[Total points: 33]

Part 1: Dynamic Programming. *In your solution to any of these problems you are required to give a recursive formula that indicates how a problem instance reduces to one or more smaller instances. This accounts for 3 of the 6 points. There are 2 points for the table filling algorithm (really a program) and its time bound, and 1 point for the algorithm that retrieves the actual choices.*

1. [worth 6 points]

[From Baase and Van Gelder, Computer Algorithms, Addison Wesley, 2000]

Suppose you have three strings $X = x_1 \cdots x_m$, $Y = y_1 \cdots y_n$, and $Z = z_1 \cdots z_{m+n}$. String Z is said to be a *shuffle* of X and Y if it can be formed by interspersing characters of X and Y while maintaining the original order of each string. For example `blmuondeasy` and `mbolnudeasy` are shuffles of `monday` and `blues`, but `blmuondeysa` is not (the a and y of `monday` are in the worng order). Give an algorithm that determines, given inputs X, Y, and Z, whether or not Z is a shuffle of X and Y. Illustrate how your algorithm works on $X =$ `my`, $Y =$ `dog` and $Z =$ `domyg`; also do this with $Z =$ `dmogy`. What is the asymptotic runtime of your algorithm as a function of m and n?

Solution: Let $S(i,j) =$ ***true*** if $z_1 \cdots z_{i+j}$ is a shuffle of $x_1 \cdots x_i$ and $y_1 \cdots y_j$. Then

$S(i,j) = $ ***true*** if $i = 0$ **or** $j = 0$ – a string is a shuffle of itself with the empty string

$S(i,j) = $ ***true*** if $S(i-1,j)$ **and** $x_i = z_{i+j}$
or if $S(i,j-1)$ **and** $y_j = z_{i+j}$
– either $z_1 \ldots z_{i+j-1}$ adds x_i or it adds y_j to becomes a longer shuffle

$S(i,j) = $ ***false*** otherwise

The table filling algorithm is straightforward – see Fig. 1. Runtime is obviously $\Theta(mn)$.

for $i \leftarrow 1$ **to** m **do**
    $S[i,0] \leftarrow$ ***true***
end do
for $j \leftarrow 1$ **to** n **do**
    $S[0,j] \leftarrow$ ***true***
end do
for $i \leftarrow 1$ **to** m **do**
    for $j \leftarrow 1$ **to** n **do**
        $S[i,j] \leftarrow (S[i-1,j]$ **and** $x_i = z_{i+j})$ **or** $(S[i,j-1]$ **and** $y_j = z_{i+j})$
    end do
end do

Figure 1: Algorithm to determine if $Z = z_1 \cdots z_{m+n}$ is a shuffle of $x_1 \cdots x_m$ and $y_1 \cdots y_n$.

The recursive algorithm in Fig. 2 determines, for each symbol of Z, whether it comes from X or from Y. Let $C[k]$ be X if $z_k = x_i$ for some i and let it be Y if $z_k = y_j$ for some j. It is initially called as BelongsTo(m,n).

For the strings $X =$ `my`, $Y =$ `dog` and $Z =$ `domyg`, here's the order in which the table is filled.

$S[0,0] \leftarrow$ ***true***
$S[1,0] \leftarrow$ ***false*** $x_1 \neq z_1$

[Illustration on the example omitted – for now.]

```
procedure BelongsTo(i, j)
    if i = 0 and j = 0 then return endif
    if S[i − 1, j] and x_i = z_{i+j} then
        C[k] ← X
        BelongsTo(i − 1, j)
    else if S[i, j − 1] and y_i = z_{i+j} then
        C[k] ← Y
        BelongsTo(i, j − 1)
    else *error*
    endif
end BelongsTo
```

Figure 2: Algorithm to determine which string each symbol of a shuffled string comes from.

2. [worth 6 points]

[Also from Baase and Van Gelder]

Suppose you have inherited the rights to 500 previously unreleased songs recorded by the popular group Raucous Rockers. You plan to release a set of 5 CD's (numbered 1 through 5) with a selection of these songs. Each disk holds a maximum of 60 minutes of music and each song must appear in its entirety on one disk. Since you are a classical music fan and have no way of judging the artistic merits of the songs, you decide to use the following criteria: the songs will be recorded in order by the date they were written, and the number of songs included will be maximized. Suppose you have a list $\ell_1, \ell_2, \ldots, \ell_{500}$ of the lengths of the songs, in order by the date they were written (no song is more than 60 minutes long). Give an algorithm to determine the maximum number of songs that can be included using the given criteria. Hint: Let $T(i, j)$ be the minimum amount of time needed for recording any i songs from among the first j songs. You should interpret T to include blank time at the end of a completed disk. For example, if a selection of songs uses all of the first disk plus 15 minutes of the second disk, the time for that selection should be 75 minutes even if there is blank space at the end of the first disk. Generalize your solution to a situation where there are m CD's, n songs, and a capacity c for each of the CD's. What is the asymptotic runtime in terms of m and n (treating c as a constant)?

Solution: The following describes the more general solution. For the specific one that applies to the Raucus Rockers, let $m = 5$, $n = 500$, and $c = 60$. First, note that based on the $T(i, j)$ of the hint, the maximum number of songs that can be recorded is the maximum value of i for which $T(i, n) \leq cm$. In order to simplify the formula for $T(i, j)$ define $R(i-1, j-1) =$ the extra space on the last CD when the songs that obtain $T(i-1, j-1)$ are chosen and let $\ell_{i,j}$ be ℓ_j if the j-th song fits into that extra space and $\ell_j + R(i-1, j-1)$ if it doesn't. Then

$$\begin{array}{rcll} R(i-1,j-1) & = & c\lceil T(i-1,j-1)/c\rceil - T(i-1,j-1) & \\ \ell_{i,j} & = & \ell_j & \text{if } \ell_j \leq R(i-1,j-1) \\ & & \ell_j + R(i-1,j-1) & \text{otherwise} \end{array}$$

Now

$$\begin{array}{rcll} T(i,j) & = & 0 & \text{if } i = 0 \text{ or } j = 0 \\ T(i,j) & = & \text{min of the following:} & \\ & & T(i,j-1) & \text{don't choose the } j\text{-th song} \\ & & T(i-1,j-1) + \ell_{i,j} & \text{choose the } j\text{-th song – adds } \ell_{i,j} \text{ to the total time.} \end{array}$$

The table filling algorithm is as expected – see Fig. 3 – and has runtime $\Theta(n^2)$. Note that m does not have any impact. To reconstruct the solution and count the number of songs that will fit, the array element $X[i, j] =$ ***true*** if and only if the j-th song is included in the solution that is used to obtain $T[i, j]$.

```
for i ← 1 to n do
    T[i,0] ← 0
end do
for j ← 1 to n do
    T[0,j] ← 0
end do
for i ← 1 to n do
    for j ← 1 to n do
        r ← c(1 + int(T[i−1,j−1]/c)          ▷ r = R(i−1,j−1)
        if ℓ_j ≤ r then ℓ ← ℓ_j else ℓ ← ℓ_j + r endif
        if T[i,j−1] < T[i,j−1] + ℓ then
            T[i,j] ← T[i−1,j−1];        X[i,j] ← false
        else
            T[i,j] ← T[i−1,j−1] + ℓ;    X[i,j] ← true
        endif
    end do
end do
```

Figure 3: Algorithm to determine minimum amount of space required to include i songs from among the first j.

```
function RECOVER(i,j) is
    if i = 0 or j = 0 then return [ ]      ▷ empty list
    ▷ if i songs don't fit, try i − 1
    else if T[i,j] > cm then return RECOVER(i−1,j)
    ▷ if the j-th song was chosen, add it to the list
    else if X[i,j] then return j + RECOVER(i−1,j−1)
    ▷ otherwise, there are i songs among the first j−1 that will fit
    else return RECOVER(i,j−1)
end RECOVER
```

Figure 4: Algorithm to recover the list of songs to be included on m CD's with c minutes each.

Recovering the actual songs that make up the minimum number can be done using the recursive algorithm in Fig. 4. It returns a list of the (indices) of the i songs to be included among the first j assuming that there is a limit of m CD's of length c minutes. The length of that list is the maximum number of songs.

3. [worth 7 points – one extra because the time bound requires additional analysis]

Suppose n jobs, numbered $1, \ldots, n$ are to be scheduled on a single machine. Job j takes t_j time units, has an integer deadline d_j, and a profit p_j. The machine can only process one job at a time and each job must be processed without interruption (for its t_j time units). A job j that finishes before its deadline d_j receives profit p_j, while a tardy job receives no profit (and might as well not have been scheduled). Give an efficient algorithm to find a schedule that maximizes total profit. You may assume that the t_j are integers in the range $[1, n]$. You should be able to come up with an $O(n^3)$ algorithm if you make the right observation about the maximum relevant deadline.

Solution: Suppose the jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$. Choosing whether or not d_n will be included in the solution leads to a subproblem involving the remaining jobs. If d_n is included, the deadline of each other job j is effectively reduced to $\min(d_j, d_n - t_n)$.

Taking this into account, let $M(k, d)$ be the maximum profit that can be obtained by jobs $1, \ldots, k$

given that all of them must finish at or before deadline d. Then

$$M(k,d) = \max \begin{cases} p_k + M(k-1, d-t_k) & \text{job } k \text{ is included} \\ M(k-1,d) & \text{job } k \text{ is not included} \end{cases}$$

The values of M can be stored in a table, which is filled by the following algorithm. The algorithm assumes that the jobs have already been sorted by increasing deadline – this can be done in $O(n \lg n)$, taking less time than the remainder of the algorithm.

function MaxProfit
input: a set of jobs $1, \ldots, n$ sorted so that $d_1 \le d_2 \le \ldots \le d_n$
output: a maximum profit schedule for the jobs
▷ $M[k,d]$ is the $M(k,d)$ defined above, stored in a table
▷ $J[k,d]$ is ***true*** iff job k is included in the schedule that achieves $M[k,d]$
$M[0,0] = 0$
for $k = 1$ **to** n **do**
for $d = 1$ **to** d_k **do**
if $d - t_k < 0$ **then** $M[k,d] = M[k-1,d]$; $J[k,d] =$ ***false***
else if $M[k-1,d] > p_k + M[k-1, d-t_k]$ **then** $M[k,d] = M[k-1,d]$; $J[k,d] =$ ***false***
else $M[k,d] = p_k + M[k-1,d-t_k]$; $J[k,d] =$ ***true***
endif

▷ Now extract the actual jobs; let $J =$ the set of jobs included
$J = \emptyset$; $d = d_n$
for $k = n$ **downto** 1 **do**
if $J[k,d]$ **then** $J = J \cup \{k\}$; $d = d - d_k$ **endif**

end MaxProfit

Technically, this algorithm runs in time $O(nd_n)$. and d_n could grow much faster than n. So it's not a $O(n^2)$ algorithm. However, we can assume, without loss of generality that no deadline exceeds $T = \sum_{1 \le k \le n} t_k$; that's the maximum time required to schedule all the jobs. In $O(n)$ preprocessing time we can set $d_k = \min(d_k, T)$. This reduces the overall runtime to $O(nT) = O(n^3)$.

Some students have claimed a $\Theta(n^2)$ algorithm, but I can't see how this can be made to work *unless the deadlines of all the jobs are distinct.* A solution that works under that assumption should receive 5 points if everything else is correct.

Part 2: Greedy Algorithms. *For any greedy algorithm you give you must be clear about how a choice reduces an instance of the problem to a smaller one and prove the greedy choice property.*

[Turns out, somewhat inadvertently on my part, that the following two problems are very similar. In particular, the proof technique of problem 5 could be for problem 4; vice-versa might be horrendously messy.]

4. [worth 6 points]

Problem 16-2(a) on page 447. Scheduling to minimize average completion time.

Solution: Schedule the tasks in order of increasing p_i. To see that this is consistent with an optimum solution, let $S = a_1 \cdots a_n$ be the sequence of tasks in a greedy schedule and pick an arbitrary optimum schedule $S^* = a_1^* \cdots a_n^*$. Let $a_i \ne a_i^*$ be the first place where the two schedules differ. Consider a schedule S' that differs from the optimum only by swapping the positions of a_i and a_i^*. So we have $S' = a_1^* \cdots a_i a_{i+1}^* \cdots a_i^* \cdots a_n^*$, where $p_i \le p_i^*$. Suppose that a_i^* ends up in position m of S'. In other words, let $S' = a_1' \cdots a_n'$. Then $a_1' \cdots a_{i-1}' = a_1^* \cdots a_{i-1}^*$, $a_i' = a_i$, $a_{i+1}' \cdots a_{m-1}' = a_{i+1}^* \cdots a_{m-1}^*$, $a_m' = a_i^*$, and $a_{m+1}' \cdots a_n' = a_{m+1}^* \cdots a_n^*$.

The completion times for tasks $a'_1, \ldots, a'_{i-1}$ are no different from those of $a^*_1, \ldots a^*_{i-1}$. Nor are those for tasks $a'_m, \ldots, a'_n$ and $a^*_m, \ldots a^*_n$. The completion times for tasks $a'_i, \ldots, a'_{m-1}$ differ from those of tasks $a^*_i, \ldots, a^*_{m-1}$ by $\sum_{j=1}^{i} p'_j - \sum_{j=1}^{i} p^*_j = p_i - p^*_i \leq 0$. So the total completion time for S' is $\leq$ that of S and the average completion time must be as well. The greedy choice property therefore holds. The algorithm can be implemented in time $\Theta(n \lg n)$ – linear time if the p_i are polynomial in n.

4 points for proof of greedy choice property, 2 points for algorithm and time bound.

5. [worth 8 points]

[from Brassard and Bratley, Fundamentals of Algorithmics, Prentice Hall, 1996]

Suppose n programs are stored on a magnetic tape (remember those?). Let $s(i)$ be the size of program i and $f(i)$ the frequency of use for program i. The time it takes to access a program is proportional to the sizes of all programs on the tape up to and including it. The goal is to minimize the total access time for all programs by finding an ordering of the programs on the tape. More formally, let π be a permutation of $1, \ldots, n$ that describes the ordering: so if $\pi(1) = i$ then program i is the first on the tape. The total cost is:

$$\sum_{i=1}^{n} f(\pi(i)) \sum_{k=1}^{i} s(\pi(k))$$

There are three possible greedy algorithms for minimizing this cost:

a. select programs in order of increasing $s(i)$ (nondecreasing if we allow for equal sizes – I use increasing to avoid confusion)

b. select programs in order of decreasing $f(i)$

c. select programs in order of decreasing $f(i)/s(i)$

For each algorithm, either find a counterexample to show that it is not optimal or prove that it always gives an optimum solution.

(a) [1 point] Consider the following counterexample:

i	1	2	3
$s(i)$	1	2	3
$f(i)$	1	1	3

The suggested algorithm selects programs in the order 1,2,3. The resulting cost is

$$f(1)s(1) + f(2)(s(1) + s(2)) + f(3)(s(1) + s(2) + s(3)) = 1 \cdot 1 + 1 \cdot 3 + 3 \cdot 6 = 22$$

A better solution arises if the order is 3,2,1. The cost is

$$f(3)s(3) + f(2)(s(3) + s(2)) + f(1)(s(3) + s(2) + s(1)) = 3 \cdot 3 + 1 \cdot 5 + 1 \cdot 6 = 20$$

(b) [1 point] Another counterexample:

i	1	2	3
$s(i)$	4	1	1
$f(i)$	3	2	1

The suggested algorithm selects programs in the order 1,2,3. The resulting cost is

$$f(1)s(1) + f(2)(s(1) + s(2)) + f(3)(s(1) + s(2) + s(3)) = 3 \cdot 4 + 2 \cdot 5 + 1 \cdot 6 = 28$$

A better solution arises if the order is 2,1,3. The cost is

$$f(2)s(2) + f(1)(s(2) + s(1)) + f(3)(s(2) + s(1) + s(3)) = 2 \cdot 1 + 3 \cdot 5 + 1 \cdot 6 = 23$$

(c) [5 points] This algorihm works. The recursive formulation is based on choosing a particular program to be last. Let P be the set of all programs and let $\pi^*(P)$ be an optimum permutation on the programs in P. Then $\pi^*(\textbf{\textit{null}}) = 0$ and

$$\pi^*(P) = \min_{1 \le i \le n} (\pi^*(P/i) \cdot i)$$

Here P/i is P with i removed and, in general, $\pi \cdot i$ is π with i added at the end. Optimal substructure is pretty clear: the amount added to the sum when i is added does not depend on the earlier choices in any way.

We prove the greedy choice property, namely that the best choice in the formula above is the i with the *smallest* $f(i)/s(i)$, as follows. Let $\overline{\pi}$ be the greedy permutation. Suppose that there is a different permutation π whose total access time is optimal. and that has the fewest number of inversions with respect to $\overline{\pi}$. Note that the number of inversions is a measure of how much π disagrees with $\overline{\pi}$.

If π and $\overline{\pi}$ are the same, we are done – the greedy solution is optimal.

If π and $\overline{\pi}$ are not the same, then there must be two programs a and b (and an index k) so that $\pi(k) = a$, $\pi(k+1) = b$ and $\overline{\pi}(k) = b$, $\overline{\pi}(k+1) = a$ – if two permutations disagree, there must be two neighboring positions where they disagree.

The greedy algorithm chose program a to appear earlier because $f(b)/s(b) \ge f(a)/s(a)$. Consider the effect of swapping a and b in π; call the swapped permutation π'. The access cost of programs $\pi(k+2), \ldots, \pi(n)$ is the same in π' and so is the cost for programs $\pi(1), \ldots, \pi(k-1)$. Let $S = \sum_{1 \le i \le k-1} s(\pi(i))$. Then the contributions of programs a and b to the cost with respect to π is

$$C = f(a)(S + s(a)) + f(b)(S + s(a) + s(b))$$

With respect to π', it is

$$C' = f(b)(S + s(b)) + f(a)(S + s(a) + s(b))$$

After some algebra, we see that $C' - C = f(a)s(b) - f(b)s(a)$. Dividing by $s(a)s(b)$ this is $f(a)/s(a) - f(b)/s(b) \le 0$. Thus π' has no more access cost than π and has fewer inversions with respect to $\overline{\pi}$ than π does, contradicting our assumption.