AARHUS UNIVERSITET
INGENIØRHØJSKOLEN

DOA

Hand-in 1

Weeks 1-2: Lists, ADTs and complexity

Group members:

Studienummer	Navn	Studieretning
201708558	Mads Søborg	SW
202007814	Martin Dietz Vad	SW
201809299	Thea Knudsen	SW

Exercise 1:

A:

```cpp
void generateRandomMIntegers(int M, int N);

#include <cstdlib>
#include <ctime>
#include <vector>
#include <iostream>

// Exercise 1
// A

int main(void)
{
    int M = 10000;
    int N = 5000;

    generateRandomMIntegers(M, N);

    return 0;
}

void generateRandomMIntegers(int M, int N)
{
    srand(time(NULL));
    int count = 0;

    std::vector<int> randomMIntegers;
    std::vector<int>::iterator it;

    for (int i = 0; i < M; i++)
    {
        // Add random integer to vector between 0 and 10000
        randomMIntegers.push_back(rand() % 10000);
    }

    for (int i = 0; i < N; i++)
    {
        for (it = randomMIntegers.begin(); it != randomMIntegers.end(); it++)
        {
            // Check if random integer is in vector of random integers from 0 to 10000
            if (*it == rand() % 10000)
            {
                count++;
            }
        }
    }
    std::cout << "Count: " << count << std::endl;

    return;
}
```

Figure 1: implementation of the generateRandomIntegers function

B:

Time complexity:

The implementation takes in two parameters, M and N, and it has a double nested loop where it first loops over N and then loops over M. Therefore, the worst-case time complexity of the algorithm is to be considered $O(M \cdot N)$.

Space complexity:

Due to the use of a Vector the space complexity must be considered O(M) as it grows with the input.

Exercise 2:

The time complexity of the algorithm is $O(N^3)$ Because of the initial three nested for loops. There is another for loop which is $N \cdot N$ which is equal to N^2, but this can be ignored due to the fact N^2 is smaller than N^3. This is since N^2 would provide an insignificant contribution in comparison to N^3 if N is large enough, which is the point of approach for worst case analysis of algorithms.

Exercise 3:

A:

1:

There is a single for loop which loops N times. Therefor the time complexity is $O(N)$.

2:

There are two nested for loops which loops N^2times. Therefor the time complexity is $O(N^2)$

3:

There are two nested for loops but the second loop loops $N \cdot N$ times. Therefor the time complexity is $O(N^3)$.

4:

There are two nested for loops where the outer loop has a complexity of $O(N)$ and the inner has a complexity of $O(N^2)$, therefor we must multiply the complexity together giving us a total of $O(N^3)$.

B:

```c
int main(void)
{

    int (*functionPointers[4])(int) = {A1, A2, A3, A4};

    int nValues[] = {100,
                     200,
                     300,
                     400,
                     500,
                     600,
                     700,
                     800,
                     900,
                     1000};

    for (int i = 0; i < 4; i++)
    {

        for (int j = 0; j < 10; j++)
        {
            int n = nValues[j];
            printf("n: %d\n", n);

            clock_t begin = clock();

            int sum = functionPointers[i](n);

            clock_t end = clock();

            long double time_spent = (long double)(end - begin) / CLOCKS_PER_SEC;
            printf("Time spent on A%d: %Lf seconds\n", i + 1, time_spent);
        }
    }

    return 0;
}
```

Figure 2: Main function for timing the functions

```
❯ ./a.out
n: 100
Time spent on A1: 0.000002 seconds
n: 200
Time spent on A1: 0.000002 seconds
n: 300
Time spent on A1: 0.000001 seconds
n: 400
Time spent on A1: 0.000002 seconds
n: 500
Time spent on A1: 0.000002 seconds
n: 600
Time spent on A1: 0.000002 seconds
n: 700
Time spent on A1: 0.000003 seconds
n: 800
Time spent on A1: 0.000002 seconds
n: 900
Time spent on A1: 0.000003 seconds
n: 1000
Time spent on A1: 0.000002 seconds
n: 100
Time spent on A2: 0.000025 seconds
n: 200
Time spent on A2: 0.000098 seconds
n: 300
Time spent on A2: 0.000210 seconds
n: 400
Time spent on A2: 0.000345 seconds
n: 500
Time spent on A2: 0.000498 seconds
n: 600
Time spent on A2: 0.000700 seconds
n: 700
Time spent on A2: 0.000943 seconds
n: 800
Time spent on A2: 0.001223 seconds
n: 900
Time spent on A2: 0.001391 seconds
n: 1000
Time spent on A2: 0.001636 seconds
n: 100
Time spent on A3: 0.002299 seconds
n: 200
Time spent on A3: 0.021638 seconds
n: 300
Time spent on A3: 0.071905 seconds
n: 400
Time spent on A3: 0.176138 seconds
n: 500
Time spent on A3: 0.348788 seconds
n: 600
Time spent on A3: 0.610352 seconds
n: 700
Time spent on A3: 0.977092 seconds
n: 800
Time spent on A3: 1.458027 seconds
n: 900
Time spent on A3: 2.078741 seconds
n: 1000
Time spent on A3: 2.869889 seconds
n: 100
Time spent on A4: 0.000010 seconds
n: 200
Time spent on A4: 0.000036 seconds
n: 300
Time spent on A4: 0.000079 seconds
n: 400
Time spent on A4: 0.000134 seconds
n: 500
Time spent on A4: 0.000212 seconds
n: 600
Time spent on A4: 0.000298 seconds
n: 700
Time spent on A4: 0.000408 seconds
n: 800
Time spent on A4: 0.000555 seconds
n: 900
Time spent on A4: 0.000728 seconds
n: 1000
Time spent on A4: 0.000872 seconds
```

Figure 3: Terminal output

As you can see on figure 3 multiple running times in seconds are provided for ten different values of N. The running time measured is behaving as to be expected except for A1 which runs so fast that its very hard to time the actual running time of the function with N values this low. Higher values of N would have to be supplied for N but that would not be fitting for the other functions due to their different growth pattern.

C:

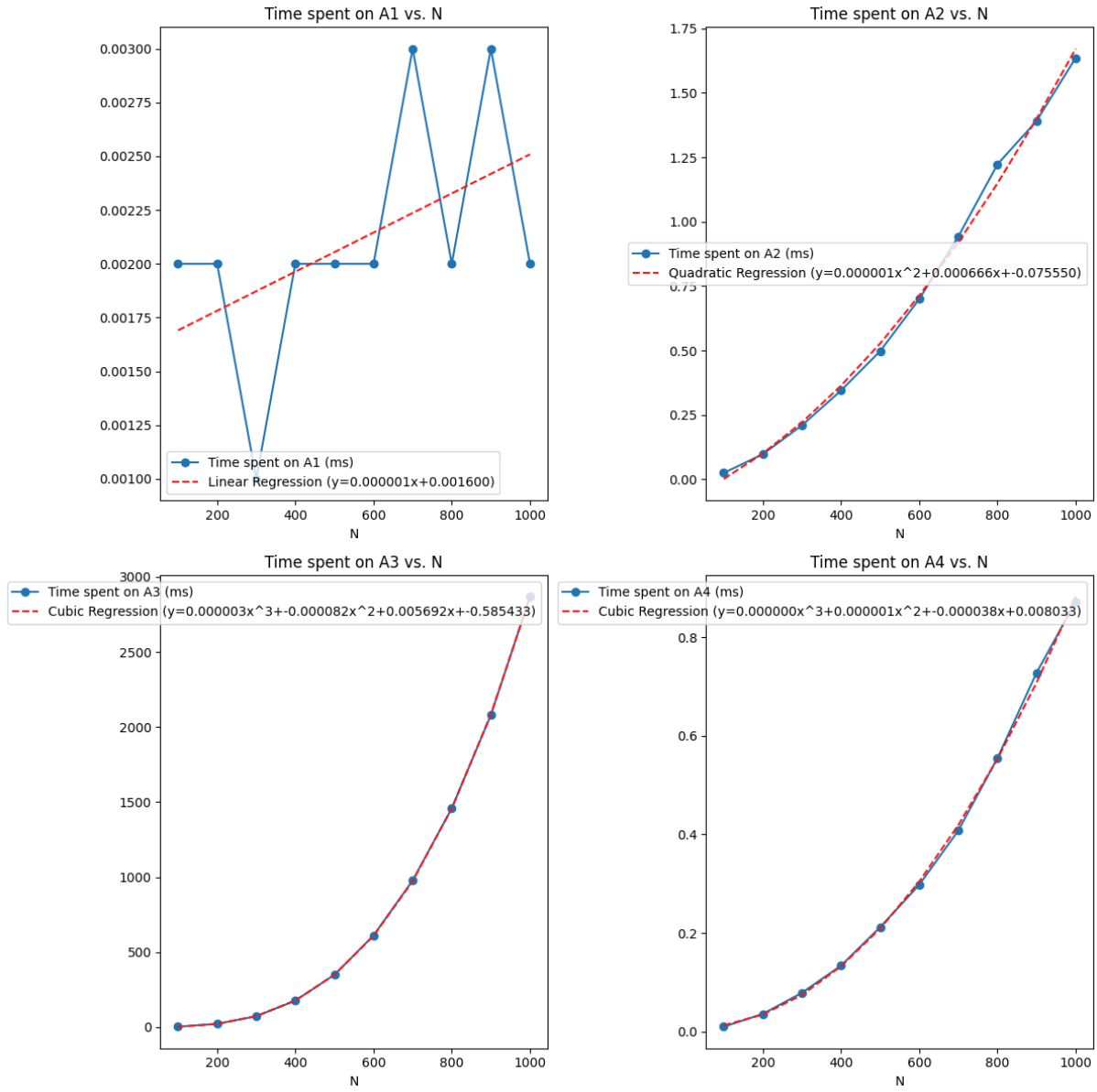

Figure 4: Python plot showing time spent vs N.

Based on the above plot we are unable to prove that A1 grows as a linear function due to N being too small. However, A2 is showing what would appear to be quadric regression which is proven by the trendline being very much in line with the dataset. The same is true A3 where the cubic regression being very similar to the cubic regression, the same is true with A4. We can therefor conclude that our initial analysis very much is in line with the actual runtime behaviour of the algorithm except for A1.

D:

No, the time complexity analysis should remain the same, however with compiler optimizations the comparison with current value of N is not possible, since the runtime speed is so massive that we are unable to get accurate timings for the runtime for the function. This behaviour is seen in figure 5. We would have to increase N in very big incremental amount to get accurate readings again. With this change we should revert to a state we are able to accurately graph out time complexity again.

```
n: 100
Time spent on A1: 0.000003 seconds
n: 200
Time spent on A1: 0.000001 seconds
n: 300
Time spent on A1: 0.000015 seconds
n: 400
Time spent on A1: 0.000000 seconds
n: 500
Time spent on A1: 0.000001 seconds
n: 600
Time spent on A1: 0.000000 seconds
n: 700
Time spent on A1: 0.000001 seconds
n: 800
Time spent on A1: 0.000002 seconds
n: 900
Time spent on A1: 0.000000 seconds
n: 1000
Time spent on A1: 0.000001 seconds
n: 100
Time spent on A2: 0.000002 seconds
n: 200
Time spent on A2: 0.000001 seconds
n: 300
Time spent on A2: 0.000001 seconds
n: 400
Time spent on A2: 0.000000 seconds
n: 500
Time spent on A2: 0.000001 seconds
n: 600
Time spent on A2: 0.000001 seconds
n: 700
Time spent on A2: 0.000000 seconds
n: 800
Time spent on A2: 0.000001 seconds
n: 900
Time spent on A2: 0.000001 seconds
n: 1000
Time spent on A2: 0.000001 seconds
n: 100
Time spent on A3: 0.000002 seconds
n: 200
Time spent on A3: 0.000000 seconds
n: 300
Time spent on A3: 0.000001 seconds
n: 400
Time spent on A3: 0.000001 seconds
n: 500
Time spent on A3: 0.000000 seconds
n: 600
Time spent on A3: 0.000002 seconds
n: 700
Time spent on A3: 0.000001 seconds
n: 800
Time spent on A3: 0.000000 seconds
n: 900
Time spent on A3: 0.000000 seconds
n: 1000
Time spent on A3: 0.000002 seconds
n: 100
Time spent on A4: 0.000001 seconds
n: 200
Time spent on A4: 0.000001 seconds
n: 300
Time spent on A4: 0.000000 seconds
n: 400
Time spent on A4: 0.000001 seconds
n: 500
Time spent on A4: 0.000001 seconds
n: 600
Time spent on A4: 0.000000 seconds
n: 700
Time spent on A4: 0.000002 seconds
n: 800
Time spent on A4: 0.000001 seconds
n: 900
Time spent on A4: 0.000000 seconds
n: 1000
Time spent on A4: 0.000001 seconds
```

Figure 5: Console output with clang -O3 optimization flag

Exercise 4:

A:

```cpp
#include <vector>

class MaxHeap
{
public:
    // is the heap empty?
    virtual bool isEmpty() const = 0;
    // number of elements in the heap
    virtual int size() = 0;
    // add an element to the heap
    virtual void insert(const int x) = 0;
    // find the maximum element in the heap
    virtual const int findMax() const = 0;
    // delete and return the maximum element of the heap
    virtual int deleteMax() = 0;
};

class MaxHeapVector : public MaxHeap
{
    std::vector<int> heap;

public:
    bool isEmpty() const;
    int size();
    void insert(const int x);
    const int findMax() const;
    int deleteMax();
};
```

Figure 6: Header file

```cpp
bool MaxHeapVector::isEmpty() const
{
    if (heap.size() == 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}

int MaxHeapVector::size()
{
    return heap.size();
}

// Unoptimized simple way to insert elements into the heap
void MaxHeapVector::insert(const int x)
{
    // Create iterator to iterate through the heap
    std::vector<int>::iterator it;
    // Place find the highest element in the heap
    for (it = heap.begin(); it != heap.end(); it++)
    {
        if (*it < x)
        {
            heap.insert(it, x);
            break;
        }
    }

    // If the element is the lowest element in the heap, place it at the end of the heap
    if (it == heap.end())
    {
        heap.push_back(x);
    }
}

const int MaxHeapVector::findMax() const
{
    // Protective programming throw exception if heap is empty
    if (heap.size() == 0)
    {
        throw std::out_of_range("Heap is empty");
    }

    // Since we already sort the heap when we insert an element, the maximum element is at the start of the heap
    return heap[0];
}

int MaxHeapVector::deleteMax()
{
    // Protective programming throw exception if heap is empty
    if (heap.size() == 0)
    {
        throw std::out_of_range("Heap is empty");
    }

    // Find the maximum element in the heap
    int max = heap[0];
    // Delete the maximum element in the heap
    heap.erase(heap.begin());

    // iterate through the heap and swap the max element to the start of the heap
    std::vector<int>::iterator it;

    for (it = heap.begin(); it != heap.end(); it++)
    {
        if (*it > max)
        {
            heap.insert(heap.begin(), *it);
            heap.erase(it);
            break;
        }
    }

    return max;
}
```

Figure 7: Cpp implementation

B:

IsEmpty:

We are simply performing a size check, and finding out if it is equal to 0 or not. Checking the size of the vector is constant time, there for the time complexity is $O(1)$

Size:

Checking the size of the vector is constant time, there for the time complexity is $O(1)$

Insert:

Inserting for this implementation is constant time $O(N)$, because we are using the expensive function insert on the vector, which is suboptimal for this kind of implementation.

findMax:

The max element is positioned at index 0 of the heap, so the complexity is constant time $O(1)$.

deleteMax:

Deleting the maximum value is trivial, but due to the subpar implementation the entire vector must be iterated through in order to find the maximum value and replace it at the front, therefor the time complexity is to be considered $O(N)$

C:

```
#include "simple_linked_list.h"

class MaxHeap
{
public:
    // is the heap empty?
    virtual bool isEmpty() const = 0;
    // number of elements in the heap
    virtual int size() = 0;
    // add an element to the heap
    virtual void insert(const int x) = 0;
    // find the maximum element in the heap
    virtual const int findMax() const = 0;
    // delete and return the maximum element of the heap
    virtual int deleteMax() = 0;
};

class MaxHeapList : public MaxHeap
{
    LinkedList<int> heap;

public:
    bool isEmpty() const;
    int size();
    void insert(const int x);
    const int findMax() const;
    int deleteMax();

private:
    void heapifyDown(int index);
    void heapifyUp(int index);
};
```

Figure 8: Header file implementation

```cpp
#include "listMaxheap.h"
#include <stdexcept>
#include <algorithm>

bool MaxHeapList::isEmpty() const
{
    return heap.empty();
}

int MaxHeapList::size()
{
    return heap.size();
}

void MaxHeapList::insert(const int x)
{
    // Add the element to the end of the linked list.
    heap.push_back(x);
    heapifyDown(heap.size() - 1);
}

const int MaxHeapList::findMax() const
{
    if (!isEmpty())
    {
        // The maximum element is at the root.
        return heap.find_kth(0);
    }
    else
    {
        // Protective programming
        throw std::runtime_error("Heap is empty.");
    }
}

int MaxHeapList::deleteMax()
{
    if (!isEmpty())
    {
        int max = heap.find_kth(0);
        int lastElement = heap.pop_back();

        if (!isEmpty())
        {
            heap.push_front(lastElement);
            heapifyDown(0);
        }

        return max;
    }

    throw std::out_of_range("Heap is empty");
}

void MaxHeapList::heapifyDown(int index)
{
    int leftChild = 2 * index + 1;
    int rightChild = 2 * index + 2;
    int largest = index;

    if (leftChild < heap.size() && heap.find_kth(leftChild) > heap.find_kth(largest))
        largest = leftChild;

    if (rightChild < heap.size() && heap.find_kth(rightChild) > heap.find_kth(largest))
        largest = rightChild;

    if (largest != index)
    {
        std::swap(heap.find_kth(index), heap.find_kth(largest));
        heapifyDown(largest);
    }
}

void MaxHeapList::heapifyUp(int index)
{
    int parent = (index - 1) / 2;
    while (index > 0 && heap.find_kth(index) > heap.find_kth(parent))
    {
        std::swap(heap.find_kth(index), heap.find_kth(parent));
        index = parent;
        parent = (index - 1) / 2;
    }
}
```

Figure 9: Cpp implementation