Студент: Яценко Кирилл

Задача о поиске минимума и максимума

Условие задания

Дан массив из $2n$ чисел. Найти минимальное и максимальное значение за $3n - 2$ сравнения.

Код на Python

```
def find_min_max(nums):
    n = len(nums) // 2
    min_num, max_num = nums[0], nums[1]

    if min_num > max_num:
        max_num, min_num = min_num, max_num

    for i in range(1, n):
        num1, num2 = nums[2 * i], nums[2 * i + 1]
        if num1 < num2:
            if num1 < min_num:
                min_num = num1
            if num2 > max_num:
                max_num = num2
        else:
            if num2 < min_num:
                min_num = num2
            if num1 > max_num:
                max_num = num1

    return min_num, max_num
```

Обяснение решения

При решении задачи используется стратегия сравнения пар чисел. Для каждой пары мы сразу определяем, какое из них меньше, а какое больше. Первую пару принимаем за начальные минимальное и максимальное значения. Затем делаем следующее:

1. Сравниваем числа пары между собой.
2. Сравниваем меньшее число пары с текущим минимумом.
3. Сравниваем большее число пары с текущим максимумом.

Таким образом, для каждой из n пар чисел проводится ровно 3 сравнения (кроме первой пары, для которой делается 1 сравнение), что в сумме даёт $3n - 2$ сравнения. Это соответствует условию задачи.

Поиск второго максимума в массиве

Условие задания

Необходимо найти второй максимум в массиве за $n + O(\log n)$ сравнений.

Код на Python

```python
def secondmax(arr):
    n = len(arr)
    if n == 0:
        return None, None
    if n == 1:
        return arr[0], None

    def compare_pairs(left, right):
        if left[0] > right[0]:
            return left, right
        else:
            return right, left

    history = [(arr[i], []) for i in range(n)]

    while len(history) > 1:
        next_level = []
        for i in range(0, len(history), 2):
            if i + 1 < len(history):
                winner, loser = compare_pairs(history[i], history[i + 1])
                winner[1].append(loser[0])
                next_level.append(winner)
            else:
                next_level.append(history[i])
        history = next_level

    max_element = history[0][0]
    fights = history[0][1]

    if not fights:
        return max_element, None

    second_max = fights[0]
    for f in fights[1:]:
        if f > second_max:
            second_max = f

    return max_element, second_max
```

Обяснение решения

Данный алгоритм использует метод турнирного дерева для определения второго максимума в массиве. На каждом шаге элементы массива сопоставляются парами, выигравшие элементы переходят на следующий уровень, а проигравшие сохраняются в истории победителя. Процесс продолжается до тех пор, пока не останется один максимальный элемент, история которого содержит всех его "соперников". Затем находится максимальный элемент из этой истории, который будет вторым максимумом массива. Это позволяет найти второй максимум за $n + O(\log n)$ сравнений, так как каждый элемент, кроме

одного, проигрывает ровно один раз, а дополнительные $\log n$ сравнения необходимы для определения второго максимума среди проигравших.

Кузнечик 1

Условие задания

Кузнечик умеет прыгать с одной ступеньки на следующую (с номером i на ступеньку с номером $i+1$) и прыгать через одну ступеньку (на ступеньку с номером $2i$). Необходимо найти количество способов добраться с нулевой ступеньки до ступеньки с номером n за время $O(n)$.

Код на Python

```python
def count_ways(n):
    dp = [0] * (n + 1)
    dp[0] = 1

    for i in range(1, n + 1):
        dp[i] += dp[i - 1]
        if i % 2 == 0:
            dp[i] += dp[i // 2]

    return dp[n]
```

Объяснение решения

В данном решении применяется метод динамического программирования. Создается массив `dp`, где `dp[i]` хранит количество способов добраться до ступеньки i. Динамическое заполнение массива начинается с того, что до начальной ступеньки (0-й) существует ровно один способ добраться — остаться на ней. Для каждой ступеньки i, количество способов добраться до нее состоит из суммы количества способов добраться до предыдущей ступеньки ($dp[i-1]$) и, в случае четного номера ступеньки, количества способов добраться до ступеньки с номером $i/2$.

Такие действия позволяют эффективно вычислить количество способов добраться до каждой ступеньки, вплоть до n-ой, при этом временная сложность алгоритма составляет $O(n)$, что удовлетворяет требуемым условиям задачи.

Кузнечик 2

Условие задания

Кузнечик умеет прыгать на две или три ступеньки вперёд, либо на одну ступеньку назад. Однако прыжок назад разрешается совершать не более одного раза подряд. Требуется найти количество способов добраться с нулевой ступеньки до n-й, используя $O(n)$ времени.

Код на Python

```python
def count_ways(n):
    if n <= 1:
        return 1
    dp = [0] * (n + 1)
    dp[0], dp[1] = 1, 1
    dp[2] = 3
    for i in range(3, n + 1):
        dp[i] = dp[i - 1] + 2 * dp[i - 2] + dp[i - 3]
    return dp[n]
```

Объяснение решения

Для решения данной задачи используется динамическое программирование. Создаётся массив `dp`, где `dp[i]` хранит количество способов добраться до i-й ступеньки. Основываясь на правилах задачи, мы можем понять, что каждая ступенька может быть достигнута из трёх предыдущих позиций: $dp[i-1]$, $2 \cdot dp[i-2]$, и $dp[i-3]$. Поэтому значение для `dp[i]` получается суммированием значений этих трёх предыдущих позиций. Этот алгоритм позволяет вычислить количество способов достижения n-й ступеньки за время $O(n)$.

Задача о минимальной стоимости достижения n-й ступеньки

Условие задания

Кузнечик может прыгать на одну или две ступеньки вперёд, при этом у каждой ступеньки есть своя стоимость. Требуется найти минимальную стоимость, чтобы добраться с нулевой ступеньки до n-й, за время $O(n)$.

Код на Python

```python
def min_cost_to_reach_nth_stair(cost):
    n = len(cost)
    if n == 0:
        return 0
    if n == 1:
        return cost[0]

    dp = [0 for _ in range(n)]
    dp[0], dp[1] = cost[0], cost[1]

    for i in range(2, n):
        dp[i] = min(dp[i - 1], dp[i - 2]) + cost[i]
    return min(dp[n - 1], dp[n - 2])
```

Объяснение решения

В этой задаче используется метод динамического программирования для нахождения минимальной стоимости достижения n-й ступеньки. Для каждой ступеньки i, рассчитываем минимальную стоимость достижения этой ступеньки как минимум из сумм стоимостей достижения предыдущих двух ступенек

$(i-1$ и $i-2)$ плюс стоимость текущей ступеньки. Результатом будет минимальное значение стоимости между последней и предпоследней ступеньками, так как кузнечик может достичь последней ступеньки как непосредственно, так и через один прыжок назад.

5.2.6

Пусть алгоритм $A(X, i)$ корректно находит i-ый по порядку элемент в любом массиве чисел X и использует только попарные сравнения элементов. Покажите, что, в массиве X можно найти все элементы, меньшие i-ого, и все элементы, большие i-ого, используя только результаты сравнений, которые сделает $A(X, i)$.

Для решения этой задачи рассмотрим, как алгоритм $A(X, i)$ работает. Алгоритм находит i-ый элемент по порядку в массиве X при помощи попарных сравнений. В процессе работы алгоритма каждое сравнение определяет, какой из двух элементов ближе к искомому i-му элементу, или же является самим i-м элементом.

Рассуждение:

После выполнения алгоритма $A(X, i)$ все элементы в массиве X могут быть разделены на три группы:

1. Элементы, меньшие i-ого по порядку $(< i)$
2. Элемент i-ый по порядку (точно найденный алгоритмом A)
3. Элементы, большие i-ого по порядку $(> i)$

Помимо того, что $A(X, i)$ находит i-ый по порядку элемент, процесс сравнения, используемый в A, также позволяет определить относительное расположение остальных элементов по отношению к i-му элементу.

Если в процессе работы сравнивались элементы a и b и было установлено, что $a < b$ и один из этих элементов соответствует искомому i-му элементу, то можно утверждать, что другой элемент находится по соответствующую сторону от i-го (либо перед ним, если он меньше, либо после, если больше). Алгоритм не может прийти к выводу о том, какой элемент является i-м, не проведя сравнения, которое неявно устанавливает отношение "меньше"или "больше"между проверяемым элементом и i-м элементом.

Вывод:

Таким образом, используя только сравнения, сделанные алгоритмом $A(X, i)$, можно не только найти i-й элемент, но и разделить остальные элементы массива на две группы: элементы, меньшие i-го, и элементы, большие i-го. Это становится возможным, потому что каждое сравнение вносит вклад в понимание того, как элементы массива относятся к находимому i-му элементу.

Таким образом, результаты сравнений, сделанных алгоритмом $A(X, i)$ достаточны для выполнения задачи.

QuickHeap: куча на основе алгоритма QuickSelect

QuickHeap представляет собой структуру данных типа кучи, которая для операций вставки элемента (Insert) и извлечения минимального элемента (ExtractMin) использует алгоритм QuickSelect, позволяя достигать амортизированного времени выполнения $O(\log n)$ для данных операций.

1 Описание структуры данных

Структура данных QuickHeap состоит из двух частей:

- Буфера для входящих элементов, куда новые элементы добавляются при операции вставки.
- Основной кучи, в которую элементы из буфера перемещаются с учетом медианы, определяемой используя алгоритм QuickSelect.

2 Операции QuickHeap

2.1 Вставка (Insert)

При вставке элемента, он сначала помещается в буфер. Когда размер буфера достигает заданного порога, выбирается медианный элемент с помощью QuickSelect. Элементы меньше медианы перемещаются в основную кучу как минимальные элементы, тем самым поддерживая свойство кучи.

2.2 Извлечение минимума (ExtractMin)

Для извлечения минимального элемента просматривается как основная куча, так и буфер (при необходимости, применяя QuickSelect для определения медианы в буфере). Используется свойство кучи для выбора и удаления минимального элемента.

3 Амортизированный анализ времени выполнения

Для анализа времени выполнения ключевым элементом является операция QuickSelect, которая требуется для перебалансировки буфера и основной кучи. QuickSelect имеет амортизированное время выполнения $O(n)$, однако так как буфер перебалансируется только при достижении порога его размера, вероятная амортизация операций вставки и извлечения минимума составляет $O(\log n)$, предполагая, что:

- Количество операций перебалансировки мало по сравнению с общим числом операций.
- Размер буфера оптимально подобран для минимизации амортизированного времени выполнения обеих операций.

Таким образом, благодаря эффективности QuickSelect и стратегии ленивой перебалансировки, QuickHeap обеспечивает эффективное амортизированное время выполнения для операций вставки и извлечения минимума.