# CS 161 (Stanford, Winter 2026) Section 7

## 1 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* (decoded as "1", "2", "3") *lc*, (decoded as "12", "3") and *aw* (decoded as "1", "23").

**Solution**

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 to decode the substring ending at $k - 1$ ($S[0 : k]$), we can use that information to count the number of decodings for the substring ending at $k$ ($S[0 : k + 1]$). 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. 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.

Let $f(i)$ denote the number of ways to encode the string up to and including $S[i]$ establish our recurrence as follows:

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

To compute this in a single forward pass using dynamic programming, we build a table for each $f(i)$ and initialize base cases for $f(0)$ and $f(1)$, as shown below. We set up an array of size $n$ and fill in each element, and return the last element. Because it takes time $O(n)$ to iterate through our array and $O(1)$ time to fill in a given array element, the total runtime of our algorithm is $O(n)$.

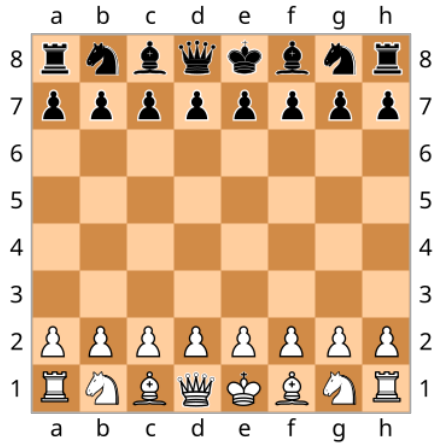
```
def num_decodings(numeric_string):
    n = len(numeric_string)
    #   t[i] := how many possible decodings for s[:i]
    t = [0 for _ in range(n)]
```

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

    for i in range(2, n):
        if int(numeric_string[i]) > 0:
            t[i] += t[i-1]

        two_digit_num = numeric_string[i-1:i+1]
        if 10 < two_digit_num <= 26:
            t[i] += t[i - 2]

    return t[-1]
```

## 2 Knight Moves

Given an 8×8 chessboard and a knight that starts at position $a1$, devise an algorithm that returns how many ways the knight can end up at position $xy$ after $k$ moves. Knights move $\pm 1$ squares in one direction and $\pm 2$ squares in the other direction. In other words, knights move in a pattern similar to a L.

Note: on a chessboard, rows are labeled from 1 to 8 and columns are labeled from $a$ to $h$, as seen below.



Solution

We can use dynamic programming to track the number of ways to reach each position after $i$ moves. 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 they could move to some $xy$, we would add up the total number of ways they could have gotten to any of his last steps (one move away in an "L" pattern - 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:

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

(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]
```

# 3 String Cutting

Suppose we have a string of length $k$, where $k$ is a positive integer. We would like to cut the string into integer-length segments such that we maximize the *product* of the resulting segments' lengths. Multiple cuts may be made. For example, if $k = 8$, the maximum product is 18 from cutting the string into three pieces of length $3, 3$, and 2. Write an algorithm to determine the maximum product for a string of length $k$.

**Solution**

To solve this problem we are going to exploit the following overlapping sub-problems. If we let $f(k)$, be the maximum product possible for a string of length $k$, then we have

$$f(k) = \max_{c \in \{2, k-1\}} (k,\ c \cdot f(k-c))$$

Another way to think of this is that we are going to try cutting the string of length $k$ into two strings of length $c$ and $k - c$ and try all possible values of $c$, taking the one which produces the maximum product. Note that not cutting the string at all is another option which we can take. Also notice that we do not need to consider cutting off a length of 1 since that will never yield the optimal product, and also do not need to try cuts any larger than $\lfloor k/2 \rfloor$ since those will already have been explored due to the symmetry of the cutting. The running time for this algorithm is $O(k^2)$ since for each value of $k$ we loop through $O(k)$ values to get the answer for that $k$.

```
def max_string_cut(k):

    # max_prods[i] := largest product for
    # cutting string of length i
    max_prods = [0 for _ in range(k + 1)]
    max_prods[1] = 1 # length 1 cannot be cut more

    for i in range(2, k + 1):

        best_prod = i # compare against not cutting at all
        for cut in range(2, i // 2 + 1):
            remaining = i - cut # the length remaining
            p = max_prods[cut] * max_prods[remaining]
```

```
            best_prod = max(best_prod, p)
        max_prods[i] = best_prod
    return max_prods[k]
```