# Exam: TDDD86

# Data Structures, Algorithms and Programming Paradigms

**2023-12-14 kl: 08-12**

On-call (jour): Ahmed Rezine (tel: 1938)

**Specific instructions for the computer exams:**

- In summary: you log in with your LiuID and your private password. You can only save files in the desktop. We might leave files for you in the read-only "given_files" folder. You will use the "student chat client", or "student client" to receive information during the exam, to ask questions and to submit your solution. More details in the "EXAM_README.pdf" under "given_files".

- Your "student client" should start automatically, if you close it and need to start it again, double click on the "fish icon" on your desktop.

- Submit one file with all your answers. We will only look at the last submitted file.

- You can access OpenDSA using chromium. The start page will list available links.

**General instructions:**

- You may answer in either English or Swedish.

- **Submit a single document (text file or pdf) with your answers. The document should only contain text** (for instance, no pictures, no drawings).

- The questions are formulated so that you can answer with any text editor (e.g., vim, emacs, gedit, etc) or an office program (e.g., Open/Libre Office).

- If in doubt about a question, write down your interpretation and assumptions.

- The exam is divided into two parts:
  - Part A with a maximum of 34 pts.
  - Part B with a maximum of 20 pts.

- Grading:
  - Grade 3 requires at least 20 pts exclusively from Part A.
  - Grade 4 requires grade 3 is secured and at least 8 pts from Part B.
  - Grade 5 requires grade 3 is secured and at least 12 pts from Part B.

# Part A

## Problem A.1: Asymptotic execution time (8 pts)

Consider the four methods `f1, f2, f3, f4` and the nine complexity classes (A)-(I) depicted below. Assume the manipulated arrays are large enough in the four methods. You can always assume the size n of the problem to be a power of 2.

```
// Size of the problem is n. You can assume n = 2^p for some integer p.
void f1(int a[], int n){
  for (int i=0; i < n - 1; i++){
    int s = i;
    for(int j = i+1; j < n; j++){
      if(a[j] < a[s]){
        s = j;
      }
    }
    int tmp = a[s];
    a[s] = a[i];
    a[i] = tmp;
}
```

```
// Size of the problem is n. You can assume n = 2^p for some integer p.
void f2(int a[], int n){
  for (int i = 0; i < n; i++){
    int j = i;
    int x = a[j];
    while(j >= 1 && a[j-1] > x){
      a[j] = a[j-1];
      j = j - 1;
    }
    a[j] = x;
}
```

```
// Size of the problem is n = v - u. You can assume n = 2^p for some integer p
int f3(int a[], int u, int v){
  if(u >= v){
    return 0;
  }
  if(u + 1 == v){
    return 1;
  }
  int m = u + (v-u)/2;
  int x = f3(a, u, m);
  int y = f3(a, m, v);
  return x + y;
}
```

```
// Size of the problem is n = v - u. You can assume n = 2^p for some integer p
int f4(int a[], int u, int v, int b){
  if(u >= v){
    return 0;
  }
  int m = u + (v-u)/2;
  if(a[m] == 0){
    return 0;
  }
  int z = 0;
  for(int i= u; i < v; i++){
    if(a[i] == b){
      z = z + 1;
    }
  }
  int x = f4(a, u, m, b + 1);
  int y = f4(a, m, v, b + 1);
  return x + y + z;
}
```

Complexity classes:

| | | |
|---|---|---|
| (A) $\Theta(1)$ | (D) $\Theta(n \log n)$ | (G) $\Theta(2^n)$ |
| (B) $\Theta(\log n)$ | (E) $\Theta(n^2)$ | (H) $\Theta(3^n)$ |
| (C) $\Theta(n)$ | (F) $\Theta(n^3)$ | (I) $\Theta(n!)$ |

1. For each one of the 4 methods above, give (without justification!) the complexity class among the classes (A-I) that best matches its asymptotic **worst-case** execution time. (4pts).

2. For each one of the 4 methods above, give (without justification!) the complexity class among the classes (A-I) that best matches its asymptotic **best-case** execution time. (4pts).

## Problem A.2: Hashing and conflict resolution (8 pts)

Assume we use an array of size 7 with indices 0 to 6. The array is used to implement a hash table where the hash function (i.e., the hash value or the "home position" of each key) is given by the following:

| Key | Hash value |
|---|---|
| A | 1 |
| B | 2 |
| C | 6 |
| D | 4 |
| E | 5 |
| F | 6 |
| G | 1 |

Assume Linear probing is used (i.e., the probe function is $p(k, i) = i$ ).

3. Give the content of each cell of the table after inserting, starting from an empty table, the sequence F, E, D, C, B, A, G. (2pts).

4. Answer with yes or no (no need for justification). A wrong answer counts negative. Is there a sequence that results, starting from an empty table, in this table? (2pts if correct, 0 if not answered, -2pts if incorrect).

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Content | C | F | D | B | E | A | G |

5. Answer with yes or no (no need for justification). A wrong answer counts negative. Is there a sequence that results, starting from an empty table, in this table? (2pts if correct, 0 if not answered, -2pts if incorrect).

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Content | F | G | A | B | D | E | C |

6. Answer with yes or no (no need for justification). A wrong answer counts negative. Is there a sequence that results, starting from an empty table, in this table? (2pts if correct, 0 if not answered, -2pts if incorrect).

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Content | G | A | B | F | D | E | C |

## Problem A.3: Binary search trees and their traversal (8 pts)

Recall that binary trees can be represented sequentially. In this problem (i.e., problem A.3), we adopt the approach described in 8.3.1 in OpenDSA. For instance, the binary tree in Figure 1 can be represented using the sequence: "A B / D / / C E G / / / F H / / I / /". The symbol "/" is used to represent a "null" child. Do not draw your trees! Use this approach instead to answer the following questions.

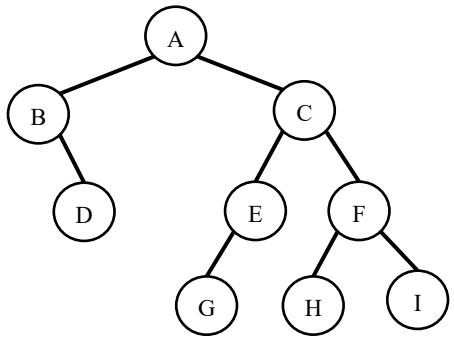


*Figure 1. Only relevant for Problem A.3*

Consider this sequence of 15 elements:

S: 10, 15, 11, 5, 1, 3, 4, 9, 13, 7, 14, 8, 12, 2, 6

7. Give a sequential representation (see the description of sequential representations of binary trees at the beginning of this problem) of the final binary search tree obtained by starting from an empty tree and inserting all the integers of the sequence S one after the other (i.e., first insert 10, then 15, then 11 …). **Do not try to balance the tree. Do not perform splay operations.** Call this tree $T_1$. (2pts).

8. List the integer values encountered in a post-order traversal of $T_1$. (2pts).

9. Give a sequential representation of the tree $T_2$ obtained by removing the value 11 from the tree $T_1$. Write your assumptions in case you make choices. (2pts).

10. Consider now the tree $T_1$ (observe the question is about the tree $T_1$ with 15 nodes obtained in question 7 above without any splay operations and without removing any node). Give the tree $T_3$ resulting from performing a splay operation on the node containing value 9 in $T_1$. (2pts).

### Problem A.4: Graphs (10pts)

Consider the directed graph in Figure 2.

11. Give a topological sort of the nodes of the directed graph from in Figure 2. (4pts).
12. List the letters in the order they are processed (i.e. marked as visited) in a **depth first traversal** that starts from node A in the graph of Figure 2. Observe the traversal needs not pass all nodes! Use the alphabetical ordering to break ties if any. (3pts)
13. List the letters in the order they are processed (i.e., marked as visited) in a **breadth first traversal** that starts from node A in the graph of Figure 2. Observe the traversal needs not pass all nodes! Use the alphabetical ordering to break ties if any. (3pts)

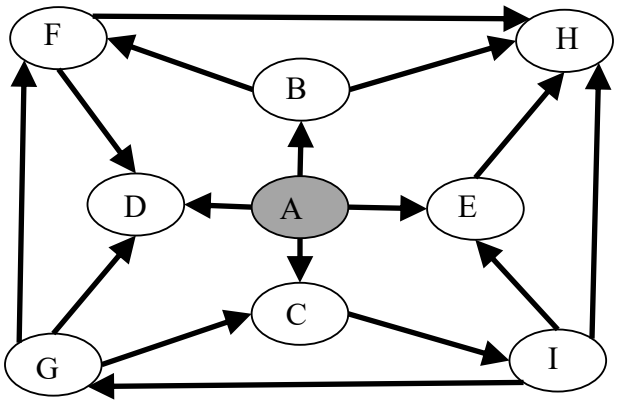


*Figure 2. Directed graph for questions 11, 12 and 13.*

# Part B:

### Problem B.1 (6pts)

Answer with yes or no (no need for justification). A wrong answer counts negative. All algorithms discussed in this problem take an array of size $n$ as input. The algorithms can

handle arrays of any size. We are interested in the asymptotic time complexity of the algorithms, i.e., in the asymptotic number of steps required as the size of the inputs increases. A step here is assumed to take a constant amount of time.

1. Assume the **worst-case** time complexity of an algorithm is in $O(n \log n)$. Does this exclude the existence of an infinite sequence of arrays $a_1, a_2, \ldots$ (where the size of each array $a_n$ is $n$) for which the algorithm always terminates in less than 10 steps? (1pts if correct, 0 if not answered, -1pts if incorrect).

2. Assume the **best-case** time complexity of an algorithm is in $O(n \log n)$. Does this exclude the possibility that the algorithm always terminates in less than 10 steps no matter the size of the input? (1pts if correct, 0 if not answered, -1pts if incorrect).

3. Assume the **best-case** time complexity of an algorithm is in $\Theta(n \log n)$. Does this exclude the existence of an infinite sequence of arrays $a_1, a_2, \ldots$ (where the size of each array $a_n$ is n) for which the algorithm always terminates in less than 10 steps? (1pts if correct, 0 if not answered, -1pts if incorrect).

4. Assume the **worst-case** time complexity of an algorithm is in $\Theta(n \log n)$. Does this exclude the possibility that the algorithm always terminates in less than 10 steps no matter the size of the input? (1pts if correct, 0 if not answered, -1pts if incorrect).

5. Assume the **worst-case** time complexity of an algorithm is in $\Omega(n \log n)$. Does this this exclude the existence of an infinite sequence of arrays $a_1, a_2, \ldots$ (where the size of each array $a_n$ is $n$) for which the algorithm always terminates in less than 10 steps? (1pts if correct, 0 if not answered, -1pts if incorrect).

6. Assume the **best-case** time complexity of an algorithm is in $\Omega(n \log n)$. Does this this exclude the existence of an infinite sequence of arrays $a_1, a_2, \ldots$ (where the size of each array $a_n$ is $n$) for which the algorithm always terminates in less than 10 steps? (1pts if correct, 0 if not answered, -1pts if incorrect).

## Problem B.2 (4 pts):

The code listed below is extracted from an template implementation of an extensible array called `ArrayList`. This data-structure allocates an array (referred to with the `T` pointer `m_elements`) of capacity `m_capacity` to store its elements. The number of elements currently stored is captured by the value of the variable `size`. As described in the listing, the implementation of the "`add`" method doubles the size of the allocated array each time it is about to insert a new element when the array is full (i.e., when `m_capacity` equals `size`).

7. Clearly show that the **<u>average number</u>** of assignments "`fresh_elements[i] = m_elements[i]`" (required to copy elements of previous arrays to fresh ones) when inserting $n$ elements in an empty `ArrayList` is in O(1). **`(4pts).`**

```
/*
 * An ArrayList is an ordered collection of elements stored and accessed
 * with 0-based integer indexes, using an array as the internal representation.
 */
template<typename T>
class ArrayList {
   public:

      ...

      // Appends the given value to the end of the list.
      void add(T value){
         if (m_size == m_capacity) {
         // out of space; resize
            if(m_capacity == 0){
               m_capacity = 1;
            }else{
               m_capacity *= 2;
            }
            int* fresh_elements = new int[m_capacity];
            for (int i = 0; i < m_size; i++) {
               fresh_elements[i] = m_elements[i];
            }
            delete[] m_elements;   // free old array's memory
            m_elements = fresh_elements;
         }
         m_elements[m_size] = value;
         m_size++;
      }


   private:
      int m_size = 0;                        // number of elements added
      int m_capacity = 0;                    // length of array
      T* m_elements = nullptr;               // array of elements

      ...

};
```

*Figure 3. Code for problem B.2*

## Problem B.3 (10 pts):

The Tower of Hanoi is a classical mathematical puzzle that is often used to introduce recursion in programming courses. The state of the puzzle can be captured using three stacks: “a”, “b” and “c”. Initially, the stack “a” contains an increasing sequence of integers $1, 2, \ldots n$ where n is the integer that was first pushed in “a” (hence 1 was the integer that was last pushed). Both “b” and “c” are initially empty. The goal of the puzzle is to move all integers from “a” to “b” while respecting the following rules:

- Each move results in one integer being moved from one stack to another.
- Each move from some stack “s” to another stack “t” consists in two steps: first, the integer “x” at the top of “s” is pushed into “t”. Then, “x” is popped from “s”.
- No integer may be pushed to a stack already containing smaller integers.

The code listed in Figure 4 describes a solution to the puzzle. Observe that `n`, `a`, `b` and `c` are global variables. (This is bad programming, but it simplifies the presentation here). The recursive method `solve` takes a number "m" of integers to be moved from a source stack "src" to a destination stack "dst" while using stack "via" as an intermediary stack. Observe that the stacks are passed by reference. Hence, the arguments to `solve` always refer to `a`, `b` and `c`, although in different orders.

8. Assume n=3. Give the contents of the three stacks "a", "b" and "c" before and after the move step in the if statement of solve (i.e., the two lines marked with "moving top of src to dst" and "finished moving top of src to dst". There will be 7 such moves (so you need to print 8 problem states). (3pts).

9. Let **solve_steps(m)** be the number of "top", "push", and "pop" instructions executed in the worst case by a call to `solve(m, src, dst, via)` when the argument `m` is smaller or equal to the number of elements in the stack src. State a recurrence definition of **solve_steps(m)** in terms of **solve_steps(m-1)**. Explain. (3pts).

10. Give a function **g** such that **solve_steps(m)** is in **Θ(g(m))**. Justify using induction. (4pts).

```
#include <iostream>
#include <stack>
using namespace std;
// State of Tower of Hanoi captured with 3 stacks.
stack<int> a,b,c;
int n = 3;

//Move m integers from src to dst using via as intermediary
void solve(int m, stack<int>& src, stack<int>& dst, stack<int>& via){
  if(m > 0){
    solve(m-1, src, via, dst);
    //moving top of src to dst
    dst.push(src.top());
    src.pop();
    //finished moving top of src to dst
    solve(m-1, via, dst, src);
  }
}

int main(){
  //Stack a contains sequence 1, ...n with 1 last pushed
  for(int i= n; i > 0; i--){
    a.push(i);
  }
  solve(n,a,b,c);
  return 0;
}
```

*Figure 1. Code for problem B.3*