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

Задание 1

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 $a_i = 1$.
5. В этом случае, $carry+ = a_i$ увеличивает значение carry на 1, и $carry = carry/2$ уменьшает его в два раза. Таким образом, значение carry не изменяется.
6. При этом a_i устанавливается в $carry\%2$, то есть равно 1. Это означает, что на каждом шаге цикла, где $carry > 0$ и $a_i = 1$, значение a_i 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.

Задание 2

```python

from typing import List, Tuple


class AdvansedList:
    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* с соответствующими запросами и получить результат.

Задание 3

```

def remove_digits(num: int, k: int) -> int:
    num_str = str(num)
    n = len(num_str)
    stack = []
    removed = 0

    for i in range(n):
        while stack and removed < k and stack[-1] < num_str[i]:
            stack.pop()
            removed += 1

        if removed == k:
            stack += num_str[i:]
            break

        stack.append(num_str[i])
```

```
18
19      stack = stack[:-k] if removed < k else stack
20      result = int(''.join(stack))
21      return result
```

- Решение представлено функцией remove_digits, которая принимает целое число num и количество цифр, которые требуется удалить k.
- Функция сначала преобразует число в строковый формат и сохраняет его длину в переменную n. Затем создается стек, в котором будут храниться цифры числа. Также создается переменная removed, которая будет отслеживать количество уже удаленных цифр.
- Затем проходим по каждой цифре числа и выполняем следующие действия:
- 1. Пока в стеке есть элементы и количество уже удаленных цифр меньше k и верхний элемент стека меньше текущей цифры, удаляем верхний элемент из стека и увеличиваем количество удаленных цифр.
- 2. Если количество удаленных цифр достигло k, добавляем оставшиеся цифры числа в стек и прерываем цикл.
- 3. Иначе, добавляем текущую цифру в стек.
- После прохода по всем цифрам, проверяем, достигли ли мы нужного количества удаленных цифр k. Если да, удаляем последние k элементов из стека, иначе оставляем стек без изменений.
- Наконец, объединяем элементы стека в строку и преобразуем ее обратно в целое число result, который является максимально возможным числом после удаления цифр. Результат выводится на экран.

Задание 4

```
1  class SingleManager:
2      def __init__(
3          self,
4      ) -> None:
5          self.stack = []
6          self.min = 0
7
8      def open(self, i: int):
9          self.stack.append(i)
10
11     def close(self, i: int):
12         if self.stack:
13             self.stack.pop(-1)+1
14         else:
15             self.min = i
16
17     def low(self) -> int:
18         if self.stack:
19             return self.stack[-1]
20         return self.min
21
22
23
24 class TripleManager:
25     def __init__(self) -> None:
```

```
        self.ans = 0
        self.n = 1
        self.managers = []

    def process(self, s: str) -> Tuple[int, int]:
        self.ans = 0
        self.n = 1

        self.managers = [SingleManager(), SingleManager(), SingleManager()]

        for i, c in enumerate(s):
            if c == ')':
                self.managers[0].close(i+1)
            elif c == '(':
                self.managers[0].open(i+1)
            elif c == '}':
                self.managers[1].close(i+1)
            elif c == '{':
                self.managers[1].open(i+1)
            elif c == ']':
                self.managers[2].close(i+1)
            elif c == '[':
                self.managers[2].open(i+1)

            cur = min([i+1 - m.low() for m in self.managers])
            if cur > 0 and cur == self.ans:
                self.n += 1
            if cur > self.ans:
                self.ans = cur
                self.n = 1
        return self.ans, self.n
```

1. Данная программа решает задачу по поиску самой длинной подстроки строки, которая является правильной скобочной последовательностью. Время работы программы составляет O(n).

2. Программа состоит из двух классов: SingleManager и TripleManager.

3. Класс SingleManager представляет собой менеджер для работы со скобками одного типа. У него есть следующие методы:

4. - init(): конструктор класса, инициализирует пустой стек и минимальное значение индекса.

5. - open(i: int): добавляет индекс открывающей скобки в стек.

6. - close(i: int): если стек не пустой, удаляет последний индекс из стека; в противном случае обновляет минимальное значение индекса.

7. - low(): возвращает верхний элемент стека (индекс последней открывающей скобки) или минимальное значение индекса.

8. Класс TripleManager представляет собой менеджер для работы со скобками трех типов. У него есть следующие методы:

9. - init(): конструктор класса, инициализирует переменные ans, n и список managers.

10. - process(s: str) -> Tuple(int, int): метод, который обрабатывает строку s, содержащую все три типа скобок. Он инициализирует переменные ans и n нулями, создает экземпляры класса SingleManager для каждого типа скобок и помещает их в список managers. Затем перебирает все символы строки и для каждого символа вызывает соответствующий метод open() или close() соответствующего

экземпляра SingleManager. Затем вычисляется текущая длина правильной скобочной последовательности и обновляются значения переменных ans и n, если текущая длина больше текущего значения ans. В конце метод возвращает значения ans и n.

11. В основной программе создается экземпляр класса TripleManager и вызывается его метод process, передавая ему строку, для которой нужно найти самую длинную правильную скобочную последовательность. Возвращаемые значения метода process - это длина найденной подстроки и количество таких подстрок в строке.

Задание 5

(a) Без ограничений на дополнительную память:

Для решения данной задачи без ограничений на дополнительную память будем использовать алгоритм Кадана.

1. Инициализируем две переменные: maxsum, которая будет хранить максимальную сумму подотрезка, и currentsum, которая будет хранить текущую сумму подотрезка. Обе переменные устанавливаем равными первому элементу массива.
2. Проходим по массиву, начиная со второго элемента.
3. Для каждого элемента, сравниваем текущий элемент с суммой текущего подотрезка. Если текущий элемент больше суммы, то обновляем текущую сумму, иначе оставляем текущую сумму без изменений.
4. Проверяем, если текущая сумма больше максимальной суммы, то обновляем максимальную сумму.
5. Повторяем шаги 3-4 для всех элементов массива.
6. В конце, возвращаем максимальную сумму.

```
def max_subarray_sum(arr):
    max_sum = arr[0]
    current_sum = arr[0]

    for i in range(1, len(arr)):
        current_sum = max(arr[i], current_sum + arr[i])
        max_sum = max(max_sum, current_sum)

    return max_sum

array = [1, -2, 3, 4, -1, 2, 1, -5, 4]
max_sum = max_subarray_sum(array)
print(max_sum)
```

(b) Имея возможность завести не более одного массива целых чисел длины n плюс O(1) памяти:

Для решения этой задачи с ограничением на память модифицируем алгоритмом Кадана, так чтобы он хранил только начальную и конечную позиции максимального подотрезка.

1. Инициализируем переменные maxsum, currentsum, startpos, endpos, которые будут хранить максимальную сумму подотрезка, текущую сумму подотрезка, начальную позицию максимального подотрезка и конечную позицию максимального подотрезка соответственно. Все переменные устанавливаем равными первому элементу массива.
2. Проходим по массиву, начиная со второго элемента.

3. Для каждого элемента, сравниваем текущий элемент с суммой текущего подотрезка. Если текущий элемент больше суммы, то обновляем текущую сумму и начальную позицию подотрезка, иначе оставляем текущую сумму и начальную позицию без изменений.

4. Проверяем, если текущая сумма больше максимальной суммы, то обновляем максимальную сумму и конечную позицию подотрезка.

5. Повторяем шаги 3-4 для всех элементов массива.

6. В конце, возвращаем максимальную сумму и подотрезок с помощью начальной и конечной позиции.

```python
def max_subarray_sum(arr):
    max_sum = arr[0]
    current_sum = arr[0]
    start_pos = 0
    end_pos = 0

    for i in range(1, len(arr)):
        if arr[i] > current_sum + arr[i]:
            current_sum = arr[i]
            start_pos = i
        else:
            current_sum += arr[i]

        if current_sum > max_sum:
            max_sum = current_sum
            end_pos = i

    return max_sum, arr[start_pos:end_pos+1]
```