Analysis Report on the Implementation of Partner's Algorithms (Student B)

Author of the report: Dmitriy Belyaikin, Student A

Selection sort

Selection Sort works by repeatedly selecting the smallest element from the unsorted part of the array and putting it into the sorted part.

Danial's code

```java
package org.example.danial;

public class SelectionSort {

    public static void selectionSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
            int minIdx = i;
            boolean swapped = false;
            for (int j = i + 1; j < n; j++) {
                if (arr[j] < arr[minIdx]) {
                    minIdx = j;
                    swapped = true;
                }
            }
            if (!swapped) {
                break;
            }
            int temp = arr[i];
            arr[i] = arr[minIdx];
            arr[minIdx] = temp;
        }
    }
}
```

Best case for this sorting algorithm is when the input array is already sorted, that would be O(1). Worst case is when the array is sorted the other way around, O(n^2). Average case is Θ(n^2).

Space complexity is O(1)

Heap Sort

Heap Sort is a sorting algorithm that uses a binary heap data structure (in our case maxheap).

```java
public class HeapSort {
    public static void heapSort(int[] arr) {
        int n = arr.length;

        for (int i = n / 2 - 1; i >= 0; i--) {
            siftDown(arr, i, n);
        }

        for (int i = n - 1; i > 0; i--) {
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;
            siftDown(arr, 0, i);
        }
    }

    private static void siftDown(int[] arr, int i, int n) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < n && arr[left] > arr[largest]) {
            largest = left;
        }
        if (right < n && arr[right] > arr[largest]) {
            largest = right;
        }
        if (largest != i) {
            int temp = arr[i];
            arr[i] = arr[largest];
            arr[largest] = temp;
            siftDown(arr, largest, n);
        }
    }
}
```

Each siftDown takes O(log n), but not all nodes sift down fully, so the total cost is O(n).

The sorting phase is O(n log n).

The space complexity is O(1).

Kadane’s Algorithm

Kadane’s algorithm finds the contiguous subarray within an array of numbers that has the largest possible sum.

```java
package org.example.danial;

public class Kadane {
    public static class Result {
        public int maxSum;
        public int start;
        public int end;
        public Result(int maxSum, int start, int end) {
            this.maxSum = maxSum;
            this.start = start;
            this.end = end;
        }
    }
    public static Result kadane(int[] arr) {
        int n = arr.length;
        Result res = new Result(arr[0], 0, 0);
        int currSum = arr[0];
        int tempStart = 0;
        for (int i = 1; i < n; i++) {
            if (currSum < 0) {
                currSum = arr[i];
                tempStart = i;
            } else {
                currSum += arr[i];
            }
            if (currSum > res.maxSum) {
                res.maxSum = currSum;
                res.start = tempStart;
                res.end = i;
            }
        }
        return res;
    }
}
```

Time complexity is O(n)

Space complexity is O(1)

Max-Heap

```java
public class MaxHeap {
    private ArrayList<Integer> heap;
    private int size;

    public MaxHeap() {
        heap = new ArrayList<>();
        size = 0;
    }

    public void insert(int key) {
        heap.add(key);
        siftUp(size);
        size++;
    }

    public void increaseKey(int index, int newKey) {
        if (index >= size || newKey < heap.get(index)) {
            return;
        }
        heap.set(index, newKey);
        siftUp(index);
    }

    public int extractMax() {
        if (size == 0) {
            throw new IllegalStateException("Куча пуста");
        }
        int max = heap.get(0);
        heap.set(0, heap.get(size - 1));
        heap.remove(size - 1);
        size--;
        if (size > 0) {
            siftDown(0);
        }
        return max;
    }

    private void siftUp(int i) {
        while (i > 0 && heap.get((i - 1) / 2) < heap.get(i)) {
            int temp = heap.get(i);
            heap.set(i, heap.get((i - 1) / 2));
            heap.set((i - 1) / 2, temp);
            i = (i - 1) / 2;
        }
    }

    private void siftDown(int i) {
        int maxIdx = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < size && heap.get(left) > heap.get(maxIdx)) {
            maxIdx = left;
        }
        if (right < size && heap.get(right) > heap.get(maxIdx)) {
            maxIdx = right;
        }
        if (maxIdx != i) {
            int temp = heap.get(i);
            heap.set(i, heap.get(maxIdx));
            heap.set(maxIdx, temp);
            siftDown(maxIdx);
        }
    }
}
```

Time complexity for insert() is O(n log n), because of siftUp()

increaseKey – O(log n), extractMax() - O(n log n) because of siftDown(),

SiftUp and Down are O(log n)

Best case for this algorithm is O(1), if there are no swaps needed. Worst and average case is O(log n)

SiftDown is recursive, which can cause StackOverflow, iterative version is safer.

Benchmarking

```
=== Тестирование сортировки выбором ===
Отсортированный массив (n=1000): 10.6408 мс
Корректность: true
Обратный порядок (n=1000): 2.0744 мс
Корректность: true
Случайный массив (n=1000): 0.2486 мс
Корректность: false
Пустой массив: 3.0E-4 мс
Корректность: true
Массив с одним элементом: 2.0E-4 мс
Корректность: true

=== Тестирование пирамидальной сортировки ===
Отсортированный массив (n=1000): 11.0221 мс
Корректность: true
Обратный порядок (n=1000): 0.1466 мс
Корректность: true
Случайный массив (n=1000): 0.1691 мс
Корректность: true
Пустой массив: 5.0E-4 мс
Корректность: true
Массив с одним элементом: 5.0E-4 мс
Корректность: true

=== Тестирование Max-Heap ===
Вставка и извлечение 1000 случайных элементов: 2.9987 мс
Корректность: true
Increase-key: 0.0067 мс
Корректность: true
Пустая куча: Ошибка выброшена корректно

=== Тестирование алгоритма Кадане ===
Случайный массив (n=1000): 25.6015 мс
Корректность: true
Все отрицательные (n=1000): 0.0199 мс
Корректность: true
Массив с одним элементом: 0.001 мс
Корректность: true
Пустой массив: Ошибка выброшена корректно
```