Due: beginning of class on Tue. May 7, 2002

Problem 1

Consider the problem of multiplying a matrix $A \in R^{n \times n}$ by a vector $v \in R^n$. (Assume that entries are stored as floating point numbers, and arithmetic operations can be performed in constant time.) The running time of the standard matrix-vector multiplication algorithm is $O(n^2)$. In this problem you are asked to devise a faster solution when matrix $A$ has a special form.

A circulant matrix is a square matrix $A$ such that every row of $A$ is obtained by cyclically rotating the previous row by one position. In other words, if the first row is $[a_1, \ldots, a_n]$, then the second row is $[a_n, a_1, a_2, \ldots, a_{n-1}]$, the third row is $[a_{n-1}, a_n, a_1, \ldots, a_{n-2}]$, and so on. Circulant matrices are important because they can be represented using only $O(n)$ storage (e.g., giving only the first row), and are used for example in cryptographic applications.

Show that if $A$ is a circulant matrix, then the product $Av$ can be computed in $O(n \log n)$ time.

Problem 2

In the subset-sum problem, one is given a list of positive integers $a_1, \ldots, a_n$ and a target value $b$, and the task is to decide if there is a subset of the $a$’s that add up to $b$. Equivalently, you want to determine if there is a $0-1$ sequence $x_1, \ldots, x_n$ such that $\sum a_ix_i = b$.

The subsetsum is a very hard problem, and no efficient algorithm is known for the general case. (In fact, subsetsum is NP-complete, so no polynomial time algorithm is likely to exist.) However, there are some special cases that can be solved in polynomial time. Define the density of a subsetsum problem as the quantity

$$\delta = \frac{n}{\log \max_i a_i}.$$

Prove that if the density is $\delta = \Omega(n/ \log n)$, then subsetsum can be solved in polynomial time. (Another case that can be solved in polynomial time is when the density is very small $\delta = O(1/n)$, but the solution is much more complicated. Ask Daniele for references if you are interested.)

Problem 3

Consider the following covering problem. Let $T$ be a tree with $n$ nodes (e.g., represented by an array $P[1, \ldots, n]$ where $P[i] = j$ if node $j$ is the parent of $i$, and $P[i] = i$ if $i$ is the root of the tree.) We say that a vertex $v$ cover an edge $e$ if $v$ is one of the end-points of $e$. The problem is, given a tree $T$, find as small a subset of vertices $S \subseteq \{1, \ldots, n\}$ as possible,
such that $S$ cover all edges, i.e., every edge $(i, p[i])$ in the tree is covered by at least one vertex in $S$.

Give as efficient an algorithm as possible to solve the problem above. Your solution should include a detailed description of how the tree is represented and manipulated in order to get a fast implementation.

**Problem 4**

Consider the following videogame, played on an $h \times w$ screen. At the beginning of the game you are moving on an aeroplane on the top row of the screen (say, row $h$). Any time (say, when you are at position $(h, x)$) you can jump off the plane (with your parachute of course.) At this point you will start moving down, from row $h - 1$ down to the ground at row 1. As you move from row $r$ to row $r - 1$, you can either stay on the same column, move one column to the left (unless you are on column 1) or move one column to the right (unless you are on column $w$).

There is a number of balloons floating in the air, and as you encounter them (say, you are at the same position of the balloon, the position immediately to the left, or the one immediately to the right), you can blow them up. The goal is to blow up as many balloons as possible.

Give an efficient algorithm that on input an array with $A[i, j] = 1$ if there is a balloon at position $(i, j)$, and $A[i, j] = 0$ otherwise, find the optimal playing strategy, i.e., the maximum number of balloons you can blow up, and the corresponding sequence of moves.

Assume that the top row does not contain any balloon, and you can blow up multiple balloons at the same time.

**Problem 5**

Give a recursive algorithm to compute binomial coefficient \( \binom{n}{k} \) based on the recurrence formula

\[
\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.
\]

(See Appendix C in the book for definition of binomial coefficients.)

The straightforward recursive implementation of this algorithm is not polynomial time. Transform your recursive procedure into a dynamic programming algorithm first using memoization, and then giving a bottom-up iterative solution. What is the asymptotic running time of the two dynamic programming solutions?

Compare the running times of your two implementations running tests on input of the form $(2k, k)$, and comment your results. Are the running times comparable? Do they match the theoretical analysis? Which implementation is faster? Why?