Лекция 1. Введение. Асимптотический анализ (O-нотация). Сортировки.

1. Время работы и модель вычислений

Когда мы говорим об эффективности алгоритма, мы в первую очередь имеем в виду два ресурса: **время** и **память**. В этом курсе мы прежде всего сосредоточимся на анализе **времени работы**.

Чтобы корректно сравнивать алгоритмы, нам нужна абстрактная модель вычислительной машины. Исторически существовало множество моделей таких, как машина Тьюринга, алгоритмы Маркова, но сегодня де-факто стандартом является **RAM-модель (Random Access Machine)**.

RAM-модель представляет собой вычислительную машину со следующими свойствами:

1. **Одна процессорная единица**: инструкции выполняются последовательно, одна за другой.
2. **Бесконечная память с произвольным доступом**: каждая ячейка памяти имеет уникальный адрес, и время доступа к любой ячейке (чтение или запись) является **константным** и не зависит от адреса или размера данных. Это ключевое допущение упрощает анализ.
3. **Базовые операции**: каждая простая инструкция (арифметические операции $+$, $-$, $*$, $/$, сравнение $<$, $>$, $==$, присваивание $=$, вызов функции, доступ к элементу массива по индексу) выполняется за **константное время**, обычно обозначаемое как **O(1)**.

На практике, разумеется, память не бесконечна, а время доступа к кешу, оперативной памяти и диску различается на порядки. Однако для анализа алгоритмов высокого уровня и их сравнения RAM-модель предоставляет прекрасный баланс между простотой и точностью.

2. Анализ времени работы на примере поиска минимума

Рассмотрим простейшую задачу: найти минимальный элемент в непустом массиве.

```python
def find_min(arr: list[int]) -> int:
    min_val = arr[0]          # O(1): присваивание
    for num in arr[1:]:       # Цикл выполнится (n-1) раз
        if num < min_val:     # O(1): сравнение
            min_val = num     # O(1): присваивание (в худшем случае)
    return min_val            # O(1): возврат значения
```

Давайте определим **время работы** этой функции $T(n)$, где $n = len(arr)$ – размер входных данных.

Инициализация min_val: **1 операция**.

Цикл for: итерируется $n - 1$ раз.

В каждой итерации:

1. Сравнение $num < min\_val$: **1 операция**.
2. Присваивание $min\_val = num$: эта операция выполняется только если условие истинно. В **худшем случае** (массив отсортирован по убыванию)

оно выполняется на **каждой** итерации. Мы часто анализируем именно **худший случай**, так как он даёт нам гарантию на время работы алгоритма для любого входа.

Возврат результата: **1 операция**.

Сложим всё вместе для худшего случая:

$$T(n) = 1 + (n-1) \cdot (1+1) + 1 = 2 + 2 \cdot (n-1) = 2n$$

Мы получили линейную функцию $T(n) = 2n$. Но насколько полезна такая точность? На реальной машине каждая «операция» из нашей модели может выполняться за разное количество тактов процессора. Поэтому нас интересует не точное число, а порядок роста функции $T(n)$ при увеличении n (при стремлении n к бесконечности, $n \to \infty$). Именно для описания порядка роста и была придумана асимптотическая нотация.

3. O-нотация (O-большое)

Определение. Говорят, что функция $f(n) = O\big(g(n)\big)$ (читается «ф от эн есть О большое от жэ от эн»), если существуют такие положительные константы c и n_0, что для всех $n \geq n_0$ выполняется неравенство:

$$0 \leq f(n) \leq c \cdot g(n)$$

На языке кванторов:

$$\exists c > 0, \exists n_0 \in \mathbb{N}: \forall n \geq n_0 \Rightarrow f(n) \leq c \cdot g(n)$$

Эта запись означает, что функция $g(n)$ является **асимптотической верхней оценкой** для $f(n)$. $f(n)$ растёт не быстрее, чем $g(n)$ с точностью до постоянного множителя c для всех достаточно больших n.

Доказательство на примере. Докажем, что $f(n) = 5n + 2 = O(n)$.

Нам нужно подобрать c и n_0 так, чтобы $5n + 2 \leq c \cdot n$ для всех $n \geq n_0$.

Преобразуем неравенство: $5n + 2 \leq c \cdot n \Rightarrow 2 \leq n \cdot (c-5)$.

Если мы возьмём, например, $c = 6$, тогда неравенство примет вид $2 \leq n \cdot (6-5) \Rightarrow 2 \leq n$. Это неравенство верно для всех $n \geq 2$.

Таким образом, мы нашли $c = 6$ и $n_0 = 2$, при которых определение выполняется.

Можно было взять $c = 7$, тогда $2 \leq n \cdot (7-5) \Rightarrow 2 \leq 2n \Rightarrow n \geq 1$. Так что $n_0 = 1$ тоже подходит.

Важно. В информатике запись $O(n)$ подразумевает именно $O\big(g(n)\big)$, где $g(n) = n$. Это стандартное сокращение.

Для нашей функции поиска минимума мы можем сказать: $T(n) = 2n = O(n)$. Мы отбросили константный множитель 2, так как он не важен для асимптотики.

4. Другие асимптотические нотации: Ω и Θ

Ω-нотация (Омега-большое) является **асимптотической нижней оценкой**.

Определение. $f(n) = \Omega\big(g(n)\big)$, если

$$\exists c > 0, \exists n_0 \in \mathbb{N}: \forall n \geq n_0 \Rightarrow 0 \leq c \cdot g(n) \leq f(n)$$

Это значит, что $f(n)$ растёт **не медленнее**, чем $g(n)$. Например, для поиска минимума в неотсортированном массиве любой корректный алгоритм должен посмотреть каждый элемент как минимум один раз, чтобы не пропустить минимум. Поэтому время работы любого алгоритма для этой задачи $\Omega(n)$. Мы можем сказать, что $find\_min(arr)$ работает за $\Theta(n)$. А вот если бы массив был отсортирован, мы могли бы взять первый элемент за O(1), и нижняя оценка была бы другой.

Θ-нотация (Тета) описывает **асимптотически точную оценку**.

Определение. $f(n) = \Theta\big(g(n)\big)$, если

$$\exists c_1 > 0, \exists c_2 > 0, \exists n_0 \in \mathbb{N}: \forall n \geq n_0 \Rightarrow 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Это означает, что функции $f(n)$ и $g(n)$ имеют **одинаковый порядок роста**. Фактически, $f(n) = \Theta\big(g(n)\big)$ тогда и только тогда, когда $f(n) = O\big(g(n)\big)$ и $f(n) = \Omega\big(g(n)\big)$.

Смысл на практике:

1. $O(\dots)$: Алгоритм работает **не дольше, чем …**. Это наша главная гарантия.
2. $\Omega(\dots)$: Алгоритм работает **не быстрее, чем …**. Это важно для понимания пределов оптимизации.
3. $\Theta(\dots)$: Алгоритм работает **примерно как …**. Идеальная ситуация, когда верхняя и нижняя оценки совпали.

5. Сортировка вставками (Insertion Sort)

Суть алгоритма. Массив мысленно делится на две части: отсортированную (в начале) и неотсортированную. Изначально отсортированная часть содержит один первый элемент. Затем мы по одному берём элементы из неотсортированной части и **вставляем** их на нужное место в отсортированной части, сдвигая элементы при необходимости.

Реализация:

```python
def insertion_sort(arr: list[int]) -> None:
    # Проходим по массиву, начиная со второго элемента (i=1)
    for i in range(1, len(arr)):           # n-1 итераций
        key = arr[i]                       # O(1)
        j = i - 1                          # O(1)
        # Сдвигаем элементы отсортированной части, которые больше key
        while j >= 0 and arr[j] > key:     # В худшем случае i итераций
            arr[j + 1] = arr[j]            # O(1)
            j -= 1                         # O(1)
        # Вставляем key на найденное место
        arr[j + 1] = key                   # O(1)
```

Анализ времени работы (худший случай – массив отсортирован в обратном порядке):

Внешний цикл for выполняется $n - 1$ раз.

Внутренний цикл $while$ на итерации i выполняется в худшем случае i раз (когда key меньше всех элементов в отсортированной части).

Считаем общее количество операций (упрощённо, считая каждую строку за O(1)):

$$T(n) = (\text{кол} - \text{во итераций } for) \cdot (\text{стоимость одной итерации } for)$$

$$T(n) = \sum_{i=1}^{n-1} [(\text{стоимость } 4 - \text{х } O(1) \text{ операций } + \text{ стоимость цикла } while)]$$

$$T(n) = \sum_{i=1}^{n-1} [4 + i] \text{ (в худшем случае для } while)$$

$$T(n) = \sum_{i=1}^{n-1} 4 + \sum_{i=1}^{n-1} i = 4 \cdot (n-1) + \frac{n \cdot (n-1)}{2}$$

Отбрасывая константы и слагаемые низших порядков, получаем $T(n) = O(n^2)$. Более строго,

$$\sum_{i=1}^{n-1} i = \frac{n \cdot (n-1)}{2} = \frac{n^2 - n}{2} = O(n^2).$$

Сортировка вставками эффективна на почти отсортированных массивах (в лучшем случае, когда массив уже отсортирован, внутренний цикл не выполняется, и время работы становится $O(n)$), но плоха на больших случайных или обратно отсортированных массивах.

6. Время работы как функция от ввода

Важно помнить, что $T(n)$ – это не одна функция. Для одного алгоритма может быть:

1. **Худший случай**: $T(n) = O(n^2)$ для Insertion Sort. Это гарантированная верхняя оценка для любого входного данных размера n.
2. **Лучший случай**: $T(n) = \Omega(n)$ для Insertion Sort (уже отсортированный массив).
3. **Средний случай**: Его анализ часто является более сложной задачей, но для Insertion Sort он также составляет $O(n^2)$.

Анализируя алгоритм, всегда важно понимать, какой случай рассматривается.

7. Примеры асимптотических оценок

1. $O(n^2)$ – два вложенных цикла:

```python
def print_pairs(n: int) -> None:
    for i in range(n):      # n итераций
        for j in range(n):  # n итераций для каждого i
            print(i, j)     # O(1)
```

Общее количество операций: $n \cdot n \cdot O(1) = O(n^2)$.

2. $O(n\sqrt{n}) = O\left(n^{\frac{3}{2}}\right)$:

```
def example_sqrt(n: int) -> None:
    for i in range(n):              # n итераций
        j = 1
        while j * j <= n:           # ~√n итераций (пока j <= √n)
            print(i, j)             # O(1)
            j += 1
```

Общее количество операций: $n \cdot \sqrt{n} \cdot O(1) = O(n\sqrt{n})$.

3. Рекурсия с $O(n)$ (линейная рекурсия):

```
def linear_recursion(n: int) -> int:
    if n <= 1:
        return 1
    return linear_recursion(n - 1) + 1  # Один рекурсивный вызов с n-1
```

Глубина рекурсии – n. На каждом уровне выполняется O(1) операций (сравнение, сложение, вызов). Итого: $n \cdot O(1) = O(n)$.

4. Рекурсия с $O(\log n)$ (деление пополам):

```
def logarithmic_recursion(n: int) -> int:
    if n <= 1:
        return 1
    return logarithmic_recursion(n // 2) + 1 # Вызов с аргументом n/2
```

Сколько раз нужно поделить n на 2, чтобы получить 1? Это классический пример $\log_2 n$. На каждом из $\log_2 n$ уровней выполняется $O(1)$ операций. Итого: $O(\log n)$.

Основание логарифма. $O(\log_2 n) = O(\log_3 n) = O(\log n)$. Почему? По формуле замены основания логарифма: $\log_a n = \log_a b \cdot \log_b n$. Множитель $\log_a b$ является константой, а константы в О-нотации отбрасываются. Поэтому основание не указывают.

8. Сортировка слиянием (Merge Sort)

Это классический алгоритм «разделяй и властвуй».

Принцип работы:

1. **Разделение.** Рекурсивно разделяйте массив на две половины, пока не останутся подмассивы размером 1 или 0 элементов (которые по определению отсортированы).
2. **Властвование.** Рекурсивно сортируйте два подмассива.
3. **Объединение.** Сливайте два отсортированных подмассива в один отсортированный массив. Это ключевой шаг.

Реализация:

```
def merge_sort(arr: list[int]) -> list[int]:
    # Базовый случай: массивы длиной 0 или 1 уже отсортированы
    if len(arr) <= 1:
        return arr

    # Рекурсивный случай: разделяй
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]
```

```
    # Властвуй: рекурсивно сортируем обе половины
    left_sorted = merge_sort(left_half)
    right_sorted = merge_sort(right_half)

    # Объединяй: сливаем два отсортированных массива
    return merge(left_sorted, right_sorted)

def merge(left: list[int], right: list[int]) -> list[int]:
    result = []
    i = j = 0
    # Сравниваем элементы из left и right и добавляем меньший в result
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    # Добавляем "хвосты" от того массива, который не закончился
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Анализ времени работы:

Рекуррентное соотношение для времени работы $T(n)$:

$T(n) = O(1)$, если $n \leq 1$

$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$, если $n > 1$

1. $2 \cdot T\left(\frac{n}{2}\right)$ – это время на рекурсивную сортировку двух половин.

2. $O(n)$ – это время, необходимое для слияния двух массивов, чей общий размер равен n.

Дерево рекурсии:

Анализ работы алгоритма можно представить в виде дерева рекурсии.

1. **На самом верхнем уровне (уровень 0)** мы выполняем слияние всего массива длиной n, что требует работы $O(n)$.

2. Этот массив разбивается на две половины. **На следующем уровне (уровень 1)** мы выполняем две операции слияния, каждая для массива размером $\frac{n}{2}$. Таким образом, общая работа на этом уровне составляет $2 \cdot O\left(\frac{n}{2}\right)$, что упрощается до $O(n)$.

3. Каждая половина, в свою очередь, разбивается ещё пополам. **На уровне 2** мы выполняем четыре операции слияния массивов размером $\frac{n}{4}$. Общая работа здесь равна $4 \cdot O\left(\frac{n}{4}\right)$, что снова составляет $O(n)$.

4. Этот паттерн продолжается. На любом уровне k у нас будет 2^k операций слияния, каждая с размером $\frac{n}{2^k}$. Их общая работа всегда будет $2^k \cdot O\left(\frac{n}{2^k}\right) = O(n)$.

Глубина дерева. На каждом шаге n делится пополам. Сколько раз нужно разделить n, чтобы получить 1? Это $\log_2 n$ уровней.

На **каждом уровне** общая работа составляет $O(n)$.

Общее время работы. Количество уровней · Работа на уровне $= O(\log n) \cdot O(n) = O(n \cdot \log n)$.

Сортировка слиянием всегда работает за $\Theta(n \cdot \log n)$, что асимптотически быстрее, чем $\Theta(n^2)$.

9. Подсчёт инверсий в массиве

Инверсия – это пара индексов (i, j) такая, что $i < j$, но $a[i] > a[j]$. Количество инверсий показывает, насколько массив близок к отсортированному состоянию.

Пример:

В массиве $[2, 3, 9, 2, 9]$ есть две инверсии:

- $(2, 4) \rightarrow a[2] = 9 > 2 = a[4]$
- $(3, 4) \rightarrow a[3] = 9 > 2 = a[4]$

Наивный алгоритм:

Перебор всех пар за $O(n^2)$.

Эффективный алгоритм:

Модификация Merge Sort за $O(n \cdot \log n)$.

Алгоритм «Разделяй и властвуй»

1. **Разделяем** массив на две половины.
2. **Рекурсивно** считаем инверсии в каждой половине.
3. **Сливаем** половины, подсчитывая **разделённые инверсии** (между элементами из разных половин).

Модифицированная процедура слияния (Merge)

```
# Функция принимает два УЖЕ ОТСОРТИРОВАННЫХ списка и возвращает:
# 1. Объединённый отсортированный список
# 2. Количество "разделённых" инверсий (когда элемент из right меньше
элемента из left)
def merge_and_count(left: list[int], right: list[int]) -> tuple[list[int],
int]:
    # Инициализируем пустой список для результата слияния
    result = []

    # Инициализируем указатели для обхода списков left и right (начинаем с
начала)
    i = j = 0

    # Счётчик для инверсий, которые мы обнаружим в процессе слияния
    inversions = 0

    # Пока оба указателя не вышли за пределы своих списков
    while i < len(left) and j < len(right):
        # Если текущий элемент left МЕНЬШЕ ИЛИ РАВЕН текущему элементу right
        if left[i] <= right[j]:
            # Добавляем элемент из left в результат (он меньше)
```

```
            result.append(left[i])
            # Перемещаем указатель в left на следующий элемент
            i += 1
        else:
            # Если элемент из right МЕНЬШЕ элемента из left
            # Добавляем элемент из right в результат
            result.append(right[j])
            # Перемещаем указатель в right на следующий элемент
            j += 1

            # Ключевой момент: все ОСТАВШИЕСЯ элементы в left (начиная с
текущего i)
            # образуют инверсию с текущим элементом right[j-1] (который мы
только что добавили)
            # Потому что left уже отсортирован, и все элементы от i до конца
больше right[j-1]
            inversions += len(left) - i

    # После того как один из списков закончился,
    # добавляем все оставшиеся элементы из другого списка в конец результата
    # (они уже отсортированы и все больше последнего элемента в result)
    result.extend(left[i:])
    result.extend(right[j:])

    # Возвращаем объединённый отсортированный список и общее количество
найденных инверсий
    return result, inversions
```

Полная реализация

```
# Рекурсивная функция, которая:
# 1. Сортирует массив
# 2. Возвращает количество инверсий в нём
def count_inversions(arr: list[int]) -> tuple[list[int], int]:
    # Базовый случай рекурсии: массивы длиной 0 или 1 уже отсортированы
    # и не содержат инверсий (не может быть пар i < j)
    if len(arr) <= 1:
        return arr, 0

    # Находим середину массива для разделения на две части
    mid = len(arr) // 2

    # Рекурсивно вызываем count_inversions для левой половины
    # left - отсортированная левая половина
    # inv_left - количество инверсий внутри левой половины
    left, inv_left = count_inversions(arr[:mid])

    # Рекурсивно вызываем count_inversions для правой половины
    # right - отсортированная правая половина
    # inv_right - количество инверсий внутри правой половины
    right, inv_right = count_inversions(arr[mid:])

    # Сливаем два отсортированных массива, попутно подсчитывая инверсии
    # МЕЖДУ элементами из разных половин (split inversions)
    merged, inv_split = merge_and_count(left, right)
```

```
    # Общее количество инверсий =
    # инверсии в левой половине + инверсии в правой половине + инверсии между половинами
    return merged, inv_left + inv_right + inv_split
```

Анализ сложности

• Рекуррентное соотношение: $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$.

• По мастер-теореме: $T(n) = O(n \cdot \log n)$.

10. Мастер-теорема

Мастер-теорема – мощный инструмент для решения рекуррентностей вида:

$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$, где $a \geq 1$, $b > 1$.

1. $T(n)$ – это общее время работы алгоритма на входе размера n.

2. $a \cdot T\left(\frac{n}{b}\right)$ – эта часть выражения описывает время, затраченное на решение подзадач, на которые делится исходная задача.

3. a – количество подзадач (или рекурсивных вызовов), на которые делится исходная задача. Условие $a \geq 1$ означает, что должна быть хотя бы одна подзадача.

4. $\frac{n}{b}$ – размер каждой подзадачи. Предполагается, что все подзадачи имеют одинаковый размер. Условие $b > 1$ гарантирует, что подзадачи действительно меньше исходной ($\frac{n}{b} < n$).

5. $T\left(\frac{n}{b}\right)$ – время решения одной подзадачи размера $\frac{n}{b}$.

6. Таким образом, $a \cdot T\left(\frac{n}{b}\right)$ – это суммарное время, необходимое для решения всех a подзадач.

7. $f(n)$ – эта функция описывает время, затраченное на работу вне рекурсивных вызовов. Сюда входят следующие операции:

1. Время, необходимое для **разделения** исходной задачи на a подзадач.

2. Время, необходимое для **комбинирования** результатов решения этих подзадач в итоговое решение исходной задачи.

3. Другая работа, не связанная с рекурсией напрямую.

Функция $f(n)$ часто называется **стоимостью разделения и объединения**.

Мастер-теорема позволяет сразу «угадать» асимптотику, сравнивая функцию $f(n)$ с функцией $n^{\log_b a}$.

Существует три случая:

1. Если $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ для некоторого $\varepsilon > 0$, то $T(n) = \Theta\left(n^{\log_b a}\right)$.

2. Если $f(n) = \Theta\left(n^{\log_b a}\right)$, то $T(n) = \Theta\left(n^{\log_b a} \cdot \log n\right)$.

3. Если $f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right)$ для некоторого $\varepsilon > 0$ и если $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ для некоторой константы $c < 1$ и всех достаточно больших n (условие регулярности), то $T(n) = \Theta\left(f(n)\right)$.

Пример для Merge Sort:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$$

Здесь $a = 2$, $b = 2$, $f(n) = n$.

Сравниваем $f(n)$ и $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.

Попадаем в **случай 2**: $f(n) = \Theta(n)$.

Следовательно, $T(n) = \Theta\left(n^{\log_b a} \cdot \log n\right) = \Theta(n \cdot \log n)$.

11. Быстрое умножение полиномов (алгоритм Карацубы)

Задача. Перемножить два полинома (или длинных числа), представленных как массивы коэффициентов. Наивный алгоритм имеет сложность $O(n^2)$.

Карацуба предложил алгоритм «разделяй и властвуй».

Пусть есть два полинома $A(x)$ и $B(x)$ степени $n-1$ (имеют n коэффициентов).

Разобьём каждый на половины:

$$A(x) = A_{high}(x) \cdot x^{\frac{n}{2}} + A_{low}(x)$$
$$B(x) = B_{high}(x) \cdot x^{\frac{n}{2}} + B_{low}(x)$$

Их произведение:

$$A \cdot B = \left(A_{high} \cdot B_{high}\right) \cdot x^n + \left(A_{high} \cdot B_{low} + A_{low} \cdot B_{high}\right) \cdot x^{\frac{n}{2}} + (A_{low} \cdot B_{low})$$

Это требует 4 умножения половинной длины. Рекуррентность: $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + O(n)$ (сложение и сдвиг (умножение на x^k) линейны). По мастер-теореме: $a = 4$, $b = 2$, $n^{\log_b a} = n^2$. $f(n) = O(n^{2-\varepsilon})$ для $\varepsilon = 1$ (случай 1). Значит, $T(n) = \Theta(n^2)$ – не лучше наивного.

Карацуба заметил, что можно обойтись **тремя** умножениями.

Вычислим:

$$D_0 = A_{low} \cdot B_{low}$$
$$D_1 = \left(A_{low} + A_{high}\right) \cdot \left(B_{low} + B_{high}\right)$$
$$D_2 = A_{high} \cdot B_{high}$$

Тогда средний коэффициент можно выразить как:

$$A_{high} \cdot B_{low} + A_{low} \cdot B_{high} = D_1 - D_0 - D_2$$

Итоговое произведение:

$$A \cdot B = D_2 \cdot x^n + (D_1 - D_0 - D_2) \cdot x^{\frac{n}{2}} + D_0$$

Теперь рекуррентность: $T(n) = 3 \cdot T\left(\frac{n}{2}\right) + O(n)$.

Применяем мастер-теорему:

$a = 3, b = 2, n^{\log_b a} = n^{\log_2 3} \approx n^{1,58496}$

$f(n) = O(n) = O(n^1)$.

Так как $1 < 1{,}58496$, имеем $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ для $\varepsilon \approx 0{,}58496$ (случай 1).

Следовательно, $T(n) = \Theta\left(n^{\log_2 3}\right)$.

Реализация (упрощённая, для чисел):

```
def karatsuba(x: int, y: int) -> int:
    # Базовый случай для простоты
    if x < 10 or y < 10:
        return x * y

    # Вычисляем размер чисел
    n = max(len(str(x)), len(str(y)))
    m = n // 2

    # Разделяем числа
    high1, low1 = x // (10 ** m), x % (10 ** m)
    high2, low2 = y // (10 ** m), y % (10 ** m)

    # 3 рекурсивных умножения
    z0 = karatsuba(low1, low2)
    z1 = karatsuba((low1 + high1), (low2 + high2))
    z2 = karatsuba(high1, high2)

    # Комбинируем результат
    return (z2 * (10 ** (2 * m))) + ((z1 - z2 - z0) * (10 ** m)) + z0
```

12. Алгоритм Штрассена для умножения матриц

Наивное умножение матриц $n \times n$ имеет сложность $O(n^3)$ (три вложенных цикла).

Фолькер Штрассен применил метод «разделяй и властвуй», разбивая каждую матрицу на 4 блока:

$C = A * B$, где $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$

Наивный расчёт:

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

Это требует 8 умножений матриц половинного размера $(\frac{n}{2} \times \frac{n}{2})$. Рекуррентность: $T(n) = 8 \cdot T\left(\frac{n}{2}\right) + O(n^2)$ (сложение матриц $\frac{n}{2} \times \frac{n}{2}$ имеет сложность $O\left(\left(\frac{n}{2}\right)^2\right) = O(n^2)$). По мастер-теореме: $a = 8$, $b = 2$, $n^{\log_b a} = n^3$. $f(n) = O(n^2) = O(n^{3-\varepsilon})$ для $\varepsilon = 1$ (случай 1). $T(n) = \Theta(n^3)$ – без улучшения.

Штрассен нашёл способ вычислять C с помощью **7** умножений.

Вычислим вспомогательные матрицы:

$$M_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22}) \cdot B_{11}$$
$$M_3 = A_{11} \cdot (B_{12} - B_{22})$$
$$M_4 = A_{22} \cdot (B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12}) \cdot B_{22}$$
$$M_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$

И затем выразим C через $M_1, \dots, M_7$:

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Теперь рекуррентность: $T(n) = 7 \cdot T\left(\frac{n}{2}\right) + O(n^2)$.

Применяем мастер-теорему:

$a = 7, b = 2, n^{\log_b a} = n^{\log_2 7} \approx n^{2,8074}$

$f(n) = O(n^2)$.

Так как $2 < 2{,}8074$, имеем $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ для $\varepsilon \approx 0{,}8074$ (случай 1).

Следовательно, $T(n) = \Theta\left(n^{\log_2 7}\right) \approx \mathrm{O}(n^{2,8074})$.

Это асимптотически быстрее наивного алгоритма. Существуют и более быстрые алгоритмы (алгоритм Копперсмита-Винограда, $O(n^{2,373})$), но они имеют огромные константные множители и непрактичны для реальных применений.