Delft University of Technology
EEMCS Faculty

CSE1305 Algorithms & Data Structures

Resit Analysis Exam

05 April 2022, 09:00–11:00

Examiners:

Examiner responsible: Joana Gonçalves and Ivo van Kreveld
Examination reviewer: Stefan Hugtenburg

Parts of the examination and determination of the grade:

Exam part	Number of questions	Question specifics	Grade (%)	Grade (points)
Multiple-choice	20 questions (equal weights)	One correct answer per question	50%	$5 \cdot \frac{\text{score}}{19}$
Open questions	4 questions (different weights)	Multiple parts	50%	$5 \cdot \frac{\text{score}}{20}$

Use of information sources and aids:

- A hand-written double-sided A4 cheat sheet can be used during the exam.
- No other materials may be used, including but not limited to books, lecture slides in any form, or devices such as laptops and phones.
- Scrap paper sheets are provided at the beginning of the exam. Additional scrap paper can be requested.

General instructions:

- Solve the exam on your own. Any form of collaboration is prohibited.
- You cannot leave the examination room during the first 30 minutes.
- If you are eligible for extra time, place the "Verklaring Tentamentijd Verlenging" on your desk, together with your student card.

Instructions for writing down your answers:

- You should answer the questions using the provided answer sheets.
- Write your name and student number on every sheet of paper.
- **For multiple-choice questions**, mark the answers on this exam paper first, and copy them to the answer form after revising.
- **For open questions**, provide all requested information and always give an explanation. Avoid irrelevant data, it could lead to deductions.
- **For proofs**, make sure your proof is properly structured and sufficiently explained. Statements or steps without justification could lead to point deductions.

This page is intentionally left blank.

Multiple-choice questions (50%, 19 points)

1. (1 point) Given two functions $f(n)$ and $g(n)$ where $f(n)$ is $\Omega(g(n))$. Which of the following statements about asymptotic algorithmic complexities must be **true**?

   A. $g(n)$ is $\mathcal{O}(f(n))$.

   B. $g(n)$ is $\Theta(f(n))$.

   C. $g(n)$ is **not** $\Omega(f(n))$.

   D. $f(n) > g(n)$ for some value of n.

Answer:

A. True. If $f(n)$ is at least the time complexity of $g(n)$, $g(n)$ is at most the time complexity of $f(n)$.

B. False. $f(n) = n^2$ and $g(n) = n$ is a counterexample.

C. False. $f(n) = n$ and $g(n) = n$ is a counterexample.

D. False. $f(n) = n$ and $g(n) = n + 1$ is a counterexample.

2. (1 point) The Java methods `sumA` and `sumB` both calculate the sum of the integer elements in array `arr`.

```
1  public static int sumA(int[] arr, int low, int high) {
2    if(low == high) {
3      return arr[low];
4    }
5    return sumA(arr, low, high - 1) + arr[high];
6  }
7
8  public static int sumB(int[] arr, int low, int high) {
9    if(low == high) {
10     return arr[low];
11   }
12   int mid = (high + low) / 2;
13   return sumB(arr, low, mid) + sumB(arr, mid + 1, high);
14 }
```

The problem size is the difference between `high` and `low`. What can we say about the tightest worst-case time and space complexity of methods `sumA` and `sumB`, expressed in big-Oh notation?

A. Method `sumA` has larger time complexity and larger space complexity than method `sumB`.

B. Method `sumA` has larger time complexity than method `sumB`, space complexities are the same for both methods.

C. Method `sumA` has larger space complexity than `sumB`, time complexities are the same for both methods.

D. The methods `sumA` and `sumB` have the same time and space complexities.

Answer: The time complexities are the same because we make a total of n calls to a method which has only operations which take constant time. The space complexity of `sumB` is lower because since it will have only $\log(n)$ calls on the stack at the same time, it can reuse the space from the calls that are already completed.

3. (1 point) Consider implementations of a stack and a queue data structures using a singly-linked list. Where do you need to add and remove elements to get a correctly working data structure with the optimal time complexities for both methods?

   A. **stack:** add to head, remove from head **queue:** add to head, remove from tail
   B. stack: add to head, remove from head queue: add to tail, remove from head
   C. **stack:** add to tail, remove from tail **queue:** add to head, remove from tail
   D. **stack:** add to tail, remove from tail **queue:** add to tail, remove from head

   Answer: For a singly-linked list, removing the tail takes $\mathcal{O}(n)$ time, while removing the head takes $\mathcal{O}(1)$ time. Adding elements takes $\mathcal{O}(1)$ time regardless of whether elements are added to the head or the tail. Thus, to get an efficient data structure, elements should be removed from the head.

4. (1 point) Consider a **sorted** sequence of integers elements. The task is to insert a new element such that the sequence is kept sorted. What is the time complexity of the most time-efficient way to do this, using an array-based implementation and a doubly-linked list (DLL) implementation?

   A. **Array:** $\mathcal{O}(\log(n))$ **DLL:** $\mathcal{O}(\log(n))$
   B. **Array:** $\mathcal{O}(\log(n))$ **DLL:** $\mathcal{O}(n)$
   C. **Array:** $\mathcal{O}(n)$ **DLL:** $\mathcal{O}(\log(n))$
   D. Array: $\mathcal{O}(n)$ **DLL:** $\mathcal{O}(n)$

   Answer: For an array, it takes $\mathcal{O}(\log(n))$ time to find the place where the element needs to be added, but it takes $\mathcal{O}(n)$ time to add the element there. For a doubly-linked list, it takes $\mathcal{O}(n)$ time to find the place where the element needs to be added, and $\mathcal{O}(1)$ time to add the element there.

5. (1 point) Consider a dynamic array, where elements are inserted or removed always at the end. There can be arbitrary insertions and deletions in no particular order. When and how could we resize the array to make sure that the amortized time complexities of adding and removing one element are both $\mathcal{O}(1)$? Here n denotes the number of elements in the array, and C is the capacity of the array.

   A. When $n = C$, increase capacity to $n + 10$. When $n = C - 20$, decrease capacity to $n + 10$.
   B. When $n = C$, increase capacity to $n + 10$. When $n = C - 10$, decrease capacity to n.
   C. When $n = C$, increase capacity to $2 \cdot n$. When $n = \frac{1}{2} \cdot C$, decrease capacity to n.
   D. When $n = C$, increase capacity to $2 \cdot n$. When $n = \frac{1}{2} \cdot C$, decrease capacity to $\frac{3}{2} \cdot n$.

   Answer: To get an amortized time complexity of $\mathcal{O}(1)$, the capacity should be increased by an amount linearly dependent on n. This is the case in answer C and D. For answer C however, it is possible that elements are added and removed alternately such that the capacity is increased or decreased each time, taking $\mathcal{O}(n)$ time.

6. (1 point) Consider the following Java implementation of an iterator `ADSListIterator` below.

```
1  public class ADSListIterator<V> implements Iterator<V> {
2    LinkedList<V> list;
3    int index;
4
5    public ADSListIterator(LinkedList<V> list) {
6      this.list = list;
7      index = 0;
8    }
9
10   public boolean hasNext() {
11     return index < list.size();
12   }
13
14   public V next() throws NoSuchElementException {
15     if (!hasNext()) throw new NoSuchElementException();
16     return list.get(index++);
17   }
18 }
```

Is this a lazy or snapshot iterator? And what is the time complexity of the `next` method?

A. It is a lazy iterator. The time complexity of the `next` method is $\mathcal{O}(1)$.

B. It is a lazy iterator. The time complexity of the `next` method is $\mathcal{O}(n)$.

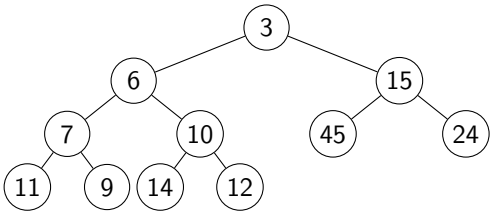
C. It is a snapshot iterator. The time complexity of the `next` method is $\mathcal{O}(1)$.

D. It is a snapshot iterator. The time complexity of the `next` method is $\mathcal{O}(n)$.

Answer: Since the `list` variable is just a reference, changing the list afer initializing the iterator will change the list in the iterator as well. Getting an element at a certain index in a linked list takes $\mathcal{O}(n)$ time.

7. (0 points) This question was removed from the exam.

8. (1 point) Consider the following heap:



Which nodes would have a different element after removing the minimum element from the heap?

A. The nodes currently containing 3, 6, 7 and 11.

B. The nodes currently containing 3, 6, 7 and 9.

C. The nodes currently containing 3, 6 and 10.

D. The nodes currently containing 3, 6, 7 and 12.

Answer: The element from the root would be removed and would be replaced by the last value, the 12. Then, it would be swapped downwards each time with the lower of the 2 childs, being 6, 7 and 9.

9. (1 point) Consider a sorting algorithm which first enqueues all elements of a sequence into a priority queue, and then dequeues all the elements from that priority queue. The priority queue uses an unsorted list implementation. What sorting algorithm does this description correspond to?

   A. This is a stable version of insertion sort.

   B. This is an unstable version of insertion sort.

   C. This is a stable version of selection sort.

   D. This is an unstable version of selection sort.

   Answer: Enqueuing elements doesn't change the order since the priority queue is not sorted. While dequeuing, the lowest element is taken each time and added to a new list. This corresponds to a stable version of selection sort.

10. (1 point) Consider the array (10,4,2,6,14,12,16,18,8). What is the content of the array after **two** iterations of the **in-place** heap sort algorithm to sort the array in **decreasing** order? Note: in case you need to apply heapify, you should apply the bottom-up algorithm with the $O(n)$ time complexity.

    A. (18,16,14,12,10,8,6,4,2)

    B. (2,4,10,6,14,12,16,18,8)

    C. (4,6,10,8,14,12,16,18,2)

    D. (6,8,10,18,14,12,16,4,2)

    Answer: After heapifying, the content of the array is (2,4,10,6,14,12,16,18,8). After the first iteration, it is (4,6,10,8,14,12,16,18,2). After the second iteration, it is (6,8,10,18,14,12,16,4,2).

11. (1 point) Consider a modified version of merge sort, which uses selection sort in the base case. How should the base case be defined such that this variant of the merge sort algorithm has a time complexity of $\mathcal{O}(n \cdot \log(n))$? If there are multiple correct options, choose the most general one.

    A. Immediately go into the base case, because selection sort is $\mathcal{O}(n \cdot \log(n))$.

    B. Go into the base case after k recursive calls (k is a constant greater than 0).

    C. Go into the base case when we have a list of size k or smaller (k is a constant greater than 0).

    D. Go into the base case when we have a list of size 2 or 1.

    Answer:

    The worst-case time to sort a list of length k using selection sort is $\Theta(k^2)$. Therefore, sorting n/k sublists, each of length k, takes $\Theta(k^2 \cdot n/k) = \Theta(nk)$ worst-case time.

    We have n/k sorted sublists, each of length k. To merge the n/k sorted sublists into one sorted list of length n, we take 2 sublists at a time and continue to merge. This will result in $\log(n/k)$ steps and we compare n elements in each step. Worst-case time to merge the sublists is thus $\Theta(n \log(n/k))$.

    This algorithm has time complexity as standard merge sort when $\Theta(nk + n\log(n/k)) = \Theta(n \log n)$.

12. (1 point) Say we have a list of n integers, with values ranging from 0 to 9. In the initial list, it is guaranteed that the integers from 0 to 4 are earlier in the list than the integers from 5 to 9 (for each integer i, if $i < 5$ then there exists no integer j earlier in the list such that $j > 4$). What is the fastest sorting algorithm to sort this list (for an arbitrarily large n)?

    A. Bucket sort

    B. Insertion sort

    C. Merge sort

    D. Quick sort

> **Answer:** Bucket sort takes $\mathcal{O}(n + N)$ time. In this case, $N = 10$, which means it takes $\mathcal{O}(n + 10)$ time, so $\mathcal{O}(N)$ time. This is faster than the other algorithms which take $\mathcal{O}(n^2)$ or $\mathcal{O}(n \log(n))$ time.

13. (1 point) What is the expected time complexity of the get, put and $remove$ methods with a hashmap with a load factor of $\lambda = 2$, for both separate chaining and linear probing? You may assume that the hash function is perfect, so each hash value in the range $[0, N - 1]$ will have an equal chance to be hashed to.

    A. Separate chaining: $\mathcal{O}(1)$, linear probing: $\mathcal{O}(n)$.

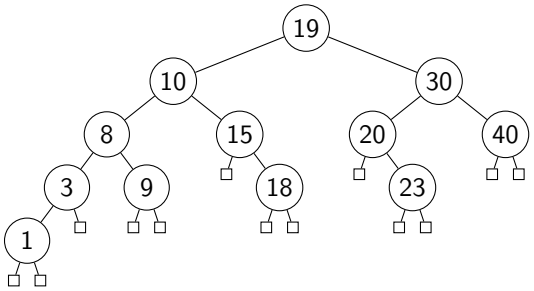
    B. Separate chaining: $\mathcal{O}(n)$, linear probing: $\mathcal{O}(n)$.

    C. Separate chaining: $\mathcal{O}(1)$, linear probing is not able to handle load factors above 1.

    D. Separate chaining: $\mathcal{O}(n)$, linear probing is not able to handle load factors above 1.

> **Answer:** The load factor is $\lambda = 2$, which means the expected values in each bucket is 2. Since this is a constant, the operations still take $\mathcal{O}(1)$ time. Linear probing is not possible, because it cannot have more values than the size of the array.

14. (1 point) Consider the following AVL tree:



    When deleting entry **19** from the tree, what trinode restructuring would occur? If the node of the entry to be deleted has non-null children, the node will be replaced with the in-order predecessor (i.e. the maximal entry in the left child of the entry).

    A. A single rotation trinode restructuring involving nodes 3, 8 and 10.

    B. A double rotation trinode restructuring involving node 3, 8 and 10.

    C. A single rotation trinode restructuring involving node 8, 10 and 18.

    D. A double rotation trinode restructuring involving nodes 8, 10 and 18.

> **Answer:** After deleting the value 19 from the root, it is replaced by 18. This means the node with value 10 has childs with a height difference of 2. Since the left child of 10 is the highest height and its left child is higher than its right child, a single rotation restructure occurs. After this, the tree is balanced.

15. (1 point) Given a corresponding (2,4) tree and red-black tree. What is **false**?

    A. The depth of the leaf nodes in the (2,4) tree is equal to the black depth of the leaf nodes in the red-black tree.

    B. The (2,4) tree has as many nodes as the red-black tree has black nodes.

    C. The (2,4) tree has as many 3-nodes and 4-nodes as the red-black tree has red nodes.

    D. The set of entries in the root of the (2,4) tree contains the entry found at the root of the red-black tree.

> **Answer:** For each 4-node a (2,4) tree has, a red-black tree has 2 red nodes. Thus, as soon as a (2,4) tree has at least 1 4-node, its corresponding red-black tree has more red nodes than the (2,4) tree has 4-nodes.

16. (1 point) Consider that we need a data structure to keep the scoreboard of a game, with entries stored as (key, value) pairs. We would like to be able to do frequent lookups of the top k players, and to do frequent lookups of all players with scores within a given range s. Note that the number of players with scores in a range s is typically much smaller than the size of the scoreboard. Which of the following data structures would be the most time-efficient for each of these operations?

    A. **Top k:** Hashtable and Heap **Score in range s:** Hashtable

    B. **Top k:** Hashtable and Heap **Score in range s:** Heap

    C. **Top k:** Heap and AVL tree **Score in range s:** Heap

    D. Top k: Heap and AVL tree **Score in range s: AVL tree**

> **Answer:** Lookups for the top k players is fastest with a heap or AVL tree, because we can avoid looking at most of the data. In a heap, we can start at the top and proceed downwards via the nodes with the smallest values. In an AVL tree, we can just look at the far left side of the tree. To get the all scores in a certain range, we can find the leaves where the minimum and maximum of this range is, and only look at the part of the tree between these leaves.

17. (1 point) Consider a graph G with n vertices and m edges implemented using an **adjacency map** and a list L of k vertices $[x_1, ..., x_k]$. The task is to determine whether this list of vertices forms a path i.e. whether for all $i \in [1, ..., k-1]$, the vertices x_i and x_{i+1} are adjacent. What is the tightest upper bound on the time complexity of this operation?

    A. $\mathcal{O}(k)$

    B. $\mathcal{O}(k \cdot n)$

    C. $\mathcal{O}(n \cdot m)$

    D. $\mathcal{O}\left(\sum_{v \in L} deg(v)\right)$

> **Answer:** Since the adjacent nodes of each node are stored as a hashmap, it takes $O(1)$ time to check whether the next node is adjacent. Since this has to be done for each node in a list with size k, this takes $\mathcal{O}(k)$ time.

18. (1 point) Consider a directed acyclic graph G and a topological sorting of that graph where node a appears before node b. What must be **true**?

    A. There is a directed path from a to b.

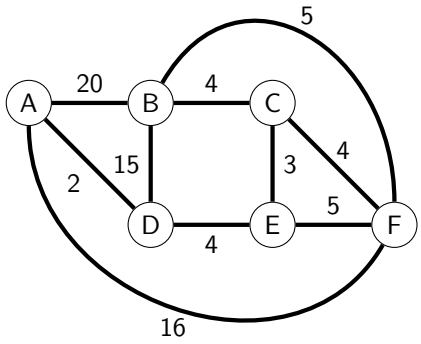
    B. There is no directed path from b to a.

    C. If node b has no incoming edges, then a also had no incoming edges.

    D. If there is a directed path from c to a and a directed path from c to b, the directed path from c to a has less edges than the directed path from c to b.

> **Answer:** If there would be a directed path from b to a, a sorting where a appears before b is not a topological sorting. Therefore, there cannot be such an edge.

19. (1 point) Consider the following weighted graph:



    If we calculate the shortest path between A and F in this graph using Dijkstra's algorithm, how many times do we need to update the distance that is stored for node B (including the update from ∞ to an initial value)?

    A. 0 times

    B. 1 time

    C. 2 times

    D. 3 times

> **Answer:** When we visit A, it is first updated to 20. Then we visit D and we update it to 17. Then, we visit E. Then, we visit C and we update it to 13. Finally, we visit F and we are done.

20. (1 point) While applying Kruskal's algorithm to a given graph, we iterate over the edges in a sorted order. What happens if we encounter an edge between 2 vertices of the same cluster?

    A. We add the edge to the minimum spanning tree, nothing else.

    B. We add the edge to the minimum spanning tree, and we remove the largest edge from the other edges that now form a cycle.

    C. We do not add the edge to the minimum spanning tree, nothing else.

    D. We do **not** add the edge to the minimum spanning tree, and the algorithm terminates because the minimum spanning tree is now complete.

> **Answer:** We do not add the edge, because that would create a cycle, and since we iterated over edges in a sorted order, all other edges in this cycle have a smaller (or equal) weight. We cannot terminate, because there could be a node which has only incident edges that have a high weight which all haven't been added yet. Since no edges to this node have been added, we don't have a spanning tree yet.

Open questions (50%, 20 points)

21. (5 points) Consider the following Java implementation of method `methodX`. Note that `remove` corresponds to the dequeue operation and `add` corresponds to the enqueue operation.

```
1  public static int methodX(Queue<Integer> q,int x) {
2    int n = q.size();
3    if(n == 1) {
4      if(q.peek() == x) {
5        return 1;
6      } else {
7        return 0;
8      }
9    }
10   int sum = 0;
11   for(int i = 0; i < n; i++) {
12     int y = q.remove();
13     sum += methodX(q,x);
14     q.add(y);
15   }
16   return sum;
17 }
```

In this method, the size of the input n is the number of elements in queue `q`.

Give the recurrence equation expressing the tightest worst-case **time** complexity of `methodX` as a function of n. Define all variables and constants, and explain which lines of code contribute to each term of the recurrence equation. Make sure that each line is mentioned in at least one of the terms.

Answer: The adjacency map is optimal. The use of an adjacency map leads to the following polynomial:

$$\begin{aligned} T(1) &= c_0 \\ T(n) &= c_1 + c_2 \cdot n + n \cdot T(n-1) \qquad \text{if } n > 1 \end{aligned}$$

where:

- c_0 accounts for the primitive instructions associated with calling the method `methodX` (line 1), initialization of `n` (line 2), the conditionals (line 3-4) and returning from the method `methodX` (line 5 or line 7);
- c_1 accounts for the primitive instructions associated with calling the method `methodX` (line 1), initialization of `n` (line 2), the conditional (line 3), the initialization of `sum` (line 10), the initialization of 'i' and the first conditional test of the loop (line 11), and returning from the method `methodX` (line 15);
- $c_2 \cdot n$ accounts for the increment 'i++' (line 12), the conditional test (line 12) and the `poll` (line 13) and `offer` (line 14) method in the loop;
- $n \cdot T(n-1)$ accounts for the recursive calls (line 13) in the loop.

22. (5 points) Derive the closed form of the following recurrence equation:

$$T(5) = c_0$$
$$T(n) = c_1 + c_2 \cdot n + T(n-1) \qquad \text{if } n > 5$$

To answer this question, you should either:

- derive the closed form solution by repeatedly unfolding the recurrence equation, or
- guess the closed form and prove the correctness of your solution by induction.

Note the base case is for $n = 5$.

Answer: Option 1. By repeated unfolding:

$$\begin{aligned}
T(n) &= c_1 + c_2 \cdot n + T(n-1) && \text{(by unfolding } T(n)\text{)}\\
&= c_1 + c_2 \cdot n + (c_1 + c_2 \cdot (n-1) + T(n-1-1)) && \text{(by unfolding } T(n-1)\text{)}\\
&= 2 \cdot c_1 + c_2 \cdot (n + (n-1)) + T(n-2) && \text{(by arithmetic)}\\
&= k \cdot c_1 + c_2 \cdot \sum_{i=0}^{k-1} (n-i) + T(n-k) && \text{(by repeating } k \text{ times)}\\
&= (n-5) \cdot c_1 + c_2 \cdot \sum_{i=0}^{n-6} (n-i) + T(n-(n-5)) && \text{(by letting } k = n-5\text{)}\\
&= (n-5) \cdot c_1 + c_2 \cdot (\frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n - 15) + T(5) && \text{(by arithmetic)}\\
&= (n-5) \cdot c_1 + c_2 \cdot (\frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n - 15) + c_0 && \text{(by definition of } T(5)\text{)}
\end{aligned}$$

Option 2. By induction:

Closed form solution. The closed form solution of the above recurrence is $T(n) = (n-5) \cdot c_1 + c_2 \cdot (\frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n - 15) + c_0$. This can be obtained by repeatedly unfolding T.

Induction proof.
Base case: For $n = 5$, prove $T(n) = (n-5) \cdot c_1 + c_2 \cdot (\frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n - 15) + c_0$.

Proof.

$$\begin{aligned}
T(n) = T(5) &= c_0\\
&= 0c_1 + 0c_2 + c_0\\
&= (5-5) \cdot c_1 + c_2 \cdot (12.5 + 2.5 - 15) + c_0\\
&= (5-5) \cdot c_1 + c_2 \cdot (\frac{1}{2} \cdot 5^2 + \frac{1}{2} \cdot 5 - 15) + c_0
\end{aligned}$$ □

Induction step: For $n > 5$, prove $T(n) = (n-5) \cdot c_1 + c_2 \cdot (\frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n - 15) + c_0$ assuming the induction hypothesis (IH) $T(n-1) = (n-1-5) \cdot c_1 + c_2 \cdot (\frac{1}{2} \cdot (n-1)^2 + \frac{1}{2} \cdot (n-1) - 15) + c_0$.

Proof.

$$\begin{aligned}
T(n) &= c_1 + c_2 \cdot n + T(n-1) && \text{(by definition of } T(n)\text{)}\\
&= c_1 + c_2 \cdot n + (n-1-5) \cdot c_1 + c_2 \cdot (\frac{1}{2} \cdot (n-1)^2 + \frac{1}{2} \cdot (n-1) - 15) + c_0 && \text{(by IH)}\\
&= (n-5) \cdot c_1 + c_2 \cdot n + c_2 \cdot (\frac{1}{2} \cdot n^2 - \frac{1}{2} \cdot n - 15) + c_0 && \text{(by arithmetic)}\\
&= (n-5) \cdot c_1 + c_2 \cdot (\frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n - 15) + c_0
\end{aligned}$$ □

23. Prove that $\frac{1}{2} \cdot n^2 + n - 10$ is $\Omega(n^2)$.

(a) (2 points) State in detail the mathematical conditions that should be proven.

Answer: We have to prove that there exists a positive (real) number c and a (positive integer) n_0, such that:

$$\frac{1}{2} \cdot n^2 + n - 10 \geq c \cdot n^2 \quad \text{for all } n \geq n_0.$$

(b) (2 points) Prove that these conditions hold. Explain all the steps in your proof.

Answer: Let $c = \frac{1}{2}$ and $n_0 = 10$. When filling out the constant c in the formula, we obtain:

$$\begin{aligned} \frac{1}{2} \cdot n^2 + n - 10 &\geq \frac{1}{2} \cdot n^2 \\ n - 10 &\geq 0 && \text{(by arithmetic)} \\ n &\geq 10 && \text{(by arithmetic)} \end{aligned}$$

This is true for all $n \geq 10$ because the right side of the equation will then be at least 10 and the left side will be 10.

24. In a construction market, wooden planks of different lengths are being sold. You are tasked to design the database that stores information about these planks. Each type of plank will be an item, which has two properties: length and price. Your database should be able to perform the following three operations:

    - A new type of plank is being sold: given length and price, add the respective item to the database.
    - A type of plank is not sold anymore: given the length, remove the respective item from the database.
    - A customer wants to buy a type of plank: given a length, retrieve the price from the database.

    (a) (3 points) What data structure would be most **time-efficient** for this database of plank type items? Explain your answer by giving the time complexities of the three operations for that data structure. Also explain how it stores the properties of each item.

    > **Answer:** A hashmap storing the length as the key and the price as the value. All operations would have a time complexity of $\mathcal{O}(1)$.

    (b) (3 points) You have noticed that customers sometimes want a plank of custom length, which is not being sold. In those cases, you can sell them a plank that is slightly longer and they can saw it themselves to get the correct length. Thus, the third operation is replaced with:

    - A customer wants to buy a type of plank: given a length, retrieve the length and price of the smallest plank that is at least as large as the given length.

    What data structure is most **time-efficient** for this new operation? Explain why the data structure you chose in question 24.a cannot do this operation as efficiently. Describe how this operation can now be efficiently performed in the new data structure (i.e. explain the algorithm). You do not need to give or explain the time complexity.

    > **Answer:** Any balanced binary search tree. Due to the hashing of the hashmap, it cannot easily find an item with a length that is a bit different than the length that was asked for. With a balanced binary search tree, you could traverse the tree downwards to find the length. If you end up in a null node, you could traverse the tree upwards again until the first time you traverse upwards via a left child. Then, traverse down again by going to the right child first, and then taking the left child until the left child is a null node.

End of the exam