ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 4

«NP-полные задачи. Задача о разделении множества»

Выполнила работу

Сторожева Евгения

Академическая группа №C3100

Санкт-Петербург

2024

1. Введение

Цель: изучить, как сложность алгоритмов для решения NP-полных задач влияет на эффективность их выполнения.

Задачи:

* 1. Изучить понятие NP-полных задач, их характеристики и примеры. Осознать, почему решение таких задач является сложной задачей и какие методы применяются для их оптимизации.
  2. Реализовать алгоритм, который разделяет массив на два подмножества с равной суммой.
  3. В процессе написания кода при помощи битовых масок осуществить комбинаторный перебор всех подмножеств
  4. Оценить время выполнения разработанного алгоритма в зависимости от объема входных данных. Сравнить результаты для различных наборов и выявить, как количество элементов влияет на время работы.
  5. Зафиксировать результаты экспериментов, сделать выводы о сложности алгоритмов и их применимости к реальным задачам.

1. Теоретическая подготовка
   1. NP-полная задача — в теории алгоритмов задача с ответом «да» или «нет» из класса NP, к которой можно свести любую другую задачу из этого класса за полиномиальное время (то есть при помощи операций, число которых не превышает некоторого полинома в зависимости от размера исходных данных).
   2. Используемые типы данных

Целые числа (int). Стандартный тип целого числа. Размер этого типа обычно составляет 4 байта. Диапазон предельных значений может варьироваться от −2 147 483 648 до 2 147 483 647.

Векторы (vector). Динамический массив, который может изменять свой размер. Вектор хранит элементы заданного типа в линейном расположении и обеспечивает быстрый случайный доступ к любому элементу. Размер вектора зависит от количества элементов, которые он содержит.

Переменная bool. Это ключевое слово является встроенным типом. Переменная этого типа может иметь значения true и false, при этом других значений быть не может. Имеет размер в 1 байт.

* 1. Алгоритм перебора подмножеств с помощью битовых масок.

Это эффективный способ генерации всех возможных подмножеств заданного множества. В этом алгоритме каждое подмножество представляется как набор битов, где каждый бит указывает, включен ли соответствующий элемент в текущее подмножество или нет.

Для каждого числа (маски) от 0 до 2^n - 1 проверяется каждый бит. Если бит на позиции j установлен (т.е. равен 1), то соответствующий элемент массива добавляется в текущее подмножество. Это делается с помощью побитовых операций. Например, для проверки, установлен ли j-й бит в числе mask, используется выражение (mask & (1 << j)). Его временная сложность: O(n \* 2^n).

* 1. Итеративный подход.

Это выполнение всех типов работ параллельно в ходе фиксированных коротких интервалов времени — итераций.

1. Реализация

Перед непосредственным написанием кода были изучены особенности представленной задачи, пример входа и выхода, условия, которые необходимо соблюсти.

Далее работа осуществлялась в несколько условных этапов.

На первом этапе были подключены библиотека <iostream> для использования стандартных потоков ввода и вывода и директива <vector>, предоставляющая множество удобных методов для работы с данными, директива «using namespace std;» для удобства.

Далее была написана часть кода, позволяющая определить, в каких случаях разделение на подмножества с равной суммой возможно, а в каких - нет. Для этого проверялась четность суммы элементов входного вектора. Использовался тип данных bool для сообщения о результате выполнения операции. Описываемая часть кода представлена на рисунке 1.

На втором этапе работы был введен цикл, осуществляющий комбинаторный перебор всех подмножеств. Мы инициализируем переменную mask с начальным значением 0. Она позволяет представить каждое подмножество можно как последовательность битов, где каждый бит указывает, включен ли элемент в подмножество или нет. В цикле сравнивается текущее значение переменной mask с результатом операции 1 <<n, чтобы убедиться не превышает ли значение mask количество всех возможных подмножеств, то есть не превышает ли2^n для n элементов. В конце каждой итерации значение mask увеличивается на 1, чтобы перейти к следующей комбинации. Данная часть кода представлена на рисунке 2.

На третьем этапе была написана часть кода для генерации подмножеств из массива чисел при помощи битовых масок, что представляет рисунок 3. Также была написана основная функция: в ней создается вектор arr, содержащий целые числа {1, 5, 11, 5} из примера входа; определяются два вектора subset1 и subset2 для хранения элементов, которые будут входить в каждое из подмножеств, если разделение возможно. Вызывается функция canPartition, проверяющая можно ли разделить массив на два подмножества с равной суммой. Если разделение возможно, функция заполняет subset1 и subset2 соответствующими элементами. Затем выводится результат. Функция main представлена на рисунке 4.

1. Экспериментальная часть

В этом разделе представлены результаты работы алгоритма с различными условиями и наборами данных. Результаты включают таблицы и графики, демонстрирующие выполнение алгоритма.

Подсчёт по памяти:

Для переменных (totalSum, num, target, n, subsetSum, mask, i): 7 \* 4 = 28 байт.

Для векторов: nums + subset1 + subset2 = n \* 4 + n \* 4 + n \* 4 = 3n \* 4 байт.

Общая память: (28 + 3n \* 4) байт

Подсчёт асимптотики:

O(n)+ O(1)+ O(2^n)+ O(n)= O(n \* 2^n),

Внешний цикл выполняется 2^n раз, а внутренний цикл — n раз.

График зависимости времени от числа элементов:

Согласно требованиям моего варианта, на вход к моему алгоритму подаётся до 25 элементов. Для тестирования алгоритма была собрана статистика, приведенная в таблице №1.

Таблица №1 - Подсчёт сложности реализованного алгоритма

|  |  |  |
| --- | --- | --- |
| Кол-во подмножеств(N) | Время выполнения (мс) | Ожидаемая сложность (O(2^N)) |
| 1 | 0.002438 | O(2) |
| 2 | 0.003641 | O(4) |
| 5 | 0.003952 | O(2^5) |
| 10 | 0.011545 | O(2^10) |
| 12 | 0.014565 | O(2^12) |
| 14 | 0.066725 | O(2^14) |
| 15 | 0.094725 | O(2^15) |
| 20 | 1.18206 | O(2^20) |
| 25 | 2.11085 | O(2^25) |

График представляющий визуально удобный формат данных из таблицы №1 представлен на изображении №1.

Изображение №1 - График работы алгоритма

Теоретическая сложность алгоритма О(n\*2^N). Реальная сложность алгоритма тоже О(n\*2^N) и в действительности он отрабатывает быстрее, поэтому 3 столбец и соответствующая линия графика, представляющие О(2n\*2^N), были убраны, так как в данном случае они непоказательные

Время выполнения алгоритма показывает экспоненциальный рост с увеличением N, что соответствует ожидаемой сложности O(2^N). Это подтверждается тем, что время выполнения значительно увеличивается даже при небольшом увеличении N.

Начиная с N = 14, алгоритм начинает демонстрировать заметное замедление, что может служить индикатором того, что для значений N выше этого порога алгоритм становится менее эффективным для практического использования.

1. Заключение

В ходе выполнения работы мною был реализован алгоритм разделения множества на два подмножества с равной суммой. Цель работы была достигнута путем изучения теории о свойствах NP-полных задач и методов их решения. Выполнение задач осуществлялось постепенно, в несколько этапов, благодаря чему получилось уделить внимание каждой части кода, существенно влияющей на выполнение алгоритма и конечный результат

При совпадении реальной и теоретической сложности, алгоритм работает быстрее, чем предполагается в теории.

Несмотря на свою простоту и наглядность, данный алгоритм имеет ограничения по производительности для больших наборов данных. Поэтому в качестве дальнейших исследований можно предложить использование более оптимизированных методов, таких как динамическое программирование или жадные алгоритмы, так как в реальных приложениях, где необходимо обрабатывать большие массивы, может потребоваться

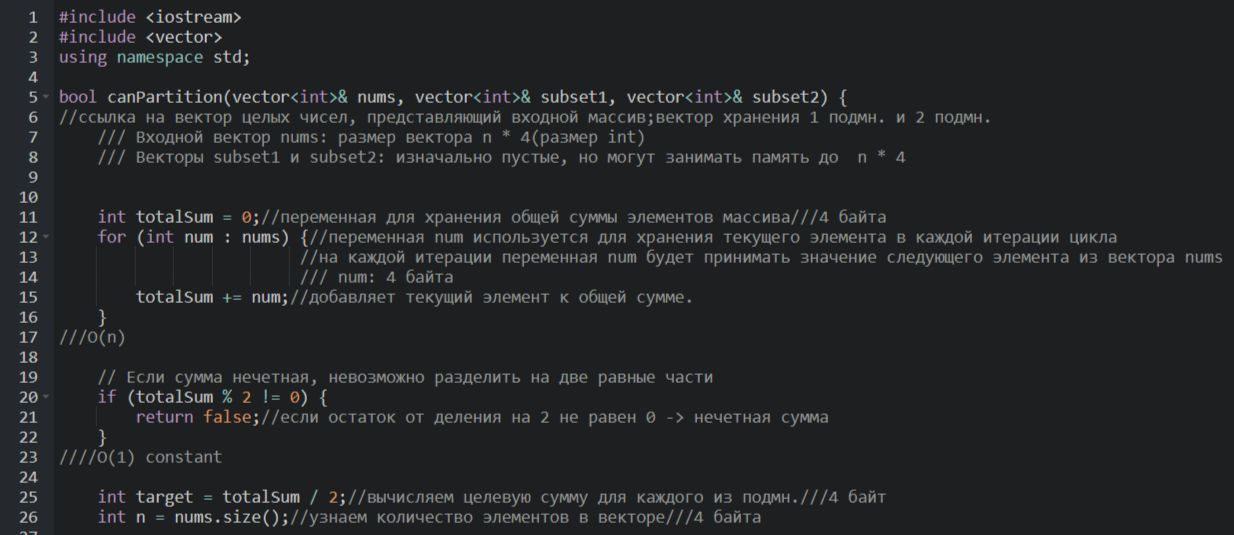
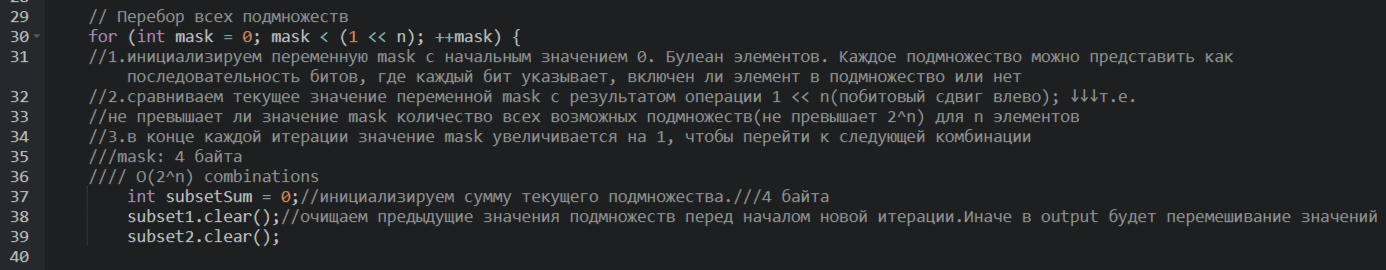
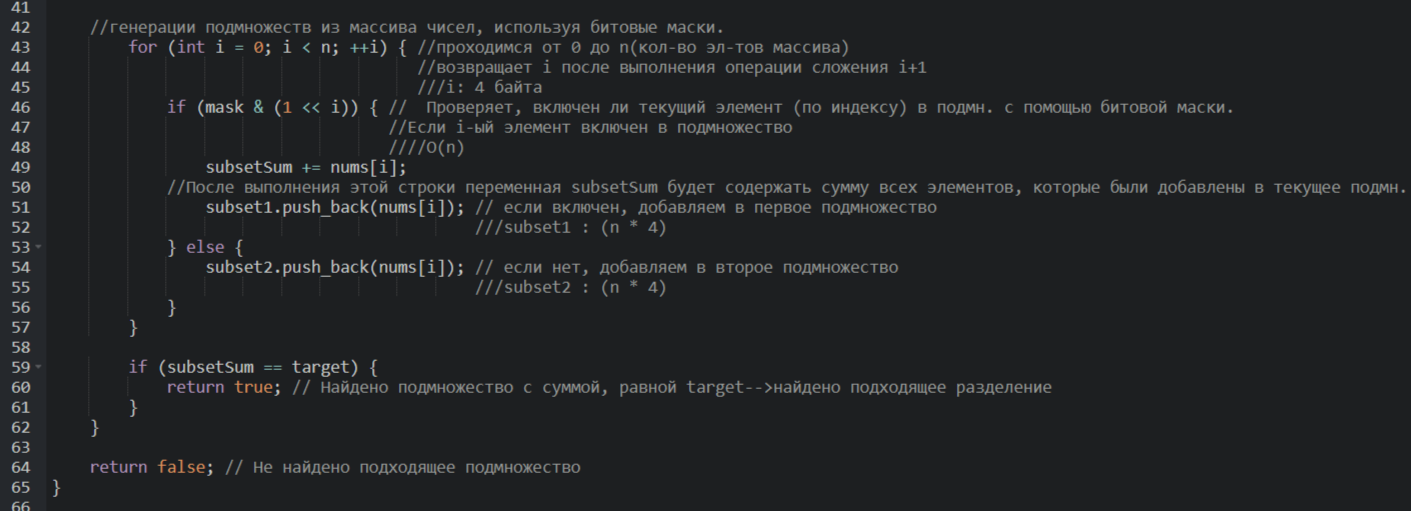
****ПРИЛОЖЕНИЕ А

Рисунок 2 – Первый этап реализации кода. Определение случаев, в которых возможно разделение на подмножества.

Рисунок 3 – Второй этап реализации кода. Комбинаторный перебор всех подмножеств при помощи битовых масок

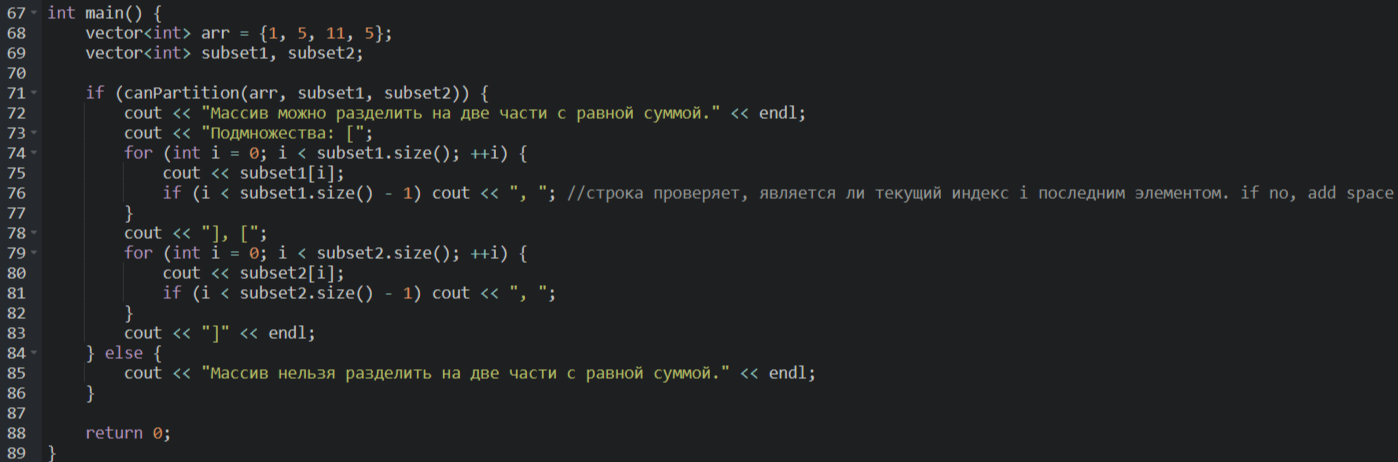
****Рисунок 4 – Третий этап реализации кода. Генерация подмножеств с использованием битовых масок

Рисунок 5 **–** Основная функция