Midterm Exam 1 Solutions

Problem 1 (Miscellaneous).

(a) Each of the items below describes a function of a parameter $n$. For each one, give a simple big-theta ($\Theta$) expression for the function, of the form $\Theta(2^{an}n^b \log^c n)$ where $a, b, c \geq 0$.

(a) The running time of Kruskal’s algorithm, implemented using a binary heap, on graphs with $n$ vertices and $n^{0.5}$ edges.

There was a typo here and the question did not count. The correct answer is $\Theta(n)$, but many people were confused by the fact that the graph has fewer edges than vertices.

(b) The solution to the recurrence given by $T(n) = 9T([n/3]) + n^2$ and $T(1) = 1$.

$\Theta(n^2 \log n)$ (since $\log_3 9 = 2$)

(c) The solution to the recurrence given by $T(n) = 10T([n/3]) + n^2$ and $T(1) = 1$.

$\Theta(n \log_{10} 10)$ (since $\log_3 10 > 2$)

(d) The solution to the recurrence given by $T(n) = T(n - 1) + 2^n$ and $T(1) = 1$.

$T(n) = 2^{n+1} - 2 = \Theta(2^n)$

(e) $\sum_{i=1}^{n} i 3^i$

$\Theta(n 3^n)$. Clearly the sum is at least $n 3^n$. One can also bound it above by $n \sum_{i=1}^{n} 3^i = n \frac{3^n - 1}{2}$.

(f) $\sum_{i=1}^{n} \sqrt{i}$

$\Theta(n^3)$. The sum $\sum_{i=1}^{K} \sqrt{i}$ is $\Theta(K^{3/2})$ (to see the lower bound, notice that at least half the terms are at least $\sqrt{K/2}$). Plugging in $K = n^2$ gives the answer.

(g) $\sum_{i=1}^{n} \frac{i}{3^i}$

$\Theta(1)$. In the limit as $n$ goes to $\infty$, this sum converges to a constant.

(b) In an effort to simplify its more obscure practices, Flying Dutchman Airlines (FDA) has decided on a new pricing model: the cost of a trip in dollars will equal the sum of the lengths, in miles, of the two longest segments. So if the two longest segments are 200 and 3000 miles long, respectively, the trip will cost $3200 (simple doesn’t mean cheap). Trips with only one segment will cost the length of that segment.
A clause in FDA’s charter states that it can only use or modify algorithms originally designed by Dutch computer scientists. They have asked you to modify Dijsktra’s algorithm to compute the cheapest flight between two given airports $s$ and $t$. The input is a directed graph $G$ in which every vertex is an airport, and an edge of length $\ell(u,v)$ represents a flight they operate from $u$ to $v$ that covers $\ell$ miles.

Indicate in the pseudocode of Algorithm 1 which lines you would change, and how.

[Hint: you’ll have to store some extra information at each node.]

---

**Algorithm 1: Dijkstra($G = (V,E), \ell(\cdot), s,t$)**

1. **for each vertex** $v \in V$ **do**
   2. $\text{previous}[v] \leftarrow \text{undefined}$; /* Previous node in optimal path from $s$ */
   3. $\text{first}[v] \leftarrow \infty$; /* Length of longest leg on shortest path to $v$ */
   4. $\text{second}[v] \leftarrow 0$; /* Length of second longest leg */
   5. $\text{dist}[v] \leftarrow \infty$; /* Unknown distance from source to $v$ */
   6. $\text{dist}[s] \leftarrow 0$;
   7. $\text{first}[s] \leftarrow 0$;
   8. $Q \leftarrow$ empty priority queue keyed on $\text{dist}[]$;
   9. $Q \leftarrow V$;
10. **while** $Q$ **is not empty** **do**
    11. $u \leftarrow \text{extract-min}(Q)$;
    12. **for each neighbor** $v$ **of** $u$ **do**
    13. $\text{alt} \leftarrow \text{dist}[u] + \ell(u,v)$;
    14. $\text{alt} \leftarrow \text{SumOfTwoLargest}(\ell(u,v), \text{first}[u], \text{second}[u])$;
       /* SumOfTwoLargest() returns the sum of its two largest arguments; $\text{alt}$ is the length of the shortest path that goes to $v$ through $u$ */
    15. **if** $\text{alt} < \text{dist}[v]$ **then**
    16. $\text{dist}[v] \leftarrow \text{alt}$; /* Implicitly performs decrease-key in $Q$. */
    17. $\text{previous}[v] \leftarrow u$;
    18. $\text{first}[v] \leftarrow \max(\ell(u,v), \text{first}[u], \text{second}[u])$;
    19. $\text{second}[v] \leftarrow \text{alt} - \text{first}[v]$;
    20. return $\text{dist}[t]$ ;
(c) Let $G$ be a weighted connected graph $G = (V,E)$ with weight function $w : V \rightarrow \mathbb{R}$. Suppose you are given an MST $T$ for $G$. Now one of the weights in $G$ gets decreased, that is, one edge $e$ gets assigned weight $w' < w(e)$. Describe a $O(n)$ time algorithm that finds a new MST $T'$. (Hint: there are two cases: either $e$ is in $T$ already, or it is not.)

If $e$ is already in $T$, return $T$.

Otherwise, suppose $u$ and $v$ are the endpoints of $e$. Use DFS (or BFS) to find a path from $u$ to $v$ in $T$. Let $f$ be the length of the heaviest edge on that path. If $w' < w(f)$, return $T \setminus \{f\} \cup \{e\}$; else return $T$.

[Note: Because $T$ has $n-1$ edges, running DFS (or BFS) takes time $O(n+n-1) = O(n)$. Scanning over the path to find the heaviest edge takes time proportional to the length of the path, which is at most $n$.]

(d) Explain why your algorithm from the previous question is correct.

If $e \in T$, then changing the weight of $e$ makes the weight of $T$ decrease by $\Delta = w(e) - w'$. The weight of every other tree $T'$ decreases by at most that amount, so the weight of $T$ remains minimum.

If $e \notin T$, then adding $e$ to $T$ would create a cycle consisting of $e$ plus the path from $u$ to $v$ from $T$. The edge our algorithm removes (either $e$ or $f$, whichever is heavier) is the heaviest edge on that cycle; it is thus safe to discard by the cycle property.

It remains to argue that the rest of $T$ can remain unchanged. One way to do this is to look at the edges not in $T$ (other than $e$) and argue that they can all be discarded. To see why, note that each of them was the heaviest edge on some cycle in the original graph $G$ (for example, that is why Kruskal’s algorithm would not add them). Reducing the weight of $e$ will not change that fact, so each of those edges should still not be in the MST. The only way to keep the graph connected is to use all the remaining edges in $T$.

Problem 2. (Interval Sub-Covers) Given a family of open intervals $(a_i, b_i)$, $i = 1,\ldots,n$, a sub-cover is subset of the intervals that covers the same area of the real line as the union of all the intervals. For example, in the following picture, the gray intervals form a sub-cover consisting of 5 intervals:

You want to design an efficient algorithm that finds a sub-cover with as few intervals as possible. In the following, you may assume that (i) all the endpoints $(a_i, b_i)$ are distinct, and (ii) the original set of intervals covers a contiguous segment of the real line.
Your friend suggests the following greedy approach: at each stage, add the interval with the most length not covered by the intervals selected so far. Give an example showing that this approach will not find the smallest sub-cover.

The friend’s greedy algorithm is described ambiguously, so there were lots of correct answers depending on how one understood the question. In any interpretation, though, the algorithm will start by grabbing the longest interval from the whole collection. So a set of three intervals provides a counter-example to the algorithm’s optimality: (1, 5), (4, 8), (2, 7). The greedy algorithm will use all three even though only the first two are necessary.

(b) Give a polynomial-time algorithm to find a smallest sub-cover, given the $a_i$’s and $b_i$’s as input. Faster (correct) algorithms are worth slightly more points than slower ones.

Idea: Start with interval 1, and repeatedly grab the latest-finishing interval that overlaps with the current set.

```
Algorithm 2: GreedySubcover(a, b, n)

1 Sort the intervals in increasing order of left end-point so that $a_1 < a_2 < \cdots < a_n$;
2 $S \gets \{1\}$; /* set of intervals selected so far */
3 $t \gets b_1$; /* right-most endpoint of intervals in S */
4 $maxsofar \gets t$; /* maximum right endpoint seen in current phase */
5 $maxindex \gets 1$; /* index of interval with right endpoint maxsofar */
6 $b_{\text{max}} \gets$maximum among all endpoints $b_i$; /* requires $O(n)$ time to compute */
7 $j \gets 1$;
8 while $t < b_{\text{max}}$ do
9    /* Look for latest-finishing interval that overlaps with current set */
10       while $a_j \leq t$ and $j \leq n$ do
11          if $b_j > maxsofar$ then
12             $maxsofar \gets b_j$;
13             $maxindex \gets j$;
14             $j \gets j + 1$;
15             $t \gets maxsofar$;
16             Add $maxindex$ to $S$;
17    end
18 return $S$;
```

(c) What is the running time of your algorithm?

$O(n \log n)$ time: sorting takes $O(n \log n)$ time and the remainder of the algorithm takes $O(n)$ time (one pass through array with constant time per entry).

(d) Explain carefully why your algorithm is correct. You may find it helpful to read the rest of the question first.

There are at least two ways to approach this. The first is to use the “spreads” discussed in the rest of the question (see below). The second is to use “greedy stays ahead”:
let’s prove, by induction, the following claim, which implies that the greedy algorithm is optimal:

Claim: For every $k$, the leftmost $k$ intervals used by the greedy solution reach as far to the right as the leftmost $k$ intervals of any feasible solution.

Proof by induction: For $k = 1$, every feasible solution has to use interval 1, so they are all equivalent.

Induction step: Suppose $k > 1$ and the claim holds for $k - 1$. Let $t_k$ be the rightmost endpoint of the $k$-th interval in the greedy solution and let $t_k'$ be the rightmost endpoint of the $k$-th interval in some other solution. By the induction hypothesis, $t_k' \leq t_k$. By definition, the greedy solution chooses the $k + 1$-st interval that reaches as far to the right as possible while containing $t_k$. The $k + 1$-st interval of the other solution can reach no farther since $t_k' \leq t_k$. This completes the induction step.

(e) (Clarified) Given a set $T$ of intervals, a spread for $T$ is a set of points on the line such that each point is contained in some interval in $T$, but no interval in $T$ contains two points in the spread. Prove that if there exists a spread of size $k$ for $T$, then every subcover of $T$ must have size at least $k$.

Suppose there is a subcover that uses $k' < k$ intervals. The spread contains $k$ points, and the subcover must contain all of them, so some interval in the subcover must contain two points in the spread. This is a contradiction.

(f) Show that for any collection of intervals, the size of the largest spread is exactly equal to the size of the smallest sub-cover. [Hint: can you find an algorithm that produces a sub-cover and a spread of the same size?]

We can modify our greedy algorithm to produce a spread for the original set of intervals. Start from the set of all the values that the variable $t$ takes on during the algorithm: there are $k + 1$ of them if $k$ is the size of the subcover. Let’s remove the last one ($b_{\text{max}}$), and let’s replace $a_1$ with a value just a hair to the right (so that it is contained in interval 1).

These $k$ values (call them $t_1, \ldots, t_k$) are all contained in some interval. Moreover, no two of them can be contained in a single interval since we chose $t_j$ to be the rightmost end-point of all the intervals containing $t_{j-1}$.

Thus, we get a spread of size $k$. This implies that the spread is the largest possible, and the subcover is the smallest possible.

Point of confusion: Many students got confused between two notions: a spread for the whole set of intervals, and a spread for just the subcover. In order to prove a lower bound on the size of the minimum subcover, you need to argue that you have a spread for the whole set, not just the subcover.
Problem 3 (Recursion). Recall that the degree of a vertex in an undirected graph is the number of vertices adjacent to it; the maximum degree of a graph $G$ is the largest degree of any vertex in $G$.

Recall that a chain graph of size $k$ is a sequence of vertices $\{v_1, \ldots, v_k\}$ with edges $(v_i, v_{i+1})$ for $i = 1, \ldots, k - 1$. A cycle graph of size $k$ is a chain graph with an additional edge $(v_k, v_1)$.

(a) Show that a graph of maximum degree 2 or less is a union of disjoint chains and cycles.

Proceed by induction on the number of edges. If $m = 0$, the vertices are all isolated (chains of length 1). Now suppose we take a graph which is a union of chains and cycles and add an edge while keeping the degree at most 2. The endpoints of the edge must have had degree at most 1. They cannot have been part of a cycle (all vertices have degree 2) so they were endpoints of chains. Connecting them either creates a cycle or a longer chain.

(b) Give an algorithm that finds the size of the largest independent set in such a graph in time $O(n)$.

Use DFS to break the graph into connected components, noting the size of the components as we go (this takes time $O(m + n)$, but $m \leq n$ since the graph has degree at most 2). A cycle on $k$ vertices has a maximum independent set size $\lfloor k/2 \rfloor$. A chain on $k$ vertices has a maximum independent set size $\lceil k/2 \rceil$. We can sum the IS sizes from each of the connected components to get the maximum possible size of an IS in the graph as a whole.

(c) Fill in the missing pieces of the following recursive algorithm for finding the size of the largest independent set in a graph $G$:

<table>
<thead>
<tr>
<th>Algorithm 3: MIS$_3(G)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 if $G$ has maximum degree 2 then</td>
</tr>
<tr>
<td>2 run the algorithm from the previous question</td>
</tr>
<tr>
<td>3 else</td>
</tr>
<tr>
<td>4 pick a vertex $u$ of degree at least 3;</td>
</tr>
<tr>
<td>5 with$u \leftarrow 1 + MIS_3(G \setminus {u} \setminus N(u))$;</td>
</tr>
<tr>
<td>6 without $u \leftarrow MIS_3(G \setminus {u})$;</td>
</tr>
<tr>
<td>7 return $\max$,(with$u$ , without$u$)</td>
</tr>
</tbody>
</table>

(d) Explain how to represent an undirected graph in memory so that the running time of this algorithm, excluding the recursive calls, is $O(n)$.

This questions was removed at the request of the ASPCGS (American Society for the Prevention of Cruelty to Graduate Students). Interested students can ask me (the professor) about the solution.

(e) Give a recurrence for the worst-case running time of this algorithm. Justify your answer briefly.
\[ T(n) \leq T(n-1) + T(n-4) + O(n^2). \] Because \( u \) has degree at least 3, removing its neighbors shrinks the size of the graph by at least 4. The work done at a given level of the recursion is (at most) \( O(n^2) \) (roughly, the time required to prepare the two copies for recursive subcalls).

\( f \) Give a polynomial \( q(x) \) such that the algorithm above runs in time \( O(\lambda^n n^2) \), where \( \lambda \) is the largest real root of \( q(x) \).

\[ q(x) = x^4 - x^3 - 1 \]

\( g \) Will this algorithm run in time \( o(1.5^n) \)? Justify your answer briefly.

Yes. Note that \( 1.5^4 = 1.5 \times (1.5^3) = 1.5^4 + \frac{1}{2} \times 1.5^3 > 1.5^3 + 1 \). So \( q(1.5) > 0 \). Since \( x^4 \) increases more quickly than \( x^3 + 1 \) for \( x > 1 \), we know that there can be no real roots of \( q \) when \( x > 1.5 \). So the \( \lambda \) in the running time is at most 1.5.

**Problem 4. (Dynamic programming and coffee shops)** You decide to spend all Saturday studying for your algorithms midterm. You want to go to a coffee shop, since your roommates will be watching the football game. The problem is that there are two coffee shops you like: Saints’ and Devils’. At any given minute, you prefer to study in the quieter of the two coffee shops, but of course switching from one coffee shop to the other takes time: you lose ten minutes each time you switch.

You’ve figured out, for every minute \( i \) of the day, the number of exercises you can get done at Saints’, \( s_i \), and at Devils’, \( d_i \) (this depends on the typical noise level at that time). So the total amount of studying you’ll get done is

\[
\left( \sum_{\text{minutes spent working at Saints'}} s_i \right) + \left( \sum_{\text{minutes spent working at Devils'}} d_i \right).
\]

The time you spend switching from one coffee shop to the other is lost. You can start the day at either coffee shop.

More formally, you’re given \( s_1, \ldots, s_n \) and \( d_1, \ldots, d_n \) as input. A plan is a choice of Saints’, Devils’, or “switching” for each of the \( n \) minutes in the day. Switching takes a block of ten consecutive minutes. Your goal is to design a dynamic programming algorithm that finds an optimal studying plan.

\( a \) What are the subproblems for which your algorithm will compute solutions?

I will assume that \( s_i, d_i \geq 0 \) for all \( i \). This makes sense (how can I get a negative amount of work done?), and it makes the solution to the problem a bit simpler.

For each \( i = 1, \ldots, n \), we will compute

- \( S(i) \): the maximum amount of work I can get done during minutes \( \{1, \ldots, i\} \) assuming that I spend minute \( i \) at Saints’.
- \( D(i) \): the maximum amount of work I can get done during minutes \( \{1, \ldots, i\} \) assuming that I spend minute \( i \) at Devils’.

I would like to know \( \max(S(n), D(n)) \) – this is the overall maximum amount of work I can get done (note that there is no point to switching in the last ten minutes—I might as well stay where I am).
(b) Give a recursive formula for each subproblem in terms of other subproblems and the \( s_i \)'s and \( d_i \)'s. Justify the formulas briefly. Don't forget the base cases.

Given that I spend minute \( i \) at Saints', there are two possible situations: either I spent the previous minute at Saints', or I spent minute \( i - 11 \) at Devils' and was switching during minutes \( \{i - 10, \ldots, i - 1\} \). In the first situation, the work I get done is \( s_i \) plus the work I got done during the first \( i - 1 \) minutes, and I'm free to use any study plan that has me at Saint's during minute \( i - 1 \). So in that case, \( S(i) = s_i + S(i - 1) \). In the situation where I just switched, I am again free to use any study plan that leaves me at Devils’ at minute \( i - 11 \), so \( S(i) = s_i + D(i - 11) \). Thus,

\[
S(i) = \begin{cases} 
0 & \text{if } i \leq 0 \\
 s_i + \max(S(i - 1), D(i - 11)) & \text{if } 1 \leq i \leq n 
\end{cases}
\]

Similarly,

\[
D(i) = \begin{cases} 
0 & \text{if } i \leq 0 \\
 s_i + \max(D(i - 1), S(i - 11)) & \text{if } 1 \leq i \leq n 
\end{cases}
\]

(c) Give pseudocode for an algorithm for this problem. [For full credit, you should find an algorithm that runs in time \( O(n) \).] It is probably easiest to separate the algorithm that finds the value of the optimal plan from one that finds the actual plan.

We build the table of answers bottom-up. See pseudocode on next page (Algorithm 4). Students got most of the credit even if their code only computed the value of the optimal plan, not the actual optimal plan.

(d) State the running time and space complexity of your algorithm and justify your answer.

The algorithm uses time \( \Theta(n) \) since each pass through the arrays \( S \) and \( D \) uses constant time for each index. Space is also \( \Theta(n) \), to store \( S \) and \( D \).

— End of the exam —
Algorithm 4: StudyPlanValue($\vec{s}, \vec{d}$)

1. Allocate two arrays of real numbers $S$ and $D$ of length $n$ with indices in $\{-10, -9, ..., n\}$;
2. Initialize $S$ and $D$ to contain 0 in each cell;
   /* First we compute solutions to all subproblems. */
3. for $i = 1$ to $n$ do
   4. $S[i] \leftarrow s_i + \max(S[i - 1], D[i - 11])$;
   5. $D[i] \leftarrow d_i + \max(D[i - 1], S[i - 11])$;
   /* Now we extract optimal plan */
6. $L \leftarrow$ empty list;
7. $i \leftarrow n$;
8. if $D[n] > S[n]$ then
   9. $\text{whichCafe} \leftarrow \text{"D"}; \text{prepend} \text{"Devils' at time n" to } L;$
10. else
   11. $\text{whichCafe} \leftarrow \text{"S"}; \text{prepend} \text{"Saints' at time n" to } L;$
12. while $i > 1$ do
   13. if $\text{whichCafe} = \text{"D"}$ then
      14. if $D[i - 1] \geq S[i - 11]$ then
         15. prepend "Devils' at time $i - 1$" to $L$; $i \leftarrow i - 1$;
      16. else
         17. prepend "Devils' at time $i - 11$, switching from time $i - 10$ to $i - 1$" to $L$; $i \leftarrow i - 11$;
      18. else
         /* Similar code for case $\text{whichCafe} = \text{"S"}$. */
   19. ... 
20. return $L$;
