Delft University of Technology
EEMCS Faculty

CSE1305 Algorithms & Data Structures

Final Written Exam

25 January 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	18 questions (equal weights)	One correct answer per question	50%	$5 \cdot \frac{\text{score}}{18}$
Open questions	3 questions (different weights)	Multiple parts	50%	$5 \cdot \frac{\text{score}}{25}$

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.

Multiple-choice questions (50%, 18 points)

1. (1 point) Which of the following statements about asymptotic algorithmic complexities is **true**?
   - A. There exists a function $f(n)$ which is $\mathcal{O}(n)$ and $\Omega(n)$, but not $\Theta(n)$.
   - B. There exists a function $f(n)$ which is $\mathcal{O}(n)$ and $\Theta(n)$, but not $\Omega(n)$.
   - C. There exists a function $f(n)$ which is $\Omega(n)$ and $\Theta(n)$, but not $\mathcal{O}(n)$.
   - **D. None of the above is true.**

Answer:

- A. False. If a function is $\mathcal{O}(n)$ and $\Omega(n)$, then it is also $\Theta(n)$.
- B. False. If a function is $\Theta(n)$, then it is also $\Omega(n)$.
- C. False. If a function is $\Theta(n)$, then it is also $\mathcal{O}(n)$.
- **D. True.**

2. (1 point) Consider a recursive method that makes two recursive calls whose input size is 1 lower than the input size with which the method was called. Which of the following statements about this recursive method is **true**?
   - A. Both the time and space complexity can be $\mathcal{O}(n)$.
   - B. The time complexity can be $\mathcal{O}(n)$, but the space complexity cannot.
   - **C. The space complexity can be $\mathcal{O}(n)$, but the time complexity cannot.**
   - D. Both the time and space complexity cannot be $\mathcal{O}(n)$.

Answer: If a given method makes 2 recursive calls with input size $n-1$, the total number of calls will be $\mathcal{O}(2^n)$. Since making each call takes time, the time complexity is $\Omega(2^n)$ which is not $\mathcal{O}(n)$. Since we can reuse space, if the recursive method does not use more than $\mathcal{O}(1)$ space in a call, we will maximally use n stack frames at each point in time, which in the end uses $\mathcal{O}(n)$ space.

3. (1 point) The Java method `rearrange` below rearranges the elements of a linked list with `head` node of type `Node`, also defined below.

```java
public class ADSList {
  private static class Node {
    private int value;
    private Node next;
    /* ... */
  }

  public static void rearrange(Node head) {
    Node p, q;
    int temp;
    if (head == null || head.getNext() == null) {
      return;
    }
    p = head;
    q = head.next;
    while (q != null) {
      temp = p.value;
      p.value = q.value;
      q.value = temp;
      p = q.next;
      if (p != null)
        q = p.next;
      else
        q = null;
    }
  }
}
```

Method `rearrange` is called with a list containing the following integer values [0, 2, 4, 6, 8, 10, 12] in the given order (left/head to right/tail). What will be the contents of the list after the call to method `rearrange` completes its execution?

A. head [2, 0, 6, 4, 10, 8, 12] tail

B. head [4, 2, 0, 10, 8, 6, 12] tail

C. head [0, 4, 2, 8, 6, 12, 10] tail

D. head [4, 0, 2, 10, 6, 8, 12] tail

Answer: The method `rearrange` swaps the values of the first pair of consecutive nodes, then proceeds to swapping the values of the subsequent pair of nodes (non-overlapping), and so on.

4. (1 point) Consider the most time-efficient algorithm to append all m elements of a sequence $S2$ one by one to the end of another sequence $S1$ already containing n elements, without removing the elements from sequence $S2$. What is the tightest worst-case time complexity of such an algorithm when sequences $S1$ and $S2$ are implemented using a dynamic array that increments the size by one with every resizing? You can assume that $n > m$.

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

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

C. $\mathcal{O}(m^2)$

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

Answer: To add the first element of $S2$, we need to:

- create a new array of size $n + 1$,
- then copy the n elements of $S1$ into this new array,
- then copy the first element of $S2$ into the new array as well.

More generally, to add the k^{th} element of $S2$ we create a new array of size $n+k$, copy the $n+k-1$ elements of $S1$ into the new array, copy the k^{th} element of $S2$ into the new array.

The time complexity for adding all elements of $S2$ one by one to the end of $S1$ is given by $c(n+1)+c(n+2)+...+c(n+m) = c((n+1)+(n+2)+...+(n+m)) = c(nm+m^2)$, which is $\mathcal{O}(\max\{nm, m^2\})$. Since we know that $n > m$, this simplifies to $\mathcal{O}(nm)$.

5. (1 point) Consider the following Java method `recursiveMethod` below.

```
private static void recursiveMethod(int[] array, int low, int high) {
  if (high < low)
    return;

```

```java
    int temp = array[low];
    array[low] = array[high];
    array[high ] = temp;
    recursiveMethod(array, low + 1, high - 1);
}
```

What is the tightest worst-case **space** complexity of calling the method for a given array of integer elements `arr`, that is, executing the call `recursiveMethod(arr, 0, arr.length-1)`? Use variable n to denote the length of the input array.

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

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

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

D. $\mathcal{O}(n^2)$

Answer: Since only variables of $\mathcal{O}(1)$ space are initialized in the body of the method, and the recursive call is called with a size which half the input size, the recurrence equation is:

$$S(1) = c_0$$
$$S(n) = c_1 + S\left(\frac{n}{2}\right) \qquad \text{if } n > 1$$

Unfolding this gives a closed-form solution which is $\mathcal{O}(n)$.

6. (1 point) Consider that a queue is implemented using a singly-linked list with both a reference to the head and a reference to the tail of the list (without dummy header and trailer nodes). Both enqueueing and dequeueing operations are implemented such that they take $\mathcal{O}(1)$ time. Which of these references will change when enqueueing an element into either an empty or a non-empty queue?

A.	**empty:** head	**non-empty:** head
B.	**empty:** tail	**non-empty:** tail
C.	**empty:** both	**non-empty:** head
D.	**empty: both**	**non-empty: tail**

Answer: Empty queue: head/tail $\rightarrow$ null tail $\rightarrow$ null After insertion: head/tail $\rightarrow$ 1 $\rightarrow$ null

```
                      tail                                        tail
                       v                                           v
Non-empty queue: head → 1 → 2 → null    After insertion: head → 1 → 2 → 3 → null
```

7. (1 point) Consider a complete binary tree whose inorder traversal is 6, 5, 1, 4, 3, 2, 10, 7. What is the parent of node 1?

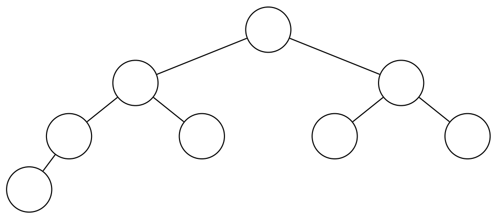
A. 5

B. 4

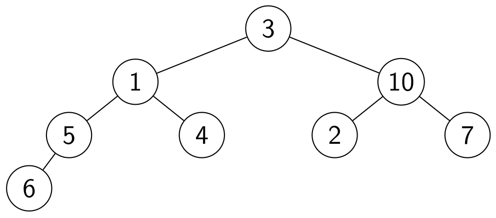
C. 3

D. 2

Answer: A complete binary tree with 8 nodes has the following shape:



If we perform an inorder traversal through this tree while filling in the values in the corresponding nodes, we get:



8. (1 point) Consider that the **heapify** algorithm is applied to build a heap of height h from the elements of a given array. What is the tightest upper bound on the total number of swaps in the calls made by heapify for the nodes at the third last level (or antepenultimate) of the complete binary tree represented by the array? You can assume that the antepenultimate level exists.

   A. $2^{h-1} - 1$
   B. $2^{h-1} - 2$
   C. $2^{h} - 1$
   D. $2^{h} - 2$

Answer: Last level is h, penultimate is $h-1$, antepenultimate is $h-2$.

The upper bound on the total number of swaps performed by all downheap calls at level $h-2$ is given by the maximum possible number of nodes at level $h-2$ multiplied by the maximum possible height of the subtrees having their root nodes at level $h-2$.

Maximum possible number of nodes at level $h-2$ is given by $2^{h-2+1} - 1 = 2^{h-1} - 1$
Maximum possible height of subtrees rooted at level $h-2$ is given by $h - (h-2) = 2$

Answer is then $\left(2^{h-1} - 1\right) \times 2 = 2^{h-1+1} - 2 = 2^{h} - 2$

9. (1 point) Consider the Java method `specialTraversal` below, which performs a special traversal of a binary tree rooted at the given `node`.

```java
public class BinaryTree {
  private static class Node {
    int value;
    Node left, right;
    public Node(int v) { value = v; }
  }

  public void specialTraversal(Node node) {
    DS1<Node> ds1 = new DS1<>();
    DS2<Node> ds2 = new DS2<>();
    ds1.insert(node);

    while (!ds1.isEmpty()) {
      node = ds1.peek();
      ds1.remove();
      ds2.insert(node);

      if (node.right != null)
        ds1.insert(node.right);
      if (node.left != null)
        ds1.insert(node.left);
    }

    while (!ds2.isEmpty()) {
      node = ds2.peek();
      System.out.print(node.data + " ");
      ds2.remove();
    }
  }
}
```

For a given complete binary tree t, we know the following: the breadth-first traversal of t visits the nodes of t in the following order [1, 2, 3, 4, 5, 6, 7], and the `specialTraversal` method prints out the nodes of t in the following order [4, 5, 6, 7, 2, 3, 1]. Knowing this, what kind of data structures are `DS1` and `DS2` ? You can assume that `specialTraversal` is called using the root of the binary tree as argument.

A. **DS1:** stack	**DS2:** queue
B. **DS1:** stack	**DS2:** stack
C. **DS1:** queue	**DS2:** queue
D. DS1: queue	**DS2: stack**

Answer: If DS1 is a queue and DS2 is a stack, we add 1 to the queue and get the following:

- In the first iteration, we remove 1 from the queue and put it in the stack, adding 3 and 2 to the queue
- In the second iteriaton, we remove 3 from the queue and put it in the stack, adding 7 and 6 to the queue
- In the third iteriation, we remove 2 from the queue and put it in the stack, adding 5 and 4 to the queue
- In the four iterions after those, we simply remove 7, 6, 5 and 4 respectively from the queue and add them to the stack, adding nothing to the queue because their children are null
- In the second while loop, we remove the elements from the stack in reverse order of how we added them, which gives us the order 4, 5, 6, 7, 2, 3, 1.

10. (1 point) Consider sorting the elements of the following array [1, 5, 7, 6, 4, 3, 2] in increasing order using the in-place insertion sort algorithm. How many elements are shifted and how many comparisons are made during the complete execution of the inner loop for **only the one** iteration of the outer loop that processes the element at index 4?

A. 3 shifts and 3 comparisons
B. 3 shifts and 4 comparisons
C. 4 shifts and 3 comparisons
D. 4 shifts and 4 comparisons

Answer: Index 0: sorted by definition Index 1: compare 5 to 1, no shift Index 2: compare 7 to 5, no shift Index 3: compare 6 to 7, shift 7; compare 6 to 5, no shift Index 4: compare 4 to 6, shift 6; compare 4 to 7, shift 7; compare 4 to 5; shift 5; compare 4 to 1

11. (1 point) Consider sorting the elements of the following array [0, 4, 3, 6, 5, 2, 1] in increasing order using the in-place quick sort algorithm that always chooses the pivot as the last element in the subarray to be sorted. How many swaps are performed in total by the algorithm when sorting this specific array?
    - A. 3
    - B. 4
    - **C. 5**
    - D. 6

Answer: Input array $[0, 4, 3, 6, 5, 2, 1]$

Partition start idx 0, end idx: 6 (pivot: 1) (swaps: 1) total swaps: 1
$[0, 1, 3, 6, 5, 2, 4]$ (finish partition: swapped 4 1 (indices 1 and 6)

Partition start idx 0, end idx: 0 (pivot: -) (swaps: 0) total swaps: 1

Partition start idx 2, end idx: 6 (pivot: 4) (swaps: 2) total swaps: 3
$[0, 1, 3, 2, 5, 6, 4]$ swapped 6 2 (indices 3 and 5)
$[0, 1, 3, 2, 4, 6, 5]$ finish partition: swapped 5 4 (indices 4 and 6)

Partition start idx 2, end idx: 3 (pivot: 2) (swaps: 1) total swaps: 4
$[0, 1, 2, 3, 4, 6, 5]$ finish partition: swaped 3 2 (indices 2 and 3)

Partition start idx 2, end idx: 1 (pivot: -) (swaps: 0) total swaps: 4

Partition start idx 3, end idx: 3 (pivot: -) (swaps: 0) total swaps: 4

Partition start idx 5, end idx: 6 (pivot: 5) (swaps: 1) total swaps: 5
$[0, 1, 2, 3, 4, 5, 6]$ finish partition: swaped 6 5 (indices 5 and 6)

Partition start idx 5, end idx: 4 (pivot: -) (swaps: 0) total swaps: 5

Partition start idx 6, end idx: 6 (pivot: -) (swaps: 0) total swaps: 5

12. (1 point) Which of the following statements about sorting and selection algorithms is **true**?
    - **A. The tightest worst-case big-Oh space complexity of the merge sort, in-place quick sort, and in-place quick select algorithms is $\mathcal{O}(n)$.**
    - B. Radix sorts, merge sort, in-place heap sort, and in-place insertion sort are all stable sorting algorithms.
    - **C. For finding the median of an unsorted array, sorting the array using in-place heap sort followed by accessing the middle element has lower worst-case big-Oh time complexity than applying quick select.**
    - D. Key-based sorting is always faster than comparison-based sorting.

Answer:

A. True.

B. False. In-place heap sort is not stable.

C. True.

D. False. This depend on the input data. For example, if N in bucket sort is a lot larger than n, $\mathcal{O}(n+N)$ might be larger than $\mathcal{O}(n\log(n))$.

13. (1 point) Consider the following fixed size hash table using linear probing (associated values are omitted):

0	1	2	3	4	5	6	7	8	9
9		11					36	16	18

The hash function is $h(k) = (k+1) \mod 10$. Assume we first delete the entry with key 11, then insert an entry with key 19, and then delete an entry with key 17. How many buckets do we need to search before concluding that 17 is not in the hash table?

A. 1

B. 4

C. 5

D. 6

Answer: After deleting the entry with key 11 (hash code 2), the hash table is:

0	1	2	3	4	5	6	7	8	9
9		$\times$					36	16	18

Here, $\times$ represents a DEFUNCT bucket.

After inserting an entry with key 19 (hash code 0), the hash table is:

0	1	2	3	4	5	6	7	8	9
9	19	$\times$					36	16	18

When deleting an entry with key 17 (hash code 8), it will probe until it finds an empty bucket, which will appear at position 3, so we searched at positions 8, 9, 0, 1, 2 and 3.

14. (1 point) Maps can be implemented in multiple ways. Consider a map implemented as a sorted list, a map implemented as a hash table, and a map implemented as a balanced search tree. For which of the following operations does the map implemented as a balanced search tree have the lowest expected time complexity?

A. Removing the element with a given key.

B. Removing the element with the smallest key.

C. Removing the element with the greatest key which is smaller than a given value.

D. Removing all elements with a given set of k keys.

Answer:

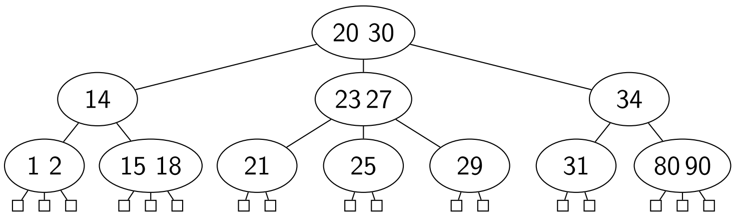
A. Incorrect. A hash table does this in $\mathcal{O}(1)$ time.

B. Incorrect. A sorted list does this in $\mathcal{O}(1)$ time.

C. Correct. The balanced search tree does this in $\mathcal{O}(\log n)$ time.

D. Incorrect. A hash table does this in $\mathcal{O}(k)$ time.

15. (1 point) Consider the following (2,4) tree:



When deleting **30** from the tree, what happens with the middle child of the root currently containing (23,27)? 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).

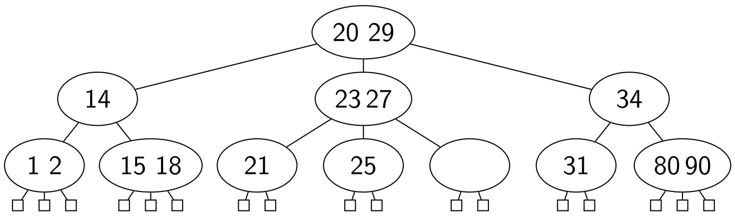
A. Node (23 27) will be fused with its right sibling.

B. Entry 23 will be moved to a child for a fusion operation.

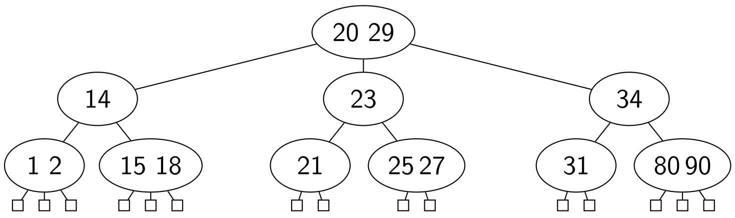
C. Entry 27 will be moved to a child for a fusion operation.

D. Nothing happens, node (23 27) will stay as it is.

Answer: After removing 30, we end up with:



After performing a transfer, we end up with:



There are no more underflows in this tree, so we are done.

16. (1 point) Given a red-black tree of height 10, what is the minimum depth that each leaf needs to have? Note that height and depth do **not** mean black height and black depth here.

A. 1

B. 5

C. 9

D. 10

Answer: Since the height of the tree is 10, the leaf with the most ancestors will have 10 proper ancestors. At most 5 of them can be red, otherwise a red node would have a red parent. Thus, at least 5 of them need to be black, so the minimum black depth is 5. Thus, the depth is also at least 5.

17. (1 point) Consider that we want to find a shortest path in a graph were all edges have a weight of 1. Which of the following statements is **true**?

    A. Dijkstra's algorithm might not return the correct result.

    B. There exists an algorithm faster than Dijkstra's to find a shortest path in such a graph.

    C. Although we could use another algorithm to find a shortest path in such a graph, Dijkstra's algorithm has the same time complexity as that alternative algorithm.

    D. Dijkstra's algorithm is the fastest algorithm to find a shortest path in all types of graphs.

Answer:

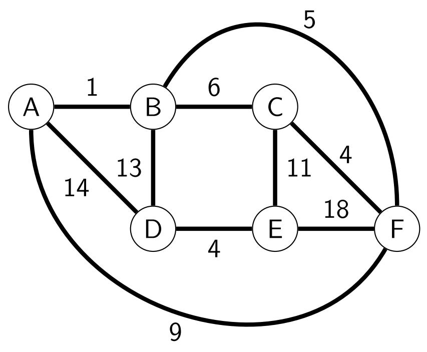
A. False. Dijkstra will also return the correct result in this case.

B. True. Using a breadth-first search is faster than using Dijkstra's algorithm, because a Queue has faster operations than a Priority Queue.

C. False. Using a breadth-first search is faster than using Dijkstra's algorithm, because a Queue has faster operations than a Priority Queue.

D. False. For some restrictions in what graphs are possible, it is better to use a different algorithm, such as in this case.

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



If we apply Kruskal's algorithm to find the minimum spanning tree in the above graph, what is the last edge that will be added to the minimum spanning tree?

A. (A, F)

B. (B, C)

C. (C, E)

D. (E, F)

Open questions (50%, 25 points)

19. Prove that $4 \cdot n^2 - 20$ is $\mathcal{O}(n^2)$.

(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:

$$4 \cdot n^2 - 20 \leq c \cdot n^2 \quad \text{for all } n \geq n_0.$$

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

Answer: Let $c = 4$ and $n_0 = 0$. When filling out the constant c in the formula, we obtain:

$$\begin{aligned} 4 \cdot n^2 - 20 &\leq 4 \cdot n^2 \\ -20 &\leq 0 && \text{(by arithmetic)} \end{aligned}$$

This is true for any n because it is independent of n, thus it is also true for any $n \geq 0$.

(c) (4 points) Prove that all functions which are $\mathcal{O}(4 \cdot n^2 - 20)$ are also $\mathcal{O}(n^2)$ i.e. prove that for an arbitrary function $f(n)$, if $f(n)$ is $\mathcal{O}(4 \cdot n^2 - 20)$, then $f(n)$ is $\mathcal{O}(n^2)$.

Answer: Take an aribrary function $f(n)$. Since $f(n)$ is $\mathcal{O}(4 \cdot n^2 + 20)$, there exists a positive (real) number c' and a (positive integer) n_0', such that:

$$f(n) \leq c' \cdot 4 \cdot n^2 - c' \cdot 20 \quad \text{for all } n \geq n_0'.$$

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

$$f(n) \leq c \cdot n^2 \quad \text{for all } n \geq n_0.$$

Let $c = 4 \cdot c'$ and $n_0 = n_0'$. When filling out the constant c in the formula, we obtain:

$$f(n) \leq 4 \cdot c' \cdot n^2$$

Since we established that $f(n) \leq c' \cdot 4 \cdot n^2 - c' \cdot 20 \quad$ for all $n \geq n_0'$, and $c' \cdot 4 \cdot n^2 - c' \cdot 20 \leq 4 \cdot c' \cdot n^2$ for any n, we can conclude that $f(n) \leq 4 \cdot c' \cdot n^2 \quad$ for all $n \geq n_0'$. Thus, $f(n)$ is $\mathcal{O}(n^2)$.

20. Consider the following Java implementation of method `methodX`.

```java
1  public static <V,E> PositionalList<Vertex<V>> methodX(Graph<V,E> g) {
2      PositionalList<Vertex<V>> ret = new LinkedPositionalList<>( );
3      Stack<Vertex<V>> ready = new LinkedStack<>( );
4      Map<Vertex<V>, Integer> inCount = new ProbeHashMap<>( );
5      for (Vertex<V> u : g.vertices()) {
6          inCount.put(u, g.inDegree(u));
7          if (inCount.get(u) == 0)
8              ready.push(u);
9      }
10     while (!ready.isEmpty( )) {
11         Vertex<V> u = ready.pop( );
12         ret.addLast(u);
13         for (Edge<E> e : g.outgoingEdges(u)) {
14             Vertex<V> v = g.opposite(u, e);
15             inCount.put(v, inCount.get(v) - 1);
16             if (inCount.get(v) == 0)
17                 ready.push(v);
18         }
19     }
20     return ret;
21 }
```

(a) (2 points) Explain what the algorithm `methodX` does. In other words, what can you tell about the output of the method given the input?

Answer: The method returns a topological order of the given graph.

(b) (6 points) Assume that graph `g` does not have parallel edges or self-loops. Graph `g` could be implement in multiple ways. Consider the use of these 3 possible data structures:

- An edge list
- An adjacency map
- An adjacency matrix

Which data structure is optimal for this method concerning worst-case time complexity? Give the polynomial expressing the tightest worst-case **time** complexity of `methodX` as a function of the number of vertices n and the number of edges m of graph `g` when the optimal data structure is used. Define all variables and constants, and explain which lines of code contribute to each term of the polynomial.

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

$$T(n,m) = c_0 + c_1 n + c_2 m$$

where:

- c_0 accounts for the primitive instructions associated with calling the method `methodX` (line 1), initialization of `ret`, `ready` and `inCount` (line 2-4), initialization of the for loop (line 5), and returning from the method `methodX` (lines 20);
- $c_1 n$ accounts for the constant time operations executed n times (line 5-12);
- $c_2 m$ accounts for the call to `g.outgoingEdges` for all vertices (line 13) and the constant time operations that are performed for all edges (line 14-17).

(c) (4 points) Describe what would change in this analysis for each of the data structures that are not optimal. Note that your answer should contain the following:

- The polynomial expression of the first data structure that is not optimal.

- The operation (and line of code) that causes the difference in the polynomial obtained for the first non-optimal data structure compared to the optimal one in your answer to question 20b. Explain why this difference occurs.
- The polynomial expression of the second data structure that is not optimal.
- The operation (and line of code) that causes the difference in the polynomial obtained for the second non-optimal data structure compared to the optimal one in your answer to question 20b. Explain why this difference occurs.

Answer: Using an edge list leads to the following polynomial expression:

$$T(n,m) = c_0 + c_1 n + c_2 m + c_3 nm$$

This is because `g.inDegree(u)` and `g.outgoingEdges()` take $\mathcal{O}(m)$ time instead of $\mathcal{O}(1)$. Since the operations is in a for loop executing n times, this causes the term $c_2 nm$.
Using an adjacency matrix leads to the following polynomial expression:

$$T(n,m) = c_0 + c_1 n + c_2 m + c_3 n^2$$

This is because `g.inDegree(u)` and `g.outgoingEdges()` take $\mathcal{O}(m = n)$ time instead of $\mathcal{O}(1)$. Since the operations is in a for loop executing n times, this causes the term $c_3 n^2$.

21. (5 points) Derive the closed form of the following recurrence equation:

$$
\begin{aligned}
T(1) &= c_0 \\
T(n) &= c_1 + 2 \cdot T\left(\frac{n}{4}\right) \qquad \text{if } n > 1
\end{aligned}
$$

You may assume that the function is only called with integers that result in all recursive calls being made with a whole number.

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.

Answer: Option 1. By repeated unfolding:

$$
\begin{aligned}
T(n) &= c_1 + 2 \cdot T\left(\frac{n}{4}\right) && \text{(by unfolding } T(n)\text{)} \\
&= c_1 + 2 \cdot \left(c_1 + 2 \cdot T\left(\frac{n/4}{4}\right)\right) && \text{(by unfolding } T\left(\tfrac{n}{4}\right)\text{)} \\
&= 3c_1 + 4 \cdot T\left(\frac{n}{16}\right) && \text{(by arithmetic)} \\
&= (2^k - 1) \cdot c_1 + 2^k \cdot T\left(\frac{n}{4^k}\right) && \text{(by repeating } k \text{ times)} \\
&= (2^{\log_4(n)} - 1) \cdot c_1 + 2^{\log_4(n)} \cdot T\left(\frac{n}{4^{\log_4(n)}}\right) && \text{(by letting } k = \log_4(n)\text{)} \\
&= (\sqrt{n} - 1) \cdot c_1 + \sqrt{n} \cdot T(1) && \text{(by arithmetic)} \\
&= \sqrt{n} \cdot c_1 - c_1 + \sqrt{n} \cdot c_0 && \text{(by definition of } T(2)\text{)}
\end{aligned}
$$

Option 2. By induction:

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

Induction proof.
Base case: For $n = 1$, prove $T(n) = \sqrt{n} \cdot c_1 - c_1 + \sqrt{n} \cdot c_0$.

Proof.

$$
\begin{aligned}
T(n) = T(1) &= c_0 \\
&= 0c_1 + c_0 \\
&= \sqrt{1} \cdot c_1 - c_1 + \sqrt{1} \cdot c_0 \qquad \square
\end{aligned}
$$

Induction step: For $n > 0$, prove $T(n) = \sqrt{n} \cdot c_1 - c_1 + \sqrt{n} \cdot c_0$ assuming the induction hypothesis (IH) $T\left(\frac{n}{4}\right) = \sqrt{\frac{n}{4}} \cdot c_1 - c_1 + \sqrt{\frac{n}{4}} \cdot c_0$.

Proof.

$$
\begin{aligned}
T(n) &= c_1 + 2 \cdot T\left(\frac{n}{4}\right) && \text{(by definition of } T(n)\text{)} \\
&= c_1 + 2 \cdot \left(\sqrt{\frac{n}{4}} \cdot c_1 - c_1 + \sqrt{\frac{n}{4}} \cdot c_0\right) && \text{(by IH)} \\
&= c_1 + \sqrt{n} \cdot c_1 - 2 \cdot c_1 + \sqrt{n} \cdot c_0 && \text{(by arithmetic)} \\
&= \sqrt{n} \cdot c_1 - c_1 + \sqrt{n} \cdot c_0 && \square
\end{aligned}
$$

End of the exam