Grade maximization Problem 20 on pp. 329-330. Note that this is essentially the same as the homework grade maximization problem on the previous assignment, except that we drop the assumption that the improvement in your grade is monotone. (The total grade is monotone, but not necessarily the improvement). 14 points correct polynomial-time algorithm, 6 points time analysis and efficiency. My best time is $O(nH^2)$.

We need to allocate $H$ hours among projects $1...n$, i.e., find $h_1,..h_n$ with $\sum h_i = H$ in order to maximize $\sum f_i(h_i)$.

A recursive algorithm to do this is based on the possible answers to the question: How many hours do I devote to project 1? The possible answers are $0..H$. The recursion just returns the best achievable total grade, not the assignment that achieves it.

BTGM($H$, $f_1,..f_n$);

1. IF $n = 1$ return $f_1(H)$
2. $BestGrade \leftarrow 0$
3. FOR $h = 0$ to $H$ do:
   4. $ThisGrade \leftarrow f_1(h) + BTGM(H-h, f_2,..f_n)$
   5. IF $ThisGrade > BestGrade$ THEN $BestGrade \leftarrow ThisGrade$
4. Return $BestGrade$.

As we make recursive calls in this algorithm, the set of projects is always of the form $f_1,..f_n$, and the number of hours we have left is an integer from 0 to $H$. Thus, we have $O(nH)$ distinct recursive calls. Let $BG[H',I]$, $0 \leq H' \leq H, 1 \leq I \leq n$ be an array to store these subproblems, where $BG[H',I]$ is intended to be the optimal total grades achievable spreading $H'$ hours among projects $I...n$. In the top-down algorithm, the index $I$ would increase by 1 in all of the recursive calls. So the bottom-up order to fill in this array is $I$ decreasing from $n$ to 1. The case $I = n$ would correspond to the base case of the recursion. This gives a DP version of the recursion as follows:

DPGM($H$, $f_1,..f_n$):

1. Initialize $BG[0..H,1..n]$.
2. FOR $h = 0$ to $H$ do: $BG[h,n] \leftarrow f_n(h)$.
3. FOR $I = n - 1$ downto 1 do:
4. FOR $H' = 0$ to $H$ do:
5. $BestGrade \leftarrow 0$.
6. FOR $h = 0$ to $H'$ do:
7. $ThisGrade \leftarrow f_I(h) + BG[H' - h, I + 1]$
8. IF $ThisGrade > BestGrade$ THEN $BestGrade \leftarrow ThisGrade$.
9. $BG[H', I] \leftarrow BestGrade$.

The total time is dominated by the three nested loops, which are $O(n), O(H)$ and $O(H)$ respectively. So the time is $O(nH^2)$.

**Protein Bonding** Let $\Sigma$ be a finite set of amino acids, and let $w = w_1...w_n$ be a sequence of acids from $\Sigma$. For $\sigma, \sigma' \in \Sigma$, let $b(\sigma, \sigma')$ be the strength of a bond between the two types of acids, a non-negative real number. A bonding of the sequence is a partial matching between positions in the word so that matched pairs can be connected with lines drawn below the word without lines crossing. Equivalently, it should satisfy: there are no two bonded pairs $i_1, j_1$ and $i_2, j_2$ with $i_1 < i_2 < j_1 < j_2$. The total bond strength is the sum over all bonded positions $i, j$ of the bond strength $b(w_i, w_j)$. Give as efficient as possible algorithm to find the bonding of a protein sequence that maximizes the total bond strength. (15 points correct, polytime algorithm with proof; 5 pts efficiency. I know an $O(n^3)$ algorithm.)

Identify a bonding with the set of pairs of positions matched, each pair listed in increasing order. Let $B$ be a bonding of $w_1...w_n$. If $B$ does not match position 1 with anything, it is also a bonding on positions 2...n. Conversely, any bonding on positions 2...n is also a bonding on 1...n.

If $B$ matches position 1 with position $i$, then every match $(j, k)$ in $B$ must either have $2 \leq j < k \leq i - 1$ or $i + 1 \leq j < k \leq n$, since if $j < i < k$, we have $1 \leq j < i < k$, which violates our constraints. Let $B_{inside}$ be the matching on $2...i - 1$ of edges in $B$ of the first type, and $B_{outside}$ of the second type. $B_{inside}$ is a bonding on $2...i - 1$, and $B_{outside}$ is a bonding on $i + 1...n$, and $Totalstrength(B) = Totalstrength(B_{inside}) + Totalstrength(B_{outside}) + strength(w_i, w_i)$. Conversely, let $B_I$ be any bonding on $2...i - 1$, and $B_O$ any bonding on $i + 1...n$. Then $B = (1, i) \cup B_{O} \cup B_{I}$ is a bonding on 1...i of total strength $Totalstrength(B_{O}) + Totalstrength(B_{I}) + strength(w_1, w_i)$.

This gives us the following recursive backtracking algorithm:

$TS(w_1...w_n \in \Sigma^n, strength[\Sigma][\Sigma]:$ array of non-negative real numbers): non-negative real number.

1. IF $n \leq 1$ return 0; {need two positions to bond }
2. $\text{Non} - \text{bond} \leftarrow TS(w_2...w_n, strength)$.
3. $\text{Best} \leftarrow \text{Non} - \text{bond}$.
4. FOR $i = 2$ to $n$ do:
5. begin;
6. $\text{CurrentCase} \leftarrow TS(w_2...w_{i-1}, strength) + TS(w_{i+1}...w_n) + strength(w_1, w_i)$
7. IF $\text{Best} < \text{CurrentCase}$ THEN $\text{Best} \leftarrow \text{CurrentCase}$.
8. end;
9. Return $\text{Best}$.

Then we note that this recursive procedure never changes $strength$ and only calls itself on consecutive subsequences $w_i...w_j$ of the original sequence. The value of $i$ is always greater in a sub-call, so bottom-up can be in decreasing order of $i$. This allows us to create the following dp version of the above recursion: $\text{DPTS}(w_1...w_n \in \Sigma^n$, $strength[\Sigma][\Sigma]$): array of non-negative real numbers: non-negative real number.

1. Initialize array $total[1..n][1..n]$ to 0’s. {includes base case}
2. FOR $I = n-1$ to 1 do :
3. FOR $J = I + 1$ to $n$ do:
4. begin;{compute strength for $w_I...w_J$}
5. $\text{Non} - \text{bond} \leftarrow total(I + 1, J)$
6. $\text{Best} \leftarrow \text{Non} - \text{bond}$.
7. FOR $K = I + 1$ to $J$ do:
8. begin;
9. $\text{Curr} \leftarrow total(I + 1, K - 1) + total(K + 1, J) + strength(w_1, w_K)$
10. IF $\text{Best} < \text{Curr}$ THEN $\text{Best} \leftarrow \text{Curr}$.
11. end;
12. $total[I, J] \leftarrow \text{Best}$
13. end;

The total time is $O(n^3)$ since we have three nested loops, each with up to $n$ iterations.

**Monotone matching** Let $G = (L, R, E)$ be a bipartite graph with $|L| = |R| = n$, and where the nodes in $L$ are ordered as $u_1...u_n$ and the nodes in $R$ as $v_1...v_n$.

Then a matching $M$ is **monotone** if whenever $u_i$ is matched to $v_j$, and $u_{i'}$ is matched to $v_{j'}$, if $i < i'$ then $j < j'$. The problem is: given such
a bipartite graph \( G \), find the maximum size of a monotone matching in \( G \). (4 points correct poly time algorithm, 10 points correctness proof, 6 points efficiency and time analysis. My algorithm is \( O(n^2) \). )

This is similar to the sequence alignment problem in the text. Let \( l_1 \) be the first node on the left, and \( r_1 \) the first node on the right. If there is an edge between \( l_1 \) and \( r_1 \), then without loss of generality we can match them in an optimal matching. Since otherwise, at most one of the two is matched in the optimal matching, and by switching that at most one edge with the one between \( l_1 \) and \( l_2 \), we create a monotone matching of the same size. If there is no such edge, then either \( l_1 \) or \( r_1 \) must go unmatched (possibly both).

So here is a recursive algorithm returning the size of the maximum monotone matching based on the above:

RMM (\( L, R \))
1. IF \( |L| = 0 \) or \( |R| = 0 \) return 0.
2. IF \( \{l_1, r_1\} \in E \), return 1 + RMM(\( L - \{l_1\}, R - \{r_1\} \))
3. Return max\( (\)RMM(\( L - \{l_1\}, R \))\), RMM(\( L, R - \{r_1\} \))\).

Note that, as we make recursive calls in the above, we delete some initial segment from \( L \) and some initial segment from \( R \), leaving \( \{l_I, \ldots, l_n\} \) and \( \{r_I, \ldots, r_n\} \) for some \( 1 \leq I, J \leq n + 1 \). (As usual, I use \( I = n + 1 \) to denote an empty \( L \), and similarly for \( J \).

This gives the following \( DP \) version: DPMM(\( L, R \))

1. Initial matrix of integers \( M[I, J] \) for \( 1 \leq I, J \leq n + 1 \)
2. FOR \( I = 1 \) to \( n + 1 \) do: \( M[I, n + 1] \leftarrow 0 \)
3. FOR \( J = 1 \) to \( n + 1 \) do: \( M[n + 1, J] \leftarrow 0 \)
4. FOR \( I = n \) downto 1 do:
   5. FOR \( J = n \) downto 1 do:
      6. IF \( \{l_I, r_J\} \in E \) then \( M[I, J] \leftarrow 1 + M[I + 1, J + 1] \)
      7. else \( M[I, J] \leftarrow \max(M[I + 1, J], M[I, J + 1]) \).
   8. Return \( M[1, 1] \).

The main loop takes \( O(n^2) \) time which dominates the algorithm.

**Job allocation** You are given a set of \( n \) jobs and a set of \( n \) machines. For each \( 1 \leq i \leq n \) and \( 1 \leq j \leq n \), you are given a quantity \( T_{i,j} \) so that if you assign job \( i \) to machine \( j \), it will take \( T_{i,j} \) time. You need to assign each job \( j \) to a machine \( M[j] \), so that every machine gets exactly one job assigned to it. Your goal is to minimize the time all jobs are finished, \( \max_j T_j, M[j] \).
(10 points correct algorithm and correctness proof, 10 points efficiency. A hint is to consider the decision version: Given $T$ is there a way to assign jobs to machine so that all are finished within $T$ time?).

(This solution is a bit sketchy; apologies.) Following the hint, given $T$, the makespan will be less than $T$ if and only if we assign each job a machine that can perform it in less than $T$ time. Thus, if we construct a bipartite graph $G_T$ with edges between those jobs and machines that can perform the job in less than $T$ time, there is a schedule with makespan at most $T$ if and only if there is a perfect matching in $G_T$. One approach then is to do a binary search over the $n^2$ possible values of $T$ to find the minimum where $G_T$ has a bipartite matching. We can solve bipartite matching using FF in time $O(|E||V|) = O(n^3)$ which is larger than the time to sort the $n^2$ entries in the table. So this leads to an $O(n^3 \log n)$ algorithm.

However, we can observe that any flow for the network with a smaller value of $T$ is also a flow for a larger value with more edges. Starting from the smallest to the largest value of $T$, insert each edge into the graph one by one and attempt to augment the current flow. Since each augmentation increases the flow, and the max flow is $n$, the number of total iterations where we successfully augment will be at most $n$. We can also maintain the set of nodes $R$ reachable from $s$ in the residual graph. When we add an edge between nodes in $R$ or between nodes not in $R$, we do not need to check whether a path is created. Only when we add a new edge from a node in $R$ to a node not in $R$ do we need to perform a DFS, and we can keep the nodes in $R$ marked as visited during the DFS, so that we only search the edges coming out of nodes when they are added to $R$. So between successes at finding an augmenting path, the total work will be at most $O(n^2)$, since we spend a linear amount of work for every node added to $R$.

So the total number of successful augmentations throughout is $n$ and the total work between augmentations is at most $O(n^2)$, giving a total time $O(n^3)$. Since we stop as soon as we find a perfect matching, we find the smallest value of $T$ where such a matching exists, which as observed above, is the smallest makespan of any schedule.

**Number puzzle**

You are trying to solve the following puzzle. You are given the sums for each row and column of an $n \times n$ matrix of integers in the range 1...$M$, and wish to reconstruct a matrix that is consistent. In other words, your input is $M, r_1, \ldots, r_n, c_1, \ldots, c_n$. Your output should be a matrix $a_{i,j}$ of integers between 1 and $M$ so that $\forall j \sum_i a_{i,j} = r_j$ and $\forall i \sum_j a_{i,j} = c_i$; if no such matrix exists, you should output, "Impossible". Give an efficient algorithm for this problem. (15 points correctness for a poly-time algorithm; 5 points efficiency. My best algorithm takes $O(n^3$)).

We reduce the problem to network flow. First, we reduce the problem to
Dynamic Programming vs. Memoization

Implement the bottom-up dynamic programming algorithm for the length of the sequence alignment algorithm (section 6.6) (page 353), and implement a top-down, memoized version of the same recurrence. For a range of sizes $n$ (say, $n = 2^4$ to $n = 2^{15}$ or as high as time permits), compare the performance of these two algorithms on inputs consisting of two random strings of length $n$ over an alphabet of size 4. (You can present times with a log-log chart. Use
all times, not just cycles.) Which approach is better and why? Is there a cross-over point?

Usually, DP is better than Memoization, not because of asymptotic complexity (which is the same or worse), but because it can utilize locality of reference in its memory accesses, and avoids having a stack of subproblems. On the other hand, memoization occasionally is better when many entries of the matrix of sub-problems don’t need to be filled in because they are not actually called in the recursion.