Name _____

Student Number _____

HAND IN
answers recorded
on question paper

BISHOP'S UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

CS 317

FIRST EXAMINATION

29 October 2025

Instructor: Stefan D. Bruda

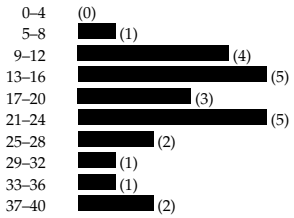
Instructions

- This examination is *80 minutes in length* and is *open book*. You are allowed to use any kind of printed documentation. Electronic devices are *not* permitted. You are *not* allowed to share material with your colleagues. *Any violation of these rules will result in the complete forfeiture of the examination.*
- There is no accident that the total number of marks add up to the length of the test in minutes. The number of marks awarded for each question should give you an estimate on how much time you are supposed to spend answering the question.
- *To obtain full marks provide all the pertinent details*. This being said, do not give unnecessarily long answers. In principle, all your answers should fit in the space provided for this purpose. If you need more space, use the back of the pages or attach extra sheets of paper. However, if your answer is not (completely) contained in the respective space, clearly mention within this space where I can find it.
- The number of marks for each question appears in square brackets right after the question number. If a question has sub-questions, then the number of marks for each sub-question is also provided.

When you are instructed to do so, turn the page to begin the test.

1	a,b,c	10	/	9				
2		10	/	10				
3	a,b,c,d	25	/	25				
4		5	/	7				
5		10	/	9				
6		20	/	20				
	Total:	80	/	80	=	40	/	40

The highest grade was 41 (103%), the lowest grade was 7 (18%), and the average grade was 20 (50%). Here is the grade distribution:



1. [9] For each relation below find *all* the $\mathbb{X} \in \{O, \Omega, \Theta, o, \omega\}$ that make the relation true. Justify your answer *using limits*.

   (a) [3] $2^{n+2} \in \mathbb{X}(8^n)$

   Answer:

   $\lim_{n\to\infty} \frac{2^{n+2}}{8^n} = \lim_{n\to\infty} \frac{1}{2^{3n-n-2}} = \lim_{n\to\infty} \frac{1}{2^{2n-2}} = 0$, so $2^{n+2} \in o(8^n)$, and therefore $\mathbb{X} \in \{o, O\}$.

   (b) [3] $\log n \in \mathbb{X}(\log n^2)$

   Answer:

   $\lim_{n\to\infty} \frac{\log n}{\log n^2} = \lim_{n\to\infty} \frac{\log n}{\log n \times n} = \lim_{n\to\infty} \frac{\log n}{\log n + \log n} = 1/2 = c$, so $\log n \in \Theta(\log n^2)$ which means that $\mathbb{X} \in \{\Theta, O, \Omega\}$.

   (c) [3] $\sqrt{n} \in \mathbb{X}(\log n)$

   Answer:

   We apply l'Hôspital's and so $\lim_{n\to\infty} \frac{\sqrt{n}}{\log n} = \lim_{n\to\infty} \frac{1/2n^{1/2}}{1/n \ln a} = \lim_{n\to\infty} \frac{n \ln a}{2n^{1/2}} = \lim_{n\to\infty} \frac{n^{1/2} \ln a}{2} = \infty$, so $\sqrt{n} \in \omega(\log n)$, and so $\mathbb{X} \in \{\omega, \Omega\}$.

   ---

2. [10] What is the complexity of the following algorithm. Explain formally how you counted the steps and then give the complexity in Θ notation.

```
i ← 1;    j ← 1
for i = 1 to n do
    while j < n do
        A_i ← A_j + 1
        j ← j × 3
    while j > 1 do
        A_j ← A_i − 1
        j ← j/2
for j = 1 to n do  PRINT(A_i)
```

ANSWER:

For the first while loop j starts at 1 and triples at every iteration. Therefore at iteration k we will have $j = 3^k$. The loop terminates as soon as $j = 3^k = n$ that is, $k = \log_3 n = \Theta(\log n)$. It follows that the loop executes $\Theta(\log n)$ times.

For the second while loop, j starts where the first loop left it that is, $j = n$. j is then halved at each iteration, meaning that at iteration k we have $j = n/2^k$. The loop terminates as soon as $j = n/2^k = 1$ that is, $n = 2^k$ or $k = \log_2 n$. Therefore this loop executes $\Theta(\log n)$ times.

The two while loops execute n times in the first for loop, followed by a $\Theta(1)$-time code executed n times in the second for loop. Therefore $T(n) = n(\Theta(\log n) + \Theta(\log n)) + n\Theta(1) = \Theta(n \log n + n) = \Theta(n \log n)$.

---

3. [25] Consider the following algorithm:

```
algorithm FindMax(A, n):
    if n = 1 then return A_1
    else
        k ← n/2
        for i = 1 to k do
            if A_i < A_{i+k} then A_i ↔ A_{i+k}
        return FindMax(A, k)
```

(a) [15] Prove formally that the algorithm returns the maximum value in the sequence $A_{1\ldots n}$.

Answer:

We proceed as usual by structural induction. For the base case $n = 1$ FindMax correctly returns the sole (and thus maximum) element A_1.

The invariant at the end of each iteration of the for loop is that all $A_{1\ldots i}$ are larger that all $A_{k\ldots i+k}$. This is clearly the case after the first iteration: if the sole value in $A_{1\ldots i}$ is smaller than the sole value in $A_{k\ldots i+k}$ then the if condition is true and so the two values are exchanged so that $A_{1\ldots i}$ becomes larger. Assuming that all $A_{1\ldots i-1}$ are less that all $A_{k\ldots i-1+k}$ at the beginning of an iteration then again if A_i is smaller than A_{i+k} then these two values are swapped and thus the whole sequence $A_{1\ldots i}$ becomes larger than $A_{k\ldots i+k}$. Thus at the end of the loop all the values in $A_{1\ldots k}$ are larger than the values in the rest of A and so the overall maximum is somewhere in there. This maximum is correctly returned by the recursive call to FindMax by inductive hypothesis.

(b) [4] Write down the recurrence relation for the running time of FindMax. Explain how you counted the steps.

Answer:

Clearly $T(1) = 0$ (we just return). For $T(n)$ we have the for loop which iterates $k = n/2$ times followed by the recursive call for $k = n/2$. That is, $T(n) = T(n/2) + n/2$.

(c) [4] Use the characteristic equation technique to solve the recurrence relation you found for Question 3b enough to give the running time of the algorithm in Θ notation.

Answer:

$T(n) = T(n/2) + n/2$. Let $n = 2^k$ (so that $k = \log n$) and $b_k = T(2^k)$. The relation becomes $b_k - b_{k-1} = 2^k/2$. The characteristic equation for the homogeneous part is $r - 1 = 0$ and so $r_1 = 1$. For the non-homogeneous part we have $b = 2$ and $d = 0$ which leads to $(r - 2)^1 = 0$ so that $r_2 = 2$. Therefore $b_k = c_0 1^k + c_1 2^k = \Theta(2^k)$. This in turn implies that $T(n) = \Theta(2^{\log n}) = \Theta(n)$.

(d) [2] Is the algorithm FindMax optimal? Explain one way or another.

Answer:

The algorithm is optimal, since in order to find the maximum of n values we need to check all the values at least once and so the lower bound for the problem is $\Omega(n)$.

---

4. [7] Give an algorithm that determines whether an undirected graph $G = (V, E)$ contains a cycle. Your algorithm must run in $O(|V|)$ time (independent on $|E|$). Explain the idea behind your algorithm and the running time, but you do not need to provide any formal proof.

ANSWER:

The graph has a cycle iff in a traversal we encounter a vertex a second time. Since G is undirected the two paths that allow us to reach a vertex twice constitute a cycle.

The usual depth-first search runs in $O(|V| + |E|)$ time, but we can just count the vertices encountered and always stop at $|V| + 1$ returning True. If we have to stop earlier then we return False. We have:

```
vcount ← 0
algorithm FindCycle(G = (V, E)):
    Let v ∈ V
    if v exists then recFindCycle(v)
    else return False
algorithm recFindCycle(v):
    vcount ← vcount + 1
    if vcount = |V| + 1 then return True
    foreach (v, u) ∈ E do recFindCycle(u)
    return False
```

---

5. [9] Recall the median-of-medians Quickselect algorithm presented in class:

```
algorithm MoMSelect(k, S, l, h):
    if h − l ≤ 25 then use brute force
    else
        m ← (h − l)/5
(a)     for i = 1 to m do M_i ← MedianOfFive(S_{l+5i−4...l+5i})
(b)     mom ← MoMSelect(m/2, M, 1, m)
        S_1 ↔ S_mom
        p ← Partition(S, l, h)
        if k = p then return S_k
        else if k < p then MoMSelect(k, S, l, p − 1)
        else MoMSelect(k, S, p + 1, h)
```

Refine the lines (a) and (b) of the algorithm so that the sequence M or medians does not require additional storage space. Explain how your modifications maintain the correctness and running time of the algorithm.

ANSWER:

The requested change is super easy to figure out once we notice that the particular location of values in the sequence *never* matters in the Quicksort family of algorithms. Indeed, we move values around all the time when we partition.

This observation suggests that we can reuse the first m indices in S to store M. We only need to make sure that the existing values therein are moved elsewhere before being overwritten, so we perform an exchange instead of mere assignment when we assign to M_i (now S_i). We assume a slightly modified MEDIANOFFIVE that returns the index of the median rather than the value (trivial to do and we do not care anyway since MEDIANOFFIVE operates on a constant number of values so we can spend as much time as we want and the running time will still be constant). With this modification the requested refinement goes like this:

(a) **for** $i = 1$ **to** m **do**
    $k \leftarrow \text{MEDIANOFFIVE}(S_{l+5i-4 \ldots l+5i})$
    $S_i \leftrightarrow S_k$
(b) $mom \leftarrow \text{MOMSELECT}(m/2, S, 1, m)$

Note that by the time we assign to S_i we are long past the need to consider it in MEDIANOFFIVE, for indeed i goes up one by one in the for loop while the range of MEDIANOFFIVE goes up five by five in that loop.

6. [20] A majority element is an element that occurs in more than half the elements of a sequence. For example, the element 3 is the majority element in $\langle 3, 2, 3, 3, 7, 8, 3, 3\rangle$, but the sequence $\langle 3, 2, 3, 3, 7, 8, 3, 2\rangle$ has no majority element.

   Design a divide and conquer algorithm that receives a sequence of integers $S_{1\ldots n}$ and returns the majority element of the sequence or NotFound if there is no majority element. Determine the running time of your algorithm. You must also explain the idea behind the algorithm well enough to convince me that it is correct, but you do not need to provide a formal proof.

   Hint: If n is a majority element of S then n must also be a majority element of either the first half of S or the second half of S (or both).

Answer:

We recursively find the majority elements in the first and second halves of S. By the hint above we just reduced the field to two candidates. We then simply count the number of occurrences of the two candidates in S and return whichever happens to be the overall majority. If no majority is thus established we return NotFound. Note that we do not care about further counting the occurrences of NotFound since that count will always be 0.

```
algorithm Majority(S, l, h):
    if l = h then return S_l                                   T(1) = 0
    else
        m ← (h − l)/2                                          Θ(1) time
        m_l ← Majority(S, l, m)                                T(n/2) time
        m_r ← Majority(S, m + 1, h)                            T(n/2) time
        if m_l = m_r then return m_l
        if Count(S, l, h, m_l) > (h − l + 1)/2 then return m_l Θ(n) time
        if Count(S, l, h, m_r) > (h − l + 1)/2 then return m_r Θ(n) time
        return NotFound
algorithm Count(S, l, h, v):
    c ← 0
    for i = l to h do
        if S_i = c then c ← c + 1
    return c
```

Let input size be $n = h - l$. Counting clearly takes linear time, and so $T(n) = 2T(n/2) + n$, with $T(0) = 0$ (also see the time annotations above). We have already seen this recurrence (it is the same as MergeSort) and so we already know that $T(n) = \Theta(n \log n)$.
