Problem Set 7

Name: Thomas Ren

Collaborators: None

Problem 7-1. *We will solve this problem following the **SRT BOT** template.*

•Subproblem

–Since Rep. Torr needs at least $\lfloor \frac{D}{2} \rfloor + 1$ to win, and the higher the **delegate count** the better.

–We will try to find the highest **delegate count** Rep. Torr can get, then use that to decide whether she can win this primary contest.

–Define subproblem using prefixes. (Using suffixes is nearly the same)

–$x(i)$: the maximum delegate count Rep. Torr can get up to day i.

•Relation

–There is a condition that make this relation not directly iterative.

–We have to **guess** whether at day i, Rep. Torr will launch a campaign.

–If she does, then at day $i-1$ and day $i-2$, she cannot launch one.

–If she doesn't, then she can freely decide whether to launch one from day 0 to day $i-1$.

–$x(i) = max\{d_i + z_{i-1} + z_{i-2} + x(i-3), z_i + x(i-1)\}$

•Topological Order

–$x(i)$ depends on $x(j)$ where j is strictly smaller than i.

–So forms a topological order (DAG).

•Base Case

–Since it's only meaningful when $n \geq 0$, and $x(0)$ depends on $x(j), \quad j \geq -3$.

–$x(n-1) = x(n-2) = x(n-3) = 0 \Longrightarrow x(0) = d_0$

•Original Problem

–The final decision of whether Rep. Torr can win depends on her **delegate count** after day n.

–Compute $x(n)$ iterative bottom up.

–If $x(n) \geq \lfloor \frac{D}{2} \rfloor + 1$, return True that she can win if she launches campaign properly.

–Otherwise, return False that no matter how she use the campaign, she will lose.

- **Time**
  - Number of subproblems: $O(n)$
  - Operations per subproblem: $O(1)$
  - $O(n)$ running time.

Problem 7-2. *We will solve this problem following the **SRT BOT** template.*

- Before we dive into the solution, we will make several observations which would help us better understand the problem.
  - The order to arrange the tigers depend on its **age** a_i and then its **size** s_i.
  - The order to allocate cages depend on its **distance** d_j and then its **capacity** c_j.
  - Kiger wants the older tiger to have smaller distance, then arrange the capacity that's possible to minimize the total discomfort.
  - Sort the tigers by **decreasing ages**, the cages by **increasing distances**, so that the oldest goes to the closest cage.
  - Designate the sorted tigers' sequence by **T** and cages' sequence by **C**.
- **Subproblem**
  - $x(i, j)$: the minimized total discomfort for $T[i :]$ and $C[j :]$
- **Relation**
  - Remember that T is in decreasing order and C is in increasing order.
  - We brute-force the situations to be binary.
  - Whether $T[i]$ stays at $C[j]$? **Guess!**
  - If stays, we need to consider the inequality of s_i and c_j. Computed as $ReLU(s_i - c_j) + x(i + 1, j + 1)$.
  - If doesn't stay, we can only recurse by incrementing j, since if i_{th} tiger doesn't stay at $C[j]$, the $i + 1_{th}$ tiger cannot stay at $C[j]$ otherwise $T[i]$ has no place to stay. Computed as $x(i, j + 1)$.
  - $x(i, j) = min\{ReLU(s_i - c_j) + x(i + 1, j + 1), x(i, j + 1)\}$ (Suffixes)
- **Topological Order**
  - The $x(i, j)$ depends on strictly larger $i + j$.
  - Although i may stay the same during recursive calls, the base case recursed to the end will cover this situation.
- **Base Case**
  - $x(n, j) = 0$, there is no more tigers we need to arrange, thus won't generate discomfort. We are happy with this that all tigers have places to stay and we don't care about wasting empty cages.
  - $x(i, n^2) = \infty$, cages have been all allocated and remain no room for tigers. We don't want this to happen, since all we need is to arrange tigers.
- **Original Problem**
  - When suffixes hit the end, i.e. $i = j = 0$.
  - Return $x(0, 0)$.
- **Time**

–Sort T and C costs $O(n \log n)$ and $O(n^2 \log n)$ separately.
–Number of subproblems: $O(n^3)$ $\longleftarrow \{(n+1) \times (n^2+1)\}$
–Operations per subproblem: $O(1)$
–$O(n^3)$ running time.

Problem 7-3. *We will solve this problem following the **SRT BOT** template.*

•*Observations*:

–Weights can be positive or negative, but forms a **DAG**.

–Optimization Problem? **Decision problem!** The weight of a path is either *even* or *odd*. So we can return boolean value.

–The parity of a path depend on subpaths. Specifically, the parity of path from *s* to *t* depends on the parity of path from s to u for $u \in Adj^-(t)$ and the parity of weight (u, t). **Solve the D.P. just like Bellman-Ford**.

–All weight integers can fit in a single machine word means that we can determine whether the path weight is odd or even in $O(1)$-time.

•**Subproblem**

–$G = (V, E, w)$ is a DAG, so all possible paths from s to t is a simple path.

–Recurse from $u \mid u \in Adj^-(t)$, then $v \mid v \in Adj^-(u)$, to s.

–For any node v and its incoming neighbour u, the parity of *path(s,v)* depends on the parity of *path(s,u)* and $w(u, v)$.
So, we have to be told about the parity of *path(s,u)*.

–$x(v, b)$: the number of paths from s to v, $b = 1$ if the all paths' weight are odd and $b = 0$ otherwise.

–For $v \in V$ and $b \in \{0, 1\}$

•**Relation**

–Locally brute force all possible states.

–For $x(v, b)$ and $u \mid u \in Adj^-(v)$, if $w(u, v)$ is odd, $x(v, 1) = x(u, 0), \quad x(v, 0) = x(u, 1)$. Otherwise, $x(v, 1) = x(u, 1), \quad x(v, 0) = x(u, 0)$. **With constraint** that the paths count goes through u.

–$x(v, i) = \sum\{x(u, parity(w(u, v) + i)) \mid u \in Adj^-(v)\}$

•**Topological Order**

–Subproblem on node v depending on its incoming neighbours.

–The graph is already a DAG.

•**Base Case**

–$x(s, 0) = 1$ Zero-weight is even.

–$x(s, 1) = 0$ Zero-weight is not odd.

–$x(v, 0) = x(v, 1) = 1$ for v that are not reachable from s.

•**Original Problem**

–$x(t, 1)$ tells the number of paths from s to t that have *odd weights*.

•**Time**

–Number of subproblems: $2|V|$ (for $b = 0\ and\ 1$)
–Operations per subproblem: $|Adj^{-}(v)|$ (for every incoming edge, compute its parity)
–Sums to $O(|V| + |E|)$ running time, which is linear to the graph size.

Problem 7-4. *We will solve this problem following the **SRT BOT** template.*

•Review the **ACG**(Alternate Coin Game) from Lecture 16.

•**Subproblem**

–Considering that once a slice of pizza is eaten, the other cannot have that slice of pizza. So, this is, again, a **zero-sum game**.

–We can exploit **subproblem expansion** to tackle this.

–Given a sequence of tastiness corresponding to the angle index, we duplicate the same sequence on both left and right side of the sequence, since the pizza slices is cyclic.

–The uneaten pizza slices is always contiguous, i.e. forms a subsequence.

–$x(i, j, p)$: the maximum tastiness Liza can get with uneaten pizza slices from angle α_i to α_j. If Liza is the chooser $p = 1$, otherwise $p = 2$.

–For $i \in \{0, 1, \cdots, 2n-1\}$, $j \in \{i+1, i+2, \cdots, 2n-1\}$, $p \in \{0, 1\}$.

•**Relation**

–For angle that's proper from α_i to α_j, only $t_i, \cdots, t_{j-1}$ are not zero.

–Every time Liza is the chooser, she wants to maximize the value she can get. As well as Lie.

–So, when Lie is the chooser, he tries to maximize his tastiness sum, that is to *minimize Liza's tastiness sum*, since it's a zero-sum game.

–$x(i, j, 0) = max\{x(max\{i, j-n\}, k, 1) + \sum_k^{k+n-1} t_m, \quad k \in \{i+1, \cdots, j\}\}$

–$x(i, j, 1) = min\{x(max\{i, j-n\}, k, 1), \quad k \in \{i+1, \cdots, j\}\}$

•**Topological Order**

–Depend on strictly smaller $j - i$. Forms a DAG.

•**Base Case**

–Dependency terminates when $j = i + 1$ leaves the last slice of pizza, that's the time to locally brute force our base case.

–$x(i, i+1, 0) = t_i$ (when Liza gets the last piece)

–$x(i, i+1, 1) = 0$ (when Lie gets the last piece)

–for $i \in \{0, 1, \cdots, 2n-1\}$

•**Original Problem**

–Liza starts as the chooser

–$x(0, 2n-1, 0)$

•**Time**

–Number of subproblems: $O(n^2)$

–Operations per subproblem: $O(n)$

–$O(n^3)$ running time.

Problem 7-5.

(a) This core of this problem is the same as in the **LIS**(Longest Increasing Subsequence), while here is the **LDS**(Longest Decreasing subsequence).
Although the dynamic programming to subproblems are quite similar, there exists many constraints. In the following algorithm describing, we will use LDS as a *black-box* and focus on other implementation details and constraints.

1. For a sequence of **length** n, LDS runs in $O(n^2)$-time. But we are only allowed for $O(k^2)$-time. What's more, we don't skip days, which means that we don't have to search for the day after tomorrow.
   So, instead of searching the whole sequence for the possible LDS, we just search for today's prices and tomorrow's. *Note that* we need to take the last day into account, since we may have indices out of range.
2. To get the **index** of the largest number rather than **the largest number**, we need to implement the *argmax(A)* function that returns the index of the largest item in a sequence.
3. The **LIS** described in Lecture 16 only returns the length of LIS, but here we need to actually return the LDS.
   Thus, we modify the function by maintaining a list of parent pointers initialized with **None**. This pointer will point to the next item's index in its LDS and the last item points to None, at when the loop terminates.
   By traversing the parent pointers until None, we append the items to the LDS list and return that.
4. Computing the LDS for p_i takes $O(nk^2)$-time, find the max of them takes $O(s)$-time. Resulting in the whole algorithm runs in $O(snk^2)$-time.

(b) Submit your implementation to `alg.mit.edu`.

```
def argmax(A):
    '''
    :param A: a list of numbers or a string
    :return: i: the index of the largest item
    '''
    max = -float('inf')
    max_index = 0
    for i, item in enumerate(A):
        if item>max:
            max = item
            max_index = i

    return max_index

def LDS(A):
    '''
```

```
    :param A: a list of items that are comparable
    :return: the longest decreasing subsequence
    '''
    a = len(A)
    x = [1]*a
    y = [None]*a
    for i in reversed(range(a)):
        for j in range(i,a):
            if A[j]<A[i]:
                x[i] = max(x[i],1+x[j])
                if x[i] == 1+x[j]:
                    y[i] = j

    parent_pointer = argmax(x)
    Longest_Subseq = []
    while parent_pointer is not None:
        Longest_Subseq.append(A[parent_pointer])
        parent_pointer = y[parent_pointer]

    return Longest_Subseq

def LDS_short_company(p,n,k):
    '''
    :param p: nk prices in n contiguous days for a company
    :param n: number of days
    :param k: number of prices per day
    :return: LDS that doesn't skip days
    '''

    a = len(p)
    x = [1]*a
    y = [None]*a

    for i in reversed(range(n)):                          # O(nk^2)
        for j in reversed(range(i*k,k*(i+1))):            # O(k^2)
            for l in range(j,min((i+2)*k,n*k)):           # O(k)
                if p[l]<p[j]:
                    x[j] = max(x[j],1+x[l])
                    if x[j] == 1+x[l]:
                        y[j] = l

    parent_pointer = argmax(x)                            # O(nk)
    Longest_Subseq = []
    while parent_pointer is not None:                     # O(nk)
        Longest_Subseq.append(p[parent_pointer])
        parent_pointer = y[parent_pointer]

    return Longest_Subseq


def short_company(C, P, n, k):
```

```
    '''
    Input:  C | Tuple of s = |C| strings representing names of companies
            P | Tuple of s lists each of size nk representing prices
            n | Number of days of price information
            k | Number of prices in one day
    Output: c | Name of a company with highest shorting value
            S | List containing a longest subsequence of
              | decreasing prices from c that doesn't skip days
    '''
    c = C[0]
    S = []
    ##################
    # YOUR CODE HERE #
    ##################
    C_lds = []
    C_subseq = []

    for p in P:                                     # O(snk^2)
        lds = LDS_short_company(p,n,k)              # O(nk^2)
        C_lds.append(len(lds))                      # O(1)
        C_subseq.append(lds)                        # O(1)

    c = C[argmax(C_lds)]                            # O(s)
    S = C_subseq[argmax(C_lds)]                     # O(s)

    return (c, S)
```