Delft University of Technology
EEMCS Faculty

CSE1305 Algorithms & Data Structures

Resit Written Exam

16 April 2019, 18:30–20:30

Examiners:

Examiner responsible: Joana Gonçalves and Robbert Krebbers
Examination reviewer: Stefan Hugtenburg

Parts of the examination and determination of the grade:

Exam part	Number of questions	Question specifics	Grade
Multiple-choice	22 questions (equal weights)	One correct answer per question	50%
Open questions	3 questions (different weights)	Multiple parts	50%

- The grade for the multiple-choice questions is computed as $10 \cdot \frac{\text{score}}{22}$.
- The grade for the open questions is computed as $10 \cdot \frac{\text{score}}{30}$.

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 may not leave the examination room during the first 30 minutes.
- If you are eligible for extra time, put the "Verklaring Tentamentijd Verlenging" on your table.

Instructions for writing down your answers:

- You should answer the questions on the provided answer sheets.
- Write your name and student number on every sheet of paper.
- **For multiple-choice questions**, the order of the choices on the answer form **might not be A-B-C-D!**
- Tip: mark multiple-choice answers on this exam paper first, 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%, 22 points)

1. (1 point) Consider the Java implementation of method `operation` below.

```
public int operation(int n) {
  int i, j, k = 0;
  for (i = n / 2; i <= n; i++)
    for (j = 2; j <= n; j = j * 2)
      k = k + n / 2;
  return k;
}
```

Let n be a positive integer. What is the tightest worst-case time complexity of method `operation`?

A. $\mathcal{O}(n)$
B. $\mathcal{O}(n \log n)$
C. $\mathcal{O}(n^2)$
D. $\mathcal{O}(n^2 \log n)$

Answer: The first loop runs exactly $n/2$ times, so it is $\mathcal{O}(n)$. The second loop runs x times, where $n = 2^x$ or $x = \log_2 n$, therefore the time complexity is $\mathcal{O}(\log n)$. This means that the combined time complexity of the two nested loops is $\mathcal{O}(n \log n)$.

2. (1 point) Consider that the space complexity of an algorithm X is $O(1)$. Which of the following statements about algorithm X is **false**?

A. The time complexity of algorithm X is guaranteed to be $\mathcal{O}(1)$.
B. The time complexity of algorithm X is guaranteed to be $\Omega(1)$.
C. The space used by algorithm X does not asymptotically depend on the input size.
D. Algorithm X uses a constant amount of space in addition to the input.

3. (1 point) Consider the insertion of n elements (using `push` operations) in two different array stacks, `stack1` and `stack2`. For `stack1`, the array grows when full from capacity C to $\lceil \frac{5}{3} C \rceil$. For `stack2`, the array grows when full from capacity C to $C + \lceil \frac{5}{3} \rceil$. What are the tightest **amortized** time complexities of the set of n `push` operations in these two stacks?

A. stack1: $\mathcal{O}(n^2)$ stack2: $\mathcal{O}(n^2)$
B. stack1: $\mathcal{O}(n^2)$ stack2: $\mathcal{O}(n)$
C. stack1: $\mathcal{O}(n)$ stack2: $\mathcal{O}(n^2)$
D. stack1: $\mathcal{O}(n)$ stack2: $\mathcal{O}(n)$

Answer: In `stack1` the array grows by a factor of 1.66(6), proportional to the array size, leading to a geometric progression. In `stack2` the array size is incremented by a constant value, leading to an arithmetic progression.

4. (1 point) Which of the following data structures is **not** suitable to implement breadth-first traversal with the same tightest big-$\mathcal{O}$ time and space complexity than the others?

A. Dynamic array
B. Queue
C. Singly-linked list
D. Stack

Answer: A queue is appropriate for breadth-first search, since it allows us to visit the nodes per level. Dynamic arrays and singly-linked lists are also fine since they allow efficient insertion at one end and efficient removal at the other end (we can easily implement a queue using these elementary data structures). A stack only allows insertion and removal from one end. This means that we will access the last visited nodes first, and the last visited nodes are the deepest nodes, which naturally results in a depth-first traversal. In order to revert this behavior, one would need to use more time or space.

5. (1 point) Consider a maximum-oriented array-based heap given by the sequence $(25, 14, 16, 13, 10, 8, 12)$. What is the sequence of elements in the array after two `removeMax` operations?

   A. (14,8,12,13,10)

   B. (14,13,12,8,10)

   C. (8,14,12,13,10)

   D. (14,10,12,13,8)

Answer: Input sequence: (25,14,16,13,10,8,12)

Remove 25: (12,14,16,13,10,8)
Down-bubble: (16,14,12,13,10,8)
Remove 16: (8,14,12,13,10)
Down-bubble: (14,13,12,8,10)

6. (1 point) Consider the partial implementation of class `SortedArray` below, which maintains a sequence of integer elements sorted in non-decreasing order.

```
public class SortedArray {
  private Integer[] array;    // array containing the sorted elements
  private int size;           // number of elements in sorted array

  /* ... */
  public Integer operation(int element) {
    int a = 0, b = size-1, m = -1;

    while(a <= b) {
      m = a + (b-a)/2;
      if (element == array[m]) break;
      if (element < array[m]) b = m - 1;
      else a = m + 1;
    }
    if (a == size || (b <= a && array[a] != element)) return null;

    int i;
    if (b > a) i = m;
    else i = a;
    int e = array[i];

    for (int k = i+1; k < size; k++)
      array[k-1] = array[k];
    size--;
    return e;
  }
  /* ... */
}
```

What operation is performed by method `operation` in the Java code above?

A. Search element.

B. Search element index.

C. Insert element.

D. Remove element.

7. (1 point) Consider the method `operation` from question 6. What kind of search does it perform?

A. Linear search.

B. Brute-force search.

C. Binary search.

D. Multiway search.

8. (1 point) Consider the method `operation` from question 6. What is its tightest time complexity in Big-$\mathcal{O}$ notation, if n denotes the number of elements `size`?

A. $\mathcal{O}(n \log n)$

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

C. $\mathcal{O}(\log n)$

D. $\mathcal{O}(1)$

9. (1 point) Which of the following statements on sorting algorithms is **false**?

A. An algorithm is called stable if it maintains the relative order of identical elements (or elements with an identical key).

B. Four sorting algorithms ordered by **best-case** time complexity: insertion sort $\leq$ quicksort $\leq$ heap sort $\leq$ selection sort.

C. The minimum possible time complexity of a comparison-based sorting algorithm is $\Omega(n \log n)$ for an input sequence with n elements.

D. Quicksort is the fastest algorithm for sorting 1 million license plate numbers.

Answer:

A. Correct, this is the definition of stability of a sorting algorithm.

B. Correct, insertion sort $\mathcal{O}(n) \leq$ quicksort (typically faster than merge/heap sort) $\mathcal{O}(n \log n) \leq$ heap sort $\mathcal{O}(n \log n) \leq$ selection sort $\mathcal{O}(n^2)$.

C. Correct.

D. The fastest algorithm for sorting 1 million license plate numbers is radix sort, since these numbers are of small length and composed of letters (26-size alphabet possible) and digits (10-size alphabet), each of which can be used individually as a key in bucket sort. In this case, radix sort has linear time complexity and is therefore faster than comparison-based algorithms.

10. (1 point) What is the recurrence equation with $n > 1$ for the **worst-case** of the quicksort algorithm, where n denotes the input size and c_1 and c_2 are positive integer constants?

A. $T(n) = T(n-2) + c_1 n + c_2$

B. $T(n) = T(n-1) + c_1 n + c_2$

C. $T(n) = T(n/2) + c_1 n + c_2$

D. $T(n) = 2T(n/2) + c_1 n + c_2$

Answer: The worst case of quicksort occurs when the pivot is always the lowest or highest element in the array. In this case, quicksort partitioning leads to one subproblem of size 0 and another subproblem of size $n-1$. A recursive call is made for the subproblem of size $n-1$. Therefore, the recurrence is expressed by: $T(n) = T(n-1) + cn$, where c is a positive integer constant.

11. (1 point) Consider that the selection sort algorithm is applied to sort the sequence (1,7,6,2,8,4,5,3) in increasing order. What is the state of the sequence after a number of iterations resulting in 3 swaps?

    A. (1,2,3,7,8,4,5,6)

    B. (1,2,6,7,8,4,5,3)

    C. (1,2,3,4,8,7,5,6)

    D. (1,2,3,4,5,7,8,6)

Answer:

Iteration 1	(1,7,6,2,8,4,5,3)	1 sorted	0 swaps
Iteration 2	(1,2,6,7,8,4,5,3)	2 sorted	1 swap
Iteration 3	(1,2,3,7,8,4,5,6)	3 sorted	2 swaps
Iteration 4	(1,2,3,4,8,7,5,6)	4 sorted	3 swaps

12. (1 point) Consider the following fixed size hash table using linear probing (associated values are omitted):

0	1	2	3	4	5	6
7	5	0			12	6

The hash function is $h(k) = k \mod 7$. Which of the following sequences **does not** denote a valid order by which the elements could have been inserted into the initially empty hash table?

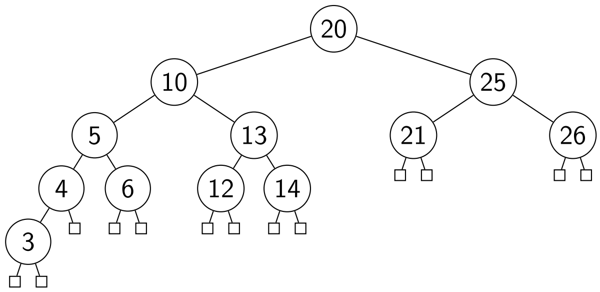
    A. (12,7,6,5,0)

    B. (12,6,7,5,0)

    C. (6,12,5,7,0)

    D. (6,12,7,5,0)

13. (1 point) Consider the following tree:



Which of the following is **true**?

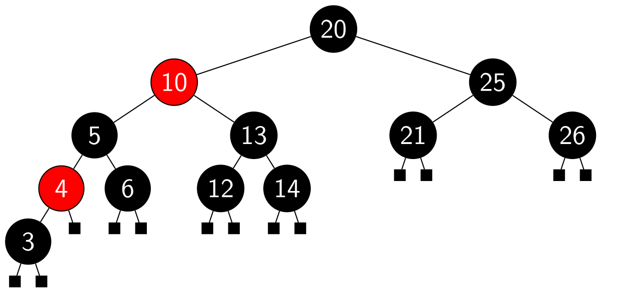
    A. The tree is an AVL tree and can be colored as a red-black tree.

    B. The tree is an AVL tree and cannot be colored as a red-black tree.

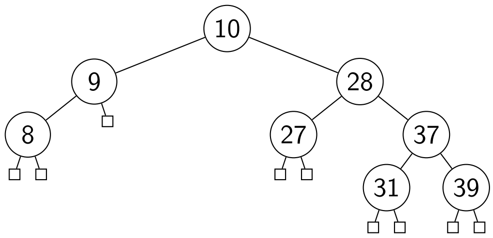
    C. The tree is not an AVL tree and can be colored as a red-black tree.

    D. The tree is not an AVL tree and cannot be colored as a red-black tree.

Answer:



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



How does the insertion of **30** affect the above AVL tree?

A. The insertion does not affect the AVL property.

B. The insertion violates the AVL property, which needs to be fixed with a single rotation.

C. The insertion violates the AVL property, which needs to be fixed with a double rotation.

D. The insertion violates the AVL property, which needs to be fixed with two double rotations.

15. (1 point) What is the maximum possible number of non-null nodes in an AVL tree of depth 4?

A. 8

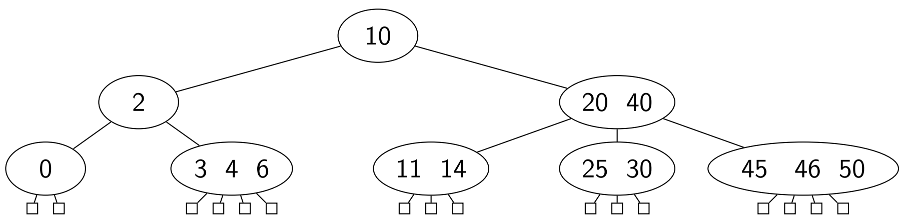
B. 14

C. 15

D. 16

Answer: The maximum number of nodes is $2^h - 1$

16. (1 point) Consider the following (2,4) tree.



What is the number of red-black trees that correspond to the given (2,4) tree?

A. 2

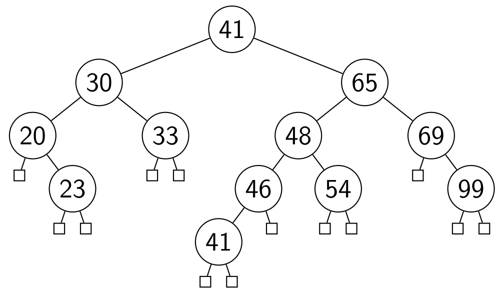
B. 3

C. 4

D. 8

Answer: Note that for each 3-node, there is a choice, but for 2 and 4-nodes there is no choice.

17. (1 point) Consider the following AVL tree:



When deleting **33** from the given AVL tree, how many tri-node restructurings are caused in the tree? If the node to be deleted has two children, the node will be replaced with the in-order predecessor (i.e. the maximal node in the left child).

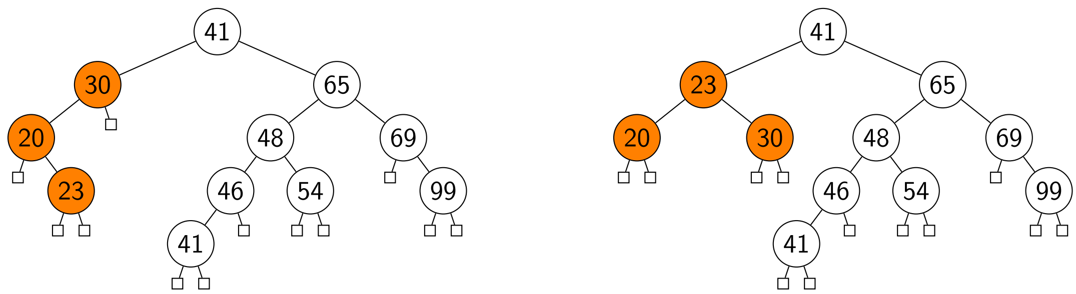
A. No tri-node restructuring.

B. One tri-node restructuring.

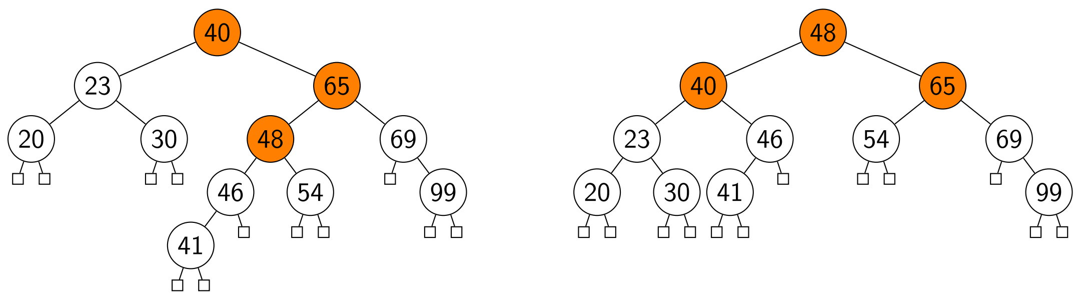
C. Two tri-node restructurings.

D. Three tri-node restructurings.

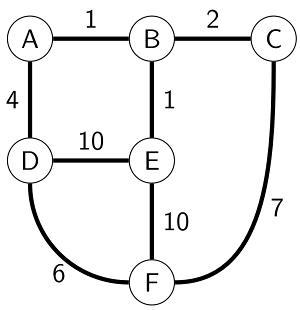
Answer: After removing 33, the node containing 30 becomes unbalanced, which needs to be fixed by a double rotation (i.e. a tri-node restructuring):



In turn, the node containing 40 becomes unbalanced, which needs to be fixed by a double rotation (i.e. another tri-node restructuring):



18. (1 point) Consider the following weighted graph:



When performing Dijkstra's algorithm starting from vertex A, how many **non-infinite distinct** labels will the vertex F have?

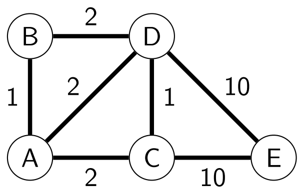
A. 1

B. 2

C. 3

D. 4

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



In general, a graph can have multiple minimum spanning trees. How many different minimum spanning trees does the above graph have?

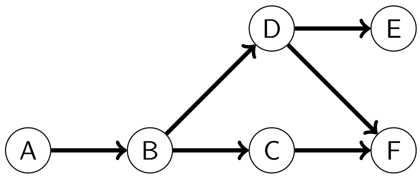
A. 3

B. 4

C. 5

D. 6

20. (1 point) Consider the following directed acyclic graph (DAG):



In general, DAGs can have multiple topological orders. How many different topological orders does the above graph have?

A. 2

B. 3

C. 4

D. 5

Answer: Note that A, B, C, F have to appear in order. Then, D can be put either after B or after C. In the first case, there are three choices for E, in the last case, there are two choices for E.

21. (1 point) Consider an undirected graph with n vertices and m edges that is a forest. Which of the following properties is **true**?

    A. $m \leq n$

    B. $m \leq n - 1$

    C. $m \geq n$

    D. $m \geq n - 1$

22. (1 point) Assume we need to represent dense graphs, and we are primarily interested in accessing edges, and removing and inserting vertices. What is the most efficient graph data structure for this purpose?

    A. Edge list.

    B. Adjacency list.

    C. Adjacency map.

    D. Adjacency matrix.

Open questions (50%, 30 points)

23. Consider the following Java implementation of `methodX`. Assume that the input array `a` is non-null.

```
1  public static int methodX(Integer[] a) {
2    int i = 0, n = a.length;
3    while (i < n) {
4      if(a[i] % 2 == 0) {
5        for (int j = i; j < n - 1; j++)
6          a[j] = a[j+1];
7        a[n-1] = null;
8        n--;
9      } else {
10       i++;
11     }
12   }
13   return n;
14 }
```

(a) (5 points) Write the polynomial expressing the worst-case **time** complexity of method `methodX` as a function of n, where n is the number of elements in array `a`. Define all variables and constants. Explain each term of the polynomial by referring to the lines of code.

Answer: The worst-case time complexity occurs in case the array only contains even elements (i.e. elements for which %2 is 0). The polynomial is:

$$\begin{aligned} T(n) &= c_0 + c_1 n + c_2((n-1) + \ldots + 2 + 1) \\ &= c_0 + c_1 n + c_2 \sum_{i=1}^{n-1} i \end{aligned}$$

where:

- n is the number of elements in the input array `a`;
- c_0 accounts for the primitive instructions associated with calling `methodX` (line 1), initializing i and n (line 2), and returning from `methodX` (line 13);
- c_1 accounts for primitive operations within the `while` loop, including the conditional test (line 3), the if statement (line 4) which always evaluates to true in the worst-case, the initialization of j for the `for` loop (line 5), and the assignment and decrement in lines 7-8.
- c_2 accounts for operations within the `for` loop, including conditional and increment of variable `j` (line 5), and the assignment `a[j] = a[j+1]` (line 6).

(b) (4 points) Simplify your polynomial expression as much as possible. State the tightest worst-case Big-$\mathcal{O}$ **time** complexity of `methodX` as a function of n. You **do not** have to give a proof, but you should clearly justify your answer.

Answer:

$$
\begin{aligned}
T(n) &= c_0 + c_1 n + c_2((n-1) + \ldots + 2 + 1) \\
&= c_0 + c_1 n + c_2 \sum_{i=1}^{n-1} i \\
&= c_0 + c_1 n + c_2 \frac{(n-1)n}{2} \\
&= c_0 + c_1 n + c_2 \frac{n^2 - n}{2} \\
&= c_0 + \left(c_1 - \frac{c_2}{2}\right) \cdot n + \frac{c_2}{2} \cdot n^2
\end{aligned}
$$

The constants can be disregarded, since $\{c_0, c_1, c_2/2\} \ll n$. The term n^2 grows faster than any other term in the polynomial when $n \to \infty$, therefore the time complexity of method `methodX` in Big-Oh notation is $\mathcal{O}(n^2)$.

(c) (3 points) Consider that the array `a` given as input to `methodX` only contains non-null elements. Explain what the algorithm `methodX` computes, i.e. what is the content of array `a` after calling `methodX(a)` and what is the return value? Describe a faster solution to perform the same operation and state its tightest worst-case **time** complexity.

Answer: The algorithm `methodX` removes all even elements from array `a`. When removing an even element, it shifts the subsequent elements to the left, and pads the end of the array with a `null` value. The return value indicates the number of non-null (or odd) elements in array `a` after all even elements have been removed.

A faster solution would be to not perform the entire removal of each even element at once, once found. Instead, when we find an odd element, we can place it immediately next to the last odd element so that any even elements in between are automatically excluded. In this way, the odd elements are shifted to the left as needed and the padding of all the remaining positions in the right part of the array is done at the end. The worst-case complexity of this solution is $\mathcal{O}(n)$.

```java
public static int removeEven(Integer[] a) {
  int numOdd = 0, n = a.length;
  for (int i = 0; i < n; i++)
    if (a[i] % 2 == 1)
      a[numOdd++] = a[i];
  for (int i = numOdd; i < n; i++)
    a[i] = null;
  return numOdd;
}
```

24. Consider the following Java implementation of a recursive algorithm.

```
1 public static int methodY(int n) {
2   if (n == 0)
3     return 0;
4
5   return 1 + methodY(n - 1) + methodY(n - 1);
6 }
```

(a) (4 points) State the base and recurrence equation for the **time** complexity of method `methodY` as a function of n. Refer to the relevant parts of the code to justify your answer.

Answer:

$$T(0) = c_0 \qquad T(n) = c_1 + 2 \cdot T(n-1)$$

where:

c_0 accounts for the constant time operations in the base case, i.e. calling the method `methodY` (line 1), the conditional (line 2), and returning from method `methodY` (line 3).

c_1 accounts for the constant time operations in the recursive case, i.e. calling the method `methodY` (line 1), the conditional (line 2), the arithmetic operations (line 5), and returning from method `methodY` (line 5).

$2 \cdot T(n-1)$ accounts for the two recursive calls (line 5).

(b) (6 points) Derive the closed form of the given recurrence equation. You should either:

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

Answer: Option 1. By repeated unfolding:

$$\begin{aligned}
T(n) &= 2 \cdot T(n-1) + c_1 && \text{(by unfolding } T(n)\text{)}\\
&= 2 \cdot (2 \cdot T(n-2) + c_1) + c_1 && \text{(by unfolding } T(n-1)\text{)}\\
&= 4 \cdot T(n-2) + 3c_1 && \text{(by arithmetic)}\\
&= 2^k \cdot T(n-k) + (2^k - 1)c_1 && \text{(by repeating } k \text{ times)}\\
&= 2^n \cdot T(0) + (2^n - 1)c_1 && \text{(by letting } k = n\text{)}\\
&= 2^n c_0 + (2^n - 1)c_1\\
&= 2^n(c_0 + c_1) - c_1
\end{aligned}$$

Option 2. By induction:

Closed form solution. The closed form solution of the above recurrence is $T(n) = 2^n(c_0 + c_1) - c_1$. This can be obtained by repeatedly unfolding $T(n)$ using $T(n) = 2T(n-1) + c_1$ and replacing $T(0)$ using $T(0) = c_0$.

Induction proof.

Base case: For $n = 0$, prove $T(0) = c_0$.

Proof.

$$\begin{aligned}
T(n) &= 2^0(c_0 + c_1) - c_1\\
&= 1(c_0 + c_1) - c_1\\
&= c_0 + c_1 - c_1\\
&= c_0
\end{aligned}$$

□

Induction step: For $h > 0$, prove $T(n) = 2^n(c_0+c_1)-c_1$ assuming $T(n-1) = 2^{n-1}(c_0+c_1)-c_1$.

Proof.

$$\begin{aligned}
T(n) &= 2 \cdot T(n-1) + c_1 && \text{(by } T(n) = 2T(n-1) + c_1\text{)} \\
&= 2 \cdot \left(2^{n-1}(c_0 + c_1) - c_1\right) + c_1 && \text{(by IH } T(n-1) = 2^{n-1}(c_0 + c_1) - c_1\text{)} \\
&= 2^{n-1+1}(c_0 + c_1) - 2c_1 + c_1 && \text{(by arithmetic)} \\
&= 2^n(c_0 + c_1) - c_1 && \square
\end{aligned}$$

(c) (2 points) State the tightest worst-case Big-$\mathcal{O}$ **time** complexity of `methodY` as a function of n. You **do not** have to give a proof, but you should clearly justify your answer.

Answer: We previously established that the closed form is $T(n) = 2^n(c_0 + c_1) - c_1$. The constants in the closed form can be disregarded, therefore the time complexity of method `methodY` in Big-Oh notation is $\mathcal{O}(2^n)$.

(d) (2 points) State the tightest worst-case Big-$\mathcal{O}$ **space** complexity of `methodY` as a function of n. You **do not** have to give a proof, but you should clearly justify your answer.

Answer: The worst-case space complexity of method `methodY` in Big-Oh notation is $\mathcal{O}(n)$ because the method will make n nested recursive calls until the base case is reached (n decreases by 1 at each level until it becomes 0). The number of stack frames is therefore proportional to n, and each stack frame uses a constant amount of space.

25. Prove that $\left(\frac{n}{2}\right)^2 + \log(2)$ is $\mathcal{O}(n^4)$.

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

Answer: In order to prove that $\left(\frac{n}{2}\right)^2 + \log(2)$ is $\mathcal{O}(n^4)$, we have to show that there exist positive constants c and n_0, such that:

$$\left(\frac{n}{2}\right)^2 + \log(2) \leq c \cdot n^4 \quad \text{for all } n \geq n_0.$$

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

Answer: Let $c = 1/4 + \log(2)$ and $n_0 = 1$. When filling out these constants in the formula and performing some simplification, we obtain:

$$1/4 \cdot n^2 + \log(2) \leq 1/4 \cdot n^4 + \log(2) \cdot n^4$$

This formula holds for any $n \geq n_0$ where $n_0 = 1$, because $n^2 \leq n^4$ and $1 \leq n^4$ for any $n \geq 1$.

End of the exam