DSA: Solved Problems for the Theory Exam and Resits 2022

Katja Tuma, Fernanda Madeiral

September 13, 2024

Abstract

This document includes a detailed explanation of the solutions to the theory quiz for the examination. We include all the theory questions (including the questions of examination and resits). We ordered the questions chronologically as they were released in the examinations (newer are on top). All questions **were** of the form 'tick all answers which apply'. **From 2024 onwards, all questions include several statements that you are asked to read and analyze. Then, each question presents 'multiple choice' where only one choice is the correct answer. Each question is worth 1 point.**

Contents

1 Computing with Big-O Notation

Problem 1.1 *(Feb 1 2023) Given the function* `x` *below, what is the complexity expressed with* n*, in terms of Big-O notation? As usual, the complexity is given as the lowest (in terms of function growth) upper bound.*

```
function x(int n){
    for(int i=1 ; i<=n ; i++){
        for(int j=1 ; j<=n ; j++){
            for(int k=3 ; k<=n ; k++){ # NOT "j<=n"
                for(int l=1 ; l<=n ; l=l*2){
                    int h=4;
                }
            }
        }
    }
}
```

- ☐ $O(n)$
- ☐ $O(n^4 - 3)$
- ☐ $O(n^3)$
- ☐ $O(n^3 \cdot \log n)$

Problem 1.2 *(Jan 10 2023) Given the fragment below, what is the complexity expressed with* n*, in terms of Big-O notation? As usual, the complexity is given as the lowest (in terms of function growth) upper bound.*

```
for (int i = 0; i < n; i++) {
    if (i % 2 ==0) {
        for (int j = i; j < n; j++) { /*some functions of O(1)*/  }
    } else {
        for (int j = 0; j < i; j++) { /*some functions of O(1)*/ }
    }
}
```

- ☐ $O(n)$
- ☐ $O(n^3)$
- ☐ $O(n^2)$
- ☐ $O(logn)$

Problem 1.3 *(Jan 10 2023) Given the fragment below, which among the following choices have the tightest upper bound complexity expressed with* n*, in terms of Big-O notation?*

```
int Fibonacci(int n)
{
    if (n <= 1)
        return n;
    else
        return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

- ☐ $O(n^2)$
- ☐ $O(2^n)$
- ☐ $O(1^n)$
- ☐ $O(2^{1.6 \log n})$

Problem 1.4 *(Jan 10 2023) Read the fragment below and select the false statement(s) below. As usual, the complexity is given as the lowest (in terms of function growth) upper bound.*

```
void method1(int[] arr) {
       for (int i = arr.length() - 1; i >= 0; i = i - 3) {
             print(arr[i]);
       }
}
void method2(int[] arr) {
       for (int i = 0; i < arr.length(); i++) {
             for (int k = arr.length() - 1; k > 0; k = k/3) {
                   print(arr[i]);
             }
       }
}
void method3(int[] arr) {
       for (int i = 0; i < arr.length(); i++) {
             method1(arr);
             method2(arr);
       }
}
```

☐ *The complexity of method3 expressed with n (where n is the length of the array arr) in terms of Big-O notation is* $O(n^2 \cdot \log n)$.

☐ *The complexity of method3 expressed with n (where n is the length of the array arr) in terms of Big-O notation is* $O(n^3 \cdot \log n)$.

☐ *The complexity of method3 expressed with n (where n is the length of the array arr) in terms of Big-O notation is* $O(n^3)$.

☐ *The complexity of method3 expressed with n (where n is the length of the array arr) in terms of Big-O notation is* $O(n \cdot (n + n^2))$.

Problem 1.5 *(Oct 25 2022) What is the complexity (expressed with n, where n is the length of the array `arr`) of `method3` (in terms of Big-O notation)?*

```
void method1(int[] arr) {
       for (int i = arr.length() - 1; i >= 0; i = i - 3) {
             print(arr[i]);
       }
}
void method2(int[] arr) {
       for (int i = 0; i < arr.length(); i++) {
             for (int k = arr.length() - 1; k > 0; k = k/3) {
                   print(arr[i]);
             }
       }
}
void method3(int[] arr) {
       for (int i = 0; i < arr.length(); i++) {
             method1(arr);
             method2(arr);
       }
}
```

- ☐ $O(n^2 \cdot \log n)$
- ☐ $O(n^3 \cdot \log n)$
- ☐ $O(n^3)$
- ☐ $O(n \cdot (n + n^2))$

2 Sorting

Problem 2.1 *(Feb 1 2023) Look at the array below and consider how the array changes at each step of going through a merge-sort algorithm implementation, where the values of the array are being sorted in decreasing order (i.e., [3, 2, 1]). Consider the state of the array before and after the merges at the bottom of the recursion tree. What does the array look like after the merges in the next level in the recursion tree?*

Array:
`[9, -1, 5, -28, -7, -6, 3, 2]`

Before the first merge step:
`[9, -1, 5, -28, -7, -6, 3, 2]`

After the first merge step:
`[9, -1, 5, -28, -6, -7, 3, 2]`

- ☐ `[-1, 5, 9, -7, -6, 3, 2, -28]`
- ☐ `[-28, -1, -6, -7, 9, 2, 3]`
- ☐ `[-28, -1, 5, 9, -7, -6, 2, 3]`
- ☐ `[9, 5, -1, -28, 3, 2, -6, -7]`

Problem 2.2 *(Jan 10 2023) Look at the array below and consider how the array changes at each step of going through an in-place implementation of quick-sort (maintaining an invariant, as shown in class), where the values of the array are being sorted in increasing order (i.e. 1 2 3). The pivot is the first value in the array (so, array[0]). What does the array look like after the first step of partitioning the values around the pivot?*

Array:
`3 1 9 4 4 5 6 7`

- ☐ `3 1 9 4 4 5 6 7`
- ☐ `1 3 4 4 5 9 6 7`
- ☐ `1 3 4 4 5 6 7 9`
- ☐ `1 3 9 4 4 5 6 7`

Problem 2.3 *(Jan 10 2023) Look at the array below and consider how the array changes at each step of going through a merge-sort program, where the values of the array are being sorted in an increasing order (i.e. 1 2 3). Below you will see the state of the array before and after the merges in the bottom of the recursion tree. Find the state of the array after the merges in the next level in the recursion tree, then select the false statement(s) below.*

Array:
`9 -1 5 -28 -7 -6 3 2`

Before the first merge step:
`9 -1 5 -28 -7 -6 3 2`

After the first merge step:
`-1 9 -28 5 -7 -6 2 3`

☐ *After the merges in the next level in the recursion tree the state of the array is* `-28 -1 5 9 -7 -6 3 2`.

☐ *After the merges in the next level in the recursion tree the state of the array is* `-28 -1 5 9 -7 -6 2 3`.

☐ *After the merges in the next level in the recursion tree the state of the array is* `-28 -7 -6 -1 2 3 5 9`.

☐ *After the merges in the next level in the recursion tree the state of the array is* `-28 -7 -6 9 2 3 -1 9`.

Problem 2.4 *(Oct 25 2022) Look at the array below and consider how the array changes at each step of going through a merge-sort program, where the values of the array are being sorted in an increasing order (i.e. 1 2 3). Below you will see the state of the array before and after the merges in the bottom of the recursion tree. What does the array look like after the merges in the next level in the recursion tree?*

Array:
`9 -1 5 -28 -7 -6 3 2`

Before the first merge step:
`9 -1 5 -28 -7 -6 3 2`

After the first merge step:
`-1 9 -28 5 -7 -6 2 3`

☐ `-28 -1 5 9 -7 -6 3 2`

☐ `-28 -1 5 9 -7 -6 2 3`

☐ `-28 -7 -6 -1 2 3 5 9`

☐ `-28 -7 -6 9 2 3 -1 9`

3 Divide-and-Conquer and Dynamic Programming

Problem 3.1 *(Feb 1 2023) Which of the following statement(s) are false about divide-and-conquer and dynamic programming for a problem instance with n elements?*

- ☐ *Differently from most divide-and-conquer algorithms, the dynamic programming approach always recurses on subproblems of equal size.*
- ☐ *Both Merge sort and Quick sort are examples of dynamic programming when n is a power of 2.*
- ☐ *Quick sort is a divide-and-conquer algorithm.*
- ☐ *If $s(k)$ is the complexity of testing that a partial solution of k elements can be extended by adding a $k+1$th element, then the complexity of the dynamic programming algorithm is bounded by $O(n \cdot s(n))$.*

Problem 3.2 *(Jan 10 2023) Which of the following statement(s) are true about divide and conquer and dynamic programming for a problem instance with n elements?*

- ☐ *Differently from most divide and conquer algorithms, the dynamic programming approach always recurses on subproblems of equal size.*
- ☐ *Both Merge sort and Quick sort are examples of dynamic programming when n is a power of 2.*
- ☐ *Quick sort is a divide and conquer algorithm.*
- ☐ *If $s(k)$ is the complexity of testing that a partial solution of k elements can be extended by adding a $k+1$th element, then the complexity of the dynamic programming algorithm is bounded by $O(n \cdot s(n))$.*

Problem 3.3 *(Jan 10 2023) Which of the following statements are false about divide and conquer and dynamic programing?*

- ☐ *Merge sort is not a divide and conquer algorithm.*
- ☐ *Differently to dynamic programming, a divide and conquer approach usually recurses on subproblems of the same size.*
- ☐ *Every dynamic programming solution is also an instance of the divide and conquer approach.*
- ☐ *The complexity of the dynamic programming solution to the 0/1 knapsack problem is $O(2^n)$, where n is the number of items considered.*

Problem 3.4 *(Oct 25 2022) Which of the following statements are true about divide and conquer and dynamic programing?*

- ☐ *Merge sort is not a divide and conquer algorithm.*
- ☐ *Differently to dynamic programming, a divide and conquer approach usually recurses on subproblems of the same size.*
- ☐ *Every dynamic programming solution is also an instance of the divide and conquer approach.*
- ☐ *The complexity of the dynamic programming solution to the 0/1 knapsack problem is $O(2^n)$, where n is the number of items considered.*

4 Operations of data structures

Problem 4.1 *(Feb 1 2023) Select the statement(s) that are true about the running times (in terms of Big-O) of the operations for data structures. As usual, the complexity is given as the lowest (in terms of function growth) upper bound.*

- ☐ *The running time (in terms of Big-O) of the* `Insert` *operation of the heap is the same as the running time of the* `Insert` *operation of the balanced search tree, which is* $O(\log n)$.
- ☐ *The running time (in terms of Big-O) of the* `Insert` *operation for a sorted array is* $O(1)$.
- ☐ *The running time (in terms of Big-O) of the* `Delete` *operation for a sorted array is* $O(n)$.
- ☐ *The running time (in terms of Big-O) of the* `Insert` *operation for a balanced search tree is* $O(n^2)$ *and not* $O(\log n)$ *because after inserting a new item additional work is required to balance the tree again.*

Problem 4.2 *(Feb 1 2023) Which of the following patterns in a computer program suggests that a heap data structure could provide a significant speed-up?*

- ☐ *Repeated lookups.*
- ☐ *Repeated minimum computations.*
- ☐ *Repeated maximum computations.*
- ☐ *None of the other options.*

Problem 4.3 *(Jan 10 2023) What is the running time (in terms of Big-O) of the operations for the min-heap data structure?* ***This question was worth 1pt each in TestVision, therefore in total 5pt***

	$O(1)$	$O(n)$	$O(\log n)$	$O(n \cdot \log n)$
Insert	☐	☐	☐	☐
Extract min	☐	☐	☐	☐
Find min	☐	☐	☐	☐
Heapify	☐	☐	☐	☐
Delete	☐	☐	☐	☐

Problem 4.4 *(Jan 10 2023) Select the false statement(s) about the running time (in terms of Big-O) of the operations for the binary search tree data structure.* ***This question was worth 1pt each in TestVision, therefore in total 5pt***

	$O(1)$	$O(n)$	$O(\log n)$	$O(n \cdot \log n)$
Insert	☐	☐	☐	☐
Extract min	☐	☐	☐	☐
Find min	☐	☐	☐	☐
Predecessor	☐	☐	☐	☐
Delete?	☐	☐	☐	☐

Problem 4.5 *(Jan 10 2023) Select the statement(s) that are true about the running time (in terms of Big-O) of the operations for the sorted array data structure.*

- ☐ *The running time (in terms of Big-O) of the* `Search` *operation for sorted array is* $O(\log n)$.
- ☐ *The running time (in terms of Big-O) of the* `Insert` *operation for sorted array is* $O(\log n)$.
- ☐ *The running time (in terms of Big-O) of the* `Delete` *operation for sorted array is* $O(n)$.
- ☐ *The running time (in terms of Big-O) of the* `Predecessor` *operation for sorted array is* $O(n \cdot \log n^2)$.

Problem 4.6 *(Oct 25 2022) What is the running time (in terms of Big-O) of the operations complexity for the balanced search tree (BST) data structure?*

	$O(1)$	$O(n)$	$O(\log n)$	$O(n \cdot \log n)$
Insert	☐	☐	☐	☐
Extract min	☐	☐	☐	☐
Find min	☐	☐	☐	☐
Predecessor	☐	☐	☐	☐
Delete?	☐	☐	☐	☐

5 Graph Algorithms

5.1 Graph algorithms: topological sort, breadth-first search (BFS) and depth-first search (DFS)

Problem 5.1 *(Feb 1 2023) Select the statement(s) that are true about graph search algorithms.*

- ☐ *Node reachability problem (whether some node in a graph is reachable from the starting node) can be solved using either breadth-first or depth-first search.*
- ☐ *For a directed acyclic graph $G = (V, E)$, the topological ordering is an assignment $f(v)$ for every node $v \in V$ to a different number, such that for every $(v, w) \in E$, $f(v) \leq f(w)$.*
- ☐ *For extremely large graphs (e.g., more than 1 million nodes), it is more memory efficient to implement iterative depth-first search as opposed to recursive depth-first search.*
- ☐ *Every graph has a topological ordering.*

Problem 5.2 *(Jan 10 2023) Select the statement(s) that are false about graph search algorithms.*

- ☐ *Node reachability problem (whether some node in a graph is reachable from the starting node) can be solved using either breadth-first or depth-first search.*
- ☐ *For a directed acyclic graph $G = (V, E)$, the topological ordering is an assignment $f(v)$ for every node $v \in V$ to a different number, such that for every $(v, w) \in E$, $f(v) \leq f(w)$.*
- ☐ *For extremely large graphs (e.g., more than 1 million nodes) it is more memory efficient to implement iterative depth-first search as opposed to recursive depth-first search.*
- ☐ *Every graph has a topological ordering.*

Problem 5.3 *(Jan 10 2023) Select the statements that are false about graph search algorithms.*

- ☐ *Topological orderings can be computed for directed acyclic graphs (DAGs), but not every DAG has a topological ordering.*
- ☐ *Breadth-first search is usually implemented using a queue, while depth-first search is usually implemented using the stack data structure.*
- ☐ *Computing a topological ordering of a DAG is the most typical application of the breath-first search algorithm.*
- ☐ *For extremely large graphs (e.g., more than 1 million nodes), the recursive implementation of the depth-first search is going to use less memory compared to the iterative implementation, which has a memory overhead for each iteration.*

Problem 5.4 *(Oct 25 2022) Select the statements that are true about graph search algorithms.*

- ☐ *Topological orderings can be computed for directed acyclic graphs (DAGs), but not every DAG has a topological ordering.*
- ☐ *Breadth-first search is usually implemented using a queue, while depth-first search is usually implemented using the stack data structure.*
- ☐ *Computing a topological ordering of a DAG is the most typical application of the breath-first search algorithm.*
- ☐ *For extremely large graphs (e.g., more than 1 million nodes), the recursive implementation of the depth-first search is going to use less memory compared to the iterative implementation, which has a memory overhead for each iteration.*

5.2 Graph algorithms: A*

Problem 5.5 *(Feb 1 2023) Consider the pseudocode for the A* algorithm in Listing 2. The pseudocode is describing an implementation of A* that finds a solution to a single source, single target shortest path problem. It is using a queue data structure (X) to store the visited nodes and a heuristic function (h).*

Listing 1: Pseudocode for A*

```
1  Input: G = (V, E), s ∈ V, t ∈ V, len(e) ≥ 0 for each edge e ∈ E (i.e.,
       the edge costs are non-negative).
2  Postcondition: the shortest path from s to t.
3
4  \\pseudocode for A*
5  X = {s}
6  len(s) = 0; len(v) = ∞ (for all vertices)
7  while there is an edge (v, w), such that v ∈ X, w ∈ X do:
8       if (v ≠ t) then exit
9       (a, b) = minimize len(v) + h(w, t)
10      add b to X
11      len(b)=len(a) + h(b, t)
```

Select the statement(s) that are true about the pseudocode from Listing 2:

☐ *The length of node b is not calculated correctly (in line 11). It should be instead initialized with the maximum value possible for the most conservative estimate: '`len(b) = len(a)` $+\infty$'.*

☐ *The exit condition is incorrect (in line 8), because this way the while loop will exit whenever the current node is not equal to the target. It should check for equality as follows: '`if` $(v = t)$ `then exit`'.*

☐ *X is incorrectly initialized (in line 5). It should contain all the vertices in the graph otherwise the algorithm will not be able to visit all the nodes. The correct way to initialize X is as follows: '`X=`$\{$`all` $v \in V\}$'.*

☐ *The while loop (in line 7) is iterating over the wrong edges. It should iterate over the outgoing edges from v that have not been visited yet, therefore they are not included in X. The correct line 7 is: '`while there is an edge` (v, w)`, such that` $v \in X$, $w \notin X$ `do:`'.*

Problem 5.6 *(Jan 10 2023) Consider the pseudocode for the A* in Listing 2 and answer the question below.*
The algorithm is describing an implementation of A that finds a solution to a single source, single target shortest path problem. It is using a queue data structure (X) to store the visited nodes and a heuristic function (h).*

Listing 2: Pseudocode for A*

```
1  Input: G=(V,E), s ∈ V, t ∈ V, len(e) ≥ 0 for each edge e ∈ E (i.e.,
       the edge costs are non-negative).
2  Postcondition: the shortest path from s to t.
3
4  \\pseudocode for A*
5  X = {s}
6  len(s) = 0; len(v) = ∞ (for all vertices)
7  while there is an edge (v,w), such that v ∈ X, w ∉ X do:
8       if (v = t) then exit
9       (a,b) = minimize len(v) + h(w,t)
10      add b to X
11      len(b)=len(a)
```

Which of the following statement(s) about the pseudocode from Listing 2 are true:

- ☐ *The length of node b is not calculated correctly (in line 11) because it is not taking the heuristic estimate into account. It should be instead calculated as follows: len(b) = len(a) + h(b,t)*
- ☐ *The exit condition is incorrect (in line 8), because this way the while loop will always exit the first time it is executed. It should check for inequality as follows: if (v ≠ t) then exit*
- ☐ *X is incorrectly initialized (in line 5). It should contain all the vertices in the graph otherwise the algorithm will not be able to visit all the nodes. The correct way to initialize X is as follows: X={all v ∈ V}*
- ☐ *The value calculated for each outgoing edge from current node v is incorrectly calculated (in line 9). It should be calculated as the length of the current node v plus the heuristic estimate of how far the current node v is to the target (and not the next node w). Therefore, the following is correct: (a,b) = minimize len(v) + h(v,t)*

Problem 5.7 *(Feb 1 2023) Which of the statement(s) about the A* algorithm are false?*

- ☐ *Regardless of the chosen heuristic function, the A* algorithm will always outperform Dijkstra.*
- ☐ *The A* algorithm will never outperform the Dijkstra algorithm.*
- ☐ *A* is guaranteed to return an optimal solution if the input graph has a large number of nodes and very few edges (the graph is sparse).*
- ☐ *The running time of A* can (in some cases) be similar to the running time of Dijkstra.*

Problem 5.8 *(Jan 10 2023) Which of the statement(s) about the A* algorithm are true?*

- ☐ *Regardless of the chosen heuristic function, the A* algorithm will always outperform Dijkstra.*
- ☐ *The A* algorithm will never outperform the Dijkstra algorithm.*
- ☐ *A* is guaranteed to return an optimal solution if the input graph has a large number of nodes and very few edges (the graph is sparse).*
- ☐ *The running time of A* can (in some cases) be similar to the running time of Dijkstra.*

Problem 5.9 *(Jan 10 2023) Consider an implementation of the A* algorithm that uses a heap data structure to store the yet unprocessed nodes. The key used for the heap is the minimum combined cost of the Dijkstra score and the heuristic function. The heuristic function h(w,t), where w is the next node to be explored and t is the target node, is equal to 0. As usual, n denotes the number of nodes, while m denotes the number of edges in a directed, weighted graph with no negative weights.*
Which of the statement(s) about this A implementation are false?*

- ☐ *The complexity of the A* algorithm in case of the described implementation is $O(n)$.*
- ☐ *The complexity of the A* algorithm in case of the described implementation is $O(nm)$.*
- ☐ *The complexity of the A* algorithm in case of the described implementation is $O(nlog(n))$.*
- ☐ *The complexity of the A* algorithm in case of the described implementation is $O((n+m)log(n))$.*

Problem 5.10 *(Oct 25 2022) What is the complexity of the A* algorithm in case of the following implementation (in terms of Big-O notation)?*
Note that, n denotes the number of nodes, while m denotes the number of edges in a directed, weighted graph with no negative weights. The A implementation uses a heap data structure to store the yet unprocessed node. The key used for the heap is the minimum combined cost of the Dijkstra score and the heuristic function. The heuristic function $h(w,t)$, where w is the next node to be explored and t is the target node, is equal to 0.*

- ☐ $O(n)$
- ☐ $O(nm)$
- ☐ $O(nlog(n))$
- ☐ $O((n+m)log(n))$

Problem 5.11 *(Jan 10 2023) Which of the statements about the A* algorithm are true?*

- ☐ *A* algorithm can be implemented using different heuristic functions.*
- ☐ *The A* algorithm will always outperform the Dijkstra algorithm.*
- ☐ *A* is guaranteed to return an optimal solutions if the heuristic function is consistent.*
- ☐ *The source and target vertex must be known for this algorithm to work.*

Problem 5.12 *(Oct 25 2022) Which of the statements about the A* algorithm are false?*

- ☐ *A* algorithm can be implemented using different heuristic functions.*
- ☐ *The A* algorithm will always outperform the Dijkstra algorithm.*
- ☐ *A* is guaranteed to return an optimal solutions if the heuristic function is consistent.*
- ☐ *The source and target vertex must be known for this algorithm to work.*

5.3 Graph algorithms: Minimum Spanning Tree (MST)

Problem 5.13 *(Jan 10 2023) Which of the statement(s) about the minimum spanning tree problem are true?*

- ☐ *A minimum spanning tree for a connected graph has exactly V-2 edges, V being the number of vertices in the graph.*
- ☐ *The most straight forward implementation of Prim's algorithm (e.g., without the use of heap) performs better when the graph on the input is dense, while the most straight forward implementation of the Kruskal algorithm performs better when the input graph is sparse.*
- ☐ *Prim's and Kruskal's algorithms will never return the same minimum spanning tree.*
- ☐ *The complexity of Prim's algorithm implemented with the heap data structure and Kruskal's algorithm is the same. Both have the complexity of $O(m * n)$, where m is the number of edges and n is the number of nodes.*

Problem 5.14 *(Jan 10 2023) Which of the statements about the minimum spanning tree problem are false?*

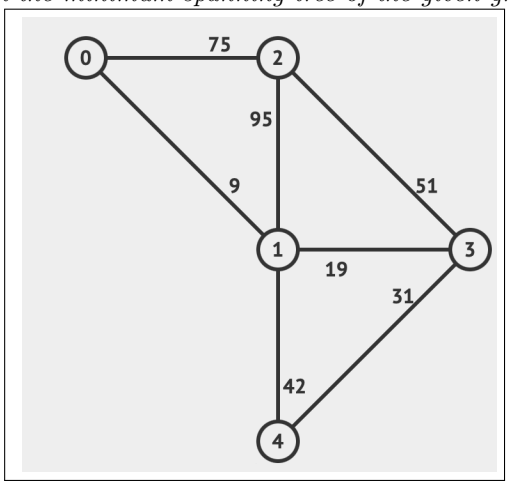
- ☐ *A minimum spanning tree for a connected graph has exactly V-1 edges, V being the number of vertices in the graph.*
- ☐ *A graph where every edge weight is unique (there are no two edges with the same weight) has a unique minimum spanning tree.*
- ☐ *Prim's and Kruskal's algorithms will always return the same minimum spanning tree.*
- ☐ *The minimum spanning tree can be used to find the shortest path between two vertices.*

Problem 5.15 *(Oct 25 2022) Which of the statements about the minimum spanning tree problem are true?*

- ☐ *A minimum spanning tree for a connected graph has exactly V-1 edges, V being the number of vertices in the graph.*
- ☐ *A graph where every edge weight is unique (there are no two edges with the same weight) has a unique minimum spanning tree.*
- ☐ *Prim's and Kruskal's algorithms will always return the same minimum spanning tree.*
- ☐ *The minimum spanning tree can be used to find the shortest path between two vertices.*

5.4 Operations on Graphs

Problem 5.16 *(Jan 10 2023) Consider the graph shown below. Which of the following are the edges in the minimum spanning tree of the given graph?*



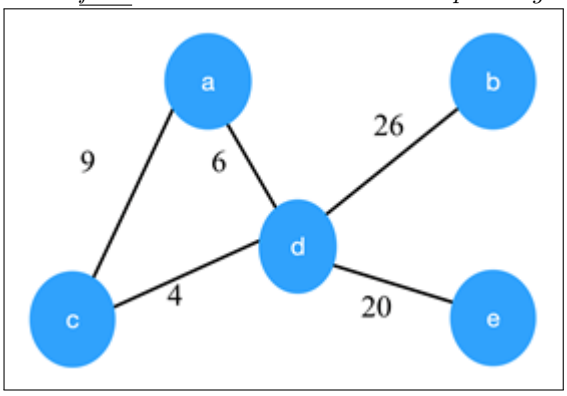
☐ $(0-1),(1-3),(2-3),(3-4)$

☐ $(0-1),(1-4),(1-2),(1-3)$

☐ $(0-1),(1-4),(2-3),(1-3)$

☐ $(0-1),(0-2),(1-4),(1-2)$

Problem 5.17 *(Jan 10 2023) Consider the graph shown below and select the statement(s) that are false about the minimum spanning tree of the given graph.*



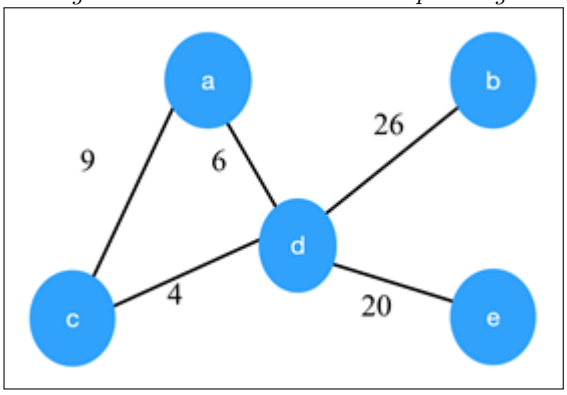
☐ *The following edges form the minimum spanning tree of the given graph:* $(a-c),(c-d),(d-b),(d-b)$.

☐ *The following edges form the minimum spanning tree of the given graph:* $(c-a),(a-d),(d-b),(d-e)$.

☐ *The following edges form the minimum spanning tree of the given graph:* $(a-d),(d-c),(d-b),(d-e)$.

☐ *The following edges form the minimum spanning tree of the given graph:* $(c-a),(a-d),(d-c),(d-b),(d-e)$.

Problem 5.18 *(Oct 25 2022) Consider the graph shown below. Which of the following are the edges in the minimum spanning tree of the given graph?*



☐ $(a-c),(c-d),(d-b),(d-b)$

☐ $(c-a),(a-d),(d-b),(d-e)$

☐ $(a-d),(d-c),(d-b),(d-e)$

☐ $(c-a),(a-d),(d-c),(d-b),(d-e)$

5.5 0/1 Knapsack problem

Problem 5.19 *(Feb 1 2023) Solve the following 0/1 knapsack problem (e.g., using the dynamic programming approach with the tabular method). The weights for the items are 3,4,5,6 and the profits for the items are 2,3,4,1, and the maximum weight the knapsack can hold is 8. Select the true statement(s) below:*

☐ *The solution to the problem is choosing items with weight 5 and 3 with the combined weight of 8 and combined profit of 6.*

☐ *The solution to the problem is choosing the first three items with the combined weight of 8 and combined profit of 9.*

☐ *The solution to the problem is choosing the first two items with the combined weight of 7 and combined profit of 5.*

☐ *The solution to the problem is choosing items with weight 4 and 5 with the combined weight of 8 and combined profit of 5.*

6 Solutions

6.1 Computing with Big-O Notation

Solution to 1.1 To answer this question we must analyze the number of operations that are executed during the four nested loops in the given function, which takes as parameter an integer n. We can break our analysis for each loop, where every time we consider how many times the body of that loop will be executed. Since we have nested for loops, we will use multiplication to combine the final result. We can choose to analyze this code fragment in a top-down approach (though the bottom-up would be an equally good choice).

The entire complexity of the four loops is:

$$t(n) \quad = \quad O\left(O(first) \cdot (O(second) \cdot (O(third) \cdot (O(fourth))))\right)$$

As we enter the function, we program will immediately enter the first for loop. The index i begins at 1, is increased upon every iteration by 1 and goes up to (including) n. Therefore, the entire body of the first loop will be executed n times. The index j is initiated also with 1, increased by 1 at every loop iteration, until it reaches the value of n. Therefore, the body of the second loop will be executed n times.

So far we have established:

$$t(n) \quad = \quad O\left(O(n) \cdot (O(n) \cdot (O(third) \cdot (O(fourth))))\right)$$

The index k of the third nested loop is initiated with 3, increased by 1 at every loop iteration, until it reaches the value of n. Despite the index skipping the first two integers this will still remain in the same complexity class in the big-O notation, where we suppress lower order terms and constants. Therefore, the body of the third loop will be also $O(n)$ despite being actually executed only $n-2$ times.

Finally, the index l of the fourth nested loop is initiated with 1, increased by $l*2$ at every loop iteration, until it reaches the value of n. Now the index does not grow linearly (by 1) anymore. We can observe a few values of l: $1, 1*2 = 2, 2*2 = 4, 4*2 = 8, 8*2 = 16 \ldots n$ and we realize that the index l grows *exponentially*. This means that the fourth loop will actually be executed much *less* than n times. It will be executed $\log_2 n$ times (which is the inverse of exponential growth). Since in big-O we suppress lower order terms and constants, we get $O(\log n)$ for the fourth loop, bringing the total complexity oft he code fragment to:

$$\begin{aligned} t(n) \quad &= \quad O\left(O(n) \cdot (O(n) \cdot (O(n) \cdot (O(\log n))))\right) \\ t(n) \quad &= \quad O\left(n^3 \cdot \log n\right) \end{aligned}$$

Thus, the following answer is correct:

☐ ~~$O(n)$~~
This is false because the complexity is in fact greater (in terms of function growth).

☐ ~~$O(n^4 - 3)$~~
This is false because we are looking for the lowers upper bound.

☐ ~~$O(n^3)$~~
This is false because the complexity is in fact greater (in terms of function growth).

☑ $O(n^3 \cdot \log n)$
As explained above, this is the true complexity expressed with the big-O notation.

Solution to 1.2 At first, we observe that we have to analyze two nested for loops, which will be executed depending on some conditional if statement. The outer for loop starts from $i = 0$ and goes till $i = n - 1$. The first inner for loop starts at $j = i$ and goes until $j = n - 1$. The second inner for loop starts at $j = 0$ and goes until $j = i - 1$. The indices i and j are in all three for loops incremented by 1 at each step of the loop. Then we can break down the complexity in a top-down fashion.

- The complexity of the outer for loop is $O(n)$ times the complexity of the conditional if statement. At this point, we observe that the conditional statement is going to switch the execution between the two inner for loops depending on whether index i is odd or even. When i is even, the condition is satisfied, so the first inner loop is executed. Else, the second inner loop is executed. Since we increment i by 1 at every step, we know that half the time the first loop is executed and the other half, the second loop is executed. Therefore, we have:

$$t_3(n) \quad = \quad O\left(n \cdot (\ 1/2 * t_1(n) + 1/2 * t_2(n))\right) = O\left(n \cdot (\ t_1(n) + t_2(n))\right)$$

- The complexity of the first inner loop is:

$$t_1(n) \quad = \quad O\left(n\right)$$

since j goes from i until n (where i will also go until n).

- The complexity of the second inner loop is:

$$t_1(n) \quad = \quad O\left(n\right)$$

since j goes from 0 until i (where i will also go until n).

Hence the overall complexity is

$$t_3(n) \quad = \quad O\left(n \cdot (\ O(n) + O(n))\right) = O\left(n^2\right)$$

Hence the complexity of the algorithm expressed in a Big-O notation is as follows:

☐ ~~$O(n)$~~

☐ ~~$O(n^3)$~~

☑ $O(n^2)$

☐ ~~$O(logn)$~~

Solution to 1.3 The program in front of us is a recursive implementation for calculating the $n-th$ Fibonacci number (where n is given as a parameter) in the Fibonacci sequence. At first, we observe that this is a recursive function, so we will have to analyse the recursion tree to understand its' complexity. Namely, we need to consider the branching factor of the tree and the depth of the tree.

The complexity of the exit condition is $O(1)$ as it is only checking an equality and returning n. Then the complexity of the entire function is:

$$t_{Fib}(n) \quad = \quad O(O(t_{Fib}(n-1)) + O(t_{Fib}(n-2)) + O(1))$$

Since the exercise does not actually require us to calculate the precise tight bound (for Fibonacci is $\approx \theta(1.6^n)$) but only the one that is tightest among the choices, we can simply find the upper bound by examining the recursion tree. First, we observe that we have two recursive calls ($Fibonacci(n-1)$ and $Fibonacci(n-2)$), therefore we know the branching factor is 2. To find the tree height, we can draw the tree for some empirically chosen n (e.g., $n = 5$) and observe the tree depth.

```
5 \\ depth 0, we call the function with n=5
.4 \\ depth 1, we call the function with n-1=5-1=4
.3 \\ depth 1, we call the function with n-2=5-2=3
..3 \\ depth 2, we call the function with n-1=4-1=3
..2 \\ depth 2, we call the function with n-2=4-2=2
..2 \\ depth 2, we call the function with n-1=3-1=2
..1 \\ depth 2, we call the function with n-2=3-2=1
...2 \\ depth 3, we call the function with n-1=3-1=1
...1 \\ depth 3, we call the function with n-2=3-2=1
...1 \\ depth 3, we call the function with n-1=2-1=1
...0 \\ depth 3, we call the function with n-2=2-2=0
...1 \\ depth 3, we call the function with n-1=2-1=1
...0 \\ depth 3, we call the function with n-2=2-2=0
```

```
...1<=1 return \\ depth 3, we call the function with 1 and immediately return
....1 \\ depth 4, we call the function with n-1=2-1=1
....0 \\ depth 4, we call the function with n-2=2-2=0
(and so on for the rest of the branches at depth 4)
.....1<=1 return \\ depth 5, we call the function with 1 and immediately return
(and so on for the rest of the branches at depth 5)
```

In the example above the maximum depth was 5 (which is equal to n). We can observe that not all the branches in the recursive tree had the depth of 5 (e.g., for some, we returned beforehand). This is precisely why the tight bound is actually $\approx \theta(1.6^n)$.

However a good approximation is to assume that $t_{Fib}(n-1))\tilde{t}_{Fib}(n-2)$ and therefore

$$\begin{aligned} t_{Fib}(n) &= t_{Fib}(n-1) + t_{Fib}(n-2) + O(1) \\ &\leq 2 \cdot t_{Fib}(n-1) + O(1) \\ &\leq 4 \cdot t_{Fib}(n-2) + 2 \cdot O(1) \\ &\leq 2^k \cdot t_{Fib}(n-k) + k \cdot O(1) \\ &= (2^n + n)O(1) \\ &= O(2^n) \end{aligned}$$

The other values are clearly wrong as $O(1^n) = O(1)$ as well as $O(2^{1.6 \log n}) = O(n^{1.6 \log 2})$ which is again a polynomial.

Hence the complexity of the algorithm expressed in a Big-O notation is as follows:

☐ ~~$O(n^2)$~~

☑ $O(2^n)$

☐ ~~$O(1^n)$~~

☐ ~~$O(2^{1.6 \log n})$~~

Solution to 1.4 This question includes the same code fragment as 1.5. The only difference is that this question asks to mark the false statements (as opposed to marking the true complexity of method3). For detailed solution explanation see solution to problem 1.5. Therefore the following options are false and should be selected:

☐ ~~The complexity of method3 expressed with n (where n is the length of the array arr) in terms of Big-O notation is $O(n^2 \cdot \log n)$.~~

☑ The complexity of method3 expressed with n (where n is the length of the array arr) in terms of Big-O notation is $O(n^3 \cdot \log n)$.

☑ The complexity of method3 expressed with n (where n is the length of the array arr) in terms of Big-O notation is $O(n^3)$.

☑ The complexity of method3 expressed with n (where n is the length of the array arr) in terms of Big-O notation is $O(n \cdot (n + n^2))$.

Solution to 1.5 At first we observe that array elements in the language C starts from 0 and go till $n-1$. For the inner loop of `method2` we observe that eventually when $k < 3$ the division by 3 yields 0 as a value. Then we break down the components of the algorithm in a top-down fashion.

- The method `method3` calls repeatedly `method1` and `method2` for the duration of the for loop which scan the entire array `arr` which has n elements. Its complexity is therefore

$$t_3(n) = O\left(n \cdot (\, t_1(n) + t_2(n))\right)$$

- The method `method1` basically prints every third element of `arr` (i.e. the n-th element, the $n-4$-th element, etc..

$$t_1(n) \quad = \quad O\left(1/3 \cdot n\right) = O\left(n\right)$$

- The method `method2` has an other loop which linearly scan the array *arr* repeating the inner loop irrespective of the value of the index i of the outerloop. The inner loop proceeds to an exponentially decreasing scan (i.e. the n-th element, the $n/3$-th element,$n/9$, $n/27$ etc.). Eventually the inner loop stops when $3^k \geq n$ i.e. when $k = \log_3 n$.

$$t_2(n) \quad = \quad O\left(n \cdot t_{innerloop}(n)\right) = O\left(n \cdot (\log_3 n)\right) = O\left(n \cdot \log n\right)$$

Hence the overall complexity is

$$t_3(n) \quad = \quad O\left(n \cdot (\ O(n) + O(n \log n))\right) = O\left(n^2 \log n\right)$$

Hence the complexity of the algorithm expressed in a Big-O notation is as follows:

☑ $O(n^2 \cdot \log n)$

☐ ~~$O(n^3 \cdot \log n)$~~

☐ ~~$O(n^3)$~~

☐ ~~$O(n \cdot (n + n^2))$~~

TYPICAL MISTAKE 6.1 *A typical mistake when answering questions of the form "which is the complexity of an algorithm" is answering them as if they were questions of the form "which functions bound [from above] the complexity of the algorithm".*

Questions of the latter type have a different answer. In the case of the algorithm `method3` from question 1.5 all possible choices would have been correct answers to the question 'which functions bounds from above the complexity of `method3`'. Indeed, if $t(n) = n^2 \log n + n^2$ then we can say that

$$t(n) = O(n^2 \log n) = O(n^3) = O(n^3 \log n) = O(2^n) = O(n!) = O\left(\underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}_{n}\right) = O(Ackermann(n,n))$$

where $Ackermann(4,n)$ is the tower of powers of 2s with height n. It is clear from this example that using the Big-O notation in this way would be completely meaningless when answering the question 'which is the complexity of the algorithm X'. A function to print every third element of an array a few times would be classified as having the complexity of an algorithms requiring more time than the life of the Universe.

Therefore, when answering question such as Question 1.5, one should only eliminate the lower order terms to simplify calculations and not introduce arbitrary (meaningless) higher order terms. The use of the verb is, as opposed to the use of the verb bounds (from above), is therefore an important indication of what is a meaningful answer.

TYPICAL MISTAKE 6.2 *Another typical mistake in the use of the Big-O notation is to interpret the equality symbol $=$ in the classical mathematical sense as a symmetric relation.*

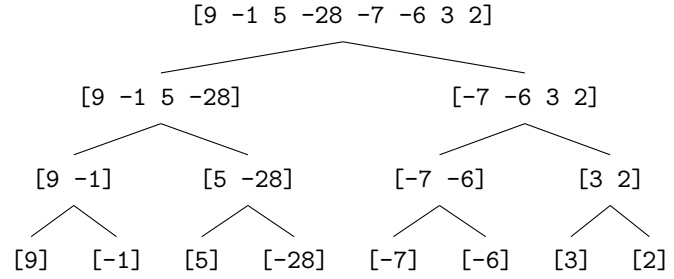
In the 'normal' mathematical notation and writing, when we write that the relation $a = b$ holds we take it from granted that the relation $b = a$ also always holds.

From the example above it is clear that this is not the case. Writing $O(n^2 \log n) = O(n^3)$ is correct because for every function f such that $f(n) \leq c \cdot n^2 \log n$ for some constant c_f, n_f and all $n > n_f$ it is also true that $f(n) \leq c_f \cdot n^3$. However, swapping the terms of the 'equality' and writing $O(n^3) = O(n^2 \log n)$ would be clearly incorrect.

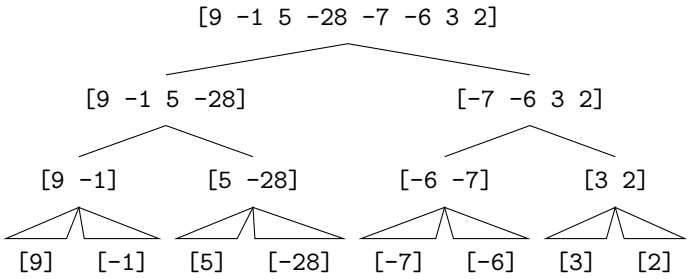
Unfortunately, this overloading of the equality symbol with what should be rather an inequality symbol is so entrenched that essentially all textbooks uses it (e.g. [**?**, Chapter 2,Part I]). To be formally correct one should use the $\leq$ symbol.

6.2 Sorting Algorithms

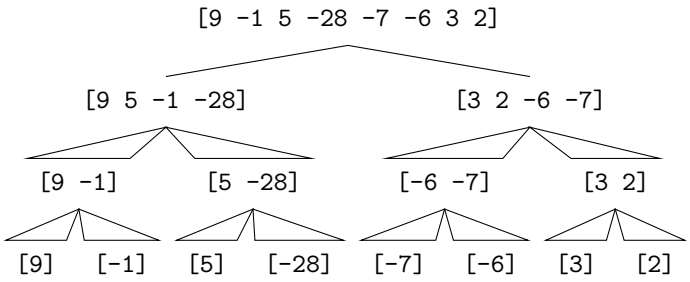
Solution to 2.1 The following tree describes the recursive calls of merge sort until it arrives at the bottom of the recursion tree:



After the leaves of the recursion tree are merged we end up as follows:



The result of the merge at the next level of the call tree, which is the one asked by the question, is therefore



Then the correct answer is as follows:

☐ ~~-1 5 9 -7 -6 3 2 -28~~

☐ ~~-28 -1 -6 -7 9 2 3~~

☐ ~~-28 -1 5 9 -7 -6 2 3~~

☑ 9 5 -1 -28 3 2 -6 -7

Solution to 2.2. First, since the pivot p is the first element, we need to partition the array around 3. We need to maintain the invariant *"all elements between p and $i < p$ and all elements between i and j and $> p$"*. We will first partition the array into the form: $p; all < p; all > p$. Finally we can place the pivot in the correct position at the end. The following describes how the array is partitioned:

```
3 1 9 4 4 5 6 7 \\ original array, pivot=3, underlined is the partitioned part
3 1 9 4 4 5 6 7 \\ 1<3
3 1 9 4 4 5 6 7 \\ 1<3 and 9>3
3 1 9 4 4 5 6 7 \\ 1<3 and 9 and 4 are both >3
...
3 1 9 4 4 5 6 7 \\ 1<3 and 9,4,4,5,6,7 are all >3
1 3 9 4 4 5 6 7 \\ last step is placing the pivot in the correct place
```

Then the correct answer is as follows:

☐ ~~3 1 9 4 4 5 6 7~~

☐ ~~1 3 4 4 5 9 6 7~~

☐ ~~1 3 4 4 5 6 7 9~~

☑ 1 3 9 4 4 5 6 7

Solution to 2.3 This question presents the same problem as 2.4. The only difference is that this question asks to mark the false statements. For detailed solution explanation see solution to problem 2.4. Therefore the following options are false and should be selected:

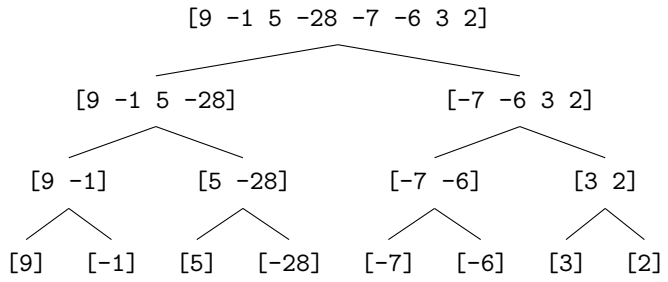
☑ After the merges in the next level in the recursion tree the state of the array is `-28 -1 5 9 -7 -6 3 2`.

☐ ~~After the merges in the next level in the recursion tree the state of the array is `-28 -1 5 9 -7 -6 2 3`.~~

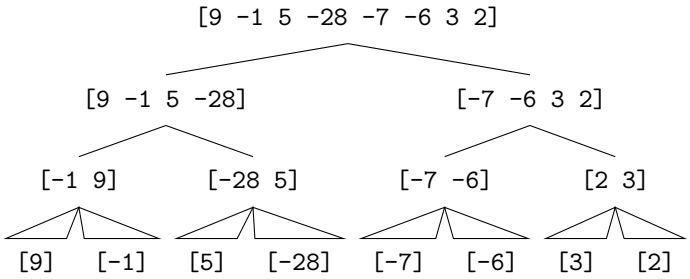
☑ After the merges in the next level in the recursion tree the state of the array is `-28 -7 -6 -1 2 3 5 9`.

☑ After the merges in the next level in the recursion tree the state of the array is `-28 -7 -6 9 2 3 -1 9`.

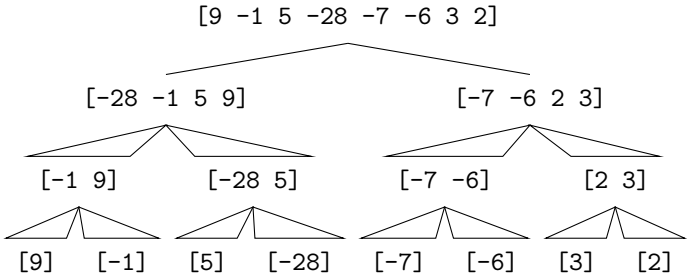
Solution to 2.4. The following tree describes the recursive calls of merge sort until it arrives at the bottom of the recursion tree:



After the leaves of the recursion tree are merged we end up as follows:



The result of the merge at the next level of the call tree, which is the one asked by the question, is therefore



Then the correct answer is as follows:

☐ ~~-28 -1 5 9 -7 -6 3 2~~

☑ -28 -1 5 9 -7 -6 2 3

☐ ~~-28 -7 -6 -1 2 3 5 9~~

☐ ~~-28 -7 -6 9 2 3 -1 9~~

6.3 Divide-and-Conquer and Dynamic Programming

Solution to 3.1. This question presents the same problem as 3.4. The only difference is that this question asks to mark the false statements. For detailed solution explanation see solution to problem 3.2. Therefore the following options are false and should be selected:

☑ Differently from most divide and conquer algorithms, the dynamic programming approach always recurses on subproblems of equal size.
This statement is false, therefore it should be selected.

☑ Both Merge sort and Quick sort are examples of dynamic programming when n is a power of 2.
This statement is false, therefore it should be selected.

☐ ~~Quick sort is a divide and conquer algorithm.~~
This statement is actually correct, therefore it should not be selected as false.

☐ ~~If $s(k)$ is the complexity of testing that a partial solution of k elements can be extended by adding a $k+1$th element, dynamic programming programming complexity is bounded by $O(n \cdot s(n))$.~~
This statement is actually correct, therefore it should not be selected as false.

Solution to 3.2. What makes dynamic programming algorithms efficient is the possibility of computing the solutions of the common sub-problems only once each time the sub-problem is encountered and having only a polynomial number of sub-problems in the size of the inputs. In this way, the first time a sub-problem of size k is encountered it may cost $s(k)$ but, from that moment on, it will only cost $O(1)$ to retrieve the solution to the sub-problems if we consider $p(n)$, the number of subproblems, we would have

$$\begin{aligned} t(n) &= \sum_{k=1}^{p(n)} s(k) \\ &\leq \sum_{k=1}^{p(n)} s(n) \\ &= p(n) \cdot s(n) \end{aligned}$$

A key issue here how many different sub-problems do we have. In absence of information the number of subproblem might be exponential. Then dynamic programming would not be very efficient. If we don't have any additional information then we might have to consider all possible subsets of elements. So for example, if we have a set of n elements and we must find some optimal subset, the possible number of subset is 2^n and, in absence of additional information we would need to test all subsets, so the overall complexity would be $O(s(n) \cdot 2^n)$.

If we are given the information that a solution set with k elements can be extended to a set of $k+1$ elements with complexity $s(k)$ then the complexity changes as we can define the following dynamic programming algorithm

```
Algoritm ExtendDynamically
Input n
Input elements e1,... en
sol = empty set;
for element ei = 1...n
do
   try = ExtendSolution(sol,ei)
   if try != fail
   then
      sol = sol U {ei};
done
```

The complexity would then be

$$\begin{aligned} t(n) &= \sum_{k=1}^{n} s(k) \\ &\leq \sum_{k=1}^{n} s(n) \\ &= n \cdot s(n) \end{aligned}$$

Then the correct answer is as follows:

☐ ~~Differently from most divide and conquer algorithms, the dynamic programming approach always recurses on subproblems of equal size~~ This statement is false as also explained above. Dynamic programming algorithms do not always subdivide a problems into equal size.

☐ ~~Both Merge sort and Quick sort are examples of dynamic programming when n is a power of 2~~.

☑ Quick sort is a divide and conquer algorithm.
While Merge sort is the prototypical example of divide and conquer, also Quicksort is a divide a conquer algorithm. It divides the array into subarrays that will have statistically equal size and then solves these subproblems in the same way.

☑ If $s(k)$ is the complexity of testing that a partial solution of k elements can be extended by adding a $k+1$th element, dynamic programming programming complexity is bounded by $O(n \cdot s(n))$.
This is precisely the case described above.

Solution to 3.3 This question presents the same problem as 3.4. The only difference is that this question asks to mark the false statements. For detailed solution explanation see solution to problem 3.4. Therefore the following options are false and should be selected:

☑ Merge sort is not a divide and conquer algorithm.

☐ ~~Differently to dynamic programming, a divide and conquer approach usually recurses on subproblems of the same size.~~

☑ Every dynamic programming solution is also an instance of the divide and conquer approach.

☑ The complexity of the dynamic programming solution to the 0/1 knapsack problem is $O(2^n)$, where n is the number of items considered.

Solution to 3.4. Both divide-and-conquer and dynamic programming break down the problem recursively into sub-problems and then combine the solutions of the sub-problems to obtain a solution of original problem in polynomial time. The major difference is that divide-and-conquer divides the original problem into non-overlapping sub-problems whereas dynamic programming is used when there might be sub-...-sub-problems common to both initial sub-problems.

What makes divide-and-conquer algorithms efficient (for some problems) is the possibility to break down the problem of initial size n into k-sub-problems of approximately equal size n/k so that the problem size decreases exponentially after each recursion step (i.e. after $O(\log_k n)$ steps one has a problem of constant size).

If one would break the problem into a sub-problem of constant size k and a sub-problem of size $n-k$ a divide-and-conquer algorithm would not be very efficient. For example it is perfectly possible to use merge-sort when the array to be sorted is divided in the way just mentioned and the result would be correct. However, the algorithm would not require $O(n \log n)$ steps but rather $O(n^2)$.

What makes dynamic programming algorithms efficient (for some problems) is the possibility of computing the solutions of the common sub-problems only once each time the sub-problem is encountered and having only a polynomial number of sub-problems in the size of the inputs. In this way, the first time a sub-problem of size $n-k$ is encountered it may costs $t(n-k)$ but, from that moment on, it will only costs $O(1)$ to retrieve the solution to the sub-problems.

In the 0/1 knapsack problem with n items and W weights we partition the problem into $n \cdot W$ sub-problems and we solve the problem by placing individually each item into the corresponding level of weights thus creating a large matrix $n \times W$. If the number of different sub-problems were exponential, dynamic programming would be far from efficient.

Then the correct answer is as follows:

☐ ~~Merge sort is not a divide and conquer algorithm~~.
Merge sort is the prototypical example of divide-and-conquer algorithms studied in the course. See the solution to 2.4 as an example.

☑ Differently to dynamic programming, a divide and conquer approach usually recurses on sub-problems of the same size.
This is precisely the case when one uses divide-and-conquer to efficiently find solutions as we illustrated above.

☐ ~~Every dynamic programming solution is also an instance of the divide and conquer approach~~.
As explained above, dynamic programming is used when we cannot properly divide the problem into distinct sub-problems as we do for divide-and-conquer. The opposite sentence is often considered to be true by several authors (e.g. [**?**, Chapter 16, Part III]: divide and conquer is a special case of dynamic programming when the sub-problems have no overlap.

☐ ~~The complexity of the dynamic programming solution to the 0/1 knapsack problem is $O(2^n)$, where n is the number of items considered.~~
$O(2^n)$ would be the solution for the exponential search of all possible combinations.

TYPICAL MISTAKE 6.3 *A typical mistake that one finds in the internet is that the divide-and-conquer is claimed to be recursive and the dynamic- programming is claimed to be not recursive.*

The authors of those Internet pages (which are typically copied and pasted from each other) confuse the way in which recursion is implemented (by iteration or by a call stack) with the conceptual recursive division of the problem.

For example, the pseudocode below is a recursive implementation for the computation of Fibonacci numbers that takes linear time $O(n)$ simply by using a static array of size $O(n)$. It is possible to create a similar version that only use a dynamic array.

```
n =...;
int f[n]; // compiler inizializes to zero

int recfib(n) {
    if (n==0 | n==1)
        f[n]=1;
    else if (f[n]==0)
        f[n]=recfib(n-1)+recfib(n-2);
    return(f[n]);
}
main() {
 print(recfib(n));
}
```

One can try to prove that the above algorithm takes linear time using the traditional recurrence equation below

$$\begin{aligned} T(1) &= a \\ T(n) &= T(n-1) + T(n-2) + b \end{aligned}$$

Unfortunately, using the equations above would result in an exponential upper bound as explained on the Master Method [**?**, Chapter 4, Part I]).

To show it, we apply to the recurrence to $n-1$ to obtain

$$T(n-1) = T(n-2) + T(n-3) + b$$

and therefore $T(n-2)+b \leq T(n-1)$ since $T(n-3)$ is some positive quantity. Then by replacing this inequality in the original recurrence equation for n we would obtain

$$T(n) \leq T(n-1) + T(n-2) + b \leq 2 \cdot T(n-1) \leq 4 \cdot T(n-2) \leq \cdots \leq 2^n a = O(2^n)$$

To prove the desired result of linear time complexity, one needs a more careful analysis and prove by induction that for all n the following two conditions are true

1. if `recfib(n)` has been calculated at least once then also `recfib(n-1)` has been already calculated at least once, and
2. any invocation of `recfib(n)` after the first one only takes $O(1)$.

Then the correct recurrence equation becomes

$$T(n) = T(n-1) + d + c$$

and this can be solved as $T(n) = O(n)$.

TYPICAL MISTAKE 6.4 *A typical mistake when describing the complexity of optimization problems is not distinguishing whether the question asked for the complexity of the algorithm in the value of the optimization parameters or the complexity in the size of the optimization parameters.*

In the classical example of the 0/1 knapsack solved by dynamic programming we consider the complexity of the algorithm in the number of items n and value of the constraint weight W. This is considered efficient because the algorithm takes $O(n \cdot W)$.

If we measured the complexity in terms of the size to store the optimization parameters (i.e. $\log_2 W$ bits for storing the weight) this would take linear time in the number of items and exponentially long time in the size of the weights $O(n \cdot W) = O(n \cdot 2^{\log W})$.

6.4 Operations of Data Structures

Solution to 4.1 To answer this question, you need to know what is a heap, balance search tree, and sorted array and what are the running times for the operations for these data structures. The following are the answers for this question:

☑ The running time (in terms of Big-O) of the `Insert` operation of the heap is the same as running time of the `Insert` operation of the balanced search tree, which is $O(\log n)$.
This statement is correct, both data structures offer the insert operation in logarithmic time.

☐ ~~The running time (in terms of Big-O) of the `Insert` operation for sorted array is $O(1)$.~~
This statement is false. After inserting a new item, additional work is required to maintain the property of the sorted array. In particular, the running time of this operation is $O(n)$ and not only $O(1)$.

☑ The running time (in terms of Big-O) of the `Delete` operation for sorted array is $O(n)$.
This statement is true. Similar to the insert operation, the delete operation is painfully slow when realized over sorted arrays. This is because the sorted array is a static data structure, which does not support well dynamic modifications. To delete an item, we would need to maintain the properties of the sorted array, which in the worse case takes $O(n)$.

☐ ~~The running time (in terms of Big-O) of the `Insert` operation for balanced search tree is $O(n^2)$ and not $O(\log n)$ because after inserting a new item additional work is required to balance the tree again.~~
This statement is false. It is true that additional computation is required to balance (see also the rotations described in the course literature [**?**] and in class during Lecture in week 4) the tree after a modification (both in case of inserting into and deleting from the search tree). However, the rotations can be implemented efficiently and thus the final complexity of is still $O(\log n)$.

Solution to 4.2 To answer this question, one must understand what a heap data structure is and which operations it is good for. The major advantage of using heaps is to support fast minimum computations with heap sort being a canonical application. Negating the key of every object turns a heap into a data structure that supports fast maximum computations. Heaps do not generally support fast lookups unless you are by some chance happen to be looking for the object with the minimum key. Therefore the following statements are correct and should be selected:

☐ ~~Repeated lookups.~~

☑ Repeated minimum computations.

☑ Repeated maximum computations.

☐ ~~None of the other options.~~

Solution to 4.3 First, finding the minimum element in a heap only requires reading from the root of the heap which stored the minimum element already, therefore `Find min` operation has the running time of $O(1)$. This is the main advantage of using a heap data structure, so if a program needs to often find a minimum value, this data structure is a very good choice.

Most operations that modify the heap data structure have the running time of $O(logn)$, where n is the number of items stored in the heap. The reason behind this is that modifications (such as updating the heap and removing items from the heap) require some work to make sure that the heap invariant is preserved also after the modification. Therefore, the running times of `Insert, Extract min, Delete` operations for the heap data structure are $O(logn)$.

An exception is `Heapify`, which is an operation that builds a heap from an array. Intuitively, one may guess that the running time of this operation is $O(n * log(n))$, however, it turns out that this operation can be more efficient, if implemented correctly. The reason for this is the fact that the operation starts with the leave nodes, and works its way from bottom to top. And since the nature of a tree-like structure is that there are more nodes at the bottom of the tree compared to the top, there are actually (on average) fewer swaps than $log(n)$ required for every node. See also [**?**, Chapter 11, Part II] and slides of the lectures.

Therefore, the correct answers (for all operations) are as follows:

	$O(1)$	$O(n)$	$O(\log n)$	$O(n \cdot \log n)$
Insert	☐	☐	☑	☐
Extract min	☐	☐	☑	☐
Find min	☑	☐	☐	☐
Heapify	☐	☑	☐	☐
Delete	☐	☐	☑	☐

Solution to 4.4 This question presents the same problem as 4.6. The only difference is that this question asks to mark the false statements. For detailed solution explanation see solution to

problem 4.6. Therefore the following options are false and should be selected:

	$O(1)$	$O(n)$	$O(\log n)$	$O(n \cdot \log n)$
Insert	☑	☑	☐	☑
Extract min	☑	☑	☐	☑
Find min	☑	☑	☐	☑
Predecessor	☑	☑	☐	☑
Delete	☑	☑	☐	☑

Solution to 4.5 To answer this question, you need to know what is a sorted array and what are the running times for the operations for this data structure. A sorted array is simply an array that maintains an order of the objects that it stores (in increasing or decreasing order of some object attribute (or key), depending on the concrete implementation). The most simple example is an array of integers, sorted by decreasing order of the integer values. However, we can easily imagine implementing a custom sorted array that stores objects by some decreasing attribute (e.g., if we wish to quickly display items in an online web-store sorted by price).

The `Search` is a supported operation of sorted arrays, where, for a key k, we return a pointer to the object in the data structure with key k (or report that no such object exists).

The `Insert` is an operation that is typically not supported in sorted arrays, however it is not impossible to realize it. Given a new object x, this operation would add this object to the data structure (while maintaining its' properties).

The `Predecessor` is a supported operation of sorted arrays, where, given a pointer to an object in the data structure, we return a pointer to the object with the closest smaller key. If the given object has the smallest key, no predecessor exists, and we report "none".

The `Delete` is an operation that is typically not supported in sorted arrays, however it is not impossible to realize it. For a key k, this operation would delete object with key k from the data structure, or report that no such object exists.

To revisit all the operations of the sorted array data structure, the diligent reader may also refer to the lecture on Search Trees and Shortest Path, and the book [**?**].

The following statements are true and should be selected:

☑ The running time (in terms of Big-O) of the `Search` operation for sorted array is $O(\log n)$.
This statement is true. The operation of `Searching` for an object in a sorted array is most efficiently implemented with a binary search, therefore the complexity of this operation is the same as for performing a binary search, that is $O(\log n)$.

☐ ~~The running time (in terms of Big-O) of the `Insert` operation for sorted array is $O(\log n)$.~~
This statement is false. The insert operation can be realized for sorted arrays, but it is painfully slow compared to for instance Insert in binary search trees (which takes $O(\log n)$). This is because the sorted array is a static data structure, which does not support well dynamic modifications. To insert a new item, we would need to maintain the properties of the sorted array, which in the worse case takes $O(n)$.

☑ The running time (in terms of Big-O) of the `Delete` operation for sorted array is $O(n)$.
This statement is true. Similar to the insert operation, the delete operation is painfully slow when realized over sorted arrays. This is because the sorted array is a static data structure, which does not support well dynamic modifications. To delete an item, we would need to maintain the properties of the sorted array, which in the worse case takes $O(n)$.

☐ ~~The running time (in terms of Big-O) of the `Predecessor` operation for sorted array is $O(n \cdot \log n^2)$.~~
This statement is false. To implement the `Predecessor` operation, we first use the `Search` operation to recover the position of the object in the array referred to by the pointer. We then return the pointer to the object in the previous position. This operation is as fast as the `Search` ($O(\log n)$, which is much faster compared to $O(n \cdot \log n^2)$). In fact, in C++, the `Predecessor` and `Successor` operations can even be implemented in $O(1)$ by using pointer arithmetic.

Solution to 4.6 The running time of every binary search tree operation (except `output sorted`) is proportional to the tree's height, which can range anywhere from the best case scenario of $log_2(n)$ (for a perfectly balanced search tree) to the worse case scenario of a linear chain, $n - 1$.

Since the difference between linear and logarithmic running time is not negligible for large trees, we can use balanced search trees (which can be binary or not). Several different implementations of balanced search trees exist, but all spend more computation balancing the tree to guarantee the running time of every operation (except `output sorted`) of $O(\log n)$. See also [**?**, Chapter 11, Part II] and slides of the lectures.

Therefore, the correct answers (for all operations) is as follows:

	$O(1)$	$O(n)$	$O(\log n)$	$O(n \cdot \log n)$
Insert	☐	☐	☑	☐
Extract min	☐	☐	☑	☐
Find min	☐	☐	☑	☐
Predecessor	☐	☐	☑	☐
Delete	☐	☐	☑	☐

6.5 Graph algorithms: topological sort, breadth-first search (BFS) and depth-first search (DFS)

Solution to 5.1 This question presents the same problem as 5.2. The only difference is that this question asks to mark the true statements. For detailed solution explanation see solution to problem 5.2. Therefore, the following statements should be selected:

☑ Node reachability problem (whether some node in a graph is reachable from the starting node) can be solved using either breadth-first or depth-first search.
This statement is true, therefore it should be selected.

☑ For a directed acyclic graph $G = (V, E)$, the topological ordering is an assignment $f(v)$ for every node $v \in V$ to a different number, such that for every $(v, w) \in E$, $f(v) \leq f(w)$.
This statement is true. It is the definition taken from the book [**?**, Chapter 8, Part II].

☑ For extremely large graphs (e.g., more than 1 million nodes) it is more memory efficient to implement iterative depth-first search as opposed to recursive depth-first search.
This statement is true, because recursive program implementations will actually copy certain variables on the program stack at every new recursive call. When the inputs are small, this memory overhead is negligible, but for very large inputs, recursive implementations (with many recursive calls) may fill up the available memory.

☐ ~~Every graph has a topological ordering.~~
This is the only statement that is false. Every *directed acyclic graph* (DAG) has a topological ordering, and not just any graph.

Solution to 5.2 To answer this question one must understand what constitutes a topological ordering and the key concept of breadth-first search and depth-first search. Both can be found in [**?**, Chapter 8, Part II].

A topological ordering in a direct graph is an assignment f(v) of every node v to a different number so that for every edge $(v, w), f(v) < f(w)$. A topological order of the nodes essentially reflects their depth in the graph, the number of edges one needs to traverse to reach them starting from some 'root' node.

Since an ordering must be transitive and well-founded, one might need to visit an arbitrary portion of the graph before deciding what could be the right function f for the ordering.

A very simple example is a graph with four edges $(v_0, v_1); (v_0, v_2); (v_2, v_3); (v_3, v_1)$. If we start from v_0 we can clearly choose a function f such that $f(v_0) < f(v_1), f(v_2)$. However, we cannot establish an ordering between v_1 and v_2 until we have visited v_3 from v_2 and further found out that from v_3 we can actually reach back v_1 so we have a path $(v_0, v_2); (v_2, v_3); (v_3, v_1)$ and therefore we must have $f(v_2) < f(v_2) < f(v_1)$.

The part of the graph that must be visited can be arbitrarily long. For example, the situation above can be generalized to a DAG with two set of edges $\{(v_0, v_{0,i}) | i = 1 \dots n\}$ and $\{(v_{0,i}, v_{i,j}) | j = 1 \dots n\}$ and an edge $(v_{nn}, v_{0,k})$ for some k. One would visit all $v_{0,i}$ nodes but cannot establish any value of the ordering among any of the $v_{0,i}$ (including $v_{0,k}$) until all $v_{i,j}$) have been visited. Until we had reached $v_{n,n}$ and found out that its successor was $v_{0,k}$ (and not $v_{0,1}$ or $v_{0,2}$ etc.) we would not know if we would need to have $f(v_{0,i}) < f(v_{0,k}$ or viceversa.

In other words, we can not know the order among siblings until we have visited their neighbors and their neighbors' neighbors and so on. This property is what determines the algorithm of choice for topological orderings.

If a directed graph has cycles such order cannot be well founded since for all nodes in a cycle $(v_1, v_2); (v_2, v_3); \dots; (v_{n-1}, v_n); (v_n, v_1)$ one would have $f(v_1) < f(v_2) < \dots < f(v_n) < f(v_1)$ which is clearly impossible. For the same reason, a topological order on an undirected graph would not be well defined.

As explained in [**?**], upon reaching a node a breadth-first search algorithm first visit all neighboring nodes, i.e. those reachable from the current node by a single edge traversal, first before increasing the depth of the search to the neighbors of the neighbors and so on.

For depth-first search, every branch of the graph is first explored in its depth before the neighboring nodes are considered. This means that upon reaching a node one of its neighboring nodes is selected and visited, then a neighbor of the latter node is visited, and so on. Only after no node can be further reached the algorithm backtracks to the first visited node. If a graph has cycles it might be that a depth-first search returns to a previously visited node but this event cannot happen for direct acyclic graphs.

Therefore, the following statements are false:

☐ ~~Node reachability problem (whether some node in a graph is reachable from the starting node) can be solved using either breadth-first or depth-first search.~~
This statement is true, therefore it should not be selected.

☐ ~~For a directed acyclic graph $G = (V, E)$, the topological ordering is an assignment $f(v)$ for every node $v \in V$ to a different number, such that for every $(v, w) \in E$, $f(v) \leq f(w)$.~~
This statement is true, as explained above. It is the definition taken from the book [**?**, Chapter 8, Part II].

☐ ~~For extremely large graphs (e.g., more than 1 million nodes) it is more memory efficient to implement iterative depth-first search as opposed to recursive depth-first search.~~
This statement is true, because recursive program implementations will actually copy certain variables on the program stack at every new recursive call. When the inputs are small, this memory overhead is negligible, but for very large inputs, recursive implementations (with many recursive calls) may fill up the available memory.

☑ Every graph has a topological ordering.
This is the only statement that is false. Every *directed acyclic graph* (DAG) has a topological ordering, and not just any graph.

TYPICAL MISTAKE 6.5 *A typical mistake is to assume that the data structure has no impact on the complexity of a graph traversal algorithm.*

Of course, it is always possible to use arbitrarily cumbersome data structures and algorithms to inefficiently (and possibly incorrectly) visit a graph. However, some data structures have a natural support for some types of traversals.

Using an 'unnatural' data structure means that the designer would have to mimic one's pet data structure the functionalities of the 'natural' data structure. Such operations will naturally have higher costs, because the data structure has not been optimized for these operations, and a higher overall complexity both in theory and in practical implementations.

There might also be an impact on correctness. For example in the $v_0, \dots v_{0,i}, \dots v_{i,1} \dots, v_{i,j}$, the breadth first algorithm cannot assign a value f and then just mark as visited the nodes $v_{0,i}$ after the first iteration has removed them from the queue. The algorithm must keep them somewhere for further processing when the edge $(v_{n,n}, v_{0,k})$ will appear.

Solution to 5.3 This question presents the same problem as 5.4. The only difference is that this question asks to mark the false statements. For detailed solution explanation see solution to problem 5.4. Therefore the following options are false and should be selected:

☑ Topological orderings can be computed for directed acyclic graphs (DAGs), but not every DAG has a topological ordering.

☐ ~~Breadth-first search is usually implemented using a queue, while depth-first search is usually implemented using the stack data structure.~~

☑ Computing a topological ordering of a DAG is the most typical application of the breath-first search algorithm.

☑ For extremely large graphs (e.g., more than 1 million nodes), the recursive implementation of the depth-first search is going to use less memory compared to the iterative implementation, which has a memory overhead for each iteration.

Solution to 5.4 To answer this question one must understand what constitutes a topological ordering and the key concept of breadth-first search and depth-first search. Both can be found in [**?**, Chapter 8, Part II].

A topological ordering in a direct graph is an assignment f(v) of every vertex v to a different number so that for every edge $(v, w), f(v) < f(w)$. A topological order of the nodes essentially reflects their depth in the graph, the number of edges one need to traverse to reach them starting from some 'root' node.

Since an ordering must be transitive and well founded one might need to visit a arbitrary portion of the graph before deciding what could be the right function f for the ordering.

A very simple example is a graph with four edges $(v_0, v_1); (v_0, v_2); (v_2, v_3); (v_3, v_1)$. If we start from v_0 we can clearly choose a function f such that $f(v_0) < f(v_1), f(v_2)$. However we cannot establish an ordering between v_1 and v_2 until we have visited v_3 from v_2 and further found out that from v_3 we can actually reach back v_1 so we have a path $(v_0, v_2); (v_2, v_3); (v_3, v_1)$ and therefore we must have $f(v_2) < f(v_2) < f(v_1)$.

The part of the graph that must be visited can be arbitrarily long. For example the situation above can be generalized to a DAG with two set of edges $\{(v_0, v_{0,i}) | i = 1 \dots n\}$ and $\{(v_{0,i}, v_{i,j}) | j = 1 \dots n\}$ and an edge $(v_{nn}, v_{0,k})$ for some k. One would visit all $v_{0,i}$ nodes but cannot establish any value of the ordering among any of the $v_{0,i}$ (including $v_{0,k}$) until all $v_{i,j}$) have been visited. Until

we had reached $v_{n,n}$ and found out that its successor was $v_{0,k}$ (and not $v_{0,1}$ or $v_{0,2}$ etc.) we would not know if we would need to have $f(v_{0,i}) < f(v_{0,k}$ or viceversa.

In other words we can not know the order among siblings until we have visited their neighbors and their neighbors' neighbors and so on. This property is what determines the algorithm of choice for topological orderings.

If a directed graph has cycles such order cannot be well founded since for all nodes in a cycle $(v_1, v_2); (v_2, v_3); \ldots; (v_{n-1}, v_n); (v_n, v_1)$ one would have $f(v_1) < f(v_2) < \ldots < f(v_n) < f(v_1)$ which is clearly impossible. For the same reason, a topological order on an undirected graph would not be well defined.

As explained in [**?**], upon reaching a node a breadth-first search algorithm first visit all neighboring nodes, i.e. those reachable from the current node by a single edge traversal, first before increasing the depth of the search to the neighbors of the neighbors and so on.

For depth-first search, every branch of the graph is first explored in it's depth before the neighboring nodes are considered. This means that upon reaching a node one of its neighboring node is selected and visited, then a neighbor of the latter node is visited and so on. Only after no node can be further reached the algorithm backtracks to the first visited node. If a graph has cycles it might be that a depth-first search returns to a previously visited node but this event cannot happen for direct acyclic graphs.

☐ ~~Topological orderings can be computed for directed acyclic graphs (DAGs), but not every DAG has a topological ordering.~~

Theorem 8.6 in [**?**, Page 47,Part II] states that every directed acyclic graph has at least one topological ordering. One can prove this theorem by induction. This implies that the statement negating the theorem cannot be true.

☑ Breadth-first search is usually implemented using a queue, while depth-first search is usually implemented using the stack data structure.

The first-in-first-out (or a queue) data structure is a better and more efficient choice for implementing a breadth first graph traversal because upon vising a node the neighboring nodes can be added to queue and extracted in that order. The neighbors of the neighbors will be added at the end of the queue. In contrast, a last-in-first-out (or a stack) is a more appropriate data structure to use when using depth-first search because the neighbors of the last visited one will be added on top and the visit will continue from that node. Either way the natural ordering of the search is preserved by the data structure. See also [**?**, Chapter 8] on graph search and its applications for a similar explanation.

☐ ~~Computing a topological ordering of a DAG is the most typical application of the breath-first search algorithm.~~

Depth-first (and not breadth-first) is the typical algorithm used to compute a topological order because there are only two changes to make to a recursive implementation of DFS in order to compute a topological ordering [**?**, 8.5.4, page 49]. Again, one can obviously explore a graph in any haphazardly way and eventually come out with a topological order of its nodes but implementing a topological ordering using breadth-first search requires significant changes to the algorithm to even guarantee the correctness of the ordering, let alone any efficiency consideration.

In particular, is we look at the example that we described above a breadth-first search starting at v_0, upon reaching $v_{n,n}$ and its neighbor would requires anyhow to backtrack all the way to the root of the tree on previous decisions on $v_{0,i}$, i.e. use a stack, to decide that one must have $f(v_{0,i}) < f(v_{0,k})$.

☐ ~~For extremely large graphs (e.g., more than 1 million nodes), the recursive implementation of the depth-first search is going to use less memory compared to the iterative implementation, which has a memory overhead for each iteration.~~

This is false because recursive programs are the ones actually causing a memory overhead on the computer stack. To understand this, one must understand what implementing a recursive program actually means (which is basic programming knowledge). When a function

is called, a block of memory is allocated and is reserved for storing the function parameters, local variables, function definition, return instruction pointers, etc. At every recursive call, the compiler will allocate a new block of memory for another copy of the local variables. When the program inputs are small this is not noticeable. When the program inputs are very large, recursive implementations will quickly fill up your stack, which will result in the computer to freeze or crash (due to lack of RAM memory).

TYPICAL MISTAKE 6.6 *A typical mistake is to assume that the data structure have no have an impact on the complexity of a graph traversal algorithm.*

Of course, it is always possible to use arbitrarily cumbersome data structures and algorithms to inefficiently (and possibly incorrectly) visit a graph. However, some data structures have a natural support for some types of traversals.

Using an 'unnatural' data structure means that the designer would have to mimic with one's pet data structure the functionalities of the 'natural' data structure. Such operations will naturally have a higher costs, because the data structure has not been optimized for these operations, and a higher overall complexity both in theory and in practical implementations.

There might also be an impact on correctness. For example in the $v_0, \dots v_{0,i}, \dots v_{i,1} \dots, v_{i,j}$, the breadth first algorithm cannot assign a value f and then just mark as visited the nodes $v_{0,i}$ after the first iteration has removed them from the queue. The algorithm must keep them somewhere for further processing when the edge $(v_{n,n}, v_{0,k})$ will appear.

6.6 Graph algorithms: A*

Solution to 5.5 This question is a variation of problem 5.6. Instead, the pseudocode of A* was manipulated to insert two mistakes.

To answer this question one must understand how the A* algorithm explores the graph. The basic intuition behind the Dijkstra shortest path algorithm is that it will always choose the closest next neighbor. Such move might not be good in the long run and might force the algorithm to explore a larger portion of the graph than necessary. The A* combines the information on the closest next neighbor with a heuristic function that represents the most promising neighbor to eventually reach the target.

To achieve this, A* starts to explore the graph at the start node. Then, for all the outgoing edges which have not been explored yet, the algorithm calculates the heuristic score and ultimately chooses to move towards the most promising. The kind of search is repeated in the following iterations, until the target has been reached.

☐ ~~The length of node b is not calculated correctly (in line 11). It should be instead initialized with the maximum value possible for the most conservative estimate: 'len(b) = len(a) + ∞'~~
This statement is false because if we want to make any use of the heuristic function (assuming it is a "good one") then we need to add it to the combined score. Updating the length of next most promising candidate to infinity would result in an incorrect algorithm.

☑ The exit condition is incorrect (in line 8), because this way the while loop will exit whenever the current node is not equal to the target. It should check for equality as follows: 'if $(v = t)$ then exit'
This statement is true as $(v \neq t)$ would result in an incorrect algorithm that would never find the shortest path. Actually, if we were to exit when the current node is not equal to the target node, then this program would exit immediately upon considering the starting node s (except if the starting node is actually the target node).

☐ ~~X is incorrectly initialized (in line 5). It should contain all the vertices in the graph otherwise the algorithm will not be able to visit all the nodes. The correct way to initialize X is as follows: 'X={all $v \in V$}'~~
This statement is also false. If we initialized the data structure X with all the nodes, then the program would never enter the while loop (as the condition would not be satisfied). Consequently, the algorithm would be incorrect as it would not be able to find the shortest path.

☑ The while loop (in line 7) is iterating over the wrong edges. It should iterate over the outgoing edges from v that have not been visited yet, therefore they are not included in X. The correct line 7 is: 'while there is an edge (v, w), such that $v \in X$, $w \notin X$ do:'
This statement is true. In fact, with $w \in X$ this while loop would exit immediately and the algorithm would terminate without finding the shortest path. Consider what happens in the very first iteration of the while loop: the only node included in X is the starting node s, therefore no such edge ($v \in X$, $w \in X$) even exists.

Solution to 5.6 To answer this question one must understand the key features of how the A* algorithm uses the heuristic function to improve over the Dijkstra shortest path algorithm.

The basic intuition behind the Dijkstra shortest path algorithm is that it will always choose the closest next neighbor. Such move might not be good in the long run and might force the algorithm to explore a larger portion of the graph than necessary. The A* combines the information on the closest next neighbor with a heuristic function that represents the most promising neighbor to eventually reach the target. So A* might choose an edge that is not going to the closest next neighbor but such a choice might pay off in the long run. If the heuristic function is cleverly chosen, then the A* algorithm will perform better compared to Dijkstra, in the best case only the nodes on the shortest path. However, if the heuristic function is poorly chosen, the A* algorithm may need to visit the same number of vertices as the Dijkstra algorithm or even more. One can find some experimental comparison of various alternatives for actual roads in [**?**].

The notion of 'cleverly chosen' can be formally defined with the notion of consistent (sometimes called feasible) heuristic function (w, t) where w is the possible candidate node and t is the target node. A heuristic function is consistent if for each node v, all neighbour vertices w_i that are closer to the target node will have a smaller or equal heuristic value $h(w_i, t) \leq h(v, t)$ and all neighbour vertices that are further away from the target node will have a strictly greater heuristic value $h(w_j, t) > h(v, t)$. Given such a heuristic function, A* is guaranteed to return an optimal solution.

The choice of heuristic function heavily depends on the specific problem and properties of the graphs on the input. A good example is using the Manhattan distance for a grid-like graph with all edge weights equal to 1 (as seen in the lecture of week 5). Manhattan distance is the distance between two points measured along axes at right angles. In a plane with p_1 at (x_1, y_1) and p_2 at (x_2, y_2), the Manhattan distance is $|x_1 - x_2| + |y_1 - y_2|$. In this case, the combined cost of Dijkstra score and the Manhattan distance will ensure that the algorithm continues exploring the next node in the direction of the target node.

A good example of a 'poorly chosen' heuristic when the weight of the edges represent the time to traverse a road is the very same Manhattan distance from the GPS coordinate in a mountainous region such the Alps, the Norwegian Fjords, the Rocky Mountains or the Sequoia National Park in the USA. A heuristic that works extremely well for flat regions such as France, the Netherlands or the costs of California is extremely poor in a different contest.

The intuition is that the heuristic function will suggest to explore narrow and kinky roads up and down a steep mountain (where traversal time increases dramatically as speeds drops) because they are 'very close' (and thus promising in the long run according to the heuristics) rather than taking the high speed straight highway around the mountain (which however requires to go to a very far node in terms of GPS Euclidean distance). Come winter and the shortest path might become a path to nowhere. This is well known to 'the locals' who do not follow Google Maps.

☑ The length of node b is not calculated correctly (in line 11) because it is not taking the heuristic estimate into account. It should be instead calculated as follows: $\text{len}(b) = \text{len}(a) + h(b, t)$
This statement is true because if we want to make any use of the heuristic function (assuming it is a "good one") then we need to add it to the combined score, which becomes the length of a current node. Since node b has been chosen as the optimal next node, we need to update its length so that the exploration of its neighbors (in the next iteration of the while loop) is accurate.

☐ ~~The exit condition is incorrect (in line 8), because this way the while loop will always exit the first time it is executed. It should check for inequality as follows: if $(v \neq t)$ then exit~~

This statement is false as this would result in an incorrect algorithm that would never find the shortest path. Actually, if we were to exit when the current node is not equal to the target node, then this program would exit immediately upon considering the starting node s (except if the starting node is actually the target node, but this would be a special case and it was not specified in the exercises).

☐ ~~X is incorrectly initialized (in line 5). It should contain all the vertices in the graph otherwise the algorithm will not be able to visit all the nodes. The correct way to initialize X is as follows: X={all $v \in V$}~~
This statement is also false. If we initialized the data structure X with all the nodes, then the program would never enter the while loop (as the condition would not be satisfied). Consequently, the algorithm would be incorrect as it would not be able to find the shortest path.

☐ ~~The value calculated for each outgoing edge from current node v is incorrectly calculated (in line 9). It should be calculated as the length of the current node v plus the heuristic estimate of how far the current node v is to the target (and not the next node w). Therefore, the following is correct: $(a, b) =$ minimize len$(v) + h(v, t)$~~
This statement is also false. As explained above, the combined cost is comprised of the Dijkstra score of the current node (i.e., cost endured so far) and the heuristic estimate of the next (not yet explored) node. Therefore the heuristic function takes as parameters the candidate next node (which has not yet been visited, and is thus $\notin X$) and the target node. If we were to keep adding the heuristic estimate of the current node (v), then the algorithm could choose a less optimal next neighbor, and consequently the performance gains of A* (over Dijkstra) could be lost.

TYPICAL MISTAKE 6.7 *A typical mistake about the A* algorithm is to assume that a heuristic function is always consistent or always not consistent irrespective of the type of graphs.*

The consistency of the heuristic function instead depends on the particular graph and it might even vary for the particular target node. See the example above.

TYPICAL MISTAKE 6.8 *Another typical mistake about the A* algorithm is to assume that we can make the algorithm work with a heuristic function that depends on only one among the candidate node and the target node.*

The whole point of the heuristic function is to estimate how far is the candidate node from the target because it is used to select the 'most promising neighbor to eventually reach the target'. If the function doesn't have the target as an argument it cannot discriminate between far and near targets and if it doesn't have the node as an argument it cannot discriminate between the neighbors.

Solutions to 5.7 This question presents the same problem as 5.8. The only difference is that this question asks to mark the false statements. For detailed solution explanation see solution to problem 5.8. Therefore the following options should be selected:

☑ Regardless of the chosen heuristic function, the A* algorithm will always outperform Dijkstra.
This statement is false, therefore it should be selected. The choice of the heuristic is a determining factor in the actual performance of A*, since if the heuristic is incorrectly chosen, A* may even exhibit worse running time compared to Dijkstra, as it may explore the graph in the opposite direction than the target. If such an unfortunate heuristic is chosen, A* may visit nodes in the graph that Dijkstra does not because it may already find the shortest path and terminate beforehand.

☐ ~~The running time of A* can (in some cases) be similar to the running time of Dijkstra.~~
This is true, therefore this statement should not be selected. This is particularly true when the heuristic function is equal to 0 (essentially, when there is no heuristic estimate).

☑ The A* algorithm will never outperform the Dijkstra algorithm.
This statement is false, therefore it should be selected. If implemented correctly, A* can (and does) outperform Dijkstra by visiting fewer nodes and finding the shortest path to the target faster. The performance gains are even more apparent for dense graphs with similar edge weights.

☑ A* is guaranteed to return an optimal solution if the input graph has a large number of nodes and very few edges (the graph is sparse).
This statement is false, therefore it should be selected. A* is guaranteed to return an optimal solution if the heuristic function is consistent.

Solution to 5.8 To answer this question one must understand the key features of on the Dijkstra shortest path algorithm because the A* algorithm combines a greedy, local search based on a heuristic (hopefully good but possibly bad in practice) with a global optimization (surely correct but possibly inefficient in practice) that is based on the Dijkstra shortest path algorithm.

When n denotes the number of nodes and m denotes the number of edges in a directed, weighted graph with no negative weights the complexity of the Dijkstra algorithm without an efficient data structure is $O(m \cdot n)$ as explained in [**?**, Section 9.4, part II],

When a heap data structure is used (as described in this exercise) the complexity of the Dijkstra algorithm can be improved to $O((m + n) \cdot \log n)$ because the heap maintains the node with a minimum Dijkstra score, which enables the algorithm to visit the graph more efficiently. See [**?**, Section 10.4, Part II] for further details.

As mentioned in the exercise the A* algorithm uses a heap to store the unprocessed node where the priority key used for the heap is the minimum combined cost of the Dijkstra score and the heuristic function. The heuristic function $h(w, t)$, where w is the next node to be explored and t is the target node, is equal to 0. This means that the heuristic function takes $O(1)$ and is basically useless.

The next net effect of this combination is that A* is actually only considering the Dijkstra score using a heap. Hence, the complexity of A* with a heuristic function equal to zero is equivalent to that of the Dijkstra algorithm as explained during the A* lecture (Week 5).

☐ ~~Regardless of the chosen heuristic function, the A* algorithm will always outperform Dijkstra.~~
This statement is false, as explained above, the choice of the heuristic is a determining factor in the actual performance of A*, since the heuristic is incorrectly chosen, A* may even exhibit worse running time compared to Dijkstra, as it may explore the graph in the opposite direction than the target. If such an unfortunate heuristic is chosen, A* may visit nodes in the graph that Dijkstra does not because it may already find the shortest path and terminate beforehand.

☑ The running time of A* can (in some cases) be similar to the running time of Dijkstra.
This is true. As explained above, this is particularly true when the heuristic function is equal to 0 (essentially, when there is no heuristic estimate).

☐ ~~The A* algorithm will never outperform the Dijkstra algorithm.~~
This statement is false. If implemented correctly, A* can (and does) outperform Dijkstra by visiting fewer nodes and finding the shortest path to the target faster. The performance gains are even more apparent for dense graphs with similar edge weights.

☐ ~~A* is guaranteed to return an optimal solution if the input graph has a large number of nodes and very few edges (the graph is sparse).~~
This statement is false. A* is guaranteed to return an optimal solution if the heuristic function is consistent.

TYPICAL MISTAKE 6.9 *A typical mistake is responding to the question about the complexity of an algorithm based on a specific configuration of the input.*

The claim about the complexity of a graph traversal algorithm must hold for all possible initial configurations. For example a graph is fully connected (every node is connected to every other node) it has $m = n^2$ edges. If the graph is made only by a simple cycle going through each node once it would have only $m = n$ edges. The complexity of an edge traversal algorithm that requires $O(m)$ operation can also be characterized as $O(n^2)$ in the first case or $O(n)$ in the second case. Choosing either $O(n^2)$ or $O(n)$, the bound would be badly wrong for the other case.

Solution to 5.9 This question presents the same problem as 5.10. The only difference is that this question asks to mark the false statements. For detailed solution explanation see solution to problem 5.10. Therefore the following options are false and should be selected:

☑ The complexity of the A* algorithm in case of the described implementation is $O(n)$.

☑ The complexity of the A* algorithm in case of the described implementation is $O(nm)$.

☑ The complexity of the A* algorithm in case of the described implementation is $O(nlog(n))$.

☐ ~~The complexity of the A* algorithm in case of the described implementation is $O((n + m)log(n))$.~~

Solution to 5.10 To answer this question one must understand the key features of on the Dijkstra shortest path algorithm because the A* algorithm combines a greedy, local search based on a heuristic (hopefully good but possibly bad in practice) with a global optimization (surely correct but possibly inefficient in practice) that is based on the Dijkstra shortest path algorithm.

When n denotes the number of nodes and m denotes the number of edges in a directed, weighted graph with no negative weights the complexity of the Dijkstra algorithm without an efficient data structure is $O(m \cdot n)$ as explained in [**?**, Section 9.4, part II],

When a heap data structure is used (as described in this exercise) the complexity of the Dijkstra algorithm can be improved to $O((m + n) \cdot \log n)$ because the heap maintains the node with minimum Dijkstra score, which enables the algorithm to visit the graph more efficiently. See [**?**, Section 10.4, Part II] for further details.

As mentioned in the exercise the A* algorithm uses a heap to store the unprocessed node where the priority key used for the heap is the minimum combined cost of the Dijkstra score and the heuristic function. The heuristic function $h(w, t)$, where w is the next node to be explored and t is the target node, is equal to 0. This means that the heuristic function takes $O(1)$ and is basically useless.

The next net effect of this combination is that A* is actually only considering the Dijkstra score using a a heap. Hence, the complexity of A* with a heuristic function equal to zero is equivalent to that of the Dijkstra algorithm as explained during the A* lecture (Week 5).

☐ ~~$O(n)$~~

This answer should be ruled out even without knowledge of either A* or Dijkstra because every edge needs to be potentially considered at least once to compute a shortest path. The proposed complexity would be true only for some graphs, but a statement on the complexity of an algorithm must hold for all possible instances.

☐ ~~$O(n \cdot n)$~~

As we mentioned, this would be the complexity of a Dijkstra implementation without an heap but we have been told that the algorithm uses an heap to store the priority key based on minimum Dijkstra score (since the heuristic function is constant and thus no therefore contribute to the minimum).

☐ ~~$O(n \cdot \log(n))$~~

The incorrectness of this possible answer rests on the same reasoning used behind the first answer: every edge needs to be traversed at least once to compute the shortest path. Hence a bound based only on the number of nodes that is less than $O(n^2)$ (the worst case), might only be true on some graphs.

☑ $O((n+m)log(n))$

This is the case that we have discussed above: the complexity of A* algorithm with a heuristic function fixed to the constant value 0 is equivalent to the complexity of Dijkstra's algorithm.

TYPICAL MISTAKE 6.10 *A typical mistake is responding to the question about the complexity of an algorithm based on a specific configuration of the input.*

The claim about the complexity of a graph traversal algorithm must hold for all possible initial configurations. For example a graph is fully connected (every node is connected to every other node) it has $m = n^2$ edges. If the graph is made only by a simple cycle going through each node once it would have only $m = n$ edges. The complexity of a edge traversal algorithm that requires $O(m)$ operation can also be characterized as $O(n^2)$ in the first case or $O(n)$ in the second case. Choosing either of $O(n^2)$ or $O(n)$ the bound would be badly wrong for the other case.

Solution to 5.11 This question presents the same problem as 5.12. The only difference is that this question asks to mark the true statements (instead of false). For detailed solution explanation see solution to problem 5.12. Therefore the following options are true and should be selected:

☑ A* algorithm can be implemented using different heuristic functions.

☐ ~~The A* algorithm will always outperform the Dijkstra algorithm.~~

☑ A* is guaranteed to return an optimal solutions if the heuristic function is consistent.

☑ The source and target vertex must be known for this algorithm to work.

Solution to 5.12 To answer this question one must understand the key features of how the A* algorithm uses the heuristic function to improve over the the Dijkstra shortest path algorithm.

The basic intuition behind the Dijkstra shortest path algorithm is that it will always choose the closest next neighbor. Such move might not be good in the long run and might force the algorithm to explore a larger portion of the graph than necessary. The A* combines the information on the closest next neighbor with an heuristic function that represent the most promising neighbor to eventually reach the target. So A* might choose an edge that is not going to the closest next neighbor but such choice might pay off in the long run. If the heuristic function is cleverly chosen, then the A* algorithm will perform better compared to Dijkstra, in the best case only the nodes on the shortest path. However, if the heuristic function is poorly chosen, the A* algorithm may need to visit the same number of vertices as the Dijkstra algorithm or even more. One can find some experimental comparison of various alternatives for actual roads in [**?**].

The notion of 'cleverly chosen' can be formally defined with the notion of consistent (sometimes called feasible) heuristic function (w, t) where w is the possible candidate node and t is the target node. An heuristic function is consistent if for each vertex v, all neighbour vertices w_i that are closer to the target vertex will have a smaller or equal heuristic value $h(w_i, t) \leq h(v, t)$ and all neighbour vertices that are further away from the target vertex will have a strictly greater heuristic value $h(w_j, t) > h(v, t)$. Given such a heuristic function, A* is guaranteed to return an optimal solution.

The choice of heuristic function heavily depends on the specific problem and properties of the graphs on the input. A good example is using the Manhattan distance for a grid-like graph with all edge weights equal to 1 (as seen in the lecture of week 5). Manhattan distance is the distance between two points measured along axes at right angles. In a plane with p_1 at (x_1, y_1) and p_2 at (x_2, y_2), the Manhattan distance is $|x_1 - x_2| + |y_1 - y_2|$. In this case, the combined cost of Dijkstra score and the Manhattan distance will ensure that the algorithm continues exploring the next vertex in the direction of the target vertex.

A good example of a 'poorly chosen' heuristic when weight of the edges represent the time to traverse a road is the very same Manhattan distance from the GPS coordinate in a mountainous

region such the Alps, the Norwegian Fjords, the Rocky Mountains or the Sequoia National Park in the USA. A heuristics that works extremely well for flat regions such as France, the Netherlands or the costs of California is extremely poor in a different contest.

The intuition is that the heuristic function will suggest to explore narrow and kinky roads up and down a steep mountain (where traversal time increases dramatically as speeds drops) because they are 'very close' (and thus promising in the long run according to the heuristics) rather than taking the high speed straight highway around the mountain (which however requires to go to a very far node in term of GPS Euclidean distance). Come winter and the shortest path might become a path to nowhere. This is well known to 'the locals' who do not follow Google Maps.

To determine the correct answer remember that the question asked which statements are false.

☐ ~~A* algorithm can be implemented using different heuristic functions.~~
An heuristic function is usually defined for each and every concrete algorithm implementation and therefore the sentence cannot be false.

☑ The A* algorithm will always outperform the Dijkstra algorithm.
This is sentence is false (and therefore the answer is correct) because the heuristic function has an impact on how well the algorithm will perform. Only if the heuristic function that is consistent for the specific problem domain the statement would be true. As we have illustrated above, even the very same heuristic such as Manhattan distance might not be good in a different context.

☐ ~~A* is guaranteed to return an optimal solutions if the heuristic function is consistent.~~
As we have illustrated above, this is always the case. This sentence is true and therefore not a correct answer as the question asked which sentence is false. The intuitive reason is that if the heuristic is consistent one can, for the purpose of identifying a shortest path, replace the weight of each edge with the difference between the heuristics functions of the vertices of the edge.

☐ ~~The source and target vertex must be known for this algorithm to work.~~
This statement is true for every shortest path algorithm between a source s and a target t. If the algorithm does not know the source and the target vertex, how can it return the shortest path between them? Further, on A* the heuristic function, described above, is a function of two arguments $h(w, t)$: the next node and the target. Therefore how one can estimate the distance from the target if one doesn't have the target?

TYPICAL MISTAKE 6.11 *A typical mistake about the A* algorithm is to assume that a heuristic function is always consistent or always not consistent irrespective of the type of graphs.*

The consistency of the heuristic function instead depends on the particular graph and it might even vary for the particular target node. See the example above.

TYPICAL MISTAKE 6.12 *Another typical mistake about the A* algorithm is to assume that we can make the algorithm work with an heuristic function that depends on only one among the candidate node and the target node.*

The whole point of the heuristic function is to estimate how far is the candidate node from the target because it is used to select the 'most promising neighbor to eventually reach the target'. If the the function doesn't have the target as an argument it cannot discriminate between far and near targets and if it doesn' have the node as an argument it cannot discriminate the between the neighbors.

6.7 Graph algorithms: Minimum Spanning Tree (MST)

Solution to 5.13 To answer this question, one must recall the problem of the minimum spanning tree. The minimum spanning tree of a (possibly weighted) graph is represented by the set of edges that connects all vertices such that the sum of the weights of the edges is the minimum among all

sets of edges that connect all nodes. A minimum spanning tree does not contain cycles, because a cycle would yield an unnecessary additional cost since it does not span a new node.

If a node can be connected by two or more edges to the remaining part of the tree with the same minimum cost, it does not matter which edge will be in the minimum spanning tree, because the cost to span that node will be the same. Such non-determinism means that, for a given graph, there might be more than one minimum spanning tree, if there are edges with the same costs. If all edges have different costs, it means that they can be ordered and therefore any algorithm for finding a minimum spanning tree of a graph, with any implementation, will always choose the next unique, cheapest edge. So the minimum spanning tree of the graph will be unique.

So the correct answers are as follows:

☐ ~~A minimum spanning tree for a connected graph has exactly V-2 edges, V being the number of vertices in the graph.~~

This statement is false because it actually requires V-1 edges to form a minimum spanning tree. Two vertices are connected with one edge, and every other vertice is spanned with one edge, which is why a minimum spanning tree has exactly V-1 edges.

☑ The most straightforward implementation of Prim's algorithm (e.g., without the use of heap) performs better when the graph on the input is dense, while the most straightforward implementation of the Kruskal algorithm performs better when the input graph is sparse.

This statement is true as was also explained during the lecture of week 6. The complexity of a straightforward implementation of Prim's algorithm (without using the heap data structure to improve performance) is $O(n*m)$, which is the same as a straightforward implementation of Kruskal. However, since Kruskal builds a forest (compared to Prim's building a tree), it will find the solution faster compared to Prim's when executed on sparse graphs. The opposite is true for dense graphs.

☐ ~~Prim's and Kruskal's algorithms will never return the same minimum spanning tree.~~

This statement is false as explained above in detail. If the given graph has only one possible minimum spanning tree, then both algorithms will surely return the same solution.

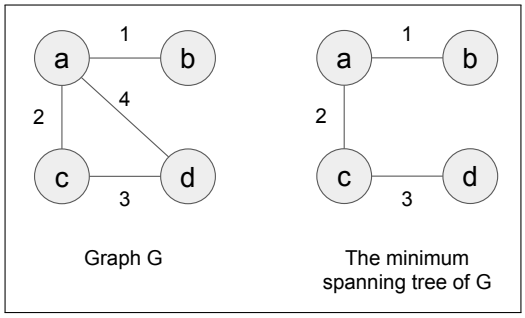
☐ ~~The complexity of Prim's algorithm implemented with the heap data structure and Kruskal's algorithm is the same. Both have the complexity of $O(m*n)$, where m is the number of edges and n is the number of nodes.~~

This statement is false because the running time of Prim's algorithm can be improved by using the heap data structure to run in $O((m+n)*logn)$ time (as shown in class).

TYPICAL MISTAKE 6.13 *A typical mistake is making confusion between the shortest path between vertices, where not all vertices of the graph are included, and the minimum spanning tree of a graph, where all vertices are included.*

The minimum spanning tree does include one path between each pair of nodes, but such a path might not be the minimal for the two particular nodes. Consider graph G and its minimum spanning tree below. The path between "a" and "d" of graph G costs 4, but, in the minimum spanning tree of G, the path costs 5 because of the edges (a, c) and (c, d). The intuition for this discrepancy is the difference in objectives. The minimum spanning tree has a global objective whereas the all pairs shortest paths is a set of individual pairwise objectives each of which is optimize as an independent functions. Such functions may be all computed at once (as the Dijkstra all-pairs shortest paths algorithm does), but there is no guarantee that the global optimum is the same as the sum of the pairwise minima. Therefore, the minimum spanning tree of a graph cannot be used to find the shortest path between two vertices, and shortest path algorithms cannot be used to find minimum spanning trees.

Solution to 5.14 This question presents the same problem as 5.15. The only difference is that this question asks to mark the false statements. For detailed solution explanation see solution to problem 5.15. Therefore the following options are false and should be selected:



☐ ~~A minimum spanning tree for a connected graph has exactly V-1 edges, V being the number of vertices in the graph.~~

☐ ~~A graph where every edge weight is unique (there are no two edges with the same weight) has a unique minimum spanning tree.~~

☑ Prim's and Kruskal's algorithms will always return the same minimum spanning tree.

☑ The minimum spanning tree can be used to find the shortest path between two vertices.

Solution to 5.15 To answer this question, one must recall the problem of the minimum spanning tree. The minimum spanning tree of a (possibly weighted) graph is represented by the set of edges that connects all vertices such that the sum of the weight of the edges is the minimum possible among all set of edges that connect all nodes.

The set of nodes and the set of edges make a graph in itself, which is called the the spanning tree of the original graph. A minimum spanning tree does not contain cycles, because a cycle would yield an unnecessary additional cost since it does not span a new vertex.

If a vertex can be connected by two or more edges to the remaining part of the tree with the same minimum cost, it does not matter which edge will be in the minimum spanning tree, because the cost to span that vertex will be the same. Of course such non-determinism means that a minimum spanning tree might not be unique if there are edges with the same costs.

If all edges have a different costs it means that they can be ordered and therefore any algorithm for finding a minimum spanning tree of a graph, with any implementation, will always choose the (unique) cheapest edge at any given time. So the minimum spanning tree will be unique. While a graph with vertices with unique edges will always have a unique minimum spanning tree (see Chapter 15, Part III [**?**]) the reasoning above is not generally correct: the algorithm described above (pick always the cheapest edge) is essentially Prim's algorithm. In this case, we know that it uniquely outputs the minimum spanning tree from the book [**?**, [Chapter 15, part III]. We illustrate after the answers a possible way to prove it without making a reference to the specific algorithm.

So the correct answers are as follows:

☑ A minimum spanning tree for a connected graph has exactly V-1 edges, V being the number of vertices in the graph.

A minimum spanning tree of a graph connects all the vertices without cycles. To connect two vertices, one edge is necessary. To connect three vertices, after the first two node are connected, only one additional edge is necessary, to connect the third edge to either the first or the second node. Eventually, to connect V vertices, then V-1 edges are necessary.

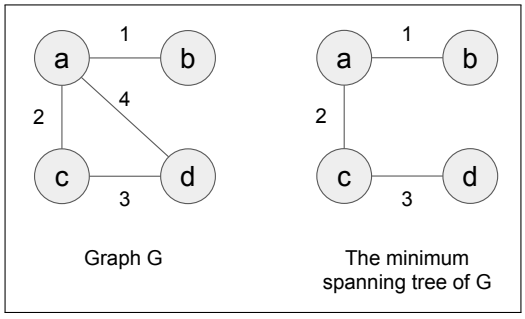
☑ A graph where every edge weight is unique (there are no two edges with the same weight) has a unique minimum spanning tree.

From the book [**?**] we know that this type of graphs have an unique minimum spanning tree. The only way that, given a graph, there could be multiple minimum spanning trees is if there

were more than one edge with the same cost because then one particular implementation could choose an edge and another implementation could choose the other edge.

☐ ~~Prim's and Kruskal's algorithms will always return the same minimum spanning tree.~~
The returned minimum spanning tree depends on the input graph and the implementation of the algorithms. However, only if the graph has unique edges (that is, there are no two or more edges with the same cost), there exists only one minimum spanning tree (see previous question). Otherwise, the question's statement does not hold for any graph, because graphs with non-unique edges have more than one minimum spanning tree, and the implementations of any algorithm (Prim's, Kruskal's, or another) could return any of them.

☐ ~~The minimum spanning tree can be used to find the shortest path between two vertices.~~
This is false because a minimum spanning tree might not contain the cheapest path between the two vertices. Consider graph G and its minimum spanning tree below. If one wants to find the shortest path between the vertices "a" and "d" of graph G, one would find the path with cost 4, which is the path containing only the edge that connects "a" and "d". However, if one uses the minimum spanning tree of G to do so, one will not find such a path because it does not belong to the minimum spanning tree. Instead, one would find a path with cost 5, which contains the edges (a, c) and (c, d). Therefore, the minimum spanning tree of a graph cannot be used to find the shortest path between two vertices.



TYPICAL MISTAKE 6.14 *A typical mistake is making confusion between the shortest path between vertices, where not all vertices of the graph are included, and the minimum spanning tree of a graph, where all vertices are included.*

Even in the case of all pairs shortest paths the two cases do no coincide. The minimum spanning tree does include one path between each pair of nodes but such path might not be minimal for the two particular nodes. The intuition for this discrepancies is the difference in objectives. The minimum spanning tree has a global objective whereas the all pairs shortest paths is a set of individual pairwise objectives each of which is optimize as an independent functions. Such functions may be all computed at once (as the Dijkstra all-pairs shortest paths algorithm does) but there is no guarantee that the global optimum is the same as the sum of the pairwise minima.

TYPICAL MISTAKE 6.15 *Another typical mistake is assuming that a property of a solution for a specific algorithm can be transformed into a general property of any algorithm.*

If we have been able to prove that a solution returned by some algorithm by using some assumptions on how the algorithm works we cannot conclude anything for every algorithms which do not make the same choices. The very example of A* and Dijkstra shortest path algorithms illustrates it: A* does not always chose the next shortest edge.

In our case, if we consider a graph with $O(n^2)$ all different edges we can imagine that there might several ways in which several sets of $n-1$ numbers (the costs of the picked edges) selected from those n^2 sum up to the same number (the cost of the minimum spanning tree). So how a unique minimum spanning tree can even exist?

The correct way to prove properties of solutions irrespective of the algorithm that produces them (i.e. the algorithm is a block box) is by induction or by contradiction. The first method of

proof assumes that property holds for a graph with n nodes (there is a unique minimum spanning tree when all edges have different costs) and if we add the $n+1$-th node the property also holds. The second one assume that the property is violated (there are two different minimum spanning trees) and then derive a logical contradiction with some parts of the solution.

In the book there are several proofs by induction and a guide [**?**, Appendix A,Part I] so we illustrate the idea here by proving by contradiction that a minimum spanning tree is unique if edges are all different. This proof is simpler than the proofs in the book because we are not forced to make any assumption about the particular order used by the algorithm in which edges are chosen. For what we know, it could have selected them in the most random way. We just look at the final products.

So suppose that there is a graph in which the costs of all edges are different and there are two minimal spanning trees T_1 and T_2 with exactly the same total cost but some edges that only belong to one of them. We can now sort the edges of both trees in order of increasing cost and start comparing these two sequences one edge at the time. Notice that we don't make any assumption on the order in which the algorithm producing T_i has selected those edges.

Since all edges of the original tree are different from each other, if an edges of sequence 1 and an edge in sequence 2 have the same cost then they actually are the same edge. However, the two trees are different, so there must be a k such that after the first $k-1$ identical and cheapest edges in both trees, the k-th edge of T_i in the sequence is not an edge of T_j.

Suppose that the edge $e_k^{(1)}$ in T_1 is the cheaper one with costs $c_k^{(1)}$ (if not then just swap the two tree and consider T_2 as the new T_1). Then we can add this edge to T_2. Since T_2 was a minimum spanning tree there must be now a cycle in $T_2 \cup \{e_k^{(1)}\}$.

Since there is a cycle in $\cup\{e_k^{(1)}\}$ we can now remove another edge from this cycle and obtain another spanning tree T_2'. If the edge that we removed has a higher cost than $c_k^{(1)}$, then we have a contradiction because T_2' has a strictly lower cost than T_2 which was supposed to be a minimum spanning tree.

Can it be that all other edges in the cycle created by adding this edges $e_k^{(1)}$ have a lower costs than $c_k^{(1)}$? At this point we observe that all edges with a cost lower than $c_k^{(1)}$ are also edges in T_1: $e_k^{(1)}$ was the first edge that was different among the two trees. So if there is cycle in T_2 plus this k-th edge there must be already a cycle in T_1 which is against the assumption that T_1 was a spanning tree (let alone a minimum one).

6.8 Operations on Graphs

Solution to 5.16 One can simply run Prim's algorithm on the graph: repeatedly picks the cheapest edge from the already spanned nodes that does not create a cycle. This means that the edge (0-1) is selected first, then (1-3), and then (3-4). The edge (4-1) can not be selected because it would create a cycle, so then the edge (2-3) is selected. The total cost of the minimum spanning tree is 110. Since all edges are unique (that is, all edges have different costs), the minimum spanning tree is unique, so there is only one correct answer for this question.

☐ ~~$(0-1),(1-4),(1-2),(1-3)$~~

☐ ~~$(0-1),(1-4),(2-3),(1-3)$~~

☑ $(0-1),(1-3),(2-3),(3-4)$

☐ ~~$(0-1),(0-2),(1-4),(1-2)$~~

Solution to 5.17 This question presents the same problem as 5.18. The only difference is that this question asks to mark the false statements. For detailed solution explanation see solution to problem 5.18. Therefore the following options are false and should be selected:

☑ The following edges form the minimum spanning tree of the given graph: $(a-c),(c-d),(d-b),(d-b)$.

☑ The following edges form the minimum spanning tree of the given graph: $(c-a),(a-d),(d-b),(d-e)$.

☐ ~~The following edges form the minimum spanning tree of the given graph: $(a-d),(d-e),(d-b),(d-e)$.~~

☑ The following edges form the minimum spanning tree of the given graph: $(c-a),(a-d),(d-c),(d-b),(d-e)$.

Solution to 5.18 Since all edges are different (and the graph is small) one can simply run Prim's algorithm: repeatedly picks the cheapest edge that does not create a cycle. This means that (c-d) is selected first, then (a-d). The edge (a-c) cannot be selected because it would create a cycle, so then (d-e) is selected and finally (b-d) is selected for a total cost of 56. When all edges are unique (that is, all edges have different costs), as in this case, the MST is unique so we don't need to consider other options.

☐ ~~(a-c)(c-d)(d-b)(d-b)~~
This is wrong because an edge is selected twice and the node e is not connected to the tree so it would not even be a spanning tree.

☐ ~~(c-a)(a-d)(d-b)(d-e)~~
This is a spanning tree but it is not minimal since it has a cost of 61.

☑ (a-d)(d-c)(d-b)(d-e)
The edges are not presented in the order in which the algorithm mentioned above would have selected them, but this is immaterial to the definition of minimum spanning tree.

☐ ~~(c-a)(a-d)(d-c)(d-b)(d-e)~~
This is wrong because there is a cycle (a-c-d) so it would not even be a spanning tree.

6.9 0/1 Knapsack problem

Solution to 5.19 To answer this question, one needs to understand the definition of a 0/1 knapsack problem. The problem is formulated as follows: Given a set of objects (v_i, w_i), what is the maximum value (v) we can achieve, such that the sum of the object weights (w) does not exceed capacity c? In a simple example such as the one given in this exercise, one could simply try all the combinations and find the solution. Such a brute-force approach surely does not work for 0/1 knapsack problem of bigger size (e.i., with more items to consider and a larger capacity). The knapsack problem is a classical problem that can be elegantly solved using dynamic programming (as shown in class). The following statements are true:

☑ The solution to the problem is choosing items with weight 5 and 3 with the combined weight of 8 and combined profit of 6.
This is a correct statement because no other combination of items will result in a greater profit while remaining under the capacity.

☐ ~~The solution to the problem is choosing the first three items with the combined weight of 8 and combined profit of 9.~~
This statement is false because while the profit is indeed 9, the combined weight is 12, which goes above the knapsack capacity of 8.

☐ ~~The solution to the problem is choosing the first two items with the combined weight of 7 and combined profit of 5.~~
This statement is also false. This option is a valid candidate for the solution, however a better choice exists with the combined profit of 6.

☐ ~~The solution to the problem is choosing items with weight 4 and 5 with the combined weight of 8 and combined profit of 5.~~

This statement is false, the profit is indeed 5 but the combined weight of these two items is 11, which is greater than the knapsack capacity.