SEMINAR 3

RECURSION

Week 9 – Seminar 4

When considering a sequence of object, such as a list or a string, we can see that data structure as a recursive data structure. For example, a list [1,2,3] can be thought of as its first element 1 (the head) and the rest of the list [2,3] (the tail) which is also a list. Similarly, the list [2,3] can be seen as its head 2 and its tail [3]. Again, [3] can be seen as its head 3 and its tail [] which is empty ("end" of the recursive data structure). This way of thinking might be useful for this week's exercises.

Exercise 1:

In Python, to compute the sum of all elements in a list, you can use the built-in function **`sum`**. For example:

```
>>> sum([1,2,3,4])
10
>>> sum([])
0
```

However, **`sum`** does not work with multidimensional lists (see example below):

```
>>> sum([1,[2,3],4])
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    sum([1,[2,3],4])
TypeError: unsupported operand type(s) for +: 'int' and
'list'
```

Write a recursive algorithm (using pseudo-code) `sum_all(numbers:list):list` that takes a multidimensional list of integers as parameters and returns the sum of all elements in that list. Note, empty lists sum to 0. For examples:

```
>>> sum_all([1,[2,3],4])
10
>>> sum_all([1,[2,[3,[4]]]])
10
>>> sum_all([1,2,3,4])
10
>>> sum_all([1,[]])
1
```

Exercise 2: *wildcard pattern*

Given a binary pattern that contains `'?'` wildcard character at few positions, find all possible combinations of binary strings that can be formed by replacing the wildcard character by either 0 or 1. For example, for wildcard pattern `1?11?00?1?`, the possible combinations are

```
1011000010    1011000011    1011000110    1011000111
1011100010    1011100011    1011100110    1011100111
1111000010    1111000011    1111000110    1111000111
1111100010    1111100011    1111100110    1111100111
```

We can easily solve this problem using recursion. The idea is to process each character of the pattern one by one and recur for the remaining pattern. If the current digit is `0` or `1`, we ignore it and if the current character is a wildcard character `'?'`, we replace it with 0 & 1 and recur for the remaining pattern. You may need a convenience function with a parameter that keep track of the partial solution you built so far, and another parameter storing all the previous solutions you found.

Exercise 3:

Suppose you have a rod of length n, and you want to cut up the rod and sell the pieces in a way that maximizes the total amount of money you get. A piece of length i is worth £ p_i.

Table 1: An example of a price list.

Length	1	2	3	4	5	6
Price p_i (£)	1	5	8	9	10	17

For example, if you have a rod of length 4, there are eight different ways to cut it, and given the price list in Table 1, the best strategy is cutting it into two pieces of length 2, which gives you £ 10.00 (see Figure 1).

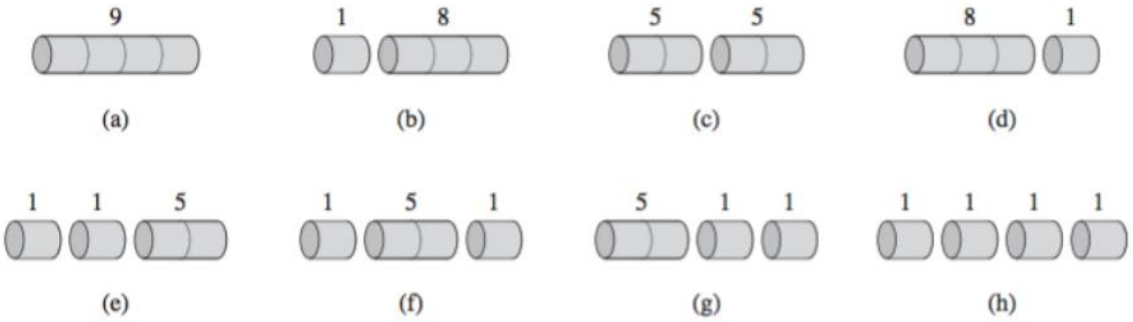

Figure 1: The 8 possible ways of cutting up a rod of length 4. Given the price list in Table 1, the optimal strategy is (c) cutting the rod into two pieces of length 2, which as a total value of £10.00.

We can view the problem recursively as follows:

- First, cut a piece off the left end of the rod, and sell it.
- Then, find the optimal way to cut the remainder of the rod.

Part A: Naïve approach

Now we don't know how large a piece we should cut off. So, we try all possible cases. First, we try cutting a piece of length 1, and combining it with the optimal way to cut a rod of length $n-1$. Then we try cutting a piece of length 2 and combining it with the optimal way to cut a rod of length $n-2$. We try all the possible lengths and then pick the best one.

This is summarised by the equation

$$r_n = \boldsymbol{max}_{1 \leq i \leq n} (p_i + r_{n-i})$$

Write in pseudo-code `rod_cutting(prices:list, length:int):int` a recursive algorithm that returns the optimal way to cut a rod of size `length` given a price list `prices`. The parameter `length` is a positive integer. The parameter `prices` is a list of strictly positive integers, where the first element is the price of a piece of length 1, the second element is the price of a piece of length 2, and so on.

Part B: A better approach using memoization

You have seen during the lecture that we can improve the recursive implementation of the Fibonacci function using memoization. We can also use this technique for this problem. Write the algorithm using memoization. What should we "remember" between each recursive call?