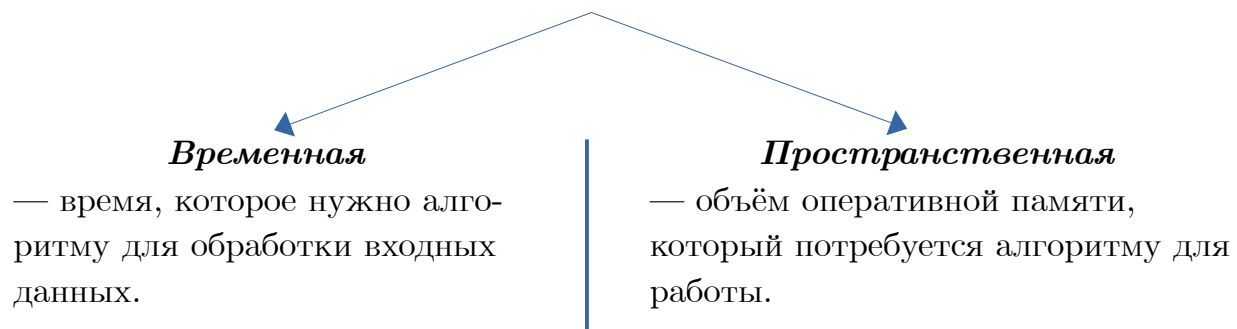


## Вопросы

- [55. Большая О Нотация. Асимптотика.](#)
- [56. Поиск элемента в массиве. Асимптотика.](#)
- [57. Пользовательские типы данных в C++.](#)
- [58. Нечёткий поиск. Расстояние Левенштейна.](#)
- [59. Оптимизированное перемножение матриц. Асимптотика.](#)
- [60. Флаги компиляций приложений.](#)
- [61. Компилирование с использованием статических библиотек.](#)
- [62. Компилирование с использованием динамических библиотек.](#)

**Вопрос 55.** Большая О Нотация. Асимптотика.

**Вычислительная сложность алгоритма** — функция, определяющая зависимость объёма работы, выполняемой некоторым алгоритмом, от свойств входных данных.



- Когда говорят о *Time Complexity* или просто *Time*, то речь идёт именно о количестве элементарных операций, осуществляемых алгоритмом.

**Замечание.** В теории алгоритмов разница в скорости выполнения между операциями обычно опускается. Поэтому сложение простых чисел и деление чисел с плавающей точкой считаются равными по сложности операциями.

- Когда говорят о *Space Complexity* или просто *Space* (редко *Memory*), то речь идёт именно о количестве ячеек памяти, необходимых для выполнения алгоритма.

**Замечание.** В теории алгоритмов все ячейки считаются равноценными. Например, `int` на 4 байта и `double` на 8 байт имеют один вес.

- *In-place* алгоритмы (*на месте*) — алгоритмы, которые используют исходный массив как рабочее пространство.

- *Out-of-place* алгоритмы (*вне места*) — алгоритмы, требующие дополнительной памяти: копии исходного массива или дополнительные структуры данных.
- Таким образом, *оптимальным* называется алгоритм, решающий поставленную задачу за наименьшее возможное время и использующий минимальное возможное количество памяти.

**Большая O Нотация (Big O Notation)** — способ описания асимптотического поведения функций (поведение функции, когда её аргумент стремится к бесконечности | вспоминаем O-большое и o-маленькое из матанализа : D), который используется для анализа временной и пространственной сложности алгоритмов.

Примеры описания вычислительной сложности алгоритмов через «O»:

#### **Временная:**

- $O(1)$  — константное время. Например, определение размера массива, сложение чисел и т.п.
- $O(\log \log n)$  — двойное логарифмическое время. Например, интерполяционный поиск
- $O(\log n)$  — логарифмическая сложность. Например, бинарный поиск, бинарное возведение в степень.
- $O(n)$  — линейное время. Например, линейный поиск.
- $O(n^c)$  — полиномиальное время (квадратичное, кубическое и т.д.)
- $O(c^n)$  — экспоненциальное время. Например, рекурсивная реализация последовательности Фибоначчи.
- $O(n!)$  — факториальное время. Например, задача о коммивояжёре или задача полного перебора.

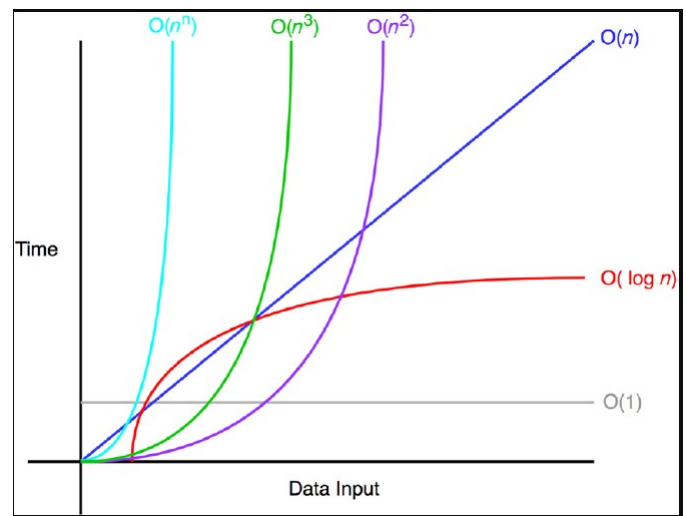
#### **Пространственная:**

- $O(1)$  — константное пространство. Например, алгоритм с фиксированным количеством переменных.
- $O(n)$  — линейное пространство. Например, одномерный массив.
- $O(n^c)$  — полиномиальное пространство. Например, многомерный массив.

#### **Лирическое отступление:**

экспоненциальное и факториальное пространства тоже существуют, но я не уверен, что их стоит упоминать (например, массив из всех возможных перестановок элементов массива длиной  $n$ ).

\*при наличии времени и желания, можете нарисовать для Вадима график



*Правила определения сложности алгоритма:*

- При расчете «О» используется два правила:
  1. Константы откидываются:

$$O(3n) = O(n)$$

$$O(10000n^2) = O(n^2)$$

$$O(2n \log n) = O(n \log n)$$

**Важное замечание.** Из-за того, что в большой О нотации константы откидываются, алгоритмы с большей асимптотической сложностью могут работать быстрее алгоритмов с меньшей асимптотической сложностью. Так, например, алгоритм Штрассена с асимптотикой  $\approx O(n^{2.81})$  (кстати, асимптотика этого алгоритма посчитана с использованием мастер-теоремы) (вопрос об оптимизированном перемножении матриц) работает быстрее алгоритма Копперсмита-Винограда с асимптотикой  $\approx O(n^{2.37})$  на малых и средних матрицах из-за очень высоких скрытых констант в алгоритме Копперсмита-Винограда.

2. Если в «О» есть сумма, то записывается только самое *быстрорастущее* слагаемое (это называется асимптотической сложностью).

$$O(n^2 + n) = O(n^2)$$

$$O(n^3 + 100n \log n) = O(n^3)$$

$$O(1.1^n + n^{100}) = O(1.1^n)$$

- Циклы и вложенные циклы:
  1. Временная сложность цикла, в котором  $n$  раз повторяется функция с временной сложностью  $O(m)$ , равна:  $n * O(m) = O(n * m)$ .
  2. При добавлении вложенных циклов сложность растёт экспоненциально: в цикл с  $O(n)$  добавили еще один цикл  $\rightarrow$  общая сложность  $O(n^2)$ .
- Мастер-теорема

Используется для оценки сложности рекурсивных алгоритмов.

#### Общая форма [\[ править \]](#) [\[ править код \]](#)

Основная теорема рассматривает следующие рекуррентные соотношения:

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

В применении к анализу алгоритмов константы и функции обозначают:

$n$  — размер задачи.

$a$  — количество подзадач в рекурсии.

$n/b$  — размер каждой подзадачи. (Предполагается, что все подзадачи на каждом этапе имеют одинаковый размер.)

$f(n)$  — оценка сложности работы, производимой алгоритмом вне рекурсивных вызовов. В неё также включается вычислительная стоимость деления на подзадачи и объединения результатов решения подзадач.

\*возможно, не стоит эту теорему расписывать, т. к. на степике Вадим Денисович даёт только: «По сути, это набор правил по оценке сложности. Он учитывает, сколько новых ветвей рекурсии создаётся на каждом шаге и на сколько частей дробятся данные в каждом шаге рекурсии. Это если вкратце.»

- Метод Монте-Карло

1. Применяется, только если невозможно оценить сложность через вложенность циклов или с помощью мастер-теоремы.
2. Алгоритм проверяется на данных разного размера. Затем строятся графики зависимости затраченных времени и памяти от размера данных. Затем по этим графикам вычисляется функция, которая лучше всего описывает полученное облако точек.

P.S. (возможно, это что-то важное). Не помню, в каком вопросе Вадим просил написать Quick Sort, поэтому пусть будет здесь:

Реализация для вектора.

*Time Complexity:*

Средний случай:  $O(n \log n)$

Худший случай:  $O(n^2)$

Настоятельно рекомендуется написать qsort самостоятельно. Этот говнокод использовать лишь в качестве подсказки

```
int partition(vector<int> &vec, int low, int high) {
    // Selecting last element as the pivot
    int pivot = vec[high];
    // Index of element just before the last element
    // It is used for swapping
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {

        // If current element is smaller than or
        // equal to pivot
        if (vec[j] <= pivot) {
            i++;
            swap(vec[i], vec[j]);
        }
    }
    // Put pivot to its position
    swap(vec[i + 1], vec[high]);
    // Return the point of partition
    return (i + 1);
}

void quickSort(vector<int> &vec, int low, int high) {
    // Base case: This part will be executed till the starting
    // index low is lesser than the ending index high
    if (low < high) {
        // pi is Partitioning Index, arr[p] is now at
        // right place
        int pi = partition(vec, low, high);
        // Separately sort elements before and after the
        // Partition Index pi
        quickSort(vec, low, pi - 1);
        quickSort(vec, pi + 1, high);
    }
}
```

## Вопрос 56. Поиск элемента в массиве. Асимптотика.

**Поиск элемента в массиве** — алгоритм, который находит индекс элемента в массиве или определяет, что элемент отсутствует.

Поиск элемента в массиве разделяют на две большие категории:

- *поиск в несортированном массиве* (например, линейный поиск)
- *поиск в сортированном массиве* (например, бинарный поиск, интерполирующий поиск)

**Замечание.** Для облегчения работы с элементами массива, в том числе для облегчения поиска элемента в массиве, нелишним бывает отсортировать массив.

### **Линейный поиск:**

- Подходит для поиска элемента в небольших несортированных массивах.

**Замечание.** Если массив большой, то рекомендуется его отсортировать и применять более эффективные алгоритмы поиска.

- Перебираются все элементы массива и проверяются на совпадение с заданными критериями или ключами.
- Time Complexity:  $O(n)$ . В массиве миллион элементов → миллион операций (в худшем случае).

```
int linSearch(int arr[], int requiredKey, int arrSize) {
    for (int i = 0; i < arrSize; i++) {
        if (arr[i] == requiredKey)
            return i;
    }
    return -1;
}
```

### **Бинарный (двоичный) поиск:**

- Подходит для поиска элемента в отсортированном массиве.
- На каждом шаге алгоритма выбирается средний элемент массива и сравнивается с искомым элементом. Если элементы

```
int SearchingAlgorithm(int arr[], int left, int right, int keys) {
    int middle = 0;
    while (true) {
        middle = (left + right) / 2;
        if (keys < arr[middle])
            right = middle - 1;
        else if (keys > arr[middle])
            left = middle + 1;
        else
            return middle;
        if (left > right)
            return -1;
    }
}
```

совпадают, то алгоритм завершает работу: элемент найден. Иначе половина элементов массива «отбрасывается» и поиск продолжается в другой половине массива.

- Time Complexity:  $O(\log n)$ . В отсортированном массиве миллион элементов → 20 операций (в худшем случае).

### **Интерполирующий поиск:**

- Подходит для поиска элемента в отсортированном массиве.
- Вместо того, чтобы брать средний элемент, как в бинарном поиске, интерполирующий поиск вычисляет позицию искомого элемента по формуле:

$$\text{pos} = \text{lo} + \frac{(x - \text{arr}[\text{lo}]) * (\text{hi} - \text{lo})}{(\text{arr}[\text{hi}] - \text{arr}[\text{lo}])}$$

```
int interpolationSearch(int arr[], int lo, int hi, int x)
{
    int pos;
    if (lo <= hi && x >= arr[lo] && x <= arr[hi]) {
        pos = lo
            + (((double)(hi - lo) / (arr[hi] - arr[lo]))
              * (x - arr[lo]));
        if (arr[pos] == x)
            return pos;
        if (arr[pos] < x)
            return interpolationSearch(arr, pos + 1, hi, x);
        if (arr[pos] > x)
            return interpolationSearch(arr, lo, pos - 1, x);
    }
    return -1;
}
```

где lo, hi — нижняя и верхняя границы поиска соответственно, arr[] — заданный массив, x — искомый элемент, pos — позиция, на которой предположительно находится искомый элемент. Далее, как и в бинарном поиске, элемент на позиции pos сравнивается с искомым элементом.

- Time Complexity:  $O(\log \log n)$ .  
**Важное замечание.** При плохих исходных данных (например, при экспоненциальном возрастании элементов) время работы может ухудшиться до  $O(n)$ .  
**Еще одно важное замечание.**

Эксперименты показали, что интерполяционный поиск не настолько снижает количество выполняемых сравнений, чтобы компенсировать требуемое для дополнительных вычислений время (пока таблица не очень велика). Кроме того, типичные таблицы недостаточно случайны, да и разница между значениями  $\log \log n$  и  $\log n$  становится значительной только при очень больших  $n$ . На практике при поиске в больших файлах оказывается выгодным на ранних стадиях применять интерполяционный поиск, а затем, когда диапазон существенно уменьшится, переходить к двоичному.

## **Вопрос 57.** Пользовательские типы данных в C++.

**Пользовательские типы данных** — типы данных, определяемые пользователем для организации данных и управления ими в программе.

Основные пользовательские типы данных:

### 1. Структуры (struct):

- пользовательский тип данных, который позволяет объединить различные типы данных в одну логическую единицу.
- Поле в структуре могут быть: переменные базовых типов C++, вложенные структуры, объединения (о них дальше), классы (тоже дальше), функции.
- Для определения структуры применяется ключевое слово struct, а сам формат структуры выглядит так:

```
struct имя_структуры {  
    компоненты_структуры  
};
```

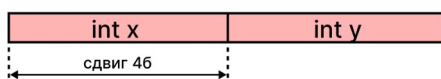
- Для определения нового типа используется ключевое слово

```
#include <conio.h>  
#include <stdio.h>  
  
//Определяем новую структуру  
struct point_t {  
    int x;  
    int y;  
};  
  
//Определяем новый тип  
typedef struct point_t Point;  
  
void main() {  
    //Обращение через имя структуры  
    struct point_t p = {10, 20};  
    //Обращение через новый тип  
    Point px = {10, 20};  
  
    getch();  
}
```

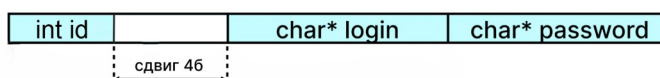
- Для обращения к элементу структуры через указатель на typedef: структуру используется операция «стрелка»: ->

```
Point p = {10, 20};  
Point* ptr = &p;  
  
ptr->x += 1;    // x = 11  
ptr->y += 2;    // y = 22
```

- Устройство структуры в памяти: размер структуры не всегда равен сумме размеров её полей, т.к. компилятор оптимизирует расположение структуры в памяти, подгоняя некоторые поля до четных адресов.



← Размер структуры: 8 байт



← Размер структуры: 24 байт  
(выравнивание под char\* - 8 байт)

**Замечание.** Есть возможность изменить упаковку структур в памяти: `#pragma pack(n)`. По умолчанию  $n = 8$ . Допустимыми значениями являются 1, 2, 4, 8 и 16. Выравнивание поля происходит по адресу, кратному  $n$  или сумме нескольких полей объекта, в зависимости от того, какая из этих величин меньше.

**Замечание к замечанию.** Использование `#pragma pack` не приветствуется: логика работы программы не должна зависеть от внутреннего представления структуры (если, конечно, вы не занимаетесь системным программированием или ломаете чужие программы и сети).

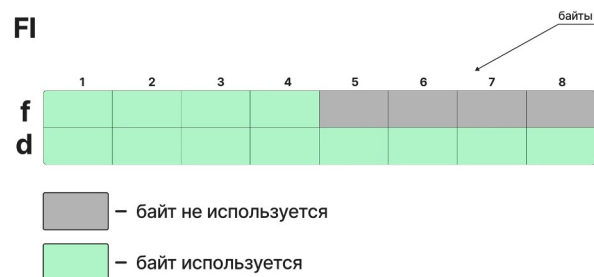
**P.S.** Нигде не указал: к полям структуры обращаемся через точку «.»: `Point.x += 1;`

## 2. Объединения (*union*):

- Объединение — группирование элементов, разделяющих одну и ту же область памяти (т.е. все переменные, что включены в объединение, начинаются с одной границы).

```
union Floats {  
    float f; // рассматривается 4 байта  
    double d; // рассматривается 8 байт  
};
```

В данном случае `sizeof(Floats) = 8`.



- Типом переменной в объединении может быть: базовый тип в C++, структура, объединение, класс.
- С указателями на объединение работаем так же, как со структурами «->». К полям обращаемся через точку. Вложенность работает так же, как со структурами.

## 3. Битовые поля:

- Позволяют формировать объекты с длиной, не кратной байту:

```
struct Example {  
    unsigned int field1 : 3; // 3 бита  
    unsigned int field2 : 5; // 5 бит  
    unsigned int field3 : 1; // 1 бит  
};
```



- Ширина поля не может превышать длину машинного слова.

#### 4. Перечисления (*enum*):

- Перечисления позволяют задавать переменные, принимающие определенный набор значений:
- Пример работы с перечислением:

```
enum Color {
    Red,
    Green,
    Blue
};
```

```
#include <iostream>

enum Season{
    Winter,
    Spring,
    Summer,
    Autumn
};

int main() {
    Season currentSeason = Summer;
    if (currentSeason == Summer) {
        std::cout << "+вайб" << std::endl;
    }
    else {
        std::cout << "-вайб" << std::endl;
    }

    return 0;
}
```

P.S. Выведет +вайб

#### 5. Ключевое слово *typedef*:

- Используется для создания псевдонимов (синонимов) для существующих типов данных.

- ```
typedef unsigned long ulong;
ulong a = 1000;
//Теперь можно использовать ulong вместо unsigned long
```

#### 6. Классы (*class*):

**Лирическое отступление.** В семестре их не было, так что я не уверен, что их надо писать, но пусть что-то будет.

- Класс в C++ — это шаблон для создания объектов, который объединяет данные и функции, работающие с этими данными.
- Объявление класса позволяет описать не только данные представления объекта, но и аспекты использования таких данных:

1. список функций доступа к объектам;
2. правила наследования объектов производными классами;
3. различная степень защиты элементов объектов.

- Пример:

```
class ClassName {  
public:  
    // Конструктор  
    ClassName();  
  
    // Методы  
    void method();  
  
private:  
    // Поля данных  
    int data;  
};
```

**Вопрос 58.** Нечёткий поиск. Расстояние Левенштейна.

***Нечёткий поиск*** — это метод поиска, который позволяет находить совпадения, даже если вводимые данные содержат ошибки, опечатки или незначительные различия от искомого значения.

***Расстояние Левенштейна*** — минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Пример:

$x = \text{“привет”}$

$y = \text{“прияет”}$

расстояние Левенштейна  $L(x, y) = 1$  — одна замена символа. Можно привести и другие примеры:

$L(\text{“test”}, \text{“ttext”}) = 2;$

$L(\text{“ОАиП”}, \text{“н”}) = 4.$

### *Алгоритм нечёткого поиска:*

- Для каждой строки словаря найдем, сколько у нее отличий с заданной строкой. Ответом будут те строки из словаря, отличий с которыми не больше заданного числа  $k$ .
- Алгоритм нахождения расстояния Левенштейна — алгоритм Вагнера-Фишера:
  1. Пусть  $s1$  и  $s2$  - строки, между которыми нужно найти расстояние. Их размеры соответственно  $n$  и  $m$
  2.  $D(i, j)$  будет означать расстояние между префиксом длины  $i$  строки  $s1$  и префиксом длины  $j$  строки  $s2$ .
  3. Тогда ответ на задачу -  $D(n, m)$ .
  4. Будем считать, что нумерация символов в строке с единицы
  5. Базовые значения:
    6.  $D(0, 0) = 0$ ,
    7.  $D(i, 0) = i, D(0, j) = j$
    8. Для  $i, j \neq 0$ :
      - $D(i, j) = D(i - 1, j - 1)$ , если  $s1[i] = s2[j]$
      - $\min(D(i, j - 1) + 1, D(i - 1, j) + 1, D(i - 1, j - 1) + 1)$ , иначе
  9. (последней операцией могли приписать символ к строке, удалить, или заменить символ)

### *Реализация на C++:*

```
int diff(string a, string b) {
    int n = (int)a.size(), m = (int)b.size();
    int d[n + 1][m + 1];
    for (int i = 0; i <= n; i++)
        d[i][0] = i;
    for (int j = 0; j <= m; j++)
        d[0][j] = j;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (a[i - 1] == b[j - 1])
                d[i][j] = d[i - 1][j - 1];
            else
                d[i][j] = min({d[i - 1][j] + 1, d[i][j - 1] + 1, d[i - 1][j - 1] + 1});
    return d[n][m];
}
```

**Time Complexity:**  $O(n*m)$ , где  $n$  – размер введенной строки,  $m$  — сумма длин всех строк словаря.

**Плюсы использования:**

- Работает довольно быстро, если длина искомой строки и суммарная длина словаря небольшие.
- Алгоритм довольно прост в понимании и реализации.

**Минусы использования:**

- Работает довольно медленно при большой длине искомой строки и/или суммарной длине словаря

**Вопрос 59.** Оптимизированное перемножение матриц. Асимптотика.

**Замечание.** Будем говорить, что размеры матриц  $A$  порядка  $m*n$  и  $B$  порядка  $p*k$  согласованы относительно умножения, если  $n = p$ .

**Виды алгоритмов перемножения матриц:**

1. Стандартный алгоритм:

- сумма произведений элементов  $i$ -ой строки матрицы  $A$  на элементы  $j$ -ой строки матрицы  $B$  (рассмотрим подробнее чуть дальше).

2. Алгоритмы, основанные на блокировании:

- матрицы разбиваются на блоки, что уменьшает время доступа к данным и позволяет лучше использовать кэш-память процессора.
- Time Complexity:  $O(n^3)$ , но на практике может работать быстрее за счёт уменьшения накладных расходов на память.

3. Алгоритмы с использованием параллельных вычислений:

- Для ускорения умножения матриц используются многоядерные процессоры и распределенные системы для параллельных вычислений.

4. Алгоритм Штрассена:

- Матрица разбивается на 4 подматрицы и используется 7 рекурсивных умножений вместо 8, что уменьшает количество операций.
- Time Complexity:  $O(n^{\log_2 7}) \approx O(n^{2.81})$ .

#### 5. Алгоритм Копперсмита-Винограда:

- Используются ассоциативные правила перемножений и требуются сложные математические выкладки.
- Time Complexity:  $O(n^{2.376})$ . На практике он медленнее, чем алгоритм Штрассена, на матрицах маленького и среднего размера, из-за огромной скрытой константы. Поэтому чаще используют алгоритм Штрассена.

#### Стандартное перемножение матриц:

- Работает за  $O(n^3)$ .
- Ниже представлен код, реализующий стандартное перемножение матриц.

```
void mulMat(vector<vector<int>>& m1, vector<vector<int>>& m2,
           vector<vector<int>>& res) {
    int r1 = m1.size();
    int c1 = m1[0].size();
    int r2 = m2.size();
    int c2 = m2[0].size();
    if (c1 != r2) {
        cout << "Invalid Input" << endl;
        exit(EXIT_FAILURE);
    }

    // Resize result matrix to fit the result dimensions
    res.resize(r1, vector<int>(c2, 0));

    for (int i = 0; i < r1; i++) {
        for (int j = 0; j < c2; j++) {
            for (int k = 0; k < c1; k++) {
                res[i][j] += m1[i][k] * m2[k][j];
            }
        }
    }
}
```

**Замечание.** Рекомендуется написать код самостоятельно и запустить на своей машине.

#### Алгоритм Штрассена:

- Используется методика *разделяй и властвуй*: делим исходную матрицу на 4 подматрицы порядка  $(N/2) * (N/2)$ , после чего выполняется 7 рекурсивных перемножений подматриц (вместо стандартных 8, как в алгоритмах, основанных на блокировании):

$$\begin{aligned}
 p1 &= a(f - h) & p2 &= (a + b)h \\
 p3 &= (c + d)e & p4 &= d(g - e) \\
 p5 &= (a + d)(e + h) & p6 &= (b - d)(g + h) \\
 p7 &= (a - c)(e + f)
 \end{aligned}$$

The A x B can be calculated using above seven multiplications.  
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A                      B                      C

A, B and C are square matrices of size N x N  
a, b, c and d are submatrices of A, of size N/2 x N/2  
e, f, g and h are submatrices of B, of size N/2 x N/2  
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

- Временная асимптотическая сложность посчитана с помощью мастер-теоремы:  $O(n^{\log_2 7}) \approx O(n^{2.81})$ . Если запомните формулу, то можете написать:  
 $T(N) = 7T(N/2) + O(N^2)$ .
- Пример кода (Да пребудет с вами сила!):

```

#include <bits/stdc++.h>
using namespace std;

#define ROW_1 4
#define COL_1 4

#define ROW_2 4
#define COL_2 4

void print(string display, vector<vector<int>> > matrix,
           int start_row, int start_column, int end_row,
           int end_column)
{
    cout << endl << display << " ==>" << endl;
    for (int i = start_row; i <= end_row; i++) {
        for (int j = start_column; j <= end_column; j++) {
            cout << setw(10);
            cout << matrix[i][j];
        }
        cout << endl;
    }
    cout << endl;
    return;
}

vector<vector<int>> >
add_matrix(vector<vector<int>> > matrix_A,
           vector<vector<int>> > matrix_B, int split_index,
           int multiplier = 1)
{
    for (auto i = 0; i < split_index; i++)

```

```

        for (auto j = 0; j < split_index; j++)
            matrix_A[i][j]
                = matrix_A[i][j]
                  + (multiplier * matrix_B[i][j]);
    return matrix_A;
}

vector<vector<int>> >
multiply_matrix(vector<vector<int>> > matrix_A,
                vector<vector<int>> > matrix_B)
{
    int col_1 = matrix_A[0].size();
    int row_1 = matrix_A.size();
    int col_2 = matrix_B[0].size();
    int row_2 = matrix_B.size();

    if (col_1 != row_2) {
        cout << "\nError: The number of columns in Matrix "
              "A must be equal to the number of rows in "
              "Matrix B\n";
        return {};
    }

    vector<int> result_matrix_row(col_2, 0);
    vector<vector<int>> > result_matrix(row_1,
                                       result_matrix_row);

    if (col_1 == 1)
        result_matrix[0][0]
            = matrix_A[0][0] * matrix_B[0][0];
    else {
        int split_index = col_1 / 2;

        vector<int> row_vector(split_index, 0);

        vector<vector<int>> > a00(split_index, row_vector);
        vector<vector<int>> > a01(split_index, row_vector);
        vector<vector<int>> > a10(split_index, row_vector);
        vector<vector<int>> > a11(split_index, row_vector);
        vector<vector<int>> > b00(split_index, row_vector);
        vector<vector<int>> > b01(split_index, row_vector);
        vector<vector<int>> > b10(split_index, row_vector);
        vector<vector<int>> > b11(split_index, row_vector);

        for (auto i = 0; i < split_index; i++)
            for (auto j = 0; j < split_index; j++) {
                a00[i][j] = matrix_A[i][j];
                a01[i][j] = matrix_A[i][j] + split_index;
                a10[i][j] = matrix_A[split_index + i][j];
            }
    }
}

```

```

        a11[i][j] = matrix_A[i + split_index]
                        [j + split_index];
        b00[i][j] = matrix_B[i][j];
        b01[i][j] = matrix_B[i][j + split_index];
        b10[i][j] = matrix_B[split_index + i][j];
        b11[i][j] = matrix_B[i + split_index]
                        [j + split_index];
    }

```

```

    vector<vector<int>> > p(multiply_matrix(
        a00, add_matrix(b01, b11, split_index, -1)));
    vector<vector<int>> > q(multiply_matrix(
        add_matrix(a00, a01, split_index), b11));
    vector<vector<int>> > r(multiply_matrix(
        add_matrix(a10, a11, split_index), b00));
    vector<vector<int>> > s(multiply_matrix(
        a11, add_matrix(b10, b00, split_index, -1)));
    vector<vector<int>> > t(multiply_matrix(
        add_matrix(a00, a11, split_index),
        add_matrix(b00, b11, split_index)));
    vector<vector<int>> > u(multiply_matrix(
        add_matrix(a01, a11, split_index, -1),
        add_matrix(b10, b11, split_index)));
    vector<vector<int>> > v(multiply_matrix(
        add_matrix(a00, a10, split_index, -1),
        add_matrix(b00, b01, split_index)));

```

```

    vector<vector<int>> > result_matrix_00(add_matrix(
        add_matrix(add_matrix(t, s, split_index), u,
            split_index),
        q, split_index, -1));
    vector<vector<int>> > result_matrix_01(
        add_matrix(p, q, split_index));
    vector<vector<int>> > result_matrix_10(
        add_matrix(r, s, split_index));
    vector<vector<int>> > result_matrix_11(add_matrix(
        add_matrix(add_matrix(t, p, split_index), r,
            split_index, -1),
        v, split_index, -1));

```

```

    for (auto i = 0; i < split_index; i++)
        for (auto j = 0; j < split_index; j++) {
            result_matrix[i][j]
                = result_matrix_00[i][j];
            result_matrix[i][j + split_index]
                = result_matrix_01[i][j];
            result_matrix[split_index + i][j]
                = result_matrix_10[i][j];
            result_matrix[i + split_index]
                [j + split_index]

```



```

        = result_matrix_11[i][j];
    }

    a00.clear();
    a01.clear();
    a10.clear();
    a11.clear();
    b00.clear();
    b01.clear();
    b10.clear();
    b11.clear();
    p.clear();
    q.clear();
    r.clear();
    s.clear();
    t.clear();
    u.clear();
    v.clear();
    result_matrix_00.clear();
    result_matrix_01.clear();
    result_matrix_10.clear();
    result_matrix_11.clear();
}

return result_matrix;
}

int main()
{
    vector<vector<int>> > matrix_A = { { 1, 1, 1, 1 },
   { 2, 2, 2, 2 },
   { 3, 3, 3, 3 },
   { 2, 2, 2, 2 } };

    print("Array A", matrix_A, 0, 0, ROW_1 - 1, COL_1 - 1);

    vector<vector<int>> > matrix_B = { { 1, 1, 1, 1 },
   { 2, 2, 2, 2 },
   { 3, 3, 3, 3 },
   { 2, 2, 2, 2 } };

    print("Array B", matrix_B, 0, 0, ROW_2 - 1, COL_2 - 1);

    vector<vector<int>> > result_matrix(
        multiply_matrix(matrix_A, matrix_B));

    print("Result Array", result_matrix, 0, 0, ROW_1 - 1,
        COL_2 - 1);
}

// Time Complexity: T(N) = 7T(N/2) + O(N^2) => O(N^2 Log7)

```

**Замечание.** Думаю, писать алгоритм Штрассена на экзамене не надо. Просто надо озвучить идею и рассказать про асимптотику алгоритма, а также написать про скрытые константы в алгоритмах Штрассена и Копперсмита-Винограда. Также рекомендую запустить код на своей машине (если он вообще запустится w :)

## Вопрос 60. Флаги компиляций приложений.

**Флаги компиляции приложения** — параметры, которые передаются компилятору для изменения поведения процесса компиляции. (Что такое компилятор? 🤔)

### 1. Флаги оптимизаций (англ. буква *O*):

- используются для улучшения производительности сгенерированного кода, уменьшая его размер или время выполнения.
- -O0 — без оптимизаций (по умолчанию).
- -O1 — основная оптимизация (уменьшает размер кода и улучшает производительность).
- -O2 — более агрессивная оптимизация (включает все оптимизации из -O1 и добавляет новые).
- -O3 — максимальная оптимизация (включает все оптимизации из -O2 и добавляет новые (может даже увеличить размер кода)).
- -Os — оптимизация для уменьшения размера кода.
- -Ofast — включает все оптимизации из -O3 и добавляет агрессивные оптимизации, которые могут нарушать стандартное поведение.

### 2. Флаги отладки:

- позволяют включить информацию для отладки (что такое *отладка*? 🤔)
- -g — включает отладочную информацию в сгенерированный код.
- -ggdb — включает отладочную информацию, специфичную для GDB (GNU Debugger).
- **Замечание.** -g и -ggdb во многом схожи, однако, если интересно, вот небольшие отличия:

`-g` and `-ggdb` are similar with some *slight* differences, I read this [here](#):

`-g` produces debugging information in the OS's native format (stabs, COFF, XCOFF, or DWARF 2).

`-ggdb` produces debugging information specifically intended for gdb.

`-ggdb3` produces extra debugging information, for example: including macro definitions.

`-ggdb` by itself without specifying the level defaults to `-ggdb2` (i.e., gdb for level 2).

### 3. *Флаги предупреждений:*

- помогают выявить потенциальные ошибки и проблемы в программе.
- `-Wall` — включает общие предупреждения.
- `-Wextra` — включает дополнительные предупреждения.
- `-Werror` — превращает предупреждения в ошибки, останавливая компиляцию при их наличии.
- `-fsanitize=address` — указывает на сложные ошибки несоответствия типов, переполнения кучи, переполнения стека, ошибки выравнивания и т. п.
- `-fsanitize=thread` — указывает на конфликты потоков.
- `-fsanitize=undefined` — указывает на возможное неопределенное поведение.

### 4. *Флаги стандартов:*

- Позволяют указать версию языка, используемого при компиляции.
- `-std=c11` — стандарт C11 для компиляции на C.
- `-std=c++17` — стандарт C++17 для C++.

### 5. *Флаги связывания:*

- используются для связывания объектных файлов и библиотек.
- `-l<library>` — указывает компилятору подключить библиотеку (например, `-lgtest`).
- `-L<Path>` — указывает путь к пользовательской библиотеке.

Пример:

```
gcc -O2 -g -Wall -std=c11 my_program.c -o my_program
```

## Вопрос 61. Компилирование с использованием статических библиотек.

**Статическая библиотека** — это коллекция объектных файлов, которые присоединяются к программе во время компоновки.

На Windows у статических библиотек расширение **.lib**, на Unix/Linux расширение **.a**

### *Создание статической библиотеки на Windows в Visual Studio:*

Например создадим статическую библиотеку для подсчёта площадей фигур:

- **Создание проекта статической библиотеки в Visual Studio**

1. в строке меню выберите **файл создать Project**, чтобы открыть диалоговое окно **создание нового Project**.
  2. в верхней части диалогового окна задайте для параметра **Language** значение **C++**, задайте для параметра **Platform** значение **Windows** и задайте для параметра **Project тип** значение **Library**.
  3. В отфильтрованном списке типов проектов выберите пункт **Мастер классических приложений Windows**, а затем нажмите кнопку **Далее**.
  4. На странице **Настроить новый проект** введите *SquareLibrary* в поле **Имя проекта**. В поле **Имя решения** введите *SquareMath*. Нажмите кнопку **Создать**, чтобы открыть диалоговое окно **Проект классического приложения Windows**.
  5. В диалоговом окне **Проект классического приложения Windows** в разделе **Тип приложения** выберите **Статическая библиотека (.lib)**.
  6. В разделе **Дополнительные параметры** снимите флажок **Предварительно откомпилированный заголовок**, если он установлен. Установите флажок **Пустой проект**.
  7. Нажмите кнопку **ОК**, чтобы создать проект.
- Затем создайте несколько .cpp файлов и в них разместите определения функций.
  - В едином заголовочном файле разместите все прототипы созданных функций.
  - Запустите проект

В папке Debug вашего проекта должен появиться *SquareLibrary.lib* - это и есть ваша библиотека.

Чтобы подключить вашу библиотеку к проекту, необходимо переместит .lib в файл в папку Debug вашего проекта и в IDE с помощью команды "Добавить/Add" контекстного меню вашего проекта добавить библиотеку.

Вам осталось лишь прописать в необходимом месте в коде `#include` к вашей библиотеке.

### *Создание статической библиотеки на Linux/Unix системах:*

- Компиляция объектных файлов библиотеки:  
`gcc -c my_library.c -o my_library.o`
- Создание статической библиотеки (создается с помощью утилиты ar):  
`ar rcs mylibrary.a my_library.o`
- Компилирование программы с линковкой статической библиотеки:  
`gcc main.c -L. -lmylibrary -o a.out`

**Замечание.** -L. флаг связывания здесь указывает на то, что наша библиотека находится в текущем каталоге (при желании её можно разместить в другой директории и указать к ней путь через -L<Path>).

- Запуск:  
./a.out

**Замечание.** При желании проверьте, работает ли этот способ. Я вообще через CMake всегда билдил и не жаловался.

## Вопрос 62. Компилирование с использованием динамических библиотек.

**Динамическая библиотека** — коллекция скомпилированных файлов, которая загружается в память и связывается с программой во время выполнения.

- На Windows динамические библиотеки имеют расширение **.dll**, а на Unix-подобных системах — **.so**
- Особенностью динамической библиотеки является то, что загрузившему её процессу доступны только экспортируемые функции, а функции, предназначенные для «внутреннего мира» библиотеки не выгружаются, чтобы не замедлять скорость загрузки библиотеки.
- Для экспортирования функции из динамической библиотеки следует указывать ключевое слово `__declspec(dllexport)`, а также `extern "C"`, чтобы отключить перегрузку функций (иначе в название функции будут кодироваться типы аргументов функции, что усложнит процесс обращения к экспортируемым функциям во внешних файлах).
- В Windows для загрузки динамической библиотеки используется *дескриптор* (указатель на область памяти) `HINSTANCE` или `HMODULE`. Переменная `HINSTANCE load = LoadLibrary(L"ПУТЬ/К/БИБЛИОТЕКЕ");` (`load` — название переменной) является указателем на начало динамической библиотеки в памяти компьютера.
- Доступ к функциям динамической библиотеки получаем с помощью функции `GetProcAddress(load, "Название функции");` которая возвращает указатель типа `(void*)` на место начала функции в памяти компьютера.

**Создание динамической библиотеки на Linux:**

- Компиляция исходного файла в библиотеку:
  - `gcc -fPIC -shared -o mylibrary.so my_library.c`
  - `-fPIC` указывает компилятору генерировать код, который может быть использован в динамической библиотеке.
  - `-shared` указывает на создание динамической библиотеки.
- Компилирование программы с подключением динамической библиотеки:
  - `gcc -o a.out main.c -L. -lmylibrary`
  - `-L.` указывает компилятору искать библиотеку в текущем каталоге (при желании можно разместить библиотеку в другой директории и указать путь к ней).
  - `-lmylibrary` — линкуем динамическую библиотеку.
- Запуск программы:
  - Для запуска программы, использующей динамическую библиотеку, необходимо убедиться, что библиотека доступна в системном пути. Для этого устанавливается *переменная окружения* (хранит информацию о среде выполнения) `LD_LIBRARY_PATH`.
  - `export LD_LIBRARY_PATH=./LD_LIBRARY_PATH`
  - `./a.out`

**Замечание.** Чтобы освежить знания о создании динамических библиотек на Windows в Visual Studio, рекомендую пересмотреть гайд на Stepik.