МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра САПР

ОТЧЁТ

По лабораторной работе №2
По дисциплине «Алгоритмы и структуры данных»

Студент гр. 3351 _____ Морозов А.А.
Преподаватель _____ Пестерев Д.О.

Санкт-Петербург
2025

ЦЕЛЬ РАБОТЫ

1. Реализовать следующие функции:
   - Перехода в цветовое пространство YCbCr из RGB
   - Даунсэмплинга матрицы цветового канала.
   - Разбиения изображения на блоки NxN. Если изображение делится на нецелое количество блоков, округлить их количество по вертикали/горизонтали вверх и заполнить неполные блоки выбранным значением (например, нулем)
   - Прямого и обратного DCT-II 2D для блока размера NxN. Удостовериться в обратимости.
   - Изменения матрицы квантования в зависимости от уровня сжатия.
   - Квантования и обратного преобразования матрицы ДКТ по заданной матрице квантования
   - Зигзаг обхода матрицы NxN
   - Разностного кодирования DC коэффициентов
   - Переменного кодирования разностей DC и AC коэффициентов.
   - RLE кодирования AC коэффициентов
   - Кодирования разностей DC коэффициентов и последовательностей Run/Size по таблице кодов Хаффмана и упаковки результата в байтовую строку.

2. Подготовить тестовые данные.
   - Lenna.png (https://en.wikipedia.org/wiki/Lenna#/media/File:Lenna_(test_image).png)
   - Нетривиальное цветное изображение размера 2048 на 2048 высокого качества.

3. Получить версии тестовых изображений в grayscale, чб с дизерингом, чб без дизеринга

4. На основе функций из пункта 3 реализовать JPEG-инспирированный компрессор со следующими этапами выполнения:
   - Переход в цветовое пространство YCbCr
   - Даунсэмплинг каналов Cb и Cr с коэффициентом 2 по каждой оси.
   - Разбиение на блоки (размер по умолчанию 8x8).
   - Применение двумерного дискретного косинусного преобразования (DCT-II 2D)
   - Квантование коэффициентов DCT по матрице квантования (143 стр. ITU-T81) и уровню качества
   - Разностное кодирование DC коэффициентов

- Переменное кодирование разностей DC коэффициентов и AC коэффициентов
- RLE AC коэффициентов
- Энтропийное кодирование табличными кодами Хаффмана (149-157 стр. ITU-T81, при желании использовать готовые спецификации таблицы из K.3.3)

5. Записать в файл в получившуюся байтовую строку и все необходимые метаданные (размер изображения, матрицы квантования, коды Хаффмана и тд).
6. Реализовать соответствующий декомпрессор. Удостовериться в корректной декомпрессии.
7. Для каждого изображения из пунктов 2-3 построить график зависимости размера сжатого от файла от коэффициента качества сжатия (шаг коэффициента не более 5)
8. Сжать тестовые изображения с уровнем качества 0, 20, 40, 60, 80, 100. Произвести декомпрессию, приложить получившиеся изображения к отчету.

Описание программы

Программное обеспечение – Microsoft Visual Studio 2022. Язык программирования – C++.

Теоретическая часть

Цветовое пространство YCbCr является одним из наиболее распространенных форматов представления цветовой информации в цифровой обработке изображений и видео. Оно было разработано для эффективной передачи и хранения цветной информации, особенно в контексте телевидения и видеосистем.

Компоненты YCbCr

1. **Y**: Этот компонент представляет яркость изображения. Он определяет, насколько светлым или темным будет пиксель, и воспринимается человеческим глазом как основная информация о свете. Y рассчитывается на основе значений RGB с учетом их восприятия человеком.
2. **Cb**: Этот компонент представляет разницу между синей составляющей и яркостью. Он указывает, насколько синий цвет преобладает в изображении.
3. **Cr**: Этот компонент представляет разницу уже между красной составляющей и яркостью. Он показывает, насколько красный цвет преобладает в изображении.

Преимущества YCbCr

- **Сжатие данных**: Поскольку человеческий глаз более чувствителен к изменениям яркости, чем к изменениям цвета, YCbCr позволяет применять методы сжатия, такие как субдискретизация (или можно просто downсэмплинг) цветовой информации (например 4:2:0 как будет представлено далее), что уменьшает объем данных без значительной потери качества.
- **Последующая обработка**: Разделение яркости и цветовой информации упрощает многие алгоритмы обработки изображений, такие как фильтрация, коррекция цвета и т.д.

Даунсэмплинг, а иначе говоря субдискретизация каналов Cb и Cr в цветовом пространстве YCbCr — это процесс уменьшения разрешения цветовой информации, который часто используется в цифровой обработке изображений и видео для оптимизации хранения и передачи данных. Этот метод особенно эффективен, поскольку человеческий глаз менее чувствителен к изменениям в цвете, чем к изменениям в яркости.

При даунсэмплинге с коэффициентом 2 по каждой оси (у нас обозначается как 4:2:0) разрешение уменьшается цветовых каналов в два раза как по горизонтали, так и по вертикали. Это означает, что для каждого блока 2x2 пикселей в каналах Y мы будем использовать только один пиксель в каналах Cb и Cr.

Преимущества даунсэмплинга

1. **Сжатие данных**: Даунсэмплинг позволяет значительно сократить объем информации, что особенно важно для видео и изображений с высоким разрешением.
2. **Сохранение качества**: Поскольку человеческий глаз менее чувствителен к изменениям в цвете, чем к изменениям в яркости, даунсэмплинг цветовых каналов может быть выполнен без заметной потери качества изображения.
3. **Упрощение обработки**: Меньшее количество данных в цветовых каналах упрощает алгоритмы обработки изображений, такие как фильтрация, сжатие и может быть другие операции.

Двумерное дискретное косинусное преобразование (DCT-II 2D) используется для преобразования изображения из пространственной области в частотную, что позволяет более эффективно представлять и сжимать информацию. DCT — это преобразование, которое преобразует изображение сумму косинусоидальных функций с различными частотами. DCT-II 2D применяется к двумерным массивам данных, таким как пиксели

изображения. Оно выполняется последовательно: сначала по строкам, а затем по столбцам. Ну ещё наоборот тоже можно.

Применение DCT-II 2D

1. **Сжатие изображений**: Одним из основных применений DCT-II 2D является сжатие изображений, например в формате JPEG. Преобразование изображения в частотную область позволяет выделить важные частоты, которые содержат большую часть визуальной информации, и отфильтровать менее значимые частоты, что приводит к уменьшению объема данных.
2. **Устойчивость к шуму**: DCT-II 2D помогает уменьшить влияние шума на изображение. При сжатии изображения в частотной области можно удалить высокочастотные компоненты, которые часто содержат шум, сохраняя при этом основные характеристики изображения.
3. **Видеокодирование**: DCT-II 2D также широко используется в видеокодировании, например в стандартах MPEG и H.264. Преобразование кадров видео в частотную область позволяет эффективно сжимать данные, что особенно важно для потоковой передачи и хранения видео.

Процесс DCT-II 2D

1. **Разделение изображения на блоки**: Обычно изображение разбивается на небольшие блоки, чтобы DCT-II 2D мог быть применен к каждому блоку отдельно.
2. **Применение DCT**: Преобразуем пространственные значения пикселей в частотные компоненты в каждом блоке.
3. **Квантование**: После применения DCT-II 2D полученные коэффициенты подвергаются квантованию, что позволяет уменьшить количество данных, сохраняя при этом визуально приемлемое качество изображения.
4. **Кодирование**: Квантованные коэффициенты кодируются с использованием методов сжатия, таких как кодирование Хаффмана, для дальнейшего уменьшения объема данных.

Квантование коэффициентов DCT позволяет уменьшить количество данных, необходимых для представления изображения, за счет удаления менее значимой информации, что в свою очередь влияет на качество изображения.

Основа

1. **Коэффициенты DCT**: После применения DCT к блоку пикселей 8x8, мы получаем 64 коэффициента, которые представляют частотные

компоненты изображения. Эти коэффициенты содержат информацию о яркости и цвете, где низкочастотные компоненты содержат основную информацию о структуре изображения, а высокочастотные компоненты содержат детали и текстуры.

2. **Матрица квантования**: Матрица квантования — это таблица, которая используется для определения, насколько сильно будут уменьшены коэффициенты DCT. Она содержит значения, которые делят соответствующие коэффициенты DCT, чтобы получить квантованные значения. Стандартная матрица квантования для Y-канала в JPEG выглядит следующим образом (для блока 8x8):

116 11 10 16 24 40 51 61
212 12 14 19 26 58 60 55
314 13 16 24 40 57 69 56
414 17 22 29 51 87 80 62
518 22 37 56 68 109 103 77
624 35 55 64 81 104 113 92
749 64 78 87 103 121 120 101
872 92 95 98 112 100 103 99

Процесс квантования

1. **Деление коэффициентов DCT на матрицу квантования**: Каждый коэффициент DCT делится на соответствующее значение в матрице квантования, и результат округляется до ближайшего целого числа. Это позволяет уменьшить значения коэффициентов, особенно для высокочастотных компонентов, которые менее заметны для человеческого глаза.
2. **Уровень качества**: Уровень качества (quality factor) влияет на матрицу квантования. При изменении уровня качества матрица может быть умножена на коэффициент, который изменяет степень квантования. Например при увеличении уровня качества матрица квантования уменьшается, что приводит к меньшему сжатию и лучшему качеству изображения. При понижении уровня качества матрица увеличивается, что приводит к большему сжатию и ухудшению качества.

Влияние на качество изображения

- **Высокий уровень качества**: При высоком уровне качества (например, 90-100) коэффициенты DCT будут менее квантованы, что приводит к меньшему потере информации и лучшему качеству изображения, но и большему объему данных.
- **Низкий уровень качества**: При низком уровне качества (например, 10-50) коэффициенты DCT будут более агрессивно квантованы, что

приводит к значительной потере информации и ухудшению качества изображения, но и к значительному уменьшению объема данных.

Разностное кодирование DC коэффициентов — это метод, используемый в сжатии изображений и видео для уменьшения объема данных, передаваемых или хранящихся. Этот метод особенно эффективен при кодировании изображений в формате JPEG и видео в формате MPEG.

Основа

1. **DC коэффициенты**: В дискретном косинусном преобразовании DC коэффициент представляет собой первый коэффициент в блоке DCT, который соответствует нулевой частоте. Он отражает среднюю яркость блока пикселей. В отличие от AC коэффициентов, которые представляют высокочастотные компоненты, DC коэффициенты содержат основную информацию о яркости изображения.
2. **Разностное кодирование**: Этот метод заключается в кодировании разностей между последовательными DC коэффициентами, а не самих значений. Это позволяет уменьшить количество бит, необходимых для представления данных, поскольку разности часто имеют меньшие значения и, следовательно, могут быть закодированы эффективнее.

Процесс разностного кодирования

1. **Получение DC коэффициентов**: После применения DCT к каждому блоку изображения извлекаются DC коэффициенты.
2. **Вычисление разностей**: Для каждого блока кроме первого вычисляется разность между текущим DC коэффициентом и предыдущим DC коэффициентом.
3. **Кодирование разностей**: Разности кодируются с использованием методов, например кодирование Хаффмана. Поскольку разности часто имеют меньшие значения, они могут быть закодированы с использованием меньшего количества бит, что приводит к уменьшению объема данных.

Преимущества разностного кодирования

1. **Сжатие данных**: Разностное кодирование позволяет значительно уменьшить объем данных, особенно в случаях, когда DC коэффициенты изменяются от блока к блоку незначительно. Это особенно актуально для изображений с плавными градиентами и небольшими изменениями яркости.
2. **Упрощение кодирования**: Кодирование разностей, а не абсолютных значений, позволяет использовать более простые и эффективные

схемы кодирования, что также способствует уменьшению объема данных.

3. **Снижение ошибок**: В случае ошибок в передаче данных, разностное кодирование может быть более устойчивым, поскольку ошибка в одном DC коэффициенте не приводит к значительным искажениям в последующих значениях.

Переменное кодирование разностей DC и AC коэффициентов — это метод, используемый в сжатии изображений и видео для эффективного представления данных, полученных после дискретного косинусного преобразования (DCT). Этот подход позволяет уменьшить объем данных, сохраняя при этом приемлемое качество изображения.

Преимущества переменного кодирования разностей

1. **Сжатие данных**: Переменное кодирование разностей позволяет значительно уменьшить объем данных, особенно в случаях, когда DC и AC коэффициенты изменяются незначительно от блока к блоку. Это особенно актуально для изображений с плавными градиентами и небольшими изменениями яркости.
2. **Эффективное представление**: Использование кодов переменной длины для представления AC коэффициентов позволяет более эффективно использовать доступное пространство, так как более частые значения могут быть представлены более короткими кодами.
3. **Устойчивость к ошибкам**: В случае ошибок в передаче данных, переменное кодирование разностей может быть более устойчивым, поскольку ошибка в одном коэффициенте не приводит к значительным искажениям в последующих значениях.

Преимущества RLE для AC коэффициентов

1. **Эффективное сжатие**: RLE особенно эффективно работает с изображениями, где много последовательных нулей, что часто происходит после DCT. Это позволяет значительно уменьшить объем данных.
2. **Простота реализации**: Алгоритм RLE прост в реализации и не требует сложных вычислений, что делает его подходящим для быстрого сжатия и декомпрессии.
3. **Устойчивость к ошибкам**: В случае ошибок в передаче данных, RLE может быть более устойчивым, поскольку ошибка в одном коде не приводит к значительным искажениям в последующих значениях.

Энтропийное кодирование с использованием таблиц Хаффмана является важным этапом в процессе сжатия изображений, особенно в

стандарте JPEG (ITU-T T.81). Этот метод позволяет эффективно представлять данные, используя переменные длины кодов, что приводит к значительному уменьшению объема данных без потери информации.

Преимущества кодирования Хаффмана

1. **Эффективность**: Кодирование Хаффмана позволяет значительно уменьшить объем данных, особенно в случаях, когда некоторые символы (коэффициенты) встречаются гораздо чаще других.
2. **Гибкость**: Таблицы Хаффмана могут быть адаптированы к различным наборам данных, что позволяет оптимизировать сжатие для конкретных изображений или типов изображений.
3. **Простота декодирования**: Декодирование данных, закодированных с использованием кодов Хаффмана, является простым и быстрым процессом, что делает его подходящим для реального времени.

Практическая часть

Я думаю следует начать с изображений, которые будут использоваться в ходе лабораторной работы. По техническому заданию у меня должно быть 2 начальных изображения: lenna.png размером 512 на 512 пикселей, а также любое нетривиальное изображение высокого разрешения - минимум 2048 на 2048 пикселей. В моём случае это картинка drum.png 3240 на 2160 пикселей. Вот они (Рис.1, Рис. 2)

Рис. 1 – lenna.png

Рис. 2 – drum.png

Также по техническому заданию было указано, что для каждого изображения необходимо несколько версий:

1. Классическая цветная
2. С оттенками серого
3. Чёрно-белая с дизерингом
4. Чёрно-белая без дизеринга

Каждое изображение было переведено в формат .raw, где каждый пиксель кодируется тремя байтами. Также для удобства сделал версии .png, чтобы можно было посмотреть как они выглядит. Прикладываю их (Рис. 3-8)

Рис. 3 – lenna.png grayscale

Рис. 4 – drum.png grayscale

Рис. 5 – lenna.png dithered

Рис. 6 – drum.png dithered

Рис. 7 – lenna.png no dither


Рис. 8 – drum.png no dither

Таким образом у нас получается 8 тестовых изображений. Вот как мы их переводим с помощью библиотеки Pillow:

- convert('RGB'): Конвертирует изображение в цветное.
- convert('L'): Конвертирует изображение в оттенки серого.
- convert('1'): Конвертирует изображение в черно-белое с дизерингом.
- point(lambda x: 255 if x >= 128 else 0, mode='1'): Применяет пороговое значение для создания черно-белого изображения без дизеринга.

Теперь пройдусь по основному коду, который я выполнял (слава аллаху) на c++. Было непросто, но всё возможно.

Изначально я собирался считывать байты из картинок .png напрямую. Для этого написал считыватель на Windows API, благо он затрагивался на курсе низкоуровневого программирования несколько ранее. В целом все работало, но я нашёл библиотеки stb_image.h и stb_image_write.h, которые работают на аналогичном принципе и использовал их, чтобы сократить код.

Перевод в YCbCr у меня вычисляет 3 компоненты: яркостную, синюю и красную цветоразностные составляющие. После перевода для цветоразностных составляющих добавляю 128. Это потому что классически их значения лежат в диапазоне от -128 до 127, а мне надо от 0 до 255. Таким образом получается, что 128 соответствует нейтральному цвету – нулю, а всё

остальное это сдвиги. При обратном преобразовании вычитаю 128, чтобы вернуть значения в исходный диапазон.

После перевода в YCbCr я сохраняю промежуточное переведённое изображение, а также монохромные изображения с трёх отдельных каналов. (Рис. 9-12)

Рис. 9 – YcbCr целиком

Рис. 10 – Канал Y

Рис. 11 – Канал Cb

Рис. 12 – Канал Cr

Далее по списку следует даунсэмплинг каналов Cb и Cr. Это надо для того, чтобы уменьшить разрешение цветоразностных компонент, поскольку человеческий глаз более чувствителен к компоненте яркости. В моём случае цветоразностные каналы (Cb и Cr) уменьшаются в 2 раза по вертикали и горизонтали. Схема 4:2:0 оставляет Y в полном разрешении, а Cb и Cr уменьшает в 2 раза.

Для каждого пикселя в уменьшенном изображении берётся соответствующий блок 2×2 в исходном изображении. Вычисляется среднее значение всех пикселей в этом блоке и записывается в результирующий пиксель. Таким образом мы уменьшаем итоговый вес изображения.

Разбиение на блоки происходит на этапе обработки каналов Y, Cb, Cr в compressImage(). Для канала яркости (Y) блоки создаются в полном разрешении:

```
int blockRowsY = (height + 7) / 8;  // Округление вверх
int blockColsY = (width + 7) / 8;
```

Для цветностных каналов (Cb, Cr) - в уменьшенном разрешении (после даунсэмплинга 4:2:0):
int blockRowsC = (Cb.size() + 7) / 8;
int blockColsC = (Cb[0].size() + 7) / 8;

При нехватке пикселей на границе изображения блок дополняется нулями.

Для преобразования DCT я использую вспомогательную функцию, которая вычисляет нормировочный коэффициент. Для частоты u=0 возвращает √(1/8), а для частоты u>0 возвращает √(2/8), что является стандартными весовыми коэффициентами из определения DCT.

В функции DCT я выполняю двумерное DCT, применяя одномерное DCT сначала к строкам, затем к столбцам промежуточного результата. В итоге получается матрица коэффициентов 8 на 8, где (0,0) - постоянная составляющая (DC-коэффициент), а остальные - частотные составляющие (AC-коэффициенты).

После DCT энергия изображения концентрируется в верхнем левом углу - низкие частоты. Высокочастотные коэффициенты могут быть отброшены или сильнее сжаты. При декодировании IDCT восстанавливает изображение с потерями, если были удалены высокочастотные компоненты. Это преобразование эффективно упаковывает информацию об изображении в небольшое число значимых коэффициентов, что делает его идеальным для сжатия.

Для квантования используется формала: block[y][x] = round(block[y][x] / (quantizationMatrix[y][x] * qualityFactor));. В моём коде каждый коэффициент DCT делится на соответствующее значение из матрицы квантования. Потом результат округляется до целого числа, что приводит к уменьшению точности коэффициентов и обнулению многих высокочастотных коэффициентов, которые после деления становятся близкими к нулю.

При обратном преобразовании каждый квантованный коэффициент умножается на соответствующее значение из матрицы квантования. Это приблизительно восстанавливает исходные значения DCT-коэффициентов. Так немного теряется информация.

Зигзаг-сканирование я использую для специфичного обхода блока 8 на 8. Это повышает эффективность сжатия за счёт группировки нулевых коэффициентов. Основная идея заключается в том, что после квантования большинство значимых, ненулевых, коэффициентов DCT сосредоточены в верхнем левом углу блока (низкие частоты). Зигзаг-сканирование переупорядочивает коэффициенты так, чтобы сначала шли низкочастотные коэффициенты, затем – высокочастотные, а в конце - длинные

последовательности нулей. Благодаря группировке нулей к ним потом можно применить RLE кодирование.

Что касается DC коэффициентов… DC-коэффициент, а иначе говоря постоянная составляющая в DCT-блоке 8×8 представляет собой среднюю яркость блока. DC-коэффициенты соседних блоков обычно близки по значению, а разница между соседними DC-коэффициентами часто мала. Это позволяет эффективно сжимать данные, кодируя только разницы.

В своём коде первый DC-коэффициент я оставляю без изменений, а для последующих вычисляю разницу с предыдущим значением. Потом возвращается массив разностей. В обратную сторону все работает аналогично: начинаем с первого значения. Для каждого следующего значения добавляем разницу к предыдущему и восстанавливаем исходные DC-коэффициенты.

В RLE кодировании я прохожу по всем коэффициентом блока уже в зигзаг-порядке. Считаем последовательные нули (в моем коде это ZeroRun). Когда встречаем ненулевой коэффициент, то формируется пара (количество_нулей, значение), а также сбрасывается счетчик нулей. При достижении 16 нулей подряд вставляется специальный маркер ZRL (15, 0) и продолжаем счет заново. В конце вставляется маркер EOB (0, 0). Ну короче говоря это самое обычное RLE кодирование, которое было разобрано в предыдущей лабораторной. Разве что ограничение в 16 бит вводится из-за таблиц хаффмана для jpeg. Там поле длины серии нулей (run) кодируется 4 битами.

Несколько слов по моему Хаффману. Для разных компонентов изображения, яркости и цветности, применяются отдельные оптимизированные таблицы кодов. Это учитывает особенности человеческого зрения, которое более чувствительно к изменениям яркости, чем цветности.

Для DC-коэффициентов вместо непосредственного кодирования значений используется разностное кодирование между соседними блоками. Сначала определяется категория величины разницы (количество необходимых бит), затем эта категория кодируется по Хаффману, а само значение передается в дополнительном коде. Такой подход эффективно сжимает плавно изменяющиеся области изображения.

Для AC-коэффициентов применяется комбинация RLE и Хаффмановского кодирования. Каждая пара (длина серии нулей, значение коэффициента) кодируется как единое целое с использованием специальных таблиц. Особое внимание уделено часто встречающимся комбинациям - они получают самые короткие коды. Для оптимизации введены специальные

маркеры: ZRL (16 нулей подряд) и EOB (окончание блока), позволяющие эффективно обрабатывать области с отсутствием высокочастотных компонент.

Результаты сжатия всех изображений я поместил в одну большую таблицу. По традиции прикладываю обычную версию и цветную, чтобы явно показать степень сжатия. (Рис. 13, Рис. 14)

Тип	Качество	Входной	Сжатый	Коэффициент
drum rgb	0	20995200	1134049	18,51348575
drum grayscale	0	20995200	1132850	18,53308028
drum dithered	0	20995200	1133281	18,52603194
drum no dither	0	20995200	1151790	18,22832287
lenna rgb	0	786432	42121	18,6707818
lenna grayscale	0	786432	42100	18,68009501
lenna dithered	0	786432	42131	18,66635019
lenna no dither	0	786432	43470	18,09137336
drum rgb	20	20995200	1355074	15,49376639
drum grayscale	20	20995200	1320236	15,90261135
drum dithered	20	20995200	2576987	8,147188946
drum no dither	20	20995200	1405548	14,93737674
lenna rgb	20	786432	51702	15,21086225
lenna grayscale	20	786432	50376	15,61124345
lenna dithered	20	786432	104436	7,530276916
lenna no dither	20	786432	60709	12,95412542
drum rgb	40	20995200	1499676	13,99982396
drum grayscale	40	20995200	1440127	14,57871424
drum dithered	40	20995200	3336946	6,291741011
drum no dither	40	20995200	1557891	13,47668097
lenna rgb	40	786432	58377	13,47160697
lenna grayscale	40	786432	55834	14,08518107
lenna dithered	40	786432	133370	5,89661843
lenna no dither	40	786432	71326	11,02588117
drum rgb	60	20995200	1798235	11,67544843
drum grayscale	60	20995200	1694962	12,38682637
drum dithered	60	20995200	3893970	5,391721046
drum no dither	60	20995200	1689423	12,42743824
lenna rgb	60	786432	64718	12,15167341
lenna grayscale	60	786432	61102	12,87080619
lenna dithered	60	786432	154487	5,090603093
lenna no dither	60	786432	80043	9,825118999
drum rgb	80	20995200	1941500	10,81390677
drum grayscale	80	20995200	1809725	11,60132064
drum dithered	80	20995200	4890746	4,292842033
drum no dither	80	20995200	1889654	11,11060543
lenna rgb	80	786432	78373	10,03447616
lenna grayscale	80	786432	72459	10,85347576
lenna dithered	80	786432	194439	4,044620678
lenna no dither	80	786432	93917	8,373691664
drum rgb	100	20995200	2656768	7,902534207
drum grayscale	100	20995200	2419724	8,676692052
drum dithered	100	20995200	7658051	2,741585294
drum no dither	100	20995200	2395095	8,765915339
lenna rgb	100	786432	159856	4,919627665
lenna grayscale	100	786432	139809	5,625045598
lenna dithered	100	786432	354613	2,217719035
lenna no dither	100	786432	129854	6,05627859

Рис. 13 – Таблица

Тип	Качество	Входной	Сжатый	Коэффициент
drum rgb	0	20995200	1134049	18,51348575
drum grayscale	0	20995200	1132850	18,53308028
drum dithered	0	20995200	1133281	18,52603194
drum no dither	0	20995200	1151790	18,22832287
lenna rgb	0	786432	42121	18,6707818
lenna grayscale	0	786432	42100	18,68009501
lenna dithered	0	786432	42131	18,66635019
lenna no dither	0	786432	43470	18,09137336
drum rgb	20	20995200	1355074	15,49376639
drum grayscale	20	20995200	1320236	15,90261135
drum dithered	20	20995200	2576987	8,147188946
drum no dither	20	20995200	1405548	14,93737674
lenna rgb	20	786432	51702	15,21086225
lenna grayscale	20	786432	50376	15,61124345
lenna dithered	20	786432	104436	7,530276916
lenna no dither	20	786432	60709	12,95412542
drum rgb	40	20995200	1499676	13,99982396
drum grayscale	40	20995200	1440127	14,57871424
drum dithered	40	20995200	3336946	6,291741011
drum no dither	40	20995200	1557891	13,47668097
lenna rgb	40	786432	58377	13,47160697
lenna grayscale	40	786432	55834	14,08518107
lenna dithered	40	786432	133370	5,89661843
lenna no dither	40	786432	71326	11,02588117
drum rgb	60	20995200	1798235	11,67544843
drum grayscale	60	20995200	1694962	12,38682637
drum dithered	60	20995200	3893970	5,391721046
drum no dither	60	20995200	1689423	12,42743824
lenna rgb	60	786432	64718	12,15167341
lenna grayscale	60	786432	61102	12,87080619
lenna dithered	60	786432	154487	5,090603093
lenna no dither	60	786432	80043	9,825118999
drum rgb	80	20995200	1941500	10,81390677
drum grayscale	80	20995200	1809725	11,60132064
drum dithered	80	20995200	4890746	4,292842033
drum no dither	80	20995200	1889654	11,11060543
lenna rgb	80	786432	78373	10,03447616
lenna grayscale	80	786432	72459	10,85347576
lenna dithered	80	786432	194439	4,044620678
lenna no dither	80	786432	93917	8,373691664
drum rgb	100	20995200	2656768	7,902534207
drum grayscale	100	20995200	2419724	8,676692052
drum dithered	100	20995200	7658051	2,741585294
drum no dither	100	20995200	2395095	8,765915339
lenna rgb	100	786432	159856	4,919627665
lenna grayscale	100	786432	139809	5,625045598
lenna dithered	100	786432	354613	2,217719035
lenna no dither	100	786432	129854	6,05627859

Рис. 14 – Цветная таблица

	...-5
	5-8
	8-12
	12-16
	16-...

Рис. 15 – Распределение

Какие же выводы я могу сделать, имея на руках все эти данные? В первую очередь самое логичное, что чем выше уровень качества, тем хуже сжимается. Это основа компрессора с потерями. **Большие изображения** (в моём случае drum) **сжимаются эффективнее** благодаря избыточности данных. Grayscale версии показывают чуть лучшие результаты, чем RGB.

На качестве 80-100 для dithered версий коэффициент ухудшается в 3-4 раза. Это связано с агрессивным дизерингом, создающим высокочастотный шум.

Для каждого изображения построил графики зависимости размера сжатого от файла от коэффициента качества сжатия (шаг коэффициента не более 5). Получилось 8 графиков, в каждом из которых сочетается 3 оси… Сейчас будет немного сложно объяснить, но я попробую:
Синяя сплошная линяя показывает зависимость размера сжатого файла от качества (левая вертикальная ось). Красная пунктирная линия показывает зависимость коэффициента сжатия от качества (правая вертикальная ось).

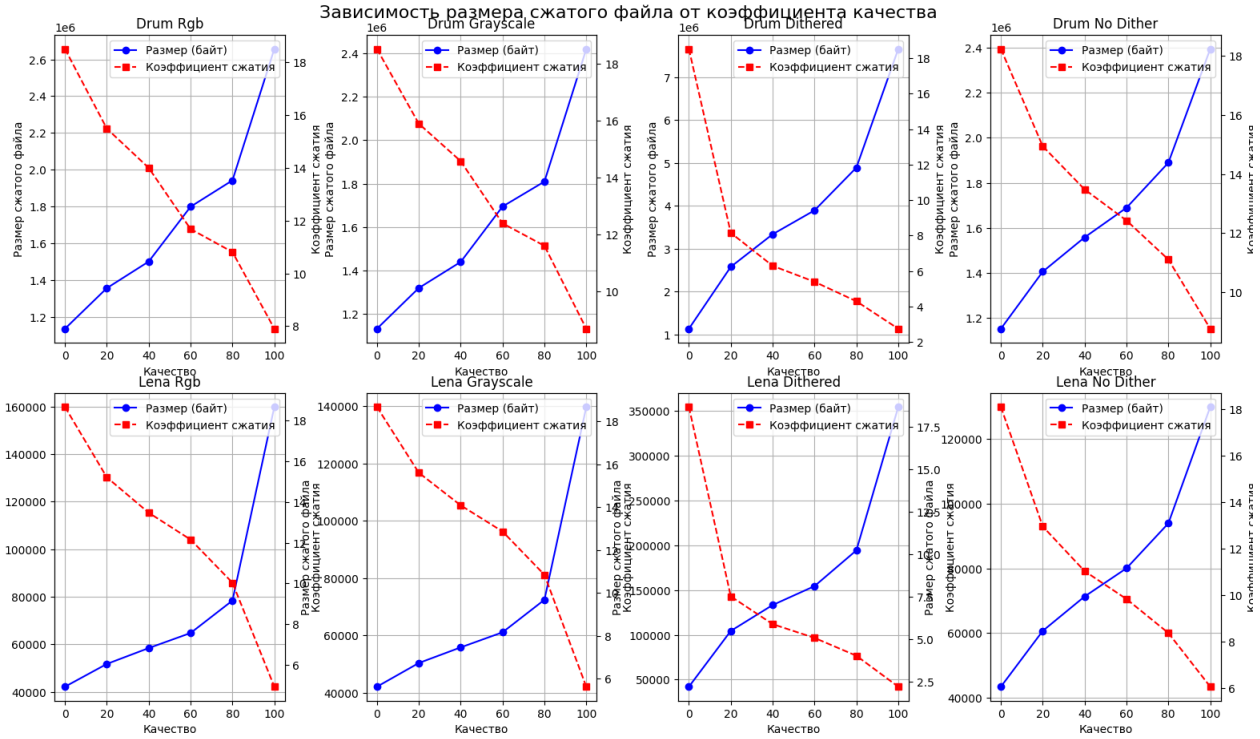

Рис. 16 - Графики

Также прикладываю несколько изображений в ходе работы компрессора. (Рис. 17-24)

Рис. 17 – drum с дизерингом в качестве 20

Рис. 18 – drum цветной в качестве 60

Рис. 19 – drum цветной в качестве 80

Рис. 20 – lenna цветная в качестве 20

Рис. 21 – lenna цветная в качестве 60

Рис. 22 – lenna с дизерингом в качестве 20

Рис. 23 – lenna в оттенках серого с качеством 20

Рис. 24 – lenna без дизеринга в качестве 40

Вывод о проделанной лабораторной работе

В ходе выполнения лабораторной работы был разработан JPEG-инспирированный компрессор с множеством этапов выполнения.

Результаты показали ожидаемую зависимость между уровнем качества сжатия и степенью компрессии — чем выше устанавливалось качество, тем меньше оказывался коэффициент сжатия. При нулевом уровне качества все изображения демонстрировали наилучшие показатели сжатия (коэффициенты 18-19 для изображений drum и 18-19 для lena), что соответствует теоретическим ожиданиям для алгоритмов сжатия с потерями.

Наибольшую эффективность компрессор показал при работе с grayscale-изображениями без дизеринга, где сохранялись плавные переходы яркости. Это подтверждает принцип JPEG о лучшей сжимаемости областей с низкочастотными компонентами. В то же время изображения с применением дизеринга, содержащие искусственно созданные высокочастотные компоненты для имитации градаций серого, показали

наихудшие результаты — их коэффициенты сжатия при высоких качествах падали в 3-4 раза по сравнению с оригиналами.

Особый интерес представляет сравнение работы алгоритма на изображениях разного разрешения. Большие изображения drum (3240×2160) сжимались с лучшими коэффициентами, чем маленькие lena (512×512), что объясняется большей избыточностью данных в высокоразрешенных изображениях. При этом кривые зависимости размера от качества для всех изображений сохраняли схожую форму, что свидетельствует о стабильности работы алгоритма. Спасибо.

Полный код программы (Внимание! 2000 строк! Тут почти все наработки!)

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <vector>
#include <cstdint>
#include <algorithm>
#include <fstream>
#include <unordered_map>
#include <sstream>
#include <bitset>
#include <array>
#include <utility>
#include <map>

#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"

using namespace std;

const double M_PI = 3.14159265358979323846;

// Матрицы квантования (стандартные JPEG)
const int QY[8][8] = {
   {16, 11, 10, 16, 24, 40, 51, 61},
   {12, 12, 14, 19, 26, 58, 60, 55},
   {14, 13, 16, 24, 40, 57, 69, 56},
   {14, 17, 22, 29, 51, 87, 80, 62},
```

```
    {18, 22, 37, 56, 68, 109, 103, 77},
    {24, 35, 55, 64, 81, 104, 113, 92},
    {49, 64, 78, 87, 103, 121, 120, 101},
    {72, 92, 95, 98, 112, 100, 103, 99}
};

const int QC[8][8] = {
    {17, 18, 24, 47, 99, 99, 99, 99},
    {18, 21, 26, 66, 99, 99, 99, 99},
    {24, 26, 56, 99, 99, 99, 99, 99},
    {47, 66, 99, 99, 99, 99, 99, 99},
    {99, 99, 99, 99, 99, 99, 99, 99},
    {99, 99, 99, 99, 99, 99, 99, 99},
    {99, 99, 99, 99, 99, 99, 99, 99},
    {99, 99, 99, 99, 99, 99, 99, 99}
};

// ------------------------------------------------------

const unordered_map<int, string> HA_Y_DC_TABLE = {
    {0, "00"},
    {1, "010"},
    {2, "011"},
    {3, "100"},
    {4, "101"},
    {5, "110"},
    {6, "1110"},
    {7, "11110"},
    {8, "111110"},
    {9, "1111110"},
    {10, "11111110"},
    {11, "111111110"}
};

const unordered_map<int, string> HA_C_DC_TABLE = {
    {0, "00"},
    {1, "01"},
    {2, "10"},
    {3, "110"},
    {4, "1110"},
    {5, "11110"},
    {6, "111110"},
```

```
    {7, "1111110"},
    {8, "11111110"},
    {9, "111111110"},
    {10, "1111111110"},
    {11, "11111111110"}
};

const map<pair<int, int>, string> HA_Y_AC_TABLE = {
    {{0, 0}, "1010"},
    {{0, 1}, "00"},
    {{0, 2}, "01"},
    {{0, 3}, "100"},
    {{0, 4}, "1011"},
    {{0, 5}, "11010"},
    {{0, 6}, "1111000"},
    {{0, 7}, "11111000"},
    {{0, 8}, "1111110110"},
    {{0, 9}, "1111111110000010"},
    {{0, 10}, "1111111110000011"},
    {{1, 1}, "1100"},
    {{1, 2}, "11011"},
    {{1, 3}, "1111001"},
    {{1, 4}, "111110110"},
    {{1, 5}, "11111110110"},
    {{1, 6}, "1111111110000100"},
    {{1, 7}, "1111111110000101"},
    {{1, 8}, "1111111110000110"},
    {{1, 9}, "1111111110000111"},
    {{1, 10}, "1111111110001000"},
    {{2, 1}, "11100"},
    {{2, 2}, "11111001"},
    {{ 2, 3}, "1111110111"},
    {{2, 4}, "111111110100"},
    {{2, 5}, "1111111110001001"},
    {{2, 6}, "1111111110001010"},
    {{2, 7}, "1111111110001011"},
    {{2, 8}, "1111111110001100"},
    {{2, 9}, "1111111110001101"},
    {{2, 10}, "1111111110001110"},
    {{3, 1}, "111010"},
    {{3, 2}, "111110111"},
    {{3, 3}, "111111110101"},
```

{{3, 4}, "1111111110001111"},
{{3, 5}, "1111111110010000"},
{{3, 6}, "1111111110010001"},
{{3, 7}, "1111111110010010"},
{{3, 8}, "1111111110010011"},
{{3, 9}, "1111111110010100"},
{{3, 10}, "1111111110010101"},
{{4, 1}, "111011"},
{{4, 2}, "1111111000"},
{{4, 3}, "1111111110010110"},
{{4, 4}, "1111111110010111"},
{{4, 5}, "1111111110011000"},
{{4, 6}, "1111111110011001"},
{{4, 7}, "1111111110011010"},
{{4, 8}, "1111111110011011"},
{{4, 9}, "1111111110011100"},
{{4, 10}, "1111111110011101"},
{{5, 1}, "1111010"},
{{5, 2}, "11111110111"},
{{5, 3}, "1111111110011110"},
{{5, 4}, "1111111110011111"},
{{5, 5}, "1111111110100000"},
{{5, 6}, "1111111110100001"},
{{5, 7}, "1111111110100010"},
{{5, 8}, "1111111110100011"},
{{5, 9}, "1111111110100100"},
{{5, 10}, "1111111110100101"},
{{6, 1}, "1111011"},
{{6, 2}, "111111110110"},
{{6, 3}, "1111111110100110"},
{{6, 4}, "1111111110100111"},
{{6, 5}, "1111111110101000"},
{{6, 6}, "1111111110101001"},
{{6, 7}, "1111111110101010"},
{{6, 8}, "1111111110101011"},
{{6, 9}, "1111111110101100"},
{{6, 10}, "1111111110101101"},
{{7, 1}, "11111010"},
{{7, 2}, "111111110111"},
{{7, 3}, "1111111110101110"},
{{7, 4}, "1111111110101111"},
{{7, 5}, "1111111110110000"},

{{7, 6}, "1111111110110001"},
{{7, 7}, "1111111110110010"},
{{7, 8}, "1111111110110011"},
{{7, 9}, "1111111110110100"},
{{7, 10}, "1111111110110101"},
{{8, 1}, "111111000"},
{{8, 2}, "111111111000000"},
{{8, 3}, "1111111110110110"},
{{8, 4}, "1111111110110111"},
{{8, 5}, "1111111110111000"},
{{8, 6}, "1111111110111001"},
{{8, 7}, "1111111110111010"},
{{8, 8}, "1111111110111011"},
{{8, 9}, "1111111110111100"},
{{8, 10}, "1111111110111101"},
{{9, 1}, "111111001"},
{{9, 2}, "1111111110111110"},
{{9, 3}, "1111111110111111"},
{{ 9, 4}, "1111111111000000"},
{{9, 5}, "1111111111000001"},
{{9, 6}, "1111111111000010"},
{{9, 7}, "1111111111000011"},
{{9, 8}, "1111111111000100"},
{{9, 9}, "1111111111000101"},
{{9, 10}, "1111111111000110"},
{{10, 1}, "111111010"},
{{10, 2}, "1111111111000111"},
{{10, 3}, "1111111111001000"},
{{10, 4}, "1111111111001001"},
{{10, 5}, "1111111111001010"},
{{10, 6}, "1111111111001011"},
{{10, 7}, "1111111111001100"},
{{10, 8}, "1111111111001101"},
{{10, 9}, "1111111111001110"},
{{10, 10}, "1111111111001111"},
{{11, 1}, "1111111001"},
{{11, 2}, "1111111111010000"},
{{11, 3}, "1111111111010001"},
{{11, 4}, "1111111111010010"},
{{11, 5}, "1111111111010011"},
{{11, 6}, "1111111111010100"},
{{11, 7}, "1111111111010101"},

{{11, 8}, "1111111111010110"},
{{11, 9}, "1111111111010111"},
{{11, 10}, "1111111111011000"},
{{12, 1}, "1111111010"},
{{12, 2}, "1111111111011001"},
{{12, 3}, "1111111111011010"},
{{12, 4}, "1111111111011011"},
{{12, 5}, "1111111111011100"},
{{12, 6}, "1111111111011101"},
{{12, 7}, "1111111111011110"},
{{12, 8}, "1111111111011111"},
{{12, 9}, "1111111111100000"},
{{12, 10}, "1111111111100001"},
{{13, 1}, "11111111000"},
{{13, 2}, "1111111111100010"},
{{13, 3}, "1111111111100011"},
{{13, 4}, "1111111111100100"},
{{13, 5}, "1111111111100101"},
{{13, 6}, "1111111111100110"},
{{13, 7}, "1111111111100111"},
{{13, 8}, "1111111111101000"},
{{13, 9}, "1111111111101001"},
{{13, 10}, "1111111111101010"},
{{14, 1}, "1111111111101011"},
{{14, 2}, "1111111111101100"},
{{14, 3}, "1111111111101101"},
{{14, 4}, "1111111111101110"},
{{14, 5}, "1111111111101111"},
{{14, 6}, "1111111111110000"},
{{14, 7}, "1111111111110001"},
{{14, 8}, "1111111111110010"},
{{14, 9}, "1111111111110011"},
{{14, 10}, "1111111111110100"},
{{15, 0}, "11111111001"},
{{15, 1}, "1111111111110101"},
{{15, 2}, "1111111111110110"},
{{15, 3}, "1111111111110111"},
{{15, 4}, "1111111111111000"},
{{15, 5}, "1111111111111001"},
{{15, 6}, "1111111111111010"},
{{15, 7}, "1111111111111011"},
{{15, 8}, "1111111111111100"},

```
    {{15, 9}, "1111111111111101"},
    {{15, 10}, "1111111111111110"}
};

const map<pair<int, int>, string> HA_C_AC_TABLE = {
    {{0, 0}, "00"},
    {{0, 1}, "01"},
    {{0, 2}, "100"},
    {{0, 3}, "1010"},
    {{0, 4}, "11000"},
    {{0, 5}, "11001"},
    {{0, 6}, "111000"},
    {{0, 7}, "1111000"},
    {{0, 8}, "111110100"},
    {{0, 9}, "1111110110"},
    {{0, 10}, "111111110100"},
    {{1, 1}, "1011"},
    {{1, 2}, "111001"},
    {{1, 3}, "11110110"},
    {{1, 4}, "111110101"},
    {{1, 5}, "11111110110"},
    {{1, 6}, "111111110101"},
    {{1, 7}, "1111111110001000"},
    {{1, 8}, "1111111110001001"},
    {{1, 9}, "1111111110001010"},
    {{1, 10}, "1111111110001011"},
    {{2, 1}, "11010"},
    {{2, 2}, "11110111"},
    {{2, 3}, "1111110111"},
    {{2, 4}, "111111110110"},
    {{2, 5}, "111111111000010"},
    {{2, 6}, "1111111110001100"},
    {{2, 7}, "1111111110001101"},
    {{2, 8}, "1111111110001110"},
    {{2, 9}, "1111111110001111"},
    {{2, 10}, "1111111110010000"},
    {{3, 1}, "11011"},
    {{3, 2}, "11111000"},
    {{3, 3}, "1111111000"},
    {{3, 4}, "1111111110010001"},
    {{3, 5}, "1111111110010010"},
    {{3, 6}, "1111111110010011"},
```

{{3, 7}, "1111111110010100"},
{{3, 8}, "1111111110010101"},
{{3, 9}, "1111111110010110"},
{{3, 10}, "1111111110010111"},
{{4, 1}, "111010"},
{{4, 2}, "111110110"},
{{4, 3}, "1111111110011000"},
{{4, 4}, "1111111110011001"},
{{4, 5}, "1111111110011010"},
{{4, 6}, "1111111110011011"},
{{4, 7}, "1111111110011100"},
{{4, 8}, "1111111110011101"},
{{4, 9}, "1111111110011110"},
{{4, 10}, "1111111110011111"},
{{5, 1}, "111011"},
{{5, 2}, "1111111001"},
{{5, 3}, "1111111110100000"},
{{5, 4}, "1111111110100001"},
{{5, 5}, "1111111110100010"},
{{5, 6}, "1111111110100011"},
{{5, 7}, "1111111110100100"},
{{5, 8}, "1111111110100101"},
{{5, 9}, "1111111110100110"},
{{5, 10}, "1111111110100111"},
{{6, 1}, "1111001"},
{{6, 2}, "11111110111"},
{{6, 3}, "1111111110101000"},
{{6, 4}, "1111111110101001"},
{{6, 5}, "1111111110101010"},
{{6, 6}, "1111111110101011"},
{{6, 7}, "1111111110101100"},
{{6, 8}, "1111111110101101"},
{{6, 9}, "1111111110101110"},
{{6, 10}, "1111111110101111"},
{{7, 1}, "1111010"},
{{7, 2}, "111111110111"},
{{7, 3}, "1111111110110000"},
{{7, 4}, "1111111110110001"},
{{7, 5}, "1111111110110010"},
{{7, 6}, "1111111110110011"},
{{7, 7}, "1111111110110100"},
{{7, 8}, "1111111110110101"},

{{7, 9}, "1111111110110110"},
{{7, 10}, "1111111110110111"},
{{8, 1}, "11111001"},
{{8, 2}, "1111111110111000"},
{{8, 3}, "1111111110111001"},
{{8, 4}, "1111111110111010"},
{{8, 5}, "1111111110111011"},
{{8, 6}, "1111111110111100"},
{{8, 7}, "1111111110111101"},
{{8, 8}, "1111111110111110"},
{{8, 9}, "1111111110111111"},
{{8, 10}, "1111111111000000"},
{{9, 1}, "111110111"},
{{9, 2}, "1111111111000001"},
{{9, 3}, "1111111111000010"},
{{9, 4}, "1111111111000011"},
{{9, 5}, "1111111111000100"},
{{9, 6}, "1111111111000101"},
{{9, 7}, "1111111111000110"},
{{9, 8}, "1111111111000111"},
{{9, 9}, "1111111111001000"},
{{9, 10}, "1111111111001001"},
{{10, 1}, "111111000"},
{{10, 2}, "1111111111001010"},
{{10, 3}, "1111111111001011"},
{{10, 4}, "1111111111001100"},
{{10, 5}, "1111111111001101"},
{{10, 6}, "1111111111001110"},
{{10, 7}, "1111111111001111"},
{{10, 8}, "1111111111010000"},
{{10, 9}, "1111111111010001"},
{{10, 10}, "1111111111010010"},
{{11, 1}, "111111001"},
{{11, 2}, "1111111111010011"},
{{11, 3}, "1111111111010100"},
{{11, 4}, "1111111111010101"},
{{11, 5}, "1111111111010110"},
{{11, 6}, "1111111111010111"},
{{11, 7}, "1111111111011000"},
{{11, 8}, "1111111111011001"},
{{11, 9}, "1111111111011010"},
{{11, 10}, "1111111111011011"},

```
    {{12, 1}, "111111010"},
    {{12, 2}, "1111111111011100"},
    {{12, 3}, "1111111111011101"},
    {{12, 4}, "1111111111011110"},
    {{12, 5}, "1111111111011111"},
    {{12, 6}, "1111111111100000"},
    {{12, 7}, "1111111111100001"},
    {{12, 8}, "1111111111100010"},
    {{12, 9}, "1111111111100011"},
    {{12, 10}, "1111111111100100"},
    {{13, 1}, "11111110010"},
    {{13, 2}, "1111111111100101"},
    {{13, 3}, "1111111111100110"},
    {{13, 4}, "1111111111100111"},
    {{13, 5}, "1111111111101000"},
    {{13, 6}, "1111111111101001"},
    {{13, 7}, "1111111111101010"},
    {{13, 8}, "1111111111101011"},
    {{13, 9}, "1111111111101100"},
    {{13, 10}, "1111111111101101"},
    {{14, 1}, "1111111010"},
    {{14, 2}, "1111111111101110"},
    {{14, 3}, "1111111111101111"},
    {{14, 4}, "1111111111110000"},
    {{14, 5}, "1111111111110001"},
    {{14, 6}, "1111111111110010"},
    {{14, 7}, "1111111111110011"},
    {{14, 8}, "1111111111110100"},
    {{14, 9}, "1111111111110101"},
    {{14, 10}, "1111111111110110"},
    {{15, 0}, "11111111000"},
    {{15, 1}, "1111111111110111"},
    {{15, 2}, "1111111111111000"},
    {{15, 3}, "1111111111111001"},
    {{15, 4}, "1111111111111010"},
    {{15, 5}, "1111111111111011"},
    {{15, 6}, "1111111111111100"},
    {{15, 7}, "1111111111111101"},
    {{15, 8}, "1111111111111110"},
    {{15, 9}, "1111111111111111"},
    {{15, 10}, "111111111000000"}
};
```

```
// ----------------------------------------------------

// Структура для записи битового потока
struct BitStream {
   vector<uint8_t> data;
   uint8_t buffer = 0;
   int bitCount = 0;
};

// Запись одного бита в поток
void writeBit(BitStream* stream, bool bit) {
   stream->buffer |= (bit << (7 - stream->bitCount));
   stream->bitCount++;
   if (stream->bitCount == 8) {
      stream->data.push_back(stream->buffer);
      stream->buffer = 0;
      stream->bitCount = 0;
   }
}

// Запись строки битов в поток
void writeBits(BitStream* stream, const string& bits) {
   for (char bit : bits) {
      writeBit(stream, bit == '1');
   }
}

// Запись байта в поток
void writeByte(BitStream* stream, uint8_t byte) {
   if (stream->bitCount == 0) {
      stream->data.push_back(byte);
   }
   else {
      for (int i = 7; i >= 0; i--) {
         writeBit(stream, (byte >> i) & 1);
      }
   }
}

// Получение данных из потока с завершением последнего байта
vector<uint8_t> getData(BitStream* stream) {
```

```
    if (stream->bitCount > 0) {
        stream->data.push_back(stream->buffer);
    }
    return stream->data;
}

// Структура для чтения битового потока
struct BitStreamReader {
    const vector<uint8_t>& data;
    size_t bytePos = 0;
    int bitPos = 0;
    bool eof = false;
};

// Инициализация ридера
BitStreamReader createBitStreamReader(const vector<uint8_t>& input) {
    return { input, 0, 0, false };
}

// Чтение одного бита
bool readBit(BitStreamReader* reader) {
    if (reader->eof || reader->bytePos >= reader->data.size()) {
        reader->eof = true;
        return false;
    }
    bool bit = (reader->data[reader->bytePos] >> (7 - reader->bitPos)) & 1;
    reader->bitPos++;
    if (reader->bitPos == 8) {
        reader->bytePos++;
        reader->bitPos = 0;
    }
    return bit;
}

// Чтение байта
uint8_t readByte(BitStreamReader* reader) {
    if (reader->eof) return 0;

    if (reader->bitPos == 0) {
        if (reader->bytePos >= reader->data.size()) {
            reader->eof = true;
            return 0;
```

```
        }
        return reader->data[reader->bytePos++];
    }
    else {
        uint8_t byte = 0;
        for (int i = 0; i < 8; i++) {
            byte |= readBit(reader) << (7 - i);
            if (reader->eof) break;
        }
        return byte;
    }
}

// Конец файла?
bool isEof(BitStreamReader* reader) {
    return reader->eof;
}

// Загрузка изображения из .raw формата (3 байта на пиксель в моем случае)
vector<vector<vector<float>>> loadRawImage(const char* filename, int width, int
height) {
    ifstream file(filename, ios::binary);
    if (!file) {
        cerr << "Error: Cannot open RAW file " << filename << endl;
        exit(1);
    }

    // Вычисляем размер файла и проверяем соответствие
    file.seekg(0, ios::end);
    size_t fileSize = file.tellg();
    file.seekg(0, ios::beg);

    if (fileSize != width * height * 3) {
        cerr << "Error: RAW file size doesn't match dimensions" << endl;
        exit(1);
    }

    vector<vector<vector<float>>>    image(height,    vector<vector<float>>(width,
vector<float>(3)));
    vector<uint8_t> buffer(width * height * 3);

    file.read(reinterpret_cast<char*>(buffer.data()), buffer.size());
```

```
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int idx = (y * width + x) * 3;
            image[y][x][0] = buffer[idx]; // R
            image[y][x][1] = buffer[idx+1]; // G
            image[y][x][2] = buffer[idx+2]; // B
        }
    }

    return image;
}

// Сохранение изображения в .raw формат
void saveRawImage(const char* filename, const vector<vector<vector<float>>>&
image) {
    int height = image.size();
    if (height == 0) return;
    int width = image[0].size();

    ofstream file(filename, ios::binary);
    if (!file) {
        cerr << "Error: Cannot create RAW file " << filename << endl;
        return;
    }

    vector<uint8_t> buffer(width * height * 3);

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int idx = (y * width + x) * 3;
            buffer[idx] = static_cast<uint8_t>(max(0.0f, min(255.0f, image[y][x][0])));
            buffer[idx+1]        =        static_cast<uint8_t>(max(0.0f,        min(255.0f,
image[y][x][1])));
            buffer[idx+2]        =        static_cast<uint8_t>(max(0.0f,        min(255.0f,
image[y][x][2])));
        }
    }

    file.write(reinterpret_cast<const char*>(buffer.data()), buffer.size());
}
```

```
void writeCompressedToFile(const string& filename, const vector<uint8_t>& compressedData, int width, int height, int quality) {
    ofstream outFile(filename, ios::binary);
    if (!outFile) {
        cerr << "Error: Cannot create file " << filename << endl;
        return;
    }

    // Записываем заголовок
    outFile.put(width >> 8); // Старший байт ширины
    outFile.put(width & 0xFF); // Младший байт ширины
    outFile.put(height >> 8); // Старший байт высоты
    outFile.put(height & 0xFF); // Младший байт высоты
    outFile.put(quality); // Качество (1 байт)

    // Записываем матрицы квантования
    for (int y = 0; y < 8; y++) {
        for (int x = 0; x < 8; x++) {
            outFile.put(QY[y][x]);
        }
    }
    for (int y = 0; y < 8; y++) {
        for (int x = 0; x < 8; x++) {
            outFile.put(QC[y][x]);
        }
    }

    // Записываем сжатые данные
    outFile.write(reinterpret_cast<const char*>(compressedData.data()), compressedData.size());
    outFile.close();
}

vector<uint8_t> readCompressedFromFile(const string& filename, int& width, int& height, int& quality) {
    ifstream inFile(filename, ios::binary);
    if (!inFile) {
        cerr << "Error: Cannot open file " << filename << endl;
        return {};
    }

    // Читаем заголовок
```

```
    width = (inFile.get() << 8) | inFile.get();
    height = (inFile.get() << 8) | inFile.get();
    quality = inFile.get();

    // Пропускаем матрицы квантования
    inFile.seekg(128, ios::cur); // 64 + 64 байт

    // Читаем сжатые данные
    vector<uint8_t>                compressedData((istreambuf_iterator<char>(inFile)),
istreambuf_iterator<char>());
    inFile.close();

    return compressedData;
}

// Кодирование DC коэффициента
string encodeHuffmanDC(int value, bool isLuminance) {
    // Определяем категорию DC коэффициента
    int category = 0;
    if (value != 0) {
        category = static_cast<int>(log2(abs(value))) + 1;
    }

    // Получаем код Хаффмана для категории
    const auto& table = isLuminance ? HA_Y_DC_TABLE : HA_C_DC_TABLE;
    auto it = table.find(category);
    if (it == table.end()) {
        cerr << "Error: DC category " << category << " not found in Huffman table" <<
endl;
        return "";
    }

    // Добавляем биты значения VLI
    string code = it->second;
    if (category > 0) {
        int vli = value > 0 ? value : (value + (1 << category) - 1);
        code += bitset<32>(vli).to_string().substr(32 - category);
    }

    return code;
}
```

```
// Функция для кодирования AC коэффициентов
string encodeAC(int run, int size, int value, bool isLuminance) {
   if (run == 0 && size == 0) { // EOB
      return isLuminance ? HA_Y_AC_TABLE.at({ 0, 0 }) : HA_C_AC_TABLE.at({ 0, 0
});
   }

   if (run == 15 && size == 0) { // ZRL
      return isLuminance ? HA_Y_AC_TABLE.at({ 15, 0 }) : HA_C_AC_TABLE.at({ 15,
0 });
   }

   string code = isLuminance ? HA_Y_AC_TABLE.at({ run, size }) :
HA_C_AC_TABLE.at({ run, size });
   if (size > 0) {
      if (value < 0) {
         value = (1 << size) - 1 + value;
      }
      bitset<16> bits(value);
      code += bits.to_string().substr(16 - size, size);
   }
   return code;
}

// RLE кодирование
vector<pair<int, int>> rleEncode(const vector<int>& block) {
   vector<pair<int, int>> rle;
   int zeroRun = 0;

   for (size_t i = 0; i < block.size(); i++) {
      if (block[i] == 0) {
         zeroRun++;
         if (zeroRun == 16) {
            rle.emplace_back(15, 0); // ZRL (Zero Run Length для хаффмана)
            zeroRun = 0;
         }
      }
      else {
         rle.emplace_back(zeroRun, block[i]);
         zeroRun = 0;
      }
   }
```

```
    // Добавляем EOB (End Of Block)
    rle.emplace_back(0, 0);
    return rle;
}

// RLE декодирование
vector<int> rleDecode(const vector<pair<int, int>>& rle) {
    vector<int> block;

    for (const auto& pair : rle) {
        int run = pair.first;
        int value = pair.second;

        if (run == 0 && value == 0) { // EOB
            // Заполняем оставшиеся нули
            while (block.size() < 64) {
                block.push_back(0);
            }
            break;
        }
        else if (run == 15 && value == 0) { // ZRL
            for (int i = 0; i < 16; i++) {
                block.push_back(0);
            }
        }
        else {
            for (int i = 0; i < run; i++) {
                block.push_back(0);
            }
            block.push_back(value);
        }
    }

    // Обрезаем до 64 коэффициентов на случай, если данных больше
    if (block.size() > 64) {
        block.resize(64);
    }

    return block;
}
```

```
// Разностное кодирование DC коэффициентов
vector<int> encodeDCDifferences(const vector<int>& dcValues) {
    if (dcValues.empty()) return {};

    vector<int> diffValues(dcValues.size());
    diffValues[0] = dcValues[0]; // Первый коэффициент остается без изменений

    for (size_t i = 1; i < dcValues.size(); i++) {
        diffValues[i] = dcValues[i] - dcValues[i - 1];
    }

    return diffValues;
}

// Декодирование разностей DC коэффициентов
vector<int> decodeDCDifferences(const vector<int>& diffValues) {
    if (diffValues.empty()) return {};

    vector<int> dcValues(diffValues.size());
    dcValues[0] = diffValues[0]; // Первый коэффициент остается без изменений

    for (size_t i = 1; i < diffValues.size(); i++) {
        dcValues[i] = dcValues[i - 1] + diffValues[i];
    }

    return dcValues;
}

// Кодирование AC коэффициента
string encodeHuffmanAC(int run, int size, int value, bool isLuminance) {
    // Специальные случаи: EOB и ZRL
    if (run == 0 && value == 0) { // EOB
        const auto& table = isLuminance ? HA_Y_AC_TABLE : HA_C_AC_TABLE;
        auto it = table.find({ 0, 0 });
        if (it == table.end()) {
            cerr << "Error: EOB code not found in Huffman table" << endl;
            return "";
        }
        return it->second;
    }
    else if (run == 15 && value == 0) { // ZRL
        const auto& table = isLuminance ? HA_Y_AC_TABLE : HA_C_AC_TABLE;
```

```
        auto it = table.find({ 15, 0 });
        if (it == table.end()) {
            cerr << "Error: ZRL code not found in Huffman table" << endl;
            return "";
        }
        return it->second;
    }

    // Определяем категорию значения
    int category = 0;
    if (value != 0) {
        category = static_cast<int>(log2(abs(value))) + 1;
    }

    // Получаем код Хаффмана для пары (run, category)
    const auto& table = isLuminance ? HA_Y_AC_TABLE : HA_C_AC_TABLE;
    auto it = table.find({ run, category });
    if (it == table.end()) {
        cerr << "Error: AC code for (run=" << run << ", size=" << category << ") not
found in Huffman table" << endl;
        return "";
    }

    // Добавляем биты значения VLI
    string code = it->second;
    if (category > 0) {
        int vli = value > 0 ? value : (value + (1 << category) - 1);
        code += bitset<32>(vli).to_string().substr(32 - category);
    }

    return code;
}

// Функция для кодирования блока
void encodeBlock(const vector<int>& block, BitStream* bitStream, bool
isLuminance) {
    // 1. Кодирование DC коэффициента
    int dcValue = block[0];
    int category = dcValue == 0 ? 0 : (int)log2(abs(dcValue)) + 1;

    const auto& dcTable = isLuminance ? HA_Y_DC_TABLE : HA_C_DC_TABLE;
    auto it = dcTable.find(category);
```

```
if (it != dcTable.end()) {
   writeBits(bitStream, it->second);
   if (category > 0) {
      int vli = dcValue > 0 ? dcValue : dcValue + (1 << category) - 1;
      for (int i = category - 1; i >= 0; i--) {
         writeBit(bitStream, (vli >> i) & 1);
      }
   }
}
else {
   cerr << "Error: DC category not found\n";
}

// 2. Кодирование AC коэффициентов
const auto& acTable = isLuminance ? HA_Y_AC_TABLE : HA_C_AC_TABLE;
int zeroRun = 0;

for (int i = 1; i < 64; i++) {
   if (block[i] == 0) {
      zeroRun++;
      if (zeroRun == 16) {
         // ZRL
         writeBits(bitStream, acTable.at({ 15, 0 }));
         zeroRun = 0;
      }
   }
   else {
      int value = block[i];
      int size = value == 0 ? 0 : (int)log2(abs(value)) + 1;
      writeBits(bitStream, acTable.at({ zeroRun, size }));

      if (size > 0) {
         int vli = value > 0 ? value : value + (1 << size) - 1;
         for (int j = size - 1; j >= 0; j--) {
            writeBit(bitStream, (vli >> j) & 1);
         }
      }
      zeroRun = 0;
   }
}

// EOB
```

```
    if (zeroRun > 0) {
        writeBits(bitStream, acTable.at({ 0, 0 }));
    }
}

// Функция для декодирования блока Хаффмана
vector<int> decodeHuffmanBlock(BitStreamReader* reader, bool isLuminance) {
    vector<int> block(64, 0);
    const auto& dcTable = isLuminance ? HA_Y_DC_TABLE : HA_C_DC_TABLE;
    const auto& acTable = isLuminance ? HA_Y_AC_TABLE : HA_C_AC_TABLE;

    // 1. Декодирование DC коэффициента
    string code;
    while (code.size() < 16) { // Максимальная длина кода DC
        code += readBit(reader) ? '1' : '0';
        auto it = find_if(dcTable.begin(), dcTable.end(),
            [&code](const auto& p) { return p.second == code; });
        if (it != dcTable.end()) {
            int category = it->first;
            int value = 0;
            if (category > 0) {
                for (int i = 0; i < category; i++) {
                    value = (value << 1) | (readBit(reader) ? 1 : 0);
                }
                if ((value & (1 << (category - 1))) == 0) {
                    value -= (1 << category) - 1;
                }
            }
            block[0] = value;
            break;
        }
    }

    // 2. Декодирование AC коэффициентов
    int index = 1;
    while (index < 64) {
        code.clear();
        bool found = false;

        // Читаем код, пока не найдем совпадение
        while (code.size() < 32 && !found) {
            code += readBit(reader) ? '1' : '0';
```

```
    // Проверяем специальные коды EOB и ZRL
    if (code == acTable.at({ 0, 0 })) { // EOB
        while (index < 64) block[index++] = 0;
        return block;
    }
    if (code == acTable.at({ 15, 0 })) { // ZRL
        for (int i = 0; i < 16 && index < 64; i++) {
            block[index++] = 0;
        }
        found = true;
        continue;
    }

    // Ищем обычные AC коды
    for (const auto& entry : acTable) {
        if (entry.second == code) {
            int run = entry.first.first;
            int size = entry.first.second;

            // Заполняем нулями согласно run
            for (int i = 0; i < run && index < 64; i++) {
                block[index++] = 0;
            }

            // Читаем значение
            if (size > 0 && index < 64) {
                int value = 0;
                for (int i = 0; i < size; i++) {
                    value = (value << 1) | (readBit(reader) ? 1 : 0);
                }
                if ((value & (1 << (size - 1))) == 0) {
                    value -= (1 << size) - 1;
                }
                block[index++] = value;
            }
            found = true;
            break;
        }
    }
}
```

```
            if (!found) {
                cerr << "Warning: AC code not found, filling with zeros\n";
                while (index < 64) block[index++] = 0;
                return block;
            }
        }
        return block;
    }

    // Функции для сохранения промежуточных изображений
    void saveChannelAsImage(const char* filename, const vector<vector<float>>&
    channel, bool scaleToRGB = false) {
        int height = channel.size();
        if (height == 0) return;
        int width = channel[0].size();

        vector<uint8_t> data(height * width * 3);

        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                int idx = (y * width + x) * 3;
                float val = channel[y][x];

                val = max(0.0f, min(255.0f, val));
                data[idx] = static_cast<uint8_t>(val);
                data[idx + 1] = static_cast<uint8_t>(val);
                data[idx + 2] = static_cast<uint8_t>(val);
            }
        }

        stbi_write_jpg(filename, width, height, 3, data.data(), 90);
    }

    void saveYCbCrImage(const char* filename, const
    vector<vector<vector<float>>>& ycbcr) {
        int height = ycbcr.size();
        if (height == 0) return;
        int width = ycbcr[0].size();

        vector<uint8_t> data(height * width * 3);

        for (int y = 0; y < height; y++) {
```

```
      for (int x = 0; x < width; x++) {
         int idx = (y * width + x) * 3;
         data[idx] = static_cast<uint8_t>(max(0.0f, min(255.0f, ycbcr[y][x][0])));    // 
Y
         data[idx + 1] = static_cast<uint8_t>(max(0.0f, min(255.0f, ycbcr[y][x][1])));
// Cb
         data[idx + 2] = static_cast<uint8_t>(max(0.0f, min(255.0f, ycbcr[y][x][2])));
// Cr
      }
   }

   stbi_write_jpg(filename, width, height, 3, data.data(), 90);
}

vector<vector<float>> inverseZigzag(const vector<int>& scanned);

// Визуализация блоков DCT
void visualizeDCTBlocks(const vector<vector<vector<int>>>& blocks, const 
string& filename) {
   // Проверка на пустые блоки
   if (blocks.empty() || blocks[0].empty()) return;

   // Создаем изображение нужного размера
   int blockRows = blocks.size();
   int blockCols = blocks[0].size();
   vector<vector<float>> image(blockRows * 8, vector<float>(blockCols * 8));

   // Заполняем изображение данными из блоков
   for (int by = 0; by < blockRows; by++) {
      for (int bx = 0; bx < blockCols; bx++) {
         // Проверка на валидность блока
         if (by >= blocks.size() || bx >= blocks[by].size()) continue;

         auto block = inverseZigzag(blocks[by][bx]);
         for (int y = 0; y < 8; y++) {
            for (int x = 0; x < 8; x++) {
               if (by * 8 + y < image.size() && bx * 8 + x < image[0].size()) {
                  image[by * 8 + y][bx * 8 + x] = block[y][x];
               }
            }
         }
      }
```

```
    }

    // Сохраняем изображение
    saveChannelAsImage(filename.c_str(), image);
}

// Конвертация RGB в YCbCr
void RGBtoYCbCr(vector<vector<vector<float>>>& image) {
    int height = image.size();
    if (height == 0) return;
    int width = image[0].size();

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            float R = image[y][x][0];
            float G = image[y][x][1];
            float B = image[y][x][2];

            float Y = 0.299f * R + 0.587f * G + 0.114f * B;
            float Cb = -0.1687f * R - 0.3313f * G + 0.5f * B + 128;
            float Cr = 0.5f * R - 0.4187f * G - 0.0813f * B + 128;

            image[y][x][0] = Y;
            image[y][x][1] = Cb;
            image[y][x][2] = Cr;
        }
    }

    // Сохраняем промежуточное изображение YCbCr
    saveYCbCrImage("1_ycbcr.jpg", image);

    // Сохраняем отдельные каналы
    vector<vector<float>> Y(height, vector<float>(width));
    vector<vector<float>> Cb(height, vector<float>(width));
    vector<vector<float>> Cr(height, vector<float>(width));

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            Y[y][x] = image[y][x][0];
            Cb[y][x] = image[y][x][1];
            Cr[y][x] = image[y][x][2];
        }
```

```
    }

    saveChannelAsImage("1_Y_channel.jpg", Y);
    saveChannelAsImage("1_Cb_channel.jpg", Cb);
    saveChannelAsImage("1_Cr_channel.jpg", Cr);
}

// Даунсэмплинг каналов Cb и Cr (2x2)
void downsample(vector<vector<float>>& channel, const string& channelName) {
    int height = channel.size();
    int width = height > 0 ? channel[0].size() : 0;
    int newHeight = (height + 1) / 2;
    int newWidth = (width + 1) / 2;

    vector<vector<float>> downsampled(newHeight, vector<float>(newWidth, 0));

    // Используем точное усреднение 4:2:0
    for (int y = 0; y < newHeight; y++) {
        for (int x = 0; x < newWidth; x++) {
            float sum = 0;
            int count = 0;

            for (int dy = 0; dy < 2; dy++) {
                for (int dx = 0; dx < 2; dx++) {
                    int ny = y * 2 + dy;
                    int nx = x * 2 + dx;
                    if (ny < height && nx < width) {
                        sum += channel[ny][nx];
                        count++;
                    }
                }
            }
            downsampled[y][x] = count > 0 ? sum / count : 0;
        }
    }
    channel = downsampled;

    // Сохраняем даунсэмплированный канал
    saveChannelAsImage(("2_" + channelName + "_downsampled.jpg").c_str(),
channel);
}
```

```
// Коэффициент для DCT
float alpha(int u) {
   return u == 0 ? sqrt(1.0 / 8.0) : sqrt(2.0 / 8.0);
}

// Функция для вычисления DCT блока 8x8
void DCT(vector<vector<float>>& block) {
   vector<vector<float>> temp(8, vector<float>(8, 0));
   vector<vector<float>> result(8, vector<float>(8, 0));

   // Первое применение для ряда
   for (int y = 0; y < 8; y++) {
      for (int u = 0; u < 8; u++) {
         float sum = 0.0f;
         for (int x = 0; x < 8; x++) {
            sum += block[y][x] * cos((2 * x + 1) * u * M_PI / 16.0);
         }
         temp[y][u] = alpha(u) * sum;
      }
   }

   // Повторное применение для столбца
   for (int u = 0; u < 8; u++) {
      for (int v = 0; v < 8; v++) {
         float sum = 0.0f;
         for (int y = 0; y < 8; y++) {
            sum += temp[y][u] * cos((2 * y + 1) * v * M_PI / 16.0);
         }
         result[v][u] = alpha(v) * sum;
      }
   }

   // Сохраняем результаты в блок
   block = result;
}

// Функция для вычисления обратного DCT блока 8x8
void IDCT(vector<vector<float>>& block) {
   vector<vector<float>> temp(8, vector<float>(8, 0));
   vector<vector<float>> result(8, vector<float>(8, 0));

   // Аналогично первое применение
```

```
    for (int y = 0; y < 8; y++) {
        for (int x = 0; x < 8; x++) {
            float sum = 0.0f;
            for (int u = 0; u < 8; u++) {
                sum += alpha(u) * block[y][u] * cos((2 * x + 1) * u * M_PI / 16.0);
            }
            temp[y][x] = sum;
        }
    }

    // Повторное применение
    for (int x = 0; x < 8; x++) {
        for (int y = 0; y < 8; y++) {
            float sum = 0.0f;
            for (int v = 0; v < 8; v++) {
                sum += alpha(v) * temp[v][x] * cos((2 * y + 1) * v * M_PI / 16.0);
            }
            result[y][x] = sum;
        }
    }

    // Сохраняем результат обратно в блок
    block = result;
}

// Тут квантование
void quantize(vector<vector<float>>& block, const int quantizationMatrix[8][8],
int quality) {
    float qualityFactor = quality <= 0 ? 0.01f :
        quality > 100 ? 8.0f :
        quality < 50 ? 50.0f / quality :
        2.0f - (2.0f * quality) / 100.0f;

    for (int y = 0; y < 8; y++) {
        for (int x = 0; x < 8; x++) {
            float q = quantizationMatrix[y][x] * qualityFactor;
            block[y][x] = round(block[y][x] / q);
        }
    }
}

// Тут деквантование
```

```
void dequantize(vector<vector<float>>& block, const int quantizationMatrix[8][8],
int quality) {
   float qualityFactor = quality <= 0 ? 0.01f :
      quality > 100 ? 8.0f :
      quality < 50 ? 50.0f / quality :
      2.0f - (2.0f * quality) / 100.0f;

   for (int y = 0; y < 8; y++) {
      for (int x = 0; x < 8; x++) {
         float q = quantizationMatrix[y][x] * qualityFactor;
         block[y][x] *= q;
      }
   }
}

// Функция для зигзаг-сканирования
vector<int> zigzagScan(const vector<vector<float>>& block) {
   vector<int> scanned(64);
   int index = 0;
   int row = 0, col = 0;
   bool goingUp = true;

   for (int i = 0; i < 64; i++) {
      scanned[index++] = static_cast<int>(block[row][col]);

      if (goingUp) {
         if (col == 7) { // Достигли правой границы
            row++;
            goingUp = false;
         }
         else if (row == 0) { // Достигли верхней границы
            col++;
            goingUp = false;
         }
         else { // Продолжаем движение вверх-вправо
            row--;
            col++;
         }
      }
      else { // Движение вниз-влево
         if (row == 7) { // Достигли нижней границы
            col++;
```

```
            goingUp = true;
        }
        else if (col == 0) { // Достигли левой границы
            row++;
            goingUp = true;
        }
        else { // Продолжаем движение вниз-влево
            row++;
            col--;
        }
    }
  }

  return scanned;
}

// Функция для обратного зигзаг-сканирования
vector<vector<float>> inverseZigzag(const vector<int>& scanned) {
  vector<vector<float>> block(8, vector<float>(8));
  int index = 0;
  int row = 0, col = 0;
  bool goingUp = true;

  for (int i = 0; i < 64; i++) {
    block[row][col] = static_cast<float>(scanned[index++]);

    if (goingUp) {
      if (col == 7) {
        row++;
        goingUp = false;
      }
      else if (row == 0) {
        col++;
        goingUp = false;
      }
      else {
        row--;
        col++;
      }
    }
    else {
      if (row == 7) {
```

```
            col++;
            goingUp = true;
        }
        else if (col == 0) {
            row++;
            goingUp = true;
        }
        else {
            row++;
            col--;
        }
    }
  }

  return block;
}

vector<vector<float>> reconstructChannel(const vector<vector<vector<int>>>&
blocks,
  int height, int width,
  const int quantMatrix[8][8],
  int quality, bool isLuminance) {
  vector<vector<float>> channel(height, vector<float>(width));

  for (size_t by = 0; by < blocks.size(); by++) {
    for (size_t bx = 0; bx < blocks[by].size(); bx++) {
      // Обратное зигзаг-сканирование
      auto block = inverseZigzag(blocks[by][bx]);

      // Обратное квантование
      dequantize(block, quantMatrix, quality);

      // Обратное DCT
      IDCT(block);

      // Запись в изображение
      for (int y = 0; y < 8; y++) {
        for (int x = 0; x < 8; x++) {
          int imgY = by * 8 + y;
          int imgX = bx * 8 + x;
          if (imgY < height && imgX < width) {
            channel[imgY][imgX] = block[y][x] + 128.0f;
```

```
            }
          }
        }
      }
    }
    return channel;
}

vector<vector<float>> upsample(const vector<vector<float>>& channel, int newHeight, int newWidth) {
    int oldHeight = channel.size();
    int oldWidth = oldHeight > 0 ? channel[0].size() : 0;
    vector<vector<float>> upsampled(newHeight, vector<float>(newWidth));

    for (int y = 0; y < newHeight; y++) {
        for (int x = 0; x < newWidth; x++) {
            float fy = (y * (oldHeight - 1)) / (float)(newHeight - 1);
            float fx = (x * (oldWidth - 1)) / (float)(newWidth - 1);

            int y1 = floor(fy), x1 = floor(fx);
            int y2 = min(y1 + 1, oldHeight - 1);
            int x2 = min(x1 + 1, oldWidth - 1);

            float a = fy - y1, b = fx - x1;
            upsampled[y][x] =
                (1 - a) * (1 - b) * channel[y1][x1] +
                (1 - a) * b * channel[y1][x2] +
                a * (1 - b) * channel[y2][x1] +
                a * b * channel[y2][x2];
        }
    }
    return upsampled;
}

// Функция для загрузки изображения
vector<vector<vector<float>>> loadImage(const char* filename, int& width, int& height) {
    int channels;
    unsigned char* data = stbi_load(filename, &width, &height, &channels, 3);
    if (!data) {
        cerr << "Failed to load image: " << filename << endl;
        exit(1);
```

```
    }

    vector<vector<vector<float>>> image(height, vector<vector<float>>(width, vector<float>(3)));

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int idx = (y * width + x) * 3;
            image[y][x][0] = data[idx];
            image[y][x][1] = data[idx + 1];
            image[y][x][2] = data[idx + 2];
        }
    }

    stbi_image_free(data);
    return image;
}

// Функция для сохранения изображения
void saveImage(const char* filename, const vector<vector<vector<float>>>& image) {
    int height = image.size();
    if (height == 0) return;
    int width = image[0].size();

    vector<uint8_t> data(height * width * 3);

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int idx = (y * width + x) * 3;
            data[idx] = static_cast<uint8_t>(max(0.0f, min(255.0f, image[y][x][0])));
            data[idx + 1] = static_cast<uint8_t>(max(0.0f, min(255.0f, image[y][x][1])));
            data[idx + 2] = static_cast<uint8_t>(max(0.0f, min(255.0f, image[y][x][2])));
        }
    }

    stbi_write_jpg(filename, width, height, 3, data.data(), 90);
}

// Функция для конвертации YCbCr в RGB
void YCbCrtoRGB(vector<vector<vector<float>>>& image) {
    int height = image.size();
```

```
    if (height == 0) return;
    int width = image[0].size();

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            float Y = image[y][x][0];
            float Cb = image[y][x][1] - 128;
            float Cr = image[y][x][2] - 128;

            float R = Y + 1.402f * Cr;
            float G = Y - 0.344136f * Cb - 0.714136f * Cr;
            float B = Y + 1.772f * Cb;

            image[y][x][0] = max(0.0f, min(255.0f, R));
            image[y][x][1] = max(0.0f, min(255.0f, G));
            image[y][x][2] = max(0.0f, min(255.0f, B));
        }
    }
}

// Основная функция сжатия с сохранением промежуточных результатов. Добавляю везде цифры, чтобы
// было проще ориентироватьс в коде...
vector<uint8_t> compressImage(const vector<vector<vector<float>>>& image, int quality, BitStream* bitStream) {
    // 1. Разделение на каналы
    int height = image.size();
    int width = image[0].size();

    vector<vector<float>> Y(height, vector<float>(width));
    vector<vector<float>> Cb(height, vector<float>(width));
    vector<vector<float>> Cr(height, vector<float>(width));

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            Y[y][x] = image[y][x][0];
            Cb[y][x] = image[y][x][1];
            Cr[y][x] = image[y][x][2];
        }
    }

    // 2. Даунсэмплинг Cb и Cr
```

```
    downsample(Cb, "Cb");
    downsample(Cr, "Cr");

    cout << "[Info] Downsampled" << endl;

    // 3. Разбиение на блоки 8x8 и DCT
    int blockRowsY = (height + 7) / 8;
    int blockColsY = (width + 7) / 8;
    int blockRowsC = (Cb.size() + 7) / 8;
    int blockColsC = (Cb[0].size() + 7) / 8;

    vector<vector<vector<int>>> blocksY(blockRowsY,
vector<vector<int>>(blockColsY, vector<int>(64)));
    vector<vector<vector<int>>> blocksCb(blockRowsC,
vector<vector<int>>(blockColsC, vector<int>(64)));
    vector<vector<vector<int>>> blocksCr(blockRowsC,
vector<vector<int>>(blockColsC, vector<int>(64)));

    // Обработка Y канала
    vector<int> dcValuesY;
    for (int by = 0; by < blockRowsY; by++) {
        for (int bx = 0; bx < blockColsY; bx++) {
            vector<vector<float>> block(8, vector<float>(8));

            // Заполнение блока
            for (int y = 0; y < 8; y++) {
                for (int x = 0; x < 8; x++) {
                    int imgY = by * 8 + y;
                    int imgX = bx * 8 + x;

                    if (imgY < height && imgX < width) {
                        block[y][x] = Y[imgY][imgX] - 128; // Сдвиг уровня
                    }
                    else {
                        block[y][x] = 0; // Заполнение нулями
                    }
                }
            }

            // DCT и квантование
            DCT(block);
            quantize(block, QY, quality);
```

```
            // Зигзаг-сканирование
            vector<int> scanned = zigzagScan(block);
            for (int i = 0; i < 64; i++) {
                blocksY[by][bx][i] = scanned[i];
            }

            // Сохраняем DC коэффициент
            dcValuesY.push_back(scanned[0]);
        }
    }

    cout << "[Info] Y channel processing" << endl;

    // Обработка Cb канала
    vector<int> dcValuesCb;
    for (int by = 0; by < blockRowsC; by++) {
        for (int bx = 0; bx < blockColsC; bx++) {
            vector<vector<float>> block(8, vector<float>(8));

            // Заполнение блока
            for (int y = 0; y < 8; y++) {
                for (int x = 0; x < 8; x++) {
                    int imgY = by * 8 + y;
                    int imgX = bx * 8 + x;

                    if (imgY < Cb.size() && imgX < Cb[0].size()) {
                        block[y][x] = Cb[imgY][imgX] - 128; // Сдвиг уровня
                    }
                    else {
                        block[y][x] = 0; // Заполнение нулями
                    }
                }
            }

            // DCT и квантование
            DCT(block);
            quantize(block, QC, quality);

            // Зигзаг-сканирование
            vector<int> scanned = zigzagScan(block);
            for (int i = 0; i < 64; i++) {
```

```
            blocksCb[by][bx][i] = scanned[i];
        }

        // Сохраняем DC коэффициент
        dcValuesCb.push_back(scanned[0]);
    }
}

cout << "[Info] Cb channel processing" << endl;

// Обработка Cr канала
vector<int> dcValuesCr;
for (int by = 0; by < blockRowsC; by++) {
    for (int bx = 0; bx < blockColsC; bx++) {
        vector<vector<float>> block(8, vector<float>(8));

        // Заполнение блока
        for (int y = 0; y < 8; y++) {
            for (int x = 0; x < 8; x++) {
                int imgY = by * 8 + y;
                int imgX = bx * 8 + x;

                if (imgY < Cr.size() && imgX < Cr[0].size()) {
                    block[y][x] = Cr[imgY][imgX] - 128; // Сдвиг уровня
                }
                else {
                    block[y][x] = 0; // Заполнение нулями
                }
            }
        }

        // DCT и квантование
        DCT(block);
        quantize(block, QC, quality);

        // Зигзаг-сканирование
        vector<int> scanned = zigzagScan(block);
        for (int i = 0; i < 64; i++) {
            blocksCr[by][bx][i] = scanned[i];
        }

        // Сохраняем DC коэффициент
```

```
            dcValuesCr.push_back(scanned[0]);
        }
    }

    cout << "[Info] Cr channel processing" << endl;

    // Визуализация блоков после DCT и квантования
    visualizeDCTBlocks(blocksY, "3_Y_DCT_quantized.jpg");
    visualizeDCTBlocks(blocksCb, "3_Cb_DCT_quantized.jpg");
    visualizeDCTBlocks(blocksCr, "3_Cr_DCT_quantized.jpg");

    // 4. Разностное кодирование DC коэффициентов
    vector<int> diffDCY = encodeDCDifferences(dcValuesY);
    vector<int> diffDCCb = encodeDCDifferences(dcValuesCb);
    vector<int> diffDCCr = encodeDCDifferences(dcValuesCr);

    cout << "[Info] DC differential coded" << endl;

    // 5. Кодирование Хаффмана и создание битового потока
    // Записываем метаданные
    writeByte(bitStream, width >> 8);
    writeByte(bitStream, width & 0xFF);
    writeByte(bitStream, height >> 8);
    writeByte(bitStream, height & 0xFF);
    writeByte(bitStream, quality);

    // Записываем DC коэффициенты
    for (int diff : diffDCY) {
        writeByte(bitStream, (diff >> 8) & 0xFF);
        writeByte(bitStream, diff & 0xFF);
    }
    for (int diff : diffDCCb) {
        writeByte(bitStream, (diff >> 8) & 0xFF);
        writeByte(bitStream, diff & 0xFF);
    }
    for (int diff : diffDCCr) {
        writeByte(bitStream, (diff >> 8) & 0xFF);
        writeByte(bitStream, diff & 0xFF);
    }

    // Кодируем блоки Y
    for (const auto& blockRow : blocksY) {
```

```
    for (const auto& block : blockRow) {
      encodeBlock(block, bitStream, true); // true for luminance
    }
  }

  // Кодируем блоки Cb
  for (const auto& blockRow : blocksCb) {
    for (const auto& block : blockRow) {
      encodeBlock(block, bitStream, false); // false for chrominance
    }
  }

  // Кодируем блоки Cr
  for (const auto& blockRow : blocksCr) {
    for (const auto& block : blockRow) {
      encodeBlock(block, bitStream, false); // false for chrominance
    }
  }

  cout << "[Info] Compress completed" << endl;

  return getData(bitStream);
}

// Основная функция декомпрессии с сохранением промежуточных результатов
vector<vector<vector<float>>> decompressImage(const vector<uint8_t>& compressedData) {
  BitStreamReader reader = createBitStreamReader(compressedData);

  // Читаем метаданные
  int width = (readByte(&reader) << 8) | readByte(&reader);
  int height = (readByte(&reader) << 8) | readByte(&reader);
  int quality = readByte(&reader);
  cout << "Info [dec]: Metadata read" << endl;

  // Определяем количество блоков
  int blockRowsY = (height + 7) / 8;
  int blockColsY = (width + 7) / 8;
  int blockRowsC = ((height / 2) + 7) / 8;
  int blockColsC = ((width / 2) + 7) / 8;
```

```
    // Чтение DC коэффициентов
    vector<int> diffDCY, diffDCCb, diffDCCr;

    for (int i = 0; i < blockRowsY * blockColsY; i++) {
        int16_t diff1 = (readByte(&reader) << 8) | readByte(&reader);
        diffDCY.push_back(static_cast<int>(diff1));
    }
    for (int i = 0; i < blockRowsC * blockColsC; i++) {
        int16_t diff2 = (readByte(&reader) << 8) | readByte(&reader);
        diffDCCb.push_back(static_cast<int>(diff2));
    }
    for (int i = 0; i < blockRowsC * blockColsC; i++) {
        int16_t diff3 = (readByte(&reader) << 8) | readByte(&reader);
        diffDCCr.push_back(static_cast<int>(diff3));
    }
    cout << "Info [dec]: DC read" << endl;

    // Декодирование DC коэффициентов
    vector<int> dcValuesY = decodeDCDifferences(diffDCY);
    vector<int> dcValuesCb = decodeDCDifferences(diffDCCb);
    vector<int> dcValuesCr = decodeDCDifferences(diffDCCr);
    cout << "Info [dec]: DC decoded" << endl;

    // Декодирование блоков Y
    vector<vector<vector<int>>>                                    blocksY(blockRowsY,
vector<vector<int>>(blockColsY, vector<int>(64)));
    int dcIndex = 0;
    for (int by = 0; by < blockRowsY; by++) {
        for (int bx = 0; bx < blockColsY; bx++) {
            vector<int> block = decodeHuffmanBlock(&reader, true);
            block[0] = dcValuesY[dcIndex++];
            blocksY[by][bx] = block;
        }
    }
    cout << "Info [dec]: Y blocks decoded" << endl;

    // Декодирование блоков Cb
    vector<vector<vector<int>>>                                  blocksCb(blockRowsC,
vector<vector<int>>(blockColsC, vector<int>(64)));
    dcIndex = 0;
    for (int by = 0; by < blockRowsC; by++) {
        for (int bx = 0; bx < blockColsC; bx++) {
```

```
            vector<int> block = decodeHuffmanBlock(&reader, false);
            block[0] = dcValuesCb[dcIndex++];
            blocksCb[by][bx] = block;
        }
    }
    cout << "Info [dec]: Cb block decoded" << endl;

    // Декодирование блоков Cr
    vector<vector<vector<int>>>                                    blocksCr(blockRowsC,
vector<vector<int>>(blockColsC, vector<int>(64)));
    dcIndex = 0;
    for (int by = 0; by < blockRowsC; by++) {
        for (int bx = 0; bx < blockColsC; bx++) {
            vector<int> block = decodeHuffmanBlock(&reader, false);
            block[0] = dcValuesCr[dcIndex++];
            blocksCr[by][bx] = block;
        }
    }
    cout << "Info [dec]: Cr blocks decoded" << endl;

    // Восстановление изображения Y
    vector<vector<float>> Y(height, vector<float>(width));
    for (int by = 0; by < blockRowsY; by++) {
        for (int bx = 0; bx < blockColsY; bx++) {
            auto block = inverseZigzag(blocksY[by][bx]);
            dequantize(block, QY, quality);
            IDCT(block);

            for (int y = 0; y < 8; y++) {
                for (int x = 0; x < 8; x++) {
                    int imgY = by * 8 + y;
                    int imgX = bx * 8 + x;
                    if (imgY < height && imgX < width) {
                        float val = block[y][x] + 128.0f;
                        Y[imgY][imgX] = max(0.0f, min(255.0f, val));
                    }
                }
            }
        }
    }
    cout << "Info [dec]: Y image recovery" << endl;
    saveChannelAsImage("4_Y_adter_IDCT.jpg", Y);
```

```
// Восстановление изображения Cb
vector<vector<float>> Cb((height + 1) / 2, vector<float>((width + 1) / 2));
for (int by = 0; by < blockRowsC; by++) {
   for (int bx = 0; bx < blockColsC; bx++) {
      auto block = inverseZigzag(blocksCb[by][bx]);
      dequantize(block, QC, quality);
      IDCT(block);

      for (int y = 0; y < 8; y++) {
         for (int x = 0; x < 8; x++) {
            int imgY = by * 8 + y;
            int imgX = bx * 8 + x;
            if (imgY < Cb.size() && imgX < Cb[0].size()) {
               float val = block[y][x] + 128.0f;
               Cb[imgY][imgX] = max(0.0f, min(255.0f, val));
            }
         }
      }
   }
}
cout << "Info [dec]: Cb image recovery" << endl;
saveChannelAsImage("4_Cb_adter_IDCT.jpg", Cb);

// Восстановление изображения Cr
vector<vector<float>> Cr((height + 1) / 2, vector<float>((width + 1) / 2));
for (int by = 0; by < blockRowsC; by++) {
   for (int bx = 0; bx < blockColsC; bx++) {
      auto block = inverseZigzag(blocksCr[by][bx]);
      dequantize(block, QC, quality);
      IDCT(block);

      for (int y = 0; y < 8; y++) {
         for (int x = 0; x < 8; x++) {
            int imgY = by * 8 + y;
            int imgX = bx * 8 + x;
            if (imgY < Cr.size() && imgX < Cr[0].size()) {
               float val = block[y][x] + 128.0f;
               Cr[imgY][imgX] = max(0.0f, min(255.0f, val));
            }
         }
      }
```

```
        }
    }
    cout << "Info [dec]: Cr image recovery" << endl;
    saveChannelAsImage("4_Cr_adter_IDCT.jpg", Cr);

    // Апсемплинг Cb и Cr
    vector<vector<float>> CbUpsampled(height, vector<float>(width));
    vector<vector<float>> CrUpsampled(height, vector<float>(width));
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            CbUpsampled[y][x] = Cb[y / 2][x / 2];
            CrUpsampled[y][x] = Cr[y / 2][x / 2];
        }
    }
    cout << "Info [dec]: Cb and Cr upsampled" << endl;

    // Создание YCbCr изображения
    vector<vector<vector<float>>> ycbcr(height, vector<vector<float>>(width,
vector<float>(3)));
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            ycbcr[y][x][0] = Y[y][x];
            ycbcr[y][x][1] = CbUpsampled[y][x];
            ycbcr[y][x][2] = CrUpsampled[y][x];
        }
    }
    cout << "Info [dec]: YCbCr recreated" << endl;

    // Сохраняем промежуточное изображение YCbCr
    saveYCbCrImage("5_decompressed_ycbcr.jpg", ycbcr);
    cout << "Info [dec]: The image is saved" << endl;

    // Конвертация обратно в RGB
    YCbCrtoRGB(ycbcr);
    cout << "Info [dec]: Image was converted back to RGB" << endl;

    return ycbcr;
}

int main() {
    // 1. Загрузка RAW изображения
    int width = 512, height = 512;
```

```
    auto image = loadRawImage("lenna_bw_nodither.raw", width, height);

    // 2. Конвертация в YCbCr и сжатие
    RGBtoYCbCr(image);
    int quality = 40;
    BitStream bitStream;
    auto compressedData = compressImage(image, quality, &bitStream);

    // 3. Сохраняем сжатые данные + метаданные в compressed.bin
    writeCompressedToFile("6_compressed.bin", compressedData, width, height,
quality);

    // 4. Читаем compressed.bin и разжимаем
    int loadedWidth, loadedHeight, loadedQuality;
    auto loadedData = readCompressedFromFile("6_compressed.bin",
loadedWidth, loadedHeight, loadedQuality);
    auto decompressedImage = decompressImage(loadedData);

    // 5. Сохраняем разжатое изображение
    saveRawImage("6_decompressed.raw", decompressedImage);
    saveImage("6_decompressed.jpg", decompressedImage);

    // 6. Сравниваем lenna_rgb.raw и decompressed.raw (например, через diff)
    cout << "Compress completed. Compare here:\n";
    cout << "- Original: lenna_rgb.raw (" << (image.size() * 512 * 3) << " bytes)\n";
    cout << "- Compressed: compressed.bin (" << compressedData.size() << "
bytes)\n"; // 133 = 5 + 128 байт заголовка
    cout << "- Decompressed: decompressed.raw (" << (decompressedImage.size()
* 512 * 3) << " bytes)\n";
    cout << "- JPG for check: decompressed.jpg\n";

    return 0;
}
```

Ссылка на GIT:
https://github.com/millkun/AiSD2.git