Студент: Кирилл Яценко
Группа: Яндекс
Дата: 3 октября 2023 г.

---

1. Мы можем доказать амортизационную сложность этого алгоритма, используя метод потенциалов.
2. Пусть потенциал P равен количеству единиц в битовом массиве a.
3. Изначально P=0, так как в a нет единиц.
4. Предположим, что на i-м шаге цикла (где i<n) выполняется carry>0 и ai=1.
5. В этом случае, carry += ai увеличивает значение carry на 1, и carry = carry / 2 уменьшает его в два раза. Таким образом, значение carry не изменяется.
6. При этом ai устанавливается в $carry\%2$, то есть равно 1. Это означает, что на каждом шаге цикла, где carry>0 и ai=1, значение ai устанавливается в 1.
7. Теперь рассмотрим следующий шаг цикла, где carry=0. В этом случае, ничего не происходит, и все остается без изменений. Внутренний цикл выполняется ровно n раз.
8. Из этого следует, что для каждой единицы в a, мы увеличиваем потенциал на 1, и этого достаточно, чтобы покрыть затраты времени для всех m прибавлений единицы.
9. Общее время работы операции можно оценить как время выполнения цикла плюс время на установку новых единиц в a.
10. Время выполнения цикла - O(n), так как цикл выполняется ровно n раз.
11. Время на установку новых единиц в a - O(m), так как у нас есть m прибавлений единицы.
12. Таким образом, суммарное время работы алгоритма составляет O(n + m), что и требовалось доказать.

```

 from typing import List, Tuple


 class AdvansedList:
   """
                                  n                                        m
   add(x, l, r):                   x
                        [l, r].
        O(n + m)                            ,
   
   """
 
   def __init__(self, data: List[int]) -> None:
     self.data: List[int] = data
     self.modifications: List[int] = [0 for i in data]
 
   def add(self, x: int, l: int, r: int):
     self.modifications[l] += x
     if r < len(self.data):
       self.modifications[r] -= x
```

```
    
    def sync(self):
        acc = 0
        for i, x in enumerate(self.modifications):
            acc += x
            self.data[i] += acc
        self.modifications = [0 for i in self.data]
    
    def apply_adds(self, qeries: List[Tuple[int, int, int]]):
        for args in qeries:
            self.add(*args)
        self.sync()
        return self.data
```

1. Данное решение представляет собой класс AdvansedList, который содержит в себе массив данных data, список модификаций modifications и несколько методов.

2. Метод $\_\_init\_\_$ инициализирует класс, принимая входные данные в виде списка чисел data и создает список модификаций modifications, состоящий из нулей, размером равным длине data.

3. Метод add принимает три аргумента: число x, номер начального элемента l и номер конечного элемента r. Он прибавляет x к каждому элементу на отрезке [l, r] массива data. Операция выполняется следующим образом: прибавляем x к элементу с индексом l и вычитаем x из элемента с индексом r. Если r выходит за границы массива data, то x просто прибавляется к элементу с индексом l.

4. Метод sync синхронизирует массив data со списком модификаций modifications. Для этого проходим по каждому элементу x и индексу i в списке модификаций и применяем следующий алгоритм: прибавляем x к текущему элементу data[i] и сохраняем сумму в переменной acc. Затем присваиваем data[i] значение acc, а затем обнуляем modifications.

5. Метод *apply_adds* применяет список запросов qeries к массиву data. Для каждого запроса вызывается метод add, а затем вызывается метод sync, чтобы синхронизировать массив data с актуальными модификациями. Наконец, метод возвращает массив data после выполнения всех запросов.

6. Таким образом, чтобы получить массив, получающийся из исходного после выполнения заданных запросов, достаточно создать объект класса AdvansedList с исходным массивом, вызвать метод $apply_adds$.