

**Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский государственный технический университет имени Н.Э. Баумана**

**(национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)**

ФАКУЛЬТЕТ *Робототехники и комплексной автоматизации*

КАФЕДРА *Системы автоматизированного проектирования (РК-6)*

**Научно-исследовательская работа**

**“Быстрые алгоритмы подсчета и поиска двоичных разрядов”**

Студент Журавлев Н.В.

Группа РК6-42Б

Тип задания Научно-исследовательская работа

Студент

**Журавлев Н.В.**

*подпись, дата фамилия, и.о.*

Преподаватель

**Родионов С.В.**

*подпись, дата фамилия, и.о.*

*Москва, 2021*

**Оглавление**

**Оглавление2**

**Техническое задание3**

**Введение4**

**Представление чисел в компьютере4**

**Действия над битами5**

**Подсчёт единичных разрядов - наивный подход6**

**Дуальный алгоритм7**

**Предпосчёт7**

**Разделяй и властвуй8**

**Умножение и остаток от деления8**

**Умножение и сдвиг10**

**Параллельное суммирование10**

**Параллельное суммирование11**

**Комбинированный метод12**

**Подсчёт ведущих нулей: бинарный поиск13**

**Двухсторонний подсчёт ведущих нулевых битов13**

**Алгоритм Харли14**

**Вариация Горявски алгоритма Харли**1**4**

**С плавающей точкой14**

**Подсчёт завершающих нулевых битов. Общее решение16**

**Бинарный поиск16**

**Поиск по бинарному дереву16**

**Наивный метод: подсчёт завершающих нулей16**

**Алгоритм Годо17**

**Алгоритм Сила17**

**С использованием цикла ДеБрейна17**

**Алгоритм Райзера18**

**Алгоритм Госперо18**

**Метод с округлениями19**

**Степени двойки19**

**Бинарный поиск19**

**Заполнение19**

**Модификация бинарного поиска20**

**Заключение21**

**Список литературы21**

**Техническое задание**

Быстрые алгоритмы подсчета и поиска двоичных разрядов

**Введение**

Данная тема состоит из 2 подтем: поиска старшего бита и подсчёт единичных битов. В первой подтеме изложенные алгоритмы помогут ускорить работу с битами, а другая больше похожа на развлечение, чем на применение в реальной жизни, хотя есть задача range minimum query (RMQ) в которой это применимо.

Рассматриваемые алгоритмы могут быть применены для чисел из 8, 16, 32, 64 бит. Для начала рассмотрим пару вещей, который понадобятся для понимания изложенных алгоритмов. За скобки оставляем то, что некоторую часть задач можно сделать на языке Ассемблера.

**Представление чисел в компьютере**

Хорошо известно, что числа в компьютерах хранятся в двоичной системе счисления в виде последовательности 0 и 1. Так же могут хранится в формате фиксированной или плавающей запятой.

Целые не отрицательные числа хранятся в формате с фиксированной запятой. На них выделяется 1 регистр (8 бит). Каждому разряду ячейки памяти соответствует всегда один и тот же разряд числа. Таким образом максимальное значение 255 = 1111111, а минимальное 0 = 0000000.

Для чисел со знаком выделяется 2 регистра памяти (16 бит). Отрицательные числа хранятся в дополнительном коде. Если число отрицательное, то в 16 разряд записывается 1, иначе пишется 0. Максимальное хранимое число -32767 = 11111111111111, максимальное число соответственно 32767 = 01111111111111.

Вещественные числа обрабатываются и хранятся в компьютере в формате с плавающей точкой). Они занимают 2,4,8 или 16 байт. Рассмотрим на примере 2 байтового: последний бит занимает знак. 5 байт занимает порядок, остальные – мантисса.

**Действия над битами**

Для получения результата мы будем выполнять некоторые доступные команды для языка C, а именно: побитовое И, побитовое ИЛИ, НЕ, исключающее ИЛИ, побитовый сдвиг на бит влево, побитовый сдвиг на бит вправо.

Побитовое И выполняет логическое умножение. Первый бит переменной равен логическому произведению первого бита числа a и первого бита числа b в случае, когда оба бита чисел a и b равны 1, то результат 1, во всех остальных 0. И так для каждого бита. Таблица истинности:

|  |  |  |
| --- | --- | --- |
| a | b | result |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Побитовое ИЛИ выполняет логическое сложение. Первый бит переменной равен логическому сложению первого бита числа a и первого бита числа b в случае, когда какой-либо бит чисел a и b равен 1, то результат 1, иначе 0. Таблица истинности:

|  |  |  |
| --- | --- | --- |
| a | b | result |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

НЕ является логическое отрицание. Инвертирует значение бита с 0 на 1, а с 1 на 0. Таблица истинности:

|  |  |
| --- | --- |
| a | result |
| 0 | 1 |
| 1 | 0 |

Исключающее ИЛИ является сложением по модулю 2. Если биты двух чисел различны, то в результате пишется 1, если они равны, то 0. Таблица истинности:

|  |  |  |
| --- | --- | --- |
| A | b | result |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Операции побитового сдвига смещает биты на определённое количество. Сдвиг в право сдвигает биты числа вправо, на освободившееся ставит нули. Сдвиг влево сдвигает биты в лево, лишние – отбрасываются. Например, для числа 01011100101010001 результатом побитовый сдвига вправо 00101110010101000, а влево 10111001010100010.

**Подсчёт единичных разрядов - наивный подход**

Этот метод при постановке задачи приходит в голову самый первый. Он заключается в том, что итерируясь по всему числу пока оно не станет равно 0, выполняем сдвиг влево на 1, но до этого делаем логическое И и результат суммируем в специальную переменную. Недостатки способы явно видны: нам нужно проитерироваться по всему числу. Из пользы данного метода, что при его реализации трудно ошибиться и им можно пользоваться для проверки более сложных алгоритмов. Так же он универсальный: работает для любого размера чисел. Скорость работы вполне закономерно возрастает с ростом размера входного параметра.

**Дуальный алгоритм**

Внешне не сильно изменился от предыдущего варианта. Допустим число n, мы используем n=n&(n-1), что убирает у числа n его младшую единицу и считаем количество таких операций пока n не равно 0. В худшем случае (когда все разряды 1) мы всё равно проходим по всему числу, однако в среднем случаи уже проходим меньше, если быть точнее, то проходим столько же раз сколько единиц в числе. Однако по времени изменения стали не на много быстрее.

|  |  |
| --- | --- |
| n | 11101000 |
| n-1 | 11100111 |
| n&n-1 | 11100000 |

**Предпосчёт**

Этот вариант подходит в том случае, когда у нас есть большое количество свободного места. Суть метода в том, что мы заводим таблицы с уже подсчитанным результатом, что очень сильно ускоряет любой подсчёт, нов равной степени требует большое количество памяти и нужно гарантировать, что обращение к памяти не вызовет проблем и является довольно быстрым. Так же нужно ещё учесть, что эти самый таблицы нужно заполнять, что тоже занимает время, что может сыграть злую шутку в случае, когда подсчёт происходит малое количество раз.

Чуточку подробнее: мы определим две таблицы на 256 и 65536 значений, и в них заранее посчитаем ответы для всех возможных 1-байтовых и 2-байтовых величин. А чтобы получить для других чисел будет делать отдельные функции, в которых число будет разбито на байты. Будет существовать 2 варианта разбиения, которые зависят от таблиц, которые мы применяем для этого.

Подытожив, данный алгоритм оказывается выгодным только лишь при соблюдении следующих двух условий: у нас есть лишняя память, нам требуется выполнять расчёт числа единичных битов намного больше раз, чем размер самой таблицы, то есть имеется возможность “отыграть” время, потраченное на предварительное заполнение таблицы каким-то либо совершенно любым способом. Пожалуй, можно также упомянуть очень экзотическое условие, которое на практике практически всегда гарантированно выполнено. Вы должны гарантировать, что обращение к памяти само по себе быстрое и не замедляет работу других функций системы. Дело в том, что обращение к таблице может выбросить из кэша то, что там было изначально и замедлить таким образом какой-то другой участок кода. Данную проблему вряд ли можно найти быстро, однако подобные нетривиальные оптимизации едва ли кому-то понадобятся на практике при реализации обычных программ.

**Разделяй и властвуй**

Суть алгоритма в том, что число мы делим в группы по 2 бита, а затем эти группы объединять, при этом записывая сумму единиц и в итоге получим готовую сумму. Данная стратегия является рекурсивной. Выполнять подзадачи на самом нижнем уровне можно осуществлять параллельно, как и суммирование значений в соседних полях, которое можно выполнить параллельно за фиксированное число шагов на любом этапе. Иллюстрация:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 01 | | 10 | | 01 | | 00 | |
| 0011 | | | | 0001 | | | |
| 00000100 | | | | | | | |

**Умножение и остаток от деления**

Переходим к более сложным алгоритмам подсчёта. Возьмём 8 битовое число и обозначим все биты буквами:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| a | b | c | d | e | f | g | h |

n =

При умножении числа на шестнадцатеричное 0x010101, мы получим число, где каждый байт исходного повторяется 3 раза.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | | abcdefgh |
| + |  | abcdefgh |  |
| + | abcdefgh |  |  |
| = | abcdefgh abcdefgh abcdefgh | | |

Затем применим логическое И к получившемуся результату и маске 0x249249. Предварительно поделим число на группы по 3 бита в каждой, таких групп 8. Данные действия приведут к тому, что в каждом блоке будет гарантированно 2 нуля и бит из исходного числа, при том в каждом блоке будет “уникальный” бит из первого числа. Визуализация данных действий для более чёткого понимания:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n\*0x010101 | abc | def | gha | bcd | efg | hab | cde | fgh |
| 0x249249 | 001 | 001 | 001 | 001 | 001 | 001 | 001 | 001 |
| result & 0x249249 | 00c | 00f | 00a | 00d | 00g | 00b | 00e | 00h |

Как можно заметить при данном методе теряется последовательность бит, одна для нашей задачи она не должна поддерживаться для достижения результата, поэтому игнорируем этот факт.

Теперь мы должны сложить все эти группы и от полученного результата взять остаток от деления 7, т.к. остаток от деления числа A на -1 даёт сумму k-битовых групп числа A, взятую по модулю -1. Или иными словами мы получаем сумму наших групп по модулю 7.

Доказательство этого действия: Разобьём число A (в двоичной записи) на блоки по k бит в каждом (при необходимости можно дополнить самый последний, старший, блок нулями). Обозначим через i-й блок. Теперь запишем значение числа A через сумму этих блоков, помноженных на соответствующую степень двойки:

A= ⋅2^(0⋅k)+ ⋅2^(1⋅k)+…+ ⋅2^((N-1)⋅k),

где N – число блоков.

Теперь рассчитаем A mod (2^(k-1)).

A mod (2^(k-1))= ⋅2^(0⋅k)+ ⋅2^(1⋅k)+…+ ⋅2^((N-1)⋅k)) mod (2^(k-1)) = (+ +…+ ) mod (2^(k-1)).

Всё благодаря тому, что 2^(i⋅k) = 1 (mod (2^k-1)) для любого целого неотрицательного i. (Здесь, правда, важно, что трюк имеет смысл, когда k>1, иначе не совсем понятно, как нам интерпретировать модуль 1). Вот мы и получили сумму блоков по модулю 2^k-1.

Однако надо ещё предусмотреть 2 случая: если мы считаем для 0(единственный случай, когда ответ 0) и считаем для 255(единственный случай когда ответ 8). В этих случаях просто напишем проверку для 0 и 255, что вернёт как результат 0 и 8. Остальное является верным результатом, так же надо не забыть учесть, что если остаток 0, то результат 7.

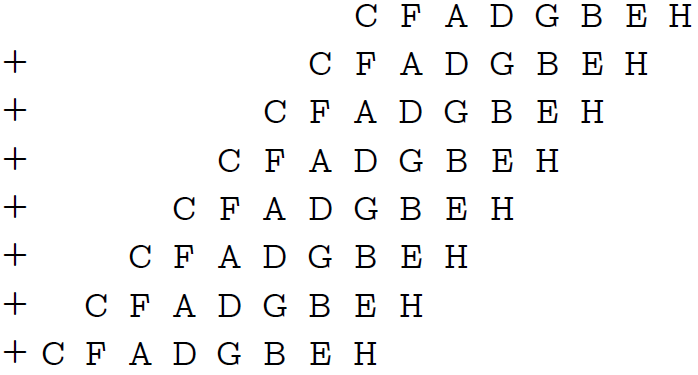
Для остальных случаев можно или выполнить данные определённое количества раз или подобрать разделения на другое по количеству групп и соответственно другие маски.

**Умножение и сдвиг**

Взятие остатка - это деление, а деление очень дорогая операция, поэтому постараемся заменить его на сдвиг. Данный алгоритм является улучшением предыдущего.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 00c | 00f | 00a | 0d | 00g | 00b | 00e | 00h |
| c | f | a | d | g | b | e | h |

Повторяем те же самые действия, но теперь вместо взятия остатка мы умножим на маску 0x249249, что эквивалентно сложение числа самого с собой 8 раз. Биты с 21 по 23 дают нам нужную сумму, но для 255 получим 0, поэтому пропишем это отдельным условием.



**Параллельное суммирование**

Является самым распространённым алгоритмом. Опять рассмотрим для 8 битового числа. Разобьём на 4 поля по 2 бита особенным образом применив маску 0x55 и побитовый сдвиг на 1 вправо, а затем сложим результат. Для лучшего понимания иллюстрация в виде таблицы:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | ab | cd | ef | gh |
| x=n&0x55 | 0b | 0d | 0f | 0h |
| y=(n>>1)&0x55 | 0a | 0c | 0e | 0g |
| x+y | a+b | c+d | e+f | g+h |

Теперь получившиеся суммы объединим в группы по 2 и аналогичным методом, но с другой маской (0x33) сделаем сумму:

|  |  |  |
| --- | --- | --- |
| n | a+b c+d | e+f g+h |
| x=n&0x33 | 0 c+d | 0 g+h |
| y=(n>>2)&0x33 | 0 a+b | 0 e+f |
| x+y | a+b+c+d | e+f+g+h |

Повторяем аналогичную операцию с маской 0x0f и смешением 4:

|  |  |
| --- | --- |
| n | a+b+c+d e+f+g+h |
| x=n&0x0f | 0 e+f+g+h |
| y=(n>>4)&0x0f | 0 a+b+c+d |
| x+y | a+b+c+d+e+f+g+h |

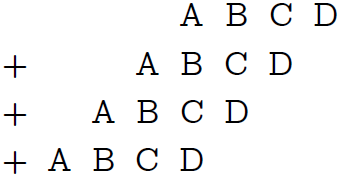
Данная сумма является искомым результатом.

**Комбинированный метод**

Хорошо видно, что наилучшие варианты подсчёта единичных битов – это параллельный метод (с определённой оптимизацией) и метод тиражирования с умножением для подсчёта суммы блоков. Мы можем объединить оба метода, и получим комбинированный алгоритм. Выполняем первые 3 суммирования из параллельного алгоритма.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n= | a | b | c | d |

Затем к нашему числу, которое состоит из 4 байт, применяем алгоритм умножение и сдвиг и получаем нужный ответ.



**Подсчёт ведущих нулей: бинарный поиск**

Суть метода заключается в том, что мы ищем первый ведущий ноль каждый раз уменьшая интервал поиска в 2 раза.

Так же есть вариация с обратным бинарным поиском. Всё происходит аналогично простому, только работающем с другого края. Он требует выполнения тем меньшего количества операций, чем больше ведущих нулей задаваемых чисел и больше сдвигов.

Существует возможность реализовать бинарный поиск с использованием цикла. Такая вариации названа бинарный поиск с использованием цикла.

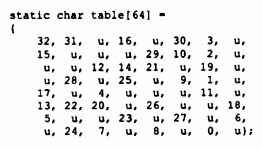
**Двухсторонний подсчёт ведущих нулевых битов**

Очевидно, для подсчёта количества ведущх нулей можно просто циклически выполнять сдвиг влево на 1, подсчитывая количества нулевых битов до тех пор, пока знаковый бит не станет равный 1; можно так же циклически сдвигать слово на одну позицию вправо до тех пор, пока всебиты слова не окажутся нулевыми. Оба алгоритма достаточно компактны и хорошо работают со словами, содержащими малое (ну, соотвественно, или очень большое) количество ведущих нулевых битов.

Этот метод улучшение “Наивного” алгоритма, а именно, если в первом мы просто итерируемся по циклу, то в данной задачи, можно делать одновременно итерацию и с другого конца слова, что не влияет на худший случай, но ускоряет средний.

**Алгоритм Харли**

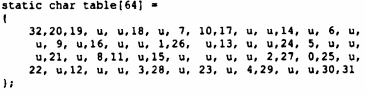
В данном алгоритме в числе выполняется сдвиг вправо, а затем логическое ИЛИ и так несколько раз, а затем умножается на определённую константу (0x06eb14f9), что позволяет нам однозначно идентифицировать количество нулей. После этого мы выполняем сдвиг и выполняем поиск в таблице для преобразования получившегося значения в число нулей. Данная таблица (u –неиспользуемые значения):

: 

**Вариация Горявски алгоритма Харли**

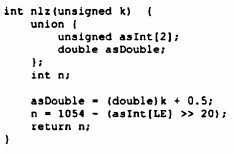
Юлиус Горявски нашёл ряд вариаций алгоритма Харли, в которых размер таблицы снижен ценой нескольких команд либо повышена степень параллельности вычислений, либо эти вариации обладают ещё какими-то желательными свойствами.

Данная вариация уменьшает размер таблицы использующийся раньше, и выполняет сдвиг на 16 и так же логическое отрицание ко всему этому, вместо 5 сдвигов, так же изменена константа. Эта вариация считается самый лучшей из придуманных. Его таблица(u – точно так же неиспользуемые значения):



**С плавающей точкой**

Можно использовать нормализованныечисла с плавающей точкой, например числа с плавающей точкой в IEEE-формате. Идея заключается в преобразовании данного беззнакового целого числа в число с плавающей точкой двойной точности, выделении из него степенной части и вычитании ее из константы.



Демонстрируемый код использует анонимное (безымянное) объединение C++ для перекрытия целого числа и величины с плавающей точкой двойной точности. Переменная LE должнабыть равна 1, если процедура выполняется на машинах, где адрес младшего байта меньше адреса старшего, и 0 в противном случае. Слагаемое 0.5 (или другое малое число) необходимо для корректной работы программы при k = 0. Оценить время выполнения этой программы практически невозможно, так как наразных машинах числа с плавающей точкой обрабатываются по-разному. Например, намногих машинах, кроме регистров для хранения целых чисел, имеются специальные регистры для хранения чисел с плавающей точкой. На таких машинах, в частности, может потребоваться пересылка данных в память для преобразования целого числа в число сплавающей точкой и наоборот.

Кстати, этот код нарушает еще один стандарт ANSI, а именно то, что арифметические действия могут выполняться только с указателями на элементы массива. Основная проблема, тем не менее, связана с первым нарушением стандартат.

Код некорректен в силу способа использования языка программирования С. Эти методы совершенно корректно будут работать при кодировании на машинном языке либо будучи сгенерированными компилятором для конкретного типа компьютера.

В силу этих проблем, использование метода может стоять под вопросом.

**Подсчёт завершающих нулевых битов. Общее решение**

Применив любой из ранее предложенных алгоритмов для вычисления ведущих нулей, то можно свести к использованию этой команды: 32-foo(~x&(x-1)). Так же при применении команды вычисления количества единичных битов тоже можно свести к тому же: foo(~x&(x-1)) или 32-foo(x|-x).

**Бинарный поиск**

Просто применяем уже известный до этого бинарный поиск: уменьшаем интервал в 2 пока не будет найдено нужное значение. Так же можно упомянуть про то, что при малых числах это можно заменить на сдвиг вправо.

**Поиск по бинарному дереву**

В данном случае мы просто проверяем все случаи ответа особым образом. Бинарным деревом же называется потому что мы проверяй случаи, и если она не совпадает проверяем другие, если совпадает, то проверяем другой и так пока не дойдём до конца и концы же вернут нам итоговый ответ.

Явным недостатком всего это огромное количество написанного (иногда трудночитаемого) кода, так же чем больше разрядность числа, тем больше программа, поэтому она теряет универсальность.

**Наивный метод: подсчёт завершающих нулей**

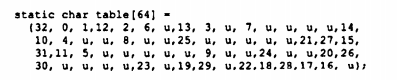
Как и в предыдущий раз суть метода в том, что мы просто проходим по числу и считаем количество. Плюсы и минусы такого алгоритма остались не изменными.

**Алгоритм Годо**

Основан данный алгоритм на 5 числах, которые позволяют определить положения единиц. Используемые числа: 0x0000ffff, 0x00ff00ff, 0x0f0f0f0f, 0x33333333, 0x55555555. Затем результат всех проверок складываем и получаем нужное нам число.

**Алгоритм Сила**

Чем-то напоминает алгоритм Харли, в основном потому что последний основан на данном. Возьмём первый бит числа и умножим его на константу составленному по определённым правилам и заведём таблицу, которая по результату смещённому на 26 вправо, позволит определить число нулей. Таблица(u – неиспользуемые значения)::



**С использованием цикла ДеБрейна**

Существуют такие циклические последовательности на заданном алфавите, которые содержат в качестве подпоследовательности каждую из последовательностей букв алфавита заданной длины ровно один раз.

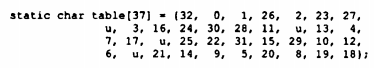
Для нашей задачи воспользуемся алфавитом {0,1}, а для работы с 32-битовыми словами интересует цикл, который содержит все 32 последовательности 00000, 00001, …, 11111. Если имеется такой цикл, который начинается по крайне мере с четырёх нулей, можно вычислить количество нулей, сначала преобразуем входное число в слово, содержащее единственный единичный бит в позиции младшего единичного бита, как в алгоритме Сила. Затем с использованием умножения выбираем 5-битовое поле цикла деБрейна, которое для каждого множителя будет иметь свое единственное значение, и по нему уже найдём количество нулей.

В целом это является усовершенствованием алгоритма Сила. Разница лишь в том, что меняется константа, что приводит к дополнительной проверки на 0, но также мы уменьшаем размер таблицы. Сама таблица:



**Алгоритм Райзера**

Ещё одна модификация алгоритма Сила. В данном алгоритме в отличии от предыдущей модификации уже есть не используемые поля в таблице, выбор ячейки уже происходит путём взятия остатка от деления на 37. Таблица (u – так же неиспользуемые значения):



**Алгоритм Госперо**

Функции выше можно использовать в алгоритме Госперо, который несмотря на свой вид является довольно эффективным.

Пусть имеется последовательность определяемая правилом . Если область значений функции f конечна, то эта последовательность – периодическая. Она состоит из некоторой ведущей последовательности, за которой идёт периодически повторяющаяся последовательность. Обнаружение цикла так ж используется в тестировании генераторов случайных чисел.

Алгоритм основан на идеи поиска циклов в последовательности. Главным достоинством этого алгоритма является то, что функцию мы вызываем 1 раз, при этом хорошая скорость работы и экономия используемой памяти

**Поиск старшего единичного бита: наивный метод**

По аналогии с предыдущими наивными методами, мы просто проходим в киле по числу, пока не найдём нужны бит.

**Метод с округлениями**

Так как номер старшего бита – показатель степени двойки, то его можно найти с помощью округления вниз логарифма. Но тут могут возникнуть проблемы с библиотекой math.h, т.к. она возвращает не всегда точное значение, поэтому придётся прибавлять константу.

**Степени двойки**

Является своего рода комбинации 2 предыдущих методов. Переберём все степени двойки, и выберем из них максимальную, которая не превосходит числа, таким образом получим нужное нам число.

**Бинарный поиск**

Запускаем ничем не изменённый бинарный поиск и ищем пока число не будет равно 1. Одно из самых быстро приходящих в голову решений.

Так же существует оптимизация, которая заменяет сдвиги и исключающее ИЛИ на сложение и логическое И добавив константный массив масок: 0x0000FFFF, 0x000000FF, 0x0000000F, 0x00000003, 0x00000001. Данная оптимизация называется Бинарный поиск с маской.

**Заполнение**

Основная идея в том, что мы заполняем все биты после нужной нам 1, единицами каждый раз в 2 раза большим числом, после чего делаем x – (x>>1), что выдаёт нужный ответ. Иллюстрация на примере первого шага:

|  |  |
| --- | --- |
| x | 00000100000 |
| x>>1 | 00000010000 |
| x|x>>1 | 00000110000 |

**Модификация бинарного поиска**

Алгоритм сохраняет основную идею бинарного поиска, одна в качестве сравнения выполняем x >= 1 << 16 и в случае 1 сдвигаем на 16 и так далее 8, 4, 2, 1. И получаем нужный ответ.

**Заключение**

При просмотре результатов измерений при малом количестве использований данный алгоритмы мало чем отличаются от самого простого наивного поиска, что разница не чувствуется в общем, одна некоторая часть поддтем даже используется в реальных задачах.

К сожалению, мы никогда не можем точно сказать, который из алгоритмов окажется лучше в том или ином случае. Под каждую задачу нужно затачивать конкретный метод, а универсального быстрого алгоритма, к сожалению, не существует.

Также некоторые алгоритмы можно модифицировать под конкретные случаи, что даже условный наивный алгоритм, будет выгоднее использовать, чем какое-либо другой.

**Список литературы**

1. Генри С. Уоррен мл. "Алгоритмические трюки для программистов", 2-е издание
2. “Обстоятельно о подсчёте единичных битов” <https://m.habr.com/ru/post/276957/>
3. “Алгоритмы поиска старшего бита” <https://m.habr.com/ru/post/93172/>
4. “Нахождение номера старшего бита числа” <http://cppalgo.blogspot.com/2012/06/blog-post.html?m=1>
5. Э. Рейнгольд, Ю. Нивергельт, Н. Део Комбинаторные алгоритмы. Теория и практика.