CMPS 102
Homework Assignment 5
Solutions

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:

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

for $1 \le i \le n-k$. 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 Quick[k]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.

Solution:
The key to this problem is the observation that any array $A[1\cdots n]$ is k-sorted, if $n \le k$. We simply run the Quicksort algorithm, but only recur down to array size k, or smaller.

```
Quick[k]sort(A, p, r)
1. if $k < r - p + 1$
2.     $q = \text{Partition}(A, p, r)$
3.     $\text{Quick}[k]\text{sort}(A, p, q-1)$
4.     $\text{Quick}[k]\text{sort}(A, q+1, r)$
```

Proof of correctness:
We show by induction on $m = r - p + 1$, that $\text{Quick}[k]\text{sort}(A,p,r)$ correctly k-sorts any subarray $A[p\cdots r]$ of length $m \ge 1$.

I. The base cases $m = 1, 2, \ldots, k$ follow from the observations made in the problem statement, i.e. there is at most one k-fold average, and hence the consecutive k-fold averages cannot be out of order.

II. Let $m > k$ and assume that $\text{Quick}[k]\text{sort}$ correctly k-sorts any subarray of length less than m. We must show that $\text{Quick}[k]\text{sort}(A,p,r)$ correctly k-sorts the subarray $A[p\cdots r]$ of length $m = r - p + 1$. Since $m > k$ the test on line 1 is true, and lines 2, 3 and 4 are executed. The call to $\text{Partition}(A,p,r)$ on line 2 rearranges $A[p\cdots r]$ and returns an integer q such that $p \le q \le r$ and

(*) $$A[p\cdots(q-1)] \le A[q] < A[(q+1)\cdots r]$$

holds. Since both

$$\text{length}(A[p\cdots(q-1)]) = (q-1) - p + 1 = q - p \le r - p < r - p + 1 = m$$

and

$$\text{length}(A[(q+1)\cdots r]) = r - (q+1) + 1 = r - q \le r - p < r - p + 1 = m,$$

the induction hypothesis guarantees that $A[p \cdots (q-1)]$ and $A[(q+1) \cdots r]$ are k-sorted after the recursive calls on lines 3 and 4 are executed. We show that this fact, together with (*), implies the subarray $A[p \cdots r]$ is now k-sorted.

The result of problem 5c on hw4 says that $A[p \cdots r]$ is k-sorted if and only if $A[i] \le A[i+k]$ for all i in the range $p \le i \le r-k$. We have 3 cases to consider.

(1) $p \le i < i+k \le q-1$
In this case, the fact that $A[p \cdots (q-1)]$ is k-sorted implies $A[i] \le A[i+k]$.

(2) $q+1 \le i < i+k \le r$
Then $A[(q+1) \cdots r]$ being k-sorted implies $A[i] \le A[i+k]$.

(3) $p \le i < q \le i+k \le r$
In this case, the inequality (*) itself implies $A[i] \le A[i+k]$.

In all cases, $A[i] \le A[i+k]$ for all i in the range $p \le i \le r-k$, and hence the subarray $A[p \cdots r]$ is k-sorted, as required.

The result follows for all $m \ge 1$ by the 2nd Principle of Mathematical Induction. ■

Analysis of worst case runtime
The recurrence for worst case number of array comparisons is very similar to that for ordinary Quicksort, the only difference being where (for which n) the recurrence halts. As before, the worst case behavior is elicited when one of the two subarrays on lines 3 or 4 is empty.

$$T(n) = \begin{cases} 0 & 1 \le n \le k \\ T(n-1) + (n-1) & n > k \end{cases}$$

The iteration method yields $T(n) = \sum_{i=1}^{s}(n-i) + T(n-s)$, stopping at the smallest s for which $1 \le n-s \le k$, that is $s = n-k$. Thus

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-k}(n-i) \\ &= \sum_{i=1}^{n-k} n - \sum_{i=1}^{n-k} i \\ &= n(n-k) - \frac{(n-k)(n-k+1)}{2} \\ &= \Theta(n^2). \end{aligned}$$
■

Remark
The average case runtime of Quick$[k]$sort can be shown to be $\Theta(n\log(n))$, as one would expect. We leave this as a further exercise for the reader. ■

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 $\frac{n}{3} \times \frac{n}{3}$.)

Solution:
We regard an $n \times n$ square matrix (where n is a power of 3) as a 3×3 matrix, each of whose 9 elements is a square submatrix of size $\frac{n}{3} \times \frac{n}{3}$. Thus, to multiply two $n \times n$ square matrices, we need to multiply two 3×3 matrices of matrices. We are to assume that this multiplication can be done by performing only k multiplications of the underlying $\frac{n}{3} \times \frac{n}{3}$ submatrices (which are non-commutative operations). Presumably k is less than the 27 multiplications ordinarily needed for multiplying two 3×3 matrices. However, k must also be at least 9 since that is the number of $\frac{n}{3} \times \frac{n}{3}$ submatrices in the answer, and each requires at least 1 multiplication to compute. Since n is a power of 3, we can recur on this process down to matrices of size 1×1, where the recursion halts and the product of real numbers is returned. At each level then, there are k recursive multiplications of 9 matrices whose size is 1/3rd the size of the current level.

Let $T(n)$ denote the running time of the algorithm described above. Then, in analogy with Strassen's algorithm, we obtain the recurrence relation

$$T(n) = kT(n/3) + \Theta(1),$$

where the first term is the cost of the k recursive calls, and the second term represents the overhead of the current invocation of the algorithm. Case 1 of the Master Theorem gives the asymptotic solution to this recurrence as

$$T(n) = \Theta\left(n^{\log_3(k)}\right).$$

Alternatively, if we included the cost of the matrix additions necessary to compute the product, we would have the recurrence $T(n) = kT(n/3) + \Theta(n^2)$, which has the same asymptotic solution as above (since $k > 9$).

We therefore seek the largest integer k satisfying $n^{\log_3(k)} = o\left(n^{\lg(7)}\right)$. Equivalently $\log_3(k) < \log_2(7)$, and hence

$$k < 3^{\log_2(7)} = 21.849 \ldots$$

We see that $\boldsymbol{k = 21}$, and the run time of the above algorithm is $T(n) = \Theta\left(n^{\log_3(21)}\right)$. ■

Remark:
Observe that nothing in this solution proves, or even suggests, that such an algorithm exists, since we know of no way to multiply two 3×3 matrices of matrices by computing only $k = 21$ submatrix products (and likewise for any other $k < 27$.) All it shows is that if such a method were to exist, we would need $k \leq 21$ to yield an algorithm better than Strassen's. ■

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.)

Solution:
An adequate description of this algorithm is obtained starting with the first paragraph of the preceding solution, and replacing each occurrence of 3 by m (also 9 by m^2, and 27 by m^3). Alternately, we express the algorithm here in (high level) pseudo-code.

Assume the existence of a subroutine called Multiply$[m, k](A, B)$ that returns the product of its $m \times m$ matrix operands A and B by performing sums and products of their respective elements, only k of which are multiplications (and which multiplications are not assumed to be commutative, so the elements can themselves be matrices.) Note here that m and k are not function arguments, but part of the name of the algorithm. Thus Strassen's algorithm explicitly defines Multiply[2,7] (see pages 80-81 of CLRS). The previous problem asks that we assume Multiply$[3, k]$ exists.

We can now define MultiplyRecursive$[m](A, B, n)$ in which A and B are $n \times n$ square matrices and n is an exact power of m.

```
MultiplyRecursive$[m](A, B, n)$
1. let $C$ be a new $n \times n$ matrix
2. if $n > 1$
3.     partition each of $A, B$ and $C$ into $m^2$ submatrices of size $\frac{n}{m} \times \frac{n}{m}$
4.     let $A', B'$ and $C'$ be the corresponding $m \times m$ matrices of matrices
5.     $C' =$ Multiply$[m, k](A', B')$
6.     let the elements of $C$ be the elements of the elements of $C'$
7. else
8.     $C = A \cdot B$  (the product of two real numbers)
9. return $C$
```

Here also, m is part of the name of the algorithm, and not a function argument. Observe this algorithm contains no explicit call to itself. The recursive call is embedded in Multiply$[m, k]$, whose general form follows.

```
Multiply$[m, k](A, B)$
1. Perform some combination of sums and products of the elements of $A$ and $B$, only $k$ of which
   are products (not assumed to commute).
2. Every product in (1) is computed by calling MultiplyRecursive$[m](A, B, n/m)$.
3. Assemble the results into an $m \times m$ matrix of matrices, which is then returned.
```

When expressed in this abstract manner, the runtime $T(n)$ of MultiplyRecursive$[m](A, B, n)$ is seen to satisfy the recurrence

$$T(n) = kT(n/m) + \Theta(1),$$

which, by case 1 of the Master Theorem, has asymptotic solution $T(n) = \Theta\left(n^{\log_m(k)}\right)$. Again, if the cost of additions were included, we would obtain the same asymptotic solution. Observe that since $k < m^3 \Rightarrow \log_m(k) < 3$, we have $T(n) = o(m^3)$ as required. ■

Remarks

The excessively abstract form of the above pseudo-code arises from the fact that the subroutine Multiply$[m, k]$ may not even exist. When it does exist, one would normally write the procedure directly into the code for MultiplyRecursive$[m]$, rather than call a separate function. The recursive branch (lines 3-6) of MultiplyRecursive$[m]$ is therefore best implemented by index calculations alone, rather than by allocating new matrices. If n is not an exact power of m, one can simply pad both A and B with zeros until they are of size $N \times N$, where N is the smallest power of m that is greater than or equal to n, i.e. $N = m^{\lceil \log_m(n) \rceil}$. One can show $N^{\log_m(k)} = \Theta\left(n^{\log_m(k)}\right)$, so nothing is lost in asymptotic run time by augmenting the matrices in this way. (This is essentially the content of Problem 4.2-3 on page 82, with $m = 2$). It's even possible to avoid allocating larger matrices altogether by restricting calculations to only the non-zero parts of the augmented matrices. This same augmentation procedure can be used to multiply non-square matrices. Verifying all of these facts is an interesting exercise, left to the reader.

Problem 4.2-5 informs us that in 1978, Victor Pan devised methods as described above for the following (m, k) pairs: $(68, 132464)$, $(70, 143640)$ and $(72, 155424)$. Note that in all cases, k is much less than m^3. Observe $\log_{68}(132464) = 2.795128\ldots$, $\log_{70}(143640) = 2.795122\ldots$ and $\log_{72}(155424) = 2.795147\ldots$, each of which is smaller than $\log_2(7) = 2.807354\ldots$. Thus Pan's algorithms are (slight) improvements over Strassen's, asymptotically speaking. This is just part of the long line of research started by Strassen's amazing discovery. An interesting discussion of that history can be found in https://theory.stanford.edu/~virgi/matrixmult-f.pdf. ■

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.  $n = \text{length}[d]$
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.              $C[1, j] = \infty$
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 $C[1 \cdots n; 0 \cdots N]$ generated by the above algorithm, prints out a sequence of $C[n, N]$ coin types which disburse *N* monetary units. In the case $C[n, N] = \infty$, print a message to the effect that no such coin disbursal is possible.

Solution:
Note that array $d = (d_1, d_2, \dots, d_n)$ is needed as input in order to navigate the table *C*.

```
PrintCoins(C, d, i, j)    Pre: $C[1 \cdots n; 0 \cdots N]$ was filled by CoinChange(d, N)
1.  if $j > 0$
2.      if $C[i, j] == \infty$
3.          print("cannot pay amount ", $j$)
4.          return
5.      if $i == 1$
6.          print("pay one coin of value ", $d_1$)
7.          PrintCoins($C, d, 1, j - d_1$)
8.      else if $j < d_i$
9.          PrintCoins($C, d, i - 1, j$)
10.     else  // both $i > 1$ and $j \geq d_i$
11.         if $C[i, j] == C[i - 1, j]$
12.             PrintCoins($C, d, i - 1, j$)
13.         else  // $C[i, j] == 1 + C[i, j - d_i]$
14.             print("pay one coin of value ", $d_i$)
15.             PrintCoins($C, d, i, j - d_i$)
```

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.)

      Solution:

```
Knapsack($v, w, W$)  (pre: $v[1 \cdots n]$ and $w[1 \cdots n]$ contain positive numbers)
1.  $n = \text{length}[v]$
2.  for $j = 0$ to $W$   // fill in first row
3.      if $j < w_1$
4.          $V[1, j] = 0$
5.      else
6.          $V[1, j] = v_1$
7.  for $i = 2$ to $n$   // fill remaining rows
8.      for $j = 0$ to $W$
9.          if $j < w_i$
10.             $V[i, j] = V[i-1, j]$
11.         else
12.             $V[i, j] = \max(\, V[i-1, j],\ v_i + V[i-1, j-w_i]\, )$
13. return $V[n, W]$
```

   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.

      Solution:

```
PrintObjects($V, w, i, j$)  (pre: $V[1 \cdots n; 0 \cdots W]$ was filled by Knapsack($v, w, W$))
1.  if $i == 1$
2.      if $j < w_i$
3.          print("do not include object ", 1)
4.      else
5.          print("include object ", 1)
6.  else  // $i > 1$
7.      if $V[i, j] == V[i-1, j]$
8.          PrintObjects($V, w, i-1, j$)
9.          print("do not include object ", $i$)
10.     else  // both $j \ge w_i$ and $V[i, j] == v_i + V[i-1, j-w_i]$
11.         PrintObjects($V, w, i-1, j-w_i$)
12.         print("include object ", $i$)
```