МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Лабораторная работа №1.1

по дисциплине: Дискретная математика
тема: «Операции над множествами»

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

Проверили:
ст. пр. Рязанов Юрий Дмитриевич
ст. пр. Бондаренко Татьяна Владимировна

Белгород 2023 г.

Лабораторная работа №1.1
Операции над множествами
Вариант 10

Цель работы: изучить и научиться использовать алгебру подмножеств, изучить различные способы представления множеств в памяти ЭВМ, научиться программно реализовывать операции над множествами и выражения в алгебре подмножеств.

1. Вычислить значение выражения (см. “Варианты заданий”, п. а). Во всех вариантах считать U = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}. Решение изобразить с помощью кругов Эйлера.

2. Записать выражение в алгебре подмножеств, значение которого при заданных множествах A, B и C равно множеству D (см. “Варианты заданий”, п. б).

3. Программно реализовать операции над множествами, используя следующие способы представления множества в памяти ЭВМ:

   а) элементы множества A хранятся в массиве A. Элементы массива A неупорядочены;

   б) элементы множества A хранятся в массиве A. Элементы массива A упорядочены по возрастанию;

   в) элементы множества A хранятся в массиве A, элементы которого типа boolean. Если i∈A, то $A_i = true$, иначе $A_i = false$.

4. Написать программы для вычисления значений выражений (см. “Задания”, п.1 и п.2).

5. Используя программы (см. “Задания”, п.4), вычислить значения выражений (см. “Задания”, п.1 и п.2).

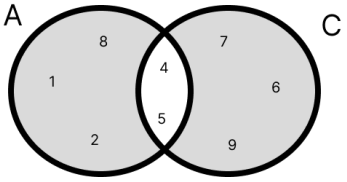
№1. Вычислить значение выражения (см. “Варианты заданий”, п. а). Во всех вариантах считать U = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}. Решение изобразить с помощью кругов Эйлера.

2 5 4 3 1

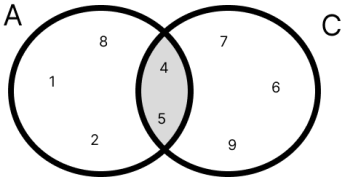
a) $D=A\cap C - B\cup B\cap(A\Delta C)$

$A=\{1,2,4,5,8\}$ $B=\{2,3,5,6,9\}$ $C=\{4,5,6,7,9\}$

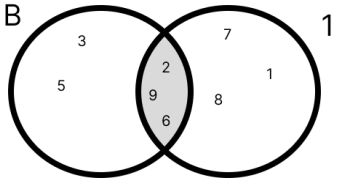
1. $A\Delta C = \{1,2,4,5,8\}\Delta\{4,5,6,7,9\} = \{1,2,6,7,8,9\}$



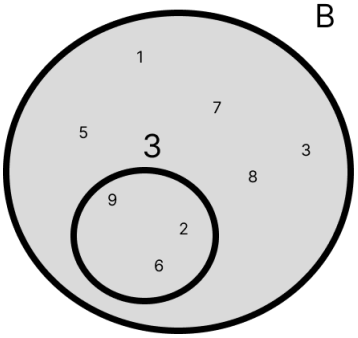
2. $A \cap C = \{1,2,4,5,8\} \cap \{4,5,6,7,9\} = \{4,5\}$



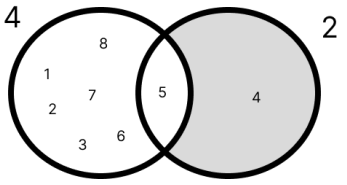
3. $B \cap 1 = \{2,3,5,6,9\} \cap \{1,2,6,7,8,9\} = \{2,6,9\}$



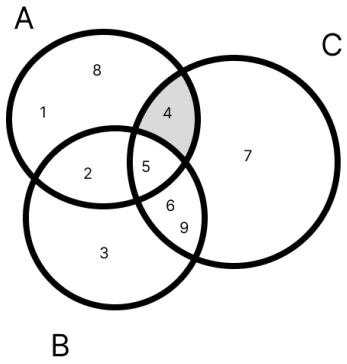
4. $B \cup 3 = \{2,3,5,6,9\} \cup \{2,6,9\} = \{2,3,5,6,9\}$



5. $2 - 4 = \{4,5\} - \{2,3,5,6,9\} = \{4\}$



$D = \{4\}$

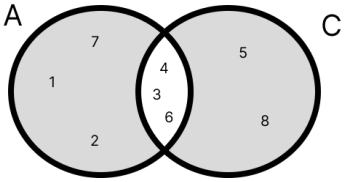


№2. Записать выражение в алгебре подмножеств, значение которого при заданных множествах A, B и C равно множеству D (см. “Варианты заданий”, п. б).

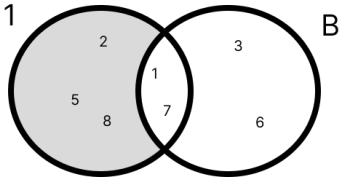
б) $A=\{1,2,3,4,6,7\}$ $B=\{1,3,6,7\}$ $C=\{3,4,5,6,8\}$
$D=\{2,5,8\}$

$$D = (A \overset{1}{\Delta} C) \overset{2}{-} B$$

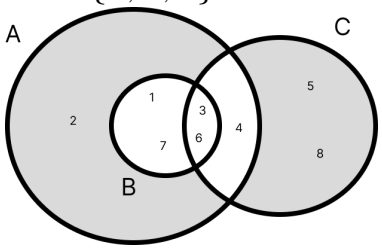
1. $A\Delta C = \{1,2,3,4,6,7\}\Delta\{3,4,5,6,8\} = \{1,2,5,7,8\}$



2. $1 - B = \{1,2,5,7,8\} - \{1,3,6,7\} = \{2,5,8\}$



$D = \{2,5,8\}$



№3. Программно реализовать операции над множествами, используя следующие способы представления множества в памяти ЭВМ:

а) элементы множества A хранятся в массиве A. Элементы массива A неупорядочены;

```
#include "../alg.h"

void uniteUnordered(const int *const arrayA, const size_t arrayASize,
```

```
const int *const arrayB, const size_t arrayBSize,
int *arrayC, size_t *const arrayCSize) {
        // Создаём указатель на первый элемент массива C
        int *cBegin = arrayC;

        // Копируем все элементы массива A в C, сдвигаем указатель на последний
↪  элемент.
        for (size_t i = 0; i < arrayASize; i++) {
                *(arrayC++) = arrayA[i];
        }

        // Аналогично копируем элементы из массива B в C, кроме того проверяем,
        // что копируемый элемент не встречается в массиве A
        for (size_t i = 0; i < arrayBSize; ++i) {

                // Проверяем, что элемента нет в массиве A
                int j = 0;
                while (j < arrayASize && arrayA[j] != arrayB[i])
                j++;

                // Если его нет, добавляем в массив C новый элемент
                if (j == arrayASize)
                *(arrayC++) = arrayB[i];
        }

        // Длина итогового массива - разница указателя на последний и первый элемент
        *arrayCSize = arrayC - cBegin;
}

void intersectUnordered(const int *const arrayA, const size_t arrayASize,
const int *const arrayB, const size_t arrayBSize,
int *arrayC, size_t *const arrayCSize) {
        // Создаём указатель на первый элемент массива C
        int *cBegin = arrayC;

        // Копируем элементы из массива B в C, кроме того проверяем,
        // что копируемый элемент встречается в массиве A
        for (size_t i = 0; i < arrayBSize; ++i) {

                // Проверяем, что элемента есть в массиве A
                int j = 0;
                while (j < arrayASize && arrayA[j] != arrayB[i])
                j++;

                // Если он есть, добавляем в массив новый элемент
                if (j != arrayASize)
                *(arrayC++) = arrayB[i];
        }

        // Длина итогового массива - разница указателя на последний и первый элемент
        *arrayCSize = arrayC - cBegin;
}

void differenceUnordered(const int *const arrayA, const size_t arrayASize,
```

```
const int *const arrayB, const size_t arrayBSize,
int *arrayC, size_t *const arrayCSize) {
        // Создаём указатель на первый элемент массива C
        int *cBegin = arrayC;

        // Копируем элементы из массива A в C, кроме того проверяем,
        // что копируемый элемент не встречается в массиве B
        for (size_t i = 0; i < arrayASize; ++i) {

                // Проверяем, что элемента нет в массиве B
                int j = 0;
                while (j < arrayBSize && arrayA[i] != arrayB[j])
                j++;

                // Если его нет, добавляем в массив C новый элемент
                if (j == arrayBSize)
                *(arrayC++) = arrayA[i];
        }

        // Длина итогового массива - разница указателя на последний и первый элемент
        *arrayCSize = arrayC - cBegin;
}

void symmetricDifferenceUnordered(const int *const arrayA, const size_t arrayASize,
const int *const arrayB, const size_t arrayBSize,
int *arrayC, size_t *const arrayCSize) {
        // Создаём указатель на первый элемент массива C
        int *cBegin = arrayC;

        // Копируем элементы из массива A в C, кроме того проверяем,
        // что копируемый элемент не встречается в массиве B
        for (size_t i = 0; i < arrayASize; ++i) {

                // Проверяем, что элемента нет в массиве B
                int j = 0;
                while (j < arrayBSize && arrayA[i] != arrayB[j])
                j++;

                // Если его нет, добавляем в массив C новый элемент
                if (j == arrayBSize)
                *(arrayC++) = arrayA[i];
        }

        // Аналогичным образом копируем элементы из B
        for (size_t i = 0; i < arrayBSize; ++i) {

                int j = 0;
                while (j < arrayASize && arrayB[i] != arrayA[j])
                j++;

                if (j == arrayASize)
                *(arrayC++) = arrayB[i];
        }
```

```
        // Длина итогового массива - разница указателя на последний и первый элемент
        *arrayCSize = arrayC - cBegin;
}

bool includesUnordered(const int *const arrayA, const size_t arrayASize,
const int *const arrayB, const size_t arrayBSize) {
        // Предположим, что B действительно содержит каждый элемент массива A
        bool result = true;

        // Проверим, что каждый элемент A находится в B, если обнаружится что это не
↪  так, то result будет false,
        // и смысла продолжать перебор далее не будет
        for (size_t i = 0; i < arrayASize && result; i++) {
                // Просто перебор
                size_t j = 0;
                while (j < arrayBSize && arrayA[i] != arrayB[j])
                j++;

                // Присваиваем result результат перебора, если что-то нашлось,
↪  result остаётся без изменений
                // Иначе - становится false.
                result = j != arrayBSize;
        }

        return result;
}

bool equalUnordered(const int *const arrayA, const size_t arrayASize,
const int *const arrayB, const size_t arrayBSize) {
        // Если массивы состоят из неповторяющихся одинаковых элементов, то логично
↪  предположить, что их
        // размеры равны.
        bool result = arrayASize == arrayBSize;

        // Проверим, что каждый элемент A находится в B, если обнаружится что это не
↪  так, то result будет false,
        // и смысла продолжать перебор далее не будет
        for (size_t i = 0; i < arrayASize && result; i++) {
                // Просто перебор
                size_t j = 0;
                while (j < arrayBSize && arrayA[i] != arrayB[j])
                j++;

                // Присваиваем result результат перебора, если что-то нашлось,
↪  result остаётся без изменений
                // Иначе - становится false.
                result = j != arrayBSize;
        }

        // Смысла проверять B нет, так как размеры массивов равны и каждому элементу
↪  A найден равный элемент из B

        return result;
}
```

```
void fillUniversumUnordered(const int *const arrayA, const size_t arrayASize,
                        const int *const universum, const size_t universumSize,
                        int *arrayC, size_t *const arrayCSize) {
    // Создаём указатель на первый элемент массива C
    int *cBegin = arrayC;

    // Здесь проверим, что universum действительно универсум
    for (size_t i = 0; i < arrayASize; ++i) {

        // Проверяем, что элемента из A нет в универсуме
        int j = 0;
        while (j < universumSize && arrayA[i] != universum[j])
            j++;

        // Если его нет в универсуме, падаем
        assert(j < universumSize);
    }

    // После проверки можем приступать к копированию из универсума в C
    for (size_t i = 0; i < universumSize; ++i) {

        // Находим элемент из универсума в A
        int j = 0;
        while (j < arrayASize && arrayA[j] != universum[i])
            j++;

        // Элемент из универсума не найден, поэтому копируем его в C
        if (j == arrayASize)
            *(arrayC++) = universum[i];
    }

    // Длина итогового массива - разница указателя на последний и первый элемент
    *arrayCSize = arrayC - cBegin;
}


bool includesStrictUnordered(const int *const arrayA, const size_t arrayASize,
const int *const arrayB, const size_t arrayBSize) {
        // Если массивы равны, то их размеры равны, а значит A не включен строго в B
        // Если массивы не равны, но их размеры равны, значит в A встречаются
↪   элементы, которых нет в B,
        // проверять дальше тоже нет смысла.
        bool result = arrayASize != arrayBSize;

        // Проверим, что каждый элемент A находится в B, если обнаружится что это не
↪   так, то result будет false,
        // и смысла продолжать перебор далее не будет
        for (size_t i = 0; i < arrayASize && result; i++) {
                // Просто перебор
                size_t j = 0;
                while (j < arrayBSize && arrayA[i] != arrayB[j])
                j++;
```

```
                // Присваиваем result результат перебора, если что-то нашлось,
↪  result остаётся без изменений
                // Иначе - становится false.
                result = j != arrayBSize;
        }

        return result;
}
```

б) элементы множества A хранятся в массиве A. Элементы массива A упорядочены по возрастанию;

```
#include "../alg.h"

void uniteOrdered(const int *const arrayA, const size_t arrayASize,
                  const int *const arrayB, const size_t arrayBSize,
                  int *arrayC, size_t *const arrayCSize) {
    // Индексы в массиве A и B
    size_t i = 0, j = 0;

    // Пока индексы не указывают на конец массива выполняем цикл
    while (i < arrayASize && j < arrayBSize)
        if (arrayA[i] < arrayB[j])
            // Первый случай. Копируем значение из A если индекс B указывает на
↪  конец массива или
            // элемент из A меньше элемента из B
            arrayC[(*arrayCSize)++] = arrayA[i++];
        else if (arrayA[i] > arrayB[j])
            // Второй случай. Копируем значение из B если индекс A указывает на
↪  конец массива или
            // элемент из B меньше элемента из A
            arrayC[(*arrayCSize)++] = arrayB[j++];
        else {
            // Третий случай. Элементы равны, сохраняем его в C и свдигаем оба
↪  индекса
            arrayC[(*arrayCSize)++] = arrayA[i];
            i++;
            j++;
        }

    // Копируем оставшиеся элементы из A если таковые есть
    while (i < arrayASize)
        arrayC[(*arrayCSize)++] = arrayA[i++];

    // Копируем оставшиеся элементы из B если таковые есть
    while (j < arrayBSize)
        arrayC[(*arrayCSize)++] = arrayB[j++];
}

void intersectOrdered(const int *const arrayA, const size_t arrayASize,
                      const int *const arrayB, const size_t arrayBSize,
                      int *arrayC, size_t *const arrayCSize) {
```

```
    // Индексы в массиве A и B
    size_t i = 0, j = 0;

    // Пока индексы не указывают на конец массива выполняем цикл
    while (i < arrayASize && j < arrayBSize)
        // Первый случай. A меньше элемента из B
        if (arrayA[i] < arrayB[j])
            i++;
            // Второй случай. A больше элемента из B
        else if (arrayA[i] > arrayB[j])
            j++;
            // Третий случай. A равен элементу из B. В третьем случае копируем
↪   элемент и сдвигаем оба индекса
        else {
            arrayC[(*arrayCSize)++] = arrayA[i];
            i++;
            j++;
        }
}

void differenceOrdered(const int *const arrayA, const size_t arrayASize,
                       const int *const arrayB, const size_t arrayBSize,
                       int *arrayC, size_t *const arrayCSize) {
    // Индексы в массиве A и B
    size_t i = 0, j = 0;

    // Пока индексы не указывают на конец массива выполняем цикл
    while (i < arrayASize && j < arrayBSize)
        if (arrayA[i] < arrayB[j])
            // Первый случай. Элемент из A оказался меньше элемента из B или индекс
↪   j указывает на конец B
            // Если A[i] < B[j], то в B больше никогда не встретится A[i], так как
↪   все последующие элементы
            // будут больше B[j], поэтому можем добавлять элемент в C.
            // Также добавляем элемент если мы достигли конца B - больше элементов
↪   не будет, и следующие элементы A
            // в нём не встретятся
            arrayC[(*arrayCSize)++] = arrayA[i++];
        else if (arrayA[i] > arrayB[j])
            // Второй случай. B[j] < A[i]. Здесь пока ничего не ясно, сдвигаем j
            j++;
        else {
            // Третий случай. Элементы равны, поэтому переходим к следующему
↪   элементу A увеличивая i и j
            i++;
            j++;
        }

    // Копируем оставшиеся элементы из A если таковые есть
    while (i < arrayASize)
        arrayC[(*arrayCSize)++] = arrayA[i++];
}

void symmetricDifferenceOrdered(const int *const arrayA, const size_t arrayASize,
```

```
                                const int *const arrayB, const size_t arrayBSize,
                                int *arrayC, size_t *const arrayCSize) {
    // Индексы в массиве A и B
    size_t i = 0, j = 0;

    // Пока индексы не указывают на конец массива выполняем цикл
    while (i < arrayASize && j < arrayBSize)
        if (arrayA[i] < arrayB[j])
            // Первый случай. Копируем значение из A если индекс B указывает на
↪   конец массива или
            // элемент из A меньше элемента из B
            arrayC[(*arrayCSize)++] = arrayA[i++];
        else if (arrayA[i] > arrayB[j])
            // Второй случай. Копируем значение из B если индекс A указывает на
↪   конец массива или
            // элемент из B меньше элемента из A
            arrayC[(*arrayCSize)++] = arrayB[j++];
        else {
            // Третий случай. Элементы равны, в этом случае нужно сдвинуть оба
↪   индекса
            j++;
            i++;
        }

    // Копируем оставшиеся элементы из A если таковые есть
    while (i < arrayASize)
        arrayC[(*arrayCSize)++] = arrayA[i++];

    // Копируем оставшиеся элементы из B если таковые есть
    while (j < arrayBSize)
        arrayC[(*arrayCSize)++] = arrayB[j++];
}

bool includesOrdered(const int *const arrayA, const size_t arrayASize,
                     const int *const arrayB, const size_t arrayBSize) {
    size_t i = 0, j = 0;
    bool result = arrayASize <= arrayBSize &&
                  arrayA[arrayASize - 1] <= arrayB[arrayBSize - 1];

    while (i < arrayASize && result)
        // Первый случай. A[i] < B[j]. Элемент в массиве не найден, присваиваем
↪   result значение false
        if (arrayA[i] < arrayB[j])
            result = false;
            // Второй случай. A[i] > B[j].  Продолжаем поиск
        else if (arrayA[i] > arrayB[j])
            j++;
        else {
            // Третий случай. Элементы равны, в этом случае нужно сдвигаем оба
↪   индекса
            i++;
            j++;
        }
```

```
    return result;
}

bool equalOrdered(const int *const arrayA, const size_t arrayASize,
                  const int *const arrayB, const size_t arrayBSize) {
    // Если упорядоченные массивы равны, логично предположить, что и размеры их тоже
↪   равны
    if (arrayASize != arrayBSize)
        return false;

    for (size_t i = 0; i < arrayASize; i++)
        if (arrayA[i] != arrayB[i])
            return false;

    return true;
}

void fillUniversumOrdered(const int *const array, const size_t arraySize,
                          const int *const universum, const size_t universumSize,
                          int *arrayC, size_t *const arrayCSize) {
    // Алгоритм схож с алгоритмом разницы множеств
    size_t i = 0, j = 0;
    // Проверяем, что универсум действительно универсум
    assert(array[arraySize - 1] <= universum[universumSize - 1]);

    while (i < universumSize && j < arraySize) {
        if (universum[i] < array[j])
            arrayC[(*arrayCSize)++] = universum[i++];
        else if (universum[i] == array[j]){
            i++;
            j++;
        // вторым его отличием будет то, что если элемент есть в A и его нет в
↪   universum, программа будет падать
        } else assert(array[j] >= universum[i]);
    }

    while (i < universumSize)
        arrayC[(*arrayCSize)++] = universum[i++];
}

bool includesStrictOrdered(const int *const arrayA, const size_t arrayASize,
                           const int *const arrayB, const size_t arrayBSize) {
    size_t i = 0, j = 0;
    // Отличием от нестрогого включения является то, что если A и включено в B, их
↪   размеры не должны быть равны
    bool result = arrayASize < arrayBSize &&
                  arrayA[arrayASize - 1] <= arrayB[arrayBSize - 1];

    while (i < arrayASize && result)
        if (arrayA[i] < arrayB[j])
            result = false;
        else if (arrayA[i] > arrayB[j])
            j++;
        else {
```

```
            i++;
            j++;
        }

    return result;
}
```

в) элементы множества A хранятся в массиве A, элементы которого типа boolean. Если i∈A, то $A_i = true$, иначе $A_i = false$.

```
#include "../alg.h"

void uniteBool(const bool *const arrayA,
                  const bool *const arrayB,
                  bool *arrayC, size_t arrayCapacity) {
    for (size_t i = 0; i < arrayCapacity; i++) {
        // Если элемент есть хотя бы в одном из массивов, добавляем его в итоговый
        arrayC[i] = arrayA[i] || arrayB[i];
    }
}

void intersectBool(const bool *const arrayA,
                     const bool *const arrayB,
                     bool *arrayC, size_t arrayCapacity) {
    for (size_t i = 0; i < arrayCapacity; i++) {
        // Если элемент есть в обоих массивах, добавлеяем его в итоговый
        arrayC[i] = arrayA[i] && arrayB[i];
    }
}

void differenceBool(const bool *const arrayA,
                      const bool *const arrayB,
                      bool *arrayC, size_t arrayCapacity) {
    for (size_t i = 0; i < arrayCapacity; i++) {
        // Добавляем элемент только если он есть в A и его нет в B. Получили формулу
↪   из таблицы истинности
        //  A   B   F
        //  0   0   0
        //  0   1   0
        //  1   0   1
        //  1   1   0
        //
        // F = A & (!B)
        arrayC[i] = arrayA[i] && (!arrayB[i]);
    }
}

void symmetricDifferenceBool(const bool *const arrayA,
                               const bool *const arrayB,
                               bool *arrayC, size_t arrayCapacity) {
    for (size_t i = 0; i < arrayCapacity; i++) {
        // Добавляем элемент если он есть только в A или только в B. Используем
↪   операцию xor
```

```
        arrayC[i] = arrayA[i] ^ arrayB[i];
    }
}

bool includesBool(const bool *const arrayA,
                  const bool *const arrayB, const size_t arrayCapacity) {

    for (size_t i = 0; i < arrayCapacity; i++) {
        // Если элемент есть в A, но его нет в B, значит A не включено в B,
↪   возвращаем false.
        if (arrayA[i] && !arrayB[i])
            return false;
    }

    return true;
}

bool equalBool(const bool *const arrayA,
               const bool *const arrayB, const size_t arrayCapacity) {
    for (size_t i = 0; i < arrayCapacity; i++) {
        // Тут всё просто, если элементы не равны - возвращаем false.
        if (arrayA[i] != arrayB[i])
            return false;
    }

    return true;
}

void fillUniversumBool(const bool *const arrayA, const bool *const universum, bool
↪   *const arrayB,
                       const size_t arrayCapacity) {
    for (size_t i = 0; i < arrayCapacity; i++) {
        // Если элемент есть в A, но его нет в универсуме, то произойдёт падение
↪   программы.
        assert(!arrayA[i] || universum[i]);
        // Инвертируем значения массива A и сохраняем в B. B - искомое множество.
↪   Также проверяем что A[i] есть в универсуме
        arrayB[i] = !arrayA[i] && universum[i];
    }
}

bool includesStrictBool(const bool *const arrayA,
                        const bool *const arrayB, const size_t arrayCapacity) {
    bool result = false;

    for (size_t i = 0; i < arrayCapacity; i++) {
        // Если элемент есть в A, но его нет в B, значит A не включено в B,
↪   возвращаем false.
        if (arrayA[i] && !arrayB[i])
            return false;

        // Если есть хотя бы один элемент, который есть в B, но его нет в A, можем
↪   утверждать, что множества
        // не равны
```

```
        result |= arrayB[i] && !arrayA[i];
    }

    return result;
}
```

№4. Написать программы для вычисления значений выражений (см. “Задания”, п.1 и п.2).

```
#include <printf.h>
#include "../libs/alg/alg.h"

#define MAX_ARRAY_SIZE 10000

int main() {
        printf("Counting Expression One using unordered sets\n");
        size_t arrayASize = 5, arrayBSize = 5, arrayCSize = 5;
        int arrayA[MAX_ARRAY_SIZE] = {5, 8, 4, 1, 2},
        arrayB[MAX_ARRAY_SIZE] = {6, 3, 9, 2, 5},
        arrayC[MAX_ARRAY_SIZE] = {4, 9, 7, 6, 5};

        // First operation
        size_t arrayFirstOperationSize = 0;
        int arrayFirstOperation[MAX_ARRAY_SIZE] = {};
        symmetricDifferenceUnordered(arrayA, arrayASize, arrayC, arrayCSize,
↪   arrayFirstOperation, &arrayFirstOperationSize);

        // Second operation
        size_t arraySecondOperationSize = 0;
        int arraySecondOperation[MAX_ARRAY_SIZE] = {};
        intersectUnordered(arrayA, arrayASize, arrayC, arrayCSize, arraySecondOperation,
↪   &arraySecondOperationSize);

        // Third operation
        size_t arrayThirdOperationSize = 0;
        int arrayThirdOperation[MAX_ARRAY_SIZE] = {};
        intersectUnordered(arrayB, arrayBSize, arrayFirstOperation,
↪   arrayFirstOperationSize, arrayThirdOperation,
        &arrayThirdOperationSize);

        // Fourth operation
        size_t arrayFourthOperationSize = 0;
        int arrayFourthOperation[MAX_ARRAY_SIZE] = {};
        uniteUnordered(arrayB, arrayBSize, arrayThirdOperation, arrayThirdOperationSize,
↪   arrayFourthOperation,
        &arrayFourthOperationSize);

        // Fifth operation
        size_t arrayDSize = 0;
        int arrayD[MAX_ARRAY_SIZE] = {};
        differenceUnordered(arraySecondOperation, arraySecondOperationSize,
↪   arrayFourthOperation, arrayFourthOperationSize,
```

```
        arrayD, &arrayDSize);

        printf("Answer: ");
        for (size_t i = 0; i < arrayDSize; i++)
        printf("%d ", arrayD[i]);

        printf("\n");
        printf("\n");


        printf("Counting Expression One using ordered sets\n");
        size_t arrayASize1O = 5, arrayBSize1O = 5, arrayCSize1O = 5;
        int arrayA1O[MAX_ARRAY_SIZE] = {1, 2, 4, 5, 8},
        arrayB1O[MAX_ARRAY_SIZE] = {2, 3, 5, 6, 9},
        arrayC1O[MAX_ARRAY_SIZE] = {4, 5, 6, 7, 9};

        // First operation
        size_t arrayFirstOperationSize1O = 0;
        int arrayFirstOperation1O[MAX_ARRAY_SIZE] = {};
        symmetricDifferenceOrdered(arrayA1O, arrayASize1O, arrayC1O, arrayCSize1O,
↪   arrayFirstOperation1O, &arrayFirstOperationSize1O);

        // Second operation
        size_t arraySecondOperationSize1O = 0;
        int arraySecondOperation1O[MAX_ARRAY_SIZE] = {};
        intersectOrdered(arrayA1O, arrayASize1O, arrayC1O, arrayCSize1O,
↪   arraySecondOperation1O, &arraySecondOperationSize1O);

        // Third operation
        size_t arrayThirdOperationSize1O = 0;
        int arrayThirdOperation1O[MAX_ARRAY_SIZE] = {};
        intersectOrdered(arrayB1O, arrayBSize1O, arrayFirstOperation1O,
↪   arrayFirstOperationSize1O, arrayThirdOperation1O,
        &arrayThirdOperationSize1O);

        // Fourth operation
        size_t arrayFourthOperationSize1O = 0;
        int arrayFourthOperation1O[MAX_ARRAY_SIZE] = {};
        uniteOrdered(arrayB1O, arrayBSize1O, arrayThirdOperation1O,
↪   arrayThirdOperationSize1O, arrayFourthOperation1O,
        &arrayFourthOperationSize1O);

        // Fifth operation
        size_t arrayDSize1O = 0;
        int arrayD1O[MAX_ARRAY_SIZE] = {};
        differenceOrdered(arraySecondOperation1O, arraySecondOperationSize1O,
↪   arrayFourthOperation1O, arrayFourthOperationSize1O,
        arrayD1O, &arrayDSize1O);

        printf("Answer: ");
        for (size_t i = 0; i < arrayDSize1O; i++)
        printf("%d ", arrayD1O[i]);

        printf("\n");
```

```
        printf("\n");

        printf("Counting Expression One using bool sets\n");
        bool arrayA1B[MAX_ARRAY_SIZE] = {false, true, true, false, true, true, false,
↪  false, true},
        arrayB1B[MAX_ARRAY_SIZE] = {false, false, true, true, false, true, true, false,
↪  false, true},
        arrayC1B[MAX_ARRAY_SIZE] = {false, false, false, false, true, true, false, true,
↪  false, true};

        // First operation
        bool arrayFirstOperation1B[MAX_ARRAY_SIZE] = {};
        symmetricDifferenceBool(arrayA1B, arrayC1B, arrayFirstOperation1B,
↪  MAX_ARRAY_SIZE);

        // Second operation
        bool arraySecondOperation1B[MAX_ARRAY_SIZE] = {};
        intersectBool(arrayA1B, arrayC1B, arraySecondOperation1B, MAX_ARRAY_SIZE);

        // Third operation
        bool arrayThirdOperation1B[MAX_ARRAY_SIZE] = {};
        intersectBool(arrayB1B, arrayFirstOperation1B, arrayThirdOperation1B,
↪  MAX_ARRAY_SIZE);

        // Fourth operation
        bool arrayFourthOperation1B[MAX_ARRAY_SIZE] = {};
        uniteBool(arrayB1B, arrayThirdOperation1B, arrayFourthOperation1B,
        MAX_ARRAY_SIZE);

        // Fifth operation
        bool arrayD1B[MAX_ARRAY_SIZE] = {};
        differenceBool(arraySecondOperation1B, arrayFourthOperation1B, arrayD1B,
↪  MAX_ARRAY_SIZE);

        printf("Answer: ");
        for (size_t i = 0; i < MAX_ARRAY_SIZE; i++)
        if (arrayD1B[i])
        printf("%zu ", i);

        printf("\n");
        printf("\n");

        printf("Counting Expression Two using unordered sets\n");
        size_t arrayASize2U = 6, arrayBSize2U = 4, arrayCSize2U = 5;
        int arrayA2U[MAX_ARRAY_SIZE] = {6, 2, 4, 7, 1, 3},
        arrayB2U[MAX_ARRAY_SIZE] = {7, 3, 6, 1},
        arrayC2U[MAX_ARRAY_SIZE] = {4, 3, 6, 5, 8};

        // First operation
        size_t arrayFirstOperationSize2U = 0;
        int arrayFirstOperation2U[MAX_ARRAY_SIZE] = {};
        symmetricDifferenceUnordered(arrayA2U, arrayASize2U, arrayC2U, arrayCSize2U,
↪  arrayFirstOperation2U, &arrayFirstOperationSize2U);
```

```
        // Second operation
        size_t arrayDSize2U = 0;
        int arrayD2U[MAX_ARRAY_SIZE] = {};
        differenceUnordered(arrayFirstOperation2U, arrayFirstOperationSize2U, arrayB2U,
↪  arrayBSize2U, arrayD2U, &arrayDSize2U);

        printf("Answer: ");
        for (size_t i = 0; i < arrayDSize2U; i++)
        printf("%d ", arrayD2U[i]);

        printf("\n");
        printf("\n");
        printf("Counting Expression Two using ordered sets\n");
        size_t arrayASize2O = 6, arrayBSize2O = 4, arrayCSize2O = 5;
        int arrayA2O[MAX_ARRAY_SIZE] = {1, 2, 3, 4, 6, 7},
        arrayB2O[MAX_ARRAY_SIZE] = {1, 3, 6, 7},
        arrayC2O[MAX_ARRAY_SIZE] = {3, 4, 5, 6, 8};

        // First operation
        size_t arrayFirstOperationSize2O = 0;
        int arrayFirstOperation2O[MAX_ARRAY_SIZE] = {};
        symmetricDifferenceOrdered(arrayA2O, arrayASize2O, arrayC2O, arrayCSize2O,
↪  arrayFirstOperation2O, &arrayFirstOperationSize2O);

        // Second operation
        size_t arrayDSize2O = 0;
        int arrayD2O[MAX_ARRAY_SIZE] = {};
        differenceOrdered(arrayFirstOperation2O, arrayFirstOperationSize2O, arrayB2O,
↪  arrayBSize2O, arrayD2O, &arrayDSize2O);

        printf("Answer: ");
        for (size_t i = 0; i < arrayDSize2O; i++)
        printf("%d ", arrayD2O[i]);

        printf("\n");
        printf("\n");
        printf("Counting Expression Two using bool sets\n");
        bool arrayA2B[MAX_ARRAY_SIZE] = {false, true, true, true, true, false, true,
↪  true},
        arrayB2B[MAX_ARRAY_SIZE] = {false, true, false, true, false, false, true, true},
        arrayC2B[MAX_ARRAY_SIZE] = {false, false, false, true, true, true, true, false,
↪  true};

        // First operation
        bool arrayFirstOperation2B[MAX_ARRAY_SIZE] = {};
        symmetricDifferenceBool(arrayA2B, arrayC2B, arrayFirstOperation2B,
↪  MAX_ARRAY_SIZE);

        // Second operation
        bool arrayD2B[MAX_ARRAY_SIZE] = {};
        differenceBool(arrayFirstOperation2B, arrayB2B, arrayD2B, MAX_ARRAY_SIZE);

        printf("Answer: ");
        for (size_t i = 0; i < MAX_ARRAY_SIZE; i++)
```

```
        if (arrayD2B[i])
        printf("%zu ", i);

        printf("\n");
        printf("\n");

        return 0;
}
```

№5. Используя программы (см. “Задания”, п.4), вычислить значения выражений (см. “Задания”, п.1 и п.2).

```
Counting Expression One using unordered sets
Answer: 4

Counting Expression One using ordered sets
Answer: 4

Counting Expression One using bool sets
Answer: 4

Counting Expression Two using unordered sets
Answer: 2 5 8

Counting Expression Two using ordered sets
Answer: 2 5 8

Counting Expression Two using bool sets
Answer: 2 5 8


Process finished with exit code 0
```

Вывод: в ходе лабораторной работы изучили и научились использовать алгебру подмножеств, изучили различные способы представления множеств в памяти ЭВМ, научились программно реализовывать операции над множествами и выражения в алгебре подмножеств.