Структуры и алгоритмы обработки данных

Оглавление

[Хороший стиль записи программ на C++ (на основе Google Code Style Guide) 2](#_Toc533598427)

[Шаблоны (templates) C++, шаблонные функции и шаблонные классы 5](#_Toc533598428)

[Перегрузка (overloading) функций и операторов в C++ 6](#_Toc533598429)

[Базовые типы данных и строки в С++ 8](#_Toc533598430)

[Класс std::vector<T>, основные свойства и методы. Циклы по элементам вектора 10](#_Toc533598431)

[Реализация матриц на основе std::vector<T>. Создание и обработка 11](#_Toc533598432)

[Стек и основные операции над ним. Реализация на основе std::vector<T> 12](#_Toc533598433)

[Ввод и вывод в C++, классы, операторы, форматирование. Консоль и файлы 13](#_Toc533598434)

[Нотация асимптотического роста для алгоритмов (О-нотация). Классификация алгоритмов по сложности 15](#_Toc533598435)

[Стековый алгоритм проверки сбалансированности скобочных выражений 17](#_Toc533598436)

[Динамическое программирование 18](#_Toc533598437)

[Поиск наибольшей возрастающей последовательности 20](#_Toc533598438)

[Быстрая сортировка 21](#_Toc533598439)

[Замена рекурсии на итерацию на примере быстрой сортировки 22](#_Toc533598440)

[Сортировка слиянием 23](#_Toc533598441)

[Поразрядная сортировка 24](#_Toc533598442)

[Алгоритм Кнута-Морриса-Пратта 25](#_Toc533598443)

[Алгоритм Бойера-Мура 26](#_Toc533598444)

[Поиск с возвратом (backtracking) 28](#_Toc533598445)

[Генерация перестановок 30](#_Toc533598446)

[Рекомендуемая литература 32](#_Toc533598447)

# 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

В С для операций ввода-вывода приходится использовать целое множество функций, таких как *printf(format, …)* для вывода на консоль и *fprintf(FILE, format, …)* для вывода в файлы.

При разработке С++ была придумана другая форма записи, более удобная и гибкая. Для неё не требуется заранее знать, куда мы будем выводить и какие именно объекты. И тот и другой список могут при необходимости расширяться.

Основу записи представляют два оператора «<<» и «>>». Изначально это операторы двоичного сдвига, но в контексте ввода-вывода они представляют собой оператор вывода и ввода соответственно. Например:

*#include <iostream>*

*std::cout << “Введите число:”;*

*int x;*

*std::cin >> x;*

*std::cout << x << “ + 1 = “ << x + 1 << std::endl;*

Поскольку это оператор, он может быть легко перекрыт и написан для любого нового типа объектов (как потока ввода-вывода, так и выводимого объекта).

Слева от оператора располагается поток ввода, который может реализовывать вывод куда угодно – в консоль, в файл или по сети Интернет. Самыми распространёнными потоками являются стандартные потоки ввода-вывода

* std::cin (стандартный ввод)
* std::cout (стандартный выход)
* std::cerr (поток сообщений об ошибках)

Если же необходимо выводить в файл, потребуется создать соответствующий поток ввода-вывода, что легко сделать так:

*#include <fstream>*

*std::ifstream input(“in.txt”);*

*std::ofstream out (“out.txt”);*

*int x;*

*input >> x;*

*output << x << “ + 1 = “ << x + 1 << std::endl;*

В случае, если кто-то напишет новый тип потока ввода-вывода (например, выводящий во всплывающее окно в трее), можно сразу начать выводить в него, не переписывая всю программу.

Точно так же, для любого нового типа данных мы можем реализовать операторы «<<» и / или «>>» и выводить их куда угодно. По соглашению операторы ввода-вывода должны возвращать переданный в них поток, чтобы вызовы операций ввода-вывода можно было сцеплять.

*#include <iostream>*

*#include <vector>*

*template <typename T>*

*std::istream& operator >>(std::istream& is, std::vector<T>& v) {*

*unsigned len;*

*is >> len;*

*v.resize(len);*

*for (auto& it : v) {*

*is >> it;*

*}*

*return is;*

*}*

*template <typename T>*

*std::ostream& operator <<(std::ostream& os, const std::vector<T>& v) {*

*auto first = true;*

*for (auto& it : v) {*

*if (!first) {*

*os << " ";*

*}*

*first = false;*

*os << it;*

*}*

*return os;*

*}*

При необходимости форматировать вывод в потоки, могут применяться специальные объекты – манипуляторы, такие как

* std::setw(n) – задаёт количество выводимых символов
* std::setf(n) – задаёт количество символов после запятой в выводимом числе
* std::left, std::right – задаёт выравнивание
* и много других

А также стоящий немного особняком манипулятор *std::endl*, который переводит строку (для чего в C использовалось *\n*).

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

Для любого алгоритма можно оценить время его работы и количество потребляемой памяти в виде некоторой функции от размера входных данных. Как правило, чем больше данных нужно обработать, тем больше времени и памяти для этого нужно.

На практике точная оценка как правило не важна и достаточно узнать самое общее поведение, причём при большом количестве данных. Для этого была разработана так называемая О-нотация, которая учитывает лишь самую существенную компоненту роста, отбрасывая второстепенные.

Например, в функции f (n) = 2n2 + n – 5 при достаточно больших n компонента n2 будет значительно превосходить остальные слагаемые, и поэтому характерное поведение этой функции определяется именно этой компонентой. Остальные компоненты можно отбросить и условно записать, что данная функция имеет оценку поведения (в смысле скорости роста ее значений) вида О(n2).

Аналогично, для функции f (n) = 2n + 3n3 – 10 начиная с некоторого n первое слагаемое будет превосходить второе и поэтому данную функцию можно описать оценкой О(2n).

Сложность алгоритма зачастую оценивается путём подсчёта числа элементарных операций, осуществляемых алгоритмом, где элементарная операция занимает для выполнения фиксированное время. Тогда полное время выполнения и число элементарных операций, выполненных алгоритмом, отличаются максимум на постоянный множитель.

Самые популярные классы сложности:

* O(1) – констатное (постоянное) время / память
* O(n) – линейное время
* O(n log n) – линейно-логарифмическое время
* O(n2) – квадратичное время
* O(n3), O(n4)… – полиномиальное время
* O(2n), O(en), O(10n)… – экспоненциальное время
* O(n!) – факториальное время
* И много других

С практической точки зрения оказывается очень важно понимать оценку времени и памяти для выбранного алгоритма. При небольших объёмах данных могут применяться любые алгоритмы, однако с ростом объёмов многие становятся непрактичны.

Так, например, малые массивы (сотни и тысячи элементов) вполне можно сортировать «наивными» алгоритмами (вроде сортировки пузырьком), однако с ростом длины массива, время сортировки начинает увеличиваться (по закону O(n2), то есть по квадрату) и приходится применять более сложные алгоритмы, например Быструю сортировку или Сортировку слиянием, потому что они работают за время O(n log n).

На практике можно даже говорить о том, что уже квадратичные алгоритмы оказываются зачастую неприменимы при современных объёмах данных, в то время как линейно-логарифмические широко применяются и хорошо работают.

Хотя в некоторых областях встречаются, например, алгоритмы, работающие за кубическое время (и доказано, что быстрее сделать работу невозможно). Их можно применять, если мы гарантируем, что объём данных не слишком велик.

В теоретических исследованиях, наоборот, исполнимыми считаются алгоритмы с полиномиальной оценкой, а слишком долгими – экспоненциальные (и факториальные). Действительно, они как правило не могут быть решены всей современной вычислительной мощью человечества уже при n > 100.

Таким образом,

1. При выборе однотипных алгоритмов предпочтение (при прочих равных условиях) следует отдавать алгоритмам с наименьшей скоростью роста трудоемкости, поскольку они позволят за одно и то же время решить задачи с большей размерностью
2. Если заранее известно, что размерность решаемых задач невелика, , имеет смысл рассмотреть возможность использования алгоритмов не с самой лучшей оценкой, поскольку при малых n «лучшие» алгоритмы могут вести себя хуже, чем «плохие»
3. Алгоритмы класса О(2n) и О(n!) следует использовать с большой осторожностью, учитывая катастрофический рост их трудоемкости уже при n>100. Например, если число базовых операций определяется соотношением 2n, то при n=100 это число будет примерно равно 1030, и если одна базовая операция выполняется за 1 микросекунду, то это потребует около 1024 секунд, т.е. порядка 1016 лет. К сожалению, задачи с подобной трудоемкостью довольно часто встречаются на практике и их точное решение пока невозможно даже на сверхбыстрых суперкомпьютерах!

Кроме того, в некоторых случаях можно делать раздельные оценки для времени выполнения программы:

* В среднем
* В лучшем случае
* В худшем случае

Так например, быстрая сортировка работает как правило за O(n log n), но при некоторых неудачных входных данных деградирует до O(n2). В противоположность ей, сортировка слиянием всегда работает за O(n log n), но как правило, её время работы несколько больше. С другой стороны, грамотно написанная сортировка пузырьком в среднем и худшем случае работает за O(n2), но иногда (для уже отсортированных данных) сработает за O(n).

Подобные сведения также могут в некоторых случаях повлиять на выбор алгоритма для решения конкретной задачи.

# Стековый алгоритм проверки сбалансированности скобочных выражений

Правильной скобочной последовательностью называется строка, состоящая только из скобок, в которой все скобки можно разбить на пары таким образом, что:

* в каждой паре есть левая и правая скобка, причем левая скобка расположена левее правой;
* для любых двух пар скобок либо одна из них полностью внутри другой пары, либо промежутки между скобками в парах не пересекаются
* в паре с круглой скобкой может быть только круглая скобка, с квадратной - квадратная, с фигурной - фигурная, с треугольной - треугольная

Примеры:

* Правильные последовательности: [{(())}({})], []{}(), {}, (), [], <>
* Неправильные последовательности: [{(})], ()]{}, >

Для того, чтобы определить, является ли заданная строка правильным скобочным выражением, можно применять разные методы, один из простейших и популярнейших – использовать стек для хранения уже разобранных, но ещё не обработанных до конца скобок:

* Просматриваем символы в строке слева направо
* Для символов открывающихся скобок ([{< кладём в стек соответствующую закрывающуюся скобку )]}>
* Для символов закрывающихся скобок )]}> - проверяем, что стек не пуст и на вершине лежит такой же символ и снимаем его. Иначе ошибка.
* Для всех остальных символов выдаём ошибку
* После просмотра всей строки, проверяем, что стек пуст

Например, при разборе последовательности «[{(})]» мы последовательно положим в стек скобки «]», «}» и «)» и после этого встретив «}», что не равно содержимому вершины стека, завершаем выполнение алгоритма с ошибкой.

Сложность алгоритма:

* Время O(n)
* Память O(n)

где n – длина исходной строки.

# Динамическое программирование

Динамическое программирование – способ решения сложных задач путём разбиения их на более простые подзадачи. Он применим к задачам, выглядящим как набор перекрывающихся подзадач, сложность которых чуть меньше исходной. В этом случае время вычислений, по сравнению с «наивными» методами, можно значительно сократить.

Словосочетание «динамическое программирование» впервые было использовано в 1940-х годах Р. Беллманом для описания процесса нахождения решения задачи, где ответ на одну задачу может быть получен только после решения задачи, «предшествующей» ей. В 1953 г. он уточнил это определение до современного.

Слово «программирование» в словосочетании «динамическое программирование» в действительности к «традиционному» программированию (написанию кода) почти никакого отношения не имеет и имеет смысл как в словосочетании «математическое программирование» или «линейное программирование», которое является синонимом слова «оптимизация». Поэтому слово «программа» в данном контексте скорее означает оптимальную последовательность действий для получения решения задачи. К примеру, определенное расписание событий на выставке иногда называют программой.

Можно дать неформальное определение: Динамическое программирование — это когда у нас есть задача, которую непонятно как решать, и мы разбиваем ее на меньшие задачи, которые тоже непонятно как решать.

В любом случае, чтобы решать задачу методом динамического программирования, нужно:

* Разбить задачу на несколько однотипных подзадач
* Научиться находить решение одних подзадач, зная решение других
* Выделить самые простые подзадачи, решение которых понятно
* Придумать порядок, в котором решать подзадачи

В смысле последнего пункта (порядка обхода подзадач) алгоритмы динамического программирования естественным образом делятся на:

* Линейные (одномерные)
* Матричные (двумерные)
* Ветвящиеся (обход дерева
* Прочие

Также в процессе решения методом динамического программирования очень часто применяется приём мемоизации, то есть запоминания результатов решения промежуточных подзадач. Иногда ничего больше добавлять к нему и не требуется.

Не все задачи могут быть решены методом динамического программирования, но когда это удаётся, сложность решения задачи как правило резко падает с экспоненциального времени для «наивного» решения до квадратичного (как правило, но не обязательно) времени для решения методом динамического программирования.

Например, если рассмотреть классическую задачу вычисления чисел Фибоначчи, то она может быть решена «по определению»

*int fib(int n) {*

*return n <= 1 ? n : fib(n - 1) + fib(n - 2);*

*}*

за время O(1.6n) и память O(n). При этом фактически происходит обход пространства состояний задач по дереву, то есть каждое fib(2), fib(3), fib(4)… считается очень много раз.

Если же использовать мемоизацию, то есть запоминать результаты вычислений

*int fib(int n) {*

*if (n <= 1) {*

*return n;*

*}*

*static std::vector<int> memS;*

*if (memS.size() < n + 1) {*

*memS.resize(n+1);*

*}*

*if (memS[n]) {*

*return memS[n];*

*}*

*return memS[n] = fib(n - 1) + fib(n - 2);*

*}*

то оценка алгоритма становится O(n) как по времени, так и по памяти. Впрочем, это пессимистичная оценка, во многих случаях этот код работает даже за константное время O(1).

Наконец, если поменять направление обхода и двигаться от 1 в сторону увеличения, то можно записать такой алгоритм:

*int fib(int n) {*

*int result = 0;*

*int next = 1;*

*for(int i = 0; i < n; ++i) {*

*auto tmp = next;*

*next += result;*

*result = tmp;*

*}*

*return result;*

*}*

Где время работы O(n), а потребление памяти констатное O(1).

# Поиск наибольшей возрастающей последовательности

Это классическая задача, решаемая методом динамического программирования. Её решение является шагом в более сложных алгоритмах.

Дана последовательность чисел, необходимо найти её подпоследовательность, все числа в которой возрастают, причём из всех таких подпоследовательностей нужно найти самую длинную. Элементы ответа не обязаны стоять подряд в исходной последовательности.

Например, для исходной последовательности 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15 ответом будет последовательность 0, 2, 6, 9, 11, 15 длины 6.

Существуют разновидности задачи, например:

* Найти не саму последовательность, а только её длину
* Найти самую «левую» (или «правую») из всех таких последовательностей

Существует также множество алгоритмов решения этой задачи. Рассмотрим классическое решение методом динамического программирования.

Будем искать для каждого элемента исходной строки самую длинную увеличивающуюся подпоследовательность, заканчивающуюся в нём. Если эта задача решена для всех элементов левее некоторого, то и для него она может быть легко получена простым перебором всех ответов и попыткой присоединить к ним этот элемент к концу.

На практике нам даже не нужно запоминать все последовательности для каждого элемента, достаточно запоминать только наибольшую длину увеличивающейся подпоследовательности, завершающейся в каждом элементе. Таким образом, если xi – исходная последовательность, а di – заполняемый массив длин, то

* Просматриваем исходную последовательность слева направо
* Для каждого i находим все xj < xi, j < i
* Из всех таких j выбираем di = 1 + max dj
* Если список j пуст, то xi = 1
* По окончании обхода находим наибольшее di, это и будет длина искомой последовательности

Если нам нужно найти не только длину, но и саму последовательность, можно либо пройти назад от элемента с максимальным di (на каждом шаге ищем такое j, что j < i, dj = di – 1, xj < xi), либо вместе с заполнением di заполняем другой массив pi, в котором для каждого xi указан предпоследний элемент в подподследовательности, кончающейся в xi.

Вот результат выполнения этого алгоритма для указанной последовательности

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| xi | 0 | 8 | 4 | 12 | 2 | 10 | 6 | 14 | 1 | 9 | 5 | 13 | 3 | 11 | 7 | 15 |
| di | **1** | 2 | **2** | 3 | 2 | 3 | **3** | 4 | 2 | **4** | 3 | 4 | 3 | **5** | 4 | **6** |
| Pi | - | 0 | 0 | 1 | 0 | 1 | 2 | 3 | 0 | 6 | 2 | 3 | 4 | 9 | 6 | 13 |

Сложность алгоритма:

* Время O(n2)
* Память O(n)

# Быстрая сортировка

Классический алгоритм сортировки произвольной последовательности за наилучшее возможное время, открытый Хоаром в 1960 году.

Алгоритм принципиально рекурсивный, общая его схема такова:

* Выбрать элемент из массива. Назовём его *опорным*.
* Разбиение: перераспределение элементов в массиве таким образом, что элементы меньше опорного помещаются перед ним, а больше или равные – после.
* Рекурсивно применить первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один элемент или отсутствуют элементы.

Возможны разные способы выбора опорного элемента:

* Взять первый или последний элемент массива
* Взять средний элемент массива
* Более сложные схемы и комбинации
* Взять случайный элемент массива (рандомизированная быстрая сортировка)

В идеале лучше всего брать «средний» элемент массива, то есть больший и меньший примерно половины элементов массива, но найти такой на практике сложно, поэтому применяются разные схемы выбора – от простейших до сложных.

Возможны также разные способы разбиения массива на две части на втором шаге:

* Наивный, когда заводятся два дополнительных массива и элементы раскладываются по ним; очень прост в реализации, но на практике не используется
* Разбиение Ломуто: проходим по массиву слева направо и переставляем при необходимости текущий элемент с элементом, находящимся на границе; этот метод сравнительно легко программируется, но на практике используется редко, а больше в учебных целях
* Разбиение Хоара: просматриваем массив с двух сторон, и если найден элемент в начале, который больше опорного, а в конце меньший опорного, то переставляем их местами; требует тщательного программирования, но зато чаще всего применяется на практике

Сложность алгоритма:

* Среднее время O(n log n)
* Худшее время O(n2)
* Память O(n)

К сожалению, для некоторых исходных массивов время работы алгоритма быстрой сортировки деградирует до квадратичного. Это происходит, например, если исходный массив (почти) отсортирован, а опорным элементом берётся первый или последний элемент. Тогда на всех этапах массив разбивается наиболее несимметрично (N+1), приходится делать O(n) разбиений и время получается O(n2).

Простейший метод борьбы с этим – выбирать в качестве опорного элемента средний, тогда на отсортированных массивах потери производительности не будет, но всё же будут некоторые массивы, которые будут сортироваться долго. Самым надёжным общим решением является рандомизированная быстрая сортировка, когда опорный элемент берётся из случайной позиции.

# Замена рекурсии на итерацию на примере быстрой сортировки

Очень многие алгоритмы являются рекурсивными, то есть содержат процедуры, которые вызывают себя или друг друга много раз, например расчёт чисел Фибоначчи или быстрая сортировка. Таким образом удаётся элегантно выразить довольно сложные алгоритмы. Недостатком этого подхода является то, что при большом объёме исходных данных может переполниться стек возвратов программы и она будет аварийно завершена.

Для борьбы с этим применяется общий приём замены рекурсии на итерации (цикл). Иногда (как при расчёте факториала или чисел Фибоначчи) удаётся переделать исходный алгоритм так, чтобы исключить рекурсию и использовать явный цикл.

Но во многих случаях сделать так не удаётся. Тогда можно воспользоваться общим приёмом – явно завести стек данных, в который складывать параметры рекурсивных вызовов. На месте рекурсивного вызова пишется операция пополнения стека текущими параметрами, а в цикле происходит извлечение параметров из вершины стека, обработка (которая, как правило, добавляет на стек другие параметры вызова – однократно или многократно). Цикл завершается по исчерпании стека.

На примере быстрой сортировки хорошо видно, как это делается.

|  |  |
| --- | --- |
| void qsort(std::vector<int>& data) {  qsort(data, 0, data.size() - 1);  }  void qsort(std::vector<int>& data,  unsigned left, unsigned right) {  if (left >= right) return;  auto border = partition(data,  left,  right,  data[(left + right) / 2]);  qsort(data, left, border);  qsort(data, border + 1, right);  } | struct StackFrame {  size\_t left;  size\_t right;  };  void qsort(std::vector<int>& data) {  std::vector<StackFrame> stack;  stack.push\_back({0, data.size() - 1});  while (!stack.empty()) {  auto range = stack.top();  stack.pop\_back();  if (range.left >= range.right) continue;  auto border = partition(data,  range.left,  range.right,  data[(range.left + range.right) / 2]);  stack.push\_back({range.left, border});  stack.push\_back({border+1, range.right});  }  } |

Время работы при этом не меняется (по крайней мере, оценка; множитель может и поменяться), но как правило удаётся сэкономить некоторое количество памяти за счёт того, что в стек добавляются только необходимые данные, так как компилятор может помещать в стек вызовов много разных вспомогательных данных.

Главное же достоинство этого процесса состоит в том, что теперь могут обрабатываться данные значительно большего объёма, которые переполнили бы стек вызовов (выделяемый программе операционной системой), но могут быть размещены в полной памяти.

# Сортировка слиянием

Это альтернативная реализация сортировки, которая пишется чуть проще, чем быстрая сортировка, работает чуть медленнее, но зато всегда работает за линейно-логарифмическое время на любых данных, то есть не деградирует.

В основе лежит та же идея разделения массива на две части (что и в быстрой сортировке), но разделение в данном случае очень простое – массив всегда делится ровно пополам.

После этого две части независимо сортируются и после этого сливаются. Сам процесс слияния – единственное, что требует программирования – и дал название всему алгоритму.

К сожалению, процесс слияния не удаётся реализовать без дополнительной памяти. Возникает задача минимизации её выделения. Ценой минимальных ухищрений удаётся свести её к однократному выделению копии исходного массива.

После этого процесс слияния выполняется просто: мы просматриваем оба массива слева направо и минимальный из двух элементов переносим во временный буфер. По окончании слияния содержимое буфера возвращается в исходный массив.

Кроме того, ввиду того что разделение носит регулярный и простой характер, удаётся легко преобразовать данный алгоритм из рекурсивного в итерационный. Для этого достаточно сливать подмассивы не с самого длинного, а наоборот, с самых коротких, длины 1.

То есть, сначала сливаем соседние пары элементов, образуя отсортированные кусочки длины 2. Потом сливаем соседние двойки, образуя отсортированные четвёрки и так далее, на каждом шаге длина отсортированных фрагментов удваивается, образуя степени двойки. Процесс завершается, когда длина очередного фрагмента превосходит длину исходного массива.

По окончании нужно освободить память, выделенную для слияния, но в С++ это происходит автоматически.

Сложность алгоритма:

* Время O(n log n)
* Память O(n)

# Поразрядная сортировка

Главное достоинство поразрядной сортировки – время работы, оно оценивается линейно, то есть лучше теоретического предела. Такой результат достигается за счёт того, что мы сортируем не произвольные объекты, поэтому их можно сравнивать не просто попарно. Этим методом можно сортировать любые конечные последовательности некоторых символов, в том числе – целые числа. А вот, например, вещественные числа так сортировать не получится.

В качестве одного шага поразрядной сортировки применяется так называемая «сортировка подсчётом». Она применима только когда сортируемые объекты можно разбить на небольшое количество классов (например, буквы английского или русского алфавита, или цифры). Тогда можно за один проход подсчитать количество объектов каждого класса, а за второй – расположить их друг за другом, сначала все нули, потом все единицы и т.д.

Очевидно, сортировка подсчётом делается за два прохода, каждый из которых требует линейное время. Также требуется дополнительная память, пропорциональная количеству значений (то есть 10 для цифр и 33 для русских букв или 256 для произвольных символов).

Вторым важным достоинством сортировки подсчётом является то, что она стабильна (или может быть реализована стабильно), то есть сохраняет взаимный порядок одинаковых элементов.

Поразрядная сортировка многократно применяет поразрядную сортировку к объектам, представляющим собой конечные цепочки элементов. Прежде всего это строки, а также целые числа.

Оказывается, что удобнее всего сделать поразрядную сортировку сначала по младшим разрядам, потом по следующим и так далее, на последнем шаге отсортировав объекты подсчётом по старшим разрядам. При этом автоматически они окажутся правильно отсортированы.

При сортировке чисел удобно выделять текущий разряд и остающиеся для будущей сортировки цифры при помощи операции целочисленного деления на 10 с остатком.

Но ещё удобнее и быстрее оказывается использовать для сортировки целых чисел поразрядную сортировку в двоичной системе счисления с использованием битовых операций сдвига влево, побитового И / ИЛИ.

Сложность алгоритма:

* Время O(n)
* Память O(n+Σ)

Где Σ- размер используемого алфавита (количество классов символов; при двоичной поразрядной сортировке это 2)

# Алгоритм Кнута-Морриса-Пратта

Алгоритм Кнута — Морриса — Пратта (КМП-алгоритм) — эффективный алгоритм, осуществляющий поиск подстроки в строке. Время работы алгоритма линейно зависит от объёма входных данных, то есть разработать асимптотически более эффективный алгоритм невозможно.

Алгоритм был разработан Д. Кнутом и В. Праттом и, независимо от них, Д. Моррисом. Результаты своей работы они опубликовали совместно в 1977 году.

Даны образец (строка) P и строка T. Требуется определить индекс, начиная с которого образец P содержится в строке T. Если P не содержится в T — вернуть индекс, который не может быть интерпретирован как позиция в строке (например, отрицательное число).

Алгоритм состоит из двух этапов, на первом вычисляется так называемая префиксная функция от образца P.

Префикс-функция строки – это наибольшая длина префикса строки, совпадающего с суффиксом, кончающимся в каждом символе строки. Значение префикс-функции может быть 0 (и всегда равно 0 для первого символа в строке) и всегда меньше длины строки. Пример расчёта префикс-функции:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | A | C | A | B | A |
| 0 | 0 | 1 | 0 | 1 | 2 | 3 |

Оказывается, что префикс-функция может быть вычислена за линейное время методом динамического программирования. Для этого надо двигаться по строке слева направо и рассчитывать префикс-функцию очередного символа на основе предыдущих.

Пусть значение префикс-функции fi = k и при этом оказывается, что Pi+1 = Pk+1. Тогда fi+1 = k+1 и расчёт для Pi+1 завершен. Иначе пробуем укорачивать суффикс, то есть взять k = fk и повторить процесс. Если мы приходим к k = 0, то и fi+1 = 0.

Хотя процесс расчёта префикс-функции содержит вложенный цикл, можно показать, что он выполняется за линейное время O(|P|).

После того, как мы знаем префикс-функцию шаблона, мы можем искать его вхождения (все или первое в строку). Для этого мы просматриваем строку слева направо и делаем расчёт, похожий на тот, которым мы считали префикс-функцию. Для каждого символа строки Si мы вычисляем k – длину префикса шаблона P, который можно приложить так, чтобы он заканчивался в Si.

Теперь, если Si+1 = Pk+1, то Si+1 = k + 1 и переходим к следующему символу в строке. Если же нет, то укорачиваем k = fk и повторяем сравнение. При этом может быть достигнуто значение k = 0.

По дороге, если мы получили k = |P| (длине шаблона), то текущий символ означает конец полного вхождения шаблона P в строку S, это ответ (или один из ответов).

Аналогично несмотря на то, что в алгоритме имеется вложенный цикл, время работы оказывается линейным

Сложность алгоритма:

* Время O(P+T)
* Память O(P+T)

# Алгоритм Бойера-Мура

Алгоритм поиска строки Бойера — Мура считается наиболее быстрым среди алгоритмов общего назначения, предназначенных для поиска подстроки в строке. Был разработан Робертом Бойером и Джеем Муром в 1977 году.

Преимущество этого алгоритма в том, что ценой некоторого количества предварительных вычислений над шаблоном (но не над строкой, в которой ведётся поиск) шаблон сравнивается с исходным текстом не во всех позициях — часть проверок пропускаются как заведомо не дающие результата.

Алгоритм основан на трёх идеях.

1. Сканирование слева направо, сравнение справа налево. Совмещается начало текста (строки) и шаблона, проверка начинается с последнего символа шаблона. Если символы совпадают, производится сравнение предпоследнего символа шаблона и т. д. Если все символы шаблона совпали с наложенными символами строки, значит, подстрока найдена, и выполняется поиск следующего вхождения подстроки.

Если же какой-то символ шаблона не совпадает с соответствующим символом строки, шаблон сдвигается на несколько символов вправо, и проверка снова начинается с последнего символа.

Эти «несколько», упомянутые в предыдущем абзаце, вычисляются по двум эвристикам.

2. Эвристика стоп-символа.

Предположим, что мы производим поиск слова «колокол». Первая же буква не совпала — «к» (назовём эту букву *стоп-символом*). Тогда можно сдвинуть шаблон вправо до последней его буквы «к».

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Строка | \* | \* | \* | \* | \* | \* | К | \* | \* |
| Шаблон | К | О | Л | О | К | О | Л |  |  |
| Далее |  |  | К | О | Л | О | К | О | Л |

Если стоп-символа в шаблоне вообще нет, шаблон смещается за этот стоп-символ.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Строка | \* | \* | \* | \* | \* | А | Л | \* | \* | \* | \* | \* | \* |
| Шаблон | К | О | Л | О | К | О | Л |  |  |  |  |  |  |
| Далее |  |  |  |  |  |  | К | О | Л | О | К | О | Л |

В данном случае стоп-символ — «а», и шаблон сдвигается так, чтобы он оказался прямо за этой буквой.

Для того, чтобы быстро вычислять сдвиги, заранее для всех символов в шаблоне заполняется таблица стоп-символов.

Если стоп-символ «к» оказался за другой буквой «к», эвристика стоп-символа не работает. В таких ситуациях выручает третья идея алгоритма Бойера — Мура:

3. Эвристика совпавшего суффикса. Неформально, если при чтении шаблона справа налево совпал суффикс S, а символ b, стоящий перед S в шаблоне (т. е. шаблон имеет вид PbS), не совпал, то эвристика совпавшего суффикса сдвигает шаблон на наименьшее число позиций вправо так, чтобы строка S совпала с шаблоном, а символ, предшествующий в шаблоне данному совпадению S, отличался бы от b, если такой символ вообще есть.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Строка | \* | \* | \* | \* | \* | \* | Р | К | А | \* | \* | \* | \* |  |  |
| Шаблон | С | К | А | Л | К | А | Л | К | А |  |  |  |  |  |  |
| Далее |  |  |  |  |  |  | С | К | А | Л | К | А | Л | К | А |

В данном случае совпал суффикс «ка», и шаблон сдвигается вправо до ближайшего «ка», перед которым нет буквы «л».

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Строка | \* | \* | Т | О | К | О | Л | \* | \* | \* | \* |
| Шаблон | К | О | Л | О | К | О | Л |  |  |  |  |
| Далее |  |  |  |  | К | О | Л | О | К | О | Л |

В данном случае совпал суффикс «окол», и шаблон сдвигается вправо до ближайшего «окол», перед которым нет буквы «л». Если подстроки «окол» в шаблоне больше нет, но он начинается на «кол», сдвигается до «кол», и т. д.

Аналогично таблице стоп-символов, заранее для всех символов в шаблоне заполняется таблица суффиксов (отдалённо напоминающая префикс-функцию в алгоритме КМП).

Алгоритм Бойера — Мура на «хороших» данных очень быстр, а вероятность появления «плохих» данных крайне мала. Поэтому он оптимален в большинстве случаев, когда нет возможности провести предварительную обработку текста, в котором проводится поиск. Разве что на коротких текстах выигрыш не оправдает предварительных вычислений.

Сложность алгоритма:

* Время O(P+T+Σ)
* Память O(P+T)

где Σ – количество видов символов в шаблоне P.

# Поиск с возвратом (backtracking)

Поиск с возвратом, бэктрекинг (англ. backtracking) — общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве М. Как правило позволяет решать задачи, в которых ставятся вопросы типа: «Перечислите все возможные варианты …», «Сколько существует способов …», «Есть ли способ …», «Существует ли объект…» и т. п.

Термин backtracking был введен в 1950 году американским математиком Дерриком Генри Лемером.

Решение задачи методом поиска с возвратом сводится к последовательному расширению частичного решения. Если на очередном шаге такое расширение провести не удается, то возвращаются к более короткому частичному решению и продолжают поиск дальше. Данный алгоритм позволяет найти все решения поставленной задачи, если они существуют. Для ускорения метода стараются вычисления организовать таким образом, чтобы как можно раньше выявлять заведомо неподходящие варианты. Зачастую это позволяет значительно уменьшить время нахождения решения.

Классическим примером использования алгоритма поиска с возвратом является задача о восьми ферзях. Её формулировка такова: «Расставить на стандартной 64-клеточной шахматной доске 8 ферзей так, чтобы ни один из них не находился под боем другого». Сперва на доску ставят одного ферзя, а потом пытаются поставить каждого следующего ферзя так, чтобы его не били уже установленные ферзи. Если на очередном шаге такую установку сделать нельзя — возвращаются на шаг назад и пытаются поставить ранее установленного ферзя на другое место.

Будем представлять текущую расстановку ферзей на доске в виде вектора

*using Pos = std::vector<unsigned>;*

Запишем вспомогательную функцию, которая определяет, допустимо ли добавить к текущей позиции ещё одного ферзя в следующий столбик

*bool ok2add(Pos& pos, unsigned col) {*

*int delta = pos.size();*

*for (auto& it : pos) {*

*if (col == it || col - it == delta || it - col == delta) {*

*return false;*

*}*

*--delta;*

*}*

*return true;*

*}*

Тогда метод поиска с возвратом может быть реализован в виде рекурсивной функции

*bool backtrack(Pos& pos, unsigned size) {*

*pos.reserve(size);*

*if (pos.size() >= size) {*

*std::cout << pos;*

*return true;*

*}*

*for (unsigned i = 0; i < size; ++i) {*

*if (!ok2add(pos, i)) continue;*

*pos.push\_back(i);*

*auto res = backtrack(pos, size);*

*pos.pop\_back();*

*if (res) return true;*

*}*

*return false;*

*}*

Сложность алгоритма:

* Время O(n!)
* Память O(n)

Помимо этого, метод поиска с возвратом позволяет решать множество других переборных задач. Например, с помощью него можно получить все подмножества, размещения, перестановки, сочетания данного множества М.

Метод поиска с возвратом является универсальным. Достаточно легко проектировать и программировать алгоритмы решения задач с использованием этого метода. Однако время нахождения решения может быть очень велико даже при небольших размерностях задачи (количестве исходных данных), причём настолько велико (может составлять годы или даже века), что о практическом применении не может быть и речи. Поэтому при проектировании таких алгоритмов, обязательно нужно теоретически оценивать время их работы на конкретных данных. Существуют также задачи выбора, для решения которых можно построить уникальные, «быстрые» алгоритмы, позволяющие быстро получить решение даже при больших размерностях задачи. Метод поиска с возвратом в таких задачах применять неэффективно.

# Генерация перестановок

Иногда в других алгоритмах возникает задача перебрать все подмножества, размещения, перестановки или сочетания некоторого данного множества. Такие задачи естественно решаются при помощи поиска с возвратом.

Сгенерируем все перестановки чисел от 1 до N (для заданного пользователем N).

Представим начало текущей перестановки в виде вектора:

*using Prm = std::vector<int>;*

и определим вспомогательную функцию, которая проверяет, что в нашей перестановке не повторяются элементы:

*bool ok2add(const Prm& start, int item) {*

*for (auto it : start)*

*if (it == item)*

*return false;*

*return true;*

*}*

Будем добавлять к проверяемой перестановке новый элемент и рекурсивно проверять, что полученная перестановка может быть продолжена до полной перестановки.

*void try2add(Prm& start, int size) {*

*if (start.size() >= size) {*

*std::cout << start << std::endl;*

*return;*

*}*

*for (int next = 0; next < size; ++next) {*

*if (!ok2add(start, next))*

*continue;*

*start.push\_back(next);*

*try2add(start, size);*

*start.pop\_back();*

*}*

*}*

Тогда вызов, генерирующий все перестановки, можно определить так:

*void try2add(int size) {*

*Prm& start;*

*try2add(start, size);*

*}*

*try2add(5);*

Сложность алгоритма:

* Время O(n!)
* Память O(n)

# Рекомендуемая литература

1. Кормен, Томас. Алгоритмы: построение и анализ : Пер. с англ. / Т. Кормен, Ч. Лейзерсон, Р. Ривест
2. Вирт, Никлаус. Алгоритмы и структуры данных
3. Дасгупта C. Алгоритмы / С. Дасгупта, Х. Пападимитриу, У. Вазирани
4. Скиена, Стивен. Алгоритмы. Руководство по разработке
5. ...