Жуковський Сергій,
кандидат педагогічних наук,
доцент кафедри комп'ютерних наук та інформаційних технологій,
Шимон Максим,
здобувач другого (магістерського) рівня вищої освіти
фізико-математичного факультету,
Житомирський державний університет імені Івана Франка,
м. Житомир, Україна

ОПТИМІЗАЦІЯ АЛГОРИТМІВ ПРИ ВИКОРИСТАННІ МАСИВУ ПРЕФІКСНИХ СУМ

Структури даних є фундаментальними елементами в програмуванні, які дозволяють зберігати і організовувати дані у відповідний спосіб. Їх можна розділити на кілька категорій: лінійні, асоціативні, ієрархічні та мережеві [1].

Під час розв'язання задач важливим елементом є оптимізація алгоритму. Алгоритм повинен працювати швидко і використовувати при можливості найменше пам'яті. Існує ряд задач де потрібно постійно знаходити суму послідовних елементів масиву. Наприклад, для знаходження сумарну кількість опадів за певний період, або прибуток на біржі за певний інтервал часу. Якщо постійно перераховувати суму елементів масиву, при великому масиві N та великій кількості запитів M кількість операцій дорівнюватиме M*N. Для прискорення роботи алгоритму використаємо масив часткових сум або масив префіксних сум.

Масив префіксних сум (prefix sum array) – це структура даних, що використовується для швидкого обчислення суми підрядка (значень з певного діапазону) масиву.

0	1	2	3	4	5	6	7	8	9	10
	3	5	4	1	3	4	7	4	3	4

Рис. 1 Вхідний масив

Для створення масиву префіксних сум, необхідно спочатку обчислити суму всіх елементів масиву та зберегти її в першому елементі масиву префіксних сум. Далі, кожний наступний елемент масиву префіксних сум буде дорівнювати сумі всіх елементів з початку масиву до поточної позиції.

0	1	2	3	4	5	6	7	8	9	10
0	3	8	12	13	16	20	27	31	34	38

Рис. 2 Масив префіксних сум

Таким чином, якщо потрібно обчислити суму підрядка масиву від i до j ($i \leq j$), то вона буде дорівнювати різниці між j-тим та (i-1)-м елементами масиву префіксних сум.

$$sum_{i,j} = sum_j - sum_{i-1}$$

Масив префіксних сум [2] дозволяє зменшити час обчислення суми підрядка з O(n) до O(1). Це знайшло застосування в різних задачах, таких як обчислення кумулятивної суми елементів масиву, знаходження максимальної суми підрядка та інших алгоритмах.

Ще однією корисною властивістю масиву префіксних сум є можливість ефективно змінювати елементи вихідного масиву. Якщо змінити один елемент, то для оновлення масиву префіксних сум достатньо змінити значення відповідного елемента.

Матриця префіксних сум (рис. 1.3) – це розширена версія масиву префіксних сум, яка дозволяє швидко обчислювати суму підрядка в двовимірному масиві.

Для створення матриці префіксних сум sum з початкової матриці A потрібно для кожного елемента матриці вирахувати значення за наступною формулою:

$$\mathbf{sum}_{i,\,j}=\mathbf{A}_{i,\,j}+\mathbf{sum}_{i-1,\,j}+\mathbf{sumi}_{,\,j-1}-\mathbf{sum}_{i-1,\,j-1}$$

Таким чином, якщо потрібно обчислити суму підрядка матриці від (i_1, j_1) до (i_2, j_2), то її можна вирахувати за значеннями чотирьох елементів матриці префіксних сум:

$$\mathbf{sum}_{i2,\,j2}-\mathbf{sumi}_{1-1,\,j2}-\mathbf{sum}_{i2,\,j1-1}+\mathbf{sum}_{i1-1,\,j1-1}$$

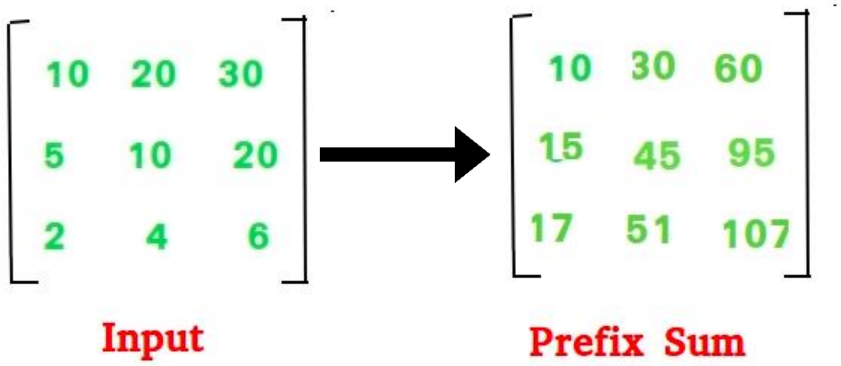

Рис. 1.2. Побудова матриці сум для двовимірного масиву.

Матриця префіксних сум дозволяє зменшити час обчислення суми підмасиву в двовимірному масиві з $O(n^2)$ до O(1). Це знайшло застосування в різних задачах, таких як розрахунок мінімальної суми в прямокутній області та інших алгоритмах.

Розглянемо приклади.

Задача 1. Мінімальний підмасив

Джерело: https://leetcode.com/problems/minimum-size-subarray-sum/description/

Умова:

Дано масив натуральних чисех *nums* та натуральне число *target*. Знайдіть мінімальну довжину підмасиву сума чисел якого не менша за *target*. Якщо такого підмасиву немає, поверніть 0.

1 <= target <= 10^9

1 <= nums.length <= 10^5

1 <= nums[i] <= 10^4

Приклад 1

Вхідні дані:

target = 7, nums = [2,3,1,2,4,3]
Вихідні дані:
2
Пояснення:
Підмасив [4,3] має мінімальну довжину та суму 7
Приклад 2
Вхідні дані:
target = 11, nums = [1,1,1,1,1,1,1,1]
Вихідні дані:
0
Пояснення:
Немає жодного підмасиву з сумою не меншою за 11
Розбір:

Для початку будемо виконувати наступний очевидний алгоритм: шукати окремим циклом суму для всіх можливих підмасивів і оновлювати ans, коли ми отримаємо кращий підмасив, який відповідає вимогам $(sum \geq s)$. Код:

```
for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
                int sum = 0;
                for (int k = i; k <= j; k++) {
                        sum += nums[k];
                }
                if (sum >= s) {
                        ans = min(ans, (j - i + 1));
                        break;
        }
}
```

Проте даний підхід отримає обмеження по часу через час виконання *O(n^3).*

В попередньому підході ви могли зауважити що підрахунов суми для кожного підмасиву виконується за *O(n)* часу. Проте ми можумо скоротити цей час до *O(1)* попередньо підрахувавши масив префіксних сум для масиву *nums*.

Далі, ми можемо скористатись бінарним пошуком. Зауважимо що ми шукаємо масив з *sum>=s* який починається з позиції *i* за *O(n)* часу. Це можна пришвидшити до *O(log(n))* використавши бінарний пошук правої позиції підмасиву який починається з позиції *i*, або використовуючи фунцію *lower_bound* в бібліотеці C++ STL.

Код:

```
int minSubArrayLen(int s, vector<int>& nums)
{
        int n = nums.size();
        if (n == 0)
                return 0;
        int ans = INT_MAX;
```

```
vector<int> sums(n + 1, 0);
for (int i = 1; i <= n; i++)
        sums[i] = sums[i - 1] + nums[i - 1];
for (int i = 1; i <= n; i++) {
        int to_find = s + sums[i - 1];
        auto bound = lower_bound(sums.begin(), sums.end(), to_find);
        if (bound != sums.end()) {
                ans = min(ans, static_cast<int>
                        (bound - (sums.begin() + i - 1)));
        }
}
return (ans != INT_MAX) ? ans : 0;
}
```

Задача 2. Максимальна сума після перестановки

Джерело: https://leetcode.com/problems/maximum-sum-obtained-of-any-permutation/description/

Умова:

Ми маємо масив цілих чисел *nums*, масив запитів *request*, де *requests[i] = [l_i, r_i]*. Результатом на кожен запипит *i* є сума елементів масиву *nums* на проміжку *[l_i, r_i]* (*nums[l_i] + nums[l_i + 1] + ... + nums[r_i - 1] + nums[r_i]*).

Знайдіть максимальну загальну суму для всіх запитів серед всіх перестановок масиву *nums*.

Оскільки результат може бути дуже великим, поверніть його за модулем $10^9 + 7$.

n == nums.length

$1 <= n <= 10^5$

$0 <= nums[i] <= 10^5$

$1 <= requests.length <= 10^5$

requests[i].length == 2

0 <= l_i <= r_i < n

Приклад 2

Вхідні дані:

nums = [1,2,3,4,5], requests = [[1,3],[0,1]]

Вихідні дані:

19

Пояснення:

Однаєю з перестановок є [2,1,3,4,5], тоді відповіді на запити будуть наступними:

requests[0] -> nums[1] + nums[2] + nums[3] = 1 + 3 + 4 = 8

requests[1] -> nums[0] + nums[1] = 2 + 1 = 3

Загальна сума: 8 + 3 = 11.

Перестановкою з максимально можливою сумою є [3,5,4,2,1], з наступними результатами:

requests[0] -> nums[1] + nums[2] + nums[3] = 5 + 4 + 2 = 11

requests[1] -> nums[0] + nums[1] = 3 + 5 = 8

Загальна сума: 11 + 8 = 19, що є максимальним результатом який можна отримати

Приклад 2

Вхідні дані:

nums = [1,2,3,4,5,10], requests = [[0,2],[1,3],[1,1]]

Вихідні дані:

47

Пояснення:

Відповіддю є перестановка [4,10,5,3,2,1], тоді суми для запитів будуть [19,18,10].

Розбір:

Ми можемо порахувати кількість входжень кожної позиції масиву в проміжки з запитів. Відповідно, для максимізації загального результату, на позиції з найбільшою кількістю входжень має стояти найбільше число з масиву, і на позиції з другою найбільшою кількістю входжень - друге найбільше число масиву, і т.д. На позиції з мінімальним входженням - найменше.

Для кожного запиту *[l, r]* ми встановимо *count[l]++* та *count[r+1]++*. І після підрахунку масиву префіксиних сум для масиву *count* ми матимемо значення *count[i]* яке відповідатиме частові входження *i* позиції в запити.

Далі, посортувавши масиви *nums* та *count* за спаданням ми можемо порахувати суму *count[i] * nums[i]* для кожного *i*.

Код програми:

```
int maxSumRangeQuery(vector<int> A, vector<vector<int>> req) {
    long res = 0, mod = 1e9 + 7, n = A.size();
    vector<int> count(n);
    for (auto &r: req) {
        count[r[0]] += 1;
        if (r[1] + 1 < n)
            count[r[1] + 1] -= 1;
    }
    for (int i = 1; i < n; ++i)
        count[i] += count[i - 1];
    sort(begin(count), end(count));
    sort(begin(A), end(A));
    for (int i = 0; i < n; ++i)
        res += (long)A[i] * count[i];
     return res % mod;
}
```

Дані алгоритми можна оптимально використовувати при розв’язанні задач, де використовується багато запитів, але масив статичний, що значно прискорить роботу алгоритму. Для змінного масиву варто використовувати структуру даних Дерево Фенвіка або Дерево відрізків.

Вивчати дані алгоритми можна з учнями, які готуються до шкільних олімпіад з інформатики на факультативних заняттях, або в профільних класах з поглибленим вивченням інформатики під час вивчення масивів та алгоритмів опрацювання масивів.

Список використаних джерел та літератури

1. Енциклопедія кібернетики : у 2 т. / за ред. В. М. Глушкова. Київ : Гол. ред. Української радянської енциклопедії, 1973.

2. Donald E. Knuth. The Art of Computer Programming. Volume 1 Fundamental Algorithms. 3rd Edition. Addison-Wesley Professional 1997. 682c.