CSE 102
Homework Assignment 5

1. (This is a continuation of problem 5 on hw assignment 4, which was itself based on problem 8-5 on page 207 of CLRS 3rd edition.) Recall an array $A[1 \cdots n]$ is k-sorted if and only if the $(n-k+1)$ consecutive k-fold averages of $A[1 \cdots n]$ are sorted:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k}$$

for $1 \leq i \leq n-k$. Observe that if $n = k$, then any array of length n is k-sorted, since there is only one k-fold average in this case. We may also *define* any $A[1 \cdots n]$ to be k-sorted if $n < k$, since in this case there are no k-fold averages. Modify Quicksort to produce an algorithm that k-sorts an array of any length. Write your algorithm in pseudo-code and call it $\text{Quick}[k]\text{sort}()$. Specifically, the call $\text{Quick}[k]\text{sort}(A, 1, n)$ will k-sort $A[1 \cdots n]$, but will not necessarily $(k-1)$-sort the same array. Prove the correctness of your algorithm, and analyze its worst case run time.

2. (This is a re-wording of problem 4.2-4 on page 82 of CLRS 3rd edition.)
Determine the largest integer k such that if there is a way to multiply 3×3 matrices using k multiplications (not assuming commutativity of multiplication of matrix elements), then there is an algorithm to multiply $n \times n$ matrices (where n is an exact power of 3) in time $o\left(n^{\lg(7)}\right)$. What would the run time of this algorithm be? (Hint: Proceed as in the analysis of Strassen's algorithm, but recur on submatrices of size $\frac{n}{3} \times \frac{n}{3}$.)

3. (This is a generalization of problem 4.2-5 on page 82 of CLRS 3rd edition.)
Suppose there is a method for multiplying $m \times m$ matrices using only k multiplications (not assuming commutativity of multiplication of matrix elements), where $k < m^3$. Explain how to recursively multiply $n \times n$ matrices (where n is an exact power of m) in time $o(n^3)$ by using this method as a subroutine. What is the runtime of your algorithm? (Hint: Imitate Strassen's algorithm.)

4. Read the Coin Changing Problem in the handout on Dynamic Programming. Its solution is presented below for reference. Recall this algorithm assumes that an unlimited supply of coins in each denomination $d = (d_1, d_2, \dots, d_n)$ are available.

```
CoinChange(d, N)
1.  n = length[d]
2.  for i = 1 to n
3.      C[i, 0] = 0
4.  for i = 1 to n
5.      for j = 1 to N
6.          if i = 1 and j < d[1]
7.              C[1, j] = ∞
8.          else if i = 1
9.              C[1, j] = 1 + C[1, j − d[1]]
10.         else if j < d[i]
11.             C[i, j] = C[i − 1, j]
12.         else
13.             C[i, j] = min(C[i − 1, j], 1 + C[i, j − d[i]])
14. return C[n, N]
```

Write a recursive algorithm that, given the filled table $C[1 \cdots n; 0 \cdots N]$ as input, prints a sequence of $C[n, N]$ coin values whose total value is N. If it happens that $C[n, N] = \infty$, print a message to the effect that no such disbursement of coins is possible.

5. Read the Discrete Knapsack Problem in the handout on Dynamic Programming. A thief wishes to steal *n* objects having values $v_i > 0$ and weights $w_i > 0$ (for $1 \le i \le n$). His knapsack, which will carry the stolen goods, holds at most a total weight $W > 0$. Let $x_i = 1$ if object *i* is to be taken, and $x_i = 0$ if object *i* is not taken ($1 \le i \le n$). The thief's goal is to maximize the total value $\sum_{i=1}^{n} x_i v_i$ of the goods stolen, subject to the constraint $\sum_{i=1}^{n} x_i w_i \le W$.

   a. Write pseudo-code for a dynamic programming algorithm that solves this problem. Your algorithm should take as input the value and weight arrays $v[\,]$ and $w[\,]$, and the weight limit *W*. It should generate a table $V[1 \cdots n; 0 \cdots W]$ of intermediate results. Each entry $V[i, j]$ will be the maximum value of the objects that can be stolen if the weight limit is *j*, and if we only include objects in the set $\{1, \dots, i\}$. Your algorithm should return the maximum possible value of the goods which can be stolen from the full set of objects, i.e. the value $V[n, W]$. (Alternatively you may write your algorithm to return the whole table.)

   b. Write an algorithm that, given the filled table generated in part (a), prints out a list of exactly which objects are to be stolen.