CSE 101 Final Exam


Time: 3 Hours

Some problems have multiple parts; do all parts. EXPLAIN ALL ANSWERS, with at least a few lines or sentences of precise English.

Order Notation

For each of the following answer “True” or “False” and give a brief explanation (1 or 2 lines or sentences.) Each is worth 4 points, 2 points for the correct answer and 2 points for the explanation or proof.

1. 1000n + 12,500 ∈ O(n log n) True, 1000n + 12,500 < 13,500n < 13500n log n. so

2. n^2 + (n - 1)^2 + (n - 2)^2 + ...1^2 ∈ O(n^2). False. n/2 of the terms are larger than n^2/4, so the sum is Ω(n^3), not upper bounded by O(n^2).

3. 2^{3n} ∈ O(2^n). False. 2^{3n}/2^n = 2^{2n}, which goes to infinity with large n. Thus, we can’t have 2^{3n} < C2^n for any constant C.

4. If n ≥ 16, an O(n log n) time algorithm is always at least four times faster than an O(n^2) time algorithm. False. Since O notation hides multiplicative constants, you cannot reach an exact numerical conclusion for specific values of n. Also, O is an upper bound, so that O(n^2) algorithm could also be O(n). (Either of these answers is enough).

5. If f and g are any positive, non-decreasing functions, then (f(n) + g(n))^2 ∈ Θ((f(n))^2 + (g(n))^2) (Prove or give counter-example.) True. (f(n) + g(n))^2 = f(n)^2 + 2f(n)g(n) + g(n)^2 ≥ f(n)^2 + g(n)^2 since f and g are non-negative. On the other hand, 2f(n)g(n) < 2max((f(n), g(n))^2 < 2f(n)^2 + 2g(n)^2 since the max squared will be one of those two terms. So all terms are upper bounded by O(f(n)^2 + g(n)^2) .

Divide and Conquer

The maximum weight sub-tree problem is as follows. You are given a balanced binary tree T of size n, where each node i ∈ T has a (not necessarily positive) weight w(i) for each node i ∈ T. (Every node in T has pointers to its left-child, right-child, and parent, and you are given a pointer to the root of the tree. A NIL field for the children means the node is a leaf, and for the parent, means the node is the root. You are given a pointer to the root r of T.) A rooted sub-tree of T is a connected sub-graph of T containing the root r. (So a sub-tree is not necessarily the entire sub-tree rooted at a node. However, it cannot contain the children of a node without containing the node.) You wish to find the maximum possible value of the sum of weights of nodes in a rooted sub-tree S of T, ∑i∈S w(i).
Here is a recursive algorithm that solves this problem, given a pointer to the root of $T$:

$\text{MaxWtSubtree}[r]$

1. IF $r = \text{NIL}$ return 0.
2. $A \leftarrow \max(O, \text{MaxWtSubtree}[r\text{.leftchild}])$
3. $B \leftarrow \max(O, \text{MaxWtSubtree}[r\text{.rightchild}])$

a. Give a recurrence and a time analysis for this algorithm in the case when $T$ is a complete binary tree of height $h$ and size $n = 2^h - 1$ (10 pts.)

We call ourselves recursively on the left and right subtree, and then have constant work. If both sub-trees are of size at most $n/2$, as in the balanced case, $T(n) \leq 2T(n/2) + O(1)$. Since $2 > 2^0$, this is in the bottom-heavy case of the Master Theorem, so the total time is $O(n^{\log_2 2}) = O(n)$.

b. Prove that the same worst-case bound holds if $T$ is any tree of size $n$. (10 pts.)

More generally, if the left subtree has size $L$ and the right sub-tree has size $R$, $n = L + R + 1$, and $T(n) \leq T(L) + T(R) + O(1)$. We can prove by strong induction that $T(n)$ is linear-time. Let $c$ be greater than the hidden constant in the $O(1)$ above, and greater than the time the algorithm takes on inputs of size 1. By definition, if the tree has size $n = 1$, the algorithm takes time less than $Cn = C$. Assume the algorithm takes time at most $Cn'$ for $1 \leq n' \leq n$. Then on an input of size $n$, it takes time $T(L) + T(R) + c'$ for $c' < c$ the hidden constant. By the inductive hypothesis, $T(L) \leq cL$ and $T(R) \leq cR$, to the total time is at most $T(n) \leq T(L) + T(R) + c' \leq cL + cR + c \leq cn$.

Alternatively, observe that we call the algorithm recursively throughout at most once per node of the tree, and each time, the non-recursive part is constant time.

Monotone matchings All the remaining questions concern variations of the following problem.

Let $G$ be a bipartite graph, with $L = \{u_1, \ldots, u_l\}$ the set of nodes on the left, $R = \{v_1, \ldots, v_r\}$ the set of nodes on the right, $E$ the set of edges, each with one endpoint in $L$ and the other in $R$, and $m = |E|$ the number of edges.

A matching in $G$ is a set of edges $M \subseteq E$ so that no two edges in $M$ share an endpoint (neither the one in $L$ nor the one in $R$). A matching $M$ is monotone if for every two edges $(u_{i_1}, v_{j_1})$ and $(u_{i_2}, v_{j_2})$ in $M$, if $i_1 < i_2$ then $j_1$ < $j_2$. 
then $j_1 < j_2$. That is, one could draw all the edges in the matching without crossing, if the nodes are put in order on the two sides.

The problem is, given a bipartite graph $G$, find the largest monotone matching in $G$.

Assume $l \leq r$. Then a monotone matching $M$ is perfect if it has size $l$, i.e., $|M| = l$.

**Greedy Algorithms and data structures Part 1 : 10 points** Below is a greedy strategy for the largest monotone matching problem. Give a counter-example where it fails to produce the optimal solution. (Hint: Since below you will show that the algorithm works when the maximum monotone matching is perfect, your example shouldn’t have a perfect monotone matching.)

Candidate Strategy A : For each $i = 1$ to $l$, if $u_i$ has at least one undeleted neighbor $v_j$, match it to the unmatched neighbor with smallest value of $j$. Then delete $u_i$ and $v_1, \ldots, v_j$, and repeat.

Say that $G$ has three nodes $u_1, u_2, u_3$ on the right and three nodes $v_1, v_2, v_3$ on the left. If $u_1$ is connected to $v_3$ only, $u_2$ to $v_1$ and $u_3$ to $v_2$, then the best monotone matching is to use the last two edges. But the greedy algorithm above picks the first edge, leaving it no additional edges to choose from.

**Part 2: 5 pts** Illustrate the above strategy on the following graph with a perfect matching: $L = \{u_1, u_2, u_3\}$, $R = \{v_1, v_2, v_3, v_4, v_5\}$, and $E = \{(u_1, v_2), (u_1, v_3), (u_1, v_4), (u_2, v_1), (u_2, v_2), (u_2, v_3), (u_2, v_5), (u_3, v_1), (u_3, v_5)\}$

Skipped

**Part 3: 10 points** Prove that, if $G$ has a perfect matching, then Candidate Strategy A finds one.

(Hint: Use one of the following two methods. Let strategy A match $u_i$ with $v_j$ (unless it can’t be matched). Let $OPT$ be a perfect matching that matches each $u_i$ with $v_k$. (Note: all left nodes will be matched by $OPT$, since it is perfect.)

Transformation method: Prove by induction on $T$ that there is a left-perfect matching $OPT_T$ that matches each $u_i$ with $v_j$, for $1 \leq i \leq T$. We show by induction, that, if there is a perfect monotone matching, then there is one that matches the first $i$ nodes on the left as does the greedy algorithm. In the base case, $i = 0$, there is nothing to prove.

Assume $Opt_T$ is a perfect monotone matching that matches the first $i$ nodes as does the greedy algorithm. Say that $Opt_T$ matches $u_i$ to some $v_j$ (as does the greedy algorithm) and matches $u_{i+1}$ to some $v_k$. We must have $k > j$ for $OPT_T$ to be monotone. So at the start of the $i + 1$st iteration of the greedy algorithm, $v_k$ has not been deleted.
The greedy algorithm will then match $i+1$ to the first neighbor that has not been deleted, if any. Since $v_k$ is some neighbor that hasn’t been deleted, it will either match $i+1$ to $v_k$ or some $v_{k'}$ with $k' < k$. In the first case, we can let $OPT_{i+1} = OPT_i$. In the second, we can replace the edge from $u_{i+1}$ to $v_k$ in $OPT_i$ with the edge from $u_{i+1}$ to $v_{k'}$ to construct $OPT_{i+1}$. In either case, $OPT_{i+1}$ is a perfect monotone matching, because the match for $u_{i+1}$ is greater than $j$ and hence after any node used for nodes before $u_{i+1}$ and less than $k$ and hence before any node used for nodes after $u_{i+1}$.

By induction, we have shown the claim for any $i$. In particular, for $i = n$, there is a perfect monotone matching that matches every node the same as the greedy algorithm does, so the greedy algorithm is a perfect monotone matching.

**Part 4: 10 points** Describe an efficient algorithm that carries out the strategy. Your description should specify which data structures you use, and any pre-processing steps. Assume the graph is given in adjacency list format. Give a time analysis, in terms of $l, r$ and $m$.

Assume the graph is in adjacency list format. We just keep track of the index $k$ of the last node $v_k$ that has been matched. For each $i$, we run through the list of neighbors, and find the smallest $k' > k$ so that $v_{k'} \in N(u_i)$. This takes time proportional to the degree of $u_i$. If no such $k'$ exists, we leave $u_i$ unmatched. Otherwise, we match it to $v_{k'}$ and set $k$ to $k'$.

Since the time for each node on the left is $O(1 + deg(u_i))$, the total time will be $O(|L| + |E|)$.

**Back-tracking and Dynamic Programming** The following recursive algorithm for the maximal monotone matching problem finds the maximum matching whether or not it is perfect. It branches on whether a node on the left is matched or unmatched. By the analysis of greedy algorithm A above, we can see that when a node is matched, it should always be matched to its smallest neighbor. The backtracking algorithm just returns the size of the maximum monotone matching, not the actual matching.

BTMMM($G = (L = \{u_1, \ldots, u_l\}, R = \{v_1, \ldots, v_r\}, E)$: bipartite graph)

1. IF $|L| = 0$ return 0.
2. Unmatched ← BTMMM($G - \{u_1\}$).
3. IF $|N(u_1)| = 0$ return Unmatched
4. Let $J$ be the first neighbor of $u_1$, i.e., the smallest value so that $v_J \in N(u_1)$
5. Matched ← $1 + BTMMM(G - \{u_1, v_1, \ldots, v_J\})$
6. Return $\texttt{Max}(\text{Matched}, \text{Unmatched})$
Part 1: 5 points Illustrate the above algorithm on your counter-example graph for the greedy strategy, (as a tree of recursive calls and answers.}

Skipped

Part 2: 5 points Give an upper bound on the number of recursive calls the above algorithm makes, in the worst-case. (Be sure to expain your answer.)

There are two ways to give this bound. We make two recursive calls at each step. In terms of the number of nodes \( n = l + r \), one call is of size \( n - 1 \), and the other of size at most \( n - 2 \). This gives us an upper bound of \( O(Fib_n) \), the \( n \)'th Fibonacci number.

On the other hand, both recursive calls reduce \( |L| \) by 1. Thus, the depth of the binary tree is at most \( |L| \) which gives an upper bound of \( O(2^{|L|}) \).

Both bounds are correct, and neither is always better than the other. We could combine them by saying the upper bound is the minimum of the two.

Part 3: 10 points Give a dynamic programming version of the recurrence.

At any point in the recursion, the remaining nodes on the left are \( u_I...u_l \) and the remaining nodes on the right are of the form \( v_K...v_r \). So we create a matrix \( MM[I,K] \) to store the answers for each \( 1 \leq I \leq l + 1 \) and each \( 1 \leq K \leq r + 1 \), \( I \) is increasing as we make calls, so we need to fill in this matrix in decreasing order of \( I \). The base cases are when \( I = l + 1 \), i.e., the left side is empty.

\[ DPMMM(G = (L = \{u_1,...u_l\}, R = \{v_1,...v_r\}, E): \text{bipartite graph}) \]

1. Initialize \( MM[1...l + 1,1...r + 1] \).
2. FOR \( K = 1 \) to \( r + 1 \) do: \( MM[l + 1,K] \leftarrow 0. \)
3. FOR \( I = l \) downto 1 do: FOR \( K = 1 \) to \( r + 1 \) do:
   4. \( Unmatched \leftarrow MM[I + 1,K] \).
   5. IF \( u_I \) has no neighbor \( u_H \) with \( H \geq K \) THEN \( MM[I,K] \leftarrow Unmatched \)
   6. ELSE let \( J \) be the smallest such \( H \)
   7. let \( Matched \leftarrow 1 + MM[I,J] \).
   8. \( MM[I,K] \leftarrow \max(Matched, Unmatched) \)
9. Return \( MM[1,1] \).

Part 4: 5 points Give a time analysis of this dynamic programming algorithm.

In the most naïve version of the algorithm above, for each node \( u_I \), we search through all of its neighbors \( r + 1 \) times. So the cost is \( \sum_I O((\text{deg}(u_I)(r + 1))) = O(r|E|) \).
But if we sort the neighbors of I before doing the loop for all J, and just keep a counter for where we left off in the search for a neighbor \( H > K \), we can update this counter in constant time when we increment \( K \). (It is either the same, or the next on the sorted list). Then the total time is \( \sum_i (O(\deg(v_i) \log \deg(v_i) + r)) \leq \sum_i O(\deg(v_i) \log r + r)) = O(|E| \log r + rl). \)

**Part 5: 5 points** Show the array or matrix that your dynamic programming algorithm produces on the example graph from Part 2 of the greedy algorithm.

Skipped.