PG4200 Algorithms and Data Structures

Exam

Question 1:

LO1: Understanding Data Structures

In a one-dimensional array, the address of any element can be calculated using the following formula (Goodrich & Tamassia, 2006, Ch.2.2):
Address(A[i]) = BaseAddress + (i – FirstIndex) x ElementSize

Algorithmic solution:
Input:

- BA = base address to the array
- FI = first index in the array
- i = wanted index
- s = size of each element in bytes

Output:

- Address to A[i]

Algorithmic steps *(adapted from Sedgewick & Wayne, 2011,* Ch.1.4)*.)*:

1. Start
2. Read BA
3. Read FI
4. Read i
5. Read s
6. Calculate Offsett ← (i – FI) x s
7. Calculate Address ← BA + Offset
8. Write out Address
9. Stop

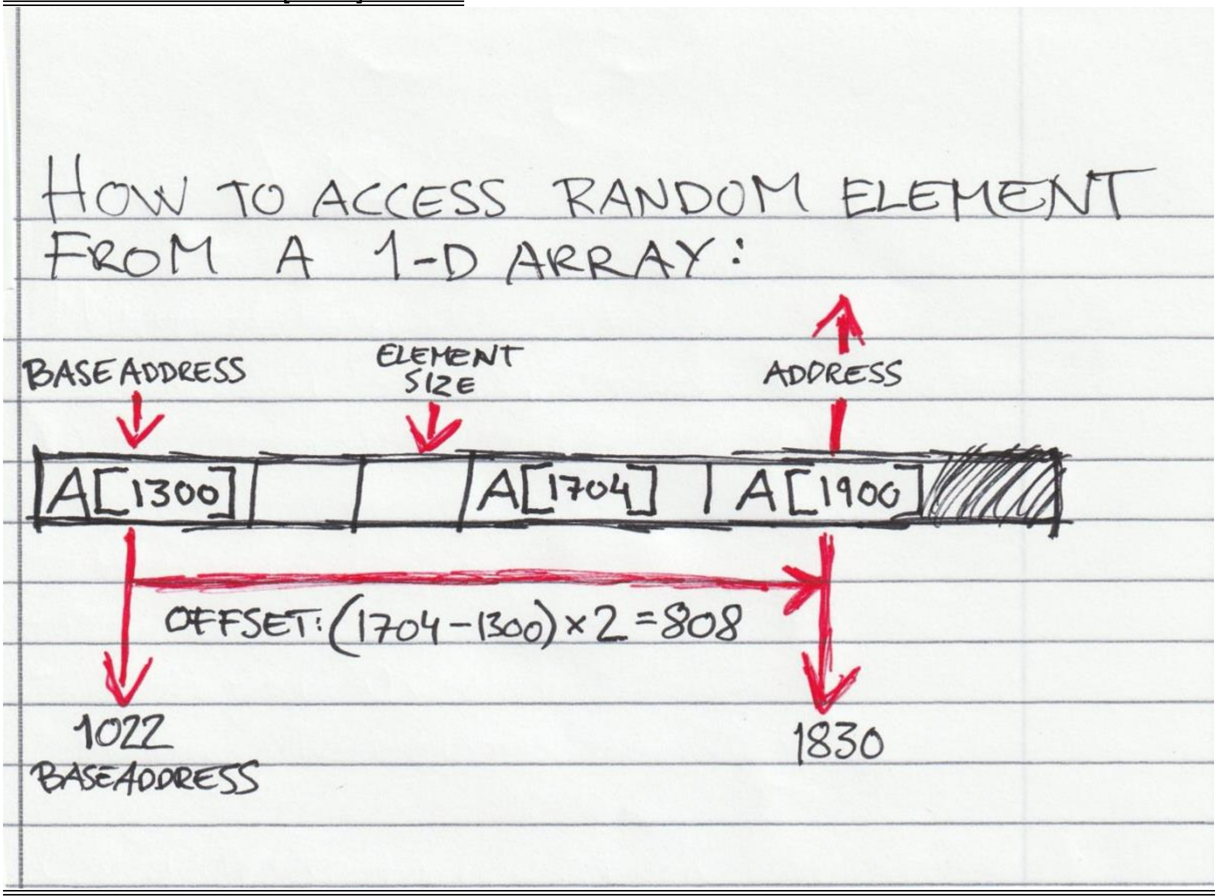
Our numbers:

- BA = 1022
- FI = 1300
- i = 1704
- s = 2

Calculation:

1. 1704 – 1300 = 404
2. 404 x 2 = 808
3. 1022 + 808 = 1830

Answer: Address to A[1740] is 1830



Question 2:

LO1: Stack(push/pop/getMin operations)

A stack is a linear data structure that follows the Last-in-First-Out (LIFO) principle (Goodrich & Tamassia, 2006, Ch.5.1). In this task we need a specialized stack that can return the minimun element in constant time.

This can be achieved by maintaining two stacks (Sedgewick & Wayne, 2011, Ch.1.3):
1. mainStack: Stores all pushed elements
2. minStack: Stores the current minimum after each push.

Algorithmic explanation:

We want a stack that supports:
- push(x) – Insert an element into the stack
- pop() – Remove and return the top element
- getMin() – Return minimum element in O(1) time

Algorithm steps (adapted from Gupta, n.d):
Push(x):
1. Push x into mainStack
2. If mainStack is empty or x <= top of mainStack, also push x into minStack

Pop():
1. If mainStack is empty → return -1
2. Pop the top value from mainStack
3. If this popped value equals the top of minStack, also pop from minStack
4. Return the popped value

getMin():
1. If minStack is empty → return -1
2. Return top of minStack

Complexity:
- push(x) → O(1)
- pop() → O(1)
- getMin → O(1)
- Space → O(n) (extra stack for minimum values)

Screenshot from IntelliJ showing the Java implementation of the mainStack and minStack:

```java
package org.example.pg4200.question2;

import java.util.Stack;

3 usages
public class MinStack {
    4 usages
    private Stack<Integer> mainStack;
    8 usages
    private Stack<Integer> minStack;

    1 usage
    public MinStack() {
        mainStack = new Stack<>();
        minStack = new Stack<>();
    }

    3 usages
    public void push(int x) {
        mainStack.push(x);
        // If minStack is empty or new element is smaller/equal, push it
        if (minStack.isEmpty() || x <= minStack.peek()) {
            minStack.push(x);
        }
    }

    2 usages
    public int pop() {
        if (mainStack.isEmpty()) {
            return -1; // stack is empty
        }
        int val = mainStack.pop();
        if (val == minStack.peek()) {
            minStack.pop();
        }
        return val;
    }

    2 usages
    public int getMin() {
        if (minStack.isEmpty()) {
            return -1; // stack is empty
        }
        return minStack.peek();
    }
}
```

This screenshot shows an example of how the code works:

```java
package org.example.pg4200.question2;

public class StackTest {
    public static void main(String[] args) {
        MinStack stack = new MinStack();

        stack.push( x: 5);    // main: [5], min: [5]
        stack.push( x: 3);    // main: [5, 3], min: [5, 3]
        stack.push( x: 7);    // main: [5, 3, 7], min: [5, 7]

        System.out.println(stack.getMin()); // Output: 3

        stack.pop();    // removes 7 -> main: [5, 3], min [5, 3]
        stack.pop();    // removes 3 -> main: [5], min [5]

        System.out.println(stack.getMin()); // Output: 5
    }
}
```

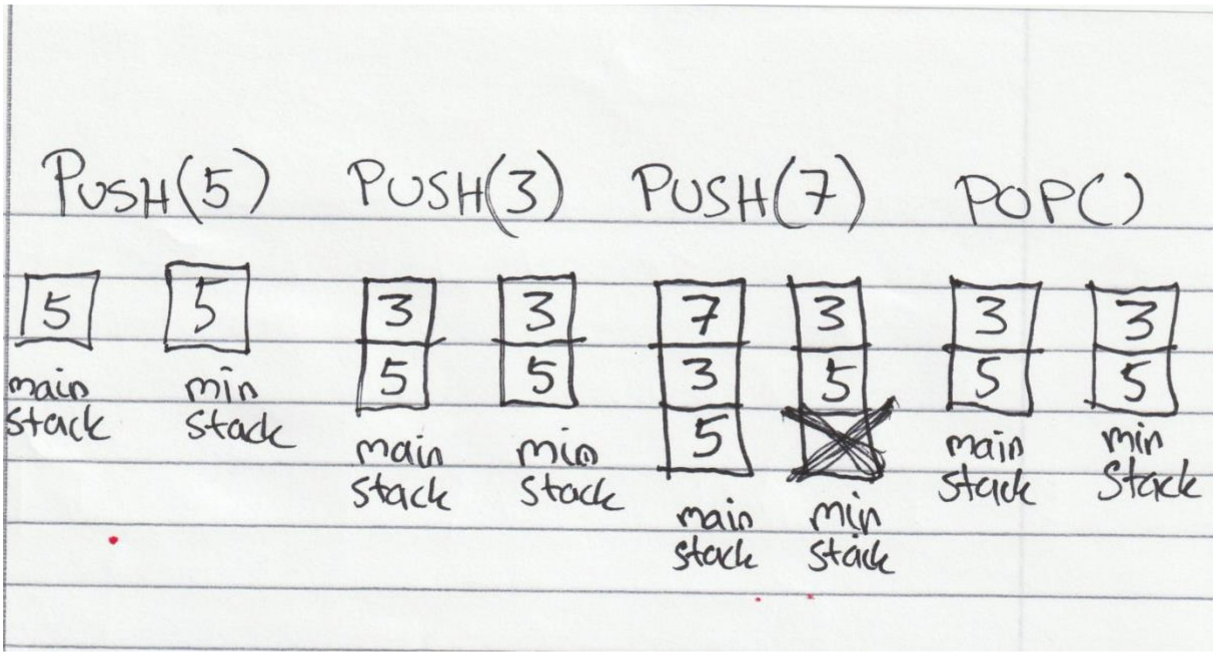
This screenshot shows the output:

```
12:40:39: Executing ':StackTest.main()'...

> Task :compileJava
> Task :processResources UP-TO-DATE
> Task :classes

> Task :StackTest.main()
3
5
```

Graphical illustration showing both stacks side-by-side at each step:



Question 3:

LO2: Searching Algorithms

Algorithm – Deleting a Node in a BST
There is three cases when deleting a node in a Binary Search Tree (Gupta, n.d):

1. Node is a leaf (No children → Simply remove it
2. Node has one child → Remove node, connect its parent to the child
3. Node has two children →
   - Find in-order successor(smalles value in right subtree) or in-order predecessor(largest value in left subtree)
   - Replace node´s value with successor/predecussor value
   - Delete the successor/predecessor from the subtree

Deleting node 60 in the given BTS (*algorithm adapted from Sedgewick & Wayne, 2011,* Ch.3.2*):*

1. Start at root 90 → go left because 60 < 90

2. Node 60 has two children
   - Left child: 25
   - Right child: 75

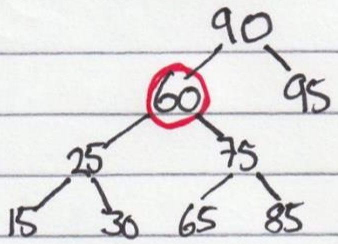
   We must find in-order successor: the smalles value in the right subtree

3. Find in-order successor – right subtree of 60 is:
```
      75
      / \
    65   85
```
   Smallest value = 65 (left in this subtree)

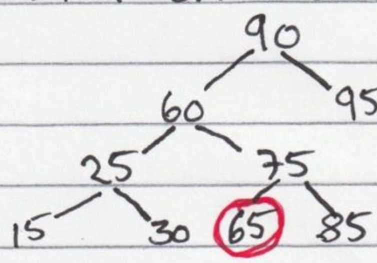
4. Replace 60 with 65
   Tree now looks like:
```
         90
         / \
      65    95
     /   \
   25     75
  / \     / \
15   30  65  85
```
5. Delete original 65 node
   Original 65 was a leaf node (no children) in the right subtree of 75. We romove it.
   Final BTS:
```
         90
         / \
      65    95
     /   \
   25    75
  / \      \
15   30     85
```

Step-by-step graphical solution:

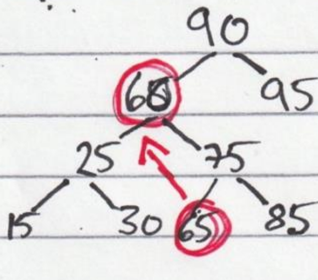
1. Initial tree:



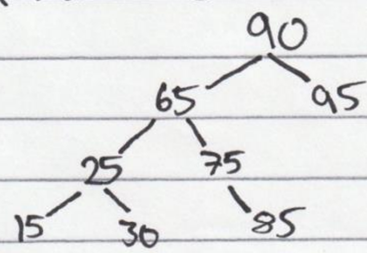
2. Find in-order successor:



3. Replace 60 with 65:



4. Delete leaf node 65 in right subtree, final BST:



Question 4:

LO3: Sorting Algorithms

Merge Sort is a divide-and-conquer algorithm. It works by repeatedly dividing the array into smaller subarrays until each contains only one element, then merging them back in sorted order (Sedgewick & Wayne, 2011, Ch.2.2; Goodrich & Tamassia, 2006, Ch.11.3).

Algorithm steps *(adapted from Sedgewick & Wayne, 2011):*

1. If the array has only one element, return (it is already sorted)

2. Otherwise:
    1. Find the middle of the array
    2. Split the array into two halves
        - Left half(U)
        - Right half(V)
    3. Recursively apply Merge Sort to both halves

3. Merge the two sorted halves:
    1. Create an empty array result
    2. Compare the first elements of both halves
    3. Append the smaller element to result and remove it from its half
    4. Repeat until one half is empty
    5. Append the remaining elements from the non-empty half to result

4. Return result

Merge Process Table, merging two arrays U and V into one array S
Initial values:

70, 50, 30, 10, 20, 40, 60

Step	U (Left halv)	V (Right halv)	S (Result)
1	70 50 30	10 20 40 60	10
2	70 50 30	10 20 40 60	10 20
3	70 50 30	10 20 40 60	10 20 30
4	70 50 30	10 20 40 60	10 20 30 40
5	70 50 30	10 20 40 60	10 20 30 40 50
6	70 50 30	10 20 40 60	10 20 30 40 50 60
7	70 50 30	10 20 40 60	**10 20 30 40 50 60 70 (Final values)**

Input: Positive integer n, array S indexed from 1 to n
For this case: S = {70, 50, 30, 10, 20, 40, 60}

Output: Array S containing the keys in non-decreasing order:
{10, 20, 30, 40, 50, 60, 70}

Merge Sort Algorithm:

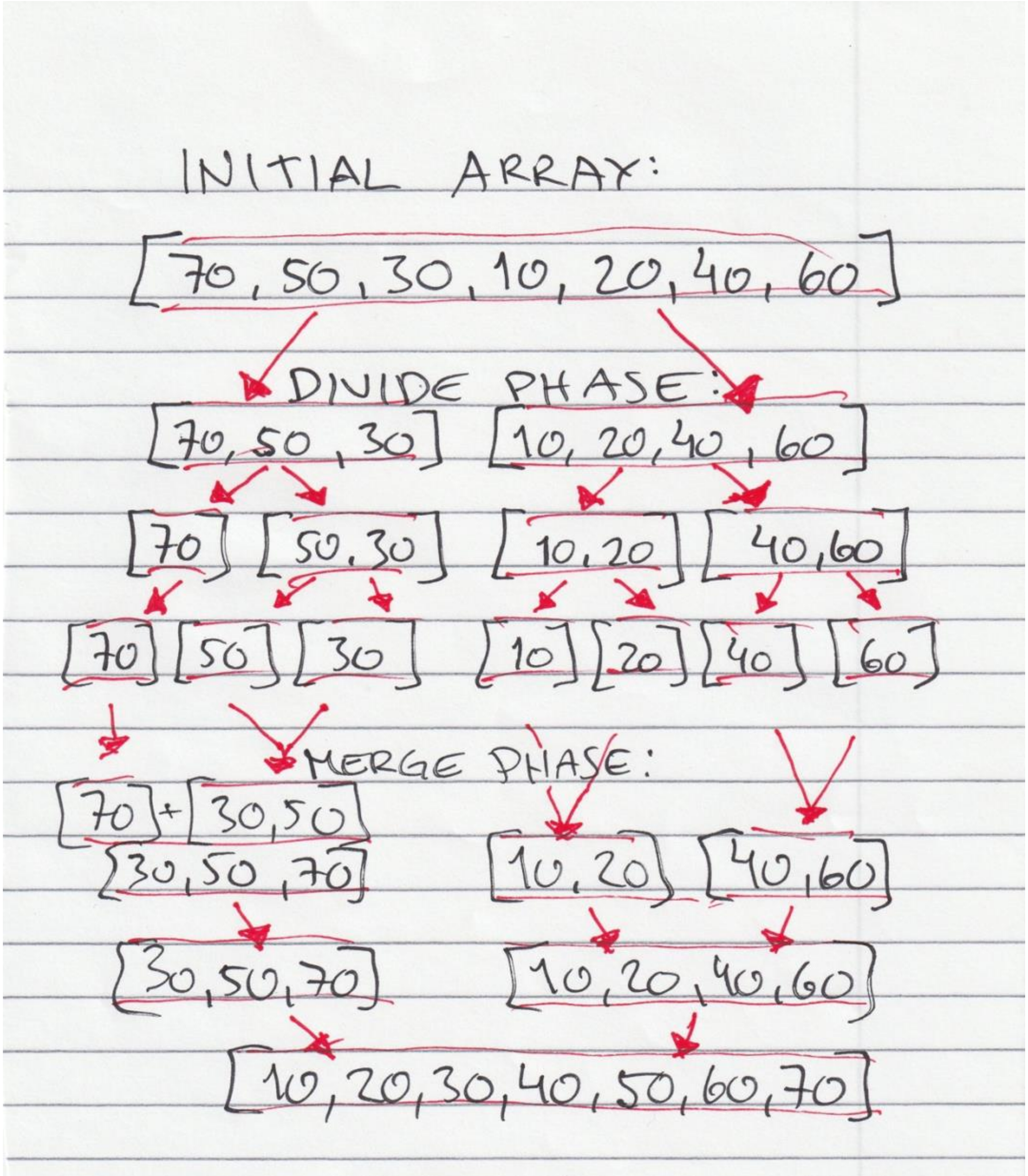
- Merges the two arrays U and V created by the recursive calls to mergesort
- Input size
    - h the number of items in U
    - m the number of items in V
- Basic operation:
    - Comparison of U[i] to V[i]

```
Void mergeSort{int n, keytype S[]) {
        If (n > 1) {
                const int h = [n/2];
                const int m = n-h;
                keytype U[1..h], V[1..m];

                copy S[1] through S[h] to U[1] through U[h];
                copy S[h+1] through S[n] to V[1] through V[m];

                mergeSort(h, U);
                mergeSort(m, V);
                merge(h, U, V, S);
                }
        }
```

Graphical representation of solving the given problem with Merge Sort:



Question 5:

LO4: Traversing Graphs Algorithms

The Fibonacci sequence is defined as:
F(0) = 0, F(1) =1
F(n) = f(n-1) + F(n-2) for n≥2
(Goodrich & Tamassia, 2006, Ch.4.3; Sedgewick & Wayne, 2011, Ch.2.3)

It can be computed in two common ways:

1. Recursive approach, directly applies the mathmatical recurrence relation.
2. Iterative approach, uses a loop to build the result from the bottom up.

Screenshot of the Fibonacci Recursive Algorithm from IntelliJ *(based on Sedgewick & Wayne, 2011, Ch.2.3):*

```java
package org.example.pg4200.question5;

public class FibonacciRecursive {
    3 usages
    public static int fib(int n) {
        if (n <= 1) {
            return n;
        }
        return fib( n: n-1) + fib( n: n-2);
    }

    public static void main(String[] args) {
        System.out.println("Recursive Fibonacci:");
        for (int i = 1; i <= 5; i++) {
            System.out.println("fib(" + i + ") = " + fib(i));
        }
    }
}
```

Screenshot of the output:

```
> Task :FibonacciRecursive.main()
Recursive Fibonacci:
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
```

Screenshot of the Fibonacci Iterative Algorithm from IntelliJ:

```java
package org.example.pg4200.question5;

public class FibonacciIterative {
    1 usage
    public static int fib(int n) {
        if (n <= 1) {
            return n;
        }
        int prev = 0, curr = 1;
        for (int i = 2; i <= n; i++) {
            int temp = curr;
            curr = curr + prev;
            prev = temp;
        }
        return curr;
    }

    public static void main(String[] args) {
        System.out.println("Iterative Fibonacci:");
        for (int i = 1; i <= 5; i++) {
            System.out.println("fib(" + i + ") = " + fib(i));
        }
    }
}
```

Screenshot of the output:

```
> Task :FibonacciIterative.main()
Iterative Fibonacci:
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
```

Efficiency justification

Recursive approach:

Logic: Breaks problem into smaller subproblems
F(n) = F(n-1) + F(n-2)

Negative: Recalculate the same Fibonacci values many times. For example:
fib(5) computes fib(3) twice, fib(2) three times, etc.

Time complexity: O(2^n), because each call generates two more calls, leading to exponential growth.

Space complexity: O(n), due to call stack repth in recursion.

Iterative approach:

Logic: Builds Fibonacci sequence from the bottom up using a loop.

Positive: Computes each Fobonacci number exactly once.

Time complexity: O(n), because it loops once per term.

Space complexity: O(1), because it only stores a few variables.

Experimental comparison for n = 1 → 5

Both produce the same output:
1, 1, 2, 3, 5

But:
Recursive approach:

- Makes many repeated calls even for small n
- Uses more memory because of call stack
- Gets much slower as n increases

Iterative approach:

- One simple loop
- No repeated computation
- Minimal memory use

Conclusion

The iterative algorithm is more efficien bacuse:

1. Time complexity: O(n) vs O(2^n)
2. Space complexity O(1) vs =(n)
3. No redundant calculations
4. Scales better for large n

Question 6:

LO5 and LO6: Computability and Complexity

1. *What is a Complexity Class? Why is it known as complexity classen in data structures?*

A complexity class is a group of computional problems that require similar amounts of time and/or memory to solve. They are called «complexity classes» because problems are organised into sets based on how difficult they are to compute (Gupta, n.d.; Goodrich & Tamassia, 2006, Ch.14; Sedgewick & Wayne, 2011, Ch.6).
In datastructures, these classes help us choose the right algorithms depending on efficiency needs.

Real-life example:

Sorting letters in a post office. If you have 10 letters, you can sort them quickly; with 10 000 letters, it takes longer. The sorting method you use (by postcode, aplhabetical, automated) changes the complexity class of the task.

2. Why do we use Big O Notation to analyse algorithm´s complexity in data structures?

We use Big O notation to describe the upper bound of an algorithm´s running time or memory use as the input size grows. It abstracts away hardware details and focuses on scalability (Sedgewick & Wayne, 2011, Ch.1; Goodrich & Tamassia, 2006, Ch.5).

Real-life example:

When loading a website, some designs scale poorly, adding more images and scripts can drastically slow it down. Big O helps predict hos performance changes with size, letting developers optimise the «algorithm» behind page rendering.

3. What is the P Class?

The P class contains problems that can be solved in polynomial time by a deterministic computer (Goupta, n.d.; Goodrich & Tamassia, 2006, Ch.14).
Features:

- Solvable in a reasonible time, even for large inputs.
- Deterministic, same input always gives the same output quickly.
- Considered tractable (practical to solve).

Real-life example:

Finding the fastest route on Google Maps is in P, even with thousands of roads, algorithms like Dijkstra´s can compute it in milliseconds. Sorting a playlist by song length is another P-class problem.

4. What is the NP class?

NP problems have solutions that can be verified in polynomial time, even if finding the solution may take much longer (Gupta, n.d.; Goodrich & Tamassia, 2006, Ch.14; Sedgewick & Wayne, 2011, Ch.6).
Features:

- Solution us easy to check, hard to find.
- Includes all the problems in P.
- Often requires brute-force search if no faster algorithm is known.

Real-life example:

Solving a jigsaw puzzle, you can quickly check if a finished puzzle is correct, but figuring out where every piece goes can be time-consuming. Another example is arranging studenst in an exam hall so no friends sit together, easy to check, hard to plan.

5. What is an NP-complete class?

NP-complete problems are the hardest problems in NP, they are both in NP and NP-hard (Gupta, n.d.; Goodrich & Tamassia, 2006, Ch.14; Sedgewick & Wayne, 2011, Ch.6).
Features:

- Any NP problem can be reduced to an NP-complete problem in polynomial time.
- If one NP-complete problem is solved in polynomial time, all NP problems an be solved in polynomial time.
- No known efficient algorithm exists to solve them exactly.

Real-life example:

The Travelling Salesman Problem, finding the shortest route visiting all delivery stops once is very hard to compute, but easy to check once a route is given. A sports tournament schedule that minimises travel time is another NP-complete problem.

Referances

Goodrich, M. T., & Tamassia, R. (2006). *Data Structures and Algorithms in Java* (4th ed.). Wiley.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.) Addison-Wesley.

Gupta, R. (n.d.). *LO 1: Understanding Data Structures* [Lecture slides]. Kristiania University College.

Gupta, R. (n.d.). *LO 2: Searching Algorithms* [Lecture slides]. Kristiania University College.

Gupta, R. (n.d.). *LO 3: Sorting Algorithms* [Lecture slides]. Kristiania University College.

Gupta, R. (n.d.). *LO 4: Traversing Graphs Algorithms* [Lecture slides]. Kristiania University College.

Gupta, R. (n.d) *LO 5 and 6: Computability and Complexity* (Lecture slides). Kristiania University College.