Delft University of Technology
EEMCS Faculty

CSE1305 Algorithms & Data Structures

Midterm Written Exam

7 December 2021, 09:00–10:30

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	14 questions (equal weights)	One correct answer per question	50%	$5 \cdot \frac{\text{score}}{14}$
Open questions	3 questions (different weights)	Multiple parts	50%	$5 \cdot \frac{\text{score}}{14}$

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 declaration form "Verklaring Tentamentijd Verlenging" together with your student card on your table. We don't want to interrupt you to be able to check this form.

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.
- 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%, 14 points)

1. (1 point) Which of the following statements is **true**?

   A. The expression $\sum_{k=1}^{n} \sum_{i=1}^{k} i$ is $\Omega(n^4)$.

   B. The Big-Ω, Big-Θ, and Big-$\mathcal{O}$ notations are used to denote respectively the best case, the average case, and the worst case runtimes of an algorithm.

   C. The Big-Ω, Big-Θ, and Big-$\mathcal{O}$ notations denote respectively a lower bound, a tight bound, and an upper bound on the runtime of an algorithm for a given case of interest.

   D. The expression $n^3 \log_2 n$ is $\mathcal{O}(n \log_8 n^3)$.

Answer:

A. False. The expression evaluates to $\frac{n^3+3n^2+2n}{6}$, so the fastest growing term of the polynomial is n^3. Since n^4 grows asymptotically faster than n^3, n^4 cannot be a lower bound for n^3.

B. False. All three notations can be used to provide bounds for different cases. For instance, we can use Big-Ω, Big-Θ, and Big-$\mathcal{O}$ to respectively provide a lower bound, a tight bound, and an upper bound for the best case.

C. True.

D. $n \log_8 n^3 = 3n \log_8 n$ and $3n \log_8 n$ cannot be an upper bound for $n^3 \log_2 n$, since the dominant factor $3n$ grows asymptotically slower than n^3.

2. (1 point) Given an algorithm with time complexity $\Theta(f(n))$ and space complexity $\Theta(g(n))$, which of the following claims is **true**?

   A. $g(n)$ is $\Omega(f(n))$.

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

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

   D. $f(n)$ is unrelated to $g(n)$.

Answer:

A. False. If $f(n)$ was a lower bound for $g(n)$, this would mean that the space used could grow faster than runtime. Since all space used has to be declared, this cannot happen.

B. True. Similar reasoning as above. The function of time spent can grow faster than the function of space used.

C. False. $f(n)$ is $\Omega(g(n))$ but it is not $\mathcal{O}(g(n))$ (for the same reason mentioned above).

D. False. There is a relationship, see above.

3. (1 point) Consider the implementation of method `select` below.

```java
public static Integer select(List<Integer> list, int k) {
  if (list == null || list.isEmpty())
    return null;
  if (list.size() == 1)
    return list.get(0);

  int elem = list.get(0);
  List<Integer> lesser = new ArrayList<>();
  List<Integer> equal = new ArrayList<>();
  List<Integer> greater = new ArrayList<>();
  for (int i : list) {
    if (i < elem) { lesser.add(i); }
    else if (i > elem) { greater.add(i); }
    else { equal.add(i); }
  }
  if (k <= lesser.size())
    return select(lesser, k);
  if (k <= lesser.size() + equal.size())
    return elem;
  return select(greater, k - lesser.size() - equal.size());
}
```

Assume that the input size n denotes the number of elements in `list`, and that the created sublists `lesser` and `greater` end up each with approximately $n/2$ of the elements of the input `list`. What is the recurrence equation for the runtime of algorithm `select`?

- **A.** $T(n) = T(n/2) + c_1 n + c_2$
- B. $T(n) = \frac{T(n/2)}{2} + c_1 n + c_2$
- C. $T(n) = 2T(n/2) + c_1 n + c_2$
- D. $T(n) = T(n-2) + c_1 n + c_2$

4. (1 point) Consider a sequence S of n elements. We would like to extract the subsequence of elements of sequence S between indices i and j, with $0 < i < j < n$ to make a new sequence P (note that extracting the subsequence means that the entire subsequence is removed from S). What are the time and space complexities of performing this operation when the sequences S and P are both implemented using either arrays or singly-linked lists (SLL), where you have full access to the array, as well as full access to the head reference and the references between the nodes of the SLL?

   - A. **array:** time $\mathcal{O}(n)$ and space $\mathcal{O}(1)$ **SLL:** time $\mathcal{O}(n)$ and space $\mathcal{O}(1)$
   - B. **array:** time $\mathcal{O}(n)$ and space $\mathcal{O}(n)$ **SLL:** time $\mathcal{O}(1)$ and space $\mathcal{O}(1)$
   - **C. array: time $\mathcal{O}(n)$ and space $\mathcal{O}(n)$ SLL: time $\mathcal{O}(n)$ and space $\mathcal{O}(1)$**
   - D. **array:** time $\mathcal{O}(n^2)$ and space $\mathcal{O}(n)$ **SLL:** time $\mathcal{O}(n^2)$ and space $\mathcal{O}(n)$

Answer:

Array create a new array P of size $j - i + 1$ in $\mathcal{O}(n)$ time, remove all elements between indices i and j from sequence S in $\mathcal{O}(n)$ time (copy elements between i and j to the new array P first, then shift the subsequent elements in array S backwards by an offset of $j + i + 1$). Space is $\mathcal{O}(n)$ because we need to create the new array P.

SLL traverse to find positions i and j in $\mathcal{O}(n)$ time, then detach the sequence of nodes between positions i and j in list S and create a new list P with the head reference pointing to the first node in the detached sublist in $\mathcal{O}(1)$ time (updating a few references to detach the sublist and make the new one). Space is $\mathcal{O}(1)$, since the sublist is kept intact, the nodes were already part of S, we only need a new head reference for the new list P.

5. (1 point) How many next and tail references need to be set or updated when inserting a new element at the head of a non-empty circularly-linked list?

   - A. 1 next reference
   - B. 1 next reference and the tail reference
   - **C. 2 next references**
   - D. 2 next references and the tail reference

6. (1 point) Consider the implementation of method `operation` below.

```
public static void operation(Queue queue) {
  if (queue == null) return;
  Stack<Integer> stack = new Stack<>();
  while (!queue.isEmpty()) {
    stack.push(queue.peek());
    queue.dequeue();
  }
  while (!stack.isEmpty()) {
    queue.enqueue(stack.peek());
    stack.pop();
  }
}
```

Assume that the queue was previously initialized with elements [1,2,3,4,5,6,7,8] enqueued into the queue in the order that they appear in the sequence from left to right. If we execute `operation(queue)` and then remove all elements in `queue` one by one by calling `queue.dequeue()`, which elements will be retrieved and in what order?

A. 1, 2, 3, 4, 5, 6, 7, 8
B. 1, 2, 3, 4, 8, 7, 6, 5
C. 8, 7, 6, 5, 1, 2, 3, 4
D. 8, 7, 6, 5, 4, 3, 2, 1

7. (1 point) Consider a positional list implemented using a doubly-linked list with header and trailer nodes (guard or dummy nodes). How many times are `prev` references used by method `addBefore` (either to get or set their value), which adds a new element before the given position? Assume the optimal implementation, that is, the references are not used more times than is needed to perform the operation.

A. 1 `prev` reference
B. 2 `prev` references
C. 3 `prev` references
D. 4 `prev` references

Answer: Access 1: `prev` of current position/node to get a reference to the predecessor node.
Access 2: set `prev` of new node to point to predecessor node.
Access 3: set `prev` current node to point to new node.

8. (1 point) Consider that a given data structure implemented using an array of capacity C contains n elements, where $C > n$. Which of the following operations does **not** necessarily place the new element at index n? Assume that the data structure is implemented such that it performs its typical operations in the most efficient way.

A. Insert the element into an unsorted list priority queue.
B. Add the element to a heap, before any bubbling operations.
C. Push the element onto the top of a stack.
D. Enqueue the element at the back of the queue.

Answer:

A. Inserts at index n. Since an unsorted list PQ doesn't care about keeping elements in order, it inserts elements where it's most efficient (in an array, this will be at the end).

B. Inserts at index n. To maintain the heap order property, a new element is always inserted just beyond the current last position of the heap. When the heap is implemented using an array, a heap with n elements will always occupy indices 0 to $n-1$. This means that a new element is inserted at index n.

C. Inserts at index n. The top of the stack is placed towards the end of the array, so that elements can be added to or removed from the top of the stack in $\mathcal{O}(1)$ time.

D. Inserts at index $(f+n)\%C$, where f denotes the front index of the queue. The elements are inserted at the end of the queue and removed from the front. The queue is allowed to wrap around the array.

9. (1 point) Which of the statements about the removal operation in a heap is **true**?

A. When removing the element at the root in an array-based heap, all subsequent elements need to be shifted backwards (or right to left) by one position.

B. After removing the element, we check if the heap-order property is satisfied between the root and its children. We perform down-heap bubbling if needed to restore the heap-order.

C. After removing the element, the bubbling operation starts at the last position in the heap and performs swaps along a single path up the tree until the heap-order is re-established.

D. Bubbling after removal takes $\mathcal{O}(n)$ time, where n is the number of elements in the heap.

Answer:

A. Shifting all elements by one position would take $\mathcal{O}(n)$, and it wouldn't guarantee the re-establishment of the heap-order property.

B. To keep the structure of the heap (completeness), the element at the root is removed by replacing or swapping with the element in the last position (this position is then excluded from the heap). The element that changes is at the root, so we need to check the heap-order property downwards, using downheap bubbling.

C. After removing the element, we may need to perform downheap bubbling from the root.

D. Bubbling after removal takes $\mathcal{O}(\log n)$ time, where n is the number of elements in the heap, since we move along one single path down the tree. In the worst-case we go from the root to a leaf, and perform a number of swaps that is proportional to the height of the heap, so $\mathcal{O}(\log n)$.

10. (1 point) Consider a linked tree implemented using nodes of the type shown below.

```
protected static class Node<E> {
  private E element;
  private Node<E> parent;
  private Node<E>[] children;
  /* ... */
}
```

Just like in family relationships, we define the cousin of a node `v` in a linked tree as a child of a sibling of the parent of `v`. Consider that node `v` is at index `k` in the array of children of its parent, and that the parent of `v` is at index `p` in the array of children of its own parent (which is also the grandparent of `v`). Where can we find the first cousin to the right of a given node `v` in a linked tree, if it exists?

A. `v.parent.children[k+1]`

B. `v.parent.parent.children[p-1]`

C. `v.parent.parent.children[p-1].children[0]`

D. `v.parent.parent.children[p+1].children[0]`

11. (1 point) Which of the following statements relating traversals and shape of a binary tree with at least 3 nodes is **false**?

    A. If the tree is complete, its postorder traversal visits all leaves before any internal nodes.

    B. If the tree is complete, its breadth-first traversal visits all internal nodes before all leaves.

    C. If the tree is a line, its preorder and breadth-first traversals visit all nodes in the same order.

    D. If the tree has the maximum possible number of nodes at every level, its inorder and breadth-first traversals visit all leaves in the same order.

Answer:

A. False. This works for a complete tree of height 1, i.e. with 3 nodes, but not for trees of larger height. With more levels, postorder will always have to visit the parent of the leaves before it can move on to other parts of the tree (including other leaves).

B. True. Breadth-first traverses the tree by level, from left to right. Since in a complete binary tree all leaves are guaranteed to be in the last positions (either all in the last level if the tree is full, or spanning the rightmost part of the second last and the leftmost part of the last level if the tree is not full), then they will also be found by the BFS in this order.

C. True. When a tree is a line, there is no choice to go wide, we can only move in depth. So breadth-first traversal becomes similar to a depth-first traversal in this kind of tree. Since both preorder and breadth-first visit the parent before the children, they end up visiting all nodes in the same order.

D. True. If the tree is full, then all leaves are at the same level. This means that breadth-first will deliver all leaves after the internal nodes, in left to right order. An inorder traversal will have to go up the tree and visit some ancestors in between visiting the leaves so that it can move sideways through the tree, but it will still visit all leaves from left to right, so the order of the leaves will be the same.

12. (1 point) Which of the following statements about sorting algorithms is **false**?

    A. Insertion and selection sort are more space efficient than merge sort.

    B. Insertion, selection, and heap sort can all be implemented in-place.

    C. Insertion sort does more comparisons than selection sort.

    D. Merge sort establishes an ordering of the elements in the combine step, when merging the two subsequences back together.

13. (1 point) What is the tightest upper bound on the total number of swap operations performed in the **worst-case** by the selection sort algorithm?

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

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

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

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

Answer: Selection sort performs $n-1$ swaps, since it always swaps the current element with the minimum element after it has found it. This can be optimized to perform 0 swaps when the minimum element happens to be at the current index being processed. If this is done, then in the best case (elements are already sorted) it will perform no swaps at all. But the worst-case will still be $n-1$.

14. (1 point) Consider that you apply the **in-place** heap sort algorithm to sort the following input sequence [23,10,4,16,5,2] in **decreasing** order. What is the state of the array after two removal operations (and any required bubbling)? Note that you are expected to use the most efficient heap construction algorithm.

    A. [5,10,23,16,4,2]

    B. [10,5,23,16,4,2]

    C. [5,16,10,23,4,2]

    D. [23,16,10,5,4,2]

Answer: In-place heap sort algorithm:
Step 1 - to sort in decreasing order in-place, we need to build a **min**-heap using heapify.
Step 2 - repeatedly call removeMin to remove the minimal element and perform any necessary bubbling (question specifies we do this only twice)

$[23, 10, \mathbf{4}, 16, 5, 2] \rightarrow [23, 10, \mathbf{2}, 16, 5, \mathbf{4}]$ heapify - downheap from index 2 (swap 4 with 2)
$[23, \mathbf{10}, 2, 16, 5, 4] \rightarrow [23, \mathbf{5}, 2, 16, \mathbf{10}, 4]$ heapify - downheap from index 1 (swap 10 with 5)
$[\mathbf{23}, 5, 2, 16, 10, 4] \rightarrow [\mathbf{2}, 5, \mathbf{4}, 16, 10, \mathbf{23}]$ heapify - downheap from index 0 (swap 23 w/ 2, then w/ 4)

$[\mathbf{2}, 5, 4, 16, 10, 23] \rightarrow [\mathbf{23}, 5, 4, 16, 10, \mathbf{2}]$ remove minimal element (swap root 2 with last 23)
$[\mathbf{23}, 5, 4, 16, 10, 2] \rightarrow [\mathbf{4}, 5, \mathbf{23}, 16, 10, 2]$ downheap from root (swap 23 with smallest child 4)
$[\mathbf{4}, 5, 23, 16, 10, 2] \rightarrow [\mathbf{10}, 5, 23, 16, \mathbf{4}, 2]$ remove minimal element (swap root 4 with last 10)
$[\mathbf{10}, 5, 23, 16, 4, 2] \rightarrow [\mathbf{5}, \mathbf{10}, 23, 16, 4, 2]$ dowheap from root (swap root 10 with smallest child 5)

Open questions (50%, 14 points)

15. (5 points) A student is asked to implement the merge sort algorithm. They remember the general idea of the algorithm, but are unsure about what data structure to use. The student decides to use a priority queue, since they remember that priority queues enables fast sorting. The student produces the following Java implementation of method `mergeSort`. Note that the `PriorityQueue` class implements a priority queue using a heap.

```
 1  public static void mergeSort(PriorityQueue<Integer> pq) {
 2    if(pq.size() < 2)
 3      return;
 4    int oldSize = pq.size();
 5    PriorityQueue<Integer> pq1 = new PriorityQueue();
 6    PriorityQueue<Integer> pq2 = new PriorityQueue();
 7
 8    while(pq.size() > oldSize/2)
 9      pq1.offer(pq.poll());
10    while(!pq.isEmpty())
11      pq2.offer(pq.poll());
12
13    mergeSort(pq1);
14    mergeSort(pq2);
15    while(!pq1.isEmpty() && !pq2.isEmpty()) {
16      if(pq1.peek() < pq2.peek())
17        pq.offer(pq1.poll());
18      else
19        pq.offer(pq2.poll());
20    }
21    while(!pq1.isEmpty())
22      pq.offer(pq1.poll());
23    while(!pq2.isEmpty())
24      pq.offer(pq2.poll());
25  }
```

As you might notice, this is a less desirable implementation of merge sort. Explain why using the recurrence equations for the runtime of this implementation and of the typical implementation of the merge sort algorithm. Give these 2 recurrence equations. Then explain what causes the difference between these recurrence equations and why the issue can be avoided (you do not have to explain all the terms that appear in both recurrence equations, explain only the terms that differ between them). You can assume that the number of elements n in the input priority queue pq is a power of two ($n = 2^k$), where k is an integer.

Answer:

$$\begin{aligned}
\text{This implemention: } & T(n) = c_1 + c_2 n + c_3 \cdot n \cdot \log_2(n) + 2 \cdot T(n/2) \\
& T(1) = c_0 \\
\text{Typical implemention: } & T(n) = c_1 + c_2 \cdot n + 2 \cdot T(n/2) \\
& T(1) = c_0
\end{aligned}$$

Methods `offer` and `poll` take $\mathcal{O}(\log k)$ time each on a heap with k elements (note that some of the `offer` calls take $\mathcal{O}(1)$ because we offer elements that are higher than all other elements in the heap, but the other `offer` and `poll` calls still make sure the $n \cdot \log_2 n$ term is there). In lines 9, 11, 17, 19, 22, and 24 these methods are performed over $n/2$ iterations each in the worst-case, on lists whose size respectively increases or decreases by one at each iteration. The time complexity over the $n/2$ iterations is given approximately by the sum $\sum_{k=1}^{n/2} \log_2 k$, which is $O(n \log n)$. So these operations take $c_3 \cdot n \cdot \log_2(n)$ time.

In the typical implementation of merge sort, using regular arrays/lists instead of priority queues, the operations to add and remove an element take $\mathcal{O}(1)$ time because it is sufficient to insert and remove at the end where it is most efficient. So if we replace the priority queue by one of these data structures, the operations above take $c_2 \cdot n$ time.

It can be seen from the recurrence equations that the priority queue is less efficient than using arrays/lists, because it reorganizes the heap every time to try to keep the elements in a certain order. This is however not necessary for merge sort to properly work.

16. (5 points) Derive the closed form of the following recurrence equation:

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

You may assume that the function is only called with integers that result in all recursive calls being called with an odd value for n, resulting in $\frac{n+1}{2}$ being a whole number. This means the function is only called for values $f(k) = 2^k + 1$ where k is a natural number, namely 2, 3, 5, 9, 17, 33, 65 etc.

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.

Note: one of the steps in the derivation/proof is a bit tricky. If you can't figure it out, do a reasonable guess and reason further from there, noting any inconsistencies that were caused by your incorrect guess. You can still get 4 out of 5 points for this question that way.

Answer: Option 1. By repeated unfolding:

$$\begin{aligned} T(n) &= c_1 + T\left(\frac{n}{2} + \frac{1}{2}\right) && \text{(by unfolding } T(n)\text{)} \\ &= c_1 + \left(c_1 + T\left(\frac{\frac{n}{2}+\frac{1}{2}}{2} + \frac{1}{2}\right)\right) && \text{(by unfolding } T(\tfrac{n+1}{2})\text{)} \\ &= 2c_1 + T\left(\frac{n}{4} + \frac{3}{4}\right) && \text{(by arithmetic)} \\ &= kc_1 + T\left(\frac{n}{2^k} + 1 - \frac{1}{2^k}\right) && \text{(by repeating } k \text{ times)} \\ &= \log_2(n-1)c_1 + T\left(\frac{n}{n-1} + 1 - \frac{1}{n-1}\right) && \text{(by letting } k = \log_2(n-1)\text{)} \\ &= \log_2(n-1)c_1 + T(2) && \text{(by arithmetic)} \\ &= \log_2(n-1)c_1 + c_0 && \text{(by definition of } T(0)\text{)} \end{aligned}$$

Option 2. By induction:

Closed form solution. The closed form solution of the above recurrence is $T(n) = \log(n-1)c_1 + c_0$. This can be obtained by repeatedly unfolding T.

Induction proof.
Base case: For $n = 2$, prove $T(n) = \log_2(n-1)c_1 + c_0$.

Proof.

$$\begin{aligned} T(n) = T(2) &= c_0 \\ &= 0c_1 + c_0 \\ &= \log_2(n-1)c_1 + c_0 \qquad \square \end{aligned}$$

Induction step: For $n > 0$, prove $T(n) = \log_2(n-1)c_1 + c_0$ assuming the induction hypothesis (IH) $T\left(\frac{n+1}{2}\right) = \log_2(\frac{n+1}{2} - 1)c_1 + c_0$.

Proof.

$$\begin{aligned}
T(n) &= c_1 + T\left(\frac{n+1}{2}\right) && \text{(by definition of } T(n)\text{)} \\
&= c_1 + \log_2\left(\frac{n+1}{2} - 1\right) c_1 + c_0 && \text{(by IH)} \\
&= \log_2\left(2 \cdot \left(\frac{n+1}{2} - 1\right)\right) c_1 + c_0 && \text{(by arithmetic)} \\
&= \log_2(n-1) + c_0 && \square
\end{aligned}$$

17. Prove that $\log_4(n)$ is $\Omega(\log_2(n))$.

(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:

$$\log_4(n) \geq c \cdot \log_2(n) \quad \text{for all } n \geq n_0.$$

(b) (2 points) Prove that these conditions hold. Explain all the steps in your proof.
Hint: you might need to use following logarithms rule: $\log_y(x) = \frac{\log_b(x)}{\log_b(y)}$

Answer: Let $c = \frac{1}{2}$ and $n_0 = 0$. When filling out the constant c in the formula, we obtain:

$$\begin{aligned}
\log_4(n) &\geq \frac{1}{2} \cdot \log_2(n) \\
\log_4(n) &\geq \log_4(2) \cdot \log_2(n) && \text{(by arithmetic)} \\
\log_4(n) &\geq \log_4(n) && \text{(by arithmetic)}
\end{aligned}$$

Since the sides are the same, this formula holds for any $n \geq n_0$ where $n_0 = 0$.