**Материалы для курса   
“Алгоритмы и структуры данных”**

**Содержание**

[Глоссарий](#_viepu6kzycjv) 3

[Лекция. Введение в алгоритмы](#_hnae6t1cxy8u) 5

[Анализ алгоритма](#_9bqmwx9qqjod) 26

[Введение в анализ алгоритмов](#_2p2v5855ofjn) 30

[Анализ алгоритмов с примерами на языке Pascal](#_bx60gctotrx6) 38

[Примеры анализа сложности алгоритмов: блог программиста](#_d9n0xrjl0qc0) 72

[Мастер-теорема](#_s2r3jks981su) 81

[Вычислительная сложность](#_cy9q8cm06gep) 84

[алгоритмов](#_upcwvcybqv43) 84

[Алгоритмы и структуры данных](#_83jtnibpnhm) 98

[Лабораторная работа № 1. Алгоритмы и их сложность](#_kk1p3rm1ttxx) 118

[Рекурсия и рекурсивные алгоритмы (Intuit)](#_9vx3svfedpom) 120

[Построение графиков функций на С++ с использованием Excel](#_yzcru3ngl8n4) 133

# Глоссарий

Алгоритм - (лат. algorithmi — от арабского имени математика Аль-Хорезми) конечная совокупность точно заданных правил решения произвольного класса задач или набор инструкций, описывающих порядок действий исполнителя для решения некоторой задачи.

АТД - Абстрактный тип данных. Это математическая модель для типов данных, где тип данных определяется поведением (семантикой) с точки зрения пользователя данных, а именно в терминах возможных значений, возможных операций над данными этого типа и поведения этих операций.

АТД вектор - расширяет конкретную структуру данных массива. Он содержит методы доступа к обычным элементам последовательности, а также методы обновления, обеспечивающие удаление и добавление элементов с определенным индексом.

АТД список - это абстрактный тип данных, представляющий собой упорядоченный набор значений, в котором некоторое значение может встречаться более одного раза. Экземпляр списка является компьютерной реализацией математического понятия конечной последовательности.

Быстрая сортировка - алгоритм обменной сортировки состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы.

Временная сложность алгоритма - (в худшем случае) это функция от размера входных данных, равная максимальному количеству элементарных операций, проделываемых алгоритмом для решения экземпляра задачи указанного размера.

Дек - от англ. deque — double ended queue; двухсторонняя очередь, очередь с двумя концами.

Очередь - абстрактный тип данных с дисциплиной доступа к элементам «первый пришёл — первый вышел». Добавление элемента возможно лишь в конец очереди, выборка — только из начала очереди, при этом выбранный элемент из очереди удаляется.

Пиромидальная сортировка - алгоритм сортировки, основанный на построении двоичного дерева.

Сортировка - это любой процесс систематического упорядочения элементов, имеющий два общих, но различных значения: упорядочение элементов в последовательности, упорядоченной по некоторому критерию; категоризация: группировка предметов с похожими свойствами.

Сортировка вставками - алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов.

Сортировка выбором - алгоритм сортировки, который состоит в поиске минимального значения элемента в массиве, и перемещения этого значения в начало массива.

Сортировка посчетом - алгоритм сортировки, в котором используется диапазон чисел сортируемого массива для подсчёта совпадающих элементов.

Сортировка слиянием - (англ. merge sort) алгоритм сортировки, который основан на повторяющемся слиянии отсортированных подмассивов.

Сортировка Шелла - алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками. Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга.

Сортировки обменом - алгоритм сортировки прямым обменом основан на принципе сравнения и обмена пары соседних элементов до тех пор, пока не будут отсортированы.

Стек - абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (last in - first out, «последним пришёл - первым вышел»).

# Лекция. Введение в алгоритмы

1. Понятие алгоритма и анализ их сложности

Понятие [*алгоритм*](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) является основным для всей области компьютерного програм­мирования, поэтому начать мы должны с тщательного анализа этого термина. Слово "[алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)" (algorithm) уже само по себе представляет большой интерес. На первый взгляд может показаться, будто кто-то собирался написать слово "логарифм" (logarithm), но случайно переставил первые четыре буквы. Этого слова еще не было в изда­нии словаря *'VeЬster's New World Dictionary,* вышедшем в 1957 году. Мы нахо­дим там только устаревшую форму "algorism" -старинное слово, которое означает "выполнение арифметических действий с помощью арабских цифр". В средние века абакисты считали на абаках (счетных досках), а [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ики использовали "algorism". Наконец историки математики обнаружили истинное происхождение слова "algorism": оно берет начало от имени автора знаменитого персидского учебника по математике – Абу Абд Аллах Мухаммед ибн Муса аль-Хорезми (ок. 825 г.), означающего буквально "Отец Абдуллы, Мухаммед, сын Мусы, уроженец Хорезма". Аральское море в Центральной Азии когда-то называлось озером Хорезм, и район Хорезма (Khwarizm) расположен в бассейне реки Амударьи южнее этого моря. Аль-Хорезми написал знаменитую книгу Китаб аль-джебр валь-мукабала – "Книга о восстановлении и противопоставлении". От названия этой книги, которая была посвящена решению линейных и квадратных уравнений, произошло еще одно слово – "алгебра".

К 1950 году слово "[алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)" чаще всего ассоциировалось с [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ом Евклида, который представляет собой процесс нахождения наибольшего общего делителя двух чисел. Данный [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) был впервые описан в книге Евклида "Начала" (около 300 г. до н.э.) и является наиболее цитируемым при рассмотрении введение в [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)изацию и программирование.

Современное значение слова "[алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)" во многом аналогично таким понятиям, как рецепт, процесс, метод, способ, процедура, программа, но все-таки слово "algorithm" имеет дополнительный смысловой оттенок.

1.1. Свойства [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов

[Алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) – это не просто набор конечного числа правил, задающих последовательность выполнения операций для решения задачи определенного типа. Помимо этого, он имеет пять важных особенностей [1].

1) Конечность. [Алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) всегда должен заканчиваться после выполнения конечного числа шагов. Количество шагов может быть сколь угодно большим; выбор слишком больших значений m и n в [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)е Евклида приведет к тому, что некоторые шаги будет выполняться более миллиона раз (неважно какая модификация [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а будет использована, циклическая или рекурсивная).

Процедура, обладающая всеми характеристиками [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а, за исключением, возможно, конечности, называется методом вычислений. Евклид предложил не только [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) нахождения наибольшего общего делителя, но и аналогичное ему геометрическое построение "наибольшей общей меры" длин двух отрезков прямой; это уже метод вычислений, выполнение которого не заканчивается, если заданные длины оказываются несоизмеримыми

2) Определенность. Каждый шаг [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а должен быть точно определен. Действия, которые нужно выполнить, должны быть строго и недвусмысленно определены для каждого возможного случая. На практике [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ы могут описываться и на обычном языке, и на формализованных псевдоязыках так и на языках программирования. Метод вычислений, выраженный на языке программирования, называется программой.

3) Ввод. [Алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) имеет некоторое (возможно, равное нулю) число входных данных, т. е. величин, которые задаются до начала его работы или определяются динамически во время его работы. Эти входные данные берутся из определенного набора объектов. Например, в [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)е есть два входных значения, а именно m и n, которые принадлежат множеству целых положительных чисел.

4) Вывод. У [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а есть одно или несколько выходных данных, т. е. величин, имеющих вполне определенную связь с входными данными. У [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а Евклида имеется только одно выходное значение, а именно – наибольший общий делитель двух входных значений.

5) Эффективность. [Алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) обычно считается эффективным, если все его операторы достаточно просты для того, чтобы их можно было точно выполнить в течение конечного промежутка времени с помощью карандаша и бумаги. В [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)е Евклида используются только следующие операции: деление одного целого положительного числа на другое, сравнение с нулем и присвоение одной переменной значения другой. Эти операции являются эффективными, так как целые числа можно представить на бумаге с помощью конечного числа знаков и так как существует, по меньшей мере, один способ ("[алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) деления") деления одного целого числа на другое. Но те же самые операции были бы неэффективными, если бы они выполнялись над действительными числами, представляющими собой бесконечные десятичные дроби, либо над величинами, выражающими длины физических отрезков прямой, которые нельзя измерить абсолютно точно.

На практике нам нужны не просто [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ы, а хорошие [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ы в широком смысле этого слова. Одним из критериев качества [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а является время, необходимое для его выполнения; данную характеристику можно оценить по тому, сколько раз выполняется каждый шаг. Другими критериями являются адаптируемость [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а к различным компьютерам, его простота, изящество и т. д.

Часто решить одну и ту же проблему можно с помощью нескольких [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов и нужно выбрать наилучший из них. Таким образом, возникает чрезвычайно интересная и крайне важная область анализа [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов. Предмет этой области состоит в том, чтобы для заданного [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а определить рабочие характеристики. Как правило, это среднее число операций, необходимых для выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а – Tn, где n – параметр, характеризующий каким-то образом исходные данные, например число входных данных.

Для обозначения области подобных исследований используется термин анализ [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов, Основная идея заключается в том, чтобы взять конкретный [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) и определить его количественные характеристики. Можно выяснять, является ли [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) оптимальным в некотором смысле. Теория [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов – это совершенно другая область, в которой, в первую [очередь](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375555&displayformat=dictionary), рассматриваются вопросы существования или не существования эффективных [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов вычисления определенных величин.

[Алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ы, как и аппаратное обеспечение компьютера, представляют собой технологию. Общая производительность системы настолько же зависит от эффективности [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а, как и от мощности применяющегося аппаратного обеспечения. В области разработки [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов происходит такое же быстрое развитие, как и в других компьютерных технологиях.

Возникает вопрос, действительно ли так важны [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ы, работающие на современных компьютерах, если и так достигнуты выдающиеся успехи в других областях высоких технологий, таких как:

* аппаратное обеспечение с высокими тактовыми частотами, конвейерной обработкой и суперскалярной архитектурой;
* легкодоступные, интуитивно понятные графические интерфейсы (GUI);
* объектно-ориентированные системы;
* локальные и глобальные сети.

Ответ — да, безусловно. Несмотря на то, что иногда встречаются приложения, – которые не требуют [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ического наполнения (например, некоторые простые Web-приложения), для большинства приложений определенное [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ическое наполнение необходимо. Например, рассмотрим Web-службу, определяющую, как добраться из одного места в другое. В основе ее реализации лежит высокопроизводительное аппаратное обеспечение, графический интерфейс пользователя, глобальная сеть и, возможно, объектно-ориентированный подход. Кроме того, для определенных операций, выполняемых данной Web-службой, необходимо использование [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов — например, таких как вычисление квадратных корней (что может потребоваться для определения кратчайшего пути), визуализации карт и интерполяции адресов.

Более того, даже приложение, не требующее [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ического наполнения на высоком уровне, сильно зависит от [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов. Известно, что работа приложения зависит от производительности аппаратного обеспечения, а при его разработке применяются разнообразные [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ы. Все мы также знаем, что приложение тесно связано с графическим интерфейсом пользователя, а для разработки любого графического интерфейса пользователя требуются [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ы.

Вспомним приложения, работающие в сети. Чтобы они могли функционировать, необходимо осуществлять маршрутизацию, которая, как уже говорилось, основана на ряде [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов. Чаще всего приложения составляются на языке, отличном от машинного. Их код обрабатывается компилятором или интерпретатором, которые интенсивно используют различные [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ы. И таким примерам нет числа. Кроме того, ввиду постоянного роста вычислительных возможностей компьютеров, они применяются для решения все более сложных задач. Как мы уже убедились на примере сравнительного анализа двух методов сортировки, с ростом сложности решаемой задачи различия в эффективности [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов проявляются все значительнее. Знание основных [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов и методов их разработки — одна из характеристик, отличающих умелого программиста от новичка. Располагая современными компьютерными технологиями, некоторые задачи можно решить и без основательного знания [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов, однако знания в этой области позволяют достичь намного большего.

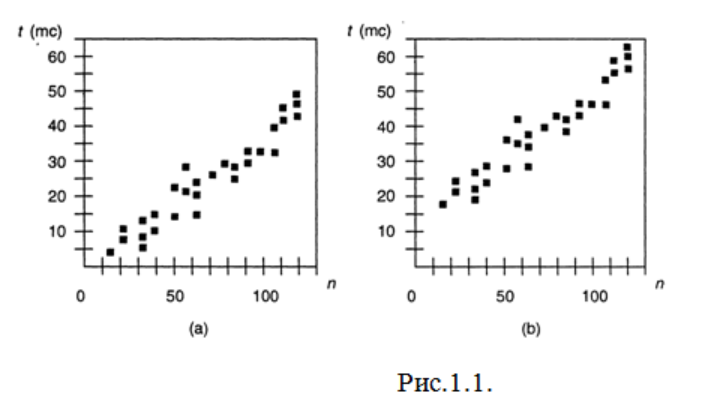
1.2. Анализ [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов

Время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а или операции над структурой данных зависит, как правило, от целого ряда факторов, вследствие чего возникает вопрос – как следует проводить его измерение. При реализации [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а можно определить затраты времени, регистрируя действительное время, затраченное на выполнение [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а в каждом отдельном случае запуска с различными исходными данными [8].

Подобные измерения должны проводиться с достаточной точностью с помощью системных вызовов, встроенных в язык или операционную систему, для которой написан данный [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) (например, метод System.currentTimeMillis() в Java или вызовом исполняющей среды с возможностью профилирования). В общем, требуется определить, каким образом время выполнения программы зависит от количества исходных данных.

Для решения этой задачи можно провести ряд экспериментов, в которых будет использовано различное количество исходных данных. Далее полученные результаты наглядно представляются с помощью графика, где каждый случай выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а обозначается с помощью точки, координата х которой равна размеру исходных данных n, а координата у — времени выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а t (см. рис. 1.1).

Чтобы сделать определенные выводы на основе полученных экспериментов, необходимо использовать качественные образцы исходных данных и провести достаточно большое число экспериментов- что позволит определить некоторые статистические характеристики в отношении времени выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а.



Результаты экспериментального исследования времени выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а. Точка с координатами (*n*, *t*) обозначает, что при размере исходных данных n время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а составило *t* миллисекунд (мс).

На рис. 1.1, а представлены результаты выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а на компьютере с быстрым процессором, на рис. 1.1, b представлены результаты выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а на компьютере с медленным процессором. В целом можно сказать, что время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а или метода доступа к полям структуры данных возрастает по мере увеличения размера исходных данных, хотя оно зависит и от типа данных, даже при равном размере. Кроме того, время выполнения зависит от аппаратного обеспечения (процессора, тактовой частоты, размера памяти, места на диске и др.) и программ программного обеспечения (операционной среды, языка программирования, компилятора, интерпретатора и др.), с помощью которых осуществляется реализация, компиляция и выполнение [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а. Например, при всех прочих равных условиях время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а определенного количества исходных данных будет меньше при использовании более мощного компьютера или при записи [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а в виде программы на машинном коде по сравнению с его исполнением виртуальной машиной, проводящей интерпретацию в байт-коды.

Экспериментальные исследования, безусловно, очень полезны, однако при их проведении существуют три основных ограничения:

* эксперименты могут проводиться лишь с использованием ограниченного набора исходных данных; результаты, полученные с использованием другого набора, не учитываются;
* для сравнения эффективности двух [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов необходимо, чтобы эксперименты по определению времени их выполнения проводились на одинаковом аппаратном и программном обеспечении;
* для экспериментального изучения времени выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а необходимо провести его реализацию и выполнение.

Обычно для сравнительного анализа рассматривается, так называемая общая методология анализа времени выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов, которая:

* учитывает различные типы входных данных;
* позволяет производить оценку относительной эффективности любых двух [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов независимо от аппаратного и программного обеспечения;
* может проводиться по описанию [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а без его – непосредственной реализации или экспериментов.

Сущность такой методологии состоит в том, что каждому [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)у соответствует функция, которая представляет время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а как функцию размера исходных данных *n*. Наиболее распространенными являются функции *n* и *n*2. Например, можно записать следующее утверждение: «Время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а *А* пропорционально n». В этом случае, в результате проведения экспериментов, окажется, что время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а *А* при любом размере входных данных *n* не превышает значения *cn*, где *с* является константой, определяемой условиями используемого аппаратного и программного обеспечения. Если имеются два [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а *А* и *В*, причем время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а *А* пропорционально *n*, а время выполнения *В* пропорционально *n*2, то предпочтительнее использовать [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) *А*, так как функция *n* возрастает медленнее, чем функция *n*2. Выводы о скорости возрастания этих функций являются, безусловно, очевидными, однако предлагаемая методология содержит точные определения данных понятий. Прежде чем перейти к описанию общей методики, определим понятие псевдокод и области его применения.

1.2.1 Псевдокод

Зачастую программистам требуется создать описание [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а, предназначаемое только для человека. Подобные описания не являются программами, но вместе с тем они более структурированы, чем обычный текст. В частности, «высокоуровневые» описания сочетают естественный язык и распространенные структуры языка программирования, что делает их доступными и вместе с тем информативными. Такие описания способствуют проведению высокоуровневого анализа структуры данных или [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а. Подобные описания принято называть псевдокодом.

Проблема «максимума в массиве» является простой задачей поиска элемента с максимальным значением в массиве А, содержащем n целых чисел. Для решения этой задачи можно использовать [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) аггауМах, который осуществляет просмотр массива А с использованием цикла for.

Псевдокод [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а аггауМах представлен во фрагменте кода , а полная реализация программы Java представлена в следующем фрагменте кода.

Input: массив А, содержащий п целых чисел *(n* > 1).

Output: элемент с максимальным значением в массиве A.

1 currentMax ←*А* [0] выполняется один раз

2 for *i* ←1 to *n* - 1 do выполняется от 1 до *n*, *n*-1 раз соответственно

3 if currentMax < *A*[*i*] then выполняется *n*-1 раз

4 currentMax ← *А* [*i*] выполняется максимально *n*-1 раз

5 return currentMax выполняется один раз

Фрагмент кода [Алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) аггауМах

// Тестируем программу [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а поиска максимального элемента массива.

public class ArrayMaxProgram {

// находит элемент с максимальным значением в массиве А, содержащем n целых //чисел.

static int arrayMax(int[ ] *A*, int *n*) {

int currentMax = *A*[0]; // выполняется один раз

for (int *i* = l; *i* < *N*; *i*++) /\* выполняется от 1 до *n*, *n*-1 раз соответственно. \*/

if (currentMax < *A*[*i*]) // выполняется *n*-1 раз

currentMax = *A*[*i*]; // выполняется максимально *n*-1 раз

return currentMax; // выполняется один раз

}

// Тестирующий метод, вызываемый после выполнения программы.

public static void main(String args[ ]) {

int[ ] num = { 10, 15, 3, 5, 56, 107, 22, 16, 85 };

int *n* = num.length;

System.out.print("Array:");

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

System.out.print("" + num[*i*]);

System.out.println(".");

System.out.println("The maximum elementis" + arrayMax(num,*n*) + ".");

}

}

Фрагмент кода [Алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) arrayMax внутри законченной Java-программы.

Как можно заметить, псевдокод выглядит компактнее Java-кода, и его легче читать и понимать. При анализе псевдокода можно поспорить о правильности [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а arrayMax c простым аргументом. Переменная currentMax первоначально принимает значение первого элемента массива А. Можно утверждать, что перед началом итерации по номеру *i* значение currentMax равно максимальному значению среди первых *i* элементов массива А. Так как при повторении I значение currentMax сравнивается с *A[i],* то, если это утверждение верно перед данной итерацией, оно будет верно и после нее для *i + 1* (которое является следующим значением счетчика *i*).Таким образом, после количества итераций *n - 1* значение currentMax будет равно элементу массива *А*, содержащему максимальное значение.

Псевдокод является сочетанием естественного языка и конструкций языка программирования, которые используются для описания основных идей, заложенных в реализацию структуры данных или [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а. Точного определения языка псевдокода не существует, так как в нем все же преобладает естественный язык. В то же время для обеспечения четкости и ясности в псевдокоде наряду с естественным языком используются стандартные конструкции языка программирования:

• Выражения. Для написания числовых и логических выражений используются стандартные математические символы. Знак стрелки применяется в качестве оператора присваивания в командах присваивания (равнозначен оператору «=» языка Java). Знак «=» используется для передачи отношения равенства в логических выражениях (что соответствует оператору «= =» языка Java).

• Объявление метода. Имя [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а (*paraml, param2,* ...) объявляет «имя» нового метода и его параметры.

• Структуры принятия решений. Условие if, then — действия, если условие верно [else — если условие не верно]. Отступы используются для обозначения выполняемых в том или другом случае действий.

• Цикл while, while — условие, do - действия. Отступ обозначает действия, выполняемые внутри цикла.

• Цикл repeat, repeat — действия, которые выполняются, пока выполняется условие until. Отступ обозначает действия, выполняемые внутри цикла.

• Цикл for. for — описание переменной и инкремента, do — действия. Отступ обозначает действия, выполняемые внутри цикла

• Ин[дек](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375554&displayformat=dictionary)сирование массива. A[*i*] обозначает *i*-ую ячейку массива А. Ячейки массива А с количеством ячеек n ин[дек](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375554&displayformat=dictionary)сируются от *A[0]* до *A[n - 1]* (как в Java).

• Обращения к методам, object.method (args) (часть object необязательна, если она очевидна).

• Возвращаемое методом значение. Значение return. Данный оператор возвращает значение, указанное в методе, вызывающим данный метод.

При записи псевдокода следует помнить, что он записывается для анализа человеком, а не компьютером, и необходимо выразить основные глобальные идеи, а не базовые детали реализации структуры. В то же время не следует упускать из виду важные этапы. Таким образом, как при любом типе человеческого общения, написание псевдокода состоит в поиске баланса между общим и частным; такое умение вырабатывается в ходе практической деятельности. Дополнительно, для лучшей передачи сущности [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а или структуры данных, описание [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а должно начинаться с краткого абстрактного объяснения характера входного и выходного потока данных, а также основных действий и используемых идей [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а.

1.2.2. Простейшие операции

После изложения высокоуровневого способа описания [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов приступим к рассмотрению различных способов анализа [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов.

Как уже отмечалось, экспериментальный анализ может быть очень полезен, но обладает рядом ограничений. Чтобы провести анализ [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а без экспериментов, связанных с изучением времени его выполнения, используем аналитический подход, который состоит из следующих операций:

1. Записать [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) в виде кода одного из развитых языков программирования (например, Java).

2. Перевести программу в последовательность машинных команд (например, байт-коды, используемые в виртуальной машине Java).

3. Определить для каждой машинной команды *i* время *ti,* необходимое для ее выполнения.

4. Определить для каждой машинной команды *i* количество повторений команды *ni* за время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а.

5. Определить произведение *ti\*ni* всех машинных команд, что и будет составлять время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а.

Данный подход позволяет точно определить время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а, однако он очень сложен и трудоемок, поскольку требуется доскональное знание машинных команд, создаваемых компилятором и средой, в которой выполняется [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary).

В силу этого удобнее проводить анализ непосредственно программы, написанной на языке высокого уровня, или псевдокода. Выделим ряд простейших операций высокого уровня, которые в целом не зависят от используемого языка программирования и могут использоваться в псевдокоде:

• присваивание переменной значения,

• вызов метода,

• выполнение арифметической операции (например, сложение двух чисел),

• сравнение двух чисел,

• ин[дек](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375554&displayformat=dictionary)сация массива,

• переход по ссылке на объект,

• возвращение из метода.

Следует отметить, что простейшие операции соответствуют машинным командам, время выполнения которых зависит от аппаратного и программного обеспечения, но тем не менее является постоянной известной величиной. Вместо попыток определения времени выполнения отдельной простейшей операции достаточно просто подсчитать -количество таких операций и использовать полученное значение в качестве критерия оценки времени выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а. Такой подсчет количества операций можно соотнести со временем выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а в условиях определенного аппаратного и программного обеспечения, так как каждая простейшая операция соответствует машинной команде, время выполнения которой есть величина постоянная, а число простейших операций известно и неизменно. Данный метод основан на неявном предположении о том, что время выполнения различных простейших операций приблизительно одинаково. Таким образом, число t простейших операций, выполняемых внутри [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а, пропорционально действительному времени выполнения данного [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а.

Рассмотрим процесс подсчета числа простейших операций, выполняемых внутри [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а, на примере [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а аггауМах. Псевдокод и Java-реализация этого [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а представлены во фрагментах кодов 3.1 и 3.2 соответственно. Данный анализ может проводиться как на основании псевдокода, так и Java-реализации.

• На этапе инициализации переменной currentMax и присваивания ей значения *A*[0] выполняются две простейшие операции (ин[дек](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375554&displayformat=dictionary)сация массива и присваивание переменной значения), которые однократно выполняются в начале [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а. Таким образом, счетчик операций равен 2.

• В начале выполнения цикла for счетчик *i* получает значение 1. Это соответствует одной простейшей операции (присваивание значения переменной).

• Перед выполнением тела цикла проверяется условие *i<n*. В данном случае выполняется простейшая операция сравнения чисел. Так как первоначальное значение счетчика *i* равно 0, а затем его значение увеличивается на 1 в конце каждой итерации цикла, сравнение *i<n* проверяется *n* раз. Таким образом, в счетчик простейших операций добавляется еще n единиц.

• Тело цикла for выполняется *n ≥ 1* раз (для значений счетчика 1, 2,..., *n*- 1). При каждой итерации *A[i]* сравнивается с currentMax (две простейшие операции — ин[дек](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375554&displayformat=dictionary)сирование и сравнение), значение *A[i],* возможно, присваивается currentMax (две простейшие операции — ин[дек](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375554&displayformat=dictionary)сирование и присваивание значения), а счетчик i увеличивается на 1 (две простейшие операции — сложение и присваивание значения). Таким образом, при каждой итерации цикла выполняется 4 или 6 простейших операций, в зависимости от того *A[i]* ≤ currentMax или *A[i]* > currentMax. Таким образом, при выполнении тела цикла в счетчик простейших операций добавляется

*4(n - 1)* или *6(n - 1)* единиц.

• При возвращений значения переменной currentMax однократно в выполняется одна простейшая операция.

Итак, число простейших операций tНет, выполняемых [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ом arrayMax, минимально равно

2 + 1 *+ n +* 4*(n -*1*) +* 1 *=* 5*n*,

а максимально

2 + 1 *+ n +* 6*(n -*1*) +* 1 *=* 7*n - 2*.

Число выполняемых операций равно минимально (*t(n) =* 5*n*) в том случае, если *A[*0*]* является максимальным элементом массива, то есть переменной currentMax не присваивается нового значения. Число выполняемых операций максимально равно (*t(n)= =* 7*n -* 2) в том случае, если элементы массива отсортированы по возрастанию, и переменной currentMax присваивается новое значение при каждой очередной итерации цикла.

1.2.3. Анализ средних и худших показателей

На примере метода аггауМах можно убедиться, что при определенном типе исходных данных [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) выполняется быстрее, чем при других. В данном случае можно попытаться выразить время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а как среднее, взятое на основе результатов, полученных при всех возможных исходных данных. К сожалению, проведение анализа с точки зрения средних показателей является весьма проблематичным, так как в этом случае требуется определить вероятностное распределение входящего потока. На рис. 3.3 схематично представлена зависимость времени выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а от распределения входного потока данных. Например, если исходные данные только типа «А» или «D».

При анализе средних показателей необходимо определить предположительное время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а при некотором распределении входного потока данных. Для проведения подобных вычислений зачастую требуется применение понятий высшей математики и теории вероятности.

В связи с этим в дальнейшем будем по умолчанию указывать худший показатель времени выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а (если не будет оговорено другое условие). Можно сказать, что в худшем случае [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) arrayMax выполняет *t(n) = 7n - 2* простейших операций, то есть максимальное число простейших операций, выполняемых [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ом при использовании всех исходных данных размера n, составляет *7n - 2*.

Подобный тип анализа намного проще анализа средних показателей, так как не требует использования теории вероятности; для него просто необходимо определить, при каком типе исходных данных время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а будет максимальным, что зачастую является вполне очевидным. Кроме того, использование такого подхода может способствовать совершенствованию [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов. Другими словами, если создаваемый [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) должен успешно работать при худших исходных данных, предполагается, что он будет работать при любом типе исходных данных.

Таким образом, проектирование [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а на основании худших показателей приводит к созданию более устойчивой «сущности» [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а, аналогично ситуации, когда чемпион по бегу во время тренировок бегает только в гору.

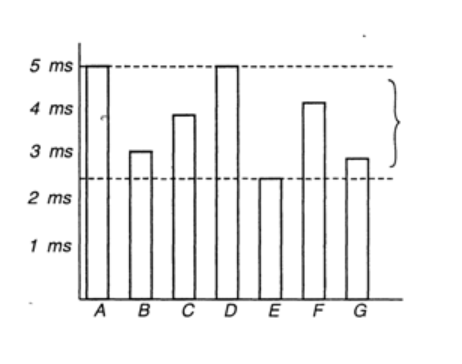


Рис. 1.2 Различие между наилучшим и наихудшим показателями времени выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а.

Каждый прямоугольник соответствует времени выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а при различных типах исходных данных

1.2.4. Асимптотическая нотация

Очевидно, что анализ времени выполнения такого простого [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а, как arrayMax, проведен слишком глубоко. В ходе анализа возникает несколько вопросов:

* Действительно ли необходим такой уровень детализации?
* Действительно ли так важно установить точное число простейших операций, выполняемых [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ом?
* Насколько досконально следует определять количество простейших операций? Например, сколько простейших операций выполняется в команде *у = a\*x + b*? (Можно определить, что выполняются две арифметические операции и одна операция присваивания, но, с другой стороны, в этом случае не учитывается еще одна «скрытая» операция присваивания результата произведения a\*x временной переменной перед выполнением операции сложения.)

В целом каждый этап псевдокода или команда программы на языке программирования содержит небольшое число простейших операций, которое не зависит от размера исходных данных. Таким образом, можно проводить упрощенный анализ, при котором число простейших операций определяется применением некоторой константы, путем подсчета выполненных шагов псевдокода или команд программы. Возвращаясь к [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)у arrayMax, упрощенный анализ даст следующий результат: [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) выполняет от 5*n* до 7*n* - 2 шагов при размере исходных данных *n*.

При анализе [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов следует рассматривать увеличение времени его выполнения как функцию исходных данных *n*, уделяя основное внимание глобальным аспектам и не вдаваясь в отдельные мелкие детали. Зачастую достаточно просто знать, что время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а, например, arrayMax, увеличивается пропорционально *n*. Это означает, что действительное время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а является произведением n на некий постоянный множитель, который определяется условиями аппаратного и программного обеспечения, и может колебаться в определенных пределах в зависимости от особенностей исходных данных.

Формализуем приведенный метод анализа структур данных и [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов, используя математическую систему функций, в которой не учитываются постоянные факторы. В частности, будем описывать время выполнения и требования к памяти с помощью функций, которые преобразуют целые числа в действительные, обращая внимание на глобальные характеристики функции времени выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а или ограничения дискового пространства.

Рассмотрим основные понятия так называемой нотация большого *О* [6].

Пусть *f(n)* и *g(n)* являются функциями, которые преобразуют неотрицательные целые числа в действительные. Докажем, что *f(n)* есть *O(g(n)),* если существует действительная константа *с > 0* и целочисленная константа *n0 >* 1, такие, что *f(n) < cg (n)* для любого целого числа *n > n0*. Такое определение функции зачастую называется нотацией большого *О*, так как иногда говорят, что *f(n)* является большим *O* функции *g(n).* Другими словами, можно сказать, что fНет есть порядковая функция gНет (определение показано на рис. 3.4).

Пример: 7*n - 2* есть *O(n).*

Доказательство: по определению нотации большого *О* необходимо найти действительную константу *с > 0*, а также целочисленную константу *n0 ≥*1, так, что 7*n-*2*n0 ≤ сn* для любого целого числа, *n≥n0*. Одним из очевидных вариантов является *с =* 7, а *n0 =* 1. Безусловно, это один из бесконечного множества вариантов; *с* в данном случае может принимать значение любого действительного числа, равного или большего 7, а *n0* - любое целочисленное значение, большее или равное 1.

Нотация большого О позволяет выразить, что функция n «меньше или равна» другой функции (в определении это выражается знаком «≤») до определенного постоянного значения (константа с в определении) и по мере того, как n стремится к бесконечности (условие «*n ≥ n0*» в определении).

Зачастую нотация большого O используется для характеристики времени выполнения и использования памяти на основании некоего параметра n, который может различаться в конкретных ситуациях, однако, как правило, зависит от целей анализа.



Рис. 1.3. Наглядное изображение нотации большого О. Функция *f(n)* есть *O(g(n)),* так как *f(n)≤ cg(n)* при *n ≥ n0*

Например, если требуется найти наибольший элемент массива целых чисел (см. аггауМах, представленный во фрагментах кодов), то вполне естественно использовать n для обозначения числа элементов массива. Нотация большого *О* позволяет не учитывать постоянные множители и частные детали, а сосредоточиться только на основных компонентах функции, определяющих ее возрастание.

Используя нотацию большого *О*, можно представить математически точное определение времени выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а аггауМах независимо от применяемых аппаратного и программного обеспечения.

Утверждение. Время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а аггауМах, определяющего максимальный элемент массива целых чисел, есть функция *O(n).*

Доказательство. Как было отмечено, максимальное число простейших операций, выполняемых [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ом аггауМах, равно *7n - 2*. Таким образом, существует положительная константа a, которая определяется единицами измерения времени, а также применяемым при реализации, компиляции и исполнении аппаратным и программным обеспечениями, при которой время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а аггауМах для размера

исходных данных *n* равно максимально *a(*7*n -* 2*).* Используя нотацию большого *О* для *с =* 7*a* ,a *n0=* 1, можно сделать вывод, что время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а аггауМах является *O(n).*

Рассмотрим еще несколько примеров, демонстрирующих использование нотации большого *О*.

Рассмотрим еще несколько примеров, демонстрирующих использование нотации большого *О*.

Пример: 20*n3+* 10*n log n +* 5 есть *О(n3).*

Доказательство: 20*n3+* 10*n log n +* 5 *<* 35*n3* для *n≥*1.

В сущности, любой многочлен (полином) *aknk + ak-1+ …+0* является *О(nk).*

Пример: 2100 есть *O*(1)

Доказательство: 2100 ≤ 2100 • 1 для *n ≥* 1. Заметьте, что переменная n не присутствует в неравенстве, так как в данном случае рассматриваются постоянные функции.

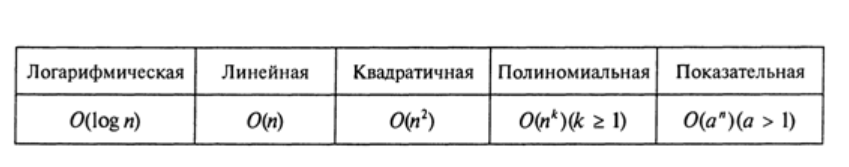
Пример: 5 */ n* есть O(1/n)

Доказательство: 5 */ n* ≤ 5*(*1*/n)* для n ≥1 (на самом деле это убывающая функция).

В целом можно сказать, что нотация большого *О* используется для максимально точного описания функции. Если верно, что функция *F(n)* = 4*n3 + Зn4/3* есть *О* (*n5*) или даже *O* (*n3 log n*), то более точно будет сказать, что *fНет* есть O(*n3*). Рассмотрим следующую аналогию. Голодный путешественник долго едет по пустынной проселочной дороге и встречает местного фермера, который возвращается домой с рынка. Если путешественник спросит фермера, сколько ему нужно ехать, чтобы добраться до ближайшего места, где он может поесть, фермер может вполне правдиво ответить: «Не более 12 часов», но более точным (и полезным) будет, если он скажет: «Всего в нескольких минутах езды отсюда находится рынок».

Таким образом, даже используя нотацию большого *O*, следует стремиться сообщать все возможные подробности.

При анализе [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов и структур данных часто применяются некоторые функции, имеющие особые названия. Например, термин «линейная функция» обозначает функции вида *OНет*. В табл. 3.1 представлены функции, часто используемые при анализе [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов.



Классы функций

Нотация большого *О* позволяет проводить асимптотический анализ, то есть определять, что функция «меньше или равна» другой функции. Существуют типы нотаций, которые позволяют проводить асимптотические сравнения других типов: *Ω-* и *Θ*- нотации, которые асимптотически задают ограничения на функцию снизу или снизу и сверху одновременно (рис 1.4)

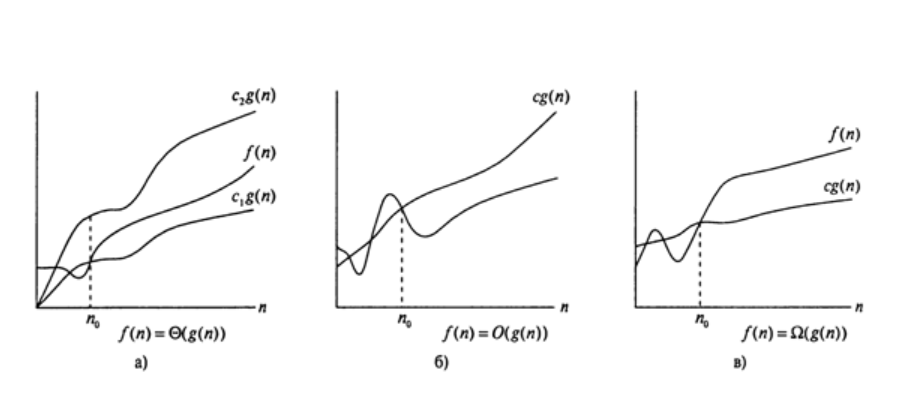


Рис. 1.4. Графические примеры *Θ, О* и *Ω*, обозначений; в каждой части рисунка в качестве n0 используется минимально возможное значение, т.е. любое большее значение также сможет выполнить роль n0

Выскажем некоторые предостережения в отношении использования асимптотических нотаций. Во-первых, следует отметить, что нотация большого *О* и аналогичные ей могут привести к ошибочным результатам, если «скрываемые» ими постоянные множители достаточно велики. Например, хотя функция 10100n есть *Θ(n),* если она обозначает время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а, сравниваемого с [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ом, время выполнения которого равно 10*n log n*, следует выбрать [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary) со временем *Θ(n logn),* даже несмотря на то, что асимптотически время выполнения, выраженное линейной функцией, меньше. Данный выбор обусловлен тем, что постоянный множитель 10100, называемый «hiding», считается верхним пределом числа атомов в наблюдаемой части вселенной. Таким образом, маловероятно, что [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)у придется решать задачу реального мира с таким размером исходных данных. Таким образом, при использовании нотации большого *О* следует обращать внимание на постоянные множители и элементы низшего порядка, которые могут «скрываться» за ними.

В целом нотации большого *O*, большой омеги и большой теты являются удобным средством анализа структур данных и [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов. Как упоминалось ранее, удобство данных нотаций состоит в том, что они позволяют сосредоточиться на основных составляющих, влияющих на время выполнения [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов без учета частных деталей.

# Анализ алгоритма

[Электронный ресурс]. – Режим доступа: <https://habr.com/ru/post/127753/>, свободный.

Что такое анализ?

Анализируя алгоритм, можно получить представление о том, сколько времени займет решение данной задачи при помощи данного алгоритма. Одну и ту же задачу можно решить с помощью различных алгоритмов. Анализ алгоритмов дает нам инструмент для выбора алгоритма.

Результат анализа алгоритмов — не формула для точного количества секунд или компьютерных циклов, которые потребует конкретный алгоритм. Нужно понимать, что разница между алгоритмом, который делает N + 5 операций, и тем, который делает N + 250 операций, становится незаметной, как только N становится очень большим.

Классы входных данных

При анализе алгоритма выбор входных данных может существенно повлиять на его выполнение. Скажем, некоторые алгоритмы сортировки могут работать очень быстро, если входной список уже отсортирован, тогда как другие алгоритмы покажут весьма скромный результат на таком списке. А вот на случайном списке результат может оказаться противоположным. Поэтому не будем ограничиваться анализом поведения алгоритмов на одном входном наборе данных. Практически нужно искать такие данные, которые обеспечивают как самое быстрое, так и самое медленное выполнение алгоритма. Кроме того, полезно оценивать и среднюю эффективность алгоритма на всех возможных наборах данных.

Наилучший случай

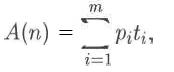
Время выполнения алгоритма в наилучшем случае очень часто оказывается маленьким или просто постоянным, поэтому подобный анализ проводится редко.

Наихудший случай

Анализ наихудшего случая чрезвычайно важен, поскольку он позволяет представить максимальное время работы алгоритма. При анализе наихудшего случая необходимо найти входные данные, на которых алгоритм будет выполнять больше всего работы.

Средний случай

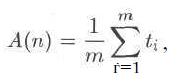
Анализ среднего случая является самым сложным, поскольку он требует учета множества разнообразных деталей. В основе анализа лежит определение различных групп, на которые следует разбить возможные входные наборы данных. На втором шаге определяется вероятность, с которой входной набор данных принадлежит каждой группе. На третьем шаге подсчитывается время работы алгоритма на данных из каждой группы. Время работы алгоритма на всех входных данных одной группы должно быть одинаковым, в противном случае группу следует подразбить. Среднее время работы вычисляется по формуле



где через n обозначен размер входных данных, через m — число групп. через pi — вероятность того, что входные данные принадлежат группе с номером i, а через ti — время, необходимое алгоритму для обработки данных из группы с номером i.

В большинстве случаев мы будем предполагать, что вероятности попадания входных данных в каждую из групп одинаковы. Другими словами, если групп пять, то вероятность попасть в первую группу такая же, как вероятность попасть во вторую, и т.д., то есть вероятность

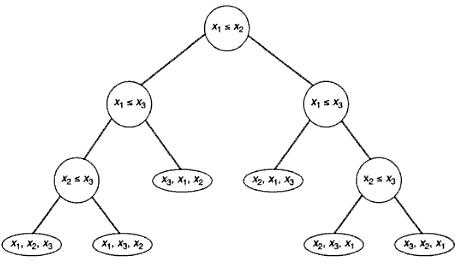
попасть в каждую группу равна 0.2. В этом случае среднее время работы можно либо оценить по предыдущей формуле, либо воспользоваться эквивалентной ей формулой



Нижние границы. Дерево решений.

Алгоритм является оптимальным, если любой любой другой алгоритм, решающий данную задачу, работает не быстрее данного. Как узнать оптимальность алгоритма? Для этого мы должны знать минимальное количество операций, необходимое для решения поставленной задачи, которое называется нижней границей. Но для этого нам нужно изучать именно ЗАДАЧУ, а не конкретный алгоритм!

Как пример, рассмотрим анализ процесса сортировки списка из 3 чисел, воспользовавшись бинарным деревом. Такое дерево называется деревом решений.



Самый длинный путь в данном дереве соответствует наихудшему случаю и наоборот, кратчайший — наилучшему. Средний случай описывается частным от деления числа ребер в дереве на число листов.

Для перестановки 2-х элементов мы имеем 1лист, для N эл-тов, по правилам комбинаторики, N! листов. Число листов на уровне K равно 2k-1, поэтому в нашем дереве решений буде L -уровней, где L — наименьшее целое число, для которого N! ≤ 2L-1. Логарифмируя, получим

image

image

image

В конечном итоге, получаем:



Вот мы и узнали, что любой алгоритм сортировки, порядка О(NlgN) является наилучшим и его можно считать оптимальным. Более того, из этого исследования вытекает, что алгоритм, решающий задачу сортировки быстрее О(NlgN) операций, не может работать правильно!

# Введение в анализ алгоритмов

[Электронный ресурс]. – Режим доступа: <http://techn.sstu.ru/kafedri/%D0%BF%D0%BE%D0%B4%D1%80%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F/1/MetMat/shaturn/theoralg/4.htm>, свободный.

1. Сравнительные оценки алгоритмов  
   При использовании алгоритмов для решения практических задач мы сталкиваемся с проблемой рационального выбора алгоритма решения задачи. Решение проблемы выбора связано с построением системы сравнительных оценок, которая в свою очередь существенно опирается на формальную модель алгоритма.  
   Будем рассматривать в дальнейшем, придерживаясь определений Поста, применимые к общей проблеме, правильные и финитные алгоритмы, т.е. алго-ритмы, дающие 1-решение общей проблемы. В качестве формальной системы будем рассматривать абстрактную машину, включающую процессор с фонНеймановской архитектурой, поддерживающий адресную память и набор «элементарных» операций соотнесенных с языком высокого уровня.

В целях дальнейшего анализа примем следующие допущения:  
каждая команда выполняется не более чем за фиксированное время;

исходные данные алгоритма представляются машинными словами по  битов каждое.

Конкретная проблема задается N словами памяти, таким образом, на входе алгоритма –  = N\* бит информации. Отметим, что в ряде случаев, особенно при рассмотрении матричных задач N является мерой длины входа алгоритма, отражающей линейную размерность.  
Программа, реализующая алгоритм для решения общей проблемы состоит из М машинных инструкций по  битов –  = М\* бит информации.

Кроме того, алгоритм может требовать следующих дополнительных ресурсов абстрактной машины:  
 – память для хранения промежуточных результатов;

 – память для организации вычислительного процесса (память, необходимая для реализации рекурсивных вызовов и возвратов).

При решении конкретной проблемы, заданной N словами памяти алгоритм выполняет не более, чем конечное количество «элементарных» операций абстрактной машины в силу условия рассмотрения только финитных алгоритмов. В связи с этим введем следующее определение:  
Определение 4.1. Трудоёмкость алгоритма.  
Под трудоёмкостью алгоритма для данного конкретного входа –  (N), будем понимать количество «элементарных» операций совершаемых алгоритмом для решения конкретной проблемы в данной формальной системе.  
Комплексный анализ алгоритма может быть выполнен на основе комплексной оценки ресурсов формальной системы, требуемых алгоритмом для решения конкретных проблем. Очевидно, что для различных областей применения веса ресурсов будут различны, что приводит к следующей комплексной оценке алгоритма:  
, где  – веса ресурсов.

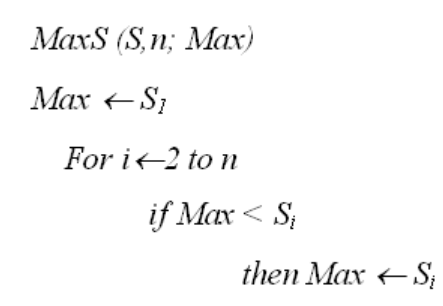
1. Система обозначений в анализе алгоритмов  
   При более детальном анализе трудоемкости алгоритма оказывается, что не всегда количество элементарных операций, выполняемых алгоритмом на одном входе длины N, совпадает с количеством операций на другом входе такой же длины. Это приводит к необходимости введения специальных обозначений, отражающих поведение функции трудоемкости данного алгоритма на входных данных фиксированной длины.  
   Пусть  – множество конкретных проблем данной задачи, заданное в формальной системе. Пусть D є  – задание конкретной проблемы и |D| = N.  
   В общем случае существует собственное подмножество множества , включающее все конкретные проблемы, имеющие мощность N:  
   обозначим это подмножество через :  = {D є ,: |D| = N};  
   обозначим мощность множества  через  -->  = | |.

Тогда содержательно данный алгоритм, решая различные задачи размерности N, будет выполнять в каком-то случае наибольшее количество операций, а в каком-то случае наименьшее количество операций. Ведем следующие обозначения:  
(N) – худший случай – наибольшее количество операций, совершаемых алгоритмом А для решения конкретных проблем размерностью N:  
 - худший случай на 

(N) – лучший случай – наименьшее количество операций, совершаемых алгоритмом А для решения конкретных проблем размерностью N:  
 – лучший случай на 

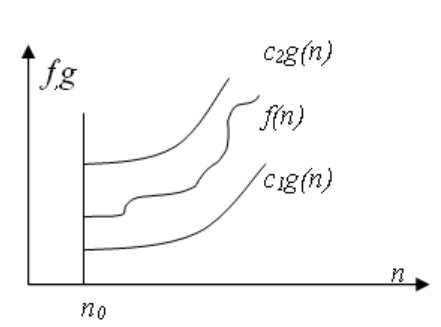
(N) – средний случай – среднее количество операций, совершаемых алгоритмом А для решения конкретных проблем размерностью N:  
 – средний случай на 

1. Классификация алгоритмов по виду функции трудоёмкости  
   В зависимости от влияния исходных данных на функцию трудоемкости алгоритма может быть предложена следующая классификация, имеющая практическое значение для анализа алгоритмов:  
   Количественно-зависимые по трудоемкости алгоритмы  
   Это алгоритмы, функция трудоемкости которых зависит только от размерности конкретного входа, и не зависит от конкретных значений:  
   (D) = (|D|) = (N)  
   Примерами алгоритмов с количественно-зависимой функцией трудоемкости могут служить алгоритмы для стандартных операций с массивами и матрицами – умножение матриц, умножение матрицы на вектор и т.д.  
   2.Параметрически-зависимые по трудоемкости алгоритмы  
   Это алгоритмы, трудоемкость которых определяется не размерностью входа (как правило, для этой группы размерность входа обычно фиксирована), а конкретными значениями обрабатываемых слов памяти:  
   (D) =  (,…,) = (,…,), m =< n  
   Примерами алгоритмов с параметрически-зависимой трудоемкостью являются алгоритмы вычисления стандартных функций с заданной точностью путем вычисления соответствующих степенных рядов. Очевидно, что такие алгоритмы, имея на входе два числовых значения – аргумент функции и точность выполняют существенно зависящее от значений количество операций.  
   а) Вычисление  последовательным умножением  (x, k) = (k).  
   б) Вычисление  =  (/n!), с точностью до    = (x, )  
    Количественно-параметрические по трудоемкости алгоритмы  
   Однако в большинстве практических случаев функция трудоемкости зависит как от количества данных на входе, так и от значений входных данных, в этом случае:  
   (D) = (||D||, ,…,) =  (N, ,…, )  
   В качестве примера можно привести алгоритмы численных методов, в которых параметрически-зависимый внешний цикл по точности включает в себя количественно-зависимый фрагмент по размерности.  
   3.1 Порядково-зависимые по трудоемкости алгоритмы  
   Среди разнообразия параметрически-зависимых алгоритмов выделим еще оду группу, для которой количество операций зависит от порядка распо-ложения исходных объектов.  
   Пусть множество D состоит из элементов (,…,), и ||D||=N,  
   Определим = {( ,…,)}-множество всех упорядоченных N-ок из ,…, , отметим, что ||=n!.  
   Если (i)   (j), где i, j є , то алгоритм будем называть порядково-зависимым по трудоемкости.  
   Примерами таких алгоритмов могут служить ряд алгоритмов сортировки, алгоритмы поиска минимума и максимума в массиве. Рассмотрим более подробно алгоритм поиска максимума в массиве S, содержащим n элементов:



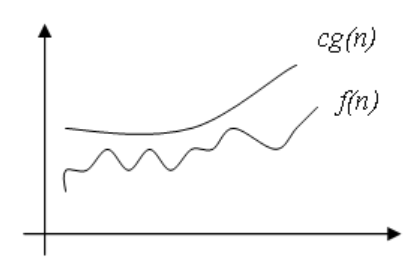
|  | (количество выполненных операций присваивания зависит от порядка следования элементов массива) |
| --- | --- |

4. Асимптотический анализ функций  
При анализе поведения функции трудоемкости алгоритма часто используют принятые в математике асимптотические обозначения, позволяющие по-казать скорость роста функции, маскируя при этом конкретные коэффициенты.  
Такая оценка функции трудоемкости алгоритма называется сложностью алгоритма и позволяет определить предпочтения в использовании того или иного алгоритма для больших значений размерности исходных данных.  
В асимптотическом анализе приняты следующие обозначения:  
1. Оценка  (тетта)  
Пусть f(n) и g(n) – положительные функции положительного аргумента, n >= 1 (количество объектов на входе и количество операций – положительные числа), тогда:



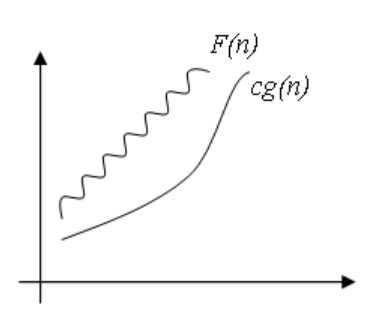
|  | f(n) = (g(n)), если существуют положительные с1, с2, n0, такие, что: с1 \* g(n) =< f(n) =< c2 \* g(n), при n > n0 |
| --- | --- |

Обычно говорят, что при этом функция g(n) является асимптотически точной оценкой функции f(n), т.к. по определению функция f(n) не отличается от функции g(n) с точностью до постоянного множителя.  
Отметим, что из f(n) = (g(n)) следует, что g(n) = (f(n)).  
Примеры:  
1) f(n)=4+nlnN+174 – f(n)= ();  
2) f(n)=(1) – запись означает, что f(n) или равна константе, не равной нулю, или f(n) ограничена константой на  : f(n) = 7+1/n = (1).  
2. Оценка О (О большое)  
В отличие от оценки , оценка О требует только, что бы функция f(n) не превышала g(n) начиная с n > n0, с точностью до постоянного множителя:



|  | c > 0, n0 > 0 : 0 =< f(n) =< c \* g(n), n > n0 |
| --- | --- |

Вообще, запись O(g(n)) обозначает класс функций, таких, что все они растут не быстрее, чем функция g(n) с точностью до постоянного множителя, поэтому иногда говорят, что g(n) мажорирует функцию f(n).  
Например, для всех функций:  
f(n)=1/n, f(n)= 12, f(n)=3\*n+17, f(n)=n\*Ln(n), f(n)=6\*  +24\*n+77 будет справедлива оценка О()  
Указывая оценку О есть смысл указывать наиболее «близкую» мажорирующую функцию, поскольку например для f(n)=  справедлива оценка О(), однако она не имеет практического смысла.  
3. Оценка  (Омега)  
В отличие от оценки О, оценка  является оценкой снизу – т.е. определяет класс функций, которые растут не медленнее, чем g(n) с точностью до постоянного множителя:



|  | c > 0, n0 >0 : 0 =< c \* g(n) =< f(n) |
| --- | --- |

Например, запись (n\*Ln(n)) обозначает класс функций, которые растут не медленнее, чем g(n) = n\*Ln(n), в этот класс попадают все полиномы со степенью большей единицы, равно как и все степенные функции с основанием большим единицы.  
Асимптотическое обозначение О восходит к учебнику Бахмана по теории простых чисел (Bachman, 1892), обозначения ,  введены Д. Кнутом (Donald Knuth).  
Отметим, что не всегда для пары функций справедливо одно из асимптотических соотношений, например для f(n)=  и g(n)=n не выполняется ни одно из асимптотических соотношений.  
В асимптотическом анализе алгоритмов разработаны специальные методы получения асимптотических оценок, особенно для класса рекурсивных алгоритмов. Очевидно, что  оценка является более прдпочтительной, чем оценка О. Знание асимптотики поведения функции трудоемкости алгоритма - его сложности, дает возможность делать прогнозы по выбору более рационального с точки зрения трудоемкости алгоритма для больших размерностей исходных данных.

# Анализ алгоритмов с примерами на языке Pascal

учеб. пособие / сост. А.В. Мезенцев. – Иркутск: ИГУ, 2015. [Электронный ресурс]. – Режим доступа: <http://old.math.isu.ru/ru/chairs/it/files/analysis.pdf>, свободный.

1. Анализ не рекурсивных алгоритмов .

Анализ алгоритма можно определить как поиск оценок затрачиваемых на выполнение алгоритма памяти и времени. Наиболее часто анализ алгоритмов используется либо для того, чтобы сравнить два различных алгоритма решения одной и той же задачи, либо для того , чтобы выявить практическую применимость алгоритма.

Оценка алгоритма по памяти достаточно прозрачна, и здесь мы ее рассматривать не будем.

Сосредоточимся на оценке алгоритмов по времени выполнения. Один из подходов к оценке алгоритмов по времени выполнения заключается в том, чтобы просто запускать алгоритм на компьютере и засекать тем или иным образом время его выполнения.

У этого подхода есть много недостатков. Во-первых, время выполнения сильно зависит от компьютера, на котором выполняется алгоритм. Во-вторых, такая оценка дает только одно значение для конкретной размерности входных данных. Даже если у нас есть целая таблица оценок для различных размерностей, получить по ней функциональную зависимость времени выполнения от размерности входных данных весьма проблематично (то есть мы не сможем получить оценку для произвольной размерности). В-третьих, такая оценка зависит от реализации (например, от того какой программист будет кодировать этот алгоритм).

Поэтому для оценки алгоритма по времени выполнения стараются найти функциональную зависимость количества выполняемых элементарных операций от размерности входных данных.

Рассмотрим оценку алгоритма на примере.

Пусть у нас есть следующий алгоритм, вычисляющий сумму значений всех компонентов одномерного массива:

(1) s:=0;

(2) for i:=1 to n do

(3) s:= s+a[i];

Алгоритм надо записать таким образом, чтобы в одной строке был один оператор. Далее, рядом с каждым выполняемым оператором надо записать выражение, зависящее от размерности входных данных указывающее то количество раз, которое этот оператор будет выполняться. Эта оценка может быть более или менее точной, главное чтобы вы выполняли ее однообразно. Например, можно считать, что каждый оператор выполняется за одну абстрактную единицу времени. Или разбивать выполнение каждого оператора на последовательность выполнений элементарных операций: прочитать из памяти, записать в память, выполнить арифметическую операцию.

При первом подходе мы получим следующие оценки. Первый оператор будет выполнен один раз и это не зависит от размерности входных данных. Количество выполнений второго оператора зависит от размерности входных данных (конкретно от длины массива n). В нашем случае это n+1 (не забываем о том, что заголовок цикла for выполняется на единицу больше раз, чем его тело). Соответственно третий оператор будет выполняться за n абстрактных единиц времени. Таким образом, имеем:

(1) s:=0; 1

(2) for i:=1 to n do n+1

(3) s:= s+a[i]; n

Осталось просуммировать оценки всех операторов и получить оценку алгоритма 2n+2.

При втором подходе имеем:

(1) s:=0; 1

(2) for i:=1 to n do 6(n+1)

(3) s:= s+a[i]; 7n

Здесь оценка первого оператора осталась такой же, так как надо выполнить только одну элементарную операцию: записать в память s значение константы 0. Второй оператор разбивается на следующие элементарные операции: прочитать из памяти i; прочитать из памяти значение шага; выполнить сложение i и шага; записать в память i; прочитать из памяти n; сравнить значения i и n. Итого 6 операций. Третий оператор разбивается на семь операций: прочитать из памяти i; прочитать из памяти адрес начала массива a; сложить эти два адреса; прочитать по этому адресу a[i]; прочитать из памяти s; сложить; записать в память s. Итого 7 операций. Оценка алгоритма – 13n+7.

Кажется, что эти два подхода дают нам две различные оценки одного и того же алгоритма. На самом деле в анализе алгоритмов в большинстве случаев используется еще более грубый подход. В качестве оценки берется выражение O(f(n)). Где в качестве f(n) берется одна из стандартных функций используемых в анализе алгоритмов, n – размерность входных данных.

Напомним, что g(n) = O(f(n)), когда

lim f (n) = C , где С произвольная

n®¥

g(n)

константа. Поэтому и первый, и второй подход дают нам одну и ту же оценку

O(n). Более точные оценки используются только когда сравнивают различные алгоритмы решения одной и той же задачи.

Наиболее часто в оценке алгоритмов используются следующие функции:

log2 n, n, n×log2 n, n2, n3, 2n, 10n, n!. Алгоритмы, имеющие оценку O(log(n))

,неважно по какому основанию, называются очень быстрыми алгоритмами. Таких алгоритмов существует не очень много. На самом деле в литературе упоминается обычно только один алгоритм, имеющий оценку O(log2 n) – это алгоритм бинарного поиска. Мы рассмотрим его позже. Алгоритмы, имеющие оценку O(n) и O(n·log n) называются быстрыми алгоритмами.

Алгоритмы имеющие оценку O(n2), O(n3), или в общем случае O(nC), где С произвольная положительная константа, называются полиномиальными алгоритмами. Наконец, алгоритмы, имеющие оценку O(2n), O(10n), O(n!) называются неполиномиальными алгоритмами.

Если алгоритм, который вы разработали, является быстрым или очень быстрым, то вам крупно повезло. Это очень эффективный алгоритм и его можно применять к данным практически неограниченной размерности.

Если алгоритм является полиномиальным, тоже не плохо. Такие алгоритмы широко используются и могут применяться к данным с практически важными размерностями.

Если же алгоритм является неполиномиальным, то это значит, что решение практически важных задач с помощью такого алгоритма найти будет невозможно.

В следующей таблице приводятся значения этих функций для некоторых аргументов:

| n | 1 | 2 | 5 | 10 | 20 | 30 | 50 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| log10 n | 0 | 0,30103 | 0,69897 | 1 | 1,30103 | 1,477121 | 1,69897 |
| ln n | 0 | 0,693147 | 1,609438 | 2,302585 | 2,995732 | 3,401197 | 3,912023 |
| log2 n | 0 | 1 | 2,321928 | 3,321928 | 4,321928 | 4,906891 | 5,643856 |
| n | 1 | 2 | 5 | 10 | 20 | 30 | 50 |
| n ln n | 0 | 1,386294 | 8,04719 | 23,02585 | 59,91465 | 102,0359 | 195,6012 |
| n2 | 1 | 4 | 25 | 100 | 400 | 900 | 2500 |
| n3 | 1 | 8 | 125 | 1000 | 8000 | 27000 | 125000 |
| 2n | 2 | 4 | 32 | 1024 | 1048576 | 1,07·109 | 1,13·1015 |
| 10n | 10 | 100 | 100000 | 1010 | 1020 | 1030 | 1050 |
| n! | 1 | 2 | 120 | 3628800 | 2,43·1018 | 2,65·1032 | 3,04·1064 |
| nn | 1 | 4 | 3125 | 1010 | 1026 | 2·1044 | 8,88·1084 |

Для того чтобы было понятно, что скрывается за этими абстрактными числами типа 1010 или 1030 рассмотрим еще одну таблицу.

Будем считать, что у нас имеются два компьютера, первый из которых может выполнить 106 операций в секунду, а второй 109 операций в секунду. Заметим, что под операциями здесь мы понимаем операции, время выполнение которых в оценке алгоритмов считается за абстрактную единицу. То есть быстродействие наших абстрактных компьютеров несравнимо с тактовой частотой современных компьютеров. Вряд ли самый современный компьютер может быть быстрее, чем наш второй абстрактный.

|  | Первый компьютер | | Второй компьютер | |
| --- | --- | --- | --- | --- |
| 1010 | 104 сек. | 2,7 часа | 10 сек. | 10 сек. |
| 1012 | 106 сек. | 10 дней | 103 сек. | 17 мин. |
| 1015 | 109 сек. | 31 год | 106 сек. | 10 дней |
| 1018 | 1012 сек. | 31710 лет | 109 сек. | 31 год |
| 1020 | 1014 сек. | 3 млн. лет | 1011 сек. | 3171 год |
| 1030 | 1024 сек. | 31 000 000  млрд. лет | 1021 сек. | 31 000  млрд. лет |

В первом столбце таблицы приводятся различные значения количества операций, которые необходимо выполнить для завершения алгоритма. Мы видим что, начиная с 1018 для первого компьютера и 1020 для второго, время выполнения алгоритмов становятся совершенно нереальными. А ведь такие числа получаются для алгоритмов с оценкой O(n!) всего-навсего при n=20. Размерности практически важных задач обычно гораздо больше.

Рассмотрим теперь примеры алгоритмов, оценка которых зависит не только от размерности входных данных. Это широко распространенный алгоритм нахождения наибольшего (наименьшего) значения среди компонентов одномерного массива.

(1) max:=a[i]; 1

(2) for i:=2 to n do n

(3) if max<a[i] then n-1

(4) max:=a[i]; от 0 до n-1

Нахождение оценок 1, 2 и 3 операторов не должно вызвать затруднений. А вот количество выполнений 4 оператора зависит от конкретных значений компонентов массива, поэтому мы не можем дать однозначной оценки. В этом случае поступают следующим образом. Дают не одну оценку, а три: наилучшую, наихудшую и среднюю. Из этих трех оценок сложнее всего найти среднюю (даже сформулировать что значит средняя), хотя она с практической точки зрения и является наиболее важной. Для студентов первого курса это, пожалуй, является трудно разрешимой задачей, поэтому здесь мы ее рассматривать не будем.

Что касается наилучшей и наихудшей оценок, то их искать проще. Надо представить себе такие входные данные, для которых соответствующий оператор будет выполняться наименьшее и наибольшее количество раз соответственно.

Для нашего примера наилучшими входными данными будет такой массив, в котором максимум стоит на первом месте. В этом случае 4 оператор ни разу не выполнится, так как условие в 3 операторе будет все время false. Наихудшими входными данными будет такой массив, в котором компоненты упорядочены по возрастанию. В этом случае условие в 3 операторе каждый раз будет true, и оператор 4 будет выполняться каждый раз.

Таким образом, лучшая оценка нашего алгоритма равна 2n, а худшая – 3n-1.

В качестве примера оценки более сложных алгоритмов содержащих вложенные циклы рассмотрим два самых простых, и обычно приходящих в голову первыми, алгоритма сортировки.

Условие задачи: дан массив целых чисел а1,а2, …,аn. Упорядочить компоненты массива по неубыванию.

Первый алгоритм называется алгоритмом «сортировки методом выбора».

Суть его в следующем. Во всем массиве ищется минимум и ставится на первое место, в оставшейся части массива ищется минимум и ставится на второе место и так далее, пока не рассмотренным не останется последний элемент массива. Этот алгоритм используют люди в повседневной жизни. Априори этот алгоритм кажется достаточно «плохим», то есть не эффективным. Рассмотрим этот алгоритм подробнее:

(1) for i:=1 to n-1 do begin n

(2) min:=a[i]; n-1

(3) k:=i; n-1

(4) for j:=i+1 to n do (n2+n-2)/2

(5) if min>a[j] then begin (n2-n)/2

(6) min:=a[j]; от 0 до (n2-n)/2

(7) k:=j от 0 до (n2-n)/2

(8) end;

(9) a[k]:=a[i]; n-1

(10) a[i]:=min n-1

(11) end

Оценка 4 оператора получена следующим образом. Этот заголовок цикла выполняется для каждого i, причем для i =1 он выполнится n раз, для i =2 – n-1 раз, и так далее, при i=n-2 – 3 раза, при i = n-1 – 2 раза. То есть мы имеем арифметическую прогрессию a1,a2, … ,am, сумма которой

s = (a1 + am ) × m .

m 2

Здесь m=n-1, a1=n, an-1=2 и сумма равна (n+2)(n-1)/2 = (n2+n-2)/2.

Оценка 5 оператора получена подобным же образом, только не надо забывать о том, что заголовок цикла for всегда выполняется на один раз больше чем его тело. Здесь m=n-1, a1=n-1, an-1=1 и сумма равна (n-1+1)(n-1)/2 = (n2-n)/2.

Что касается оценки 5 и 6 операторов, то с этим мы уже сталкивались ранее. В зависимости от входных данных эти операторы могут выполняться различное количество раз. Для «хороших» данных (массив уже упорядочен) эти операторы не выполнятся ни разу. Для «плохих» данных эти операторы будут выполняться при каждой проверке min>a[j]. Для алгоритмов сортировки «плохие» данные представляют собой массив, упорядоченный в обратном порядке.

Таким образом, лучшая оценка алгоритма «сортировки методом выбора» равна n2+5n-5, худшая оценка равна 2n2+4n-5. Плохо это, или хорошо? Пока можно заметить только, что алгоритм тратит достаточно много времени, когда массив уже упорядочен. Это несомненный минус этого алгоритма.

Рассмотрим теперь другой алгоритм, который называется алгоритмом

«сортировки методом пузырька». Суть его заключается в том, что сравнивается пара расположенных рядом элементов массива и, если они стоят не месте, они меняются местами. Проблема заключается в том, что за один просмотр всех пар массив можно и не упорядочить. Поэтому требуется еще один как бы «холостой» прогон. Кажется, что этот алгоритм намного лучше предыдущего, но давайте не будем торопиться. Оценим алгоритм

«сортировки методом пузырька»:

лучший худший

(1) do

(2) f:=false; 1 n

(3) for i:=1 to n-1 do n n2

(4) if a[i]>a[i+1] then begin n-1 n(n-1)

(5) t:=a[i]; 0 n(n-1)

| (6) | a[i]:=a[i+1]; | 0 | n(n-1) |
| --- | --- | --- | --- |
| (7) | a[i+1]:=t; | 0 | n(n-1) |
| (8) | f:=true | 0 | n(n-1) |
| (9) | end |  |  |

(10)until not(f); 1 n

Лучшая оценка алгоритма «сортировки методом пузырька» равна 2n+1. Худшая оценка – 6n2-4n. Теперь мы можем сравнить эти два алгоритма решающие одну и ту же задачу. Метод пузырька для «хороших» данных работает лучше, но для «плохих» данных он работает почти в три раза дольше, чем метод выбора.

Таким образом, выбор того или иного алгоритма сортировки зависит от того, какие входные данные вы ожидаете для своего алгоритма. Если предполагается, что массивы будут отсортированы или почти отсортированы, то следует отдать предпочтение методу пузырька. Если предполагается, что массивы будут отсортированы или почти отсортированы в обратном порядке, то следует отдать предпочтение методу выбора. Следует заметить, что для любых самых быстрых алгоритмов сортировки оценка в худшем случае O(n2).

2. Оценка алгоритмов, количество итераций которых не задано явно

Рассмотрим следующую задачу:

Вычислить сумму бесконечного сходящегося ряда

x x 2 xn

1 + + + ... +

+ ... .

1! 2! n!

Легко видеть, что это разложение в ряд широко известной экспоненциальной

функции ( e x ). Ряд сходящийся, то есть каждое очередное слагаемое должно

быть меньше по абсолютной величине предыдущего. Это значит, что всегда найдется такое слагаемое, начиная с которого все остальные будут добавлять

к сумме пренебрежимо малую величину. Таким образом, мы можем написать алгоритм, вычисляющий искомую сумму с любой заданной точностью.

Например, если заданная точность 0,01, то все слагаемые, значения которых меньше 0,01, мы отбросим, и наша вычисленная сумма будет отличаться от истинной не больше чем на 0,01.

Для решения этой задачи можно составить два алгоритма. Первый представляет собой «лобовое» решение. Степень и факториал вычисляются явно:

y:=0;

p:=1;

n:=1;

while Abs(p)<=eps do begin y:=y+p;

f:=1;

for i:=1 to n do f:=f\*I;

p:=Power(x, n)/f; n:=n+1;

end;

Второй алгоритм более «хитрый». Он учитывает тот факт, что для того чтобы получить второе слагаемое, достаточно первое слагаемое умножить на х и разделить на 1. Для того чтобы получить третье слагаемое достаточно второе слагаемое умножить на х и разделить на 2, и так далее.

y:=0;

p:=1;

n:=1;

while Abs(p)<=eps do begin y:=y+p;

p:=p\*x/n; n:=n+1;

end;

Даже на первый взгляд второй алгоритм кажется более эффективным, чем первый. Давайте оценим это точно.

При оценке подобных алгоритмов мы сталкиваемся с одной проблемой. Дело в том, что мы не можем знать заранее, сколько раз будет выполняться тело цикла в этом алгоритме. Мы не можем вычислить это по заданной точности eps, все зависит от конкретного ряда. Проблема кажется трудно разрешимой, однако заметим, что в обоих алгоритмах для одинаковых входных данных циклы выполняются одинаковое количество раз. А так как нам надо лишь выяснить во сколько раз один алгоритм эффективнее другого, то можно просто задать какое-либо абстрактное количество итераций цикла. Тогда алгоритмы будут выглядеть так.

Первый алгоритм:

y:=0;

p:=2;

for n:=1 to m do begin y:=y+p;

f:=1;

for i:=1 to n do f:=f\*I;

p:=Power(x, n)/f; end;

Второй алгоритм:

y:=0;

p:=2;

for n:=1 to m do begin

y:=y+p; p:=p\*x/n;

end;

Оценим первый алгоритм:

y:=0; 1

p:=1; 1

for n:=1 to m do begin m+1

y:=y+p; m

f:=1; m

for i:=1 to n do (m2+3m)/2

f:=f\*I; (m2+m)/2

p:=Power(x, n)/f; m

end;

Суммарная оценка: m2+6m+3.

Оценка достаточно грубая, в частности мы не учитываем тот факт, что умножение требует больше времени, чем сложение, а возведение в степень – больше чем умножение.

Оценим второй алгоритм:

y:=0; 1

p:=1; 1

for n:=1 to m do begin m+1

y:=y+p; m

p:=p\*x/n; m

end;

Суммарная оценка: 3m+3.

Даже с учетом того, что мы проигнорировали разницу в выполнении операций возведения в степень (первый алгоритм) и умножения (второй алгоритм), второй алгоритм является гораздо более эффективным. Например, при m=10 оценка первого алгоритма 163, а второго 33. То есть, второй алгоритм выполнится почти в пять раз быстрее. При m=20 эти оценки равны 523 и 63 соответственно. Второй алгоритм быстрее почти в девять раз.

Умножение двух матриц

Рассмотрим задачу, алгоритм решения которой имеет оценку O(n3). Это задача умножения двух матриц. Требуется вычислить произведение двух матриц: a размерности n´k и b размерности k´m. Результатом будет матрица c размерности n´m.

for i:=1 to n do n+1

for j:=1 to m do begin n×(m+1)

c[i,j]:=0; n×m

for ii:=1 to k do n×m×(k+1)

c[i,j]:=a[i,ii]\*b[ii,j]; n×m×k

end;

Суммарной оценкой будет - 2×n×m×k+3×n×m+2×n+1, т.е. O(n×m×k).

Если же обе исходные матрицы будут квадратными, т.е. n=m=k, то суммарная оценка алгоритма будет - 2×n3+3×n2+2×n+1, или более грубо - O(n3).

3. Анализ рекурсивных алгоритмов

Рассмотрим рекурсивный алгоритм, вычисляющий сумму компонентов одномерного массива.

function sum(a: array of integer, n: integer): integer; begin

if n<1 then

sum:=0

else

sum:=sum(a, n-1)+a[n]

end

Основная операция – суммирование. При изменении аргумента на 1 количество выполненных основных операций равно 1. При значении аргумента равным 0 (n<1), количество выполненных основных операций равно 0. Таким образом, мы получаем следующее рекуррентное соотношение

S(n) = S(n-1) + 1, с начальным условием

S(0) = 0.

Решаем это рекуррентное соотношение методом обратной подстановки. Прямая подстановка:

S(n-1) = S(n-2) + 1 S(n-2) = S(n-3) + 1 S(n-3) = S(n-4) + 1

… и т. д.

Теперь подставляем найденные значения обратно в исходное соотношение:

S(n) = (S(n-2) + 1) + 1 =

((S(n-3) + 1) + 1) + 1 =

(((S(n-4) + 1) + 1) + 1) + 1) = S(n-4) + 4

Или в общем виде, для произвольного k - S(n) = S(n-k) + k.

Для того, чтобы воспользоваться начальными условиями, аргумент у S в правой части должен быть равным 0. То есть, n-k = 0. Отсюда k = n.

S(n) = S(n-k) + k = S(n-n) + n = S(0) + n = 0 + n = n

Оценка эффективности нашего алгоритма по времени – O(n).

Рекурсивный алгоритм нахождения наибольшего значения среди компонентов одномерного массива.

int max(int [] a, int n){ if(n<=1)

return a[n]; else

if(max(a, n-1)>a[n]) return max(a, n-1);

else

return a[n];

}

Основная операция – сравнение (max(a, n-1)>a[n]). Заметим, что время работы нашего алгоритма зависит не только от размерности, поэтому будем строить худшую оценку Cw.

В худшем случае, при изменении значения размерности на 1, задача распадается на две, меньшей размерности, плюс одно сравнение. Получаем следующее рекуррентное соотношение:

Cw(n) = 2×Cw(n-1) + 1, с начальным условием:

Cw(1) = 0.

Решаем рекуррентное соотношение методом обратной подстановки.

Прямая подстановка: Cw(n-1) = 2×Cw(n-2) + 1 Cw(n-2) = 2×Cw(n-3) + 1 Cw(n-3) = 2×Cw(n-4) + 1

… и т. д.

Теперь подставляем найденные значения обратно в исходное соотношение:

Cw(n) = 2×(2×Cw(n-2) + 1) + 1 =

= 2×(2×(2×Cw(n-3) + 1) + 1) + 1 =

= 2×(2×(2×(2×Cw(n-4) + 1) + 1) + 1) + 1 =

= 24×Cw(n-4) +23 + 22 + 21 + 20

Или в общем виде, для произвольного k:

Cw(n) = 2k×Cw(n-k) +2k-1 + 2k-2 + … + 21 + 20

Для того, чтобы воспользоваться начальными условиями, аргумент у Cw в правой части должен быть равным 1. То есть, n-k = 1. Отсюда k = n-1.

Cw(n) = 2k×Cw(1) +2n-2 + 2n-3 + … + 21 + 20

Cw(1) = 0, таким образом

Cw(n) = 2n-2 + 2n-3 + … + 21 + 20

В правой части – сумма степеней 2: 2n-2 + 2n-3 + … + 21 + 20 = 2n-1 - 1

Cw(n) = 2n-1 - 1

Оценка эффективности нашего алгоритма по времени - O(2n). Это очень

«плохая» оценка. Но почему оценка именно такая? Дело в том, что в худшем случае мы два раза рекурсивно вызываем экземпляр нашего алгоритма с

одним и тем же аргументом! Если его вызывать один раз, и сохранять результат, можно значительно улучшить наш алгоритм.

int max(int [] a, int n){ int m;

if(n<=1)

return a[n]; else{

m= max(a, n-1);

if(m>a[n])

return m; else

return a[n];

}

}

Основная операция по прежнему – сравнение (m>a[n]).

В худшем случае, при изменении значения размерности на 1, требуется один раз решить задачу меньшей размерности и выполнить одно сравнение. Получаем следующее рекуррентное соотношение:

Cw(n) = Cw(n-1) + 1, с начальным условием:

Cw(1) = 0.

Такое рекуррентное соотношение мы уже решали выше и знаем ответ:

Cw(n) = n

Оценка эффективности улучшенного алгоритма по времени - O(n).

Задача поиска.

Рассмотрим алгоритмы решения задачи поиска. Дан массив ключей

a1, a2, …, an и искомое значения ключа b, требуется выяснить, существует ли

в массиве ключей ключ с таким значением, и, если он есть, вычислить его индекс в массиве ключей.

Первый алгоритм называется алгоритмом последовательным поиском. Поочередно сравниваем b с элементами массива a1, a2, …, an. Обнаружив совпадение, мы возвращаем индекс найденного элемента; если же элемент не найден, то возвращаем -1.

ind:=-1;

for i:=1 to n do

if a[i]=b then begin ind:=I;

break end;

Достаточно понятно, что придется проверить в худшем случае n элементов, в среднем n/2. Таким образом, сложность (эффективность) алгоритма последовательного поиска в худшем случае – O(n).

Второй алгоритм называется алгоритмом бинарного (двоичного,

дихотомического) поиска.

Предполагается, что массив ключей отсортирован по возрастанию. При этом условии можно значительно повысить эффективность алгоритма. На каждой итерации алгоритма мы будем вычислять средний индекс в той части массива ключей, в которой предполагается искомое значение. Сравниваем этот средний элемент массива с искомым значением. Может быть три варианта. Первый – они равны; искомый индекс найден. Второй – элемент массива меньше искомого значения; необходимо продолжить поиск в правой половине массива. Третий – элемент массива больше искомого значения; необходимо продолжить поиск в левой половине массива. Так как на каждой итерации длина той части массива, в которой осуществляется поиск, уменьшается в два раза, поиск достаточно быстро завершится.

l:=1; r:=n; und:=-1; while l<=r do begin

m:=(l+r) div 2;

if a[m]=b then begin ind:=m;

break end;

if b<a[m] then r:=m-1 else l:=m+1 end;

Для оценки эффективности этого алгоритма не годится метод, который мы до этого использовали так как, мы не сможем для каждого из операторов алгоритма указать количество раз выполнения.

Мы будем использовать метод, который разработан в основном для оценки рекурсивных алгоритмов. Хотя наш алгоритм не рекурсивный, в принципе, он имеет рекурсивный характер.

План анализа эффективности рекурсивных алгоритмов

1. Выберите параметр (параметры), по которым будет оцениваться размер входных данных алгоритма.

2. Определите основную операцию алгоритма.

3. Проверьте, зависит ли число выполняемых основных операций только от размера входных данных, Если оно зависит и от других факторов, рассмотрите при необходимости, как меняется эффективность алгоритма для наихудшего, среднего и наилучшего случаев.

4. Составьте рекуррентное соотношение, выражающее количество выполняемых основных операций алгоритма, и укажите соответствующие начальные условия.

5. Найдите решение рекуррентного уравнения или, если это невозможно, определите хотя бы его порядок роста.

Оценим количество сравнений искомого ключа с элементами массива. Из соображений простоты, мы будем считать, что одно сравнение b и a[m] позволяет определить b < a[m], b = a[m], или b > a[m].

Обозначим через Cw(n) – количество сравнений в наихудшем случае. Вообще говоря, существуют два варианта окончания работы алгоритма в худшем случае: первый – массив не содержит искомое значение; второй - b=a[m] и l=r=m.

Поскольку после одного сравнения алгоритм попадает в ту же ситуацию, что и до сравнения, только область поиска становится вдвое меньше, можно записать следующее рекуррентное соотношение (уравнение):

Cw(n) = Cw(ën/2û)+1, если n>1 Cw(1) = 1 (здесь ëxû - округление в меньшую сторону).

Рекуррентные соотношения для оценки сложности алгоритма выводятся непосредственно из вида алгоритма, однако с их помощью нельзя быстро вычислить эту оценку.

Для этого следует привести рекуррентное соотношение к так называемому

замкнутому виду, отказавшись от их рекурсивной природы.

Производится такое приведение посредством последовательных подстановок, позволяющих уловить общий принцип.

Рекуррентные соотношения.

Рекуррентные соотношения для оценки сложности алгоритма выводятся непосредственно из вида алгоритма, однако с их помощью нельзя быстро вычислить эту оценку.

Для этого следует привести рекуррентное соотношение к так называемому

замкнутому виду, отказавшись от их рекурсивной природы.

Производится такое приведение посредством последовательных подстановок, позволяющих уловить общий принцип.

Рекуррентное соотношение задается в одной из двух форм:

1) Когда простых случаев немного: T(n) = 2´T(n-2) -15;

T(2) = 40;

T(n) = 40.

2) Когда количество простых случаев относительно велико:

C*w*(*n*) – количество сравнений искомого ключа с элементами массива длины *n*

в худшем случае.

(1) C*w*(*n*) = C*w*(ë*n*/2û)+1, если *n*>1 C*w*(1) = 1

Положим, сначала, что *n* = 2*k* и решаем (1) методом обратной подстановки. Так как *n* = 2*k*, то при делении n/2 будем получать всегда целое число (надобность в функции ë*n*/2û отпадает):

C*w*(*n*) = C*w*(*n*/2)+1, если *n*>1 C*w*(1) = 1

Осуществляем прямую подстановку: C*w*(*n*/2) = C*w*(*n*/4)+1

C*w*(*n*/4) = C*w*(*n*/8)+1 C*w*(*n*/8) = C*w*(*n*/16)+1 C*w*(*n*/16) = C*w*(*n*/32)+1

…

Теперь делаем обратную подстановку в исходное соотношение: C*w*(*n*) = (C*w*(*n*/4)+1)+1 =

= ((C*w*(*n*/8)+1)+1)+1 =

= (((C*w*(*n*/16)+1)+1)+1)+1 =

= ((((C*w*(*n*/32)+1) +1)+1)+1)+1 = C*w*(*n*/32)+5 = C*w*(*n*/32)+5×*log*22

C*w*(*n*) = C*w*(*n*/32)+5×*log*22 = C*w*(*n*/32)+ *log*225 = C*w*(*n*/32)+ *log*232

Закончить понижение аргумента мы должны при его значении равным 1, то есть *n*/*n* = 1.

C*w*(*n*) = C*w*(*n*/*n*)+ *log*2*n* и так как C*w*(1) = 1 следовательно

(2) C*w*(*n*) = 1+ *log*2*n*

Докажем теперь, что (2) с небольшими изменениями:

(3) C*w*(*n*) = 1+ /*log*2*n*/

Будет решением (1) для любого положительного целого *n*.

Убедимся с помощью непосредственной подстановки, что (3) удовлетворяет

(1) для любого положительного четного *n* = 2×*i* (*i*>0).

Левая часть (1):

Cw(n) = 1+ |log22×i| = 1+|log22+log2i| = 2+|log2i|

Правая часть (1):

C*w*(|*n*/2|)+1 = C*w*(|(2×*i*) /2|)+1 = C*w*(*i*)+1 = (1+ |*log*2*i*|)+1 = 2+|*log*2*i*|

Оба выражения равны, что доказывает наше утверждение.

Докажем теперь, что (3) является решением (1) для любого положительного нечетного *n* = 2×*i*-1 (*i*>0).

Левая часть (1):

C*w*(*n*) = 1+ |*log*2(2×*i*-1)| = *log*22+|*log*2(2×*i*-1)| =

= |*log*22×(2×*i*-1)| = |*log*2 (4×*i*-1)|

Правая часть (1):

C*w*(|*n*/2|)+1 = C*w*(|(2×*i*-1)/2|)+1 = (1+ |*log*2((2×*i*-1)/2)|)+1 =

= |2+*log*2((2×*i*-1)/2)| = | *log*222+*log*2((2×*i*-1)/2)| =

= |*log*2(4×(2×*i*-1)/2)| = |*log*2 (4×*i*-1)|

Оба выражения равны, утверждение доказано.

Таким образом, C*w*(*n*)=1+|*log*2*n*| является решением исходного рекуррентного соотношения для любого целого *n*>0.

1. P, NP и NP-полные задачи.

При изучении вычислительной сложности задач первое, на что следует обратить внимание – может ли данная задача быть решена при помощи некоторого алгоритма за полиномиальное время.

Говорят, что алгоритм решает задачу за полиномиальное время, если его временная эффективность в наихудшем случае принадлежит классу O(p(n)), где p(n) – полином от размера входных данных n.

Задачу, которая может быть решена за полиномиальное время называют легкой, а задачу, которая не может быть решена за полиномиальное время – трудной.

Мы не можем решить трудные задачи за реальное время, за исключением очень малых размеров входных данных.

Хотя время работы может сильно отличаться для различных степеней полинома, имеется очень мало практичных полиномиальных алгоритмов со степенью полинома больше трех. Кроме того, полиномы, обычно не содержат очень больших коэффициентов.

Полиномиальные функции обладают многими удобными свойствами; в частности, как сумма, так и композиция двух полиномов всегда остаются полиномами.

Говоря не строго, задачи, решаемые за полиномиальное время, можно рассматривать как множество, которое в теории вычислительной сложности называется P.

Более формально, в множество P включаются только задачи принятия решения, представляющие собой задачи, ответ на которые – «ДА» либо

«НЕТ».

Класс P представляет собой класс задач принятия решения, которые могут быть решены (детерминистическим) алгоритмом за полиномиальное время. Этот класс задач называется полиномиальным.

Многие важные задачи, не являющиеся задачами принятия решения в своей естественной формулировке, могут быть приведены к ряду задач принятия решения, которые проще изучать.

Например, вместо выяснения, какое наименьшее количество цветов надо для раскраски вершин графа так, чтобы никакие две смежные вершины не были одного цвета, мы можем выяснить, существует ли такая раскраска вершин графа не более чем m цветами при m=1,2,… .Первое значение m в этом ряду для которого задача раскраски имеет решение, дает ответ на оптимизационную задачу раскраски графа.

Возникает вопрос: каждая ли задача принятия решения может быть решена за полиномиальное время? Ответ – нет. Некоторые задачи принятия решения не могут быть решены в принципе, никакими алгоритмами (неразрешимые задачи). Пример – задача останова (Тьюринг, 1936 г.): для данной компьютерной программы и входных данных определить, завершится ли выполнение программы или она будет выполняться бесконечно?

Второй вопрос: существуют ли задачи разрешимые, но трудные? Ответ – да. Но количество известных примеров невелико, в особенности тех, которые возникают естественным путем, а не построены в качестве теоретического доказательства.

Имеется, однако, большое количество важных задач, для которых не найден алгоритм с полиномиальным временем работы, но и не доказана невозможность его существования (несколько сотен из различных областей информатики, математики и исследования операций).

Вот несколько из наиболее известных задач:

1. Гамильтонов цикл. Определить, имеется ли в данном графе гамильтонов цикл (путь, который начинается и заканчивается в одной и той же вершине и проходит по всем остальным вершинам ровно по одному разу).

2. Задача коммивояжера. Найти кратчайший маршрут по n городам с известными расстояниями между ними (найти кратчайший гамильтонов цикл в полном графе с положительными целыми весами).

3. Задача о рюкзаке. Найти подмножество с наибольшей стоимостью из n предметов с заданными положительными целыми весами и стоимостями, которое может быть помещено в рюкзак с заданной положительной целой емкость (грузоподъемностью).

4. Задача о разделении. Даны n положительных целых чисел; требуется определить, можно ли разделить их на два непересекающихся подмножества с одинаковыми суммами.

5. Упаковка корзин. Даны n предметов, размеры которых представляют собой положительные рациональные числа, не превышающие 1. Их надо разместить в наименьшее количество корзин размером 1.

6. Раскраска графа (карты). Для данного графа найти его хроматическое число (наименьшее количество цветов, которыми можно раскрасить вершины графа так, чтобы никакие две смежные вершины не были окрашены в один и тот же цвет).

7. Целочисленное линейное программирование. Найти максимальное (минимальное) значение функции нескольких целочисленных переменных при условии выполнения конечного множества ограничений в виде линейных равенств и/или неравенств.

Некоторые из этих задач являются задачами принятия решения. Общее у всех этих задач то, что они обладают экспоненциальным (или еще более быстрым) ростом количества вариантов выбора решения с ростом размерности задачи n.

Заметим, что имеется ряд задач с очень похожими условиями, но решаемых за полиномиальное время. Например, задача об Эйлеровом цикле, то есть, о существовании цикла, который проходит по всем ребрам данного графа ровно по одному разу, может быть решена за время O(n2) путем проверки связности графа и все ли его вершины имеют четную степень (Задача о мостах в Кенигсберге).

Этот пример поразителен: интуитивно, совершенно не ожидаешь, что задача обхода всех ребер по одному разу (Эйлеров цикл) настолько проще кажущейся похожей задачи о цикле, обходящем по одному разу все вершины (гамильтонов цикл).

Еще одно общее свойство огромного большинства задач принятия решения заключается в том, что при том, что решение задач может быть вычислительно сложным, проверка предложенного решения обычно достаточно проста и может быть выполнена за полиномиальное время (такие решения можно представить как случайным образом генерируемые кем-то и предлагаемые нам для проверки их корректности).

Например, для того чтобы проверить, что предложенный список вершин является гамильтоновым циклом для данного графа с n вершинами, необходимо убедиться, что список содержит n+1 вершину графа, что первые n вершин в списке различны, а последняя совпадает с первой, и что каждая пара соседних вершин в списке соединяется ребром.

Это свойство приводит нас к понятию недетерминистического алгоритма. Недетерминистическим алгоритмом называется двух этапная процедура, которая получает в качестве входа экземпляр I задачи принятия решения и делает следующее:

- недетерминистический этап («угадывание»): генерируется произвольная строка S, которая может рассматриваться как кандидат в решение данной задачи I (но может оказаться и полной ерундой);

- детерминистический этап («проверка»): получает I и S в качестве входных данных и выдает «да», если S является решением I. (Если S не является решением I, алгоритм либо возвращает «нет», либо может вообще не завершить работу).

| Недетерминистический | алгоритм | является недетерминистическим |
| --- | --- | --- |
| полиномиальным, если | временная | эффективность этапа проверки |
| полиномиальная. |  |  |

Класс NP – это класс задач принятия решения, которые могут быть решены недетерминистическим полиномиальным алгоритмом (nondeterministic polynomial).

P Í NP (для P на этапе проверки просто игнорируем строку S, генерируемую на этапе угадывания).

Большинство задач принятия решения принадлежат классу NP. Задача останова не принадлежит классу NP.

Важный открытый вопрос теоретической информатики: является ли класс P

собственным подмножеством NP или эти классы совпадают?

Задача принятия решения D1 называется полиномиально приводимой к задаче принятия решения D2, если имеется функция t, которая преобразует экземпляры D1 в экземпляры D2 так, что

1. t отображает все экземпляры D1 с положительным ответом на экземпляры D2 с положительным ответом, и все экземпляры D1 с отрицательным ответом на экземпляры D2 с отрицательным ответом.

2. t – вычислима при помощи алгоритма с полиномиальным временем работы.

Если задача D1 полиномиально приводима к некоторой задаче D2, которая может быть решена за полиномиальное время, то задача D1 также может быть решена за полиномиальное время.

Задача принятия решения D называется NP-полной, если:

1. она принадлежит классу NP;

2. любая задача в NP полиномиально приводима к D.

Понятие NP-полноты требует полиномиальной приводимости всех задач в

NP, как известных, так и неизвестных, к рассматриваемой задаче.

При огромном количестве задач принятия решения вызывает изумление тот факт, что были найдены конкретные примеры NP-полных задач.

Этого добились независимо друг от друга Стивен Кук (в 1971 году) и Леонид Левин (в 1973 году). Они показали, что задача КНФ-выполнимости является NP-полной. (КНФ (CNF) – конъюнктивная нормальная форма булева выражения). Каждое булево выражение может быть представлено в КНФ. Спрашивается, можно ли назначить в булевом выражении переменным (true или false) так, чтобы результат вычисления всего выражения был true.

С того времени найдены сотни примеров NP-полных задач. Некоторые из них были приведены выше. Известно, однако, что если P≠NP, то должны существовать NP-задачи, которые не являются ни P-задачами, ни NP-полными задачами.

Непосредственно из определения NP-полноты следует, что если будет найден детерминистический алгоритм решения одной из NP-полных задач, то все задачи в NP могут быть решены за полиномиальное время при помощи детерминистического алгоритма, следовательно, P=NP.

Исчерпывающий перебор.

Исчерпывающим перебором называется поиск элемента со специфическими свойствами в области экспоненциально (или еще быстрее) растущей с увеличением размера задачи. Обычно такие задачи возникают в случаях, когда присутствуют – явно или не явно – комбинаторные объекты: перестановки, сочетания, подмножества.

Многие подобные задачи являются задачами оптимизации: в них требуется найти элемент, который максимизирует (минимизирует) некоторую целевую функцию как, например, длина пути или стоимость.

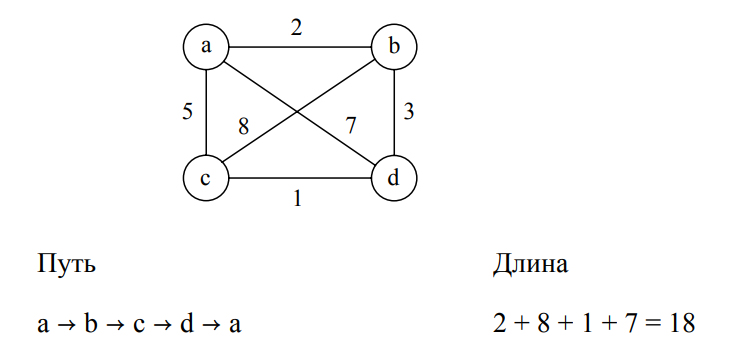
Исчерпывающий перебор представляет собой подход к комбинаторным задачам с позиции грубой силы. Он предполагает генерацию всех возможных элементов из области определения задачи, и последующий поиск нужного элемента (например, оптимизирующего значение целевой функции задачи).

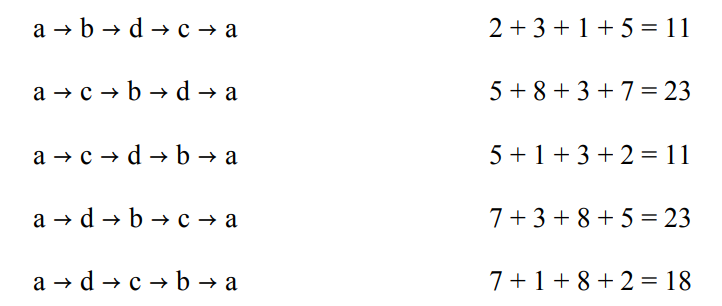
Идея исчерпывающего перебора весьма проста, ее реализация обычно требует алгоритма для генерации определенных комбинаторных объектов.

Задача коммивояжера.

Под задачей коммивояжера понимается поиск кратчайшего гамильтонова цикла неориентированного графа.

Гамильтонов цикл в графе содержащем ровно n вершин можно определить как последовательность n+1 смежных вершин vi0, vi1, …, vin-1, vi0, где первая вершина в последовательности совпадает с последней, в то время как все остальные n–1 вершин различны. Далее без потери общности можно предположить, что все циклы начинаются и заканчиваются в одной конкретной вершине (в конце концов, все они циклы). Значит, можно получить все возможные маршруты, генерируя все перестановки n–1 промежуточных городов, вычисляя длину соответствующих путей и находя кратчайший из них.





При внимательном рассмотрении можно выявить три пары обходов, которые отличаются друг от друга только направлением. Таким образом, можно снизить количество перестановок вершин вдвое.

Можно, например, выбрать две промежуточные вершины, скажем b и c, и рассматривать только те перестановки, в которых b предшествует c (это неявно определяет направление обхода).

Однако это улучшение алгоритма не дает заметного эффекта. Общее количество перестановок остается равным (n−1)!/2, что делает исчерпывающий перебор неприемлемым для всех значений n, кроме самых малых.

Заметим, что если бы не ограничение, чтобы все пути начинались в одной и той же вершине, то количество перестановок было бы в n раз больше.

Задача о рюкзаке.

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

Исчерпывающий перебор в этой задаче приводит к рассмотрению всех подмножеств данного множества из n элементов, вычислению общего веса каждого из них для того, чтобы выяснить, допустим ли такой набор предметов (не превосходит ли его общий вес возможности рюкзака), и выбору из допустимых подмножества с максимальным весом.

Общее количество подмножеств n-элементного множества равно 2n, исчерпывающий перебор приводит к алгоритму со временем работы O(2n), вне зависимости от того, насколько эффективным методом генерируются рассматриваемые подмножества.

Предметы: 1) w = 7 v = 42

2) w = 3 v = 12

3) w = 4 v = 40

4) w = 5 v = 25

Грузоподъемность рюкзака W = 10

| Подмножество | Общий вес | Общая стоимость |  |
| --- | --- | --- | --- |
| ∅ | 0 | 0 |
| {1} | 7 | 42 |
| {2} | 3 | 12 |
| {3} | 4 | 40 |
| {4} | 5 | 25 |
| {1, 2} | 10 | 36 |
| {1, 3} | 11 | недопустим |
| {1, 4} | 12 | недопустим |
| {2, 3} | 7 | 52 |
| {2, 4} | 8 | 37 |
| {3, 4} | 9 | 65 | ответ |
| {1, 2, 3} | 14 | недопустим |  |
| {1, 2, 4} | 15 | недопустим |  |
| {1, 3, 4} | 16 | недопустим |  |
| {2, 3, 4} | 12 | недопустим |  |

{1, 2, 3, 4} 19 недопустим

Задача о назначениях

Имеется n работников, которые должны выполнить n заданий, по одному заданию каждый. Стоимость выполнения i-ым работником j-го задания известна и равна Cij для всех пар i и j.

Надо распределить задания между работниками таким образом, чтобы они были выполнены с наименьшей общей стоимостью.

Легко видеть, что экземпляр задачи с назначением заданий полностью определяется матрицей стоимости C.

В терминах данной матрицы требуется выбрать по одному элементу из каждой строки матрицы так, чтобы выбранные элементы находились в разных столбцах, а их общая сумма имела наименьшее возможное значение.

Заметим, что очевидной стратегии решения данной задачи не существует. Например, нельзя выбирать наименьшие элементы в каждой строке, так как они могут оказаться в одном и том же столбце. В действительности, наименьшие элементы матрицы могут вообще не входить в оптимальное решение. Так что исчерпывающий перебор для решения задачи может оказаться неизбежным.

Допустимое решение можно описать в виде кортежа из n значений <j1, …, jn>, в котором i-ый компонент указывает столбец, где находится выбранный в i-ой строке элемент матрицы.

| Задание 1 | | Задание 2 | Задание 3 | Задание 4 |
| --- | --- | --- | --- | --- |
| Работник 1 | 9 | 2 | 7 | 8 |
| Работник 2 | 6 | 4 | 3 | 7 |
| Работник 3 | 5 | 8 | 1 | 8 |
| Работник 4 | 7 | 6 | 9 | 4 |

Кортеж <2, 3, 4, 1> указывает допустимое назначение:

2 задание работнику 1

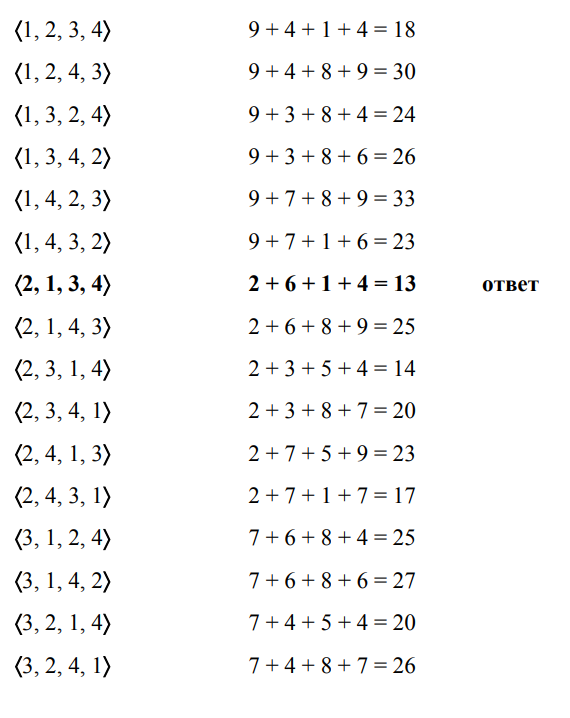
3 задание работнику 2

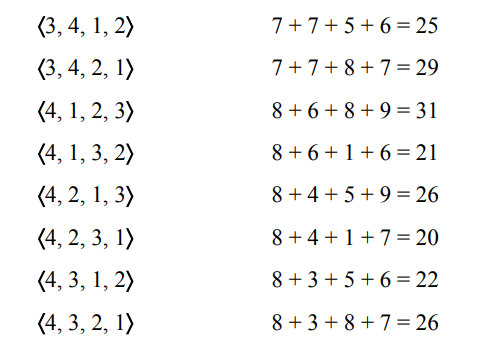
4 задание работнику 3

1 задание работнику 4

Из условия задачи вытекает, что имеется однозначное соответствие между допустимыми назначениями и перестановками первых ***n*** натуральных чисел.

Следовательно, исчерпывающий перебор потребует генерации всех перестановок натуральных чисел 1, 2, …, ***n***, вычисления общей стоимости и выбора назначения с минимальной стоимостью.





5. Алгоритмы генерации комбинаторных объектов.

Наиболее важные типы комбинаторных объектов – перестановки и подмножества данного множества. В математике, в первую очередь, интересуются формулами для подсчета количества таких объектов. Эти формулы предупреждают нас о том, что это количество с увеличением размерности задачи растет экспоненциально или еще быстрее.

Генерация перестановок.

Дано множество натуральных чисел от 1 до n. В общем случае их можно интерпретировать как индексы элементов n-элементного множества

{a1, …, an}.

Нам нужно сгенерировать n! перестановок множества {1, 2, …, n}.

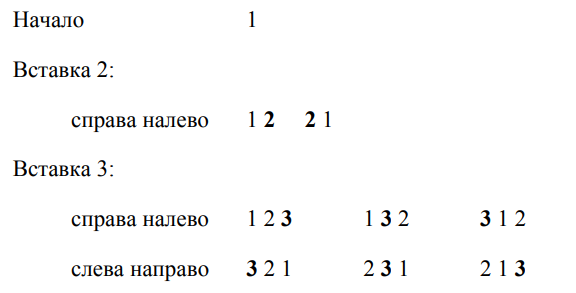
Применим метод уменьшения размерности задачи.

Задача меньшего на 1 размера состоит в генерации всех (n-1)! перестановок. Полагая, что задача меньшего размера успешно решена, мы можем получить решение большей задачи путем вставки n в каждую из n возможных позиций среди элементов каждой из перестановок n-1 элементов.

Мы можем вставлять n в ранее сгенерированные перестановки слева направо или справа налево. Оказывается, выгодно начинать вставку n в

последовательность 1 2 3 … (n-1) справа налево и изменять направление всякий раз при переходе к новой перестановке множества {1, 2, …, n-1}.

Пример:



Преимущество такого порядка генерации – он удовлетворяет требованию минимальных изменений: каждая перестановка получается из своей непосредственной предшественницы при помощи обмена местами только двух элементов.

Требование минимальных изменений выгодно как с точки зрения скорости работы алгоритма, так и для приложения использующего перестановки. Например, в задаче коммивояжера нам нужны перестановки городов. Если такие перестановки генерируются алгоритмом, удовлетворяющим требованию минимальных изменений, то мы можем вычислить длину нового маршрута на основании длины старого маршрута за постоянное, а не линейное время (как ?).

Можно получить тот же порядок перестановок n элементов и без явной генерации перестановок для меньших значений n. Это можно сделать, связав с каждым компонентом k перестановки так называемое направление. Будем указывать это направление при помощи стрелки:

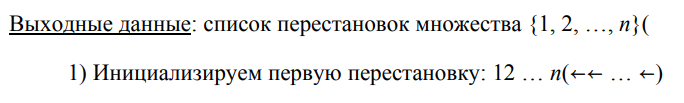


Компонент k в такой перестановке с использованием стрелок называется

мобильным, если стрелка указывает на меньшее соседнее число. Например, числа 3 и 4 мобильны, а 2 и 1 – нет.

Воспользовавшись понятием мобильного компонента, мы получим следующий алгоритм Джонсона-Троттера.

Входные данные: n– натуральное число



2) Пока имеется мобильное число k выполнять

3) Найти наибольшее мобильное число k

4) Поменять местами k и соседнее число, на которое указывает стрелка у k

5) Поменять направлении стрелок у всех чисел, больших k

Пример для n = 3



Этот алгоритм – один из наиболее эффективных для генерации всех перестановок (хотя все они имеют оценку O(n!)).

Порядок перестановок, генерируемых алгоритмом Джонсона-Троттера, не совсем естественный: было бы естественным ожидать, что последняя перестановка в списке будет иметь вид: n (n-1) … 2 1. Именно такая перестановка окажется последней, если перестановки будут упорядочены в соответствии с лексикографическим порядком.

1 2 3 1 3 2 2 1 3 2 3 1 3 1 2 3 2 1

Каким образом мы можем сгенерировать перестановку, следующую за

*a*1, *a*2, …, *a*n-1, *a*n в лексикографическом порядке?

Если an-1 > an, мы должны обратиться к элементу an-2. Если an-2 < an-1, мы должны переставить последние три элемента, минимально увеличивая (n-2)-ой элемент, т.е. помещая на это место следующий больший an-2 элемент, выбранный из an-1 и an, и заполняя позиции (n-1) и (n) оставшимися двумя из трех элементов an-2, an-1, an в возрастающем порядке.

В общем случае мы просматриваем текущую перестановку справа налево в поисках первой пары соседних элементов ai и ai+1 таких, что ai<ai+1 (и, следовательно, ai+1> …>an). Затем мы находим наименьший элемент из

«хвоста», больший ai, т.е. min{aj|aj>ai, j>i}, и помещаем его в позицию i; позиции с i+1-ой по n-ую заполняются элементами ai, ai+1, …, an, из которых изъят элемент для вставки в позицию i, в возрастающем порядке.

Пример.

1 2 3 4 5

1 2 3 5 4

1 2 4 3 5

1 2 4 5 3

1 2 5 3 4

1 2 5 4 3

1 3 2 4 5

1 3 2 5 4

1 3 4 2 5

1 3 4 5 2

1 3 5 2 4

1 3 5 4 2

1 4 2 3 5

и т.д.

Генерация подмножеств.

Рассмотрим алгоритм для генерации всех 2n подмножеств абстрактного множества A = {a1, a2, …, an}.

К этой задаче также применим метод уменьшения размерности на единицу. Все подмножества множества A = {a1, a2, …, an} разделить на две группы – те, которые содержат элемент an, и те, которые не содержат его. Первая группа представляет собой не что иное, как все подмножества множества

{a1, a2, …, an-1}, в то время как все элементы второй группы можно получить путем добавления элемента an к каждому подмножеству множества

{a1, a2, …, an-1}. Следовательно, как только мы получим список всех подмножеств множества {a1, a2, …, an-1}, мы можем получить все подмножества множества {a1, a2, …, an}, просто добавляя к списку все его элементы с добавление к каждому из них элемента an.

Пример 1. Сгенерируем все подмножества множества {a1, a2, a3} n Сгенерированные подмножества

0 ∅

1 ∅ {a1}

2 ∅ {a1} {a2} {a1, a2}

3 ∅ {a1} {a2} {a1, a2} {a3} {a1, a3} {a2, a3} {a1, a2, a3}

Как и в случае перестановок, мы не обязаны генерировать все подмножества для множеств меньших размеров. Удобный способ решения основан на взаимно однозначном соответствии между всеми 2n подмножествами n-элементного множества A = {a1, a2, …, an} и всеми 2n битовыми строками b1b2…bn длины n.

Простейший способ установить такое соответствие – назначить подмножеству битовую строку, в которой bi=1, если ai принадлежит данному подмножеству, и bi=0 в противном случае.

Используя такое соответствие, мы можем сгенерировать все битовые строки длиной n, просто генерируя последовательно двоичные числа от 0 до 2n-1, добавляя при необходимости соответствующее количество ведущих нулей.

Пример 2. n = 3

Битовые строки

000 001 010 011 100 101 110 111

Подмножества

∅ {a3} {a2} {a2,a3} {a1} {a1,a3} {a1,a2} {a1,a2,a3}

Заметим, что в то время, как битовые строки, сгенерированные данных алгоритмом, находятся в лексикографическом порядке (для алфавита {0, 1}), порядок подмножеств выглядит далеко не естественно.

Например, мы можем захотеть получить так называемый плотный порядок, когда подмножество, включающее aj, может находиться в списке только после всех подмножеств, включающих элементы a1, a2, …, aj-1, как, например, в примере 1.

Более интересный вопрос – о существовании алгоритма генерации битовых строк, соответствующих требованию минимальных изменений, так, чтобы каждая строка отличалась от непосредственно предшествующей ей только одним битом (каждое подмножество будет отличаться от своего предшественника в списке либо добавлением, либо удалением одного элемента, но не более).

Это код Грея. При n = 3:

000 001 011 010 110 111 101 100.

Код Грея легко получить:

function Gray(n: integer): integer; begin

Gray := n xor (n shl 1)

end;

Здесь используются две побитовые операции: исключающее или и сдвиг вправо на один разряд.

# Примеры анализа сложности алгоритмов: блог программиста

# 

[Электронный ресурс]. – Режим доступа: <https://pro-prof.com/archives/4275>.

Еще одна статья по анализу алгоритмов?

Давным давно я написал статью, в которой была [параллельная реализация алгоритма быстрой сортировки](https://pro-prof.com/archives/1220) [1]. По ней я получил обратную связь. Одни ребята написали мне, что быстрая сортировка будет работать медленнее, чем их реализация (мне скинули “слегка оптимизированный” алгоритм сортировки пузырьком). В качестве аргумента я получил такую фразу:

“он (алгоритм быстрой сортировки) очевидно медленнее (‘посмотри сколько там кода вообще?’). Чем больше кода – тем медленнее работает, ведь код процессор должен выполнить”.

Позже я опубликовал еще две статьи, в одной из которых описал [математический аппарат, необходимый для асимптотического анализа](https://pro-prof.com/archives/1660) [2], а в другой – [методы анализа рекурсивных алгоритмов](https://pro-prof.com/archives/813) [3] (для них стандартные методы не работают). Очень старался описать все понятно и доступно, был уверен, что любой гуманитарий теперь сможет разобраться в этой теме, но нашлись другие ребята, которые тоже не все поняли. Статью они прочитали, но им не понравилось, что анализ ведется при

n→∞

. Цитирую:

обычный размер массива в программах в районе 50-100 элементов, а на нем (мы запустили и проверили) быстрая сортировка работает медленнее пузырьковой на X миллисекунд, поэтому весь этот анализ – математическое фуфло. Никогда n не будет равно бесконечности, объем памяти ограничен же.

Этими примерами я пытался обрисовать непонятные мне до получения обратной связи, проблемы, которые возникают у многих людей при попытке сравнения двух алгоритмов. Замечу, что самый активный из моих собеседников закончил ВУЗ (на программиста учился), хотя по специальности в тот момент не работал. В статье я попробую представить часть информации в графической форме, возможно картинки убедят кого-нибудь посмотреть уже наконец в мат.часть приведенную в предыдущих статьях.

Статья ориентирована на студентов первых курсов (и выпускников, которым не повезло с преподавателями). Вы не узнаете из нее ничего нового, если понимаете (можете доступно объяснить коллегам) почему у этого фрагмента трудоемкость

Θ(n⋅log(n)):

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

for (int j = 1; j < n; j \*= 2) {

++k;

}

}

А у этого – Θ(n3):

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

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

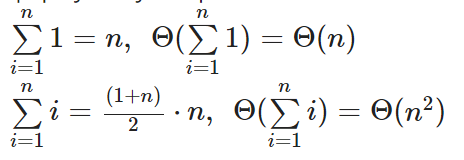
for (int k = i; k < j; ++k) {

++k;

}

}

}

Основные математические сведения, которые понадобятся нам для понимания материала – формулы суммирования:  


Пример 1

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

**for** (**int** j = 1; j < m; ++j) { // A

a[i][j]++; // С

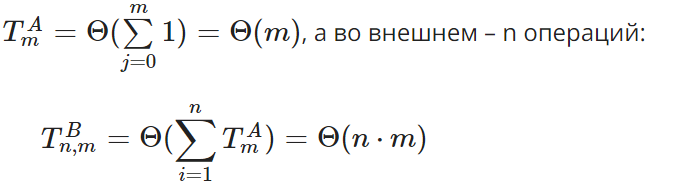
}

}

Трудоемкость операции инкремента значения элемента массива не зависит от размерности задачи (в данном случае n и m), т.е. ее трудоемкость равна

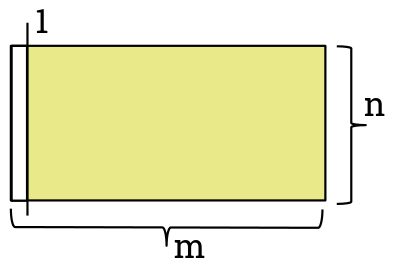
Θ(1)

. Во внутреннем цикле будет выполнено m операций, т.е.



Тут все просто, при этом не случайно в одном цикле начальное значение счетчика равно нулю, а в другом единице (на их месте могли бы стоять другие числа) – на трудоемкость это не влияет. В отличии от Скиены [4], который предлагает своим читателям попробовать посчитать число операций (поэтому вложенный цикл у него содержит что-то типа

k++), я предлагаю перейти к графической интерпретации, поэтому использую массивы.



Графическая интерпретация примера 1

Пример 2

// пример 2.1

**for** (**int** i = 0; i < n-1; ++i) { // B

**for** (**int** j = 0; j < n-i; ++j) { // A

a[i][j]++;

}

}

Если в этом примере заменить

a[i][j]++ на

**if** (a[j] > a[j+1])

swap(a[j], a[j+1])

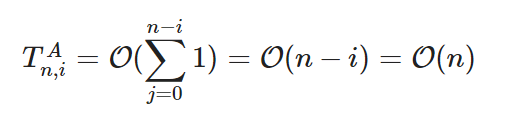
Получится пузырьковая сортировка (дальше используются обозначения

O вместо Θ, т.к. после замены в лучшем случае будет выполнено

Ω(n) операций обмена, подробнее это описано в [2]).

Зачем это тут и чем отличается от примера 1? – Во вложенном цикле теперь перебираются элементы не до

n, а до n-i, при этом i не является константой, а зависит от n, ряд моих читателей уверены, что это изменит асимптотическую сложность алгоритма. На самом же деле:



Последнее действие понятно не всем. Ну в самом деле, считаете вы трудоемкость и у вас получается



, человек, не знакомый с асимптотическим анализом скажет, что в результате получится

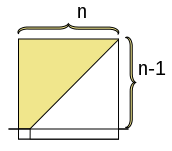
Θ(n). Однако, нельзя забывать, что асимптотические обозначения задают *класс функций*, т.е. нет разницы между



, поэтому вне зависимости от того, что *вычитается*, при верхней оценке трудоемкости берется самая быстро растущая функция. В данном случае это



Таким образом, трудоемкость приведенного алгоритма можно оценить как:  


Графическая интерпретация примера 2.1

По приведенной картинке видно, что обработано будет примерно

(n−1)2/2

элементов матрицы. Из этого примера можно сделать вывод о том, что “игры” с начальным и конечным значением счетчика цикла обычно на трудоемкость не влияют. В частности у такого алгоритма:

// пример 2.2

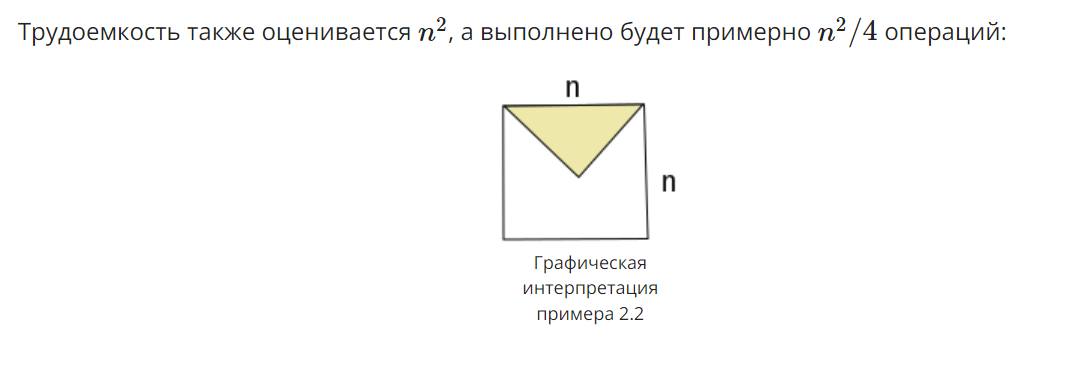
**for** (**int** i = 0; i < n; ++i) { // B

**for** (**int** j = i; j < n-i; ++j) { // A

a[i][j]++;

}

}



Пример 3

Если в примере 1 на каждой итерации цикла увеличивать счетчик не на 1, а скажем, на 2 – оценка трудоемкости также не изменится (будет обработана половина элементов массива). Однако что будет если счетчик изменять в 2 раза?

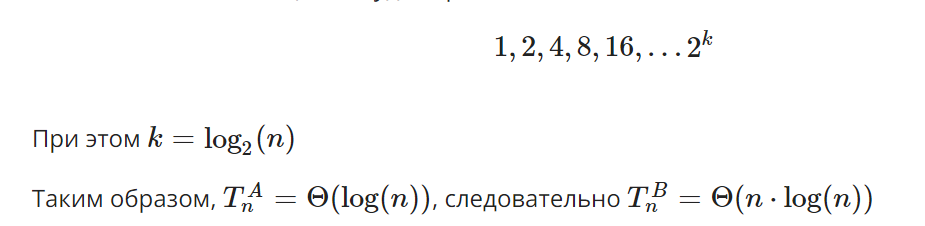
**for** (**int** i = 0; i < n; ++i) { // B

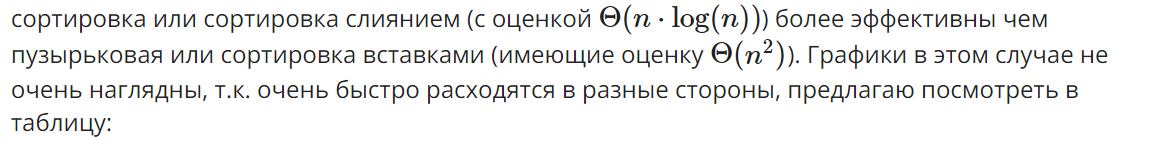
**for** (**int** j = 1; j < n; j \*= 2) { // A

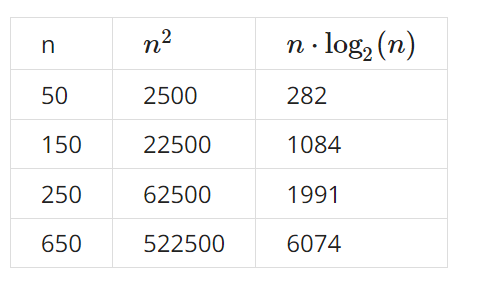
a[i][j]++;

}

}

Счетчик вложенного цикла будет принимать значения:  


Иллюстрация… представьте себе массив у которого обработаны только столбцы, индексы которых рассчитываются по приведенной выше формуле. Очень важно представить себе насколько медленно растет функция логарифма – это поможет понять почему быстрая 

****

Пример 4

Очень надеюсь, что все кто не понимал почему при асимптотическом анализе можно отбрасывать константы (даже очень большие) разберутся посмотрев в приведенную таблицу. Но, что если эти константы являются границами цикла, как в примере?:

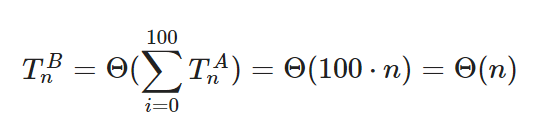
**for** (**int** i = 0; i < 100; ++i) { // B

**for** (**int** j = 1; j < n; ++j) { // A

++count;

}

}



Часто в задачах ограничено n, но это не значит что всегда можно его игнорировать. Так например, Скиена [4] приводит случай из жизни, когда к нему обратились за эффективным алгоритмом, который должен выполнить некоторые расчеты с

0<i<1000000000

. Очевидно, что будь в нашем примере такое большое значение – его нельзя было бы не учитывать (его надо было бы считать за n).

Пример 5



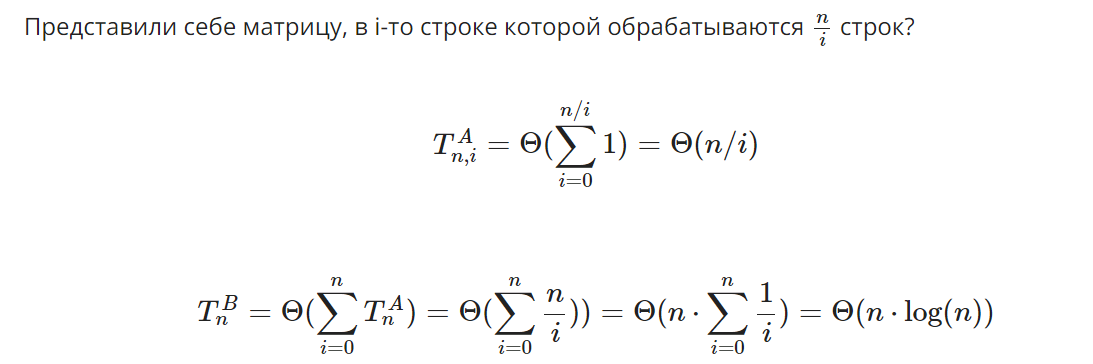
**for** (**int** i = 0; i < n; ++i) { // B

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

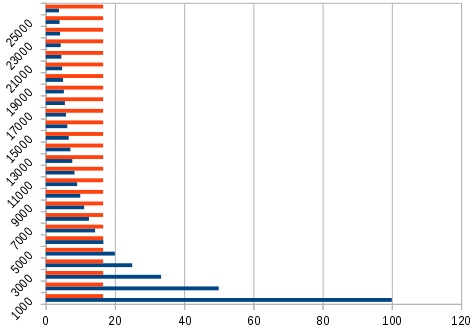
matrx[i][i]++;

}

}



Примеры 5 и 4 на самом деле очень сильно отличаются. В примере 3 каждая итерация вложенного цикла выполняла одинаковое число итераций (их количество log(n)), а в примере 5 для каждой следующей строки матрицы обрабатывалось меньше столбцов, чем для предыдущей. Тем не менее, асимптотические оценки обоих примеров совпадают.



Графическая интерпретация примеров 3 и 5

На рисунке показаны зависимости количества итераций внутреннего цикла от номера итерации внешнего цикла для примеров 4 (оранжевый) и 5 (синий) для

n=100000

. Надеюсь эта картинка поможет понять почему асимптотические оценки совпадают. Синяя функция быстро убывает (может показаться, что оценка этой функции должна быть “лучше”), но для маленький значений i ее значения огромны (именно поэтому на графике начальное значение

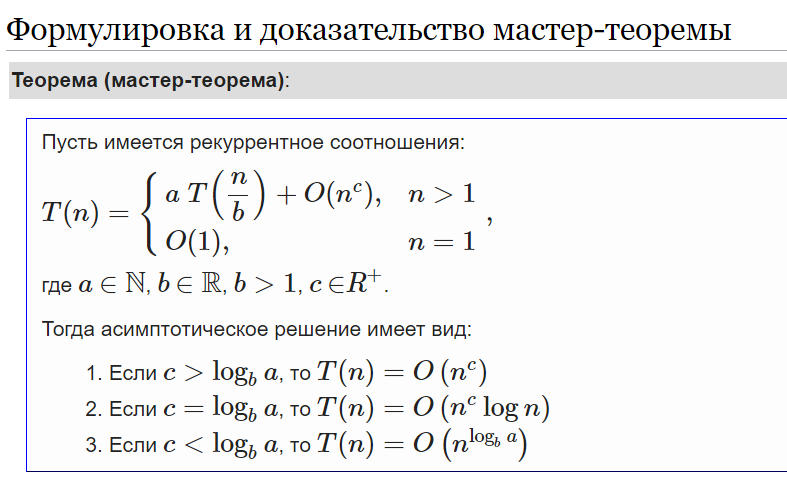
i=1000

.

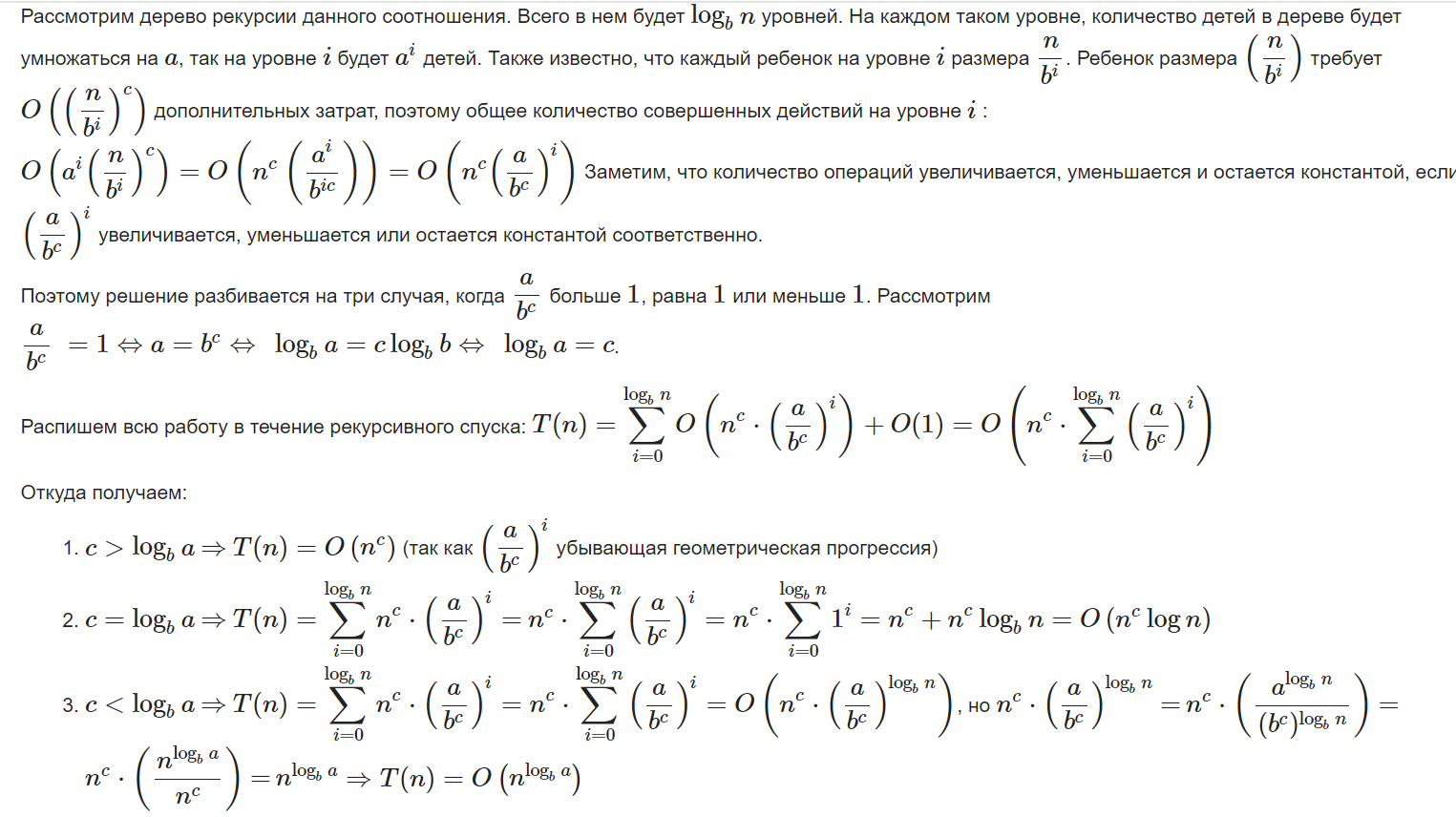
# Мастер-теорема

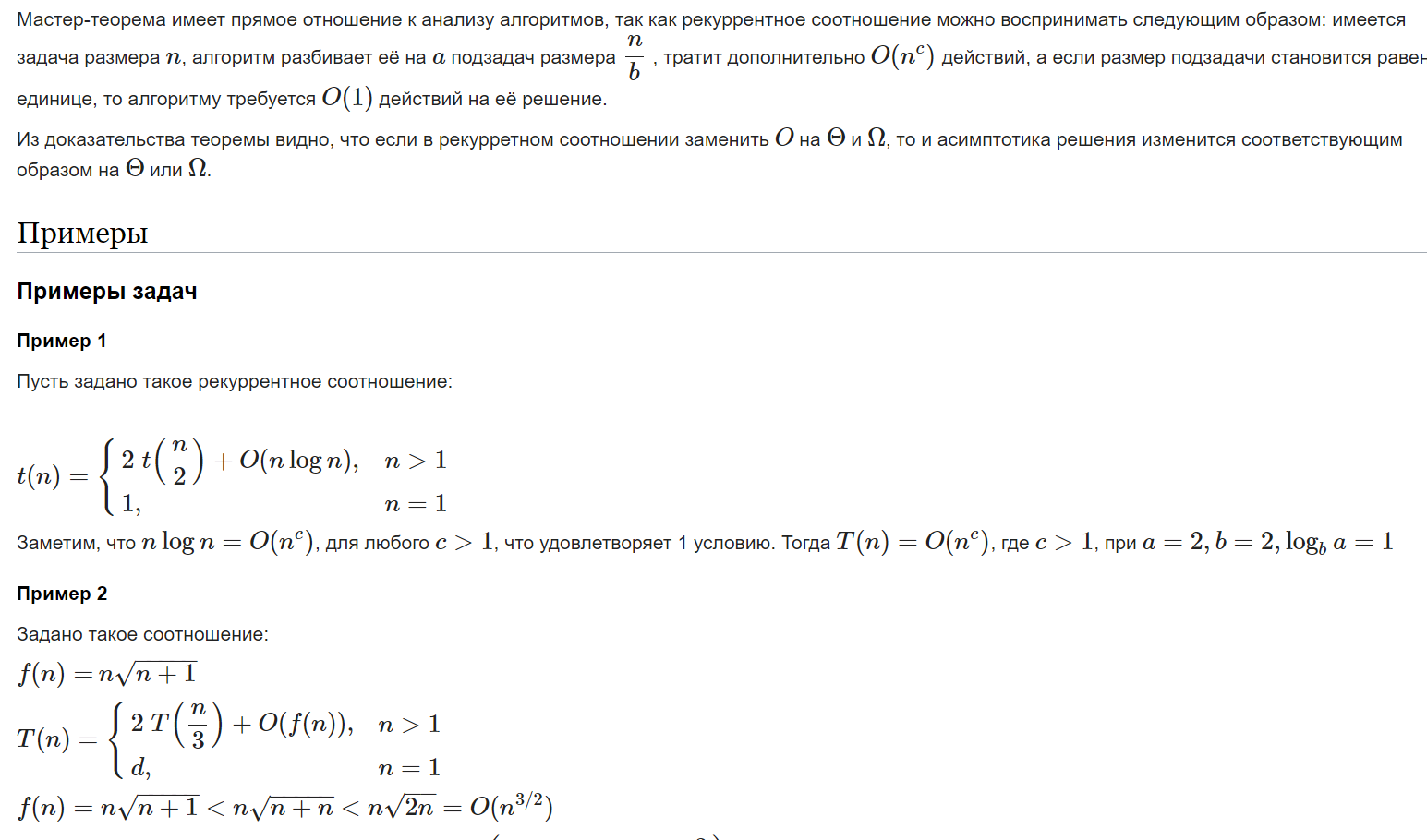
https://neerc.ifmo.ru/wiki/index.php?title=%D0%9C%D0%B0%D1%81%D1%82%D0%B5%D1%80-%D1%82%D0%B5%D0%BE%D1%80%D0%B5%D0%BC%D0%B0

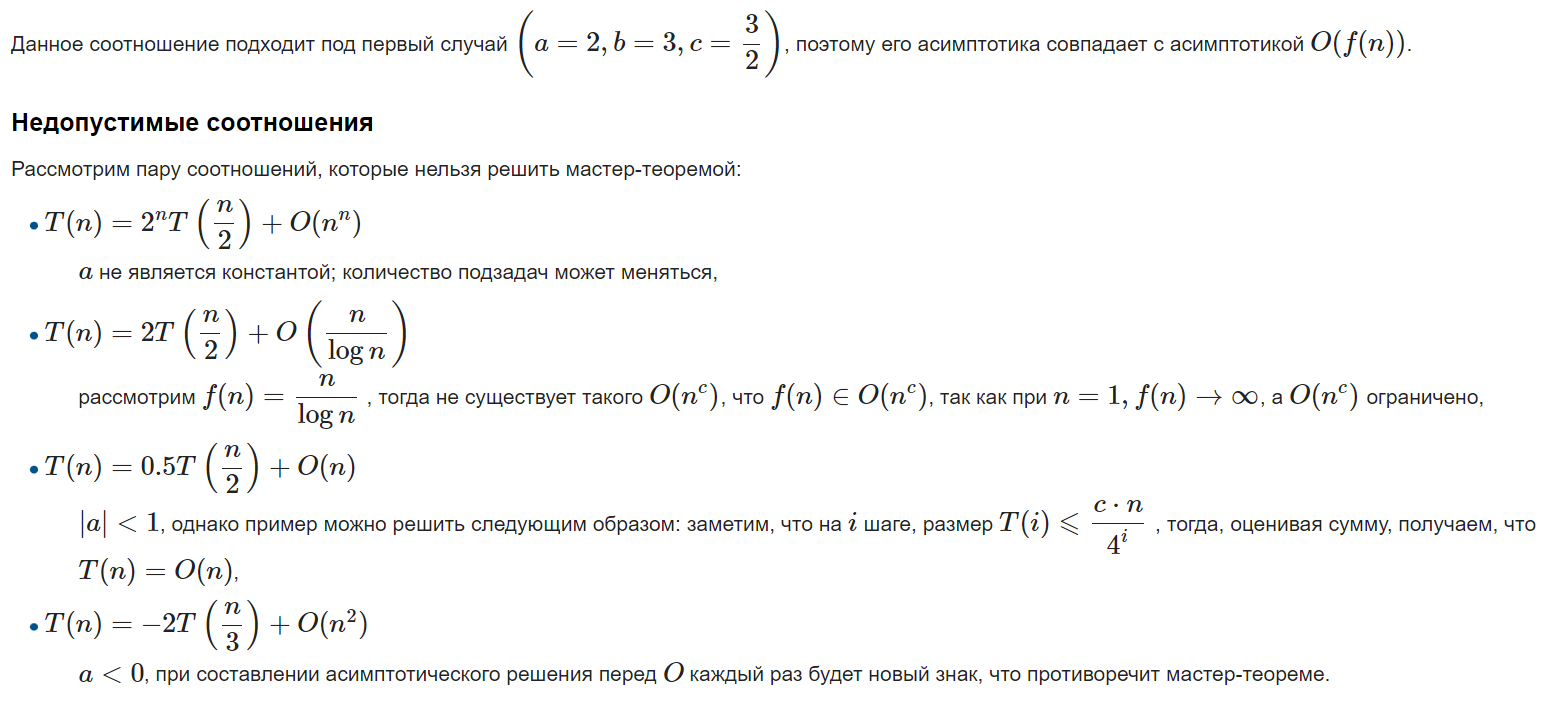
Мастер теорема (англ. Master theorem) позволяет найти асимптотическое решение рекуррентных соотношений, которые могут возникнуть в анализе асимптотики многих алгоритмов. Однако не все рекуррентные соотношения могут быть решены через мастер теорему, ее обобщения включаются в метод Акра-Бацци[[1]](https://neerc.ifmo.ru/wiki/index.php?title=%D0%9C%D0%B0%D1%81%D1%82%D0%B5%D1%80-%D1%82%D0%B5%D0%BE%D1%80%D0%B5%D0%BC%D0%B0#cite_note-1).

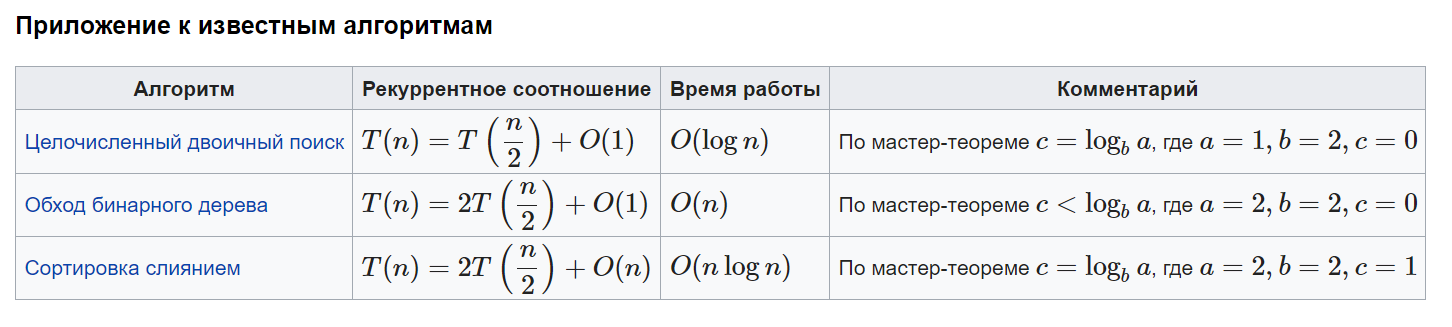


Доказательство









# Вычислительная сложность

# алгоритмов

🞂 Алгоритм ‒ конечное множество четко определенных правил, которые задают последовательность действий для выполнения конкретной задачи (Источник: ГОСТ Р 51904- 2002: Программное обеспечение встроенных систем. Общие требовани я к разработке и документированию).

🞂 Алгоритм ‒ однозначное описание последовательности операций над исходными данными (из некоторой совокупности возможных исходных данных), направленной на получение результата, полностью определяемого этими исходными данными (Источник: МИ 2174- 91: Рекомендация. Государственная система обеспечения единства изм ерений. Аттестация алгоритмов и программ обработки данных при изм ерениях. Основные положения).

Алгоритм (лат. algorithmi — от имени среднеазиатского математика Аль-Хорезми) — совокупность точно заданных правил решения некоторого класса задач или набор инструкций, описывающих порядок действий исполнителя для решения определённой задачи (Источник: википедия).

Характеристики алгоритмов:

Ø Дискретность — алгоритм должен представлять процесс решения задачи как упорядоченное выполнение некоторых простых шагов. При этом для выполнения каждого шага алгоритма требуется конечный отрезок времени, то есть преобразование исходных данных в результат осуществляется во времени дискретно.

Ø Детерминированность (определённость). В каждый момент времени следующий шаг работы однозначно определяется состоянием системы.

Ø Понятность — алгоритм должен включать только те команды,

которые доступны исполнителю и входят в его систему команд.

Ø Завершаемость (конечность) — в более узком понимании алгоритма как математической функции, при правильно заданных начальных данных алгоритм должен завершать работу и выдавать результат за определённое число шагов.

Ø Массовость (универсальность). Алгоритм должен быть применим к

разным наборам начальных данных.

Ø Результативность — завершение алгоритма определёнными результатами.

Характеристики качества алгоритма:

Ø Вычислительная сложность ‒ количество элементарных операций в зависимости от объема входных данных.

Ø Элементарная операция не может быть разбита на более простые операции (например, умножение, сложение, сравнение), т.е., не может быть сокращено время ее выполнения.

Ø Память ‒ объем памяти в зависимости от объема входных данных.

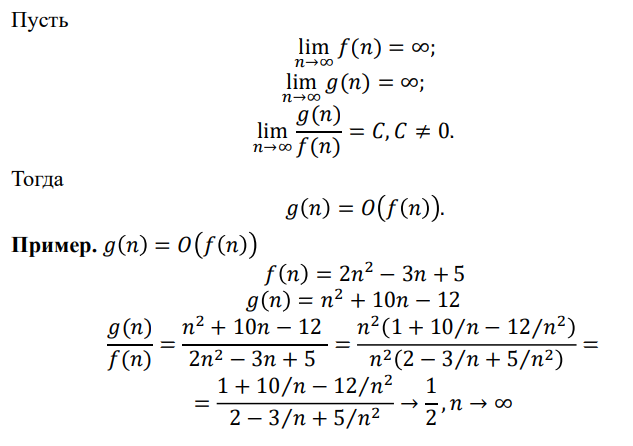
Ø Обычно вычисляется в байтах, килобайтах, и т.д.

Ø Существуют различные понятия сложности: в худшем, среднем или лучшем случае.

Ø Обычно оценивают сложность в худшем случае.

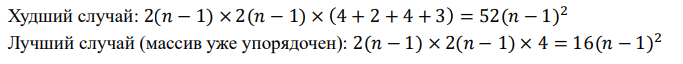
Нотация О, о

«O» большое и «o» малое (O и o) — математические обозначения для сравнения асимптотического поведения (асимптотики) функций. Используются в различных разделах математики, но активнее всего — в математическом анализе, теории чисел и комбинаторике, а также в информатике и теории алгоритмов. Фраза «сложность алгоритма есть 𝑶(𝒇(𝒏)) » означает, что с увеличением параметра 𝑛, характеризующего количество входной информации алгоритма, время работы алгоритма будет возрастать не быстрее, чем 𝑓(𝑛), умноженная на некоторую константу.



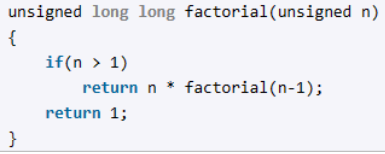


function bubbleSort(arr) { for (let i = 0; i < arr.length-1; i++) { // 2n for (let j = 0; j < arr.length-1; j++) {//2(n-1) // Сравниваем соседние элементы массива if (arr[j] > arr[j + 1]) { // 4 // меняем элементы местами let tmp = arr[j]; // 2 arr[j] = arr[j + 1]; // 4 arr[j + 1] = tmp; // 3 } } } }



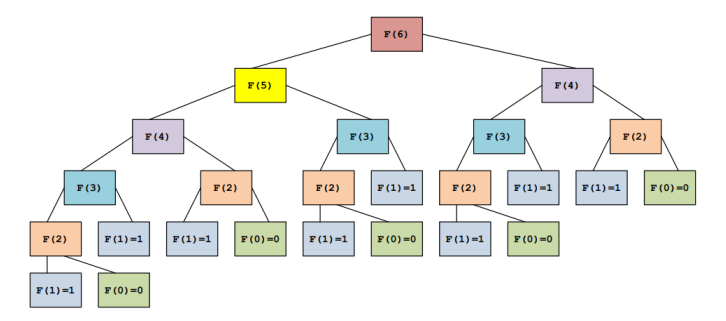
Рекурсивные алгоритмы

Рекурсия – это определение объекта через обращение к самому себе. Рекурсивный алгоритм – это алгоритм, в описании которого прямо или косвенно содержится обращение к самому себе. В технике процедурного программирования данное понятие распространяется на функцию, которая реализует решение отдельного блока задачи посредством вызова из своего тела других функций, в том числе и себя самой. Если при этом на очередном этапе работы функция организует обращение к самой себе, то такая функция является рекурсивной.

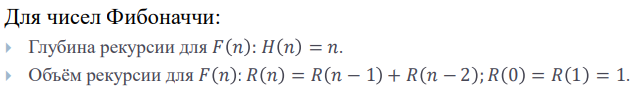


Рекурсивные алгоритмы относятся к классу алгоритмов с высокой ресурсоемкостью, так как при большом количестве самовызовов рекурсивных функций происходит быстрое заполнение стековой области. Кроме того, организация хранения и закрытия очередного слоя рекурсивного стека являются дополнительными операциями, требующими временных затрат. На трудоемкость рекурсивных алгоритмов влияет и количество передаваемых функцией параметров. Полное дерево рекурсии представляет собой граф, вершинами которого являются наборы фактических параметров при всех вызовах функции, начиная с первого обращения к ней, а ребрами – пары таких наборов, соответствующих взаимным вызовам. При этом вершины дерева рекурсии соответствуют фактическим вызовам рекурсивных функций.

Числа Фибоначчи 𝐹(0) = 0; 𝐹(1) = 1; 𝐹(𝑛) = 𝐹(𝑛 − 1) + 𝐹(𝑛 − 2) , 𝑛 ≥ 2.



Глубина рекурсивных вызовов – наибольшее одновременное количество рекурсивных обращений функции, определяющее максимальное количество слоев рекурсивного стека, в котором осуществляется хранение отложенных вычислений. Объем рекурсии ‒ количество вершин полного рекурсивного дерева без единицы



Основная теорема о рекуррентных соотношениях (master theorem) используется в анализе алгоритмов для получения асимптотической оценки рекурсивных соотношений (рекуррентных уравнений), часто возникающих при анализе алгоритмов типа «разделяй и властвуй», например, при оценке времени их выполнения. Теорема была введена и доказана Джоном Бентли, Доротеном Хакеном и Джеймсом Хакеном в 1980 году. Теорема была популяризована в книге ”Алгоритмы: построение и анализ” (Томас Кормен, Чарльз Лейзерстон, Рональд Ривест, Клиффорд Штайн), в которой она была приведена.

Рекурсивный алгоритм в общем виде:

function T(input n: размер задачи):

if n < k:

решить задачу относительно n нерекурсивно

else:

определить множество из a подзадач, каждая

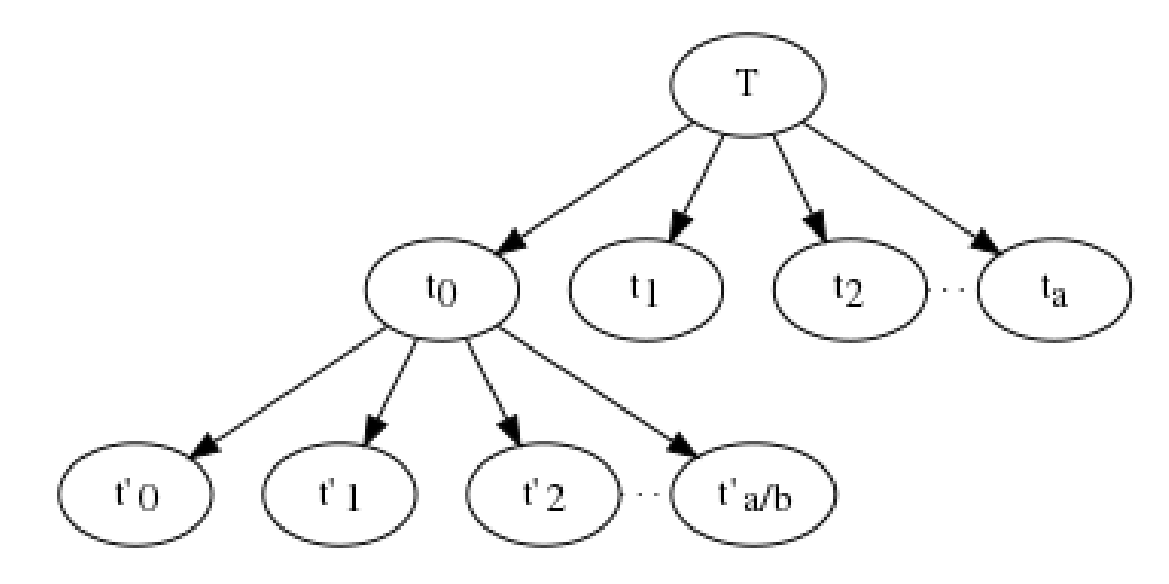
размера n/b

вызвать функцию T рекурсивно для каждой

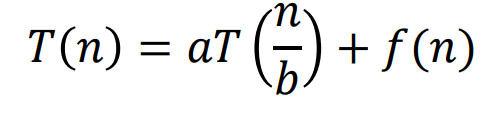
подзадачи

объединить решения

end



Рекурсивный алгоритм в общем виде:



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.

Пример: двоичный поиск в упорядоченном массиве

Вопрос: определите 𝑎, 𝑏, 𝑓(𝑛).

int binSearch(int[] a, int key): // Запускаем

бинарный поиск

int l = -1 // l, r — левая и правая границы

int r = len(a)

while l < r - 1 // Запускаем цикл

m = (l + r) / 2

// m — середина области поиска

if a[m] < key

l = m

else

r = m // Сужение границ

return r



Бинарный поиск

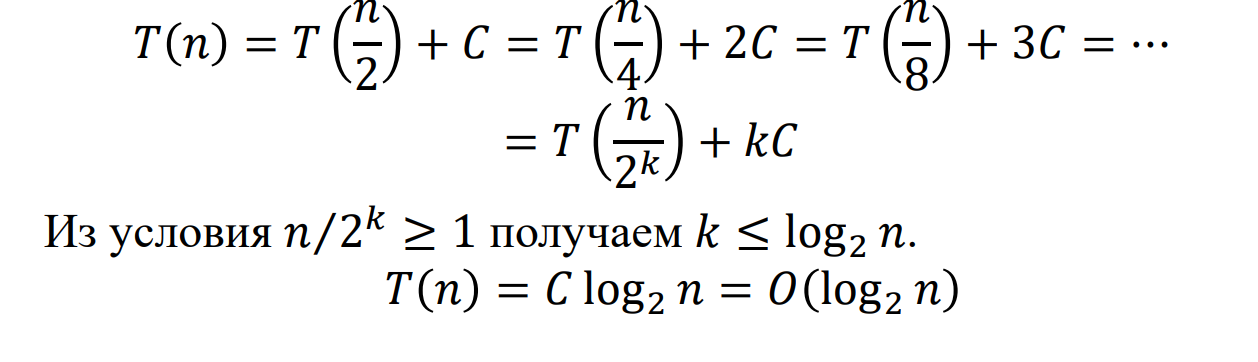
𝑎 = 1 (выполняем бинарный поиск в одной из половин

массива);

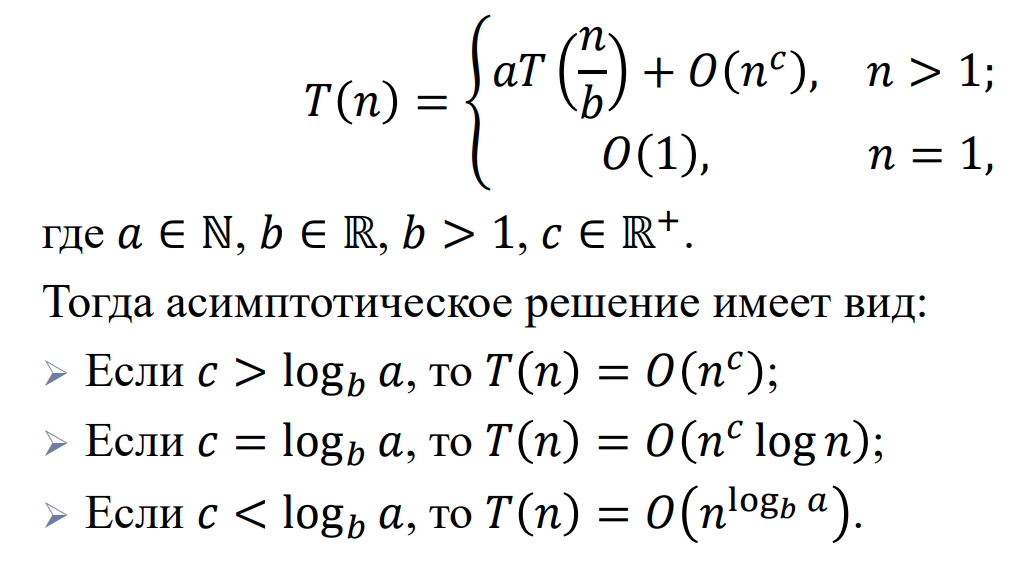
𝑏 = 2 (массив делится на две равные части);

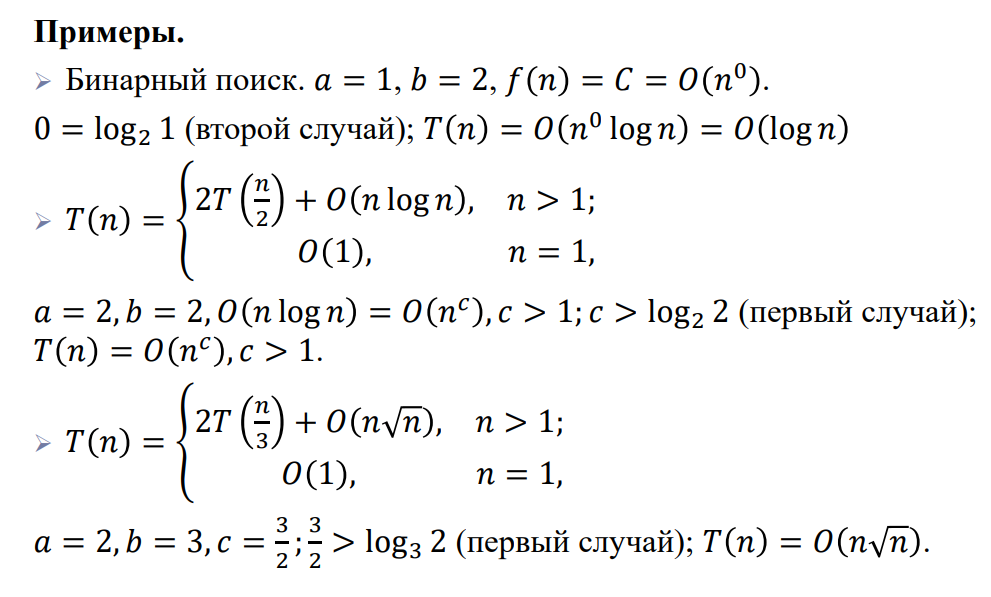
𝑓 (𝑛) = 𝐶 (выполняется одна операция деления и одно

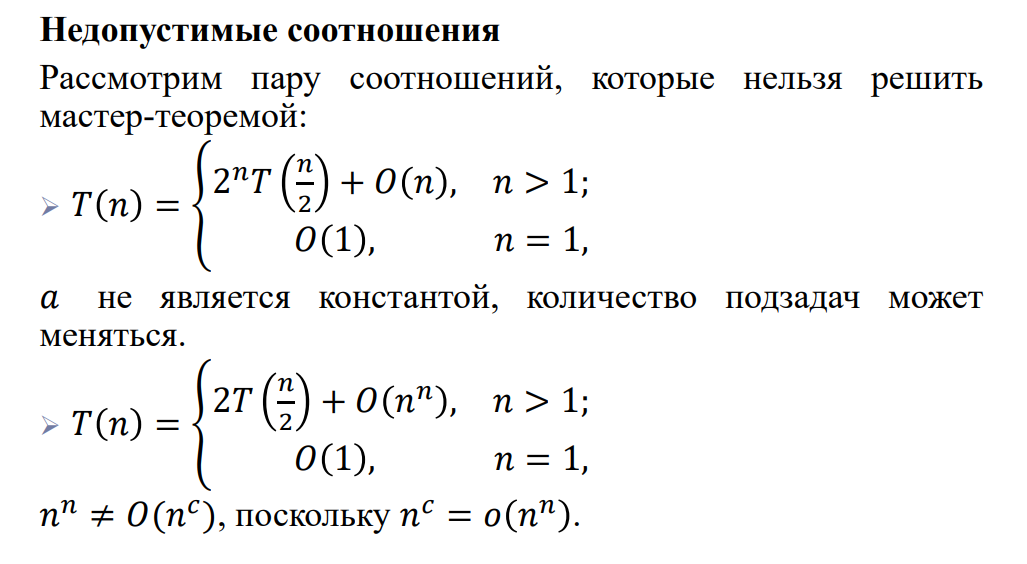
сравнение).

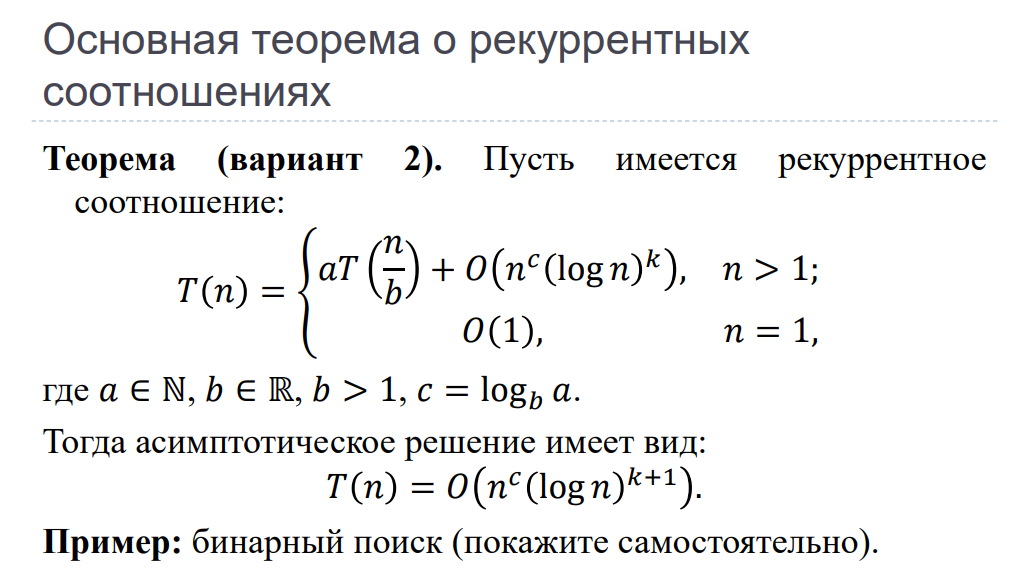


Теорема (вариант 1). Пусть имеется рекуррентное соотношение:









Алгоритм Карацубы.

В 1960-м году Андрей Николаевич Колмогоров и несколько

других советских пионеров информатики собрались на научном

семинаре и выдвинули «гипотезу »: невозможно перемножить

два 𝑛-значных числа, быстрее, чем за 𝑂(). Это подразумевает,

что умножение «в столбик», придуманное шумерами как

минимум четыре тысячи лет назад, является асимптотически

оптимальным алгоритмом умножения двух чисел.

Через неделю 23-летний аспирант Анатолий Карацуба

предложил метод умножения с оценкой времени работы и тем

самым опроверг гипотезу.

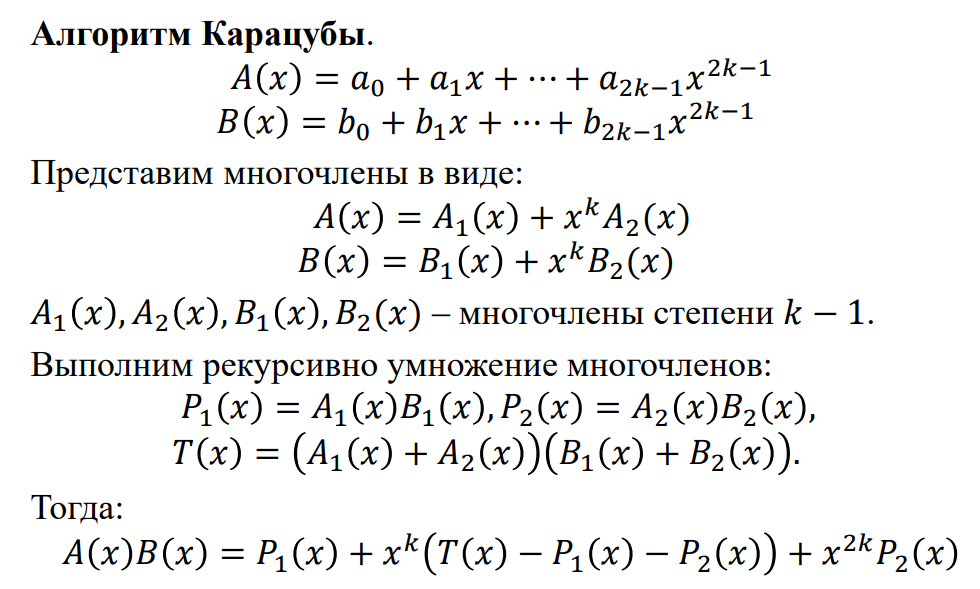
Историческое примечание: эта задача сейчас решается за

𝑂(𝑛 log 𝑛) с помощью алгоритма быстрого преобразования

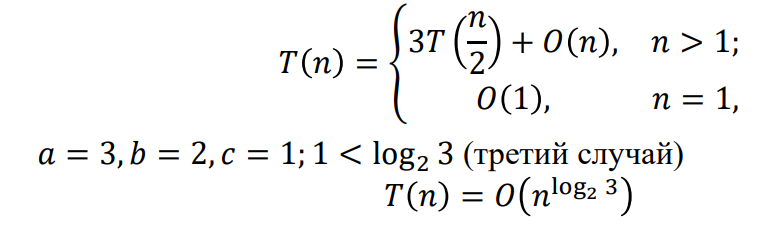
Фурье, который к концу 50-х годов уже изобрели, но

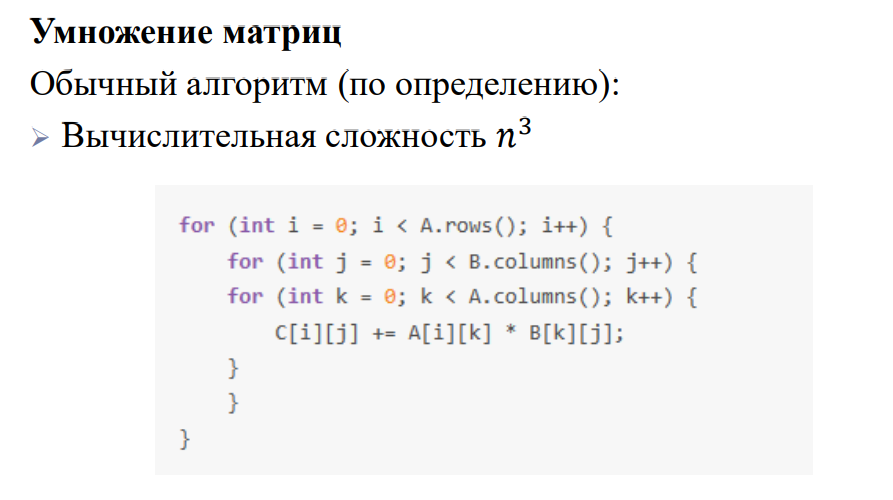
использовали только «по назначению»: для обработки сигналов,

а не для умножения чисел.

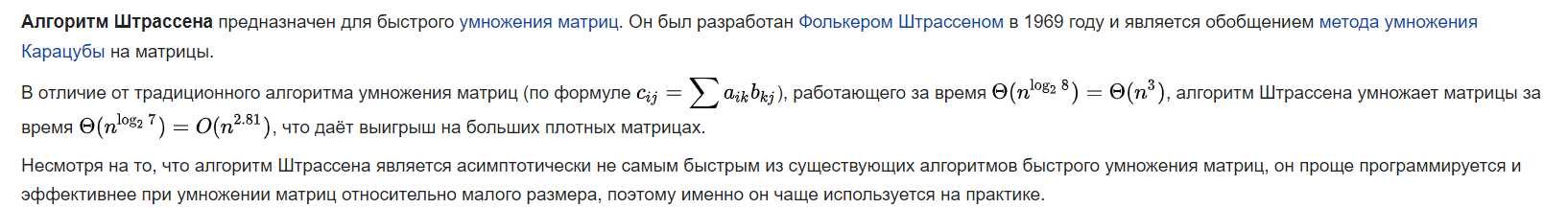


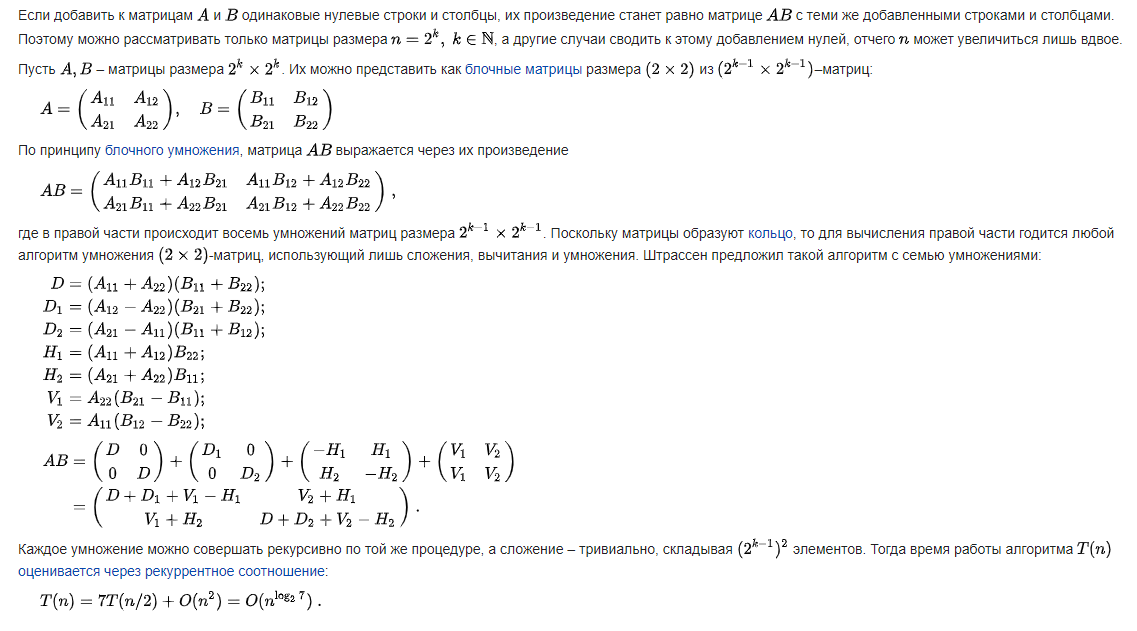
Алгоритм Карацубы. Для умножения двух многочленов с n коэффициентами нужно: Три раза вычислить произведение двух многочленов с 𝑛/2 коэффициентами; Выполнить два вычитания для многочленов с 𝑛/2 коэффициентами, два умножения таких многочленов на число и два сложения многочленов.

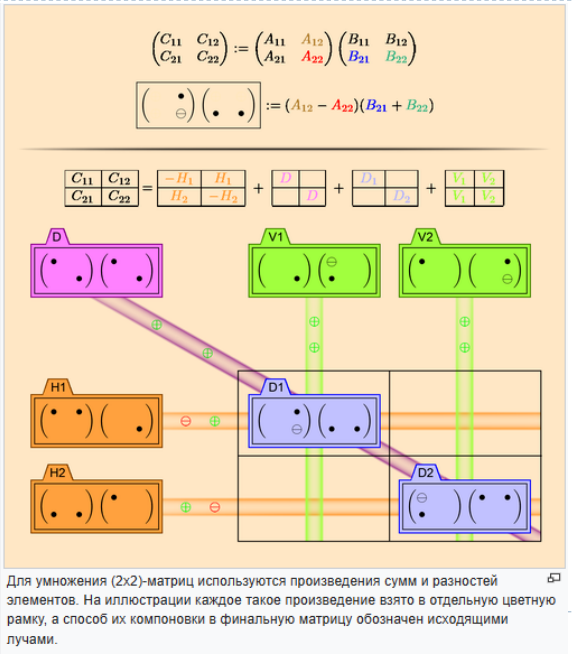




Алгоритм Штрассена







Ниже приведён пример реализации алгоритма на языке [Python](https://ru.wikipedia.org/wiki/Python) с использованием библиотеки [NumPy](https://ru.wikipedia.org/wiki/NumPy) для быстрого взятия подматриц. Основная функция – strassen\_mul. Предполагается, что все матрицы квадратны, представлены типом numpy.array и их размер является степенью 2.

При небольших размерах матрицы прямое умножение оказывается быстрее алгоритма Штрассена из-за большого числа сложений в последнем. Граница таких размеров зависит от соотношения времени сложения и умножения элементов и поэтому может варьироваться в зависимости от аппаратной среды. В коде за её назначение отвечает константа TRIVIAL\_MULTIPLICATION\_BOUND.

**from** **itertools** **import** product

**import** **numpy** **as** **np**

**def** split\_to\_2x2\_blocks(matrix):

**return** list(map(

**lambda** row: np.hsplit(row, 2),

np.vsplit(matrix, 2)

))

**def** strassen\_mul\_2x2(lb, rb):

d = strassen\_mul(lb[0][0] + lb[1][1], rb[0][0] + rb[1][1])

d\_1 = strassen\_mul(lb[0][1] - lb[1][1], rb[1][0] + rb[1][1])

d\_2 = strassen\_mul(lb[1][0] - lb[0][0], rb[0][0] + rb[0][1])

left = strassen\_mul(lb[1][1], rb[1][0] - rb[0][0])

right = strassen\_mul(lb[0][0], rb[0][1] - rb[1][1])

top = strassen\_mul(lb[0][0] + lb[0][1], rb[1][1])

bottom = strassen\_mul(lb[1][0] + lb[1][1], rb[0][0])

**return** [[d + d\_1 + left - top, right + top],

[left + bottom, d + d\_2 + right - bottom]]

**def** trivial\_mul(left, right):

height, mid\_size = left.shape

mid\_size, width = right.shape

result = np.zeros((height, width))

**for** row, col, mid **in** product(\*map(range, [height, width, mid\_size])):

result[row][col] += left[row][mid] \* right[mid][col]

**return** result

TRIVIAL\_MULTIPLICATION\_BOUND = 8

**def** strassen\_mul(left, right):

**assert**(left.shape == right.shape)

**assert**(left.shape[0] == left.shape[1])

**if** left.shape[0] <= TRIVIAL\_MULTIPLICATION\_BOUND:

**return** trivial\_mul(left, right)

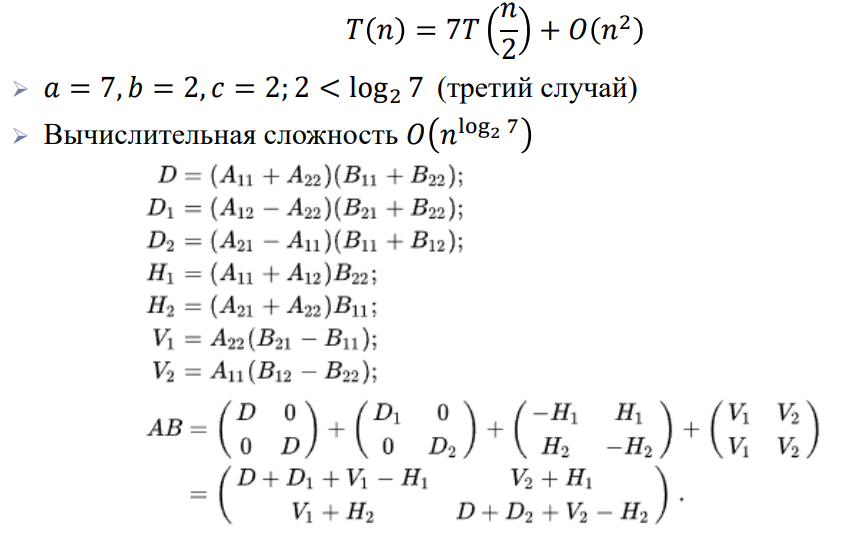
**assert**(left.shape[0] % 2 == 0)

**return** np.block(

strassen\_mul\_2x2(\*map(split\_to\_2x2\_blocks, [left, right]))

)

Алгоритм Штрассена



# Алгоритмы и структуры данных

• Алгоритм – это формально описанная вычислительная процедура, получающая входные данные (input), и выдающая результат на выход (output)

• Алгоритм – точное предписание, которое задает вычислительный процесс, начинающийся с произвольного исходного данного и

направленный на получение полностью определенного этим исходным данным результата.

• Алгоритм – это четкое описание по выполнению некоторого

процесса обработки данных, который через разумное конечное

число шагов приводит к решению задачи данного типа для любых допустимых вариантов исходных данных.

Свойства алгоритмов

• Дискретность (прерывность) - алгоритм должен представлять процесс решения задачи как последовательное выполнение простых шагов.

• Определенность - каждое правило алгоритма должно быть четким, однозначным и не оставлять места для вариаций.

• Результативность (конечность) - алгоритм должен приводить к решению задачи за конечное число шагов.

• Детерминированность. После каждого шага необходимо указывать, какой шаг выполняется следующим, либо давать команду остановки

• Массовость - алгоритм решения задачи разрабатывается в общем виде и должен быть применим для некоторого класса задач, различающихся только входными данными. При этом входные данные могут выбираться из некоторой области, которая называется областью применимости алгоритма

Структура данных

Структура данных – программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных.

Типичные операции:

• добавление данных;

• изменение данных;

• удаление данных;

• поиск данных.

Анализ алгоритмов

• Эффективность алгоритмов определяется различными характеристиками, зависящими от исходных данных (размерность обозначим как n):

• Время работы Т(n);

• Количество выполняемых операций (кол-во арифметических операций,

операций сравнений, операций обращения к диску и т.п.);

• Объемом используемой памяти M(n);

• Адаптируемость алгоритма к различным компьютерам;

Простота, изящество.

Анализ трудоемкости алгоритма

• Целью анализа трудоёмкости алгоритмов является нахождение оптимального алгоритма для решения данной задачи.



Вычислительная сложность алгоритма

• Вычислительная сложность алгоритма — это функция, определяющая зависимость объёма работы, выполняемой

некоторым алгоритмом, от свойств входных данных. Объём

работы обычно измеряется абстрактными понятиями времени и пространства, называемыми вычислительными ресурсами.

• Время определяется количеством элементарных шагов, необходимых для решения проблемы, тогда как пространство определяется объёмом памяти или места на носителе данных.

• Центральный вопрос разработки алгоритмов: «как изменится время

исполнения и объём занятой памяти в зависимости от размера входа и выхода?»

Асимптотический анализ

«По сути, задача их сводилась к анализу кривой

относительного познания в области ее асимптотического приближения к абсолютной истине.» А. и Б. Стругацкие. Понедельник начинается в субботу.

Асимптотический анализ — метод описания предельного поведения функций. Например, в функции f(n)=n2+3n при стремлении n к бесконечности слагаемое 3n

становится пренебрежительно малым, поэтому про функцию f(n) говорят, что она асимптотически эквивалентна n2, при n → ∞ или записывают как f(n) ~ n2

Асимптотическая сложность алгоритмов

• Асимптотическая сложность (асимптотическое описание временной сложности) - оценка скорости роста времени работы алгоритмов,

предназначенных для решения одной и той же задачи, при больших объемах входных данных.

• Количество элементарных операций, затраченных алгоритмом для решения конкретного экземпляра задачи, зависит не только от размера входных данных, но и от самих данных.

• Асимптотическая сложность алгоритмов обычно рассматривается для худшего случая входных данных, т.е. сами данные приводят к наибольшему числу элементарных операций. Например, при сортировке пузырьком, худший случай представляют исходные данные отсортированные в обратном порядке.

• Асимптотическая сложность алгоритмов записывается через нотацию большого О, или Big O Notation

Хороший, плохой, средний

• Худший случай (worst case) — это когда входные данные требуют максимальных затрат времени и памяти.

• Лучший случай (best case) — полная противоположность worst case, самые удачные входные данные. Пример: В случае поиска — когда алгоритм находит нужный элемент с первого раза.

• Средний случай (average case) — это алгоритм, который выполняет среднее количество шагов над входными данными из n элементов.

Пример: В случае поиска элемент находится в середине массива, либо проводится ряд вычислительных экспериментов со случайными данными.

Big O Notation

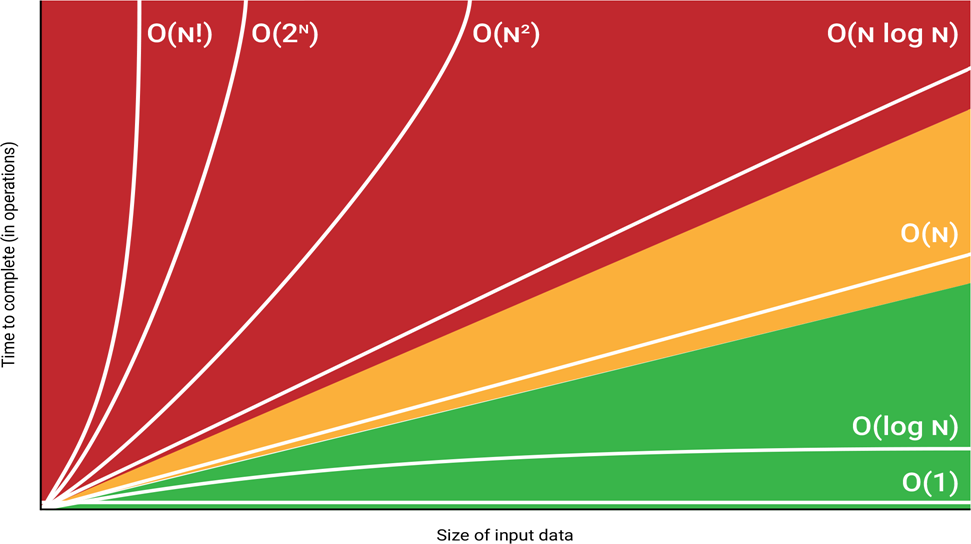
• Big O обозначает верхнюю границу сложности алгоритма. Это идеальный инструмент для поиска worst case.

• Big Omega (Ω) обозначает нижнюю границу сложности, и её правильнее использовать для поиска best case.

• Big Theta (ϴ) располагается между О и омегой и показывает точную функцию сложности алгоритма. С её помощью

правильнее искать average case.

Big O complexity Chart



Градации сложности алгоритмов (1)

• 𝑓 (𝑛) = 𝑂(1) – константная

• Примеры:

• алгоритм, обращения к элементу массива

• оператор присвоения с арифметическими вычислениями

• Алгоритм расчета суммы первых 10 элементов массива

• 𝑓 (𝑛) = 𝑂(𝑙𝑜𝑔 𝑛) – логарифмическая

• Примеры:

• бинарный поиск в отсортированном массиве

• алгоритмы быстрого возведения в степень

Алгоритм быстрого возведения в степень

• Алгоритм предназначен для возведения числа x в натуральную степень n. При этом не обязательно перемножать число x n раз.

Используется свойство x 2n = (x 2 ) n

long mypow(long x, long n)

{

long b = 1;

wh le (n != 0)

{

f (n % 2 == 0) { n /= 2; x \*= x; }

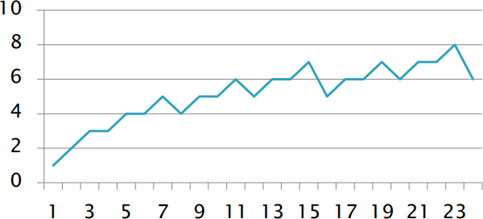
else { n--; b \*= x; }

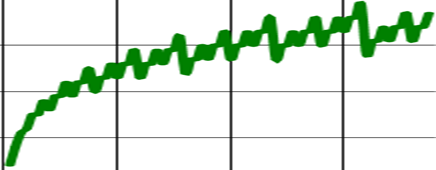
}

return b;

}

• Сложность выполнения алгоритма 𝑂 𝑙𝑜𝑔2(n)





Градации сложности алгоритмов (2)

• 𝑓 (𝑛) ~ 𝑂(n1/2) – сублинейная

• Примеры:

• Поиск Гровера

• 𝑓 (𝑛) ~ 𝑂(n) – линейная

• Примеры:

• перебор всех элементов массива, матрицы, в том числе линейный поиск

• «оптимизированный» алгоритм вычисления числа Фибоначчи (без рекурсии)

• простая рекурсия для вычисления суммы:

int sum( int n)

{

f (n == 1) return 1; return n + sum(n - 1);

}

• 𝑓 (𝑛) = 𝑂(n 𝑙𝑜𝑔 𝑛) – линейно-логарифмическая

• Примеры:

• сортировка с помощью двоичного дерева

• Алгоритм QuickSort

• 𝑓 (𝑛) ~ 𝑂(𝑛k) – полиномиальная (задачи класса P)

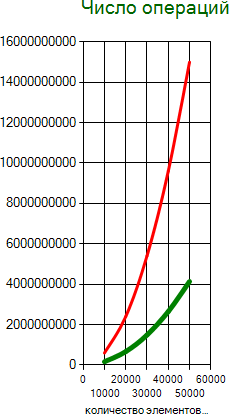
• Примеры:

• Сортировка пузырьком (n2), сортировка вставками (n2) (субквадратичная сложность)

Обычное умножение матриц nxn (n3)

• Buble sort

• Insert sort



Алгоритм вычисления числа Фибоначчи

• 𝑓3 = 𝑓1 + 𝑓2 → 𝑓4 = 𝑓2 + 𝑓3 → ⋯ 𝑓𝑛 = 𝑓𝑛−2 + 𝑓𝑛−1

double Fibonacci ( int IndexNumber)

{

double f1, f2, f3; //Объявляем переменные

f1 = f2 = 1.0; //Задаем известные значения для 1го и 2го числа

f3 = 0.0;

int i = 3; //Объявляем и задаем значение переменной шага

while (i <= IndexNumber)//Пока не достигнем необходимого номера числа Фибоначчи

{

| f3 | = | f2 + f1; | // Определяем I ое число по формуле |
| --- | --- | --- | --- |
|  |  |  | //Меняем для следующего шага переменные |
| f1 | = | f2; //f2 | становится f1 |
| f2 | = | f3; //f3 | становится f2 |

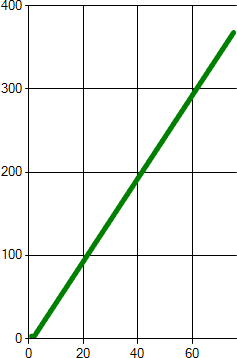
i++; //Увеличиваем шаг (номер числа) на единицу

}

return f3; //Возвращаем последнее посчитанное значение

}

• Сложность выполнения алгоритма 𝑂 (n)



Градации сложности алгоритмов (3)

• 𝑓 (𝑛) = 𝑂(сn) – экспоненциальная

• Примеры:

• Нативный рекурсивный алгоритм вычисления чисел Фибоначчи

• В криптографии атака методом "грубой силы"

• 𝑓 (𝑛) = 𝑂(n!) – факториальная

• Примеры:

• задача коммивояжёра

•

Поиск всех перестановок или размещений

• Вычисление факториала через рекурсию:

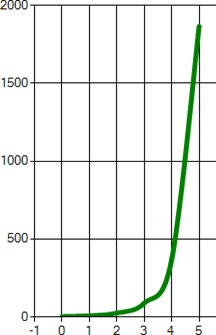
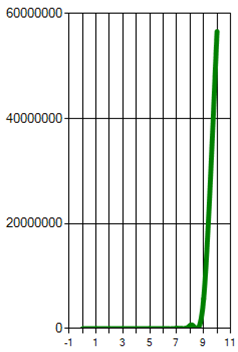
int Factoral ( int n)

{ i int num = n;

f (n == 0) return 1;

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

num = n \* Factorial(n - 1); return num;}

Рекурсивный алгоритм вычисления числа Фибоначчи

int RFibonacci( int n)

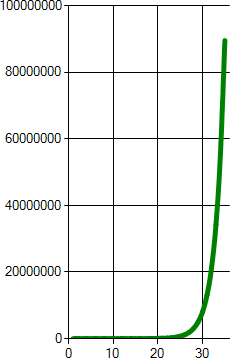
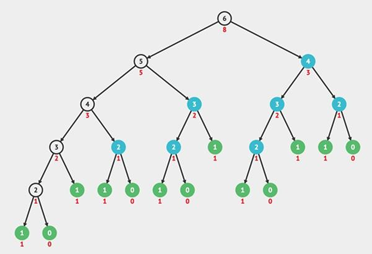
{

f (n <= 1) return n;

else return (RFibonacc(n - 2) + RFibonacc(n - 1);

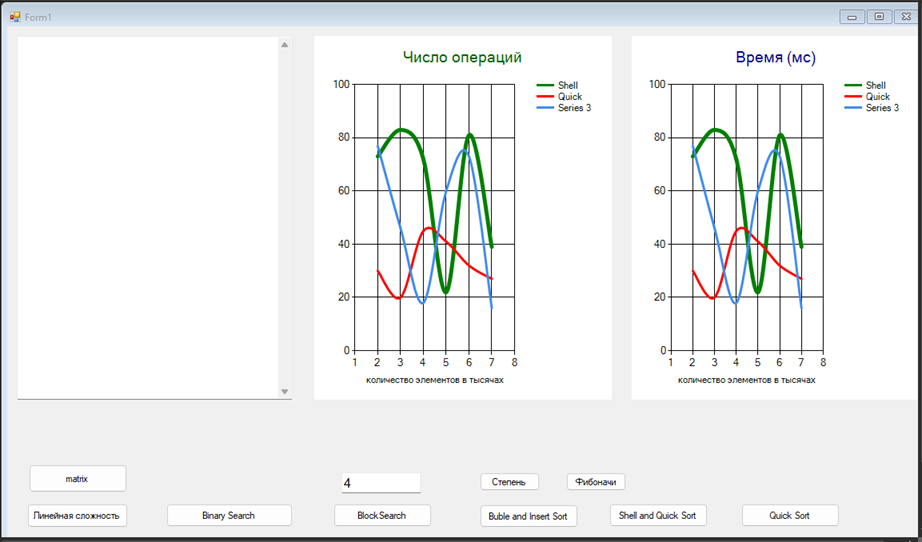
}

• Сложность выполнения алгоритма 𝑂 (2n)

Выполнение лабораторных работ

Создание интерфейса C#



Создание интерфейса на Web странице



Создание интерфейса на html

<table>

<tr>

<td>

<p>

<button onclick=**"AllElements();"**> **Поиск минимума в матрице** </button>

</p>

<p>

<button onclick=**"MainDiagonal();"**> **Поиск минимума на главной диагонали** </button>

</p>

</td>

<td>

<div id = **"container"** style = **"width: 550px; height: 400px; margin: 0 auto"**> </div>

</td>

</tr>

</table>

Организация серии вычислительных экспериментов

***C***#

for ( int n = 10000; n <= 2000000; n += 50000)

{

int[] myarray = new int[n];

//Генерация массива

Random rand = new Random();

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

myarray[ ] = rand.Next();

//Средний случай для бинарного поиска: int item = myarray[0];

//Сортировка массива Array.Sort(myarray); k = 0;

//Вызов метода для n-го эксперимента

}

**Python**

**for i in range(10, 1000, 1): myList = []**

**r = 100000**

**for j in range(i):**

**myList.append(randint(-r, r))**

**myList.sort()**

**#Худший случай для бинарного поиска:**

**n = 1000002 #Элемент для поиска**

**k = 0**

**#**Вызов метода для n-го эксперимента

Организация серии вычислительных экспериментов JavaScript

**function** matrix**(**m**,** n**)**

**{ var** result **= []**

**for(var** i **=** 0**;** i **<** n**;** i**++)**

**{** result**.**push**(new** Array**(**m**).**fill**(**0**)) }**

**return** result**}**

**function** getRandom**(**min**,** max**)**

**{return** Math**.**random**() \* (**max **-** min**) +** min **}**

**function** setRandonMatrix**(**matrix**)**

**{let** row **=** matrix**.**length**;**

**let** column **=** matrix**[**0**].**length**; for (var** i **=** 0**;** i **<** row**;** i**++)**

**for (var** j **=** 0**;** j **<** column**;** j**++)**

matrix**[**i**][**j**] =** getRandom**(-**100**,**100**); return** matrix**;}**

//Цикл для организации численных экспериментов

**for (var** i **=** 1000**;** i **<=** 10000**;** i **+=** 1000**)**

**{**

//Создаем матрицу NxN myarray **=** matrix**(**i**,**i**);**

//Заполняем матрицу случайными числами

myarray **=** setRandonMatrix**(**myarray**);**

//Проводим i вычислительный эксперимент

k **=** 0**;**

min **=** getMinmatrix**(**myarray**);**

**}**

Оценки времени эксперимента

C# внутри цикла вычислительных экспериментов Stopwatch stopwatch = new Stopwatch(); stopwatch.Start();

//Вызов метода для n-го эксперимента

stopwatch.Stop();

Результат в stopwatch.ElapsedTcks stopwatch.ElapsedMll seconds

При описании объекта stopwatch вне цикла: stopwatch.Reset(); stopwatch.Start();

//Вызов метода для n-го эксперимента

stopwatch.Stop();

Python внутри цикла вычислительных экспериментов

import time

start\_time = time.monotonic\_ns()

# вызов n-го эксперимента

Результат в

time.monotonic\_ns() - start\_time

аналогично JavaScript:

**const {**performance**} =** require**(**'perf\_hooks’**);**

**var** startTime **=** performance**.**now**()**

**//Вызов метода для n-го эксперимента**

**var** endTime **=** performance**.**now**()**

Результат в

endTime **-** startTime

Оценка количества операций C#

private int BnarySearch( int[] array,

int searchedValue, int left, int right)

{

while (left <= right)

{ var middle = (left + right) / 2;

if (searchedValue == array[middle]) return middle;

else f (searchedValue < array[middle]) right = middle -- 1;

else

left = middle + 1;

}

return -1;

}

private int BnarySearch( int[] array, int searchedValue, int left, int right)

{

while (left <= right)

{ k++; //Проверка условия в while var middle = (left + r ight) / 2; k++; //вычисление middle

k += 2; //проверка условия и обращение к элементу массива

if (searchedValue == array[middle])

{return middle;}

else if (searchedValue < array[middle])

{ right = middle - 1;

k += 3; //Проверка условия и обращение к элементу массива и вычисление

}

else

{ left = middle + 1;

k++; //вычисление left

}

}

return -1;

}

Оценка количества операций Python

**def BinarySearch(myList, SearchedValue):**

**left = 0**

**right = len(myList)-1 index = -1**

**global k**

**k += 4**

**while (left<=right):**

**middle = (left+right)//2 k += 2**

**if SearchedValue == myList[middle]:**

**index = middle k += 3**

**break else:**

**if SearchedValue < myList[middle]: right = middle -1**

**k += 3**

**else:**

**left = middle + 1 k += 1**

**return index**

Оценка количества операций JavaScript

**function** getMinmatrix**(**matrix**)**

**{ let** row **=** matrix**.**length**;** k**++;**

**let** column **=** matrix**[**0**].**length**;** k**++;** result **=** matrix**[**0**][**0**];** k **=+**2**;**

**for (var** i **=** 0**;** i **<** row**;** i**++)**

**{** k **+=** 2**;**

**for (var** j **=** 0**;** j **<** column**;** j**++)**

**{** k **+=** 2**;**

k **+=** 2**;**

**if (**matrix**[**i**][**j**] <** result**)**

**{** result **=** matrix**[**i**][**j**];** k **+=** 2**; }**

**}**

**}**

**return** result**;**

**}**

Построение графиков С#

• Приложение WinForms (.NetFrameWork)

• компонент Chart

• Свойство Series (2-3 серии для графиков)

• Свойство ChartType – Spline

• Свойство BorderWidth – толщина линии графика

• Свойство Color – цвет графика

• Cвойство LegendText –текст легенды для графика

• Свойство Titles (один заголовок для графика)

• Свойство Text

• Доступ из кода программы к свойству Points

• Метод Clear – очистка всех точек графика

• Метод AddXY – добавление точки на график

this.chart1.Series[0].floints.Clear(); this.chart2.Series[0].floints.Clear(); this.chart1.Series[1].floints.Clear(); this.chart2.Series[1].floints.Clear();   
В цикле вычислительных экспериментов:

this.chart1.Series[0].floints.AddXY(n, k);

this.chart2.Series[0].floints.AddXY(n, stopwatch.ElapsedTicks);

Построение графиков Python

import matplotlib.pyplot as plt

x = []

y = []

| for | i | in | range(10, | 1000, | 1): |
| --- | --- | --- | --- | --- | --- |
|  | k | = 0 | |  |  |

result = BinarySearch(myList, n)

x.append(i) #добавление размерности данных

y.append(k) #добавление числа операций

plt.plot(x,y)

Построение графиков JavaScript.

Подготовка HTML

**Код HTML**

**<head> <title>Google Charts Complexity Graphic</title>**

**<script type="text/javascript"src="https://**[**www.gstatic.com/charts/loader.js"**](http://www.gstatic.com/charts/loader.js)**>**

**</script>**

**</head>**

**<script language='JavaScript'> google.charts.load('current',{packages:['corechart']}); google.charts.setOnLoadCallback(drawChart); </script>**

Построение простого графика JavaScript

**Код Javascript**

**var** data**; var** options**; var** chart**;**

**function** drawChart**()**

**{** // Задаем chart и первоначальный график

data **= new** google**.**visualization**.**DataTable**();** data**.**addColumn**(**'number'**,** 'Размерность данных’**);** data**.**addColumn**(**'number'**,** 'Вычислительная сложность’**);**

data**.**addRows**([ [**100**,** 154**], [**200**,** 987**], [**300**,** 1376**] ]);** // задаем 3 точки

options **= {**'title'**:**'Вычислительная сложность алгоритма'**,** 'width'**:**550**,**

'height'**:**400**};** // Задаем options

chart **= new** google**.**visualization**.**LineChart**(document.**getElementById

**(**'container'**));** chart**.**draw**(**data**,** options**);**

Построение графика на JavaScript по результатам численных экспериментов

**function AllElements()**

**{ //Очищаем data для графика var n = data.getNumberOfRows(); data.removeRows(0, n)**

**//Цикл для организации численных экспериментов**

**for (var i = 1000; i <= 10000; i += 1000)**

**{ //пропущена подготовка матриц**

**//Проводим i вычислительный эксперимент**

**k = 0;**

**min = getMinmatrix(myarray); let x = i\*i;**

**let y = k;**

**//Добавляем в график точку**

**data.addRows ([ [x, y], ]); }**

**//Вызываем прорисовку графика**

**chart.draw(data, options); }**

Коллизия при определении временной и вычислительной сложности

//подсчет отрицательных элементов в матрице

private int doMatrix( int[,] array, int n)

{ int m = 0;

int count = 0; ;

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

{ k += 2;

for ( int j = 0; j < n; j++)

{ k += 2;

if (array[ , j] < 0)

{ count++;

k += 3;

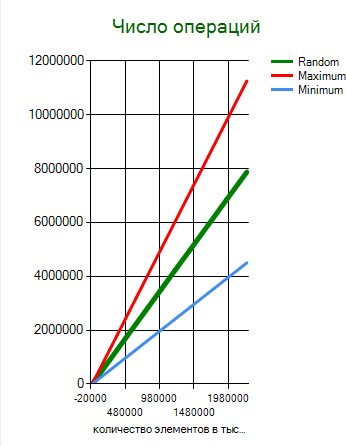
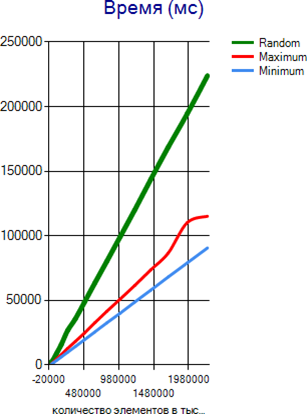
}

}

}

return count;

}

Случайные числа

Все элементы положительные

Все элементы отрицательные

# Лабораторная работа № 1. Алгоритмы и их сложность

Цель работы: Научиться исследовать вычислительную и временную сложность [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов на различных наборах входных данных.

Задание:  
1. Разработать интерфейс программы содержащий элементы для вывода и проверки результата вычислений, для вывода графики временной сложности [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а и вычислительной сложности [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а.  
2. Разработать и проверить программу реализующую [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ы приведенные в индивидуальном задании. Для каждого [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а внедрить в код программы элементы для подсчета числа вычислений.  
3. Разработать и реализовать стратегии формирования исходных данных а) случайным образом, б) минимизирующим число вычислений в [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)е, в) максимизирующее число вычислений в [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)е.  
4. Определить временную и вычислительную сложность [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а. Построить графики зависимости времени выполнения от размерности исходных данных. При построении графиков для заданий с матрицами, размерность задачи считать как количество элементов в матрице. Максимальный размер данных для массивов необходимо брать не менее 1 млн. элементов, для матриц не менее 5 тысяч на 5 тысяч элементов.  
5. Провести анализ полученных результатов.

Инструкция по выполнению:   
1. Номер варианта определяется в элементе курса [Выбор варианта лабораторной работы № 1](https://stud.lms.tpu.ru/mod/choice/view.php?id=603280).  
2. Матрицы и вектора заполняются случайными целыми числами в соответствии со стратегией формирования исходных данных.  
4. В отчет необходимо включить: скриншоты, иллюстрирующие выполнение.  
5. Приложение должно быть реализовано на С++ или Java или C#.

Критерии оценки:

1. Полнота и правильность описания [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)а (20%).
2. Полнота тестирования приложения (20%).
3. Эффективность организации структуры приложения (20%)
4. Защита лабораторной работы (40%).

Содержание отчета по заданию:

1. Титульный лист.
2. Задание
3. Описание стратегии формирования данных
4. Текст программы
5. Результаты проверки работоспособности программы
6. Результаты исследования [алгоритм](https://stud.lms.tpu.ru/mod/glossary/showentry.php?eid=375558&displayformat=dictionary)ов на сложность
7. Аналитические развернутые выводы.

Оформление отчета в соответствии со [стандартом ТПУ](http://standard.tpu.ru/docs/standorg/stp42i.mht)

# Рекурсия и рекурсивные алгоритмы (Intuit)

<https://intuit.ru/studies/courses/648/504/lecture/11462?ysclid=ls02vc2uq4946572523>

Цель лекции: изучить понятие, виды рекурсии и рекурсивную триаду, научиться разрабатывать рекурсивную триаду при решении задач на языке C++.

Одной из идей процедурного программирования, которая оформилась в начале шестидесятых годов ХХ века, стало активное применение в практике программирования некоторого метода, основанного на организации серий взаимных обращений программ (функций) друг к другу. Вопросы об эффективности использования данного метода при разработке алгоритмических моделей актуальны и в настоящее время, несмотря на существование различных парадигм программирования, создание новых и совершенствование существующих языков программирования. Речь идет о рекурсивном методе в программировании, который рассматривается альтернативным по отношению к итерационному.

Рекурсия – это определение объекта через обращение к самому себе.

Рекурсивный алгоритм – это алгоритм, в описании которого прямо или косвенно содержится обращение к самому себе. В технике процедурного программирования данное понятие распространяется на функцию, которая реализует решение отдельного блока задачи посредством вызова из своего тела других функций, в том числе и себя самой. Если при этом на очередном этапе работы функция организует обращение к самой себе, то такая функция является рекурсивной.

Прямое обращение функции к самой себе предполагает, что в теле функции содержится вызов этой же функции, но с другим набором фактических параметров. Такой способ организации работы называется прямой рекурсией. Например, чтобы найти сумму первых n натуральных чисел, надо сумму первых (n-1) чисел сложить с числом n, то есть имеет место зависимость: Sn=Sn-1+n. Вычисление происходит с помощью аналогичных рассуждений. Такая цепочка взаимных обращений в конечном итоге сведется к вычислению суммы одного первого элемента, которая равна самому элементу.

При косвенном обращении функция содержит вызовы других функций из своего тела. При этом одна или несколько из вызываемых функций на определенном этапе обращаются к исходной функции с измененным набором входных параметров. Такая организация обращений называется косвенной рекурсией. Например, поиск максимального элемента в массиве размера n можно осуществлять как поиск максимума из двух чисел: одно их них – это последний элемент массива, а другое является максимальным элементом в массиве размера (n-1). Для нахождения максимального элемента массива размера (n-1) применяются аналогичные рассуждения. В итоге решение сводится к поиску максимального из первых двух элементов массива.

Рекурсивный метод в программировании предполагает разработку решения задачи, основываясь на свойствах рекурсивности отдельных объектов или закономерностей. При этом исходная задача сводится к решению аналогичных подзадач, которые являются более простыми и отличаются другим набором параметров.

Разработке рекурсивных алгоритмов предшествует рекурсивная триада – этапы моделирования задачи, на которых определяется набор параметров и соотношений между ними. Рекурсивную триаду составляют параметризация, выделение базы и декомпозиция.

На этапе параметризации из постановки задачи выделяются параметры, которые описывают исходные данные. При этом некоторые дальнейшие разработки решения могут требовать введения дополнительных параметров, которые не оговорены в условии, но используются при составлении зависимостей. Необходимость в дополнительных параметрах часто возникает также при решении задач оптимизации рекурсивных алгоритмов, в ходе которых сокращается их временная сложность.

Выделение базы рекурсии предполагает нахождение в решаемой задаче тривиальных случаев, результат для которых очевиден и не требует проведения расчетов. Верно найденная база рекурсии обеспечивает завершенность рекурсивных обращений, которые в конечном итоге сводятся к базовому случаю. Переопределение базы или ее динамическое расширение в ходе решения задачи часто позволяют оптимизировать рекурсивный алгоритм за счет достижения базового случая за более короткий путь обращений.

Декомпозиция представляет собой сведение общего случая к более простым подзадачам, которые отличаются от исходной задачи набором входных данных. Декомпозиционные зависимости описывают не только связь между задачей и подзадачами, но и характер изменения значений параметров на очередном шаге. От выбранных отношений зависит трудоемкость алгоритма, так как для одной и той же задачи могут быть составлены различные зависимости. Пересмотр отношений декомпозиции целесообразно проводить комплексно, то есть параллельно с корректировкой параметров и анализом базовых случаев.

Анализ трудоемкости рекурсивных алгоритмов методом подсчета вершин дерева рекурсии

Рекурсивные алгоритмы относятся к классу алгоритмов с высокой ресурсоемкостью, так как при большом количестве самовызовов рекурсивных функций происходит быстрое заполнение стековой области. Кроме того, организация хранения и закрытия очередного слоя рекурсивного стека являются дополнительными операциями, требующими временных затрат. На трудоемкость рекурсивных алгоритмов влияет и количество передаваемых функцией параметров.

Рассмотрим один из методов анализа трудоемкости рекурсивного алгоритма, который строится на основе подсчета вершин рекурсивного дерева. Для оценки трудоемкости рекурсивных алгоритмов строится полное дерево рекурсии. Оно представляет собой граф, вершинами которого являются наборы фактических параметров при всех вызовах функции, начиная с первого обращения к ней, а ребрами – пары таких наборов, соответствующих взаимным вызовам. При этом вершины дерева рекурсии соответствуют фактическим вызовам рекурсивных функций. Следует заметить, что одни и те же наборы параметров могут соответствовать разным вершинам дерева. Корень полного дерева рекурсивных вызовов – это вершина полного дерева рекурсии, соответствующая начальному обращению к функции.

Важной характеристикой рекурсивного алгоритма является глубина рекурсивных вызовов – наибольшее одновременное количество рекурсивных обращений функции, определяющее максимальное количество слоев рекурсивного стека, в котором осуществляется хранение отложенных вычислений. Количество элементов полных рекурсивных обращений всегда не меньше глубины рекурсивных вызовов. При разработке рекурсивных программ необходимо учитывать, что глубина рекурсивных вызовов не должна превосходить максимального размера стека используемой вычислительной среды.

При этом объем рекурсии - это одна из характеристик сложности рекурсивных вычислений для конкретного набора параметров, представляющая собой количество вершин полного рекурсивного дерева без единицы.

Будем использовать следующие обозначения для конкретного входного параметра D:

R(D) – общее число вершин дерева рекурсии,

RV(D) – объем рекурсии без листьев (внутренние вершины),

RL(D) – количество листьев дерева рекурсии,

HR(D) – глубина рекурсии.

Например, для вычисления n -го члена последовательности Фибоначчи разработана следующая рекурсивная функция:

int Fib(int n){ //n – номер члена последовательности

if(n<3) return 1; //база рекурсии

return Fib(n-1)+Fib(n-2); //декомпозиция

}

Тогда полное дерево рекурсии для вычисления пятого члена последовательности Фибоначчи будет иметь вид ( [рис. 34.1](https://intuit.ru/studies/courses/648/504/lecture/11462?page=1#image.34.1)):

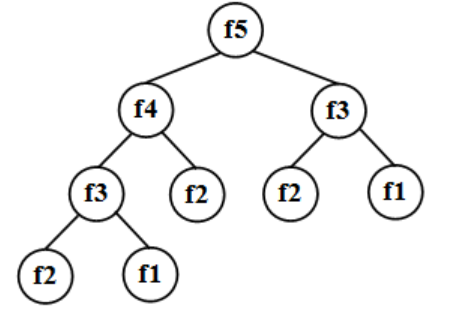
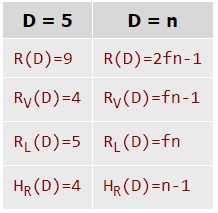


Рис. 34.1. Полное дерево рекурсии для пятого члена последовательности Фибоначчи

Характеристиками рассматриваемого метода оценки алгоритма будут следующие величины.



Пример 1. Задача о разрезании прямоугольника на квадраты.

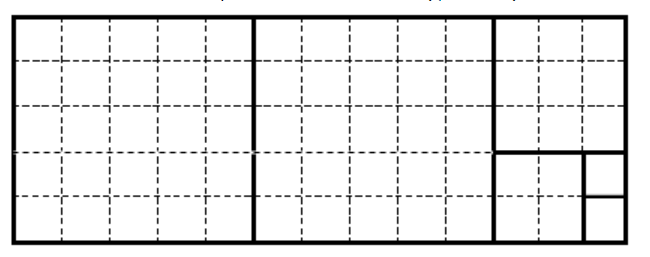
Дан прямоугольник, стороны которого выражены натуральными числами. Разрежьте его на минимальное число квадратов с натуральными сторонами. Найдите число получившихся квадратов.

Разработаем рекурсивную триаду.

Параметризация: m, n – натуральные числа, соответствующие размерам прямоугольника.

База рекурсии: для m=n число получившихся квадратов равно 1, так как данный прямоугольник уже является квадратом.

Декомпозиция: если m не равно n, то возможны два случая m < n или m > n. Отрежем от прямоугольника наибольший по площади квадрат с натуральными сторонами. Длина стороны такого квадрата равна наименьшей из сторон прямоугольника. После того, как квадрат будет отрезан, размеры прямоугольника станут следующие: большая сторона уменьшится на длину стороны квадрата, а меньшая не изменится. Число искомых квадратов будет вычисляться как число квадратов, на которые будет разрезан полученный прямоугольник, плюс один (отрезанный квадрат). К получившемуся прямоугольнику применим аналогичные рассуждения: проверим на соответствие базе или перейдем к декомпозиции ( [рис. 34.2](https://intuit.ru/studies/courses/648/504/lecture/11462?page=1#image.34.2)).



**Рис. 34.2.** Пример разрезания прямоугольника 13x5 на квадраты

#include "stdafx.h"

#include <iostream>

using namespace std;

int kv(int m,int n);

int \_tmain(int argc, \_TCHAR\* argv[]) {

int a,b,k;

printf("Введите стороны прямоугольника->");

scanf("%d%d",&a,&b);

k = kv(a,b);

printf("Прямоугольник со сторонами %d и %d можно разрезать

на %d квадратов",a,b,k);

system("pause");

return 0;

}

int kv(int m,int n){ //m,n – стороны прямоугольника

if(m==n) return 1; //база рекурсии

if(m>n) return 1+kv(m-n,n); //декомпозиция для m>n

return 1+kv(m,n-m); //декомпозиция для m<n

}

Характеристиками рассматриваемого метода оценки алгоритма будут следующие величины ( [рис. 34.3](https://intuit.ru/studies/courses/648/504/lecture/11462?page=1#image.34.3)).



Рис. 34.3. Пример полного дерева рекурсии для разрезания прямоугольника 13x5 на квадраты

[**Академия Microsoft**](https://intuit.ru/academies/companiesn/42/info)**:** [**Структуры и алгоритмы компьютерной обработки данных**](https://intuit.ru/studies/courses/648/504/info)

**[+]**

Пример 2. Задача о нахождении центра тяжести выпуклого многоугольника.

Выпуклый многоугольник задан на плоскости координатами своих вершин. Найдите его центр тяжести.

Разработаем рекурсивную триаду.

Параметризация: x, y – вещественные массивы, в которых хранятся координаты вершин многоугольника; n – это число вершин многоугольника, по условию задачи, n>1 так как минимальное число вершин имеет двуугольник (отрезок).

База рекурсии: для n=2 в качестве многоугольника рассматривается отрезок, центром тяжести которого является его середина ([рис. 34А](https://intuit.ru/studies/courses/648/504/lecture/11462?page=2#image.34.4)). При этом середина делит отрезок в отношении 1 : 1. Если координаты концов отрезка заданы как (x0,y0) и (x1,y1), то координаты середины вычисляются по формуле:

cx=\frac{x_0+x_1}{2},\quad cy=\frac{y_0+y_1}{2}.

Декомпозиция: если n>2, то рассмотрим последовательное нахождение центров тяжести треугольника, четырехугольника и т.д.

Для n=3 центром тяжести треугольника является точка пересечения его медиан, которая делит каждую медиану в отношении 2 : 1, считая от вершины. Но основание медианы – это середина отрезка, являющегося стороной треугольника. Таким образом, для нахождения центра тяжести треугольника необходимо: найти центр тяжести стороны треугольника (отрезка), затем разделить в отношении 2 : 1, считая от вершины, отрезок, образованный основанием медианы и третьей вершиной ([рис. 34B](https://intuit.ru/studies/courses/648/504/lecture/11462?page=2#image.34.4)).

Для n=4 центром тяжести четырехугольника является точка, делящая в отношении 3 : 1, считая от вершины, отрезок: он образован центром тяжести треугольника, построенного на трех вершинах, и четвертой вершиной ([рис. 34C](https://intuit.ru/studies/courses/648/504/lecture/11462?page=2#image.34.4)).

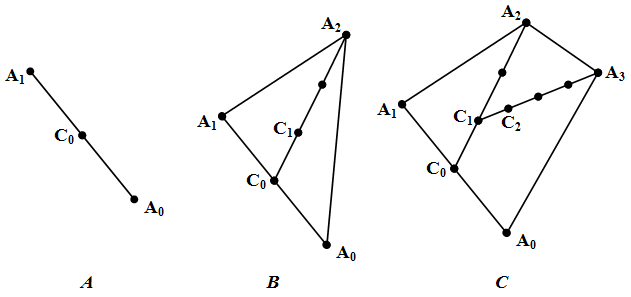


Рис. 34.4. Примеры построения центров тяжести многоугольников

Таким образом, для нахождения центра тяжести n -угольника необходимо разделить в отношении (n-1): 1, считая от вершины, отрезок: он образован центром тяжести (n-1) -угольника и n -ой вершиной рассматриваемого многоугольника. Если концы отрезка заданы координатами вершины (xn,yn) и центра тяжести (n-1) -угольника (cxn-1,cyn-1), то при делении отрезка в данном отношении получаем координаты:

cx_n=\frac{x_n+(n-1)cx_{n-1}}{n},\quad cy_n=\frac{y_n+(n-1)cy_{n-1}}{n}

#include "stdafx.h"

#include <iostream>

using namespace std;

#define max 20

void centr(int n,float \*x, float \*y, float \*c);

int \_tmain(int argc, \_TCHAR\* argv[]){

int m, i=0;

FILE \*f;

if ( ( f = fopen("in.txt", "r") ) == NULL )

perror("in.txt");

else {

fscanf(f, "%d",&m);

printf("\n%d",m);

if ( m < 2 || m > max ) //вырожденный многоугольник

printf ("Вырожденный многоугольник");

else {

float \*px,\*py,\*pc;

px = new float[m];

py = new float[m];

pc = new float[2];

pc[0] = pc[1] = 0;

while(i<m) {

fscanf(f, "%f %f",&px[i], &py[i]);

printf("\n%f %f",px[i], py[i]);

i++;

}

centr(m,px,py,pc);

printf ("\nЦентр тяжести имеет координаты:

(%.4f, %.4f)",pc[0],pc[1]);

delete [] pc;

delete [] py;

delete [] px;

}

fclose(f);

}

system("pause");

return 0;

}

void centr(int n,float \*x, float \*y, float \*c){

//n - количество вершин,

//x,y - координаты вершин,

//c - координаты центра тяжести

if(n==2){ //база рекурсии

c[0]=(x[0]+x[1])/2;

c[1]=(y[0]+y[1])/2;

}

if(n>2) { //декомпозиция

centr(n-1,x,y,c);

c[0]= (x[n-1] + (n-1)\*c[0])/n;

c[1]= (y[n-1] + (n-1)\*c[1])/n;

}

}

Характеристиками рассматриваемого метода оценки алгоритма будут следующие величины.

| **D = 4** | **D = n** |
| --- | --- |
| R(D)=3 | R(D)=n-1 |
| RV(D)=1 | RV(D)=n-3 |
| RL(D)=1 | RL(D)=1 |
| HR(D)=3 | HR(D)=n-1 |

Однако в данном случае для более достоверной оценки необходимо учитывать емкостные характеристики алгоритма.

Пример 3. Задача о разбиении целого на части.

Найдите количество разбиений натурального числа на сумму натуральных слагаемых.

Разбиение подразумевает представление натурального числа в виде суммы натуральных слагаемых, при этом суммы должны отличаться набором чисел, а не их последовательностью. В разбиение также может входить одно число.

Например, разбиение числа 6 будет представлено 11 комбинациями:

6

5+1

4+2, 4+1+1

3+3, 3+2+1, 3+1+1+1

2+2+2, 2+2+1+1, 2+1+1+1+1

1+1+1+1+1+1

Рассмотрим решение в общем виде. Пусть зависимость R(n,k) вычисляет количество разбиений числа n на сумму слагаемых, не превосходящих k. Опишем свойства R(n,k).

Если в сумме все слагаемые не превосходят 1, то такое представление единственно, то есть R(n,k)=1.

Если рассматриваемое число равно 1, то при любом натуральном значении второго параметра разбиение также единственно: R(n,k)=1.

Если второй параметр превосходит значение первого , то имеет место равенство R(n,k)=R(n,n), так как для представления натурального числа в сумму не могут входить числа, превосходящие его.

Если в сумму входит слагаемое, равное первому параметру, то такое представление также единственно (содержит только это слагаемое), поэтому имеет место равенство: R(n,n)=R(n,n-1)+1.

Осталось рассмотреть случай (n>k). Разобьем все представления числа n на непересекающиеся разложения: в одни обязательно будет входить слагаемое k, а другие суммы не содержат k. Первая группа сумм, содержащая k, эквивалентна зависимости R(n-k,k), что следует после вычитания числа k из каждой суммы. Вторая группа сумм содержит разбиение числа n на слагаемые, каждое из которых не превосходит k-1, то есть число таких представлений равно R(n,k-1). Так как обе группы сумм не пересекаются, то R(n,k)=R(n-k,k)+R(n,k-1).

Разработаем рекурсивную триаду.

*Параметризация*: Рассмотрим разбиение натурального числа n на сумму таких слагаемых, которые не превосходят натурального числа k.

*База рекурсии*: исходя из свойств рассмотренной зависимости, выделяются два базовых случая:

при n=1 R(n,k)=1,

при k=1 R(n,k)=1.

*Декомпозиция*: общий случай задачи сводится к трем случаям, которые и составляют декомпозиционные отношения.

при n=k R(n,k)=R(n,n-1)+1,

при n<k R(n,k)=R(n,n),

при n>k R(n,k)=R(n-k,k)+R(n,k-1).

#include "stdafx.h"

#include <iostream>

using namespace std;

unsigned long int Razbienie(unsigned long int n,

unsigned long int k);

int \_tmain(int argc, \_TCHAR\* argv[]){

unsigned long int number, max,num;

printf ("\nВведите натуральное число: ");

scanf ("%d", &number);

printf ("Введите максимальное натуральное слагаемое в

сумме: ");

scanf ("%d", &max);

num=Razbienie(number,max);

printf ("Число %d можно представить в виде суммы с

максимальным слагаемым %d.", number, max);

printf ("\nКоличество разбиений равно %d",num);

system("pause");

return 0;

}

unsigned long int Razbienie(unsigned long int n,

unsigned long int k){

if(n==1 || k==1) return 1;

if(n<=k) return Razbienie(n,n-1)+1;

return Razbienie(n,k-1)+Razbienie(n-k,k);

}

Пример 4. Задача о переводе натурального числа в шестнадцатеричную систему счисления.

Дано натуральное число, не выходящее за пределы типа unsigned long. Число представлено в десятичной системе счисления. Переведите его в систему счисления с основанием 16.

Пусть требуется перевести целое число n из десятичной в р -ичную систему счисления (по условию задачи, р = 16), то есть найти такое k, чтобы выполнялось равенство n10=kp.

Параметризация: n – данное натуральное число, р – основание системы счисления.

База рекурсии: на основании правил перевода чисел из десятичной системы в систему счисления с основанием р, деление нацело на основание системы выполняется до тех пор, пока неполное частное не станет равным нулю, то есть: если целая часть частного n и р равна нулю, то k = n. Данное условие можно реализовать иначе, сравнив n и р: целая часть частного равна нулю, если n < р.

Декомпозиция: в общем случае k формируется из цифр целой части частного n и р, представленной в системе счисления с основанием р, и остатка от деления n на p.

#include "stdafx.h"

#include <iostream>

using namespace std;

#define maxline 50

void perevod( unsigned long n, unsigned int p,FILE \*pf);

int \_tmain(int argc, \_TCHAR\* argv[]){

unsigned long number10;

unsigned int osn=16;

char number16[maxline];

FILE \*f;

if ((f=fopen("out.txt", "w"))==NULL)

perror("out.txt");

else {

printf ("\nВведите число в десятичной системе: ");

scanf("%ld", &number10);

perevod(number10, osn, f);

fclose(f);

}

if ((f=fopen("out.txt", "r"))==NULL)

perror("out.txt");

else {

fscanf(f,"%s",number16);

printf("\n %ld(10)=%s(16)", number10, number16);

fclose(f);

}

system("pause");

return 0;

}

void perevod(unsigned long n, unsigned int p, FILE \*pf){

char c;

unsigned int r;

if(n >= p) perevod (n/p, p, pf);//декомпозиция

r=n%p;

c=r < 10 ? char (r+48) : char (r+55);

putc(c, pf);

}

Ключевые термины

База рекурсии – это тривиальный случай, при котором решение задачи очевидно, то есть не требуется обращение функции к себе.

Глубина рекурсивных вызовов – это наибольшее одновременное количество рекурсивных обращений функции, определяющее максимальное количество слоев рекурсивного стека.

Декомпозиция – это выражение общего случая через более простые подзадачи с измененными параметрами.

Корень полного дерева рекурсивных вызовов – это вершина полного дерева рекурсии, соответствующая начальному обращению к функции.

Косвенная (взаимная) рекурсия – это последовательность взаимных вызовов нескольких функций, организованная в виде циклического замыкания на тело первоначальной функции, но с иным набором параметров.

Объем рекурсии - это характеристика сложности рекурсивных вычислений для конкретного набора параметров, представляющая собой количество вершин полного рекурсивного дерева без единицы.

Параметризация – это выделение из постановки задачи параметров, которые используются для описания условия задачи и решения.

Полное дерево рекурсии – это граф, вершинами которого являются наборы фактических параметров при всех вызовах функции, начиная с первого обращения к ней, а ребрами – пары таких наборов, соответствующих взаимным вызовам.

Прямая рекурсия – это непосредственное обращение рекурсивной функции к себе, но с иным набором входных данных.

Рекурсивная триада – это этапы решения задач рекурсивным методом.

Рекурсивная функция – это функция, которая в своем теле содержит обращение к самой себе с измененным набором параметров.

Рекурсивный алгоритм – это алгоритм, в определении которого содержится прямой или косвенный вызов этого же алгоритма.

Рекурсия – это определение объекта посредством ссылки на себя.

Краткие итоги

1. Рекурсия характеризуется определением объекта посредством ссылки на себя.
2. Рекурсивные алгоритмы содержат в своем теле прямое или опосредованное обращение с самим себе.
3. Рекурсивные функции содержат в своем теле обращение к самим себе с измененным набором параметров в виде прямой рекурсии. При этом обращение к себе может быть организовано посредством косвенной рекурсии – через цепочку взаимных обращений функций, замыкающихся в итоге на первоначальную функцию.
4. Решение задач рекурсивными способами проводится посредством разработки рекурсивной триады.
5. Целесообразность применения рекурсии в программировании обусловлена спецификой задач, в постановке которых явно или опосредовано указывается на возможность сведения задачи к подзадачам, аналогичным самой задаче.
6. Рекурсивные методы решения задач широко используются при моделировании задач из различных предметных областей.
7. Рекурсивные алгоритмы относятся к ресурсоемким алгоритмам. Для оценки сложности рекурсивных алгоритмов учитывается число вершин полного рекурсивного дерева, количество передаваемых параметров, временные затраты на организацию стековых слоев.

# Построение графиков функций на С++ с использованием Excel

Для построения графика функции в Excel необходимо подготовить и сохранить данные для графика в файле csv. В таком файле разделение ячеек происходит с помощью разделителей в виде точки с запятой. Рассмотрим пример построение параболы на целочисленных значениях. Нижеследующий код позволяет сгенерировать файл example.csv, в первом столбце которого будут хранится значение x, во втором - y.

#include <iostream>

#include <fstream>

int main()

{

// открытие файла для записи

std::ofstream myfile;

myfile.open("example.csv");

myfile << "X;Y\n"; // запись заголовка из двух ячеек

int y;

// цикл генерации значения для графика

for (int x = -8; x <= 8; x++)

{

y = x \* x; // вычисление y

myfile << x; // запись в файл столбца с x

myfile << ";"; // запись разделителя ячеек

myfile << y; // запись в файл столбца с y

myfile << "\n"; // запись в файл перехода на новую строку

}

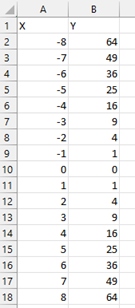
myfile.close(); // закрытие

return 0;

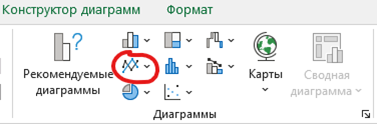
}

После запуска программы в папке с проектом будет создан файл example.csv, который

можно открыть в Excel:



Выделяем второй столбец (ячейки B2:B18) и в меню Вставка выбираем на панели вставку графика функции.



В результате получаем график:



Для построения графика двух функций модифицируем код таким образом:

#include <iostream>

#include <fstream>

int main()

{

// открытие файла для записи

std::ofstream myfile;

myfile.open("example.csv");

myfile << "X;Y2;Y3\n"; // запись заголовка из двух ячеек

int y2, y3;

// цикл генерации значения для графика

for (int x = -8; x <= 8; x++)

{

y2 = x \* x; // вычисление y

y3 = x \* x \* x;

myfile << x; // запись в файл столбца с x

myfile << ";"; // запись разделителя ячеек

myfile << y2; // запись в файл столбца с y2

myfile << ";"; // запись разделителя ячеек

myfile << y3; // запись в файл столбца с y3

myfile << "\n"; // запись в файл перехода на новую строку

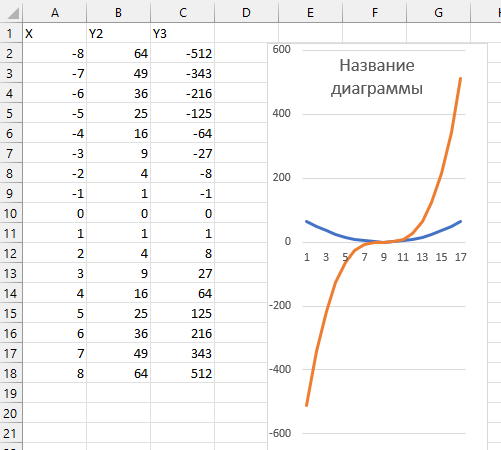
}

myfile.close(); // закрытие

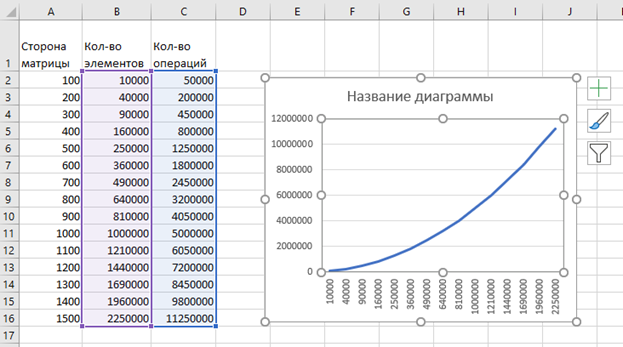
return 0;

}

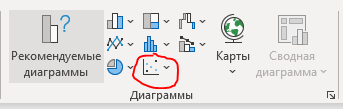
В результате csv файл в Excel со вставленным графиком имеет вид:



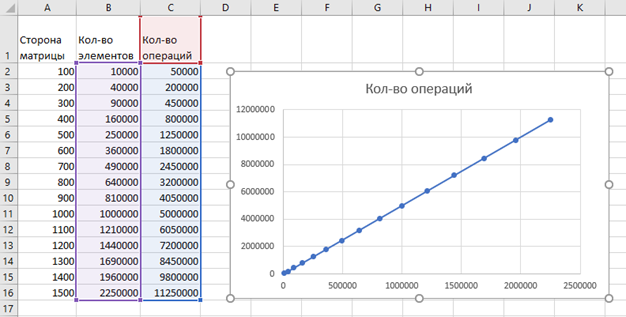
При построении графиков функций может возникнуть ситуация неравномерных замеров по оси **Х**. Например, если в задаче используются квадратные матрицы и сторона матрицы увеличивается на константу, то есть берутся матрицы 100 на 100, далее 200 на 200, далее 300 на 300 и т.д. По оси **X** откладывается размерность входных данных, которая в соответствии с заданием определяется как число элементов в матрице. Для нашего случая это будет 10 000, 40 000, 90 000 и т.д., то есть между первой и второй точкой будет 30 000, между второй и третьей 50 000 и т.д. В этом случае вставленный график в Excel может строится некорректно, даже при выборе правильных подписей данных:



Для решения этой проблемы можно использовать точечный график:



Для нашего примера, выделите два столбца **B** и **C**, далее Вставка, далее любую точечную диаграмму. Столбец **B** будет содержать значение по оси **X**, столбец **C** значения по **Y**. Получим корректный график:



Компонент Chart в Winfroms (C#) и подобный в Java представляют графики функций корректно.