Banner number: Name:

Midterm Exam

CSCI 3110: Design and Analysis of Algorithms

October 30, 2015

Group 1		Group 2		Group 3		$\sum$
Question 1.1		Question 2.1		Question 3.1		
Question 1.2		Question 2.2		Question 3.2		
$\sum$		$\sum$		$\sum$		

Instructions:

- The questions are divided into three groups: Group 1 (36%), Group 2 (40%), and Group 3 (24%). You have to answer **all questions in Groups 1 and 2** and **exactly one questions 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. If you select none or both questions, I will randomly choose which one 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.**
- **Make sure your answers are clear and legible. If I can't decipher an answer or follow your train of thought with reasonable effort, you'll receive 0 marks for your answer.**
- If you are asked to design an algorithm and you cannot design one that achieves the desired running time, design a slower algorithm that is correct. A correct and slow algorithm earns you 50% of the marks for the algorithm. A fast and incorrect algorithm earns 0 marks.
- When designing an algorithm, you are allowed to use algorithms and data structures you learned 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 8 pages, including this title page. Notify me immediately if your copy has fewer than 8 pages.**

Question 1.1 (Worst-case and average-case running time) — 9 marks

(a) Define what the worst-case running time of an algorithm is.

The worst-case running time of an algorithm is a function T of the input size n such that $T(n)$ is the maximum running time over all possible inputs of size n.

(b) Define what the average-case running time of an algorithm is.

The average-case running time of an algorithm is a function T of the input size n such that $T(n)$ is the average running time over all possible inputs of size n.

Question 1.2 (Asymptotic growth of functions) — 9 marks

(a) Define the set $\Theta(f(n))$.

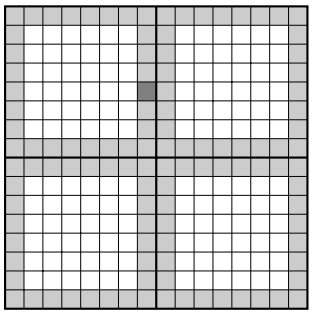
$\Theta(f(n))$ *is a set of functions. A function* $g(n)$ *belongs to* $\Theta(f(n))$ *if there exist constants* $0 < c_1 \leq c_2$ *and* $n_0 \geq 0$ *such that for all* $n \geq n_0$, $c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$.

(b) Define the set $o(f(n))$.

$o(f(n))$ *is a set of functions. A function* $g(n)$ *belongs to* $o(f(n))$ *if, for all* $c > 0$, *there exists a constant* $n_0 \geq 0$ *such that for all* $n \geq n_0$, $g(n) \leq c \cdot f(n)$.

Question 2.1 (Recurrence relations) — 10 marks

Assume you're given an $n \times n$ matrix M, where n is a power of 2 and no two cells in the matrix store the same value. Finding the maximum element in M takes $\Theta(n^2)$ time. This maximum element is also a *local maximum* in the sense that it is greater than its (at most) eight neighbours in the matrix. What if we care only about finding a local maximum, even if it is not the maximum element in the matrix? Here's an algorithm that solves this problem. Specifically, the algorithm returns the position (i, j) of a local maximum. If $n = 1$, then we return the position of the only element in M. If $n > 1$, we split M into four $\frac{n}{2} \times \frac{n}{2}$ submatrices and inspect the elements in the first row, last row, first column, and last column of each submatrix. We recurse on the submatrix that contains the largest of these elements and return whatever position (i, j) this recursive call produces.



For a 16×16 matrix, we inspect the grey cells. If the dark grey cell holds the largest element among these cells, we recurse on the top-left submatrix.

Give a recurrence relation for the running time of this algorithm, explain why the function it describes is indeed the running time of the algorithm, and solve this recurrence relation whichever way you like. Show the steps you take to solve the recurrence.

The algorithm inspects four complete rows and four complete columns of the input matrix. Each such row or column contains n elements, so in total less than 8n elements are inspected, at constant cost per element. After that, the algorithm recurses on one $\frac{n}{2} \times \frac{n}{2}$ submatrix. Therefore, its running time is given by the recurrence

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n).$$

Now $\log_2 1 = 0 < 1$ and, for $f(n) = n$, we have $1 \cdot f\left(\frac{n}{2}\right) = \frac{n}{2} = \frac{1}{2} \cdot f(n)$. Thus, the Master Theorem states that this recurrence has the solution $T(n) \in \Theta(n)$.

Question 2.2 (Correctness proofs) 10 marks

Prove that the algorithm in Question 2.1 returns the position of a local maximum in M. Specifically, you should prove the following claim, where a *boundary element* of M is an element in the first row, last row, first column or last column of M:

> If the algorithm returns the position (i, j), then the element stored at this position in M is a local maximum and is no less than any boundary element of M.

Again, assume for simplicity that no two elements in M are the same.

By induction on n.

If $n = 1$, the algorithm returns the only element in M. This element is a local maximum and is no less than any boundary element of M.

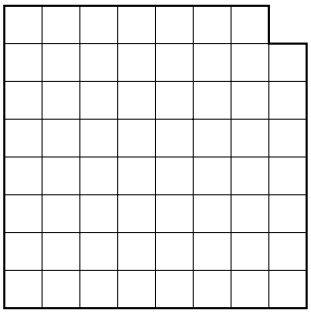
If $n > 1$, let M_{11}, M_{12}, M_{21}, and M_{22} be the four submatrices into which we divide M and assume w.l.o.g. that we recurse on M_{11}. Thus, the maximum boundary element b of M_{11}, M_{12}, M_{21}, and M_{22} is a boundary element of M_{11}. By the inductive hypothesis, the recursive call on M_{11} returns a local maximum m of M_{11} such that $m \geq b$.

First we argue that m is no less than any boundary element b' of M: Since b is the maximum boundary element of M_{11}, M_{12}, M_{21}, and M_{22}, we have $b \geq b'$. Since $m \geq b$, this implies that $m \geq b'$ for every boundary element b' of M.

Next we argue that m is a local maximum of M: If m is not a boundary element of M_{11}, then it is also a local maximum of M because it has the same eight neighbours in M_{11} as in M. If m is a boundary element of M_{11}, then $m = b$. Indeed, m is no less than any boundary element of M_{11}, m is a boundary element of M_{11}, b is the maximum boundary element of M_{11}, and no two elements in M are the same. Now, since m is a local maximum in M_{11}, all its neighbours in M_{11} are less than m. All its neighbours not in M_{11} are boundary elements of M_{12}, M_{21} or M_{22}. Since $m = b$ and b is the maximum boundary element of M_{11}, M_{12}, M_{21}, and M_{22}, m is also greater than its neighbours outside of M_{11}, that is, it is greater than all *its neighbours in M. Thus, it is a local maximum of M.*

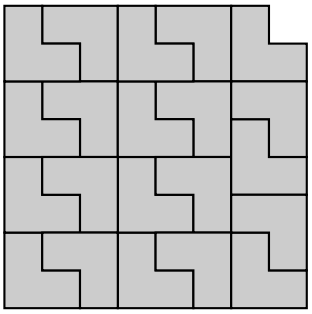
Question 3.1 (Divide and conquer) — 12 marks

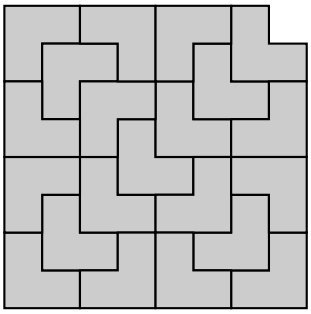
A *deficient* $n \times n$ grid is an $n \times n$ grid with its top-right corner missing. Let's refer to this missing corner as the grid's *deficiency*. A *tromino* is a deficient 2×2 grid. A *tiling* of a deficient $n \times n$ grid with trominoes is a partition of the grid into trominoes. Alternatively, you can think of it as placing trominoes on the deficient $n \times n$ grid so that the entire grid is covered and no two trominoes overlap. In the tiling, you are allowed to rotate trominoes, that is, the deficiency of the tromino does not have to be in the top-right corner.



A deficient 8×8 grid

A tromino



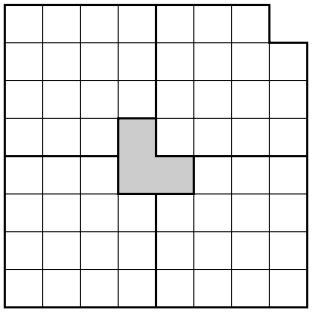


Two possible tilings of the deficient 8×8 grid

Develop an algorithm that tiles a deficient $n \times n$ grid with trominoes. Assume that n is a power of 2. Describe your algorithm and argue briefly that it is correct. The running time of the algorithm should be $O(n^2)$, that is, linear in its output size (the output consists of $(n^2-1)/3$ trominoes). Give a recurrence for the running time of your algorithm and argue briefly that its solution is indeed $O(n^2)$. In your description of the algorithm, do not worry about how the output is to be represented. You only need to ensure that your description makes it clear where you place trominoes, in which orientation.

If $n = 2$, the grid to be tiled has exactly the same shape as a tromino. So we tile it by placing a single tromino, with its deficiency in the top-right corner:

If $n > 2$, we start by placing a tromino in the center of the grid, with its deficiency in the top-right corner. This leaves us with four deficient $\frac{n}{2} \times \frac{n}{2}$ grids. The bottom-left and top-right ones once again have their deficiencies in their top-right corners. The top-left one has its deficiency in the bottom-right corner. The bottom-right one has its deficiency in the top-left corner:



We can tile the bottom-left and top-right grids by calling the algorithm recursively. The other two grids can also be tiled by calling the algorithm recursively, after rotating the coordinate system 90 degrees to the right and left, respectively.

Extra space for Question 3.1

In the base case ($n = 2$), the algorithm produces the only correct solution. If $n > 2$, the four recursive class produce correct tilings of the four subgrids, by the induction hypothesis. The combination of these four tilings leaves three of the four center grid cells unoccupied. The tromino we place in the center of the grid covers exactly these three cells, so we obtain a valid tiling of the entire $n \times n$ deficient grid.

The algorithm spends constant time to place the center tromino and then makes four recursive calls on $\frac{n}{2} \times \frac{n}{2}$ grids. Thus, the running time is given by the recurrence

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(1).$$

By the Master Theorem, this recurrence has the solution $T(n) \in \Theta(n^2)$.

Question 3.2 (Graph algorithms) 12 marks

Consider a connected graph $G = (V, E)$ each of whose edges is coloured either red or blue. Your goal is to partition the vertex set V into two subsets U and W such that every blue edge either has both its endpoints in U or both its endpoints in W whereas every red edge has one endpoint in U and one endpoint in W. Develop an algorithm that takes $O(n + m)$ time to decide whether such a partition exists; $n = |V|$, $m = |E|$. If it does, the algorithm should output the partition. Argue briefly that your algorithm takes $O(n + m)$ time and that it is correct. (*Hint:* If all edges were red, this would be bipartiteness testing.)

Solution 1:

We start by computing a spanning tree T of G and placing its root r into U. Using breadth-first search or depth-first search to compute T takes $O(n + m)$ time.

The other vertices are placed into U or W using a preorder traversal of T. This ensures that every vertex v is visited after its parent p_v. If the edge between v and p_v is blue, we add v to the same set as p_v; if it is red, we add v to the opposite set. This second step of the algorithm involves a preorder traversal of T and otherwise spends constant time per vertex. Thus, the cost of this step is $O(n)$.

Finally, we inspect all edges of G. For every red edge (u, v), we check whether u and v belong to opposite sets. For every blue edge (u, v), we check whether u and v belong to the same set. If none of these tests fails, we output the pair of sets (U, W). If any of these tests fails, we output that there is no partition (U, W) that satisfies all edges. This final step that tests whether every edge is satisfied takes constant time per edge, $O(m)$ time in total. By summing the costs of the three steps, we obtain that the whole algorithm takes $O(n + m)$ time.

Now for the correctness: If the algorithm outputs a partition (U, W), it obviously satisfies all edges of G because the final step of the algorithm verifies explicitly that all edges of G are satisfied. So the algorithm gives the correct answer in this case. If the algorithm claims that there is no partition (U, W) that satisfies all edges, we argue as follows that this is correct:

Once we place r into U, the partition (U, W) the second step of the algorithm computes is the only partition that satisfies all edges of T. This is easily shown for every vertex $v \in G$ by induction on the distance from r to v in T. If this distance is 0, then $v = r$ and $v \in U$. If the distance from r to v is $d > 0$, then v has a parent p_v with distance $d - 1$ from r. By the inductive hypothesis, the set we place p_v in is the only valid choice for p_v. Thus, if the edge (p_v, v) is blue, the only valid choice is to place v into the same set; if the edge is red, the only valid choice is to place v into the opposite set.

Now, since the only partition that satisfies the edges of T is the one computed in Step 2 of our algorithm, then if this partition does not satisfy all edges of G, there is no partition that satisfies all edges of G. Thus, our algorithm's answer is correct.

Extra space for Question 3.2

Solution 2:

First we introduce a new vertex v_e for every blue edge $e = (u, w)$ and replace this edge with two red edges (u, v_e) and (v_e, w). Next we apply the bipartiteness testing algorithm from class to the resulting graph G'. If this algorithm reports that G' is not bipartite, we report that there is no partition that satisfies all edges of G. Otherwise, let (U', W') be the partition of the vertex set of G' produced by the bipartiteness testing algorithm. We obtain the two sets U and W by removing the new vertices v_e introduced above from U' and W', respectively, and return the resulting pair of sets (U, W).

The first step takes $O(m)$ time, constant time per edge. The second step takes $O(n' + m')$ time, where n' and m' are the numbers of vertices and edges in G', respectively. Every vertex of G' is a vertex of G or one of the at most m vertices introduced for the blue edges. Thus, $n' \leq n + m$. Every edge of G' is either an edge of G or one of the two edges replacing a blue edge of G. Thus, $m' \leq 2m$. The second step of the algorithm thus takes $O(n + m + 2m) = O(n + m)$ time. The third step finally takes $O(n') = O(n + m)$ time because it only requires scanning U' and W' and discarding the vertices that do not belong to G.

Now the correctness: First assume we output a partition (U, W), that is, G' is bipartite and the second step produces a partition (U', W'). Consider an edge $e = (u, w)$ of G. If this edge is red, it is also an edge of G'. Thus, w.l.o.g. $u \in U'$ and $w \in W'$, that is, $u \in U$ and $w \in W$ as required. If the edge is blue, G' contains two edges (u, v_e) and (v_e, w). Thus, neither u nor w belongs to the same set as v_e, that is, u and w belong to the same set as required.

Now assume there exists a partition (U, W) that satisfies all edges of G. We obtain a valid bipartition of G' by initializing $U' = U$ and $W' = W$ and placing every vertex v_e corresponding to a blue edge (u, w) in the opposite set from the one containing u and w. This is well defined because the endpoints of every blue edge belong to the same set. The partition (U', W') satisfies each of the two edges replacing a blue edge. It also satisfies every edge of G' that is a red edge of G because its two endpoints belong to opposite sets. Thus, (U', W') is a valid bipartition of G', that is, our algorithm does not report failure.