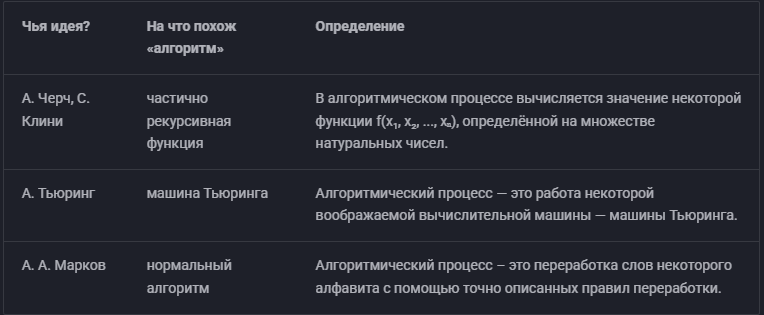
# Формализация понятия алгоритма. Машина Тьюринга. Нормальные алгоритмы Маркова. Лямбда-исчисление Черча. Тезис Черча. Теорема о рекурсии.



## Машина Тьюринга

Алан Тьюринг, работая над проблемой алгоритмической вычислимости, предложил в 1936 году концепцию **абстрактной машины**, известной как **машина Тьюринга**.

Основные компоненты машины Тьюринга:  
- **Лента**, разделённая на ячейки, содержащие символы из конечного алфавита.  
- **Головка**, которая может читать символы с ленты, записывать новые символы и перемещаться влево или вправо.  
- **Конечный набор состояний**, определяющих, как машина реагирует на прочитанные символы.  
- **Таблица переходов**, содержащая правила, указывающие, какое действие совершить в зависимости от текущего состояния и символа на ленте.

Тьюринг доказал, что его машина может моделировать работу любого алгоритма, если представить его в виде последовательности элементарных операций. Машины Тьюринга стали одной из фундаментальных моделей вычислений и сыграли важную роль в развитии теории сложности.

## Нормальные алгоритмы Маркова

Советский математик Андрей Марков в 1950-х годах предложил **нормальные алгоритмы** — формальную модель, основанную на преобразовании строк символов с помощью четко определённых подстановочных правил.

Основные принципы нормальных алгоритмов Маркова:  
- Алгоритм представлен в виде набора правил подстановки.  
- Вычисления сводятся к последовательному применению этих правил к входному слову.  
- Процесс завершается, когда не остаётся применимых подстановок.

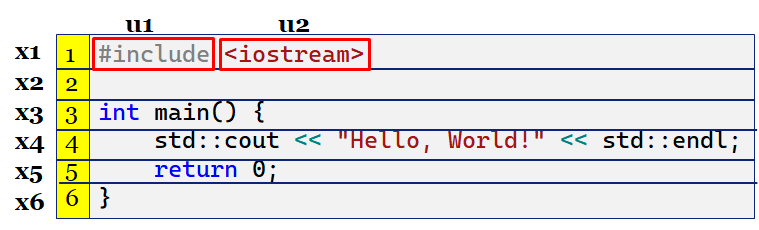
Нормальные алгоритмы эквивалентны машинам Тьюринга и λ𝜆-исчислению, что подтверждает гипотезу Черча — Тьюринга. Они оказались удобными для изучения свойств формальных языков и стали одной из ключевых моделей в теории алгоритмов.

## lambda-исчисление Черча

Вариант строгого определения алгоритма, предложенный Черчем и Клини, рассматривает алгоритмический процесс как процесс вычисления значений некоторой функции f, определённой на множестве натуральных чисел N.

Предположим, что имеется некоторый алгоритм, который работает следующим образом:

* На вход алгоритма поступает объект P, который представляет собой набор данных u1,u2,…,un, где каждый ui является словом в некотором алфавите A.
* Слова в алфавите A образуют счётное множество, и эти слова могут быть занумерованы натуральными числами.
* Таким образом, каждый конструктивный объект ui имеет номер x∈N, где x — это натуральное число, которое используется для идентификации объекта.



В данном контексте предполагается, что алгоритм выполняет вычисления, результатом которых является некоторое значение функции f, определённой на множестве натуральных чисел N. Функция f представляет собой отображение, которое, в общем случае, может быть определено не для всех входных данных.

Когда имеется некоторый объект P на входе алгоритма, это вовсе не означает, что алгоритм всегда приводит к некоторому объекту Q на выходе. В некоторых случаях алгоритм может не иметь результата, что делает вычисление функции f **неопределённым** для определённых наборов данных.  
В связи с этим возникает понятие **частичной функции**. Частичная функция — это такая функция, которая может быть определена только для подмножества входных данных. Например, алгоритм может правильно обработать один набор входных данных и не дать результата для другого.

Частичная функция играет ключевую роль в контексте алгоритмов, поскольку:

* Алгоритм не всегда имеет гарантированный результат для всех возможных входных данных.
* Для некоторых наборов данных алгоритм может не завершиться или не выдать корректного результата.

Таким образом, несмотря на то, что алгоритм представляет собой конечную последовательность операций, его поведение может зависеть от конкретных входных данных, что и ведёт к необходимости использования частичных функций.

## Тезис Черча

**Тезис Черча**

***Каждая вычислимая функция частично рекурсивна.***

Сможем ли мы доказать тезис Черча? Нет, поскольку у нас нет точного определения вычислимой функции. Тезис Черча — это просто строгое определение алгоритма. Это положение является соглашением, возникшим в результате длительного исследования интуитивного понятия алгоритма. Тезис Черча является естественнонаучным фактом, который подтверждается опытом, накопленным математикой за весь период её развития. Тезис Черча является достаточным средством, чтобы придать необходимую точность формулировкам алгоритмических проблем и делает возможным доказательство их неразрешимости.

## Теорема о рекурсии

Теорема о рекурсии утверждает, что всегда можно найти эквивалентную этой функции p(y)=V(p,y), которая будет использовать саму себя для вычисления значения.

Пусть V(n,x)— вычислимая функция. Тогда найдется такая вычислимая функция p𝑝, что для всех y𝑦:

p(y)=V(p,y)

Иначе говоря, если рассмотреть V(x, y), как программу, использующую x в качестве исходного кода и выполняющую действие над y, то теорема о рекурсии показывает, что мы можем написать эквивалентную ей программу p(y) = V(p, y), которая будет использовать собственный исходный код.

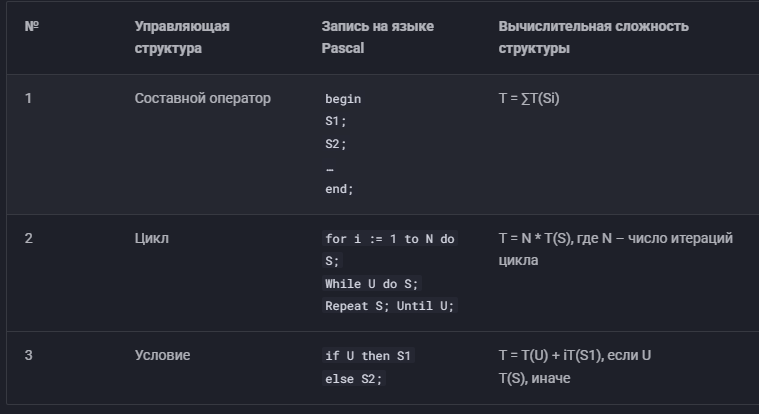
# Абстрактная модель вычислений. Вычислительные ресурсы. Асимптотический анализ алгоритмов (нотация big-O).

## Абстрактная модель вычислений

Поскольку мы работаем с абстрактными алгоритмами, то следует зафиксировать некоторые набор утверждений

1. **Алгоритм рассматривается как набор операций и управляющих структур.** Каждая операция выполняет определённое действие, которое занимает некоторое время, а управляющие структуры (например, циклы, условия) определяют, как именно выполняются операции.
2. **Каждому виду операций сопоставляется временная стоимость.** Стоимость операции измеряется в абстрактных единицах времени (шага), которые учитываются при оценке времени работы алгоритма.
3. **Время работы алгоритма в целом равно сумме стоимостей составляющих его операций с учётом вложенности управляющих структур.** Это означает, что для точной оценки необходимо учитывать не только время, которое тратится на отдельные операции, но и то, как они расположены внутри других операций (например, сколько раз выполняется вложенный цикл).

Время работы алгоритма в целом равно сумме стоимостей составляющих его операций с учетом вложенности управляющих структур



## Вычислительные ресурсы

аааээээ написано ниче не было, но чисто логически ЦПУ/ГПУ (в особых случаях), оперативная память

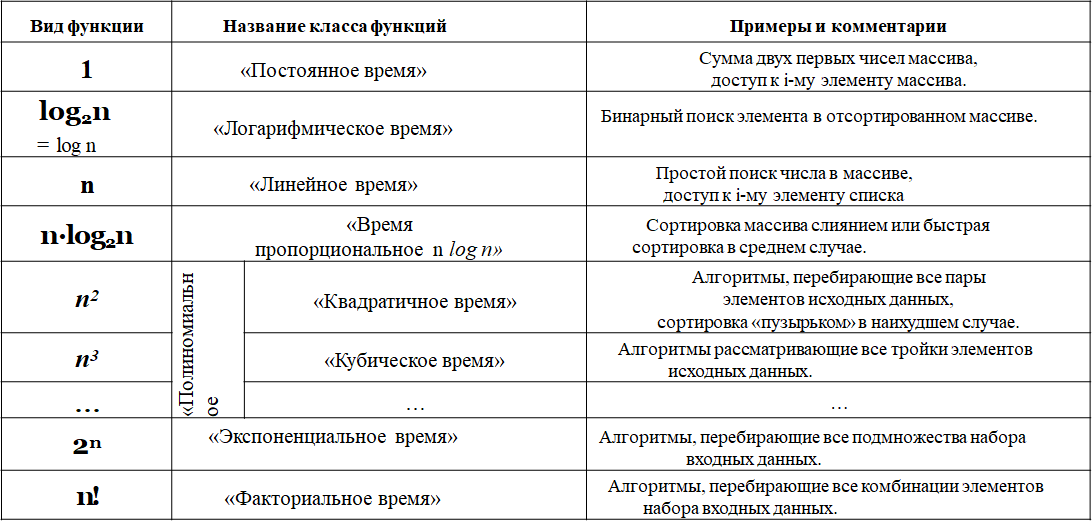
## Асимптотический анализ алгоритмов (нотация big-O)

Ο(g(n)) — асимптотически верхняя граница, для которой все функции будут находиться ниже.

Θ(g(n)) — асимптотически точная граница.

Ω(g(n)) — асимптотически нижняя граница, для которой все функции будут находиться выше.

Примеры:

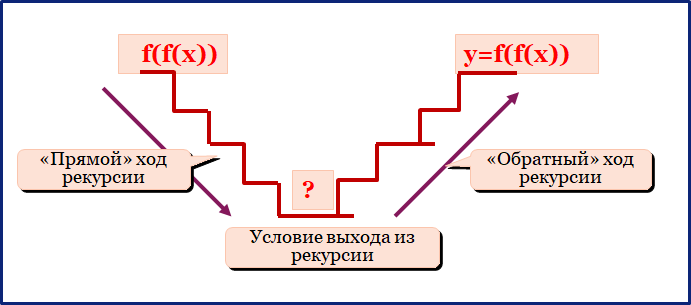


# Вычисление рекурсивной функции. Асимптотический анализ алгоритмов рекурсивных алгоритмов. Анализ рекуррентных соотношений. Мастер-теорема.

## Вычисление рекурсивной функции

Рекурсивная функция работает в два этапа:

* **Прямой ход рекурсии** (углубление) – функция вызывает саму себя, пока не достигнет условия выхода.
* **Обратный ход рекурсии** (возвращение) – начинается, когда выполнено условие выхода, и вычисления сворачиваются, возвращаясь по цепочке вызовов.



Рассмотрим вызов функции factorial(5)

function factorial(int:a):

    if a == 0 return 1

    return a \* factorial(a-1)

1. factorial(5) вызывает factorial(4), factorial(4) вызывает factorial(3), и так далее… (прямой ход).
2. Когда factorial(0) возвращает 1, начинается **обратный ход (разворачивание стека)**

Чтобы хранить порядок операций, необходимых для выполнения компьютер формируется **стек вызова**. Системный стек — это область памяти, используемая для хранения:  
- **Адресов возврата** (куда возвращаться после выполнения функции).  
- **Локальных переменных** функции.  
- **Аргументов, переданных в функцию**.

Когда вызывается функция, для неё выделяется **кадр стека (stack frame)**.  
При выходе из функции её кадр удаляется, и **управление передаётся по адресу возврата**.

Если программа использует слишком много памяти в стеке, например, из-за слишком глубокой рекурсии, может произойти **переполнение стека** (stack overflow).  
Это может привести к сбоям в работе программы или системы.

## Асимптотический анализ алгоритмов рекурсивных алгоритмов

Рекурсивный алгоритм можно описать уравнением вида:

𝑇(𝑛)=𝑎⋅𝑇(𝑛/𝑏)+𝑓(𝑛)

где:

* 𝑇(𝑛)— время работы алгоритма на входе размера 𝑛*n*,
* 𝑎 — количество рекурсивных вызовов,
* 𝑛𝑏— размер подзадачи,
* 𝑓(𝑛) — время на разбиение задачи и комбинирование результатов.

это больше лирическое отступление, подробнее – следующие два пункта

## Анализ рекуррентных соотношений

Один из методов анализа алгоритмов — это метод рекуррентных соотношений. Этот метод позволяет вычислять временную сложность рекурсивных алгоритмов на основе их рекуррентных формул. Рассмотрим на примере задачи вычисления чисел Фибоначчи с помощью рекурсии.

Шаг 1: Подставляем рекуррентное соотношение для T(n−1)

Мы начинаем с рекуррентного соотношения:

T(n)=2⋅T(n−1)+1

Теперь подставляем выражение для T(n−1) из предыдущего шага, которое равно:

T(n−1)=2⋅T(n−2)+1

Тогда получаем:

T(n)=2⋅(2⋅T(n−2)+1)+1=4⋅T(n−2)+3

Шаг 2: Подставляем рекуррентное соотношение для T(n−2)

Теперь подставим для T(n−2):

T(n−2)=2⋅T(n−3)+1

Тогда выражение для T(n) примет вид:

T(n)=2⋅(4⋅T(n−3)+3)+1=8⋅T(n−3)+7

Шаг 3: так же для n-3…

Шаг 4: Закономерность и общее решение

Из вышеизложенного видно, что на каждом шаге множитель перед T(n−k) удваивается, а добавляемое число увеличивается по закономерности 2k−1.

Таким образом, можно записать общую форму для T(n):

T(n)=2k⋅T(n−k)+(2k−1)

Для любого k, где k — это количество шагов рекурсии.

Шаг 5: Определение k

Для того чтобы решить это уравнение, нам нужно подставить T(0)=1, то есть:

T(0)=1при n−k=0

Следовательно, k=n.

Шаг 6: Подставляем k=n

Подставляем k=n в общее решение:

T(n)=2n⋅T(0)+(2n−1)=2n+(2n−1)=2n+1−1

Таким образом, мы получаем:

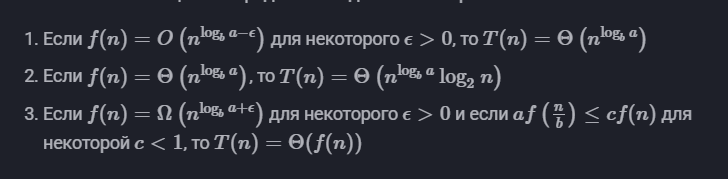
T(n)=O(2n) (+1-1 убрали т.к. наибольший эффект имеет 2n, можно и оставить для большей точности)

## Мастер-теорема

(работает только для рекурсивных алгоритмов?) Суть Мастер-теоремы в том, что рекуррентное соотношение можно представить в общем виде как:

T(n)=a\*T(n/b)+f(n)

где a≥1, b>1, а f(n) — заданная функция. Здесь a означает количество подзадач, на которое делится исходная подзадача, а b — во сколько раз уменьшается количество данных при вызове рекурсивной функции. f(n) — это работа по обработке 1 вызова метода рекурсии. Далее ответ зависит от одного из трёх случаев, с помощью которого асимптотические оценки определяются достаточно просто (след. скрин):



# Представление чисел в памяти компьютера. Проблемы компьютерных вычислений, вызванные использованием стандарта IEEE754. Применение побитовых операторов.

## Представление чисел в памяти компьютера

Машинное представление чисел неразрывно связано с понятием **типа данных**.

Тип данных – это характеристика данных, определяющая:

- диапазон их значений,

- набор допустимых операций над ними,

- внутреннее представление в памяти компьютера.

Типы данных для хранения **целых чисел** делятся на **беззнаковые** и **знаковые**.

* **Беззнаковые** типы данных могут хранить только **неотрицательные целые числа**.
* **Знаковые** предназначены для хранения **и положительных, и отрицательных чисел**.

Числа беззнаковых типов хранятся в памяти **в двоичном виде**.  
Для получения представления беззнакового десятичного числа в памяти компьютера:  
1. Переводим число в **двоичную систему счисления**.  
2. Дописываем **незначащие нули слева** до нужной разрядности.

Для представления знаковых целых чисел используются три способа:  
1. **Прямой код**;  
2. **Обратный код**;  
3. **Дополнительный код**.

Все три способа используют самый левый (старший) разряд битового набора  
длины k для кодирования знака числа:  
- знак **"плюс"** кодируется **нулем** (00),  
- знак **"минус"** — **единицей** (11).

Остальные k−1 разрядов (называемые **мантиссой** или **цифровой частью**)  
используются для представления **абсолютной величины** числа.

При записи числа в **прямом коде** (англ. *Signed magnitude representation*) старший разряд является знаковым разрядом. Пример: Число **−5** в восьмибитном типе данных, использующем прямой код, будет выглядеть так: 10000101.

Достоинства представления чисел с помощью прямого кода:  
1. Получить прямой код числа достаточно просто.  
2. Из-за того, что 0 обозначает **+**, коды положительных чисел относительно беззнакового кодирования остаются неизменными.  
3. Количество положительных чисел равно количеству отрицательных.

Недостатки представления чисел с помощью прямого кода:  
1. Выполнение арифметических операций с отрицательными числами требует усложнения архитектуры центрального процессора. Например, для вычитания невозможно использовать сумматор, необходима отдельная схема для этого.  
2. Существуют два нуля: **−0** (100…000) и **+0** (000…000), что усложняет арифметическое сравнение.  
3. Из-за весьма существенных недостатков прямой код используется очень редко.

**Обратный код** двоичного числа образуется по следующему правилу:  
- Обратный код **положительных чисел** совпадает с их **прямым кодом**.  
- Обратный код **отрицательного числа** содержит единицу в знаковом разряде числа, а значащие разряды числа заменяются на инверсные, т.е. нули заменяются единицами, а единицы — нулями.

Пример:  
1. A10=+10, A2=+1010, [A2]ок=[A2]п=0|1010  
2. V10=−15, V2=−1111, [V2]ок=1|0000

Важные свойства обратного кода:  
1. **Сложение положительного числа** C с его **отрицательным значением** в обратном коде дает т.н. **машинную единицу** MEок=1|11…11, состоящую из единиц в знаковом и в значащих разрядах числа.  
2. **Нуль в обратном коде** имеет двоякое значение. Он может быть как **положительным числом** 0|00…000|00…00, так и **отрицательным** 1|11…11. Значение отрицательного числа совпадает с машинной единицей MEок.  
3. Двойственное представление нуля явилось причиной того, что в современных ЭВМ все числа представляются не обратным, а **дополнительным кодом**.

Чаще всего для представления отрицательных чисел используется **код с дополнением до двух** (*Two's Complement*). (был и до единицы, но че то решил не вставлять… надеюсь не попадется ¯\\_(ツ)\_/¯ )

Алгоритм получения дополнительного кода числа:

1. Если число неотрицательное, то в старший разряд записывается ноль, далее записывается само число.
2. Если число отрицательное:
3. Все биты модуля числа инвертируются (все единицы меняются на нули, а нули — на единицы).
4. К инвертированному числу прибавляется единица.
5. К результату дописывается знаковый разряд, равный единице.

Пример:  
Переведём число **−5** в дополнительный восьмибитный код:  
- Прямой код модуля −5: 00000101  
- Обратный код: 1111010  
- Прибавляем 1: 1111011  
- Приписываем 1 как знаковый разряд: 11111011

Математическое представление:  
Дополнительный код отрицательного числа **A**, хранящегося в **n** битах, равен:

2n−|A|

Пример восстановления числа из дополнительного кода:  
Переведём **11111011** обратно в число:  
- Инвертируем: 00000100  
- Прибавляем 1: 00000101  
- Получаем модуль исходного числа **−5**.

Достоинства представления чисел с помощью кода с дополнением до двух:

* Возможность заменить операцию вычитания операцией сложения, что упрощает архитектуру процессора и увеличивает быстродействие.
* Нет проблемы двух нулей (однозначность представления нуля).

Недостатки представления чисел с помощью кода с дополнением до двух:

* Ряд положительных и отрицательных чисел несимметричен, но это не так важно, поскольку дополнительный код решает гораздо более важные задачи для представления целых чисел.
* Числа в дополнительном коде нельзя сравнивать как беззнаковые или вычитать без расширения разрядности.

## Проблемы компьютерных вычислений, вызванные использованием стандарта IEEE754

float не может точно представить 123456789, вместо этого число округляется.  
В результате a ≈ 123456792, а b ≈ 123456784. (короче говоря фигня с флоат поинтом, примеры: слишком далеко залетел в роблоксе или любой другой 3д игре с открытым миром и вещи начинает в прямом смысле плющить)

Если a ≈ 123456792, а b ≈ 123456784, то разница будет **8**, а не **1**.

Ошибки вызванные сдвигом мантисс. Циклические дыры.

Хотя с относительной погрешностью здесь всё в порядке, есть и другие проблемы.  
Во-первых, работать с числами можно только в узком диапазоне числовой оси, где мантиссы пересекаются.  
Во-вторых, для каждого исходного числа существует предел выполнения цикла, называемый **"Циклической дырой"**. Если существует цикл, в котором исходное число суммируется с суммой, то существует численный предел суммы для этого числа.  
То есть, сумма, достигнув определённой величины, перестаёт увеличиваться при её сложении с исходным числом.

"Грязный ноль" — это ситуация, когда вы или программа считаете, что переменная, не равная нулю, равна нулю.

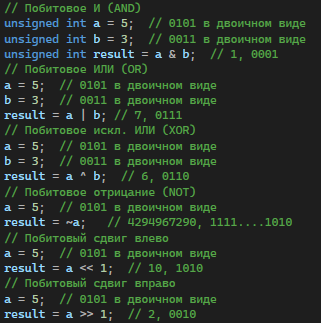
При компьютерных вычислениях можно выделить два вида округления:

1. **Результат выполненной арифметической операции всегда округляется**.
2. **Вывод и ввод вещественного числа в окно Windows происходит с округлением**.

В первом случае переменная округляется по одному из четырёх типов округления IEEE754. По умолчанию округление происходит к ближайшему целому.  
При этом переменная принимает новое округлённое значение.

Во втором случае переменная не меняет своего значения, а просто в окно Windows выводится округлённое значение вещественного числа.  
Получается, что исходная переменная и её отображение в окне Windows — это разные числа. Это не ошибка формата IEEE754, а ошибка Windows.

## Применение побитовых операторов



# Основы параллелизации. Параллелизация на уровне процессора. Векторные операции. Интрисики.

## Основы параллелизации

??? просто название темы ???

## Параллелизация на уровне процессора

Я, честно говоря, хз что сюда вставить, так что в основном вода… картинка на всякий случай



**С программной точки зрения**, процессорная архитектура определяет набор регистров, команд, их структуру и способ выполнения, в результате чего, с одной стороны, программы, собранные для процессоров одной архитектуры, могут выполняться практически на всех процессорах одинаковой (или подобной) архитектуры, а с другой – не смогут работать на процессорах иной архитектуры

## Векторные операции

**Векторные вычисления** — такие вычисления, когда при выполнении одной инструкции процессора производится одновременно несколько однотипных операций. Этот принцип в настоящее время реализован не только в специализированных процессорах, но и в процессорах архитектуры x86 и ARM в виде векторных расширений. Эти расширения представляют собой специальные векторные регистры с повышенной относительно регистров общего назначения разрядностью. Для работы с этими регистрами имеются специальные векторные инструкции, которые дополняют систему инструкций процессора.  
В основном работа с векторами отходит на ГПУ, чем ЦПУ, т.к. первые более оборудованы для их расчёта

## Интрисики

Можно получить доступ к векторным инструкциям и в программе на языке высокого уровня (в частности, C/C++) без ассемблерных вставок. Для этого используются так называемые **интринсики** (intrinsics) — встроенные объекты компилятора. В заголовочном файле объявлен один или несколько типов данных (с точки зрения программиста, это массив фиксированной длины, но без возможности доступа к элементам этого массива), переменной одного из этих типов соответствует векторный регистр. Также в заголовочном файле объявлены функции, которые принимают аргументы, возвращают значения указанных выше типов и производят с точки зрения программиста те же операции над данными, что и соответствующие им векторные инструкции. С помощью интринсиков можно написать программу на языке высокого уровня, близкую или равную по производительности программе, написанной на языке ассемблера.

# Процесс и поток. Параллелизация на уровне данных. Проблема разделения ресурсов. Задача обедающих философов. Средства синхронизации.

## Процесс и поток

Основным понятием в любой операционной системе является процесс: абстракция, описывающая выполняющуюся программу. Процесс — это просто экземпляр выполняемой программы, включая текущие значения счетчика команд, регистров и переменных. Для реализации модели процессов операционная система ведет **таблицу процессов** (состоящую из массива структур), в которой каждая запись соответствует какому-нибудь процессу. (тот же диспетчер задач)

Потоки позволяют распараллеливать работу процесса(-ов), если процессор на то пригоден (лучше я не придумал понятия ¯\\_(ツ)\_/¯ )

## Параллелизация на уровне данных

Опять название темы?? Ну или то что какую-нибудь там сортировку массива можно распараллелить, разбив массив на несколько частей тдтп. Вспомнить типы сортировок короче

## Проблема разделения ресурсов

Многопоточные приложения, хотя и предлагают значительные преимущества в плане производительности и эффективности использования ресурсов, также связаны с рядом потенциальных проблем и сложностей, связанных с параллельным выполнением кода.

Рассмотрим ситуацию, когда два процесса имеот общие данные и, предположим, результат выполнения зависит от того, какой поток получит доступ первым. Начинаются так называемые гонки на запись ресурсов.

В качестве решения можно использовать механизмы блокировки, такие как мьютексы или семафоры, для обеспечения взаимоисключающего доступа к общим ресурсам

## Задача обедающих философов

Задача об обедающих философах — классический пример, используемый в информатике для иллюстрации проблем синхронизации при разработке параллельных алгоритмов и техник решения этих проблем.

Задача была сформулирована в 1965 году Эдсгером Дейкстрой как экзаменационное упражнение для студентов. В качестве примера был взят конкурирующий доступ к ленточному накопителю.

ЗАДАЧА

Пять безмолвных философов сидят вокруг круглого стола, перед каждым философом стоит тарелка спагетти. На столе между каждой парой ближайших философов лежит по одной вилке.

Каждый философ может либо есть, либо размышлять. Приём пищи не ограничен количеством оставшихся спагетти — подразумевается бесконечный запас. Тем не менее, философ может есть только тогда, когда держит две вилки — взятую справа и слева (альтернативная формулировка проблемы подразумевает миски с рисом и палочки для еды вместо тарелок со спагетти и вилок).

Каждый философ может взять ближайшую вилку (если она доступна) или положить — если он уже держит её. Взятие каждой вилки и возвращение её на стол являются раздельными действиями, которые должны выполняться одно за другим.

Вопрос задачи заключается в том, чтобы разработать модель поведения (параллельный алгоритм), при котором ни один из философов не будет голодать, то есть будет вечно чередовать приём пищи и размышления. (конец задачи)

Задача сформулирована таким образом, чтобы иллюстрировать проблему избежания взаимной блокировки (англ. deadlock) — состояния системы, при котором прогресс невозможен.

Решение задачи

**Официант**

**Описание**  
В этом решении задача решается с помощью посредника — официанта, который управляет доступом к вилкам. Каждый философ должен сначала запросить у официанта вилки, прежде чем приступить к еде, и вернуть их, когда закончит.

**Иерархия ресурсов**

**Описание**  
В этом решении философы придерживаются порядка при выборе вилок. Чтобы избежать взаимных блокировок, философы всегда берут вилку с меньшим номером. Это гарантирует, что никто не окажется заблокированным, пытаясь захватить те же вилки.

**Решение с использованием мьютексов**

**Описание**  
В этом решении для каждого философа создается объект, который управляет доступом к его вилкам. Используются мьютексы (простые механизмы синхронизации), чтобы гарантировать, что вилки берутся и освобождаются корректно.

## Средства синхронизации

Атомарные переменные

Атомарные переменные позволяют выполнять операции (чтение, запись, модификация) над переменными (в случае C++ над объектами) без использования традиционных средств синхронизации (например мьютексов) в многопоточных приложениях. Для достижения такого результата используются аппаратные инструкции (lock-free)

Мьютексы

**Мьютексы («mutual exclusion» — взаимное исключение)** используются для обеспечения взаимного исключения, что позволяет предотвратить одновременный доступ к общим ресурсам в многопоточных приложениях. Они представляют собой блокировку, которая может находиться в одном из двух состояний: заблокированном и разблокированном.

Мьютексы предоставляют механизм блокировки, который позволяет потокам последовательно получать доступ к общему ресурсу. Когда поток захватывает мьютекс, другие потоки не могут получить доступ к защищенному им ресурсу до тех пор, пока первый поток не освободит мьютекс.

Семафоры

**Семафор** — примитив синхронизации работы процессов и потоков, в основе которого лежит счётчик, над которым можно производить две атомарные операции: увеличение и уменьшение значения на единицу.

Условные переменные (Condition variables)

Условные переменные представляют собой механизм синхронизации, который позволяет потокам приостанавливать выполнение до тех пор, пока не наступит определённое условие. Условные переменные используются в сочетании с мьютексами для координации действий между несколькими потоками, позволяя одним потокам ждать («спать»), пока другие потоки не изменят состояние программы и не сигнализируют об этом.

Условные переменные используются для блокировки потока до тех пор, пока не наступит определенное условие. Они тесно связаны с мьютексами и часто используются для ожидания определенных условий выполнения задач.

# Алгоритмы численных приближений. Идея целочисленного интегрирования. Ресурс параллелизации в алгоритмах численных приближений.

## Алгоритмы численных приближений

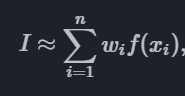
Яхз лол, похоже просто название темы

## Идея целочисленного интегрирования

Численное интегрирование (историческое название: (численная) квадратура) — вычисление значения определённого интеграла (как правило, приближённое). Под численным интегрированием понимают набор численных методов для нахождения значения определённого интеграла.

Численное интегрирование применяется, когда:

1. Сама подынтегральная функция не задана аналитически. Например, она представлена в виде таблицы (массива) значений в узлах некоторой расчётной сетки.
2. Аналитическое представление подынтегральной функции известно, но её первообразная не выражается через аналитические функции. Например, f(x)=exp(−x2)

Основная идея большинства методов численного интегрирования состоит в замене подынтегральной функции на более простую, интеграл от которой легко вычисляется аналитически. При этом для оценки значения интеграла получаются формулы вида: , где n — число точек, в которых вычисляется значение подынтегральной функции. Точки xi называются узлами метода, числа wi — весами узлов.

## Ресурс параллелизации в алгоритмах численных приближений

Если правильно понимаю, то разбиение области на подобласти и параллельный расчёт их и будет ресурсом параллелизации

# Методы оптимизации "Разделяй и властвуй". Бинарное возведение в степень. Алгоритм Карацубы. Ресурс параллелизации при реализации метода декомпозиции.

## Методы оптимизации "Разделяй и властвуй"

Скорее всего просто тема, но на всякий случай: Важнейший шаг в работе с алгоритмами «разделяй и властвуй» — это соединить решения подзадач в решение изначальной задачи.

**ПАРАДИГМА РАЗДЕЛЯЙ И ВЛАСТВУЙ**

1. Разделить входные данные на более мелкие подзадачи.

2. Решить подзадачи рекурсивным методом.

3. Объединить решения подзадач в решение исходной задач

## Бинарное возведение в степень

**Бинарное возведение в степень** — приём, позволяющий возводить любое число в n-ую степень за O(logn) умножений (вместо n умножений при обычном подходе).

Бинарное возведение в степень применимо не только к умножению чисел, но и к любым ассоциативным операциям — таким, для которых для любых a, b и c выполняется:

(a∘b)∘c=a∘(b∘c)

Наиболее очевидное обобщение — на остатки по модулю:

anmodb

## Алгоритм Карацубы

Основная идея алгоритма очень простая: в нём умножение двух чисел длины n небольшим алгебраическим трюком сводится к трём умножениям чисел длины n/2. Число элементов на каждом уровне рекурсии будет расти, но на самом нижнем будет суммарно всего O(nlog23) элементов, чем и объясняется такая странная асимптотика.

В нашей версии алгоритма мы будем перемножать не числа, а многочлены. Любое число можно можно представить в виде многочлена, если вместо x подставить основание системы счисления, а в качестве коэффициентов можно взять последовательность цифр числа

## Ресурс параллелизации при реализации метода декомпозиции

Опять разбиение на подмножества???

# Оценка эффективности параллельных вычислений. Модель вычислений в виде графа "операции-операнды". Показатели эффективности параллельного алгоритма.

## Оценка эффективности параллельных вычислений

Оценка эффективности параллельных вычислений может быть проведена двумя подходами:  
- **оценка распараллеливания конкретного алгоритма**  
- **оценка максимально возможного ускорения**

Для точной оценки параллельности используется модель вычислений в виде **графа "операции-операнды"**, которая позволяет описать существующие **информационные зависимости** между операциями и представить вычислительный процесс.

Эти методы дают возможность оценить **максимально возможный параллелизм**, что помогает определить пределы возможного ускорения при оптимальном использовании параллельных вычислений.

## Модель вычислений в виде графа "операции-операнды"

Для описания информационных зависимостей в алгоритмах решения задач используется модель в виде графа "операции-операнды" (см., например, Bertsekas and Tsitsiklis (1989), Воеводин В.В. и Воеводин Вл.В. (2002)). При этом предполагается, что время выполнения любых вычислительных операций одинаково и равно 1 (в тех или иных единицах измерения), а передача данных между вычислительными устройствами осуществляется мгновенно, без затрат времени. Это предположение может быть справедливо, например, в системах с общей разделяемой памятью.

Мб еще че то ???

## Показатели эффективности параллельного алгоритма

Ускорение (Speedup)

Ускорение, которое достигается при использовании параллельного алгоритма для p процессоров, по сравнению с последовательным вариантом вычислений, определяется величиной:

S(n,p)=T(n)Tp(n)

где T(n) — время выполнения последовательного алгоритма, а Tp(n) — время выполнения параллельного алгоритма с pp процессорами. Ускорение используется для параметризации вычислительной сложности задачи и может быть интерпретировано, например, как количество входных данных задачи.

Эффективность (Efficiency)

Эффективность использования параллельных процессоров при решении задачи определяется соотношением:

E(n,p)=S(n,p)p=T(n)pTp(n)

Эффективность определяет среднюю долю времени, в течение которой процессоры реально используются для решения задачи.

Сверхлинейное ускорение

При определённых обстоятельствах ускорение может превышать число процессоров, в этом случае говорят о **сверхлинейном ускорении**. Хотя это явление может казаться парадоксальным, оно может иметь место на практике. Причины сверхлинейного ускорения могут быть следующими:

1. **Неравноправность выполнения последовательной и параллельной программ**: при решении задачи на одном процессоре может не хватить оперативной памяти для хранения всех данных, и необходимым становится использование более медленной внешней памяти. При использовании нескольких процессоров оперативная память может быть разделена между ними, что улучшает ситуацию.
2. **Нелинейный характер зависимости сложности задачи от объема обрабатываемых данных**: например, алгоритм пузырьковой сортировки имеет квадратичную зависимость от числа упорядочиваемых элементов, что может привести к ускорению, превышающему число процессоров при распределении данных.
3. **Различие в вычислительных схемах последовательных и параллельных методов**: в некоторых случаях различия в способах вычислений могут привести к ускорению, которое превышает число процессоров.

Противоречие между ускорением и эффективностью

Попытки улучшить один из показателей качества параллельных вычислений (ускорение или эффективность) могут привести к ухудшению другого. Например:

* Повышение ускорения часто достигается за счёт увеличения числа процессоров, что обычно ведет к снижению эффективности.
* Повышение эффективности достигается при уменьшении числа процессоров, и идеальная эффективность (E(n)=1) может быть достигнута при использовании одного процессора.

Как результат, разработка методов параллельных вычислений часто требует выбора компромиссного варианта, учитывая желаемые показатели ускорения и эффективности.

Стоимость вычислений

Стоимость вычислений может быть определена как произведение времени параллельного решения задачи на число используемых процессоров:

C=p⋅Tp

Стоимостно-оптимальный (cost-optimal) параллельный алгоритм — это метод, стоимость которого пропорциональна времени выполнения наилучшего последовательного алгоритма.

В следующем пункте будет рассмотрен учебный пример решения задачи вычисления частных сумм для последовательности числовых значений, где будут использованы эти показатели для характеристики эффективности рассматриваемых параллельных алгоритмов.

# Оценка эффективности параллельных вычислений. Закон Амдала. Закон Густавсона - Барсиса. Практические приложения закона Амдала.

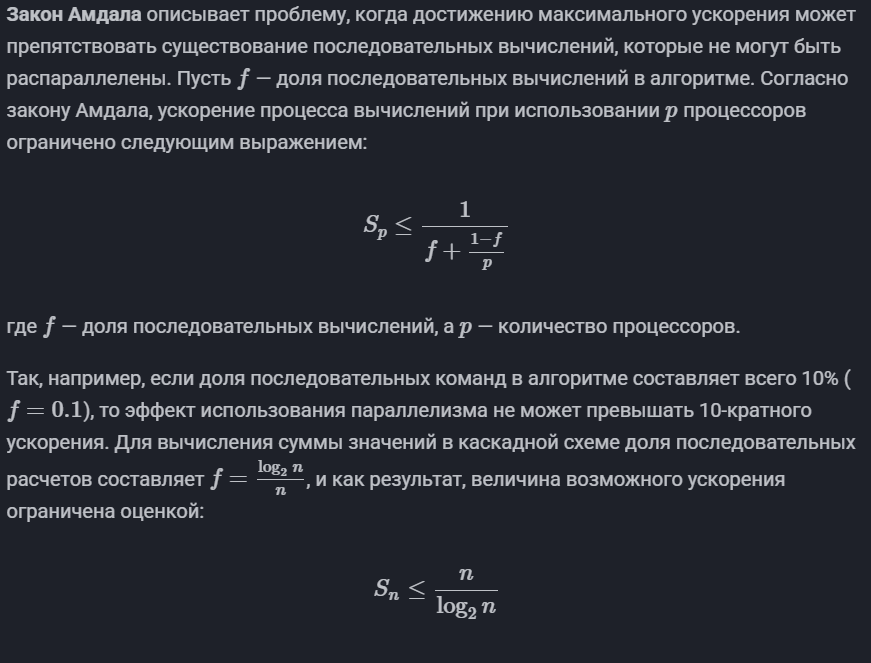
## Оценка эффективности параллельных вычислений

Оценка эффективности параллельных вычислений может быть проведена двумя подходами:  
- **оценка распараллеливания конкретного алгоритма**  
- **оценка максимально возможного ускорения**

Для точной оценки параллельности используется модель вычислений в виде **графа "операции-операнды"**, которая позволяет описать существующие **информационные зависимости** между операциями и представить вычислительный процесс.

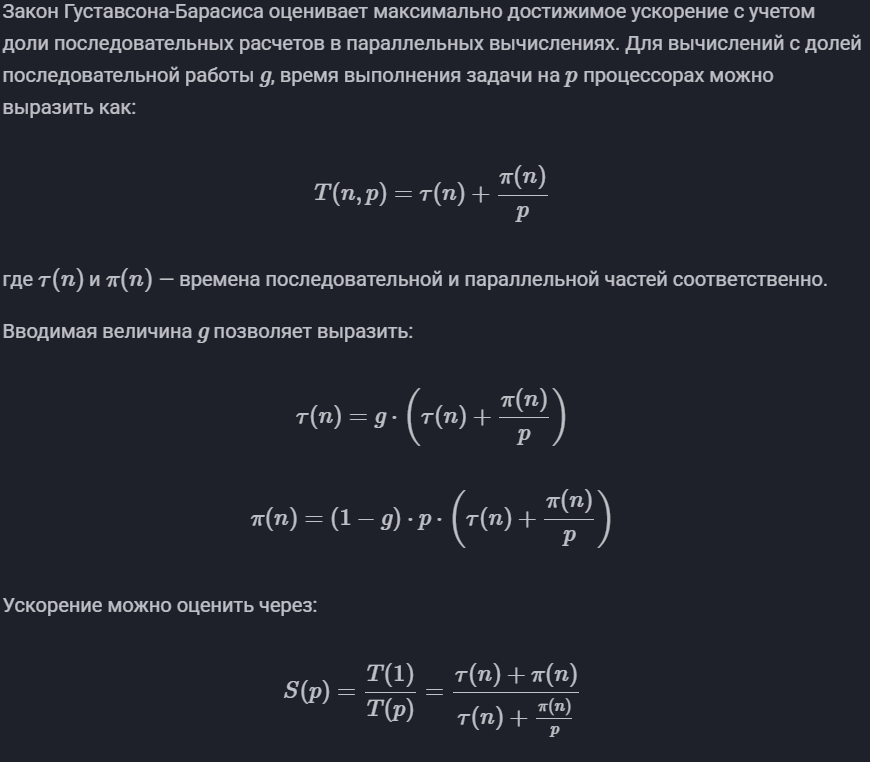
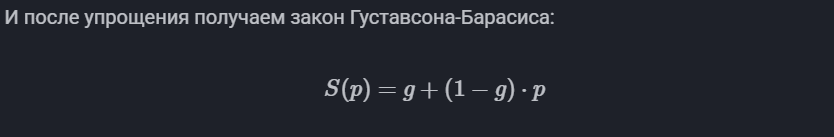
Эти методы дают возможность оценить **максимально возможный параллелизм**, что помогает определить пределы возможного ускорения при оптимальном использовании параллельных вычислений.

## Закон Амдала

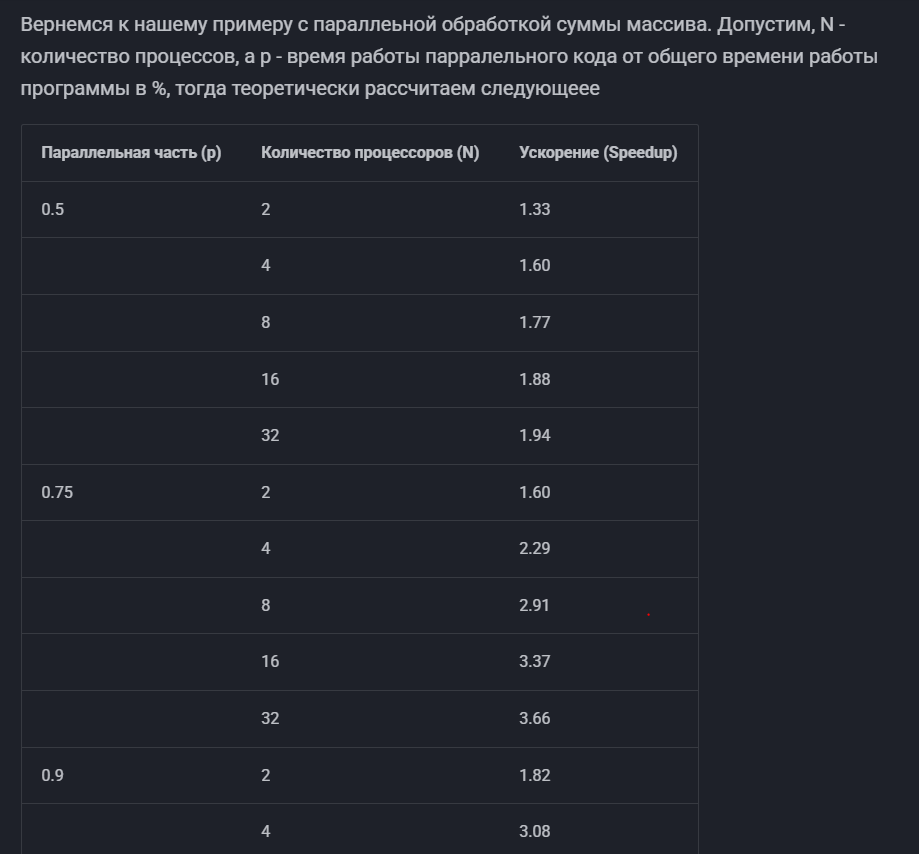
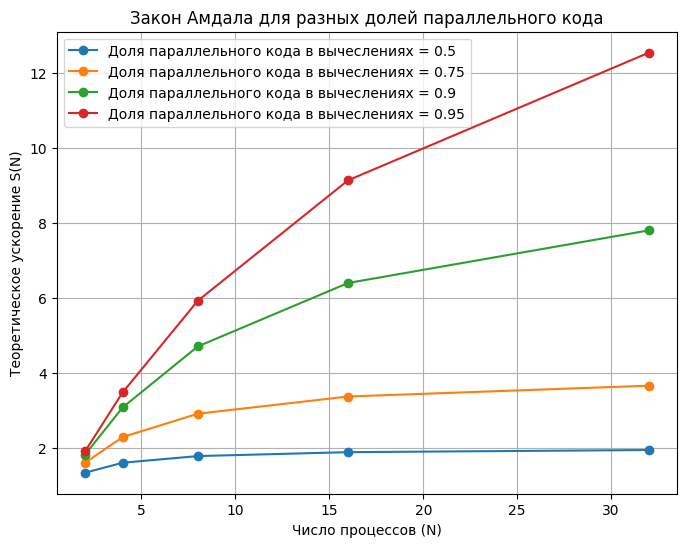


Следствие: Амдал указывает, что всякий раз, когда мы реализуем в какой-либо программе совместные и параллельные методы обработки, присущая ей последовательная часть накладных расходов всегда устанавливает некую верхнюю границу того, насколько можно разогнать эту программу. Именно это является одним из следствий, которые далее предлагает Закон Амдала.

## Закон Густавсона – Барсиса

## Практические приложения закона Амдала

Я честно хз лол, единственное что было

# Реализация параллельных алгоритмов в Python. Global Interpreter Lock (GIL). Готовые решения для многопоточных приложений в Python.

## Реализация параллельных алгоритмов в Python

**CPython** — популярная реализация интерпретатора Python — имеет **встроенный механизм**, который обеспечивает **выполнение ровно одного потока в любой момент времени**.

**GIL облегчает реализацию интерпретатора**, защищая объекты от одновременного доступа из нескольких потоков.  
Поэтому **создание нескольких потоков не приведёт к их одновременному исполнению на разных ядрах процессора**.

Однако, некоторые модули, как стандартные, так и сторонние, созданы для освобождения GIL при выполнении тяжелых вычислительных операций (например, сжатие или хеширование). К тому же, GIL всегда свободен при выполнении операций ввода-вывода.

Питон слывёт дружелюбным и простым в общении, но есть у него причуды. **Нельзя просто взять и воспользоваться всеми преимуществами многопоточности в Python!** Дорогу вам преградит огромный шлюз… Даже так — **глобальный шлюз** (*Global Interpreter Lock*, он же **GIL**), который ограничивает многопоточность на уровне интерпретатора.

## Global Interpreter Lock (GIL)

**Глобальный шлюз** (*Global Interpreter Lock*, он же **GIL**), технически, это один на всех mutex, созданный по умолчанию. Такого нет ни в C, ни в Java. Задача шлюза — пропускать потоки строго по одному, чтоб не летали наперегонки, как печально известные стритрейсеры, и не создавали угрозу работе интерпретатора.

Без GIL потоки **подрезали бы друг друга**, чтобы первыми добраться до памяти, но это еще не всё. Они имеют обыкновение **внезапно засыпать за рулём!** Операционная система не спрашивает, вовремя или невовремя — просто **усыпляет их в ей одной известный момент**.

Из-за этого **неупорядоченные потоки** могут неожиданно **перехватывать друг у друга инициативу** в работе с общими ресурсами.

**Дезориентированный спросонок поток**, который видит перед собой совсем не ту ситуацию, при которой засыпал, рискует **разбиться и повалить интерпретатор**, либо попасть в **тупиковую ситуацию (*deadlock*)**.  
Пример:  
Перед сном **Поток 1** начал работу со списком, а после пробуждения **не нашёл в этом списке элементов**, т.к. их **удалил или перезаписал Поток 2**..

Чтобы такого не было, **GIL в предсказуемый момент** (*по умолчанию раз в 5 миллисекунд для Python 3.2+*) командует отработавшему потоку:

«СПАААТЬ!»

Тот отключается и **не мешает проезжать следующему желающему**.  
Даже если желающего нет, **блокировщик всё равно подождёт**, прежде чем вернуться к предыдущему активному потоку.

✅ Благодаря GIL **однопоточные приложения работают быстро**, а потоки **не конфликтуют**.  
❌ **Но многопоточные программы при таком подходе выполняются медленнее** — слишком много времени уходит на **регулировку «дорожного движения»**.

А значит обработка графики, расчет математических моделей и поиск по большим массивам данных c GIL идут неприемлемо долго.

**(послесловие?) Глобальный шлюз** — наследие времён, когда программисты боролись за **достойную реализацию многозадачности**, и у них **не очень получалось**. Как объяснил **Гвидо ван Россум**:  
- **Без GIL** не будут нормально работать **C-расширения для Python**.  
- **Производительность однопоточных приложений упадёт**:  
- Python 3 станет **медленнее, чем Python 2**.  
- А **это никому не нужно**.

## Готовые решения для многопоточных приложений в Python

Библиотека multiprocessing позволяет организовать параллелизм вычислений за счет создания подпроцессов. Т.к. каждый процесс выполняется независимо от других, этот метод параллелизма позволяет избежать проблем с GIL.  
Предоставляемый библиотекой API схож с тем, что есть в threading, хотя есть уникальные вещи. Создание процесса происходит поутем создания объекта класса Process. Аргументы конструктора аналогичны тем, что есть в конструкторе Thread. В том числе аргумент daemon позволяет создавать служебные процессы. Служебные процессы завершаются вместе с родительским процессом и не могут порождать свои подпроцессы.

Для **сложных научных расчётов** в Python существует несколько библиотек, которые помогают обойти проблемы с производительностью из-за GIL:

* **Numba**: ускоряет вычисления с помощью JIT-компиляции.
* **NumPy**: используется для научных вычислений и эффективно работает с многомерными массивами.
* **SciPy**: библиотеки для научных и технических вычислений.

Эти библиотеки помогают значительно улучшить производительность, поскольку они **освобождают GIL** для тяжёлых вычислений.

# Алгоритмы сортировки. Асимптотический анализ сортировки пузырьком, поразрядной сортировки, быстрая сортировки. Ресурс параллелизации в алгоритме БС.

## Алгоритмы сортировки

Название темы, сортируем просто всякие массивы и всякое другое что сортируется ИДИ ДАЛЬШЕ КОРОЧЕ ГОВОРЯ

## Асимптотический анализ сортировки пузырьком, поразрядной сортировки, быстрая сортировки

**Сортировка пузырьком (Bubble Sort)**

* **Лучший случай**: O(n) (если массив уже отсортирован).
* **Средний случай**: O(n2) (много обменов из-за случайного порядка элементов).
* **Худший случай**: O(n2) (если массив отсортирован в обратном порядке).

Поразрядная сортировка:

* Все случаи: O(d\*n), d – макс. кол-во разрядов в числах

Быстрая сортировка:

* Лучший/средний случай: O(n\*logn)
* Худший случай: O(n2)

## Ресурс параллелизации в алгоритме БС

Разделение сортируемого чего-то на части, например массива на подмассивы, сортировка подмассивов и последующее их соединение

# Полный перебор. Методы оптимизации полного перебора. Алгоритмы поиска с возвратом.

## Полный перебор

Тема? А так ну come on, имя само за себя говорит, перебираем весь массив, оценка О(n), n – длина массива

## Методы оптимизации полного перебора

Оптимизация? Хз но единственное че в голову приходит например бинарный поиск если перебираем отсортированный массив

## Алгоритмы поиска с возвратом

Ключевой элемент **поиска с возвратом** — рекурсия, то есть функция, которая вызывает сама себя. Работа рекурсии часто требует много оперативной памяти, но из-за особенностей алгоритма памяти нужно гораздо меньше, чем при полном переборе «в лоб».

Алгоритм **поиска с возвратом** состоит из трёх частей:

* **Рекурсия**, которая получает очередную комбинацию для проверки и уходит вглубь себя, раскладывая всё по полочкам.
* **Генератор вариантов**, который создаёт новую цепочку данных и отправляет их в рекурсию.
* **Модуль проверки**, который оценивает, идёт ли рекурсия по верному пути, и если да — то нашла ли она решение в итоге или нет.

Основная идея перебора с возвратом (на примере судоку)

На каждом шаге выбирается число для очередной клетки. Если ни одно из чисел не подходит, алгоритм возвращается назад на ближайший шаг, где ещё есть альтернативные варианты. Таким образом, перебираются только возможные решения, исключая заведомо неверные.

# Граф. Виды графов. Представление графов в памяти. Примеры графовых задач. Граф как инструмент научного анализа.

## Граф

Граф - множество вершин и ребер. (подробнее) **Граф** – совокупность **точек**, соединенных **линиями**.  
- Точки называются **вершинами** (или **узлами**),  
- Линии – **ребрами** (или **дугами**).

## Виды графов

**Полный граф** – граф, содержащий ребра **между всеми парами вершин**.

**Взвешенный граф** – граф, в котором ребрам поставлено в соответствие конкретное **числовое значение**. Это значение называется **весом ребра**.

**Обыкновенный граф** — структура, где между двумя вершинами может быть **не более одного ребра**, и **петли отсутствуют**.  
Подходит для ситуаций, где важна **единственная связь** между элементами.

**Мультиграф** — допускает **кратные рёбра** и **петли**.  
Используется для моделирования систем с **разными видами связей** между одними и теми же вершинами.

**Граф с петлями** — разрешает наличие рёбер, соединяющих **вершину саму с собой**.  
Актуален, если важно учитывать **самоотношения**.

**Пустой граф** — содержит **только вершины**, **без рёбер**.  
Редко используется на практике, но важен в **теоретических исследованиях**.  
Применяется для анализа **крайних случаев** и как **отправная точка** для построения других графов.

**По связности:**

* **Связные графы** – между **любой парой вершин** существует **как минимум один путь**.
* **Несвязные графы** – существует **хотя бы одна вершина**, **не связанная** с другими.

**По направленности:**

* **Ориентированные графы** – ребра **направленные**,  
  переход возможен **только в одном направлении** между вершинами.
* **Неориентированные графы** – по ребрам возможен переход **в обоих направлениях**.
* **Смешанные графы** – содержат **как ориентированные, так и неориентированные ребра**.

**По заполненности**:

* Насыщенный граф – кол-во ребёр близко к максимальному
* Разреженный граф – кол-во ребёр значительно меньше максимального

## Представление графов в памяти

Граф может быть представлен (сохранён) **разными способами**:

* **Матрица смежности**
* **Матрица инцидентности**
* **Список смежности (инцидентности)**
* **Список рёбер**

**Матрица смежности** и **матрица инцидентности** — представляют граф в виде **двумерного массива (матрицы)**.  
Размер такой матрицы зависит от **количества вершин** и/или **рёбер** в графе. Эти методы удобны для **алгоритмов на матрицах** и при необходимости **быстрого доступа** к информации о связях между вершинами.

**Список рёбер** — это представление графа, в котором каждая строка описывает **две смежные вершины** и **вес рёбер** (для взвешенных графов).  
Количество строк в списке рёбер соответствует общему количеству рёбер в графе:  
- Для **ориентированных рёбер** — каждая пара рёбер считается отдельно.  
- Для **неориентированных рёбер** — каждое ребро учитывается дважды (с обеих сторон)

**Список смежности** используется, когда количество рёбер графа относительно небольшое, и большинство элементов матрицы смежности равны 0. В таких случаях матрица смежности становится неэффективной, а использование списка смежности является более оптимальным.

По отношению к памяти **список смежности** менее требователен, чем матрица смежности. Он представляется в виде таблицы с двумя столбцами:  
- **Первый столбец**: вершина, из которой выходят рёбра.  
- **Второй столбец**: список вершин, в которые входят рёбра из текущей вершины.

Для **взвешенных графов** список смежности усложняется, так как каждый элемент списка должен содержать два значащих поля:  
- Номер вершины, с которой соединяется текущая вершина.  
- Вес ребра.

**Преимущества списка смежности:**  
- Рациональное использование памяти.  
- Быстрый перебор соседей вершины.  
- Удобство проверки наличия рёбер и их удаления.

**Недостатки списка смежности:**  
- При работе с насыщенными графами (где много рёбер) скорость может быть недостаточной.  
- Нет быстрого способа проверить наличие ребра между двумя вершинами.  
- Требуется заранее знать количество вершин графа.

## Примеры графовых задач

Проверка наличия пути, задача коммивояжёра, поиск кратчайшего пути (дейкстра и а\*)

## Граф как инструмент научного анализа

Графовая модель — это не только способ описания физической или социальной структуры, но и мощный инструмент для анализа сложных систем и больших объёмов данных. Многие исследовательские задачи сводятся к построению графа и последующему анализу его свойств.

**Пример 1: Граф интернета**

**Пример 2: Граф цитирования научных публикаций**

# Деревья. Обход деревьев. Двоичная куча. Поиск в двоичном дереве. Идея индексирования данных.

## Деревья

**Дерево** — это связный ациклический граф. Связность означает наличие маршрута между любой парой вершин, ацикличность — отсутствие циклов. (подробнее далее)

* **Деревья** являются в некотором смысле простейшим классом **графов**. Для них выполняются многие интересные утверждения, которые не всегда выполняются для графов в общем случае. Применительно к деревьям многие доказательства и рассуждения оказываются намного проще. Выдвигая какие-то гипотезы при решении задач **теории графов**, целесообразно сначала их проверять на деревьях.
* **Деревья** являются самым распространённым классом **графов**, применяемых в **программировании**, причём в самых разных ситуациях. Более половины объёма этой главы посвящено рассмотрению конкретных применений деревьев в программировании.

## Обход деревьев

Самые простые в реализации обходы дерева — **прямой (Pre-Order)**, **обратный (Post-Order)** и **центрированный (In-Order)**

При **прямом обходе** мы посещаем родительские узлы до посещения узлов-потомков. В случае с нашим деревом мы будем обходить узлы в таком порядке: 1, 2, 4, 5, 3.

**Обратный обход** двоичного дерева — это когда вы сначала посещаете узлы-потомки, а затем — их родительские узлы. В нашем случае порядок посещения узлов при обратном обходе будет таким: 4, 5, 2, 3, 1.

При **центрированном обходе** мы посещаем все узлы слева направо. Центрированный обход нашего дерева — это посещение узлов 4, 2, 5, 1, 3.

Прямой обход (КЛП): корень → левое поддерево → правое поддерево

Центрированный обход (ЛКП): левое поддерево → корень → правое поддерево

Обратный обход (ЛПК): левое поддерево → правое поддерево → корень

## Двоичная куча

**Двоичная куча** (пирамида, сортирующее дерево) — структура данных, представляющая собой двоичное дерево, для которого выполнены три условия:

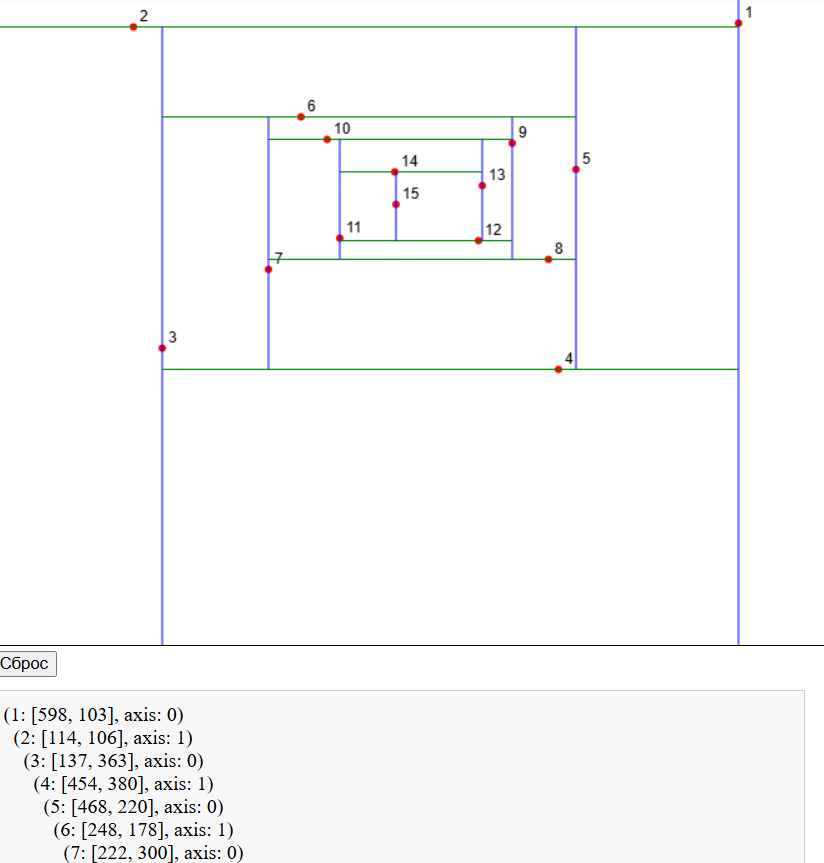
1. Значение в любой вершине не больше, чем значения её потомков.
2. У любой вершины не более двух сыновей.
3. Слои заполняются последовательно сверху вниз и слева направо, без «дырок».Поиск в двоичном дереве

Куча — это структура данных, которая поддерживает приоритеты и идеально подходит для задач, где необходимо часто извлекать минимальные (или максимальные) элементы. В нашем случае, мы всегда ищем вершину с минимальным расстоянием, что делает кучу хорошим выбором.

При применении релаксации (обновлении расстояний до соседних вершин), стандартная версия алгоритма требует перебора всех вершин, чтобы обновить минимальное расстояние, что также может занимать O(V) времени. С использованием кучи эта операция ограничена O(log V) для каждой вершины, благодаря чему алгоритм становится гораздо быстрее.

## Идея индексирования данных

Честно? Хз. Но как будто бы напрашивается kD-дерево (РЕЙКАСТИНГ!!! МОЙ СЛОНЯРА!!!!).

KD-дерево (K-мерное дерево) — структура данных, которая разделяет многомерное пространство на области, связанные с определёнными точками данных 

# Граф. Виды графов. Представление графов в памяти. Специализированные форматы хранения графов. Поиск в глубину. Поиск в ширину.

## Граф

Граф - множество вершин и ребер. (подробнее) **Граф** – совокупность **точек**, соединенных **линиями**.  
- Точки называются **вершинами** (или **узлами**),  
- Линии – **ребрами** (или **дугами**).

## Виды графов

**Полный граф** – граф, содержащий ребра **между всеми парами вершин**.

**Взвешенный граф** – граф, в котором ребрам поставлено в соответствие конкретное **числовое значение**. Это значение называется **весом ребра**.

**Обыкновенный граф** — структура, где между двумя вершинами может быть **не более одного ребра**, и **петли отсутствуют**.  
Подходит для ситуаций, где важна **единственная связь** между элементами.

**Мультиграф** — допускает **кратные рёбра** и **петли**.  
Используется для моделирования систем с **разными видами связей** между одними и теми же вершинами.

**Граф с петлями** — разрешает наличие рёбер, соединяющих **вершину саму с собой**.  
Актуален, если важно учитывать **самоотношения**.

**Пустой граф** — содержит **только вершины**, **без рёбер**.  
Редко используется на практике, но важен в **теоретических исследованиях**.  
Применяется для анализа **крайних случаев** и как **отправная точка** для построения других графов.

**По связности:**

* **Связные графы** – между **любой парой вершин** существует **как минимум один путь**.
* **Несвязные графы** – существует **хотя бы одна вершина**, **не связанная** с другими.

**По направленности:**

* **Ориентированные графы** – ребра **направленные**,  
  переход возможен **только в одном направлении** между вершинами.
* **Неориентированные графы** – по ребрам возможен переход **в обоих направлениях**.
* **Смешанные графы** – содержат **как ориентированные, так и неориентированные ребра**.

## Представление графов в памяти

Граф может быть представлен (сохранён) **разными способами**:

* **Матрица смежности**
* **Матрица инцидентности**
* **Список смежности (инцидентности)**
* **Список рёбер**

**Матрица смежности** и **матрица инцидентности** — представляют граф в виде **двумерного массива (матрицы)**.  
Размер такой матрицы зависит от **количества вершин** и/или **рёбер** в графе. Эти методы удобны для **алгоритмов на матрицах** и при необходимости **быстрого доступа** к информации о связях между вершинами.

**Список рёбер** — это представление графа, в котором каждая строка описывает **две смежные вершины** и **вес рёбер** (для взвешенных графов).  
Количество строк в списке рёбер соответствует общему количеству рёбер в графе:  
- Для **ориентированных рёбер** — каждая пара рёбер считается отдельно.  
- Для **неориентированных рёбер** — каждое ребро учитывается дважды (с обеих сторон)

**Список смежности** используется, когда количество рёбер графа относительно небольшое, и большинство элементов матрицы смежности равны 0. В таких случаях матрица смежности становится неэффективной, а использование списка смежности является более оптимальным.

По отношению к памяти **список смежности** менее требователен, чем матрица смежности. Он представляется в виде таблицы с двумя столбцами:  
- **Первый столбец**: вершина, из которой выходят рёбра.  
- **Второй столбец**: список вершин, в которые входят рёбра из текущей вершины.

Для **взвешенных графов** список смежности усложняется, так как каждый элемент списка должен содержать два значащих поля:  
- Номер вершины, с которой соединяется текущая вершина.  
- Вес ребра.

**Преимущества списка смежности:**  
- Рациональное использование памяти.  
- Быстрый перебор соседей вершины.  
- Удобство проверки наличия рёбер и их удаления.

**Недостатки списка смежности:**  
- При работе с насыщенными графами (где много рёбер) скорость может быть недостаточной.  
- Нет быстрого способа проверить наличие ребра между двумя вершинами.  
- Требуется заранее знать количество вершин графа.

## Специализированные форматы хранения графов

(у д а ч и .)

При работе с большими графами в инженерных и научных приложениях важно обеспечить эффективное хранение и быстрый доступ к структуре графа. Одним из популярных специализированных форматов хранения является **формат CRS (Compressed Row Storage)** — разновидность списков смежных вершин, заимствованная из методов хранения разреженных матриц.

Пусть граф G = (V, E) содержит N вершин и M рёбер. В формате CRS используется два основных массива:

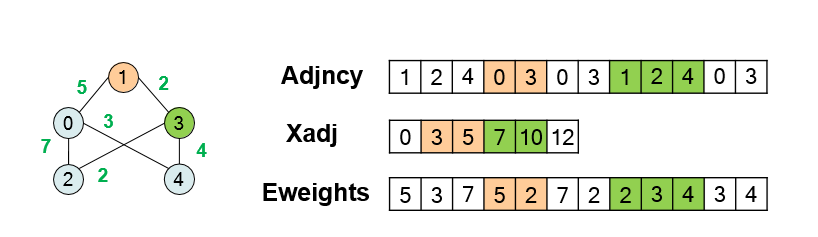
* **Adjncy** — массив смежных вершин, хранящий последовательность всех соседей для каждой вершины. Вершины перечисляются подряд: сначала соседи вершины 0, затем вершины 1 и так далее. Размер массива — 2 \* M, если граф ненаправленный.
* **Xadj** — массив индексов, указывающий, с какой позиции в Adjncy начинается список смежности очередной вершины. Для N вершин этот массив содержит N + 1 элемент. Последний элемент указывает на конец последнего списка.

Для получения всех соседей вершины i нужно обратиться к отрезку массива Adjncy от Xadj[i] до Xadj[i+1] - 1.

Такой способ экономит память и обеспечивает компактное хранение структуры графа, особенно эффективен для **разреженных графов**, где число рёбер примерно пропорционально числу вершин.

Если граф взвешенный, то дополнительно используется массив:

* **Eweights** — содержит веса рёбер в том же порядке, в каком рёбра перечислены в массиве Adjncy. Размер массива также 2 \* M (для ненаправленного графа).



При хранении большого графа на /P/ процессах удобно использовать **распределённый формат CRS (Compressed Row Storage)**. Он позволяет эффективно хранить и обрабатывать граф, распределяя вершины и рёбра между вычислительными узлами.

При хранении графа на /P/ процессах в формате **CRS (Compressed Row Storage)** используется следующая структура:

* **Adjncy** — массив всех соседей локальных вершин (аналогично обычному CRS).
* **Xadj** — массив индексов начала списков смежности в Adjncy для каждой локальной вершины.
* Нумерация вершин — с нуля (0-based indexing).

Дополнительно используется массив **VertexDist** размера /P + 1/, общий для всех процессов. Он описывает, какие вершины считаются *локальными* на каждом процессе:

* Для процесса /i/ локальными являются вершины с номерами от VertexDist[i] до VertexDist[i+1] – 1 включительно.
* Массив VertexDist хранится на **каждом** процессе.



## Поиск в глубину

**Обход в глубину (DFS)** — один из самых простых алгоритмов обхода графа. Входными параметрами для него являются граф и стартовая вершина. Алгоритм заключается в следующем:

1. Перебираем все рёбра, исходящие из стартовой вершины, и рекурсивно запускаем себя из каждой.
2. По окончании работы алгоритм обойдёт все вершины и рёбра, достижимые из стартовой вершины.
3. Ключевая деталь, делающая этот алгоритм быстрым, — пропуск уже посещённых вершин. Для этого вводится дополнительный массив из n булевых переменных, в которых хранится информация о том, посещал ли обход в глубину каждую вершину или нет.
4. Рекурсивные запуски будем производить только из тех вершин, которые ещё не помечены как посещённые.

Время работы алгоритма обхода в глубину зависит от способа представления графа. Однако важно отметить, что обход в глубину посещает каждую вершину не более одного раза, и при каждом посещении просматривает список исходящих рёбер только один раз.

## Поиск в ширину

Алгоритм обхода в ширину (англ. **breadth-first search**, или, сокращенно, **BFS**) действует таким образом, что он постепенно удаляется от стартовой вершины, двигаясь от неё по всевозможным направлениям.

**Поиск в ширину (BFS)** подразумевает поуровневое исследование графа:

1. Вначале посещается корень – произвольно выбранный узел.
2. Затем посещаются все потомки данного узла.
3. После этого посещаются потомки потомков и т.д.
4. Вершины просматриваются в порядке возрастания их расстояния от корня.
5. Алгоритм прекращает свою работу после обхода всех вершин графа или в случае выполнения требуемого условия (например, найти кратчайший путь из вершины 1 в вершину 6).

# Граф. Специализированные форматы хранения графов. Параллельный поиск в ширину.

## Граф

Граф - множество вершин и ребер. (подробнее) **Граф** – совокупность **точек**, соединенных **линиями**.  
- Точки называются **вершинами** (или **узлами**),  
- Линии – **ребрами** (или **дугами**).

## Специализированные форматы хранения графов

(у д а ч и .)

При работе с большими графами в инженерных и научных приложениях важно обеспечить эффективное хранение и быстрый доступ к структуре графа. Одним из популярных специализированных форматов хранения является **формат CRS (Compressed Row Storage)** — разновидность списков смежных вершин, заимствованная из методов хранения разреженных матриц.

Пусть граф G = (V, E) содержит N вершин и M рёбер. В формате CRS используется два основных массива:

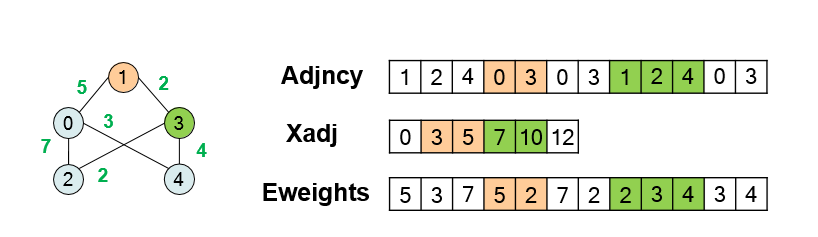
* **Adjncy** — массив смежных вершин, хранящий последовательность всех соседей для каждой вершины. Вершины перечисляются подряд: сначала соседи вершины 0, затем вершины 1 и так далее. Размер массива — 2 \* M, если граф ненаправленный.
* **Xadj** — массив индексов, указывающий, с какой позиции в Adjncy начинается список смежности очередной вершины. Для N вершин этот массив содержит N + 1 элемент. Последний элемент указывает на конец последнего списка.

Для получения всех соседей вершины i нужно обратиться к отрезку массива Adjncy от Xadj[i] до Xadj[i+1] - 1.

Такой способ экономит память и обеспечивает компактное хранение структуры графа, особенно эффективен для **разреженных графов**, где число рёбер примерно пропорционально числу вершин.

Если граф взвешенный, то дополнительно используется массив:

* **Eweights** — содержит веса рёбер в том же порядке, в каком рёбра перечислены в массиве Adjncy. Размер массива также 2 \* M (для ненаправленного графа).



При хранении большого графа на /P/ процессах удобно использовать **распределённый формат CRS (Compressed Row Storage)**. Он позволяет эффективно хранить и обрабатывать граф, распределяя вершины и рёбра между вычислительными узлами.

При хранении графа на /P/ процессах в формате **CRS (Compressed Row Storage)** используется следующая структура:

* **Adjncy** — массив всех соседей локальных вершин (аналогично обычному CRS).
* **Xadj** — массив индексов начала списков смежности в Adjncy для каждой локальной вершины.
* Нумерация вершин — с нуля (0-based indexing).

Дополнительно используется массив **VertexDist** размера /P + 1/, общий для всех процессов. Он описывает, какие вершины считаются *локальными* на каждом процессе:

* Для процесса /i/ локальными являются вершины с номерами от VertexDist[i] до VertexDist[i+1] – 1 включительно.
* Массив VertexDist хранится на **каждом** процессе.



## Параллельный поиск в ширину

В классическом алгоритме BFS вершины графа обрабатываются по уровням, где каждый уровень представляет собой набор вершин, находящихся на одинаковом расстоянии от начальной вершины.

1. **Начальный уровень**: Начинаем с вершины, которая является начальной.
2. **Следующий уровень**: После того как все вершины текущего уровня обработаны, переходим к следующему уровню, который состоит из всех соседей вершин текущего уровня, которые ещё не были посещены.

В **синхронизированном по уровням** подходе весь уровень вершин обрабатывается одновременно, и переход к следующему уровню возможен только после завершения обработки текущего. Важно отметить, что такая синхронизация обеспечивается, например, с использованием очереди, где каждый элемент хранит информацию о текущем уровне.

Предположим, у нас есть граф, и мы начинаем обход с вершины 0. Ожидаем следующий порядок обработки:

1. **Уровень 0**: Обрабатываем начальную вершину 0.
2. **Уровень 1**: После обработки уровня 0 переходим к соседям вершины 0 (вершины 1 и 2).
3. **Уровень 2**: После завершения уровня 1, обрабатываем соседей вершин 1 и 2 (например, вершина 3).

Этот процесс продолжается до тех пор, пока не будут обработаны все вершины

В параллельных реализациях BFS важное место занимает выбор подхода к обходу графа. Рассмотрим два подхода: **прямой обход** и **обратный обход**.

**Прямой обход**

В прямом обходе алгоритм начинает с начальной вершины и обрабатывает её соседей, а затем переходит к их соседям, продолжая процесс до тех пор, пока все вершины не будут посещены. Такой подход схож с традиционным алгоритмом BFS.

1. **Начальный уровень**: Начинаем с вершины 0.
2. **Обработка уровня**: На каждом уровне все вершины обрабатываются одновременно, в случае параллельной реализации — каждый поток отвечает за обработку части уровня.
3. **Переход к следующему уровню**: После завершения обработки уровня, переходим к соседям, помечая их и добавляя в очередь.

**Обратный обход**

В обратном обходе задача меняет направление: неактивные вершины пытаются найти активные вершины среди своих соседей. Этот подход позволяет инвертировать традиционный процесс и, возможно, улучшить производительность на определённых типах графов.

1. **Активизация вершин**: На каждом шаге неактивные вершины пытаются найти среди своих соседей активные вершины.
2. **Синхронизация**: Как и в прямом обходе, переход к следующему уровню возможен только после того, как все вершины текущего уровня найдены.

Наиболее очевидная возможность для параллелизации — это обход вершин и обработка их соседей, так как эти операции можно выполнить независимо для каждой вершины. Этот цикл можно распараллелить, потому что операции для каждой вершины не зависят от других вершин. Например, можно использовать параллельный цикл с помощью OpenMP

# Динамическое программирование. Задача о кратчайшем пути. Алгоритм Дейкстры. Ресурс оптимизации алгоритма Дейкстры.

## Динамическое программирование

**Динамическое программирование** (Dynamic programming) – метод решения задач (преимущественно оптимизационных) путем разбиения их на более простые подзадачи

Решение задачи идет от простых подзадач к сложным, периодически **используя ответы для уже решенных подзадач** (как правило, через рекуррентные соотношения)

Основная идея – запоминать решения встречающихся подзадач на случай, если та же подзадача встретится вновь

Чтобы успешно решить задачу динамикой нужно:

1) Состояние динамики: параметр(ы), однозначно задающие подзадачу.

2) Значения начальных состояний.

3) Переходы между состояниями: формула пересчёта.

4) Порядок пересчёта.

5) Положение ответа на задачу: иногда это сумма или, например, максимум из значений нескольких состояний.

Существует три порядка пересчёта:

1) Прямой порядок:

Состояния последовательно пересчитывается исходя из уже посчитанных.

2) Обратный порядок:

Обновляются все состояния, зависящие от текущего состояния

3) Ленивая динамика:

**Рекурсивная мемоизированная** функция пересчёта динамики. Это что-то вроде поиска в глубину по ацикличному графу состояний, где рёбра — это зависимости между ними.

## Задача о кратчайшем пути

**Определение.**  
*Кратчайшим путём* между вершинами *a* и *b* в неориентированном графе называется путь между ними, содержащий **наименьшее количество рёбер**.

В зависимости от контекста, под длиной пути могут понимать:

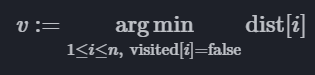
Кратчайший путь между парой вершин **может быть не единственным**.

## Алгоритм Дейкстры

Алгоритм Дейкстры можно представить как итеративный алгоритм.  
Начиная с неизвестных расстояний до всех остальных вершин, итерации этого алгоритма постепенно уточняют и улучшают эти расстояния, так что к моменту завершения алгоритма он приходит к искомому ответу.

Алгоритм Дейкстры применим только в том случае, когда веса всех ребер неотрицательны.

**Пошаговое описание алгоритма:**

1. Заведём массив текущих расстояний dist и массив отметок посещённых вершин visited.  
   Изначально массив dist заполнен значениями "бесконечность", кроме стартового элемента dist[s], равного нулю.  
   Массив отметок visited заполнен значениями false.
2. Изо всех не посещённых пока вершин выбирается вершина v с наименьшим значением dist: 
3. Выбранная вершина v отмечается посещённой:  
   visited[v] := true
4. Просматриваются все рёбра, исходящие из вершины v, и вдоль каждого ребра производится *релаксация* — попытка улучшить текущее расстояние до новой вершины.  
   Если конец текущего ребра обозначить как t, а его вес — как c, то выполняется: 
5. Если больше не осталось непосещённых вершин с конечным расстоянием, алгоритм завершает работу.  
   Иначе — переход к шагу 2.

## Ресурс оптимизации алгоритма Дейкстры

Скорее всего оптимизация в использование других форматов хранения вершин, а именно двоичной куче. Куча — это структура данных, которая поддерживает приоритеты и идеально подходит для задач, где необходимо часто извлекать минимальные (или максимальные) элементы. В нашем случае, мы всегда ищем вершину с минимальным расстоянием, что делает кучу хорошим выбором.

При применении релаксации (обновлении расстояний до соседних вершин), стандартная версия алгоритма требует перебора всех вершин, чтобы обновить минимальное расстояние, что также может занимать O(V) времени.

С использованием кучи эта операция ограничена O(log V) для каждой вершины, благодаря чему алгоритм становится гораздо быстрее.

# Динамическое программирование. Задача о кратчайшем пути. Алгоритм Дейкстры. Ресурс параллелизации алгоритма Дейкстры.

## Динамическое программирование

**Динамическое программирование** (Dynamic programming) – метод решения задач (преимущественно оптимизационных) путем разбиения их на более простые подзадачи

Решение задачи идет от простых подзадач к сложным, периодически **используя ответы для уже решенных подзадач** (как правило, через рекуррентные соотношения)

Основная идея – запоминать решения встречающихся подзадач на случай, если та же подзадача встретится вновь

Чтобы успешно решить задачу динамикой нужно:

1) Состояние динамики: параметр(ы), однозначно задающие подзадачу.

2) Значения начальных состояний.

3) Переходы между состояниями: формула пересчёта.

4) Порядок пересчёта.

5) Положение ответа на задачу: иногда это сумма или, например, максимум из значений нескольких состояний.

Существует три порядка пересчёта:

1) Прямой порядок:

Состояния последовательно пересчитывается исходя из уже посчитанных.

2) Обратный порядок:

Обновляются все состояния, зависящие от текущего состояния

3) Ленивая динамика:

**Рекурсивная мемоизированная** функция пересчёта динамики. Это что-то вроде поиска в глубину по ацикличному графу состояний, где рёбра — это зависимости между ними.

## Задача о кратчайшем пути

**Определение.**  
*Кратчайшим путём* между вершинами *a* и *b* в неориентированном графе называется путь между ними, содержащий **наименьшее количество рёбер**.

В зависимости от контекста, под длиной пути могут понимать:

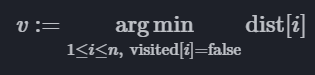
Кратчайший путь между парой вершин **может быть не единственным**.

## Алгоритм Дейкстры

Алгоритм Дейкстры можно представить как итеративный алгоритм.  
Начиная с неизвестных расстояний до всех остальных вершин, итерации этого алгоритма постепенно уточняют и улучшают эти расстояния, так что к моменту завершения алгоритма он приходит к искомому ответу.

Алгоритм Дейкстры применим только в том случае, когда веса всех ребер неотрицательны.

**Пошаговое описание алгоритма:**

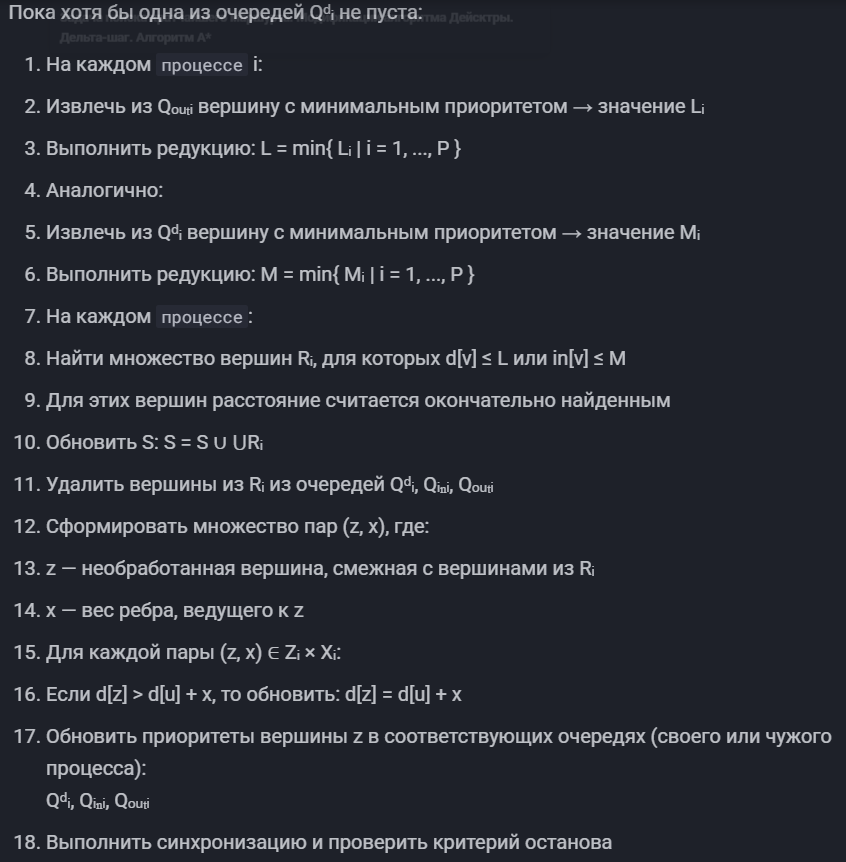
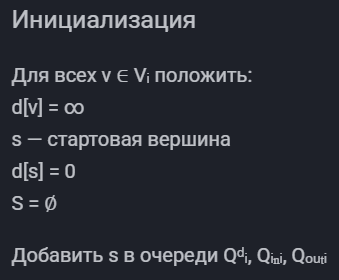
1. Заведём массив текущих расстояний dist и массив отметок посещённых вершин visited.  
   Изначально массив dist заполнен значениями "бесконечность", кроме стартового элемента dist[s], равного нулю.  
   Массив отметок visited заполнен значениями false.
2. Изо всех не посещённых пока вершин выбирается вершина v с наименьшим значением dist: 
3. Выбранная вершина v отмечается посещённой:  
   visited[v] := true
4. Просматриваются все рёбра, исходящие из вершины v, и вдоль каждого ребра производится *релаксация* — попытка улучшить текущее расстояние до новой вершины.  
   Если конец текущего ребра обозначить как t, а его вес — как c, то выполняется: 
5. Если больше не осталось непосещённых вершин с конечным расстоянием, алгоритм завершает работу.  
   Иначе — переход к шагу 2.

## Ресурс параллелизации алгоритма Дейкстры

Подход предполагает разделение графа на более мелкие подграфы и распараллеливание выполнения алгоритма Дейкстры на каждом подграфе с использованием подхода распараллеливания с общей памятью.

Основные компоненты алгоритма, которые можно распределить между процессами:

1. **Обработка вершин**
2. Распределение множества вершин V между процессами (V = V₁ ∪ V₂ ∪ ... ∪ Vₚ)
3. Локальные вычисления для "своих" вершин
4. **Работа с очередями**
5. Параллельное обслуживание трех типов очередей:
   * Q\_d (основные расстояния)
   * Q\_in (входящие ребра)
   * Q\_out (исходящие ребра)
6. **Обновление расстояний**
7. Независимая обработка смежных вершин
8. Пакетное обновление значений



# Динамическое программирование. Задача о кратчайшем пути. Алгоритм Дельта-шага. Ресурс парраллелизации алгоритма Дельта-шага.

## Динамическое программирование

**Динамическое программирование** (Dynamic programming) – метод решения задач (преимущественно оптимизационных) путем разбиения их на более простые подзадачи

Решение задачи идет от простых подзадач к сложным, периодически **используя ответы для уже решенных подзадач** (как правило, через рекуррентные соотношения)

Основная идея – запоминать решения встречающихся подзадач на случай, если та же подзадача встретится вновь

Чтобы успешно решить задачу динамикой нужно:

1) Состояние динамики: параметр(ы), однозначно задающие подзадачу.

2) Значения начальных состояний.

3) Переходы между состояниями: формула пересчёта.

4) Порядок пересчёта.

5) Положение ответа на задачу: иногда это сумма или, например, максимум из значений нескольких состояний.

Существует три порядка пересчёта:

1) Прямой порядок:

Состояния последовательно пересчитывается исходя из уже посчитанных.

2) Обратный порядок:

Обновляются все состояния, зависящие от текущего состояния

3) Ленивая динамика:

**Рекурсивная мемоизированная** функция пересчёта динамики. Это что-то вроде поиска в глубину по ацикличному графу состояний, где рёбра — это зависимости между ними.

## Задача о кратчайшем пути

**Определение.**  
*Кратчайшим путём* между вершинами *a* и *b* в неориентированном графе называется путь между ними, содержащий **наименьшее количество рёбер**.

В зависимости от контекста, под длиной пути могут понимать:

Кратчайший путь между парой вершин **может быть не единственным**.

## Алгоритм Дельта-шага

Алгоритм дельта-шаган предназначен для решения задачи поиска кратчайшего пути на графе. Для заданного ориентированного взвешенного графа с неотрицательными весами алгоритм находит кратчайшие расстояния от выделенной вершины-источника до всех остальных вершин графа. Алгоритм изначально проектировался с целью эффективной параллелизации. **Идея**: модифицировать алгоритм Дейкстры, сгруппировав вершины с близкими значениями расстояний, чтобы пересчитывать их одновременно.

* Вводится параметр Δ > 0 для разделения рёбер:
* **Лёгкие**: вес < Δ
* **Тяжёлые**: вес ≥ Δ
* Вершины группируются в "карманы" (buckets) по текущей оценке расстояния:
* 𝐵[𝑖] содержит вершины с 𝑑[𝑣] ∈ [𝑖Δ, (𝑖+1)Δ)

## Ресурс парраллелизации алгоритма Дельта-шага

Синхронизация: потоки (процессы) обмениваются подмножествами Req, чтобы каждый обрабатывал запросы только для своих локальных вершин. Дубли удаляются при формировании Req.  
Дополнительно синхронизируются:  
- переход к следующему непустому карману,  
- проверка критерия останова.

Предобработка (в параллель): для всех v∈V сформировать списки L[v] и H[v]

Инициализация (в параллель): для всех v∈V: d[v]=∞, B[0]={s}

Основной цикл:

1. На каждом потоке k: S[k]=∅
2. Пока карман B[i] не пуст:
3. Параллельно: для локальных вершин из B[i] построить Req[k] без дублей
4. S[k]=S[k]∪B[i], B[i]=∅
5. Обменяться Req[k], выполнить релаксацию локально
6. Параллельно: построить Req[k] из рёбер H[v], где v∈S[k]
7. Обменяться Req[k], выполнить релаксацию
8. Синхронизация: найти следующий непустой карман

# Жадные алгоритмы. Задача о рюкзаке. Пример жадной оптимизации (Алгоритм А\*).

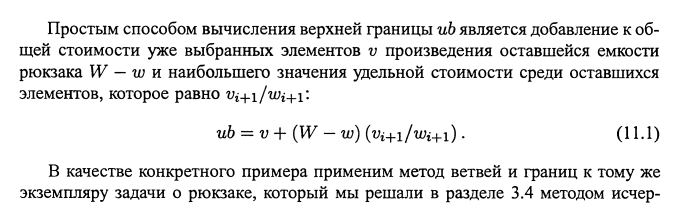
## Жадные алгоритмы

**Жадный алгоритм** (*Greedy algorithm*) — алгоритм, заключающийся в принятии локально оптимального решения на каждом его этапе, допуская, что конечное решение также окажется оптимальным. К примеру, алгоритм Дейкстры нахождения кратчайшего пути в графе вполне себе жадный, потому что мы на каждом шагу ищем вершину с наименьшим весом, в которой мы еще не бывали, после чего обновляем значения других вершин. При этом можно доказать, что кратчайшие пути, найденные в вершинах, являются оптимальными.

## Задача о рюкзаке

Дано n предметов весом w1, ... , Wn и ценой v1, ... , vn, а также рюкзак, выдерживающий вес W. Требуется найти подмножество предметов, которое можно разместить в рюкзаке, и которое имеет при этом максимальную стоимость

Применим метод ветвей и границ к решению задачи о рюкзаке. С этой задачей мы также познакомились в разделе: дано n предметов с весами w1, ... , Wn и ценами vi, ... , Vn, а также рюкзак, выдерживающий вес W. Требуется найти подмножество предметов, которое можно разместить в рюкзаке и которое имеет при этом максимальную цену. Оказывается, удобным упорядочить предметы в убывающем порядке по их удельной цене (отношению цены к весу), с разрешением неоднозначностей произвольным образом: 



## Пример жадной оптимизации (Алгоритм А\*)

A\* — это модификация алгоритма Дейкстры, оптимизированная для единственной конечной точки. Алгоритм Дейкстры может находить пути ко всем точкам, A\* находит путь к одной точке. Он отдаёт приоритет путям, которые ведут ближе к цели.

A\* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Как и все информированные алгоритмы поиска, он просматривает сначала те маршруты, которые «кажутся» ведущими к цели. Он использует **эвристическую функцию**, которая помогает выбрать наиболее перспективное направление поиска.

* g(n) — стоимость пути от начальной вершины до вершины n.
* h(n) — эвристическая оценка расстояния от n до цели.
* f(n) = g(n) + h(n) — общая оценка стоимости пути через вершину n.

Алгоритм выбирает вершины с наименьшим значением f(n).

# Задача коммивояжера. Разбор решений: метод грубой силы (полный перебор), динамическое программирование, жадный алгоритм.

## Задача коммивояжера

Суть задачи сводится к поиску **оптимального (кратчайшего, быстрейшего или самого дешевого) пути**, проходящего через промежуточные пункты по одному разу и возвращающегося в исходную точку. К примеру, нахождение наиболее выгодного маршрута, позволяющего **коммивояжеру посетить со своим товаром определенные города** по одному разу и вернуться обратно.

Мерой выгодности маршрута может быть **минимальное время поездки**, **минимальные расходы на дорогу** или **минимальная длина пути**. В наше время, когда стоимость доставки часто бывает сопоставима со стоимостью самого товара, а **скорость доставки** — один из главных приоритетов, задача нахождения **оптимального маршрута** приобретает огромное значение.

Задача коммивояжера является **NP-трудной** задачей, то есть её решение требует экспоненциального времени для поиска оптимального пути при увеличении количества городов.

## Разбор решений: 1) метод грубой силы (полный перебор);

Перебираются все возможные перестановки городов, выбирается маршрут с минимальным расстоянием.

Сложность:  
Полный перебор имеет факториальную сложность O(n!), так как рассматриваются все возможные маршруты.  
Это делает алгоритм непрактичным для больших n, например, уже при n=10 количество возможных маршрутов достигает 3,628,800.

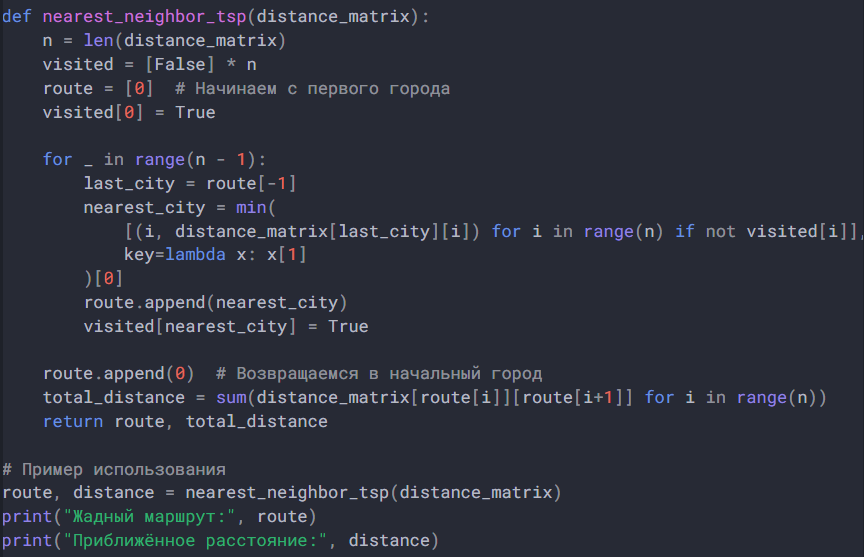
✅ Достоинства решения:  
- Гарантированно находит оптимальное решение.  
- Простая реализация.

❌ Недостатки решения:  
- Экспоненциальная сложность — не подходит для больших входных данных.  
- Требует хранения всех возможных маршрутов, что потребляет много памяти.

## 2) динамическое программирование;

Это и будет жадным алгоритмом?

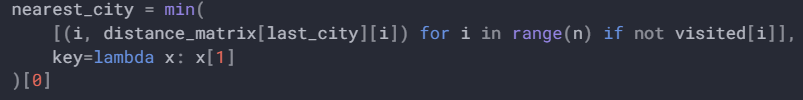
## 3) жадный алгоритм

Простые эвристики, такие как «ближайший сосед» (Nearest Neighbor), позволяют быстро находить приближённые решения, но не гарантируют их оптимальность. (решил код все таки привести)  


Разбор кода

Функция nearest\_neighbor\_tsp(distance\_matrix)  
Инициализация:  
visited = [False] \* n – массив отслеживает, какие города уже посещены.  
route = [0] – начинаем маршрут с первого города.  
visited[0] = True – помечаем его как посещённый.

Основной цикл (for \_ in range(n - 1)):  
Находим ближайший непосещённый город:



Добавляем его в маршрут и помечаем как посещённый.

**Завершение маршрута**:

* Возвращаемся в стартовый город route.append(0).
* Вычисляем общее расстояние маршрута.

**Возвращает**:

route – полученный маршрут.  
total\_distance – длина маршрута.  
Временная сложность: O(n^2), так как:

Каждую итерацию находим ближайший город (O(n))  
Делаем n-1 итераций → O(n^2).

✅ Достоинства решения:

* Быстрый алгоритм – работает за O(n^2), что намного лучше полного перебора O(n!) и ДП O(n^2 \* 2^n).
* Простая реализация.
* Подходит для больших n (до 1000+).

❌ Недостатки решения:  
- Не гарантирует оптимальный маршрут.  
- Эффект жадности: может привести к "локальному минимуму", когда хороший выбор на ранних этапах приводит к плохому конечному решению.  
- Зависит от стартового города: разные стартовые точки могут давать разные маршруты.

# Поиск с возвратом. Метод ветвей и границ. Задача Коммивояжера. Использование эвристик для оптимизации перебора (алгоритм Литтла).

## Поиск с возвратом

Ключевой элемент **поиска с возвратом** — рекурсия, то есть функция, которая вызывает сама себя. Работа рекурсии часто требует много оперативной памяти, но из-за особенностей алгоритма памяти нужно гораздо меньше, чем при полном переборе «в лоб».

Алгоритм **поиска с возвратом** состоит из трёх частей:

* **Рекурсия**, которая получает очередную комбинацию для проверки и уходит вглубь себя, раскладывая всё по полочкам.
* **Генератор вариантов**, который создаёт новую цепочку данных и отправляет их в рекурсию.
* **Модуль проверки**, который оценивает, идёт ли рекурсия по верному пути, и если да — то нашла ли она решение в итоге или нет.

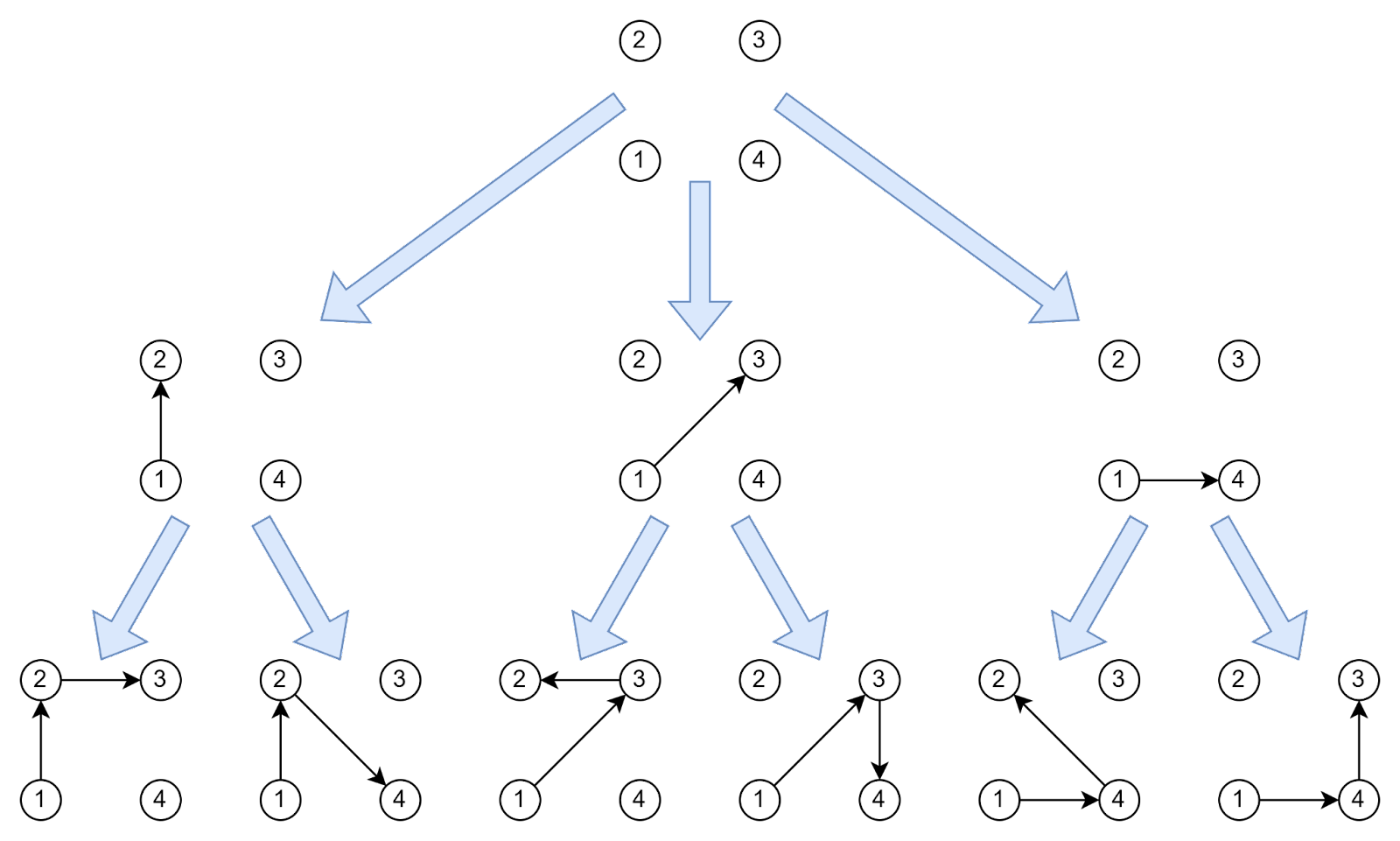
Основная идея перебора с возвратом (на примере судоку)

На каждом шаге выбирается число для очередной клетки. Если ни одно из чисел не подходит, алгоритм возвращается назад на ближайший шаг, где ещё есть альтернативные варианты. Таким образом, перебираются только возможные решения, исключая заведомо неверные.

## Метод ветвей и границ

Метод ветвей и границ концентрируется на том, чтобы отбрасывать заведомо плохие варианты.

Как и перебор, метод ветвей и границ гарантирует нахождение лучшего решения, но при этом он может найти его за приемлемое время. Но есть потенциальная проблема — в худшем случае он вырождается в метод перебора и может работать слишком долго.



**Как работает метод ветвей и границ**

Метод ветвей и границ довольно сложный. Он позволяет решать разные задачи на графах — задачу о рюкзаке, о коммивояжере и другие. Именно поэтому его описание достаточно абстрактное.

Кроме того, сам метод состоит из нескольких шагов. Их детальное описание может отличаться от задачи к задаче.

Рекурсивная функция tsp(node, visited, cost, path):

Базовый случай:  
- Если len(path) == n, значит обошли все города.  
- Добавляем расстояние до стартового города: cost += distance\_matrix[node][0].  
- Если новый маршрут лучше найденного, обновляем min\_distance и best\_route.

Рекурсивный случай:  
- Перебираем все возможные города next\_node, которые ещё не посещены.  
- Вызываем tsp() для следующего города с обновлённым cost и path.

Временная сложность:  
- В худшем случае: O(n!) (как полный перебор).  
- В среднем случае: значительно быстрее, так как отсекает ненужные ветви.  
- Использует нижнюю границу (bound), что сокращает количество перебираемых маршрутов.

Пространственная сложность:  
- O(n) – из-за хранения visited и path.

✅ Достоинства решения:  
- Находит оптимальное решение (если не прерывать алгоритм).  
- Лучше, чем полный перебор (O(n!)), так как отсекает плохие маршруты.  
- Не требует большого объёма памяти, как динамическое программирование (O(n \* 2^n)) или ACO.

❌ Недостатки решения:  
- Всё ещё экспоненциальная сложность в худшем случае.  
- Может работать долго при n > 20, если не удаётся отсечь много ветвей.  
- Зависит от порядка посещения городов – можно ускорить, если предварительно отсортировать рёбра.

## Задача коммивояжера

Суть задачи сводится к поиску **оптимального (кратчайшего, быстрейшего или самого дешевого) пути**, проходящего через промежуточные пункты по одному разу и возвращающегося в исходную точку. К примеру, нахождение наиболее выгодного маршрута, позволяющего **коммивояжеру посетить со своим товаром определенные города** по одному разу и вернуться обратно.

Мерой выгодности маршрута может быть **минимальное время поездки**, **минимальные расходы на дорогу** или **минимальная длина пути**. В наше время, когда стоимость доставки часто бывает сопоставима со стоимостью самого товара, а **скорость доставки** — один из главных приоритетов, задача нахождения **оптимального маршрута** приобретает огромное значение.

Задача коммивояжера является **NP-трудной** задачей, то есть её решение требует экспоненциального времени для поиска оптимального пути при увеличении количества городов.

## Использование эвристик для оптимизации перебора (алгоритм Литтла)

Алгоритм Литтла является частным случаем МВиГ (метод ветвей и границ), т.е. в худшем случае его сложность равна сложности полного перебора. Теоретическое описание выглядит следующим образом:

Имеется множество S всех гамильтоновых циклов графа. На каждом шаге в S ищется ребро (i, j), исключение которого из маршрута максимально увеличит оценку снизу. Далее происходит разбиение множества на два непересекающихся S1 и S2. S1 — все циклы, содержащие ребро (i, j) и не содержащие (j, i). S2 — все циклы, не содержащие (i, j).

Далее вычисляется оценка снизу для длины пути каждого множества и, если она превышает длину уже найденного решения, множество отбрасывается. Если нет — множества S1 и S2 обрабатываются так же, как и S.

(как работает далее, просто вспомни как делали с матрицами фигню с вычеркиваниями. Это оно и есть)

Имеется матрица расстояний M. Диагональ заполняется бесконечными значениями, т.к. не должно возникать преждевременных циклов. Также имеется переменная, хранящая нижнюю границу.

Стоит оговориться, что нужно вести учет двух видов бесконечностей — одна добавляется после удаления строки и столбца из матрицы, чтобы не возникало преждевременных циклов, другая — при отбрасывании ребер. Случаи будут рассмотрены чуть позже. Первую бесконечность обозначим как inf1, вторую — inf2. Диагональ заполнена inf1.

Алгоритм состоит из двух этапов:  
**Первый этап**  
Приведение матрицы затрат и вычисление нижней оценки стоимости маршрута r.

1. Вычисляем наименьший элемент в каждой строке (константа приведения для строки)
2. Переходим к новой матрице затрат, вычитая из каждой строки ее константу приведения
3. Вычисляем наименьший элемент в каждом столбце (константа приведения для столбца)
4. Переходим к новой матрице затрат, вычитая из каждого столбца его константу приведения.  
   Как результат имеем матрицу затрат, в которой в каждой строчке и в каждом столбце имеется хотя бы один нулевой элемент.
5. Вычисляем границу на данном этапе как сумму констант приведения для столбцов и строк (данная граница будет являться стоимостью, меньше которой невозможно построить искомый маршрут)

**Второй этап**  
1 .Вычисление штрафа за неиспользование для каждого нулевого элемента приведенной матрицы затрат.  
Штраф за неиспользование элемента с индексом (h,k) в матрице, означает, что это ребро не включается в наш маршрут, а значит минимальная стоимость «неиспользования» этого ребра равна сумме минимальных элементов в строке h и столбце k.

а) Ищем все нулевые элементы в приведенной матрице  
б) Для каждого из них считаем его штраф за неиспользование.  
в) Выбираем элемент, которому соответствует максимальный штраф (любой, если их несколько)

2 . Теперь наше множество S разбиваем на множества — содержащие ребро с максимальным штрафом(Sw) и не содержащие это ребро(Sw/o).  
3 . Вычисление оценок затрат для маршрутов, входящих в каждое из этих множеств.  
а) Для множества Sw/o все просто: раз мы не берем соответствующее ребро c максимальным штрафом(h,k), то для него оценка затрат равна оценки затрат множества S + штраф за неиспользование ребра (h,k)  
б) При вычислении затрат для множества Sw примем во внимание, что раз ребро (h,k) входит в маршрут, то значит ребро (k,h) в маршрут входить не может, поэтому в матрице затрат пишем c(k,h)=infinity, а так как из пункта h мы «уже ушли», а в пункт k мы «уже пришли», то ни одно ребро, выходящее из h, и ни одно ребро, приходящее в k, уже использоваться не могут, поэтому вычеркиваем из матрицы затрат строку h и столбец k. После этого приводим матрицу, и тогда оценка затрат для Sw равна сумме оценки затрат для S и r(h,k), где r(h,k) — сумма констант приведения для измененной матрицы затрат.  
4 . Из всех неразбитых множеств выбирается то, которое имеет наименьшую оценку.

Так продолжаем, пока в матрице затрат не останется одна не вычеркнутая строка и один не вычеркнутый столбец.

# Эвристические алгоритмы. Алгоритм имитации отжига. Примеры использования алгоритма имитации отжига.

## Эвристические алгоритмы

**Эвристический алгоритм** — это алгоритм решения задачи, правильность которого для всех возможных случаев не доказана, но про который известно, что он даёт достаточно хорошее решение в большинстве случаев.  
Эвристические алгоритмы обычно применяются при решении плохо формализованных или сложных задач.

## Алгоритм имитации отжига

**Алгоритм имитации отжига** (англ. *simulated annealing*) — эвристический алгоритм глобальной оптимизации, особенно эффективный при решении дискретных и комбинаторных задач. Алгоритм вероятностный и не даёт почти никаких гарантий сходимости, однако хорошо работает на практике при решении NP-полных задач. Иногда на контестах им удаётся сдать сложные комбинаторные задачи, у которых есть нормальное решение.

## Примеры использования алгоритма имитации отжига

Задача коммивояжёра: было видно, как сначала города на рандом берутся, но потихоньку расстояние между выбираемыми городами становилось короче до тех пор, пока не достигаем нулевой температуры.

Еще раз повторим суть процесса отжига:

1. Генерируем начальное случайное решение (или получаем **feasible** при помощи эвристик);
2. Задаем начальную **“температуру”** – некий глобальный метапараметр, суть которого станет ясна далее;
3. Выполняем отжиг заданное число итераций;
4. Выполняем случайную модификацию решения;
5. Если значение функции стоимости для нового решения лучше, чем для старого, то принимаем его;
6. Если нет, то все равно можем принять новое решение, но лишь с некоторой вероятностью, которая тем больше, чем выше температура и чем ближе друг к другу по оценке старое и новое решение.

# Эвристические алгоритмы. Генетический алгоритм. Примеры использования алгоритма генетического алгоритма.

## Эвристические алгоритмы

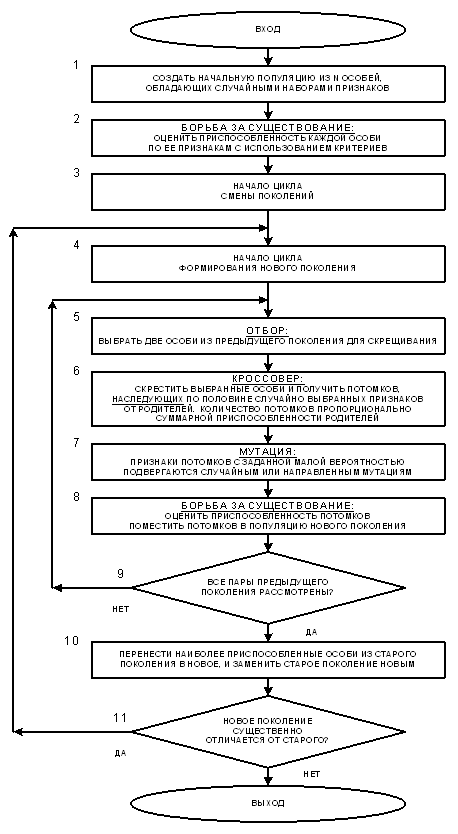
**Эвристический алгоритм** — это алгоритм решения задачи, правильность которого для всех возможных случаев не доказана, но про который известно, что он даёт достаточно хорошее решение в большинстве случаев.  
Эвристические алгоритмы обычно применяются при решении плохо формализованных или сложных задач.

## Генетический алгоритм

**Генетический алгоритм** (англ. genetic algorithm) — эвристический алгоритм поиска, используемый для решения задач оптимизации и моделирования путём случайного подбора, комбинирования и вариации искомых параметров с использованием механизмов, аналогичных естественному отбору в природе. Несмотря на то, что **биологи пока не пришли к единой системе критериев**, позволяющей оценивать значимость процессов, воспроизводимых в ГА, такие алгоритмы всё же представляют собой **важный инструмент концептуального анализа** и экспериментальной проверки эволюционных гипотез.

Работа **генетического алгоритма (ГА)** представляет собой итерационный процесс, который продолжается до тех пор, пока поколения не перестанут существенно отличаться друг от друга, или не пройдет заданное количество поколений, или заданное время. Для каждого поколения реализуются **отбор**, **кроссовер** (скрещивание) и **мутация**.

На примере символов (1234, abcd):  
Популяция – (1234, abcd)  
Хромосома – (1234)  
Ген – 1



## Примеры использования алгоритма генетического алгоритма

Подбор слов? Как получали из символов/цифр “Hello world”. Как будто бы это в ИИ используется

Был ещё коммивояжёр где-то, как именно хз. Но пускай лол

# Эвристические алгоритмы. Муравьиный алгоритм. Примеры использования алгоритма муравьиного алгоритма.

## Эвристические алгоритмы

**Эвристический алгоритм** — это алгоритм решения задачи, правильность которого для всех возможных случаев не доказана, но про который известно, что он даёт достаточно хорошее решение в большинстве случаев.  
Эвристические алгоритмы обычно применяются при решении плохо формализованных или сложных задач.

## Муравьиный алгоритм

Муравьиный алгоритм (Ant Colony Optimization, ACO) — это метаэвристический метод для решения задач оптимизации, вдохновленный поведением муравьев в поиске пищи. Алгоритм был разработан в 1992 году Марко Дориги и его коллегами, и с тех пор широко применяется для решения различных задач, таких как **задача коммивояжера** (TSP), **расписания**, **оптимизация маршрутов** и др.

Основные идеи

1. **Поведение муравьев**: Муравьи находит кратчайший путь к пище, следуя за феромонами, которые они оставляют на пути.
2. **Феромоны**: Муравьи оставляют следы феромонов на пройденном пути. Эти следы испаряются со временем, но феромоны остаются на пути, по которому прошел муравей.
3. **Алгоритм**: Исключает полное переборное решение. Вместо этого множество агентов (муравьев) исследуют пространство решений параллельно, используя феромоны для принятия решений.
4. **Использование феромонов**: Пути, по которым проходит больше муравьев, становятся более привлекательными для других муравьев, так как оставшиеся феромоны усиливаются.

Шаги работы алгоритма

1. **Инициализация**: Инициализация феромонов на всех путях с равными значениями.
2. **Построение решения**: Каждый муравей строит решение, выбирая следующий шаг на основе вероятности, которая зависит от количества феромонов и длины пути.
3. **Обновление феромонов**:
   * Феромоны испаряются со временем (уменьшаются).
   * Добавляются новые феромоны на пути, пройденные муравьями. Чем лучше путь, тем больше феромонов добавляется.
4. **Повторение**: Алгоритм повторяется, пока не будет найдено оптимальное решение или не исчерпано количество итераций.

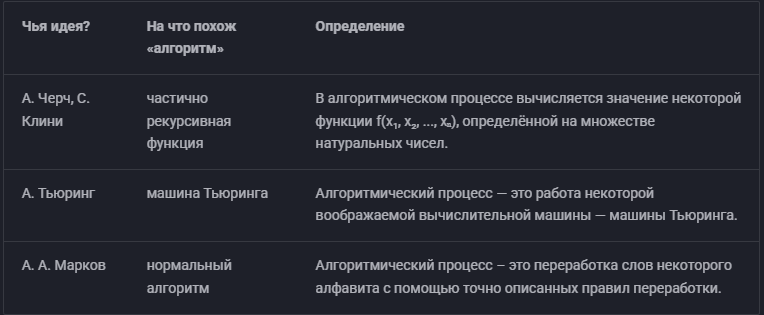
Параметры алгоритма

1. **Количество муравьев**: Количество агентов, которые будут искать решение.
2. **Испарение феромонов**: Коэффициент, определяющий скорость, с которой феромоны исчезают со времени.
3. **Параметры α и β**: Они контролируют влияние феромонов (α) и расстояния (β) на выбор пути.
4. **Число итераций**: Количество циклов, в которых муравьи будут искать решение.

## Примеры использования алгоритма муравьиного алгоритма

Муравьиный алгоритм применим для различных задач:  
- **Задача коммивояжера** (TSP): Поиск кратчайшего пути, который посещает все города.  
- **Оптимизация маршрута**: Для логистики и доставки товаров.  
- **Распределение задач**: Например, при решении задачи о распорядке работы оборудования.

# Формализация понятия алгоритма. Лямбда-исчисление Черча. Тезис Черча. Алгоритмически неразрешимые задачи.

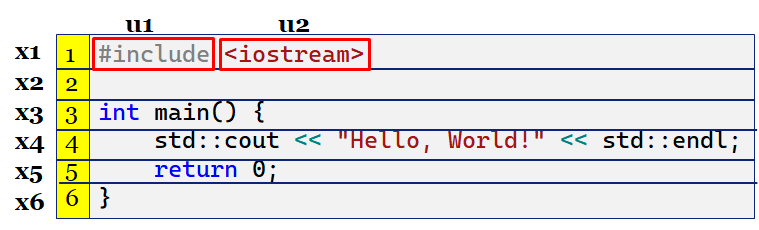


## lambda-исчисление Черча

Вариант строгого определения алгоритма, предложенный Черчем и Клини, рассматривает алгоритмический процесс как процесс вычисления значений некоторой функции f, определённой на множестве натуральных чисел N.

Предположим, что имеется некоторый алгоритм, который работает следующим образом:

* На вход алгоритма поступает объект P, который представляет собой набор данных u1,u2,…,un, где каждый ui является словом в некотором алфавите A.
* Слова в алфавите A образуют счётное множество, и эти слова могут быть занумерованы натуральными числами.
* Таким образом, каждый конструктивный объект ui имеет номер x∈N, где x — это натуральное число, которое используется для идентификации объекта.



В данном контексте предполагается, что алгоритм выполняет вычисления, результатом которых является некоторое значение функции f, определённой на множестве натуральных чисел N. Функция f представляет собой отображение, которое, в общем случае, может быть определено не для всех входных данных.

Когда имеется некоторый объект P на входе алгоритма, это вовсе не означает, что алгоритм всегда приводит к некоторому объекту Q на выходе. В некоторых случаях алгоритм может не иметь результата, что делает вычисление функции f **неопределённым** для определённых наборов данных.  
В связи с этим возникает понятие **частичной функции**. Частичная функция — это такая функция, которая может быть определена только для подмножества входных данных. Например, алгоритм может правильно обработать один набор входных данных и не дать результата для другого.

Частичная функция играет ключевую роль в контексте алгоритмов, поскольку:

* Алгоритм не всегда имеет гарантированный результат для всех возможных входных данных.
* Для некоторых наборов данных алгоритм может не завершиться или не выдать корректного результата.

Таким образом, несмотря на то, что алгоритм представляет собой конечную последовательность операций, его поведение может зависеть от конкретных входных данных, что и ведёт к необходимости использования частичных функций.

## Тезис Черча

**Тезис Черча**

***Каждая вычислимая функция частично рекурсивна.***

Сможем ли мы доказать тезис Черча? Нет, поскольку у нас нет точного определения вычислимой функции. Тезис Черча — это просто строгое определение алгоритма. Это положение является соглашением, возникшим в результате длительного исследования интуитивного понятия алгоритма. Тезис Черча является естественнонаучным фактом, который подтверждается опытом, накопленным математикой за весь период её развития. Тезис Черча является достаточным средством, чтобы придать необходимую точность формулировкам алгоритмических проблем и делает возможным доказательство их неразрешимости.

## Алгоритмически неразрешимые задачи

Теорема Геделя о неполноте

Теорема Гёделя, сформулированная Куртом Гёделем в 1931 году, устанавливает важные пределы в математической логике и вычислениях, что связано с понятием разрешимости.

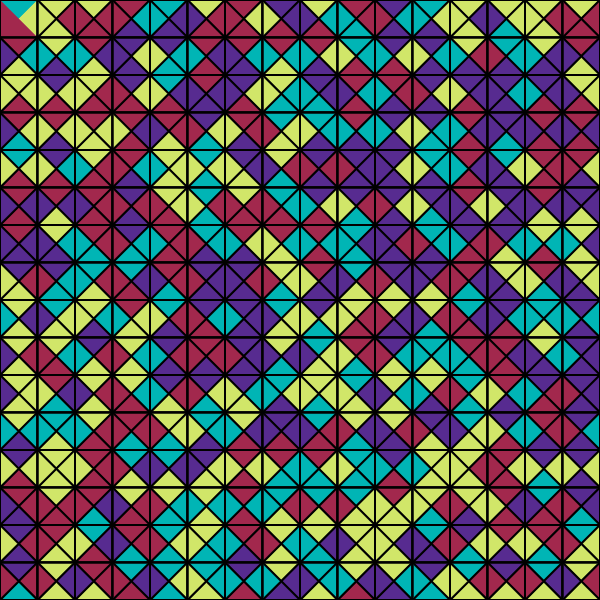
**Формулировка Теоремы Гёделя о неполноте**

1. **Первая теорема о неполноте**: В любой достаточно мощной формальной системе, которая включает арифметику натуральных чисел, существуют утверждения, которые не могут быть доказаны или опровергнуты в рамках этой системы. То есть система не является полной.
2. **Вторая теорема о неполноте**: Если система непротиворечива, то она не может доказать свою собственную непротиворечивость. Это означает, что невозможно внутри формальной системы доказать, что система сама по себе не содержит противоречий.

Проблема замощения плитки Вана

Плитки Вана (или домино Вана), впервые предложенные математиком, логиком и философом Хао Ваном в 1961, — это класс формальных систем. Они моделируются визуально с помощью квадратных плиток с раскрашиванием каждой стороны. Определяется набор таких плиток (например, как на иллюстрации), затем копии этих плиток прикладываются друг к другу с условием согласования цветов сторон, но без вращения или симметрического отражения плиток.

Основной вопрос о наборе плиток Вана — могут ли они замостить плоскость или нет, то есть может ли плоскость быть заполнена таким способом. Следующий вопрос, может ли она быть заполнена в виде периодического узора.



Задача усердного бобра

Задача усердного бобра (или задача о бобре) (англ. busy beaver) — это классическая неразрешимая задача в теории вычислений, которая связана с вычислением максимального числа шагов, которое может выполнить машина Тьюринга до того, как она остановится, при условии, что эта машина начинает в пустом состоянии и на входе нет данных. Задача состоит в том, чтобы найти максимальное количество шагов, которые может сделать машина Тьюринга с заданным количеством состояний, прежде чем она остановится. Этот максимальный результат и называется числом усердного бобра для заданного числа состояний.

# Основы параллелизации. Средства синхронизации и их реализация. Атомарные переменные. Мьютексы. Семафоры.

## Основы параллелизации

??? просто название темы ???

## Средства синхронизации и их реализация

Всё что ниже

## Атомарные переменные

Атомарные переменные позволяют выполнять операции (чтение, запись, модификация) над переменными (в случае C++ над объектами) без использования традиционных средств синхронизации (например мьютексов) в многопоточных приложениях. Для достижения такого результата используются аппаратные инструкции (lock-free). Операции над атомарной переменной могут выполняться без блокировок на аппаратном уровне. Проверить отсутствие блокировок можно с помощью метода **is\_lock\_free**. Также есть переменная **is\_always\_lock\_free**, определяемая на этапе компиляции, но принимает true, если отсутствие блокировок не зависит от аппаратного уровня.

Есть три метода для сравнения и обмена:

* **exchange** — атомарный способ значения атомарной переменной новым значением, возвращая предыдущее значение.
* **compare\_exchange\_weak** — атомарно сравнивает значение с ожидаемым и, если они равны, устанавливает новое значение, возвращая флаг успешности замены.
* **compare\_exchange\_strong** — аналогично **compare\_exchange\_weak**, но с гарантией отсутствия ложных отказов, возвращая флаг успешности замены.

## Мьютексы

**Мьютексы («mutual exclusion» — взаимное исключение)** используются для обеспечения взаимного исключения, что позволяет предотвратить одновременный доступ к общим ресурсам в многопоточных приложениях. Они представляют собой блокировку, которая может находиться в одном из двух состояний: заблокированном и разблокированном.

Мьютексы предоставляют механизм блокировки, который позволяет потокам последовательно получать доступ к общему ресурсу. Когда поток захватывает мьютекс, другие потоки не могут получить доступ к защищенному им ресурсу до тех пор, пока первый поток не освободит мьютекс.

В C++ есть несколько типов мьютексов:  
- **std::mutex** — самый простой мьютекс, который блокирует доступ к ресурсу и не позволяет одному потоку захватывать его несколько раз.  
- **std::recursive\_mutex** — позволяет одному потоку захватывать мьютекс несколько раз, предотвращая возникновение взаимных блокировок.  
- **std::timed\_mutex** и **std::recursive\_timed\_mutex** — позволяют пытаться захватить мьютекс в течение ограниченного времени, предотвращая бесконечное ожидание блокировки.

## Семафоры

**Семафор** — примитив синхронизации работы процессов и потоков, в основе которого лежит счётчик, над которым можно производить две атомарные операции: увеличение и уменьшение значения на единицу.

**Основные методы семафоров:**

* acquire — захватывает семафор, уменьшая счетчик (если счетчик равен нулю — поток блокируется);
* release — освобождает семафор, увеличивая счетчик (если есть заблокированный поток — он просыпается);
* try\_acquire — неблокирующая попытка захвата счетчика, возвращает **true** в случае успешного захвата.

**В отличие от мьютексов** семафоры не привязаны к потокам выполнения — acquire и release могут быть вызваны в разных потоках.

Начальное значение передается параметром в конструкторе.