МИНОБРНАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «ТУЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт прикладной математики и компьютерных наук

Кафедра информационной безопасности

КУРСОВАЯ РАБОТА

по дисциплине: «Языки программирования»

на тему:

«Префиксные суммы»

Выполнил:

студент 1 курса группы 230711

Павлова Виктория Сергеевна

Преподаватель:

доц. кафедры ИБ

Басалова Галина Валерьевна

Тула 2022

Оглавление

Задание 2

Введение 4

1 Постановка задачи 5

2 Теоретическая часть 6

3 Возможности применения алгоритма 12

4 Решение задач с использованием алгоритма 15

Заключение 34

Библиографический список 35

Приложение А 36

Приложение Б 37

Введение

Долгие годы различные структуры данных остаются строительной единицей в основе любой программы, поскольку позволяют обрабатывать или преобразовывать информацию для эффективной работы с ней. Вкупе с алгоритмами и разнообразными алгоритмическими методами создаются программы, которые способны быстро обрабатывать большие объёмы информации. Одна из таких структур данных являются темой данной курсовой работы – это массивы префиксных сумм данных.

Мы живём в эпоху господства информационных технологий: наше утро начинается с погружения в цифровое пространство, которое сопровождает нас до самого вечера. С каждым новым витком развития становится всё труднее представить себе жизнь без передовых технологий. Коммуникации, экономика, да едва ли не вся техносфера в целом – всё это потребляет и порождает огромные массивы данных. Перед передовыми программистами ставится непростая задача о создании таких программ, которые смогут осилить так называемые «Big Data» – информационные объёмы, исчисляемые уже терабайтами. Эта потребность подчёркивает актуальность выбранной темы, поскольку именно структуры данных, в том числе и массивы префиксных сумм, помогают в создании таких программ.

Таким образом, объектом исследования данной курсовой работы являются структуры данных под названием «префиксные суммы», а предметом исследования – алгоритм их образования и способ применения на практике. Целью курсовой работы является разработка задач и алгоритмов их решения с использованием вышеупомянутой структуры данных. Для изучения данной темы были поставлены конкретные задачи, определенные в разделе «1 Постановка задачи».

1 Постановка задачи

Для изучения такой структуры данных, как массивы префиксных сумм, были поставлены следующие задачи:

1. Определить теоретическую и практическую значимость такого массива данных, как префиксная сумма, для этого необходимо:
   а) рассмотреть суть понятия «префиксная сумма»;
   б) рассмотреть варианты реализации алгоритмов, связанных с префиксными суммами.
2. Разработать задачи, необходимые для демонстрации практического применения структуры данных «префиксные суммы», для этого необходимо:
   а) разработать условие для четырёх задач по указанной теме;
   б) разработать алгоритм решения для указанных задач;
   в) определить группы тестов для проверки эффективности выбранного пути решения.
3. Сделать выводы об итогах проведенного исследования.

2 Теоретическая часть

При обработке большого количества запросов к статическим данным для удобства можно заранее производить некоторые несложные операции с массивами входных данных. Например, можно определять для них массивы так называемых префиксных сумм за O(n) [1]. Слово «префиксная» довольно полно раскрывает саму суть понятия: поскольку префиксом в информатике называют подстроку, начинающуюся с начала основной строки, по аналогии образуются и массивы префиксных сумм – они насчитываются с первого элемента массива входных данных.

Пример 1. Создание массива префиксных сумм.

Пусть дан массив A: $[a_0, a_1, a_2, \ldots, a_{n-1}]$. Для него можно подсчитать такие суммы, что k-тая сумма равна сумме первых k элементов данного массива, то есть с a_0 до a_{k-1}. Такие префиксные суммы имеют вид $S_k = a_{k-2} + a_{k-1}$, отсюда можно выделить формулу (1) для поиска префиксных сумм:

$$S_{n+1} = S_n + a_n \quad (1)$$

Префиксные суммы удобно хранить в массиве S: $[S_0, S_1, S_2, \ldots, S_n]$, причём его длина всегда будет на единицу больше, чем длина первоначального массива A, потому что нулевой элемент для корректности дальнейших вычислений обязательно принимается равным нулю ($S_0 = 0$). [4]

Для простоты понимания можно представить, что элементы массива находятся в ячейках, а префиксные суммы находятся между ними — на перегородках – и содержат в себе суммы всего того, что находится левее этой перегородки. Именно поэтому для нулевого элемента сумма ноль – левее неё ничего нет.

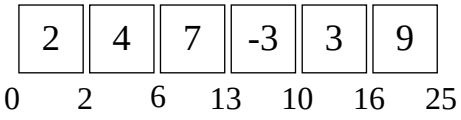


Рис. 1. Пример образования последовательности префиксных сумм.

Пример кода программы:

```
#include <iostream>
#include <stdlib.h>
#include <iomanip>
using namespace std;
int main()
{
	setlocale(LC_ALL, "RUSSIAN");
	int n;
	cout << "Введите длину массива: "; cin >> n;
	int *a,*s;
	a = new int[n];
	s = new int[n + 1];
	s[0] = 0;
	srand((unsigned)time(NULL));
	for (int i = 0; i < n; i++)
		{
		a[i] = rand() % 100;
		s[i + 1] = s[i] + a[i];
		cout << a[i] << setw(5);
		}
	cout << "\nМассив префиксных сумм:\n";
	for (int i = 0; i < n; i++)
		{
		cout << s[i] << setw(5);
		}
	return 0;
}
```

Для предварительной обработки массива входных данных можно применять не только операцию суммирования. Среди операций, подходящих для использования в данном алгоритме необходимым условием является только тот факт, что функция, которая считается на отрезке, обладает свойством обратимости, то есть для неё реализуема возможность по двум префиксам восстановить значение на отрезке. Потому простейшим примером и является суммирование, ведь операция суммы обратима, потому что, если было прибавлено лишнее, потом это можно отнять обратно. К таким операциям можно отнести сложение по модулю (обратным ему является вычитание так же, как и у простых сумм), а ещё умножение и умножение по модулю, для которых обратной операцией является деление.

Помимо вышеперечисленных, одна из часто употребляемых в алгоритмах операций – это операция «побитового исключающего или», которая еще называется XOR и обозначается $\oplus$. При этом для неё выполнено тождество x $\oplus$ x = 0 для любого числа x, что означает, что операция XOR обратна сама себе.

Существуют и другие варианты обработки массивов входных данных, помимо образования префиксных сумм, например, иногда удобнее насчитывать суффиксные суммы, которые представляю из себя суммы концов, а не начал массива. Также можно работать с массивами разниц – префиксными разностями, и разностными массивами, которые нетрудно получить из массива префиксных сумм [3].

Рассмотрим подробнее понятие разностного массива. Работа с ним представляет собой представленный ранее алгоритм, только развёрнутый наоборот. То есть по массиву префиксных сумм S: $[S_0, S_1, S_2, \ldots, S_n]$ можно восстановить исходный A: $[a_0, a_1, a_2, \ldots, a_{n-1}]$, который при этом будет являться *разностным массивом* для S и определяться по формуле (2):

$$a_{n-1} = S_n - S_{n-1} \quad (2)$$

Формула (2) получается путём несложного преобразованная рекуррентной формулы (1) для поиска префиксных сумм. Проводя аналогию с математикой, можно сделать вывод о том, что переход к разностному массиву — это дискретное дифференцирование, а переход к массиву префиксных сумм — дискретное интегрирование.

Стоит различать разностные массивы и массивы разниц (или массивы разностей), ведь последние применяют несколько иначе.

Пример 2. Массив разниц.

Пусть дан массив входных данных A: $[a_0, a_1, a_2, \ldots, a_{n-1}]$. Массив разниц D: $[d_0, d_1, d_2, \ldots, d_n]$ образуется на его основе путем вычитания из текущего значения предыдущего, причём первое значение такого массива $D_0 = a_0$, поскольку подразумевает разницу между «нулевым» и первым значением, а последнее значение $D_n = 0$. Формула для нахождения элемента массива разниц имеет вид:

$$D_i = a_i - a_{i-1} \quad (3)$$

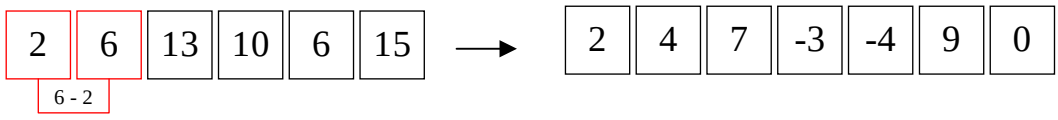


Рис. 2. Образование массива разниц.

Важным для практического применения свойством алгоритма является возможность применения его не только на одномерных массивах данных, но и на многомерных.

Пример 3. Создание многомерного массива префиксных сумм.

Пусть имеется двумерный массив A. Тогда массив его префиксных сумм B можно найти, подсчитав сперва префиксные суммы на одномерных массивах, то есть провести сложение сперва по горизонталям, а после – сложив полученные суммы по вертикалям.

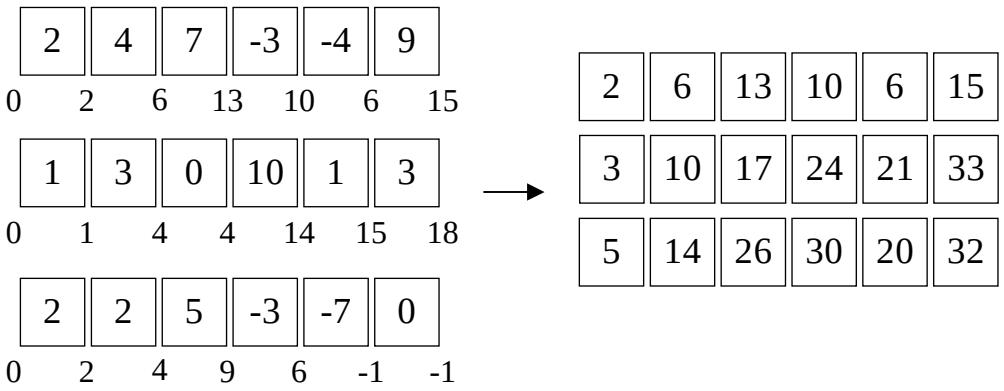


Рис. 3. Образование двумерного массива префиксных сумм.

Пример кода программы:

```
#include <iostream>
#include <stdlib.h>
#include <iomanip>
using namespace std;

int main()
{
	setlocale(LC_ALL, "RUSSIAN");
	srand((unsigned)time(NULL));
	int **a, **s, m, n, k = 0,sum_g=0,sum_v=0;
	cout << "Введите размеры массива MxN: "; cin >> m >> n;
	a = new int*[m];
	s = new int*[m+1];
	for (int i = 0; i < m; i++) a[i] = new int[n];
	for (int i = 0; i < m+1; i++) s[i] = new int[n+1];
```

```
	for (int i = 0; i < n+1; i++) s[0][i] = 0;
	for (int j = 0; j < m+1; j++) s[j][0] = 0;

	for (int i = 0; i < n; i++)
	{
		sum_g = 0;
		for (int j = 0; j < m; j++)
		{
			a[j][i] = rand() % 10;
			cout << setw(3) << a[j][i];
			sum_g += a[j][i];
			s[j+1][i+1] = sum_g+s[j+1][i];
		}
		cout << "\n";
	}
	cout << "\nМассив префиксных сумм данного массива:\n";

	for (int i = 0; i < n + 1; i++)
	{
		for (int j = 0; j < m + 1; j++)
		{ cout << setw(3) << s[j][i];}
		cout << "\n"; }
 return 0;
}
```

3 Возможности применения алгоритма

Префиксные суммы удобно брать за основу алгоритма в таких прикладных задачах программирования, когда, например, нужно найти сумму на полуинтервале или отрезке с позиции *l* до позиции *r*. У префиксных сумм есть замечательное свойство (4):

$$a_l + a_{l+1} + \cdots + a_{r-1} = S_r - S_l \quad (4)$$

Таким образом, в префиксной сумме S_r содержатся все интересующие нас элементы полуинтервала от a_l до a_r, однако вместе с ними в такую сумму неизбежно попадают и «посторонние» элементы, оставшиеся при расчётах от начала массива – это элементы a_0, a_1, …, a_{l-1}. Сумма эти элементов в свою очередь равна уже посчитанной префиксной сумме S_l. Тогда для ответа на вопрос задачи о поиске суммы на произвольном полуинтервале нужно просто найти разность между суммами с индексами, равными концам полуинтервала ($S_r - S_l$).

Ответить на вопрос о том, почему использовать полуинтервалы предпочтительнее поможет конкретный пример. Рассмотрим следующий случай: дан массив A [1, 2, 3, 4], для него есть массив префиксных сумм S [0, 1, 3, 6, 10], и запрос о сумме на отрезке [1,3].

Сумма на этом отрезке может быть определена как сумма на полуинтервале [1,4) и равна разности $S_4 - S_1 = 6 - 0 = 6$. Если бы мы взяли$S_3 - S_1 = 3$, то получили бы некорректный ответ. Таким образом, если оба конца произвольного отрезка вида $[l, r]$ включены, то для получения правильного ответа в формулу приходится добавлять единицы ($S_r - S_{l-1}$), усложняющие как запись, так и понимание, к тому же случай, когда $l = 0$ приходится выделять отдельным условием, из чего можно сделать вывод о том, что полуинтервалы действительно практичней в использовании.

Аналогично это будет работать и с многомерными массивами. Например, после подсчёта двумерного массива префиксных сумм можно будет находить суммы по произвольным полупрямоугольникам (по аналогии с полуинтервалами):

$$b_{i+1,j+1} = b_{i,j+1} + b_{i+1,j} - b_{i,j} + a_{i,j} = \boldsymbol{y} + \boldsymbol{z} - \boldsymbol{x} + \boldsymbol{a_{i,j}} \quad (5)$$

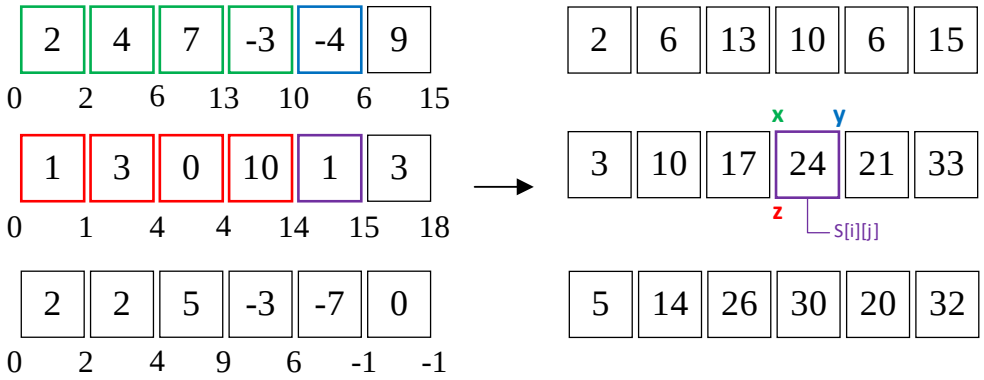


Рис. 4. Образование суммы на полупрямоугольнике.

Другой пример постановки вопроса задачи: необходимо найти любой непустой подотрезок с нулевой суммой элементов. Поскольку суммы на отрезках — это разности префиксных сумм, то если сумма на отрезке равна нулю, это равносильно тому, что префиксные суммы его концов равны.

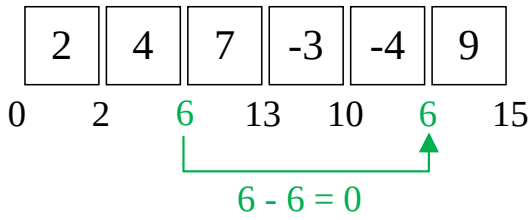


Рис. 5. Поиск подотрезка с нулевой суммой элементов.

Таким образом, задача сводится к нахождению подотрезка нулевой суммы, то есть к задаче нахождения двух одинаковых элементов в массиве S префиксных сумм. Использовать здесь префиксные суммы удобно потому, что получается, что суммы на отрезках — это разности префиксных сумм, если массив в ходе запросов не меняется. Потому что в обратном случае, если какой-

либо элемент из числа входных данных поменялся, нужно пересчитывать заново все префиксные суммы, в которые он входит, а это достаточно долго.

Массивы разниц рационально использовать для выполнения задач, где имеются запросы на обновление некоторого диапазона матрицы на значение x, то есть на добавление x к определенной подматрице. Изменив граничные значения диапазона на величину x в массиве разниц и пересчитав на его основе массив исходных данных заново, получим искомый обновлённый массив.

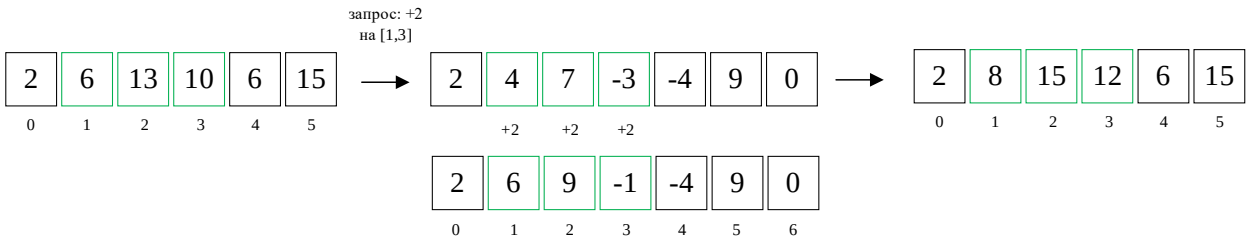


Рис. 6. Изменение входного массива с помощью массива разниц.

Среди прочих возможностей алгоритма стоит упомянуть, что разностные массивы используются в таких приложениях, как тестирование программного обеспечения, проверка подлинности и сжатие данных. [2]

4 Решение задач с использованием алгоритма

Задача 1. Таверна «Доля ангелов»

Где-то посреди одуванчиковых полей стоит, возвышаясь, обдуваемый ветрами, город свободы – Мондштадт. В нём испокон веков стоит таверна с интригующим названием, именуемая «Долей ангелов». Господин Рангвиндр – потомственный хозяин этой таверны. Он захотел узнать, как идут дела в заведении, и поручил своему помощнику в течение N дней записывать количество посетителей таверны. Это поможет узнать, сколько человек посещало «Долю ангелов» за определенные отрезки времени от *l* до *r*.

Помощник господина Рагнвиндра оказался человеком неглупым и придумал алгоритм, с помощью которого можно будет узнать число гостей быстро. Необходимо реализовать этот алгоритм в программе.

Входные данные.

На вход подаются целые положительных числа: на первой строке вводится N – число дней наблюдения ($1 \leq \mathrm{N} \leq 10^6$), на следующей строке через пробел подаётся N значений x_i – количество посетителей в *n*-ый день ($0 \leq x_i \leq 10^3$).

За ними следует число M – количество запросов статистики ($1 \leq \mathrm{M} \leq 10^6$), а после M строк со значениями y_i и z_i – границами дней сбора статистики ($1 \leq z_i \leq \mathrm{N}$, $1 \leq y_i \leq z_i$). В конце ввода Enter не ставится.

Выходные данные.

Число – ответ на каждый из запросов, т.е. M значений, представляющих собой количество посетителей за каждый из отрезков от y_i до z_i (оба конца отрезка включаются). Каждое значение – с новой строки.

Пример 1

На входе:

5

14 22 17 9 12

1

1 5

На выходе:

74

Решение.

Входные данные будет удобно записывать в массив длиной N, создадим его и назовём *array*. Тип данных имеет модификатор long long ввиду того, что вычисления сумм могут производиться с достаточно большими числами, и необходимо избежать переполнения. Для удобства дальнейшей работы с массивом *array* прямо в цикле ввода будет предподсчитываться массив префиксных сумм *pref*, причём размер данного массива будет на единицу больше, чем у исходного по причине того, что нулевой элемент массива префиксных сумм $pref_0 = 0$.

Ввод и вывод данных в программу осуществляется с помощью текстовых файлов input и output.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
	setlocale(LC_ALL, "RUSSIAN");
	ifstream input("input.txt");
	ofstream output("output.txt");
	if (!input.is_open()) {cout << "ERROR: FILE IS NOT OPEN";}
	else
	{	long int n, m, y, z, l, r;
		long long int* array, * pref;
		input >> n;
```

```
            array = new long long int[n];
            pref = new long long int[n + 1];
            pref[0] = 0;
            for (int i = 0; i < n; i++)
            {
                  input >> array[i];
                  pref[i + 1] = pref[i] + array[i];
            }

            input >> m;
            for (int i = 0; i < m; i++)
            {
                  input >> y >> z;
                  l = y-1;
                  r = z;
                  output << pref[r] - pref[l] << "\n";
            }
      }
      input.close();
      output.close();
      return 0;
}
```

Рассмотрим *группы тестов* для проверки решения данной задачи. Пусть сперва число запросов будет небольшим (меньше 10).

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

	1 7 2 7	
4	10 10 5 20 5 35 30	3 40 7 5

Теперь увеличим количество запросов для проверки эффективности алгоритма, а также попробуем ввести значительно большие числа. Для удобства представления результаты запросов записаны в строчку.

7 35 38 92 50 90 6 43 11 4 4 1 4 5 6 1 2 3 3 1 3 2 4 4 5 1 2 1 1 1 2	3 102344 544523 1000000 6 1 1 1 2 1 3 2 2 2 3 3 3	9 951 593 541 64 556 623 848 270 366 7 1 3 1 6 1 1 1 1 1 1 4 5 4 7

50, 215, 96, 73, 92, 165, 180, 140, 73, 35, 73	1102344,646867,1646867 544523,1544523,1000000	2085,3328,951,951, 951, 620, 2091

Описание групп тестов:

№1-3: Массивы данных небольшие, вводимые данные не превышают сотни, количество запросов не превышает 10. Проверены «крайние» случаи, то есть, когда имеется только один элемент в массиве, все значения одинаковы и когда значения идут в возрастающем порядке.
№4-6: Увеличен размер вводимых данных до 10^6, а также количество запросов увеличено и теперь не превышает 20.
№7-30: Алгоритм применяется при значительно большем количестве запросов, до 10^6, используются как большие размеры данных, так и маленькие. Поскольку представить в таблице все результаты такого большого числа запросов не представляется возможным, поэтому они будут представлены в отдельном файле.

Задача 2. Доходы и расходы

Один из старейших виноградных наделов в Мондштадте – это поместье господина Рагнвиндра. Винокурня «Рассвет» ведёт дела не только в пределах столицы, но и в других регионах: изысканные виноградные напитки, особые сочетания вкусов, смелые пробы ингредиентов – всё это, безусловно, приносит им добрую славу, но также напрямую влияет на доходы и расходы винодельни.

Владелец поместья имеет полезную привычку записывать все свои денежные сделки. Журнал доходов ведётся им в течение N дней и содержит записи о прибыли или расходах.

Господину Рагнвиндру нужно понять, за какой промежуток времени происходит выход «в ноль». Для этого требуется найти среди его записей такой подотрезок $[l, r]$, когда сумма трат и доходов становится равной нулю. Если

таких несколько, то вывести первый из них. Если таковых не имеется, вывести «EMPTY».

Входные данные.

На вход подаются целые числа: на первой строке вводится число дней журналирования N ($1 \leq N \leq 10^6$), далее с новой строки N значений x_i ($-10^6 \leq x_i \leq 10^6$), соответствующих доходу в денежных единицах за i-ый день, если число x_i положительно, или же расходу, если оно отрицательно. Отсчёт дней ведётся с $i = 1$ (первого дня). В конце ввода Enter не ставится.

Выходные данные.

Два числа, разделённые пробелом – границы первого найденного нулевого подотрезка, то есть такого, на котором сумма трат и доходов становится равной нулю. Первым найденным подходящим отрезком считается тот, у которого начало раньше, независимо от его длины.

Пример 1

На входе:

5

1 2 0 -2 5

На выходе:

2 4

Решение.

Для организации входных данных образуем массив a размером N. Для него насчитаем массив префиксных сумм s, с помощью которого мы и будем искать интересующий нас подотрезок. Размер массива s на единицу больше, чем у массива входных данных, потому что для корректности работы должно быть учтено $S_0 = 0$. Как было сказано ранее в теоретической части, суммы на отрезках — это разности префиксных сумм, то есть для ответа на вопрос задачи

необходимо найти два одинаковых элемента в массиве *s* и вывести их индексы.

Поскольку контекст задачи подразумевает довольно большие числа, для массива префиксных сумм был выбран тип данных long с модификатором long. Для того, чтобы вложенный цикл поиска двух одинаковых элементов в массиве префиксных сумм завершился правильно (то есть на первой найденной паре чисел), используется оператор break.

Алгоритм решения данной задачи представлен также в виде блок-схемы (Приложение А). Ввод и вывод данных в программу осуществляется с помощью текстовых файлов input и output.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
	setlocale(LC_ALL, "RUSSIAN");
	ifstream input("input.txt");
	ofstream output("output.txt");
	if (!input.is_open()) { cout << "ERROR: FILE IS NOT OPEN"; }
	else
	{
		int n, k = 0;
		input >> n;
		long long int *a, * s;
		int l, r;
		a = new long long int[n];
		s = new long long int[n + 1];
		s[0] = 0;
		for (int i = 0; i < n; i++)
		{
			input >> a[i];
			s[i + 1] = s[i] + a[i];
```

```
            }
            for (int j = 0; j < n + 1; j++)
            {
                  for (int i = j+1; i < n + 1; i++)
                  {
                        if (s[j] == s[i])
                        {
                              output << j+1 << " " << i << "\n";
                              k++;
                              break;
                        }
                  }
                  if (k != 0) break;
            }
            if (k == 0) output << "EMPTY";

      }
      input.close();
      output.close();
      return 0;
}
```

Рассмотрим *группы тестов* для проверки решения данной задачи. Сперва проверим алгоритм на небольшом количестве запросов.

1 1000000	10 53 53 53 53 53 53 53 53 53 53	10 2560 2440 3500 8002 -8001 -3501 5000 -4000 6000 -2000
EMPTY	EMPTY	3 6

Проверим случаи, при которых подходящие отрезки «перекрываются». Необходимо, чтобы выводился тот, у которого начало левее.

10	11	10

2560 2440 3500 8002 -8001 -3501 0 0 0 -2440	33642 33643 -61984 -55256 -3750 700 2078 -2778 67286 20061 33643	33642 33643 -61984 -55256 -3250 0 2078 -5556 67286 23039
2 10	1 10	2 10

Описание групп тестов:

№1-6: значения не превышают по модулю 10000, учтены случаи с несколькими перекрывающимися подотрезками, например, для теста №3 это [3, 6] и [8, 10], но выводится только первый;

для №4: перекрываются отрезки [2,10], [3,6], [3,7], [3,8] и [3,9], однако подходящим является именно [2,10];

для теста №5: перекрываются отрезки [2,10], [3,11], [6,8] но подходит только [2,10]; в тесте

№6: перекрываются отрезки [2,10] и [6,6], однако подходит только первый.

№7-25: вводимые числа существенно увеличены, однако ввиду большого количества данных они будут приведены в отдельном файле.

Задача 3. Кристальные бабочки

Недалеко от Мондштадта есть чудесное место, наполненное ароматом трав и теплом вечного лета – Одуванчиковое море. Конечно, поэтичное название «море», данное этому месту бардами, немного преувеличено. На самом же деле это живописная поляна, сплошь покрытая золотистыми цветами. Одуванчиковое море очень приглянулось редкому виду насекомых – кристальным бабочкам.

Поле представляет собой полосу из N условных клеток. На каждой клетке поля живёт определенное количество бабочек *x*. Бабочки не сидят на месте – они то прилетают, то улетают сразу по *y* штук с некоторых отрезков поля вида [*l, r*], где оба конца отрезка включены. Всего таких перелётов – M штук.

Требуется написать программу, которая определит, сколько останется бабочек на каждой из клеток после всех изменений. Считается, что за один запрос с участка от *l* до *r* улетает (или прилетает) одинаковое количество бабочек *y*.

Входные данные.

На вход подаются целые числа: на первой строке вводится число клеток на поле $1 \leq \mathrm{N} \leq 10^6$, далее с новой строки вводится N значений x_i ($0 \leq x_i \leq 10^4$), соответствующих количеству бабочек на клетке. На следующей строке вводится число запросов $1 \leq \mathrm{M} \leq 10^6$, а на последующих M строчках вводятся через пробел два числа: *l* – левая граница диапазона, *r* – правая граница ($1 \leq l \leq r$, а $1 \leq r \leq \mathrm{N}$) и y – численное изменение бабочек на каждой клетке ($-10^6 \leq \mathrm{y} \leq 10^6$). В конце ввода Enter не ставится.

Выходные данные.

Количество бабочек на каждой клетке поля после выполнения всех запросов, через пробел. Если бабочек совсем не осталось, вывести для этой клетки «0».

Пример 1

На входе:

3

34 42 88

1

0 2 2

На выходе:

36 44 90

Решение.

Для ответа на запросы о добавлении или вычитании числа на определенном диапазоне удобно использовать массив разниц, назовём его *diff* и после получения основного массива *a* пересчитаем разницу между имеющимися значениями по формуле (5):

$$diff_i = a_i - a_{i-1} \tag{5}$$

Далее воспользуемся свойством массива разниц и изменим в нём пограничные значения: левую границу $diff_l$ изменим на величину +x, а правую $diff_r$ на обратную величину –x. Таким образом выполним все M запросов, а после по массиву *diff* восстановим интересующий нас изменённый массив, использовав для вычисления *i*-го элемента массива *a* равенство (6), полученное из формулы (5):

$$a_i = diff_i + a_{i-1} \tag{6}$$

Алгоритм решения данной задачи представлен также в виде блок-схемы (Приложение Б). Ввод и вывод данных в программу осуществляется с помощью текстовых файлов input и output.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
	setlocale(LC_ALL, "RUSSIAN");
	ifstream input("input.txt");
	ofstream output("output.txt");
	if (!input.is_open()) { cout << "ERROR: FILE IS NOT OPEN"; }
	else
	{
		int n, k = 1, l, r, x, m;
```

```
        input >> n;
        long long int* a = new long long int[n];
        for (int i = 0; i < n; i++)
        {
                input >> a[i];
        }

        long long int* diff = new long long int[n + 1];
        diff[0] = a[0];
        diff[n] = 0;
        for (int i = 1; i < n; i++)
        {
                diff[i] = a[k] - a[k - 1];
                k++;
        }

        input >> m;
        for (int j = 0; j < m; j++)
        {
                input >> l >> r >> x;
                diff[l - 1] += x;
                diff[r] -= x;
                for (int i = 0; i < n; i++)
                {
                        if (i == 0) a[i] = diff[i];
                        else a[i] = diff[i] + a[i - 1];
                }
        }
        for (int i = 0; i < n; i++)
        {
                if (a[i] >= 0) output << a[i] << " ";
                else output << 0 << " ";
        }

}
input.close();
output.close();
```

```
    return 0;
}
```

Рассмотрим *группы тестов* для проверки решения данной задачи. Сперва проверим алгоритм на небольшом количестве запросов.

1 19 1 1 1 2	11 8 8 8 8 8 8 8 8 8 8 8 2 1 5 -2 1 5 -3	7 88 84 56 50 16 3 3 1 6 14 1 2 -10 2 3 1
21	3 3 3 3 3 8 8 8 8 8 8	88 84 56 50 16 4 4

Теперь проверим работу алгоритма с большими числами, а также проверим случай, когда бабочек совсем не остаётся.

7 8008 10000 5236 8020 1346 9999 3452 3 1 6 1000 1 2 1000 2 3 1000	4 13270 52377 1000000 555009 4 1 4 10500 2 4 5 2 4 -200000 2 2 105900	8 3465 8755 2184 5777 1231 9939 212 3 4 1 8 1000 2 4 13 6 8 -400 2 2 1000
10008 13000 7236 9020 2346 10999 3452	23770 0 810505 365514	4465 10768 3197 6790 2231 10539 812 603

Описание групп тестов:

№1-6: количество запросов невелико, как и число клеток поля. Проверены

«крайние» случаи, а именно: для теста №1 имеется только один элемент в массиве;
для теста №2 все значения одинаковы;
для теста №3 значения идут в убывающем порядке.
№7-25: число запросов существенно больше и достигает 10^6, однако ввиду большого количества данных они будут приведены в отдельном файле.

Задача 4. Винокурня «Рассвет»

Господин Рагнвиндр давным-давно получил в наследство от отца винокурню, и вполне успешно справляется с ней – семейное дело процветает и приносит винные дары во все уголки Мондштадта. На винокурне имеется поле размерами MxN условных клеток, засаженное различными сортами винограда. Со временем виноградные лозы разрослись, ведь они такие же свободолюбивые, как и сама страна свободных ветров. Теперь на каждой условной клетке поля имеется различное количество виноградных лоз.

Владельцу винокурни стало однажды любопытно, насколько рационально работает его надел, и он сосчитал, сколько на каждой клетке поля имеется виноградных лоз. Известно, что чётное количество лоз в полтора раза плодороднее, нежели нечётное. Необходимо узнать, какая часть поля является самой большой и при этом самой плодородной, то есть вывести координаты двух её противоположных краёв.

Входные данные.

На вход подаются целые положительные числа: на первой строке вводятся размеры поля M, N, где M, N $\in [1; 10^3]$, далее M*N значений $x_{i,j}$ ($0 \leq x_{i,j} \leq 1000$), соответствующих количеству лоз на клетке с координатами i и j. В конце ввода Enter не ставится.

Выходные данные.

Четыре целых числа, разделенные пробелом – координаты краёв самого плодородного участка в следующем порядке: сначала координаты левого верхнего угла (x_1, y_1), затем координаты правого нижнего (x_2, y_2).

Пример 1

На входе:

2 3

1 1 0 0 0 0

На выходе:

1 1 2 1

Решение.

Поле размером MxN образует двумерный массив *array*, каждому элементу которого соответствует количество лоз на клетке. Если предпосчитать массив префиксных сумм *pref* для исходной матрицы, далее по нему будет удобно узнавать, является ли суммарное количество лоз на определенной площади поля чётным или нечётным – и ставить в соответствии этим площадям некоторое число k – коэффициент плодородности, равное количеству лоз n, умноженных на полтора, если n чётное, или умноженных на единицу (т.е. остаётся равным n). Таким образом в цикле можно перебрать все возможные площади и определить, какая из них наиболее полезна и плодородна.

Если большая часть клеток массива *array* ненулевые, может оказаться так, что всё поле целиком суммарно производительнее, чем отдельно взятые участки. Если же, напротив, нулевых клеток много, то отдельный участок, наоборот, будет плодороднее целого.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
```

```
	setlocale(LC_ALL, "RUSSIAN");
	ifstream input("input.txt");
	ofstream output("output.txt");
	if (!input.is_open()) { cout << "ERROR: FILE IS NOT OPEN"; }
	else
	{
		int** array, ** pref, m, n, c, count_null = 0, sum_g = 0, sum_v
= 0, max_s = 0, x1 = 1, y1 = 1, x2 = 1, y2 = 1, l, r;
		int b_x1=0, b_y1=0, b_x2=0, b_y2=0;
		input >> m >> n;
		array = new int* [m];
		pref = new int* [m + 1];
		for (int i = 0; i < m; i++) array[i] = new int[n];
		for (int i = 0; i < m + 1; i++) pref[i] = new int[n + 1];

		for (int i = 0; i < n + 1; i++) pref[0][i] = 0;
		for (int j = 0; j < m + 1; j++) pref[j][0] = 0;

		for (int i = 0; i < n; i++)
		{
			sum_g = 0;
			for (int j = 0; j < m; j++)
			{
				input >> array[j][i];
				sum_g += array[j][i];
				pref[j + 1][i + 1] = sum_g + pref[j + 1][i];
			}
		}
		while (x1 <= m)
		{
			y1 = 1;
			while (y1 <= n)
			{
				if (array[x1 - 1][y1 - 1] == 0) { y1++;  continue; }
				while (x2 <= m)
				{
					y2 = 1;
```

```
                        while (y2 <= n)
                        {
                            if ((x1 > m) || (y1 > n) || (x2 > m) ||
(y2 > n)) break;
                            c = pref[x2][y2] - pref[x1 - 1][y2] -
pref[x2][y1 - 1] + pref[x1 - 1][y1 - 1];
                            if (c % 2 == 0) c = (int)(c * 1.5);
                            if (c > max_s)
                            {
                                max_s = c;
                                b_x1 = x1; b_y1 = y1; b_x2 = x2;
b_y2 = y2;
                            }
                            y2++;
                        }
                        x2++;
                    }
                    y1++;
                }
                x1++;
            }

            output << b_x1 << " " << b_y1 << " " << b_x2 << " " << b_y2;

        }
        input.close();
        output.close();
        return 0;
}
```

Рассмотрим *группы тестов* для проверки решения данной задачи.

Сначала количество нулевых и ненулевых клеток приблизительно одинаковое.

5 5 0 0 0 0 0 2 2 2 4 2 0 1 1 1 1 3 3 0 0 0 0 0 7 2 2	3 5 2 4 2 0 0 1 2 8 4 0 2 2 0 0 2	3 4 345 7 22 50 515 6 35 6 2 0 43 0
1 2 4 5	1 1 3 5	1 1 3 3

В следующей группе тестов количество нулевых клеток преобладает.

5 3 0 0 0 0 0 0 0 150 151 0 0 0 0 0 0	3 5 2 4 6 8 2 0 0 0 0 0 0 0 1 1 1	4 2 0 0 2 2 0 0 4 4
3 2 4 2	1 1 3 2	3 1 4 2

Все или почти все клетки заполнены числами больше нуля.

7 4 45 21 12 28 28 49 30 41 11 29 46 50 22 41 9 1 30 25 47 25 31 27 38 21 23 41 45 33	6 5 0 21 1 43 26 48 40 29 18 9 45 4 28 47 45 5 19 43 30 19 26 50 3 2 11 8 28 24 6 22	7 5 296 853 334 658 834 440 231 692 388 206 319 87 325 225 736 62 913 111 973 522 247 224 879 282 429 936 961 113 967 577 710 845 2 516 194
1 1 6 4	1 2 5 5	1 1 7 4

Описание групп тестов:

№1-3: количество нулевых и ненулевых клеток приблизительно одинаковое, результат охватывает большую часть поля. Используются числа не больше 500.
№4-6: количество нулевых клеток преобладает, результат является отдельным участком поля. Используются числа не больше 200.
№7-9: все клетки заполнены числами, результат охватывает большую часть поля. Используются числа до 10^3.
№10-25: вводимые числа существенно увеличены, однако ввиду большого количества данных они будут приведены в отдельном файле.

Заключение

В ходе выполнения данной курсовой работы был рассмотрен такой вид структуры данных, как префиксная сумма, упомянуты смежные с ним темы, приведены и разобраны примеры создания массивов префиксных сумм и их обработки. Были рассмотрены некоторые типовые варианты прикладных задач, для которых использование исследуемых структур наиболее целесообразно.

Также были разработаны условия для четырёх задач, необходимых для иллюстрации практического применения изученной структуры данных. К задачам приведено решение и группы тестов, алгоритм решения нескольких из них дополнительно определен блок-схемами, приведенными в приложении.

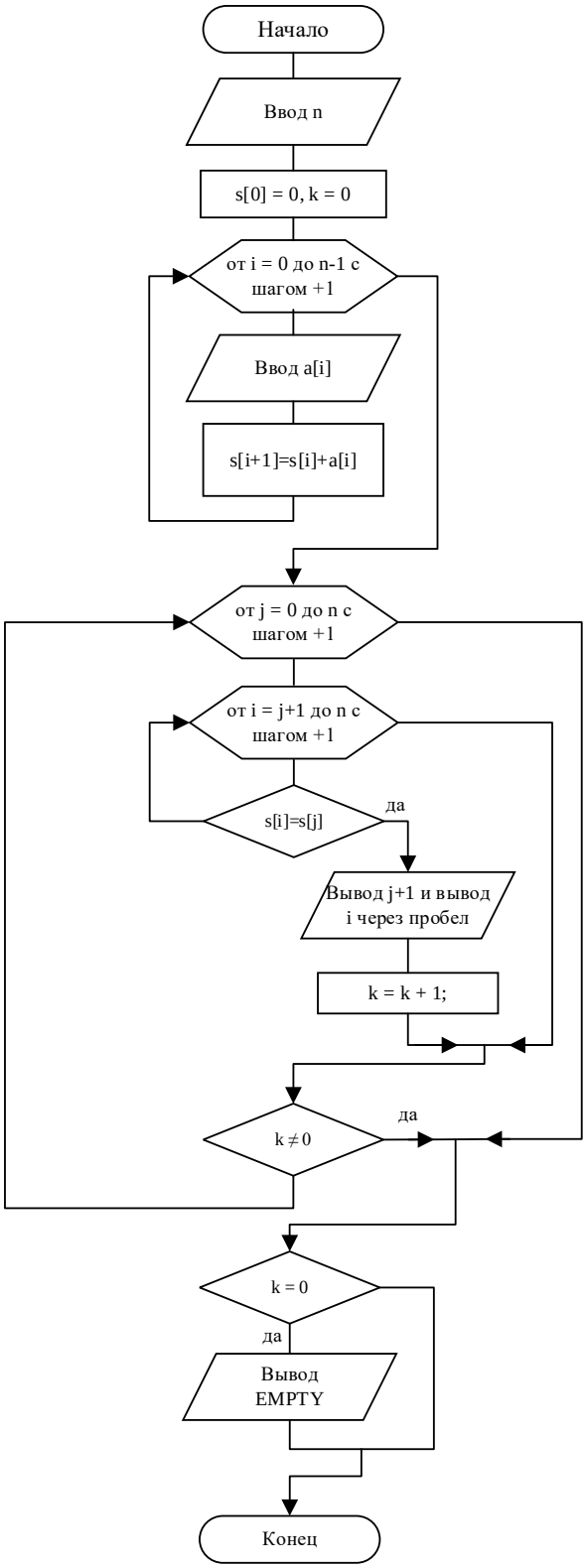
По итогам проведенной работы можно сделать вывод о том, что префиксные суммы, как и смежные с ними префиксные структуры данных, являются полезным и нужным инструментом для решения практических задач по обработке больших массивов данных.

Библиографический список

1. Peltorator - Префиксные суммы. Определения, основы и сила полуинтервалов [Электронный ресурс]. - URL: https://peltorator.ru/posts/prefix_sums/ (Дата обращения: 01.05.2022).
2. Демиркале Ф., Донован Д. М., Халл Д. Ф., Кходкар А., Рао А. В. Разностные покрывающие массивы и псевдо-ортогональные латинские квадраты: . - Vic 3000, Australia: School of Mathematical and Geospacial Sciences RMIT University, January 19. - 17 с.
3. Петр Калинин. Курс "Префиксные суммы и смежные темы" [Электронный ресурс]. - URL: https://notes.algoprog.ru/shortideas/03_x_prefix_sums.html (Дата обращения: 01.05.2022).
4. Prefix Sums and Their Applications Guy E. Blelloch // StudyLib.net URL: https://studylib.net/doc/14141067/1-prefix-sums-and-their-applications-guy-e.-blelloch (дата обращения: 01.05.2022).

Приложение А

Блок-схема алгоритма, представленного в задаче №2



Приложение Б

Блок-схема алгоритма, представленного в задаче №3