Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Instructors: Jason Ku, Julian Shun, and Virginia Williams

November 8, 2019
Problem Set 8

Problem Set 8

All parts are due on November 15, 2019 at 6PM. 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.mit.edu`.

Please solve each of the following problems using **dynamic programming**. For each problem, 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 subproblems, and analyze running time. Correct but inefficient dynamic programs will be awarded significant partial credit.

Problem 8-1. [15 points] Peter Piper

Peter Piper wants a peck of pickled peppers, so he planted a patch of n pepper plants in a line. The i^{th} pepper plant possesses a known positive integer p_i number of peppers. Peter Piper didn't plant his peppers properly, and now his poor plants are over-packed. Peter Piper plans to pick and pickle some peppers prematurely so the unpicked plants can prosper. For each plant, Peter Piper will either leave the plant **unpicked**, or **prune** the plant and pick all its peppers; but for each plant left unpicked, Peter Piper must prune both its neighboring plants to provide the unpicked plant space to prosper. Peter Piper prefers to pick as parsimoniously[1] as possible because he's picking prematurely. Please propose an efficient algorithm to help Peter Piper pick which plants to prune.

Solution:

1. **Subproblems**
   - $x(i)$: the minimum number peppers Peter can pick from plants 1 to i, where plant i may be pruned or left unpicked
2. **Relate**
   - Guess whether plant i is pruned or left unpicked
   - If plant i is pruned, then plant $i-1$ may be pruned or left unpicked
   - If plant i is left unpicked, then plant $i-1$ must be pruned, while plant $i-2$ may be pruned or left unpicked
   - $x(i) = \min(p_i + x(i-1), p_{i-1} + x(i-2))$
   - Subproblem $x(i)$ only depends on subproblems with strictly smaller i, so acyclic
3. **Base**
   - If there is no plant or only one plant, Peter need not prune any plant
   - $x(0) = x(1) = 0$

[1]He wants to minimize the sum of p_i for all picked peppers, subject to his pruning requirement.

4. **Solution**
   - $x(n)$ is the minimum number of peppers Peter can pick from all the plants, as requested
   - Store parent pointers to reconstruct which plants an optimal solution prunes
5. **Time**
   - # subproblems: $n + 1$, $x(i)$ for $i \in \{0, 1, \ldots, n\}$
   - work per subproblem: $O(1)$
   - $O(n)$ running time

Rubric:

- 4 points for a correct subproblem description
- 3 points for a correct recursive relation
- 1 points for indication that relation is acyclic
- 1 point for correct base cases in terms of subproblems
- 1 point for correct solution in terms of subproblems
- 2 points for correct analysis of running time
- 3 points if correct algorithm is $O(n)$
- Partial credit may be awarded

Problem 8-2. [15 points] **Coin Crafting**

Ceal Naffrey is a thief in desperate need of money. He recently acquired n identical gold coins. Each coin has distinctive markings that would easily identify them as stolen if sold. However, using his amateur craftsman skills, Cael can melt down gold coins to craft other golden objects. Ceal has a buyer willing to purchase golden objects at different rates, but will only purchase one of any object. Ceal has compiled a list of the n golden objects, listing both the positive integer **purchase price** the buyer would be willing to pay for each object and each object's positive integer **melting number**: the number of gold coins that would need to be melted to craft that object. Given this list, describe an efficient algorithm to determine the maximum revenue that Ceal could make, by melting down his coins to craft into golden objects to sell to his buyer.

Solution:

1. **Subproblems**
   - Label each craft-able object with a unique integer from 1 to n
   - Let p_i be the purchase price of object j, with k_i its melting number
   - $x(i, j)$: the maximum revenue possible from i coins, being able to craft any of the objects from the objects from 1 to j
2. **Relate**

- Guess whether or not to craft object j
- If $i < k_i$, object j cannot be crafted
- If object j is not crafted, may recurse on remaining items
- If object j is crafted, receive p_i in revenue and lose k_i coins
- $x(i,j) = \begin{cases} x(i, j-1) & \text{if } i < k_i \\ \max(p_i + x(i-k_i, j-1), x(i, j-1)) & \text{otherwise} \end{cases}$
- Subproblem $x(i,j)$ only depends on subproblems with strictly smaller j, so acyclic

3. **Base**
   - If there are not more coins, or no more items, cannot gain any revenue
   - $x(0, j) = 0$ for $i \in \{0, \dots, n\}$
   - $x(i, 0) = 0$ for $j \in \{0, \dots, n\}$
4. **Solution**
   - $x(n, n)$ is the maximum revenue possible from n coins, being able to craft any of the $n$objects, as requested
5. **Time**
   - # subproblems: $(n+1)^2 = O(n^2)$, $x(i,j)$ for $i, j \in \{0, 1, \dots, n\}$
   - work per subproblem: $O(1)$
   - $O(n^2)$ running time

Rubric:

- 4 points for a correct subproblem description
- 3 points for a correct recursive relation
- 1 points for indication that relation is acyclic
- 1 point for correct base cases in terms of subproblems
- 1 point for correct solution in terms of subproblems
- 2 points for correct analysis of running time
- 3 points if correct algorithm is $O(n^2)$
- Partial credit may be awarded

Problem 8-3. [15 points] **Building Blocks**

Saggie Mimpson is a toddler who likes to build block towers. Each of her blocks is a 3D rectangular prism, where each block b_i has a positive integer width w_i, height h_i, and length ℓ_i, and she has at least three of each type of block. Each block may be **oriented** so that any opposite pair of its rectangular faces may serve as its **top** and **bottom** faces, and the **height** of the block in that orientation is the distance between those faces. Saggie wants to construct a tower by stacking her blocks as high as possible, but she can only stack an oriented block b_i on top of another oriented block b_j if the dimensions of the bottom of block b_i are strictly smaller[2] than the dimensions of the top of block b_j. Given the dimensions of each of her n blocks, describe an $O(n^2)$-time algorithm to determine the height of the tallest tower Saggie can build from her blocks.

Solution:

1. **Subproblems**
   - Each block may be used in one of three vertical orientations
   - (without loss of generality, block bases can always be stacked with the base's shorter side pointed in one direction)
   - Because the stacking requirement requires base dimensions to strictly decrease, any optimal tower may use any block type at most three times (once in each orientation)
   - Sort dimensions of each block and remove duplicates (e.g., in $O(n)$ time with hash table)
   - For each block type with sorted dimensions $a \le b \le c$, construct three block orientations (a, b, c), (a, c, b), (b, c, a) (last dimension corresponding to height), and add them to an oriented block list in $O(n)$ time
   - (triplicating block types allowable since there are at least three of each type)
   - Sort lexicographically (first dimension most significant) in $O(n \log n)$ time
   - Re-number sorted list, where block i has oriented dimensions (w_i, ℓ_i, h_i) with $w_i \le \ell_i$
   - Any stackable tower of oriented blocks must be a subsequence of this sorted list since the w_i are sorted, though not every subsequence of this list comprises a valid tower, since the ℓ_i are not necessarily sorted
   - $x(i)$: the maximum tower height of any tower that uses block i and any subset of the remaining blocks 1 through $i - 1$
2. **Relate**
   - Guess the next lower block in the tower, only from blocks with strictly smaller ℓ_i
   - $x(i) = h_i + \max\{0\} \cup \{x(j) \mid j \in \{1, \dots, i-1\} \text{ s.t. } \ell_i < \ell_j\}$
   - Subproblem $x(i)$ only depends on subproblems with strictly smaller i, so acyclic
3. **Base**
   - $x(1) = h_1$, since the maximum in the recurrence is trivially zero

[2] If the bottom of block b_i has dimensions $p \times q$ and the top of block b_j has dimensions $s \times t$, then b_i can be stacked on b_j in this orientation if either: $p < s$ and $q < t$; or $p < t$ and $q < s$.

4. **Solution**
   - Some block must be the top block, so compare all possible top blocks
   - $\max\{x(i) \mid i \in \{1, \ldots, n\}\}$
5. **Time**
   - pre-processing: $O(n \log n)$
   - # subproblems: n, $x(i)$ for $i \in \{1, \ldots, n\}$
   - work per subproblem: $O(n)$
   - computing the solution: $O(n)$
   - $O(n^2)$ running time
   - Note that this problem can be solved in $O(n \log n)$ with a similar optimization as in Longest Increasing Subsequence from Recitation 15.

Rubric:

- 4 points for a correct subproblem description
- 3 points for a correct recursive relation
- 1 points for indication that relation is acyclic
- 1 point for correct base cases in terms of subproblems
- 1 point for correct solution in terms of subproblems
- 2 points for correct analysis of running time
- 3 points if correct algorithm is $O(n^2)$
- Partial credit may be awarded

Problem 8-4. [15 points] **Diffing Data**

Operating system Menix has a `diff` utility that can compare files. A **file** is an ordered sequence of strings, where the ith string is called the ith **line** of the file. A single **change** to a file is either:

- the insertion of a single new line into the file;
- the removal of a single line from the file; or
- swapping two adjacent lines in the file.

In Menix, swapping two lines is cheap, as they are already in the file, but inserting or deleting a line is expensive. A **diff** from a file A to a file B is any sequence of changes that, when applied in sequence to A will transform it into B, under the conditions that any line may be swapped at most once and any pair of swapped lines appear adjacent in A and adjacent in B. Given two files A and B, each containing exactly n lines, describe an $O(kn + n^2)$-time algorithm to return a diff from A to B that minimizes the number of changes that are **not swaps**, assuming that any line from either file is at most k ASCII characters long.

Solution:

1. **Subproblems**
   - First, use a hash table to assign each unique line a number in $O(kn)$ time
   - Now each line can be compared to others in $O(1)$ time
   - $x(i, j)$: the minimum non-swap changes to transform $A[: i]$ into $B[: j]$
2. **Relate**
   - If $A[i] = B[j]$, then recurse on remainder
   - Otherwise maximize a last change applied:
     - $A[i]$ is deleted
     - an insertion matches with $B[j]$
     - the last two in $A[: i]$ are swapped to match the last two in $B[: j]$
   - if $A[i] = A[j]$, $x(i, j) = x(i - 1, j - 1)$
   - otherwise, $x(i, j) = \min \left\{ \begin{array}{ll} 1 + x(i-1, j) & \text{delete} \\ 1 + x(i, j-1) & \text{insert} \\ x(i-2, j-2) & \text{swap if } A[i] = B[j-1] \text{ and } A[i-1] = B[j] \end{array} \right\}$
   - Subproblem $x(i, j)$ only depends on subproblems with strictly smaller $i + j$, so acyclic
3. **Base**
   - $x(0, 0) = 0$, all lines converted
   - $x(i, 0) = i$, must delete remainder
   - $x(0, j) = j$, must insert remainder
4. **Solution**
   - $x(n, n, 0)$ by definition
   - Store parent pointers to reconstruct which changes were made (for cases where $A[i] \neq A[j]$, remember whether a deletion, insertion, or swap occurred)
5. **Time**
   - pre-processing: $O(kn)$
   - # subproblems: $(n + 1)^2 = O(n^2)$, $x(i, j)$ for $i, j \in \{0, 1, \ldots, n\}$
   - work per subproblem: $O(1)$
   - $O(n^2)$ running time

Rubric:

- 4 points for a correct subproblem description
- 3 points for a correct recursive relation
- 1 points for indication that relation is acyclic
- 1 point for correct base cases in terms of subproblems
- 1 point for correct solution in terms of subproblems

- 2 points for correct analysis of running time
- 3 points if correct algorithm is $O(nk + n^2)$
- Partial credit may be awarded

Problem 8-5. [40 points] **Princess Plum**

Princess Plum is a video game character collecting mushrooms in a digital haunted forest. The forest is an $n \times n$ square grid where each grid square contains either a tree, mushroom, or is empty. Princess Plum can move from one grid square to another if the two squares share an edge, but she cannot enter a grid square containing a tree. Princess Plum starts in the upper left grid square and wants to reach her home in the bottom right grid square[3]. The haunted forest is scary, so she wants to reach home via a **quick path**: a route from start to home that goes through at most $2n - 1$ grid squares (including start and home). If Princess Plum enters a square with a mushroom, she will pick it up. Let k be the maximum number of mushrooms she could pick up along any quick path, and let a quick path be **optimal** if she could pick up k mushrooms along that path.

(a) [15 points] Given a map of the forest grid, describe an $O(n^2)$-time algorithm to return the number of distinct optimal quick paths through the forest, assuming that some quick path exists.

Solution:

1. **Subproblems**
   - Let upper left grid square be $(1, 1)$ and bottom right grid square be $O(n, n)$
   - Let $F[i][j]$ denote the contents of grid square (i, j)
   - Define two types of subproblems: one for optimal mushrooms and one to count number of paths
   - $k(i, j)$: the maximum number of mushrooms to reach grid square (i, j) from grid square $(1, 1)$ on a path touching $i + j - 1$ squares
   - $x(i, j)$: the number of paths from $(1, 1)$ to (i, j) collecting $k(i, j)$ mushrooms and touching $i + j - 1$ squares
2. **Relate**
   - A path touching $i + j - 1$ squares ending at (i, j) must extend a path touching $i + j - 2$ squares to either its left or upper neighbor
   - If $F[i][j]$ contains a tree:
     - There are no paths to (i, j)
     - $k(i, j) = -\infty$ and $x(i, j) = 0$
   - Otherwise:
     - Let $m(i, j)$ be 1 if $F[i][j]$ is a mushroom and 0 otherwise
     - $k(i, j) = m(i, j) + \max(k(i - 1, j), k(i, j - 1))$

[3] Assume that both the start and home grid squares are empty.

- $x(i,j) = \sum \left\{ \begin{array}{ll} 0 & \text{always} \\ x(i-1,j) & \text{if } k(i-1,j) + m(i,j) = k(i,j) \\ x(i,j-1) & \text{if } k(i,j-1) + m(i,j) = k(i,j) \end{array} \right\}$
- Subproblem $k(i,j)$ only depends on k subproblems with strictly smaller $i+j$, so acyclic
- Subproblem $x(i,j)$ only depends on x subproblems with strictly smaller $i+j$ (and k subproblems which do not depend on x subproblems), so acyclic

3. **Base**
   - $k(1,1) = 0$, no mushrooms to start
   - $x(1,1) = 1$, one path at start
   - negative grid squares impossible
   - $k(0,i) = k(i,0) = -\infty$ for $i \in \{0,\ldots,n\}$
   - $x(0,i) = x(i,0) = 0$ for $i \in \{0,\ldots,n\}$
4. **Solution**
   - $x(n,n)$ by definition
5. **Time**
   - # subproblems: $2(n+1)^2 = O(n^2)$, $k(i,j)$ and $x(i,j)$ for $i,j \in \{0,1,\ldots,n\}$
   - work per subproblem: $O(1)$
   - $O(n^2)$ running time

Rubric:

- If two dynamic programs defined, split evaluation of rubric items between them
- 4 points for a correct subproblem description
- 3 points for a correct recursive relation
- 1 points for indication that relation is acyclic
- 1 point for correct base cases in terms of subproblems
- 1 point for correct solution in terms of subproblems
- 2 points for correct analysis of running time
- 3 points if correct algorithm is $O(n^2)$
- Partial credit may be awarded

(b) [25 points] Write a Python function `count_paths(F)` that implements your algorithm from (a). You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.

Solution:

```
def count_paths(F):
    n = len(F)
    K = [[-float('inf')]*(n + 1) for _ in range(n + 1)]      # init K memo
    X = [[0]*(n + 1) for _ in range(n + 1)]                   # init X memo
    for i in range(1, n + 1):                      # bottom-up dynamic program
        for j in range(1, n + 1):
            if F[i - 1][j - 1] == 't':             # base case
                continue
            if i == 1 and j == 1:                  # base case
                K[1][1], X[1][1] = 0, 1
                continue
            if F[i - 1][j - 1] == 'm':             m = 1
            else:                                  m = 0
            K[i][j] = m + max(K[i - 1][j], K[i][j - 1])
            if K[i - 1][j] + m == K[i][j]:  X[i][j] += X[i - 1][j]
            if K[i][j - 1] + m == K[i][j]:  X[i][j] += X[i][j - 1]
    return X[n][n]
```