Структуры и алгоритмы обработки данных

Оглавление

[Хороший стиль записи программ на C++ (на основе Google Code Style Guide) 2](#_Toc533508072)

[Шаблоны (templates) C++, шаблонные функции и шаблонные классы 5](#_Toc533508073)

[Перегрузка (overloading) функций и операторов в C++ 6](#_Toc533508074)

[Базовые типы данных и строки в С++ 8](#_Toc533508075)

[Класс std::vector<T>, основные свойства и методы. Циклы по элементам вектора 10](#_Toc533508076)

[Реализация матриц на основе std::vector<T>. Создание и обработка 11](#_Toc533508077)

[Стек и основные операции над ним. Реализация на основе std::vector<T> 12](#_Toc533508078)

[Ввод и вывод в C++, классы, операторы, форматирование. Консоль и файлы 13](#_Toc533508079)

[Нотация асимптотического роста для алгоритмов (О-нотация). Классификация алгоритмов по сложности 14](#_Toc533508080)

[Стековый алгоритм проверки сбалансированности скобочных выражений 15](#_Toc533508081)

[Динамическое программирование 16](#_Toc533508082)

[Поиск наибольшей возрастающей последовательности 17](#_Toc533508083)

[Быстрая сортировка 18](#_Toc533508084)

[Замена рекурсии на итерацию на примере быстрой сортировки 19](#_Toc533508085)

[Сортировка слиянием 20](#_Toc533508086)

[Поразрядная сортировка 21](#_Toc533508087)

[Алгоритм Кнута-Морриса-Пратта 22](#_Toc533508088)

[Алгоритм Бойера-Мура 23](#_Toc533508089)

[Поиск с возвратом (backtracking) 24](#_Toc533508090)

[Генерация перестановок 25](#_Toc533508091)

# Хороший стиль записи программ на C++ (на основе Google Code Style Guide)

Главное, что нужно понимать: современный код пишется не для компилятора (он разберётся и скомпилирует что угодно; или откажется компилировать), а для человека. На код должно быть приятно смотреть, его должно быть легко читать. Вы его пишете один раз, сохраняете, после чего его читают много раз, поэтому выгодно потратить при написании немного времени на приведение кода в порядок, чтобы в последствии сократить своё и чужое время на чтение. Простые правила ниже служат для улучшения визуального восприятия.

1. Вокруг всех бинарных операторов (=, ==, +, -, \*, /, >, << и др.) должны быть пробелы с обеих сторон. Исключением являются операторы ., ->, ::.

2. После запятой должен быть пробел.

3. Между закрывающейся круглой скобкой и открывающейся фигурной должен быть пробел.

4. Не жадничайте с пустыми строками. Вставляйте всегда пустые строки между определениями глобальных функций, классов, констант, typedef’ов, #include’ов, между объявлениями методов и функций, между реализациями функций, между объявлениями классов и реализациями функций и т.д.

5. Вставляйте пустые строки в код реализации функций, чтобы подчеркнуть разделение логических частей кода.

6. Не размещайте if, else, for, while и др. на одной строке со своим statement вот так:

*if (condition) statement;*

*else statement;*

*...*

*for (...) statement;*

Это, во-первых, ухудшает читаемость кода. Вы можете вообще один из statement’ов не заметить или ошибочно решить, что он относится к if’у:

*if (number % 2 == 0) std::cout << "Even\n"; even = true;*

А во-вторых, при отладке debugger’ом невозможно понять, выполнив команду “Step Over”, выполнилось или не выполнилось условие (или сколько итераций цикла прошло).

Также рекомендуется всегда обрамлять if, else, for, while фигурными скобками вот так:

*for (int index = 0; index < array.size(); ++index) {*

*statement1;*

*statement2;*

*...*

*}*

даже если внутри только один statement.

*if (number % 2 == 0) {*

*std::cout << "Even\n";*

*}*

Это более читаемо и безопасно. В варианте без скобок легко ошибиться, например, вот так:

*if (number % 2 == 0)*

*std::cout << "Even\n";*

*even = true;*

Легко подумать, что код even = true; тоже находится под if-ом.

Кроме того, в процессе разработки и использования языка C++ родились некоторые полезные правила (которые иногда прямо противоположны тому, что было принято в языке C):

1. using namespace std; так делать не рекомендуется, включать целый namespace, т.к. из-за этого может возникнуть конфликт имен. Вследствие чего могут возникнуть нетривиальные ошибки компиляции/линковки, а если не повезёт, то переменная из namespace может совпасть по названию с какой-то вашей переменной, про которую вы не помните ее область видимости, что приведет к еще более сложно находимым багам, хоть все и скомпилируется, но иногда вы будете использовать переменную, думая, что это ваша переменная, и в ней такое-то значение, а значение будет совсем другим. Если нужно использовать много раз std::vector, напишите using std::vector; если cout, то using std::cout; и т.д. Кроме того, включая namespace, вы сам принцип namespace’ов, разделяющих имена, нарушаете.

2. Не используйте массивы фиксированной длины int[], int\* используйте вместо них std::vector<int>.

3. Не используйте C-type строки char[] и char\* используйте вместо них std::string.

4. Не используйте ввод-вывод в стиле С через функции scanf, printf используйте вместо них операторы >> и << у std::cin и std::cout соответственно.

5. Если используется значение типа истина/ложь, то используйте тип bool, а не int.6. main должен заканчиваться return 0;

7. Вставляйте слово const везде, где только это возможно по смыслу. Если какая-то переменная, по сути, меняться в функции не должна, она должна быть const. Если метод класса не меняет при вызове содержимое класса, он должен быть const-методом. Таким образом вы обезопасите себя от многих глупых ошибок: они отловятся еще на этапе компиляции.

Если у вас из-за того, что вы где-то поставили в правильном месте const, не компилируется код, то const выполнил свою главную задачу. Тогда надо не его убирать, а найти и исправить проблему в другом месте: вы где-то еще забыли поставить const или изменяете переменную, которую не собирались изменять. Надо в этом разобраться, доставить const туда, где он еще нужен, а не удалять там, где он вам “мешает”.

8. Используйте везде в программе индексацию с нуля. Если какие-то входные или выходные данные в задаче используют индексацию с единицы, лучше в функции ввода, соответственно вывода, переведите индексацию из одной системы в другую, а везде внутри программы, помимо функций ввода и вывода пользуйтесь индексацией с нуля. Весь язык С++ так спроектирован, что индексация с нуля гораздо удобнее, а как только вы начинаете использовать индексацию с единицы, становится неудобно, появляются вычитания единицы из переменных по всему коду и т.д.

9. Не пользуйтесь макросами для определения констант. Макросы – это очень опасная и неудобная вещь. Их раскрывает специальный препроцессор, который начинает работать еще до компилятора C++, и он ничего не знает о самом языке. Все конструкции раскрываются буквально. В связи с этим есть множество возможных неочевидных побочных эффектов, а у компилятора нет возможности выполнить проверку типов, константность и т.д. Итак, неправильный вариант:

*#define MAX\_LENGTH 100000 // Wrong! Don’t use macros!*

Правильный вариант:

*const int MAX\_LENGTH = 100000; // Correct*9. Имейте в виду, что функция abs по стандарту принимает на вход int и возвращает int, а для взятия модуля вещественного числа (float, double) необходимо пользоваться функцией fabs.

Кроме того, не зависимо от используемого языка полезно придерживаться таких правил:

1. У каждой переменной должна быть одна-единственная явная цель. Никогда не создавайте переменных tmp, выполняющих несколько разных вспомогательных функций во всем коде. Используйте переменную только с одной целью. Переменные, в названии которых используется tmp или temp, почти всегда либо бессмысленные и ненужные, либо неправильно названы.2. Имена переменных должны быть длинными и понятными. Каждый раз, когда вы пишете одно-двух-буквенное название переменной или используете что-то вроде cur, должно возникать неприятное чувство. Единственное место, где можно позволить себе однобуквенные переменные, – в качестве счетчика в очень коротком for’е без вложенных циклов.

3. Объявляйте переменные как можно ближе к месту их первого использования. Старайтесь сразу же инициализировать переменные. Если переменная используется только внутри функции, она должна быть локальной для функции. Если только внутри цикла, она должна быть локальной для цикла. Никогда не делайте глобальных переменных. Локальные переменные блока предпочтительнее по сравнению с локальными переменными функции, локальные переменные функции – по сравнению с переменными-членами класса, а последние – по сравнению с глобальными переменными. Стремитесь сократить “время жизни” каждой переменной: чем меньше время жизни переменных, тем меньше переменных приходится одновременно держать в голове при чтении и написании кода. Исследования показывают, что человек может эффективно держать в памяти не более 5-7 переменных одновременно. Большее количество неизбежно приводит к ошибкам.

4. Разделяйте программу на ввод, решение и вывод, это делает ваш код более модульным. Способы ввода и вывода часто меняются. Записывайте вход в отдельные переменные и результат работы -в отдельные. Для их заполнения и вывода напишите отдельные функции. В частности, ваш код становится легче тестируемым, что является важным свойством.

Вообще это две принципиально разные области ответственности: ввод-вывод и преобразование данных. Не смешивайте в одном классе или функции несколько разных областей ответственности: один класс отвечает ровно за одну область. Иначе он разрастается, становится слишком сложным, а две разные области ответственности начинают быть слишком сильно связанными. Это плохо, потому что чем более независимы разные части программы, тем меньше поводов для ошибок и тем проще тестировать части программы по отдельности.

5. Пишите комментарии только по делу. В идеальном случае лучше обходиться вообще без них - ваш код прокомментирует сам себя. Конечно, так редко удаётся, поэтому комментарии к классам и функциям бывают полезными. Не нужно оправдывать плохое имя подробным комментарием.

# Шаблоны (templates) C++, шаблонные функции и шаблонные классы

Шаблоны (template) – средство языка C++предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию).

Шаблоны позволяют создавать параметризованные классы и функции. Параметром может быть любой тип или значение одного из допустимых типов.

Шаблон начинается с ключевого слова template (или по старинке слово class), за которым в угловых скобках следует список параметров. Затем следует объявление функции или класса.

*template <typename T>*

*T min(T a, T b) {*

*return a < b ? a : b;*

*}*

Такая функция может применяться для разных типов входных данных

*min(1, 2);*

*min('a', 'b');*

*min(string("abc"), string("cde"));*

Причём как правило в полной записи *min<double>(3.14159, 2.71828)* имя типа можно опускать, компилятор сам догадается и подставит его. В сложных случаях имя типа указывать можно и нужно.

Подобно функциям, классы тоже могут быть шаблонными, самым ярким примером является класс std::vector<T>, который может хранить вектор любого типа – от самых простых, до сколь угодно сложных (например, вектор векторов).

При использовании класса, как правило, тип-параметр опустить невозможно, потому что компилятор не имеет возможности догадаться.

*std::vector<double> Speed(3);*

Хотя шаблоны предоставляют краткую форму записи участка кода, на самом деле их использование не сокращает исполняемый код, так как для каждого набора параметров компилятор создаёт отдельный экземпляр функции или класса. Как следствие, исчезает возможность совместного использования скомпилированного кода в рамках разделяемых библиотек. Однако, этим недостатком в наше время как правило пренебрегают в пользу того увеличения производительности труда программиста.

Шаблонное метапрограммирование в С++ также страдает от множества ограничений, включая проблемы переносимости, отсутствие поддержки отладки или ввода/вывода в процессе инстанцирования шаблонов, длительное время компиляции, низкую читабельность кода, скудную диагностику ошибок и малопонятные сообщения об ошибках. В некоторых других языках программирования шаблоны реализованы, пожалуй, удачнее.

Однако, именно шаблоны C++ сделали этот вид программирования известным и популярным.

# Перегрузка (overloading) функций и операторов в C++

Термин «перегрузка» – калька английского слова overloading. Такой перевод появился в книгах по языкам программирования в первой половине 1990-х годов. В изданиях советского периода аналогичные механизмы назывались переопределением или повторным определением, перекрытием операций / функций.

Когда одинаковые по смыслу операции применяются к операндам различных типов, их вынужденно приходится называть по-разному. Невозможность применять для разных типов функции с одним именем приводит к необходимости выдумывать различные имена для одного и того же, что создаёт путаницу, а может и приводить к ошибкам. Например, в классическом языке Си существует два варианта стандартной библиотечной функции нахождения модуля числа: abs() и fabs() — первый предназначен для целого аргумента, второй — для вещественного. Такое положение, в сочетании со слабым контролем типов Си, может привести к трудно обнаруживаемой ошибке: если программист напишет в вычислении abs(x), где x — вещественная переменная, то некоторые компиляторы без предупреждений сгенерируют код, который будет преобразовывать x к целому путём отбрасывания дробной части и вычислять модуль от полученного целого числа.

Гораздо лучше давать нескольким функциям, выполняющим сходные действия, одинаковые имена при условии, что компилятор сможет различить их и правильно выбрать нужную функцию. Например, можно определить две функции

|  |  |
| --- | --- |
| *int min(int a, int b) {*  *return a > b ? b : a;*  *}* | *double min(double a, double b) {*  *return a > b ? b : a;*  *}* |

И тогда оба вызова *min(3,5)* и *min(1.2, 1.3)* будут корректно работать, каждый вызывая свою функцию.

Для того, чтобы перегрузка работала, две функции должны отличаться количеством или типом параметров. Не могут перегружаться функции, отличающиеся только типом возвращаемого значения.

Кроме того, перегружаемых функций не обязательно должно быть только две. Наоборот, как правило, можно написать множество функций с одним именем, но различающихся количеством и типом параметров, скажем min(int, int), min(int, int, int), min(int, int, int, int) и т.д, с одной стороны и аналогично min(double, double, double), min(float, float) и т.п.

Разумеется, писать большое количество однотипных функций – утомительно и может приводить к ошибкам, но можно воспользоваться механизмом шаблонов C++ и сгенерировать сразу множество функций, например:

|  |  |
| --- | --- |
| *Template <typename T>*  *T min(T a, T b) {*  *return a > b ? b : a;*  *}* | *Template <typename T>*  *T min(T a, T b, T c) {*  *return min(a, min(b, c));*  *}* |

Возможно даже написать умный шаблон, который сгенерирует функцию *min<T>(T,T,T…)* для любого количества параметров.

Аналогично функциям, в C++ могут перегружаться методы классов и операторы, например «+» (сложения) или «<<» (двоичный сдвиг). Благодаря этому можно в удобной форме записывать операции над любыми типами данных. Так, если мы определим тип «скорость», то можем удобно складывать скорости:

*using speed = std::vector<double>;*

*speed operator + (const speed& X, const speed& Y);*

*speed A, B, C;*

*C = A + B;*

Этот механизм широко используется для удобной записи операций ввода-вывода в C++. Для этой цели перегружаются операторы «<<» и «>>», которые вместо битовых сдвигов получают смысл вывода в поток и чтения из потока соответственно.

*std::ostream& operator << (std::ostream& output, const speed& data);*

*std::istream& operator >> (std::istream& output, speed& data);*

*std::cin >> A >> B;*

*std::cout << (A + B) << std::endl;*

# Базовые типы данных и строки в С++

В основном C++ наследует базовые типы из языка C.

Самым популярным типом является *int*, для хранения целых чисел. На самом деле это целое семейство типов, остальные типы получаются добавлением префиксов signed, unsigned, short, long. Например, *signed int = int, unsigned = unsigned int, long = signed long int* и т.п. Есть даже тип *long long (int)*.

Все эти типы отличаются между собой прежде всего размером: sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long). Как правило, тип short 16-битовый, int 32-битовый, long long 64-битовый, но это может отличаться в зависимости от компилятора и операционной системы.

Кроме того, они отличаются по тому, допустимы ли отрицательные числа (signed) или значения считаются не меньшими 0 (unsigned). Последние оказываются удобны (и чаще всего применяются) для нумерации элементов векторов и массивов.

Особняком стоит тип char, который тоже является целым, но как правило 8-битным. Он используется преимущественно не для расчётов, а для работы с текстами. При этом он тоже может быть и знаковым и беззнаковым.

В C++ добавлен отдельный тип *bool*. В С для этого использовался int, но в С++ рекомендуется явно писать bool, если переменная принимает именно логическое значение – для того, чтобы программа была понятнее. Кроме того, в некоторых случаях компилятор сможет построить более эффективную программу, если знает точный тип переменной, как например происходит для *std::vector<bool>*.

Для хранения вещественных чисел (с плавающей точкой) используются типы float и double (а также long double), которые также отличаются потребляемой памятью (как правило sizeof(float)==4, sizeof(double)==8, sizeof(long double)==10). В наше время, в подавляющем большинстве случае используется *double*, только если нет каких-то (исторических) причин применять другие типы.

В С любые типы могут соединяться в массивы, однако синтаксис массивов в С противоречив и часто вводит программистов в путаницу. По этой причине, в С++ стараются сократить использование традиционных массивов С. В общем случае вместо них как правило используются вектора *std::vector<T>*.

В частности, для хранения строк в С применялись так называемые ASCIIZ-строки, которые по сути дела представляют собой массив, в конце которого добавлен нулевой элемент (как символ конца строки). Их использование крайне неудобно, так как заставляет программиста всё время заниматься выделением и освобождением памяти. Кроме того, обработка длинных строк может быть неэффективна, так как всё время придётся вычислять их длину.

Вместо них в С++ используется библиотечный тип std::string, который эффективно (и незаметно для программиста) управляет памятью – строки при необходимости выделяются и после использования автоматически освобождаются. Кроме того, для строк реализованы часто используемые операции (сложения оператором «+»), поиска, выделения подстрок и т.п., а также определены операторы ввода-вывода «<<» и «>>» в стандартные потоки.

*#include <string>*

*std::string Text;*

*std::cout << “Введите строку: “;*

*std::cin >> Text;*

*std::cout << “Вы ввели “ << Text << std::endl;*

Длину строки можно узнать при помощи метода .size() или .length() (как и для вектора), причём за время O(1).

# Класс std::vector<T>, основные свойства и методы. Циклы по элементам вектора

Шаблонный класс std::vector<T> по сути предназначен в С++ для замены традиционных массивов С ввиду их серьёзных недостатков и неудобств для современных программистов:

1. Массив – это не полноценный объект, а просто ссылка на первый элемент
2. В большинстве случаев размер массива определить невозможно (а иногда – можно)
3. Массив сложно выделить динамически и почти невозможно изменить его размер

Как следствие, использование массивов в программах на С++ не рекомендуется.

В противовес этому, вектор из стандартной библиотеки лишен этих недостатков. Синтаксис его использования несколько отличается от массива, но в общем он интуитивно понятен и легко осваивается

|  |  |
| --- | --- |
| *int a[10];*  *a[0]=1;*  *printf(“%d”, a[1]);* | *#include <vector>*  *std::vector<int> a(10);*  *a[0]=1;*  *std::cout << a[1];* |

Вектор – это полноценный объект, для которого реализовано множество методов:

* V.size() – возвращает размер вектора
* V.resize(n) – изменяет размер вектора, при необходимости выделяя или освобождая память
* V.reserve(n) – размер вектора не изменяется, но мы даём подсказку компилятору, что он дорастёт до указанного размера, за счёт чего компилятор сможет эффективнее управлять выделением памяти
* V.clear() – очистить вектор
* V.empty() – проверить, что вектор пуст
* V.push\_back(x) – добавить элемент в конец вектора
* V.pop\_bak() – удалить элемент из конца вектора
* V.front() – первый элемент вектора (V[0])
* V.back() – последний элемент вектора (V[V.size()-1])

Удобно, что вектор скрывает выделение и освобождение памяти внутри себя и программисту больше не нужно этим заниматься вообще.

Кроме того, традиционные циклы в С часто подвержены ошибкам и опечаткам. Обход элементов вектора может быть записан гораздо короче и – главное! – надёжнее при помощи нового синтаксиса циклов

|  |  |  |
| --- | --- | --- |
| *int a[10]*  *for (int I = 0; i<10; ++i) {*  *a[i] = 7;*  *}* | *std::vector a(10);*  *for (auto& it : a) {*  *it = 7;*  *}* | *std::vector a(10);*  *for (auto it : a) {*  *std::cout << a << “ “;*  *}* |

Такой цикл проще пишется, лучше понимается, а иногда может даже быстрее исполняться.

# Реализация матриц на основе std::vector<T>. Создание и обработка

В общем случае, std::vector<T> – это одномерный вектор с элементами произвольного типа. С другой стороны, это очень общая структура данных, на основе которой можно строить более сложные структуры данных по мере потребности.

Если мы хотим каким-либо образом обрабатывать двумерные матрицы в программах на С++, это можно сделать разными способами.

С точки зрения быстродействия лучше «вытянуть» матрицу в вектор и хранить все её элементы подряд. То есть, если мы хотим завести матрицу размером 10\*20, её можно описать как

*std::vector<double> Matrix(10\*20);*

К сожалению, обращаться к отдельным элементам может быть неудобно, например элемент в третьей строке и пятом столбце будет *Matrix[(3-1)\*20 + 5 - 1]*. На практике придётся написать специальный класс (обёртку над вектором), который будет предоставлять такой доступ в удобной форме, например *Matrix.at(3, 5)*. В этом нет ничего невозможного, более того, при необходимости большой работы с матрицами именно так и делают.

Однако, если нам нужно на скорую руку обработать матрицу, это можно сделать и проще, без написания дополнительных классов.

Воспользуемся тем, что элемент вектора может быть любого типа, в частности, он сам может быть вектором. Тогда можно записать (для краткости и удобства) наш тип данных так:

*using Matrix = std::vector<std::vector<>double>;*

Обращение к элементам такой матрицы очень удобно и интуитивно понятно: *m[3][5]*.

Менее удобным становится выделение памяти под такую матрицу. В общем случае, компилятор никак не гарантирует нам, что отдельные вектора в составе матрицы имеют одинаковый размер, поэтому мы должны сами проследить за этим:

*using Matrix = std::vector<std::vector<>double>;*

*Matrix m(10);*

*for (auto& row : m) {*

*row.resize(20);*

*}*

После этого наш объект является полноценной матрицей 10\*20 и мы можем легко обращаться с ним. При желании мы можем даже при помощи такого же цикла изменить размер нашей матрицы.

Приятной особенностью С++ является то, что нам не нужно заботиться об освобождении памяти: когда матрица станет не нужна, все входящие в её состав вектора автоматически уничтожатся и занимаемая ими память станет доступна для дальнейшего использования. Делать этого вручную не надо.

# Стек и основные операции над ним. Реализация на основе std::vector<T>

Стек – очень важная структура данных, которая часто используется во многих алгоритмах.

Понятие ввёл Алан Тьюринг в 1946 году. Стек – абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).

Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно снять верхнюю.

Самым известным применением стека является реализация подпрограмм во всех современных языках программирования. Широко применяются стеки, также, например, во многих компиляторах для разбора и вычисления обычных выражений со скобками. Некоторые языки программирования и виртуальные машины также используют стек для обработки данных.

Стеки часто реализуют в виде однонаправленного списка, однако в С++ в качестве стека эффективно применяется стандартный вектор std::vector<T>.

Любой стек должен уметь выполнять такие операции:

* Поместить элемент на вершину стека
* Проверить, что стек не пуст
* Посмотреть элемент, содержащийся на вершине стека
* Удалить элемент с вершины стека, тем самым открыв доступ к предыдущему элементу в стеке

Если мы используем вектор, то эти операции выглядят так:

* V.push\_back(element)
* V.empty()
* V.back()
* V.pop\_back()

Потенциально стек может оказаться бесконечным, то есть в него потребуется поместить очень много элементов. Всякая реализация стека имеет свои пределы, которые зачастую могут быть меньше, чем вся доступная память. Например, стандартный стек вызова программ может быть ограничен размером 1 Мегабайт. При его переполнении произойдёт аварийное завершение программа.

Стек на основе std::vector<T> может при необходимости занимать очень большие объёмы памяти. Потенциально он может использовать под себя всю память, выделенную программе операционной системой (гигабайты и даже больше), поэтому он может обрабатывать очень большие объёмы данных без опасности аварийного завершения программы.

# Ввод и вывод в C++, классы, операторы, форматирование. Консоль и файлы

# Нотация асимптотического роста для алгоритмов (О-нотация). Классификация алгоритмов по сложности

# Стековый алгоритм проверки сбалансированности скобочных выражений

# Динамическое программирование

# Поиск наибольшей возрастающей последовательности

# Быстрая сортировка

# Замена рекурсии на итерацию на примере быстрой сортировки

# Сортировка слиянием

# Поразрядная сортировка

# Алгоритм Кнута-Морриса-Пратта

# Алгоритм Бойера-Мура

# Поиск с возвратом (backtracking)

# Генерация перестановок