CS 161 Winter 2020 Section 7

February 27-28, 2020

Warm-up: Greedy or Not?

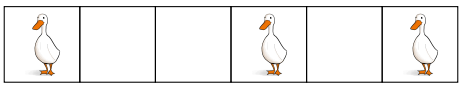
Sometimes it can be tricky to tell when a greedy algorithm applies. For each problem, say whether or not the greedy solution would work for the problem. If it wouldn't work, give a counter example.

1. You have unlimited objects of different sizes, and you want to completely fill a box with as few objects as possible. (Greedy: Keep putting the largest object possible in for the space you have left)
2. You have unlimited objects, all of which are size 3^k for some integer k, and you want to completely fill a box with as few objects as possible. (Greedy: same approach as the previous problem)
3. You have lines that can fit a fixed number of characters. You want to print out a fixed series of words while using as few lines as possible. (Greedy: Fit as many words as you can on a given line)
4. You want to get from hotel 1 to hotel n, and you can travel at most k distance between hotels before collapsing from exhaustion. Each hotel has a fixed cost per night. Find the minimum cost possible to spend on hotels. (Greedy: Go as far as you can before stopping at a hotel)

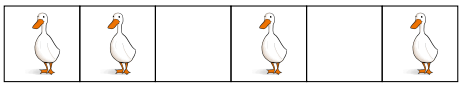
1. Greedy does not work! Consider a box of size 14 and objects of size 10, 7, and 1.
2. Greedy works! This is basically how you would write a number in base 3.
3. Greedy works!
4. Greedy does not work! Consider hotel costs [10, 20, 100, 10], where each hotel is 1 apart and k = 2.

Ducks in a Row

You have a dance floor made up of n squares in a line. The troupe of dancing ducks are back, and you'd like to assign ducks to squares. Different spots along the line will result in different quality of dancing: suppose that the location i will result in a dance of quality $T[i]$, where $T[i]$ is a positive integer. Further, you cannot place two ducks directly next to each other: this cramps their style.
Your goal is, given the array T, to create the best dance line-up possible, when summed over all of the n squares on the dance floor.
For example, if the input was $T = [21, 4, 6, 20, 2, 5]$, then you should place ducks in the pattern



and you would obtain dance quality $21 + 20 + 5 = 46$. You would **not** be allowed to place ducks in the pattern



because there are two ducks next to each other.
Design a dynamic programming algorithm which runs in time $O(n)$ which takes as input the array T and returns a single number which is the best total quality of dancing possible. Your algorithm does not need to output the optimal way to place the ducks.

First, we'll reason about which subproblems we'll use, and the recursive relationship that is satisfied between our subproblems. Our sub-problems will be:

$A[i]$ = the best quality of dancing you can get from the first $i+1$ positions of the dance floor.

I.e. using the dance floor only up through index i. The recursive relationship they satisfy is:

$$A[i] = \max\{A[i-2] + T[i], A[i-1]\},$$

for any $i \geq 2$. The base case is

$$A[0] = T[0] \qquad A[1] = \max\{T[0], T[1]\}.$$

The reason for this relationship is as follows. Suppose that we have an optimal solution for $A[i]$. There are two cases: either there is a duck dancing at position i, or there is not.

- In the case where there is a duck dancing at position i, we have

  $$A[i] = A[i-2] + T[i].$$

  This is because if we just ignore the $i-1, i$ positions of the garden, this gives us an optimal solution to $A[i-2]$. (Otherwise, if there were a better solution, we could just take that solution and place a duck at position i, and this would give us a better solution for $A[i]$).

- In the case where there is no duck dancing at position i, we have

  $$A[i] = A[i-1].$$

  This follows from similar reasoning: our optimal solution to $A[i]$ is also an optimal solution to $A[i-1]$; otherwise we'd have a better solution for $A[i]$.

Thus, since exactly one of the two cases occurs and $A[i]$ is the optimal solution, we have

$$A[i] = \max\{A[i-2] + T[i], A[i-1]\}.$$

The pseudocode is as follows:

```
def placeDucks(T):
   Initialize an array A = [T[0], max{T[0],T[1]}, None, None, ..., None ] of length n
   for j = 2,...,n-1:
       A[j] = max{ A[j-2] + T[j], A[j-1] }
   return A[n-1]
```

That is, the algorithm walks through A in order, applying the recursive relationship defined above.

Encoding

Suppose we encode lowercase letters into a numeric string as follows: we encode a as 1, b as 2 . . . and z as 26. Given a numeric string S of length n, develop an $O(n)$ algorithm to find how many letter strings this can correspond to. For example, for the numeric string 123, the algorithm should output 3 because the letter strings that map to this numeric string are abc, lc, and aw.

Intuitively, a one digit substring can be decoded to a letter if it's > 0, and a two digit substring can be decoded to a letter if it's between 10 and 26 (inclusive).
To turn this into a dynamic programming problem, we can count the number of ways to decode the substring ending at i. (ie the substring $S[0 : i+1]$ in python notation). We start at $i = 0$ and build up.

If we know the number of ways possible to decode the substring up until the $k-1$st letter ($S[0 : k]$) and $k-2$nd letter ($S[0 : k-1]$), we can use that information to find the number of decodings for the kth substring ($S[0 : k+1]$). If the last digit isn't zero, we can convert that to a letter directly, and check the array for the number of decodings for $S[0 : k]$ to get the total number of decodings here (we have the same number of decodings, since each previous decoding now just has an extra letter tacked onto the end). Or, if the last two digits make a valid letter (between 10 and 26 inclusive), the total number of decodings using this option is equal to the total number of decodings for the substring $S[0 : k-1]$. If both interpretations are valid, we add the number of decodings from the first case to the number of decodings for the second case to get the total number of decodings.

More formally, we can define a recurrence as follows:
Subproblem: $A[i]$ = the number of ways to encode the string up to and including S[i]

Recursive relationship:

$$A[i] = \sum \begin{cases} A[i-1] & \text{if } S[i] \in \{1, ..., 9\} \\ A[i-2] & \text{if } S[i-1 : i+1] \in \{10, ..., 26\} \end{cases}$$

To compute this in a single forward pass using dynamic programming, we build a table for each $A[i]$ and initialize base cases for $A[0]$ and $A[1]$

```
def num_decodings(S):
    n = len(S)
    #   A[i] := how many possible decodings for S[:i+1]
    A = [0 for _ in range(n)]

    # base cases for 0 and 1
    A[0] = int(S[0])>0
    two_digit_num = int(S[:2])
    A[1] = 10 <= two_digit_num <= 26
    if int(S[1]) > 0:
        A[1] += A[0]

    for i in {2,...,n-1}:
        if int(S[i]) > 0:
            A[i] += A[i-1]

        two_digit_num = S[i-1:i+1]
        if 10 <= two_digit_num <= 26:
```

```
            A[i] += A[i - 2]

    return A[-1]
```

Knight Moves

Given an 8×8 chessboard and a knight that starts at position $a1$ (the letter denotes the column and the number the row), devise an algorithm that returns how many ways the knight can end up at position xy after k moves. Knights move in an "L" shape: ± 1 square in one direction and ± 2 squares in the other direction.

We can store an array of the number of paths to each position after i moves, for each i. The base case is simple - after 0 moves, the knight has one way to end up in his starting position - not move at all!
If we know how many ways to get to each of the positions after round $i-1$, to get the number of ways he could move to some xy, we would add up the total number of ways he could have gotten to any of his last steps - 1 away in some direction and 2 away in the other. For example, there are 3 ways to get to $a2$ - either from $c1$, $c3$, or $b4$ - so we would simply add up the number of ways to get to each of these positions after $i-1$ steps to count how many ways to end up in position $a2$ after i steps.
Finally, once we compute the array for k, we can return position xy. (Notice that even though we only cared about position xy, we still computed the number of ways to get to any point on the chessboard in the previous steps - this is because these points could be on the path, despite not being the end point.)
More formally, we can define the function $f(x, y, i)$ to be the number of ways the knight can get to position xy in i moves. This gives us the following recurrence: (written out for clarity)

$$\begin{aligned} f(x,y,i) =& f(x-1,y-2,i-1) + f(x-1,y+2,i-1) + f(x+1,y-2,i-1) + f(x+1,y+2,i-1) \\ & f(x-2,y-1,i-1) + f(x-2,y+1,i-1) + f(x+2,y-1,i-1) + f(x+2,y+1,i-1) \end{aligned}$$

(where the function evaluates to 0 if the position is off the board)

```
def knight_moves(end_position, num_moves):
    # num_ways stores how many ways to get to each reachable position
    num_ways = collections.defaultdict(int)
    num_ways[(0, 0)] = 1 #(0,0) corresponds to A1
    move_directions = [(1, 2), (1, -2), (-1, 2),
        (-1, -2), (2, 1), (2, -1), (-2, 1), (-2, -1)]

    for i in range(num_moves):
        new_num_ways = collections.defaultdict(int)
        for cur_row, cur_col in num_ways.keys():
            for move_row , move_col in move_directions:
                new_row = cur_row + move_row
                new_col = cur_col + move_col
                # check to make sure new position stays within the board
                if new_row >= 0 and new_row < 8 and
                   new_col >= 0 and new_col < 8:
                    new_num_ways[(new_row, new_col)] +=
                        num_ways[(cur_row, cur_col)]
        num_ways = new_num_ways

    return num_ways[ end position ]
```