Banner number: Name:

Midterm Exam

CSCI 3110: Design and Analysis of Algorithms

June 24, 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 (18 marks = 36%), Group 2 (20 marks = 40%), and Group 3 (12 marks = 24%). You have to answer **all questions in Groups 1 and 2** and **exactly one question in Group 3**. In the above table, put a check mark in the **small** box beside the one question in Group 3 you want me to mark. If you select 0 or 2 questions in Group 3, I will mark neither.
- 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. Keey your answers short and to the point. If you need scrap paper, you can also use the backs of the pages.
- **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 questions.
- **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 9 pages, including this title page. Notify me immediately if your copy has fewer than 9 pages.**

Question 1.1 (Asymptotic growth of functions) — 9 marks

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

$o(f(n))$ is the set of all functions $g(n)$ with the property that, for all $c > 0$, there exists a constant $n_c \geq 0$ such that

$$g(n) \leq c \cdot f(n) \quad \forall n \geq n_c.$$

b. Running times of algorithms are always positive functions, that is, $T(n) > 0$ for all n. Use the definition of o-notation given above to show that, for two positive functions $f(n)$ and $g(n)$, $f(n) \in o(g(n))$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

$$\begin{aligned}
f(n) \in o(g(n)) &\iff \forall c > 0\, \exists n_c \geq 0\, \forall n \geq n_c : f(n) \leq c \cdot g(n) && \textit{(definition of o-notation)} \\
&\iff \forall c > 0\, \exists n_c \geq 0\, \forall n \geq n_c : \frac{f(n)}{g(n)} \leq c && \textit{(because } g(n) > 0 \textit{ for all } n\textit{)} \\
&\iff \forall c > 0\, \exists n_c \geq 0\, \forall n \geq n_c : \left|\frac{f(n)}{g(n)}\right| \leq c && \textit{(because } \frac{f(n)}{g(n)} > 0 \textit{ for all } n\textit{)} \\
&\iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 && \textit{(definition of limits)}
\end{aligned}$$

Question 1.2 (Running time of algorithms) 9 marks

Let A and B be two algorithms that solve the same problem P. Assume A's average-case running time is in $\Theta(n)$ while its worst-case running time is in $\Theta(n^2)$. B's average-case running time is in $\Theta(n \lg n)$, as is its worst-case running time. The constants hidden by the Θ-notation are much smaller for A than for B and A is much easier to implement than B. Now consider a number of real-world scenarios where you would have to solve problem P. State which of the two algorithms would be the better choice in each of the following scenarios and briefly justify your answer.

a. The inputs are fairly small.

In this case, the smaller constant factors in algorithm A will likely make A faster than B, even in the worst case, so we'd prefer A over B. The better worst-case running time of B outweighs the smaller constant factors of A only for sufficiently large inputs.

b. The inputs are big and fairly uniformly chosen from the set of all possible inputs. You want to process a large number of inputs and would like to minimize the total amount of time you spend on processing them all.

Since the inputs are chosen fairly uniformly from the set of all possible inputs, the average time it takes to process one of the inputs using A is in $\Theta(n)$. If there are m inputs, the total time to process the inputs can therefore be expected to be in $\Theta(mn)$. Using B, processing these m inputs takes $\Theta(mn \lg n)$ time. Combined with the smaller constant factors in algorithm A, this clearly makes A the better choice.

c. The inputs are big and heavily skewed towards A's worst case. As in the previous case, you want to process a large number of inputs and would like to minimize the total amount of time you spend on processing them all.

Since the inputs are heavily skewed towards A's worst case, most of the inputs will take $\Theta(n^2)$ time to process, for a total time of $\Theta(mn^2)$ to process m inputs. Using B, it takes only $\Theta(mn \lg n)$ time to process the same inputs. Since the inputs are big, the better constant factors of A are unlikely to outweigh this advantage of B, so B is likely the better choice.

d. The inputs are of moderate size, neither small nor huge. You would like to process them one at a time in real time, as part of some interactive tool for the user to explore some data collection. Thus, you care about the response time on each individual input.

Since the inputs are not small, quadratic running time cannot be expected to be negligible. Since I care about response time on each individual input, the resulting delay in processing the input is unacceptable. Thus, I need an algorithm that guarantees a performance better than $\Theta(n^2)$ for every single input: algorithm B.

Question 2.1 (Asymptotic growth of functions) 10 marks

a. Order the following functions by increasing order of growth:

$$n \lg n \quad n^{\lg n} \quad (\lg n)^n \quad 2^{\frac{\lg n}{2}} \quad n^2$$

$$2^{\frac{\lg n}{2}} = \sqrt{n} \quad n \lg n \quad n^2 \quad n^{\lg n} \quad (\lg n)^n$$

b. Prove that you have arranged the second and third functions in the sorted sequence in the right order; that is, if the sorted sequence is $f_1(n), f_2(n), \ldots, f_5(n)$, prove that $f_2(n) \in o(f_3(n))$.

The second function is $n \lg n$, the third function is n^2. We need to prove that $\lim_{n\to\infty} \frac{n \lg n}{n^2} = 0$, which holds if and only if $\lim_{n\to\infty} \frac{\lg n}{n} = 0$. To prove the latter, we apply l'Hôpital's rule:

$$\lim_{n\to\infty} \frac{\lg n}{n} = \lim_{n\to\infty} \frac{1/(n \ln 2)}{1} = \lim_{n\to\infty} \frac{1}{n \ln 2} = 0.$$

c. Prove that $f(n) = 3n^2 + n \lg n - 10n \in \Theta(n^2)$ by providing constants $0 < c_1 < c_2$ and $n_0 \geq 0$ such that $c_1 n^2 \leq f(n) \leq c_2 n^2$ for all $n \geq n_0$.

For all $n \geq 0$, we have $10n \geq 0$, so $3n^2 + n \lg n - 10n \leq 3n^2 + n \lg n$. We also have $2 \geq \lg 2$, $\frac{dn}{dn} = 1$, and $\frac{d \lg n}{dn} = \frac{1}{n \ln 2} < 1$ for all $n \geq 2$, so $n \geq \lg n$ for all $n \geq 2$. This shows that, for all $n \geq 2$, $n \lg n \leq n^2$ and thus $3n^2 + n \lg n \leq 4n^2$. In summary, we have $f(n) \leq 4n^2$ for all $n \geq 2$.

For all $n \geq 1$, we have $n \lg n \geq 0$, so $3n^2 + n \lg n - 10n \geq 3n^2 - 10n$. For $n \geq 10$, we have $n^2 \geq 10n$, so $3n^2 - 10n \geq 2n^2$. In summary, we have $f(n) \geq 2n^2$ for all $n \geq 10$.

Putting these two inequalities together, we have $2n^2 \leq f(n) \leq 4n^2$ for all $n \geq \max(2, 10) = 10$; $c_1 = 2$, $c_2 = 4$, and $n_0 = 10$.

Question 2.2 (Correctness proofs) — 10 marks

Here's yet another minimum spanning tree algorithm:

MST(G)

```
T := G
while T contains a cycle
  do Choose an arbitrary cycle C in T
     Remove the heaviest edge in C from T
return T
```

Prove that this algorithm does indeed produce a minimimum spanning tree of G. You may assume that G is connected and that there are no two edges in G that have the same weight. Your proof needs to show that the algorithm terminates, that the result is a spanning tree of G, and that it has minimum weight among all spanning trees of G.

First we argue that the algorithm terminates. Since we start with the finite graph $T = G$ and each iteration of the while loop removes one edge from T, we leave T without edges after a finite number of iterations. At this point, T is guaranteed to be cycle-free, so the while-loop exits and the algorithm terminates.

It is easy to show that T is a tree when the while loop exits: Since G is connected and we initially set $T = G$, T is initially connected. Removing an edge from a cycle cannot disconnect the graph. Thus, T remains connected at all times. The while loop exits only once the graph contains no more cycles. At this point, T is connected and contains no cycles, that is, it is a tree.

Next assume that T is not an MST of G. Let T' be an MST of G. In particular, $T' \neq T$, so there exists an edge $e' \in T'$ that does not belong to T. The removal of e' from T' splits T' into two subtrees T'_1 and T'_2. Let U be the vertex set of T'_1 and W be the vertex set of T'_2.

Since $e' \in G$ but $e' \notin T$, the construction of T from G must have removed e' as the heaviest edge of some cycle C. The path P obtained by removing e' from C has one endpoint in U and one in W. Thus, it must contain an edge e that crosses the cut (U, W). Since $e \in C$, e' is the heaviest edge in C, and no two edges have the same weight, we have $w(e) < w(e')$. Let $T'' = T' \setminus \{e'\} \cup \{e\}$. Since T' is connected, $T' \cup \{e\}$ is connected. T'' is obtained by removing the edge e' from a cycle in $T' \cup \{e\}$, so T'' is also connected. The only cycle in $T' \cup \{e\}$ is the one formed by adding e because T' is a tree. This cycle is broken by the removal of e', so T'' contains no cycle. Thus, T'' is connected and acyclic, that is, it is a tree. The weight of T'' is $w(T'') = w(T') + w(e) - w(e') < w(T')$ because $w(e) < w(e')$. Thus, since T'' is a spanning tree of G, T' cannot be an MST of G, a contradiction.

Question 3.1 (Graph algorithms) 12 marks

Given a graph G whose edges have positive weights, a *shortest path* between two vertices u and w is a path from u to w whose edges have minimum total weight among all paths from u to w. The *distance* $\text{dist}(u,w)$ between u and w is the length of such a shortest path. The *diameter* of G is the maximum distance between any two vertices in G: $\text{diam}(G) = \max_{u,v \in G} \text{dist}(u,v)$. In general, computing the diameter of G boils down to computing all pairwise vertex distances and reporting the largest one. If G is a tree, however, things get quite a bit simpler.

Develop an algorithm that computes the diameter of a tree T whose edges have positive weights. The running time of your algorithm should be in $O(n)$, where n is the number of vertices of T. Describe your algorithm, argue that its running time is in $O(n)$, and prove that it does indeed compute the diameter of T.

The first observation is that, since the edge weights are positive, the two vertices u and v such that $\text{diam}(T) = \text{dist}(u,v)$ are leaves. Indeed, if v is not a leaf, then there exists a leaf v' such that v belongs to the path from u to v', that is, $\text{dist}(u,v') = \text{dist}(u,v) + \text{dist}(v,v') > \text{dist}(u,v)$ because all edge weights are positive and thus $\text{dist}(v,v') > 0$. The same argument can be applied to argue that u must be a leaf. Thus, all we have to find is the maximum distance between two leaves of T.

Let r be an arbitrary leaf of T. From here on, we no longer consider r to be a leaf of T and instead call it the root of T. The parent *of a node v is now the first node after v on the path from v to r in T. For two nodes u and w, the lowest common ancestor (LCA) of u and w is the node closest to r on the path from u to w.*

Now let P be the set of all pairs of leaves of T. Then, as we observed already $\text{diam}(T) = \max_{(u,v)\in P} \text{dist}(u,v)$. We can partition P into sets P_x, for $x \in V(T)$, such that $(u,v) \in P_x$ if and only if x is the LCA of u and v. This gives $\text{diam}(T) = \max_{x \in V(T)} \max_{(u,v)\in P_x} \text{dist}(u,v)$. Since T has n vertices, computing $\text{diam}(T)$ thus takes linear time once we are given $m_x := \max_{(u,v)\in P_x} \text{dist}(u,v)$ for all $x \in V(T)$.

To compute these values m_x for all $x \in V(T)$, we start by computing values d_x for all $x \in V(T)$. For every $x \in V(T)$, d_x is the maximum distance from x to some descendant leaf y of x. Formally, $d_x := \max_{y \in D_x} \text{dist}(x,y)$, where D_x is the set of all leaves y such that x belongs to the path from r to y. Assume for now, we have already computed these values. Then we observe that we can compute m_x, for every $x \in V(T)$, in linear time in the number of its children in T:

If $x = r$, then r is the LCA of u and v if and only if w.l.o.g. $u = r$ and $v \in D_r$. Thus, $P_r = \{(r,y) \mid y \in D_r\}$ and $m_r = d_r$.

If $x \neq r$, consider a pair $(u,v) \in P_x$ and let u' and v' be the children of x such that $u \in D_{u'}$ and $v \in D_{v'}$. Then $\text{dist}(u,v) = \text{dist}(u,u') + w(u',x) + w(x,v') + \text{dist}(v',v)$. If $\text{dist}(u,v) = m_x$, then $\text{dist}(u,u') = d_{u'}$ and $\text{dist}(v',v) = d_{v'}$. Assume $\text{dist}(u,u') < d_{u'}$. Then there exists a leaf $u'' \in D_{u'}$ with $\text{dist}(u'',u') = d_{u'} > \text{dist}(u,u')$. We have $(u'',v) \in P_x$ and $\text{dist}(u'',v) = \text{dist}(u'',u') + w(u',x) + w(x,v') + \text{dist}(v',v) > \text{dist}(u,u') + w(u',x) + w(x,v') + \text{dist}(v',v) = \text{dist}(u,v)$, contradicting the assumption that $\text{dist}(u,v) = m_x$. The same argument shows that $\text{dist}(v',v) = d_{v'}$. Thus, to compute m_x, all we need to find is the pair of children (u',v') such that $d_{u'} + w(u',x) + d_{v'} + w(v',x)$ is maximized. This sum is maximized if we maximize the two terms $h_{u'} := d_{u'} + w(u',x)$ and $h_{v'} := d_{v'} + w(v',x)$ individually, respecting the constraint that $u' \neq v'$. Thus, we can compute m_x using a single scan of the list of children of x during which we keep track of the two children u' and v' of x we have seen so far that maximize $h_{u'}$ and $h_{v'}$.

Extra space for Question 3.1

Since we can compute m_x in $O(|C_x|)$ time for every $x \in V(T)$, where C_x is the set of x's children and every node $y \in V(T)$ is the child of at most one node $x \in V(T)$, computing m_x for all $x \in V(T)$ thus takes $O(n)$ time.

It remains to compute d_x for all $x \in V(T)$. To do this, we process the nodes by decreasing distance (measured in number of edges, not their weight) from r. This is easily done in linear time using a DFS traversal of T starting from r and processing every node x immediately before the DFS backtracks from x. Since the children of x have been visited by this time, the value d_y is known for each such child y by the time we attempt to compute d_x. Now it suffices to observe that $d_x = \max_{y \in C_x}(w(x,y) + d_y)$, where C_x is the set of x's children, that is, d_x can be computed in $O(|C_x|)$ time from the d_y values of its children. In total, this takes linear time over all nodes of T.

To see that $d_x = \max_{y \in C_x}(w(x,y)+d_y)$, we start by observing that $d_x \geq \max_{y \in C_x}(w(x,y)+d_y)$. Indeed, choose $y' \in C_x$ such that $w(x,y') + d_{y'} = \max_{y \in C_x}(w(x,y) + d_y)$. Then there exists a leaf $y'' \in D_{y'}$ such that $d_{y'} = \mathrm{dist}(y', y'')$. For this leaf, $\mathrm{dist}(x, y'') = w(x,y') + \mathrm{dist}(y', y'') = w(x,y') + d_{y'}$. Since $y'' \in D_x$, this shows that $d_x = \max_{z \in D_x} \mathrm{dist}(x,z) \geq \mathrm{dist}(x,y'') = w(x,y') + d_{y'}$.

To see that $d_x \leq \max_{y \in C_x}(w(x,y)+d_y)$, let $z'' \in D_x$ such that $\mathrm{dist}(x,z'')$, and let z' be the child of x on the path from x to z''. Then $\mathrm{dist}(x,z'') = w(x,z') + \mathrm{dist}(z',z'') \leq w(x,z') + d_{z'} \leq \max_{y \in C_x}(w(x,y)+d_y)$.

Question 3.2 (Greedy algorithms) — 12 marks

In this question, you are given a set $C = \{1, 5, 10, 25\}$ of coin denominations, that is, pennies, nickels, dimes, and quarters. Your goal is to design an algorithm which, for any input value V in cents, outputs the minimum number of coins with denominations in C whose total value is exactly V. For example, to represent the value $V = 37$, we would choose one quarter, one dime, one nickel, and two pennies, a total of 5 coins. Your algorithm should take $O(|C|) = O(1)$ time. An $O(V + |C|)$-time algorithm is also acceptable.

Describe your algorithm, argue briefly that it does indeed achieve the desired running time, and prove formally that it outputs the minimum number of coins needed to represent a given value V, for any value V it is given as input.

Here's a simple constant-time algorithm:

***MinimumNumberOfCoins*(V)**

$n_q := \lfloor V/25 \rfloor$
$V_1 := V - 25n_q$
$n_d := \lfloor V_1/10 \rfloor$
$V_2 := V_1 - 10n_d$
$n_n := \lfloor V_2/5 \rfloor$
$V_3 := V_2 - 5n_n$
$n_p := V_3$
Use n_q quarters, n_d dimes, n_n nickels, and n_p pennies to represent V

The algorithm clearly takes constant time. The chosen coins have a total value of V: $V_3 = n_p$. $V_2 = V_3 + 5n_n = n_p + 5n_n$. $V_1 = V_2 + 10n_d = n_p + 5n_n + 10n_d$. $V = V_1 + 25n_q = n_p + 5n_n + 10n_d + 25n_q$. Moreover, the produced solution is feasible because $n_q, n_d, n_n, n_p \geq 0$. To see this, observe that $V \geq 0$ and hence $n_q = \lfloor V/25 \rfloor \geq 0$. Since $n_q = \lfloor V/25 \rfloor$, we have $25n_q \leq V$, so $V_1 = V - 25n_q \geq 0$. By the same arguments, $V_2, V_3, n_d, n_n, n_p \geq 0$.

We refer to the solution produced by our algorithm as the tuple (n_q, n_d, n_n, n_p). The number of coins in this solution is $|(n_q, n_d, n_n, n_p)| := n_q + n_d + n_n + n_p$. We prove that this is the unique optimal solution, that is, there exists no optimal solution (n'_q, n'_d, n'_n, n'_p) that does not satisfy $n'_q = n_q$, $n'_d = n_d$, $n'_n = n_n$, and $n'_p = n_p$. So consider any optimal solution (n'_q, n'_d, n'_n, n'_p).

First observe that $n'_p < 5$. Otherwise, $(n'_q, n'_d, n'_n + 1, n'_p - 5)$ would be a better solution than (n'_q, n'_d, n'_n, n'_p). Similarly, $n'_n < 2$ because otherwise $(n'_q, n'_d + 1, n'_n - 2, n'_p)$ would be a better solution.

Next observe that $n'_q \leq n_q$ because $25n'_q \leq V$ and $n_q = \lfloor V/25 \rfloor$. If $n'_q < n_q$, then $(0, n'_d, n'_n, n'_p)$ is a solution for the value $V_1 = V - 25n'_q$, and it must be an optimal solution because if there existed a better solution $(n''_q, n''_d, n''_n, n''_p)$ for V_1, then $(n'_q + n''_q, n''_d, n''_n, n''_p)$ would be a better solution for V. Now observe that $V_1 = V - 25n'_q \geq 25n_q - 25n'_q \geq 25$.

If $n'_d \geq 3$, then $(0, n'_d, n'_n, n'_p)$ is not optimal because the solution $(1, n'_d - 3, n'_n + 1, n'_p)$ also represents V_1 and uses one less coin.

If $n'_d = 2$, then $n'_n \geq 1$ or $n'_p \geq 5$ because $10n'_d + 5n'_n + n'_p = V_1 \geq 25$. Since we already observe that $n'_p < 5$, we must have $n'_n \geq 1$ and $(1, n'_d - 2, n'_n - 1, n'_p)$ is a better solution for V_1. So $n'_d \neq 2$.

Extra space for Question 3.2

If $n'_d < 2$, then $n'_n \geq 2$ or $n'_p \geq 5$ because $10n'_d + 5n'_n + n'_p = V_1 \geq 25$. However, we observed that $n'_n < 2$ and $n'_p < 5$, so this is also impossible.

Since we obtain a contradiction to the optimality of (n'_q, n'_d, n'_n, n'_p) for every choice of n'_d if $n'_q < n_q$, we must have $n'_q = n_q$, both $(0, n'_d, n'_n, n'_p)$ and $(0, n_d, n_n, n_p)$ represent the value $V_1 = V - 25n_q$ and, as observed above, the representation $(0, n'_d, n'_n, n'_p)$ must be optimal.

Once again, since $10n'_d \leq V_1$ and $n_d = \lfloor V_1/10 \rfloor$, we have $n'_d \leq n_d$. If $n'_d < n_d$, then $(0, 0, n'_n, n'_p)$ must be an optimal representation of the value $V_2 = V_1 - 10n'_d \geq 10n_d - 10n'_d \geq 10$. We observed that $n'_n < 2$. This, however, implies that $n'_p \geq 5$ because $5n'_n + n'_p = V_2 > 10$, which is also impossible. Thus, $n'_d < n_d$ is impossible and we have $n'_d = n_d$.

One final time: since $(0, 0, n'_n, n'_p)$ is an optimal representation of V_2 and $n_n = \lfloor V_2/5 \rfloor$, we have $n'_n \leq n_n$. If $n'_n < n_n$, then $n'_p = V_2 - 5n'_n \geq 5n_n - 5n'_n \geq 5$, a contradiction because $n'_p < 5$. Thus, $n'_n = n_n$.

Since $n'_q = n_q$, $n'_d = n_d$, and $n'_n = n_n$, we also have $n'_p = V - 25n'_q - 10n'_d - 5n'_n = V - 25n_q - 10n_d - 5n_n = n_p$, so the two solutions (n_q, n_d, n_n, n_p) and (n'_q, n'_d, n'_n, n'_p) are the same.