A. Cathedral

Автор задачі: Дмитро Садовий
Кількість команд, що здали: 14 з 23 (61%)

В задачі потрібно відсортувати масив та знайти сьомий з кінця елемент. При цьому слід перевірити, що на сьомому місці стоїть унікальне число, тобто сусіди зліва або справа відрізняються від поточного. Також треба перевірити, що в масиві є більше 6 елементів, інакше відповіді не існує. Оскільки у відповідь необхідно вивести початковий номер (до сортування) знайденого елементу, потрібно або зберігати в масиві пари (висота, початковий номер), або зробити копію масиву перед сортуванням і знайти в цій копії індекс елементу, що дорівнює сьомому з кінця елементу у відсортованому масиві.

B. Річка та міста

Автор задачі: Леонід Беркович
Кількість команд, що здали: 5 з 23 (22%)

Можно использовать два массива (или один массив пар) с входными данными по всем городам.

Будем считать, что переходов (с левого берега на правый и с правого на левый) должно быть два и они должны относиться к разным городам. Во всех других случаях мы ничего не выигрываем.

Пусть исходные данные будут собраны в массив пар `pair<int, int> pairs[N]`, и каждый элемент массива содержит данные (числа объектов на левом и правом берегах) очередного города по пути следования.

Будем последовательно проходить в цикле все города по порядку.

Для каждого очередного города с индексом i рассмотрим три возможных случая: переходов через реку ещё не было, один переход уже был (с левого на правый берег), два перехода уже были.

Предположим, что у нас есть предыдущие (посчитанные для предыдущего города $i-1$) значения числа собранных объектов для всех трех случаев:

- `dp[2]` — для двух переходов,
- `dp[1]` — для одного перехода,
- `dp[0]` — без переходов.

Для сохранения этих значений я использовал массив `dp[]`, можно хранить в трех переменных.

Число объектов на берегах данного города `pairs[i].first` и `pairs[i].second`. Теперь получим новые значения:

- `dp[2] = max(dp[2] + pairs[i].first, dp[1] + pairs[i].first + pairs[i].second)`
  Число `dp[2] + pairs[i].first` получается, если оба перехода были раньше. Число `dp[1] + pairs[i].first + pairs[i].second` — если второй переход происходит в данном городе. Из этих чисел нас интересует максимальное значение.
- `dp[1] = max(dp[1] + pairs[i].second, dp[0] + pairs[i].first + pairs[i].second)`
  Первое число — если переход был раньше, второе — если в этом городе.
- `dp[0] += pairs[i].first`
  Переходов не было, поэтому просто прибавляем число объектов на левом берегу.

Так мы получили рекуррентное соотношение между прежними и новыми значениями. Для первого города:

- `dp[0] = pairs[0].first`
- `dp[1] = pairs[0].first + pairs[0].second`
- `dp[2] = 0`

Когда пройдем в цикле все города, результатом решения будет `dp[2]`, так как при двух переходах получаем максимальное число собранных объектов.

Асимптотическая временная сложность алгоритма $O(N)$.

Примечание. Если перебирать все возможные наборы двух переходов с применением префиксных сумм объектов на каждом берегу, надо пройти в цикле все сочетания из N по 2. Число сочетаний равно $N \cdot (N-1)/2$, сложность алгоритма была бы $O(N^2)$.

C. Корінь квантової переваги

Автор задачі: Денис Остапенко
Кількість команд, що здали: 14 з 23 (61%)

Дану задачу можна розв'язати двома способами. Перший спосіб — просто обчислити значення квадратного кореня (що б там не говорилося в легенді, квантовий комп'ютер для цього не потрібен). Це досить легко зробити при використанні мов програмування, в яких є вбудована довга арифметика. В Java та Kotlin можна використати клас `BigInteger` та його метод `sqrt()`. У Python усі числа автоматично є довгими, а квадратний корінь можна обчислити за допомогою функції `math.isqrt()`. У C++ на процесорах x86 можна використати тип `long double` та функцію `std::sqrt()` (за виключенням компілятору Visual Studio, де `long double` — те саме, що `double`). Звісно, на відміну від довгої арифметики, обчислення в `long double` не дасть точну відповідь, але кількість розрядів і перша цифра будуть правильними. Тип `double` використати не можна, бо він не здатний представляти такі великі числа. Після обчислення квадратного кореня результат можна конвертувати в рядок і вивести його довжину й перший символ.

Другий спосіб розв'язання задачі — математичний і більш оптимальний. Розглянемо його докладно. Інтуїтивно зрозуміло, що в квадратному корені цифр буде приблизно в два рази менше, ніж у квадраті, тому що $(10^n)^2 = 10^{2n}$. Знайдемо точну формулу для кількості цифр. Припустимо, що в нас є число x, десятковий запис якого має довжину $n+1$ символів. Це число задовільняє нерівність:

$$10^n \le x < 10^{n+1}$$

Піднесемо нерівність у квадрат:

$$10^{2n} \le x^2 < 10^{2n+2}$$

Перепишемо її для довжин десяткових записів відповідних чисел:

$$2n+1 \le len(x^2) < 2n+3$$

Перетворивши цю нерівність, можна отримати:

$$\frac{len(x^2)-1}{2} - 1 < n \le \frac{len(x^2)-1}{2}$$

Оскільки n є цілим, це означає, що:

$$n = \lfloor \frac{len(x^2)-1}{2} \rfloor$$

Тут $\lfloor \rfloor$ позначає округлення вниз. Отже, потрібно обчислити n для заданого на вході квадрату і вивести у відповідь $n+1$, тобто:

$$len(x) = \lfloor \frac{len(x^2)-1}{2} \rfloor + 1$$

Тепер знайдемо першу цифру. Інтуїтивно зрозуміло, що перша цифра квадратного кореня залежить лише від кількох перших цифр квадрату. Знайдемо, скільки цифр потрібно врахувати. Припустимо, що в нас є число x довжиною $n+1$ цифр, першою цифрою якого є d. Це число задовільняє нерівність:

$$d \cdot 10^n \le x < (d+1) \cdot 10^n$$

Знову піднесемо нерівність у квадрат і перетворимо її:

$$d^2 \cdot 10^{2n} \le x^2 < (d+1)^2 \cdot 10^{2n}$$

$$d^2 \le \frac{x^2}{10^{2n}} < (d+1)^2$$

$$d \le \sqrt{\frac{x^2}{10^{2n}}} < d+1$$

Оскільки d є цілим, це означає, що:

$$d = \lfloor \sqrt{\lfloor \frac{x^2}{10^{2n}} \rfloor} \rfloor$$

При цьому операцію $\lfloor \frac{x^2}{10^{2n}} \rfloor$ можна виконати, відкинувши останні $2n$ цифр числа x^2. Тоді від числа x^2 залишаться лише перші $len(x^2) - 2n = len(x^2) - 2\lfloor \frac{len(x^2) - 1}{2} \rfloor = 2 - len(x^2) \bmod 2$ цифри. Отже, потрібно взяти від заданого на вході квадрату перші одну чи дві цифри, обчислити квадратний корінь отриманого числа, округлити його вниз і вивести у відповідь.

D. AI замінить програмістів

Автор задачі: Володимир Мшанецький
Кількість команд, що здали: 18 з 23 (78%)

В даній задачі потрібно обчислити значення виразу $\lceil \frac{H \cdot W \cdot D}{h \cdot w \cdot d} \rceil$, де $\lceil x \rceil$ означає округлення вгору. Виконати ділення з округленням вгору можна двома способами: або виконати ділення в числах з рухомою комою і потім округлити результат за допомогою функції `ceil()`, або виконати ділення в цілих числах за формулою `(a + b - 1) / b` (де `/` — операція цілочисельного ділення з округленням вниз; у Python — `//`). У другому варіанті повний вираз буде `(H * W * D + h * w * d - 1) / (h * w * d)`. Чому це працює читачу пропонується подумати самостійно, розглянувши кілька прикладів із невеликими числами.

Під час реалізації розв'язку слід звернути увагу на діапазон можливих значень чисел у вхідних даних. Оскільки кожне з чисел може бути до 2000, то як проміжний результат обчислень, так і відповідь, можуть досягати $8 \cdot 10^9$, тому ані діапазону значень 32-бітового цілого (`int`), ані точності `float` для обчислень недостатньо. Тому слід використовувати 64-бітове ціле (`long long` в C++, `long` в Java/Kotlin) або `double`. В останньому випадку також слід переконатися, що відповідь виводиться у вигляді цілого числа. Наприклад, якщо відповідь дорівнює 10^9 і змінна має тип `double`, то стандартні функції виводу можуть вивести її як «`1e9`», що не відповідає очікуваному формату. Це можна виправити, наприклад, привівши кінцевий результат до `long long` у C++, `long` у Java/Kotlin або `int` у Python.

До речі, враховуючи назву задачі, я радий повідомити вам, що GPT впорався з цією задачею з першої спроби.

E. AI замінить страйкарів

Автор задачі: Володимир Мшанецький
Кількість команд, що здали: 2 з 23 (9%)

Розв'язок даної задачі складається з двох етапів. На першому етапі потрібно знайти всі відрізки-кандидати, тобто такі трійки точок, дві з яких є кінцями відрізку, а третя — його серединою. На другому етапі потрібно порахувати кількість хрестиків, що утворюються знайденими відрізками-кандидатами, тобто кількість пар знайдених відрізків-кандидатів, що задовільняють вимогам про співвідношення довжини і незнаходження на одній прямій. Розглянемо ці етапи окремо.

Пошук відрізків-кандидатів можна реалізувати двома вкладеними циклами: перший буде перебирати середню точку, а другий — перший кінець відрізка. Маючи перший кінець і середину, ми можемо легко розрахувати очікувані координати другого кінця. Далі потрібно перевірити, чи дійсно існує точка з такими координатами серед заданих. Для цього можна перед початком помістити всі задані точки у множину. Тоді перевірку існування розрахованого другого кінця можна швидко виконувати за допомогою пошуку у множині. Якщо точка із розрахованими координатами другого кінця існує, то ми знайшли відрізок-кандидат. Але помітимо, що шукаючи відрізки-кандидати таким чином, ми знайдемо кожен з них два рази: перший раз, коли внутрішній цикл буде проходити через один з кінців відрізка, і другий раз — коли він буде проходити через інший кінець (тобто, коли перший та другий кінець поміняються місцями). Щоб позбутися дублікатів, можна відкидати всі відрізки-кандидати, в яких $x_1 > x_2$, або $x_1 = x_2$ та $y_1 > y_2$.

Згадаємо, що хрестики завжди складаються з відрізків-кандидатів, що мають спільну середню точку. Завдяки цьому, під час пошуку відрізків-кандидатів, знайшовши всі відрізки, що мають поточну зафіксовану у зовнішньому циклі середню точку, ми можемо одразу їх опрацювати (тобто, підрахувати й додати до відповіді кількість хрестиків серед них) і забути. Можна було би замість цього спочатку знайти всі відрізки-кандидати і згрупувати їх по середній точці, а потім окремо рахувати кількість хрестиків в кожній групі, але це менш ефективно. Отже, насправді «другий етап» слід виконувати всередині першого.

Усі знайдені відрізки-кандидати для поточної середньої точки потрібно запам'ятовувати. Можна запам'ятовувати їхні кінці, але краще одразу розраховувати та запам'ятовувати квадрат їхньої довжини та їхній «напрямок» (про нього нижче). Слід використовувати саме квадрат довжини, щоб не мати проблем із похибками при обчисленні квадратного кореня, а також щоб зекономити час на його обчислені. Позначимо кількість знайдених відрізків-кандидатів для поточної середньої точки як M, квадрати їхніх довжин як L_i^2, а напрямки — як D_i ($1 \le i \le M$).

Тепер потрібно порахувати кількість хрестиків серед знайдених відрізків-кандидатів. Спочатку забудемо про вимогу, що відрізки не повинні знаходитись на одній прямій, та порахуємо кількість пар відрізків-кандидатів, які задовольняють вимогу про співвідношення довжин відрізків. В умові задане співвідношення $\frac{min(L_1, L_2)}{max(L_1, L_2)} \ge \frac{R}{10^3}$, але, оскільки ми працюємо з квадратами довжин, його слід звести у квадрат: $\frac{min(L_1^2, L_2^2)}{max(L_1^2, L_2^2)} \ge \frac{R^2}{10^6}$. Щоб позбутися ділення та пов'язаних із ним похибок, перепишемо нерівність як $min(L_1^2, L_2^2) \cdot 10^6 \ge max(L_1^2, L_2^2) \cdot R^2$. Для того, щоб порахувати кількість пар відрізків-кандидатів, що задовольняють цю нерівність, відсортуємо всі знайдені відрізки-кандидати за зростанням квадрату довжини, та застосуємо метод двох вказівників. Перший вказівник i буде перебирати поточний короткий відрізок від 1 до M, а другий, j — найдовший відрізок, який задовольняє нерівність разом із поточним коротким відрізком. Очевидно, що всі відрізки в діапазоні $i \dots (j-1)$ також будуть задовольняти нерівність. На кожній ітерації циклу будемо збільшувати i на 1, після чого збільшувати j, поки нерівність виконується. Після цього будемо додавати до відповіді $j-i$, тобто довжину діапазону $(i+1) \dots j$ (пару i з самим собою рахувати не слід).

Тепер повернемося до вимоги, що відрізки, з яких складається хрестик, не повинні знаходитися на одній прямій. Обчислимо для кожного відрізка-кандидата його «напрямок». Напрямком можна

вважати, наприклад, коефіцієнт k із рівняння прямої $y=kx+b$. Оскільки середня точка відрізка у нас зафіксована, то якщо у двох відрізків збігається k, тоді вони знаходяться на одній прямій. Але, щоб уникнути ділення і пов'язаних із ним похибок, краще в якості напрямку використовувати вектор із першої точки відрізку в другу. При цьому, щоб для всіх відрізків, що знаходяться на одній прямій, цей вектор був однаковим, слід розділити його на $gcd(\Delta x, \Delta y)$, і після цього, якщо $\Delta x<0$, то помножити його на -1. Ситуації $\Delta x=0$ та $\Delta y=0$ слід обробити окремо. Вектор, отриманий таким чином для відрізка-кандидата i ми і будемо вважати його напрямком D_i. Як зазначалося вище, розрахувати значення D_i для всіх знайдених відрізків-кандидатів краще ще на етапі їхнього пошуку.

Тепер ми можемо модифікувати описаний вище підрахунок кількості пар відрізків-кандидатів, що задовольняють вимогу про співвідношення довжин, таким чином щоби він не рахував пари відрізків-кандидатів, що мають однаковий напрямок. Для цього для кожного i будемо віднімати від відповіді кількість відрізків-кандидатів у діапазоні $i \le k \le j$, у яких $D_k=D_i$. Для цього створимо map, ключем якого буде вектор напрямку, а значенням — кількість відрізків-кандидатів у діапазоні $i \dots j$, які мають такий напрямок. Позначимо його C. Після кожного збільшення вказівника j будемо збільшувати $C[D_j]$ на одиницю, а перед кожним збільшенням i — зменшувати $C[D_i]$ на одиницю. При цьому, на кожній ітерації циклу по i потрібно замість $j-i$ додавати до відповіді $j-i-C[D_i]$, тобто кількість відрізків, що підходять до i за довжиною мінус кількість відрізків серед них, що також знаходяться на одній прямій із відрізком i.

Описаний вище розв'язок є авторським і є достатнім, щоб вкластися у ліміт часу виконання. Тим не менше, під час дорішування один з учасників, Максим Молчанов (ExOPNU_MaxOrTru), знайшов розв'язок, який працює у кілька разів швидше. Обидва розв'язки мають однакову асимптотику ($O(N^2 \log(N))$), але розв'язок Максима Молчанова містить дві суттєві оптимізації. Якщо ви дочитали до цього місця, то вам, ймовірно, буде цікаво про них дізнатися, тому розглянемо їх.

Перша оптимізація — оптимізація перевірки існування парної точки на етапі пошуку відрізків-кандидатів. Замість того, щоб використовувати для цього множину, відсортуємо всі задані точки за зростанням x, а при рівних x — за зростанням y. Тепер не складно помітити, що, якщо в нас зафіксована поточна середня точка i, то у кожного відрізка-кандидата індекс одного з кінців j завжди буде більшим за i, а індекс другого k — меншим за i. Отже, цикл перебору першого кінця відрізку j можна запускати від $i+1$ до N. Більш того, можна довести, що по мірі зростання j, індекс другого кінця k буде весь час зменшуватися. Тому можна спочатку присвоїти $k=i-1$, а потім для кожного j зменшувати k поки не знайдемо парну точку, або поки точка k не стане меншою за критерієм сортування ніж очікувана парна точка. При цьому, якщо k досягне 0 раніше, ніж j досягне $N+1$, цикл можна перервати достроково, бо знайти нові відрізки-кандидати вже неможливо. Таким чином, пошук усіх відрізків-кандидатів для однієї середньої точки виконується лінійно.

Друга оптимізація — використовувати для C не map, а масив. Для цього потрібно перевести вектори напрямку відрізків-кандидатів у послідовні числа. Для цього створимо новий map ключем якого буде вектор, а значенням — послідовний номер цього вектору серед усіх знайдених унікальних векторів. Використовуючи цей новий map під час пошуку відрізків-кандидатів ми можемо, знайшовши новий відрізок-кандидат, перевести його вектор у номер і вже цей номер запам'ятати як D_i. На перший погляд може здатися, що ми замінили один map іншим, але таким чином ми зменшимо кількість пошуків у map вдвічі, що суттєво зменшує час роботи програми. Якщо при цьому використовувати hash-map, то можна скоротити час ще трошки. Після цієї оптимізації підрахування хрестиків серед знайдених відрізків-кандидатів також працюватиме лінійно.

До речі, ідею легенди цієї задачі, а також картинку до неї, створив Gemini. Отже, штучний інтелект замінить ще й авторів задач. Так, стоп, зачекайте...

F. Chat limits

Автор задачі: Дмитро Садовий
Кількість команд, що здали: 19 з 23 (83%)

Тут є два розв'язки задачі. Перший з них — це в циклі промоделювати кожен місяць та порахувати на якому з них кількість повідомлень перетне межу в S.

Другий розв'язок — математичний. Давайте спробуємо порахувати кількість повідомлень на момент кінця місяця X. Для цього використаємо формулу суми арифметичної прогресії:

$$S_n = \frac{n(2a_1 + d(n-1))}{2}$$

У нашому випадку це буде:

$$S_X = \frac{X(2NM + EM(X-1))}{2}$$

Трохи змінивши формулу, отримуємо квадратне рівняння:

$$EMX^2 + (2NM - EM)X - 2S = 0$$

Знайдемо X через дискримінант:

$a = EM$

$b = 2NM - EM$

$c = -2S$

$D = b^2 - 4ac$

$x_1 = \frac{-b + \sqrt{D}}{2a}$

Знайдений x_1 потрібно округлити в меншу сторону.

Також існує і третій варіант розв'язку, де замість рівняння ми просто бінарним пошуком намагаємось підібрати цей X.

G. Знайти цифру в степені двійки

Автор задачі: Денис Остапенко
Кількість команд, що здали: 10 з 23 (43%)

Розв’язок даної задачі ґрунтується на тому, що десяткові цифри в числах 2^N зустрічаються практично рівноймовірно, тому ймовірність того, що деяка десяткова цифра не зустрінеться серед певної, достатньо великої, кількості останніх розрядів експоненційно падає з ростом кількості цих розрядів. Автор задачі підрахував, що при $N \leq 10^7$ у останніх 178 розрядах числа 2^N зустрічаються усі десяткові цифри, які є в числі, і при цьому починаючи з $N=168$ у кожному числі є всі десяткові цифри. Отже, потрібно обчислити щонайменше 178 останніх розрядів 2^N і виконати пошук потрібної цифри серед них. На практиці, шукати магічне число 178 не потрібно, бо можна просто обчислити кілька тисяч останніх розрядів. Скільки розрядів обчислювати можна підібрати, наприклад, замірявши час роботи вашого розв’язку, та поставивши максимальну кількість, при якій ваш розв’язок вкладається в ліміт на час виконання з певним запасом.

Дану задачу значно легше розв'язати на мовах програмування, в яких є вбудована довга арифметика. У Java та Kotlin можна використати клас `BigInteger` та його метод `modPow()`. У Python усі числа є довгими, а для піднесення до степеня можна використати вбудовану (не з модулю `math`) функцію `pow()` із трьома аргументами. В обох випадках останнім аргументом функції є модуль, за яким потрібно обчислити результат піднесення до степеня. Відповідно, щоби отримати останні, наприклад, 1000 цифр, потрібно вказати модуль 10^{1000}. Отриманий результат можна конвертувати у рядок і виконати пошук потрібної цифри в ньому. Крім того, можна обчислити повний результат піднесення до степеня (`BigInteger.pow()` або `**` у Python) і вже після цього окремо обчислити його залишок від ділення на 10^{1000} (`BigInteger.mod()` або `%`). Можна й не обчислювати залишок від ділення на 10^{1000} взагалі, але в такому випадку конвертувати результат у рядок не можна, бо така конвертація займе занадто багато часу. Замість цього можна виконати пошук цифри у циклі, послідовно обчислюючи залишок від ділення на 10 та ділячи число на 10.

Розв’язати дану задачу на C++ значно складніше, бо для її розв'язання необхідно реалізувати власну довгу арифметику. Розгляд реалізації довгої арифметики на C++ виходить за рамки цього розбору (пропонуємо пошукати в інтернеті, або запитати у вашої улюбленої LLM), але, в двох словах, потрібно зберігати розряди числа в масиві та вручну реалізувати всі потрібні математичні операції (додавання, множення, тощо) як у стовпчик. При цьому потрібно врахувати, що, оскільки N може бути досить великим, виконання піднесення до степеня лінійно (тобто, шляхом послідовного множення на два N разів) може працювати занадто довго. Тому рекомендується реалізувати алгоритм бінарного піднесення до степеня. Для нього буде потрібно реалізувати множення довгого на довге. Це досить трудомістко, тому альтернативно можна реалізувати довгу арифметику, зберігаючи по 18 розрядів на елемент масиву (у типі `long long`). Тоді лінійне піднесення до степеня також буде проходити за лімітом на час виконання, якщо обчислювати лише останні 200-300 розрядів. У цьому випадку буде потрібно реалізувати лише множення довгого на коротке, що значно простіше. При реалізації бінарного піднесення до степеня можна зберігати по одному розряду на елемент масиву.

H. Bicycle Paths

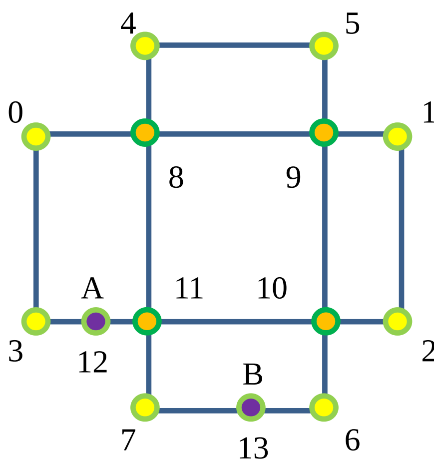
Автор задачі: Олександр Нестерюк
Кількість команд, що здали: 2 з 23 (9%)

Задача пошуку найкоротшого шляху на велосипедних доріжках може бути розв'язана кількома можливими способами. По-перше, можна вручну перебрати всі окремі випадки розташування прямокутників один щодо одного (не перетинаються, перетинаються в одній або декількох точках або мають частково або повністю спільну сторону) і розташування початкової (A) і кінцевої (B) точок на прямокутниках (лежать на різних прямокутниках чи одному, збігаються чи ні зі своїми вершинами чи взагалі друг з одним). По-друге, можна відсіяти спочатку невелику кількість окремих випадків (такі, як збіг точок A і B), а для всіх інших побудувати граф і шукати мінімальний шлях на ньому. І по-третє, реалізувати одразу універсальний розв'язок, побудувавши відповідний граф, що складається з вершин, що є вершинами прямокутників, точок перетину сторін прямокутників, а також точок A та B.

У цьому розборі розглянемо саме останній підхід до розв'язання цієї задачі.

Отже, задача може бути зведена до задачі пошуку шляху на графі, для розв'язання якої можна скористатися одним із відомих методів розв'язку. Для ситуації конкретної задачі можна застосувати алгоритм Дейкстри знаходження найкоротшого шляху. Цей алгоритм передбачає роботу з матрицею суміжності досліджуваного графа. Таким чином, розв'язання задачі зводиться до наступного:

1. Перетворення шляхів між вершинами прямокутників та їх точками перетинів у відповідний граф;
2. Побудова матриці суміжності одержаного графа;
3. Знаходження найкоротшого шляху між початковою та кінцевою вершинами отриманого графа, причому сам шлях можна не обчислювати, обмежившись знаходженням його довжини.



Перший етап розв'язку може бути поділено на 4 частини.

Перша частина — підготовча — полягає у формуванні двох масивів, у першому з яких зберігатимуться вершини створюваного графа, тоді як у другому — його ребра. Тут можна відзначити, що граф міститиме від 6 до 14 вершин (оскільки повний збіг двох прямокутників неможливий згідно з логікою завдання, інакше мінімальна кількість вершин скоротилася б до 4). Далі будемо орієнтуватися саме на найскладніший випадок із максимальною кількістю вершин у графі. Такий випадок показаний малюнку. Тут номерами 0-3 та 4-7 позначені вершини — кути прямокутників, 8-11 — їх можливі точки перетинів, 12 — початкова точка шляху A, 13 — кінцева точка B. Як програмну реалізацію тут можливе використання як контейнерів, так і статичних масивів.

Друга частина першого етапу полягає у додаванні в дані масиви вершин прямокутників 0-3 та 4-7, а також зв'язків між цими 4 вершинами для першого та другого прямокутників.

Третя частина першого етапу полягає у знаходженні можливих точок перетинів прямокутників та додаванні відповідних вершин 8-11 та ребер графа у відповідні масиви. Тут необхідно перевірити на можливі перетину всі вертикальні та горизонтальні сторони обох прямокутників. Однак, тут можна перевірити на перетин лише сторони 0-1 з 4-7 і 5-6, а також 3-2 з 4-7 і 5-6. В останньому випадку виникне необхідність повторити розв'язання задачі, помінявши місцями значення координат вершин першого та другого прямокутників, з наступним вибором мінімального із двох отриманих значень результату. Тут також при розрахунку довжин ребер між вершинами 8-11 та 0-7 також можливі два шляхи розв'язання. Перший полягає у видаленні вже доданих у попередній частині етапу ребер (сторін прямокутників), що розриваються вершинами 8-11. Другий — у простому додаванні нових ребер зі збереженням тих, що існують. Вибір першого шляху при витраті

часу на пошук та видалення дещо прискорить подальшу роботу алгоритму Дейкстри, проте через невелику кількістю вершин обидва шляхи тут можливі.

У ході четвертої частини в граф, що будується, додаються вершини початкової і кінцевої точок A і B. На малюнку вони позначені як 12 і 13 відповідно. Для їх додавання, а також пов'язаних з ними ребер повністю застосовні обидва шляхи вирішення попередньої частини. Тут потрібно послідовно перевірити належність даних точок усім сторонам обох прямокутників.

У процесі заключної частини розв'язку необхідно перевірити всі вершини сформованого графа на збіги по координатах. Тут знову можливі два шляхи реалізації. Перший полягає в об'єднанні двох вершин, що збіглися, і видаленні дублікатів з подальшим внесенням змін до масиву ребер. Цей шлях також дозволить надалі скоротити обчислювальні витрати на алгоритм Дейкстри. Другий шлях полягає в додаванні спеціальних ребер між вершинами з координатами, що збігаються, з нульовою вагою. Цей шлях надалі вимагатиме деякої модифікації самого алгоритму Дейкстри, оскільки нульова вага ребра сприйматиметься як відсутність зв'язку між вершинами в матриці суміжності. В цьому випадку як відсутність зв'язків необхідно вибрати, наприклад, значення -1 з відповідними змінами реалізації алгоритму.

Перший етап розв'язку може бути з'єднаний з другим — одразу побудувати матрицю суміжності необхідного графа. Для отримання шуканого графа потрібно послідовно пронумерувати всі вершини і точки перетинів прямокутників (наприклад, так як це показано на малюнку), які при цьому стають вершинами графа, що формується, і захопити двовимірний квадратний масив пам'яті для зберігання матриці суміжності C розмірності $N \times N$ (де N — кількість отриманих першому етапі вершин). Значення елементів матриці визначаються можливістю переходу між двома сусідніми вершинами графа і становлять ту ціну (тобто відстань), яку необхідно витратити на здійснення даного переходу, якщо даний перехід можливий, або 0 (або -1, якщо використовувався другий шлях у п'ятій частині першого етапу) в іншому випадку.

Всі інші елементи матриці C дорівнюватимуть 0 (або -1, якщо використовувався другий шлях у п'ятій частині першого етапу). Як видно з наведених формул, отримана матриця C є квадратною матрицею з нульовими елементами на головній діагоналі та елементи над головною діагоналлю є відображенням елементів під головною діагоналлю (оскільки граф не є спрямованим, тобто переміщення можливі з однаковою ціною в будь-якому напрямку по ребрах). Звідси випливає, що кількість елементів, що зберігаються, може бути скорочена як мінімум в 2 рази, що не є необхідним при заданих обмеженнях.

Далі застосовується алгоритм Дейкстри, який полягає у знаходженні мінімальної ціни, яку необхідно заплатити задля досягнення кожної з вершин. Для цього захоплюється одномірний масив D розмірністю, що дорівнює кількості вершин N, елементи якого дорівнюватимуть ціні на поточний крок алгоритму. Спочатку всі елементи даного масиву дорівнюють максимально можливому значенню цілого числа `MAX_INT`, крім відповідного початковій точці A, який дорівнює 0. При такій побудові кінцевий елемент масиву D відповідає цільовій точці B, і результатом задачі можна вважати значення, отримане в останньому елементі масиву D. На початковому етапі вершина A розглядається як поточна. Ще один масив такої ж розмірності знадобиться для зберігання інформації про те, чи було розглянуто («відвідано») відповідну вершину графа на попередніх кроках алгоритму.

Кількість кроків алгоритму визначається кількістю зв'язків між вершинами. Крок алгоритму полягає у виборі поточної вершини (з найменшою ціною в масиві D) з тих, які мають зв'язки з уже «відвіданими» на попередніх кроках та переборі всіх «невідвіданих» вершин, пов'язаних із нею. Після закінчення кроку поточна вершина позначається як «відвідана».

На кожному кроці алгоритму розраховуються нові значення елементів масиву D. Значення елементів D_i для вершини i, пов'язаної з поточною вершиною j, виходить рівним значенню $D_j + C_{ij}$. Тобто, обчислення організуються для всіх ненульових елементів матриці i-го рядка C. При обчисленні нових значень елементів масиву D вибирається мінімальне значення з наявного і нового. Наприклад, якщо вершину i можна потрапити двома способами, витративши 5 чи 8 одиниць (старе значення 5, а нове 8), то значення елемента $D_i = 5$. Також необхідно надходити, якщо попереднє значення елемента $D_i = \text{MAX_INT}$ (тобто, у цю вершину ще не було знайдено

шляху з вихідної і новий шлях є єдиним).

Алгоритм закінчується тоді, коли не залишилося жодного не розглянутого зв'язку між усіма вершинами, тобто, всі вершини відзначені як «відвідані» або з зв'язків, що залишилися, не існує.

Після закінчення роботи алгоритму залишається перевірити останній елемент масиву D і вивести його значення або, якщо воно дорівнює `MAX_INT`, тоді -1, оскільки шляху з початкової точки A в кінцеву точку B не існує.

Крім описаного вище авторського розв’язку, існує ще альтернативний, значно менш оптимальний, але простий для реалізації. В цьому розв’язку в граф додаються не лише вершини прямокутників і точки перетину сторін, а абсолютно всі точки з цілочисельними координатами, які лежать на сторонах прямокутників. При такому підході побудова графу стає тривіальною: потрібно лише пройти чотирма циклами по кожній зі сторін прямокутника, створити нову вершину для кожної точки (якщо для цієї точки ще не існує вершини) та з’єднати її з попередньою, після чого повторити все це для другого прямокутника. Пошук найкоротшого шляху в цьому випадку виконується за допомогою пошуку в ширину, бо довжина всіх ребер дорівнює 1. Такий розв’язок є досить повільним, але при даних обмеженнях на максимальне значення координат він вкладається в ліміт на час виконання.