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 sentence using as few lines as possible. It doesn’t matter if words are split between lines (Greedy: Fit as many characters 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. Find the minimum cost of 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 $k = 2$.

Cutting Ropes

Suppose we are given $n$ ropes of different lengths, and we want to tie these ropes into a single rope. The cost to connect two ropes is equal to sum of their lengths. We want to connect all the ropes with the minimum cost.

For example, suppose we have 4 ropes of lengths 7, 3, 5, and 1. One (not optimal!) solution would be to combine the 7 and 3 rope for a rope of size 10, then combine this new size 10 rope with the size 5 rope for a rope of size 15, then combine the rope of size 15 with the rope of size 1 for a final rope of size 16. The total cost would be $10 + 15 + 16 = 41$. (Note: the optimal cost for this problem is 29. How might you combine the ropes for that cost?)

Find a greedy algorithm for the minimum cost and prove the correctness of your algorithm.

Solution: Always combine the smallest ropes available to you until you have one single rope.

Justification:

We can write the strings as a graph, where the leaves are the original ropes and every node with two children is a sum of two ropes.
From here we can use a similar argument from our Huffman analysis. Our Huffman analysis relied on these two key points:
Lemma 1: if x and y are ropes with the shortest length, there is an optimal tree where they are siblings.
Lemma 2: if we treat the nodes at a given level as leaves, we can still apply Lemma 1.

Now we proceed with the proof.
Inductive hypothesis. By combining the two smallest ropes available to us at any given point, there is a minimal solution that extends the current solution.
Base case. When we haven’t combined any of the ropes, there is clearly a minimal solution that extends the current (empty) solution.
Inductive Step. Suppose that we have combined ropes k times (meaning there are n – k ropes remaining). Lemma 2 tells us that we can basically treat previously combined ropes the same as ropes that haven’t been combined, and Lemma 1 tells us that there’s an optimal solution where the shortest length ropes are ‘siblings’ to a parent node that’s the sum of them – in other words, there’s an optimal solution where the smallest ropes available are tied together.
Conclusion. By the nth step, we have not ruled out the optimal solution. Therefore, the solution we chose is optimal.

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.

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: the chance of rolling a 1, the chance of having a sum of s – 1 in the previous round, plus the chance of rolling a 2, the chance of having a sum of s – 2 in the previous round, 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:

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

```python
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.items():
        # 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]

Food for Thought: The above pseudo code uses a standard library for a hash data structure. What are the tradeoffs in terms of runtime and space complexity if one instead used a fixed size array?

Egg Dropping

We have $e$ eggs and $f$ floors. We want to find the minimum number of drops needed in the worst case to find the highest floor in which an egg will not break after dropping. Assume that all of the eggs are the same strength; if one egg breaks after dropping from a floor, all eggs will break after dropping from that floor. Design an algorithm that returns the minimum number of drops needed to accomplish this task in the worst case, without actually dropping any eggs.

Given $e$ eggs and $f$ floors, we will solve this problem by considering dropping a single egg from floor $x$ for $x \in \{1, 2, \ldots, f\}$. We will consider the worst case scenario for dropping the egg from each of these floors and choose the floor which gives the best worst-case guarantee.

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

1. If the egg breaks after dropping from the $x$’th floor, then we only need to check for floors lower than $x$ with remaining eggs; so the problem reduces to $x - 1$ floors and $e - 1$ eggs.
2. If the egg doesn’t break after dropping from the $x$’th floor, then we only need to check for floors higher than $x$; so the problem reduces to $f - x$ floors and $e$ eggs.

With this approach, we can write our recurrence as follows:

$$F(e, f) = 1 + \min_{x \in \{1, \ldots, f\}} \max \left\{ F(e - 1, x - 1), F(e, f - x) \right\}$$

egg breaks at floor $x$
egg doesn’t break at floor $x$

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

```python
def egg_drops(eggs, floors):
    # table[e][f] how many drops we need if we have e eggs and f floors
    table = [[0 for _ in range(floors + 1)] for e in range(eggs + 1)]

    for e in range(1, eggs + 1):
        table[e][1] = 1 # one floor requires just one drop
```
for f in range(1, floors + 1):
    table[1][f] = f  # only 1 egg, must consecutively drop upward
    table[0][f] = float("inf")  # 0 eggs is impossible

for e in range(2, eggs + 1):
    for f in range(2, floors + 1):
        # drop from floor that minimizes the worst case num. drops
        # drop_floor := the floor we drop at to get to smaller problems
        # the max here goes over these two cases:
        # 1. doesn’t break, same egg count, search floors above
        # 2. breaks, we lose one egg, search floors below
        drops_per_floor = list()
        for drop_floor in range(1, f + 1):
            drops = max(table[e][f - drop_floor],
                        table[e - 1][drop_floor - 1])
            drops_per_floor.append(drops)
        table[e][f] = 1 + min(drops_per_floor)

return table[eggs][floors]