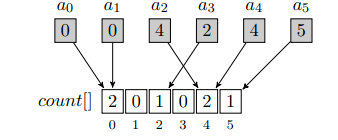
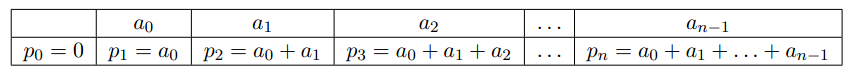
Уроки Codility:

1. **Подсчет элементов.**

Идея в том, чтобы создать дополнительный массив. И в нем считать только количество повторяющихся чисел в соответствующем индексе «счетного массива».  


1. **Префиксные суммы.**

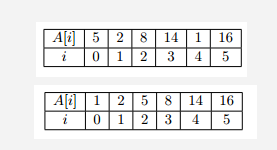
Идея в том, чтобы хранить в текущем элементе сумму всех предыдущих элементов.



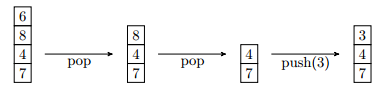
Тем самым мы запоминаем префиксные суммы, чтобы выдавать ответы на «интервальные запросы» за константное время.

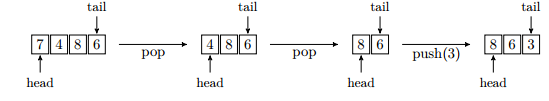
1. **Сортировка.**

Идея в том, чтобы предварительно отсортировать массив, тем самым группируя схожие элементы вместе.



1. **Стеки и очереди.**

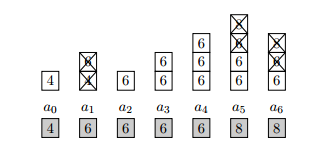
Стеки хорошо подходят для рекурсивных алгоритмов.  
  
Очереди



1. **Лидер.**

Лидер в последовательности – это элемент, который встречается больше чем n/2 раз.

  
Лидер может быть найден за линейное время используя стек.



Идея в том, что мы вставляем каждый элемент в стек. Если элемент не совпадает с предыдущим, то мы исключаем их из стека. Если совпадает, значит оставляем. После этого в стеке останется только потенциальный лидер. Для того чтобы проверить, что потенциальный лидер действительно является лидером, нужно пробежать по всему массиву и подсчитать появления данного элемента.

1. **Проблема максимальной выборки.**

В данном массиве серым цветом показана выборка с максимальной суммой. Точнее, мы должны найти индекс начала и конца данной выборки.



Сумма максимальной выборки находится за линейное время.  
Идея в том, что бы бежать по массиву и сохранять текущую сумму выборки. Если она меньше нуля, то сбрасываем ее значение в 0. Также постоянно обновляем максимальную сумму исходя из текущего значения суммы выборки.

1. **Простые числа.**

Серым цветом обозначены простые числа.



Подсчитать делители числа можно за время sqrt(N):

int i = 2;

while (i \* i <= N)

{

if (N % i == 0)

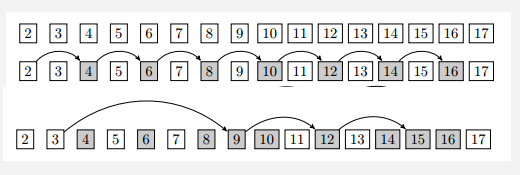
// I - это делитель для числа N

i += 1;

}

1. **Решето Эратосфена.**

Решето Эратосфена используется для нахождения простых чисел в диапазоне от 2 до n.



vector<int> Eratosphen(int N)

{

vector<bool> marked(N + 1, true);

for (int i = 2; i \* i <= N; i++)

{

if (marked[i])

{

for (int k = i \* i; k <= N; k+=i)

{

marked[k] = false;

}

}

}

vector<int> primes;

for (int i = 2; i <= N; i++)

{

if (marked[i])

primes.push\_back(i);

}

return primes;

}

Мы последовательно убираем все множители каждого числа от 2 до sqrt(N). Сложность такого решения = O(N\*log log N).

**Факторизация.** Это процесс разложения числа на простые числа.

****

Используя решето Эратосфена мы находим у каждого числа только наименьший простой делитель. Дальше для каждого числа i делители будут a[i] и факторизация числа i/a[i]. Сложность такого решения O(log i).

1. **Евклидовый алгоритм.**

Данный алгоритм определяет наибольший общий делитель.

int gcd(int a, int b)

{

int k = a % b;

if (k == 0)

return b;

return gcd(b, k);

}

**Наименьший общий множитель.** Это наименьшее число, которое делится на a и b.

Least Common Multiple. LCM = (a\*b) / gcd(a, b).

1. **Числа Фибоначчи.**

Каждое число является суммой предыдущих двух.



std::vector<int> SolveFibNumbers(int n)

{

std::vector<int> fib;

if (n>47)

{

fib.push\_back(-1);//-1 при переполнении int32 на n>=48

return fib;

}

if (n<1)

return fib;

fib.push\_back(0);

if (n<2)

return fib;

fib.push\_back(1);

for(int i=0; i<n-2; ++i)

{

fib.push\_back( \*(fib.end() - 1) + \*(fib.end() - 2) );

}

return fib;

}

Это решение за O(N). Можно решить за O(log N). Но это не тривиальный алгоритм.

1. **Двоичный поиск.**

Алгоритм способен дать на линейном массиве скорость поиска любого элемента за логарифмическое время, вместо линейного. Ограничение для данного алгоритма: массив должен быть отсортирован.



Чтобы найти число 11 нужно сначала проверить средний элемент (8). Дальше если он меньше искомого, идем вправо, иначе идем влево. Идем вправо. Далее сравниваем с 12. Идем влево. Сравниваем с 10. Идем вправо. Нашли 11.

1. **Метод Гусеницы.**

Метод решения алгоритмических задач, в которых требуется двигаться по массиву и запоминать одновременно несколько элементов сзади. Причем наша «гусеница» двигается вся целиком.

1. **«Жадные» алгоритмы.**

Данный метод как правило повышает среднее быстродействие алгоритма. Жадность заключается в том, что мы на каждом локальном шаге выбираем на наш взгляд оптимальный следующий шаг. Этот выбор может не всегда оказаться правильным. Это зависит от критерия оценки.