Data structures

- Binary heaps are implemented using a heap-ordered balanced binary tree. Binomial heaps use a collection of $B_k$ trees, each of size $2^k$. Fibonacci heaps use a collection of trees with properties a bit like $B_k$ trees. (The operation HEAPIFY below makes a heap with $n$ elements without doing $n$ INSERTs.)

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary heap (worst case)</th>
<th>Binomial heap (worst case)</th>
<th>Fibonacci heap (amortized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>HEAPIFY</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$O(\lg n)$</td>
<td>$O(\lg n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$O(1)$</td>
<td>$O(\lg n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$O(\lg n)$</td>
<td>$O(\lg n)$</td>
<td>$O(\lg n)$</td>
</tr>
<tr>
<td>UNION</td>
<td>$O(n)$</td>
<td>$O(\lg n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$O(\lg n)$</td>
<td>$O(\lg n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$O(\lg n)$</td>
<td>$O(\lg n)$</td>
<td>$O(\lg n)$</td>
</tr>
</tbody>
</table>

- Binary search trees implement all the operations of heaps (except UNION) in addition to SEARCH. Virtually all the operations take time $O(\log n)$.

- The union-find data structure implements the operations UNION($x, y, \text{label}$) and $\text{label} \leftarrow \text{FIND}(x)$ on a collection of disjoint sets. Initially (before any UNION operation) each of $n$ elements is in its own set of size one. A total of $m$ disjoint set operations take time $O(m \alpha(m, n))$, where $\alpha(m, n)$ is a ridiculously slowly growing function. $\alpha(m, n)$ is $o(\log^* m)$, $o(\log m)$, and $\Omega(1)$.

Sorting

- For comparison-based sorting algorithms, Heapsort and mergesort take time $O(n \log n)$, and quicksort takes expected time $O(n \log n)$. An information theoretic lower bound for any comparison based sort is $\log(n!) = \Omega(n \log n)$.

- For $n$ numbers known to fall within a range $\{1, \ldots, N\}$, radix sort will take time $O(n + N)$. Linear time algorithms are often possible if more can be known about the numbers besides how to pairwise compare them.

- Order statistics (the $k^{th}$ largest element of $n$ elements, or the median of $n$ elements) can be found in time $\Theta(n)$ by a comparison based algorithm. The algorithm chooses a pivot by recursively computing the median of the medians of $n/5$ subsets of 5 elements each. Once all elements are compared to the pivot, $1/4$ of the elements can be discarded, as they are all known to be greater than (or, less than) the $k^{th}$ largest element.
Exploring graphs

- Breadth first search (BFS) takes $O(E)$ time and finds shortest paths.
- Depth first search (DFS) takes $O(E)$ time. Also in this time you can have preorder numberings (or discover times), postorder numberings (or finish times), classification of edges as forward, back, back-cross or back-cross-tree edges.
- A topological sort of a dag can be found in $O(E)$ time by reversing the postorder numbers in a DFS.
- Strongly connected components (SCC’s) can be found in $O(E)$ time. Note that the component graph, $G^{SCC}$ is acyclic, so you can topologically sort it.

Minimum Spanning Trees (MST’s)

- If $A \subset E$ is part of a MST, and $S$ is a cut which no edge in $A$ crosses, then the minimum edge across the cut can be added to $A$ to yield part of a MST.
- Kruskal’s grows a collection of trees by always adding the cheapest edge which connects two trees, taking time $O(E \lg V)$ to sort the edges.
- Prim’s algorithm grows a single tree from a vertex by always adding the cheapest edge out from the tree, taking time $O(E + V \lg V)$ if a Fibonacci heap is used.

Shortest path problems

- Single-source shortest paths for non-negative edge weights can be found by Dijkstra’s algorithm. Like Prim’s, grow a tree, always adding the vertex with the cheapest path from the source by extending the tree by only one edge. Takes $O(E + V \lg V)$ using a Fibonacci heap.
- Single-source shortest paths for edge weights which may be negative can be found using Bellman-Ford. Make $V$ passes over the graph, updating shortest path estimates to each vertex relaxing edges. Either a negative cycle will be found or a shortest path tree in time $O(VE)$.
- All-pairs shortest path problems had a few algorithms:
  - Matrix-multiply like algorithms taking time $O(V^3)$ or $O(V^3 \log V)$.
  - Floyd Warshall takes time $O(V^3)$. They use dynamic programming to solve
    \[ d^{(k)}_{ij} = \text{the shortest path from } v_i \text{ to } v_j \text{ using paths going through } \{v_1, \ldots, v_k\} \]
  - Johnson’s algorithm first solves a single source shortest path problem from an added vertex, $s$, (with edges $(s, v), v \in V$ of weight $0$) to reweight the edges by:
    \[ w = w(u, v) + \delta(s, u) - \delta(s, v) \]

    This results in positive edge weights, and Bellman-Ford can be used to take time $O(VE)$.

Linear programming

- In linear programming, the goal is to optimize a linear objective function subject to linear inequality constraints. No algorithm were discussed, but polynomial time algorithms exist for linear programming.
- In integer linear programming, the goal is to find an integer solution to a linear programming problem. No polynomial time algorithm is known nor is likely to exist for this NP-complete variant.

Flow networks and maximum flows

- Capacities satisfy $c(u, v) \geq 0$. Flows satisfy
  \[ \text{Capacity constraints : } f(u, v) \leq c(u, v) \]
  \[ \text{Skew symmetry: } f(u, v) = -f(v, u) \]
  \[ \text{Flow conservation: } \forall u \in V - \{s, t\} : \sum_{u \in E} f(u, v) = 0 \]
- The residual capacities are given by $c_f(u, v) = c(u, v) - f(u, v)$.
- The min-cut max-flow theorem proves the maximum flow is equal to the minimum capacity over all cuts. Further, if there are no augmenting paths in the residual graph, a maximum flow has been obtained.
- Ford-Fullerson finds paths from $s$ to $t$ in the residual graph to augment the flows until no more paths can be found, taking time $O(Ef^*)$, where $f^*$ is the value of the max-flow.
Edmonds-Karp improves on this by always choosing the shortest augmenting path (i.e., fewest edges), finding the max-flow in time $O(VE^2)$. 

**Number theoretic algorithms** Throughout, define $\beta$ to be the length of bits in all the numbers involved.

- The greatest common divisor $\gcd(a, b) = \gcd(b, a \mod b)$, yielding Euclid’s algorithm taking $O(\beta)$ arithmetic operations.
- If $\gcd(a, b) = d$ then there is an $x$ and $y$ so that $ax + by = d$. Euclid’s algorithm can be adjusted to calculate $x$ and $y$ efficiently.
- (Chinese Remainder Theorem) If $\gcd(n_1, n_2) = 1$ and $n = n_1n_2$, then there is a one-to-one mapping between numbers $a \mod n$ and pairs $(a \mod n_1, a \mod n_2)$ so that $a = a \mod n_i$. To compute $a$, find $x$ and $y$ so that $n_1x + n_2y = 1$ and notice that $(1, 0)$ maps to $n_2y \mod n$ and $(0, 1)$ maps to $n_1x \mod n$. So, $(a_1, a_2)$ maps to $a_1n_2y + a_2n_1x \mod n$.
- (Fermat’s Little Theorem) For $p$ prime, $1 \leq a < p$, $a^{p-1} \equiv 1 \pmod{p}$.
- A pseudo-prime test is to check if $2^{n-1} \equiv 1 \pmod{n}$; output “prime?” if yes, “composite!” if no. Very few composites look like primes. A randomized primality test chooses $k$ random values of $a$ in the range $1 \leq a < n$. For each, calculate $a^{n-1} \equiv 1 \pmod{n}$. If one is not $\pm 1$ output “composite!”. If all are 1 output “prime?”. Otherwise output “prime?”.
- In the RSA public-key cryptosystem, a participant creates her public and private keys with the following procedure.
  1. Select at random two large prime numbers $p$ and $q$.
  2. Compute $n$ by the equations $n = pq$.
  3. Select a small odd integer $e$ that is relatively prime to $\phi(n) = (p - 1)(q - 1)$.
  4. Compute $d$ as the multiplicative inverse of $e \mod \phi(n)$.
  5. Publish the pair $P = (e, n)$ as her RSA public key.
  6. Keep secret the pair $S = (d, n)$ as her RSA secret key.

To encode message $M$, compute $M^e \mod n$. To decode cyphertext $C$, compute $C^d \mod n$. 

You should not need to write any code for this exam. Please make your answers as brief and as clear as possible. I highly recommend crossing out mistakes with a few dark strokes of the pen rather than erasing the work completely in case it is worth partial credit.

Below is a summary line for each question on the exam. You may use the backs of pages if you need more space, but please indicate for the grader you’ve done so: For example, “Continued on back of page 4”. Please leave the exam stapled. You can look pick up a solution set (with the questions repeated) when you leave.

| 1. | Edmonds-Karp algorithm | /20 |
| 2. | Linear programming definitions | /15 |
| 3. | Job scheduling linear program | /30 |
| 4. | Non-cycle edges | /20 |
| 5. | Unit-capacity edges | /30 |
| Total | (Extra points possible) | /115 |
1. (20 points) The following gives capacities and the flow after executing one round of the Edmonds-Karp improvement to the Ford-Fulkerson algorithm:

(a) Draw the residual graph, $G_f$ in the space below. The dotted edges (and the extra graph) are for your convenience. You need not include edges into $s$ or from $t$ in $G_f$.

(b) Draw the flow obtained after one more iteration of Edmonds-Karp. Please do not indicate the capacities; you only need to draw edges with flow.
2. (15 points) Consider the five problems labeled (a)-(e) below:

\[
\begin{align*}
\text{(a)} & \quad & \text{min } 3x_1 + 4x_2 & \quad \text{s.t.} & \quad & \text{min } x_1 + x_2 & \quad \text{s.t.} \\
& & x_2^2 + x_2 & \leq & 5 & x_1, x_2 & \in \{\ldots, -2, -1, 0, 1, 2, 3, \ldots \} \\
& & x_2 & \geq & 0 & 2x_1 + x_2 & \leq 5 \\
& & & & & .5x_1 - 3x_2 & \geq -4 \\
\text{(b)} & \quad & \text{max } x - 3y + 5z & \quad \text{s.t.} & \quad & 2x_1 + y & = 4 \\
& & -y + z & \geq & 2 & -x + z & \leq 6 \\
\text{(c)} & \quad & \text{min } x_1 + x_2 & \quad \text{s.t.} & \quad & 2x_1 + x_2 & \leq 5 \\
& & .5x_1 - 3x_2 & \geq & -4 & 3x_1 + 2x_2 & \geq 10 \\
\text{(d)} & \quad & \text{min } x_1 + x_2 & \quad \text{s.t.} & \quad & 2x_1 + x_2 & \leq 5 \\
& & .5x_1 - 3x_2 & \geq & -4 & 3x_1 + 2x_2 & \geq 10 \\
\end{align*}
\]

Of the five, two are linear programs and one is an integer linear program. Indicate which in the boxes below:

- Linear program
- Linear program
- Integer linear program
3. (30 points) We want to determine the optimal scheduling of \( m \) jobs to a machine such that:

- All jobs must be completed within \( n \) weeks.
- Job \( i \) requires a total of \( r_i \) hours.
- At most \( h_j \) hours can be scheduled on the machine during week \( j \).
- There is a cost \( c_{ij} \) for each hour that job \( i \) is assigned to the machine during week \( j \).

This problem can be formulated as a linear program with \( m \times n \) variables, where \( x_{ij} \) is the number of hours the machine spends on job \( i \) during week \( j \).

(a) Write the objective function that minimizes the total cost for a possible schedule.

(b) For job \( i \) which requires \( r_i \) hours, write the corresponding linear constraint.

(c) For week \( j \) which has at most \( h_j \) hours available, write the constraint corresponding to the jobs scheduled during this week.

(By the way, the additional constraints, “\( \forall i, j: x_{ij} \geq 0, \)” will complete the linear program.)
4. (20 points) Give an $O(E + V)$ algorithm to find all edges in a directed graph $G = (V, E)$ which are not contained in any cycle. Hint: Use depth first search, breadth first search, strongly connected components and/or topological sort as subroutines. (If your algorithm is simple and clearly stated, no justification is required. A one sentence solution could receive full credit.)
5. (30 points) Let $G = (V, E)$ be a flow network with source $s$, sink $t$, and suppose each edge $e \in E$ has capacity $c(e) = 1$. Assume also, for convenience, that $|E| = \Omega(V)$.

(a) Suppose we implement the Ford-Fulkerson maximum-flow algorithm by using depth-first search to find augmenting paths in the residual graph. What is the worst case running time of this algorithm on $G$?

(b) Suppose a maximum flow for $G$ has been computed, and a new edge with unit capacity is added to $E$. Describe how the maximum flow can be efficiently updated. (Note: It is not the value of the flow that must be updated, but the flow itself.) Analyze your algorithm.

(c) (Extra credit) Suppose a maximum flow for $G$ has been computed, but an edge is now removed from $E$. Describe how the maximum flow can be efficiently updated in $O(E + V)$ time.