Problem Set 9

All parts are due on April 26, 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`.

Problem 9-1. [25 points] **Bidirectional Search**

(a) [5 points] **General Optimality:** Breadth-first search determines a shortest path from s to t in an unweighted graph $G = (V, E)$ in $O(|V| + |E|)$ time. Show that the running time of this algorithm is asymptotically optimal applied to general graphs, by describing an unweighted undirected graph $G = (V, E)$ containing vertices s and t for which the shortest path from s to t has size $\Omega(|V| + |E|)$.

Solution: Consider a graph that is simply a chain of vertices (v_{i-1}, v_i) for i ranging from 1 through n. There are $n + 1$ vertices and n edges. If $s = v_0$ and $t = v_n$, then the shortest (and only) path between s and t is $n = |E| = \Omega(|V| + |E|)$ edges long, so BFS must search at least that many levels.

Rubric:

- 3 points for a valid description of a correct graph
- 2 points for running time analysis

(b) [10 points] **Specific Improvement:** For many graphs, single source breadth-first search can be done faster by making the search **bidirectional**: perform a breadth-first search separately from s and t, alternating the exploration of level i from s, then level i from t, then level $i + 1$ from s, etc., until the frontier levels from the two searches intersect, returning the concatenation of the two shortest paths found[1]. Consider the family of unweighted undirected graphs having bounded degree b, where a shortest path from vertices s to t is known to contain d edges. Describe a graph from this family for which breadth-first search from s takes at least $\Omega((b-1)^d)$ time, while bidirectional breadth-first search from s and t takes at most $O(b^{d/2})$ time.

Solution: Consider a tree rooted at s with lower branching factor $b - 1$ and depth d. Let t be any leaf of the tree, so that there is a unique path from s to t. Let the vertices on this path be labeled $s = v_0$ through $t = v_d$. Regular breath-first search from s would explore the entire tree, which has size $\Omega((b-1)^d)$, so breath-first search will take at least that long to touch all nodes. On the other hand, a bidirectional search would terminate a breadth-first search from s at vertex $v_{d/2}$, exploring a subtree of depth $d/2$ of size $O((b-1)^{d/2})$. The search from t will also explore out $d/2$ levels before reaching node $v_{d/2}$. Each node at level $i > 0$ in the search from t has one

[1] An interesting exercise is to prove that this path is indeed a shortest path.

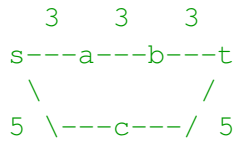
parent and at most $b-1$ unexplored neighbors to add to level $i+1$, so the search from t also touches at most $O((b-1)^{d/2})$ nodes. Thus bidirectional search runs in at most $O((b-1)^{d/2}) = O(b^{d/2})$ time.

Rubric:

- 3 points for a valid description of a correct graph
- 3 points for $\Omega((b-1)^d)$ analysis
- 4 points for $O(b^{d/2})$ analysis
- Partial credit may be awarded

(c) [10 points] **Bidirectional Dijkstra:** A similar algorithm can speed up Dijkstra on graphs containing non-negative edge weights, by running Dijkstra separately from s and t until their 'frontiers' intersect. Each Dijkstra search maintains a priority queue (e.g., Q_s) initially containing all vertices, and the set of vertices already extracted from the queue (e.g., $V \setminus Q_s$) represents the frontier of that search. To interleave their execution, extract and process a vertex from a search queue if its minimum path weight estimate to its source is smallest in both queues. Naively one might think the same stopping condition as bidirectional BFS might work here: stop the search as soon as the same vertex appears in the frontier of both searches, returning the concatenation of the two shortest paths found; however this stopping condition does not always return a shortest path[2]. Describe a weighted undirected graph on five vertices such that running bidirectional Dijkstra between two of the vertices using the above stopping condition returns a path that does not have minimum weight.

Solution: Consider the graph below, where there are two simple paths from s to t: one consisting of three edges of weight 3, and another with two edges of weight 5.



Bidirectional Dijkstra using the incorrect stopping condition would add $(3, a)$ and $(5, c)$ to Q_s, then $(3, b)$ and $(5, c)$ to Q_t. Then, pop $(3, a)$ so Q_s contains $(5, c)$ and $(6, b)$. Similarly, Q_t will contain $(5, c)$ and $(6, a)$ after popping $(3, b)$. Then popping $(5, c)$ from both Q_s and Q_t, would declare the shortest path (s, c, t), which is not a minimum weight path.

Rubric:

- 5 points for description/drawing of graph
- 5 points for argument that stopping condition does not find minimum weight path
- Partial credit may be awarded

[2] A correct stopping condition for bidirectional Dijkstra is to: maintain a vertex v whose minimum weight path estimates to s and t sum to the smallest value $\mu = d_s(v) + d_t(v)$, and stop the search when μ is no larger than the sum of the smallest minimum path estimates remaining in both searches' priority queues.

Problem 9-2. [20 points] **Katalan**

Kat A. Lan is taking a class in combinatorics. Having already learning about binary trees in 6.006, she is intrigued to learn that something called the nth **Catalan number** C_n corresponds to the number of rooted binary trees containing $n-1$ non-leaf vertices, where every non-leaf vertex has exactly two children. There are many ways to compute Catalan numbers, but Kat is intrigued by the following recursive definition:

$$C_1 = 1 \qquad C_n = \sum_{i=1}^{n-1} C_i C_{n-i}.$$

Kat writes the following code to compute Catalan numbers according to this definition.

```
def catalan(n):
    if n == 1:
        return 1
    out = 0
    for i in range(1, n):
        out += catalan(i) * catalan(n - i)
    return out
```

(a) [5 points] Is Kat's code correct? Does Kat's code run in polynomial time?

Solution: Kat's code is correct as it implements exactly the formula provided. However, a call to `catalan(n)` performs two recursive calls to `catalan(n - 1)`, as well as many other recursive calls. Thus the running time is lower bounded by $T(n) > 2T(n-1)$, so $T(n) = \Omega(2^n)$, which is not polynomial.

Rubric:

- 2 points saying that code is correct
- 3 points for arguing that code does not run in polynomial time

(b) [5 points] Describe a directed graph of computational dependencies of the first n Catalan numbers based on the recursive definition above. How many vertices and edges are in this graph?

Solution: The graph of dependencies consists of n vertices, with vertex v_i representing Catalan number C_i. For vertex v_k, there is a directed edge from v_k to vertex v_i for all $1 \leq i < k$, since the recursive definition depends on every smaller Catalan number. Thus, there are $\sum_{i=1}^{n-1} i = n(n-1)/2$ directed edges in the graph.

Rubric:

- 2 points for description of graph
- 1 point for number of vertices
- 2 points for number of edges

(c) [10 points] Improve Kat's code by writing a Python function to compute the nth Catalan number in $O(n^2)$ time. Note there are faster ways to compute Catalan numbers; we are looking for a **dynamic programming** solution based on the **provided recursive definition**. Your code may use either a bottom up or top down approach. Include your code in your PDF submission. Your code should not be more than 15 lines long (our solution is under 10 lines).

Solution:

```
# iterative solution
def catalan_i(n):
    memo = [None for _ in range(n + 1)]
    memo[1] = 1
    for i in range(2, n + 1):
        memo[i] = 0
        for j in range(1, i):
            memo[i] += memo[j] * memo[i - j]
    return memo[n]
```

```
# recursive solution
def catalan_r(n, memo = None):
    if memo is None:
        memo = [None for _ in range(n + 1)]
        memo[1] = 1
    if memo[n] is None:
        memo[n] = 0
        for i in range(1, n):
            memo[n] += catalan_r(i, memo) * catalan_r(n - i, memo)
    return memo[n]
```

Rubric:

- 10 points for code that looks correct (graders need not run the code)
- Partial credit may be awarded

Problem 9-3. [20 points] **Dynamic Consulting**

Solve each of the following problems using **dynamic programming**. Specifically: define subproblems, relate them recursively, show that subproblems are acyclic, state base cases, show how to compute the solution, and show that your dynamic program achieves the requested running time.

(a) [10 points] **Future Investing:** Tiffany Bannen stumbles upon a lottery chart dropped by a time traveler from the future, which lists winning lottery numbers and cash payout for the next n days. Tiffany is excited to use this information to make money, but is worried that if she plays winning numbers every day, the organizers of the lottery will get suspicious. As such, she decides to limit herself to playing the lottery at most twice in any seven day period. Describe a $O(n)$ time algorithm to determine the maximum amount of lottery winnings Tiff can get in the next n days, subject to playing the lottery at most twice in any seven day period.

Solution:

1. **Subproblems**
   - Let $L(i)$ be the cash payout of playing the lottery on day $i \in [0, n-1]$
   - $x(i, j)$: maximum lottery winnings playing on suffix of days from i to $n-1$, assuming play on day i and next allowable play is day $i + j$ for $j \in [1, 6]$
   - It is never optimal to go 11 days without playing the lottery, as playing on the 6th day would be valid and strictly increase winnings
2. **Relate**
   - Guess the next day Tiffany plays the lottery
   - The next play can be between j and 11 days after i, on day $i + k$
   - If next play on $i + k$ for $k \in [1, 6]$, next allowable play is on day $i + 7$
   - If next play on $i + k$ for $k \in [7, 11]$, next allowable play is on day $i + k + 1$
   - $x(i, j) = \max_{k \in [j, 11]} L(i) + x(i + k, \max(1, 7 - k))$
3. **DAG**
   - Subproblems $x(i, j)$ only depend on strictly larger i, so acyclic
   - Solve in order of decreasing i, then any order of j
   - Base case: $x(i, j) = 0$ for $n \leq i$ (no days to play, no winnings)
4. **Evaluate**
   - Solve subproblems via recursive top down or iterative bottom up
   - Solution to original problem is maximum of $x(i, 1)$ for $i \in [0, n-1]$
   - (Can store parent pointers to reconstruct days played)
5. **Analysis**
   - # subproblems: $6 \times O(n)$
   - work per subproblem: $O(11) = O(1)$
   - $O(n)$ 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) [10 points] **Light Show:** Glark Criswold recently purchased a programmable string of n LEDs to accentuate his house for the holidays. Each LED on the string can be independently assigned a different color at high frequency to simulate elaborate visual animations. The programmable interface of the LED string allows a user to update any contiguous subsequence of LEDs along the string with a set of new colors values. However, because of issues concerning electrical impedance and caching, modifying contiguous subsequences of different length takes different amounts of time. For example, updating the color of a single LED might take $2\mu s$, updating a subsequence of ten LEDs might take $30\mu s$, and updating a subsequence of twenty LEDs might take $20\mu s$ (these values are not necessarily proportional to the number of updated LEDs). Given the current color a_i and desired color b_i of LED i, for each of the n LEDs, and the time t_k of performing an update on any contiguous subsequence of k LEDs, describe an $O(n^2)$ time algorithm to determine the minimum time to update each a_i to b_i. (Note that if $a_i = b_i$, LED i does not need to be updated.)

Solution:

1. **Subproblems**
   - Assume that times t_k to update k adjacent LEDs ranges from $k \in [1, m]$ with $m \leq n$
   - $x(i)$: minimum time to update prefix of LEDs ending at LED i
   - If an optimal set of updates goes outside range of a subproblem, updates can be shifted to stay within range $[0, n-1]$ since $m \leq n$
2. **Relate**
   - If $a_i = b_i$, then $x(i) = x(i-1)$, as LED i does not need to be updated
   - Otherwise, we guess an update intersecting the subproblem range
   - Either the range can be completely covered by a single update, or an update ending at LED i partially covers the range (this is because $x(i)$ must increase monotonically with i)
   - $x(i) = \begin{cases} x(i-1) & \text{if } a_i = b_i \\ \min\left(\min_{k \in [i,m]} t_k, \min_{k \in [1,i-1]} t_k + x(i-k)\right) & \text{otherwise} \end{cases}$
3. **DAG**
   - Subproblems $x(i)$ only depend on strictly smaller i, so acyclic
   - Solve in order of increasing i

- Base case: $x(0) = 0$ (we could alternatively set $x(i) = 0$ for $i \leq 0$ to simplify the recurrence relation when $a_i \neq b_i$ to $x(i) = \min_{k \in [1,m]} t_k + x(i-k)$)

4. **Evaluate**
   - Solve subproblems via recursive top down or iterative bottom up
   - Solution to original problem is $x(n)$
   - (Can store parent pointers to reconstruct updates)
5. **Analysis**
   - # subproblems: $O(n)$
   - work per subproblem: $O(m) = O(n)$
   - $O(n^2)$ 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

Problem 9-4. [35 points] **Choosing Prizes**

Alyssa P. Hacker just won the game show, "Who Wants to Be A Theoretical Computer Scientist!" For winning the contest, she is shown a row of prizes. Alyssa is allowed to choose and take home any subset of the prizes, so long as no two chosen prizes are next to each other in the row. Each prize is displayed with a monetary value, so naturally Alyssa wants to maximize the total value of the prizes she selects. For instance, if the prize values in the row are: [14, 30, 27, 4, 5, 15, 1], Alyssa should pick the prizes worth 14, 27, and 15, as this is a subset with the maximum total.

(a) [10 points] Design a linear time algorithm to determine which prizes Alyssa should choose to maximize the total value of the prizes she selects.

Solution:

1. **Subproblems**
   - Let $p(i)$ be the value of prize i
   - $x(i)$: maximum possible value of prizes chosen from first i prizes
2. **Relate**
   - Prize i is either chosen or not chosen
   - If it is chosen, the previous prize could not have been chosen
   - $x(i) = \max(x(i-1), p(i) + x(i-2))$
3. **DAG**
   - Subproblems $x(i)$ only depend on strictly smaller i, so acyclic
   - Solve in order of increasing i
   - Base case: $x(i) = 0$ for $i \leq 0$ (no value if no prizes)
4. **Evaluate**
   - Solve subproblems via recursive top down or iterative bottom up
   - Solution to original problem is $x(n)$
   - Store parent pointers and whether prize was chosen or not to reconstruct maximizing subsequence of prizes
5. **Analysis**
   - # subproblems: $O(n)$
   - work per subproblem: $O(1)$
   - $O(n)$ 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) [25 points] Write the Python function `choose_prizes(prize_values)` that implements your algorithm from part (a). The input `prize_values` is a list of length n containing the prize values, which are positive integers. Your function should return a list of the indices between 0 and $n - 1$ inclusive not containing any consecutive values, corresponding to the prizes that Alyssa should select. Submit your code online at `py.mit.edu/6.006`.

Solution:

```
def choose_prizes(prize_values):
    '''
    Find list of prizes to choose with largest total value.
    Input:  list of positive integers corresponding to prize values
    Output: list of indices corresponding to prize values to select.
    No two consecutive prize indices should be included.
    '''
    n = len(prize_values)
    x = [0] * n                         # memo
    parent = [None] * n                 # parent pointers
    choose = [True] * n                 # chose prize? default True
    x[0] = prize_values[0]              # base case
    for i in range(1, n):               # dynamic program
        x[i] = prize_values[i]          # try choosing prize i
        if 0 <= i - 2:                  # if i - 2 in range
            x[i] += x[i - 2]            # chose? add in nonadjacent subproblem
            parent[i] = i - 2
        if x[i] < x[i - 1]:             # check if better to not choose prize i
            x[i] = x[i - 1]
            parent[i] = i - 1
            choose[i] = False
    i = n - 1                           # start at solution x[n - 1]
    prizes = []                         # initialize list of prizes
    while i is not None:                # walk back to start
        if choose[i]:                   # if optimal chose i
            prizes.append(i)            # add i to prize list
        i = parent[i]
    return prizes
```