**CPSC-320 Intermediate Algorithm Design and Analysis Winter 2000**

**Solutions to Practice Homework # 6**

1. (a) To show that the Class Scheduling Problem (CSP) is in NP, we describe a polynomial time verification algorithm, $V$, for the problem. $V$ takes as input an instance of CSP and a witness $y$. The instance is of the form $< 1, 2, \ldots, n, C_1, C_2, \ldots, C_s, b >$ where $n$ is the number of courses, $C_i \subseteq \{1, 2, \ldots, n\}$ is the set of courses that student $i$ wishes to take, $1 \leq i \leq s$, and $b \leq n$ is the bound on the number of courses that can be offered.

   The verification algorithm checks that (i) that $B$ is a set of courses of size at most $b$ and (ii) for each $C_i$, there is at least one course in $y$ that is also in $C_i$ (i.e. that at least one course on each student's list is offered). If so, the algorithm outputs "yes", else it outputs "no".

   The algorithm can be implemented to run in polynomial time: Check (i) takes time $O(b)$; this is at most linear in the size of the input since $b \leq n$ and the courses are listed explicitly. For each student $i$, a naive implementation of step (ii) that compares each course listed in $C_i$ with each course in $y$ takes time proportional to the size of $C_i$ times the size of $y$, and thus at most the square of the input size. (Can you figure out how to do that step in less than quadratic time?)

   Correctness: Note that if $x$ is a "yes" instance of the CSP, then when $y$ is the set $B$ of courses that satisfies the students, the verification algorithm outputs "yes" on the inputs $x, y$. But if $x$ is a "no" instance, no witess $y$ causes the verification algorithm to accept.

   (b) To complete the proof of NP-completeness, we next show that the Vertex Cover Problem (VCP) can be reduced to the CSP. Let $(G, k)$ be an instance of the vertex cover problem. We map $(G, k)$ to an instance of CSP as follows. The list of courses of the CSP is the list of nodes $1, 2, \ldots, n$ of $G$. The number of students equals the number of edges of $G$. The courses that a student $e$ is interested in are the two endpoints of $e$ in the graph $G$. Finally, the bound $b$ is set to equal $k$. The mapped instance is therefore $< 1, 2, \ldots, n, E_1, E_2, \ldots, E_s, b >$ where $b = k$, $s$ is the number of edges of $G$, and $E_i$ is the set of two endpoints of the $i$th edge of $G$.

   This reduction can be computed in polynomial time (details omitted).

   Correctness: Suppose that $(G, k)$ is a "yes"-instance of VCP. In this case, there is a set of nodes, call it $B$, which is of size $k$ that covers all edges of the graph. That is, every edge of the graph has an endpoint in $B$. Therefore, in the mapped instance, $B$ is a set of courses that satisfies all students and has size $b$.

   Conversely, suppose that $< 1, 2, \ldots, n, E_1, E_2, \ldots, E_s, b >$ is a "yes" instance of the CSP. Then there is a set $B$ of courses that satisfies all students, i.e. such that one course is in each set $E_i$. Therefore, the set $B$ corresponds to a set of nodes that covers all edges in $G$. Therefore $G$ has a vertex cover of size $b = k$ and so $(G, k)$ is a "yes" instance of vertex cover.

2. (i) The problem $\{< G, k > \mid G$ has a minimum spanning tree whose total cost is at most $k\}$ is in P. (The Bellman-Ford algorithm can solve the problem in polynomial time.)

   (ii) The Clique problem is in the class of NP-complete problems.

   (iii) Any problem in P is in NP and is not NP-Complete unless the unlikely event NP = P is true. Factoring is another important example of a problem that appears not to be NP-complete, but doesn't seem to be in P either.

3.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $\pi$ | 0 | 1 | 0 | 1 | 2 | 3 | 4 | 0 |

4. (a) Here is an algorithm that solves the random permutation problem. We assume that the function *rand(i,j)* returns a random integer in the range i through j inclusive.

```
for i from 1 to n do
    k <-- rand(i,n)
    Swap(A[i], A[k])
```

The running time of the algorithm is linear in $n$.

(b) Correctness: We need to show that each possible permutation of the elements is equally likely. To distinguish between the contents of array $A$ before and after the algorithm, we let $A'[1,\ldots,n]$ be the array after the algorithm. Let $a_1, a_2, \ldots, a_n$ be the elements of the array.

There are $n!$ possible permutations of the elements, so each should be chosen by the algorithm with probability $1/n!$.

Consider the permutation $a_{i_1}, a_{i_2}, \ldots, a_{i_n}$ of the array elements. The probability that $A'[1] = a_{i_1}$ is $1/n$, since the index $k$ of the element to be swapped is chosen uniformly and randomly from the range $1,\ldots,n$, and so each elements $a_i$ is equally likely to be chosen.

Now, given that $A'[1] = a_{i_1}$, what is the probability that $A'[2] = a_{i_2}$? The elements $a_{i_2}, a_{i_3}, \ldots, a_{i_n}$ are in positions 2 through $n$ of $A$ after the second step of the algorithm. The index $k$ of the position to be swapped is now chosen randomly and uniformly from the range $2,\ldots,n$, and so $a_{i_2}$ is chosen with probability $1/(n-1)$.

Similarly, given that the first $j$ elements in the permuted array $A'$ are $a_{i_1}, a_{i_2}, \ldots, a_j$, the probability that $a_{j+1}$ is chosen to be the $(n-j+1)$st element of the permuted array at step $j+1$ of the algorithm is $1/(n-j+1)$.

The probability that $a_{i_1}, a_{i_2}, \ldots, a_{i_n}$ is the output is the product of the probabilities above, namely

$$\frac{1}{n}\frac{1}{/}n-1\ldots\frac{1}{2}\frac{1}{1}$$

which is $1/n!$, as desired. Hence, the algorithm is correct.

5. See the Cormen et al. text, page 186, for notes on the solution to this one. Briefly, if we count comparisons of array elements, we have that $T(n) = 2T(n/2) + 2$ for $n$ even, and $T(2) = 1$. Solving this, we get that $T(n) = 3n/2 - 2$.

   If the algorithm divides the array into three parts instead of two, it is as follows:

```
function find-max-min(i,j)
   {returns a pair (Max, Min)}
   if i+2 = j then using three comparisons, return the max and min elements

   else k <-- (j+i-1)/3
      (MaxL, MinL) <-- find-max-min(i,k)
      (MaxM, MinM) <-- find-max-min(k+1,2k)
      (MaxR, MinR) <-- find-max-min(2k+1,j)

      if MaxL > MaxM then
         if MaxL > Max R then Max <-- MaxL else Max <-- MaxR
         else if MaxM > Max R then Max <-- MaxM else Max <-- MaxR
      if MinL > MinM then
         if MinL > Min R then Min <-- MinL else Min <-- MinR
         else if MinM > Min R then Min <-- MinM else Min <-- MinR
   return (Max,Min)
```

   Now, the recurrence is $T(n) = 3T(n/3) + 4$, and $T(3) = 3$. The solution to this is $T(n) = 2n - 2$.