Problem Set 8

All parts are due on November 21, 2017 at 11:59PM. Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on the course website, and any code should be submitted for automated checking on alg.csail.mit.edu .

Please solve the following problems using dynamic programming.

Problem 8-1. [10 points] **Binomial Recreation**

Pascal is chilling at home doing some recreational combinatorics in his notebook, and decides he needs to compute a binomial coefficient,

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}.$$

Pascal is terrible with arithmetic, so he gets out his calculator to perform some multiplications and divisions only to find that his calculator is mostly broken: only the digits and plus operator work, allowing only entry and addition of numbers. Pascal however is not deterred for he knows that binomial coefficients can also be defined recursively:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k},$$

with $\binom{n}{k} = 1$ when $k = 0$ or $k = n$. Help Pascal calculate his binomial coefficient using nothing but his broken calculator and notebook.

Solution:

The recursive definition of a binomial coefficient is a natural dynamic program:

1. Subproblems: $p(i, j)$ represents $\binom{i}{j}$.
2. Relation: $p(i, j) = p(i-1, j-1) + p(i-1, j)$
3. DAG:
   - Subproblems only depend on other subproblems with strictly smaller $n + k$, so acyclic
   - Base case: $p(i, 0) = p(i, i) = 1$ for any $i \in [0, n]$
4. Solution: Calculating $\binom{n}{k}$ is simply evaluating $p(n, k)$
5. Algorithm:
   - Use notepad to memoize results of smaller subproblems, either top-down or bottom-up

- Running Time: At most $(n+1)(k+1)$ subproblem solutions written on notepad, $O(1)$ work per subproblem using the calculator, so $O(nk)$ time

Rubric:

- +1 points for subproblems
- +1 points for recursive relation
- +2 points for acyclic dependencies
- +2 points for base case
- +2 points for solution
- +2 points for correctly evaluated running time
- Partial credit may be awarded
- Max 8 points for a correct but inefficient dynamic program, i.e. $\omega(nk)$

Problem 8-2. [15 points] **Optimal Trading**

Seorge Goros is a stock trader down on his luck. He has been consistently making the wrong trades of FredEx stock. Each day he is authorized to either buy or sell exactly s shares of FredEx at its daily price, or take no action. Ignoring inactive days, Seorge is required to alternate between buying and selling, so will never hold more than s shares between trades. To learn from his mistakes, Seorge decides to review the daily price of FredEx stock from the last d days. Describe an efficient algorithm to calculate an assignment of each day to either 'buy', 'sell', or 'none' that would have maximized his revenue during those days.

Solution:

Let $c(i)$ be the price of FredEx stock on day i

1. Subproblems: $p(i,j)$ maximum revenue trading only in the first i days while holding j shares at the end of day i (j can be either 0 or s).
2. Relation:
   - To maximize revenue in first i days, either Seorge sell shares on day i or not
   - If sell on day i, must be holding s shares to sell, so $j = s$
   - To buy on day i, must not be holding any shares, so $j = 0$
   - $p(i,0) = \max(p(i-1,0), p(i-1,s) + sc(i))$
   - $p(i,s) = \max(p(i-1,s), p(i-1,0) - sc(i))$
3. DAG:
   - Subproblems only depend on other subproblems with strictly smaller i, so acyclic
   - Base case: First day, can buy or not buy, so $p(1,s) = -sc(1)$, $p(1,0) = 0$
4. Solution:

- $p(d, 0)$ represents the maximum return in d days (holding shares at end is suboptimal)
- Order of trades constructable by storing parent pointers to maximizing subproblems

5. Algorithm:
   - Memoize subproblems in topological sort order, either top-down or bottom-up
   - Running Time: $2d$ subproblems each taking $O(1)$ time, so $O(d)$ time

Of course this problem can also be solved using a simpler dynamic program in $O(d^2)$ time by computing subproblems using $O(d)$ other subproblems. A correct quadratic dynamic program may be awarded up to $12/15$ points.

Rubric:

- +3 points for subproblems
- +3 points for recursive relation
- +2 points for acyclic dependencies
- +3 points for base case
- +2 points for solution
- +2 points for correctly evaluated running time
- Partial credit may be awarded
- Max 12 points for a correct but inefficient dynamic program, i.e. $\omega(d^2)$

Problem 8-3. [15 points] **Constrained Gift**

Ally Gorhythm just received a gift certificate to her favorite electronics store, though the gift certificate comes with a special condition: she must spend exactly d dollars at the store. Everything at the store is priced at a whole dollar amount, while the smallest component costs a dollar. Ally wants to spend her gift certificate by purchasing the fewest possible items at the store, allowing items to be purchased more than once. Given a price list of items in the store, describe an efficient algorithm to return a shortest shopping list to Ally.

Solution:

Assume there are n items, and let $c(i)$ be the price of item i, where items are indexed in an arbitrary linear order

1. Subproblems: $p(j)$ is the smallest number of items that can be chosen whose respective prices sum to j (i.e. filling a j dollar gift certificate).
2. Relation:
   - Either a minimizing subset uses item i or it does not.
   - Similar to Knapsack, where 'size' is the cost to fill, and the 'price' to minimize is the same for all items (1 per item used).

- $p(j) = \min_{i \in [1,n]} p(j - c(i)) + 1$

3. DAG:
   - Subproblems only depend on other subproblems with strictly smaller j, so acyclic
   - Base case: zero or negative gift certificate
     $p(0) = 0$, $p(j) = \infty$ for $j < 0$
4. Solution:
   - $p(d)$ represents filling a d dollar gift certificate using the fewest items
   - Shopping list is constructable by storing parent pointers to minimizing subproblems
5. Algorithm:
   - Memoize subproblems in topological sort order, either top-down or bottom-up
   - Running Time: $d + 1$ subproblems each taking $O(n)$ time, so $O(nd)$ time

Rubric:

- +3 points for subproblems
- +3 points for recursive relation
- +2 points for acyclic dependencies
- +3 points for base case
- +2 points for solution
- +2 points for correctly evaluated running time
- Partial credit may be awarded
- Max 12 points for a correct but inefficient dynamic program, i.e. $\omega(nd)$

Problem 8-4. [20 points] **Picking Teams**

A group of elementary school students take their recess very seriously. Each child has a dodge ball ability indicating how good they are at dodge ball, and every child knows every other child's ability. Every day at recess, two children are elected captains and must alternate choosing players from the remaining children to complete their teams. When picking teams the children stand in a line, and each captain takes turns adding a child to their team by choosing one of the two children at either end of the line. If you are the captain who picks first, describe a strategy to maximize the sum of dodge ball abilities on your team, assuming the other captain is also picking players to maximize her team.

Solution:

Assume there are n students, with $a(i)$ being the dodge ball ability of the ith student in the initial line up.

1. Subproblems: $p(i, j)$ and $q(i, j)$ are the maximum ability score you can achieve, when either you or your opponent choose next respectively, from among students remaining in a line up from student i to j, where $i \leq j$.
2. Relation:
   - You will maximize ability sum, while your opponent will minimize.
   - $p(i, j) = \max(q(i + 1, j) + a(i), q(i, j - 1) + a(j))$
   - $q(i, j) = \min(p(i + 1, j), p(i, j - 1))$
3. DAG:
   - Subproblems only depend on other subproblems with strictly smaller $j - i$, so acyclic
   - Base case: Only one choice if only one student, $p(i, i) = a(i), q(i, i) = 0$
4. Solution:
   - $p(1, n)$ represents a maximal score for you when choosing first
   - Optimal choices are constructable by storing parent pointers to maximizing subproblems
5. Algorithm:
   - Memoize subproblems in topological sort order, either top-down or bottom-up
   - Running Time: $O(n^2)$ subproblems each taking $O(1)$ time, so $O(n^2)$ time

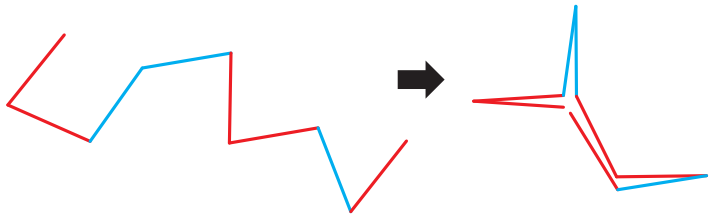
Rubric:

- +4 points for subproblems
- +6 points for recursive relation
- +2 points for acyclic dependencies
- +4 points for base case
- +2 points for solution

- +2 points for correctly evaluated running time
- Partial credit may be awarded
- Max 16 points for a correct but inefficient dynamic program, i.e. $\omega(n^2)$

Problem 8-5. [20 points] **Protein Folding**

Proteins are chains of amino acids that fold up into complex shapes. Some amino acids have natural chemical attractions to each other while others do not, and these attractions induce the protein to self assemble. Nature tends to find a folding that maximizes adjacencies between attractive pairs. Let's analyze a simplified model of this behavior. Consider a chain of n equal length edges, each edge colored either red or blue. A *valid folding* of the chain will match edges to each other to form a doubly covered tree in the plane. Describe an efficient algorithm to find a valid folding of the chain that maximizes same color matchings. An example of a valid folding that maximizes same color matchings is shown below.



Solution:

Because the folding results in a doubly covered tree, a valid folding pairs each edge uniquely to one other edge. We can think of the chain as a cycle since doubling covering a tree will bring the two endpoints together. Then bringing together any pair creates two smaller cycles. We will guess a pair of edges to bring together first and recurse on the remaining unmatched subsets. Let $c(i) \in \{Blue, Red\}$ be the color of edge i.

1. Subproblems: $p(i, j)$ is the maximum number of matchings from among any valid folding of the contiguous subset of edges from the ith to the jth, matching only within the subset.
2. Relation:
   - Guess the edge k that matches to edge i (could be any other edge from $i + 1$ to j)
   - $p(i, j) = \max_{k \in [i+1, j]} p(i + 1, k - 1) + p(k + 1, j) + (\ 1$ if $c(i) = c(k)$, and 0 otherwise)
3. DAG:
   - Subproblems only depend on other subproblems with strictly smaller $j - i$, so acyclic
   - Number of segments to match is given by $j - i + 1$
   - Base case: If odd or negative number of segments, valid folding impossible, so $p(i, j) = -\infty$ when $j - i + 1$ is odd or less than zero.
   - Base case: If exactly zero segments, no matchings, so $p(i, j) = 0$ when $j - i + 1 = 0$
4. Solution:

- $p(1, n)$ represents the maximum number of matchings for the whole chain
- Optimal matching is constructable by storing parent pointers to maximizing subproblems

5. Algorithm:
   - Memoize subproblems in topological sort order, either top-down or bottom-up
   - Running Time: $O(n^2)$ subproblems each taking $O(n)$ time, so $O(n^3)$ time

A natural but less efficient dynamic program runs in $O(n^4)$ time by matching all possible pairs in each possible cycle. Actually, an asymptotically faster greedy algorithm exists to solve the problem, but is not naturally representable as a dynamic program with overlapping subproblems. We will give full credit to any **dynamic programming** solution up to cubic running time.

For completeness, here's the faster greedy algorithm. First we prove that there always exists an optimal solution that matches every adjacent same-color pair of edges to each other.

To prove the claim, consider two adjacent same-color edges A and B, and consider an optimal valid folding where A and B are not matched to each other. Then A is matched to some other edge X, and B is matched to some other edge Y in the folding. Modify the folding to separate these two matchings, and then rematch A to B and X to Y. If we can show that this new folding is also valid and optimal, we can use this procedure to rematch any optimal solution so that every adjacent same-color pair of edges are matched to each other, thus proving the claim.

The new folding is valid because we didn't change any other matchings between edges within the three arcs between A and X; X and Y; and Y and B. To show the new folding remains optimal, assume without loss of generality that A and B are both red. Then if both X and Y are also red, the new folding doesn't change the number of matchings. If X and Y are not the same color, we also didn't change the number of matchings in the new folding. Lastly, X and Y cannot both be blue or else the modified folding would increase the number of matchings, contradicting the minimality of the optimal folding with which we started.

So the greedy algorithm looks through the cycle, and if there are any adjacent edges that have the same color, fold them and remove them from the cycle. After folding, check locally if it results in additional adjacent edges. In the base case, the unmatched edges form an alternating cycle of red and blue edges, which can never match same colored edges to each other in any doubly covered tree. This algorithm runs in $O(n)$ time.

Note that there are variants of this problem for which the greedy algorithm fails but the dynamic programming approach remains valid. For example, if we modify the problem to not count matches between same color edges if they are adjacent to each other in the chain, the greedy algorithm breaks down, but the dynamic program can be modified to apply (this is an interesting exercise).

Rubric:

- +4 points for subproblems
- +6 points for recursive relation
- +2 points for acyclic dependencies

- +4 points for base case
- +2 points for solution
- +2 points for correctly evaluated running time
- Partial credit may be awarded
- Max 16 points for a correct but inefficient dynamic program, i.e. $\omega(n^3)$

Problem 8-6. [20 points] **Return of Dr. Frick**

Dr. Crancis Frick no longer thinks that permutations of a disease DNA substring in someone's DNA results in the same disease. That was a silly idea. However, he has noticed that "similar" sequences to a disease substring results in a milder form of the disease. Given two strings, the *distance* between them is the smallest number of individual insertions, deletions, or replacements of DNA letters needed to transform one string into the other. Dr. Frick observes that the severity of disease symptoms later in life is inversely proportional to the minimum distance between the disease substring and any substring of a person's DNA. For example, if the presence of disease substring (C, A, T, A, C, T) induces someone to exhibit mannerisms indistinguishable from a kitten, a human containing the substring (C, A, G, T, A, C, A) might only lead to audible purring by the individual, being a distance 2 away from the disease via removal of G and replacement of the last A by T. Given a patient's DNA, help Dr. Frick diagnose the severity of (i.e. distance to) a given disease.

Solution:

Let A be the n characters of the person's DNA, and let B be the k characters of the disease substring.

1. Subproblems: $p(i, j)$ is the minimum distance between any substring of $A[:i]$, and disease substring $B[:j]$.
2. Relation: Minimize over five local options
   (a) If last characters match, match them and recurse on smaller problem. Otherwise:
   (b) Replace $A[i]$ with $B[j]$ and match them at a cost of one modification
   (c) Insert $B[j]$ to the end of $A[:i]$ and match them at a cost of one modification
   (d) If we've already started matching, delete $A[i]$ and recurse at cost one
   (e) If we haven't started matching, find best match in smaller substring

$$
\begin{aligned}
p(i,j) = \min(&p(i-1,j-1) \text{ if } A[i] = B[j], && (a)\\
&p(i-1,j-1)+1, && (b)\\
&p(i,j-1)+1, && (c)\\
&p(i-1,j)+1 \text{ if } j < k, && (d)\\
&p(i-1,j) \text{ if } j = k) && (e)
\end{aligned}
$$

3. DAG:
   - Subproblems only depend on other subproblems with strictly smaller $i + j$, so acyclic
   - Base case: Done if no disease to match, $p(i, 0) = 0$
   - Base case: Negative number of characters impossible, $p(i, j) = \infty$ for negative i or j
4. Solution:
   - $p(n, k)$ represents the minimum distance match between B and any substring of A
5. Algorithm:
   - Memoize subproblems in topological sort order, either top-down or bottom-up
   - Running Time: $O(nk)$ subproblems each taking $O(1)$ time, so $O(nk)$ time

A natural but inefficient dynamic program runs in $O(n^3k)$ time, running a similar edit distance evaluation as above, for each contiguous subset of A.

Rubric:

- +4 points for subproblems
- +6 points for recursive relation
- +2 points for acyclic dependencies
- +4 points for base case
- +2 points for solution
- +2 points for correctly evaluated running time
- Partial credit may be awarded
- Max 16 points for a correct but inefficient dynamic program, i.e. $\omega(nk)$