Problem Set 8

All parts are due on April 27, 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`.

Problem 8-1. [15 points] **Sunny Studies**

Tim the Beaver needs to study for exams, but it's getting warmer, and Tim wants to spend more time outside. Tim enjoys being outside more when the weather is warmer: specifically, if the temperature outside is t integer units above zero, Tim's happiness will increase by t after spending the day outside (with a decrease in happiness when t is negative). On each of the n days until finals, Tim will either study or play outside (never both on the same day). In order to stay on top of coursework, Tim resolves never to play outside more than two days in a row. Given a weather forecast estimating temperature for the next n days, describe an $O(n)$-time dynamic programming algorithm to determine which days Tim should study in order to increase happiness the most.

Solution:

1. Subproblems
   - Let $t(i)$ be the temperature on day i
   - $x(i)$: the maximum happiness increase possible during days i to n
2. Relate
   - Guess whether studying or playing on day i will yield more happiness
   - If study, no change in happiness, but may study or play on the next day, $x(i+1)$
   - If play, change happiness by $t(i)$ and either
     - study on day $i+1$, $t(i) + x(i+2)$, or
     - play on day $i+1$ and study on $i+2$, $t(i) + t(i+1) + x(i+3)$
   - $x(i) = \max\{x(i+1), t(i) + x(i+2), t(i) + t(i+1) + x(i+3)\}$ for $i \in \{1, \ldots, n-1\}$
   - Subproblem $x(i)$ only depends on subproblems with strictly larger i, so acyclic
3. Base
   - $x(n) = \max\{0, t(n)\}$, if only one day, either play or study
   - $x(n+1) = x(n+2) = 0$, if no more days, can't increase happiness
4. Solution
   - $x(1)$ is the maximum happiness increase possible from day 1 to day n
   - Store parent pointers to reconstruct days studied

5. Time
   - # subproblems: $x(i)$ for $i \in \{1, \ldots, n+2\}$, $n+2 = O(n)$
   - work per subproblem: $O(1)$ (constant branching factor)
   - $O(n)$ running time

Rubric:

- 4 points for 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
- 3 points if correct algorithm is $O(n)$
- Partial credit may be awarded

Problem 8-2. [15 points] **Selling Smoothies**

Little Tan Trotting Hood is a young entrepreneur. Her grandmother, Granny, makes delicious smoothies, and Little Tan wants to sell them at the local farmer's market. Granny has a diverse set of n bottles in her cupboard: one bottle b_i that holds exactly i milliliters of smoothie for each $i \in \{1, \ldots, n\}$. Little Tan has conducted some market research to determine how much each bottle could sell for at the market when completely filled with smoothie. Granny has enough fruit to make at most $3n$ milliliters of smoothie. Given a revenue estimate r_i for each bottle b_i, describe an $O(n^2)$-time dynamic programming algorithm to determine which bottles to fill completely and sell in order maximize projected revenue at the market (partially filled bottles may not be sold).

Solution:

1. Subproblems
   - $x(i, s)$: the maximum revenue possible by filling bottles b_i to b_n with s mL of smoothie
2. Relate
   - Guess whether or not to fill bottle b_i with smoothie
   - If not fill, gain no revenue but may fill remaining bottles with s mL of smoothie
   - If fill, gain revenue r_i and may fill remaining bottles with $s - i$ mL of smoothie
   - $x(i, s) = \begin{cases} \max\{x(i+1, s), r_i + x(i+1, s-i)\} & \text{if } s \geq i \\ x(i+1, s) & \text{if } s < i \end{cases}$ for $i \in \{1, \ldots, n\}$
   - Subproblem $x(i, s)$ only depends on subproblems with strictly larger i, so acyclic
3. Base

- $x(n+1, s) = 0$ for all $s \in \{0, \ldots, n\}$, no more bottles to fill!

4. Solution
   - $x(1, 3n)$ is the maximum revenue possible by filling bottles b_1 to b_n with n mL of smoothie
   - Store parent pointers to reconstruct bottles filled
5. Time
   - # subproblems: $x(i, s)$ for $i \in \{1, \ldots, n+1\}$ and $s \in \{0, \ldots, 3n\}$, $(n+1)(3n+1) = O(n^2)$
   - work per subproblem: $O(1)$
   - $O(n^2)$ running time

Rubric:

- 4 points for 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
- 3 points if correct algorithm is $O(n)$
- Partial credit may be awarded

Problem 8-3. [15 points] **Counting Courses**

Lance Weaklegs organizes the annual bike race in Gridville, a city laid out on an orthogonal grid of n streets, orthogonally crossing m avenues. Streets run East-West and are numbered sequentially $(1, \ldots, n)$ from South to North, while avenues run North-South and are numbered sequentially $(1, \ldots, m)$ from East to West. The race always occurs along some **North-Western course** through the city: a route along streets and avenues which: (1) begins at the intersection of 1st St. and 1st Ave., (2) ends at the intersection of n-th St. and m-th Ave, and (3) always travels North on avenues and West on streets (but may switch between them at intersections). However, some roads are closed for maintenance between certain intersections and may not be used in this year's race course. Lance wants to know how many different routes are possible. Given a list of all road closures, describe an $O(nm)$-time dynamic programming algorithm to count the number of possible North-Western courses through Gridville that avoid the road closures.

Solution:

1. Subproblems
   - $x(s, a)$: number of North-Western courses ending at s-th St. and a-th Ave.

2. Relate
   - Let $S(s, a)$ be 0 if avenue south of (s, a) (from $(s, a-1)$) is closed, and 1 otherwise
   - Let $E(s, a)$ be 0 if street east of (s, a) (from $(s-1, a)$) is closed, and 1 otherwise
   - Number of North-Western courses to (s, a) is number from south plus number from east
   - $x(s, a) = S(s, a)x(s, a-1) + E(s, a)x(s-1, a)$ for $s \in \{1, \ldots, n\}$, $a \in \{1, \ldots, m\}$
   - Subproblem $x(s, a)$ only depends on subproblems with strictly smaller $s + a$, so acyclic
3. Base
   - $x(1, 1) = 1$, path from start exists!
   - $x(s, 0) = 0$ for all $s \in \{0, \ldots, n\}$, out of grid!
   - $x(0, a) = 0$ for all $a \in \{0, \ldots, m\}$, out of grid!
4. Solution
   - $x(n, m)$ is number of North-Western courses ending at n-th St. and m-th Ave.
5. Time
   - # subproblems: $x(s, a)$ for $s \in \{0, \ldots, n\}$ and $a \in \{0, \ldots, m\}$, $(n+1)(m+1) = O(nm)$
   - work per subproblem: $O(1)$
   - $O(nm)$ running time

Rubric:

- 4 points for 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
- 3 points if correct algorithm is $O(n)$
- Partial credit may be awarded

Problem 8-4. [15 points] **Capricious Contagion**

A strange and highly infectious virus has been spreading across the globe: it is deadly to some, yet others seem to be immune. CDC investigator Dr. Leah Boegist has found the DNA sequence of the virus to be string V. A **DNA sequence** is represented by a string of bases, with each base represented by a letter, either `A`, `C`, `G`, or `T`. Dr. Boegist observes that if the virus's DNA string V appears as a subsequence[1] of a patient's DNA sequence k or more times, then that patient will be immune to the virus and never get sick. Given V, integer k, and the DNA sequence S of a patient,

[1] Subsequences are not necessarily contiguous!

describe an $O(|V||S|)$-time dynamic programming algorithm to determine whether the patient is in danger of ever getting sick. For example, if $V =$ `CAT` with $k = 2$, a patient with DNA $S =$ `TAGCATGTAG` will be immune from getting sick, because V appears twice as a subsequence of S.

Solution:

1. Subproblems
   - $x(i, j)$: number of times suffix `V[i:]` appears as a subsequence of suffix `S[j:]`
2. Relate
   - If `V[i]` and `S[j]` are same, then can use `S[j]` or not
   - If `V[i]` and `S[j]` are not same, then it suffices to check for `V[i:]` in `S[j + 1:]`
   - $x(i,j) = \begin{cases} x(i,j+1) + x(i+1,j+1) & \text{if } V[i] = S[j] \\ x(i,j+1) & \text{otherwise} \end{cases}$
   - Subproblem $x(i, j)$ only depends on subproblems with strictly larger j, so acyclic
3. Base
   - $x(i, |S|) = 0$ for $i \in \{0, \ldots, |V| - 1\}$, no more patient DNA
   - $x(|V|, j) = \begin{cases} 1 & \text{if } V[|V| - 1] = S[j - 1] \\ 0 & \text{otherwise} \end{cases}$ for $j \in \{1, \ldots, |S|\}$, match last virus
4. Solution
   - $x(0, 0)$ is number of times `V[0:] = V` appears as a subsequence of `S[0:] = S`
   - So return whether $x(0, 0) \geq k$
5. Time
   - # subproblems: $x(i, j)$ for $i \in \{0, \ldots, |V| - 1\}$ and $i \in \{0, \ldots, |S|\}$, so $O(|V||S|)$
   - work per subproblem: $O(1)$
   - $O(|V||S|)$ running time

```
def is_immune(V, S, k):
    x = {}
    for i in range(len(V)):
        x[(i, len(S))] = 0
    for j in range(1, len(S) + 1):
        if V[-1] == S[j - 1]:
            x[(len(V), j)] = 1
        else:
            x[(len(V), j)] = 0
    for j in range(len(S) - 1, -1, -1):
        for i in range(len(V)):
            if V[i] == S[j]:
                x[(i,j)] = x[(i,j + 1)] + x[(i + 1, j + 1)]
            else:
                x[(i,j)] = x[(i,j + 1)]
    return x[(0,0)] >= k
```

Rubric:

- 4 points for 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
- 3 points if correct algorithm is $O(n)$
- Partial credit may be awarded

Problem 8-5. [40 points] **Annoying Boxes**

You want to give a small gift to a friend, but not for nothing: they'll have to work for it! You will put the gift inside a box, and then fit that box inside another box, repeatedly as many times as you can. The gift will have **wrapping annoyance** k if it is nested inside k boxes. You have access to n rectangular boxes of known dimensions. The **dimensions** of box b_i is represented by a triple of integers (h_i, w_i, ℓ_i), corresponding to its height, width, and length respectively. One box can **fit inside** another box if the second box can be oriented so that all of its dimensions are strictly smaller than the dimensions of the first box. You may assume that the gift will fit inside any box. For example, a box with dimensions $(3, 6, 5)$ can fit inside a box with dimensions $(7, 4, 6)$ (after proper re-orientation), which would constitute a wrapping with annoyance 2.

(a) [15 points] Given a list B of triples representing box dimensions, describe an $O(|B|^2)$-time dynamic programming algorithm to determine a set of boxes that will maximize wrapping annoyance.

Solution: First we note that we can determine whether box b_i can fit inside b_j by sorting each triple so they are increasing, and then checking that each dimension in b_j is larger than its ordered counterpart in b_i. So we first sort each b_i increasing, which takes constant time for each of the n triples. Then if b_i can fit inside b_j, b_i must appear before b_j in a lexicographical sort (sorting first by smallest dimension, then middle dimension, then largest dimension), and any nesting sequence of boxes will appear as a subsequence of the list of boxes ordered this way. So sort the boxes lexicographically in $O(n \log n)$ time, and now assume that the boxes b_i are ordered lexicographically.

1. Subproblems
   - $x(i)$: maximum wrapping annoyance, using box b_i and only boxes from b_1 to b_i
2. Relate
   - Check which next box will maximize annoyance

- $x(i) = \max\{1 + x(j) \mid j \in \{1, \ldots, i-1\}$ and b_j fits inside $b_i\}$ for $i \in \{2, \ldots, |B|\}$
- Subproblem $x(1)$ only depends on subproblems with strictly smaller i, so acyclic

3. Base
   - $x(1) = 1$, smallest box alone can only have wrapping annoyance one
4. Solution
   - Subset ends at i for whichever $x(i)$ is maximum for $i \in \{1, \ldots, |B|\}$, find in $O(|B|)$ time
   - Store parent pointers to reconstruct which boxes were used
5. Time
   - # subproblems: $x(i)$ for $i \in \{1, \ldots, |B|\}$, so $O(|B|)$
   - work per subproblem: $O(|B|)$
   - $O(|B|^2)$ running time

Rubric:

- 4 points for 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
- 3 points if correct algorithm is $O(n)$
- Partial credit may be awarded

(b) [25 points] Write a Python function `annoy(B)` that implements your algorithm. The output of your algorithm should be a list of indices $A = [a_0, \ldots, a_{k-1}]$ such that box $B[a_{i-1}]$ can fit inside box $B[a_i]$ for all $i \in \{1, \ldots, k-1\}$, and k is the maximum wrapping annoyance possible using boxes from B. You **are allowed** to use Python's built-in sort functions if desired. You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.

Solution:

```
def fits_inside(b1, b2):
    # assumes dimensions b1 and b2 are each sorted
    return all([b1[i] < b2[i] for i in range(len(b1))])

def annoy(B):
    n = len(B)
    B_ = [sorted(b) for b in B]
    C = [i for i in range(n)]
    C.sort(key = lambda i: B_[i])
    x = [1 for _ in range(n)]
    inner = [None for _ in range(n)]
    best = 0
    for i in range(n):
        for j in range(i):
            if fits_inside(B_[C[j]], B_[C[i]]):
                if 1 + x[j] > x[i]:
                    x[i] = 1 + x[j]
                    inner[i] = j
        if x[i] > x[best]:
            best = i
    A = []
    while not (best is None):
        A.append(C[best])
        best = inner[best]
    A.reverse()
    return A
```