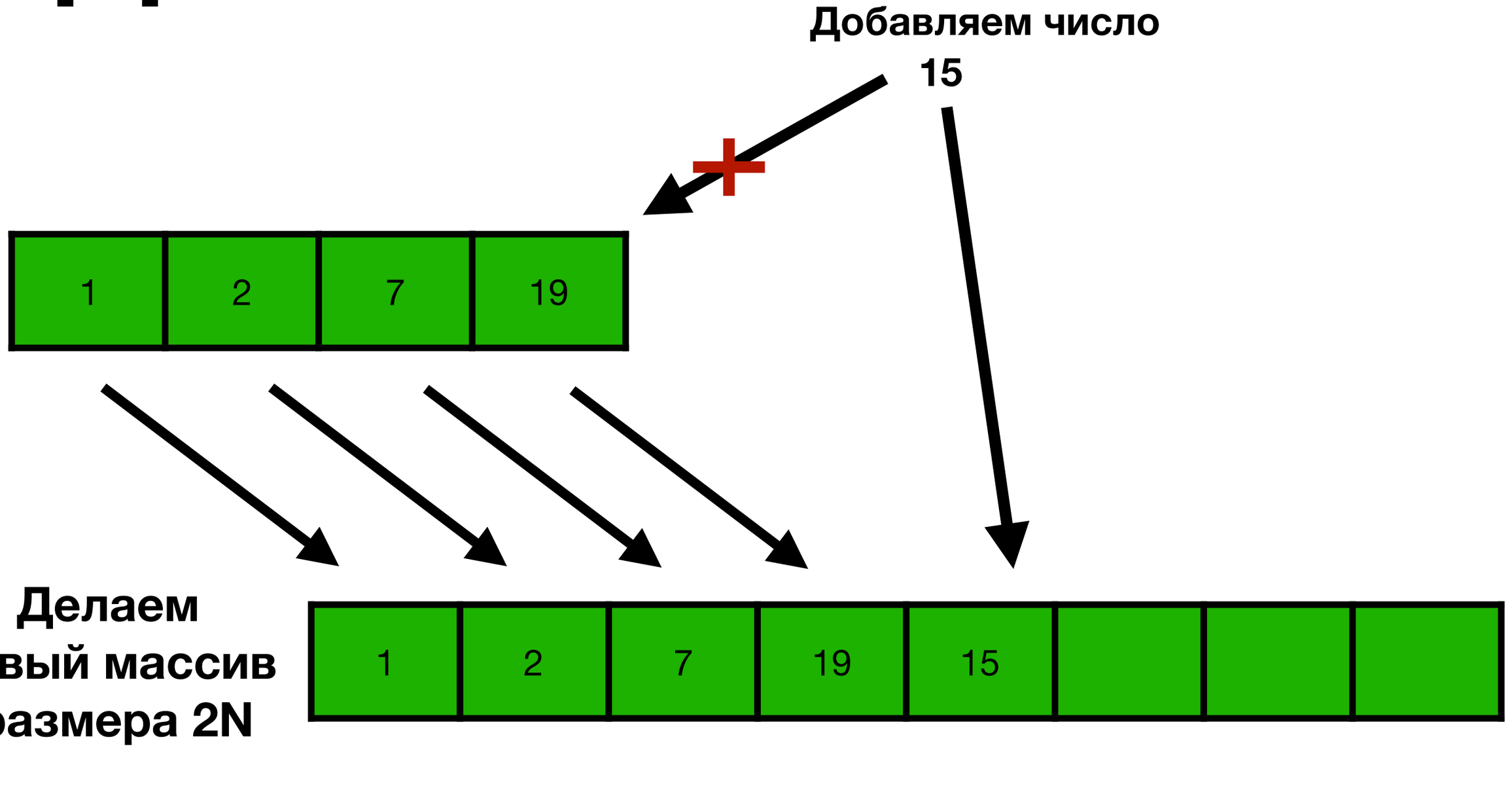
Амортизационный анализ

Амортизационный анализ (англ. *amortized analysis*) — метод подсчёта времени, требуемого для выполнения последовательности операций над структурой данных. При этом время усредняется по всем выполняемым операциям, и анализируется средняя производительность операций в худшем случае.

Динамический массив



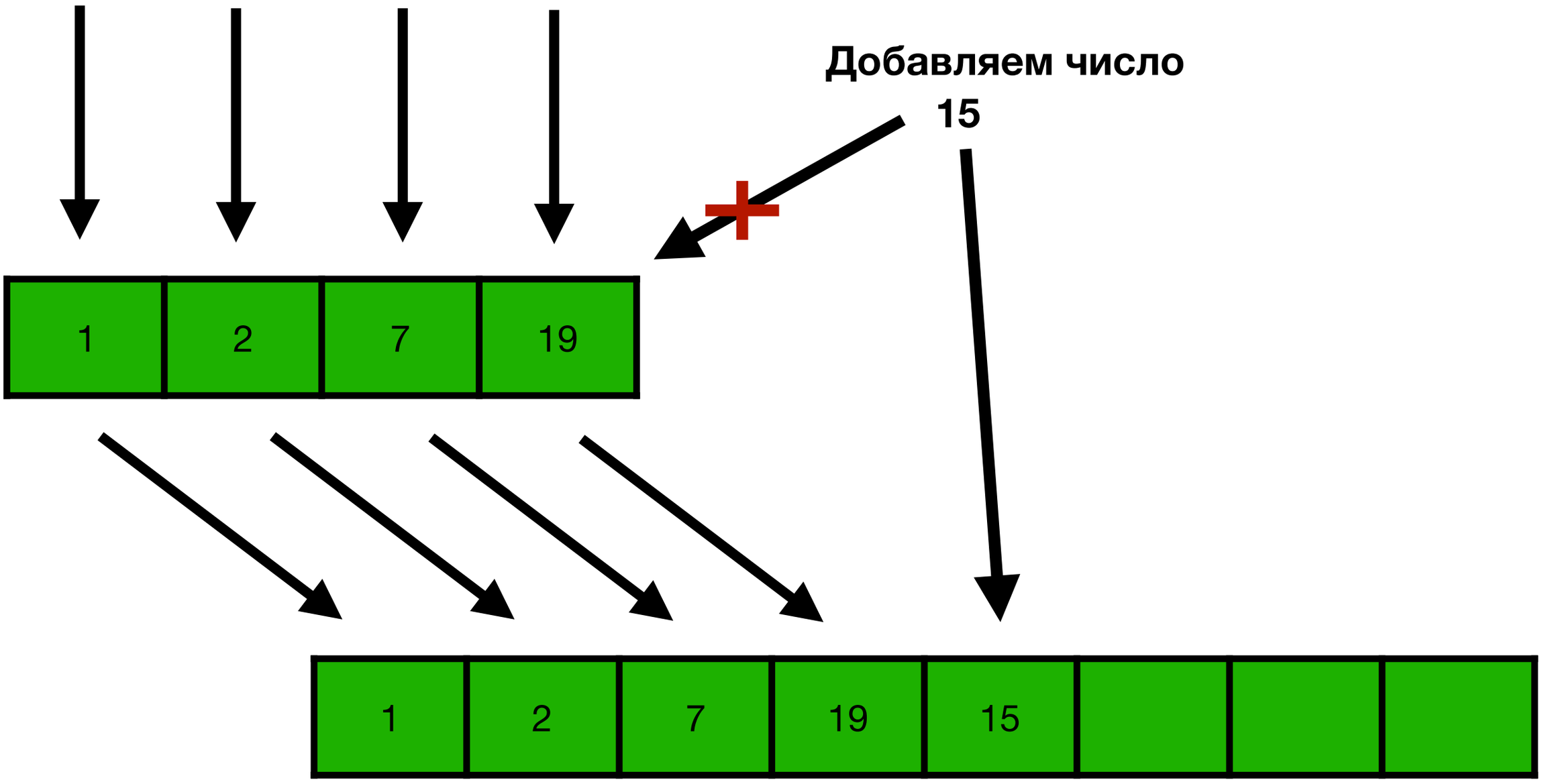
Надо скопировать N элементов - работает за O(N)

Сколько будет стоить добавить N элементов в массив в худшем случае?

Казалось бы, O(N) * N = O(N^2).

Но это неверно

Но прежде чем добавить N+1 число, нам надо было добавить N элементов, они добавлялись за O(1)



Надо скопировать N элементов - работает за O(N)

Подсчитаем, сколько в среднем на каждый элемент

$$\frac{O(N) + O(1) * N}{N+1} \approx \frac{O(N) + O(1) * N}{N} = O(1) + O(1) = O(1)$$

При этом мы не можем попасть в худший случай O(N), не пройдя случаи, когда мы добавляли за O(1).

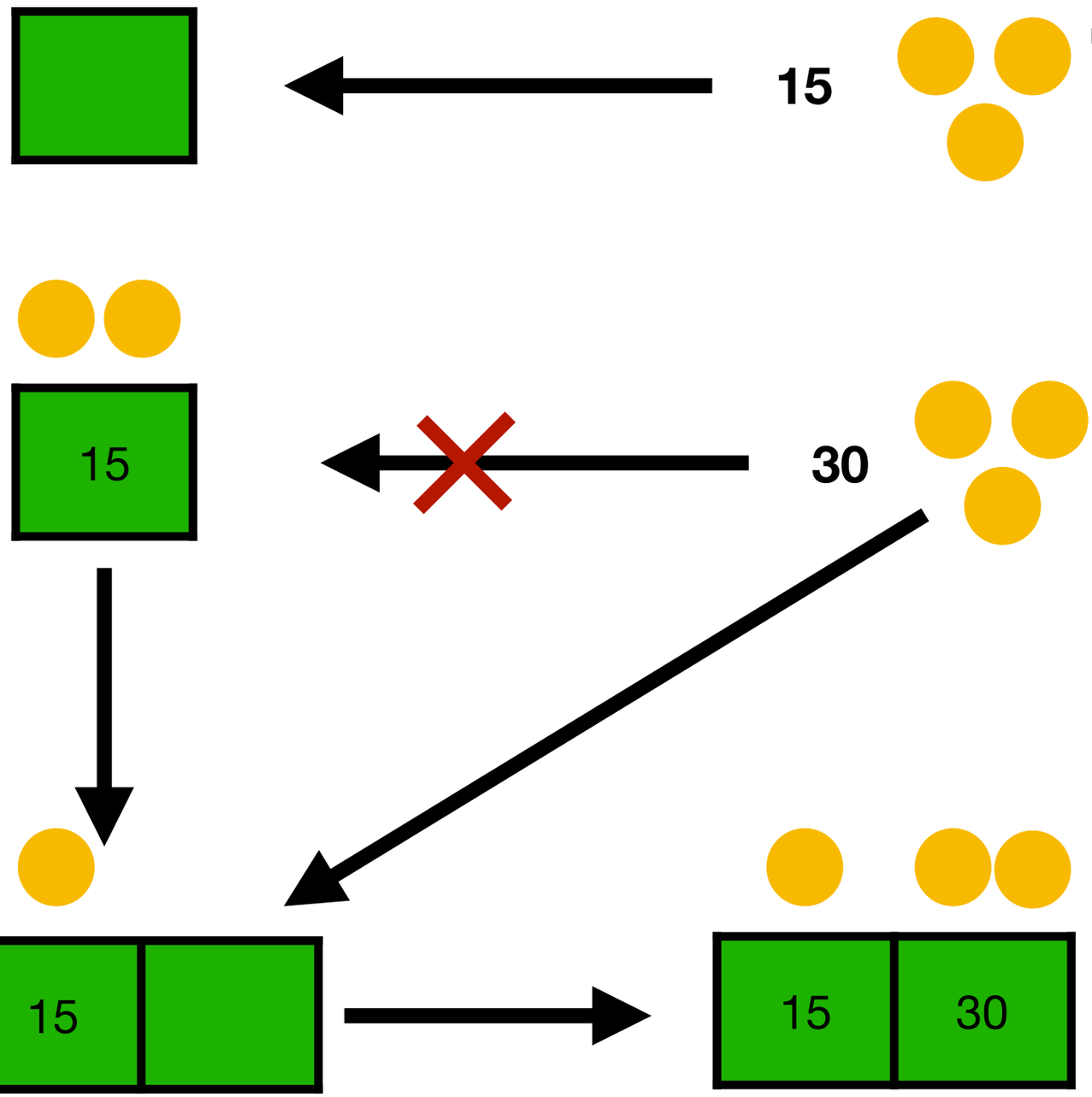
Потому добавить N элементов в массив будет стоить O(1) на каждый элемент или O(N) в целом.

Метод предоплаты (бухгалтерского учета)

Пусть каждая операция стоит сколько-то момент (платим жадному гному)
Мы не хотим в какой-то момент остаться без возможности заплатить.
Потому в какие-то моменты времени мы откладываем монеты, которые заплатим в дальнейшем.

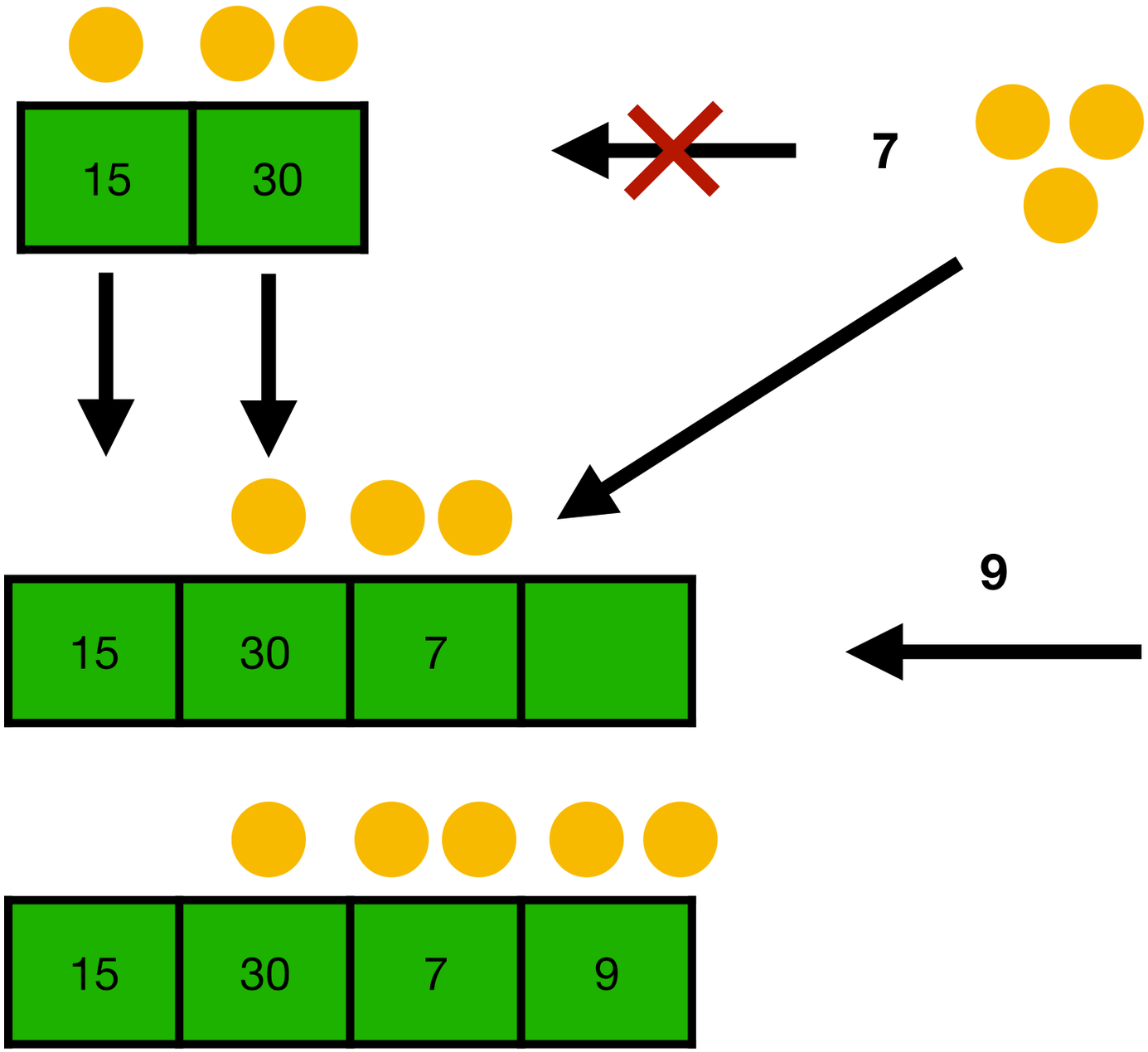
Динамический массив

Сначала имеем массив размера 1, с 1 пустым местом



Добавление будет стоить 1 монету. Будем при ПЕРВОМ добавлении элемента сразу брать три монеты, из которых одна потратится

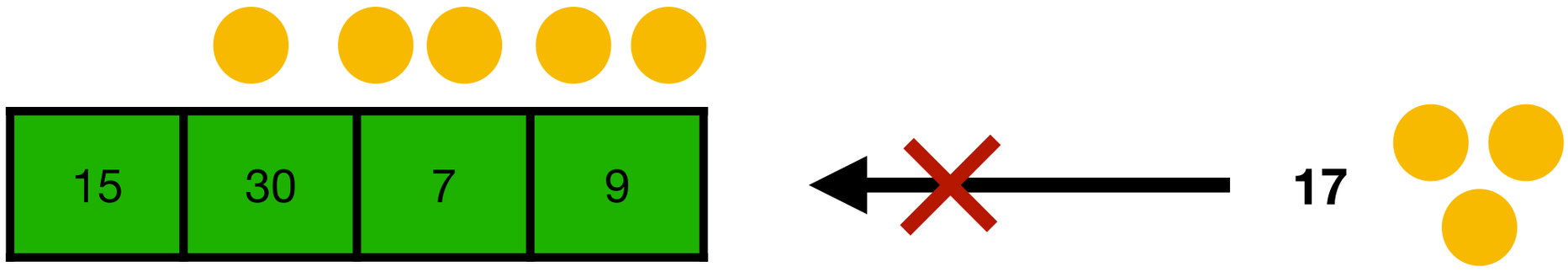
Динамический массив



Добавление будет стоить 1 монету. Будем при ПЕРВОМ добавлении элемента сразу брать три монеты, из которых одна потратится

Динамический массив

Сначала имеем массив размера 1, с 1 пустым местом

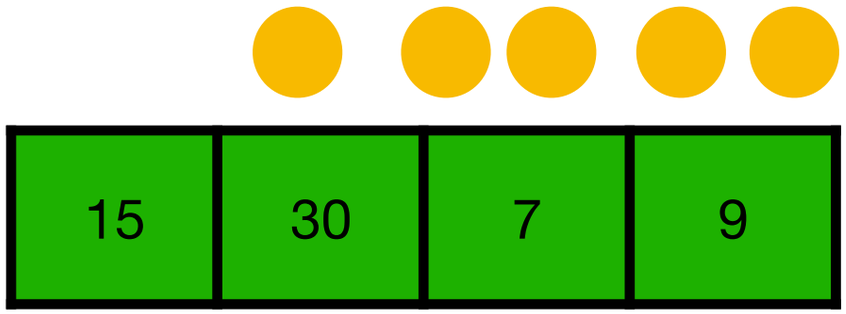


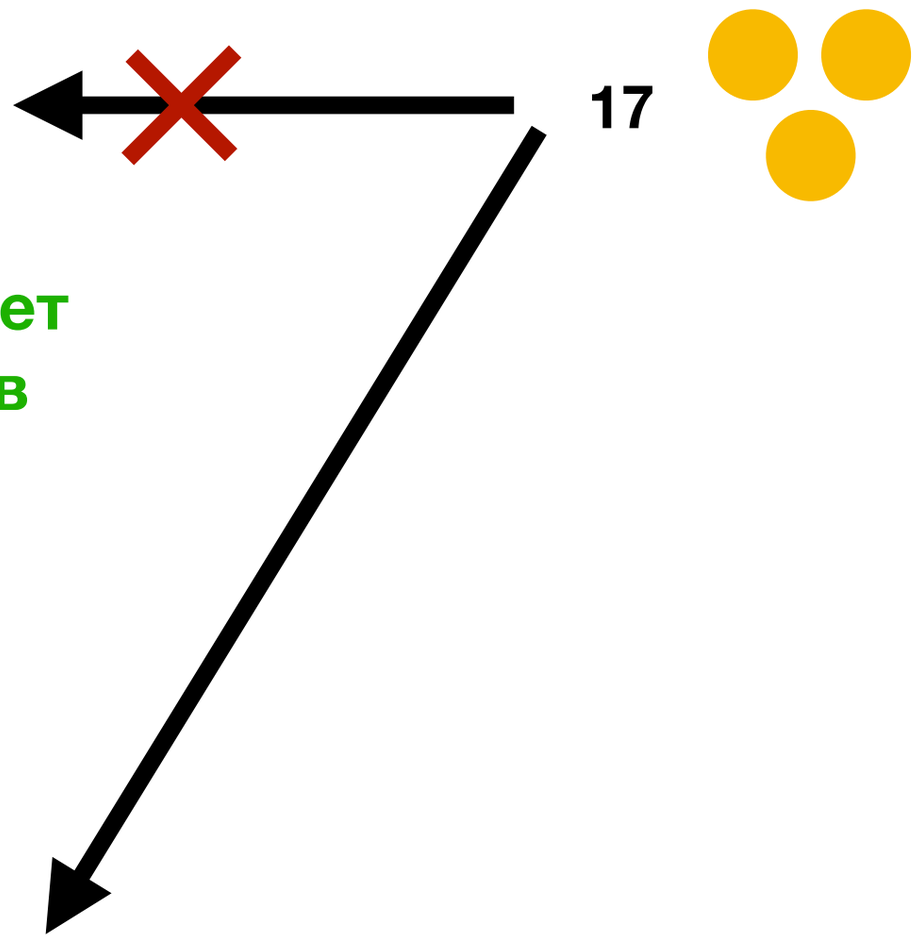
У 15 нет монеток


ПРОЛЕТАРИИ ВСЕХ
СТРАН, СОЕДИНЯЙТЕСЬ!

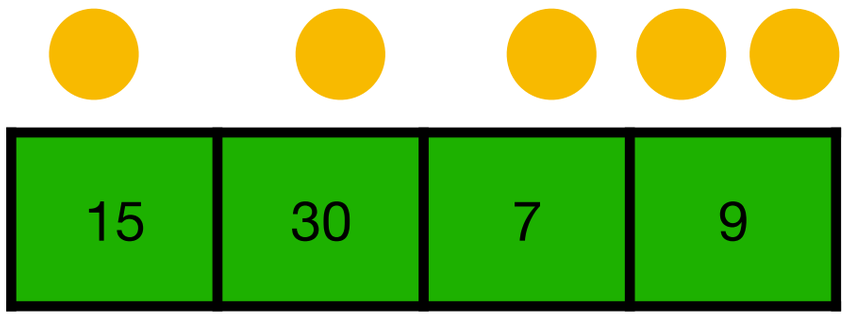
Динамический массив

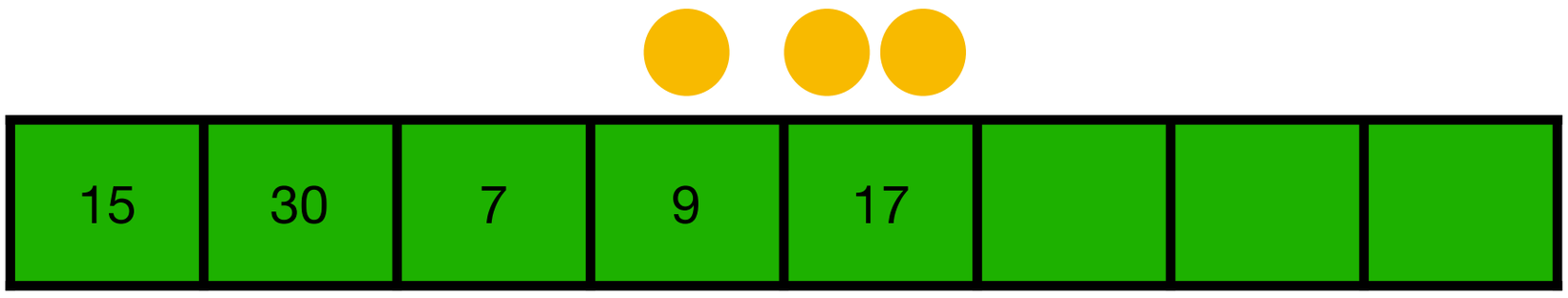
Сначала имеем массив размера 1, с 1 пустым местом





У 15 нет монеток, но с ним может поделиться один из элементов второй части массива



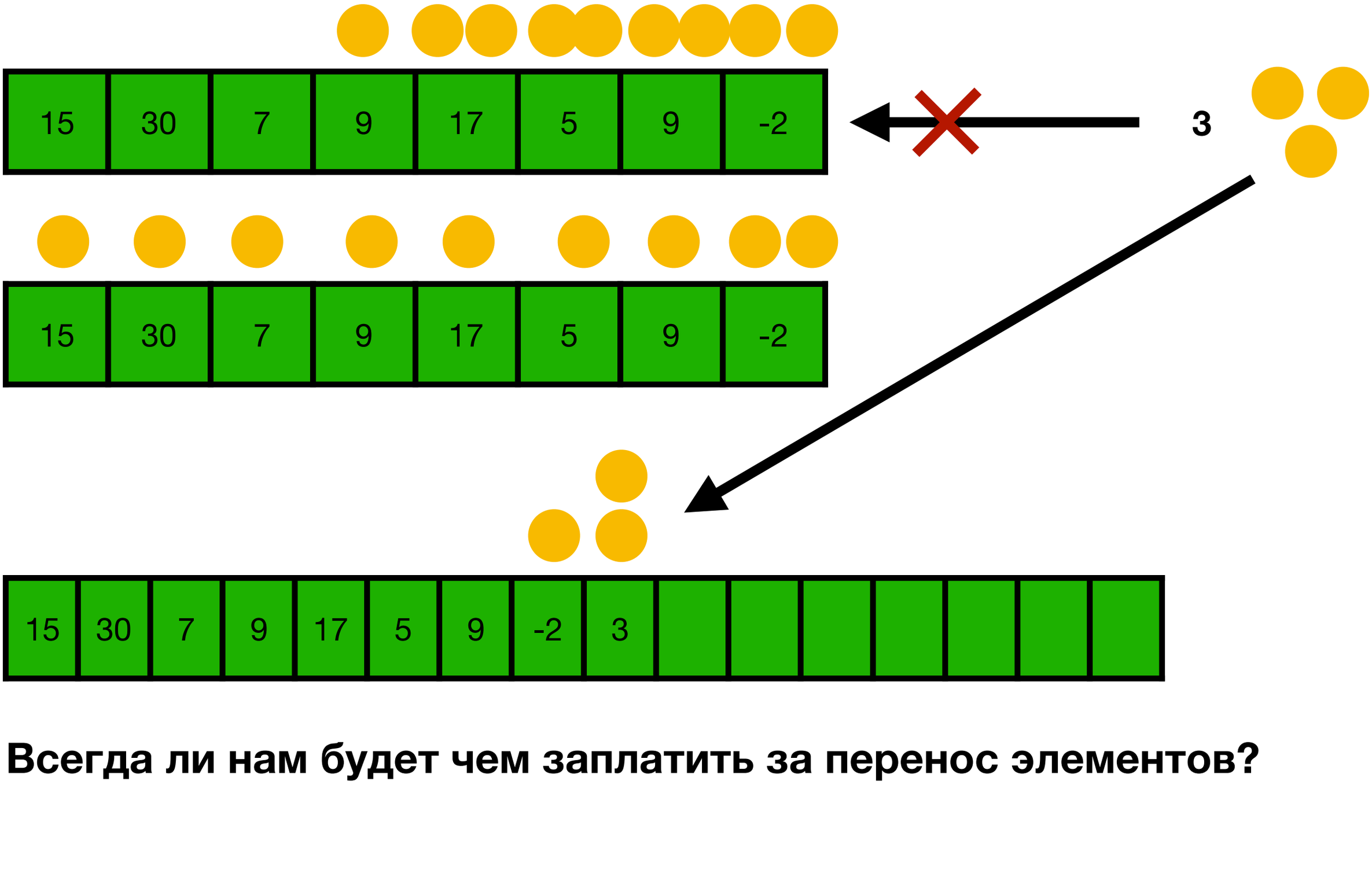


Динамический массив

Сначала имеем массив размера 1, с 1 пустым местом

Динамический массив

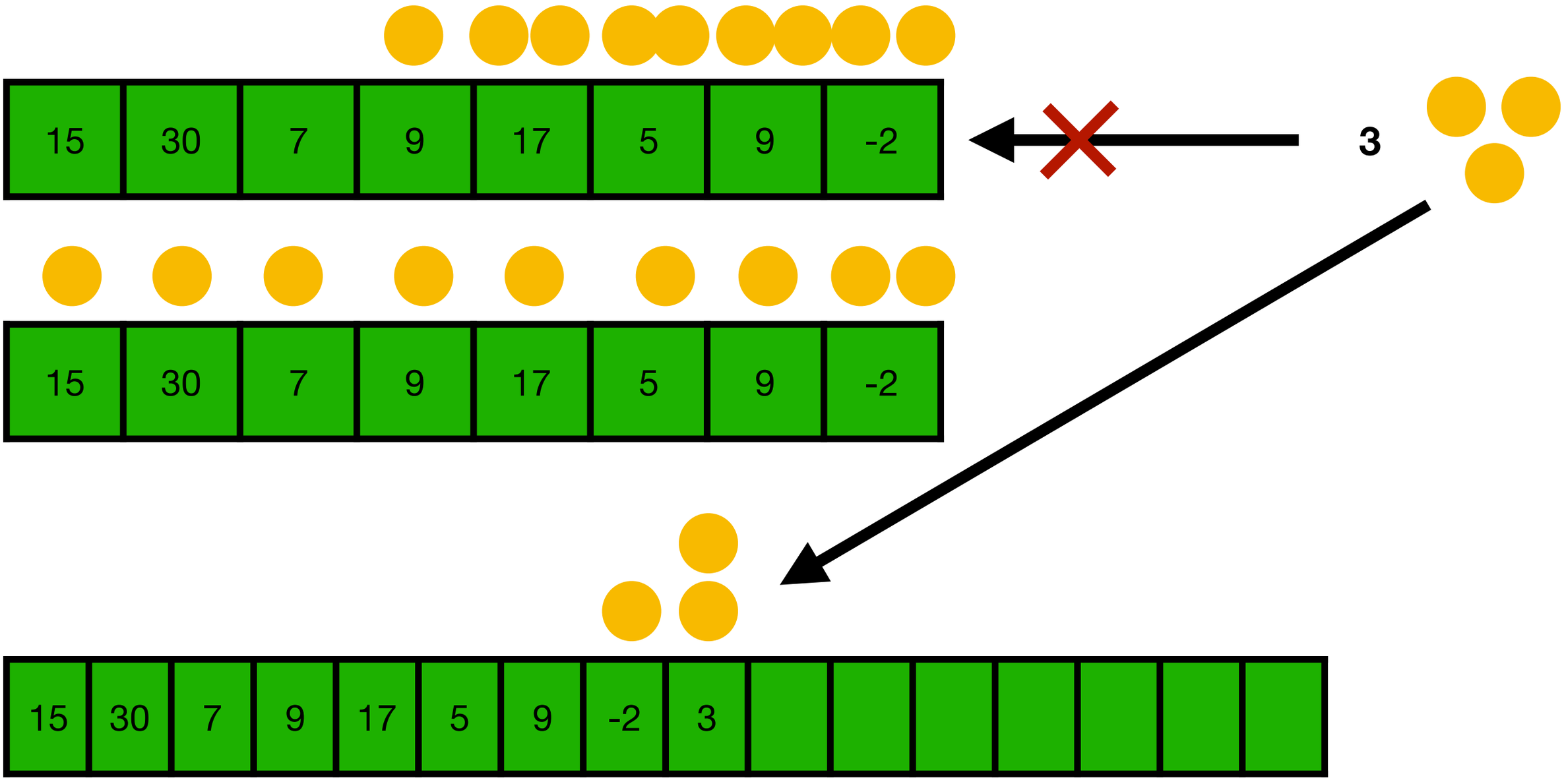
Сначала имеем массив размера 1, с 1 пустым местом



Всегда ли нам будет чем заплатить за перенос элементов?

Динамический массив

Сначала имеем массив размера 1, с 1 пустым местом



Всегда, так как у нас есть не больше N/2 без монеток и не меньше N/2 элементов с двумя монетками

Динамический массив

Таким образом, если теперь мы переведем монетки в элементарные операции, то получается, что если мы оцениваем сложность каждого из N добавлений в динамический массив как 3 O(1) операции, то наша оценка является правильной верхней оценкой.

Тогда суммарно N последовательных операций добавлен займут O(3 * N) = O(N) времени.

Говорим, что добавление в динамический массив занимает амортизационно O(1)

Подсчет префикс-суфикс функции

```
p = [0] * N;
p[0] = 0;
j = 0;

for i in range(1, N) {
    while j >= 0 and s[i] != s[j] {
        j = p[j - 1];
    }
    if s[i] == s[j] {
        j += 1;
    }
    p[i] = j;
}
```

Если считать в лоб:

1) **внешний цикл крутится N раз**
2) **внутренний крутится пока j != 0**
3) **j не может быть больше N (так как это индекс)**
4) **j на каждом шаге цикла while уменьшается минимум на 1**
5) **Значит, while выполняется максимально N раз**
6) **N * N = N^ 2**
7) **Значит, алгоритм работает за O(N^2)**

НЕВЕРНО

Подсчет префикс-суфикс функции

```
p = [0] * N;
p[0] = 0;
j = 0;

for i in range(1, N) {
    while j > 0 and s[i] != s[j] {
        j = p[j - 1];
    }
    if s[i] == s[j] {
        j += 1;
    }
    p[i] = j;
}
```

У нас есть процедура (внутренний цикл while), который выполняется на N итерациях внешнего цикла.

(это не равносильно, что сам он исполнится N раз), например, внутренний цикл ниже суммарно работает O(N^2)

```
for i in range(1, N) {
    j = 0
    while j < i {
        j += 1;
    }
}
```

Метод потенциалов

```
p = [0] * N;
p[0] = 0;
j = 0;

for i in range(1, N) {
    while j > 0 and s[i] != s[j] {
        j = p[j - 1];
    }
    if s[i] == s[j] {
        j += 1;
    }
    p[i] = j;
}
```

Мы хотим посчитать стоимость N последовательных процедур, связанных между собой.
Это задача амортизационного анализа

Применим метод потенциалов

Метод потенциалов

Мы вводим особую величину, называемую потенциалом.

Для этой величины должно выполняться:

1) **Она больше или равна нулю все время выполнения последовательности из N процедур**
2) **Она связана с процедурами и определяет время их выполнения**

```
p = [0] * N;
p[0] = 0;
j = 0;

for i in range(1, N) {
    while j > 0 and s[i] != s[j] {
        j = p[j - 1];
    }
    if s[i] == s[j] {
        j += 1;
    }
    p[i] = j;
}
```

Такими чертами обладает j:

1) **в начале j инициализируется 0. Далее условие в цикле while не дает j стать меньше 0.**
2) **Как только j становится равным 0, цикл while прекращает свое выполнение, то есть j очевидным образом с ним связан**

Метод потенциалов

```
p = [0] * N;
p[0] = 0;
j = 0;

for i in range(1, N) {
    while j > 0 and s[i] != s[j] {
        j = p[j - 1];
    }
    if s[i] == s[j] {
        j += 1;
    }
    p[i] = j;
}
```

Чертами потенциала обладает j:

1) **в начале j инициализируется 0. Далее условие в цикле while не дает j стать меньше 0.**
2) **Как только j становится равным 0, цикл while прекращает свое выполнение, то есть j очевидным образом с ним связан**

Увеличение j: **Заметим, что j может быть увеличена только в строке j += 1, когда у нас совпадают символы на позиции i и j.**
Таким образом, максимально j может быть увеличено N - 1 раз (например, в случаи строки из одинаковых символов).
В строке j = p[j-1] она не может увеличится из свойств префикс-функции.

Уменьшение j: **Заметим, что j может быть уменьшена только в строке j = p[j-1]. Минимально уменьшится она при этом может на 1.**

Метод потенциалов

Увеличение j: Заметим, что j может быть увеличена только в строке j += 1, когда у нас совпадают символы на позиции i и j.
Таким образом, максимально j может быть увеличено N - 1 раз (например, в случаи строки из одинаковых символов).
В строке j = p[j-1] она не может увеличится из свойств префикс-функции.

Уменьшение j: Заметим, что j может быть уменьшена только в строке j = p[j-1]. Минимально уменьшится она при этом может на 1.

Таким образом, так как
1)j>=0
2) j = 0 в начале программы
3) оно может быть увеличено на больше N - 1 раз.
4) на каждом исполнении внутренней части цикла while оно уменьшается минимум на 1.

Получаем, что внутренняя часть цикла while исполнится не больше N раз.

```
p = [0] * N;
p[0] = 0;
j = 0;

for i in range(1, N) {
    while j > 0 and s[i] != s[j] {
        j = p[j - 1];
    }
    if s[i] == s[j] {
        j += 1;
    }
    p[i] = j;
}
```

Метод потенциалов

Внутренняя часть цикла while исполнится не больше N раз.

Посчитаем сложности в каждой строке

```
p = [0] * N;
p[0] = 0;
j = 0;

for i in range(1, N) {                    O(N)
    while j > 0 and s[i] != s[j] {
        j = p[j - 1];                     O(N)
    }
    if s[i] == s[j] {                     O(N)
        j += 1;                           O(N)
    }
    p[i] = j;                             O(N)
}
```

O(N) - на каждый заход в внутренний цикл мы потратим максимум 2 сравнения (чтобы зайти и выйти)

$$O(N) + O(N) + O(N) + O(N) + O(N) + O(N) = O(N)$$

Таким образом, алгоритм подсчета префикс-функции работают за линейной время O(N)