# CS 170 Spring 2008 - Solutions to Midterm #1

## Problem 1

1. $2^n = \Omega(n!)$.

   **False**: $\lim\limits_{n\to\infty}(2^n/n!) = \lim\limits_{n\to\infty} \frac{2}{1}\cdot\frac{2}{2}\cdot\ldots\cdot\frac{2}{n} \le 2\cdot \lim\limits_{n\to\infty}\frac{2}{n} = 0$.

2. $3^{\log_2 \sqrt{n}} = \Theta(\sqrt{n})$.

   **False**: $3^{\log_2\sqrt{n}} = 3^{(\frac{1}{2}\log_2 n)} = n^{(\frac{1}{2}\log_2 3)} = (\sqrt{n})^{\log_2 3}$.

3. If $f(n) = O(g(n))$, then $g(n) = \Omega(f(n))$.

   **True**: By definition of $\Omega$ (this is in fact how your textbook defines $\Omega$).

4. $\sum_{j=1}^{n} j = O(n\log n)$.

   **False**: $\sum_{j=1}^{n} j = \frac{1}{2}n(n+1) = \Theta(n^2)$.

5. If $T(n) = 4T(n/2) + O(n)$, then $T(n) = O(n^3)$.

   **True**: By the master theorem, $T(n) = O(n^2)$. Since $O$-notation represents an upper bound, $T(n)$ is also $O(n^3)$.

6. If $T(n) = 4T(n/2) + O(n)$, then $T(n) = O(n^2\log n)$.

   **True**: By the master theorem, $T(n) = O(n^2)$. Since $O$-notation represents an upper bound, $T(n)$ is also $O(n^2\log n)$.

7. If $a^{n-1} \equiv 1 \mod n$ for all positive integers $a < n$, then $n$ is prime.

   **False**: Carmichael numbers are rare composite numbers $n$ satisfying Fermat's test for all positive integers $a < n$.

8. The probability that an $n$-bit integer is prime is roughly $2^{-n}$.

   **False**: By Lagrange's theorem, the number of primes less than an $n$-bit integer $x$, for large $n$, is roughly $\frac{x}{\ln x}$. Therefore, the probabilty that an $n$-bit integer is prime is roughly $\frac{(x/\ln x)}{x} = \frac{1}{\ln x} \approx \frac{1}{\ln(2^n)} = \frac{1}{n\ln 2} \approx \frac{1.44}{n}$.

9. Any DAG with a unique source and sink has a unique topological ordering.

   **False**: Consider a directed graph with four nodes A, B, C, and D and four edges (A,B), (A,C), (B,D), and (C,D). There are two possible topological orderings: A,B,C,D and A,C,B,D.

10. Breadth first search and depth first search produce the same tree on connected undirected graphs if and only if the graph is a tree.

    **True**: Suppose the input graph is an undirected tree $T$. Both DFS and BFS must produce a tree, so they must contain all the edges of $T$ (all trees have $|V| - 1$ edges). Since two trees must be identical if they have the same root and same edges, both DFS and BFS will produce $T$.

    Conversely, suppose the input graph $G$ is undirected and connected but is not a tree. Then $G$ must contain a cycle $C$. Suppose $C$ consists of the $k$ nodes $v_1, v_2, \ldots, v_k$, i.e. $C$ is the cycle $v_1 \to v_2 \to \ldots \to v_k \to v_1$. Now in the DFS tree, nodes $v_1, v_2, \ldots, v_k$ will all be on the same path from the root to a leaf. Why? Suppose $v_f$ is the first of these nodes to be visited. Then, the remaining nodes must be visited at some point during `explore(`$v_f$`)` since the $v_i$ are all strongly connected. However, in the BFS tree, nodes $v_1, v_2, \ldots, v_k$ will form at least two branches, braching from the node first visited (imagine performing BFS on the cycle).

    Therefore, BFS and DFS produce the same tree iff the input graph is a tree.

## Problem 2.1 (5 points each)

The number of bits in an integer $x$ is $\Theta(\log x)$.

Therefore, the number of bits in $a^b$ is $\Theta(\log(a^b)) = \Theta(b \log a)$. Now $\log a = O(n)$ and $b = O(2^n)$ since both $a$ and $b$ are $n$-bit numbers, so $b \log a = O(n \cdot 2^n)$. Thus, $O(n \cdot 2^n)$ is an upper bound on the number of bits in $a^b$.

Similarly, the number of bits in $a^{b^c}$ is $\Theta(\log(a^{(b^c)})) = \Theta(b^c \log a)$. Again, $\log a = O(n)$, $b = O(2^n)$, and $c = O(2^n)$ since $a$, $b$, and $c$ are $n$-bit numbers, so $b^c \log a = O(n \cdot (2^n)^{(2^n)}) = O(n \cdot 2^{(n \cdot 2^n)})$. Thus, $O(n \cdot 2^{(n \cdot 2^n)})$ is an upper bound on the number of bits in $a^{b^c}$.

## Problem 2.2 (5 points each)

(a) The secret key $d$ equals $e^{-1} \mod (p-1)(q-1)$. Since the $n$-bit primes $p$ and $q$ are given as input, $d$ can be found in $O(n^3)$ time using the extended Euclid's algorithm.

(b) The ciphertext $c$ equals $m^e \mod pq$. Therefore, $c$ can be computed in $O(n^2 \log e)$ time using the repeated squaring method for modular exponentiation.

## Problem 3.1

First off, $o(kl)$ means asymptotically faster than $O(kl)$. This threw most people off, and we gave no credit for $\Omega(kl)$ solutions since they were trivial.

Here is a solution for selecting the $s$th item.

Find the median of each element of each array by choosing the middle element. Then, find the median of these medians which takes $O(k)$ time using linear time median finding or $O(k \log k)$ time by sorting them. Then, we use this element to partition the elements in each list (by binary search we can find which are bigger and which are smaller; requiring $O(k \log l)$ time.

Then, we will recurse with the following modification on the side with the $s$th item.

Notice, that some lists may remain large in the recursive call but some will be smaller. So, in the recursion we don't take the median of medians but the weighted median of medians; sort them and add up the size of the previous lists until they add up to half.

Now, the median of medians will be less than at least $n/4$ items and greater than at least $n/4$ items, so the recursion becomes as follows.

$$T(n) \leq O(k \log l) + O(k \log k) + T(3n/4).$$

The first term is the time to find the split index for the arrays using binary search in each array. The second for finding the weighted median and the third for the recursive call.

Initially there were $n = kl$ elements in all, and after $O(\log l)$ recursive calls we have $O(k)$ elements, in which case we can finish in time $O(k)$. Thus, the total cost is bounded by the time for $O(\log l)$ recursive calls. This is bounded by $O(k \log^2 l + k \log l \log k)$.

## Problem 3.2

Since $\omega = e^{2\pi i/4} = i$, the FFT matrix is

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

The first row is the FFT of $[1, 0, 0, 0]$, the last row is the FFT of $[0, 0, 0, 1]$.

## Problem 4

### 4.1 (10 pts)

$u$ and $v$ are on a cycle if and only if there is a path from $u$ to $v$ and a path from $v$ to $u$. Thus, when running DFS starting from $u$, $v$ must be a descendant of $u$ ($pre(u) < pre(v) < post(v) < post(u)$), and when running DFS starting from $v$, $u$ must be a descendant of $v$ ($pre(v) < pre(u) < post(u) < post(v)$). The algorithm proceeds as follows. Run DFS from each node, keep pre/post number of each DFS run. If there exists a pair of nodes $u$ and $v$, satisfying $pre(v) < pre(u) < post(u) < post(v)$ for one DFS run and $pre(u) < pre(v) < post(v) < post(u)$ for another DFS run, then $u$ and $v$ are on a cycle. If there doesn't exist such a pair of nodes, then output "No two nodes are on a cycle".

### 4.2 (4 pts)

$pre(u) < post(u) < pre(v) < post(v)$. Suppose $v$ is explored before $u$. Since $u$ is reachable from $v$ and this is a directed graph, we must have $pre(v) < pre(u) < post(u) < post(v)$, contradicting the assumption that the intervals are disjoint. Therefore, $u$ must be explored before $v$ giving $pre(u) < post(u) < pre(v) < post(v)$.

It should also be pointed out that in case of an undirected graph, both $pre(u) < post(u) < pre(v) < post(v)$ and $pre(v) < post(v) < pre(u) < post(u)$ are possible. However, we gave full credit if you only answered the directed graph case.

### 4.3 (6 pts)

In the case of a connected graph, $pre(s) = 1$ and $post(s) = 2|V|$, where $|V|$ is the total number of nodes in the graph. In the case of an unconnected graph, $pre(s) = 1$ and $post(s) = 2|C| < 2|V|$, where $|C|$ is the number of nodes of the connected component containing the starting point $s$ (i.e. the number of the nodes reachable from the starting point, including the starting point itself).

## Problem 5

### Solution 1 (Full Credit)

Start from any node and run Dijkstra's algorithm. Then $k + 1$ times, do the following: update all the negative edges (not strictly necessary but easier to prove correctness), place everything back in the priority queue, and run the loop of Dijkstra's that pulls nodes out and updates the adjacent edges. The graph has a negative cycle if and only if any $dist(v)$ changes in the last iteration (i.e. the $k + 1$ iteration). The top level algorithm `NegativeCycle`, which uses subroutines `DijkstraLoop` and `UpdateNegativeEdges`, is given below.

**Algorithm 1**: NegativeCycle

```
Input: Input graph $G$
Output: True if $G$ contains a negative cycle, False otherwise
$s$ = any vertex $\in V$;
for all $u \in V$ do
    $dist(u) = \infty$;
end
$dist(s) = 0$;
$Q = makequeue(V, dist)$;
$DijkstraLoop(G, Q)$;
for $i = 1$ to $k$ do
    $UpdateNegativeEdges(G, dist)$;
    $Q = makequeue(V, dist)$;
    $DijkstraLoop(G, dist, Q)$;
end
if $UpdateNegativeEdges(G, dist)$ then
    return True
else
    $Q = makequeue(V, dist)$;
    return $DijkstraLoop(G, dist, Q)$

```

**Algorithm 2**: DijkstraLoop

```
Input: Input graph $G$, array of dist values $dist$, priority queue $Q$
Output: True if any $dist$ value is updated, False otherwise
$retValue = False$;
while $Q$ is not empty do
    $u = deletemin(H)$;
    for all edges $(u, v) \in E$ do
        if $dist(v) > dist(u) + l(u, v)$ then
            $dist(v) = dist(u) + l(u, v)$;
            $decreasekey(H, v)$;
            $retValue = True$;

    end
end
return retValue
```

**Algorithm 3**: UpdateNegativeEdges

```
Input: Input graph $G$, array of dist values $dist$
Output: True if any $dist$ value is updated, False otherwise
1 $retValue = False$;
2 for $all\ edges\ (u, v) \in E$ do
3     if $l(u, v) < 0\ AND\ dist(v) > dist(u) + l(u, v)$ then
4         $dist(v) = dist(u) + l(u, v)$;
5         $retValue = True$;
6
7 end
8 return $retValue$
```

We will prove the following claim by induction:

> After the $i$th iteration of the for loop (lines 8-12) of our algorithm `NegativeCycle`, $dist(v)$ is less than or equal to the length of any path to $v$ that uses up to $i$ negative edges.

For the base case (after 0 iterations), we have already run Dijkstra's algorithm on the graph (line 7 of `NegativeCycle`). Thus, $dist(v)$ is at most the length of any path of positive edges in the graph. This is by the fact that the unmodified Dijkstra's algorithm returns a path no longer than the shortest positive edge path.

Now by the inductive hypothesis, after iteration $i$, $dist(v)$ at most the length of any path to $v$ that uses up to $i$ negative edges. Then, after updating the negative edges, every $dist(v)$ is less than or equal to the length of any path with $i$ negative edges that ends with a negative edge. Finally, the pass through the queue in `DijkstraLoop` ensures that, for each node, $dist(v)$ is less than or equal to the $dist$ value of any other node plus the length of a postive edge path to $v$. Thus, after iteration $i + 1$, $dist(v)$ is at most the length of *any* path to $v$ using up to $i + 1$ negative edges.

Therefore, as a specific case of our claim, after $k$ iterations of the for loop, $dist(v)$ is less than or equal to the length of any path to $v$ that uses up to $k$ negative edges. If a $dist(v)$ is updated on the $k + 1$ iteration (lines 13-17 of `NegativeCycle`), then the path $P$ corresponding to that $dist$ value must use at least $k + 1$ negative edges. Since there are only $k$ negative edges in our graph, $P$ must contain a cycle. This cycle must be a negative cycle because $dist(v)$ is strictly decreasing ($P$ would not contain a cycle unless it had negative weight).

Conversely, assume our graph has a negative cycle. Then, for at least one vertex $v$, we can create a path to $v$ of arbitrarily small length and drive it's $dist$ value arbitrarily low. Now, if a particular iteration of our algorithm does not update any $dist$ values, then the $dist(v)$ values will never change again. So if a particular iteration of our algorithm does not update any $dist$ values, there cannot be a negative cycle because we can no longer obtain arbitrarily low $dist$ values. Specifically, if the $k + 1$ iteration of our algorithm does not update any $dist$ values, then the graph does not have a negative cycle.

For the runtime, we have $\Theta(k)$ iterations of Dijkstra's algorithm (which each take $O(m + n \log n)$ time using a Fibonacci heap to implement the priority queue) and $\Theta(k)$ updates of the negative edges (which each take $O(m)$ time). So the total running time is $O(k(m + n \log n))$.

## Solution 2 (3/4 Credit)

Remove all the negative edges. Find a single source shortest path from each endpoint of a negative edge to each other endpoint (using Dijkstra's algorithm). Construct a graph on the (at most) $2k$ nodes with edge distances that correspond to the shortest path distances between the corresponding point in the positive edge graph. Put in the negative edges between the original nodes. This is a graph on $2k$ nodes with $O(k^2)$ edges. One can check for a negative cycle in the is graph in time $O(k^3)$.

Any negative cycle in the new graph corresponds to a negative cycle in the original since any edge in this graph corresponds to a path in the original graph.

Moreover, any negative cycle $C$ in the original graph must contain one or more negative edges (denote the set of negative edges by $N$). Suppose $C$ does not consist of positive edge shortest paths between endpoints of negative edges. Then the cycle $C'$ formed by connecting the endpoints of $N$ using positive edges shortest paths has weight less than $C$. Thus, $C'$ is a negative cycle. This shows that if there is a negative cycle $C$ in the original graph, then there is a negative cycle consisting of positive edge shortest paths between endpoints of negative edges, i.e. a negative cycle in the new graph.

Thus, our algorithm finds a negative cycle if and only if a negative cycle existed in the original graph.

We have $\Theta(k)$ iterations of Dijkstra's algorithm following by performing Bellman-Ford on graph with $\Theta(k)$ nodes and $\Theta(k^2)$ edges. Thus, the running time for this is $O(k(m + n \log n) + k^3)$ by implementing the priority queue with a Fibonacci heap.