Математика.

Рассмотрим два алгоритма, которые работают за logn и вычисляют a в степени n.

Назовем функцию exp и она будет принимать a и n.

Если n нечетный, то вызовем ф-цию рекурсивно и вместо n передадим n-1 и умножим весь ответ на a.

Если n четный, то вызовем ф-цию рекурсивно и вместо n передадим n/2 и возведем весь ответ в квадрат.

Если n = 0, то ответ 1.

Рассмотрим число 13, а именно в бинарном виде 1101, если в конце 1, то оно нечетное => используем верхнюю ф-цию, чтобы спустить на единицу. Однако потом мы спокойно используем 2 версию ф-ции, тк 12 = 1100 четная. И после этого мы можем два раза спустить нашу степень 6 = 110 и 3 = 11. Можно заметить, что верхняя ф-ция вызывается по кол единиц в бинарном представлении нашего числа. 1101.

Соответственно сложность равна O(logn).

Второй алгоритм. Для след алгоритма нам понадобиться предпочитать некоторые значения. А именно a, a^2, a^4, a^16 … a^2^k, k = 0…infinity.

Возьмем число n = 13 = 1101. Мы можем a^13 разложить как a^8 \* a^4 \* a = a^13. 1101 = 8+4+1 все степени двойки. Следовательно, нам нужно вычислить a в степени 2 в степени 3, 2 и 0. Как понять какие именно степени нам нужно вычислять. 1101, пронумеруем каждый бит 3210. Видно, что именно тот бит, на котором стоит 1, именно та степень нам нужна будет.

Сложность O(logn).

…код

Бинарный поиск.

У нас есть отсорт массив arr = [1, 3, 4, 6, 7, 8, 12], мы хотим проверить, есть ли число a в нашем массиве. Если мы будем просто в тупую проходиться по массиву, то сложность будет O(n), а мы хотим O(logn).

Например a = 8. Возьмем две переменные l = 0, r = arr.size()-1. Это индексы первого и последнего элемента, которые мы рассматриваем. На каждом шагу вместо того, чтобы проходится от начала до конца, мы будем брать число m = (l+r)/2. Это середина. Те 0 + (6 - 0) / 2 = 3. Сразу рассмотрим эл под индексом 3, те число 6 и мы знаем, что до нее нет 8, поэтому отбрасываем часть массива.

Теперь мы знаешь, что все индексы на промежутке [l, m], не удов нам. Это значит, что l мы перезапишем как l = m + 1. Это значит, что наш диапазон начинается с 4 и заканчивается концом массива.

И далее делаем все то же самое. Если нам не нужна правая часть массива, мы можем r = m – 1. В конце у нас будет l = 5 и r = 5, m = 5 + (5 - 5) / 2 = 5. А на 5 позиции у нас 8, что мы и искали.

Если эл нет в массиве, l = m + 1 = 6, r = 5. Тк левая граница стала больше, а должна была быть меньше, то что то здесь не так. Значит эл нет. Он работает за O(logn)

…код

Алгоритм Эвклида, НОД и НОК

НОД.

Нужен для того, чтобы найти наибольший общий делитель у двух чисел. Обозначается как gcd(a, b).

Например, gcd(12, 28) будет равен 4, тк 12 делится на 1, 2, 3, 4, 6, 12, а наиб общий делитель у 28 будет равен 4.

Как вычислять. Назовем ф-цию gcd и она принимает два числа a и b. Если b = 0, return a. Иначе gcd(b, a mod b).

Например gcd(21, 15) = gcd(15, a % b => 6) = gcd(6, 3) = gcd(3, 0) = 3.

Заметим, что второй аргумент при каждой итерации строго убывает => поскольку он неотриц, то алгоритм Евклида всегда завещается.

Для док-ва нам необходимо показать, что gcd(a, b) = gcd(b, a mod b), при любых a >= 0 и b > 0. Для док-ва корректности покажем, что величина стоящая в левой части равенства, делится на величину, которая стоит в правой части и наоборот.

Обозначим через d = gcd(a, b) это будет значит, что d/a и d/b. Далее разложим остаток от деления a на b через их частное.

a mod b = a – b \* a/b- целую часть. => d / (a mod b), вспоминая, что d/b, мы можем получить след систему:

{ a/b и d/(a mod b)} => d/gcd(b, (a mod b)). Допустим для чисел p q r верно, что p/q и p/r => p/gcd(q, r).

gcd(a, b) / gcd(b, a mod b). gcd(a, b) = gcd(b, a mod b).

Сложность. Сложность вычисляется с помощью теорему Ламе, которая показывает связб между алгоритмом Евклида и послед Фибоначчи.

O(log min(a, b))

НОК.

Lcm(a, b) = (a\*b)/gcd(a, b) => a\*b = lcm(a, b) \* gcd(a ,b).

… код

Решето Эратосфена

Этот алгоритм позволяет найти все простые числа в отрезке [1, n]. Запишем все числа от 1 до n

1, 2, 3 … n и будем вычеркивать которые делятся на 2, кроме самого 2, на 3, кроме 3, на 5, кроме 5 итд. Тогда мы получим простые числа от 1 до n.

Пример:

[1, 20]: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 => все эл кроме 1 простые числа.

Дело в том, что когда мы имеем данный массив чисел, мы идем сначала до конца и смотрим если конкретное число не зачеркнуто, то оно простое.

Мы будем пропускать все составные числа, все числа, которые делятся на простые числа меньше чем они.

Все составные числа будут удалены из нашего массива.

Сложность O(nloglogn). Док-во через суммы и интегралы.

Доп памяти O(n).

Посмотрим чему равно loglogn, если n = 10^9

10^9 ~ 2^31

Log(log(10^9)) = log(31) ~ 5 очень приближенно к n. Все Log с основанием 2.

Самый большой недостаток алгоритма заключается в том, что он прыгает по эл нашего массива и поэтому получается так, что он выходит за пределы кэша. Также занимает много памяти.

Оптимизация Эратосфена.

Для того, чтобы найти все простые числа [1, n], достаточно выполнить просеивание только теми простыми числами, которые не превосходят корня нашего n.

В данном случае мы могли пройтись до sqrt(20) = 4.

Зачем хранить четные числа, если мы знаем, что четное только одно простое число у нас есть. И поэотму мы можем в двое сократить объем памяти, а также уменьшить число операций примерно в 2 раза.

Также мы можем сделать операцию, которая оперируется конкретно битами, те мы будем хранить массив не bool типа а например 32 битное integer, 32 битное целое число и проходиться не просто по этому числу, а по битам этого числа. Потребляемая память уменьшится примерно в 8 раз. Не n байт памяти, а n/8 байт.

Однако это замедляет наш алгоритм из-за увеличения ариф. выражений.

…код

Умножение матриц

Матрица размером nxm записывается в виде прямоугольной таблицы, в которой есть n строк и m элементов.

Есть две матрицы Anxm и Bkxr. Умножение матриц определено только в том случае, если m = k. Например A2x3 и B3x5. При этом C2x5 будет равна.

Для того, чтобы узнать эл c11, нам нужно взять строку из первой матрицу, столбец из второй матрицы и поэлементно умножать = a11 \* b11 + a12 \* b21 + a13 \* b31 + … + a1m \* bm1. Итд

Сложность. O(n \* k \* m) времени. O(n \* k) доп памяти.

Функция Эйлера

Она равна кол позитивных чисел, с которыми взаимно проста наша n. [1; n]

Например хотим вычислить для 9. Следовательно мы должны проверить со сколькими числами в диапазоне от 1 до 9, 9 явл взаимно простой.

Два числа a и b явл взаимно простыми, если их НОД = 1.

Фи(9) = 6.

Модулярная арифметика

Одно из самых популярных сфер, где используется модулярная арифметика явл криптография. Сразу приводя пример, мы можем взять например 24 часовые часы и 12 часовые часы. И если смотреть на часы, то для нас 6 и 18 это абсолютно одни и те же числа. В данном случае это значит, что наши часы работают по остатку mod 12.

Говоря немного формально, она записывается след образом: a mod b. Например, 5 mod 3 = 2 или 7 + 3 mod 4 = 2. Ответ может быть только положительным числом. Но если у нас берется модуль n или m, то ответ может получится [0, m - 1]. Например -7 mod 4 = 1. Слева mod не может быть деление !!!

Например: 7 + 3 \* 9 mod 4. Мы можем взять каждый эл, отдельно друг от друга взять модуль от него. Типа ((7 % 4) + ((3 % 4)(9 % 4)) % 4) % 4 = (3 + (3 \* 1)) = 2. Всегда лучше делать так. На каждом шагу.

9/3 mod 4 = (9 % 4) / (3 % 4) = 1/3. Операция деления не поддерживается.

Однако мы можем сделать так. a/b = c. Вместо того, чтобы делить на b, мы будем a \* b^-1 и получать c. Чтобы сделать это, нам поможет теорема Эйлера, которая также известна как малая теорема Ферма. И записывается след образом:

a^фи(n) тождественно равно 1 mod n, только в том случае, если gcd(a, n) = 1.

Сразу перейдем к случаю a^фи(n) \* a^-1 тождественно равно 1 \* a^-1 mod n, при этом слева мы получим a^фи(n)-1 тождественно равно a^1 mod n. Если n – простое число, то фи(n) = n – 1, а след a^фи(n) – 1 = a ^ n-2

Пример:

5 \* a = 2 mod 7

И мы хотим найти a, при этом

a = 2/5 mod 7

a = 2 \* 5^-1 mod 7

a = 2 \* 5^(7-2) mod 7

И в итоге получим a = 2 \* 3125 mod 7. a = 6 mod 7

Проверка 5 \* 6 = 30 mod 7 = 2. Все правильно.