Instructions:

- The questions are divided into three groups. You have to answer all questions in Groups 1 and 2 and one question in Group 3. In the above table, put a check mark in the small box beside the question in Group 3 you want me to mark.

- Provide your answer in the box after each question. If you absolutely need extra space, use the backs of the pages; but try to avoid it. Keep your answers short and to the point.

- You are not allowed to use a cheat sheet.

- For the design questions (Group 3), if you cannot design an algorithm that achieves the stated running time, design a slower algorithm that is correct. A correct and slow algorithm earns you 50% of the marks. A fast and incorrect algorithm earns 0 marks.

- In the design questions, you are allowed to use algorithms and data structures you learnt in class as black boxes, without explaining how they work, as long as these algorithms and data structures do not directly answer the question.

- Read every question carefully before answering. In particular, do not waste time on an analysis if none is asked for, and do not forget to provide one if it is required.

- Do not forget to write your banner number and name on the top of this page.

- This exam has 13 pages, including the title page. Notify me immediately if your copy has fewer than 13 pages.
Question 1.1 (Asymptotic running time) 15 marks

a. Explain the following two terms.

- Worst-case running time

The worst-case running time of an algorithm is $T(n)$ if there is no input of size $n$ on which the algorithm takes longer than $T(n)$ time, and there is an input of size $n$ on which the algorithm takes $T(n)$ time.

- Average-case running time

The average-case running time of an algorithm is $T(n)$ if $T(n)$ is the average over the running times of the algorithm for all possible inputs of size $n$.

b. Given two algorithms $A$ and $B$ with running times $T_A(n)$ and $T_B(n)$, what does it mean that $T_A(n) = O(T_B(n))$.

If $T_A(n) = O(T_B(n))$, then for sufficiently large $n$, algorithm $A$ is by at most a constant factor slower than algorithm $B$. 
Question 1.2 (Asymptotic growth)  

20 marks

a. Sort the following functions by order of growth:

1. $2^{\log_4 n} = \sqrt{n}$
2. $\frac{n^3}{2} \lg n$
3. $\frac{n^{3/2}}{\lg n}$

b. Prove that the function $f(n)$ in box 1 and the function $g(n)$ in box 2 satisfy $f(n) = o(g(n))$.

$$\lim_{n \to \infty} \frac{\sqrt{n}}{n \lg n} = \lim_{n \to \infty} \frac{1}{\sqrt{n} \lg n} = 0.$$ 

c. Prove that the function $g(n)$ in box 2 and the function $h(n)$ in box 3 satisfy $g(n) = o(h(n))$.

$$\lim_{n \to \infty} \frac{n \lg n}{n^{3/2} / \lg n} = \lim_{n \to \infty} \frac{\lg^2 n}{\sqrt{n}}$$

$$= \left( \lim_{n \to \infty} \frac{\lg n}{n^{1/4}} \right)^2$$

$$= \left( \lim_{n \to \infty} \frac{\lg e^{1/n}}{n^{-3/4}/4} \right)^2$$

$$= \left( \lim_{n \to \infty} \frac{4 \lg e}{n^{1/4}} \right)^2$$

$$= 0^2$$

$$= 0.$$
Consider the following procedure:

\[
\text{MAGIC}(x) \\
\quad \triangleright x \text{ is an integer} \\
1 \quad \text{if } x > 0 \\
2 \quad \text{then MAGIC}([x/2]) \\
3 \quad \text{if } x \text{ is odd} \\
4 \quad \text{then print '1'} \\
5 \quad \text{else print '0'}
\]

a. Given a positive integer \(x\), what does this procedure print on the screen? Do not prove your claim. (Your statement should be simple and shouldn’t simply rephrase the computation of the algorithm in mathematical terms.)

The procedure prints out \(x\) in binary.

b. Let the input “size” be the value of \(x\). Give a recurrence for the running time of the algorithm in terms of the value of \(x\).

\[
T(x) = \begin{cases} 
\Theta(1) & x = 0 \\
\Theta(1) + T([x/2]) & x > 0 
\end{cases}
\]
c. Solve this recurrence using whatever method you like, in order to determine the running time of procedure MAGIC.

Observe that we have $T(x) \leq \Theta(1) + T(x/2)$ for $x > 0$, that is, we can ignore the floor. Master theorem:
In the language of the Master theorem, we have $a = 1$, $b = 2$, and $f(n) = \Theta(1)$. Since $n^{\log_b a} = n^0 = \Theta(1) = \Theta(f(n))$, the second case applies, and we get $T(x) = \Theta(\log x)$.
Substitution:
We guess that $T(x) = O(\log x)$, that is, $T(x) \leq c \log x$, for some $c > 0$ and all $x \geq 2$. For $x \in \{2, 3\}$, this is most certainly true because in this case $T(x) = \Theta(1)$, and we can choose $c$ large enough so that $T(x) \leq c \log x$.
For $x \geq 4$, we use induction:

\[
T(x) \leq a + c \log(x/2)
= a + c(\log x - 1)
= c \log x + (a - c)
\leq c \log x,
\]
as long as $c \geq a$. 

Question 2.2 (Correctness proof) 20 marks

Consider the following procedure:

DIVISIBLEBYNINE(x)
1 if x < 10
2 then if x = 0 or x = 9
3 then return “yes”
4 else return “no”
5 else y ← 0
6 while x > 0
7 do y ← y + (x mod 10)
8 x ← ⌊x/10⌋
9 return DIVISIBLEBYNINE(y)

a. Argue that the algorithm always terminates, that is, that the algorithm does not call itself recursively until all eternity. (Hint: Prove that the value of y in Line 9 is less than the value of x at the beginning of the procedure and draw the right conclusions.)

We prove this by induction on x. For x < 10, the algorithm obviously terminates because it executes Lines 1–4 and, which take constant time.

For x ≥ 10, we prove that the value of y in Line 9 is less than x. Thus, for x ≥ 10, the algorithm spends constant time in Lines 1, 5–8 and then invokes itself on y in Line 9. By the induction hypothesis, the recursive call on y terminates. Thus, the current invocation on x terminates.

To prove that x > y, let us write

\[ x = \sum_{i=0}^{k} x_i 10^i. \]

Then it is easy to see that

\[ y = \sum_{i=0}^{k} x_i. \]

This implies that

\[ x - y = \sum_{i=0}^{k} (10^i - 1)x_i. \]

Every term in this summation is non-negative because \( x_i \geq 0 \) and \( 10^i - 1 \geq 0 \), for all \( 0 \leq i \leq k \). Since \( x \geq 10 \), there must be an \( i > 0 \) such that \( x_i > 0 \). For this value of i, we have \( x_i > 0 \) and \( 10^i - 1 \geq 10 - 1 > 0 \). Thus, \( x - y > 0 \), that is, \( x > y \).
b. Prove that the algorithm always gives the correct answer. (Hint: Use induction. For the inductive step, argue that \( y \) is divisible by 9 if and only if this is true for \( x \). Use the fact that 9 divides \( 10^k - 1 \) for any integer \( k \).)

The proof is again by induction. For \( x < 10 \), the algorithm obviously returns the correct answer because 0 and 9 are the only two numbers less than 10 that are divisible by 9.

If \( x \geq 10 \), we need to prove that \( x - y \) is divisible by 9. Indeed, since \( y < x \), we can assume inductively that \( \text{DIVISIBLEBYNINE}(y) \) returns “yes” if and only if \( y \) is divisible by 9. If \( y \) is divisible by 9, then \( y = 9y' \), for some integer \( y' \). Similarly \( x - y = 9q \), for some integer \( q \). Thus, \( x = y + (x - y) = 9y' + 9q = 9(y' + q) \), that is, \( x \) is divisible by 9, and the recursive call gives the correct answer for \( x \). If \( y \) is not divisible by 9, then \( y = 9y' + y'' \), where \( y' \) and \( y'' \) are integers and \( 0 < y'' < 9 \). Then \( x = y + (x - y) = 9y' + y'' + 9q = 9(y' + q) + y'' \), that is, \( x \) also leaves remainder \( y'' \) when divided by 9; \( x \) is not divisible by 9, and the “no” answer returned by the recursive call is again correct for \( x \). It remains to prove that \( x - y \) is divisible by 9.

We have observed in Part (a), however, that

\[
x - y = \sum_{i=0}^{k} (10^i - 1)x_i.
\]

\( 10^i - 1 \) is divisible by 9 for any integer \( i \). Thus, every term in the summation is divisible by 9 and, thus, \( x - y \) is divisible by 9.
Question 3.1 (Greedy algorithms) 25 marks

You are going on a trip from Halifax to San Francisco. You know that, along the route, there are gas stations at kilometers \( k_1, k_2, \ldots, k_n \). You also know that your car can go \( m \) kilometers on a tank full of gas. Develop an algorithm that determines at which gas stations to stop so that

a. The distance between two consecutive stops is at most \( m \) (that is, you can always get from one stop to the next on a tank full of gas) and

b. You make as few stops as possible while respecting the previous condition.

The running time of your algorithm should be linear. You do not have to prove that it is. Prove that your algorithm is correct, that is, that it indeed produces a sequence of stops that satisfies the above conditions.

The question is in fact somewhat underspecified because, as it stands, we would have no reason ever to stop at the last gas station because we don’t know how long the trip will continue after it. So let me assume here that the last gas station is in San Francisco (one can assume that a city of the size of San Francisco has a gas station), and let us assume that I want to arrive with a full tank. So I always stop at the last gas station to fill up the car. This leads to the following algorithm.

\[
\text{ComputeStops}(k, n, m)
\]
\[
\begin{align*}
\triangleright & \text{ } k \text{ is the array of gas station positions; } n \text{ is the number of gas stations} \\
\triangleright & \text{ } m \text{ is the mileage of my car} \\
1 & s \leftarrow 0 \triangleright \text{The position of my first stop, the start.} \\
2 & S \leftarrow \emptyset \triangleright \text{The set of gas stations where I stop.} \\
3 & \text{for } i \leftarrow 1 \text{ to } n \text{ - } 1 \\
4 & \quad \text{do if } k[i + 1] - s > m \\
5 & \quad \quad \text{then } S \leftarrow S \cup \{k[i]\} \\
6 & \quad s \leftarrow k[i] \\
7 & \text{return } S \cup \{k_n\}
\end{align*}
\]

The running time of the algorithm is obviously linear. In each step, we include the current gas station if we cannot make it to the next one from the last stop on one tank full of gas. Let \( S = (k_{i_1}, k_{i_2}, \ldots, k_{i_q}, k_n) \) be the sequence of stops we compute, and let \( O = (k_{j_1}, k_{j_2}, \ldots, k_{j_r}, k_n) \) be the optimal sequence. We prove by induction \( h \) that \( k_{i_h} \geq k_{j_h} \), for all \( h \). This then implies that \( q \leq r \): Indeed, assume that \( q > r \). Then we know that \( n - k_r \leq m \) because, as \( O \) is an optimal sequence of stops, we must be able to reach for the stop at kilometer \( k_{j_r} \) to our final stop at kilometer \( k_n \). But \( k_n - k_{j_r} \geq n - k_r \) because \( k_{j_r} \leq n \). Hence, \( k_{i_{r+2}} - k_{i_r} \leq k_n - k_{i_r} \leq m \). This contradicts that we included stop \( k_{i_r+1} \) in our sequence of stops. Thus, \( q \leq r \), and our stop sequence is as short as possible. It remains to prove the central claim: \( k_{i_h} \geq k_{j_h} \), for all \( h \).

For \( h = 1 \), the claim is most certainly true because our first stop is at kilometer \( k_{i_1} \) because we could not reach kilometer \( k_{i_1+1} \) on a tank full of gas from the start kilometer \( 0 \).

For \( h > 1 \), we distinguish two cases: If \( k_{i_{h-1}} \geq k_{j_h} \), then the claim is trivial because \( k_{i_h} \geq k_{i_{h-1}} \). So let us assume that \( k_{i_{h-1}} < k_{j_h} \). In that case, our sequence \( S \) cannot include any stop at a kilometer strictly between \( k_{i_{h-1}} \) and \( k_{j_h} \) because \( k_{j_h} - k_{i_{h-1}} \leq k_{j_h} - k_{j_{h-1}} \leq m \).

(Note that this uses the induction hypothesis: \( k_{i_{h-1}} \geq k_{j_{h-1}} \).) This implies that \( k_{i_h} \geq k_{j_{h-1}} \), and we are done.
Extra space for Question 3.1:
Question 3.2 (Divide and conquer)  

Let \( A = (x_1, \ldots, x_n) \) be an array storing \( n \) numbers, and let \( [a_1, b_1], \ldots, [a_n, b_n] \) be \( n \) intervals with \( 1 \leq a_i \leq b_i \leq n \), for all \( 1 \leq i \leq n \). Your task is to develop an algorithm which, for every interval \( [a_i, b_i], \) computes the value \( m_i = \min(x_j | a_i \leq j \leq b_i) \). Note that your algorithm has to compute all values \( m_1, m_2, \ldots, m_n \) at the same time. The running time of your algorithm should be \( \mathcal{O}(n \lg n) \). Do not analyze your algorithm and do not prove its correctness.

The algorithm is based on the following idea: Split array \( A \) into pieces \( L = A[1 \ldots n/2] \) and \( R = A[n/2 + 1 \ldots n] \). Any minimum \( m_i \) with \( b_i \leq n/2 \) can be computed by recursively applying the algorithm to \( L \). Similarly, any minimum \( m_i \) with \( a_i > n/2 \) can be computed recursively by applying the algorithm to \( R \). Thus, the only minima we have to worry about are those where \( a_i \leq n/2 \) and \( b_i > n/2 \). For these, we observe that \( m_i = \min(s_{a_i}, p_{b_i}) \), where \( s_j = \min(A[j], A[j + 1], \ldots, A[n/2]) \) and \( p_j = \min(A[n/2 + 1], A[n/2 + 2], \ldots, A[j]) \).

What does this buy us? Well, once we have all the \( s_j \) an \( p_j \) values, we can compute all the \( m_i \) that aren’t computed in recursive calls in constant time. The \( s_j \) and \( p_j \) can be computed in linear time because \( s_j = \min(A[j], s_{j+1}) \) and \( p_j = \min(p_{j-1}, A[j]) \). Thus, we obtain the following algorithm:

\[
\text{RangeMinima}(A, Q, l, r)
\]

\[\begin{align*}
\triangleright & \ A \text{ is the array; } Q \text{ contains the query} \\
\triangleright & \ A[l \ldots r] \text{ is the subarray on which the current recursive call operates.} \\
& 1 \text{ if } Q = \emptyset \\
& 2 \quad \text{then return immediately} \\
& 3 \quad \text{if } l = r \\
& 4 \quad \text{then for every interval } I \in Q \\
& 5 \quad \quad \text{do Output } A[l] \text{ as the answer} \\
& 6 \quad \text{else } q \leftarrow \lceil \frac{l + r}{2} \rceil \\
& 7 \quad \quad s[q - 1] \leftarrow A[q - 1] \\
& 8 \quad \quad \text{for } i \leftarrow q - 2 \text{ downto } l \\
& 9 \quad \quad \quad \text{do } s[i] \leftarrow \min(s[i + 1], A[i]) \\
& 10 \quad \quad \text{for } i \leftarrow q + 1 \text{ to } r \\
& 11 \quad \quad \quad \text{do } p[i] \leftarrow \min(p[i - 1], A[i]) \\
& 12 \quad \quad Q_l \leftarrow \emptyset \\
& 13 \quad \quad Q_r \leftarrow \emptyset \\
& 14 \quad \text{for every interval } [a, b] \in Q \\
& 15 \quad \quad \text{do if } a < q \text{ and } b \geq q \\
& 16 \quad \quad \quad \text{then Output } \min(s[a], p[b]) \text{ as the answer} \\
& 17 \quad \quad \text{else if } \text{then } Q_l \leftarrow Q_l \cup \{[a, b]\} \\
& 18 \quad \quad \text{else } Q_r \leftarrow Q_r \cup \{[a, b]\} \\
& 19 \quad \quad \text{RangeMinima}(A, Q_l, l, q - 1) \\
& 20 \quad \quad \text{RangeMinima}(A, Q_r, q, r)
\end{align*}\]

The running time of each invocation, excluding the recursive calls, is obviously linear in the size of the current subarray \( A[l \ldots r] \). The two recursive calls are on subarrays of \( A[l \ldots r] \) containing only half the elements. Thus, the recurrence is the same as for MergeSort: \( T(n) = 2T(n/2) + \Theta(n) \), which solves to \( T(n) = \Theta(n \lg n) \).
Extra space for Question 3.2:
Given an undirected graph $G$, a $3$-cycle is a triple of vertices $(u, v, w)$ such that edges $(u, v)$, $(u, w)$, and $(v, w)$ belong to $G$. Assume that $G$ is given in adjacency list representation. Develop an $O(nm)$-time algorithm that finds all such cycles in $G$. Make sure that your algorithm reports every cycle exactly once. Prove that the running time of your algorithm is indeed $O(nm)$.

We start by observing that a $3$-cycle is defined by an edge $(u, v)$ whose endpoints $u$ and $v$ have a common neighbour $w$. Thus, the basic idea is to iterate over all $m$ edges and identify all common neighbours of the endpoints of each of them. This would of course report every $3$-cycle three times, once per edge. We ensure that we report it only once by outputting the $3$-cycle $(u, v, w)$ only for the edge $(u, v)$ such that $w > u, v$ in some numbering of the vertices. So our algorithm is as follows:

**Find3Cycles($G$)**

1. Number the vertices $1$ through $n$ in an arbitrary order.
2. For every vertex $v \in G$, sort the edges in its adjacency list $E(v)$ by the endpoints not equal to $v$.
3. for every edge $(u, v) \in G$
   4. do $(u, x) \leftarrow$ the first edge in $E(u)$
   5. $(v, y) \leftarrow$ the first edge in $E(v)$
   6. while $(u, x) \neq \text{nil}$ and $(v, y) \neq \text{nil}$
      do if $x = y$ and $x > u$ and $x > v$
         then Output the $3$-cycle $(u, v, x)$
      8. $(v, y) \leftarrow$ next edge in $E(v)$
      9. $(u, x) \leftarrow$ next edge in $E(u)$
   11. else if $x > y$
       then $(v, y) \leftarrow$ next edge in $E(v)$
   12. else $(u, x) \leftarrow$ next edge in $E(u)$

The running time of this algorithm is easy to analyze. Numbering the vertices takes $O(n)$ time because there are $n$ vertices. Sorting the adjacency lists takes $O(|E(v)| \lg |E(v)|)$ per adjacency list $E(v)$. Since $|E(v)| \leq n$ (every vertex can be adjacent to at most $n - 1$ other vertices), the cost per adjacency list is $O(|E(v)| \lg n)$, and the total cost is

$$O \left( \sum_{v \in G} |E(v)| \lg n \right) = O(m \lg n) = O(mn).$$

Finally, the cost of the while-loop is bounded by $O(nm)$ because it has $m$ iterations, one per edge, and every iteration spends constant time per entry in two adjacency list of total size at most $2n - 2$. 
Extra space for Question 3.3: