# 11. Memory Serves You Well

*The advantage of a bad memory is that one enjoys several times the same good things for the first time. — Friedrich Nietzsche*

Programming constructs and algorithmic paradigms covered in this puzzle: Dictionary creation and lookup. Exceptions. Memoization in recursive search.

Here's a cute coin row game that corresponds to an optimization problem. We have a set of coins in a row all with positive values. We need to pick a subset of the coins to maximize their sum, but with the additional constraint that we are not allowed to pick two adjacent coins.

Given:

```
14  3  27  4  5  15  1
```

You should pick `14`, skip `3`, pick `27`, skip `4` and `5`, pick `15` and skip `1`. This gives a total of `56`, which is optimal. Note that alternately picking and skipping coins does not work for this example (or in general). If we picked `14`, `27`, `5` and `1`, we would only get `47`. And if we picked `3`, `4`, and `15`, we would get a pathetic score of `22`.

*Can you find the maximum value for the coin row problem below?*

```
3  15  17  23  11  3  4  5  17  23  34  17  18  14  12  15
```

The optimal value for the coin row problem

```
3 15 17 23 11 3 4 5 17 23 34 17 18 14 12 15
```

is `126` obtained by selecting coins `15`, `23`, `4`, `17`, `34`, `18`, and `15`.

Obviously, our goal is a general-purpose algorithm that we can code and run to find the optimal selection. We will first use recursive search to solve this problem – we will recur on different choices, picking a coin or skipping it. If we skip a coin, we have the choice of either picking or skipping the next coin. The catch is that if we pick a coin, we are forced to skip the next coin.

## Recursive Solution

Here's code that recursively solves the coin row problem.

```
 1.   def coins(row, table):
 2.       if len(row) == 0:
 3.           table[0] = 0
 4.           return 0, table
 5.       elif len(row) == 1:
 6.           table[1] = row[0]
 7.           return row[0], table
 8.       pick = coins(row[2:], table)[0] + row[0]
 9.       skip = coins(row[1:], table)[0]
10.       result = max(pick, skip)
11.       table[len(row)] = result
12.       return result, table
```

The procedure takes a coin row as input, which is assumed to be a list. It also takes a dictionary `table` as input. The dictionary will contain information about the optimal value for the original problem as well as subproblems of the original problem. The dictionary will be empty for the initial call. The dictionary will get filled in during the recursive search and will need to be passed to the recursive calls. Note that we could have used a list representation for `table`, in which case we would have to allocate a table with `len[row] + 1` entries beforehand. Using a dictionary will come in handy when we memoize the `coins` procedure and other recursive procedures later in the book.

We have two base cases in Lines 2-7. The first base case is for the empty row, in which case we simply return `0` as the maximum value, and the updated dictionary. The dictionary `table` is updated with `0` as the value for key `0` as shown on Line 3. If the row has length `1`, then we can simply return the coin value as the maximum value. In the one coin case, we update key `1` of the dictionary with the coin value (Line 6).

Lines 8 and 9 make recursive calls corresponding to picking or skipping the first coin on the row, respectively. If we add the value `row[0]` to our value, then we had better not pick `row[1]`, and so the recursive call on Line 8 has `row[2:]` as the argument. This means that the first two elements in the row are dropped, the first because we picked it, and the second because of the adjacency constraint. On Line 9, we make a recursive call with `row[1:]` as argument and without adding in the `row[0]` value. Because we did not pick `row[0]` we are allowed to pick `row[1]` if we want to. Lines 10 figures out which of the recursive calls won and uses that call's value as the result and Line 11 fills in the appropriate dictionary entry. In the general case, the key/index for the dictionary is the length of the row for which we have computed the optimal value, and the value stored for that key/index is the optimal value found for that row.

A word about how the recursion works. We are selecting or skipping coins from the front of the list. So the smaller problems associated with the smaller length rows correspond to dropping the elements from the *front* of the list, or the coins to the left of the row. In our example:

```
14 3 27 4 5 15 1
```

the length `5` sublist that `coins` considers is the sublist:

```
27 4 5 15 1
```

If you are only interested in the maximum value that can be obtained for a coin row problem we would simply return `result` and we would not even need `table`. But we want to know what coins were picked. Suppose that someone solved the long coin row problem (our second example) and told you that the optimal value was 126, you would need quite some work to verify that. You would have to solve the coin row problem yourself. The dictionary returned has the information necessary to efficiently figure out what coins were picked and the trace back procedure we will describe shortly shows the operations required.

If we run:

```
coins([14, 3, 27, 4, 5, 15, 1], table={})
```

It returns:

```
(56, {0: 0, 1: 1, 2: 15, 3: 15, 4: 19, 5: 42, 6: 42, 7: 56})
```

The first value is the optimal value, and the dictionary is printed between the curly braces as a listing of key: value pairs. For example, `table[0] = 0`, `table[4] = 19`, `table[7] = 56`. The dictionary is telling us not only what the optimal value is for the original row that has length `7`, but is also telling us what the optimal value is for smaller coin row problems so we can trace back the coin selection. For example, `table[4]` tells us that the optimal value is `19` for the sublist corresponding to the last four elements of the list, namely, `4, 5, 15, 1`. The maximum value is obtained by picking `4` and `15.`

You might think that it would have been convenient to set the default value of the dictionary `table` to `{}` in `coins` and not specify the second argument in the invocation.

Python has one copy of default arguments per function. As a result, using the default argument on multiple coin row problems would result in spilling values over from the previous instance of the problem. Mutable default arguments should be used with care!

We will now show how to use the `table` values to conveniently trace back what coins were picked.

## Tracing Back Coin Selection

```
 1.   def traceback(row, table):
 2.       select = []
 3.       i = 0
 4.       while i < len(row):
 5.           if (table[len(row)-i] == row[i]) or \
 5a.             (table[len(row)-i] == \
 5b.              table[len(row)-i-2] + row[i]):
 6.               select.append(row[i])
 7.               i += 2
 8.           else:
 9.               i += 1
10.       print ('Input row = ', row)
11.       print ('Table = ', table)
12.       print ('Selected coins are', select,
                 'and sum up to', table[len(row)])
```

The procedure `traceback` takes both the coin row and the dictionary as input. Note that the keys of `table` range from `0` to `len(row)`, inclusive, whereas the indices of `row` will, as always, range from `0` to `len(row)-1`.

The procedure works backwards in looking at dictionary keys that are the largest, i.e., those that store information for the longest row problems. Line 5 is the crucial line in the procedure. First, focus on the second part (after the first `'\'`) of Line 5. If we are working backwards from the end of the list and see two table entries `table[len(row)-i-1]` and `table[len(row)-i]`, where the latter is larger than the former by `row[i]`, then it means that we have picked the coin `row[i]` (this would be the `i+1`th coin on the row). For example, suppose `i = 0`. Then, the last two entries of the dictionary `table` are compared. These correspond to the optimal solutions for the original problem skipping the very first element (`table[len(row)-1`) and the original problem (`table[len(row]`), respectively. If the latter is larger by `row[0]`, it means that the optimal solution for the original problem picked the first element `row[0]`.

Why do we have the condition `table[len(row)-i] == row[i]` in the first part of Line 5? This is simply to take the corner case into account, when `i = len(row)-1`. In this case, the second part of Line 5 would crash since `len(row)-i-2 < 0`. Thanks to the first condition and the disjunctive **`or`** the second part is never executed – the first condition will evaluate to **`True`**, since `table[1]` is always set to `row[len(row)-1]`.

In general, if we pick `row[i]` we could not have picked `row[i+1]` and so we increment `i` by `2` and keep going (Line 7). If we did not pick `row[i]`, we increment `i` by `1` and keep going (Line 9).

What happens for our example? Suppose we run:

```
row = [14, 3, 27, 4, 5, 15, 1]
result, table = coins(row, {})
traceback(row, table)
```

We get:

```
Input row =  [14, 3, 27, 4, 5, 15, 1]
Table =  {0: 0, 1: 1, 2: 15, 3: 15, 4: 19, 5: 42, 6: 42, 7: 56}
Selected coins are [14, 27, 15] and sum up to 56
```

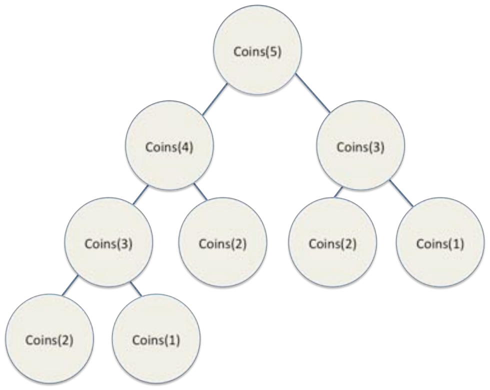
Since `table[7]` equals `table[5] + row[0]`, i.e., `56 = 42 + 14`, we choose `row[0] = 14` and increment the counter `i` by `2`. Since `table[5]` equals `table[3] + row[2]`, i.e., `42 = 15 + 27`, we pick `row[2] = 27` and increment `i` by `2`. We next check `table[3]`, which is not equal to `table[1] + row[4]`, i.e., `15 ≠ 1 + 5`, so we increment `i` by `1`. `table[2]` equals `table[0] + row[5]`, i.e., `0 = 15 + 15`, so we include `row[5] = 15`.

We now have an automated way of finding the optimum for an arbitrarily sized list. There is a small problem, however. For a list of size n, we end up calling the procedure with a list of size n – 1 and a list of size n – 2. Therefore, the number of calls for a list of size n is given by:

$$A_n = A_{n-1} + A_{n-2}$$

If n = 40, $A_n$ = 102,334,155. Not good ☹

The reason for all these calls is the redundant work that the recursive `coins` function does. Below are the recursive calls `coins` makes for a length 5 list. We only indicate what the length of the list is below since it does not matter what the list elements are to chart recursive calls.



Can we make the recursive solution to this puzzle more efficient?

## Memoization

Yes, we can, through a technique called *memoization* that eliminates redundant calls. We already have a dictionary for our coin row problem and all we need to do is to look up the dictionary `table` to see if we have already computed the solution to the problem!

Here's how memoization works in the recursive solution to our coin row problem.

```
def coinsMemoize(row, memo):
    if len(row) == 0:
        memo[0] = 0
        return (0, memo)
    elif len(row) == 1:
        memo[1] = row[0]
        return (row[0], memo)
    if len(row) in memo:
        return (memo[len(row)], memo)
    else:
        pick = coinsMemoize(row[2:], memo)[0] + row[0]
        skip = coinsMemoize(row[1:], memo)[0]
        result = max(pick, skip)
        memo[len(row)] = result
        return (result, memo)
```

The amazing thing is that we added 3 lines of code to get an exponential improvement in runtime. The memoized function only computes the solution to each problem once and stores it in the dictionary `memo`. There are only `len(row) + 1` entries in `memo`, each is computed exactly once, but looked up many times.

We renamed the variable `table` in `coins` to `memo` in `coinsMemoize` to reflect the different functionality of this variable. We are looking up the memo table during recursion in `coinsMemoize` to make the computation significantly more efficient, whereas variable `table` was only written and not read in `coins`.

## Dynamic Programming

Dynamic programming is a method for solving a problem by dividing it into a collection of simpler, possibly repeated and overlapping subproblems. Dynamic programming differs from Divide and Conquer in that in the latter the subproblems are disjoint or non-overlapping. For example, in Merge Sort or Quicksort the two arrays are disjoint. Similarly, in coin weighing, the coins are broken into disjoint groups. However, in our coin selection example, the two subproblems that we defined are overlapping, in the sense that they have coins in common.

This overlap of subproblems means that we might solve some subproblems repeatedly. In dynamic programming, each of these subproblems is solved just once, and their solutions stored. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time. Each of the subproblem solutions is indexed in some way, typically based on the values of the subproblem's input parameters, for efficient lookup. The technique of storing solutions to subproblems instead of recomputing them is called "memoization."

## Exercises

**Puzzle Exercise 1**: Solve a variant coin row problem where if you pick a coin you can pick the next one, but if you pick two in a row, you have to skip *two* coins. Write recursive, and recursive memoized versions of the variant problem where as before the objective is to maximize the value of selected coins. To obtain the selected coins, write the code to trace back the coin selection.

For our simple row example:

```
[14, 3, 27, 4, 5, 15, 1]
```

your code should produce:

```
(61, {0: (0, 1), 1: (1, 2), 2: (16, 3), 3: (20, 3), 4: (20, 1),
5: (47, 2), 6: (47, 1), 7: (61, 2)})
```

The maximum value `61` that can be selected corresponds to selecting `14`, `27`, `5`, and `15`.

*Hint*: You will need to make three recursive calls to obey the new adjacency constraint and pick the maximum value obtained from these three calls. Instead of the two choices of pick a coin and skip a coin in the original problem, you will have to code three choices in this variant: skip a coin, pick a coin and skip the next, and pick two adjacent coins. You may find the recursive solution for this problem easier to write than the iterative solution.