САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

# Отчет по лабораторной работе №3
# по курсу «Алгоритмы и структуры данных»
# Тема: Быстрая сортировка, сортировки за линейное время.

Выполнил:
Мезенцев Богдан
К 3139
Проверил:
Афанасьев Антон Владимирович

Санкт-Петербург
2024 г.

# Содержание отчёта

# Задачи по варианту

## Задание 1. Улучшение Quick sort

1) В начале реализуем функцию *Partition3()* для разделения массива на на три подмассива по опорным элементам, которая на вход получает массив($A$), индекс левого элемента массива($l$) и индекс правого элемента($r$).

```
def Partition3(A, l, r):
    """Трехстороннее разделение массива"""
    x = A[l]  # индекс опорного элемента
    i = l + 1  # индекс для элементов меньших опорного
    j = r  # индекс для элементов больших опорного
    k = l + 1  # индекс текущего элемента

    while k <= j:
        if A[k] < x:
            A[i], A[k] = A[k], A[i]  # Перемещаем меньший элемент в левую часть
            i += 1
            k += 1
        elif A[k] > x:
            A[j], A[k] = A[k], A[j]  # Перемещаем больший элемент в правую часть
            j -= 1
        else:
            k += 1  # Если элемент равен опорному, просто переходим к следующему

    A[l], A[i - 1] = A[i - 1], A[l]  # Перемещаем опорный элемент на его правильную позицию
    return i - 1, j  # Возвращаем индексы границ частей
```

Описание алгоритма данной функции приведено в комментариях на скриншоте(см. выше)

2) Теперь реализуем функцию *Randomized_QuickSort()*, которая будет выполнять рекурсивный алгоритм сортировки подмассивов.

```
def Randomized_QuickSort(A, l, r):
    """Рандомизированная быстрая сортировка с трехсторонним разделением"""
    if l < r:
        k = random.randint(l, r)
        A[l], A[k] = A[k], A[l]
        m1, m2 = Partition3(A, l, r)   # m1, m2 - опорные элементы
        Randomized_QuickSort(A, l, m1 - 1)   # сортировка правого подмассива
        Randomized_QuickSort(A, m2 + 1, r)   # сортировка левого подмассива
    return A
```

Для большей эффективности алгоритма сортировки функция каждый раз выбирает случайный начальный элемент и ставит его на первое место в массиве. Далее определяются опорные элементы и рекурсивно вызывается эта же функция для сортировки подмассивов. В итоге возвращается отсортированный массив A.

## Задание 2. Анти-quick sort

Худший случай для QuickSort возникает, когда массив уже отсортирован или отсортирован в обратном порядке, или когда опорный элемент всегда выбирается так, что в результате разделения один из подмассивов будет содержать почти все элементы (а другой — только один или два). Это происходит, если каждый раз выбирается первый или последний элемент в массиве как опорный.
Генерируем массив отсортированный в обратном порядке с помощью функции *generate_array()*:

```
def genearte_array(n):
    array = list(range(n, 0, -1))  # Массив отсортированный в обратном порядке
    return array
```

## Задание 3. Сортировка пугалом

Реализуем функцию *can_sort_by_scarecrow()*, которая проверяет можно ли отсортировать полученный массив данным способом.

```python
def can_sort_by_scarecrow(n, k, sizes_of_matryoshkas):
    # Разбиваем индексы на блоки по модулю k
    blocks = [[] for _ in range(k)]
    for i in range(n):
        blocks[i % k].append(sizes_of_matryoshkas[i])

    # Сортируем каждый блок
    for block in blocks:
        block.sort()

    # Объединяем блоки в один отсортированный массив
    sorted_sizes = []
    for i in range(n):
        sorted_sizes.append(blocks[i % k][i // k])

    # Проверяем, является ли объединённый массив отсортированным
    if sorted_sizes == sorted(sizes_of_matryoshkas):
        return "ДА"
    else:
        return "НЕТ"
```

1) Функция на вход получает n - количество матрёшек, k - размах рук, массив с размерами матрёшек.
   Сначала необходимо создать массив с блоками, в которых будут индексы разделенные по модулю k.
2) Далее создаем отсортированный массив с блоками
3) Далее проверяем массив на отсортированность и получаем ответ.

## Задание 5. Индекс Хирша

Реализуем функцию *hirsch_index_search()*, которая вычисляет индекс Хирша на основе полученных статей.

```
def hirsch_index_search(citations):
    n = len(citations)  # количество статей
    count = [0] * (n + 1)  # массив для подсчёта процетированных статей


    for citation in citations:
        if citation >= n:
            count[n] += 1  # если статья цитируется >= n раз,
                           # то добаляем её в последнюю ячейку списка count
        else:
            count[citation] += 1

    h = 0
    t = 0
    for i in range(n, -1, -1):
        t += count[i]
        if t >= i:
            h = i
            break

    return h
```

1) Создаем массив count для подсчета процитированных статей
2) Далее проверяем статью на количество цитат
3) Возвращаем индекс h

## Задание 6. Сортировка целых чисел

Реализуем функцию *sorting_integers()*

```
def sorting_integers(A, B):
    C = []  # массив для перемноженных чисел
    for b in B:
        for a in A:
            C.append(a*b)
    Randomized_QuickSort(C, 0, len(C) - 1)  # используем быструю сортировку для массива C
    return sum(C[i] for i in range(0, len(C), 10))
```

Создаем массив, элементами которого будут попарные произведения элементов массивов A и B. И далее сортируем его алгоритмом быстрой сортировки. В результате возвращаем сумму, состоящую из каждого 10-го элемента массива  C.

## Задание 8. K ближайших точек к началу координат

Реализуем функцию distance_search() для поиска ближайших точек

```
def distance_search(n, k, points):
    # Вычисляем расстояние и сохраняем в виде кортежей (расстояние, (x, y))
    distances = [((x ** 2 + y ** 2), (x, y)) for x, y in points]
    # Сортируем по расстоянию
    distances.sort(key=lambda item: item[0])
    # Извлекаем первые K точек
    result_points = [point for _, point in distances[:k]]

    return result_points
```

Описание работы алгоритма представлено на скриншоте (см. выше)