Assignment 5

Gradescope IDs: 605 & 557

All group members have read and followed the guidelines for academic conduct in CPSC 320. As part of those rules, when collaborating with anyone outside my group, (1) I and my collaborators took no record but names (and Gradescope information) away, and (2) after a suitable break, my group created the assignment I am submitting without help from anyone other than the course staff.

1. Ruler of My Non-Domain

1.1 Do Be So Naive

1.1.1

```
naive(n):
    return naiveHelper(n, 1)

naiveHelper(i, j):
    if i = 1:
        return 1
    else:
        return naiveHelper(i-1, j+1) + naiveHelper(i-1, j)
```

1.1.2

Assuming that storing one value of A takes constant space and that the matrix is not copied on each recursive call, a slightly less naive memoized version of this algorithm would take $\Theta(n^2)$ space as we would store $1 + 2 + ... + (n - 1) + n = n(n + 1)/2$ computed entries. In other words, the algorithm fills in all entries above (and including) the anti-diagonal of an $n \times n$ matrix, as shown below for n=4:

1	1	1	1
2	2	2	x
4	4	x	x
8	x	x	x

As $\Theta(n^2)$ entries are computed in constant time each, and the set-up work would take no greater than $O(n^2)$ time, the slightly less naive memoized algorithm would run in $\Theta(n^2)$ time.

1.1.3

```
for i = 1 to n:
    for j = 1 to n - i + 1:
        compute A[i,j]
```

1.2 Take a Memo!

As a specified solution index is not requested, we return the whole solution array.

```
memoized(C):
    A = array of length length(C)
    initialize each entry of A to null
    memoizedHelper(A, C, length(C))
    return A

memoizedHelper(A, C, i):
    if A[i] is null:
        if IsPow2(i):
            A[i] = 1
        else:
            min = infinity

            for j = ceil(i / 2) to i - 1:
                candidate = memoizedHelper(A, C, j) + C[i] - C[j]
                if candidate < min:
                    min = candidate

            A[i] = min
    return A[i]
```

1.3 Be a Dynamo!

```
dynamo(C):
    n = length(C)
    mid = floor(n/2)
    A = array of length n

    if n <= 9:
        return 1

    for i = 1 up to 4:
        A[i] = 1

    for i = n down to n - 4:
        A[i] = 1

    for i = 5 up to mid - 1:
        A[i] = min(C[i-1] + A[i-1], C[i-2] + A[i-2])

    for i = n down to mid + 1:
        A[i] = min(C[i+1] + A[i+1], C[i+2] + A[i+2])
```

```
    A[mid] = min(C[i-1] + A[i-1], C[i+1] + A[i+1])
    return A[mid]
```

2. Parking in Wonderland

2.1 Permission Accomplished

2.1.1

$C(n) = C(n - d_t) + p_t$

2.1.2

for $n <= 0$, $C(n) = 0$

2.1.3

Assuming that for all d_t in D, $d_t >= 1$, and that every p_t in P is positive. Using 1-based indexing and a memoized solution. If there were 0-day permits, we could remove them before processing, and use the resulting solution.

```
D = global array of durations, size k
P = global array of prices, size k
A = global array of yet unknown length, uninitialized

findIdealCost(n):
    A = empty array of size n
    initialize all entries in A to null
    return permitHelper(n)

permitHelper(i):
    if i <= 0:
        return 0

    else if A[i] is null:
        min = infinity

        for t in 1 to k:
            candidate = permitHelper(i - D[t]) + P[t]
            if candidate < min:
                min = candidate

        A[i] = min
    return A[i]
```

Now, a dynamic programming version written in Python (0-indexing):

```
D = [1, 2, 3]
P = [2, 3, 4]
```

```
A = []

def find_ideal_cost(n):
    assert(len(D) == len(P))
    k = len(D)

    # Iterate over days, increasing. Day 1 corresponds to index 0.
    for d in range(n):
        minimum = float("inf")  # infinity.

        # Iterate over permit types, calculating the minimum cost to get us
        # to this day.
        for permit_idx in range(k):
            if d - D[permit_idx] < 0:
                # Base case: only one of this permit to get us to today.
                candidate = P[permit_idx]
            else:
                # If this permit type didn't cover us from the first day,
                # calculate the cumulative cost.
                candidate = A[d - D[permit_idx]] + P[permit_idx]

            # Take the minimum cost of all the permit types for today.
            if candidate < minimum:
                minimum = candidate
        # Add to end of the list.
        A.append(minimum)
    return A[-1]
```

2.2 Where Did We Park?

We return a list of the indices of the permits used (1-indexed in this case).

```
explain_permit(a):
    k = len(D)
    n = len(a)

    if len(a) == 0:
        return []
    else:
        minimum = infinity
        permit_idx_chosen = -1

        for permit_idx in 1 to k:

            if n - D[permit_idx] <= 0:
                candidate = P[permit_idx]

                if candidate < minimum:
                    minimum = candidate
                    permit_idx_chosen = permit_idx

            else if a[end - D[permit_idx]] == a[n] - P[permit_idx]:
                candidate = a[n - D[permit_idx]]

                if candidate < minimum:
                    minimum = candidate
```

```
                    permit_idx_chosen = permit_idx

        if permit_idx_chosen == -1:
            throw an Error        // should not occur

        # Add our permit to the front of the list (so that it is displayed in order)
        return list(permit_idx_chosen) + explain_permit(a[n - (D[permit_idx_chosen])])
```

Now, in Python, with 0-indexing:

```
def explain_permit(a):
    k = len(D)
    end = len(a) - 1

    if len(a) == 0:
        return []
    else:
        minimum = float("inf")  # infinity.
        permit_idx_chosen = -1
        for permit_idx in range(k):

            if end - D[permit_idx] < 0:
                # Using this pass would fulfill our time. Use cost of pass itself.
                candidate = P[permit_idx]

            elif a[end - D[permit_idx]] == a[end] - P[permit_idx]:
                # We have used this pass to get to the next step.
                candidate = a[-D[permit_idx]]

            else:
                # This pass has not been used at this step. Ignore it.
                continue

            # Take the minimum over all iterations at this step.
            if candidate < minimum:
                minimum = candidate
                permit_idx_chosen = permit_idx

        # Make sure we've chosen a pass.
        assert(permit_idx_chosen != -1)

        return [permit_idx_chosen] +
                explain_permit(a[:-(D[permit_idx_chosen])])
```

3. Pwner of All I Survey

3.1 A Profound Dis-Likert for Greedy

Let us have question lengths 4,5,3,5,2,6 with m = 6

- **Greedy Solution:** {{4,5,3}, {5,2}, {6}}, score = $(12-12)^2 + (12-7)^2 = 0^2 + 5^2 = 25$
- **Optimal Solution:** {{4,5}, {3,5,2}, {6}}, score = $(12 - 9)^2 + (12-10)^2 = 3^2 + 2^2 = 9 + 4 = 13$

3.2 A Fair and Balanced Survey

here we want to try every length we can for the current page, recursing the rest of the questions e.g.: 1,2,3,4,5 with m = 5 try {{1}, recurse}, {{1,2}, recurse}, {{1,2,3}, recurse}, {{1,2,3,4}, recurse}

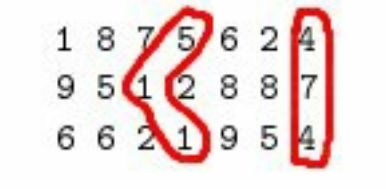
our function LowestScore computes the lowest score possible given a list of questions (lengths), Q, and the max length of a question, m LowestScore(Q, m) // Form a page in all the different lengths possible with elements off of Q, recurse, get min

```
        init total, index and to 0
        init score to infinity
        while(total < 2m)
            if(Q[index] != null)
                total += Q[index]
                score = min(score, LowestScore(Q[index+1:end], m))
                index++
            else
                // if Q[index] == null then we ran out of questions and
                // we just formed the last page, which scores 0
                return 0
        // end while

        return score
```

4 Seam Carving

4.1 Seamingly Simple

1. 

    - 4+7+4 = 15
    - 5+1+1 = 7

2. C(1,j) = A[1][j]

4.2 Seamy Details

C(i , j) = min(C(i , j-1), C(i , j), C(i , j+1)) + A[i][j]