**Petropolitan Science (Re) Search**

Вычислительные технологии (фундаментальная информатика и информационные технологии) (бакалавриат) 2025

**Задача 1.**

<https://github.com/Denigmma/olimp_PSRS_task1_2025>

Данная программа предназначена для игры в классические русские шашки против пользователя. При запуске, пользователь выбирает цвет: если выбраны «белые» — игрок ходит первым, если «чёрные» — компьютер сразу делает стартовый ход за белых и выводит игровое поле. Так же учитывается расположение доски в зависимости от выбора цвета шашек игрока (как прописано в условии). На экран выводится доска 10х10 (с координатами A–H и 1–8), а пользователь вводит ход в формате (A3 B4). Если ход неверен, программа просит повторить ввод. Так же, рассмотрена логика обязательных рубок и их приоритетность, если у игрока есть принудительная рубка, то программа ожидает только этого обязательного хода. После валидного хода компьютер рассчитывает свой ответ параллельным алгоритмом Minimax и ходит в ответ, чередуя таким образом ходы, пока одна из сторон не лишится возможности двигаться. Поддерживаются все классические правила русских шашек (включая дамки, движение по диагонали, принудительную рубку и т.д.). Так же учтены приоритетные рубки (так как в правилах русских шашек говорится, что в случае нескольких вариантов рубок, игрок обязан выбирать наибольшую по количеству срубленных шашек). Программа стабильно работает и корректно обрабатывает все виды ходов. Исходный код не требует сторонних библиотек и может быть собран с помощью Cmake.

В основе программы лежит класс CheckersBoard. Он хранит текущее состояние игрового поля в виде двумерного массива (вектора векторов) размером 8×8, а каждый элемент этого массива представляет собой перечислимый тип Piece. В этом перечислении определены:

1. W (простая белая шашка)
2. B (простая чёрная шашка)
3. DW (белая дамка)
4. DB (чёрная дамка)
5. EMPTY для пустых клеток.

Поле whiteToMove (логического типа) показывает, чья очередь хода: true означает, что ходят белые, false — что ходят чёрные. В классе предусмотрен метод initBoard для начальной расстановки фигур (белые занимают верхнюю часть доски на тёмных клетках, чёрные — нижнюю), а также функция printBoard для вывода доски и координат строк (1–8) и столбцов (A–H).

Самая сложная часть программы — это логика, позволяющая корректно обрабатывать ходы и параллельно рассчитывать наилучший компьютерный ход. Метод makeMove принимает объект Move (внутри него хранится цепочка координат, которая описывает потенциально многошаговый ход, особенно при рубке), проверяет, корректен ли ход (наискось, не заняты ли клетки, есть ли там шашка соперника для рубки) и применяет его к доске. Если простая шашка достигает противоположной стороны, она становится дамкой (DW или DB). Программа также учитывает принудительную рубку: сначала ищет все «рубящие» ходы, и только если их нет, допускается обычное перемещение. Для этого служит метод getAllPossibleMoves, который перебирает каждую шашку текущего игрока и собирает все возможные ходы. Если хотя бы один ход с рубкой нашёлся, метод возвращает только такие рубящие ходы. После хода makeMove проверяет, может ли соперник сделать ход. Если нет, партия завершается, и текущий ходивший выигрывает.

Чтобы компьютер выбирал свой ход оптимально, в программе используется алгоритм Minimax с альфа-бета-отсечением на заданную глубину поиска, обычно равную пяти. Это означает, что мы строим дерево ходов на 5 слоёв вперёд. Каждый уровень соответствует одному полуходу — то есть ход либо белых, либо чёрных. Пусть, компьютер играет за чёрных. Когда мы вызываем функцию getBestMove(5), она составляет список всех доступных ходов для текущей стороны (в данном случае для чёрных), на которой сейчас «очередь хода». Для каждого такого хода виртуально (на копии доски) делает этот ход. Затем рекурсивно вызывает алгоритм Minimax на глубину 4, переключив сторону на ход белых и так же перебираем ходы. Так продолжается, пока не погружаемся на глубину 5 уровней. Когда глубина пройдена - программа вызывает функцию evaluateBoard(), которая присваивает числовую оценку позиции. Положительное число говорит о преимуществе белых, а отрицательное — о преимуществе чёрных. Далее Minimax идёт наверх по рекурсии: Если на данном уровне хода максимизирующий игрок (например, белые) — мы выбираем ход, дающий наибольшую оценку.Если же «минимизирующий» (чёрные), — мы выбираем наименьшую оценку (потому что чёрным выгодно делать позицию хуже для белых и лучше для себя).При этом в коде есть параметр maximizingPlayer, который в зависимости от того, кто сейчас делает ход (белые или чёрные), решает, ищем ли мы максимум или минимум. Когда компьютер играет за чёрных, мы фактически считаем: чёрные — минимизирующий игрок, белые — максимизирующий.

Критерий, по которому выбирается лучший ход, — это итоговая оценка - score, которую возвращает Minimax с глубины перебора. Чем лучше (выше или ниже в зависимости от цвета) оценка, тем выше приоритет у этого хода. Оценка позиции реализована очень просто и опирается лишь на количественное соотношение фигур — простых и дамок:

1. Белая простая шашка даёт +1 к счёту.
2. Белая дамка даёт +3 к счёту (считается более ценной).
3. Чёрная простая шашка даёт –1 (уменьшает итоговый счёт).
4. Чёрная дамка даёт –3.

Чтобы не тратить слишком много времени на заведомо невыгодные ветки, используется альфа-бета-отсечение: как только становится понятно, что оценка одной ветви не улучшит итоговый результат (или наоборот уже слишком хороша для соперника), дальнейший перебор веток отсекается.Для ускорения перебора используется многопоточность через std::thread. На верхнем уровне (или нескольких верхних уровнях) алгоритм Minimax создаёт для каждого доступного хода отдельный поток. В каждом потоке в лямбда-выражении берётся копия текущей доски, выполняется метод makeMove, затем рекурсивно вызывается Minimax, и результат сохраняется в общий массив. После завершения всех потоков с использованием join результаты объединяются, и программа выбирает лучший или худший ход в зависимости от фазы (максимизирующий или минимизирующий уровень).Ограничение создания потоков верхними уровнями позволяет избежать экспоненциального роста их количества, что могло бы замедлить выполнение. Это также сохраняет эффективность альфа-бета-отсечения, так как потоки на более низких уровнях не создаются, пока один из потоков не обновит значения альфа или бета, что предотвращает преждевременное отсечение. Логика многопоточности: для каждого хода создаётся поток, результаты сохраняются в общий массив, а затем объединяются для выбора оптимального хода.  
  
**Задача 2.**

<https://github.com/Denigmma/olimp_PSRS_task2_2025>

Для выполнения задачи выравнивания изображения документа с текстом и изображением я разработал алгоритм, основанный на использовании библиотеки OpenCV, которая предоставляет эффективные инструменты для обработки изображений. OpenCV позволяет быстро обрабатывать изображения и проводить требуемые преобразования, такие как преобразования в серые тона, выделение контуров изображений и т.д.

Для начала, мы читаем исходное изображение и преобразовываем его в градации серого. Это необходимо для того, чтобы упростить последующую обработку. Преобразование в серый цвет позволяет нам фокусироваться на яркости пикселей, а не на их цвете, что особенно важно при анализе текстов, где цвет часто не играет роли. Этот подход позволяет уменьшить объем данных и упростить последующую обработку. Он существенно снижает требования к памяти, обработка изображений в оттенках серого требует меньше вычислительных ресурсов, чем работа с полноцветными изображениями.

Чтобы уменьшить влияние шума и устранить мелкие артефакты, которые могут повлиять на точность нахождения угла наклона, применяется Гауссово размытие. Этот фильтр помогает удалить высокочастотные компоненты (шумы), сглаживая резкие изменения яркости. Гауссово размытие работает путём усреднения яркости пикселей в окрестности каждого пикселя, что делает переход между различными участками изображения более плавным и улучшает качество дальнейшего выделения контуров.

Для того чтобы выделить границы объектов, таких как текст и графику, используется алгоритм Кэнни. Он работает следующим образом:

1. Удаление шума — на первом шаге применяется фильтрация изображения с использованием оператора Гаусса, чтобы убрать мелкие шумы и сгладить изображение.
2. Выделение границ — алгоритм затем находит границы, анализируя изменения яркости между соседними пикселями. Он вычисляет градиенты изображения и ищет максимальные изменения этих градиентов, которые указывают на края объектов.
3. Двойной порог — после нахождения градиентов алгоритм применяет двойной порог для отделения сильных и слабых границ. Это позволяет избавиться от слабых и ненадёжных контуров, оставив только те, которые с высокой вероятностью являются истинными границами объектов.

Результатом работы алгоритма является карта контуров, где чётко видны линии текста, границы графиков, диаграмм и других объектов. Эти контуры далее будут использоваться для определения угла наклона документа.

Для нахождения угла наклона документа используется преобразование Хафа. Этот метод позволяет находить прямые линии на изображении, представив их в полярной системе координат, где каждая линия описывается двумя параметрами: расстоянием от центра изображения и углом наклона. На основе карты контуров строятся линии, каждая из которых определяется набором точек, представляющих краевые участки изображения. Преобразование Хафа преобразует эти линии в полярные координаты, и для каждой прямой вычисляется её угол наклона. Все найденные углы собираются в список, после чего для всех углов рассчитывается медианный угол. Медианный угол выбирается, так как он является более устойчивым к выбросам и случайным отклонениям, например, из-за диаграмм или других графических элементов, не относящихся к основному тексту. Медианный угол наклона документных объектов (текста и графики) считается углом, на который необходимо повернуть изображение, чтобы оно стало горизонтальным.

После нахождения угла наклона и его коррекции, изображение вращается на противоположный угол. Для этого применяется аффинное преобразование. В процессе поворота сохраняется структура изображения, а пустые участки, образующиеся после поворота, заполняются белым цветом.

**Обоснование решения с точки зрения работы алгоритмов, памяти и ресурсов:**

Выбор описанных алгоритмов обусловлен их высокой эффективностью и оптимизацией в библиотеке OpenCV. Преобразование изображения в градации серого не только снижает объем вычисляемых данных (сокращая количество каналов с трёх до одного), но и позволяет сосредоточиться исключительно на яркостных характеристиках, что критически для выделения текстовых линий. Гауссово размытие, используемое для устранения шума, помогает обеспечить более стабильное выделение контуров, что важно при работе с неидеальными изображениями.

Алгоритм Кэнни, благодаря этапам фильтрации, вычисления градиентов и применению двойного порога, позволяет точно выделить значимые контуры, минимизируя влияние мелких артефактов. Преобразование Хафа эффективно обрабатывает обнаруженные контуры, переводя их в полярное представление, что даёт возможность надёжно оценить общий угол наклона документа даже при наличии выбросов. Выбор медианного угла как итогового результата обеспечивает устойчивость алгоритма к резким изменениям.

С точки зрения памяти и вычислительных ресурсов, все этапы обработки оптимизированы: преобразования выполняются над матрицами NumPy, что позволяет экономно расходовать оперативную память.

**Оценка сложности:**

* Преобразование изображения в градации серого выполняется за O(n), где n – общее число пикселей, поскольку операция применяется к каждому пикселю отдельно.
* Гауссово размытие реализовано через свёртку с фильтром фиксированного размера (5×5), что приводит к сложности O(n \* k), где k – размер ядра (константа). Таким образом, сложность сводится к O(n) с постоянным множителем.
* Алгоритм выделения границ Кэнни включает несколько этапов (вычисление градиентов, подавление немаксимумов, применение двойного порога). Каждый из этих этапов требует прохода по всему изображению, что даёт общую сложность O(n) с небольшими константными факторами.
* Преобразование Хафа является наиболее ресурсоёмкой частью алгоритма. Его сложность зависит от количества найденных краевых пикселей (n\_edge) и количества дискретных значений углов (A), что в худшем случае даёт O(n\_edge \* A).
* Аффинное преобразование для поворота изображения также имеет линейную сложность O(n), так как каждая точка изображения обрабатывается один раз.

Все операции реализованы с использованием оптимизированных функций OpenCV и обработки массивов через NumPy, что обеспечивает эффективное использование памяти и ускорение вычислений за счёт векторизации и низкоуровневых оптимизаций.

**Экспериментальная проверка:**

Для проверки работы алгоритма были использованы 6 различных изображений-документов, содержащих как текст, так и графические элементы (диаграммы, графики и изображения).

При экспериментальной проверке использовались следующие методики:

* Визуальный анализ: Каждый результат поворота сравнивался с исходным изображением, и оценивалась корректность выравнивания текста, а также сохранение структуры графических элементов. Выявлено что текст на изображениях стал горизонтальным, а графика не подверглась искажению.
* Количественная оценка: На изображениях проводилось измерение исходного угла наклона и сравнение его с результатом после коррекции. Полученные данные показали, что погрешность коррекции не превышает 1-2 градуса, что является приемлемым.

Результаты экспериментальной проверки показали, что алгоритм успешно корректирует угол наклона документа: текст на изображениях стал горизонтальным, а остальные элементы (графика, диаграммы) сохранили свою структуру. Это подтверждает, что предложенный подход полностью соответствует требованиям задачи и демонстрирует высокую устойчивость к различным типам входных данных.

**Задача 3.**

<https://github.com/Denigmma/olimp_PSRS_task3_2025>

В данном задании данные зашифрованы с помощью шифра Цезаря с разными сдвигами для каждого адреса. Принцип шифра Цезаря заключается в подстановке, в которой каждый символ в слове (тексте) заменяется символом, находящимся на некотором постоянном числе позиций левее или правее него в алфавите. Таким образом, чтобы расшифровать данные таблицы адресов я написал программу на Python, которая осуществляет перебор, который заключается в подстановки вместо каждой буквы – другой буквы, которая находится на определенном расстоянии (сдвиге – “key”) в алфавите. Эти сдвиги (“key”) – перебираются вдоль всего алфавита. Так как обычные адреса и email написаны на разных языках, то используется русский и английский алфавиты соответственно.   
Давайте рассмотрим конкретный случай:  
Осуществим такой перебор для первого адреса из таблицы:  
 **“ Ыу. Лпмшорхщтцкцл.6 Тй.466”**и получим такие результаты:  
0 ЫУ. ЛПМШОРХЩТЦКЦЛ.6 ТЙ.466

1 ЬФ. МРНЩПСЦЪУЧЛЧМ.6 УК.466

2 ЭХ. НСОЪРТЧЫФШМШН.6 ФЛ.466

3 ЮЦ. ОТПЫСУШЬХЩНЩО.6 ХМ.466

4 ЯЧ. ПУРЬТФЩЭЦЪОЪП.6 ЦН.466

5 АШ. РФСЭУХЪЮЧЫПЫР.6 ЧО.466

6 БЩ. СХТЮФЦЫЯШЬРЬС.6 ШП.466

7 ВЪ. ТЦУЯХЧЬАЩЭСЭТ.6 ЩР.466

8 ГЫ. УЧФАЦШЭБЪЮТЮУ.6 ЪС.466

9 ДЬ. ФШХБЧЩЮВЫЯУЯФ.6 ЫТ.466

10 ЕЭ. ХЩЦВШЪЯГЬАФАХ.6 ЬУ.466

11 ЁЮ. ЦЪЧГЩЫАДЭБХБЦ.6 ЭФ.466

12 ЖЯ. ЧЫШДЪЬБЕЮВЦВЧ.6 ЮХ.466

13 ЗА. ШЬЩЕЫЭВЁЯГЧГШ.6 ЯЦ.466

14 ИБ. ЩЭЪЁЬЮГЖАДШДЩ.6 АЧ.466

15 ЙВ. ЪЮЫЖЭЯДЗБЕЩЕЪ.6 БШ.466

16 КГ. ЫЯЬЗЮАЕИВЁЪЁЫ.6 ВЩ.466

17 ЛД. ЬАЭИЯБЁЙГЖЫЖЬ.6 ГЪ.466

18 МЕ. ЭБЮЙАВЖКДЗЬЗЭ.6 ДЫ.466

19 НЁ. ЮВЯКБГЗЛЕИЭИЮ.6 ЕЬ.466

20 ОЖ. ЯГАЛВДИМЁЙЮЙЯ.6 ЁЭ.466

21 ПЗ. АДБМГЕЙНЖКЯКА.6 ЖЮ.466

22 РИ. БЕВНДЁКОЗЛАЛБ.6 ЗЯ.466

23 СЙ. ВЁГОЕЖЛПИМБМВ.6 ИА.466

24 ТК. ГЖДПЁЗМРЙНВНГ.6 ЙБ.466

25 УЛ. ДЗЕРЖИНСКОГОД.6 КВ.466

26 ФМ. ЕИЁСЗЙОТЛПДПЕ.6 ЛГ.466

27 ХН. ЁЙЖТИКПУМРЕРЁ.6 МД.466

28 ЦО. ЖКЗУЙЛРФНСЁСЖ.6 НЕ.466

29 ЧП. ЗЛИФКМСХОТЖТЗ.6 ОЁ.466

30 ШР. ИМЙХЛНТЦПУЗУИ.6 ПЖ.466

31 ЩС. ЙНКЦМОУЧРФИФЙ.6 РЗ.466

32 ЪТ. КОЛЧНПФШСХЙХК.6 СИ.466  
  
Замечаем, что на 25 сдвиге мы получили правильный (расшифрованный) ответ: “25 УЛ. ДЗЕРЖИНСКОГОД.6 КВ.466”

Теперь перейдем к коду. Чтобы унифицировать методы работы с данными и упростить их, я написал небольшую программу (csv\_writer.py), которая читает исходные данные из exel таблицы и записывает в CSV файл. Так же была написана функция - “contains\_key” - в качестве вспомогательной. А к ней еще два массива с возможными ключами (ключевые слова) для поиска расшифрованных ответов (address\_key\_words и mail\_key\_words), пример их содержания:

1. УЛ. – улица
2. КВ. – квартира
3. GMAIL.COM - популярные email адреса

И так далее…

Затем, программа (task3.py) читает эти данные и начинает перебирать все возможные сдвиги по соответствующему алфавиту. Правильный (расшифрованный) ответ выбирается с помощью функции “contains\_key” – которая проверяет, содержит ли строка хотя бы одно из заданных ключевых слов (из массивов, описанных выше). После чего, найденные расшифрованные данные, вместе с key-сдвигом, записываются в csv файл (итоговый датасет – “decoded\_dataset.csv”) с помощью функции “write\_to\_csv” по определенным столбцам: ('email', 'email\_key', 'address', 'address\_key').  
  
**Блок схема решения:**

