Problem Set 10

All parts are due on May 11, 2018 at 11PM. 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 `py.mit.edu/6.006`.

Please solve the following problems using dynamic programming. Be sure to define a set of subproblems, relate the subproblems recursively, argue the relation is acyclic, provide base cases, construct a solution from the subprob- lems, and analyze running time. Correct but inefficient polynomial time dynamic programs will be awarded significant partial credit.

Problem 10-1. [15 points] **Mountain Finding**

A sequence of integers is considered a **mountain** if it strictly monotonically increases to a maximum and then strictly monotonically decreases. For example, $A = (2, 3, 5, 4, 1)$ is a mountain, but $B = (3, 6, 8, 7, 9, 6, 4)$ is not. However B does contain a long subsequence $B' = (3, 6, 7, 9, 6, 4)$ which is a mountain. Given a sequence of n **unique** integers, describe an $O(n^2)$ time dynamic programming algorithm that finds its longest (not necessarily contiguous) subsequence that is also a mountain.

Solution: A subsequence that is a mountain can be interpreted as an increasing subsequence s_1, followed by a decreasing subsequence s_2, where s_2 shares its first element with s_1's last element. Therefore, for every integer a_i in the original sequence, if we compute the longest increasing subsequence that **ends** on a_i and the longest decreasing subsequence that **starts** on a_i, we can find the longest mountain with a_i as its peak. So the original problem reduces to solving two instances of the longest increasing subsequence problem in $O(n^2)$ time, and finding the integer whose mountain is longest in $O(n)$ time, for $O(n^2)$ total running time. Note that since longest increasing subsequence can be solved in $O(n \log n)$ time (see Recitation Notes 16), this problem can also be solved in $O(n \log n)$ time.

Rubric:

- 15 points for a correct reduction to longest increasing subsequence, or solve directly:
- 3 points for subproblem definition
- 7 points for recursive relation
- 1 point for arguing acyclic/providing topological sort order
- 2 points for base case(s)
- 1 point for solution from subproblems
- 1 points for running time analysis
- Partial credit may be awarded

Problem 10-2. [15 points] **Professor Devkusha**

The nefarious Professor Devkusha enjoys playing games with his students. He calls you to his office, and places a row of n coins with positive values $c_1, c_2, \ldots c_n$, onto a table in front of you, and proceeds to describe the rules of an asymmetric turn-based game. Professor Devkusha will first point to the space between any two coins, breaking the row into two **sections**. Then, you in turn may choose to remove any one coin on the table to obtain for yourself, but all other coins in the same section as the chosen coin will be removed from the table and discarded. These turns alternate until either zero or one coin remain on the table. Professor Devkusha will let you pass his class, only if you can obtain coins whose values total to at least k by the end of the game. Assuming that Professor Devkusha always plays optimally, describe an $O(n^3)$ time dynamic programming algorithm to determine whether you can pass his class.

Solution:

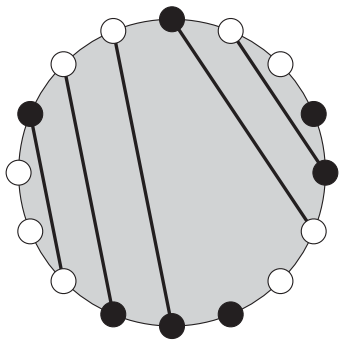
1. **Subproblems**
   - The professor wants to choose a breaking point that minimizes your possible total, while you want to choose a coin from a side that will maximize your possible total
   - $x(i, j)$: maximum coin value total you can achieve if it is the professor's turn dividing the contiguous subset of coins c_i to c_j
   - It is always optimal to choose the maximum coin value on one of the two sections
   - $m(i, j)$: maximum coin value from the contiguous subset of coins from c_i to c_j
2. **Relate**
   - Guess professor's division (minimization) and the section you choose (maximization)
   - $x(i, j) = \min_{k \in [i, j-1]}(\max(m(i, k) + x(k + 1, j), x(i, k) + m(k + 1, j)))$
   - $m(i, j) = \max(m(i, j - 1), c_j)$
3. **DAG**
   - Subproblems $m(i, j)$ and $x(i, j)$ only depend on strictly smaller $j - i$, so acyclic
   - Solve in order of increasing $j - i \in [1, n - i]$ and increasing $i \in [1, n]$
   - Base case: $x(i, i) = 0$ (one coin, game is over), $m(i, i) = c_i$
4. **Evaluate**
   - Solve subproblems via recursive top down or iterative bottom up.
   - Solution to original problem is $x(1, n) \geq k$.
5. **Analysis**
   - # subproblems: $O(n^2)$
   - work per subproblem: $O(n)$ (if $m(i, j)$ not precomputed, would take $O(n^2)$ time)
   - $O(n^3)$ running time

Rubric:

- 3 points for subproblem definition
- 7 points for recursive relation
- 1 point for arguing acyclic/providing topological sort order
- 2 points for base case(s)
- 1 point for solution from subproblems
- 1 points for running time analysis
- Partial credit may be awarded

Problem 10-3. [15 points] **Wafer Power**

A start-up is working on a new electronic circuit design for highly-parallel computing. Evenly-spaced along the perimeter of a circular wafer sits n ports for either a power source or a computing unit. Each computing unit needs energy from a power source, transferred between ports via a wire etched into the top surface of the wafer. However, if a computing unit is connected to a power source that is too close, the power can overload and destroy the circuit. Further, no two etched wires may cross each other. The circuit designer needs an automated way to evaluate the effectiveness of different designs, and has asked you for help. Given an arrangement of power sources and computing units plugged into the n ports, describe an $O(n^3)$ time dynamic programming algorithm to match computing units to power sources by etching non-crossing wires between them onto the surface of the wafer, in order to maximize the number of powered computing units, where wires may not connect two adjacent ports along the perimeter. Below is an example wafer, with non-crossing wires connecting computing units (white) to power sources (black).



Solution:

1. **Subproblems**
   - Let a_i be the type of object in port p_i around the circle, for $i \in [1, n]$.
   - Want to match opposite ports which can be connected by non-intersecting wires.
   - $x(i, j)$: maximum number of matchings when restricted to ports in the interval from p_i to p_j, for i, j such that $1 \leq i < j \leq n$.

2. **Relate**
   - Guess what port to match with first port in interval, or that first port does not match.
   - Non-adjacency condition restricts possible matchings with i to $t \in [i+2, j]$, except when $(i, j) = (1, n)$ where $t \in [3, n-1]$, as 1 cannot match with n.
   - $x(i, j) = \max(x(i+1, j), \max_{t \in [i+2, j], a_i \neq a_t}(1 + x(i+1, t-1) + x(t+1, j)))$
   - for $i \in [1, n]$ and $j \in [i, n]$, except for $(i, j) = (1, n)$:
   - $x(1, n) = \max(x(2, n), \max_{t \in [3, n-1], a_i \neq a_t}(1 + x(2, t-1) + x(t+1, n)))$
3. **DAG**
   - Subproblems $x(i, j)$ only depend on strictly smaller $j - i$, so acyclic
   - Solve in order of increasing $j - i \in [0, n-1]$, then in increasing $i \in [1, n]$
   - Base case: $x(i, j) = 0$ for $j - i \in \{0, 1\}$ (too small for any matches)
4. **Evaluate**
   - Solve subproblems via recursive top down or iterative bottom up.
   - Solution to original problem is $x(1, n)$.
   - Can store t if matching at t achieved maximum or `None` if no match acheived maximum, in order to reconstruct matching.
5. **Analysis**
   - # subproblems: $O(n^2)$
   - work per subproblem: $O(n)$
   - $O(n^3)$ running time

Rubric:

- 3 points for subproblem definition
- 7 points for recursive relation
- 1 point for arguing acyclic/providing topological sort order
- 2 points for base case(s)
- 1 point for solution from subproblems
- 1 points for running time analysis
- Partial credit may be awarded

Problem 10-4. [15 points] **Oh Charlie, Where Art Thou?**

A wealthy family, Alice, Bob, and their young son Charlie set off on a sailing journey around the world in their yacht. However, during their voyage the family encounters a massive storm, which throws Charlie from the boat, lost to the sea. Twenty years later, Alice and Bob are approached by a young man claiming to be Charlie, having survived for many years on a deserted island. Alice and Bob are excited at the prospect of reconnecting with their long lost son but are also skeptical of

this man's claim. They order a DNA matching test from the genetic testing company 46AndThee. Given equal length n DNA sequences from Alice, Bob, and Charlie, the testing center uses the following criteria to test biological relations: Charlie is Alice and Bob's son if and only if Charlie's DNA can be partitioned into two (not necessarily contiguous) subsequences of equal length, such that one is a subsequence of Alice's DNA, and the other is a subsequence of Bob's DNA. For example, suppose Alice's DNA is `AATT` and Bob's DNA is `CCGG`. If Charlie's DNA were to be `CATG`, he would be matched as a son, as Charlie's DNA would contain disjoint subsequences CG and AT which are subsequences of Alice and Bob's DNA respectively. However, Charlie would be found to be an imposter if his DNA were `AGTC`. Describe an $O(n^4)$ time dynamic programming algorithm to determine whether Charlie is indeed the couple's son.

Solution:

1. **Subproblems**
   - Let A, B, and C be the relevant length n DNA sequences from Alice, Bob, and Charlie.
   - Want to match characters of A and B to all characters of C
   - $x(i, j, k, l)$: true T if can match k sequential characters from $A[:i]$ and l from $B[:j]$ to all characters in $C[:k+l]$, false F otherwise.
2. **Relate**
   - If $A[i-1] = C[i-1]$ or $B[i-1] = C[i-1]$, check whether matching yields solution.
   - Alternatively, do not use $A[i-1]$ or $B[i-1]$
   - $x(i, j, k, l) = \text{OR} \left\{ \begin{array}{ll} x(i-1, j, k-1, l) & \text{if } A[i-1] = C[i-1] \\ x(i, j-1, k, l-1) & \text{if } B[i-1] = C[i-1] \\ x(i-1, j, k, l) & \text{always} \\ x(i, j-1, k, l) & \text{always} \end{array} \right\}$
3. **DAG**
   - Subproblems $x(i, j, k, l)$ only depend on strictly smaller $i + j + k + l$, so acyclic
   - Solve in order of increasing i, j, k, l
   - Base case: $x(i, j, 0, 0) = T$ for $i, j \geq 0$ (all matched!)
   - Base case: $x(i, j, k, l) = F$ for $i < \max(k, 0)$ or $j < \max(l, 0)$ (not enough characters!)
4. **Evaluate**
   - Solve subproblems via recursive top down or iterative bottom up.
   - Solution to original problem is $x(n, n, n/2, n/2)$.
5. **Analysis**
   - # subproblems: $O(n^4)$
   - work per subproblem: $O(1)$
   - $O(n^4)$ running time

Rubric:

- 3 points for subproblem definition
- 7 points for recursive relation
- 1 point for arguing acyclic/providing topological sort order
- 2 points for base case(s)
- 1 point for solution from subproblems
- 1 points for running time analysis
- Partial credit may be awarded

Problem 10-5. [40 points] **Winning Elections**

Jane 'The Paper' Dwonson is running for president of the Federated Districts of Algorithmia. Each district has a population and a designated electoral count. If greater than 50% of the population of a district votes for a candidate, that candidate receives the electoral count associated with that district (a winner-take-all system). Given a list of n districts in Algorithmia and their corresponding populations and electoral counts, Jane would like to determine the minimum number of votes she must win, in order to receive exactly c electoral counts, for a given value of c. For example, here is a list of districts, with associated populations and electoral counts:

District	Population	Electoral Count
Balaska	1056	30
Nine	1256	30
Bexastama	77	10
Twelve	1223	20
Dashington W.C.	454	20

To receive exactly $c = 40$ electoral counts, Jane must win at a minimum 568 votes, accomplished by winning districts Balaska and Bexastama, with 529 and 39 votes respectively.

(a) [10 points] Describe a dynamic programming algorithm to determine the minimum number of votes Jane needs to win, in order to receive exactly c electoral counts.

Solution:

1. **Subproblems:**
   - For a district $i \in [1, n]$, let p_i be its population and let c_i be its electoral count
   - Idea: Fix order on districts, is last district n in a minimizing set? (Guess!)
   - If yes, need to win population $\lfloor p_n/2 \rfloor + 1$, and still need to receive $c - c_n$ electoral counts with remaining districts
   - If no, then try to gain c electoral counts using remaining districts
   - $x(i, j)$: minimum population to win exactly j electoral counts using districts from 1 to i

2. **Relate:**
   - $x(i,j) = \begin{cases} \min(x(i-1,j), \lfloor p_i/2 \rfloor + 1 + x(i-1, j-c_i)) & \text{if } j \geq c_i \\ x(i-1,j) & \text{otherwise} \end{cases}$
   - $i \in [0,n], j \in [0,c]$
3. **DAG:**
   - Subproblems $x(i,j)$ only depend on strictly smaller i, so acyclic
   - Solve in order of increasing i, then increasing (or arbitrary) j
   - Base case: $x(i,0) = 0$ for $i \in [0,n]$, $x(0,j) = \infty$ for $j \in [1,c]$
4. **Evaluate:**
   - Solve subproblems via recursive top down or iterative bottom up
   - Minimum population for c counts given by $x(n,c)$ (or `None` if $x(n,c) = \infty$)
   - (Can store parent pointers to reconstruct states achieving minimum population)
5. **Analysis:**
   - # subproblems: $O(nc)$
   - work per subproblem $O(1)$
   - $O(nc)$ running time

Rubric:

- 2 points for subproblem definition
- 4 points for recursive relation
- 1 point for arguing acyclic/providing topological sort order
- 1 points for base case(s)
- 1 point for solution from subproblems
- 1 points for running time analysis

(b) [5 points] Briefly argue why a recursive top-down implementation of a dynamic programming algorithm might be preferable to a bottom-up implementation.

Solution: A recursive top-down implementation of a dynamic program may not evaluate all possible subproblems in the subproblem space (as discussed in lecture 19). Thus, a top-down implementation may do significantly less work than a bottom-up implementation for some dynamic programs.

Rubric:

- 5 points for a correct argument
- Partial credit may be awarded

(c) [25 points] Write the Python function `min_votes(districts, c)` that implements your algorithm from part (a) using a **top-down** dynamic programming implementation. The input `districts` is a list of length n containing the information for

districts, which are tuples consisting of the population and electoral count for the district. Your function should return the minimum number of people who must vote for Jane in order to obtain exactly c electoral counts, or output `None` if it is not possible to obtain the exact number. Submit your code online at `py.mit.edu/6.006`.

Solution:

```
def min_votes(districts, c, i = None, memo = None):
    '''
    Find minimum votes needed to obtain c electoral counts.
    Input:  list of district tuples containing (population, electoral count)
            for each district, and target electoral count c
    Output: Minimum number of popular votes necessary, or None if not possible
    '''
    if i is None:            # initialize memo
        return min_votes(districts, c, len(districts) - 1, {})
    if (c, i) in memo:       # read from memo
        return memo[(c, i)]
    if c == 0:               # base case: no count so need no population
        return 0
    if i < 0:                # base case: no districts left
        return None
    v1 = min_votes(districts, c, i - 1, memo)
    if c - districts[i][1] >= 0:
        v2 = min_votes(districts, c - districts[i][1], i - 1, memo)
        if v2 is not None:
            v2 += (districts[i][0] // 2) + 1
            v1 = v2 if (v1 is None) else min(v1, v2)
    memo[(c, i)] = v1        # memoize
    return v1
```