Algorithms: Design and Analysis - CS 412

Weekly Challenge 08: Dynamic Programming

Ali Muhammad Asad - aa07190

We are going to implement the rod cutting algorithm from CLRS 15.1 in a file `cutrod.py`. We will implement the following three versions of the solution: *top-down recursive*, *top-down memoized*, and *bottom-up*. Your code will be tested by `pytest` using the file, `test_cutrod.py`, given in `WC8_DP.zip`. To test your code implementation, open the directory containing `test_cutrod.py` and `cutrod.py` in the terminal, and run the following command:

```
pytest test_cutrod.py
```

TASKS:

(a) Write the top-down recursive version in a function, `cut_rod`.

(b) Write the top-down memoized version in a function, `cut_rod_memoized`.

(c) Write the bottom-up version in a function, `cut_rod_bottom_up`.

(d) All functions take two arguments, `p` and `n`, where `p` is the price array and `n` is the length of the rod. `p[i]` is the price of a rod of length `i`. All prices are positive and increase with length.

(e) Write all functions in the file, `cutrod.py`.

(f) Ensure that all tests pass by running `pytest` locally.

(g) Do not include any external packages.

(h) You may modify the error messages in `test_cutrod.py` to convey more information if you wish, but you may not alter any other functionality in it.

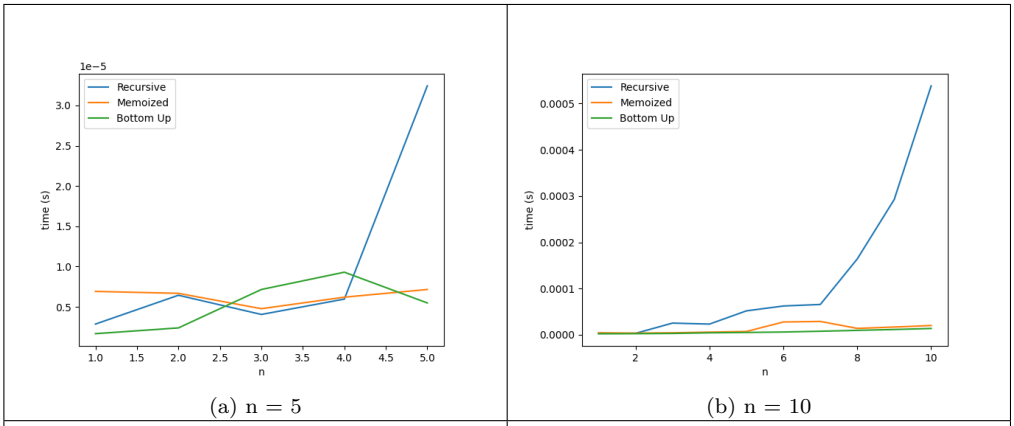
(i) Plot the running time of the three versions against `n` and include them below along with any relevant observations. Your code for plotting should be in `plot.py`.

Solution: We defined the `plot.py` script as follows:

```
import time, matplotlib.pyplot as plt
from cutrod import *

def measure_time(func, p, n):
    start = time.time()
    func(p, n)
    end = time.time()
    return end - start

def plot_times(n_values, times, labels):
    for i in range(len(times)):
        plt.plot(n_values, times[i], label=labels[i])
    plt.xlabel('n'); plt.ylabel('time (s)'); plt.legend(); plt.show()

def main():
    n_values = list(range(1, 25))
    p = [i for i in range(1, 25)]
    functions = [cut_rod, cut_rod_memoize, cut_rod_bottom_up]
    labels = ['Recursive', 'Memoized', 'Bottom Up']
    times = []

    for func in functions:
        func_times = []
        for n in n_values:
            t = measure_time(func, p, n)
            print(f'{func.__name__}({n=}) took {t:.6f} seconds')
            func_times.append(t)
        times.append(func_times)

    plot_times(n_values, times, labels)

if __name__ == '__main__':
    main()
```

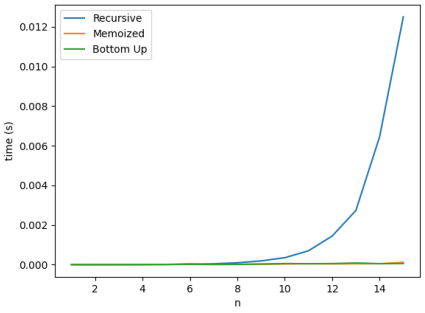
The above script basically measures the running time of the three functions for increasing values of n and then plots the running times against n. The running times are measured using the `time` module in Python. The `measure_time` function takes a function, `func`, and measures the time it takes to run the function for a given `n`. The `plot_times` function takes the running times of the three functions and plots them against n. The `main` function then measures the running times of the three functions for increasing values of n and then plots the running times against n.

For our three functions, we got the running times as follows:

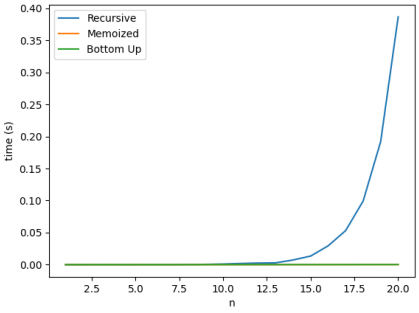


(a) n = 5

(b) n = 10



(c) n = 15



(d) n = 20

From the above plots for increasing values of n upto 20, we can see that the running time of the recursive version is the highest, which exponentially shoots up with increasing n while the memoized and bottom-up version have barely any increase in their runtimes. This is because the the recursive version recursively branches into two possiblities on each iteration from up to n; cut or don't cut. Then for each length from 0 to n, the function is exploring two possiblities - making a cut or not making a cut. This results in a binary tree where each node represents a function call, and each branch gets a decision to cut or not to cut. Then the runtime of the recursive version is $O(2^n)$, which is exponential. The memoized version, on the other hand, stores the results of the subproblems in an array and uses them to avoid recomputation of the same subproblems. This results in a polynomial time complexity of $O(n^2)$. The bottom-up version, on the other hand, iteratively computes the subproblems from the bottom up, and hence has a polynomial time complexity of $O(n^2)$ as well. Therefore, the bottom-up and memoized versions are much faster than the recursive version. (In practice, the bottom up is most often faster since it doesn't have the overhead of recursion).