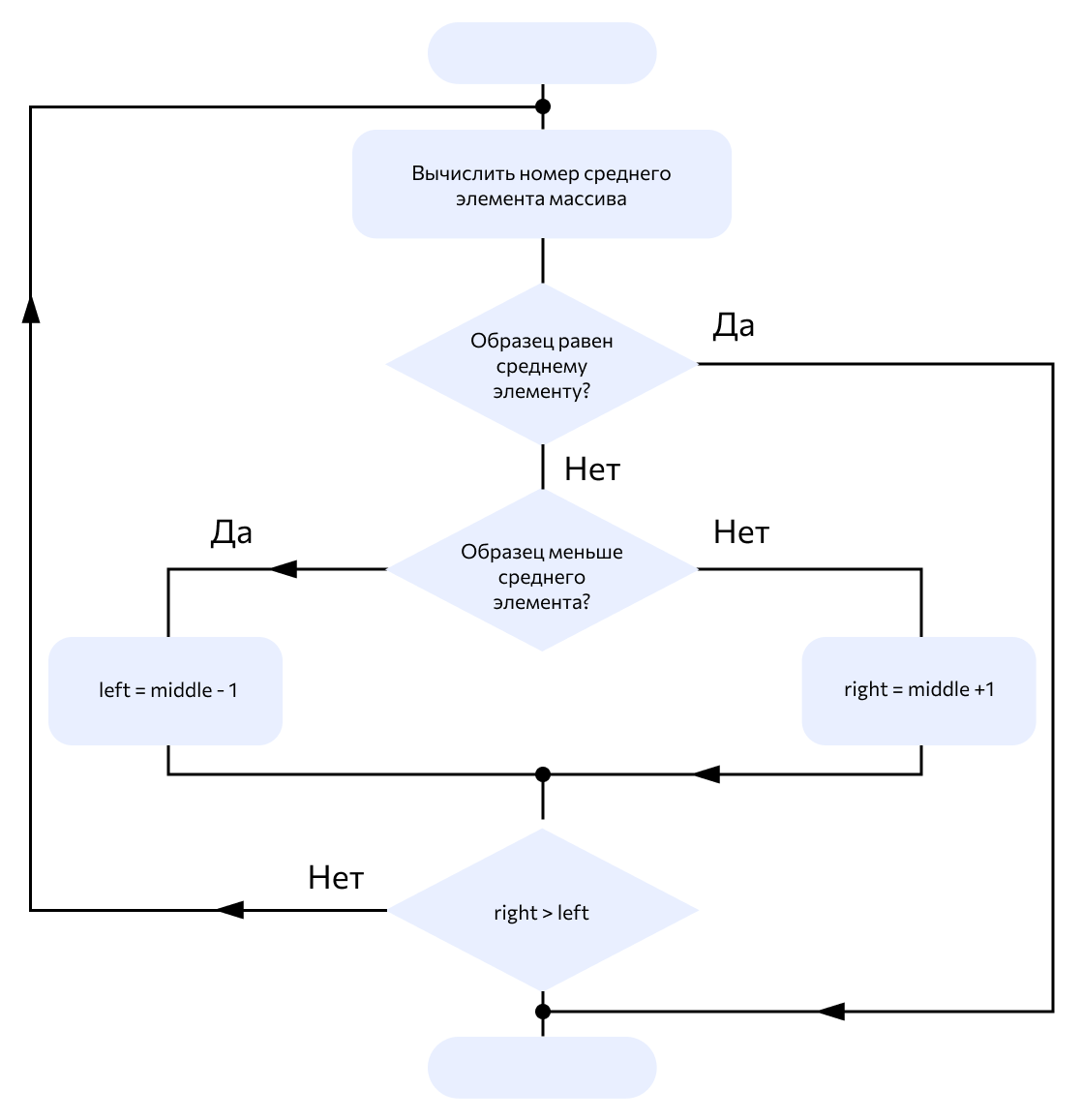
Бинарный поиск — тип поискового алгоритма, который последовательно делит пополам заранее отсортированный массив данных, чтобы обнаружить нужный элемент. Другие его названия — двоичный поиск, метод половинного деления, дихотомия.

**Принцип работы алгоритма бинарного поиска**

Основная последовательность действий алгоритма выглядит так:

1. Сортируем массив данных.
2. Делим его пополам и находим середину.
3. Сравниваем срединный элемент с заданным искомым элементом.
4. Если искомое число больше среднего — продолжаем поиск в правой части массива (если он отсортирован по возрастанию): делим ее пополам, повторяя пункт 3. Если же заданное число меньше — алгоритм продолжит поиск в левой части массива, снова возвращаясь к пункту 3.



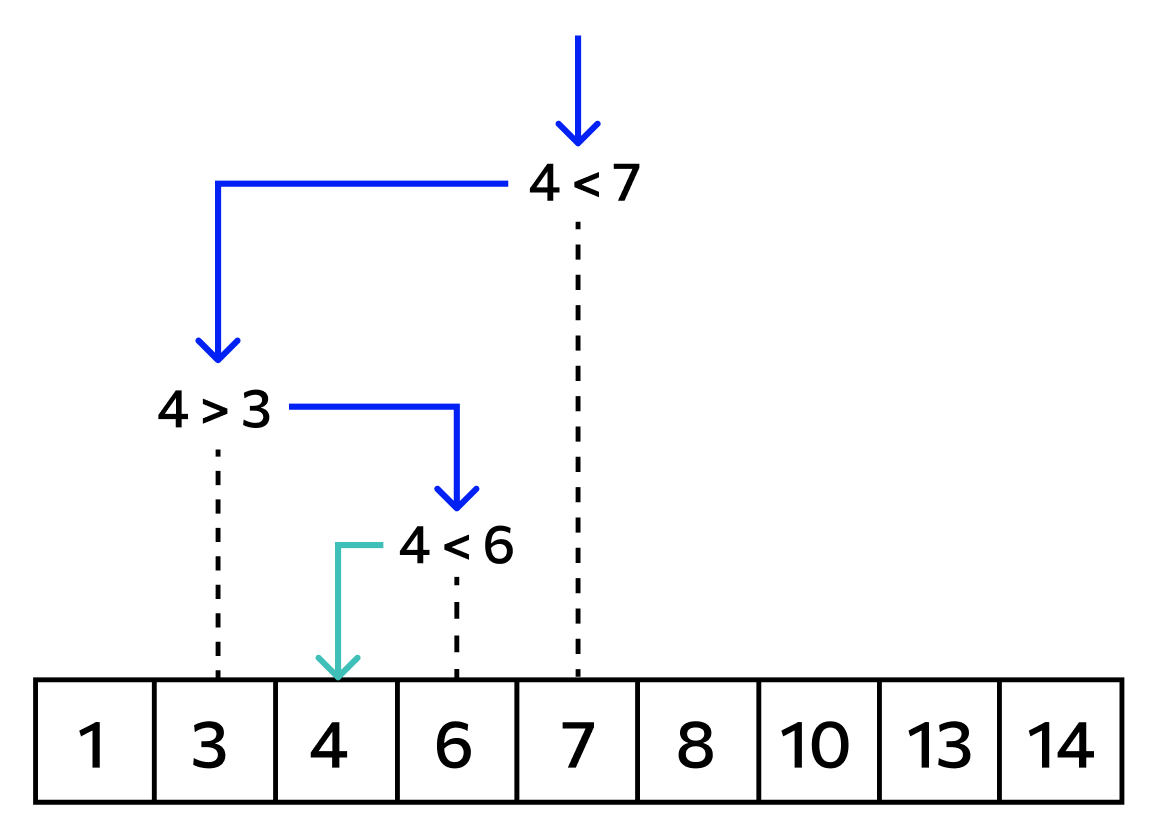
Блок-схема алгоритма двоичного поиска

**Реализация бинарного поиска**

Существуют два способа реализации бинарного поиска.

**1. Итерационный метод.** При таком подходе используется цикл, тело которого повторяется, пока не найдется заданный элемент либо не будет установлено, что его нет в массиве. Например, в Python для этой цели удобно использовать цикл while.

**2. Рекурсивный подход.** В этом случае пишется функция, которая вызывает сама себя (рекурсивно), пока не будет найден искомый элемент в массиве.



Поиск позиции заданного элемента в списке

Приведем примеры реализации этих методов на Python.

Пусть есть массив чисел [5, 8, 9, 1, 23, 7, 3, 0, 15], и необходимо найти позицию числа 5 в упорядоченном списке. На вход такой функции необходимо подать уже отсортированный массив, для этого воспользуемся встроенным методом **sorted(),** который упорядочивает массив данных по возрастанию.

Код, использующий итерационный подход, будет выглядеть так:

*def****findPosition****(num\_list, number):  
first = 0  
last = len(num\_list) — 1  
while first <= last:  
middle = first + (last — first) // 2  
if num\_list[middle] == number:  
return middle  
elif num\_list[middle] < number:  
first = middle + 1  
else:  
last = middle — 1  
return -1  
num\_list = sorted([5, 8, 9, 1, 23, 7, 3, 0, 15])  
number = 5  
print(findPosition(num\_list, number))*

При использовании рекурсивного поиска код на Python можно написать так:

*def****findPosition****(num\_list, number, first, last):  
if last >= first:  
middle = first + (last — first) // 2  
if num\_list[middle] == number:  
return middle  
elif num\_list[middle] < number:  
return findPosition(num\_list, number, middle + 1, last)  
else:  
return findPosition(num\_list, number, first, middle — 1)  
else:  
return -1  
num\_list = sorted([5, 8, 9, 1, 23, 7, 3, 0, 15])  
number = 5  
print(findPosition(num\_list, number, 0, len(num\_list) — 1))*

В некоторых языках программирования, включая Python, есть готовые функции для выполнения бинарного поиска. Модуль бинарного поиска называется **bisect.** Проиллюстрируем его работу на примере:

*from bisect import bisect\_left  
def****findPosition****(num\_list, number):  
pos = bisect\_left(num\_list, number)  
if pos < len(num\_list):  
return pos  
return False  
num\_list = sorted([5, 8, 9, 1, 23, 7, 3, 0, 15])  
number = 5  
print(findPosition(num\_list, number))*

**В каких случаях используют бинарный поиск**

Двоичный поиск подходит для нахождения позиций элемента в упорядоченном списке: в этом случае он эффективнее линейного, поскольку массив данных на каждом шаге разделяется надвое и одна половина сразу отбрасывается. Последовательная сложность двоичного метода в худшем и среднем случаях равна O(log n), в лучшем — O(1) (если обнаруживаем искомый элемент на первой итерации). Для сравнения: вычислительная сложность линейного поиска равна O(n) (обычный проход по всем элементам в поисках нужного).

У бинарного поиска есть недостаток — он требует упорядочивания данных по возрастанию. Сложность сортировки — не менее O(n log n). Поэтому, если список короткий, используется все-таки линейный поиск.

**Разновидности алгоритма**

На основе бинарного поиска разработано несколько дополнительных разновидностей поисковых алгоритмов:

* однородный бинарный поиск, при котором аргумент поиска А сравнивается с ключом Ki, где i — золотое сечение интервала (оно выбирается так, чтобы отношение длины большего отрезка к длине всего интервала равнялось отношению длины меньшего отрезка к длине большего отрезка);
* троичный поиск, когда интервал делится на три части вместо двух. Обычно применяется для поиска положения экстремума функции;
* интерполирующий поиск, который предсказывает позицию нужного элемента на основе разницы значений. Эффективен, если элементы распределены достаточно равномерно;
* дробный спуск — применяется для ускорения двоичного поиска в многомерных массивах данных, и другие.

У алгоритма бинарного поиска есть определенный затруднительный случай. Так как алгоритм всякий раз делит список пополам, можно предположить, что n = 2k-1 для k. На некотором проходе цикл имеет дело со списком из 2j-1 элементов. При j = 1 производится последняя итерация цикла, когда размер оставшейся части становится равным единице. Можно сделать вывод, что количество проходов не превышает k при n = 2k-1. Таким образом, в наихудшем случае количество проходов – k = log2(n+1).

Следовательно, алгоритм двоичного поиска имеет сложность по времени O(log2n), где n – число элементов отсортированного массива.

Метод двоичного поиска предназначен для сортированных элементов на смежной памяти фиксированного размера. Таким образом, если размерность вектора динамически изменяется, то экономия от использования двоичного поиска не компенсирует затрат на поддержание упорядоченного расположения al < a2 < … < an.

**Выводы – выбор метода поиска.**Выбор того или другого метода поиска зависит от определенных условий.

Если массив является неупорядоченным, то единственный алгоритм поиска, применимый в этом случае – последовательный. Этот метод применим для неупорядоченной информации, но также можно использовать его и на отсортированных данных. Когда данных немного, последовательный поиск работает достаточно быстро. Но если информация хранится на диске, поиск может занимать продолжительное время. Последовательный поиск достаточно прост и его легко запрограммировать, но время его работы прямо пропорционально количеству данных, которые нужно просмотреть; удвоение количества элементов приведет к удвоению времени на поиск, если искомого элемента в массиве нет. Это линейное соотношение (время выполнения является линейной функцией от размера данных), поэтому такой метод также называется линейным поиском. Потому эффективность этого метода очень низкая, так как для отыскания одного элемента в массиве размерности N в среднем нужно сделать N/2 сравнений. В лучшем случае он проверяет только один элемент, а в худшем — n.

## Только 10% программистов способны написать двоичный поиск

[Алгоритмы\*](https://habr.com/ru/hub/algorithms/)

Дональд Кнут (известный тем, что его книги никто не читает) пишет, что хотя первый двоичный поиск был опубликован в 1946 году, первый двоичный поиск *без багов* был опубликован только в 1962.  
  
Алгоритм двоичного поиска похож на то, как мы ищем слово в словаре. Открываем словарь посередине, смотрим в какой из половин будет нужное нам слово. Допустим, в первой. Открываем первую часть посередине, продолжаем половинить, пока не найдем нужное слово.  
  
С массивами так: есть упорядоченный массив, берем число из середины массива, сравниваем с искомым. Если оно оказалось больше, значит искомое число в первой половине массива, если меньше — во второй. Продолжаем делить оставшуюся половину, когда находим нужное число возвращаем его индекс, если не находим возвращаем null.  
  
  
В статье утверждалось, что только 10% программистов могут решить эту задачу. Да не может быть! Вот лохи, подумал я, зарядил Firebug, каких-то 5 минут и… нерабочая версия готова. Еще одна итерация, и еще, и еще. В сумме полтора часа, и в конечном решении все равно 2 ошибки. Стыдно как!  
  
Если вы никогда не писали двоичный поиск, я предлагаю вам написать этот алгоритм на любимом языке и выложить его в комменты *без тестирования*. Любой хороший программист сходу напишет этот поистине детский алгоритм. Потратьте столько времени, сколько нужно.   
  
Маленький гостинец для тех, кому все это кажется банальным. Почти все реализации двоичного поиска и сортировки слиянием содержат ошибки. Это говорит человек, написавший двоичный поиск для JDK.  
  
**Спойлер**.  
Распространенные ошибки:  
— не работает с массивом из 0/1/2 элементов  
— не находит первый или последний элемент  
— некорректно работает, если элемента в массиве нет  
— некорректно работает, если в массиве есть повторяющиеся элементы  
— обращение к элементами за пределами массива  
— козырная, которая была в JDK, переполнение целого при вычислении среднего индекса  
  
**P.S.**  
Кто-то скажет, что эта функция уже есть в стандартной библиотеке.  
Это так, согласен. Но это не значит, что от решения таких задач нет толку. Смотрите, если все простые задачи уже решены за нас в стандартной библиотеке, значит нам остались только более сложные задачи, которых там нет. Как мы будем решать эти более сложные задачи, если мы не умеем решить даже простую задачу из стандартной библиотеки?

## Использование бинарного поиска для оптимизации запроса на выборку данных

[MySQL\*](https://habr.com/ru/hub/mysql/)

Из песочницы

#### Введение

Сейчас очень популярна тем оптимизации работы с различными СУБД. На многочисленных форумах ведутся дискуссии о «самой лучшей СУБД в мире», но часто все это перетекает в необоснованные выкрики о том, что «я познал смысл жизни и понял, что самое лучшее хранилище данных — Х».  
  
Да, несомненно, сейчас мы можем наблюдать активное развитие NoSQL решений, которые позволяют делать многое. Но данная статья не о них. Так вышло, что я сменил работу и в нагрузку мне достался один очень интересный проект на связке php+MySQL. В нем есть много хороших решений, но он писался без расчёта на большую аудиторию. За несколько лет существования количество активных пользователей начало приближаться к числам с 7 нулями. Так как проект представляет из себя подобие социальной сети с игровыми элементами, то таблица с пользователями оказалась не самой «тяжёлой» из всех. В наследство мне достались таблицы с десятками миллионов вещей пользователей, личных сообщений, биллинговыми записями и т. п. Проект начали рефакторить, разбивать на несколько серверов и достигли значительных результатов. Сейчас все стабильно.  
  
Но недавно мне на почту прислали новую задачу. Суть заключалась в сборе статистики. Проанализировав требования я понял, что для выполнения достаточно написать один единственный запрос, выполняющий 3 INNER JOIN'а на таблицы, размеры которых впечатляли. Каждая таблица в среднем содержала 40 миллионов записей. Получается, что временная таблица состояла бы из 4\*4\*4\*10^21 = 64\*10^21 записей. Это колоссальная цифра. И загружать СУБД таким запросом для сбора статистики — непозволительная роскошь.  
  
Далее, собственно, я и хочу представить решение данной абстрактной задачи, которое пришло мне в голову, когда я вспоминал занятия по информатике на первом курсе университета.  
  
  
  
(в проекте используется СУБД MySQL, но алгоритм не имеет каких-либо специфических особенностей)

#### Что такое бинарный поиск

Думаю, что многие из вас писали лабораторные работы, посвящённые поиску элемента в массиве с использование бинарного алгоритма. Постараюсь кратко изложить его суть.  
  
Пусть у нас имеется отсортированный массив из **n** элементов:  
  
*Первый элемент массива = 1  
Последний элемент массива = n*  
  
Нам необходимо найти индекс элемента со значением **f**.  
  
Каждый шаг заключается в том, что мы вычисляем середину массива:  
  
*Середина массива = round(Первый элемент + Последний элемент)/2*  
  
Затем вычисляем значение этого элемента и проверяем больше или меньше полученное значение в сравнении с искомым **f**. Диапазон поиска уменьшается в 2 раза:  
  
*Если <значение середины> > f, то  
Последний элемент массива = значение середины  
Иначе  
Первый элемент массива = значение середины*  
  
Данные шаги повторяются пока не наступит одно из условий:

* Модуль разности значений середины и **f** меньше некоторого **эпсилон**(где эпсилон, некоторая погрешность)
* Количество итераций превысило значение **log2(количество элементов в массиве)**

Думаю, суть ясна. Таким образом мы значительно ускоряем поиск нужного элемента за счёт сокращения диапазонов поиска, но жертвуем некоторой точностью вычислений (для статистики это не критично, если пару элементов из миллионов не будут учтены. В противном случае необходимо сделать эпсилон нулевым и завершать поиск только после достижения последнего уровня дерева).

#### Переходим к практике

Итак, предположим, что нам необходимо сделать INNER JOIN на 3 таблицы, а затем задать условие «столбец **х** в диапазоне между 10 и 20». Причём столбец **х** не имеет индекса. Это будет очень долго выполнятся. Тут и приходит на помощь это простой способ.  
  
Берем таблицу с этим самым столбцом и применяем на ней бинарный поиск, для поиска диапазона первичных ключей, удовлетворяющих условию 10<=**x**<=20. Учитывая, что для подобной выборки мы будем использовать только индексы, то очень скоро мы получим нужную пару значений.  
  
Стоит сказать, что бинарный поиск используется для нахождения одного элемента, а не диапазона, но никто не мешает нам найти первый элемент со значением 10 и последний элемент со значением 20. Их первичные ключи и буду ограничениями диапазонов.  
  
С этим диапазоном мы идём снова к нашему запросу, но теперь вместо условия **WHERE x >= 10 AND x <= 20** мы пишем **WHERE id\_x BETWEEN min\_id\_x AND max\_id\_x**, где **min\_id\_x** и **max\_id\_x** — значение нижнего и верхнего краёв диапазона, удовлетворяющего условию.  
  
Что мы получаем: теперь мы делаем выборку не по некому столбцу **х**, а по первичному ключу. Время потраченное на обход одной таблицы сэкономлено. Аналогичную процедуру можно провести с другими условиями в запросе.  
  
Приводить здесь код не вижу смысла, т. к. код бинарного поиска можно найти на википедии, а сам запрос — ничего сверхъестественного.

#### Выводы

Данный алгоритм позволяет перенести условия с полей без индексов на первичные ключи, что значительно ускоряет работу запроса. Но метод нельзя считать панацеей.  
  
**Во-первых**, трудно подготовить универсальное решение для всех запросов. В любом случае будет необходимо учитывать детали реализации той или иной таблицы и, как следствие, каждый раз тратить время на оптимизацию.  
  
**Во-вторых**, данной способ подходит не для всех решений, т. к. строки в таблице должны быть отсортированы в некотором порядке.