**Министерство науки и высшего образования Российской Федерации**

**федеральное государственное бюджетное**

**образовательное учреждение высшего образования**

**«Российский экономический университет имени Г.В. Плеханова»**

Высшая школа кибертехнологий, математики и статистики

Базовая кафедра цифровой экономики института развития информационного общества

**КУРСОВАЯ РАБОТА**

по дисциплине «Структуры и алгоритмы компьютерной обработки данных»

на тему «Генерация комбинаторных объектов»

Выполнил обучающийся группы 15.11Д-МОСИП12/21б

очной формы обучения

Высшей школы кибертехнологий  
математики и статистики  
Степаниденко Д.Е.

Научный руководитель:

Комлева Нина Викторовна

Москва – 2023 г.

Оглавление

[ВВЕДЕНИЕ 4](#_Toc131123587)

[Глава 1. Теоретические основы генерации комбинаторных объектов 7](#_Toc131123588)

[**1.1 Постановка задачи 7**](#_Toc131123589)

[**1.2 Перестановки 7**](#_Toc131123590)

[**1.2.1 Алгоритм Штейнхауза-Джонсона-Троттера 9**](#_Toc131123591)

[**1.2.2 Алгоритм Хипа 11**](#_Toc131123592)

[**1.3 Разбиение чисел на слагаемые 14**](#_Toc131123593)

[**1.3.1 Генерации разбиения числа с помощью динамического программирования 16**](#_Toc131123594)

[**1.4 Разбиение числа n на m слагаемых 19**](#_Toc131123595)

[**1.4.1 Алгоритм Карла Фридриха Гинденбурга 19**](#_Toc131123596)

[Глава 2. Разработка программ 22](#_Toc131123597)

[**2.1 Разработка алгоритма Хипа 22**](#_Toc131123598)

[**2.2 Разработка алгоритма генерации всех разбиений числа с помощью динамического программирования 23**](#_Toc131123599)

[**2.3 Разработка алгоритма Карла Фридриха Гинденбурга 25**](#_Toc131123600)

[Глава 3. Машинный эксперимент 27](#_Toc131123601)

[**3.1 Тестирование алгоритма Хипа 27**](#_Toc131123602)

[**3.2 Тестирование алгоритма динамического программирования для генерации всех разбиений числа на слагаемые 29**](#_Toc131123603)

[**3.3 Тестирование алгоритма Карла Фридриха Гинденбурга 31**](#_Toc131123604)

[Заключение 32](#_Toc131123605)

[Список литературы 32](#_Toc131123606)

[Приложения 34](#_Toc131123607)

[**Приложение 1 34**](#_Toc131123608)

[**Алгоритм Карла Фридриха Гинденбурга 34**](#_Toc131123609)

[**Приложение 2 34**](#_Toc131123610)

[**Алгоритм Хипа 34**](#_Toc131123611)

[**Приложение 3 35**](#_Toc131123612)

[**Алгоритм генерации разбиения числа с помощью динамического программирования 35**](#_Toc131123613)

ВВЕДЕНИЕ

**Актуальность темы исследования**

Тема исследования генерации комбинаторных объектов является актуальной и важной в различных областях науки и техники, таких как математика, информатика, физика, биология и другие.

Комбинаторика - это наука, изучающая комбинаторные структуры и комбинаторные объекты, такие как перестановки, сочетания, разбиения и т.д. Генерация комбинаторных объектов - это процесс создания всех возможных комбинаций объектов, которые удовлетворяют определенным критериям.

Генерация комбинаторных объектов имеет множество практических применений. Например, в анализе данных генерация комбинаторных объектов может использоваться для создания всех возможных вариантов сценариев исхода событий, которые могут произойти в заданной ситуации. В криптографии генерация комбинаторных объектов может использоваться для создания всех возможных паролей, которые могут быть использованы для взлома защиты. В математике генерация комбинаторных объектов может быть полезна для создания новых теорем и формулирования гипотез.

Кроме того, генерация комбинаторных объектов является важной темой исследования в информатике и компьютерной науке. Эта тема имеет множество применений в областях, таких как алгоритмы оптимизации, искусственный интеллект, машинное обучение и т.д.

Например, алгоритмы генерации комбинаторных объектов могут использоваться для создания тестовых данных для тестирования программного обеспечения, для оптимизации производственных процессов в промышленности и для решения других задач в различных областях.

Таким образом, генерация комбинаторных объектов является важной темой исследования, которая имеет множество практических применений в различных областях науки и техники.

**Цель и задачи курсовой работы**

Целью курсовой работы является разработка, написание и тестирование программного средства для решения задач по генерации комбинаторных объектов.

Для реализации поставленной цели, необходимо решить следующие задачи:

1. Изучить различную литературу по данной теме;

2. Рассмотреть различные алгоритмы и их особенности;

3. Разработка, написание и тестирование программ, реализующих изученные алгоритмы.

**Объект исследования**

Структуры и алгоритмы компьютерной обработки данных

**Предмет исследования**

Использование различных алгоритмов и методов генерации комбинаторных объектов, их свойств, анализ эффективности их работы на различных входных данных.

**Методы исследования**

1. Анализ литературы - изучение научных статей, учебников и других публикаций, посвященных теме генерации комбинаторных объектов.
2. Экспериментальные методы - тестирование алгоритмов и методов генерации комбинаторных объектов на различных входных данных, анализ их эффективности и точности.
3. Математические методы - применение математических методов для изучения свойств комбинаторных объектов и алгоритмов их генерации.

**Структура работы**

Первая глава курсовой работы посвящена теоретической части. В ней подробно разобраны определения, связанные с комбинаторикой. Представлено постановка задачи, а также различные алгоритмы для её реализации.

Вторая глава рассказывает про процесс создания программы. Для полного понимания представлены фрагменты кода программы

В третьей главе рассматривается тестирование программы. Создаются реальные условия, в которых происходит тестирование. Также представлены скриншоты работы программы. Результат тестирование показывает, что программа работает успешно и справляется с заданными задачами.

**Глава 1. Теоретические основы генерации комбинаторных объектов**

* 1. **Постановка задачи**

Задача генерации комбинаторных объектов состоит в том, чтобы создавать наборы объектов в соответствии с заданными комбинаторными правилами.

Комбинаторные объекты могут быть различными, в зависимости от того, какие правила задаются. Некоторые из них включают перестановки, сочетания, разбиения чисел на слагаемые, графы, деревья, последовательности и многие другие.

При генерации комбинаторных объектов основной вопрос заключается в том, каким образом эти объекты должны быть сгенерированы. Существует множество методов для решения этой задачи, например, методы, основанные на рекурсии, алгоритмы перебора и комбинаторные методы.

Кроме того, важно учитывать, каким образом будут использоваться комбинаторные объекты, полученные при решении задачи. Например, в задачах оптимизации комбинаторные объекты могут использоваться для поиска оптимального решения, а в задачах криптографии - для создания криптографических ключей.

Одной из сложностей при решении задачи генерации комбинаторных объектов является вычислительная сложность. Некоторые комбинаторные объекты могут иметь очень большое число возможных комбинаций, что может существенно затруднить их генерацию и использование.

В своей курсовой работе я рассмотрю две важные задачи – генерация перестановок и разбиение чисел на слагаемые.

* 1. **Перестановки**

Перестановки - это комбинаторный объект, который описывает все возможные способы переставления элементов множества. Формально, перестановка - это упорядоченный набор из n элементов, в котором каждый элемент появляется ровно один раз.

Общее количество перестановок из n элементов вычисляется как n! (n факториал). Например, для множества из 3 элементов (например, {1, 2, 3}), есть 3! = 6 возможных перестановок: {1, 2, 3}, {1, 3, 2}, {2, 1, 3}, {2, 3, 1}, {3, 1, 2} и {3, 2, 1}.

Перестановки применяются в очень широком спектре областей, таких как:

1) Комбинаторика: перестановки являются одним из основных объектов комбинаторики и используются для решения широкого спектра задач, таких как распределение комитетов, размещение шахматных фигур на доске, выбор паролей и т.д.

2) Математический анализ: перестановки используются в теории вероятностей, для вычисления вероятности появления различных исходов, включая распределения случайных событий.

3) Криптография: перестановки используются для зашифровки и расшифровки сообщений в различных алгоритмах шифрования.

4) Алгоритмы и оптимизация: перестановки используются в задачах перебора, когда нужно перебрать все возможные варианты для выбора оптимального решения.

5) Информатика: перестановки используются для решения задач в области искусственного интеллекта, машинного обучения и анализа данных, например, для сортировки и фильтрации больших объемов данных.

Это лишь некоторые примеры областей, в которых перестановки находят свое применение. Как видно, перестановки являются важным объектом в математике и науке в целом, и их применение охватывает многие области науки и техники, поэтому очень важно изучить алгоритмы, с помощью которых можно генерировать перестановки.

Перейдём к рассмотрению самых популярных алгоритмов генерации перестановок.

* + 1. **Алгоритм Штейнхауза-Джонсона-Троттера**

Алгоритм Штейнхауза-Джонсона-Троттера - это алгоритм генерации перестановок без повторений. Он был разработан тремя математиками: Штейнхаузом, Джонсоном и Троттером в 1962 году. Основная идея алгоритма заключается в том, чтобы начать с самой маленькой перестановки (базовая перестановка) и последовательно добавлять элементы до тех пор, пока не будет сформирована полная перестановка. При этом каждый новый элемент добавляется с учетом порядка, заданного дополнительным параметром, который называется направлением.

Направление – указатель на соседний элемент (может указывать на элемент справа или слева). Элемент перестановки называется мобильным, если его направление указывает на меньший соседний элемент.

Рассмотрим описание работы данного алгоритма

1. Создаётся первая перестановка. Ряд чисел по возрастанию 1,2,…,n. Направление каждого элемента указывает влево.
2. Ищем наибольший мобильный элемент. Если не находим, то алгоритм закончен.
3. Производим обмен найденного мобильного элемента с элементом, на который указывает направление найденного мобильного элемента.
4. Меняем направление у всех элементов, которые больше чем найденный на шаге 2 элемент.
5. Переходим к шагу 2.

Преимущества данного алгоритма:

1. Относительная легкость не рекурсивной реализации
2. Малый расход памяти

Заметим, что данный алгоритм не является рекурсивным, а значит, не потребляет очень больших объёмов памяти, так как не нужно хранить все предыдущие перестановки, а только текущую, данное замечание как раз и является преимуществом этого алгоритма.

Поговорим об асимптотике данного алгоритма

Сложность по времени в наихудшем случае – O(n!)

Затраты памяти – O(n), где n – количество всех элементов в заданной последовательности.

Рассмотрим, как работает данный алгоритм на конкретном примере.

Допустим, нам нужно сгенерировать все перестановки длиной 3. Согласно алгоритму создаём базовый массив и массив направлений и возвращаем базовую перестановку 1,2,3. Ищем наибольший мобильный элемент – это число 3. Далее меняем в массиве этот элемент, с элементом, на который он указывает и меняем направление у всех элементов, которые больше чем найденный.

Получаем и – возвращаем 1,3,2 как текущую перестановку. Ищем наибольший мобильный элемент – это число 3. Далее меняем в массиве этот элемент, с элементом, на который он указывает и меняем направление у всех элементов, которые больше чем найденный.

Получаем и – возвращаем 3,1,2 как текущую перестановку. Ищем наибольший мобильный элемент – это число 2 (так как тройка не указывает ни на какой элемент). Далее меняем в массиве этот элемент, с элементом, на который он указывает и меняем направление у всех элементов, которые больше чем найденный.

Получаем и – возвращаем 3,2,1 как текущую перестановку. Ищем наибольший мобильный элемент – это число 3. Далее меняем в массиве этот элемент, с элементом, на который он указывает и меняем направление у всех элементов, которые больше чем найденный.

Получаем и – возвращаем 2,3,1 как текущую перестановку. Ищем наибольший мобильный элемент – это число 3. Далее меняем в массиве этот элемент, с элементом, на который он указывает и меняем направление у всех элементов, которые больше чем найденный.

Получаем и – возвращаем 2,1,3 как текущую перестановку. Но как мы видим теперь в нашем массиве вообще нет мобильного элемента – значит алгоритм завершается.

* + 1. **Алгоритм Хипа**

Следующий алгоритм для генерации перестановок, который мы рассмотрим это алгоритм Хипа, названный в честь B.R. Heap, который открыл его в 1963 году. Алгоритм минимизирует перемещение: он генерирует каждую перестановку из предыдущей путём замены одной пары элементов; остальные (n-2) элемента не нарушаются. В 1977 году в обзоре алгоритмов генерации перестановок Роберт Седжвик (американский специалист по информатике) пришёл к выводу, что на тот момент это был наиболее эффективный алгоритм для генерации перестановок с помощью компьютера.

Рассмотрим описание данного алгоритма

Для первого вызова функции используем последовательность (из элементов которой хотим получить перестановки) и значение k равное её длине.

1) Проверить значение k. В случае k == 1 -> вернуть последовательность как текущую перестановку. Закончить вызов функции.

2) Используя цикл от индекса первого элемента последовательности до k с шагом в 1 выполнить действия:

2.1) Рекурсивно вызвать функцию передав последовательность и значение (k-1).

2.2) Если k чётное выполнить обмен i и k элемента местами, в противном случае выполнить обмен первого элемента последовательности и k.

Рассмотрим асимптотику данного алгоритма.

Сложность по времени – O(n!).

Сложность по памяти – O(n) - для рекурсивного стекового пространства размером n, где n – размер изначально заданной последовательности.

Давайте рассмотрим пример работы алгоритма на конкретном примере.

Пусть у нас есть массив и мы хотим сгенерировать все перестановки элементов данного массива. Введём переменную k, которая изначально будет показывать длину массива, то есть . Автор предложил этот алгоритм, основываясь на индексации массива с 1. У нас переменная не равняется единице, значит вызываем для каждого элемента с начала и до эту функцию с параметром на единицу меньше. Для начала вызовем функция для элемента 1 с параметром . Получаем, что и . У нас переменная не равняется единице, значит вызываем для каждого элемента с начала и до эту функцию с параметром на единицу меньше. То есть вызываем для 1 элемента функцию с параметром . Переменная равняется одному, значит возвращаем данную перестановку как следующую – то есть 1,2,3. Далее функция возвращает управление на момент, где и . Далее сама идея Хипа, мы должны поменять элементы. У нас – чётное число, а значит меняем местами k-ый элемент и тот с которым только что работали – то есть первым элементом. То есть получаем и . Теперь вызываем функцию для второго элемента с параметром . То есть получаем и . Переменная равняется одному, значит возвращаем данную перестановку как следующую – то есть 2,1,3. Далее функция возвращает управление на момент, где и .

У нас k – чётное число, а значит меняем местами k-ый элемент и тот с которым только что работали, то есть со вторым элементом. Но k-ый элемент и так указывает на второй элемент, с которым мы только что работали, значит функция возвращает управление на момент, где и . В данном случае – нечётное число, следовательно делаем обмен первого элемента последовательности и k. То есть получаем и .

Теперь рекурсивно вызываем функцию, но уже от второго элемента с параметром . У нас переменная не равняется единице, значит вызываем для каждого элемента с начала и до эту функцию с параметром на единицу меньше. То есть вызываем для 1 элемента функцию с параметром . Переменная равняется одному, значит возвращаем данную перестановку как следующую – то есть 3,1,2.

Далее функция возвращает управление на момент, где и . У нас – чётное число, а значит меняем местами k-ый элемент и тот с которым только что работали – то есть первым элементом. То есть получаем и . Теперь вызываем функцию для второго элемента с параметром . Переменная равняется одному, значит возвращаем данную перестановку как следующую – то есть 1,3,2. Далее функция возвращает управление на момент, где и . У нас k – чётное число, а значит меняем местами k-ый элемент и тот с которым только что работали, то есть со вторым элементом. Но k-ый элемент и так указывает на второй элемент, с которым мы только что работали, значит функция возвращает управление на момент, где и . В данном случае – нечётное число, следовательно делаем обмен первого элемента последовательности и k. То есть получаем и . И теперь остаётся вызывать рекурсивно функцию для последнего элемента с параметром . У нас переменная не равняется единице, значит вызываем для каждого элемента с начала и до эту функцию с параметром на единицу меньше. То есть вызываем для 1 элемента функцию с параметром . Переменная равняется одному, значит возвращаем данную перестановку как следующую – то есть 2,3,1.

Далее функция возвращает управление на момент, где и . У нас – чётное число, а значит меняем местами k-ый элемент и тот с которым только что работали – то есть первым элементом. То есть получаем и . Теперь вызываем функцию для второго элемента с параметром . Переменная равняется одному, значит возвращаем данную перестановку как следующую – то есть 3,2,1. У нас k – чётное число, а значит меняем местами k-ый элемент и тот с которым только что работали, то есть со вторым элементом. Но k-ый элемент и так указывает на второй элемент, с которым мы только что работали, значит функция возвращает управление на момент, где и . Но на данном этапе мы прошли все элементы от 1 до k – значит алгоритм завершает свою работу.

* 1. **Разбиение чисел на слагаемые**

Задача о разбиении чисел на слагаемые заключается в том, чтобы найти все способы представления натурального числа n в виде суммы положительных целых чисел. Например, число 5 можно разбить на слагаемые несколькими способами: 5, 4+1, 3+2, 3+1+1, 2+2+1, 2+1+1+1, 1+1+1+1+1. При этом разбиения, отличающиеся только порядком частей, считаются равными.

Задача о разбиении чисел на слагаемые имеет множество исследовательских и практических задач, связанных с поиском оптимальных алгоритмов, изучением свойств различных видов разбиений и их применением в решении задач оптимизации и проектировании систем. В данной задачи важным является также изучение свойств различных видов разбиений, таких как упорядоченные, неупорядоченные, с повторениями, без повторений и т.д. Это позволяет рассматривать задачу о разбиении чисел на слагаемые с различных точек зрения и применять ее в различных областях.

Кроме того, задача о разбиении чисел на слагаемые имеет интересные математические свойства, которые могут быть использованы для исследования других комбинаторных задач. Например, теория разбиения чисел на слагаемые используется в теории представлений, теории модулей, теории функций Лиувилля и теории модулярных форм.

Одним из приложений задачи о разбиении чисел на слагаемые является проблема расписания. Например, если некоторая компания должна выполнить задачу, которая занимает n единиц времени, и у нее есть m работников с разной производительностью, то задача расписания может быть решена путем разбиения n на m слагаемых, где каждое слагаемое соответствует количеству времени, которое каждый работник должен потратить на выполнение задачи.

К тому же, задача о разбиении чисел на слагаемые может быть использована для моделирования ряда прикладных задач, например, для анализа производительности сетей связи, распределения производственных ресурсов и оптимизации процесса обработки данных.

Таким образом, задача о разбиении чисел на слагаемые имеет широкий спектр приложений и может быть использована для решения различных задач в различных областях, начиная от математики и заканчивая прикладными задачами в инженерии и экономике, поэтому её значение очень велико.

Но для начала поговорим про количество разбиений, если n стремится к бесконечности.

Асимптотическое выражение:

(1.1)

Данное выражение было получено Харди и Рамануджаном в 1918 г., а в 1920 г. независимо от них российским математиком Успенским.

Как видно из формулы при больших n количество разбиений становится очень большим.

Давайте рассмотрим алгоритм, который с помощью динамического программирования генерирует все разбиения.

* + 1. **Генерации разбиения числа с помощью динамического программирования**

Опишем сам алгоритм:

1. Создаём последовательность длиной n. Заполняем её единицами. Переходим к шагу 2.
2. Если длина последовательности равна 1, то заканчиваем алгоритм. В противном случае последовательность есть следующее разбиение. Используем его и переходим к шагу 3.
3. Находим ближайший к началу последовательности минимальный элемент (поиск проводим от начала до предпоследнего элемента). Увеличиваем его на единицу. Уменьшаем на единицу последний элемент последовательности. Вычисляем сумму элементов правее найденного минимального элемента. Создаём новую последовательность из элементов предыдущей последовательности, начиная с начала и до найденного минимального элемента и такого количества единиц, как найденная ранее сумма. Переходим в шагу 2.

Рассмотрим сложность алгоритма.

Сложность по времени в наихудшем случае – O(n).

Затраты памяти – O(n).

Где n – искомое число, разбиение которого мы хотим найти.

Но внимание – это на одно разбиение, а как мы выяснили ранее, количество разбиений возрастает очень быстро, поэтому на больших данных алгоритм будет работать очень долго.

Давайте посмотрим на то, как работает этот алгоритм на конкретном примере.

Пусть нам дано число . Согласно алгоритму мы создаём массив . Далее проверяем длину этого массива – она не равна одному, а значит это и есть текущее разбиение, то есть записываем 1,1,1,1,1 как наше первое разбиение.

Затем ищем минимальный элемент, начиная с начала массива и до предпоследнего элемента – это число 1. Увеличиваем этот элемент на 1, а последний уменьшаем на единицу. Получаем . Вычисляем сумму элементов правее минимального и до конца (будем обозначать далее её как S). В нашем случае . И создаём новый массив, в котором будут элементы из начало старого до минимального элемента и столько единиц, какова суммы . По итогу получаем . Это и есть наше следующее разбиение, то есть 2,1,1,1.

Ищем минимальный элемент в новом массиве – это опять число 1. Увеличиваем его на единицу, а также уменьшаем последний элемент на 1.

Получаем и . Теперь новый массив это . Это и есть наше следующее разбиение, то есть 2,2,1. Ищем минимальный элемент в новом массиве – это число 2 (так как смотрим от начального до предпоследнего элемента). Увеличиваем минимальный элемент на 1, а также последний уменьшаем на 1. Получаем и . Теперь новый массив это , а следовательно мы нашли ещё одно разбиение.

Ищем минимальный элемент в новом массиве – это число 1.Увеличиваем его на единицу, а также последний уменьшаем на единицу. Получаем и .

Теперь новый массив это , а также наше новое разбиение. Ищем минимальный элемент от начало до предпоследнего элемента – это число 3. Увеличиваем его на единицу, а также последний уменьшаем на 1. Получаем и .

Теперь новый массив это – это и есть наше следующее разбиение. Теперь ищем минимальный элемент от начала до предпоследнего элемента – это число 5. Увеличиваем его на 1, а также последний уменьшаем на 1. Получаем и .

Теперь новый массив – как мы видим, длина этого массива равняется единицы, а значит согласно алгоритму – мы нашли все разбиения на слагаемые числа 5.

* 1. **Разбиение числа n на m слагаемых**

Задача разбиения числа n на m слагаемых заключается в том, чтобы найти все возможные способы представления числа n в виде суммы m неотрицательных целых чисел.

Другими словами, нам нужно найти все упорядоченные наборы из m неотрицательных целых чисел, сумма которых равна n. Заметим, что порядок частей в разбиении не имеет значения, то есть разбиение 1+1+3 эквивалентно, например разбиению 3+1+1.

При этом должно выполняться условие, что n>=m>=2.

Например, разбиение числа 8 на 4 части можно представить так:

1. 8 = 5+1+1+1
2. 8 = 4+2+1+1
3. 8 = 3+3+1+1
4. 8 = 3+2+2+1
5. 8 = 2+2+2+2

Данная задача является фундаментальной в теории чисел, а также применяется во многих областях, таких как криптография, экономика, поэтому её рассмотрение столь важно.

Рассмотрим алгоритм, который реализует поставленную задачу.

* + 1. **Алгоритм Карла Фридриха Гинденбурга**

Рассмотрим алгоритм, который был предложен в 1779 г. немецким математиком К.Ф.Гиденбургом.

Посмотрим описание алгоритма:

1) Создаём последовательность длинной m. Устанавливаем , остальные элементы равные 1.Переходим к шагу 2.

2) Возвращаем последовательность в качестве разбиения. Выполнить проверку , если это условие истинно, то перейти к шагу 3, в противном случае перейти к шагу 4.

3) Установить . Перейти к шагу 2.

4) Начиная со второго элемента последовательности, ищем элемент удовлетворяющий условию . Если такого элемента не найдено, то заканчиваем алгоритм. Вычисляем сумму всех элементов до найденного элемента минус 1 (обозначим сумма как S). Увеличим значение найденного элемента на единицу. Перейти к шагу 5.

5) Выполняем проход от второго элемента до найденного элемента последовательности. На каждом шагу устанавливаем значение равное значению найденного элемента, при этом уменьшая значение S на эту величину. После завершения прохода, устанавливаем значение первого элемента последовательности как S. Перейти к шагу 2.

Рассмотрим сложность алгоритма.

Сложность по времени в наихудшем случае – O(m)

Но обратим внимание, что сложность указана для одного разбиения, а как мы выяснили ранее, количество разбиений при растёт экспоненциально, то есть очень быстро. Это означает, что на больших данных программа будет выполняться очень и очень долго.

Рассмотрим подробно, как работает алгоритм на конкретном примере.

Пусть нам нужно разбить число на части.

Согласно алгоритму нам нужно создать масcив из четырёх элементов, где , остальные элементы заполняем единичками. То есть – это и есть наше первое разбиение.

Далее проверяем условие – для нашего массива это выполнено (), значит устанавливаем и возвращаем полученное разбиение, то есть . Проверяем условие , для нашего массива это выполнено(), значит устанавливаем и возвращаем полученное разбиение, то есть .

Проверяем условие , но для нашего массива это уже не выполнено. Значит, начиная со второго элемента последовательности, ищем элемент удовлетворяющий условию – как мы можем видеть это число 1.Вычисляем сумму элементов (далее будем обозначать её как ) от начального до не включая его, и вычитаем из этой суммы один. Получаем, что . Увеличиваем на единицу и, начиная со второго элемента и до найденного, на каждом шагу устанавливаем значение равное значению найденного элемента, при этом уменьшая значение S на эту величину. После завершения прохода, устанавливаем значение первого элемента последовательности как . В нашем случае в конце получится и , поэтому первый элемент так и остаётся равным трём. И мы как раз получили наше следующее разбиение – это 3,2,2,1. Проверяем условие , но для нашего нового массива это не выполняется. Значит, начиная со второго элемента последовательности, ищем элемент удовлетворяющий условию – как мы можем видеть это число 1.

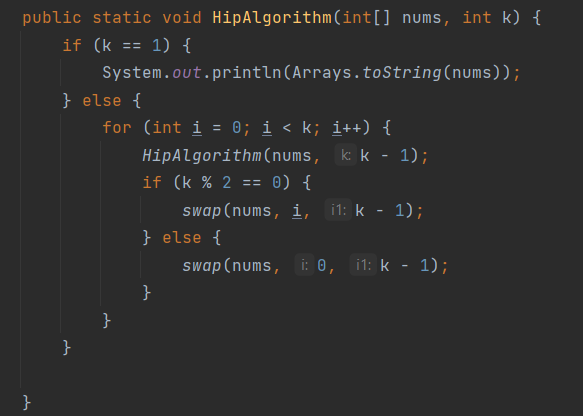
Вычисляем сумму элементов от начального до не включая его, и вычитаем из этой суммы один. Получаем, что .

Увеличиваем на единицу и, начиная со второго элемента и до найденного, на каждом шагу устанавливаем значение равное значению найденного элемента, при этом уменьшая значение на эту величину. После завершения прохода, устанавливаем значение первого элемента последовательности как . В нашем случае в конце получится и и возвращаем полученное новое разбиение. Проверяем условие , но для нашего массива это не выполнено. Значит, начиная со второго элемента последовательности, ищем элемент удовлетворяющий условию , но в нашем массиве такого элемента нет – значит завершаем алгоритм.

**Глава 2. Разработка программ**

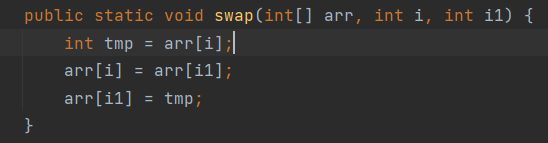
* 1. **Разработка алгоритма Хипа**

Как мы помним в алгоритме Хипа мы пользовались рекурсией, а значит создадим функцию (рис.2.1.1), которая не будет ничего возвращать, а лишь выводить перестановки на экран.



*Рисунок 2.1.1 Функция HipAlgorithm*

На вход мы принимаем сам массив из элементов, перестановки которых мы хотим найти, а также переменную, которая изначально равняется длине массива. Как мы видим если , то выводим текущую перестановку. Если нет, то создаём цикл от первого элемента массива до , и от каждого элемента вызываем функцию с параметром . Также в функции прописана замена элементов, согласно действиям алгоритма. Функция создана самостоятельно и показана на рис. 2.1.2.

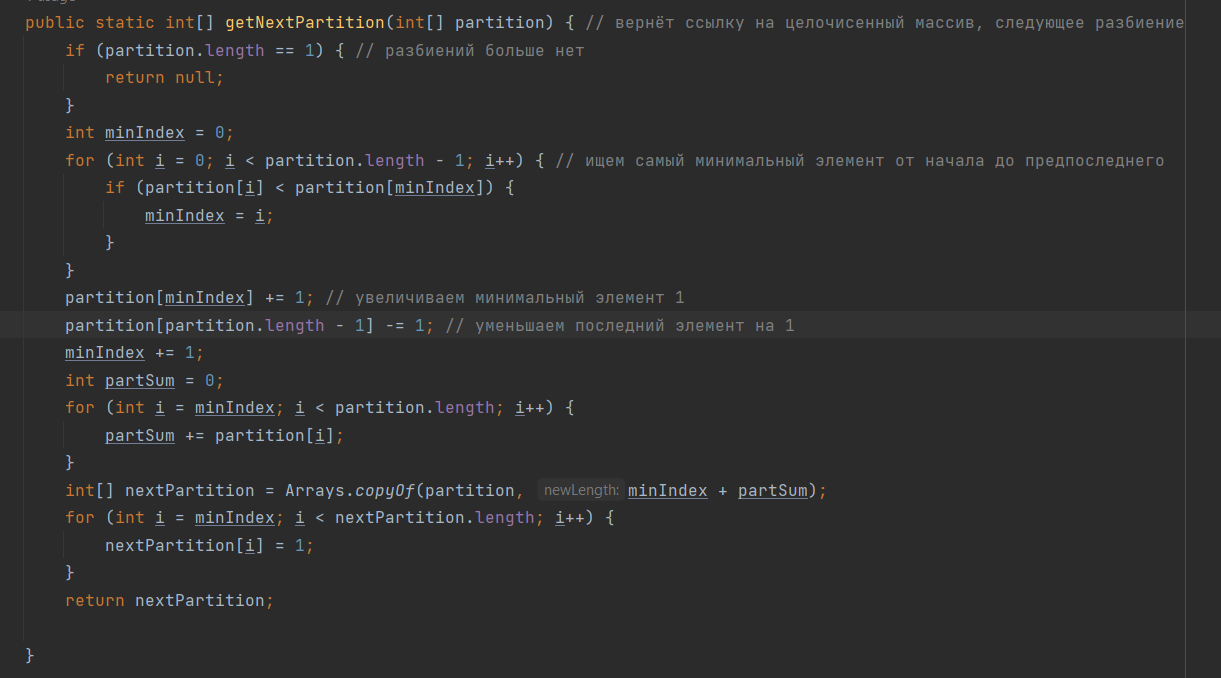


*Рисунок 2.1.2. Функция swap*

Функция работает крайне просто – принимает на вход массив, и два индекса. А далее с помощью третьей переменной меняем местами элементы массива с заданными индексами.

**2.2 Разработка алгоритма генерации всех разбиений числа с помощью динамического программирования**

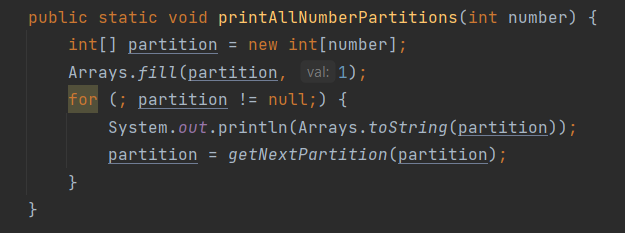
Создадим функцию (рис. 2.2.1) , которая будет возвращать следующее разбиение числа на слагаемые.



*Рисунок 2.2.1 Функция getNextPartition*

На вход данная функция принимает массив, который изначально будет заполнен одними единицами, согласно алгоритму – но потом будет видоизменяться, добавляя новые разбиения заданного числа на слагаемые.

Вначале проверяется длина массива . Если равняется единице, то алгоритм завершает свою работу и возвращает пустую ссылку. Если нет, то согласно алгоритму, ищется минимальный элемент – это мы делаем с помощью обычного цикла . Далее увеличиваем минимальный элемент на единицу, а последний уменьшаем на единицу. Затем с помощью цикла ищем сумму элементов от минимального и до последнего элемента. И наконец создаём новый массив с помощью копирования старого массива, и новой длиной , где – сумма элементов от минимального и до последнего элемента. Операцию копирования элементов в новый массив из старого делаем с помощью метода . И с помощью цикла заполняем элементы с индексом от и до последнего – единицами. Полученный массив и будет следующим разбиением заданного числа на слагаемые. Но это мы только научились создавать следующее разбиение на основе текущего, теперь создадим функцию, которая выведет все разбиения числа – это будет функция ( рис. 2.2.2).

**

*Рисунок 2.2.2 Функция printAllNumberPartitions*

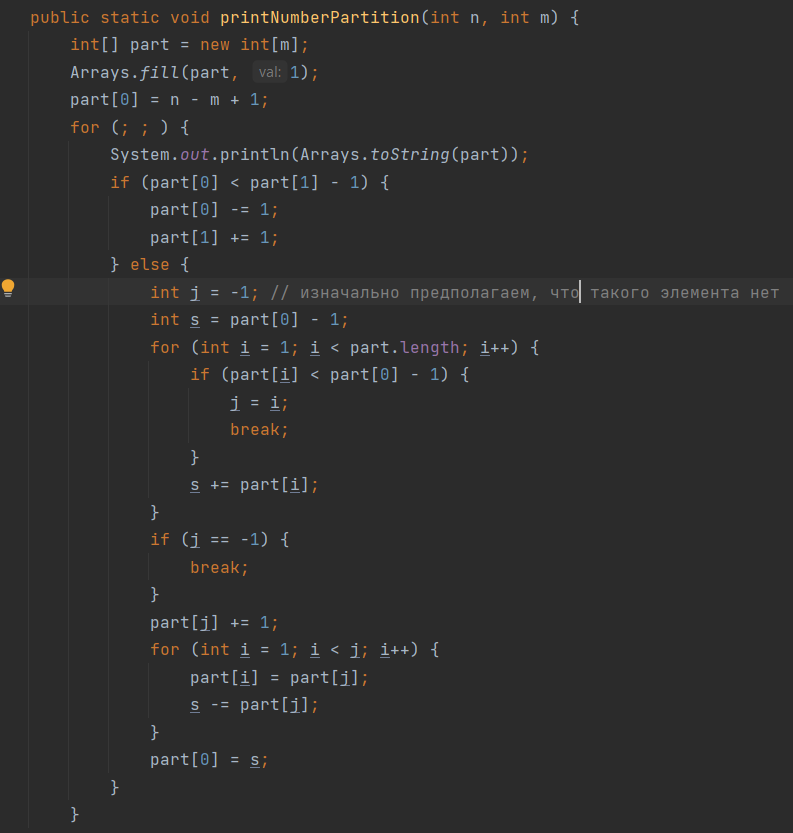
Как мы видим, на вход данная функция принимает число , разбиение которого мы хотим найти. Далее согласно алгоритму создаётся массив, размером , и с помощью метода заполняется единицами. И наконец с помощью цикла , генерируются все разбиения числа. Обратим внимание, что условие выхода из цикла, это то, что ссылка на массив равняется , как раз для этого в функции возвращается ссылка в случае равенства длины массива единице.

* 1. **Разработка алгоритма Карла Фридриха Гинденбурга**

Создадим функцию (рис. 2.3.1) , которая будет возвращать все разбиения числа на слагаемых.

На вход данная функция будет принимать два параметра – число, которое нужно разбить на слагаемые (), а также число слагаемых ().

Согласно алгоритму, вначале мы создаём массив длины , где первый элемент находится по формуле . Далее запускаем бесконечный цикл , в котором проверяем условие . Если оно выполняется, то меняем элементы массива с индексами 0 и 1 согласно алгоритму и далее возвращаемся в начало цикла . Если же оно не выполняется, то ищем элемент удовлетворяющий условию . Сразу инициализируем переменную для подсчёта суммы. Обратите внимание, что сразу вычитаем из этой суммы единицу, чтобы в конце не забыть этого сделать. Далее для того чтобы нам понять что такого элемент удовлетворяющего условию может и не быть, устанавливаем значение переменной , равное -1, чтобы в конце можно было понять, нашли ли мы такой элемент или нет. Если его нашли, то увеличиваем его на единицу, и далее чтобы избежать двух циклов , в одном цикле приравниваем значения элементов массива к переменной (тот самый элемент, который мы нашли на предыдущем шаге) и также уменьшаем сумму на значение этих элементов. И в конце не забываем установить значение первого элемента массива как сумму .



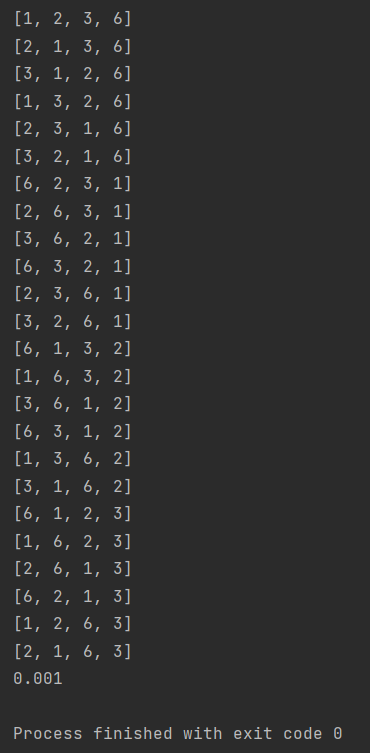
*Рисунок 2.3.1 Функция printNumberPartition*

**Глава 3. Машинный эксперимент**

* 1. **Тестирование алгоритма Хипа**

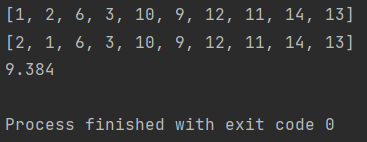
Пусть у нас есть массив и попробуем протестировать алгоритм Хипа и сгенерировать все перестановки элементов данного массива. И также в конце будем выводить время, за которое алгоритм сработал.

Как мы видим на рис. 3.1.1, алгоритм сработал верно и сгенерировал все перестановки за секунд, что довольно быстро.

**

*Рисунок 3.1.1 Результаты работы на массиве a = [1,2,3,6]*

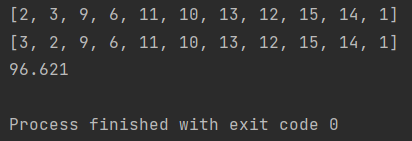
Но как мы знаем что у любого алгоритма, который генерирует перестановки сложность , то есть с увеличением , время работы алгоритма очень быстро увеличивается. Проверим это на практике на массиве . Как можно вычислить у нас будет перестановок. Все перестановки показать не получится, но давайте посмотрим на время, за которое алгоритм сработал. Это видно на рис. 3.1.2.



*Рисунок 3.1.2 Результаты работы на массиве a = [1,2,3,6,9,10,11,12,13,14]*

Как мы видим понадобилось всего лишь 9 секунд, это не так и много.

Но давайте увеличим количество элементов в массиве хотя бы на один, то есть массив и в нём 11 элементов. Количество перестановок в нашем случае будет равняться . Как мы видим из рис. 3.1.3 время работы программы уже 96 секунд, то есть с добавлением одного элемента время работы увеличилось в 16 раз!

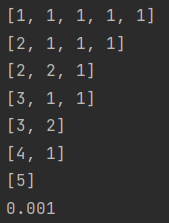


*Рисунок 3.1.3 Результаты работы на массиве a = [1,2,3,6,9,10,11,12,13,14,15]*

И на самом деле если , то даже самая быстрая машина за 18 часов не сможет получить все перестановки. Это простое следствие того факта, что растёт весьма быстро с ростом .

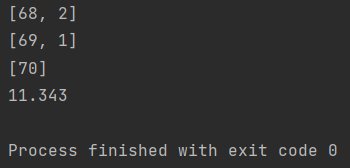
* 1. **Тестирование алгоритма динамического программирования для генерации всех разбиений числа на слагаемые**

Давайте для начала протестируем алгоритм на примере, который был в теории для . Как мы видим из рис. 3.2.1 алгоритм сработал верно и выдал все разбиения числа 5 на слагаемые, и причём время работы очень мало – секунд.

**

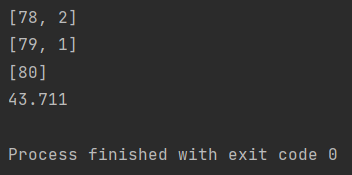
*Рисунок 3.2.1 Результаты работы при n = 5*

Но как мы помним из теории, количество разбиений числа при растёт экспоненциально, то есть очень и очень быстро. Давайте, например, протестируем алгоритм при . Как мы видим на рис. 3.2.2 время работы уже 11 секунд. То есть время работы заметно увеличилось.

**

*Рисунок 3.2.2 Результаты работы при n = 70*

Следующий тест сделаем для . Как мы видим на рис. 3.2.3 время работы уже составило 43 секунды, то есть по сравнению с прошлым результатом увеличилось в 3 раза.

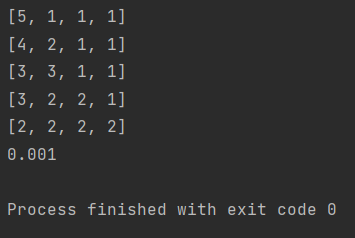
**

*Рисунок 3.2.3 Результаты работы при n = 80*

И заметим, что уже при количество разбиений по асимптотической формуле будет больше одного миллиарда, и за разумное время машина не сгенерирует все возможные способы.

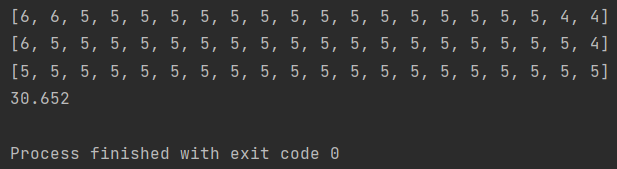
* 1. **Тестирование алгоритма Карла Фридриха Гинденбурга**

Для начала протестируем действия данного алгоритма на примере из теории, то есть для и . Как мы видим из рис. 3.3.1 алгоритм сработал верно, и очень быстро, за секунд.

**

*Рисунок 3.3.1 Результаты работы при n = 8 и m = 4*

Как мы знаем, количество разбиение растёт экспоненциально, поэтому давайте увеличим наши входные параметры. Пусть и . Как мы видим на рис. 3.3.2 время работы алгоритма уже составляет 30 секунд.

**

*Рисунок 3.3.2 Результаты работы при n = 100 и m = 20*

Далее понятно, что если мы будем увеличивать наши входные параметры, то за адекватное время машина не сможет сгенерировать все разбиения числа.

# Заключение

В рамках данной курсовой работы была изучена различная литература по теме генерации комбинаторных объектов, рассмотрены различные алгоритмы и их особенности, а также разработаны и протестированы программы, реализующие эти алгоритмы.

Были рассмотрены классические алгоритмы генерации комбинаторных объектов, такие как алгоритм Хипа, алгоритм Штейнхауза-Джонсона-Троттера, динамический алгоритм для генерации всех разбиений числа, а также алгоритм Карла Фридриха Гинденбурга.

В заключение можно отметить, что генерация комбинаторных объектов является важной и актуальной задачей в математике и информатике. Благодаря различным методам генерации комбинаторных объектов можно решать множество задач, связанных с различными областями знаний, включая теорию графов, криптографию, теорию чисел и др.

# Список литературы

1. Положение о курсовых работах (проектах) в федеральном государственном бюджетном образовательном учреждении высшего образования «Российский экономический университет имени Г. В. Плеханова».
2. ГОСТ 2.105-95. ЕСКД. Общие требования к текстовым документам.
3. ГОСТ Р 54593-2011. Информационные технологии. Свободное программное обеспечение. Общие положения.
4. Комбинаторика: методы, результаты, проблемы / под ред. В. Д. Белоусова. Москва: Наука, 1975. 544 с.
5. Мендельсон, Б. Комбинаторика: введение в математический анализ. Москва: Наука, 1984. 296 с.
6. Косточка, А. В. Комбинаторика: генерация комбинаторных объектов. Москва: МФТИ, 2017. 220 с.
7. Кузнецов, А. А. Элементы комбинаторики: перестановки, размещения, сочетания. Москва: Логос, 2005. 224 с.
8. Штарев, А. М. Комбинаторика. Москва: МЦНМО, 2015. 400 с.
9. Горбачев, А. В. Комбинаторика: генерация комбинаторных объектов. Москва: МГУ, 2012. 336 с.
10. Голубцов, В. В. Комбинаторика: задачи, теория, методы. Москва: Издательство Физико-математической литературы, 2002. 256 с.
11. Зельдин, И. А. Комбинаторика для начинающих. Москва: МЦНМО, 2010. 240 с.
12. Малаховский, В. А. Комбинаторика: задачи и теория. Москва: МЦНМО, 2013. 288 с.
13. Попов, В. Н. Комбинаторика. Москва: Издательство ЛКИ, 2005. 256 с.
14. Смоляков, С. А. Комбинаторика: основы теории и приложения. Москва: ФИЗМАТЛИТ, 2007. 432 с.
15. Липский В. Комбинаторика для программистов. –М. :Мир, 1988.–213 с.

# Приложения

## Приложение 1

### Алгоритм Карла Фридриха Гинденбурга

**import** **java.util.Arrays**;

**public** **class** **Solution** {

// вывести все разбиения числа n на m слагаемых

**public** **static** **void** **printNumberPartition**(**int** n, **int** m) {

**int**[] part = **new** **int**[m];

Arrays.fill(part, **1**);

part[**0**] = n - m + **1**;

**for** (; ; ) {

System.out.println(Arrays.toString(part));

**if** (part[**0**] < part[**1**] - **1**) {

part[**0**] -= **1**;

part[**1**] += **1**;

} **else** {

**int** j = -**1**; // изначально предполагаем, что такого элемента нет

**int** s = part[**0**] - **1**;

**for** (**int** i = **1**; i < part.length; i++) {

**if** (part[i] < part[**0**] - **1**) {

j = i;

**break**;

}

s += part[i];

}

**if** (j == -**1**) {

**break**;

}

part[j] += **1**;

**for** (**int** i = **1**; i < j; i++) {

part[i] = part[j];

s -= part[j];

}

part[**0**] = s;

}

}

}

}

## Приложение 2

### Алгоритм Хипа

**public** **static** **void** **HipAlgorithm**(**int**[] nums, **int** k) {

**if** (k == **1**) {

System.out.println(Arrays.toString(nums));

} **else** {

**for** (**int** i = **0**; i < k; i++) {

HipAlgorithm(nums, k - **1**);

**if** (k % **2** == **0**) {

swap(nums, i, k - **1**);

} **else** {

swap(nums, **0**, k - **1**);

}

}

}

}

**public** **static** **void** **swap**(**int**[] arr, **int** i, **int** i1) {

**int** tmp = arr[i];

arr[i] = arr[i1];

arr[i1] = tmp;

}

## Приложение 3

### Алгоритм генерации разбиения числа с помощью динамического программирования

**import** **java.util.Arrays**;

**public** **class** **Solution** {

**public** **static** **int**[] **getNextPartition**(**int**[] partition) { // вернёт ссылку на целочисенный массив, следующее разбиение

**if** (partition.length == **1**) { // разбиений больше нет

**return** **null**;

}

**int** minIndex = **0**;

**for** (**int** i = **0**; i < partition.length - **1**; i++) { // ищем самый минимальный элемент от начала до предпоследнего

**if** (partition[i] < partition[minIndex]) {

minIndex = i;

}

}

partition[minIndex] += **1**; // увеличиваем минимальный элемент 1

partition[partition.length - **1**] -= **1**; // уменьшаем последний элемент на 1

minIndex += **1**;

**int** partSum = **0**;

**for** (**int** i = minIndex; i < partition.length; i++) {

partSum += partition[i];

}

**int**[] nextPartition = Arrays.copyOf(partition, minIndex + partSum);

**for** (**int** i = minIndex; i < nextPartition.length; i++) {

nextPartition[i] = **1**;

}

**return** nextPartition;

}

**public** **static** **void** **printAllNumberPartitions**(**int** number) {

**int**[] partition = **new** **int**[number];

Arrays.fill(partition, **1**);

**for** (; partition != **null**;) {

System.out.println(Arrays.toString(partition));

partition = getNextPartition(partition);

}

}