## Dice Probabilities

We wish to find the probability that rolling $k$ 6-sided fair dice will result in a sum $S$. Devise an algorithm to find this probability.

**SOLUTION:** We can store an array with the probability of getting each sum with $i$ dice. It's easy to compute this for $i = 0$ (the probability of sum $= 0$ is just 1) or $i = 1$ (the numbers $1 - 6$ have probability 1/6). We can calculate these arrays for increasing $i$ as follows:

The probability of achieving the sum $s$ after rolling $i$ dice is calculated as the sum of 6 different probabilities:

1. the chance of rolling a $1\cdot$ the chance of having a sum of $s - 1$ in the previous round
2. the chance of rolling a $2\cdot$ the chance of having a sum of $s - 2$ in the previous round
3. . . . and so on

More, formally, we can define the function $f(s, i)$ to be the probability of getting sum $s$ in round $i$. Then we get the following recursion:

$$f(s,i) = \sum_{j=1}^{6} f(s-j, i-1) \cdot \frac{1}{6}$$

Calculating the runtime for this algorithm below is less straightforward. At the $i$-th iteration of the outermost for-loop, the second for-loop performs $5i + 1$ iterations (why? what is the possible range of sums at iteration $i$?). The third for-loop performs 6 iterations.
At a fixed iteration for each for-loops, the amount work done is O(1). Summing the total amount of work done at each iteration of the outermost for-loop reveals that this algorithm has a runtime of $O(k^2)$.

```
def dice_probs(num_dice, total):
    # probs stores the probability of having a certain sum
    probs = collections.defaultdict(float)
    probs[0] = 1.0

    for i in range(num dice):
        new_probs = collections.defaultdict(float)
        for prior_total, prob in probs.iteritems():
            # for each previous sum, we have 6 possible rolls
            for roll in range(1, 7):
                new_total = prior_total + roll
                if new_total <= total :
                    new_probs[new total] += prob/6.0
        probs = new probs
    return probs[total]
```

**Problem Solving Notes:**

1. *Read and Interpret:* What is the question asking of you? Is there a way you can break down the probability of getting a sum over $k$ dice into something sequential? Does it matter if the dice are all rolled at once, or one after the other? I.e., are the dice independent?

2. *Information Needed:* What is the probability of getting any particular value at a single roll of a die? If that value is rolled for some die $i$, what does your subproblem become, so that the sum of all rolls becomes $S$?

3. *Solution Plan:* Keeping the runtime low here can be tricky. Note that you can create an entirely new *new_probs* dictionary for each new die, as the recurrence only relies on the stored values from $i-1$. This helps with our runtime as we don't keep accumulating keys in our dictionary! It would be okay (and expected!) for you to write your first draft of the solution without realizing this. As you try to do a runtime analysis of your code, you might realize that you can optimize the number of values you are storing, and can eventually end up at this optimal solution!

## Knight Moves

Given an $8\times8$ 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 $\pm1$ squares in one direction and $\pm2$ 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 1-8 and columns are labeled from $a-h$.

**SOLUTION:** 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 they could move to some $x, y$, we would add up the total number of ways they 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 ]
```

**Problem Solving Notes:**

1. *Read and Interpret:* The question is asking you to return the number of ways a knight can end up in a specific part of the board. Note that you are given $x$, $y$, and $k$ which should all be used in the formulation of your solution.

2. *Information Needed:* How many ways are there in which a knight can get from some other spot on the board to $x, y$? How many steps would it have taken to get to that previous spot? How can I connect these values to $x$, $y$, and $k$?

3. *Solution Plan:* While it might seem like a complex problem when you enumerate all possibilities, realize that once you have that, you are basically done! There is some clever code-writing involved in making sure this can be implemented iteratively - think carefully about the base case and what it means to make 0 moves! Can you calculate all values for $k = 1$ if you have $k = 0$? Do you *need* to store all combinations of $x, y$?

## 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 $aw$.

**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 $k$ ($S[0 : k + 1]$). Let's do some case-by-case analysis:

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.

2. 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 $S[0 : k - 1]$.

3. 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 decode 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]
```

**Problem Solving Notes:**

1. *Read and Interpret:* First we should understand why this is a DP problem. If you see an input "12", that can either mean "ab" or "l". With a larger input string, the number of possibilities will only grow. Thus, we need DP to solve this question! If we try to do it naively, we can end up with an exponential runtime.

2. *Information Needed:* If you are looking at the $i$-th number in the input string, what information about the string up to the $i-1$-th number help you? Which are the numbers that can cause ambiguity and increase the number of possible decodings? What should we be doing differently for these numbers as compared to other numbers?

3. *Solution Plan:* Once you have answers to the above questions, you should be able to translate the recursive formulation you have into a simple loop implementation! Does this implementation look familiar to any of the ones we've solved above?

## Exact Acorn Dropping

We have $a$ acorns and $b$ branches. We want to compute the *exact* minimum number of drops needed (in the worst case) to find the highest branch in which an acorn will not break after dropping. Assume that all of the acorns are the same strength; if one acorns breaks after dropping from a branch, all acorns will break after dropping from that branch. Design an algorithm that returns the minimum number of drops needed to accomplish this task in the worst case, without actually dropping any acorns.

**SOLUTION:** Given $a$ acorns and $b$ branches, we will solve this problem by considering dropping a single acorn from branch $x$ for $x \in \{1, 2, ...b\}$. We will consider the worst case scenario for dropping the acorn from each of these branches and choose the branch which gives the best worst-case guarantee.

When we drop an acorn from a branch $x$, there can be two cases: (1) The acorn breaks or (2) The acorn doesn't break.

(1) If the acorn breaks after dropping from $x$'th branch, then we only need to check for branches lower than $x$ with remaining acorns; so the problem reduces to $x-1$ branches and $a-1$ acorns.

(2) If the acorn doesn't break after dropping from the $x$'th branch, then we only need to check for branches higher than $x$; so the problem reduces to $b-x$ branches and $a$ acorns

With this approach, we can write our recurrence as follows

$$F(a,b) = 1 + \min_{x \in \{1,...b\}} \max \begin{cases} F(a-1, x-1) & \text{acorn breaks at branch } x \\ F(a, b-x) & \text{acorn doesn't break at branch } x \end{cases}$$

In the recurrence above, the added 1 term comes from the fact that we drop our acorn at branch $x$. The min term comes from our desire to minimize the number of drops that we make over any of our branch choices of $x$, and the max term comes from our consideration of the worst-case scenario of dropping the acorn at a specified branch.

We can calculate this using dynamic programming as show below. With this approach we set up an $a \times b$ matrix and fill in each element, with the final solution being the last element to be filled in. Because it takes time $O(b)$ to consider all values of $x$ in order to fill in a single value, the total running time is $O(ab^2)$

```
def acorn_drops(acorns, branches):
    # table[a][b] number of drops needed with a acorns and b branches
    table = [[0 for _ in range(branches + 1)] for a in range(acorns + 1)]
```

```
for a in range(1, acorns + 1):
    table[a][1] = 1 # one branch requires just one drop

for b in range(1, branches + 1):
    table[1][b] = b # only 1 acorn, must consecutively drop upward in
                    # the worst case
    table[0][b] = float("inf") # 0 acorns is impossible

for a in range(2, acorns + 1):
    for b in range(2, branches + 1):
        # drop from branch that minimizes the worst case num. drops
        # drop_branch := the branch we drop at to get to smaller problems
        # the max here goes over these two cases:
        #   1. doesn’t break, same acorn count, search branches above
        #   2. breaks, we lose one acorn, search branches below
        drops_per_branch = list()
        for drop_branch in range(1, b + 1):
            drops = max(table[a][b - drop_branch],
                        table[a - 1][drop_branch - 1])
            drops_per_branch.append(drops)

        table[a][b] = 1 + min(drops_per_branch)

return table[acorns][branches]
```

**Problem Solving Notes:**

1. *Read and Interpret:* This question might look familiar, as it was an older homework problem! But note that we are asking a different question now. Before, you were asked to determine the procedure to determine this branch within a certain Big-O amount of drops. Now, we want to find the exact *minimum* number of drops (not acorns!) needed in the *worst* case. Note that you cannot actually drop any acorns!

   How does this change the problem? One hint is that now that we are trying to optimize something, DP might be useful here! What could your subproblem be? If you knew the minimum amount of drops needed for a smaller set of branchs, can you use that to solve for the larger problem?

2. *Information Needed:* What happens if an acorn breaks? What is the new subproblem you are now trying to solve? What if it does not break? How do we try to incorporate the idea that we want to consider the worst case scenario, and with that in mind, we want to calculate the minimum number of drops? What part of the outcomes of dropping an acorn from a specific branch can we control? What are the base cases?

3. *Solution Plan:* Once you have a recurrence relationship down, the tricky part is to convert it into an iterative solution. Note here that in order to solve for $F(a, b)$ you need $F$ filled

out for smaller indices than $a$ and $b$ respectively. It is okay to be a bit more verbose in your implementation for min (max (...)) as long as the result is easier for you to understand.

## Rod Cutting

Suppose we have a rod of length $k$, where $k$ is a positive integer. We would like to cut the rod 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 rod into three pieces of length $3, 3$, and 2. Write an algorithm to determine the maximum product for a rod 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 rod 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 rod of length $k$ into two rods 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 rod 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 $k$.

Here is the pseudocode:

```
def max_rod_cut(k):

    # max_prods[i] := largest product for cutting rod of length i
    max_prods = [0 for _ in range(k + 1)]
    max_prods[1] = 1 # base case. 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 = cut * max_prods[remaining]
            best_prod = max(best_prod, p)
        max_prods[i] = best_prod

    return max_prods[k]
```

**Note:** It is also correct to implement the recurrence relation as

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

in which case the pseudocode becomes:

```
def max_rod_cut(k):

    # max_prods[i] := largest product for cutting rod of length i
    max_prods = [0 for _ in range(k + 1)]
    max_prods[1] = 1 # base case. 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]
```

**Problem Solving Notes:**

1. *Read and Interpret:* This question will look a bit different from the others at first glance! We are trying to maximize the product of a list of values. Why can this be a DP problem?

2. *Information Needed:* We can either choose to not cut the rod at all, or cut it in some spot. How many possible spots can the rod be cut in? What does the length of the remaining segment look like? What will the resulting product of segments look like if a particular cut is made?

3. *Solution Plan:* Once the recurrence relation is established, this problem is very similar to the ones above in terms of implementation. That is, larger values of $k$ utilize pre-filled solutions for smaller values of $k$.