі числа реалізуються окремо з асимптотикою  $O(n)$  або замінюються на відповідну кількість операцій додавання.

3. Визначити

$$F_{0,1} = F_{1,0} - F_{0,0},$$

$$F_{1,1} = F_{2,0} - F_{1,0},$$

$$F_{2,1} = F_{3,0} - F_{2,0},$$

$$F_{3,1} = F_{4,0} - F_{3,0}.$$

4. Визначити

$$F_{0,2} = (F_{1,1} - F_{0,1})/2,$$

$$F_{1,2} = (F_{2,1} - F_{1,1})/2,$$

$$F_{2,2} = (F_{3,1} - F_{2,1})/2.$$

На цьому та подальших кроках ділення на коротке число має бути визначене як окрема операція з асимптотикою  $O(n)$ . Всі ділення згідно з теорією повинні виконуватися націло, якщо в результаті якого ділення залишається ненульова остача, це свідчить про помилку в

реалізації алгоритма.

5. Визначити  $F_{0,3} = (F_{1,2} - F_{0,2})/3$ ,  $F_{1,3} = (F_{2,2} - F_{1,2})/3$ .

6. Визначити  $F_{0,4} = (F_{1,3} - F_{0,3})/4$ .

7. Визначити поліном

$$P(x) = F_{0,0} + F_{0,1}x + F_{0,2}x(x-1) + \\ + F_{0,3}x(x-1)(x-2) + F_{0,4}x(x-1)(x-2)(x-3).$$

8. Розкрити дужки в отриманому поліномі та отримати поліном вигляду

$$Q(x) = C_4x^4 + C_3x^3 + C_2x^2 + C_1x + C_0.$$

9. Повернути значення

$$C_4 \ll (4n/3) + C_3 \ll n + C_2 \ll (2n/3) + C_1 \ll (n/3) + C_0.$$

Дно рекурсії визначається аналогічно до попереднього методу. Звісно, на кроках 7 та 8 не треба визначати поліном за допомогою нового класу, з ним можна працювати як з набором коефіцієнтів в рамках даного алгоритму. Процедура розкриття дужок — це просто підрахунок комбінацій коефіцієнтів  $F$  для утворення нових коефіцієнтів  $C$ . Отримання конкретного вигляду цих комбінацій залишається як вправа читачеві.

Суть методу Тоома-Кука полягає в тому, що на основі чисел  $A$  і  $B$  будуються два квадратичних поліноми  $P_1(x) = A_2x^2 + A_1x + A_0$  та  $P_2 = B_2x^2 + B_1x + B_0$ . Алгоритм Тоома-Кука — це процедура знаходження коефіцієнтів поліному-добутку за допомогою процедури інтерполяції Ньютона по 5 точкам.

$$P_1(x)P_2(x) = P_3(x) = C_4x^4 + C_3x^3 + C_2x^2 + C_1x + C_0.$$

Тоді, оскільки  $P_1(p^{n/3}) = A$ ,  $P_2(p^{n/3}) = B$ , то  $P_3(p^{n/3}) = AB$ , що і є шуканим результатом.

Асимптотична складність алгоритму:  $O(n^{\log_3 5}) = O(n^{1.46497})$ .

## 2.6. Швидке множення: алгоритм Шенхаге

Алгоритм Шенхаге використовується тільки для двійкових чисел. Він також є рекурсивним та базується на представлення числа через сукупність чисел меншої довжини, проте, на відміну від алгоритмів Карацуби і Тоома-Кука, він не проводить розділення чисел за розрядами. Натомість він представляє довге число  $A$  через 6 коротших довгих чисел  $A_1, \dots, A_6$ , які є результатами операції  $A \bmod M_i$ ,  $i = \overline{1, 6}$ , тобто у вигляді залишків від ділення на певним чином визначені числа  $M_i$ . Оригінальне число  $A$  можна відновити за допомогою китайської теореми про залишки. У використанні цієї теореми і полягає основна ідея алгоритму Шенхаге.

Ще однією відмінністю алгоритма Шенхаге є визначення рівня рекурсії відповідно до довжини числа. У попередніх алгоритмах рівень рекурсії визначався тим, скільки разів можна поділити число  $n$  на 2 або 3, поки ми не досягнемо значення 0. Тоді числа розбивалися на вдвічі та втричі коротші. Тут ми введемо зростаючу послідовність чисел  $p_n$  та будемо множити числа довжиною  $p_n$  за допомогою рекурсивних викликів алгоритму Шенхаге для множення чисел довжиною  $p_{n-1}$ .

Також на додачу до стандартних операцій в цьому алгоритмі використовується операція взяття остачі від ділення на числа  $M_i$ . Кожне таке число є по суті  $2^{e_i} - 1$ , де  $e_i$  — коротке додатне число. Тобто в двійковому записі кожне з чисел  $M_i$  є просто набором одиниць довжини  $e_i$ . У зв'язку з цим самі числа  $M_i$  не потрібно зберігати у пам'яті, а працювати з ними можна, передаючи у відповідні функції короткі числа  $e_i$ , оскільки кожне  $M_i$  повністю визначається своїм  $e_i$ .

Алгоритм Шенхаге використовує операції модулярної арифметики за модулями  $M_i$ , такі як додавання та віднімання.

Додавання за модулем — це просто додавання та подальше знаходження остачі від ділення результату на  $M_i$ . Віднімання за модулем працює трохи

інакше (тут  $A, B < M_i$ ):

$$A - B \bmod M_i = \begin{cases} A - B, & \text{if } A \geq B, \\ M_i + A - B, & \text{if } A < B. \end{cases}$$

Сама операція взяття числа  $A$  по модулю  $M_i$  за рахунок спеціального вигляду  $M_i$  реалізується наступним чином: якщо  $A < M_i$ , то результатом є просто число  $A$ , якщо ні, то число  $A$  розбивається на блоки  $\overline{A_k A_{k-1} \dots A_1 A_0}$  порозрядно, так що кожен блок  $A_j$  містить  $e_j$  цифр. Тоді

$$A \bmod M_i = \sum_{j=0}^k A_j \bmod M_i,$$

тобто виконується рекурсивний виклик.

В тексті операцію взяття числа по модулю зазвичай пишуть в самому кінці, як у виразі

$$A + B - C + D \bmod M_i.$$

Дана приписка свідчить, що всі операції насправді є операціями модулярної арифметики і даний вираз насправді є еквівалентним такому.

$$((((((A + B) \bmod M_i) - C) \bmod M_i) + D) \bmod M_i.$$

Саме згідно з останнім виразом рекомендується реалізовувати всі операції в межах цього алгоритму.

Сам алгоритм Шенхаге виглядає наступним чином. Вважається, що числа  $A$  і  $B$  мають довжину  $n$ .

1. Рахувати рекурентну числову послідовність  $q_0 = 1$ ,  $q_{k+1} = 3q_k - 1$ , допоки число  $n$  не стане менше за  $p_k = 18q_k + 8$ . Нехай  $q_k$  — найменше число з послідовності, коли ця умова досягнута.
2. Визначити  $e_1 = 6q_k - 1$ ,  $e_2 = 6q_k + 1$ ,  $e_3 = 6q_k + 2$ ,  $e_4 = 6q_k + 3$ ,  $e_5 = 6q_k + 5$ ,  $e_6 = 6q_k + 7$  та відповідні числа  $M_1, M_2, M_3, M_4, M_5, M_6$ .

3. Визначити  $A_i = A \bmod M_i$ ,  $B_i = B \bmod M_i$ ,  $i = \overline{1, 6}$ . Числа  $A_i, B_i$  мають довжину менше за  $p_{k-1} = 18q_{k-1} + 8$ .
4. Виконати рекурсивно 6 множень та отримати числа  $C_i = A_i B_i$ ,  $i = \overline{1, 6}$ .
5. Знайти  $W_i = C_i \bmod M_i$ ,  $i = \overline{1, 6}$ .
6. За допомогою визначеної нижче процедури  $C$  виконати кроки 7-12.
7.  $W_1^1 = W_1 \bmod M_1$ .
8.  $W_2^1 = C(W_2 - W_1^1, M_1, M_2) \bmod M_2$ .
9.  $W_3^1 = C(C(W_3 - W_1^1, M_1, M_3) - W_2^1, M_2, M_3) \bmod M_3$ .
10.  $W_4^1 = C(C(C(W_4 - W_1^1, M_1, M_4) - W_2^1, M_2, M_4) - W_3^1, M_3, M_4) \bmod M_4$ .
11.  $W_5^1 = C(C(C(C(W_5 - W_1^1, M_1, M_5) - W_2^1, M_2, M_5) - W_3^1, M_3, M_5) - W_4^1, M_4, M_5) \bmod M_5$ .
12.  $W_6^1 = C(C(C(C(C(W_6 - W_1^1, M_1, M_6) - W_2^1, M_2, M_6) - W_3^1, M_3, M_6) - W_4^1, M_4, M_6) - W_5^1, M_5, M_6) \bmod M_6$ .
13. Повернути значення  

$$((((W_6^1 M_5 + W_5^1) M_4 + W_4^1) M_3 + W_3^1) M_2 + W_2^1) M_1 + W_1^1$$

Дном рекурсії в даному випадку є числа довжиною менше за  $18q_0 + 8 = 26$  бітів. Такі числа можна множити в 64-розрядному процесорі без втрати точності.

Множення на  $M_i$  в останньому кроці алгоритму виконується дуже просто:  $AM_i = (A \ll e_i) - A$ .

Процедура  $C(U, M_i, M_j)$  виконується наступним чином.

1. За допомогою розширеного алгоритму Євкліда знайти число  $b$  таке, що  $be_i = 1 \bmod e_j$ .
2. Нехай бітове представлення короткого числа  $b$  наступне:  $b_s b_{s-1} \dots b_2 b_1 b_0$ .  
Визначити  $a_0 = e_i$ ,  $d_0 = b_0 e_i$ ,  $U_0 = U$ ,  $V_0 = b_0 U$ ,  $k = 0$ .
3. Інкрементувати значення  $k$ . Визначити

$$a_k = 2a_{k-1} \bmod e_j,$$

$$d_k = (d_{k-1} + b_k a_k) \bmod e_j,$$

$$U_k = (U_{k-1} + U_{k-1} \ll a_{k-1}) \bmod M_j,$$

$$V_k = (V_{k-1} + b_k(U_k \ll d_{k-1})) \bmod M_j.$$

4. Якщо  $k = s$ , повернути значення  $V_s$ . Інакше повторити крок 3.

Ця процедура фактично дає результат множення за модулем  $M_j$  числа  $U$  та числа  $C_{ij}$ , що є оберненим до числа  $M_i$  по модулю  $M_j$ . Саме число  $C_{ij}$  при цьому не рахується, оскільки це займає багато часу. Процедура  $C$  є доволі креативним способом отримати значення бажаного добутку без знаходження  $C_{ij}$ . За рахунок цього алгоритм і досягає своєї асимптотичної складності.

Асимптотична складність алгоритму Шенхаге —  $O(n^{\log_3 6}) = O(n^{1.6309})$ .

## 2.7. Швидке множення: алгоритм Штрасена

Алгоритм Штрасена є ускладненням алгоритму Тоома-Кука, оскільки він також використовує зведення множення довгих чисел до множення поліномів, проте поліноми ці набагато складніші. Кількість блоків розрядів, на які розбиваються числа  $A$  та  $B$  в цьому алгоритмі, тепер визначається динамічно і залежить від довжини чисел. Так само, як і метод Шенхаге, алгоритм Штрасена використовується тільки для чисел з основою числення 2.

Як проміжну процедуру, алгоритм Штрасена використовує швидке перетворення Фур'є. Алгоритм цього перетворення буде описаний в наступному пункті.

Як і у випадку алгоритму Шенхаге, всі операції тут будуть виконуватися в модулярній арифметиці, тільки модулі в даному випадку будуть виглядати як  $2^N + 1$ . Додавання та віднімання за цим модулем не відрізня-

ється від аналогічних процедур, описаних в попередньому пункті. Проте саме взяття числа по такому модулю відбувається трохи інакше.

Якщо  $A < 2^N + 1$ , то  $A \bmod (2^N + 1) = A$ , інакше число  $A$  розбивається на блоки  $\overline{A_k A_{k-1} \dots A_1 A_0}$  порозрядно, так що кожен блок  $A_j$  містить  $N$  цифр. Тоді

$$A \bmod (2^N + 1) = (((A_0 + A_2 + A_4 + \dots) \bmod (2^N + 1)) - ((A_1 + A_3 + A_5 + \dots) \bmod (2^N + 1))) \bmod (2^N + 1),$$

тобто виконується два рекурсивних виклики.

### Алгоритм Штрассена:

Вхідні дані: двійкові довгі числа  $A$  та  $B$  довжиною  $n$  цифр.

1. Знайти найменше натуральне  $m$  таке, що  $2n \leq 2^m$ .
2. Повернути результат виконання процедури **Множення за модулем** з параметрами  $A$ ,  $B$  та  $m$ .

### Множення за модулем:

Вхідні дані: двійкові довгі числа  $U, V < 2^{2^n} + 1$  та число  $n$  (це вже не довжина чисел).

Вихідні дані: результат формули  $UV \bmod (2^{2^n} + 1)$ .

1. Якщо  $U = 2^{2^n}$ , повернути  $2^{2^n} + 1 - V$ . Якщо  $V = 2^{2^n}$ , повернути  $2^{2^n} + 1 - U$ . Інакше виконати кроки 2-16.
2. На цьому кроці виконується  $U, V < 2^{2^n}$ . Визначити  $k = \lceil n/2 \rceil$ ,  $l = \lfloor n/2 \rfloor$ . Якщо  $k < 6$ , виконати множення чисел  $U$  і  $V$  методом Карацуби та повернути значення  $UV \bmod (2^{2^n} + 1)$ . Інакше виконати кроки 3-16.
3. Визначити  $N = 2^n$ ,  $K = 2^k$ ,  $L = 2^l$ . Ці числа насправді також є короткими. Запис цих трьох величин через великі літери є винятком із сформульованого нами на початку розділу правила. Такий запис обраний для підкреслення різниці в порядках між даними величи-

нами та величинами  $n$ ,  $k$  та  $l$ .

4. Представити числа  $U$  та  $V$  як блоки бітів

$$\overline{U_{K-1}U_{K-2}\dots U_1U_0},$$

$$\overline{V_{K-1}V_{K-2}\dots V_1V_0}.$$

Кожне  $U_i, V_i$  складається з  $L$  цифр.

5. Визначити  $U_r^1 = U_r \bmod 2^k$  та  $V_r^1 = V_r \bmod 2^k$  для  $r = \overline{0, K-1}$  (просте обрізання  $\text{std} :: \text{vector}$ ).

6. Визначити

$$U^{1,1} = \sum_{r=0}^{K-1} (U_r^1 \lll 3kr),$$

$$V^{1,1} = \sum_{r=0}^{K-1} (V_r^1 \lll 3kr),$$

$$U^{1,2} = \sum_{r=0}^{K-1} (U_{K-1-r}^1 \lll 3kr),$$

$$V^{1,2} = \sum_{r=0}^{K-1} (V_{K-1-r}^1 \lll 3kr).$$

7. Методом швидкого множення Штрассена знайти добутки  $W^{1,1} = U^{1,1}V^{1,1}$  та  $W^{1,2} = U^{1,2}V^{1,2}$

8. Визначити

$$W_r^1 = ((W^{1,1} \ggg (3kr)) \bmod 2^k - (W^{1,2} \ggg (3k(K-r-2))) \bmod 2^k) \bmod 2^k, \quad r = \overline{0, K-2},$$

$$W_{K-1}^1 = (W^{1,1} \ggg (3k(K-1))) \bmod 2^k.$$

9. Визначити  $\psi = 2^{2^{l+1-k}}$ ,

$$A_r = U_r \lll r(2^{l+1-k}) \bmod (2^{2L} + 1),$$

$$B_r = V_r \lll r(2^{l+1-k}) \bmod (2^{2L} + 1),$$

$$r = \overline{0, K-1}.$$

10. Здійснити швидке перетворення Фур'є для масивів  $A_r, B_r$  з параметрами  $\omega = \psi^2$  та  $M = 2^{2L} + 1$ , отримавши масиви  $A_r^1, B_r^1$ .

11. Рекурсивно для кожного  $r = \overline{0, K-1}$  викликати процедуру **Множення за модулем** з параметрами  $A_r^1, B_r^1$  та  $l+1$ . Результати позначити як  $C_r^1$  відповідно.
12. Виконати швидке перетворення Фур'є для масиву  $C_r^1$  з параметрами  $\omega = \psi^2$  та  $M = 2^{2L} + 1$ , отримавши масив  $C_r$ .
13. Визначити
 
$$W_r^2 = (C_{K-r} \ll (4L - k - r(2^{l+1-k}))) \bmod (2^{2L} + 1), \quad r = \overline{1, K-1},$$

$$W_0^2 = (C_0 \ll (4L - k)) \bmod (2^{2L} + 1).$$
14. Визначити
 
$$W_r^3 = ((W_r^1 - W_r^2) \bmod 2^k) \ll (2L) + ((W_r^1 - W_r^2) \bmod 2^k) + W_r^2,$$

$$r = \overline{0, K-1}.$$

Віднімання за модулем  $2^k$  визначено так само, як і для загального значення модуля  $M$ .

15. Для кожного  $r = \overline{0, K-1}$  визначити  $W_r$  наступним чином: якщо  $W_r^3 < (r+1)2^{2L}$ , то  $W_r = W_r^3 \bmod (2^N + 1)$ , інакше  $W_r = (W_r^3 - 2^k(2^{2L} + 1)) \bmod (2^N + 1)$ .
16. Повернути значення  $(\sum_{r=0}^{K-1} (W_r \ll rL)) \bmod (2^N + 1)$ .

По суті величини  $W_r$  в даній процедурі є так званими “згортками” коефіцієнтів  $U_r$  та  $V_r$ .

$$W_r = (U_r V_0 + U_{r-1} V_1 + \dots + U_0 V_r) - (U_{K-1} V_{r+1} + \dots + U_{r+1} V_{K-1}) \bmod (2^N + 1).$$

Ці величини є ключовими в обрахунку цифр числа-добутку. Швидке отримання згорток і є суттю більшості алгоритмів швидкого множення. В алгоритмі Штрасена це досягається за рахунок того, що спочатку визначаються більш короткі складові  $W_r^1$  та більш довгі складові  $W_r^2$ , з яких на кроці 14 за формулою, яка нагадує формули методу Шенхаге, збирається

число  $W_r^3$ , яке не дуже сильно відрізняється від потрібного нам значення  $W_r$ . Фактично, формула з кроку 14 може бути переписана наступним чином.

$$W_r^3 = (2^{2L} + 1)((W_r^1 - W_r^2) \bmod 2^k) + W_r^2.$$

Коротші компоненти  $W_r^1$  отримуються за рахунок двох рекурсивних викликів алгоритму Штрасена для згрупованих разом компонент  $U_r^1$  та  $V_r^1$  (крок 7). Для довших компонент згортки обраховуються за допомогою швидкого перетворення Фур'є. Саме розбиття на дві частини (коротшу та довшу) кожної компоненти обумовлено потребами оптимального зменшення кількості цифр під час виклику рекурсії. Коротша частина в даному випадку просто уточнює довшу. За рахунок цього алгоритм має найкращу асимптотичну швидкість.

Асимптотична складність алгоритму Штрасена —  $O(n \ln n \ln \ln n)$ .

## 2.8. Швидке перетворення Фур'є

Під час опису алгоритму швидкого перетвоорення Фур'є ми будемо часто записувати короткі числа у двійковому записі  $t = (t_{k-1}t_{k-2}\dots t_1t_0)_2$ . Варто пам'ятати, що цей набір цифр є одним числом.

### Алгоритм швидкого перетворення Фур'є.

Вхідні дані: масив довгих цілих невід'ємних чисел  $(A_0, A_1, \dots, A_{K-1})$ ,  $K = 2^k$ , число  $\omega$  та модуль  $M$ .

Вихідні дані: масив нових довгих цілих невід'ємних чисел  $(A_0^1, A_1^1, \dots, A_{K-1}^1)$ .

Перед початком виконання зручно порахувати значення  $\omega^2, \omega^4, \dots, \omega^{2^{k-1}}$ . Тоді, коли нам треба буде порахувати степінь  $\omega^t = \omega^{(t_{k-1}t_{k-2}\dots t_1t_0)_2}$ , ми можемо формувати цю степінь як добуток обчислених значень  $\omega^t = \prod \omega^{2^i}$ , де добуток береться по тим степеням, для яких відповідне  $t_i \neq 0$ .

Спочатку ми визначаємо масив  $B^0$  з  $K$  довгих чисел, використовуючи вхідні дані:

$$B^0[(t_{k-1}t_{k-2}\dots t_1t_0)_2] = A_{(t_{k-1}t_{k-2}\dots t_1t_0)_2}, \quad t = (t_{k-1}t_{k-2}\dots t_1t_0)_2, \quad 0 \leq t < K.$$

На 1 кроці для  $t$  від 0 до  $K - 1$  заповнюємо новий масив довгих чисел

$$B^1[(t_{k-1}t_{k-2}\dots t_1t_0)_2] = (B^0[(0t_{k-2}\dots t_1t_0)_2] + \omega^{(t_{k-1}0\dots 0)_2} B^0[(1t_{k-2}\dots t_1t_0)_2]) \bmod M.$$

На 2 кроці виконуємо аналогічну процедуру, зміщаючись по розрядам індексів.

$$B^2[(t_{k-1}t_{k-2}\dots t_1t_0)_2] = (B^1[(t_{k-1}0t_{k-3}\dots t_1t_0)_2] + \omega^{(t_{k-2}t_{k-1}0\dots 0)_2} B^1[(t_{k-1}1t_{k-3}\dots t_1t_0)_2]) \bmod M.$$

Зверніть увагу, що степінь  $\omega$  заповнюється в зворотньому порядку.

Продовжуємо аналогічні дії, поки не досягнемо кроку  $k$ .

$$B^k[(t_{k-1}t_{k-2}\dots t_1t_0)_2] = (B^{k-1}[(t_{k-1}t_{k-2}\dots t_10)_2] + \omega^{(t_0t_1\dots t_{k-1})_2} B^{k-1}[(t_{k-1}t_{k-2}\dots t_11)_2]) \bmod M.$$

Для кожного  $t = (t_{k-1}t_{k-2}\dots t_1t_0)_2$  визначаємо компоненти масива-відповіді:

$$A^1[(t_{k-1}t_{k-2}\dots t_1t_0)_2] = B^k[(t_0t_1\dots t_{k-1})_2].$$

В цій формулі також присутня зміна порядку бітів.

Суть застосування перетворення Фур'є в рамках алгоритму Штрассена полягає в тому, що за допомогою цього перетворення ми можемо швидко перейти від поліномів до їхніх значень на наборі точок. Так, вхідні дані перетворення Фур'є можна інтепретувати як коефіцієнти деякого полінома. Вихідні — як значення цього полінома у точках  $1, \omega, \omega^2, \dots$ . Тоді, отримавши набори значень для двох поліномів, ми перемножуємо їх покомпонентно (крок 11 процедури **Множення за модулем**) та повертаємося назад до коефіцієнтів поліному-добутку. Ідея така сама, як і в алгоритмі Тома-Кука. Проте за рахунок спеціального вибору степені поліномів та

значення  $\omega$  процедура суттєво спрощується, оскільки, по-перше, степінь поліному-добутка співпадає зі степенями поліномів-множників, і, по-друге, для повернення “назад” від значень до поліномів нам не треба вигадувати нову процедуру оберненого перетворення Фур’є, а достатньо виконати ще одне перетворення “вперед” (крок 12 процедури **Множення за модулем**) і після цього лише змінити порядок отриманих значень та трохи їх змістити (крок 13 процедури **Множення за модулем**). За цими простими діями приховується доволі складна та цікава математика, про яку можна почитати у відповідній літературі.

## 2.9. Швидке множення: алгоритм Штрасена: приклад

В цьому пункті ми покажемо, що алгоритм Штрасена дійсно працює, розглянувши приклад множення двох маленьких чисел. Ми проігноруємо перевірку умови дна рекурсії в процедурі **Множення за модулем** та спробуємо знайти добуток чисел  $A = 7$  та  $B = 5$ . При цьому ми будемо також ігнорувати рекурсивні виклики, вважаючи, що всі допоміжні множення ми можемо зробити руками. Оскільки алгоритм Штрасена працює для двійкових чисел, то в цьому пункті там, де це неочевидно, ми будемо вказувати, що записані нами числа мають основу числення 2. Це буде позначатися відповідним індексом внизу біля числа.

Алгоритм Штрасена( $A = 7 = 111_2$ ,  $B = 5 = 101_2$ ).

Крок 1. Обидва числа мають по  $n = 3$  цифри, тому мінімальне  $m$  дорівнює 3 ( $2 \cdot 3 = 6 \leq 8 = 2^3$ ).

Крок 2. Виклик процедури **Множення за модулем** з параметрами  $111_2$ ,  $101_2$  та 3.

**Множення за модулем**( $U = 00000111_2$ ,  $V = 00000101_2$ ,  $n = 3$ ).

Крок 1. Жодне з чисел  $U$ ,  $V$  не дорівнює  $2^{2^3} = 256 = 100000000_2$ , тому переходимо до кроку 2.

Крок 2.  $k = \lceil 3/2 \rceil = 2$ ,  $l = \lfloor 3/2 \rfloor = 1$ . Дно рекурсії ми ігноруємо та переходимо на крок 3.

Крок 3.  $N = 2^3 = 8$ ,  $K = 2^2 = 4$ ,  $L = 2^1 = 2$ .

Крок 4.

$$U_0 = 11_2, V_0 = 01_2,$$

$$U_1 = 01_2, V_1 = 01_2,$$

$$U_2 = 00_2, V_2 = 00_2,$$

$$U_3 = 00_2, V_3 = 00_2.$$

Крок 5.

$$U_0^1 = 11_2 \bmod 2^2 = 11_2, V_0^1 = 01_2 \bmod 2^2 = 01_2,$$

$$U_1^1 = 01_2 \bmod 2^2 = 01_2, V_1^1 = 01_2 \bmod 2^2 = 01_2,$$

$$U_2^1 = 00_2 \bmod 2^2 = 00_2, V_2^1 = 00_2 \bmod 2^2 = 00_2,$$

$$U_3^1 = 00_2 \bmod 2^2 = 00_2, V_3^1 = 00_2 \bmod 2^2 = 00_2.$$

Крок 6.

$$\begin{aligned} U^{1,1} &= 11_2 + 01_2 \ll (3 \cdot 2 \cdot 1) + 00_2 \ll (3 \cdot 2 \cdot 2) + 00_2 \ll (3 \cdot 1 \cdot 3) = \\ &= 1000011_2 = 67, \end{aligned}$$

$$\begin{aligned} V^{1,1} &= 01_2 + 01_2 \ll (3 \cdot 2 \cdot 1) + 00_2 \ll (3 \cdot 2 \cdot 2) + 00_2 \ll (3 \cdot 1 \cdot 3) = \\ &= 1000001_2 = 65, \end{aligned}$$

$$\begin{aligned} U^{1,2} &= 00_2 + 00_2 \ll (3 \cdot 2 \cdot 1) + 01_2 \ll (3 \cdot 2 \cdot 2) + 11_2 \ll (3 \cdot 1 \cdot 3) = \\ &= 11000001000000000000_2 = 193 \cdot 2^{12}, \end{aligned}$$

$$\begin{aligned} V_{1,2} &= 00_2 + 00_2 \ll (3 \cdot 2 \cdot 1) + 01_2 \ll (3 \cdot 2 \cdot 2) + 01_2 \ll (3 \cdot 1 \cdot 3) = \\ &= 10000010000000000000_2 = 65 \cdot 2^{12}. \end{aligned}$$

Крок 7.

$$W^{1,1} = 1000100000011_2 = 4355,$$

$$W_{1,2} = 11000100000001000000000000000000000000_2 = 12545 \cdot 2^{24}.$$

Тут ми отримали ситуацію, коли для знаходження добутку трицифрових чисел, нам потрібно помножити набагато довші числа. Зрозуміло, що це викликано тим, що ми проігнорували дно рекурсії кроці 2. Така ж ситуація буде і на кроці 11.

Крок 8.

$$\begin{aligned}
W_0^1 &= ((W^{1,1}) \bmod : 2^2 - (W^{1,2} \gg (3 \cdot 2(4 - 0 - 2))) \bmod 2^2) \bmod 2^2 = \\
&= ((W^{1,1}) \bmod : 2^2 - (W^{1,2} \gg 12) \bmod 2^2) \bmod 2^2 = 11_2 = 3, \\
W_1^1 &= ((W^{1,1} \gg (3 \cdot 2 \cdot 1)) \bmod : 2^2 - \\
&\quad - (W^{1,2} \gg (3 \cdot 2(4 - 1 - 2))) \bmod 2^2) \bmod 2^2 = \\
&= ((W^{1,1} \gg 6) \bmod : 2^2 - (W^{1,2} \gg 6) \bmod 2^2) \bmod 2^2 = 00_2 = 0, \\
W_2^1 &= ((W^{1,1} \gg (3 \cdot 2 \cdot 2)) \bmod : 2^2 - \\
&\quad - (W^{1,2} \gg (3 \cdot 2(4 - 2 - 2))) \bmod 2^2) \bmod 2^2 = \\
&= ((W^{1,1} \gg 12) \bmod : 2^2 - (W^{1,2}) \bmod 2^2) \bmod 2^2 = 01_2 = 1, \\
W_3^1 &= ((W^{1,1} \gg (3 \cdot 2 \cdot 3)) \bmod 2^2 = \\
&= (W^{1,1} \gg 18) \bmod : 2^2 = 00_2 = 0.
\end{aligned}$$

Крок 9.  $2^{1+1-2} = 1$ ,  $\psi = 2^{2^{1+1-2}} = 2^{2^0} = 2$ .  $2^{2 \cdot 1} + 1 = 5$

$$\begin{aligned}
A_0 &= 11_2 \ll (0 \cdot 1) \bmod (5) = 11_2, \quad B_0 = 01_2 \ll (0 \cdot 1) \bmod 5 = 01_2, \\
A_1 &= 01_2 \ll (1 \cdot 1) \bmod (5) = 10_2, \quad B_1 = 01_2 \ll (1 \cdot 1) \bmod 5 = 01_2, \\
A_2 &= 00_2 \ll (2 \cdot 1) \bmod (5) = 00_2, \quad B_2 = 00_2 \ll (2 \cdot 1) \bmod 5 = 00_2, \\
A_3 &= 00_2 \ll (3 \cdot 1) \bmod (5) = 00_2, \quad B_3 = 00_2 \ll (3 \cdot 1) \bmod 5 = 00_2.
\end{aligned}$$

Крок 10.  $\omega = 2^2 = 4 = 100_2$ ,  $M = 2^{2 \cdot 2} + 1 = 17$ .

Робимо перетворення Фур'є для параметрів  $B^0 = [11_2, 10_2, 00_2, 00_2]$ ,  $\omega = 4$  та  $M = 17$ .

$$B_{00_2}^0 = 11_2, \quad B_{01_2}^0 = 10_2, \quad B_{10_2}^0 = 00_2, \quad B_{11_2}^0 = 00_2.$$

$$\begin{aligned}
B_{00_2}^1 &= B_{00_2}^0 + \omega^{00_2} B_{10_2}^0 \quad \text{mod } 17 = 11_2 + 1 \cdot 00_2 \text{ mod } 17 = 11_2, \\
B_{01_2}^1 &= B_{01_2}^0 + \omega^{00_2} B_{11_2}^0 \quad \text{mod } 17 = 10_2 + 1 \cdot 00_2 \text{ mod } 17 = 10_2, \\
B_{10_2}^1 &= B_{00_2}^0 + \omega^{10_2} B_{10_2}^0 \quad \text{mod } 17 = 11_2 + 16 \cdot 00_2 \text{ mod } 17 = 11_2, \\
B_{11_2}^1 &= B_{01_2}^0 + \omega^{10_2} B_{11_2}^0 \quad \text{mod } 17 = 10_2 + 16 \cdot 00_2 \text{ mod } 17 = 10_2.
\end{aligned}$$

$$\begin{aligned}
B_{00_2}^2 &= B_{00_2}^1 + \omega^{00_2} B_{01_2}^1 \quad \text{mod } 17 = 11_2 + 1 \cdot 10_2 \text{ mod } 17 = \\
&= 5 \text{ mod } 17 = 5,
\end{aligned}$$

$$\begin{aligned}
B_{01_2}^2 &= B_{00_2}^1 + \omega^{10_2} B_{01_2}^1 \quad \text{mod } 17 = 11_2 + 16 \cdot 10_2 \text{ mod } 17 = \\
&= 35 \text{ mod } 17 = 1,
\end{aligned}$$

$$\begin{aligned}
B_{10_2}^2 &= B_{10_2}^1 + \omega^{01_2} B_{11_2}^1 \quad \text{mod } 17 = 11_2 + 4 \cdot 10_2 \text{ mod } 17 = \\
&= 11 \text{ mod } 17 = 11,
\end{aligned}$$

$$\begin{aligned}
B_{11_2}^2 &= B_{10_2}^1 + \omega^{11_2} B_{11_2}^1 \quad \text{mod } 17 = 11_2 + 64 \cdot 10_2 \text{ mod } 17 = \\
&= 131 \text{ mod } 17 = 12.
\end{aligned}$$

Визначаємо  $A_r^1$ .

$$\begin{aligned}
A_{00_2}^1 &= B_{00_2}^2 = 5, \quad A_{01_2}^1 = B_{10_2}^2 = 11, \\
A_{10_2}^1 &= B_{01_2}^2 = 1, \quad A_{11_2}^1 = B_{11_2}^2 = 12.
\end{aligned}$$

$$A^1 = [5, 11, 1, 12].$$

Тепер робимо перетворення Фур'є для параметрів  $B^0 = [01_2, 10_2, 00_2, 00_2]$ ,  
 $\omega = 4$  та  $M = 17$ .

$$B_{00_2}^0 = 01_2, \quad B_{01_2}^0 = 10_2, \quad B_{10_2}^0 = 00_2, \quad B_{11_2}^0 = 00_2.$$

$$\begin{aligned}
B_{00_2}^1 &= B_{00_2}^0 + \omega^{00_2} B_{10_2}^0 \quad \text{mod } 17 = 01_2 + 1 \cdot 00_2 \text{ mod } 17 = 01_2, \\
B_{01_2}^1 &= B_{01_2}^0 + \omega^{00_2} B_{11_2}^0 \quad \text{mod } 17 = 10_2 + 1 \cdot 00_2 \text{ mod } 17 = 10_2, \\
B_{10_2}^1 &= B_{00_2}^0 + \omega^{10_2} B_{10_2}^0 \quad \text{mod } 17 = 01_2 + 16 \cdot 00_2 \text{ mod } 17 = 01_2, \\
B_{11_2}^1 &= B_{01_2}^0 + \omega^{10_2} B_{11_2}^0 \quad \text{mod } 17 = 10_2 + 16 \cdot 00_2 \text{ mod } 17 = 10_2.
\end{aligned}$$

$$\begin{aligned}
B_{00_2}^2 &= B_{00_2}^1 + \omega^{00_2} B_{01_2}^1 \pmod{17} = 01_2 + 1 \cdot 10_2 \pmod{17} = \\
&= 3 \pmod{17} = 3, \\
B_{01_2}^2 &= B_{00_2}^1 + \omega^{10_2} B_{01_2}^1 \pmod{17} = 01_2 + 16 \cdot 10_2 \pmod{17} = \\
&= 33 \pmod{17} = 16, \\
B_{10_2}^2 &= B_{10_2}^1 + \omega^{01_2} B_{11_2}^1 \pmod{17} = 01_2 + 4 \cdot 10_2 \pmod{17} = \\
&= 9 \pmod{17} = 9, \\
B_{11_2}^2 &= B_{10_2}^1 + \omega^{11_2} B_{11_2}^1 \pmod{17} = 01_2 + 64 \cdot 10_2 \pmod{17} = \\
&= 129 \pmod{17} = 10.
\end{aligned}$$

Визначаємо  $B_r^1$ .

$$\begin{aligned}
B_{00_2}^1 &= B_{00_2}^2 = 3, \quad B_{01_2}^1 = B_{10_2}^2 = 9, \\
B_{10_2}^1 &= B_{01_2}^2 = 16, \quad B_{11_2}^1 = B_{11_2}^2 = 10.
\end{aligned}$$

$$B^1 = [3, 9, 16, 10].$$

Крок 11.  $l + 1 = 2$ ,  $2^{2^{l+1}} + 1 = 17$ .

$$\begin{aligned}
C_0^1 &= 5 \cdot 3 \pmod{17} = 15 \pmod{17} = 15, \\
C_1^1 &= 11 \cdot 9 \pmod{17} = 99 \pmod{17} = 14, \\
C_2^1 &= 1 \cdot 16 \pmod{17} = 16 \pmod{17} = 16, \\
C_3^1 &= 12 \cdot 10 \pmod{17} = 120 \pmod{17} = 1.
\end{aligned}$$

Крок 12.

Робимо перетворення Фур'є для параметрів  $B^0 = [15, 14, 16, 1]$ ,  $\omega = 4$  та  $M = 17$ .

$$B_{00_2}^0 = 15, \quad B_{01_2}^0 = 14, \quad B_{10_2}^0 = 16, \quad B_{11_2}^0 = 1.$$

$$\begin{aligned}
B_{00_2}^1 &= B_{00_2}^0 + \omega^{00_2} B_{10_2}^0 \pmod{17} = 15 + 1 \cdot 16 \pmod{17} = 14, \\
B_{01_2}^1 &= B_{01_2}^0 + \omega^{00_2} B_{11_2}^0 \pmod{17} = 14 + 1 \cdot 1 \pmod{17} = 15, \\
B_{10_2}^1 &= B_{00_2}^0 + \omega^{10_2} B_{10_2}^0 \pmod{17} = 15 + 16 \cdot 16 \pmod{17} = 16, \\
B_{11_2}^1 &= B_{01_2}^0 + \omega^{10_2} B_{11_2}^0 \pmod{17} = 14 + 16 \cdot 1 \pmod{17} = 13.
\end{aligned}$$

$$\begin{aligned}
B_{00_2}^2 &= B_{00_2}^1 + \omega^{00_2} B_{01_2}^1 \pmod{17} = 14 + 1 \cdot 15 \pmod{17} = 12, \\
B_{01_2}^2 &= B_{00_2}^1 + \omega^{10_2} B_{01_2}^1 \pmod{17} = 14 + 16 \cdot 15 \pmod{17} = 16, \\
B_{10_2}^2 &= B_{10_2}^1 + \omega^{01_2} B_{11_2}^1 \pmod{17} = 16 + 4 \cdot 13 \pmod{17} = 0, \\
B_{11_2}^2 &= B_{10_2}^1 + \omega^{11_2} B_{11_2}^1 \pmod{17} = 16 + 64 \cdot 13 \pmod{17} = 15.
\end{aligned}$$

Визначаємо  $C_r$ .

$$\begin{aligned}
C_{00_2} &= B_{00_2}^2 = 12, \quad C_{01_2} = B_{10_2}^2 = 0, \\
C_{10_2} &= B_{01_2}^2 = 16, \quad C_{11_2} = B_{11_2}^2 = 15.
\end{aligned}$$

$$C = [12, 0, 16, 15].$$

Крок 13.

$$\begin{aligned}
W_0^2 &= (12 \ll (4 \cdot 2 - 2)) \pmod{17} = 12 \cdot 64 \pmod{17} = 3, \\
W_1^2 &= (15 \ll (4 \cdot 2 - 2 - 1 \cdot 1)) \pmod{17} = 15 \cdot 32 \pmod{17} = 4, \\
W_2^2 &= (16 \ll (4 \cdot 2 - 2 - 2 \cdot 1)) \pmod{17} = 16 \cdot 16 \pmod{17} = 1, \\
W_1^2 &= (0 \ll (4 \cdot 2 - 2 - 3 \cdot 1)) \pmod{17} = 0 \cdot 8 \pmod{17} = 0.
\end{aligned}$$

Крок 14.

$$\begin{aligned}
W_0^3 &= ((3 - 3) \pmod{2^2}) \ll (2 \cdot 2) + (3 - 3) \pmod{2^2} + 3 = 3, \\
W_1^3 &= ((0 - 4) \pmod{2^2}) \ll (2 \cdot 2) + (0 - 4) \pmod{2^2} + 4 = 4, \\
W_2^3 &= ((1 - 1) \pmod{2^2}) \ll (2 \cdot 2) + (1 - 1) \pmod{2^2} + 1 = 1, \\
W_3^3 &= ((0 - 0) \pmod{2^2}) \ll (2 \cdot 2) + (0 - 0) \pmod{2^2} + 0 = 0.
\end{aligned}$$

Крок 15.

$$\begin{aligned} 3 < (0 + 1)2^{2 \cdot 2} &\implies W_0 = 3 = 11_2, \\ 4 < (1 + 1)2^{2 \cdot 2} &\implies W_1 = 4 = 100_2, \\ 1 < (2 + 1)2^{2 \cdot 2} &\implies W_2 = 1, \\ 0 < (3 + 1)2^{2 \cdot 2} &\implies W_3 = 0. \end{aligned}$$

Крок 16.

$$\begin{aligned} 7 \cdot 5 \text{ mod } 257 &= 11_2 + 100_2 \ll (1 \cdot 2) + 1 \ll (2 \cdot 2) + 0 \ll (3 \cdot 2) = \\ &= 11_2 + 10000_2 + 10000_2 + 0 = 100011_2 = 35. \end{aligned}$$

Таким чином, ми отримали правильний результат.

## 2.10. Знаходження оберненого числа та швидке ділення

Нехай ми маємо довге ціле невід'ємне число  $A = a_1a_2\dots$  у двійковому записі. Алгоритм Кука дозволяє для дробового числа  $V = 0.a_1a_2\dots$  знайти число  $X$ , записане у форматі  $xxxx.xxxx$ , таке що виконується нерівність

$$|X - 1/V| \leq 2^{-n},$$

де  $n$  — довільне наперед задане додатне число,  $1/V$  — математично точне значення оберненого до  $V$  числа.

Перш ніж описувати алгоритм Кука, необхідно зазначити, що для його реалізації необхідно вміти зберігати та обробляти довгі невід'ємні дробові числа. Це можна зробити, утворивши клас-обгортку ULF для класу ULI, який в якості полів буде містити довге ціле число  $N$  та ще один параметр point, який вказуватиме на місцезнаходження в масиві цифр точки, що розділяє цілу та дробову частини числа.

```
class ULF {
```

```
...
```

```

private:
    ULI N;
    unsigned int point;
};

```

Усі операції, визначені для ULI, доволі простим чином реалізуються для ULF.

### Алгоритм Кука.

Вхідні дані: двійкове довге невідємне дробове число  $V = 0.v_1v_2\dots$ , додатне число  $n$ .

1. Визначити  $X = \frac{1}{4} \lfloor \frac{32}{4v_1+2v_2+v_3} \rfloor$ ,  $k = 0$ .
2. За допомогою операції швидкого множення знайти число  $U = V_k X^2$ , де  $V_k = (0.v_1v_2\dots v_{2^{k+1}+3})$ .
3. Оновити  $X = 2X - U$  та округлити його вгору до  $2^{k+1} + 1$  знака після крапки.
4. Інкрементувати  $k$ .
5. Якщо  $2^k < n$ , повернутися на крок 2, інакше — повернути число  $X$ .

Фактично алгоритм Кука — це ітераційний процес розв'язання рівняння  $1/x - V = 0$  методом дотичних (методом Ньютона). Детальніше про цей метод можна дізнатися в літературі з чисельних методів.

Цілочисельне ділення двох довгих невід'ємних цілих чисел відбувається за тим принципом, що поділити на якесь число — це все одно, що помножити на обернене до нього. Таким чином в діленні використовується алгоритм Кука.

### Алгоритм цілочисельного ділення.

Вхідні дані: довгі числа  $U$ ,  $V$  довжиною  $n$ .

Вихідні дані: Довге число  $Q$ , таке що  $0 \leq U - QV < V$ .

1. Якщо  $U < V$ , повернути 0. Інакше виконати кроки 2-4.

2. За допомогою алгоритма Кука знайти число  $X$ , обернене до  $V$  з точністю  $n$ .
3. За допомогою швидкого множення знайти добуток  $Q = XU$ .
4. Перевірити, що  $0 \leq U - QV < V$ . Якщо нерівності виконуються, повернути  $Q$ . Якщо ні, то збільшити або зменшити  $Q$  на 1 в залежності від того, яка саме нерівність не виконується. Повторити цей крок.

Якщо в якості алгоритму швидкого множення використовувати алгоритм Штрассена, то асимптотична складність як пошуку оберненого числа, так і ділення складатиме  $O(n \ln n \ln \ln n)$ .

На основі алгоритму ділення доволі легко написати алгоритм, що реалізує знаходження остачі від ділення. Таким чином, ми можемо доповнити таблицю операцій з пункту 2.3 операціями '/' та '%' з відповідними асимптотиками.

### 2.11. Рекомендації до написання лабораторної роботи без використання бібліотеки STL

Єдиний контейнер зі стандартної бібліотеки, який використовується у цій лабораторній, це `std::vector`. Тобто, для того, щоб позбутися залежності від STL, потрібно лише побудувати свій клас-обгортку для динамічного масиву цифр.

Задача це нескладна, єдині дві операції, які потребують обговорення, — це розширення масиву за потреби та перевірка, чи не потрібно масив зменшувати.

Обидві ці операції відбуваються за наступною схемою:

1. Створення нового масиву.
2. Копіювання даних.
3. Видалення старого масиву.

Дані операції мають лінійну асимптотику по кількості елементів масива цифр і не впливають на асимптотику більшості операцій. Проте як розширення, так і зменшення масиву може бути спричинене операцією, яка дає доступ до окремої цифри. Ця операція повинна мати константну асимптотику, але досягти її можливо лише “в середньому”.

Так, якщо масив потребує розширення, його варто збільшувати одразу вдвічі. Тоді за “нормальної” роботи з цифрами його ще деякий не треба буде збільшувати, а, отже, і викликати відповідну лінійну функцію.

Так само і з процедурою зменшення. Викликається вона, лише якщо верхня половина розрядів у поточному масиві зберігає нулі. І зменшення також відбувається вдвічі.

Звісно, замість збільшення та зменшення вдвічі, можна реалізувати і зміну втричі розміру масиву, або в чотири рази, або визначити для цього довільну константу. При цьому варто орієнтуватися на те, якої довжини бувають числа в програмі та наскільки сильно ця довжина може змінюватися.

## 2.12. Алгоритми перевірки на простоту

### Алгоритм Люка-Лемера.

Вхідні дані: число Мерсенна  $M_q$  (число вигляду  $2^q - 1$ ).

Вихідні дані: однозначна відповідь “так” або “ні” на питання “чи є число  $M_q$  простим?”

1. Обробити випадок  $q < 3$  як сукупність часткових випадків.
2. Визначити послідовність довгих чисел  $L_0 = 4$ ,  $L_{n+1} = (L_n^2 - 2) \bmod M_q$ .
3. Обчислити  $L_{q-2}$ . Якщо результатом є 0, повернути відповідь так, інакше — ні.

Зрозуміло, що для використання цього методу зручно зберігати числа в двійковому записі. Тоді операцію взяття залишку можна обчислювати за

алгоритмом з пункту 2.6.

Наступні три алгоритми є ймовірносними, тобто вони не дають точну відповідь, чи є число простим чи ні, проте відповідь може бути правильною з певною ймовірністю.

Фактично, результати всіх трьох методів можна описати однаково: якщо метод каже, що число складене, то воно точно складене, якщо метод каже, що число просте, то воно може бути як простим, так і складеним з певною ймовірністю. Щоб підвищити ймовірність правильної відповіді, тести Рабіна-Міллера та Соловея-Штрассена використовують раунди перевірки. Чим більша кількість раундів, тим надійніша відповідь.

В алгоритмах Рабіна-Міллера та Соловея-Штрассена використовуються випадкові довгі числа. Отримати таке число можна різними способами: або адаптувати алгоритми першої лабораторної роботи так, щоб вони працювали з довгими цілими числами, або генерувати кожну цифру випадкового довгого числа, або генерувати одразу блоки цифр випадкового довгого числа. Конкретна стратегія генерації довгого випадкового числа залишається на розсуд студента.

### **Алгоритм Рабіна-Міллера.**

Вхідні дані: довге непарне число  $N > 2$ , кількість раундів  $k$ .

Вихідні дані: відповідь з певною ймовірністю.

1. Визначити  $s$  та  $T$ , такі що  $2^s T = N - 1$ .
2. Повторити кроки 3-10  $k$  разів (раундів). Якщо під час виконання цих раундів алгоритм не зупиниться, перейти на крок 11.
3. Згенерувати довге випадкове ціле число  $A$  з проміжку  $[2, N - 2]$ .
4. Визначити  $X = A^T \bmod N$ .
5. Якщо  $X = 1$  або  $X = N - 1$ , перейти до наступного раунду, інакше — перейти на крок 6.
6. Виконати кроки 7-10  $s - 1$  разів.

7. Оновити  $X = X^2 \bmod N$ .
8. Якщо  $X = 1$ , перейти на наступну ітерацію циклу по  $s$ , інакше — перейти на крок 9.
9. Якщо  $X = N - 1$ , перейти до наступного раунду, інакше — перейти на крок 10.
10. Повернути результат “складене”
11. Повернути результат “ймовірно просте”.

Вираз на кроці 4 обчислюється за допомогою алгоритмів швидкого зведення до степені за модулем. Інакше проміжні обчислення займають дуже багато часу.

### Алгоритм Солов'єя-Штрассена.

Вхідні дані: довге непарне число  $N > 2$ , кількість раундів  $k$ .

Вихідні дані: відповідь з певною ймовірністю.

1. Повторити кроки 2-4  $k$  разів (раундів). Якщо під час виконання цих раундів алгоритм не зупиниться, перейти на крок 5.
2. Згенерувати довге випадкове ціле число  $A$  з проміжку  $[2, N - 2]$ .
3. Якщо найбільший спільний дільник чисел  $A$  та  $N$  більше за 1, повернути результат “складене”.
4. Якщо  $A^{\frac{N-1}{2}} \neq \text{Jacobi}(A, N) \bmod N$ , повернути результат “складене”. Тут  $\text{Jacobi}(A, N)$  — символ Якобі.
5. Повернути результат “ймовірно просте”.

Обчислення НСД двох чисел відбувається за алгоритмом Євкліда, обчислення символу Якобі для двох чисел виконується за наступним рекурсивним алгоритмом.

### Символ Якобі $\text{Jacobi}(A, N)$ .

Вхідні дані: числа  $A, N$ .

Вихідні дані: одне з чисел  $-1, 0, 1$ .

1. якщо  $A = 0$ , повернути  $0$ .
2. якщо  $A = 1$ , повернути  $1$ .
3. Визначити число  $e$  та непарне число  $A_1$ , такі що  $A = 2^e A_1$
4. Якщо  $e$  — парне, визначити  $s = 1$ . Інакше, якщо  $N = \pm 1 \pmod 8$ , визначити  $s = 1$ . Інакше визначити  $s = -1$ .
5. Якщо  $N = 3 \pmod 4$  і  $A_1 = 3 \pmod 4$ , змінити знак  $s$ .
6. Визначити  $N_1 = N \pmod{A_1}$ .
7. Якщо  $A_1 = 1$ , повернути  $s$ . Інакше повернути  $s \cdot \text{Jacobi}(N_1, A_1)$ .

Метод Фробеніуса використовує алгебраїчні операції над виразами типу  $A + B\sqrt{c}$ , де  $A, B, c$  — деякі цілі числа. Такі вирази називаються квадратичними ірраціональностями. Додавання цих виразів працює за правилом додавання компонент

$$(A_1 + B_1\sqrt{c}) + (A_2 + B_2\sqrt{c}) = (A_1 + A_2) + (B_1 + B_2)\sqrt{c}.$$

Віднімання працює аналогічно. При множенні використовується величина  $c$ .

$$(A_1 + B_1\sqrt{c})(A_2 + B_2\sqrt{c}) = (A_1A_2 + B_1B_2c) + (A_2B_1 + A_1B_2)\sqrt{c}.$$

Взяття такого виразу за модулем цілого числа також відбувається покомпонентно.

$$(A + B\sqrt{c}) \pmod N = (A \pmod N) + (B \pmod N)\sqrt{c}.$$

### **Алгоритм Фробеніуса.**

Вхідні дані: довге непарне число  $N > 2$ .

Вихідні дані: відповідь з певною ймовірністю.

1. Визначити число  $c$  як найменше число з послідовності  $-1, 2, 3, 5, 7, 11, 13, \dots$ , таке що  $Jacobi(c, N) = -1$ .
2. Якщо  $c \leq 2$ , то визначити  $A = 2$ , інакше  $A = 1$ .
3. Визначити  $X = A + \sqrt{c}$ .
4. Якщо  $X^N \bmod N = A + (N - 1)\sqrt{c}$ , повернути результат “ймовірно просте”, інакше — “складене”

Як видно, даний алгоритм також використовує символ Якобі. В пункті 4 для обчислень знову доцільно використовувати алгоритми швидкого зведення до степені за модулем.

## РОЗДІЛ 3

### ЛАБОРАТОРНА РОБОТА №3: АЛГОРИТМИ ОБЧИСЛЮВАЛЬНОЇ ГЕОМЕТРІЇ

Алгоритми обчислювальної геометрії стосуються задач, які виникають під час роботи з графікою, оскільки більшість обчислень в ній мають геометричний зміст. Як і в попередній лабораторній, критичним показником є швидкість алгоритмів.

Третя лабораторна складається з двох блоків: перші два — пов'язані між собою алгоритми побудови діаграми Вороного та тріангуляції Делоне, та чотири алгоритми побудови опуклої оболонки. Ці задачі є доволі розповсюдженими в графіці. Вхідними даними в цій лабораторній завжди є масив точок на площині з координатами типу **double**. У якості вихідних даних пропонується використовувати спеціальну структуру подвійно зв'язаного списку ребер ПЗСР (DCEL), яка є доволі зручною для роботи з плоскими графами. Сформований ПЗСР далі передається в допоміжну програму для виведення графіки.

Методичні рекомендації будуть стосуватися саме принципів роботи алгоритмів. Спосіб виведення графіки та бібліотеки, які при цьому використовуються, оглядатися не будуть.

У зв'язку з тим, що в цій лабораторній робота ведеться зі значеннями типу **float** та **double**, рекомендується порівнювати ці числа з деякою точністю  $\epsilon$  для уникнення проблем, пов'язаних з втратою точності в малих розрядах. Так, варто визначити глобальну змінну  $\epsilon$  з достатньо малим значенням ( $\approx 10^{-15}$ ) та якщо треба виконати, наприклад, перевірку  $a < b$ , реалізувати перевірку  $a < b + \epsilon$ . В інших операціях порівняння слід робити аналогічно.

### 3.1. Умова

1. Реалізувати алгоритм Форчуна побудови діаграми Вороного для набору точок на площині.

2. Здійснити триангуляцію Делоне для набору точок на площині.

Реалізувати наступні алгоритми побудови опуклої оболонки набору точок на площині:

3. Алгоритм Кейла-Кіркпатріка

4. Алгоритм Ендрю-Джарвіса

5. Алгоритм Грехема.

6. Рекурсивний алгоритм.

### 3.2. Діаграма Вороного, алгоритм Форчуна та триангуляція Делоне

Нехай на площині обрано  $n$  точок  $p_1, p_2, \dots, p_n$  з різними координатами. Побудова діаграми Вороного — це задача розбиття площини на  $n$  підмножин (граней)  $A_1, A_2, \dots, A_n$ , так щоб кожна грань містила одну точку з початкового набору ( $p_i \in A_i$ ). Оскільки працювати з необмеженими гранями не дуже зручно, надалі вважатимемо, що замість площини ми розглядаємо достатньо великий прямокутник, що містить всі точки  $p_i$ . Тоді всі грані будуть обмеженими.

Проте не будь-яке розбиття прямокутника на грані є діаграмою Вороного. Діаграма Вороного характеризується наступною властивістю: якщо взяти точку  $(x, y)$  з грані  $A_i$  і порахувати всі відстані від неї до точок  $p_1, p_2, \dots, p_N$ , то відстань до точки  $p_i$  з цієї ж грані буде найменшою (рис. 3.1)

Нескладно довести, що кожна грань в діаграмі Вороного є опуклим багатокутником, причому, якщо дві грані  $A_i, A_j$  є суміжними, то сторона,

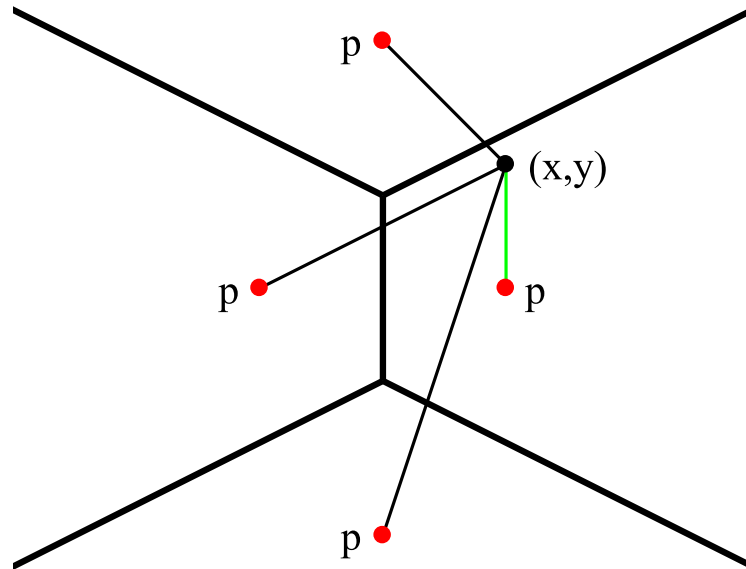


Рис. 3.1: Діаграма Вороного та її основна властивість.

якою вони дотикаються, лежить на серединному перпендикулярі до відрізка  $p_i p_j$ . Звідси можна отримати примітивний алгоритм побудови діаграми Вороного: для кожної точки  $p_i$  треба побудувати свій опуклий багатокутник, що обмежується серединними перпендикулярами до інших точок. Проте даний алгоритм має асимптотичну складність  $O(n^2)$ , бо фактично має вкладений цикл по всім точкам  $p_i$ .

Алгоритм Форчуна для побудови діаграми Вороного має асимптотичну складність  $O(n \ln n)$ , що є суттєво кращим результатом. Досягається цей результат завдяки оригінальній ідеї побудови діаграми та використанню спеціально визначених та тісно пов'язаних між собою таких структур даних, як бінарне дерево пошуку, черга з пріоритетами та ПЗСР.

Побудова триангуляції Делоне є двоїстою задачею до побудови діаграми Вороного. Це означає, що як тільки буде сформована структура ПЗСР, що описує діаграму Вороного для набору точок  $p_i$ , то шляхом нескладних (як асимптотично, так і алгоритмічно) перетворень можна побудувати триангуляцію Делоне.

По суті триангуляція Делоне є розбиттям опуклої оболонки точок  $p_i$  на трикутники, вершини яких знаходяться в точках  $p_i$ . Причому, якщо описати коло навколо будь-якого з цих трикутників, то всередині нього не буде жодної з точок  $p_i$  (рис. 3.2).

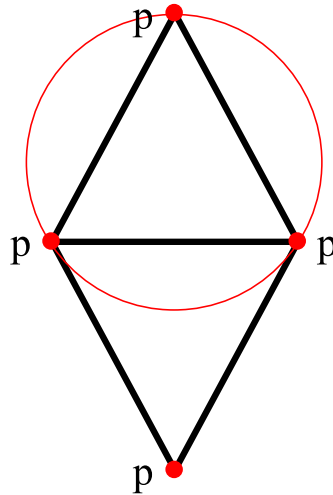


Рис. 3.2: Триангуляція Делоне та її основна властивість.

### 3.3. Алгоритм Форчуна: Загальний опис

Алгоритм Форчуна використовує так звану замітаючу пряму. Принцип роботи цієї прямої полягає в тому, що вона рухається в певному напрямку на площині (згори — донизу) і там, де вона вже пройшла, формується структура діаграми Вороного. Там, куди вона ще не дісталася, жодних відомостей про структуру діаграми немає (рис. 3.3).

Замітаюча пряма працює з певною затримкою, бо насправді структура діаграми Вороного будується лише вище берегової лінії, що є кусково заданою функцією, де кожна з окремих функцій є частиною параболи (дугою). Ці дуги характеризуються точками  $p_i$ , які вже були пройдені замітаючою прямою, та  $y$ -координатою самої прямої. Фактично кожна з парабол за-

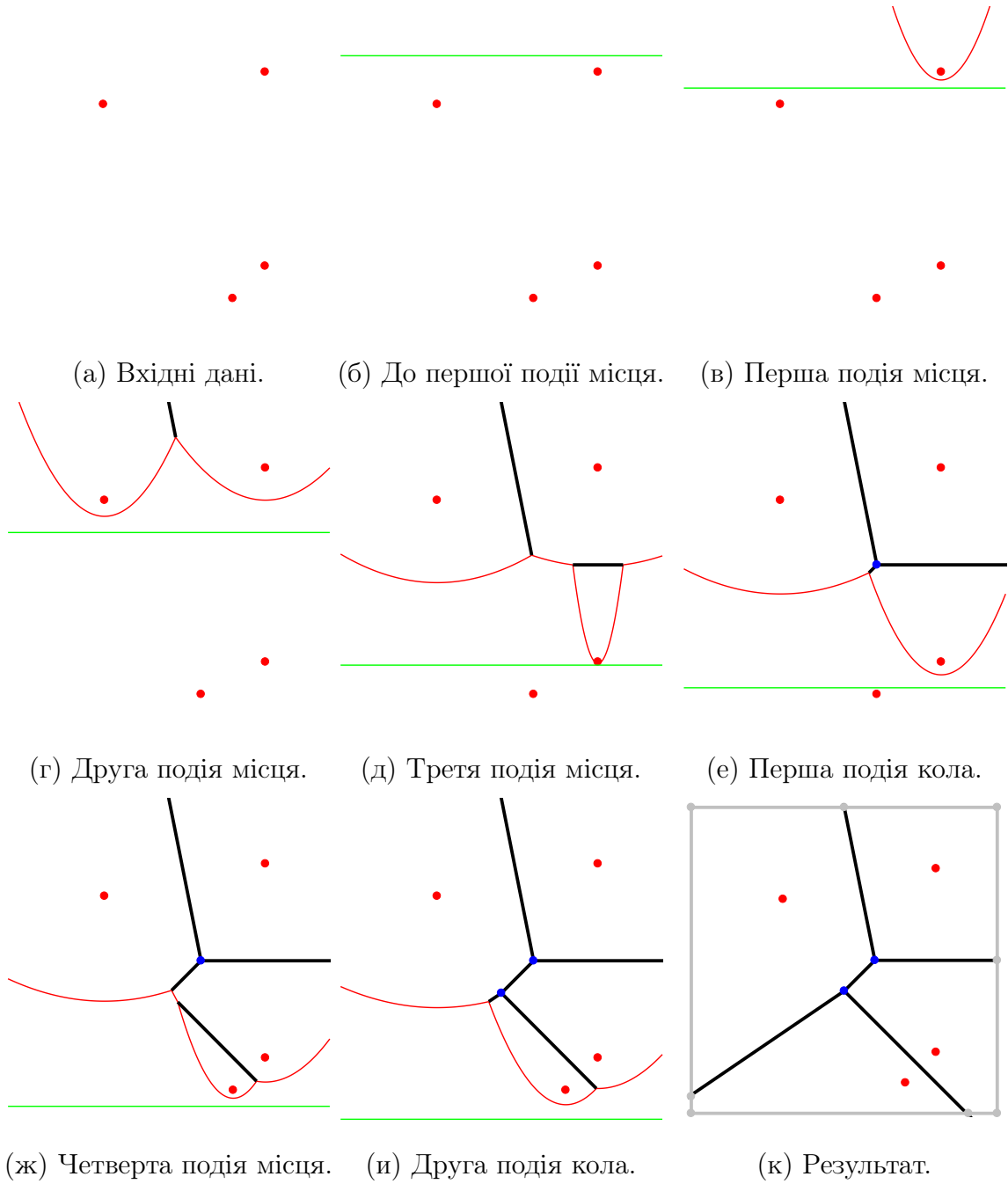


Рис. 3.3: Побудова діаграми Вороного за допомогою алгоритму Форчуна.

дається через свої фокус та директрису. Зміна  $y$ -координати замітаючої прямої спричиняє зміни в береговій лінії: деякі дуги розширюються, інші — стискаються. Місця перетину сусідніх дуг берегової лінії (так звані “точки зустрічі”) зміщуються та своїм рухом описують нові ребра діаграми Вороного. Зміни у структурі берегової лінії називаються подіями. Події бувають двох типів: **подія місця** та **подія кола**.

Подія місця настає, коли замітаюча пряма досягає нової точки  $p_i$ . Тоді “навколо” цієї точки починає зростати нова парабола, яка розриває дугу над  $p_i$ . Тобто берегова лінія стає складнішою (рис. 3.4).

Подія кола стається, коли одна з дуг затискається сусідніми дугами. На місці зникнення дуги утворюється вершина діаграми Вороного, в цій вершині закінчуються ребра, що відповідали старим точкам зустрічі, та починається нове ребро, яке відповідає новій точці зустрічі. Берегова лінія стає меншою на одну дугу (рис. 3.5). Зникнення дуги можна передбачити заздалегідь, оскільки берегова лінія рухається за складними, але відомими законами. Таким чином зникненню дуги, якщо таке планується, ставиться у відповідність певна  $y$ -координата і ми знаємо, що коли замітаюча пряма досягне цієї координати, дуга зникне, якщо нічого їй до цього не завадить і наше передбачення не зміниться.

Тобто ми маємо 2 типи подій, кожна подія має свою  $y$ -координату та кожна стається, коли замітаюча пряма досягає цієї координати. При цьому очевидно, що події треба обробляти в тій черзі, в якій їх досягає замітаюча пряма. Це означає, що всі події можна поєднати в чергу з пріоритетами, що є однією з основних структур алгоритму Форчуна.

Берегова лінія складається з дуг та точок зустрічі і основні дії, які відбуваються над нею, — це додавання нової дуги, видалення дуги та пошук дуги, що має розбитися новою дугою. Очевидно, що для таких цілей в якості структури даних для зберігання берегової лінії підійде бінарне дерево пошуку.

Тобто ми отримали 3 основні структури даних для опису алгоритму Форчуна та геометричну інтерпретацію до них. Бінарне дерево пошуку відповідає за берегову лінію та змінюється відповідно до настання подій місця або подій кола. Черга з пріоритетами зберігає події місця та події кола та впливає на берегову лінію. В той самий час зміни в береговій лінії можуть спричинити зміни в черзі подій. Зміни в береговій лінії також спричиняють зміни в ПЗСР, яке нам потрібно побудувати.

Тепер залишилося тільки прибрати саму замітаючу пряму. Дійсно, вона нам не потрібна як окрема структура, оскільки між подіями вона ніяк не впливає на жодну з перелічених структур, а координати всіх подій зберігаються в черзі з пріоритетами як ключі. Таким чином ми отримуємо “стрибкоподібне” замітання, де пряма може розглядатися як певний об'єкт лише під час обробки подій.

Поєднаємо усі наші міркування в один алгоритм.

### **Алгоритм Форчуна.**

Вхідні дані: набір точок  $\{p_1, p_2, \dots, p_n\}$  з різними  $y$ -координатами.

Вихідні дані: ПЗСР діаграми Вороного.

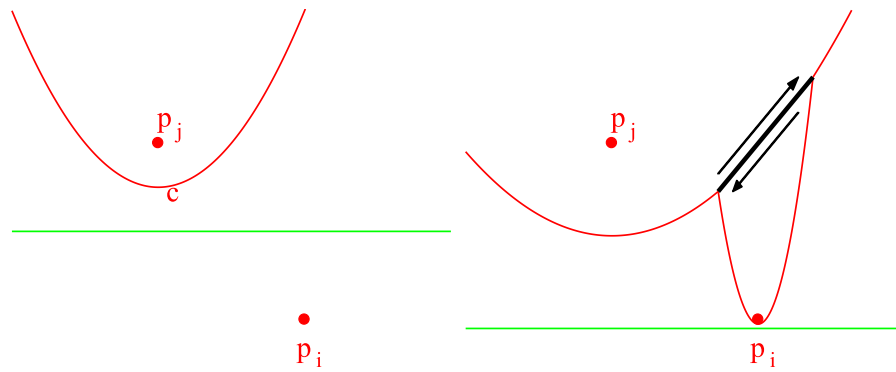
1. Додати всі точки  $\{p_1, p_2, \dots, p_n\}$  в чергу подій  $Q$  як події місця з ключами, які дорівнюють  $y$ -координатам точок.
2. Поки черга не порожня, виконувати крок 3. Коли черга  $Q$  стане порожньою, перейти до кроку 4.
3. Дістати з черги  $Q$  подію з найбільшим пріоритетом. Якщо це подія місця — **Обробити подію місця**, якщо це подія кола — **Обробити подію кола**.
4. Додати до ПЗСР відомості про охоплюючий прямокутник та обробити “напівнескінченні” ребра, які залишилися після виконання попередніх кроків.

**Обробка події місця  $p_i$ .**

1. Якщо бінарне дерево пошуку  $T$  порожнє, вставити в нього дугу, яка відповідає  $p_i$ , без події кола та вийти з процедури. Інакше виконати кроки 2-6.
2. Знайти дугу  $c$  в  $T$ , яка знаходиться вертикально над точкою  $p_i$ . Для цього використовувати результати порівняння координати  $p_i.x$  з **координатами точок зустрічі** сусідніх з  $c$  дуг.
3. Якщо  $c$  має вказівник на подію кола, то видалити цю подію з черги  $Q$ .
4. Замінити вузол  $c$  на піддерево з рис. 3.4г. Дане піддерево складається з трьох дуг та двох точок зустрічі між ними. Пов'язати середню дугу з точкою  $p_i$ , бокові дуги з точкою  $p_j$ , до якої була прив'язана дуга  $c$  до свого видалення. У ліву та праві точки зустрічі зберегти кортежі  $\langle p_j, p_i \rangle$  та  $\langle p_i, p_j \rangle$  відповідно.
5. Створити дві нові пари близнюків-півребер в ПЗСР  $D$ , кожна пара півребер відповідає новій точці зустрічі та розділяє точки  $p_i$  та  $p_j$ . Зберегти посилання на ці півребра в нових точках зустрічі.
6. Перевірити **умову зникнення** лівої дуги з щойно доданого піддерева. Якщо умова виконується, то додати до  $Q$  подію кола з відповідним пріоритетом та додати в цю подію посилання на дугу, що зникає, та навпаки. Зробити те ж саме для правої дуги з піддерева.

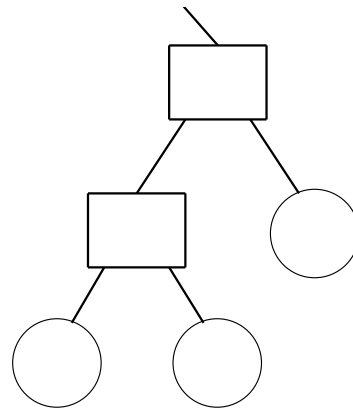
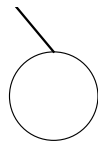
### Обробка події кола.

1. Перейти по посиланню в події кола до дуги  $c_i$  в дереві  $T$ .
2. Видалити вузол, що відповідає дузі  $c_i$  та дві сусідні з ним точки зустрічі. Замість них додати нову точку зустрічі між двома (тепер сусідніми) дугами  $c_s, c_r$  (див рис. 3.5).
3. Додати до ПЗСР  $D$  вершину з координатами в центрі кола, яке проходить через точки  $p_s, p_i, p_r$ , які відповідають дугам  $c_s, c_i, c_r$  відповідно.



(а) До події місяця.

(б) Після події місяця.



(в) До події місяця.

(г) Після події місяця.

Рис. 3.4: Зміни в береговій лінії та ПЗСР під час обробки події місяця.

4. Додати в ПЗСР  $D$  два напівребра-блюзнюка, які розділяють точки  $p_s$  та  $p_r$  та додати до нової точки зустрічі посилання на ці напівребра.
5. Пов'язати між собою нову вершину ПЗСР, два нові напівребра та чотири старі напівребра, на які посилалися видалені точки зустрічі, згідно з правилами заповнення полів в структурі ПЗСР.
6. Видалити з черги  $Q$  можливі події кіл, які відповідають дугам  $c_s$  та  $c_r$ .
7. Перевірити **умову зникнення дуги**  $c_s$  в оновленому дереві  $T$ . Якщо умова виконується, то додати до  $Q$  подію кола з відповідним пріоритетом та додати в цю подію посилання на дугу, що зникає, та навпаки. Зробити те ж саме для дуги  $c_r$ .

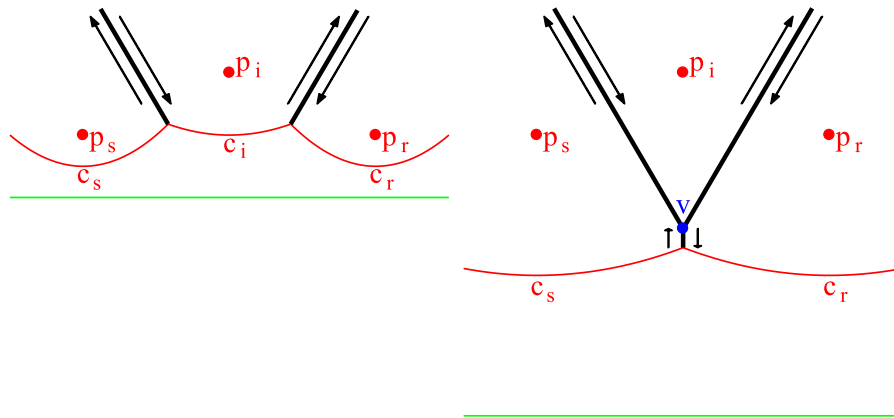
### Обрахунок координати точки зустрічі.

Для пошуку дуги, що знаходиться над новою подією місця, нам потрібно порівнювати  $x$ -координату події місця з  $x$ -координатами точок зустрічі на береговій лінії. А для цього треба визначити алгоритм обрахунку координат кожної точки зустрічі. Робиться це тривіальним чином, використовуючи знання про дуги берегової лінії.

Кожна точка зустрічі з'єднує дві сусідні дуги берегової лінії. Ці дуги є параболою і формула однієї такої параболи залежить від координати точки  $p_i$ , яка цю параболу породила, та  $y$ -координати замітаючої прямої  $L$  ( $y$ -координата нової події місця) за наступною формулою.

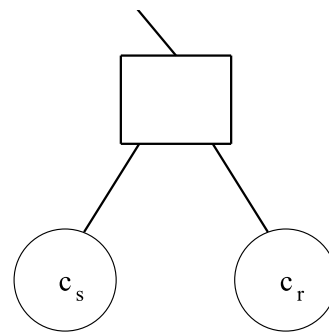
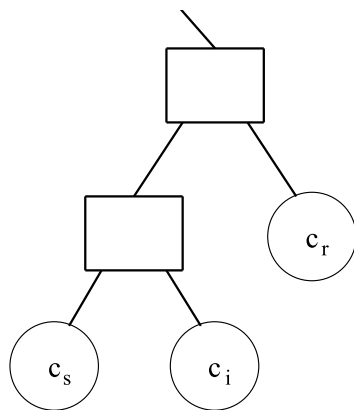
$$y = \frac{(x - p_i.x)^2}{2(p_i.y - L)} + \frac{p_i.y + L}{2}. \quad (3.1)$$

Дві сусідні дуги обов'язково породжуються двома різними точками  $p_i$  і  $p_j$ , а оскільки значення  $y$ -координат цих точок різні, то ці дві параболи мають рівно 2 точки перетину. Одна з цих точок і є необхідною нам точкою зустрічі. Щоб обрати правильний розв'язок рівняння 3.1, нам треба



(а) До події кола.

(б) Після події кола.



(в) До події кола.

(г) Після події кола.

Рис. 3.5: Зміни в береговій лінії та ПЗСР під час обробки події кола.

використати знання про те, що для даної точки зустрічі зліва знаходиться дуга, породжена саме  $p_i$ , а справа — породженна  $p_j$ . Ця додаткова інформація однозначно визначає координати точки зустрічі, коли замітаюча пряма знаходиться на певному фіксованому рівні.

### Перевірка умови зникнення дуги.

Для того, щоб перевірити, що в послідовній трійці дуг  $c_{k-1}$ ,  $c_k$ ,  $c_{k+1}$  дуги  $c_{k-1}$ ,  $c_{k+1}$  збираються затискати дугу  $c_k$ , нам достатньо знати координати точок  $p_s$ ,  $p_i$ ,  $p_r$ , які породжують вказані три дуги. Тоді умова зникнення центральної дуги записується наступним чином.

$$\begin{vmatrix} p_i \cdot x - p_s \cdot x & p_r \cdot x - p_s \cdot x \\ p_i \cdot y - p_s \cdot y & p_r \cdot y - p_s \cdot y \end{vmatrix} < 0.$$

Якщо дана умова виконується, то в чергу додається подія кола з пріоритетом, який обчислюється як  $y$ -координата найнижчої точки кола, яке проходить через точки  $p_s, p_i, p_r$ .

### Обробка напівнескінчених ребер в ПЗСР та додавання охоплюючого прямокутника.

Суть кроку 4 полягає в доповненні ПЗСР до завершеної форми, оскільки деякі півребра можуть не мати наступного ребра при обході нескінченної грані, або ж не мати точки початку. Це доповнення відбувається за рахунок додавання “фіктивних” вершин та ребер, які за своїм положенням на площині знаходяться на охоплюючому квадраті. На рис. 3.3к ці ребра та вершини позначені сірим кольором. Координати цих вершин знаходяться за допомогою інформації про напівнескінченні ребра з побудованої на кроці 3 незавершеної структури ПЗСР. Також додається одна “фіктивна” зовнішня до охоплюючого прямокутника грань.

### Коректність вхідних даних.

Також наостанок варто зазначити, що якщо початковий набір точок містить дві точки з однаковою  $y$ -координатою, то для того, щоб забезпечити коректність вхідних даних, можна повернути цей набір точок відносно деякого полюса  $P$  на певний кут  $\alpha$ . При цьому і сам полюс, і кут можна обирати випадковим чином, ймовірність того, що після такого повороту “новий” набір точок також буде мати точки з однаковою  $y$ -координатою, буде дуже близькою до 0. Тоді після завершення виконання алгоритму Форчуна треба лише повернути всю діаграму Вороного на кут  $-\alpha$  відносно того ж самого полюса.

### 3.4. Алгоритм Форчуна: Подвійно зв'язаний список ребер

Найпростішою структурою з присутніх в алгоритмі Форчуна є подвійно зв'язаний список ребер. По суті це структура, що складається з трьох списків: списку вершин, списку граней та списку напівребер. Кожне звичайне ребро складеться з двох напівребер, що направлені в протилежні сторони по лінії ребра.

Ци списки пов'язані між собою системою вказівників:

- кожна вершина містить свої координати та вказівник на одне з напівребер, яке починаються в цій вершині;
- кожна грань містить координати якоїсь з точок, що належить цій грані, та вказівник на напівребро, для якого дана грань знаходиться зліва;
- кожне напівребро містить вказівник на симетричне напівребро (близнюка), на вершину, з якої воно починається, на грань, яка знаходиться зліва та на напівребро, яке є наступним в обході вказаної грані.

Оскільки кожна точка  $p_i$  має належати власній грані, то в якості ко-

ординат грані можна використовувати координати точки  $p_i$ . Для побудови діаграми Вороного це є правилом.

Для прикладу можна розглянути граф на рис. 3.6 та його представлення за допомогою подвійно зв'язаного списку ребер (табл. 3.1, 3.2, 3.3).

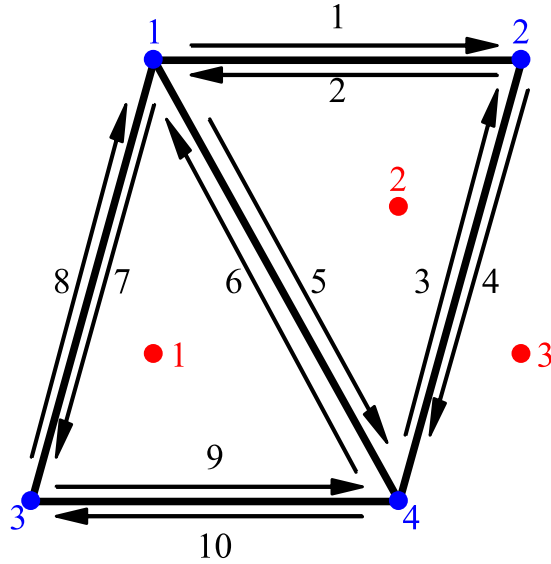


Рис. 3.6: Приклад ПЗСР.

Таблиця 3.1

**Структура ПЗСР з рис. 3.6: грані.**

Номер грані	Координати	Напівребро
1	$(-1.5, 0)$	7
2	$(1, 1)$	2
3	$(2, 0)$	4

З означення ПЗСР можна вивести конкретні дії, необхідні для встановлення всіх коректних вказівників під час обробки події кола в алгоритмі Форчуна.

Таблиця 3.2

**Структура ПЗСР з рис. 3.6: вершини.**

Номер вершини	Координати	Напівребро
1	$(-1, 2)$	1
2	$(2, 2)$	2
3	$(-2, -1)$	9
4	$(1, -1)$	6

Таблиця 3.3

**Структура ПЗСР з рис. 3.6: напівребра.**

Номер напівребра	Вершина	Грань	Близнюк	Наступне
1	1	3	2	4
2	2	2	1	5
3	4	2	4	2
4	2	3	3	10
5	1	2	6	3
6	4	1	5	7
7	1	1	8	9
8	3	3	7	1
9	3	1	10	6
10	4	3	9	8

Для реалізації даної структури доцільно використувувати `std::list`, а в якості вказівників зберігати ітератори на відповідні списки. Тоді структура ПЗСР буде мати наступний вигляд.

```

struct Point {
    double x;
    double y;
};
struct Site {
    Point p;
    std::list<Edge>::iterator e;
};
struct Vertex {
    Point coord;
    std::list<Edge>::iterator e;
};
struct Edge {
    std::list<Site>::iterator site;
    std::list<Vertex>::iterator v;

    std::list<Edge>::iterator twin;
    std::list<Edge>::iterator next;
};
struct DCEL {
    std::list<Site> sites;
    std::list<Vertex> vertices;
    std::list<Edge> edges;
};

```

Сформований ПЗСР можна передавати в функцію для графічного представлення.

### 3.5. Алгоритм Форчуна: Берегова лінія

Ключовою особливістю реалізації берегової лінії в алгоритмі Форчуна в його класичному вигляді є той факт, що упродовж всього часу роботи алгоритму листи бінарного дерева пошуку відповідають дугам берегової лінії, а інші вузли дерева — точкам зустрічі. Тоді процедури додавання та видалення формулюються таким чином, щоб ця особливість дерева зберігалася, процедура пошуку дуги в дереві фактично зводиться до спуску від кореня до листа, на проміжних кроках порівнюючи шукане значення з координатами точок зустрічі.

Інтерфейс та функціонал стандартного контейнеру `map` не дозволяє так строго контролювати структуру дерева. Фактично користувач не знає, як саме зберігаються дані в самому дереві і як це дерево балансується. Отже, нам потрібно вгадати, як ми можемо адекватно виконувати дії над береговою лінією, якщо ми точно не знаємо її структури. Проте ми можемо змусити `map` зберігати дуги та місця зустрічі у “правильній” послідовності, щоб при прямому обході `map` ми отримували спочатку найлівішу дугу, потім найлівішу точку зустрічі, потім наступну за нею дугу і так далі і так далі.

Для цього нам достатньо вміти порівнювати точки зустрічі та дуги між собою за розташуванням по координаті  $x$ . Порівняти дві точки зустрічі між собою не є проблемою: ми знаходимо їхні  $x$ -координати за допомогою алгоритма з пункту 3.3 та порівнюємо їх. Щоб порівняти дугу та точку зустрічі, ми можемо використати інформацію про крайові точки цієї дуги, тобто про точки зустрічі, де дуга починається та закінчується. Тоді ми порівнюємо ці дві точки зустрічі з координатою початкової точки зустрічі. При цьому, звісно, не кожна дуга має обидві точки зустрічі на кінцях. Крайня зліва та крайня справа дуги на береговій лінії мають лише праву та ліву точки зустрічі відповідно, тому їх варто обробляти окремо. Ну і,

звісно, на самому початку алгоритму після обробки першої події місця наше дерево буде складатися з однієї дуги без точок зустрічі взагалі. Про це також варто не забувати. Порівнювати дуги між собою також можна через порівняння їхніх кінців.

Перейти від дуги до сусідніх точок зустрічі в `std::map` можна за допомогою `std::map::iterator`. Оскільки ми забезпечуємо правильну послідовність зберігання елементів берегової лінії, то інкрементація та декрементація ітератора, який посилається на вузол з дугою, буде давати ітератори з посиланням на праву та ліву точки зустрічі відповідно. Проте у функціях `find` та під час вставки і видалення порівнюються саме ключі елементів мапи, а це означає, що використовувати ітератор мапи для пошуку сусідніх вузлів не вдасться. Проте цю проблему можна обійти, якщо в ключі для дуги просто зберігати ітератори на сусідні точки зустрічі.

Тепер залишилося лише описати процедуру пошуку дуги для певного значення координати  $x$  нової події місця. Це означає, що нам треба порівняти дійсне число з дугою або з точкою зустрічі. Це питання вирішується аналогічно: координата  $x$  події місця порівнюється або з  $x$ -координатою точки зустрічі, або з краями дуги. При цьому точка  $x$  вважається меншою за дугу, якщо вона менша за її лівий край і більшою за дугу, якщо вона більше за її правий край. Процедуру пошуку в `find` вважає, що всі ключі формують лінійно упорядковану множину, тому, якщо ми передамо їй в якості ключа значення  $x$ , яке знаходиться під дугою  $c$ , то згідно з нашою процедурою порівняння  $x$  буде не меншим і не більшим за  $c$ . Отже, в такому випадку `find` поверне ітератор дуги  $c$ .

Підсумуємо наші міркування: структура берегової лінії, реалізована за допомогою `std::map`, буде обробляти 3 типи об'єктів: дуги, точки зустрічі та окремі координати  $x$ . Координати дуги задаються двома своїми сусідніми точками зустрічі, координата точки зустрічі задається упорядкованою парою точок  $\langle p_i, p_j \rangle$ , які породжують дві сусідні параболи. Координата

$x$  задається координатою  $x$ . При цьому нам ще потрібно враховувати поточну висоту замітаючої прямої, оскільки без неї розрахувати координати точки зустрічі неможливо. Оскільки нам потрібно утворити єдиний ключ для всіх об'єктів і ми при цьому знаємо, що інших за своєю природою та структурою об'єктів у нас під час роботи з `std::map` не буде, то всі ці параметри ми можемо в єдину структуру-ключ, використавши `std::variant` для об'єднання різних типів ключів.

Оскільки таких типів три, то спочатку опишемо структури для збереження даних в кожному з цих типів.

```
struct TKeyLeaf {
    std::map<TKey,TValue>::iterator end_left;
    std::map<TKey,TValue>::iterator end_right;
};
struct TKeyNode {
    std::list<Site>::iterator site_left;
    std::list<Site>::iterator site_right;
};
struct TKeyX {
    double x;
};
```

Тоді структура загального ключа буде виглядати наступним чином.

```
struct TKey {
    std::variant<TKeyLeaf, TKeyNode, TKeyX> key;

    //for all types of key
    double line;
};
```

Для цього ключа нам треба перевантажити оператор `less`, щоб цю структуру можна було використовувати як ключ в `map`. Сигнатура оператора буде наступною

```
struct TKey {
    ...
```

```

    bool operator<(const TKey& other) const;
};

```

Для реалізації цього методу можна використати метод `std::visit`, який в певній мірі дозволяє уникнути перебору варіантів. При цьому якщо два ключа мають різні значення `line`, то з них обирається найменше та саме воно використовується для подальших обчислень. Така логіка вибору спільного `line` не протирічить логіці порівняння вузлів, оскільки, якщо порівнюється координата  $x$  події місця з чимось іншим, то `line` має бути  $x$ -координатою цієї події місця, а це найнижче значення замітаючої прямої на момент досягнення нею даної події місця. Якщо ж порівнюються дуги або точки зустрічі, то їхнє взаємне розташування на береговій лінії не залежить від  $y$ -координати замітаючої прямої, бо воно не змінюється з часом. Тоді реалізація методу порівняння буде виглядати наступним чином.

```

    bool operator<(const TKey& other) const
    {
        double y = std::min(line, other.line);
        return std::visit([y](auto&& a, auto&& b) -> bool
            { return compare(a, b, y); }, key, other.key );
    };

```

Для коректної роботи `std::visit` необхідно буде додати реалізації логіки порівняння `compare(const T1&, const T2&, double)` для усіх пар типів з набору `TKeyLeaf`, `TKeyNode` та `TKeyX`.

Проте у цього ключа є один суттєвий недолік. Він добре працює, коли дуги мають додатню довжину, проте іноді буває так, що в одній точці площини одночасно зникають дві або більше дуги, а на додачу там ще може виникати нова дуга (рис. 3.7). Обмеження на вхідні дані не гарантує, що такого не станеться. Проте з точки зору алгоритму Форчуна в цьому немає проблеми. В такому випадку неважливо, яка саме дуга знає першою, або що спочатку утворюється нова дуга, проте видалення та додавання вузлів до дерева спричиняє порівняння дуг нульової або майже нульової довжини

(що з точки зору роботи з числами типу **double** одне і те саме). В такому випадку орієнтуватися на результати порівняння крайових точок таких дуг для визначення їхнього взаємного розташування вже не можна.

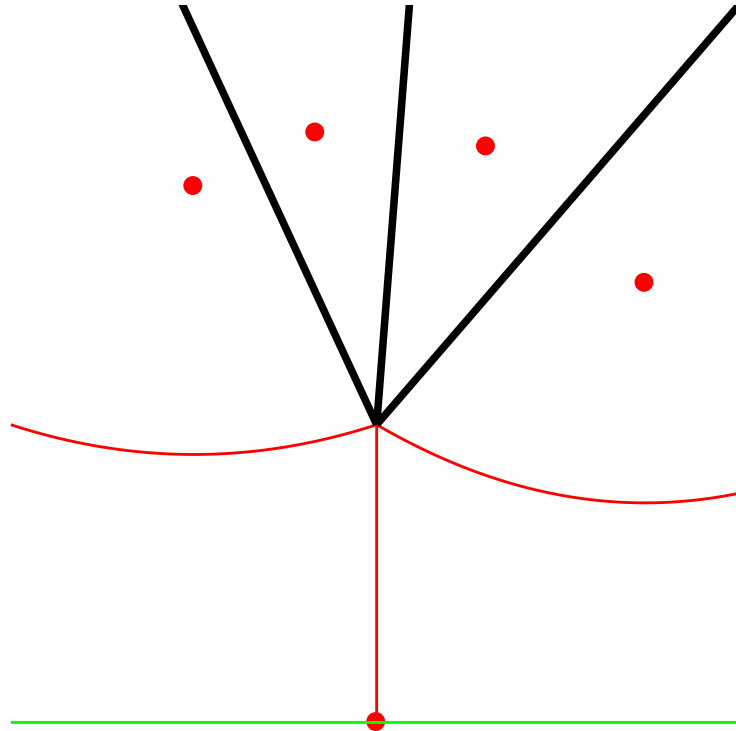


Рис. 3.7: Ситуація “одночасних” подій кола та події місяця.

Для того, щоб вирішити цю проблему, ми додамо фіктивний ключ — дійсне число, не прив’язане до жодних координат, яке підпорядковується лише одному правилу: якщо вузол *B* вирішено було вставити між вузлами *A* та *C*, то фіктивний ключ *B* повинен бути більшим за фіктивний ключ *A* та меншим за фіктивний ключ *C*. Тоді порівняння дуг та точок зустрічі між собою можна надалі проводити за цим фіктивним ключем. При цьому порівняння дуг та точок зустрічі з *x*-координатою події місяця буде відбуватися за старою логікою, а це означає, що нам лише треба додати відповідні поля до структур, що описують листи та внутрішні вузли дерева.

```
struct TKeyLeaf {
    std::map<TKey, TValue>::iterator end_left;
    std::map<TKey, TValue>::iterator end_right;
```

```

    double key;
};
struct TKeyNode {
    std::list<Site>::iterator site_left;
    std::list<Site>::iterator site_right;

    double key;
};

```

Структура ключа TKey при цьому не зміниться. Загальна логіка процедури порівняння двох ключів (operator<) при цьому не зміниться, але доведеться внести зміни в логіку порівняння двох ключів, кожен з яких має тип, відмінний від TKeyX. При цьому очевидно, що відповідні процедури суттєво полегшаться.

Процедура пошуку в такому випадку повністю коректна. Процедура видалення вузла відбувається за його ітератором без зайвих проблем. Для того, щоб коректно додати піддерево до мапи, необхідно спочатку визначити фіктивні ключі для кожного вузла піддерева, так, щоб вони узгоджувалися з порядком берегової лінії, та додати спочатку точки зустрічі, а потім дуги. Не в іншому порядку, бо інакше в ключі до дуг неможливо буде вставити коректні ітератори, що посилаються на сусідні точки зустрічі.

Також не варто забувати, що вузли дерева повинні містити додаткову інформацію, яка пов'язує цю структуру з чергою та ПЗСР. Оскільки ця інформація не впливає на процедури додавання, видалення вузлів та пошук, її можна перенести в значення Value. Для різних типів вузлів знову можна поєднати різні значення Value в одну структуру TValue.

```

struct TValueLeaf {
    std::list<Site>::iterator p;
    ... q; //посилання на можливу подію кола в черзі
};
struct TValueNode {

```

```

    std::list<Edge>::iterator e;
};
struct TValue {
    std::variant<TValueLeaf, TValueNode> value;
};

```

Тепер у нас лише 2 варіанти типів даних, які можуть зберігатися, бо  $x$ -координата події місця, хоч і фігурує в пошуку, ніколи не зберігається в самому дереві. У вузлі, що відповідає точці зустрічі, зберігаються посилання на ребра ПЗСР, у вузлах-дугах — посилання на можливу подію в черзі.

### 3.6. Алгоритм Форчуна: Черга подій

В бібліотеці STL є контейнер `std::priority_queue`, який дозволяє працювати з чергою з пріоритетами. Проте в алгоритмі Форчуна елементи черги можуть видалятися з неї в довільний момент часу з довільного місця. `std::priority_queue` не має інтерфейсу для подібних дій, тому замість цього контейнеру треба використати інший.

На щастя, для сортування подій за пріоритетом у нас завжди є `std::map`. Використовуючи пріоритет як ключ, ми можемо додавати нові події, видаляти старі через посилання на відповідний ітератор та знаходити поточну подію за допомогою `std::begin()`. І все це за логарифмічний час. Тобто контейнер `std::map` виконує всі необхідні процедури з коректною асимптотикою.

Проте він не зовсім підходить для нашої задачі, оскільки у нас можуть траплятися події з однаковим пріоритетом. Це можуть бути або “одночасні” події кола, або подія місця та подія кола (рис. 3.7). Дві події місця не можуть мати один і той самий пріоритет, оскільки всі початкові точки мають різні  $y$ -координати.

З точки зору алгоритму Форчуна абсолютно неважливо, яка з подій з

однаковим пріоритетом буде оброблятися першою, а це означає, що нам треба лише мати можливість зберігати їх в будь-якому порядку. Для цього нам цілком підходить контейнер `std::multimap`. Він зберігає асимптотику всіх ключових процедур та дозволяє зберігати події з однаковим пріоритетом. Тоді черга  $Q$  буде визначатися як

```
struct QValueSite {
    std::list<Site>::iterator site;
};
struct QValueCircle {
    std::map<TKey,TValue>::iterator tree_node;
};
struct QValue {
    std::variant<QValueSite, QValueCircle> value;
};
std::multimap<double,QValue> Q;
```

### 3.7. Рекомендації до реалізації алгоритма Форчуна без використання бібліотеки STL

У випадку, якщо ми хочемо запрограмувати структури даних самостійно, ми можемо реалізувати чергу з пріоритетами так, як це робиться стандартно: за допомогою купи. При цьому треба явно прописати операцію видалення елемента з черги.

Для представлення берегової лінії ми можемо не вигадувати складні структури ключів та процедури порівняння, а реалізувати бінарне дерево пошуку таким чином, щоб листи завжди відповідали дугам, а внутрішні вузли — точкам зустрічі. Процедура пошуку листа тоді може бути реалізована як окремий метод класу нашого дерева, де ми напряму порівнюємо  $x$ -координату тільки з внутрішніми вузлами, поки не дістанемося листа.

У якості реалізації бінарного дерева пошуку можна використовувати AVL-дерево. Воно відносно легко реалізується та принцип його балансува-

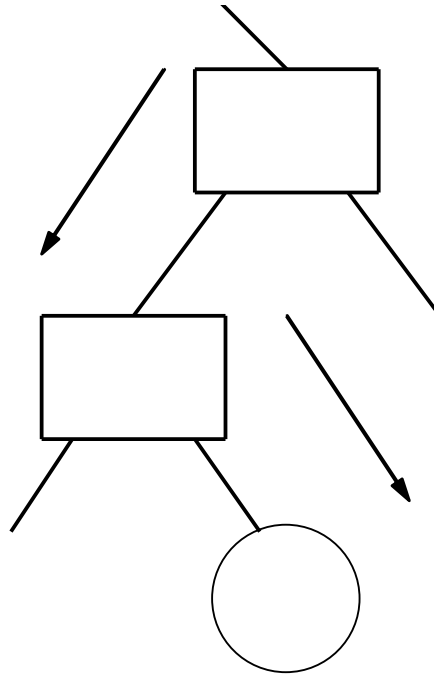


Рис. 3.8: Пошук листа в дереві.

ння інтуїтивно зрозумілий.

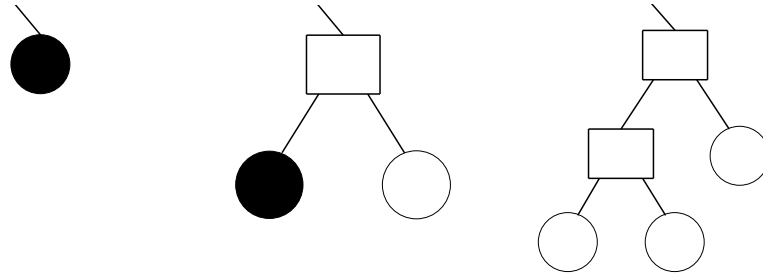
Залишаються лише питання видалення та додавання вузлів. Видалення листа та двох сусідніх внутрішніх вузлів можна реалізувати окремим методом, який працюватиме за схемою з рис. 3.5 з подальшим балансуванням.

Додавання піддерева реалізується у два кроки з почерговим додаванням піддерев висотою 1. Після кожного додавання відбувається балансування (рис. 3.9).

Тоді з такими допоміжними методами пошуку, додавання та видалення AVL-дерево буде завжди зберігати в своїх листах дані про дуги, а у внутрішніх вузлах — дані про точки зустрічі.

### 3.8. Алгоритм Форчуна: приклад

В цьому пункті ми застосуємо алгоритм Форчуна для побудови діаграми Вороного для множини точок  $\{p_1 = (3, 3), p_2 = (-2, 2), p_3 = (3, -3), p_4 =$



(а) Лист, що має зні- (б) Один рівень під- (в) Все піддерево до-  
кнуті. дерева доданий. дане.

Рис. 3.9: Додавання піддерева висотою 2 шляхом почергового додавання дерев висотою 1. Після кожного додавання відбувається перебалансування.

$(2, -4)$ }. Схематично процес виконання алгоритму зображений на рис. 3.3, проте тут ми відслідкуємо покроково зміни в усіх структурах. Черга подій та ПЗСР будуть зображуватися на кожному кроці у вигляді таблиць, в той час як берегова лінія буде зображена рисунком. Для зручності ми не будемо проводити балансування двійкового дерева та будемо зображувати його згідно з принципами оригінального алгоритму (листи відповідають дугам, інші вузли — точкам зустрічі). Також будуть пропущені деякі “технічні” обчислення, як розв’язання квадратичних рівнянь. Проте важливі обчислення будуть присутні в прикладі.

Також в таблицях та на рисунках деякі компоненти структур даних будуть пронумеровані або проіменовані. Ці номери та імена є по суті замінами вказівникам. Тому варто враховувати, що всі імена або номери, які будуть присутні в цьому прикладі, повинні бути перетворені на вказівники або просто прибрані в програмі.

Перед початком роботи алгоритму ми можемо одразу сформувати список граней діаграми Вороного, використавши у якості координат точки  $p_i$ . Після цього ми додаємо всі точки як події місця в чергу подій  $Q$ . Колонка дані зараз для кожної події містить посилання на відповідну грань.

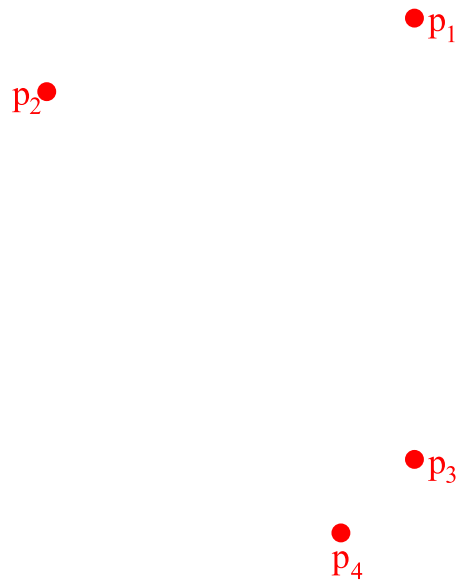


Рис. 3.10: Графічне зображення (до початку).

Таблиця 3.4

**Черга подій Q (до початку).**

Назва події	Пріоритет	Тип події	Дані
$p_1$	3	Місце	1
$p_2$	2	Місце	2
$p_3$	-3	Місце	3
$p_4$	-4	Місце	4

Таблиця 3.5

**ПЗСР: грані (до початку).**

Номер грані	Координати	Напівребро
1	(3, 3)	
2	(-2, 2)	
3	(3, -3)	
4	(2, -4)	

Таблиця 3.6

**ПЗСР: вершини (до початку).**

Номер вершини	Координати	Напівребро

Таблиця 3.7

**ПЗСР: напівребра (до початку).**

Номер напівребра	Вершина	Грань	Близнюк	Наступне

Подальші видалення подій з черги та процеси їхніх обробок ми для зручності будемо називати ітераціями. В таблицях та на рисунках будуть показані результати обробки події, які будуть супроводжуватися коментарями. Також варто зазначити, що рисунки будуть містити конфігурацію замітаючої прямої, берегової лінії та ПЗСР в моментах “між подіями”. Тобто коли вже замітаюча пряма пройшла певну відстань від останньої обробленої події. В такому випадку рисунки є найбільш інформативними.

**Ітерація 1.**

Перша подія місця обробляється тривіальним чином: до порожнього дерева додається одна дуга  $c_1$ , прив’язана до точки  $p_1$ .

Таблиця 3.8

**Черга подій Q (ітерація 1).**

Назва події	Пріоритет	Тип події	Дані
$p_2$	2	Місце	2
$p_3$	-3	Місце	3
$p_4$	-4	Місце	4

**Ітерація 2.**

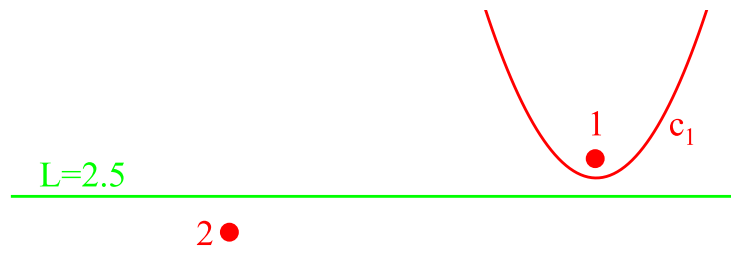


Рис. 3.11: Графічне зображення (ітерація 1).

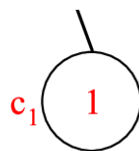


Рис. 3.12: Берегова лінія Т (ітерація 1).

Таблиця 3.9

**ПЗСР: грані (ітерація 1).**

Номер грані	Координати	Напівребро
1	(3, 3)	
2	(-2, 2)	
3	(3, -3)	
4	(2, -4)	

Таблиця 3.10

**ПЗСР: вершини (ітерація 1).**

Номер вершини	Координати	Напівребро

Таблиця 3.11

**ПЗСР: напівребра (ітерація 1).**

Номер напівребра	Вершина	Грань	Близнюк	Наступне

Перед обробкою другої події місця  $T$  складається з одного листа, тому цей лист замінюється на піддерево, що тепер представляє нову дугу  $s_3$ , яка розбиває параболу, породжену  $p_1$ , на дві дуги  $s_2$  та  $s_4$ . В листі не було вказівників на події кола, тому з черги нічого не видаляється.

В точках перетину утворюється пара напівребер-близнюків. Ці напівребра заносяться до ПЗСР. Ребра розділяють грані  $p_1$  та  $p_2$ , тому ми можемо одразу заповнити для цих напівребер поля граней, а для самих граней вказати поля ребер.

Оскільки у дуги  $s_2$  немає лівого сусіда, а у  $s_4$  — правого, то ці дуги не будуть зникати, тому перевіряти умову зникнення через визначник не потрібно.

Таблиця 3.12

**Черга подій Q (ітерація 2).**

Назва події	Пріоритет	Тип події	Дані
$p_3$	-3	Місце	3
$p_4$	-4	Місце	4

**Ітерація 3.**

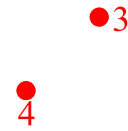
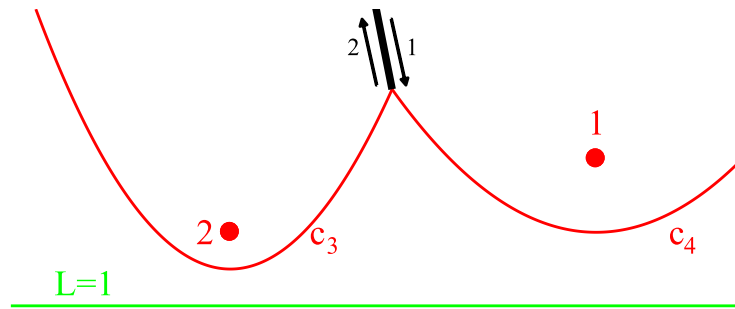


Рис. 3.13: Графічне зображення (ітерація 2).

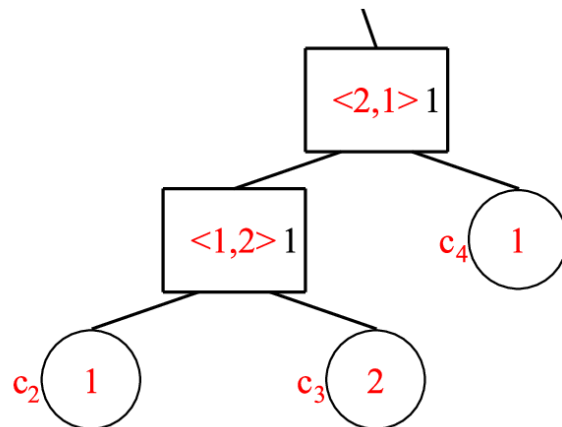


Рис. 3.14: Берегова лінія T (ітерація 2).

Таблиця 3.13

**ПЗСР: грані (ітерація 2).**

Номер грані	Координати	Напівребро
1	(3, 3)	1
2	(-2, 2)	2
3	(3, -3)	
4	(2, -4)	

Таблиця 3.14

**ПЗСР: вершини (ітерація 2).**

Номер вершини	Координати	Напівребро

Таблиця 3.15

**ПЗСР: напівребра (ітерація 2).**

Номер напівребра	Вершина	Грань	Близнюк	Наступне
1		1	2	
2		2	1	

Нова подія місця має  $x$ -координату 3. Порівнюючи її з координатами точок зустрічі, ми спускаємося до найправішого листа в  $T$ , який визначає дугу  $c_4$ . Точка  $p_3$  лежить прямо під цією дугою, коли замітаюча пряма знаходиться на висоті  $-3$ . Лист  $c_4$  замінюється на піддерево, що тепер представляє нову дугу  $c_6$ , яка розбиває параболу, породжену  $p_1$ , на дві дуги  $c_5$  та  $c_7$ . В листі не було вказівників на події кола, тому з черги нічого не видаляється.

В точках перетину утворюється пара напівребер-близнюків. Ці напівребра заносяться до ПЗСР. Ребра розділяють грані  $p_1$  та  $p_3$ , тому ми можемо одразу заповнити для цих напівберер поля граней, а для грані 3 вказати поле ребра.

Оскільки у дуги  $c_7$  немає правого сусіда, то ця дуга не буде зникати, тому перевіряти умову зникнення через визначник не потрібно.

Проте у дуги  $c_5$  є лівий та правий сусіди. Це дуги  $c_3$  та  $c_6$  відповідно. Три послідовні дуги  $c_3$ ,  $c_5$ ,  $c_6$  породжені точками  $p_2$ ,  $p_1$  та  $p_3$ , тому для них перевірка умови зникнення середньої дуги буде виглядати наступним чином:

$$\begin{vmatrix} p_1.x - p_2.x & p_3.x - p_2.x \\ p_1.y - p_2.y & p_3.y - p_2.y \end{vmatrix} = \begin{vmatrix} 3 - (-2) & 3 - (-2) \\ 3 - 2 & -3 - 2 \end{vmatrix} = \begin{vmatrix} 5 & 5 \\ 1 & -5 \end{vmatrix} = -30 < 0.$$

Це означає, що дуга  $c_5$  буде зникати. Для того, щоб визначити пріоритет нової події кола, ми повинні знайти  $y$ -координату найнижчої точки кола, що проходить через  $p_1$ ,  $p_2$  та  $p_3$ . Рівняння цього кола наступне.

$$(x - 1)^2 + y^2 = 13. \quad (3.2)$$

Найнижча точка кола має координати  $(1, -\sqrt{13}) \approx (1, -3.60555)$ , тому ми додаємо до черги  $Q$  подію кола з пріоритетом  $-3.6055$  та додаємо до цього елемента черги посилання на дугу  $c_5$ . Дана подія одразу стає першою

в черзі в силу свого пріоритету. До дуги  $c_5$  додається посилання на цю щойно створену подію.

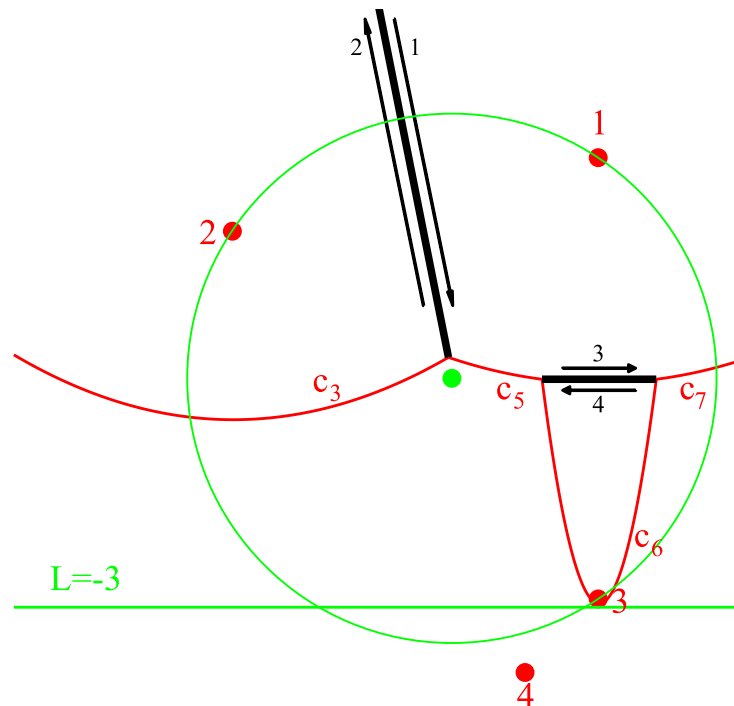


Рис. 3.15: Графічне зображення (ітерація 3).

Таблиця 3.16

### Черга подій Q (ітерація 3).

Назва події	Пріоритет	Тип події	Дані
$q_1$	-3.60555	Коло	$c_5$
$p_4$	-4	Місце	4

### Ітерація 4.

На цій ітерації ми обробляємо подію кола.

Зникає дуга  $c_5$ , разом з нею з берегової лінії видаляються дві сусідні точки зустрічі. На їхнє місце додається новий вузол, який представляє точку зустрічі дуг  $c_3$  та  $c_6$ . Для цієї точки зустрічі створюються нові два напівребра-близнюка, які розділяють грані  $p_2$  та  $p_3$ .

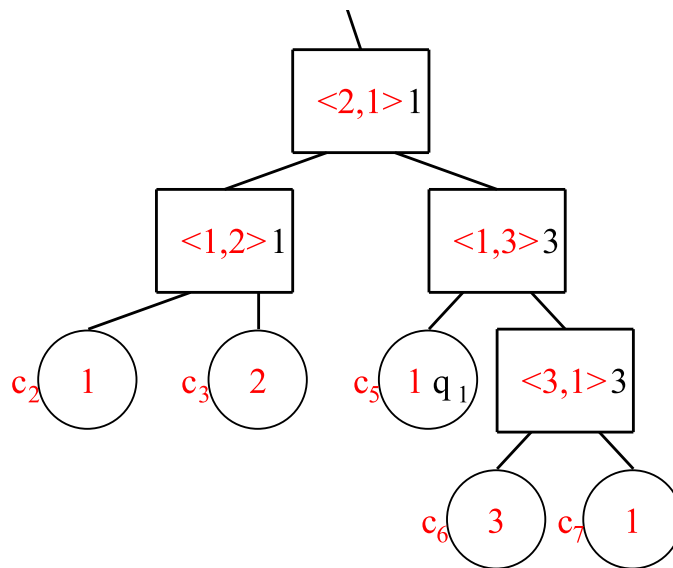


Рис. 3.16: Берегова лінія T (ітерація 3).

Таблиця 3.17

ПЗСР: грані (ітерація 3).

Номер грані	Координати	Напівребро
1	(3, 3)	1
2	(-2, 2)	2
3	(3, -3)	4
4	(2, -4)	

Таблиця 3.18

ПЗСР: вершини (ітерація 3).

Номер вершини	Координати	Напівребро

Таблиця 3.19

**ПЗСР: напівребра (ітерація 3).**

Номер напівребра	Вершина	Грань	Близнюк	Наступне
1		1	2	
2		2	1	
3		1	4	
4		3	3	

Центр кола, описаного рівнянням (3.2), знаходиться у точці  $(1, 0)$ . Це будуть координати нової вершини діаграми Вороного. З цієї вершини повинні виходити 3 напівребра: два з вже побудованих до цієї ітерації, посилення на які були у видалених точках зустрічі та одне з двох напівребер, які утворилися разом з новою точкою зустрічі. Для деяких з цих напівребер ми також можемо вказати наступні напівребра при обході відповідних граней.

Жодна з дуг  $c_3$  та  $c_6$  не мала посилень на події кола, тож видаляти з них нічого не потрібно. Проте для кожної з них треба перевірити умову зникнення дуги.

Перевірка для  $c_3$ :

Трійка послідовних дуг:  $c_2, c_3, c_6$ . Відповідні точки:  $p_1, p_2, p_3$ . Умова зникнення:

$$\begin{vmatrix} p_2 \cdot x - p_1 \cdot x & p_3 \cdot x - p_1 \cdot x \\ p_2 \cdot y - p_1 \cdot y & p_3 \cdot y - p_1 \cdot y \end{vmatrix} = \begin{vmatrix} -2 - 3 & 3 - 3 \\ 2 - 3 & -3 - 3 \end{vmatrix} = \begin{vmatrix} -5 & 0 \\ -1 & -6 \end{vmatrix} = 30 > 0.$$

Умова не виконується, дуга  $c_3$  не зникає, додаткові дії не потрібні.

Перевірка для  $c_6$ :

Трійка послідовних дуг:  $c_3, c_6, c_7$ . Відповідні точки:  $p_2, p_3, p_1$ . Умова зникнення:

$$\begin{vmatrix} p_3.x - p_2.x & p_1.x - p_2.x \\ p_3.y - p_2.y & p_1.y - p_2.y \end{vmatrix} = \begin{vmatrix} 3 - (-2) & 3 - (-2) \\ -3 - 2 & 3 - 2 \end{vmatrix} = \begin{vmatrix} 5 & 5 \\ -5 & 1 \end{vmatrix} = 30 > 0.$$

Умова не виконується, дуга  $c_6$  не зникає, додаткові дії не потрібні.

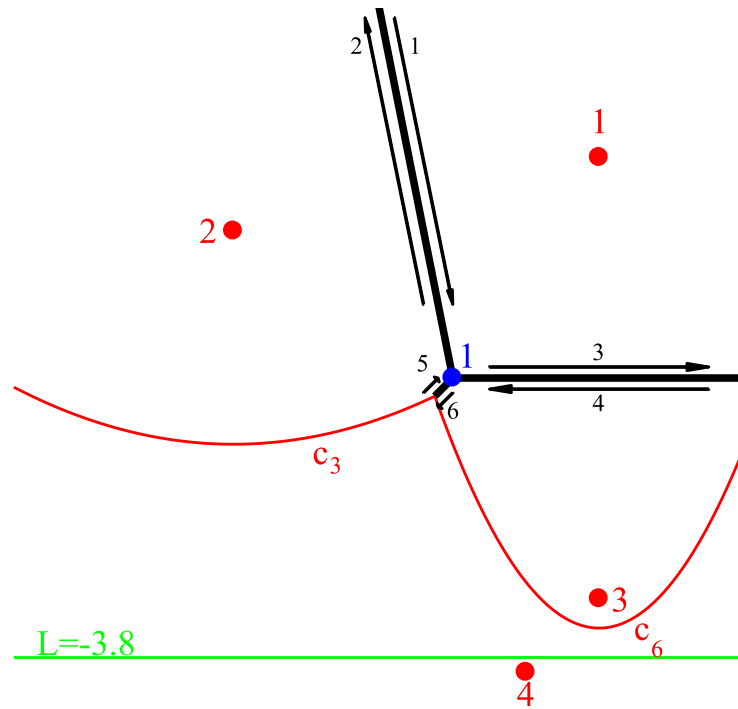


Рис. 3.17: Графічне зображення (ітерація 4).

Таблиця 3.20

### Черга подій Q (ітерація 4).

Назва події	Пріоритет	Тип події	Дані
$p_4$	-4	Місце	4

### Ітерація 5.

Остання подія місця має  $x$ -координату 2. Порівнюючи її з координатами точок зустрічі, ми спускаємося до листа в T, який визначає дугу  $c_6$ . Точка  $p_4$  лежить прямо під цією дугою, коли замітаюча пряма знаходиться на

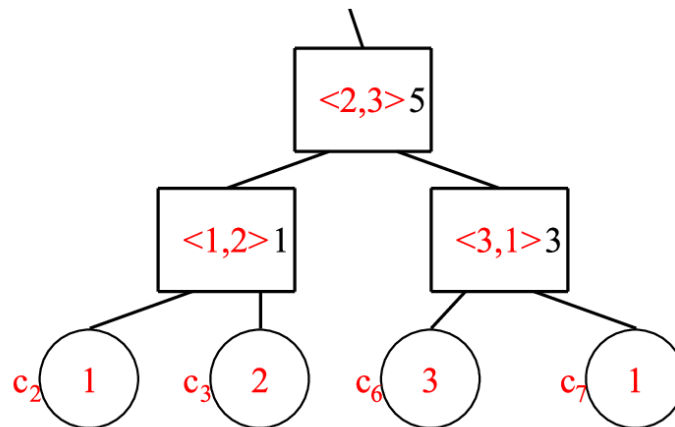


Рис. 3.18: Берегова лінія T (ітерація 4).

Таблиця 3.21

ПЗСР: грані (ітерація 4).

Номер грані	Координати	Напівребро
1	(3, 3)	1
2	(-2, 2)	2
3	(3, -3)	4
4	(2, -4)	

Таблиця 3.22

ПЗСР: вершини (ітерація 4).

Номер вершини	Координати	Напівребро
1	(1, 0)	2

Таблиця 3.23

## ПЗСР: напівребра (ітерація 4).

Номер напівребра	Вершина	Грань	Близнюк	Наступне
1		1	2	3
2	1	2	1	
3	1	1	4	
4		3	3	6
5		2	6	2
6	1	3	5	

висоті  $-4$ . Лист  $c_6$  замінюється на піддерево, що тепер представляє нову дугу  $c_9$ , яка розбиває параболу, породжену  $p_3$ , на дві дуги  $c_8$  та  $c_{10}$ . В листі не було вказівників на події кола, тому з черги нічого не видаляється.

В точках перетину утворюється пара напівребер-близнюків. Ці напівребра заносяться до ПЗСР. Ребра розділяють грані  $p_3$  та  $p_4$ , тому ми можемо одразу заповнити для цих напівберер поля граней, а для грані 4 вказати поле ребра.

У дуги  $c_8$  є лівий та правий сусіди. Це дуги  $c_3$  та  $c_9$  відповідно. Три послідовні дуги  $c_3$ ,  $c_8$ ,  $c_9$  породжені точками  $p_2$ ,  $p_3$  та  $p_4$ , тому для них перевірка умови зникнення середньої дуги буде виглядати наступним чином:

$$\begin{vmatrix} p_3.x - p_2.x & p_4.x - p_2.x \\ p_3.y - p_2.y & p_4.y - p_2.y \end{vmatrix} = \begin{vmatrix} 3 - (-2) & 2 - (-2) \\ -3 - 2 & -4 - 2 \end{vmatrix} = \begin{vmatrix} 5 & 4 \\ -5 & -6 \end{vmatrix} = -10 < 0.$$

Це означає, що дуга  $c_8$  буде зникати. Для того, щоб визначити пріоритет нової події кола, ми повинні знайти  $y$ -координату найнижчої точки кола, що проходить через  $p_2$ ,  $p_3$  та  $p_4$ . Рівняння цього кола наступне.

$$x^2 + (y + 1)^2 = 13. \quad (3.3)$$

Найнижча точка кола має координати  $(0, -1 - \sqrt{13}) \approx (0, -4.60555)$ , тому ми додаємо до черги Q подію кола з пріоритетом  $-4.6055$  та додаємо до цього елементу черги посилання на дугу  $c_8$ . До дуги  $c_8$  додається посилання на цю щойно створену подію.

У дуги  $c_{10}$  також є лівий та правий сусіди. Це дуги  $c_9$  та  $c_7$  відповідно. Три послідовні дуги  $c_9$ ,  $c_{10}$ ,  $c_7$  породжені точками  $p_4$ ,  $p_3$  та  $p_1$ , тому для них перевірка умови зникнення середньої дуги буде виглядати наступним чином:

$$\begin{vmatrix} p_3.x - p_4.x & p_1.x - p_4.x \\ p_3.y - p_4.y & p_1.y - p_4.y \end{vmatrix} = \begin{vmatrix} 3 - 2 & 3 - 2 \\ -3 - (-4) & 3 - (-4) \end{vmatrix} = \begin{vmatrix} 1 & 1 \\ 1 & 7 \end{vmatrix} = 6 > 0.$$

Це означає, що дуга  $c_{10}$  не буде зникати.

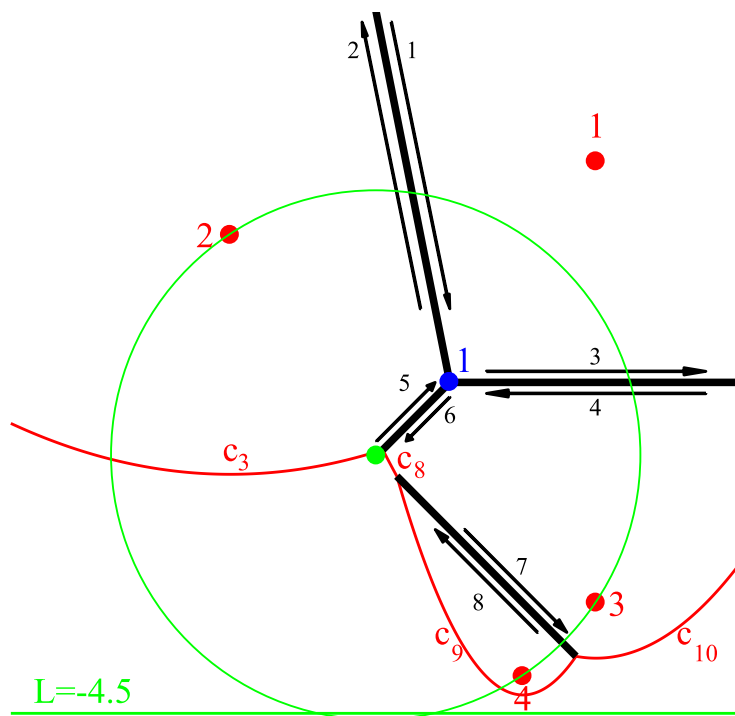


Рис. 3.19: Графічне зображення (ітерація 5).

**Ітерація 6.**

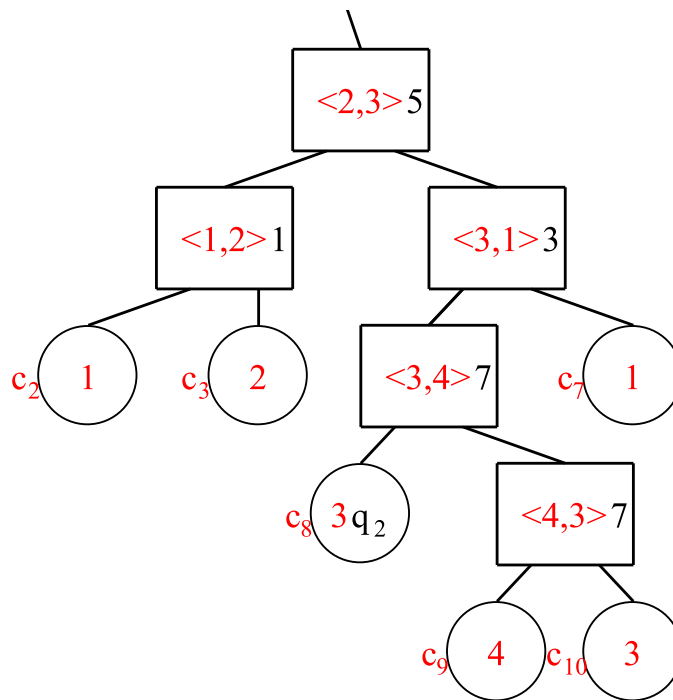


Рис. 3.20: Берегова лінія T (ітерація 5).

Таблиця 3.24

**Черга подій Q (ітерація 5).**

Назва події	Пріоритет	Тип події	Дані
$q_2$	-4.60555	Коло	$c_8$

Таблиця 3.25

**ПЗСР: грані (ітерація 5).**

Номер грані	Координати	Напівребро
1	(3, 3)	1
2	(-2, 2)	2
3	(3, -3)	4
4	(2, -4)	8

Таблиця 3.26

ПЗСР: вершини (ітерація 5).

Номер вершини	Координати	Напівребро
1	(1, 0)	2

Таблиця 3.27

ПЗСР: напівребра (ітерація 5).

Номер напівребра	Вершина	Грань	Близнюк	Наступне
1		1	2	3
2	1	2	1	
3	1	1	4	
4		3	3	6
5		2	6	2
6	1	3	5	
7		3	8	
8		4	7	

На цій ітерації ми знову обробляємо подію кола.

Зникає дуга  $c_8$ , разом з нею з берегової лінії видаляються дві сусідні точки зустрічі. На їхнє місце додається новий вузол, який представляє точку зустрічі дуг  $c_3$  та  $c_9$ . Для цієї точки зустрічі створюються нові два напівребра-близнюка, які розділяють грані  $p_2$  та  $p_4$ .

Центр кола, описаного рівнянням (3.3), знаходиться у точці  $(0, -1)$ . Це будуть координати нової вершини діаграми Вороного. З цієї вершини повинні виходити 3 напівребра: два з вже побудованих до цієї ітерації, посилення на які були у видалених точках зустрічі та одне з двох напівребер, які утворилися разом з новою точкою зустрічі. Для деяких з цих напівребер ми також можемо вказати наступні напівребра при обході відповідних граней.

Жодна з дуг  $c_3$  та  $c_9$  не мала посилень на події кола, тож видаляти з них нічого не потрібно. Проте для кожної з них треба перевірити умову зникнення дуги.

Перевірка для  $c_3$ :

Трійка послідовних дуг:  $c_2, c_3, c_9$ . Відповідні точки:  $p_1, p_2, p_4$ . Умова зникнення:

$$\begin{vmatrix} p_2 \cdot x - p_1 \cdot x & p_4 \cdot x - p_1 \cdot x \\ p_2 \cdot y - p_1 \cdot y & p_4 \cdot y - p_1 \cdot y \end{vmatrix} = \begin{vmatrix} -2 - 3 & 2 - 3 \\ 2 - 3 & -4 - 3 \end{vmatrix} = \begin{vmatrix} -5 & -1 \\ -1 & -7 \end{vmatrix} = 34 > 0.$$

Умова не виконується, дуга  $c_3$  не зникає, додаткові дії не потрібні.

Перевірка для  $c_9$ :

Трійка послідовних дуг:  $c_3, c_9, c_{10}$ . Відповідні точки:  $p_2, p_4, p_1$ . Умова зникнення:

$$\begin{vmatrix} p_4 \cdot x - p_2 \cdot x & p_1 \cdot x - p_2 \cdot x \\ p_4 \cdot y - p_2 \cdot y & p_1 \cdot y - p_2 \cdot y \end{vmatrix} = \begin{vmatrix} 2 - (-2) & 3 - (-2) \\ -4 - 2 & 3 - 2 \end{vmatrix} = \begin{vmatrix} 4 & 5 \\ -6 & 1 \end{vmatrix} = 34 > 0.$$

Умова не виконується, дуга  $c_9$  не зникає, додаткові дії не потрібні.

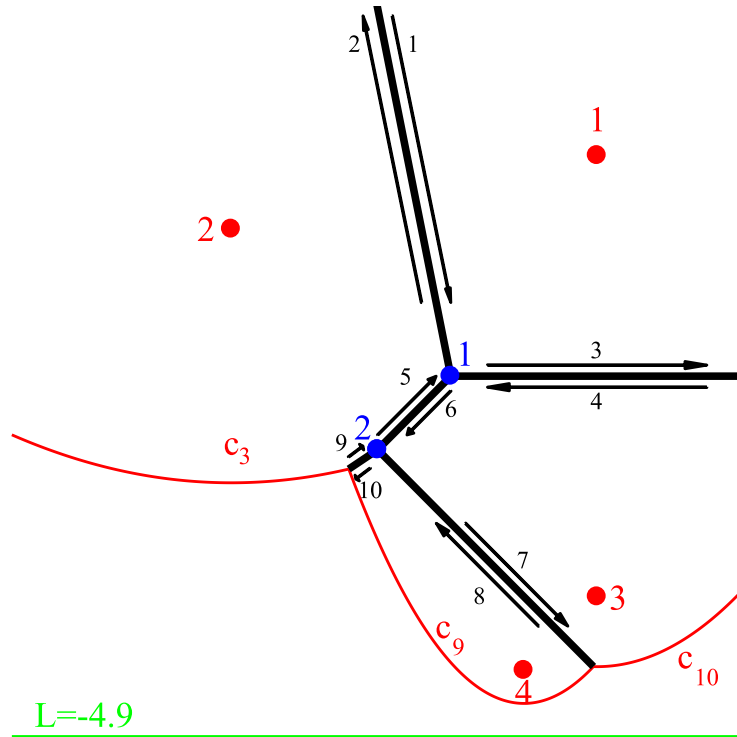


Рис. 3.21: Графічне зображення (ітерація 6).

Таблиця 3.28

**Черга подій Q (ітерація 6).**

Назва події	Пріоритет	Тип події	Дані

**Створення охоплюючого прямокутника.**

Черга подій порожня, а це означає, що нам лишилося тільки зробити крок 4 алгоритму **Форчуна**, тобто створити охоплючий прямокутник та доповнити всі списки ПЗСР фіктивними вершинами, напівребрами та гранню.

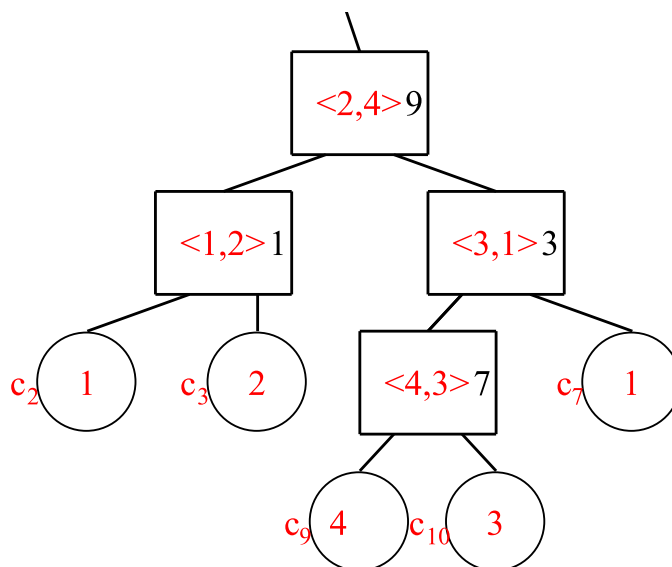


Рис. 3.22: Берегова лінія T (ітерація 6).

Таблиця 3.29

**ПЗСР: грані (ітерація 6).**

Номер грані	Координати	Напівребро
1	(3, 3)	1
2	(-2, 2)	2
3	(3, -3)	4
4	(2, -4)	8

Таблиця 3.30

**ПЗСР: вершини (ітерація 6).**

Номер вершини	Координати	Напівребро
1	(1, 0)	2
2	(0, -1)	5

Таблиця 3.31

**ПЗСР: напівребра (ітерація 6).**

Номер напівребра	Вершина	Грань	Близнюк	Наступне
1		1	2	3
2	1	2	1	
3	1	1	4	
4		3	3	6
5	2	5	6	2
6	1	3	5	7
7	2	3	8	
8		4	7	10
9		2	10	5
10	2	4	9	

Для нашого прикладу нехай у якості охоплюючого прямокутника буде квадрат  $\max(|x|, |y|) = 5$ . Тоді нам треба лише пройтися по незавершеним напівребрам побудованої діаграми Вороного та визначити точки їхніх перетинів з цим квадратом. Ці точки перетину будуть новими фіктивними вершинами. Будь-яка точка поза квадратом буде фіктивною гранню. Напівребра визначаються згідно з логікою ПЗСР.

Читачу пропонується самостійно позначити на рис. 3.23 фіктивні напівребра згідно з даними з табл. 3.34.

### **3.9. Побудова тріангуляції Делоне за допомогою діаграми Вороного**

Якщо ми маємо ПЗСР  $D_1$ , який описує діаграму Вороного, то для отримання ПЗСР  $D_2$  для тріангуляції Делоне того самого набору точок, достатньо здійснити наступні дії:

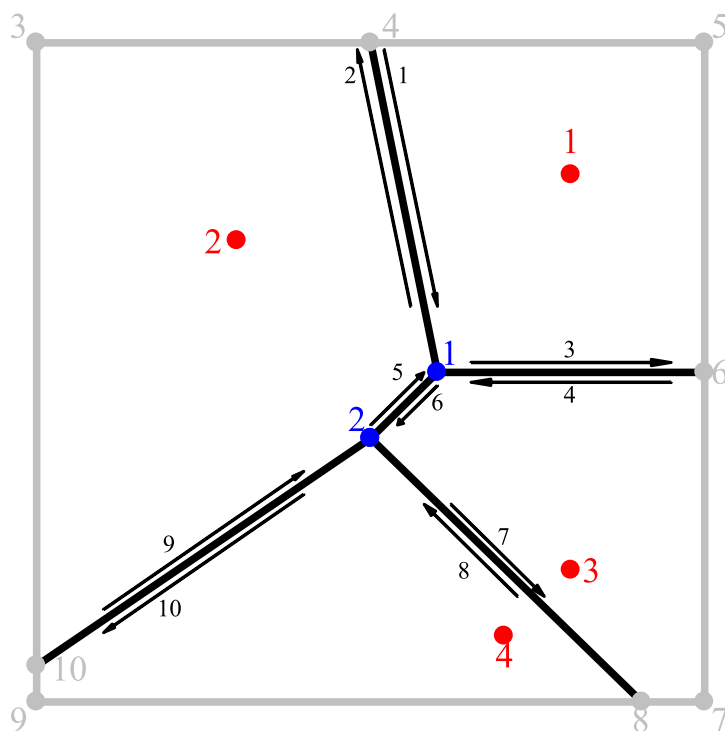


Рис. 3.23: Графічне зображення (кінець).

Таблиця 3.32

ПЗСР: грані (кінець).

Номер грані	Координати	Напівребро
1	(3, 3)	1
2	(-2, 2)	2
3	(3, -3)	4
4	(2, -4)	8
5	(6, 6)	11

Таблиця 3.33

**ПЗСР: вершини (кінець).**

Номер вершини	Координати	Напівребро
1	(1, 0)	2
2	(0, -1)	5
3	(-5, 5)	12
4	(0, 5)	1
5	(5, 5)	16
6	(5, 0)	4
7	(5, -5)	21
8	(4.06, -5)	8
9	(-5, -5)	25
10	(-5, -4.45)	9

1. Визначити координати граней  $D_2$  за допомогою координат вершин  $D_1$ .
2. Визначити координати вершин  $D_2$  за допомогою координат граней  $D_1$ .
3. Замінити кожну пару напівребер, яка з'єднує дві вершини в  $D_1$ , на пару півребер, яка з'єднує нові вершини в  $D_2$ . Після цього треба правильно визначити всі вказівники в  $D_2$  згідно з правилами заповнення ПЗСР.

**3.10. Алгоритми побудови опуклої оболонки**

Алгоритми побудови опуклої оболонки є відносно нескладними в порівнянні з алгоритмом Форчуна. Опукла оболонка множини  $A$  — це найменша за включенням опукла множина  $C$ , для якої  $A \subseteq C$ . Якщо  $A$  є набором точок на площині, то опуклою оболонкою є багатокутник, вершинами якого

Таблиця 3.34

## ПЗСР: напівребра (кінець).

Номер напівребра	Вершина	Грань	Близнюк	Наступне
1	4	1	2	3
2	1	2	1	14
3	1	1	4	18
4	6	3	3	6
5	2	2	6	2
6	1	3	5	7
7	2	3	8	22
8	8	4	7	10
9	10	2	10	5
10	2	4	9	26
11	10	5	12	13
12	3	2	11	9
13	3	5	14	15
14	4	2	13	12
15	4	5	16	17
16	5	1	15	1
17	5	5	18	19
18	6	1	17	16
19	6	5	20	21
20	7	3	19	4
21	7	5	22	23
22	8	3	21	20
23	8	5	24	25
24	9	4	23	8
25	9	5	26	11
26	10	4	25	24

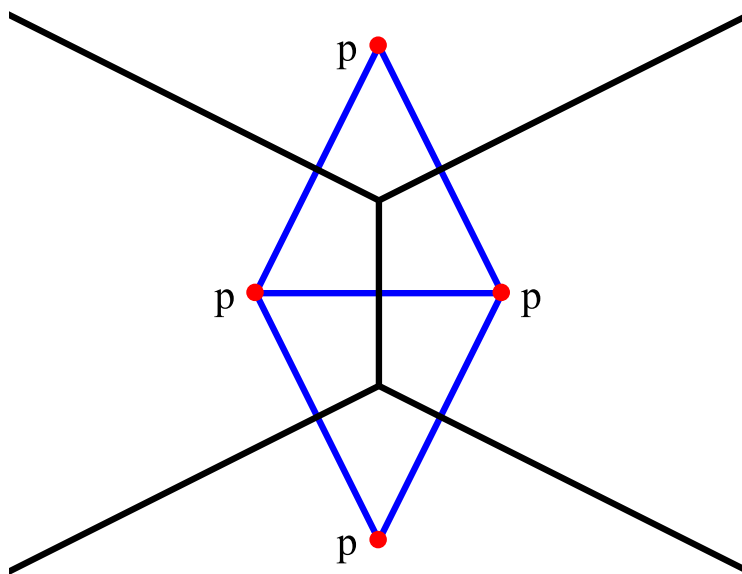
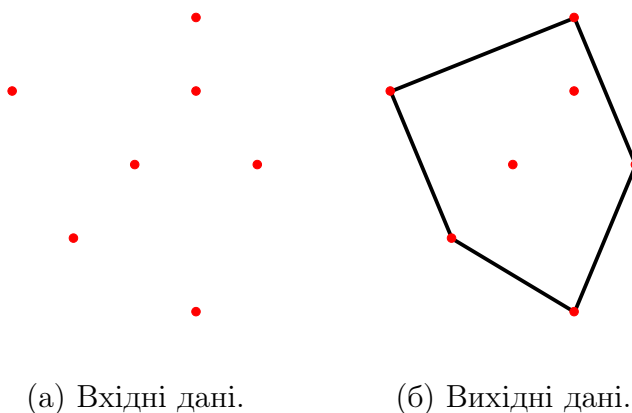


Рис. 3.24: Даграма Вороного (чорна) та триангуляція Делоне (блакитна).

є деякі точки з множини  $A$ .



(а) Вхідні дані.

(б) Вихідні дані.

Рис. 3.25: Опукла оболонка множини точок.

Більшість з методів побудови опуклої оболонки базується на понятті повороту при обході цієї самої оболонки. Так, якщо ми уявимо, що йдемо по ламаній, яка є межею опуклої оболонки деякої множини точок, причому йдемо в напрямку проти годинникової стрілки, то ми маємо завжди повертати ліворуч при такому обході. Якщо ж ми будемо йти за годинниковою стрілкою, то, навпаки, повертати будемо лише праворуч.

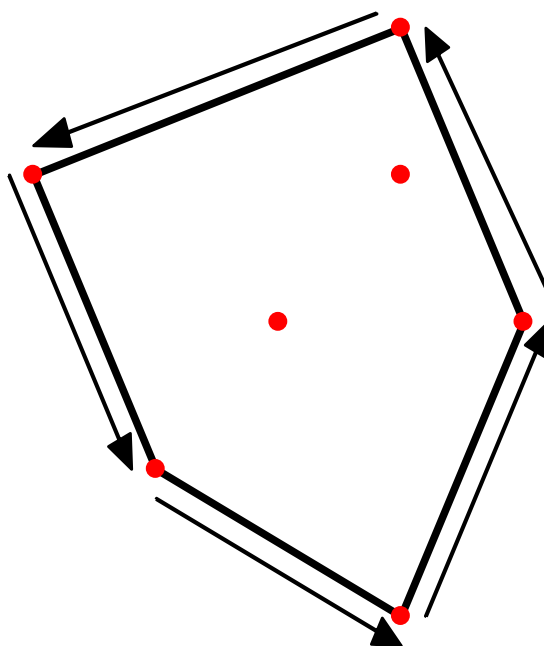


Рис. 3.26: Обхід опуклої оболонки проти годинникової стрілки з поворотами наліво.

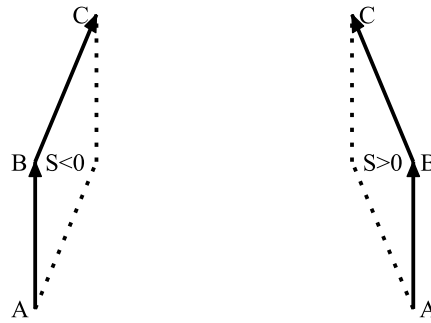
На цій інтуїтивно зрозумілій закономірності побудовні перші алгоритми Кейла-Кіркпатрика, Ендрю-Джарвіса та Грехема. Для того, щоб перевіряти, чи відбувається поворот “направо” чи “наліво”, зручно використовувати формулу для орієнтованої площі паралелограма. Припустимо, що ми йдемо по відрітку від точки  $A$  до точки  $B$  і в ній маємо повернути до  $C$  (рис. 3.27). Тоді для того, щоб визначити, в який бік відбувається поворот, треба порахувати визначник

$$\begin{vmatrix} B.x - A.x & C.x - A.x \\ B.y - A.y & C.y - A.y \end{vmatrix}$$

Якщо визначник більше 0, то поворот відбувається наліво, якщо менше 0 — направо.

#### **Алгоритм Кейла-Кіркпатрика.**

Для даного алгоритму точки  $p_1, p_2, \dots, p_N$  повинні мати цілі координати.



(а) Поворот направо.

(б) Поворот наліво.

Рис. 3.27: Зв'язок напрямку повороту та орієнтованої площі.

Для інших алгоритмів обмежень на координати точок немає.

1. Відсортувати точки  $p_1, p_2, \dots, p_N$  за  $y$ -координатою кишеньковим сортуванням, при цьому для кожного значення  $y$  визначити найлівішу та найправішу точки. В результаті буде отримано масив “лівих” та “правих” точок. Серед цих точок мають бути вершини опуклої оболонки.
2. Пройти по масиву лівих точок знизу вгору, проводячи відрізки між сусідніми точками. При цьому якщо поворот при побудові нового відрізка відбувається наліво, то дані два відрізка не є частиною опуклої оболонки, і їх треба видалити та замінити на “коротший” шлях (рис. 3.28). Якщо ж поворот відбувається направо, то ніяких проблем немає.
3. Провести аналогічну процедуру для масиву правих точок. Якщо йти по ним знизу вгору, то тепер поворот наліво є правильним поворотом.
4. Побудувати відрізки, що з'єднують найнижчі точки лівого та правого масивів.
5. Побудувати відрізки, що з'єднують найвищі точки лівого та правого масивів.

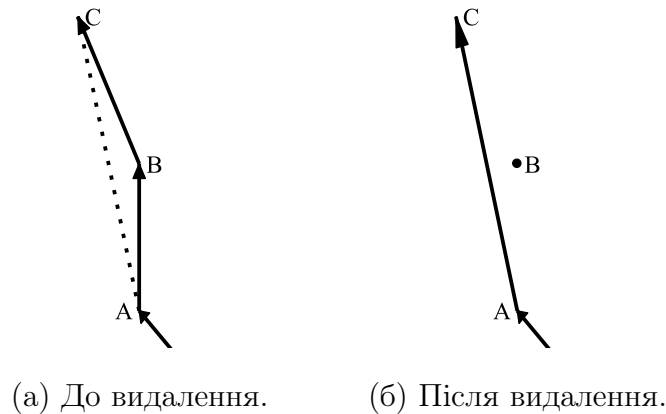


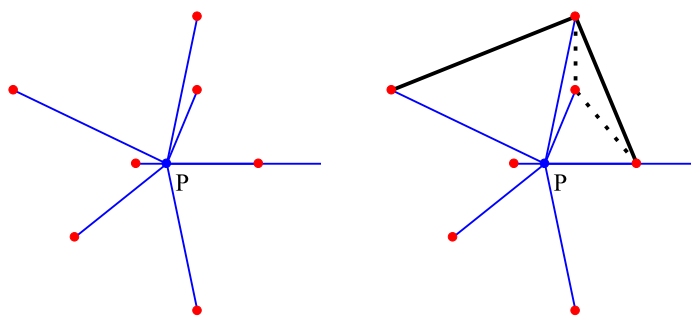
Рис. 3.28: Видалення повороту наліво під час побудови лівої частини опуклої оболонки методом Кейла-Кіркпатрика.

### Алгоритм Грехема.

1. Обрати полюс  $P$ : точку, яка знаходиться десь між точками  $p_i$ . Можна в якості полюсу обрати центр мас точок  $p_i$ , але не обов'язково саме його.
2. Відсортувати точки  $p_i$  за полярним кутом методом сортування з асимптотичною швидкістю  $O(n \ln n)$ .
3. Пройти по відсортованому набору точок проти годинникової стрілки, видаляючи при цьому повороти направо (аналогічно алгоритму Кейла-Кіркпатрика).
4. Досягнувши кінця відсортованого масиву, з'єднати дві крайні точки та пройти по опуклій оболонці проти годинникової стрілки знову для видалення поворотів направо. Ці повороти можуть залишитися через те, що ми з'єднали останню та першу точки без додаткових перевірок.

### Алгоритм Ендрю-Джарвіса.

1. Знайти найлівішу точку  $P$  серед набору  $p_i$ . Якщо таких декілька, обрати серед них найвищу.
2. Знайти точку  $Q$  серед набору  $p_i$ , таку що всі інші точки лежать по

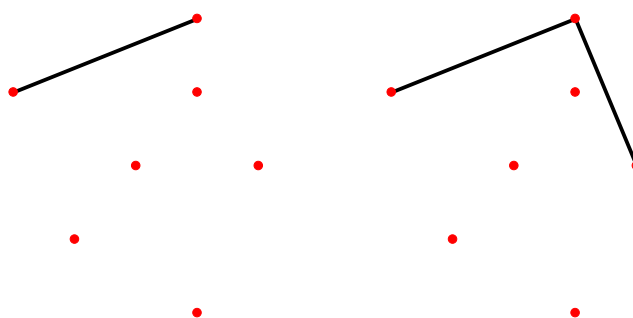


(а) Визначення полюсу та (б) Побудова проти годин-  
полярних кутів.                      никової стрілки.

Рис. 3.29: Побудова опуклої оболонки методом Грехема.

правий бік від прямої, що з'єднує  $P$  і  $Q$  (рис. 3.30).

3. Додати відрізок  $PQ$  до опуклої оболонки та встановити  $P = Q$ .
4. Повторювати кроки 2-3, поки не відбудеться повернення в точку, обрану на кроці 1.



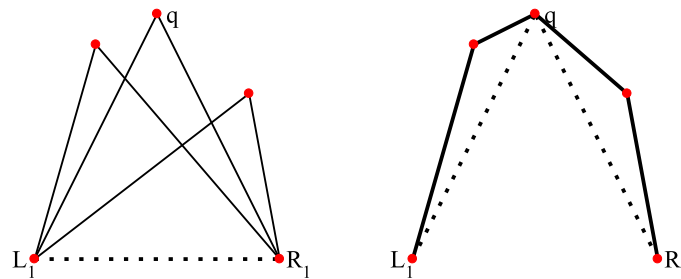
(а) Перше ребро.                      (б) Друге ребро.

Рис. 3.30: Перші дві складові опуклої оболонки, побудовані алгоритмом Ендрю-Джарвіса.

Останній, рекурсивний алгоритм для побудови опуклої оболонки, не використовує повороти. Натомість він будує опуклу оболонку, максимізуючи площу охопленої нею ділянки.

### Рекурсивний алгоритм.

1. Знайти найлівіші та найправіші точки з набору  $p_i$ . Серед найлівіших точок обрати найвищу та найнижчу, позначити їх  $L_1, L_2$ . Аналогічно визначити  $R_1, R_2$  для масиву правих точок.
2. З'єднати точки  $L_1$  та  $L_2$  між собою. Додати отриманий відрізок до опуклої оболонки. Аналогічно вчинити з  $R_1$  і  $R_2$ .
3. Визначити множину точок  $q_i$  серед множини  $p_i$ , які лежать вище прямої, що з'єднують  $L_1$  та  $R_1$ . Якщо таких точок немає, додати відрізок  $L_1R_1$  до опуклої оболонки.
4. Серед точок  $q_i$  обрати таку точку  $q$ , що площа трикутника  $L_1qR_1$  максимальна.
5. (Рекурсія) Повторити кроки 3-5, спочатку обираючи у якості  $L_1$  та  $R_1$  спочатку точки  $L_1, q$ , а потім —  $q, R_1$ . (ліва та права частини)
6. Провести процедуру, аналогічну крокам 3-5 для нижньої половини опуклої оболонки та точок  $L_2, R_2$ .

(а) Пошук  $q$ .

(б) Рекурсія.

Рис. 3.31: Побудова верхньої частини опуклої оболонки рекурсивним методом.

## ЗАВДАННЯ

1. Якщо генератор випадкових чисел мільярд разів підряд видасть значення 0, чи можна сказати, що він завжди видає 0? А якщо розглядати генератор псевдовипадкових чисел?
2. Узагальнити клас `Rand`, щоб з його допомогою можна було описувати недійснозначні випадкові величини (векторні). На що тоді мають змінитися поля `Dom_min` та `Dom_max`?
3. Реалізацією нормального розподілу може бути довільне дійсне число. Як тоді треба обрати `Dom_min` та `Dom_max` для коректного виведення гістограми?
4. Модифікувати алгоритми 7-8 лабораторної роботи №1, так щоб з їхньою допомогою можна було генерувати нормально розподілені випадкові величини із наперед заданим матсподіванням  $m$  та квадратичним відхиленням  $\sigma$ .
5. Реалізувати алгоритм шкільного ділення двох довгих цілих чисел. Яка асимптотична складність буде у цієї процедури, якщо ділене має довжину  $n$  цифр, а дільник —  $n/2$  цифр?
6. Реалізувати алгоритм шкільного множення в рекурсивній формі.
7. Довести асимптотичну оцінку для методу Карацуби, використовуючи теореми про асимптотичні оцінки рекурсивних алгоритмів.
8. Довести, що всі коефіцієнти  $F$  в методі Тоома-Кука є невід'ємними.
9. Реалізувати алгоритм швидкого множення Тоом-5, який полягає в модифікуванні алгоритму Тоома-Кука. В Тоом-5 на основі чисел-аргументів будуються поліноми 5 степені.
10. Довести, що описана в пункті 2.6 процедура взяття остачі за модулем  $2^a - 1$  є коректною. Довести коректність аналогічної процедури

для знаходження остачі по модулю  $2^N + 1$  з пункту 2.7.

11. Що має передаватися в якості першого аргументу в функцію  $C$  на кроці 8 алгоритму Шенхаге, якщо  $W_2 < W_1$ ? Дати відповіді на аналогічні питання стосовно кроків 9-12.
12. Чому в пункті 1 алгоритму Штрасена в оцінці для  $n$  в лівій частині нерівності присутній коефіцієнт 2?
13. Чи можна зменшити дно рекурсії в процедурі **Множення за модулем** з пункту 2.7? (Підказка: звернути увагу на пункт 7 даної процедури)
14. Знайти асимптотичні оцінки складності алгоритмів перевірки довгого числа на простоту з пункту 2.12, вважаючи, де це потрібно, що кількість раундів є фіксованою константою.
15. Що є діаграмою Вороного для однієї точки? Для двох точок? Дати відповіді на аналогічні питання по відношенню до триангуляції Делоне та опуклої оболонки?
16. Що є діаграмою Вороного для  $n$  точок, розташованих на одній прямій? Чи коректно обробляється такий випадок алгоритмом Форчуна?
17. Чи можна узагальнити алгоритм Форчуна на набір точок, серед яких є точки з однаковими  $y$ -координатами? Якщо так, то які пункти алгоритму або які допоміжні процедури при цьому зміняться?
18. Як буде виглядати діаграма Вороного для набору точок, що розташовані по колу? Як при цьому буде виглядати триангуляція Делоне? Пов'язати відповіді на дані питання з міркуваннями, що передують рис. 3.7.
19. Якого типу має бути поле  $q$  в структурі `TValueLeaf`?
20. Де в полях структур лабораторної роботи №3 можна замість ітераторів використовувати константні ітератори?
21. Що є ключем в бінарному дереві пошуку, якщо його реалізовувати

згідно з принципами, описаними в пункті 3.7?

22. Чи може статися так, що в процесі виконання алгоритма Форчуна в чергу  $Q$  додається подія з пріоритетом вищим, ніж у щойно обробленої події?
23. Оцінити асимптотичний час виконання процедур **Обробки події місця** та **Обробки події кола**. Оцінити асимптотично кількість подій в черзі  $Q$ . Поєднати отримані оцінки та довести, що алгоритм Форчуна працює за час  $O(n \ln n)$ .
24. Показати, що в геометричній інтерпретації алгоритму Форчуна замітаючу пряму можна замінити на замітаюче коло, яке розширюється зі свого центра до нескінченності, причому центр є довільною наперед заданою точкою площини. При цьому діаграма Вороного коректно будується всередині кола по мірі його розширення. Як тоді зміниться структура берегової лінії та якими кривими будуть описуватися дуги берегової лінії?
25. Показати, як з діаграми Вороного та триангуляції Делоне можна отримати опуклу оболонку заданого набору точок.
26. Отримати асимптотичні оцінки складності алгоритмів 3-6 лабораторної роботи №3 в залежності від кількості точок.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2001). *Introduction to Algorithms*. The MIT Press. ISBN: 0262032937
2. Crandall, Richard; Pomerance, Carl (2001), "Section 4.2.1: The Lucas–Lehmer test Prime Numbers: A Computational Perspective (1st ed.)", Berlin: Springer, pp. 167–170, ISBN 0-387-94777-9
3. Deitel, Paul, Deitel, Harvey. (2010). *C++ : how to program (7)*. New Jersey: Pearson.
4. Feller, W. (1968). *An Introduction to Probability Theory and Its Applications, Vol. 1 (3rd ed.)*. Wiley.
5. Feller, W. (1971). *An Introduction to Probability Theory and Its Applications, Vol. II (2nd ed.)*. Wiley.
6. Fortune S. A sweepline algorithm for Voronoi diagrams. Proceedings of the second annual symposium on Computational geometry. Yorktown Heights, New York, United States, pp.313–322. 1986. ISBN 0-89791-194-6.
7. Голубєва К.М., Кашпур О.Ф., Ключин Д.А. Чисельні методи (для студентів факультету комп'ютерних наук та кібернетики, ОП «Системний аналіз»): навчальний посібник. Київ: 2022. – 145 с.
8. Karatsuba, A. A.; Ofman, Y. P. (1962). "Multiplication of Many-Digital Numbers by Automatic Computers". Proceedings of the USSR Academy of Sciences (in Russian). 145: 293–294. Translation in the academic journal *Physics-Doklady*, 7 (1963), pp. 595–596.
9. Knuth, D. E. (1997). *The art of computer programming: Vol. 1. Fundamental algorithms (3rd ed.)*. Addison-Wesley.
10. Knuth, D. E. (1997). *The art of computer programming: Vol. 2 Seminumerical Algorithms (3rd ed.)*. Addison-Wesley.

11. Knuth, D. E. (1997). *The art of computer programming: Vol. 3 Sorting and Searching* (3rd ed.). Addison-Wesley.
12. Rabin, Michael O. (1980), "Probabilistic algorithm for testing primality *Journal of Number Theory*, 12 (1): 128–138, doi:10.1016/0022-314X(80)90084-0
13. Schönhage, Arnold; Strassen, Volker (1971). "Schnelle Multiplikation großer Zahlen"[Fast multiplication of large numbers]. *Computing* (in German). 7 (3–4): 281–292. doi:10.1007/BF02242355
14. Seysen M. A Simplified Quadratic Frobenius Primality Test // *Cryptology ePrint Archive*. — 2005. — P. 4–13
15. Solovay, Robert M.; Strassen, Volker (1977). "A fast Monte-Carlo test for primality". *SIAM Journal on Computing*. 6 (1): 84–85. doi:10.1137/0206006. See also Solovay, Robert M.; Strassen, Volker (1978). "Erratum: A fast Monte-Carlo test for primality". *SIAM Journal on Computing*. 7 (1): 118. doi:10.1137/0207009
16. Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.
17. Tóth, C. D., O'Rourke, J., Goodman, J. E. (Eds.). (2017). *Handbook of Discrete and Computational Geometry* (3rd ed.). Chapman and Hall/CRC.