Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

(Университет ИТМО)

# ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №3
по курсу «Алгоритмы и структуры данных»

Тема: СОРТИРОВКА СЛИЯНИЕМ, МЕТОД ДЕКОМПОЗИЦИИ
Вариант 21

Санкт-Петербург, 2024

СОДЕРЖАНИЕ

ЗАДАЧИ ПО ВАРИАНТУ

# 1 Сортировка слиянием

## Задание

Реализовать алгоритм сортировки слиянием

## Код

```
import typing as tp




def merge(list1: tp.List[int], list2: tp.List[int], target: tp.List[int], start_index:
int = 0) -> None:
    """
    Слияние list1 и list2 в target, начиная с индекса target = start
    :param list1: отсортированный список 1
    :param list2: отсортированный список 2
    :param target: список, в который будет помещён результат слияния списков 1 и 2
    :param start_index: индекс в списке target, откуда начнётся запись элементов
    """
    cur1 = cur2 = 0
    cur_target = start_index


    while cur1 < len(list1) and cur2 < len(list2):
        if list1[cur1] <= list2[cur2]:
            target[cur_target] = list1[cur1]
            cur1 += 1
        else:
            target[cur_target] = list2[cur2]
            cur2 += 1
        cur_target += 1


    if cur1 == len(list1):
        for i in range(cur2, len(list2)):
            target[cur_target] = list2[i]
            cur_target += 1
    else:
        for i in range(cur1, len(list1)):
            target[cur_target] = list1[i]
            cur_target += 1


def merge_sort(lst: tp.List[int]) -> None:
    """
    Сортировка списка слиянием на месте
    :param lst: список, который нужно отсортировать
    """

    len_merging_lists = 1
    while len_merging_lists < len(lst):
        start_index = 0
        while start_index + len_merging_lists < len(lst):
            if lst[start_index + len_merging_lists - 1] <= lst[start_index +
len_merging_lists]:
                start_index += 2 * len_merging_lists
                continue

            merge(lst[start_index : start_index + len_merging_lists],
                  lst[start_index + len_merging_lists : min(len(lst), start_index + 2
```

```
* len_merging_lists)],
                  lst, start_index)

            start_index += 2 * len_merging_lists
        len_merging_lists *= 2
```

## Анализ кода

Сортировка слиянием (Merge Sort) — это алгоритм "разделяй и властвуй", который рекурсивно делит список на две половины до тех пор, пока не останутся списки длиной 1, которые затем последовательно сливаются в один отсортированный список. Основная идея — разбить массив на части, отсортировать каждую и затем слить их обратно, сохраняя порядок.

Алгоритм состоит из следующих шагов:

1. Разбиваем список на две половины.
2. Рекурсивно применяем сортировку слиянием к каждой половине.
3. Выполняем слияние двух отсортированных половин.

Так как рекурсии не слишком практичны: они используют много памяти и могут работать дольше итеративных алгоритмов, - было принято решение реализовывать сортировку без рекурсии. Вместо разделения массива на две части каждый раз (движение сверху вниз по дереву рекурсии), список сразу делится на минимальные отсортированные блоки, затем они попарно сливаются в блоки побольше, пока размер блока не станет равным размеру самого списка (движение снизу вверх по дереву рекурсии).

Функция `merge` позволяет объединять два отсортированных подмассива в один на месте, что снижает расходы на память.

Функция `merge_sort` сортирует массив in-place, не создавая дополнительных массивов, кроме необходимых для временного хранения частей.

**Оценка времени работы**:

Время работы: $O(n*\log_2 n)$, где n - длина списка. На каждом уровне рекурсии выполняется операция слияния, которая требует $O(n)$, а уровней рекурсии – $O(\log_2 n)$.

Затраты памяти: $O(n)$ на временные массивы, поскольку используется in-place модификация.

## Вывод

Итеративные алгоритмы в большинстве случаев более эффективны по памяти и времени выполнения. Рекурсия удобна и проста для понимания в некоторых задачах, но из-за ограничений стека вызовов, накладных расходов и проблем с производительностью рекурсивные алгоритмы часто уступают итеративным.

## 2 Нахождение количества инверсий

### Задание

Найти количество инверсий в перестановке за $O(n*\log_2 n)$

### Код

```
import typing as tp


def merge_count_inversions(list1: tp.List[int], list2: tp.List[int], target:
tp.List[int], start_index: int = 0) -> int:
    """
    Слияние list1 и list2, подсчет суммы по i от 0 до длины list1: количество
элементов из list2, меньших list1[i]
    :param list1: отсортированный список 1
    :param list2: отсортированный список 2
    :param target: список, в который будет помещён результат слияния списков 1 и 2
    :param start_index: индекс в списке target, откуда начнётся запись элементов
    :return: сумма по i от 0 до длины list1: количество элементов из list2, меньших
list1[i]
    """

    cnt = 0
    cur1 = cur2 = 0   # текущие индексы в list1 и list2 соотв.
    cur_target = start_index # текущий индекс в target
    while cur1 < len(list1) and cur2 < len(list2):
        if list1[cur1] <= list2[cur2]:
            target[cur_target] = list1[cur1]
            cur1 += 1
            cnt += cur2
        else:
            target[cur_target] = list2[cur2]
            cur2 += 1
        cur_target += 1


    if cur1 == len(list1):
        for i in range(cur2, len(list2)):
            target[cur_target] = list2[i]
            cur_target += 1
    else:
        for i in range(cur1, len(list1)):
            target[cur_target] = list1[i]
            cur_target += 1
        cnt += (len(list1) - cur1) * len(list2)

    return cnt


def merge_sort_count_inversions(lst: list) -> int:
    """
    Нахождение количества инверсий
    :param lst: список, в котором нужно найти количество инверсий (отсортируется в
процессе)
    :return: количество инверсий в lst
    """
    count_inv = 0 # счетчик инверсий
    len_merging_lists = 1 # длина сливающихся подсписков
    while len_merging_lists < len(lst):
        start_index = 0  # начальный индекс пары сливающихся подсписков
        while start_index + len_merging_lists < len(lst):
            if lst[start_index + len_merging_lists - 1] <= lst[start_index +
len_merging_lists]:
                start_index += 2 * len_merging_lists
                continue
```

```
            count_inv += merge_count_inversions(lst[start_index : start_index +
len_merging_lists],
                    lst[start_index + len_merging_lists : min(len(lst), start_index + 2
* len_merging_lists)],
                    lst, start_index)

            start_index += 2 * len_merging_lists
        len_merging_lists *= 2

    return count_inv
```

## Анализ кода

Инверсии в массиве — это такие пары элементов, что `i < j`, а `A[i] > A[j]`. Подсчёт количества инверсий можно провести при помощи модифицированного алгоритма сортировки слиянием, используя тот факт, что количество инверсий списка равно сумме количества инверсий двух его частей и суммы, где каждое `i`-е слагаемое – количество элементов из правой части, таких, что они больше `i`-ого элемента из левой части. Первое и второе слагаемые находятся рекурсивно, а третье можно посчитать при слиянии. Причем, чтобы сделать эти вычисления быстрее, все-таки необходимо отсортировать части списка. Тогда, если при сравнении текущих элементов частей списков получается, что текущий элемент левой части меньше текущего из правой, можно утверждать, что все элементы правой, меньше текущего элемента из правой части (а таких элементов = индекс текущего правого), меньше текущего из левой.

Алгоритм состоит из:

1. Разделения списка на две части.
2. Рекурсивного подсчёта инверсий в каждой из частей.
3. Подсчёта инверсий, возникающих при слиянии двух частей.

**Оценка времени работы**: аналогично обычной сортировке слиянием

Время работы: $O(n*\log_2 n)$, аналогично сортировке слиянием.

Затраты памяти: $O(n)$ на временные массивы для слияния.

## Вывод

Встроенная модификация алгоритма сортировки слиянием позволяет одновременно с сортировкой подсчитывать инверсии.

# 3 Бинарный поиск

## Задание

Реализовать алгоритм бинарного поиска элемента в списке.

## Код

```python
import typing as tp


def bin_pow(lst: tp.List[int], value: int) -> int:
    """
    Поиск value в списке lst
    :param lst: список, в котором будет производиться поиск
    :param value: искомый элемент
    :return: индекс искомого элемента или -1, если его нет
    """

    lst_indexes = list(range(len(lst)))
    lst_indexes.sort(key=lambda index: lst[index])


    left = 0
    right = len(lst) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if lst[lst_indexes[mid]] == value:
            return lst_indexes[mid]
        if lst[lst_indexes[mid]] < value:
            left = mid + 1
        else:
            right = mid - 1

    return -1
```

## Анализ кода

Бинарный поиск — это алгоритм поиска элемента в отсортированном списке. Он работает по принципу "разделяй и властвуй", на каждом шаге деля список пополам и проверяя, находится ли искомый элемент в левой или правой части.

Чтобы быстро выводить индекс элемента в неотсортированном списке, мы сортируем не сами элементы, а их индексы: индекс `i` является больше индекса `j`, если `lst[i]` больше `lst[j]`. Т.е. на позиции `i` хранится значение позиции в неотсортированном массиве, где стоял элемент, который в отсортированном массиве имеет индекс `i`.

Вместо рекурсии опять же использовался итеративный алгоритм, где роль разделения массива на две части играют два указателя, показывающие, с какой частью списка мы работаем. Алгоритм итеративного бинарного поиска начинается с установки начального значения границ поиска (`left = 0, right = n-1`). Затем определяется позиция `mid` как среднее между `left` и `right`. Элемент на позиции `mid` сравнивается с целевым значением. Если они равны, поиск завершен успешно. В противном случае, если элемент меньше целевого, `left` увеличивается, а если больше, `right` уменьшается. Процесс продолжается до тех пор, пока `left` не станет больше `right`, указывая на отсутствие элемента.

**Оценка времени работы**: аналогично обычной сортировке слиянием

Время работы: $O(\log_2 n)$, где $n$ - это количество элементов в массиве. Это происходит потому, что каждый шаг поиска делит массив на две половины, сокращая область поиска примерно вдвое.

Затраты памяти: $O(1)$ на переменные.

## Вывод

Бинпоиск – понятный (даже без использования рекурсий) и эффективный алгоритм.

# 4 Поиск мажорирующего элемента

## Задание

Найти элемент, который встречается в списке более половины раз, если такой есть

## Код 1

```python
import typing as tp


def majority_element_recursion(lst: tp.List[int], start: int = 0, end: int = -1) ->
tp.Tuple[tp.Optional[int], int]:
    """
    Поиск мажорирующего элемента на части lst от start до end (не включительно)
    :param lst: список
    :param start: начальный индекс списка
    :param end: конечный индекс (end = len(lst) <-> end == -1)
    :return: значение мажорирующего элемента (None, если его нет) и
    количество раз, которое он встречается в списке (-1, если его нет)
    """

    if end == -1:
        end = len(lst)

    num_elements = end - start
    if num_elements == 0:
        return None, -1
    if num_elements == 1:
        return lst[start], 1

    mid = start + num_elements // 2
    majority_left, num_left = majority_element_recursion(lst, start, mid)
    majority_right, num_right = majority_element_recursion(lst, mid, end)

    if num_left != -1:
        for i in range(mid, end):
            if lst[i] == majority_left:
                num_left += 1
        if num_left > num_elements // 2:
            return majority_left, num_left

    if num_right != -1:
        for i in range(start, mid):
            if lst[i] == majority_right:
                num_right += 1
        if num_right > num_elements // 2:
            return majority_right, num_right

    return None, -1
```

## Анализ кода 1

Алгоритм реализован рекурсивно. Каждый раз он делит исходный список на две части, находит мажорирующий элемент для левой и правой частей, после чего проверяет, являются ли мажорирующие элементы частей (если они найдены) мажорирующими элементами на всем списке.

**Оценка времени работы**:

Время работы: $O(\log_2 n)$, где $n$ - это количество элементов в массиве. Это происходит потому, что каждый шаг поиска делит массив на две половины, сокращая область поиска примерно вдвое.

Затраты памяти: $O(\log_2 n)$.

## Код 2

```python
def majority_element_line(lst: tp.List[int]) -> tp.Tuple[tp.Optional[int], int]:
    """
    Поиск мажорирующего элемента в lst за линию
    :param lst: список
    :return: значение мажорирующего элемента (None, если его нет) и
    количество раз, которое он встречается в списке (-1, если его нет)
    """

    num_without_pair = 0 # количество элементов которые пока без пары
    candidate = None # Значение элементов без пары

    for elem in lst:
        if num_without_pair == 0:
            candidate = elem
            num_without_pair += 1
        elif elem == candidate:
            num_without_pair += 1
        else:
            num_without_pair -= 1

    cnt = 0
    for elem in lst:
        if elem == candidate:
            cnt += 1

    return (candidate, cnt) if cnt > len(lst) // 2 else (None, -1)
```

## Анализ кода 2

Алгоритм основан на том факте, что если в списке есть мажорирующий элемент, то если поделить элементы на пары, такие что ни в какой паре не встречается 2 одинаковых элемента, без пары останутся элементы, значение которых и будет мажорирующим.

Таким образом, алгоритм предполагает, что в списке есть мажорирующий элемент, находит кандидата и проверяет, действительно ли он мажорирующий.

**Оценка времени работы**:

Время работы: $O(n)$, где $n$ - это количество элементов в массиве. Т.к. алгоритм просто проходится по списку два раза.

Затраты памяти: $O(1)$.

## Вывод

Возможно, первая реализация интуитивно больше понятна, тем не менее она сильно уступает по времени и затратам памяти второму алгоритму.

# 5 Поиск подмассива с максимальной суммой за линию

## Задание

Найти подмассив с максимальной суммой и вывести его сумму, начальный и конечный индексы.

## Код

```
import typing as tp


def find_max_subarray(lst: tp.List[int]) -> tp.Tuple[int, int, int]:
    """
    Поиск максимального подмассива
    :param lst: список
    :return: сумма элементов максимального подмассива, стартовый индекс, конечный
индекс
    """

    max_subarray = lst[0] # сумма элементов максимального подмассива
    left, right = 0, 1 # границы максимального подмассива


    sm = 0 # сумма подмассива от 0 до текущего индекса
    min_sm = min_sm_len = 0 # подмассив от 0 до min_sm_len с минимальной суммой
    for ind, elem in enumerate(lst):
        sm += elem
        if max_subarray < sm - min_sm:
            max_subarray = sm - min_sm
            left = min_sm_len
            right = ind + 1

        if min_sm > sm:
            min_sm = sm
            min_sm_len = ind + 1

    return max_subarray, left, right
```

## Анализ кода

Алгоритм основан на следующих фактах:

1. Зная сумму всех подмассивов, начинающихся с первого элемента, можно легко вычислить сумму любого подмассива. Т.е., чтобы узнать, какая сумма у подмассива lst[left : right] нужно из суммы подмассива lst[: right] вычесть сумму подмассива lst[: left].
2. Чтобы узнать сумму максимального подмассива, заканчивающегося элементом cur – 1, нужно знать сумму подмассива lst[: cur] и сумму минимального подмассива, начинающегося с первого элемента (заканчиваться он может максимум элементом cur - 2). Отнимая от первого значения второе, получаем исковый резуьтат.

Используя эти факты, алгоритм совершает следующие действия:

Считает текущую сумму подмассива lst[: cur] = lst[: cur - 1] + cur.

От этого значения отнимает сумму минимального подмассива, начинающегося с первого элемента. Таким образом получается значение максимального подмассива, заканчивающегося элементом cur-1.

Если полученная сумма больше всех подобных, ранее подсчитанных сумм, т.е. больше максимума из сумм подмассивов массива lst[:cur - 1], то максимальная сумма подмассивов массива [:cur] будет равна значению максимального подмассива, заканчивающегося элементом cur - 1. Иначе она останется равной максимальной сумме подмассивов массива lst[:cur - 1].

В конце обновляется минимум из сумм подмассивов, начинающихся первым элементом.

**Оценка времени работы**:

Время работы: `O(n)`, где `n` - это количество элементов в массиве. Т.к. все происходит за один проход по списку.

Затраты памяти: `O(1)` на переменные.

## Вывод

Алгоритм Кадане, реализованный выше, - эффективный алгоритм нахождения максимального подмассива, который можно оптимизировать так, чтобы память практически не использовалась.

## 6 Умножение полиномов

### Задание

Реализовать алгоритм умножения полиномов менее чем за `O(n^2)`

### Код

```
def mult_polynomials(f: tp.List[int], g: tp.List[int]) -> tp.List[int]:
    """
    Умножение полинома f на полином g (если степени полиномов разные, они дополняются
нулями)
    Полиномы представляются в виде:
    f = [a0 + a1 * x + a2 * x^2 + ...], g = [b0 + b1 * x + b2 * x^2 + ...]
    :param f: [a0, a1, a2, ...]
    :param g: [b0, b1, b2, ...]
    :return: полином степени deg(f) + deg(g) - 1 в аналогичном виде
    """

    if not f or not g:
        raise ValueError("Empty polynomials")

    if len(f) < len(g):
        f.extend([0] * (len(g) - len(f)))
    elif len(f) > len(g):
        g.extend([0] * (len(f) - len(g)))


    def mult_polynomials_recursion(f: tp.List[int], g: tp.List[int],
                                   f_start: int, f_end: int,
                                   g_start: int, g_end: int) -> tp.List[int]:
        len_f = f_end - f_start
        len_g = g_end - g_start
        assert len_f == len_g

        if len_f == 1:
            return [f[f_start] * g[g_start]]

        mid = len_f // 2
        f_mid = f_start + mid
        g_mid = g_start + mid

        f_left_g_left = mult_polynomials_recursion(f, g, f_start, f_mid, g_start,
g_mid)
        f_right_g_right = mult_polynomials_recursion(f, g, f_mid, f_end, g_mid, g_end)

        f_left_plus_f_right = f[f_mid:f_end]
        g_left_plus_g_right = g[g_mid:g_end]
        for i in range(f_start, f_mid):
            f_left_plus_f_right[i - f_start] += f[i]
        for i in range(g_start, g_mid):
            g_left_plus_g_right[i - g_start] += g[i]

        sum_f_sum_g = mult_polynomials_recursion(f_left_plus_f_right,
g_left_plus_g_right,
                                                 0, f_end - f_mid,
                                                 0, g_end - g_mid)

        result = [0] * (len_f * 2 - 1)

        for i in range(len(f_left_g_left)):
            result[i] += f_left_g_left[i]
            result[mid + i] -= f_left_g_left[i]

        for i in range(len(f_right_g_right)):
            result[mid * 2 + i] += f_right_g_right[i]
            result[mid + i] -= f_right_g_right[i]
```

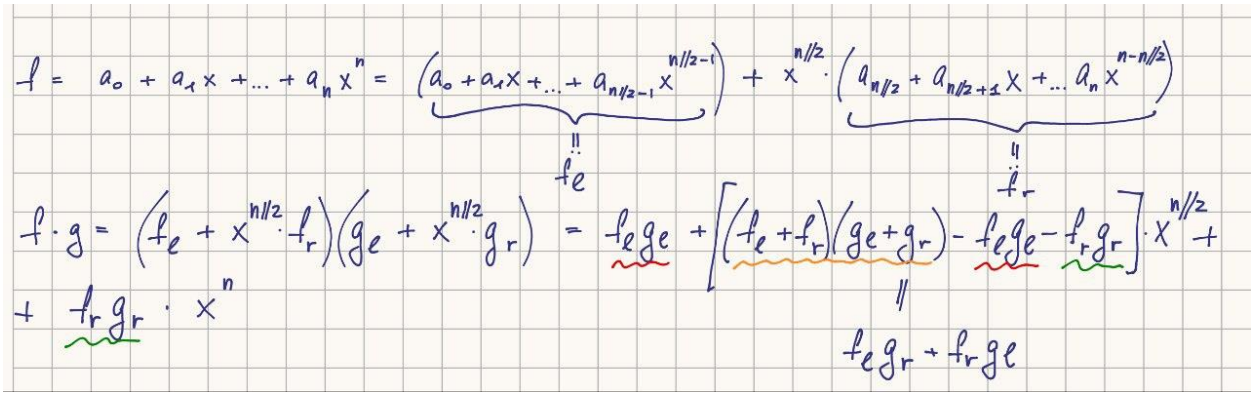
```
        for i in range(len(sum_f_sum_g)):
            result[mid + i] += sum_f_sum_g[i]

        return result

    return mult_polynomials_recursion(f, g, 0, len(f), 0, len(g))
```

## Анализ кода

Алгоритм умножения полиномов, представленный в коде, основан на методе *разделяй и властвуй*. Этот подход разбивает полиномы на части, умножает их рекурсивно и собирает результат, используя стратегию, похожую на умножение чисел с разбиением по разрядам.



Таким образом, можно рекурсивно вызвать три умножения полиномов (частей исходного), а потом собрать результат с учетом сдвига (степени x, на который умножается произведение)

1. **Разбиение полиномов**:

Алгоритм делит каждый полином на две части:

- Левая часть — это полиномы с меньшими степенями.
- Правая часть — полиномы с большими степенями.

Например, если полином `f(x) = a0 + a1·x + a2·x^2 + a3·x^3`, то он будет разделен на:

- Левую часть: `f_left(x) = a0 + a1·x`
- Правую часть: `f_right(x) = a2 + a3·x`

2. **Рекурсивное умножение**:

После разбиения двух полиномов `f` и `g` на левые и правые части, алгоритм рекурсивно умножает:

- Левую часть полинома `f` на левую часть полинома `g`.
- Правую часть полинома `f` на правую часть полинома `g`.

3. **Вычисление промежуточных сумм**:

Чтобы уменьшить количество операций умножения, алгоритм также складывает левую и правую части полиномов `f` и `g` (результаты копируются в новые списки) и умножает суммы. Это позволяет исключить лишние вычисления.

4. **Сборка результата**:

После рекурсивного умножения левых и правых частей, а также суммы левых и правых частей, результат собирается в один полином с поправками на позиции коэффициентов. Это делает алгоритм более эффективным по сравнению с наивным умножением, которое требует умножения каждого коэффициента одного полинома на каждый коэффициент другого.

**Оценка времени работы**:

Время работы: $O(n^{\log_2 3})$, где n - длина списка.

## Вывод

Ужасный алгоритм с точки зрения использования памяти. Была попытка исправить это, но в рекурсии в любом случае нужно копирование списков (для вычисления списка суммы левой и правой частей полинома) перед рекурсивным вызовом. Тем не менее, алгоритм работает быстрее наивного.

# ВЫВОД ПО ЛАБОРАТОРНОЙ РАБОТЕ

Метод "разделяй и властвуй" широко используется в алгоритмах и имеет как преимущества, так и недостатки, которые зависят от природы задачи и особенностей реализации. Вот ключевые плюсы и минусы этого подхода:

**Преимущества**

1. Эффективность на больших задачах:

Алгоритмы, использующие метод разделяй и властвуй, часто имеют логарифмическую или линейно-логарифмическую сложность. Например, быстрая сортировка и сортировка слиянием работают за $O(\log_2 n)$, что делает их более эффективными по сравнению с квадратичными алгоритмами на больших входных данных.

2. Простота проектирования для рекурсивных задач:

Разделяя задачу на более мелкие подзадачи, можно сократить сложность логики каждой из них. Это упрощает разработку рекурсивных алгоритмов, таких как нахождение максимального подмассива, умножение матриц и вычисление дискретного преобразования Фурье.

**Недостатки**

1. Затраты на рекурсивные вызовы:

Рекурсивные реализации требуют дополнительной памяти для стека вызовов. Если глубина рекурсии велика, например, при неблагоприятных входных данных в быстрой сортировке, это может привести к переполнению стека. Тем не менее многие алгоритмы, использующие рекурсию можно переписать в итеративные, просто делать это иногда трудно, читаемость кода и его понимание ухудшаются.

2. Избыточное копирование данных:

В некоторых алгоритмах приходится копировать подмассивы или подстроки для обработки отдельных частей, что приводит к увеличению затрат по памяти и времени. Тем не менее некоторые алгоритмы вполне реально оптимизировать в этом плане, но в таком случае, код становится менее читаемым, подключается больше параметров.

Метод "разделяй и властвуй" подходит для задач, где большие объемы данных могут быть разбиты на независимые и простые подзадачи. Несмотря на его преимущества для повышения производительности и возможности параллелизации, он требует тщательной реализации, чтобы избежать затрат на копирование данных и учитывать потенциальные проблемы с глубокой рекурсией.