NAME: ______________________________

CSE 421
Introduction to Algorithms
Final Exam Winter 2005

P. Beame

14 March 2005

DIRECTIONS:

- Answer the problems on the exam paper.
- Open book. Open notes.
- If you need extra space use the back of a page
- You have 1 hour and 50 minutes to complete the exam.
- Please do not turn the page until you are instructed to do so.
- Good Luck!

1	/33
2	/20
3	/18
4	/20
5	/20
6	/20
7	/20
8	/10
Total	/161

1. (33 points, 3 each) For each of the following problems circle **True** or **False**. You do not need to justify your answer.

   (a) If f and g are two different flows on the same flow graph (G, s, t) and if $\nu(f) \geq \nu(g)$ then every edge e in G has $f(e) \geq g(e)$. **True** **False**
   False

   (b) If f is a maximum flow on a flow graph $(G = (V, E), s, t)$ and B is the set of vertices in V that can reach t in the residual graph G_f then $(V - B, B)$ is a minimum capacity s-t cut in G.
   True **False**
   True

   (c) If f is a maximum flow on a flow graph (G, s, t) and (S, T) is a minimum capacity s-t cut in G then *every* edge e having endpoints on different sides of (S, T) has $f(e)$ equal to the capacity of e. **True** **False**
   False

   (d) If problem B is NP-hard and $A \leq_P B$ then A is NP-hard. **True** **False**
   False

   (e) If problem A is NP-hard and $A \leq_P B$ then B is NP-hard. **True** **False**
   True

   (f) If $P \neq NP$ then every problem in NP requires exponential time. **True** **False**
   False

   (g) If problem A is in P then $A \leq_P B$ for every problem B in NP. **True** **False**
   True

   (h) If G is a weighted graph with n vertices and m edges that does *not* contain negative-weight cycle, then the iteration of the Bellman-Ford algorithm will reach a fixed point in at most $n - 1$ rounds. **True** **False**
   True

   (i) If G is a weighted graph with n vertices and m edges that *does* contain negative-weight cycle, then for *every* vertex v in G the shortest path from v to t in G containing n edges is strictly shorter than the shortest path from v to t in G containing $n - 1$ edges.
   True **False**
   False

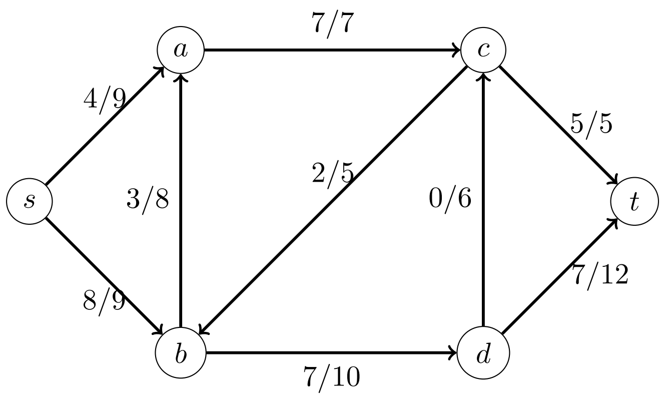
   (j) A divide and conquer algorithm for $n \times n$ matrix multiplication that solves the problem using 24 matrix multiplication subproblems of size $n/3$ and $O(n^2)$ additional work for recombining would be more efficient than the usual $\Theta(n^3)$ matrix multiplication algorithm.
   True **False**
   We did not cover this problem

(k) A divide and conquer algorithm for the selection problem that reduced the problem on lists of size n to one selection problem of size $3n/20$ and another selection problem of size $9n/10$ plus a linear amount of extra work would yield a linear time algorithm.

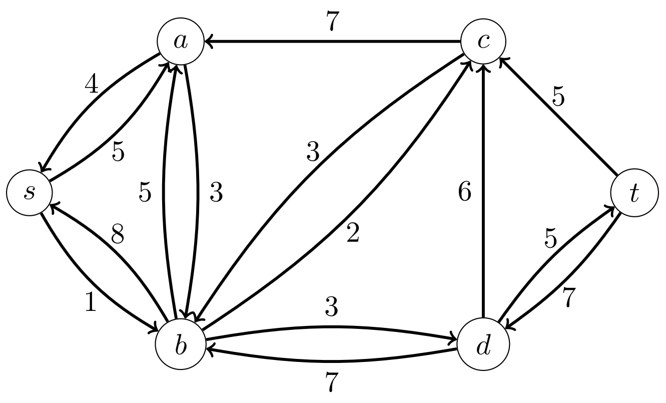
True **False**

False

2. (20 points) Consider the following flow network with a flow f shown. An edge labelled "b" means that it has capacity b and flow 0. An edge labelled "a/b" means that the flow on that edge is a and the capacity is b.



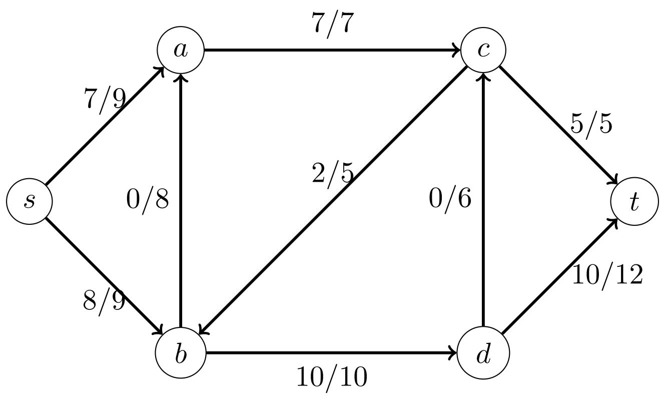
(a) (10 points) Draw the residual graph G_f.



(b) (3 points) What augmenting path in this graph would result in the greatest increase in flow value? (List the names of the vertices on this path in order.)

s, a, b, d, t. It results in an increase of 3 because minimum capacity is 3.

(c) (7 points) On the diagram below, indicate the new flow values resulting from augmenting along the path you found in part (b).



3. (18 points) For each of the following recurrences give the time and space that would be required by a simple dynamic programming algorithm to compute the answer where the values of the w function are given as input.

   (a) Compute $OPT(n)$ where $OPT(1) = 1$ and $OPT(i) = \min_{1 \le j < i}\{OPT(j)/j + w(j)\}$.
   Time=$O(n^2)$, Space=$O(n)$.

   (b) Compute $OPT(m, n)$ where $OPT(i, 0) = OPT(0, j) = 0$ for all i and j and
   $OPT(i, j) = \min\{OPT(i-1, j) + 2,\ OPT(i, j-1) + 1,\ OPT(i-1, j-1)/2 + w(j)\}$ for $i, j > 0$.
   Time=$O(mn)$, Space=$O(mn)$.

   (c) Compute $OPT(1, n)$ where $OPT(i, i) = 0$ for all i, and
   $OPT(i, j) = \min\{OPT(i, k) + OPT(k+1, j) + w(k)\ :\ i \le k < j\}$ for all $i < j$.
   Time=$O(n^3)$, Space=$O(n^2)$.

4. (20 points) You are given a sequence of n bits $x_1, \ldots,\ x_n \in \{0,1\}$. Your output is to be either

- any i such that $x_i = 1$ or
- the value 0 if the input is all 0's.

The only operation you are allowed to use to access the inputs is a function **Group-Test** where

$$\textbf{Group-Test}(i,j) = \begin{cases} 1 & \text{if some bit in } x_i, \ldots, x_j \text{ has value 1} \\ 0 & \text{if all bits in } x_i, \ldots, x_j \text{ have value 0} \end{cases}$$

(Historical Note: In World War I, the army was testing recruits for syphilis which was rare but required a time-consuming though accurate blood test. They realized that they could pool the blood from several recruits at once and save time by eliminating large groups of recruits who didn't have syphilis.)

(a) (15 points) Design a divide and conquer algorithm to solve the problem that uses only $O(\log n)$ calls to **Group-Test** in the worst case. Your algorithm should *never* access the x_i directly.

(b) (5 points) Briefly justify your bound on the number of calls.

```
$i \leftarrow 1, j \leftarrow n$
while $i < j$ do
    $mid \leftarrow \lfloor \frac{i+j}{2} \rfloor$
    if Group-Test$(i, mid) == 1$ then
        $j = mid$
    else
        $i = mid + 1$
    end if
    Return Group-Test$(i, i)$
end while
```

Correctness Given an interval $\{i, i+1, \ldots, j\}$ we partition it into two (almost) equal size intervals where the first interval is $\{i, \ldots, \lfloor (i+j)/2 \rfloor\}$ and the second interval is $\{\lfloor (i+j)/2 \rfloor + 1, \ldots, j\}$. Then we test if there is a 1 bit in the first interval using the Group-Test$(i, \lfloor (i+j)/2 \rfloor)$; if the answer is 1 then we can recursively solve the problem on the first interval. Otherwise, if Group-Test$(i, \lfloor (i+j)/2 \rfloor) = 0$ there is no solution in the first interval. So, either there is a solution in the second interval or there is no solution in the second interval. In either case we can reduce the problem to the second interval.

Runtime: The length of the interval decreases by a factor 2 each time. So, we are only going to call Group-Test at most $O(\log n)$ many times.

5. (20 points) In classifying butterflies of several species, a biologist makes time-consuming comparisons between butterfly features to determine whether or not two butterflies are the same species. A biologist wants to determine as efficiently as possible whether or not there is a *dominant species* among n butterfly specimens, $B = (b_1, \ldots, b_n)$. (A dominant species is one that occurs strictly more than $n/2$ times in the list of specimens.)

   (a) Describe a divide and conquer algorithm $\mathit{Dominant}(B, n)$ to tell the biologist how to do this in $O(n \log n)$ applications of operation $\mathit{Same}(b, b')$ which returns **true** if b and b' are the same species and **false** otherwise.

   (b) Explain why your solution is correct.

   (c) Write the recurrence that justifies your claim for the $O(n \log n)$ running time.

Algorithm 1 FindDominant

```
procedure DOMINANT($B, n$)                                    ▷ $B$ is passed as a reference
    if $n == 1$ then return $B[1]$
    else
        Left ← $\mathit{Dominant}(B[1, \ldots, \lfloor \frac{n}{2} \rfloor])$
        Right ← $\mathit{Dominant}(B[\lfloor \frac{n}{2} \rfloor + 1, \ldots, n])$
        if Left $\neq$ empty then LeftCount=number of Left appearing in $A[1, \ldots, n]$
        end if
        if Leftcount> $\frac{n}{2}$ then
            return Left
        else
            if Right $\neq$ empty then
                RightCount = number of Left appearing in $A[1, \ldots, n]$
            end if
            if RightCount> $\frac{n}{2}$ then
                return Right
            end if
        end ifreturn empty
    end if
end procedure
```

Correctness First we show that if there is a dominant species in the whole array, then it must be a dominant species in either the first half of the array or the second half of the array. Let e be the dominant species of $B[1, \ldots, n]$. If e is not the dominant species in the first part and not the dominant species in the second part, then e appears at most $\frac{1}{2} \cdot \lfloor \frac{n}{2} \rfloor + \frac{1}{2}(n - (\lfloor \frac{n}{2} \rfloor + 1) + 1) = \frac{n}{2}$ times in the whole array, and so it cannot be the majority element overall.

Hence the possible candidate for the majority element can only be Left or Right, and the above algorithm checks the validity. To count the number of appearance of Left in the whole array we simply sum up Same(Left,B[i]) over all $1 \leq i \leq n$ (we do similarly for Right). Moreover, the algorithm also works for base case, namely array containing one element. So the algorithm will return a correct answer.

Running Time This algorithm solves two subproblems of size $\frac{n}{2}$ and then uses linear number of calls of Same function to merge. Therefore, the running time corresponds to the recurrence $T(n) = 2T(\frac{n}{2}) + O(n)$, so $T(n) = O(n \log n)$ by Master theorem.

6. (20 points) You are given a *directed acyclic graph* $G = (V, E)$ and a node node $t \in V$. Design a linear time algorithm to compute for each vertex $v \in V$ the *number* of different paths from v to t in G. Analyze its running time in terms of $n = |V|$ and $m = |E|$.

   Answer. Compute a topological sorting of the vertices and let's rename them to $1, 2, \ldots, n$ such that for every edge (i, j) we have $i < j$. Now, let $OPT(i)$ be the number of paths from vertex i to t. Observe that if $i > t$ then $OPT(i) = 0$ because every path from i only goes to vertices with higher index but $t < i$.

   Now, let's compute $OPT(i)$. Obviously $OPT(t) = 1$ because if we go out of t we cannot get back to t.

   Now, suppose we have computed $OPT(j)$ for all $i < j \leq t$. We compute $OPT(i)$. Let's try to characterize $OPT(i)$. The first edge of $OPT(i)$ out of vertex i goes to a neighbor of i say j. At this moment we don't know j but later we will brute force for all possibilities of j. Observe that by IH there are $OPT(j)$ many paths from j to t. Any of paths from j to t will give us a path from i to t once we also include the edge (i, j). Therefore in this case we get

$$OPT(i) = OPT(j).$$

   Taking all possibilities of j we get

$$OPT(i) = \sum_{j:(i,j)\in E} OPT(j).$$

   In the above sum we have counted all paths because every path out of i starts with an edge (i, j) and we have not double count any path because if the starting edge of two paths are different, they are different paths. So, we get the recurrence

$$OPT(i) = \begin{cases} 0 & \text{if } i > t \\ 1 & \text{if } i = t \\ \sum_{j:(i,j)\in E} OPT(j) & \text{otherwise.} \end{cases}$$

   Running Time: The algorithm takes $\deg(i)$ steps to compute $M[i]$ for each vertex $i < t$.

```
Find a topological sorting of G and rename vertices such that $i < j$ for all $(i, j) \in E$.
for $i = t + 1 \to n$ do
    $M[i] \leftarrow 0$
end for
$M[t] = 1$
for $i = t - 1 \to 1$ do
    $M[i] \leftarrow 0$
    for each edge $(i, j) \in E$ do
        $M[i] \leftarrow M[i] + M[j]$
    end for
end for
```

   For $i \geq t$, we compute $M[i]$ in $O(1)$ steps. Therefore the algorithm runs in $O(\sum_i \deg(i)) = O(n + m)$ in the worst case.

7. (20 points) The *Number Partition* problem asks, given a collection of non-negative integers $y_1, \ldots, y_n$ whether or not it is possible to partition these numbers into two groups so that the sum in each group is the same. Prove that *Number Partition* is NP-complete by solving the following problems.

   (a) Show that *Number Partition* is in NP.
   Answer. Given a partition of $y_1, \ldots, y_n$ into two groups the verifier sums up the numbers in each group and outputs yes if the two sums are equal and no otherwise. The verifier obviously runs in time polynomial in n and $\log y_1, \ldots, \log y_n$ because we can add up any two integers a, b in time $O(\log a + \log b)$.

   (b) Show that *Subset Sum* $\leq_P$ *Number Partition*
   Recall that in the *Subset Sum* problem we are given a collection of integers (which can be positive and negative), we want to see if it is possible find a subset that sum up to 1.
   Hint: Given an input to *Subset Sum* include two large numbers whose size differs by $S - 2$ where S is the sum of all input numbers.

Answer. Given an input $x = \{x_1, \ldots, x_n\}$ to the Subset sum problem. Let $S = x_1 + \ldots + x_n$. Add two numbers $y = 10S$ and $z = 11S - 2$. Now, we give $f(x) = \{x_1, \ldots, x_n, y, z\}$ as an input to the Number Partition problem. Now, we prove the x is a yes answer to subset sum if and only if $f(x)$ is a yes answer to number partition.

Forward Direction: If x is a yes instance of Subset sum then $f(x)$ is a yes answer of Number partition.
Since x is a yes answer of Subset sum there exists a set $A \subseteq \{x_1, \ldots, x_n$ such that $\sum_{x_i \in A} x_i = 1$. Then, we claim $\{z \cup A, y \cup (\{x_1, \ldots, x_n\} - A)\}$ is a yes instance of Number partition. It is enough to see

$$z + \sum_{x_i \in A} x_i = z + 1 = 11S - 1 = 10S + (S - 1) = y + \sum_{x_i \notin A} x_i$$

where the last identity uses that $x_1 + \ldots + x_n = S$ and $\sum_{x_i \in A} x_i = 1$.

Backwards Direction: If $f(x)$ is a yes instance of Number Partition, then x is a yes instance of Subset sum.
Since $f(x)$ is a yes instance of Number partition there exists a partition A, B of $\{x_1, \ldots, x_n, y, z\}$ such that the numbers in A, B sum up to the same amount. Since $x_1 + \ldots + x_n + y + z = 22S - 2$ the numbers in A must sum up to $11S - 1$ and similarly numbers in B must sum up to $11S - 1$. Therefore, y belongs to one of A, B and z to the other one. Without loss of generality suppose $z \in A$ and let $C = A - \{z\}$. Then, $\sum_{x_i \in C} x_i = 11S - 1 - z = 1$. Therefore, the answer to the Subset Sum problem is yes.

8. (10 points) The *two processor* interval scheduling problem takes as input a sequence of request intervals $(s_1, f_1), \dots, (s_n, f_n)$ just like the unweighted interval scheduling problem except that it produces *two* disjoint subsets $A_1, A_2 \subset [n]$ such that all requests in A_1 are compatible with each other and all requests in A_2 are compatible with each other and $|A_1 \cup A_2|$ is as large as possible. (A_1 might contain requests that are incompatible with requests in A_2.) Does the following greedy algorithm produce optimal results? If yes argue why it does; if no produce a counterexample.

```
Sort requests by increasing finish time
$A_1 = \emptyset$
$A_2 = \emptyset$
while there is any request $(s_i, f_i)$ compatible with either $A_1$ or $A_2$ do
    Add the first unused request, if any, compatible with $A_1$ to $A_1$.
    Add the first unused request, if any, compatible with $A_2$ to $A_2$.
end while
```

Answer. Greedy is not optimal in this case. Consider the following example: $(1, 2), (3, 4), (1, 5)$ These jobs are sorted by increasing finish time. The optimum solutions sends $(1, 2), (3, 4)$ the first machine and $(1, 5)$ to the second machine so it schedules all jobs.

The above Greedy algorithm first allocates $(1, 2)$ to A_1, then it allocates $(3, 4)$ to A_2. Then it cannot allocate $(1, 5)$ to either of the machines.