Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

**«Уральский федеральный университет**

**имени первого Президента России Б.Н. Ельцина»**

**Институт фундаментального образования**

**Кафедра интеллектуальных информационных технологий**

**КУРСОВАЯ РАБОТА ПО ДИСЦИПЛИНЕ  
 «Алгоритмы и анализ сложности»**

Пояснительная записка

Руководитель А.А. Мокрушин

Студенты гр.ФО-260003 А.С. Шаповалов

П.Д. Черноскутова

Екатеринбург – 2018

**ОГЛАВЛЕНИЕ**

[1 ПОСТАНОВКА ЗАДАЧИ 3](#_Toc516687819)

[2 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 4](#_Toc516687820)

[2.1 Обзор алгоритмов 4](#_Toc516687821)

[2.2 Разработка алгоритма 5](#_Toc516687822)

[2.3 Анализ алгоритма 7](#_Toc516687823)

[3 ПРАКТИЧЕСКАЯ ЧАСТЬ 8](#_Toc516687824)

[3.1 Реализация 8](#_Toc516687825)

[3.2 Практический эксперимент 9](#_Toc516687826)

[4 ВЫВОДЫ 11](#_Toc516687827)

[5 СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ 12](#_Toc516687828)

[ПРИЛОЖЕНИЯ 13](#_Toc516687829)

# 1 ПОСТАНОВКА ЗАДАЧИ

**Разбиение множества целых чисел.**

Задано множество целых чисел, для простоты считаем, что все числа натуральные. Необходимо найти все подмножества этого множества, сумма элементов в которых равна некоторому наперёд заданному числу. Размер множества может быть очень большим (начиная с 10000 элементов), задаваемое число тоже может быть большим.

Пример:

Input : arr[] = {1, 2, 3, 4, 5} sum = 10

Output : {4, 3, 2, 1}   
 {5, 3, 2}  
 {5, 4, 1}

# 2 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

## 2.1 Обзор алгоритмов

Для выбора оптимального пути решения при выполнении задач иногда требуется перебирать большое количество комбинаций данных, что нагружает память компьютера. Для того, чтобы избежать выполнения лишней работы в случае, если подзадачи взаимозависимы, используется метод динамического программирования, предложенный американцем Р.Беллманом в 50-х годах.

Динамическое программирование заключается в определении оптимального решения n-мерной задачи, разделяя её на n отдельных этапов. Каждый из них является подзадачей по отношению к одной переменной.

Целесообразно применять динамическое программирование в тех случаях, когда подзадачи взаимосвязаны, то есть имеют общие модули. Алгоритмом предусмотрено решение каждой из подзадач один раз, и сохранение результатов выполняется в специальною таблицу. Это даёт возможность не вычислять ответ заново при встрече с аналогичной подзадачей.

Именно с помощью динамического программирования мы решим поставленную задачу.

Для этого надо выполнить 2 подзадачи.

1. Записать в булевскую таблицу solution[set.Length, sum + 1] значения true или false, где solution[i,j] означает, можно ли сформировать сумму j, используя set[i].
2. Рекурсивно вывести все подмножества.

## 2.2 Разработка алгоритма

Идея алгоритма заключается в следующем: строится таблица S[A.Length, sum+1] (A - начальное множество чисел), у которой количество строк равно количеству чисел в множестве, а количество столбцов равно заданной сумме + 1.

Чтобы заполнить таблицу S мы используем следующую формулу: S[i][j] = S[i-1][j] ∨ S[i-1][j-A[i]]. Но в начале S[i][0] = true для i=0..n. Эта инициализация удобна в следующем случае: допустим j=A[i]. Тогда это можно представить j как подмножество чисел, а именно путём выбора A[i], так S[i][j] имеет значение true. Вспомним наше отношение S[i][j] = S[i-1][j] ∨ S[i-1][j-A[i]]. Поскольку мы выбираем A[i] мы знаем, что S[i][j] = S[i-1][j-A[i]]. Из S[i][j] = true следует, что S[i-1][j-A[i]] = S[i-1][0] должно иметь значение true также. Таким образом, S[i][0] = true для i=0..n.

При заполнении строк, каждая последующая строка должна опираться на значения в предыдущей строке. Номер строки — это рассматриваемая позиция в исходном массиве. Номер столбца – это значение суммы элементов подмножества.

Пример: A[] = {1, 2, 3, 4} sum = 5

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i/j | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | T |  |  |  |  |  |
| 1 | T |  |  |  |  |  |
| 2 | T |  |  |  |  |  |
| 3 | T |  |  |  |  |  |

Таблица 2.2.1

После этого мы проходим по S[0...i][1...j] и заполняем оставшиеся ячейки, где i=A.Length-1, а j = sum+1. При заполнении последующих строк применяем правило: рассматриваем в предыдущей строке  
 1) Значение этого же столбца.  
 2) Значение в столбце с номером [текущий столбец - рассматриваемое значение в исходном массиве] (если такой номер существует). Если хотя бы в одной из рассматриваемых ячеек находится значение True, записываем в текущую ячейку True, иначе False.

В итоге получаем такую таблицу.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i/j | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | T | T | F | F | F | F |
| 1 | T | T | T | T | F | F |
| 2 | T | T | T | T | T | T |
| 3 | T | T | T | T | T | T |

Таблица 2.2.2

После заполнения таблицы необходимо пройти по этой таблице и вывести множество решений. Проход осуществляется по следующему правилу:

1) Проверяем, имеется ли хотя бы одно решение. Для этого проверяем значение в ячейке последней строки крайнего правого столбца, если там стоит значение True, то решение имеется, иначе не имеется

2) Рассматриваем значение в ячейке выше, если там стоит значение True, то перемещаясь в эту ячейку мы пойдём по одной ветви решения. Так же просматриваем значение в предыдущей строке в столбце с индексом [номер столбца - рассматриваемое значение в исходном массиве], и если там стоит значение True, то перемещаясь в эту ячейку мы пойдём по второму пути решения.

3) Так как на каждом шаге просмотра таблицы мы можем получить по два возможных пути решения, а нам необходимо найти все решения, то далее обход таблицы производим при помощи рекурсии

4) Условием выхода из рекурсии будет нахождение решения, либо, когда возникнет ситуация, что мы дошли до нулевой строки

5) На каждом шаге рекурсии мы имеем все элементы данной ветви решения и значение, которое равно заданной сумме минус сумма элементов, по которые образуют данную ветвь решения. Когда это значение становится равным нулю, то мы понимаем, что решение найдено и записываем список элементов данной ветви решения в список найденных решений.

## 2.3 Анализ алгоритма

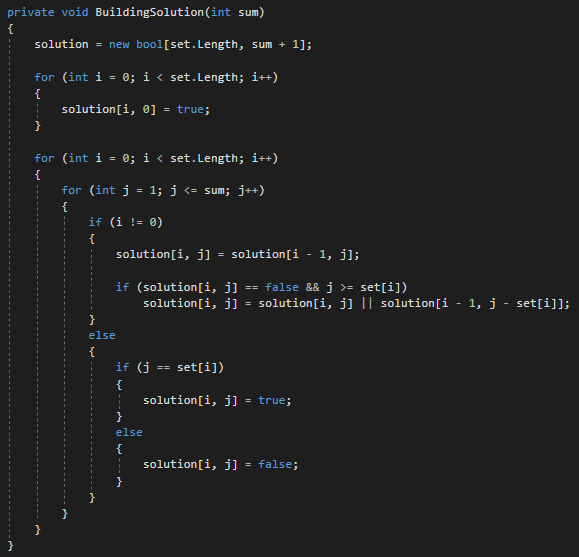
Теперь оценим сложность алгоритма: весь алгоритм сводится к тому, чтобы заполнить таблицу. Для этого имеется внешний цикл для прохода по заданному множеству. И внутренний цикл, который будет выполняться sum раз, тогда оба цикла дадут сложность О(set.Length \* sum).

Рисунок 2.3.1

# 3 ПРАКТИЧЕСКАЯ ЧАСТЬ

## 3.1 Реализация

При выборе языка программирования мы опирались на возможности этого языка и свой опыт программирования на нем. C# был выбран потому, что мы с ним хорошо знакомы. Также для удобства использования интерфейса программы, были использованы принципы Объектно-Ориентированного Программирования. А именно, был создан класс SubSetSum с полями и свойствами:

* public List<List<int>> Subsets { get; private set; } - итоговый список, который хранит списки с подмножествами.
* private bool[,] solution = null; - матрица, где будет рассматриваться 2 случая: можно ли сформировать сумму столбца j, используя A[i].

Также в классе SubSetSum есть 3 метода:

public void Search(int[] set, int sum) – метод начала поиска, где вызываются другие 2 метода, для решения задачи.(См. Приложение 1.)

private void BuildingSolution(int[] set, int sum) - метод для построения и заполнения таблицы.(См. Приложение 2.)

private void SubsetsRec(int[] set, int i, int sum, List<int> p) - метод, с помощью которого рекурсивно выводятся все подмножества.(См. Приложение 3.)

В главном классе MainWindow есть 5 основных методов, которые вызываются, когда нажата определённая кнопка на форме.

Кнопка “Поиск”, которая запускает алгоритм, ищет все подмножества и выводит их на экран. Кнопка “Автозаполение” для того, чтобы не вводить большое количество чисел вручную. И кнопка “Удалить все элементы” для того, чтобы очистить оба поля TextBox от данных. Кнопки “Очистить элементы множества” и “Очистить подмножества” для очистки соответствующих полей TextBox.  
Интерфейс программы представлен на рисунке 3.1.1.

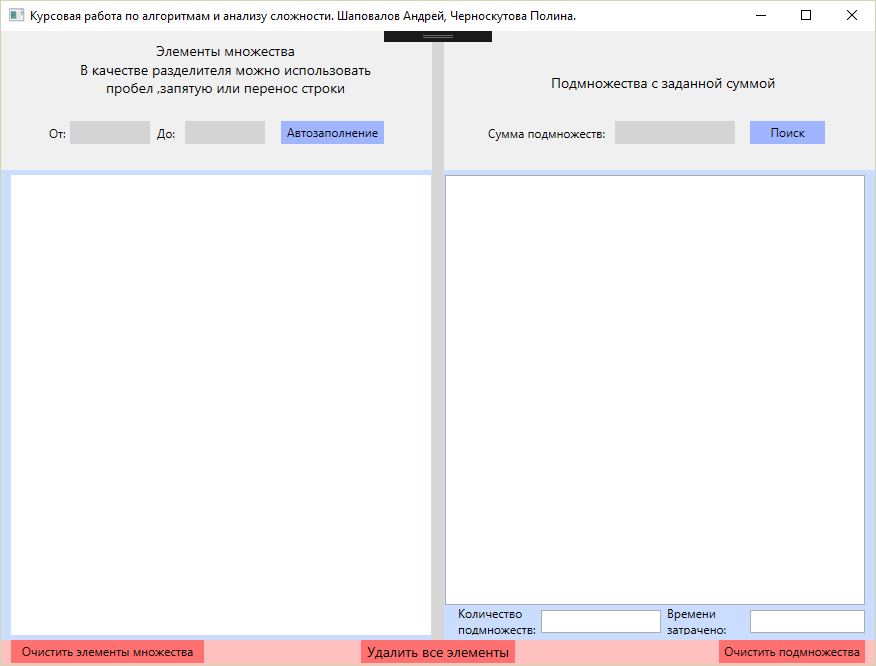


Рисунок 3.1.1

## 3.2 Практический эксперимент

Для измерения времени работы алгоритма практически, мы запускали нашу программу несколько раз на определённом количестве элементов. Каждое испытание повторялось 5 раз. Примерные результаты испытаний:

1) Для множества от 1 до 50, с суммой = 50, Среднее время: ~ 0,0001 секунды. Всего найдено 3658 подмножеств.

2) Для множества от 1 до 50, с суммой = 100, Среднее время: ~ 0,0412 секунды. Всего найдено 416 868 подмножеств.

3) Для множества от 1 до 50, с суммой = 120, Среднее время: ~ 1,0988 секунды. Всего найдено 1 924 094 подмножеств.

4) Для множества от 1 до 100, с суммой = 100, Среднее время: ~ 0,0579 секунды. Всего найдено 444 793 подмножеств

5) Для множества от 1 до 100, с суммой = 150, Среднее время: ~ 15,0028 секунды. Всего найдено 14 682 366 подмножеств

6) Для множества, состоящего из 3-х подмножеств от 1 до 20, с суммой = 50, Среднее время: ~ 55,0021 секунды. Всего найдено 29 786 886 подмножеств.

# 4 ВЫВОДЫ

В ходе выполнения курсовой работы были закреплены навыки нахождения эффективных алгоритмов для решения поставленных задач, а также анализа сложности данных алгоритмов.

В данной работе разработан эффективный алгоритм для нахождения всех подмножеств из заданного множества, сумма элементов которого равна заданному числу. Результатом работы алгоритма является список подмножеств.

Теоретическая оценка сложности алгоритма практически совпала с реальными показателями.

# 5 СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. https://cyberleninka.ru/article/n/obzor-algoritmov-resheniya-zadachi-o-nahozhdenii-summy-elementov-podmnozhestva
2. <https://habrahabr.ru/post/104219/>
3. https://ru.wikipedia.org/wiki/Динамическое\_программирование

# ПРИЛОЖЕНИЯ

Приложение 1.

public void Search(int[] set, int sum)

{

Subsets = new List<List<int>>();

this.set = set;

if (set.Length == 0 || sum < 0)

{

return;

}

BuildingSolution(sum);

if (solution[set.Length - 1, sum] == false)

{

return;

}

List<int> p = new List<int>();

SubsetsRec(set.Length - 1, sum, p);

}

Приложение 2.

private void BuildingSolution(int sum)

{

solution = new bool[set.Length, sum + 1];

for (int i = 0; i < set.Length; i++)

{

solution[i, 0] = true;

}

for (int i = 0; i < set.Length; i++)

{

for (int j = 1; j <= sum; j++)

{

if (i != 0)

{

solution[i, j] = solution[i - 1, j];

if (solution[i, j] == false && j >= set[i])

{

solution[i, j] = solution[i, j] || solution[i - 1, j - set[i]];

}

}

else

{

if (j == set[i])

{

solution[i, j] = true;

}

else

{

solution[i, j] = false;

}

}

}

}

}

Приложение 3.

private void SubsetsRec(int i, int sum, List<int> subset)

{

if (i == 0 && sum != 0 && solution[0, sum])

{

subset.Add(set[i]);

Subsets.Add(subset);

return;

}

if (i == 0 && sum == 0)

{

Subsets.Add(subset);

return;

}

if (i != 0 && solution[i - 1, sum])

{

List<int> recreate = subset.GetRange(0, subset.Count);

SubsetsRec(i - 1, sum, recreate);

}

if (i != 0 && sum >= set[i] && solution[i - 1, sum - set[i]])

{

subset.Add(set[i]);

SubsetsRec(i - 1, sum - set[i], subset);

}

}