Midterm 1 Solutions

1 True/False

1. The Mayan base 20 system produces representations of size that is asymptotically different from that of decimal (base 10) system.

   Solution: False $\log_{20}(n)$ = number of digits using base 20 and $\log_{10}(n)$ = number of digits using base 10. $\log_{20}(n) \in \Theta(\log_{10}(n))$

2. If $f(n) = O(g(n))$, then there is some value of n where $f(n) \geq g(n)$.

   Solution: False $g(n)$ can be greater than $f(n)$ for all n, for example $g(n) = f(n) + 1$

3. If $f(n) = O(g(n))$ and $\log g(n) \geq 1$ for all n, then $\log f(n) = O(\log g(n))$.

   Solution: True $f(n) \leq cg(n) \implies \log f(n) \leq \log g(n) + \log c \leq 2 \log c \log g(n)$

4. If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$.

   Solution: False Consider $f(n) = n$ and $g(n) = n/2$.

5. $x^{(p-1)(q-1)} = 1 \pmod{pq}$ for all $x \neq 0 \pmod{pq}$ if p and q are distinct primes.

   Solution: False This implies that x has an inverse mod pq and that $\gcd(x, pq) = 1$, which would not be true if $x = p$

6. $x^{(p-1)(q-1)+1} = x \pmod{pq}$ if p and q are distinct prime.

   Solution: True Same as the correctness of RSA when $ed = (p-1)(q-1) + 1$

7. If there is an $x \geq 2$ where $x^{(N-1)} = 1 \mod N$ then N is prime or Carmichael.

   Solution: False $8^8 = 1 \mod 9$. In general, for any *odd* number N, we have $(N-1)^{N-1} = (-1)^{N-1} \mod N = 1 \mod N$ (since $N - 1$ is even).

8. Recall, that a valid coloring for a graph is an assignment of colors for a graph where each edge's endpoints do not have the same color. A k coloring is a valid coloring using at most k colors.

   If a graph has a degree 4 node, it requires 5 colors to have a valid coloring.

   Solution: False A star can be two colored.

9. A valid two coloring of a two colorable graph can be produced by coloring using the low order bit (odd/even) of the level in a BFS tree.

   Solution: True By alternating colors with depth we know that this coloring is valid if only considering the BFS tree edges. If there is still an edge where both endpoints point to nodes of the same color, then the graph has an odd cycle and is not two colorable.

10. Briefly justify or give a counterexample: A valid coloring for a two colorable undirected graph can be given by using the last bit of the pre numbers.

    Solution: True For a tree edge (u, v) pre(u) and pre(v) differ by an odd number since when one has returned from all previous explores when exploring v. For a back edge, if the pre(v) differs by an even number, we have found an odd cycle.

11. Every DAG has exactly one topological ordering

    Solution: False Consider the graph $\{(A, B), (A, C)\}$, which has two topological orderings ABC and ACB.

12. If a DFS in an undirected graph G contains exactly one back-edge, then G can be made acyclic by removing one edge.

    Solution: True Component with back edge remains connected with $k - 1$ edges where k is number of nodes in that component, so it is a tree and is acyclic.

13. If G is a directed graph and can be made acyclic by removing exactly one edge, then any DFS in G will only contain exactly one back-edge

    Solution: False Two different cycles can share one edge, thus by removing that one edge the entire graph is made acyclic.

14. Given a valid pre order numbering of a directed graph, there is only one corresponding post order numbering.

    Solution: True A valid DFS is well-defined by how it breaks ties among a vertex's children, which leads to exactly one pre and post-number ordering.

15. A primitive 8th root of unity is $\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}i$.

    Solution: True A primitive 8^{th} root of unity is $e^{i\frac{2\pi}{8}} = \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}i$

2 How Many?

1. What is the number of modular multiplications to compute $x^{49} \mod N$ using repeated squaring?

   Solution: $\boxed{7}$ Five squarings (to get $x^2, x^4, x^8, x^{16}, x^{32}$), then two multiplications ($x^{49} = x^{32} \times x^{16} \times x^1$).

2. What is the inverse of $19 \pmod{21}$?

   Solution: $\boxed{10 \pmod{21}}$ $19 \times 10 \equiv 190 \equiv (9 \times 21) + 1 \equiv 1 \pmod{21}$.

3. For integers a, x, b, y where $ax + by = 1$, what is the inverse of x modulo y?

   Solution: $\boxed{\text{a}}$ $ax \equiv 1 - by \equiv 1 \pmod{y} \implies a \equiv x^{-1} \pmod{y}$

4. What is the probability that $x + y + z = 1 \pmod{p}$, when x, y and z are chosen uniformly at random from $\{0, \ldots, p-1\}$?

   Solution: $\boxed{\frac{1}{p}}$ This is similar to universal hashing. Assume that x and y have already been picked. There is exactly 1 value of z that can satisfy $x + y + z \equiv 1 \pmod{p}$. Thus, regardless of what x and y are, there is a $\frac{1}{p}$ chance that the chosen z will satisfy $x + y + z \equiv 1 \pmod{p}$.

5. A *separator* in a graph is a set of nodes whose removal produces a graph with more than one connected component. Give the best possible upper bound on the size of a separator in a graph that has a breadth first search tree with depth $D \geq 3$ in terms of D and n, the number of nodes in the graph.

   Solution: $\boxed{\frac{n-2}{D-2}}$ Removing any of the middle $D-2$ levels of the BFS tree leaves at least two components. The maximum number of nodes in the middle $D-2$ levels is $n-2$ (exclude the BFS starting node and the leaves on the outermost level; to get an upper bound on the size of the separator, we consider the case where there is only be one leaf on the outmost level). We get $\frac{n-2}{D-2}$ by averaging over all middle $D-2$ levels.

6. What is the maximum number of edges in a 10 node undirected two colorable graph?

   Solution: $\boxed{25}$ From homework, we know that a two colorable graph is bipartite. Let x be the number of nodes in the smaller of a 10-node bipartite graph's two partitions. The maximum number of edges in such a graph is obtained if every node in one partition has an edge to every node in the other partition (and vice versa). Thus, the maximum number of nodes a 10-node bipartite graph could have is $x(10-x)$, which has a maximum value of 25 at $x = 5$.

3 Sun Tzu's and Euler's Napkins

Short answer!

1. (a) What is $56a + 22b \pmod{11}$?
   Solution: $22 = 0 \pmod{11}$ and $56 = 1 \pmod{11}$, thus $56a + 22b = a \pmod{11}$.

   (b) What is $56a + 22b \pmod{7}$?
   Solution: $22 = 1 \pmod{7}$ and $56 = 0 \pmod{7}$, thus $56a + 22b = b \pmod{7}$.

   (c) Find a number $x \pmod{77}$ where $x = 1 \pmod{11}$ and $x = 2 \pmod{7}$.
   Solution: Plug in $a = 1$, and $b = 2$ into the formula from parts (a) and (b)! You get $56(1) + 22(2) = 100 = 23 \pmod{77}$.

2. Let p be prime.

   (a) How many numbers are relatively prime to p^2 in $\{0, \ldots, p^2 - 1\}$.
   Solution: $p^2 - p = p(p-1)$ since exactly the p multiples of p less then p^2 are not relatively prime wih p^2.

   (b) What is the number of numbers that have inverses $\pmod{p^2}$?
   Solution: $p(p-1)$ by the numbers with inverses $\pmod{p^2}$ are those that are relatively prime to p^2 according to Euclid's Theorem. Part (a) gives that number.

   (c) For what value of l in terms of p would it always be true that $x^l = 1 \pmod{p^2}$, if x is relatively prime with p. Briefly justify (must do so).
   Solution: $p(p-1)$ by proof analogy to Fermat's Theorem; Let S is the all the numbers in $\{1, \ldots, p^2 - 1$ that are relatively prime to p^2, and $T = \{ax | x \in S\}$. Since x has an inverse $\pmod{p^2}$, $S = T$, and the product of their elements is the same. This allows us to conclude that $x^{|S|} = 1 \pmod{p^2}$. From part (b), $|S| = p(p-1)$.

   This is a special case of Euler's Totient Theorem as is the identity $x^{(p-1)(q-1)} = 1 \pmod{p)q}$ for *distinct* primes. That inequlity does not hold when $p = q$, e.g., $2^4 \neq 1 \pmod{9}$.

   This is a difficult problem with a short answer.

4 Short Answer

No need for proofs of correctness or runtime! (Briefly justify if unsure.)

1. Asymptotically solve $T(n) = 8T(n/4) + O(n^{1.3})$.

   Solution: $\boxed{O(n^{1.5}).}$ Putting this recurrence in the form $T(n) = aT(n/b) + O(n^d)$, we see by the Master theorem that since $\log_b a = \log_4 8 = \frac{\log_2 8}{\log_2 4} = 1.5 > d = 1.3$, the answer is $O(n^{1.5})$.

   Most people got this right. But some students lost points for incorrectly assuming that $\log_4 8$ *was either* 1.3 *or* 2*. Students also lost some points for not writing the answer in the* O*-notation (though we were lenient with people who used* Θ *instead of* O*).*

2. Asymptotically solve $T(n) = \sqrt{n}T(\sqrt{n}) + O(n)$.

   Solution: $\boxed{O(n \log \log n).}$ The simplest solution is to consider the function $S(n) = T(n)/n$, which satisfies the recurrence $S(n) = S(\sqrt{n}) + O(1)$. It is clear that this has $O(\log \log n)$ levels (since the size of the problem is $n^{1/2^k}$ at the kth level), so that we get $S(n) = O(\log \log n)$, and hence $T(n) = O(n \log \log n)$. Another solution is to directly observe (via the recurrence tree) that the work done per level in the original recurrence is $O(n)$.

   We gave partial credit to students who had the right estimate for the work done per level, even if they erred in calculating the number of levels. Some students lost points for not using the O*-notation, as in the last problem.*

3. Given a tree and a depth first search based pre/post order, give a fast method for determining whether u is an ancestor of v in the tree (where the root is the node where the search is started.)

   Solution: u is an ancestor of v in the DFS tree **if and only if** $\boxed{\text{pre(u) < pre(v) < post(v) < post(u).}}$ Answers of the form $\text{pre}(u) < \text{pre}(v) < \text{post}(u)$ also received full credit (because they imply the last condition through the non-intersection property of pre-post numbers).

   Some students lost points for not counting parents as ancestors. Some people also lost points for giving a condition for when u *is a* descendant *of* v*. Some students also lost points for saying that checking the last inequality is optional (since if you do not do that, it can happen that the* v *is not reacahble from* u *in the tree). We gave (almost) full credit if it was clear that the student had the right justification but made a typo in writing down the final answer.*

4. The Fourier Transform of $(1, 0, 1, -1)$ is $(1, i, 3, -i)$. What is the Fourier Transform of $(1, 0, 0, 0, 1, 0, -1, 0)$?

   Solution: $\boxed{(1, i, 3, -i, 1, i, 3, -i).}$ The simplest way to see this is to use the definition of the FFT algorithm. Let ω be a primitive 8th root of unity. Then, in order to compute the FFT of the vector $(1, 0, 0, 0, 1, 0, -1, 0)$, we first compute the FFT of the even co-ordinates: $(s_0, s_1, s_2, s_3) = FFT[(1, 0, 1, -1), \omega^2]$, which we are already told is $(1, i, 3, -i)$. We then consider the FFT (s'_0, s'_1, s'_2, s'_3) of the odd co-ordinates. Since these co-ordinates are all 0, the FFT (s'_0, s'_1, s'_2, s'_3) is $(0, 0, 0, 0)$ as well. We now combine the results to get that the FFT of the original vector is simply $(1, i, 3, -i, 1, i, 3, -i)$ (that is, the (already given) FFT of the even co-ordinates repeated twice).

   Many students did not use the extra information in the question and proceeded to compute the FFT directly. This approach is error-prone, and we were able to give partial credit only when the number of calculation errors was not too high (typically, when it was less than 2). However, correct answers received full credit irrespective of the method used.

5. Briefly describe a method that finds the coefficients of a polynomial whose roots are $r_1, \ldots, r_n$. (You may produce one where the leading coefficient is 1.) It should run in $O(n \log^2 n)$ time and can use FFT as a black box.

   Solution: We are trying to find the coefficients of the polynomial $(x - r_1)(x - r_2) \cdots (x - r_n)$. Split the roots into two (approximately) equal halves: $r_1, r_2, \ldots, r_{\lfloor n/2 \rfloor}$ and $r_{\lfloor n/2 \rfloor + 1}, \ldots, r_n$. Recursively find the polynomial whose roots are $r_1, \ldots, r_{\lfloor n/2 \rfloor}$, and the polynomial whose roots are $r_{\lfloor n/2 \rfloor + 1}, \ldots, r_n$. Multiply these two polynomials together using FFT, which takes $O(n \log n)$. When the base case is reached with only 1 root r, return $(x - r)$. The recurrence for this algorithm is $T(n) = 2T(n/2) + O(n \log n)$, which gives $O(n \log^2 n)$.

6. Given an undirected graph with nonnegative edge costs, along with nonintersecting subsets S and T of vertices, give a short description of an efficient method to determine the distance between the closest pair of nodes between S and T; that is, $u \in S, v \in T$ such that $dist(u, v)$ is minimal in terms of shortest path costs.

   Solution: Add node s with zero weight edges to every node in S and a node t with zero weight arcs from every node in T and use Dijkstra's to compute the s-t distance. Running time is the same as that of Dijkstra's.

   Some other methods of doing the same thing:

   1. Collapse the vertices in S to a single metanode and do the same for T, and run Dijkstra's.
   2. Collapse the vertices in S to a single metanode, run Dijkstra's, and stop as soon as you call `delete-min` on a vertex in T.
   3. Run Dijkstra's, but initialize the distances of all nodes in S to 0.
   4. Set the weight of any edge between vertices in S to 0, and run Dijkstra's from any vertex in S.

   A lot of students incorrectly assumed that $S \cup T = V$ (the set of all vertices), and so the algorithm would be to find the minimal edge (u, v) such that $u \in S, v \in T$. It was only stated that S and T were non-intersecting subsets, not that every vertex had to be in S or T.

5 Divide and Conquer

1. Given an (unsorted) array of numbers in an array, a local maximum is at index i if $A[i]$ is larger than $A[i-1]$ and $A[i+1]$. (The last and first elements only need be larger than the previous and next elements respectively.) Give an efficient algorithm to find a local maximum in an array of length n where no entries are repeated.

   Solution: Both parts use divide and conquer to narrow down the search space of a local maximum. In both the 1-d and 2-d cases, we have the following

   > **Observation** If we start at ANY element, keep moving to ANY larger adjacent element if there exists one, and stop when all adjacent elements are smaller, it is obvious that our path must end at a local maximum. Furthermore, since elements are distinct, the numbers on the path strictly increase, so we never visit the same element more than once, which means the path must end at some point.

   Our algorithm first pick the center element $A[\lfloor \frac{n}{2} \rfloor]$, and check if it is a local maximum. If it is, output it, and stop. If it is not, there must exists a larger neighbor. If $A[\lfloor \frac{n}{2} \rfloor - 1] > A[\lfloor \frac{n}{2} \rfloor]$, recursively search for a local maximum in $A[0, \ldots, \lfloor \frac{n}{2} \rfloor - 1]$. Otherwise, search for a local maximum in $A[\lfloor \frac{n}{2} \rfloor + 1, n-1]$. We only look at one half, even when both neighbors are larger. At the base case, we have a list of just one element, which will be a local maximum.

   The algorithm is correct in all cases. When $A[\lfloor \frac{n}{2} \rfloor]$ is a local maximum, the algorithm outputs it. If $A[\lfloor \frac{n}{2} \rfloor]$ is not, suppose we are in the case that the algorithm chooses to search $A[0, \ldots, \lfloor \frac{n}{2} \rfloor - 1]$ (same argument for the other case). By our ealier observation, there must exists a local maximum in that half (consider the path starting with $A[\lfloor \frac{n}{2} \rfloor] \to A[\lfloor \frac{n}{2} \rfloor - 1]$, the path must end in that half, thus a local maximum exists there). Furthermore, exactly one end of this half list is not an end element of the original list, that is $A[\lfloor \frac{n}{2} \rfloor - 1]$, but we know $A[\lfloor \frac{n}{2} \rfloor - 1] > A[\lfloor \frac{n}{2} \rfloor]$, so if $A[\lfloor \frac{n}{2} \rfloor - 1]$ is a local maximum in the half list, it is a local maximum in the original list.

   The running time is $O(\log n)$, since we at each level we solve **one** problem of size $\frac{n}{2}$, plus constant work (comparing the middle elements to neighbors). $T(n) = T(\frac{n}{2}) + O(1) \qquad T(n) = O(\log n)$.

2. Given an $n \times n$ array of numbers A, a local maximum is defined as an (i, j) where $A[i, j]$ is larger than any "adjacent" entry; any (i', j') where either i' or j' differs by 1. For example, for entry $(2, 2)$ the adjacent entries are $(1, 2), (3, 2), (2, 1)$ and $(2, 3)$. Also, entry $(0, 0)$ has only two adjacent entries: $(1, 0)$ and $(0, 1)$.

   Give an algorithm that given an $n \times n$ array, A, of distinct numbers finds a local maximum in $O(n)$ time. (Note that this is "sublinear" in the size of the array. An $O(n^2)$ solution will recieve no credit.)

   Solution: We will solve a slightly more general problem:

   > Given $n \times n$ array A, and a threshold element $A[i^*, j^*]$ in the array, find a local maximum with the additional constraint that the local maximum is $\geq A[i^*, j^*]$.

   It is easy to see we can solve the original problem by solving this problem with an arbitrary $A[i^*, j^*]$. Notice there must exists such a local maximum by considering an increasing path starting at $A[i^*, j^*]$

   For the base case, when $n \leq 3$, we just search all elements, and output the global maximum.
   Otherwise, let $r = \lceil \frac{n}{2} \rceil$, and let the four quadrants, each of size $\lceil \frac{n}{2} \rceil \times \lceil \frac{n}{2} \rceil$, contain the elements $A[1 \ldots r, 1 \ldots r], A[1 \ldots r, r \ldots n], A[r \ldots n, 1 \ldots r]$, and $A[r \ldots n, r \ldots n]$ respectively. Notice elements on row r and column r are contained in more than one quadrants.
   First the algorithm looks at all elements on the borders of the four quadrants (i.e. rows $1, r, n$, and columns $1, r, n$), as well as all the neighbors of the border elements. There are $O(n)$ such elements, and we find the maximum element among them, call it $A[\hat{i}, \hat{j}]$. We have three cases

   (i) $A[\hat{i}, \hat{j}] < A[i^*, j^*]$, recursively find a local maximum in the quadrant containing $A[i^*, j^*]$, with the old threshold $A[i^*, j^*]$.

   (ii) $A[\hat{i}, \hat{j}] \geq A[i^*, j^*]$, and $A[\hat{i}, \hat{j}]$ is on the border of a quadrant. Since $A[\hat{i}, \hat{j}]$ is the maximum among the points we considered, which include all its neighbors, it is a local maximum, and also $\geq A[i^*, j^*]$, we just return it, and stop.

   (iii) $A[\hat{i}, \hat{j}] \geq A[i^*, j^*]$, and $A[\hat{i}, \hat{j}]$ is in the interior of a quadrant. Recursively find a local maximum in that quadrant, with threshold element $A[\hat{i}, \hat{j}]$.

   To see the algorithm is correct, we need to check the three cases

   (i) Use the observation at the beginning of 5.1, start the path at $A[i^*, j^*]$, which is an interior point of the quadrant, since the path strictly increases, we will never visit any element on the border since they are all $\leq A[\hat{i}, \hat{j}] < A[i^*, j^*]$, so our increasing path never leaves the quadrant. This guarantees that a local maximum $\geq$ the threshold exists in this quadrant, and it must be an interior element. So the result returned by the recursion will be a valid answer for our original problem, because all neighbors of that local maximum in the original problem are present in the subproblem.

   (ii) This case is obvious.

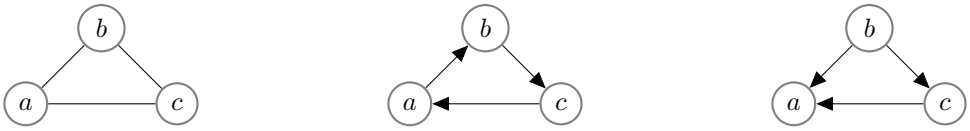
   (iii) This case is similar to case (i). We know a local maximum $\geq A[\hat{i}, \hat{j}]$ exists, and it must be an interior element of this quadrant.

   Running time: At each level we do $O(n)$ work, and look at **one** subproblem of size $\frac{n}{2} \times \frac{n}{2}$, so we have the recurrence $T(n) = T(\frac{n}{2}) + O(n)$, which gives $O(n)$ running time.

6 Directing those without direction

Suppose we have an undirected *connected* graph, and we would like to find a strongly connected orientiation of its edges: that is, an assignment of directions to its edges such that the entire graph becomes strongly connected (i.e. every node is reachable from every other node via a directed path).

For the undirected connected graph below the next figure is a strongly connected orientation, and the last is an orientation that is not strongly connected.



1. Give an example of a connected undirected graph where this cannot be done.

   Solution: 3pts: Any graph with a bridge. The simplest example is the 2-node graph $A - B$.

2. A *bridge* in a connected undirected graph is an edge whose removal leaves a graph with two connected components. Give a linear time algorithm to find a strongly connected orientation in any connected undirected graph without a bridge. Justify its correctness.

   Solution: Algorithm (4pts): DFS. Label the tree edges in the explored direction, label the back edges toward the ancestors.

   Correctness (3pts): The main way to solve this was by contradiction, but there were some interesting inductive solutions. Here, we only present the contradiction way.

   By Contradiction: If this fails, consider the sink strongly connected component. Consider the first edge into the component that was explored, explore would not return until the entire component is explored and any other edge incident to this component would then be considered. If the other end was previously explored, dfs would have already explored this edge, thus, dfs would have explored this edge, and it would be directed outward, contradicting the notion that this is the sink component. Thus, there are no other edges incident to this component. Thus, the first edge is a bridge, which contradcts the claim that we can not have a bridge, so this can not fail

   Running time (1pt): Mention DFS or say "linear time"

   Some comments: Some students said use the highest post number vertex after doing one DFS. But, since this graph is undirected and connected, the vertex which you start the search will have highest post number (so you're essentially doing DFS twice for no reason).

   Also, please do note the difference between what algorithms we can apply to directed graphs and undirected graphs. It doesn't really make much sense to say strongly connected components in undirected graphs (since the graph is connected, we will only get back one connected component). However, this notion does make sense in this problem since we are assigning directions to undirected edges.

   One more misconception: A strongly connected component in a directed graph does not imply that there is a cycle going through every single node. However, it is true that every node is part of a cycle, just not necessarily the same one.

3. Give a linear time algorithm to determine whether there is a bridge in a graph.

   Solution: (4pts): Do the orientation above, and check if it is strongly connected (by making SCCs in directed graph and check if there is more than one component)

   The sentence above was enough to get full credit, but it does miss the fact that you still need to prove that an strongly connected orientation of a graph with a bridge is impossible. But, this is easy to show, since the bridge separates the graph into two components S, T, and if we orient the bridge one way, we have no way to get from S to T, or vice versa if we orient it the other way. Solutions that did not prove this direction were not penalized.

   Please do note that this is an undirected graph. Thus, running the SCC algorithm from class does not work, since that is only for *directed* graphs. We must change it to a directed graph first from our algorithm in part (b), then apply the SCC algorithm.