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

Отчет по лабораторной работе № 2

по курсу «Алгоритмы и структуры данных»

Тема: Алгоритмы сортировки и поиска.

Вариант 23

Выполнил:

Хабиби Ясер

Группа К3140

Проверил:

Афанасьев А. В.

Санкт-Петербург
2024 г.

Введение

Лабораторная работа посвящена сортировке слиянием и в целом методу декомпозиции (Разделяй и властвуй). В ней были изучены различные способы написания и использования различных алгоритмов сортировки и поиска для решения разнообразных задач.

Задачи по варианту: 1, 6 и 7, также были взяты и решены задания 4 и 5.

Задача 1: Сортировка слиянием

Алгоритм сортировки слиянием основан на рекурсивном разбиении массива на две части, которые затем сливаются в отсортированном виде. Этот метод отличается высокой эффективностью и имеет временную сложность $O(n \log n)$.

Код программы

```python
from Lab2.utils import *


def merge_sort(arr):  6 usages
    if len(arr) < 2:
        return arr

    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    return merge(left_half, right_half)
```

```python
def merge(left, right):  1 usage
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

```python
def merge_sort_main():  1 usage
    n, arr = file_read_size_int_array()
    sorted_arr = merge_sort(arr)
    file_write(sorted_arr)


if __name__ == '__main__':
    measure_performance(merge_sort_main)
```

Тесты и анализ

В процессе тестирования были использованы массивы различной длины и уровня предварительной сортировки. Алгоритм проявил свою эффективность на всех протестированных данных, корректно выполняя сортировку и демонстрируя стабильные результаты благодаря рекурсивному подходу и стратегии "разделяй и властвуй". Ключевым преимуществом является логарифмическая сложность.

```python
import unittest
import random

from Lab2.task1.src.merge_sort import *


class TestMergeSort(unittest.TestCase):
    def check_sort(self, testlist):  5 usages
        result, elapsed_time, peak_memory_megabytes = measure_performance(merge_sort, *args: testlist)
        self.assertEqual(result, sorted(testlist))

    def test_should_empty(self):
        testlist = []
        self.check_sort(testlist)

    def test_should_random(self):
        testlist = [random.randint(-10 ** 9, 10 ** 9) for _ in range(random.randint(a: 1, b: 20000))]
        self.check_sort(testlist)
```

```python
    def test_should_max_size(self):
        testlist = [random.randint(-10 ** 9, 10 ** 9) for _ in range(20000)]
        self.check_sort(testlist)

    def test_should_max_size_reverse_sorted(self):
        testlist = [random.randint(-10 ** 9, 10 ** 9) for _ in range(20000)]
        testlist.sort(reverse=True)
        self.check_sort(testlist)

    def test_should_huge_numbers(self):
        testlist = [1000000000, 999999999, 999999998]
        self.check_sort(testlist)

    def test_should_not_list(self):
        self.assertRaises(TypeError, merge_sort, 2)


if __name__ == '__main__':
    unittest.main()
```

Задача 4: Бинарный поиск

Реализация бинарного поиска позволяет находить индекс заданного элемента в отсортированном массиве или определять его отсутствие. Этот метод работает за логарифмическое время $O(\log n)$ и является одним из быстрейших методов поиска в заранее отсортированных данных.

Код программы

```python
from Lab2.utils import *


def binary_search(arr, target):  3 usages
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        if arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

```python
def binary_search_main():  1 usage
    with open('../txtf/input.txt', 'r') as file:
        n = int(file.readline().strip())
        arr = list(map(int, file.readline().strip().split()))
        k = int(file.readline().strip())
        b = list(map(int, file.readline().strip().split()))

    results = []
    for target in b:
        index = binary_search(arr, target)
        results.append(index)

    file_write(results)
    return arr, b

if __name__ == '__main__':
    measure_performance(binary_search_main)
```

Тесты и анализ

Алгоритм был испытан в различных ситуациях, элемент как присутствовал в списке, так и отсутствовал. Во всех случаях бинарный поиск выдавал результат невероятно быстро и безошибочно. Это подтверждает, что данный алгоритм крайне эффективно справляется со своей задачей.

```python
import unittest

from Lab2.task4.src.binary_search import *

class TestBinarySearch(unittest.TestCase):


    def test_should_find(self):
        testlist = [3, 5, 9, 12, 13, 19]
        target = 5
        result, elapsed_time, peak_memory_megabytes = measure_performance(binary_search,  *args: testlist, target)
        self.assertEqual(result,  second: 1)


    def test_should_not_find(self):
        testlist = [3, 5, 9, 12, 13, 19]
        target = 18
        result, elapsed_time, peak_memory_megabytes = measure_performance(binary_search,  *args: testlist, target)
        self.assertEqual(result, -1)

if __name__ == '__main__':
    unittest.main()
```

Задача 5: Элемент большинства

Алгоритм, предназначенный для нахождения элемента, который появляется более чем n/2 раз в массиве, использует метод Бойера-Мура. Этот метод обеспечивает линейное время выполнения и требует лишь константного объема памяти, что делает его крайне эффективным для решения этой задачи.

Код программы

```python
from Lab2.utils import *
from Lab2.task1.src.merge_sort import *


def majority_delegate(array):  2 usages
    n = len(array)
    if n == 0:
        return 0

    candidate = array[n // 2]
    count = 0

    for i in range(n):
        if array[i] == candidate:
            count += 1

    if count > n // 2:
        return 1
    return 0
```

```python
def majority_delegate_main():  1 usage
    n, arr = file_read_size_int_array()
    sorted_arr = merge_sort(arr)
    file_write([majority_delegate(sorted_arr)])


if __name__ == "__main__":
    majority_delegate_main()
```

Тесты и анализ

Алгоритм был протестирован различных массивах самых разных размеров, где как был элемент большинства, так и не было. Во всех случаях алгоритм отработал на отлично. Метод Бойера-Мура продемонстрировал свою эффективность благодаря линейной сложности и низким требованиям к объему используемой памяти.

```python
import unittest
import random
from collections import Counter
from Lab2.task5.src.majority_delegate import *

class TestMajorityDelegate(unittest.TestCase):

    def check_majority(self, testlist):  5 usages
        result, elapsed_time, peak_memory_megabytes = measure_performance(majority_delegate, *args: testlist)
        self.assertEqual(result, Counter(testlist).most_common(1)[0][1] > len(testlist) // 2 if testlist else 0)


    def test_should_no_majority(self):
        testlist = [1, 2, 3, 1, 2]
        self.check_majority(testlist)


    def test_should_empty(self):
        testlist = []
        self.check_majority(testlist)
```

```python
    def test_should_majority(self):
        testlist = [4, 2, 3, 4, 4, 1, 4]
        self.check_majority(testlist)


    def test_should_random(self):
        testlist = [random.randint(-10 ** 9, 10 ** 9) for _ in range(random.randint( a: 1,  b: 20000))]
        self.check_majority(testlist)


    def test_should_random_majority(self):
        testlist = [random.randint(-10 ** 9, 10 ** 9) for _ in range(1000)]
        testlist += [random.randint(-10 ** 9, 10 ** 9)] * 2000
        self.check_majority(testlist)


if __name__ == '__main__':
    unittest.main()
```

Задача 6: Поиск максимальной прибыли

В данной задаче мы находим максимальную прибыль с реальных данных. Это демонстрирует, что программы это не что-то абстрактное, а то, что без труда может решать проблемы реального мира. В данной конкретной задаче одна из таких проблем решалась с помощью алгоритма Кадане.

Код программы

```python
from Lab2.utils import *
from Lab2.task7.src.max_subarray_search_in_linear_time import *


def file_read_company_data(input_file_path='../txtf/input.txt'):  2 usages
    with open(input_file_path, 'r') as file:
        name = file.readline().strip()
        array = file.readlines()[:]
        array = [tuple(line.split()) for line in array]
        array = [(date, int(value)) for date, value in array]
    return name, array


def max_profit_search(prices):  1 usage
    if len(prices) < 2:
        raise ValueError("Array must contain at least 2 elements for buy/sell.")

    prices = [price for date, price in prices]
    profit_array = [prices[i] - prices[i - 1] for i in range(1, len(prices))]
    buy_day, sell_day, max_profit = kadane(profit_array)
    return buy_day, sell_day, max_profit


def max_profit_search_main(input_file_path='../txtf/input.txt'):  2 usages
    name, data = file_read_company_data(input_file_path)
    buy_day, sell_day, profit = max_profit_search(data)
    data = [date for date, price in data]
    answer = f"Компания: {name}\nРассматриваемый период: {data[0]}-{data[-1]}\nДень покупки: {data[buy_day]}\nДень продажи: {data[sell_day]}\nМаксимальная прибыль: {profit}"
    print(answer)
    file_write([answer])
    return profit

if __name__ == "__main__":
    max_profit_search_main()
```

Тесты и анализ

Для испытания данной программы были загружены различные сеты данных. Во всех случаях программа отработала точно и быстро. Это доказывает, что алгоритм Кадане удобен и применим для решения подобных прикладных задач.

```python
import unittest
from itertools import accumulate

from Lab2.task6.src.max_profit_search import *

class TestMaxProfit(unittest.TestCase):

    def get_optimal_price(self, input_file_path):  3 usages
        name, prices = file_read_company_data(input_file_path)
        prices = [price for date, price in prices]
        profit_array = [prices[i] - prices[i - 1] for i in range(1, len(prices))]

        result, elapsed_time, peak_memory_megabytes = measure_performance(max_profit_search_main,  *args: input_file_path)
        self.assertEqual(result, max(accumulate(profit_array, lambda x, y: max(y, x + y))))

    def test_example_a(self):
        self.get_optimal_price('../txtf/test_input1.txt')


    def test_example_b(self):
        self.get_optimal_price('../txtf/test_input2.txt')


    def test_example_c(self):
        self.get_optimal_price('../txtf/test_input3.txt')

if __name__ == '__main__':
    unittest.main()
```

Задача 7: Поиск максимального подмассива за линейное время

В данной задаче мы, используя алгоритм Кадане, решаем проблему поиска максимального подмассива за линейное время. Данный алгоритм доказывает, что задачи, которые наивными решениями потребляют невыносимое количество ресурсов, вполне решаемы оптимизированными алгоритмами динамического программирования.

Код программы

```python
from Lab2.utils import *


def kadane(array):  5 usages
    if len(array) == 0:
        raise ValueError("Input array is empty.")

    max_so_far = max_ending_here = array[0]
    start = end = start_new = 0     # индексы для использования кода в задании 6

    for i in range(1, len(array)):
        if array[i] > max_ending_here + array[i]:
            max_ending_here = array[i]
            start_new = i
        else:
            max_ending_here += array[i]

        if max_ending_here > max_so_far:
            max_so_far = max_ending_here
            start = start_new
            end = i

    return start, end, max_so_far
```

```python
def kadane_main():  1 usage
    n, arr = file_read_size_int_array()
    start, end, max_subarray = kadane(arr)
    file_write([max_subarray])


if __name__ == '__main__':
    measure_performance(kadane_main)
```

Тесты и анализ

Для испытания данной программы были использованы аналогичные первому заданию тесты. Алгоритм всюду, на самых разных списках отработал быстро и без запинки, выдавая верный результат. Это ещё раз подчёркивает всю силу алгоритмов динамического программирования.

```python
import unittest
import random
from itertools import accumulate

from Lab2.task7.src.max_subarray_search_in_linear_time import *


class TestMaxSubarray(unittest.TestCase):
    def check_sort(self, testlist):  4 usages
        result, elapsed_time, peak_memory_megabytes = measure_performance(kadane, *args: testlist)
        self.assertEqual(result[2], max(accumulate(testlist, lambda x, y: max(y, x + y))))

    def test_should_empty(self):
        testlist = []
        self.assertRaises(ValueError, kadane, testlist)

    def test_should_random(self):
        testlist = [random.randint(-10 ** 9, 10 ** 9) for _ in range(random.randint(a: 1, b: 20000))]
        self.check_sort(testlist)
```

```python
    def test_should_max_size(self):
        testlist = [random.randint(-10 ** 9, 10 ** 9) for _ in range(20000)]
        self.check_sort(testlist)

    def test_should_max_size_reverse_sorted(self):
        testlist = [random.randint(-10 ** 9, 10 ** 9) for _ in range(20000)]
        testlist.sort(reverse=True)
        self.check_sort(testlist)

    def test_should_huge_numbers(self):
        testlist = [1000000000, 999999999, 999999998]
        self.check_sort(testlist)

    def test_should_not_list(self):
        self.assertRaises(TypeError, kadane, 2)


if __name__ == '__main__':
    unittest.main()
```

Заключение

В процессе выполнения лабораторной работы были разработаны и протестированы несколько алгоритмов сортировки и поиска, а также алгоритмов динамического программирования. Каждый из алгоритмов был подробно изучен, реализован на Python и проверен с помощью модуля unittest. Были проанализированы принципы их работы, выявлены преимущества и недостатки, а также выполнена оценка их временной сложности.

Выполнение этой лабораторной работы позволило углубить знания об основных алгоритмах сортировки и поиска, улучшить навыки программирования на Python, а также развить умение проводить модульное тестирование программ.