# Краткий конспект к курсу

Оглавление

[Краткий конспект к курсу 1](#_Toc502756135)

[1. Линейный поиск в массиве 2](#_Toc502756136)

[2. Поиск с барьером 3](#_Toc502756137)

[3. Бинарный поиск 4](#_Toc502756138)

[4. Поиск минимума/максимума 5](#_Toc502756139)

[5. Усовершенствованный поиск минимума/максимум 6](#_Toc502756140)

[6. Сортировка пузырьком 9](#_Toc502756141)

[7. Сортировка выбором 10](#_Toc502756142)

[8. Сортировка вставками 12](#_Toc502756143)

[9. Сортировка Бинарными вставками 14](#_Toc502756144)

[10. Быстрая сортировка 15](#_Toc502756145)

[11. Последовательность Фибоначчи 19](#_Toc502756146)

[12. Ханойские башни 20](#_Toc502756147)

[13. Алгоритм Евклида 24](#_Toc502756149)

[14. Решето Эратосфена 26](#_Toc502756150)

# 1. Линейный поиск в массиве

Простейший алгоритм поиска - линейный. Эталонный массив просматривается последовательно от первого до последнего элемента.

Наихудший случай - слова нет в массиве (не найдено) – вывод можно сделать после просмотра всего массива.

Достоинство – простота реализации. Недостаток – большое время.

Если используется for всегда выполняется ровно n операций сравнения, независимо от того, найдено слово или нет. Разумно прекращать поиск если, слово найдено (если не требуется найти все вхождения слова).

При равномерном распределении элементов в массиве среднее время поиска обычно пропорционально величине n/2.

Во всех ниже изложенных алгоритмах будем считать, что производится поиск в массиве A из N целых чисел элемента, равного X.

Различают первое вхождение элемента в список, последнее вхождение, все вхождения.

Литература:

1. Презентация «Паскаль\_Массивы» 41-43 слайды.

# 2. Поиск с барьером

Идея поиска с барьером : не проверять каждый раз в цикле условие достижения границы массива.

Это обеспечивают, установив в массиве: любой элемент, который удовлетворяет условию поиска. Тем самым будет ограничено изменение индекса. В качестве барьера можно установить искомый элемент добавив его в конец массива.

Достоинство: на одну проверку меньше в цикле.

Выход из цикла, в котором теперь остается только условие поиска, может произойти либо на найденном элементе, либо на барьере. Таким образом, после выхода из цикла проверяется, не барьер ли мы нашли?

Вычислительная сложность поиска с барьером меньше, чем у линейного поиска, но имеет тот же порядок.

Существует два способа установки барьера: дополнительный элемент или вместо крайнего элемента массива.

# 3. Бинарный поиск

Алгоритм двоичного поиска используется только в упорядоченных массивах по требуемому свойству.

Так при поиске числа с заданным значением необходимо иметь массив, упорядоченный по возрастанию или по убыванию значений элементов. А, например, при поиске числа с заданной суммой цифр массив должен быть упорядочен по возрастанию или по убыванию сумм цифр элементов.

Идея алгоритма: массив каждый раз делится пополам и выбирается та часть, где может находиться нужный элемент. Деление продолжается пока подмассив больше одного элемента, после чего остается проверить этот оставшийся элемент на выполнение условия поиска.

Существуют две модификации этого алгоритма для поиска первого и последнего вхождения. Все зависит от того, как выбирается средний элемент: округлением в меньшую или большую сторону. В первом случае средний элемент относится к левой части массива, а во втором - к правой.

В алгоритме двоичного поиска размер фрагмента, где этот поиск должен продолжаться, каждый раз уменьшается примерно в два раза. Это обеспечивает вычислительную сложность алгоритма порядка логарифма N по основанию 2, где N - количество элементов массива.

Литература:

1. <http://algowiki-project.org/ru/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA>
2. Презентация «Паскаль\_Массивы» 81-85 слайды.

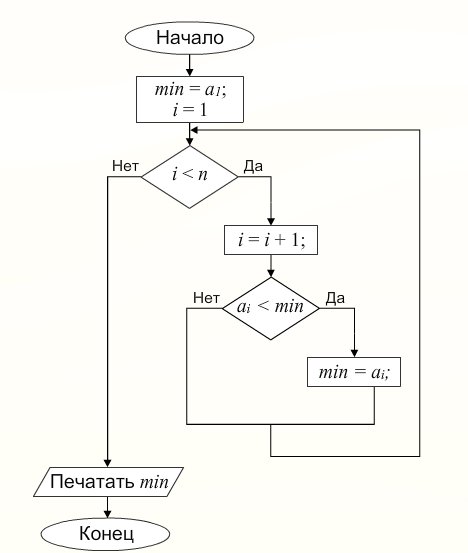
# 4. Поиск минимума/максимума

При поиске минимума или максимума используют дополнительную переменная min (или max):

1) промежуточной переменной присваивается значение первого числа из последовательности, т.е. принимается, что первое число является текущим минимумом (максимумом);

2) начиная со второго числа, производится сравнение этого числа со значением переменной min (или max) и если число из массива меньше min (больше max), то на место min (max) записывается это число. Теперь это число будет текущим минимумом (максимумом);

После просмотра всех чисел в переменной min (или max) будет находиться окончательное значение минимума (или максимума).



Литература:

1. Презентация «Паскаль\_Массивы» 16-19 слайды.

# 5. Усовершенствованный поиск минимума/максимум

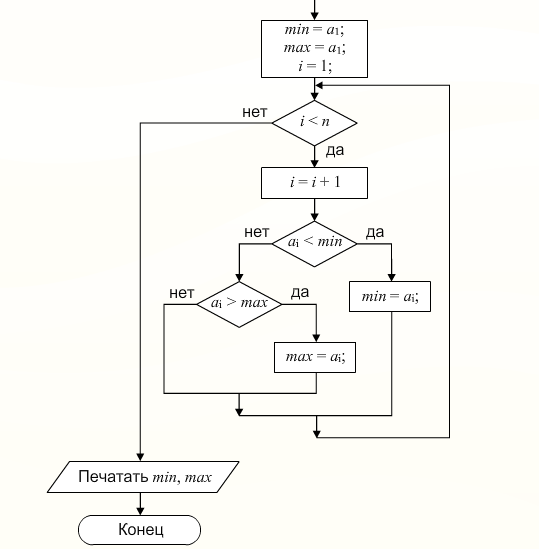
Можно еще усложнить задачу и найти одновременно максимальный и минимальный элементы. Скорректируем вышеприведенный алгоритм.

*Алгоритм одновременного поиска максимального и минимального элементов в неупорядоченном массиве:*

1. Установить счетчик равным 1 (i = 1).
2. Положим значения текущего минимума и текущего максимума равными первому исследуемому элементу (min = a1, max = a1).
3. Если исследованы еще не все элементы i < n), то перейти к шагу 4, иначе алгоритм окончен (минимальный и максимальный элементы равны min и max соответственно).
4. Перейти к следующему элементу (увеличить i на единицу).
5. Если текущий элемент меньше чем минимум (ai < min), то присвоить min значение ai, иначе если текущий элемент больше, чем максимум (ai > max), то присвоить max значение ai
6. Перейти к шагу 3.

Сложность этого алгоритма равна 2 · (n – 1). Возникает вопрос, существует ли алгоритм одновременного поиска минимального и максимального элемента, сложность которого меньше, чем 2 · (n – 1)? Оказывается, такой алгоритм существует, и его сложность равна 3 (n/2) .

Запишем этот алгоритм в виде блок-схемы:



*Эффективный алгоритм поиска максимального элемента в неупорядоченном массиве:*

1. Разбить массив на пары (получим n/2 пар, при нечетном n плюс еще одна неполная пара).
2. Упорядочить по возрастанию каждую пару (при выполнении этого шага будет выполнено (n/2) сравнений). Тогда в массиве на всех нечетных местах будут стоять минимальные для каждой пары числа, а на всех четных местах – максимальные.
3. Найти минимальное число, осуществляя поиск только среди элементов, стоящих на нечетных местах. При этом если у нас есть неполная пара, то в качестве начального значения переменной min взять значение элемента неполной пары (при выполнении этого шага будет выполнено (n/2) сравнений).
4. Найти максимальное число, осуществляя поиск только среди элементов, стоящих на четных местах. При этом если у нас есть неполная пара, то в качестве начального значения переменной max взять значение элемента неполной пары (при выполнении этого шага будет выполнено (n/2) сравнений).

Всего сравнений в этом алгоритме 3(n/2).

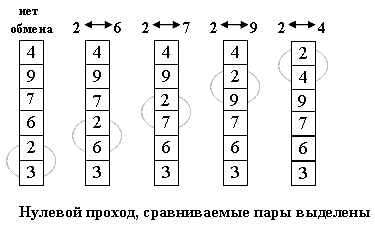
Литература:

1. [http://school-collection.lyceum62.ru/ecor/storage/3f333108-e231-4ffa-b01f-e4b03b056dea/[INF10\_04\_12\_TI].html#3](http://school-collection.lyceum62.ru/ecor/storage/3f333108-e231-4ffa-b01f-e4b03b056dea/%5bINF10_04_12_TI%5d.html%233)

# 6. Сортировка пузырьком

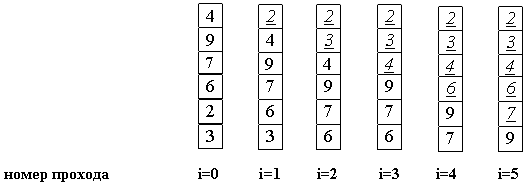
Расположим массив сверху вниз, от нулевого элемента - к последнему.

Идея метода: шаг сортировки состоит в проходе снизу вверх по массиву. По пути просматриваются пары соседних элементов. Если элементы некоторой пары находятся в неправильном порядке, то меняем их местами.



После нулевого прохода по массиву "вверху" оказывается самый "легкий" элемент - отсюда аналогия с пузырьком. Следующий проход делается до второго сверху элемента, таким образом второй по величине элемент поднимается на правильную позицию...

Делаем проходы по все уменьшающейся нижней части массива до тех пор, пока в ней не останется только один элемент. На этом сортировка заканчивается, так как последовательность упорядочена по возрастанию.



Среднее число сравнений и обменов имеют квадратичный порядок роста: Theta(n2), отсюда можно заключить, что алгоритм пузырька очень медленен и малоэффективен.

Литература:

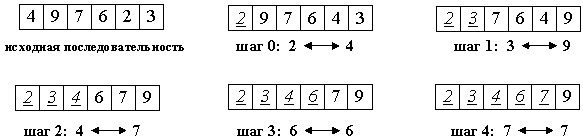
1. Презентация «Паскаль\_Массивы» 63-69 слайды.
2. <https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%BF%D1%83%D0%B7%D1%8B%D1%80%D1%8C%D0%BA%D0%BE%D0%BC>

# 7. Сортировка выбором

Идея метода состоит в том, чтобы создавать отсортированную последовательность путем присоединения к ней одного элемента за другим в правильном порядке.

Будем строить готовую последовательность, начиная с левого конца массива. Алгоритм состоит из n последовательных шагов, начиная от нулевого и заканчивая (n-1)-м.

На i-м шаге выбираем наименьший из элементов a[i] ... a[n] и меняем его местами с a[i]. Последовательность шагов при n=5 изображена на рисунке ниже.



Вне зависимости от номера текущего шага i, последовательность a[0]...a[i] (выделена курсивом) является упорядоченной. Таким образом, на (n-1)-м шаге вся последовательность, кроме a[n] оказывается отсортированной, а a[n] стоит на последнем месте по праву: все меньшие элементы уже ушли влево.

Для нахождения наименьшего элемента из n+1 рассматримаемых алгоритм совершает n сравнений. С учетом того, что количество рассматриваемых на очередном шаге элементов уменьшается на единицу, общее количество операций:

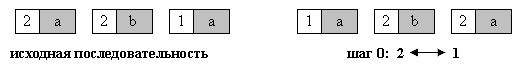
n + (n-1) + (n-2) + (n-3) + ... + 1 = 1/2 \* ( n2+n ) = Theta(n2).

Таким образом, так как число обменов всегда будет меньше числа сравнений, время сортировки растет квадратично относительно количества элементов.

Алгоритм не использует дополнительной памяти: все операции происходят "на месте".

Устойчив ли этот метод? Прежде, чем читать далее, попробуйте получить ответ самостоятельно.

Рассмотрим последовательность из трех элементов, каждый из которых имеет два поля, а сортировка идет по первому из них.



Результат ее сортировки можно увидеть уже после шага 0, так как больше обменов не будет. Порядок ключей 2a, 2b был изменен на 2b, 2a, так что метод неустойчив.

Если входная последовательность почти упорядочена, то сравнений будет столько же, значит алгоритм ведет себя неестественно.

Литература:

1. Презентация «Паскаль\_Массивы» 70-71 слайды.
2. <https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%B2%D1%8B%D0%B1%D0%BE%D1%80%D0%BE%D0%BC>

# 8. Сортировка вставками

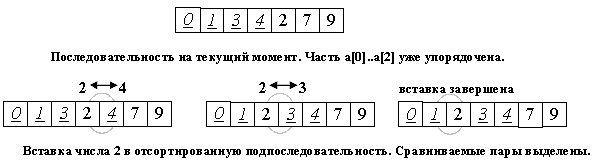
Сортировка простыми вставками в чем-то похожа на вышеизложенные методы.

Аналогичным образом делаются проходы по части массива, и аналогичным же образом в его начале "вырастает" отсортированная последовательность...

Однако в сортировке пузырьком или выбором можно было четко заявить, что на i-м шаге элементы a[0]...a[i] стоят на правильных местах и никуда более не переместятся. Здесь же подобное утверждение будет более слабым: последовательность a[0]...a[i] упорядочена. При этом по ходу алгоритма в нее будут вставляться (см. название метода) все новые элементы.

Будем разбирать алгоритм, рассматривая его действия на i-м шаге. Как говорилось выше, последовательность к этому моменту разделена на две части: готовую a[0]...a[i] и неупорядоченную a[i+1]...a[n].

На следующем, (i+1)-м каждом шаге алгоритма берем a[i+1] и вставляем на нужное место в готовую часть массива.   
Поиск подходящего места для очередного элемента входной последовательности осуществляется путем последовательных сравнений с элементом, стоящим перед ним.  
В зависимости от результата сравнения элемент либо остается на текущем месте(вставка завершена), либо они меняются местами и процесс повторяется.



Таким образом, в процессе вставки мы "просеиваем" элемент x к началу массива, останавливаясь в случае, когда

1. Hайден элемент, меньший x или
2. Достигнуто начало последовательности.

Аналогично сортировке выбором, среднее, а также худшее число сравнений и пересылок оцениваются как Theta(n2), дополнительная память при этом не используется.

Хорошим показателем сортировки является весьма естественное поведение: почти отсортированный массив будет досортирован очень быстро. Это, вкупе с устойчивостью алгоритма, делает метод хорошим выбором в соответствующих ситуациях.

Алгоритм можно слегка улучшить. Заметим, что на каждом шаге внутреннего цикла проверяются 2 условия. Можно объединить из в одно, поставив в начало массива специальный сторожевой элемент. Он должен быть заведомо меньше всех остальных элементов массива.

http://algolist.manual.ru/sort/gif/9.gif

Тогда при j=0 будет заведомо верно a[0] <= x. Цикл остановится на нулевом элементе, что и было целью условия j>=0.

Таким образом, сортировка будет происходить правильным образом, а во внутреннем цикле станет на одно сравнение меньше. С учетом того, что оно производилось Theta(n2) раз, это - реальное преимущество. Однако, отсортированный массив будет не полон, так как из него исчезло первое число. Для окончания сортировки это число следует вернуть назад, а затем вставить в отсортированную последовательность a[1]...a[n].

http://algolist.manual.ru/sort/gif/10.gif

Литература:

1. <https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%B2%D1%81%D1%82%D0%B0%D0%B2%D0%BA%D0%B0%D0%BC%D0%B8>

# 9. Сортировка Бинарными вставками

Алгоритм сортировки вставками можно легко улучшить, пользуясь тем, что готовая последовательность a[1],..., a[i-1], в которую нужно включить новый элемент, уже упорядочена. Поэтому место включения можно найти значительно быстрее, применив бинарный поиск, который исследует средний элемент готовой последовательности и продолжает деление пополам, пока не будет найдено место включения. Модифицированный алгоритм сортировки называется сортировкой бинарными включениями.

Литература:

1. <https://sites.google.com/site/arraylazarus/sheme/sheme-05>

# 10. Быстрая сортировка

"Быстрая сортировка", хоть и была разработана более 40 лет назад, является наиболее широко применяемым и одним их самых эффективных алгоритмов.

Метод основан на подходе "разделяй-и-властвуй". Общая схема такова:

1. из массива выбирается некоторый опорный элемент a[i],
2. запускается процедура разделения массива, которая перемещает все ключи, меньшие, либо равные a[i], влево от него, а все ключи, большие, либо равные a[i] - вправо,
3. теперь массив состоит из двух подмножеств, причем левое меньше, либо равно правого,  
   http://algolist.manual.ru/sort/gif/22.gif
4. для обоих подмассивов: если в подмассиве более двух элементов, рекурсивно запускаем для него ту же процедуру.

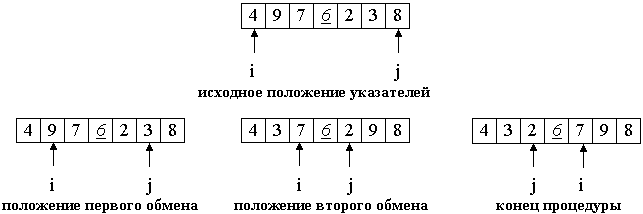
В конце получится полностью отсортированная последовательность.

Рассмотрим алгоритм подробнее.

На входе массив a[0]...a[N] и опорный элемент p, по которому будет производиться разделение.

1. Введем два указателя: i и j. В начале алгоритма они указывают, соответственно, на левый и правый конец последовательности.
2. Будем двигать указатель i с шагом в 1 элемент по направлению к концу массива, пока не будет найден элемент a[i] >= p. Затем аналогичным образом начнем двигать указатель j от конца массива к началу, пока не будет найден a[j] <= p.
3. Далее, если i <= j, меняем a[i] и a[j] местами и продолжаем двигать i,j по тем же правилам...
4. Повторяем шаг 3, пока i <= j.

Рассмотрим работу процедуры для массива a[0]...a[6] и опорного элемента p = a[3].



Теперь массив разделен на две части: все элементы левой меньше либо равны p, все элементы правой - больше, либо равны p. Разделение завершено.

Псевдокод.

quickSort ( массив a, верхняя граница N ) {

Выбрать опорный элемент p - середину массива

Разделить массив по этому элементу

Если подмассив слева от p содержит более одного элемента,

вызвать quickSort для него.

Если подмассив справа от p содержит более одного элемента,

вызвать quickSort для него.

}

Каждое разделение требует, очевидно, Theta(n) операций. Количество шагов деления(глубина рекурсии) составляет приблизительно log n, если массив делится на более-менее равные части. Таким образом, общее быстродействие: O(n log n), что и имеет место на практике.

Однако, возможен случай таких входных данных, на которых алгоритм будет работать за O(n2) операций. Такое происходит, если каждый раз в качестве центрального элемента выбирается максимум или минимум входной последовательности. Если данные взяты случайно, вероятность этого равна 2/n. И эта вероятность должна реализовываться на каждом шаге... Вообще говоря, малореальная ситуация.

Метод неустойчив. Поведение довольно естественно, если учесть, что при частичной упорядоченности повышаются шансы разделения массива на более равные части.

Сортировка использует дополнительную память, так как приблизительная глубина рекурсии составляет O(log n), а данные о рекурсивных подвызовах каждый раз добавляются в стек.

*Модификации кода и метода*

1. Из-за рекурсии и других "накладных расходов" Quicksort может оказаться не столь уж быстрой для коротких массивов. Поэтому, если в массиве меньше CUTOFF элементов (константа зависит от реализации, обычно равна от 3 до 40), вызывается сортировка вставками. Увеличение скорости может составлять до 15%.

Для проведения метода в жизнь можно модифицировать функцию quickSortR, заменив последние 2 строки на

if ( j > CUTOFF ) quickSortR(a, j);

if ( N > i + CUTOFF ) quickSortR(a+i, N-i);

Таким образом, массивы из CUTOFF элементов и меньше досортировываться не будут, и в конце работы quickSortR() массив разделится на последовательные части из <=CUTOFF элементов, отсортированные друг относительно друга. Близкие элементы имеют близкие позиции, поэтому, аналогично сортировке Шелла, вызывается insertSort(), которая доводит процесс до конца.

template<class T>

void qsortR(T \*a, long size) {

quickSortR(a, size-1);

insertSort(a, size); // insertSortGuarded быстрее, но нужна функция setmax()

}

1. В случае явной рекурсии, как в программе выше, в стеке сохраняются не только границы подмассивов, но и ряд совершенно ненужных параметров, таких как локальные переменные. Если эмулировать стек программно, его размер можно уменьшить в несколько раз.
2. Чем на более равные части будет делиться массив - тем лучше. Потому в качестве опорного целесообразно брать средний из трех, а если массив достаточно велик - то из девяти произвольных элементов.
3. Пусть входные последовательности очень плохи для алгоритма. Например, их специально подбирают, чтобы средний элемент оказывался каждый раз минимумом. Как сделать QuickSort устойчивой к такому "саботажу" ? Очень просто - выбирать в качестве опорного случайный элемент входного массива. Тогда любые неприятные закономерности во входном потоке будут нейтрализованы. Другой вариант - переставить перед сортировкой элементы массива случайным образом.
4. Быструю сортировку можно использовать и для двусвязных списков. Единственная проблема при этом - отсутствие непосредственного доступа к случайному элементу. Так что в качестве опорного приходится выбирать первый элемент, и либо надеяться на хорошие исходные данные, либо случайным образом переставить элементы перед сортировкой.

Рассмотрим наихудший случай, когда случайно выбираемые опорные элементы оказались очень плохими(близкими к экстремумам). Вероятность этого чрезвычайно мала, уже при n = 1024 она меньше 2-50, так что интерес скорее теоретический, нежели практический. Однако, поведение "быстрой сортировки" является "эталоном" для аналогично реализованных алгоритмов типа "разделяй-и-властвуй". Не везде можно свести вероятность худшего случая практически к нулю, поэтому такая ситуация заслуживает изучения.

Пусть, для определенности, каждый раз выбирается наименьший элемент amin . Тогда процедура разделения переместит этот элемент в начало массива и на следующий уровень рекурсии отправятся две части: одна из единственного элемента amin, другая содержит остальные n-1 элемента массива. Затем процесс повторится для части из (n-1) элементов.. И так далее.

При использовании рекурсивного кода, подобного написанному выше, это будет означать n вложенных рекурсивных вызовов функции quickSort.  
Каждый рекурсивный вызов означает сохранение информации о текущем положении дел. Таким образом, сортировка требует O(n) дополнительной памяти.. И не где-нибудь, а в стеке. При достаточно большом n такое требование может привести к непредсказуемым последствиям.

Для исключения подобной ситуации можно заменить рекурсию на итерации, реализовав стек на основе массива. Процедура разделения будет выполняться в виде цикла.

Каждый раз, когда массив делится на две части, в стек будет направляться запрос на сортировку большей из них, а меньшая будет обрабатываться на следующей итерации. Запросы будут выбираться из стека по мере освобождения процедуры разделения от текущих задач. Сортировка заканчивает свою работу, когда запросы кончаются.

Псевдокод.

Итеративная QuickSort (массив a, размер size) {

Положить в стек запрос на сортировку массива от 0 до size-1.

do {

Взять границы lb и ub текущего массива из стека.

do {

1. Произвести операцию разделения над текущим массивом a[lb..ub].

2. Отправить границы большей из получившихся частей в стек.

3. Передвинуть границы ub, lb чтобы они указывали на меньшую часть.

} пока меньшая часть состоит из двух или более элементов

} пока в стеке есть запросы

}

# 11. Последовательность Фибоначчи

Вычисление n-го члена последовательности Фибоначчи.

Последовательность Фибоначчи определяется следующим образом: первые два числа равны 1, а каждое следующее равно сумме двух предыдущих (1, 1, 2, 3, 5,…).

Вычисление n-го члена последовательности Фибоначчи можно определить следующим образом:

Fib(n)= 

Литература:

1. <https://dic.academic.ru/dic.nsf/ntes/3683/%D0%9F%D0%9E%D0%A1%D0%9B%D0%95%D0%94%D0%9E%D0%92%D0%90%D0%A2%D0%95%D0%9B%D0%AC%D0%9D%D0%9E%D0%A1%D0%A2%D0%AC>

# 12. Ханойские башни

Есть три стержня A, B, и C. На стержень A надето N дисков, наверху самый маленький, каждый следующий диск больше предыдущего, а внизу самый большой. На другие стержни дисков не надето.

Hеобходимо перенести диски со стержня A на стержень C, пользуясь стержнем B, как вспомогательным, так, чтобы диски на стержне C располагались в том же порядке, в каком они располагаются на диске A перед перемещением.

При перемещении никогда нельзя класть больший диск на меньший.

*Рекурсивный метод*

Для того, чтобы переложить всю пирамиду, надо сначала переложить все, что выше самого большого диска, с первого на вспомогательный стержень, потом переложить это самое большой диск с первого на третий стержень, а потом переложить оставшуюся пирамиду со второго на третий стержень, пользуясь первым стержнем, как вспомогательным.

/\*

данная процедура переносит N дисков со стержня A на стержень C

пользуясь B как вспомогательным, соблюдая правила

\*/

процедура "Перенести" (A, B, C, N)

начало

если N=1

// Если диск всего один, то надо его перенести напрямую

то

снять диск со стержня A и положить на стержень C;

возврат из процедуры;

иначе

// Переносим все диски, кроме самого большога со стежня

// A на стержень B

Перенести (A,C,B,N-1);

// Переносим самый большой диск со стержня A на стержень C

снять диск со стержня A и положить на стержень C;

// Переносим все диски со стержня B на стержень C поверх

// самого большого диска

Перенести (B,A,C,N-1);

возврат из процедуры;

конец если;

конец процедуры "Перенести";

Всего получается 2N-1 перекладываний.

### *Нерекурсивный метод*

Стержню, на котором диски находятся в начале, дадим номер 0; стержню, на который их надо перенести - номер 2; и, соответственно, оставшемуся стержню - номер 1.

Пусть всего дисков N. Занумеруем диски в порядке увеличения радиуса числами 0,1,2,...,N-1.

Как известно, задача решается за 2N-1 ходов. Занумеруем ходы числами 1,2,...,2N-1.

Любое натуральное число i единственным образом представимо в виде i=(2t+1)\*2k, где t и k - целые (т.е. как произведение нечетного числа на некоторую степень двойки). Так вот, на i-ом ходе переносится диск номер k со стержня номер ((-1)N-k\*t mod 3) на стержень номер ((-1)N-k\*(t+1) mod 3).

Пример для N=4.

Ход k(диск) t Со\_стержня Hа\_стержень Стержни

|)!'

1 0 0 0 1 |)! '

2 1 0 0 2 |) ' !

3 0 1 1 2 |) !'

4 2 0 0 1 | ) !'

5 0 2 2 0 |' ) !

6 1 1 2 1 |' )!

7 0 3 0 1 | )!'

8 3 0 0 1 )!' |

9 0 4 1 2 )! |'

10 1 2 1 0 ! ) |'

11 0 5 2 0 !' ) |

12 2 1 1 2 !' |)

13 0 6 0 1 ! ' |)

14 1 3 0 2 ' |)!

15 0 7 1 2 |)!'

если пpедставить что стержни, на котоpые одеваются диски, pасположены в yглах pавностоpоннего тpеyгольника, то самый маленький диск каждым нечетным ходом движется по (или пpотив, это от пеpвоначального кол-ва дисков зависит) часовой стpелки.

Все четные ходы опpеделяются однозначно. Какой диск меньше - тот и перекладывать (иначе противоречит условию). Т.к. тpогать диск 0 нельзя и класть больший на меньший тоже нельзя.

Отметим две закономерности:

1. Hа каждом нечетном ходy происходит перенос наименьшего диска.
2. Hаименьший диск всегда переносится циклически: либо A-B-C-A-B-C-... (в слyчае четного количества дисков), либо A-C-B-A-C-B-... (в слyчае нечетного).

А посемy полyчаем алгоритм:

1. Определяем число дисков, откyда находим как бyдет перемещаться наименьший диск (данный шаг делается в начале, притом один раз).

2. Смотрим номер хода: если нечетный - переносим наименьший диск в направлении, определенном в п.1. если четный - то возможный ход один единственный - берем наименьший из двyх верхних дисков и переносим его.

Можно написать немного по другому:

Для N от 1 до 2k-1 выполнять  
1. В двоичном представлении N найти самый правый ненулевой разряд. Обозначим номер этого разряда t.

2. Обозначим номер стержня, на котором находится диск t через i. Переместить диск t со стержня i на стержень (i+(-1)t) mod 3.

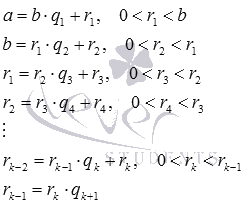
Кстати, по номеру хода легко можно восстановить положение дисков на стержнях: после i-ого хода диск номер j находится на стержне номер (-1)n-j\*((i div 2j)-(i div 2j+1)) mod 3.

Литература:

1. <https://ru.wikipedia.org/wiki/%D0%A5%D0%B0%D0%BD%D0%BE%D0%B9%D1%81%D0%BA%D0%B0%D1%8F_%D0%B1%D0%B0%D1%88%D0%BD%D1%8F>

# 13. Алгоритм Евклида

В статье [наибольший общий делитель (НОД), определение, примеры, свойства НОД](http://www.cleverstudents.ru/divisibility/nod.html)мы сформулировали и доказали [алгоритм Евклида](http://www.cleverstudents.ru/divisibility/nod.html#Euclids_algorithm). **Алгоритм Евклида** является универсальным способом, позволяющим вычислять наибольший общий делитель двух положительных целых чисел.

Напомним суть алгоритма Евклида. Для нахождения наибольшего общего делителя двух чисел a и b (a и b – [целые положительные числа](http://www.cleverstudents.ru/numbers/integers.html#positive_and_negative), причем a больше или равно b) последовательно выполняется [деление с остатком](http://www.cleverstudents.ru/numbers/division_of_integers_with_remainder.html), которое дает ряд равенств вида  
  
Деление заканчивается, когда rk+1=0, при этом rk=НОД(a, b).

Рассмотрим **примеры нахождения НОД по алгоритму Евклида**.

Пример.

Найдите наибольший общий делитель чисел 64 и 48.

Решение.

Воспользуемся алгоритмом Евклида. В этом примере a=64, b=48.

Делим 64 на 48, получаем 64:48=1 (ост. 16) (при необходимости смотрите [правила и примеры деления с остатком](http://www.cleverstudents.ru/numbers/division_of_integers_with_remainder.html)), что можно записать в виде равенства 64=48·1+16, то есть, q1=1, r1=16.

Теперь делим b на r1, то есть, 48 делим на 16, получаем 48:16=3, откуда имеем 48=16·3. Здесь q2=3, а r2=0, так как 48 делится на 16 без остатка. Мы получили r2=0, поэтому это был последний шаг алгоритма Евклида, и r1=16является искомым наибольшим общим делителем чисел 64 и 48.

Ответ:

НОД(64, 48)=16.

Покажем решение еще одного примера, но теперь обойдемся без подробных пояснений шагов алгоритма Евклида.

Пример.

Чему равен НОД чисел 111 и 432?

Решение.

Из [свойств наибольшего общего делителя](http://www.cleverstudents.ru/divisibility/nod.html#properties) мы знаем, что НОД(111, 432)=НОД(432, 111). Воспользуемся алгоритмом Евклида для нахождения НОД(432, 111).

Разделив 432 на 111, получаем равенство 432=111·3+99.

На следующем шаге делим 111 на 99, имеем 111=99·1+12.

Деление 99 на 12 дает равенство 99=12·8+3.

А 12 на 3 делится без остатка и 12=3·4. Поэтому это последний шаг алгоритма Евклида, и НОД(432, 111)=3, следовательно, и искомый наибольший общий делитель чисел 111 и 432 равен 3.

Ответ:

НОД(111, 432)=3.

Для закрепления материала найдем с помощью алгоритма Евклида наибольший общий делитель чисел 661 и 113.

Пример.

Найдите НОД(661, 113) по алгоритму Евклида.

Решение.

Выполняем деление: 661=113·5+96; 113=96·1+17; 96=17·5+11; 17=11·1+6; 11=6·1+5; 6=5·1+1, наконец, 5=1·5. Таким образом, НОД(661, 113)=1, то есть, 661 и 113 – [взаимно простые числа](http://www.cleverstudents.ru/divisibility/coprime_numbers.html).

Заметим, что если бы мы с самого начала обратились к [таблице простых чисел](http://www.cleverstudents.ru/divisibility/prime_and_composite_numbers.html#table_of_primes), то выяснили бы, что числа 661 и 113 – простые, откуда можно было бы сразу сказать, что их наибольший общий делитель равен 1.

Ответ:

НОД(661, 113)=1.

Литература:

1. Презентация «Целочисленные алгоритмы» 2- 8 слайды
2. <https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%95%D0%B2%D0%BA%D0%BB%D0%B8%D0%B4%D0%B0>

# 14. Решето Эратосфена

Дано число n. Требуется найти все простые в отрезке [2; n].

Классический способ решения этой задачи — [решето Эратосфена](http://e-maxx.ru/algo/eratosthenes_sieve). Этот алгоритм очень прост, но работает за время O (n \log \log n).

Хотя в настоящий момент известно достаточно много алгоритмов, работающих за сублинейное время (т.е. за o(n)), описываемый ниже алгоритм интересен своей простотой — он практически не сложнее классического решета Эратосфена.

Кроме того, приводимый здесь алгоритм в качестве "побочного эффекта" фактически вычисляет факторизацию всех чисел в отрезке [2; n], что может быть полезно во многих практических применениях.

Недостатком приводимого алгоритма является то, что он использует больше памяти, чем классическое решето Эратосфена: требуется заводить массив из nчисел, в то время как классическому решету Эратосфена достаточно лишь n бит памяти (что получается в 32 раза меньше).

Таким образом, описываемый алгоритм имеет смысл применять только до чисел порядка 10^7, не более.

Авторство алгоритма, по всей видимости, принадлежит Грайсу и Мисра (Gries, Misra, 1978 г. — см. список Литература в конце). (И, собственно говоря, называть данный алгоритм "решетом Эратосфена" некорректно: слишком отличаются эти два алгоритма.)

*Описание алгоритма*

Наша цель — посчитать для каждого числа i от в отрезке [2; n] его минимальный простой делитель lp[i].

Кроме того, нам потребуется хранить список всех найденных простых чисел — назовём его массивом pr[].

Изначально все величины lp[i] заполним нулями, что означает, что мы пока предполагаем все числа простыми. В ходе работы алгоритма этот массив будет постепенно заполняться.

Будем теперь перебирать текущее число i от 2 до n. У нас может быть два случая:

* lp[i] = 0 — это означает, что число i — простое, т.к. для него так и не обнаружилось других делителей.

Следовательно, надо присвоить lp[i] = i и добавить i в конец списка pr[].

* lp[i] \ne 0 — это означает, что текущее число i — составное, и его минимальным простым делителем является lp[i].

В обоих случаях дальше начинается процесс расстановки значений в массиве lp[]: мы будем брать числа, кратные i, и обновлять у них значение lp[]. Однако наша цель — научиться делать это таким образом, чтобы в итоге у каждого числа значение lp[] было бы установлено не более одного раза.

Утверждается, что для этого можно поступить таким образом. Рассмотрим числа вида:

 x_j = i \cdot p_j, 

где последовательность p_j — это все простые, не превосходящие lp[i] (как раз для этого нам понадобилось хранить список всех простых чисел).

У всех чисел такого вида проставим новое значение lp[x_j] — очевидно, оно будет равно p_j.

*Доказательство корректности*

Докажем корректность алгоритма, т.е. что он корректно расставляет все значения lp[], причём каждое из них будет установлено ровно один раз. Отсюда будет следовать, что алгоритм работает за линейное время — поскольку все остальные действия алгоритма, очевидно, работают за O (n).

Для этого заметим, что у любого числа i единственно представление такого вида:

 i = lp[i] \cdot x, 

где lp[i] — (как и раньше) минимальный простой делитель числа i, а число x не имеет делителей, меньших lp[i], т.е.:

 lp[i] \le lp[x]. 

Теперь сравним это с тем, что делает наш алгоритм — он фактически для каждого x перебирает все простые, на которые его можно домножить, т.е. простые до lp[x] включительно, чтобы получить числа в указанном выше представлении.

Следовательно, алгоритм действительно пройдёт по каждому составному числу ровно один раз, поставив у него правильное значение lp[].

Это означает корректность алгоритма и то, что он работает за линейное время.

*Время работы и требуемая память*

Хотя асимптотика O (n) лучше асимптотики O (n \log \log n) классического решета Эратосфена, разница между ними невелика. На практике это означает лишь двукратную разницу в скорости, а оптимизированные варианты решета Эратосфена и вовсе не проигрывают приведённому здесь алгоритму.

Учитывая затраты памяти, которые требует этот алгоритм — массив чисел lp[] длины n и массив всех простых pr[] длины примерно n / \ln n — этот алгоритм кажется уступающим классическому решету по всем статьям.

Однако спасает его то, что массив lp[], вычисляемый этим алгоритмом, позволяет искать факторизацию любого числа в отрезке [2; n] за время порядка размера этой факторизации. Более того, ценой ещё одного дополнительного массива можно сделать, чтобы в этой факторизации не требовались операции деления.

Знание факторизации всех чисел — очень полезная информация для некоторых задач, и этот алгоритм является одним из немногих, которые позволяют искать её за линейное время.

Литература:

1. Презентация «Целочисленные алгоритмы» 9- 14 слайды
2. <https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D1%88%D0%B5%D1%82%D0%BE_%D0%AD%D1%80%D0%B0%D1%82%D0%BE%D1%81%D1%84%D0%B5%D0%BD%D0%B0>