FINAL EXAM SOLUTIONS
AMTH 377/COEN 279
DESIGN AND ANALYSIS OF ALGORITHMS
SPRING 2012

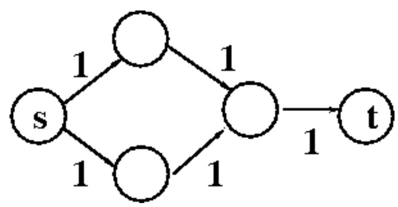
1. (10 points) True or false:

   If all edge capacities in a flow network are the same, then the maximum flow value is *outdegree*(s), the number of edges starting at the source.

   If true, give a general proof; if false, give one counter-example.

   Answer:

   False. The maximum flow value of the network below is 1, while the outdegree of the source is 2.



2. (10 points) A town consists of n houses built along a highway at positions p_1, p_2, ..., p_n. We want to install wireless routers to service these houses, assuming that a router placed at position x can service houses located in the interval $[x-1, x+1]$.

   Write a *greedy* algorithm `router_placement(p[1..n])` that takes an array of house positions and outputs a *shortest* list of positions where wireless routers should be placed in order to service all of the houses.

   Prove that your algorithm is correct.

   Answer:

   In the following, assume that the locations are in sorted order: $p_1 \leq p_2 \leq \cdots \leq p_n$.

   Proposition 1. *There is an optimal solution with a router positioned at $p_1 + 1$.*

   Proof. Suppose $r_1 < r_2 < \cdots < r_k$ is a shortest list of locations of wireless routers that can service all the houses. Since this is an optimal list, the router at r_1 must service house p_1, and hence $r_1 \in [p_1 - 1, p_1 + 1]$. Since there are no houses in $[p_1 - 1, p_1)$, relocating the router at r_1 to $p_1 + 1$ does not affect coverage. □

   The above observation leads to the greedy strategy of always placing a router at one unit to the right of the location of the leftmost house:

```
router_placement(p[1..n])
{
    sort(p);
    L = empty list;
    covered_right_edge = -infinity;
    for (i = 1; i <= n; ++i)
    {
       if (p[i] > covered_right_edge)
       {
           L += {p[i] + 1};
           covered_right_edge = p[i]+2;
       }
    }
    return L;
}
```

3. (10 points) Modify the dynamic-programming algorithm `coin_change(n, d[1..m])` for the coin change problem to output an optimal set of coins instead of just its size. The running time of your algorithm should remain $\Theta(mn)$.

Answer:

Again, we use an array of size n to store the optimal answers for amount 1, 2, ..., n. Each array element has two fields: `s`, which is the *size* of the optimal answers as before, and `d`, which is the denomination of the first coin in an optimal answer.

```
coin_change(n, d[1..m])
{
    a[0].s = 0;

    for (i = 1; i <= n; ++i)
    {
         a[i].s = infinity;
         for (j = 1; j <= m; ++j)
         {
             remainder = i - d[j];
             if (remainder >= 0)
             {
                 if (1 + a[remainder].s < a[i].s)
                 {
                     a[i].s = 1 + a[remainder].s;
                     a[i].d = d[j];
                 }
             }
          }
    }

    x = n;
    while (x > 0)
    {
         output a[x].d;
         x -= a[x].d;
    }
}
```

4. (10 points) Give a polynomial-time algorithm for HAMILTONIAN PATH *when the input graph is a tree*:

   INPUT: a tree $T = (V, E)$;

   OUTPUT: yes iff there is a path going through each vertex of T exactly once.

   Answer: A Hamiltonian path itself is a tree, so if a tree has a Hamiltonian path, the two must be identical. Hence it suffices to check that the input graph is a tree, and that there are 2 vertices of degree 1 and $n-2$ vertices of degree 2.

```
tree_hp(V, E)
{
    if (!is_tree(V, E))
        return false;

    c1 = c2 = 0;
    for each vertex v in V
    {
       if (deg(v) == 1)
          ++c1;
       else if (degree(v) == 2)
          ++c2;
       else
          return false;
    }
    return (c1 == 2 && c2 == n-2);
}
```

   The algorithm runs in time $\Theta(n+m)$.

5. (10 points) Show that the following problem P5 is NP-complete using PARTITION:

INPUT: a set of n positive integers $S = \{a_1, a_2, \ldots, a_n\}$;

OUTPUT: yes iff there are subsets S_1, S_2 of S such that $S_1 \cup S_2 = S$, $S_1 \cap S_2 = \emptyset$, and $|\sum_{x \in S_1} x - \sum_{x \in S_2} x| = 1$.

Answer:

Proposition 2. P5 *is in NP.*

Proof.

```
verify-p5(S[1..n], C[1..n])
{
    s1 = s2 = 0;
    for (i = 1; i <= n; ++i)
    {
       if (C[i] == 1)
          s1 += S[i];
       else
          s2 += S[i];
    }
    return (abs(s1 - s2) == 1);
}
```

The algorithm runs in time $\Theta(n)$. □

Proposition 3. PARTITION $\leq$ P5.

Proof. We use the following reduction, which runs in time $\Theta(n)$:

```
f(S[1..n])
{
    for (i = 1; i <= n; ++i)
        T[i] = 3*S[i];

    T[n+1] = 1;

    return T[1..n+1];
}
```

In the following, we say that X and Y is a c-partition of Z if $X \cup Y = Z$, $X \cap Y = \emptyset$, and $|\sum_{x \in X} x - \sum_{y \in Y} y| = c$. Also, if X is a set, then let $sum(X) = \sum_{x \in X} x$, and $k * X$ be the set $\{kx : x \in X\}$.

First, if S_1 and S_2 is a 0-partition of S, then $3 * S_1 \cup \{1\}$ and $3 * S_2$ is a 1-partition of T.

Conversely, suppose T_1 and T_2 is a 1-partition of T, such that $sum(T_1) = 1 + sum(T_2)$. Then the element 1 must be in T_1, or else $sum(T_1) - sum(T_2) = 2$, which is impossible, since each element in T is a multiple of 3. Hence $sum(T_1 - \{1\}) = sum(T_2)$, and hence $(T_1 - \{1\})/3$ and $T_2/3$ is a 0-partition of S.

□