Министерство науки и высшего образования Российской Федерации
ФГАОУ ВО «УрФУ имени первого Президента России Б.Н. Ельцина»
Кафедра «школы бакалавриата (школа)»

Оценка работы_____
Руководитель от УрФУ Сеньчонок Т.А.

Тема задания на практику

О кратчайших последовательностях элементарных преобразований в решетке разбиений натуральных чисел

ОТЧЕТ
Вид практики Производственная практика
Тип практики Производственная практика, преддипломная

Руководитель практики от УрФУ Сеньчонок Татьяна Александровна
Студент Катяева Дарья Александровна
Специальность (направление подготовки) 02.03.01 Математика и компьютерные науки
Группа МЕН-400207

Екатеринбург 2024

Содержание

Введение

Теория разбиений является важной частью математики, находясь на стыке комбинаторики и теории чисел. Понятие разбиения натурального числа впервые было представлено в 1669 году в письме Лейбница к Иоганну Бернулли. Основы же теории разбиений чисел были заложены Эйлером. Разбиения чисел играют важную роль во многих областях математики и современной физики.

Представление натурального числа в виде суммы натуральных чисел называется разбиением числа. Классические задачи комбинаторики занимаются подсчётом и перечислением разбиений определённого типа, в то время как в теории чисел исследуются вопросы, связанные с аддитивными представлениями чисел при наличии арифметических ограничений на слагаемые. Примерами таких задач являются проблемы Гольдбаха и Варинга. Решение задач о разбиениях часто сталкивается с серьёзными трудностями, требующими создания специализированных методов в рамках теории разбиений и проявления изобретательности для их преодоления.

Во второй половине XX века теория разбиений пережила существенные новые продвижения. Развитие компьютерных технологий и новые методы исследования привели к расширению границ этой области и открыли новые перспективы в изучении различных типов разбиений. Новые результаты в теории чисел, комбинаторике и математической физике были частично обусловлены вкладом теории разбиений, что подчеркивает ее важность и актуальность в настоящее время.

В данной работе будут рассмотрены элементарные преобразования разбиений, позволяющие переходить от одного разбиения к другому путем простых операций. Элементарные преобразования играют важную роль в теории разбиений, облегчая анализ различных типов разбиений и их свойств. Алгоритм поиска всех кратчайших последовательностей

элементарных преобразований будет иметь практическую значимость, позволяя эффективно решать различные задачи в области комбинаторики и теории чисел.

Постановка задачи

Разбиением называется последовательность $\lambda = (\lambda_1, \lambda_2, \ldots)$ целых неотрицательных чисел такая, что λ содержит лишь конечное число ненулевых компонент, $\lambda_1 \geq \lambda_2 \geq \cdots$ и $\sum_{i=1}^{\infty} \lambda_i = m$, где m — натуральное число. Говорят, что λ является разбиением числа m, а m называют весом разбиения λ, также пишут $m = sum(\lambda)$. Натуральное число $l = l(\lambda)$ такое, что $\lambda_l > 0$ и $\lambda_{l+1} = \lambda_{l+2} = \cdots = 0$, называют длиной разбиения λ. Для удобства разбиение λ будем иногда записывать в виде конечной последовательности следующих типов:

$$\lambda = (\lambda_1, \ldots, \lambda_t) = (\lambda_1, \ldots, \lambda_{t+1}) = (\lambda_1, \ldots, \lambda_{t+2}) = \ldots,$$

где $t = l(\lambda)$, то есть будем опускать нули, начиная с некоторой компоненты, но при этом будем помнить, что рассматривается бесконечная последовательность.

На множестве всех разбиений всех натуральных чисел, обозначаемом NPL, и множестве всех разбиений заданного натурального числа m ($NPL(m)$) будем рассматривать отношение доминирования $\trianglelefteq$, такое что $\lambda \trianglelefteq \mu$, если

$$\lambda_1 \leq \mu_1,$$

$$\lambda_1 + \lambda_2 \leq \mu_1 + \mu_2,$$

$$\ldots$$

$$\lambda_1 + \lambda_2 + \cdots + \lambda_i \leq \mu_1 + \mu_2 + \cdots + \mu_i,$$

$$\ldots$$

где $\lambda = (\lambda_1, \lambda_2, \ldots)$ и $\mu = (\mu_1, \mu_2, \ldots)$.

Разбиение будем изображать в виде диаграммы Ферре, которая представляет из себя конечный набор квадратных блоков одинакого размера, составляющих ступенчатую фигуру. Пример диаграммы Ферре для разбиения $\lambda = (6, 5, 4, 4, 3, 2, 1, 1)$ длины 8 и веса 26:

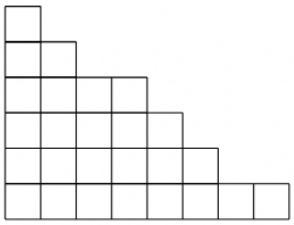

Рис. 1

Также дадим определения элементарных преобразований для разбиения $\lambda = (\lambda_1,\ \lambda_2,\ \ldots)$. Для натуральных чисел $i, j \in \{1, \ldots, t\}$ таких, что $i < j \leq t = l(\lambda) + 1$, выполняются условия:

1) $\lambda_{i-1} \geq \lambda_{i+1}$
2) $\lambda_{j-1} \geq \lambda_{j+1}$
3) $\lambda_i \geq 2 + \lambda_j$

Будем говорить, что разбиение $\mu = (\lambda_1, \ldots, \lambda_i - 1, \ldots, \lambda_j + 1, \ldots, \lambda_t)$ получено из разбиения $\lambda = (\lambda_1, \ldots, \lambda_i, \ldots, \lambda_j, \ldots, \lambda_t)$ элементраным преобразованием первого типа (или перекидыванием блока). Разбиения λ и μ отличаются на двух компонентах с номерами i и j. На диаграмме Ферре такое преобразование означает перемещение верхнего блока i-го столбца вправо на верх j-го столбца. Условия 1) и 2) гарантируют, что после преобразования снова получится разбиение. Вес разбиения тоже сохраняется.

Пусть для некоторого столбца i такого, что $1 \leq i \leq l(\lambda)$ выполняется условие $\lambda_i - 1 \geq \lambda_{i+1}$. Преобразование заменяющее разбиение λ на разбиение $\mu = (\lambda_1, \ldots,\ \lambda_{i-1},\ \lambda_i - 1,\ \lambda_{i+1} \ldots)$ будем называть элементарным преобразованием второго типа (или удалением блока). Удаление блока уменьшает вес разбиения на 1.

Если разбиение μ получено из разбиения λ с помощью элементарного преобразования первого или второго типа, то кратко будем писать $\lambda \rightarrow \mu$.

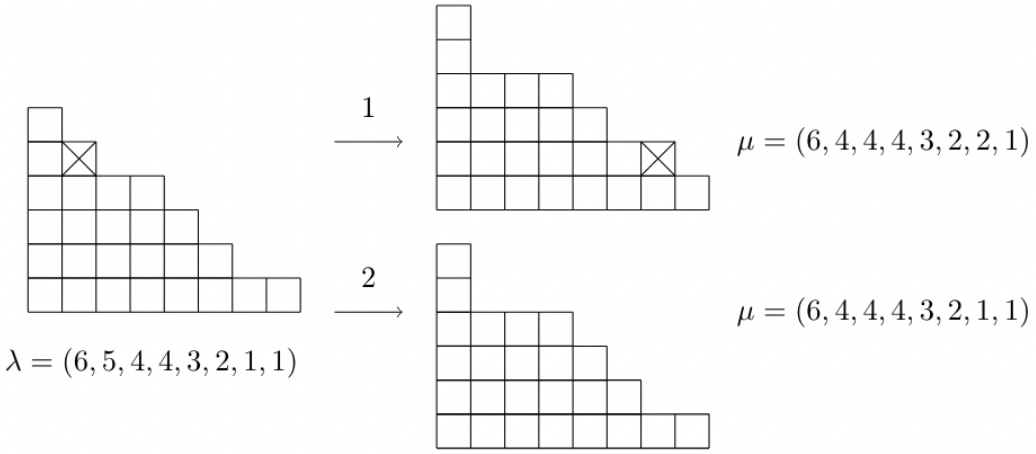

Рис. 2

Определим отношение ≥, полагая $\lambda \geq \mu$, если разбиение μ можно получить из разбиения λ с помощью последовательного применения конечного числа (возможно нулевого) элементарных преобразований указанных типов. Множества NPL и $NPL(m)$ являются решетками относительно ≤.

Пусть λ и μ— два произвольных разбиения и $\lambda \geq \mu$.

Высотой $height(\lambda, \mu)$ разбиения λ над разбиением μ будем называть число преобразований в кратчайшей последовательности элементарных преобразований, преобразующей λ в μ.

Цель преддипломной практики состоит в построении алгоритма, который для двух произвольных разбиений λ и μ таких, что $\lambda \geq \mu$, находит все кратчайшие последовательности элементарных преобразований от λ до μ. Для того, чтобы реализовать алгоритм, который строит все такие кратчайшие последовательности, нужно научиться находить некоторую кратчайшую последовательность, поэтому подцелью данной работы будет реализация алгоритма нахождения некоторых кратчайших последовательностей элементарных преобразований от λ до μ.

Для достижения поставленной цели, в работе будут решаться следующие задачи:

- рассмотреть теоритические аспекты теории разбиений;

- научиться находить некоторую кратчайшую последовательность элементарных преобразований от разбиения λ до разбиения μ;
- на основе первого алгоритма, научиться находить все кратчайшие последовательности элементарных преобразований от λ до μ;
- реализовать описанный алгоритм;
- провести анализ результатов на нескольких примерах разбиений;

Алгоритм

Описание алгоритма, находящего некоторые кратчайшие последовательности элементарных преобразований

Для того, чтобы приступить к описанию алгоритма нужно привести еще некоторые дополнительные замечания и определения.

Пусть $\boldsymbol{\lambda}$ и $\boldsymbol{\mu}$— два произвольных разбиения и $\boldsymbol{\lambda} \geq \boldsymbol{\mu}$, а t - максимальная из длин $\boldsymbol{\lambda}$ и $\boldsymbol{\mu}$. Заметим, что если $\boldsymbol{\lambda} \geq \boldsymbol{\mu}$, то $\boldsymbol{sum}(\boldsymbol{\lambda}) \geq \boldsymbol{sum}(\boldsymbol{\mu})$ и $\boldsymbol{C} = \boldsymbol{sum}(\boldsymbol{\lambda}) - \boldsymbol{sum}(\boldsymbol{\mu}) \geq \mathbf{0}, \boldsymbol{C} \in \mathbb{Z}$. Условие $\boldsymbol{\lambda} \geq \boldsymbol{\mu}$ можно задать следующей системой:

$$\begin{cases} \lambda_1 + \cdots + \lambda_t = \mu_1 + \cdots + \mu_t + C \\ \lambda_1 + \cdots + \lambda_{k-1} \geq \mu_1 + \cdots + \mu_{k-1} \ (k = 2, \dots, t) \end{cases}$$

Так как для любого $k = 2, \dots, t$ выполняется

$$\lambda_1 + \cdots + \lambda_{k-1} + \lambda_k + \cdots + \lambda_t = \mu_1 + \cdots + \mu_{k-1} + \mu_k + \cdots + \mu_t + C,$$

то система эквивалента системе:

$$\begin{cases} \lambda_1 + \cdots + \lambda_t = \mu_1 + \cdots + \mu_t + C \\ \lambda_k + \cdots + \lambda_t \leq \mu_k + \cdots + \mu_t + C \ (k = 2, \dots, t) \end{cases}$$

Перепишем эту систему в эквивалентной форме:

$$\begin{cases} \sum\limits_{j=1,\, \lambda_j > \mu_j}^{t} (\lambda_j - \mu_j) = \sum\limits_{j=1,\, \lambda_j < \mu_j}^{t} (\mu_j - \lambda_j) + C, \\ \sum\limits_{j=k,\, \lambda_j > \mu_j}^{t} (\lambda_j - \mu_j) \leq \sum\limits_{j=k,\, \lambda_j < \mu_j}^{t} (\mu_j - \lambda_j) + C \ (k = 2, \dots, t). \end{cases}$$

Для всех $j = 1, \dots, t$ будем говорить, что j-компонента разбиения λ имеет горку относительно разбиения μ, если $\lambda_j > \mu_j$. В этом случае верхние $\lambda_j - \mu_j$ блоков диаграммы Ферре разбиения λ образуют горку высоты $\lambda_j - \mu_j$.

Для $j = 1, \dots, t$ будем говорить, что j-компонента разбиения λ имеет ямку относительно разбиения μ, если $\lambda_j < \mu_j$ и над j-столбцом диаграммы Ферре разбиения λ имеется ямка глубины $\mu_j - \lambda_j$.

Условия из последней системы можно переформулировать в следующем виде:

- сумма высот всех горок равна сумме числа C и суммы глубин всех ямок,

- для любого $k = 2, \ldots, t$ сумма высот всех j-горок таких, что $j \geq k$, не превосходит суммы числа C и суммы глубин всех j-ямок таких, что $j \geq k$.

Пусть $\lambda \geq \mu$ и λ имеет i-ямку для некоторого $i = 1, \ldots, t$ относительно μ. Тогда в силу условий, в вышеприведенной системе, существует ближайшая к ней слева в диаграмме Ферре i'-горка такая, что $1 \leq i' < i$. Тогда между i-ямкой и i'-горкой нет горок. Заметим, что $\lambda_{i'} > \mu_{i'} \geq \mu_{i'+1} \geq \lambda_{i'+1}$, то есть $\lambda_{i'} > \lambda_{i'+1}$. В этом случае будем говорить, что в i'-столбце диаграммы имеется уступ.

Будем говорить, что ямка допустима если $\lambda_{i-1} < \lambda_i$. Ближайшая к i'-горке ямка допустима. Предположим, что такая ямка в k-столбце, тогда $\lambda_{k-1} > \mu_{k-1} \geq \mu_k \geq \lambda_k$, то есть $\lambda_{k-1} > \lambda_k$. Таким образом, если имеется хотя бы одна ямка, то имеется и допустимая ямка.

Давайте рассмотрим допустимую ямку. Если рассматриваемая i-ямка допустима, то к разбиению можно применить (i', i) перекидывание блока, то есть перекидывание верхнего блока i'-столбца на верх i-столбца. Полученное разбиение обозначим λ'. Указанное преобразование будем называть перекидыванием верхнего блока в допустимую ямку из ближайшей к ней слева горки. Для полученного разбиения λ' сохраняются условия системы, то есть $\lambda' \geq \mu$.

Теперь рассмотрим последнюю горку разбиения λ, то есть горку с наибольшим значением индекса i. Тогда $\lambda_i > \mu_i \geq \mu_{i+1} \geq \lambda_{i+1}$, то есть в i-столбце имеется уступ. Тогда можно применить преобразование второго типа к разбиению λ. Будем называть такое преобразование удалением верхнего блока с последней горки. Очевидно, что для полученного разбиения λ' условия системы сохраняются и $sum(\lambda') - sum(\mu) = C - 1$, то есть $\lambda' \geq \mu$.

Теперь можем приступить к описанию алгоритма, который строит некоторые кратчайшие последовательности длины $height(\lambda, \mu)$ элементарных преобразований от λ до μ, если $\lambda \geq \mu$.

Алгоритм 1:

Пусть $\lambda \geq \mu$ и $sum(\lambda) - sum(\mu) = C$.

1) Пусть $\lambda' = \lambda$ и $C' = C$.

2) К текущему разбиению λ' и числу C' применим одно любое из следующих преобразований:

a) если λ' имеет ямку, то заменяем λ' на разбиение, полученной из λ' с помощью перекидывания верхнего блока в некоторую допустимую ямку из ближайшей к ней слева горки;

b) если $C' > 0$, то заменяем C' на $C' - 1$, а разбиение λ' на разбиение, полученное из λ' с помощью удаления верхнего блока из последней горки.

3) Выполняем шаг 2 до тех пор, пока это возможно. Процесс обязательно завершится. Выполненные преобразования составят кратчайшую последовательность элементарных преобразований от λ до μ.

Доказательство корректности алгоритма:

Исходя из предыдущих замечаний, справедливо следующее:

- если у разбиения имеется ямка, то имеется и допустимая ямка;
- при каждом выполнении шага два сумма высот всех горок уменьшается точно на 1;
- для каждого следующего значения λ' выполняется $\lambda' \geq \mu$.

В силу этих утверждений у полученного разбиения λ' не будет горок и ямок и будет выполняться равенство $C' = 0$, поэтому будет выполняться и $\lambda' = \mu$.

Описание алгоритма, находящего все кратчайшие последовательности элементарных преобразований

Переборный алгоритм для поиска всех кратчайших последовательностей преобразований от разбиения λ к разбиению μ может быть реализован с использованием поиска в глубину (DFS) или поиска в ширину (BFS). В данном случае мы рассмотрим подход с использованием BFS, так как он удобен для нахождения всех кратчайших путей.

Алгоритм 2:

Пусть $\lambda \geq \mu$ и $sum(\lambda) - sum(\mu) = C$.

1) Инициализация:
   - Создаем очередь, в которой будем хранить состояние в виде тройки: $(\lambda', C',\ path)$, где λ' – текущее разбиение, C' – текущая разница сумм разбиений, а $path$ - последовательность преобразований;
   - Добавляем начальное состояние $(\lambda, C, [])$ в очередь;
   - Создаём множество для отслеживания посещённых состояний, чтобы избежать циклов;
2) Основной цикл:
   – Пока очередь не пуста, выполняем следующие шаги:
      1) Извлекаем первое состояние из очереди;
      2) Если текущее состояние (λ', C') равно целевому состоянию $(\mu, 0)$, то сохраняем текущий путь $path$, добавляя в множество, как одно из решений;
      3) Иначе генерируем все возможные элементарные преобразования для текущего состояния (λ', C'), используя Алгоритм 1:

      а) если λ' имеет ямку:

- перебираем все возможные перекидывания верхнего блока из ближайшей слева горки в ямку;
- для каждого нового состояния λ'' добавляем состояние $(\lambda'', C',\ path + [shift\ block\ from\ i\ to\ j])$ в очередь;

b) если $C' > 0$:

- заменяем C' на $C' - 1$, а разбиение λ' на разбиение, полученное из λ'' с помощью удаления верхнего блока из последней горки;
- добавляем новое состояние $(\lambda'', C' - 1,\ path + [remove\ last\ top\ block\ from\ i])$;

– Помечаем текущее состояние как посещенное;

3) Завершение: когда очередь будет пуста, все возможные кратчайшие последовательности преобразований будут найдены.

Данный полный переборный алгоритм всех возможных путей может занять экспоненциальное время $O(2^n)$, а многие вычисления для найденных решений будут избыточными, так как решения окажутся не самыми минимальными. Чтобы избавиться от недостатков текущего алгоритма, можно попытаться его оптимизировать.

Для поиска всех кратчайших последовательностей преобразований можно рассмотреть оптимизацию алгоритма с использованием поиска с отсечением (Branch and Bound) или поиска с возвратом (Backtracking). Эти подходы могут быть эффективнее в некоторых случаях, так как позволяют избегать явно ненужных путей.

Алгоритм 3:

1) Поиск минимального количества преобразований с помощью BFS:

- Определяем начальное состояние (λ', C');

- Инициализируем очередь для BFS и добавляем в нее состояние с нулевым числом шагов;

- Инициализируем множество посещенных состояний для отслеживания уже проверенных;

Пока очередь не пуста выполняем:

1) Извлекаем текущее состояние $(\lambda', C', steps)$ из очереди;

2) Если текущее разбиение равно целевому и $C' = 0$, то возвращаем текущее количество шагов steps, как наименьшее;

3) Если состояние уже посещено, то пропускаем его;

4) Если текущее разбиение содержит ямку, то выполняем преобразование перекидывания верхнего блока из ближайшей слева горки в ямку и добавляем новое состояние в очередь с увеличением числа шагов на единицу;

5) Если $C' > 0$, то выполняем удаление верхнего блока с последней горки и добавляем в очередь новое состояние с уменьшением C' на единицу и с увеличением числа шагов;

2) Нахождение всех путей минимальной длины с помощью Backtracking:

- Инициализируем простой список для хранения всех возможных решений;

- Определяем рекурсивную функцию backtrack, которая принимает текущее разбиение, текущее значение C и текущий путь преобразования;

В функции backtrack:

- Если длина текущего пути превышает минимальное количество шагов, то возвращаемся;

- Если текущее преобразование содержит ямку, то выполняем преобразование с перекидыванием верхнего блока, обновляем путь и вызываем функцию backtrack с новым состоянием;

- Если $C > 0$, то выполняем соответствующее преобразование – удаление верхнего блока, обновляем путь и также вызываем функцию backtrack с новым состоянием;

- Вызываем функцию backtrack с начальным состоянием;

3) Вывод результатов

Преимущества оптимизации алгоритма с использованием поиска с возвратом:

- BFS для нахождения минимального количества шагов:

BFS гарантирует нахождение кратчайшего пути в графе состояний. Это свойство используется для определения минимального количества шагов, необходимых для преобразования одного разбиения в другое.

Применение BFS до поиска с возвратом позволяет ограничить глубину поиска и сосредоточиться только на путях минимальной длины.

- Поиск с возвратом (backtracking):

После нахождения минимального количества шагов backtracking используется для нахождения всех возможных путей такой длины.

Ограничение поиска только путями минимальной длины значительно сокращает пространство поиска, так как исключаются все более длинные пути.

- Использование структуры данных:

Хранение состояний в виде пар (разбиение, оставшееся количество блоков) и использование множества посещенных состояний предотвращает повторные вычисления и оптимизирует процесс поиска.

Без оптимизации:

- Полный перебор всех возможных путей может занять экспоненциальное время $O(2^n)$;
- Множество проверок и вычислений, многие из которых будут избыточными;

С оптимизацией:

- BFS находит минимальное количество шагов за полиномиальное время в худшем случае.
- Backtracking ограничивается путями минимальной длины, что существенно сокращает общее количество проверяемых путей.

Реализация алгоритма

Для реализации алгоритма поиска некоторых кратчайших последовательностей преобразований был язык программирования Kotlin.
Kotlin — это современный, лаконичный и безопасный язык разработки от компании JetBrains, работающий поверх Java Virtual Machine. Это статически-типизированный, объектно-ориентированный язык программирования.
Выбор этого языка можно обосновать несколькими причинами:

1) Выразительность и читаемость кода: Kotlin предлагает чистый и выразительный синтаксис, который облегчает написание и понимание кода. Это важно для разработки сложных алгоритмов, так как помогает сделать код более понятным и поддерживаемым.

2) Производительность и надежность: Kotlin использует ту же виртуальную машину Java (JVM), что и Java, что обеспечивает высокую производительность и надежность. Это особенно важно при реализации сложных алгоритмов, где производительность играет решающую роль.

3) Богатая стандартная библиотека: язык поставляется с обширной стандартной библиотекой, которая включает множество полезных функций и инструментов. Это позволяет разработчикам сократить время разработки, используя готовые решения для общих задач.

Перейдем к реализации первого алгоритма. Для этого создадим класс Partition для манипуляции с разбиением. В этом классе будут доступны методы для преобразования разбиения такие как: сдвиг блока в ближайшую допустимую ямку, поиск ямки относительно второго разбиения, удаление верхнего блока последней горки. А разбиение будет храниться в классе в виде конечного списка чисел неравных нулю.

Рассмотрим подробнее функции, выполняющие преобразования, в нашем алгоритме.

1) Перекидывание верхнего блока в некоторую допустимую ямку из ближайшей к ней слева горки:

Для применения этого преобразования нужно наличие ямки в разбиении, это проверяется в функции containsPit:

```
fun containsPit(p: Partition): Boolean {
    for (i in sequence.indices) {
        if (sequence[i] - p.sequence[i] < 0) {
            return true
         }
        }
    return false
}
```

Функция shifting выполняет само преобразование:

```
fun shifting(p: Partition): Pair<Partition, String> {
        // поиск последней допустимой горки
        var lastAcceptablePitIndex = 0
        for (i in 1 until sequence.size) {
            val diff = sequence[i] - p.sequence[i]
            if (diff < 0 && sequence[i - 1] > sequence[i]) {
                lastAcceptablePitIndex = i
            }
        }

        // поиск ближайшей слева горки
        var leftNearestSlideIndex = 0
        for (i in lastAcceptablePitIndex - 1 downTo 0) {
            if (sequence[i] > sequence[lastAcceptablePitIndex] && sequence[i]
> 1) {
                leftNearestSlideIndex = i
                break
            }
        }

        // сдвиг блока
        val newSequence = List(sequence.size) { i ->
            when (i) {
                lastAcceptablePitIndex -> sequence[i] + 1
                leftNearestSlideIndex -> sequence[i] - 1
                else -> sequence[i]
            }
        }

        val transformation = "Shift block from index $leftNearestSlideIndex
to $lastAcceptablePitIndex"
        return Pair(Partition(newSequence), transformation)
    }
```

В этой функции в цикле мы находим индекс последней допустимой ямки, проверяя, что предыдущее число в разбиении больше и разность значений на текущей позиции разбиений отрицательна.

Далее, также в цикле, мы ищем ближайшую слева горку проверяя условие, что текущее значение больше значения в ямке и что значение больше единицы, чтобы при перекидывании блока с текущей позиции свойства разбиения сохранились. Функция возвращает новое разбиение и обновленную последовательность преобразований.

2) Удаления верхнего блока из последней горки:

Для этого преобразования создадим функцию removingLastTopBlock, в которой в цикле будем искать последнюю горку, проверяя условие, что разность текущих позиций в разбиениях строго положительна. Запоминаем индекс, удовлетворяющий условию, и далее отнимаем единицу у значения находящегося по данному индексу. Функция также возвращает новое разбиение и обновленную последовательность преобразований.

```
fun removingLastTopBlock(p: Partition): Pair<Partition, String> {
        // поиск последней горки
        var lastSlideIndex = 0
        for (i in sequence.indices) {
            val diff = sequence[i] - p.sequence[i]
            if (diff > 0) {
                lastSlideIndex = i
            }
        }

        // удаление верхнего блока последней горки
        val newSequence = sequence.mapIndexed { i, value ->
            if (i == lastSlideIndex) (value - 1) else value
        }

        val transformation = "Remove top block from index $lastSlideIndex"
        return Pair(Partition(newSequence), transformation)
}
```

Также создадим функцию с самим алгоритмом поиска некоторых кратчайших последовательностей:

```
fun searchShortestSequence(p1: Partition, p2: Partition) {
    val pair = Partition.equalizeSequenceLengths(p1, p2)
    var a = pair.first
    val b = pair.second
    var c = a.sum - b.sum
    var isContainsPit = a.containsPit(b)

    var numberOfTransformations = 0

    while (c > 0 || isContainsPit) {
        if (isContainsPit) {
            a = a.shifting(b)
        } else {
            c -= 1
            a = a.removingLastTopBlock(b)
        }
        isContainsPit = a.containsPit(b)
        numberOfTransformations += 1

        println(a.sequence)
    }

    println(numberOfTransformations
}
```

Метод equalizeSequenceLengths уравнивает длину разбиений, добавляя нули в конец списка чисел разбиения, которое оказалось короче. Далее определяем переменную c, как разность сумм разбиений a и b, где $a \geq b$. Также проверяем наличие ямки с помощью метода containsPit класса Partition. Далее в цикле пока у нас либо есть ямка, либо переменная c больше нуля выполняем одно из двух действий: при наличии ямки, выполняем метод shifting, который перекидывает верхний блок в некоторую допустимую ямку из ближайшей к ней слева горки, иначе сдвигаем верхний блок с последней горки с помощью метода removingLastTopBlock.

Переменная numberOfTransformations хранит в себе количество итераций данного цикла, то есть количество преобразований разбиения a до разбиения b.

Перейдем к реализации алгоритма поиска всех кратчайших элементарных преобразований.

Поиск минимального количества преобразований с помощью BFS реализуем в методе findMinSteps. Он проверяет все возможные состояния, избегая повторных посещений, и выполняет необходимые преобразования на каждом шаге. Если целевое разбиение достигается, возвращается минимальное количество шагов, иначе — максимальное значение целого числа:

```
fun findMinSteps(start: Partition, target: Partition): Int {
    val (startEq, targetEq) = Partition.equalizeSequenceLengths(start,
target)
    val initialState = Triple(startEq, startEq.sum - targetEq.sum, 0)
    val queue: Queue<Triple<Partition, Int, Int>> = LinkedList()
    queue.add(initialState)
    val visited = mutableSetOf<Pair<Partition, Int>>()

    while (queue.isNotEmpty()) {
        val (partition, C, steps) = queue.poll()
        val state = Pair(partition, C)

        if (visited.contains(state)) continue
        visited.add(state)

        if (partition.sequence == targetEq.sequence && C == 0) {
            return steps
        }

        if (partition.containsPit(targetEq)) {
            val (newPartition, _) = partition.shifting(targetEq)
            queue.add(Triple(newPartition, C, steps + 1))
        }

        if (C > 0) {
            val (newPartition, _) = partition.removingLastTopBlock(targetEq)
            queue.add(Triple(newPartition, C - 1, steps + 1))
        }
    }

    return Int.MAX_VALUE
}
```

Метод backtrack использует алгоритм поиска с возвратом (backtracking) для нахождения всех возможных кратчайших последовательностей преобразований от начального разбиения к целевому. Метод рекурсивно генерирует все возможные пути преобразований минимальной длины, обеспечивая нахождение всех кратчайших последовательностей преобразований между разбиениями:

```
fun backtrack(partition: Partition, C: Int, path: List<Pair<Partition,
String>>) {
        if (partition.sequence == targetEq.sequence && C == 0) {
            if (path.size == minSteps) {
                solutions.add(path)
```

```
            }
            return
        }

        if (path.size > minSteps) return

        if (partition.containsPit(targetEq)) {
            val (newPartition, transformation) = partition.shifting(targetEq)
            backtrack(newPartition, C, path + Pair(newPartition,
transformation))
        }

        if (C > 0) {
            val (newPartition, transformation) =
partition.removingLastTopBlock(targetEq)
            backtrack(newPartition, C - 1, path + Pair(newPartition,
transformation))
        }
 }
```

Метод findShortestTransformations объединяет все шаги алгоритма 3. В нем первым шагом уравниваются разбиения, путем добавления нулей в конец разбиения меньшей длины. Вторым шагом вызывается метод findMinSteps для нахождения минимального количества преобразований. Далее вызывается функция backtrack с начальным состоянием. Функция возвращает список всех решений, то есть список всех путей преобразований длины minSteps.

```
fun findShortestTransformations(start: Partition, target: Partition):
List<List<Pair<Partition, String>>> {
    val (startEq, targetEq) = Partition.equalizeSequenceLengths(start,
target)
    val solutions = mutableListOf<List<Pair<Partition, String>>>()
    val minSteps = findMinSteps(startEq, targetEq)
    backtrack(startEq, startEq.sum - targetEq.sum, emptyList())
    return solutions
}
```

Таким образом, мы реализовали все шаги алгоритма 3, то есть реализовали алгоритм поиска всех кратчайших элементарных преобразований от λ до μ.

Численный эксперимент

Рассмотрим несколько примеров выполнения алгоритма поиска некоторых кратчайших последовательностей преобразований.

Пример 1:

Разбиения a = [5, 4, 4, 1] и b = [4, 3, 2, 1, 1], $a \geq b$.

Результат работы программы:

1:

Shift block from index 2 to 4: [5, 4, 3, 1, 1]

Remove top block from index 2: [5, 4, 2, 1, 1]

Remove top block from index 1: [5, 3, 2, 1, 1]

Remove top block from index 0: [4, 3, 2, 1, 1]

2:

Remove top block from index 2: [5, 4, 3, 1, 0]

Shift block from index 2 to 4: [5, 4, 2, 1, 1]

Remove top block from index 1: [5, 3, 2, 1, 1]

Remove top block from index 0: [4, 3, 2, 1, 1]

В результате работы программы алгоритм нашел два пути решения длины 4.

Пример 2:

Разбиения a = [12, 9, 7, 6, 1, 1] и b = [10, 9, 6, 3, 1, 1, 1], $a \geq b$.

Результат:

1:

Shift block from index 3 to 6: [12, 9, 7, 5, 1, 1, 1]

Remove top block from index 3: [12, 9, 7, 4, 1, 1, 1]

Remove top block from index 3: [12, 9, 7, 3, 1, 1, 1]

Remove top block from index 2: [12, 9, 6, 3, 1, 1, 1]

Remove top block from index 0: [11, 9, 6, 3, 1, 1, 1]

Remove top block from index 0: [10, 9, 6, 3, 1, 1, 1]

2:

Remove top block from index 3: [12, 9, 7, 5, 1, 1, 0]

Shift block from index 3 to 6: [12, 9, 7, 4, 1, 1, 1]

Remove top block from index 3: [12, 9, 7, 3, 1, 1, 1]

Remove top block from index 2: [12, 9, 6, 3, 1, 1, 1]

Remove top block from index 0: [11, 9, 6, 3, 1, 1, 1]

Remove top block from index 0: [10, 9, 6, 3, 1, 1, 1]

3:

Remove top block from index 3: [12, 9, 7, 5, 1, 1, 0]

Remove top block from index 3: [12, 9, 7, 4, 1, 1, 0]

Shift block from index 3 to 6: [12, 9, 7, 3, 1, 1, 1]

Remove top block from index 2: [12, 9, 6, 3, 1, 1, 1]

Remove top block from index 0: [11, 9, 6, 3, 1, 1, 1]

Remove top block from index 0: [10, 9, 6, 3, 1, 1, 1]

Получилось 3 пути решения длины 6.

Заключение

Целью курсовой работы являлась реализация алгоритма нахождения всех кратчайших последовательностей элементарных преобразований в решетке разбиений натуральных чисел.

В рамках данной курсовой работы были изучены основы теории разбиений натуральных чисел и её применение для решения задач по поиску кратчайших последовательностей элементарных преобразований разбиений.

На основе изученных концепций и методов был реализован алгоритм для поиска кратчайших последовательностей элементарных преобразований между двумя разбиениями на языке программирования Kotlin.

Реализованный алгоритм был протестирован на различных входных данных. Полученные результаты подтвердили корректность работы алгоритма и его способность находить кратчайшие последовательности преобразований.

Таким образом, курсовая работа позволила изучить основы теории разбиений и реализовать эффективный алгоритм для решения практических задач в данной области. Полученные результаты могут быть полезны для дальнейших исследований в области комбинаторики, теории чисел и связанных с ними дисциплин.

Литература

1. Andrews, G. E. (1976). The Theory of Partitions.
2. V.A. Baransky, T.A. Koroleva, The lattice of partitions of a positive integer.
3. В. А. Баранский, Т. А. Сеньчонок, О кратчайших последовательностях элементарных преобразований в решетке разбиений натуральных чисел