**Dynamic Programming: Time Complexity Analysis of Memoization and Tabulation**

Seth Brenneman

Eastern Mennonite University

1. **Abstract**

Good algorithm design and reductions in time complexity are only becoming more important as algorithms get applied to far reaching aspects of our lives. In the 21st century, data is king. But data is useless unless you are able to parse the data and extract the desired solutions to big data problems with reasonable time complexity. To combat exponential time complexity and reduce algorithms to polynomial time, methods such as tabulation and memoization can be used to achieve this end. While tabulation and memoization achieve very similar results in a host of dynamic programming problems, they can differ quite substantially in terms of time complexity. In this specific application of tabulation and memoization, both methods were applied to a weighted interval schedule problem. More specifically, a minimum cost of travel problem between two cities in addition to a movement cost. Both implementations vastly outperformed the brute force implementation which had an exponential time complexity and was then reduced to polynomial timein both the tabulation and memoization implementations. For this specific problem, tabulation seemed to have the faster execution speed over the memoized algorithm.

1. **Background**

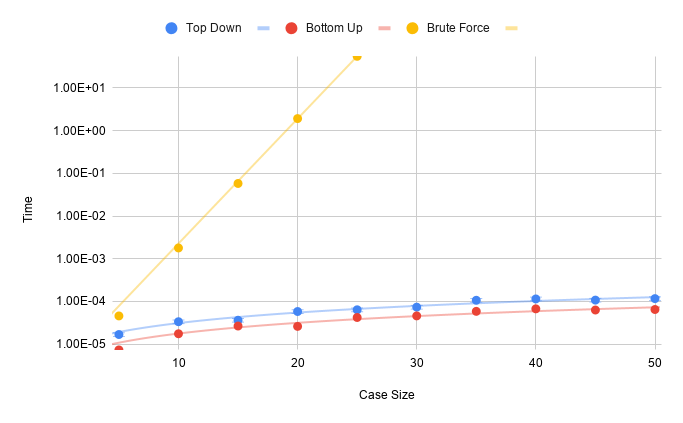
Though memoization and tabulation both vastly reduce time complexity in comparison to brute force algorithm, they go about it in different ways. The process used in memoization starts by creating a “memo” object where previously computed results can be stored and easily retrieved to cut down on redundant computations and only calculate values that have not yet been computed [1]. Memoization is also commonly used in tandem with recursion. In contrast, tabulation requires the procedural building of a table and is iterative as opposed to recursive in the case of memoization. Values are sequentially computed by looking at previously stored values in the table [2]. An array is a common data structure with which tabulation solutions can be built.

Much research has been done on how to implement both tabulation and memoization within dynamic programming problems. Most research focus on one of the implementations where in this test, both tabulation and memoization were compared to each other and a brute force algorithm.

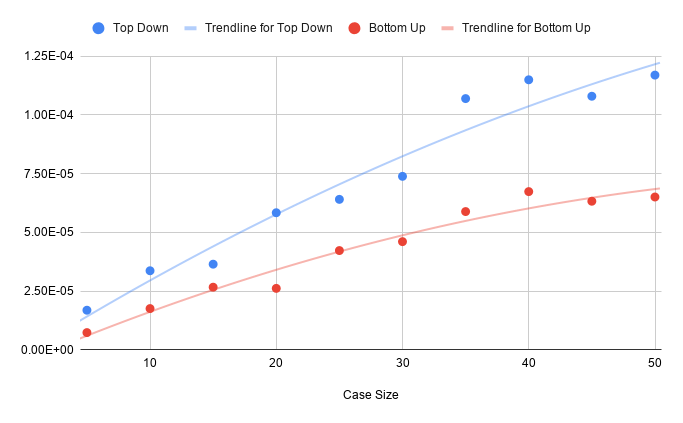
1. **Methods**

All three algorithms were applied to a minimum cost of travel problem from *Algorithm Design* by Jon Kleinberg and Eva Tardos. Specifically, chapter 6 exercise 4 [3]. Case sizes were randomly generated and ranged from a list of five integers for both New York and San Francisco travel costs all the way to an *n* of fifty in increments of five. Both the tabulation and memoization algorithms took all 10 case sizes while the brute force algorithm was only able to handle *n* up to 25 due to its inefficient design. The algorithms were written in Python version 3.8.2 on a laptop with an Intel Core i5-8250U CPU alongside 8 GB of memory. To time the speeds of each of the algorithms, the Python timeit module was used. Credit to Brandon Chupp for creating the wrapper function to make this possible.

1. **Complexity**

The brute force implementation of the algorithm clearly displayed an exponential time complexity of O(2n). In the scatter plot below of all three algorithms, the brute force implementation quickly diverges from the tabulation (bottom up) and memoization (top down) versions of the algorithm. 

The best fit line for the brute force algorithm was an exponential line and had an *R2* of 1. As previously mentioned, there are only 5 data points for the brute force algorithm as opposed to 10 in the case of the memoization and tabulation algorithms due to its inefficiency and inability to take an *n* larger than 25.

When comparing the top down and bottom up implementations, even in the graph above it is quite clear that the tabulation algorithm slightly outperformed the memoized version of the algorithm. Both of these dynamic versions reduced the time complexity from O(2n) to polynomial time of O(n2). The top down and bottom up implementations had an *R2* of .995 and .9963 respectively. The bottom scatter plot more clearly highlights the differences in speed between the two algorithms. 

1. **Experiments**

All three algorithms used the same list of randomly generated numbers ranging from one to one hundred. Different lists were generated for both New York and San Francisco movement costs by using for loops to populate lists. A movement cost for moving between the cities was a single randomly generated number between one and thirty and was generated using Python’s random module. Python’s timeit module was used to get the speed from each of the different algorithms. In the case of the tabulation and memoization algorithms, each were ran ten thousand times and divided by that same number in order to get the average run time. In the case of the brute force algorithm, as the input size *n* increased the number of times the algorithm ran had to be reduced due to inefficiencies. Implementations for each of the three algorithms can be found in the appendix.

1. **Conclusion**

When comparing dynamic programming solutions to brute force solutions to algorithm design it is clear that dynamic solutions such as tabulation and memoization clearly outperform brute force implementations. But it is more interesting and important to compare bottom up solutions and top down solutions to find which works best for the problem trying to be solved. For this specific problem of finding the minimum cost of movement between two cities, it was clear that tabulation produced the faster result. This is not to say that tabulation will always outperform memoization, rather, for this specific problem it seemed to work better. Tabulation also could have been quicker than memoization in this instance due to the specific design of each of the algorithms.

**References**

[1] Marty Hall, J. Paul McNamee. (1997). “Improving Software Performance with Automatic Memoization”. Johns Hopkins APL Technical Digest, Volume 18, Number 2.

[2] R. S. Bird. (1980). “Tabulation Techniques for Recursive Programs”. Department of Computer Science, University of Reading, Reading, Berkshire RG6 2AX, England.

[3] Jon Kleinberg, Eva Tardos. (2006). Algorithm Design. Pearson Education, Inc.

**Appendix – Code**

"""

Seth Brenneman

--------------

Analysis of Algorithms

----------------------

March 30, 2021

--------------

Presentation Code

-----------------

"""

from timeit import timeit

from random import randint

*#\* Chapter 6, Exercise 4: Given NY[] and SF[] find the minimum value*

*#\* -----------------------------------------------------------------*

NY = []

SF = []

M = randint(1, 30)

for i in range(1, 51):

    NY.append(randint(1, 100))

for i in range(1, 51):

    SF.append(randint(1, 100))

N = len(NY)

*#\* Memoized Solution (Top-down)*

*#\* ----------------------------*

def optNYSF():

    cacheNY = {0:0}

    def optNY(j):

        if j in cacheNY:

            return cacheNY[j]

        else:

            cacheNY[j] = NY[j - 1] + min(optSF(j - 1) + M, optNY(j - 1))

            return cacheNY[j]

    cacheSF = {0:0}

    def optSF(j):

        if j in cacheSF:

            return cacheSF[j]

        else:

            cacheSF[j] = SF[j - 1] + min(optNY(j - 1) + M, optSF(j - 1))

            return cacheSF[j]

    return min(optNY(len(NY)), optSF(len(SF)))

*#optNYSF()*

*#\* Tabulated Solution (Bottom-up)*

*#\* ------------------------------*

def tabulatedOptNYSF():

    MNY = [0] \* (len(NY) + 1)

    MSF = [0] \* (len(SF) + 1)

    for j in range(1, len(MNY)):

        MNY[j] = NY[j - 1] + min(MSF[j - 1] + M, MNY[j - 1])

        MSF[j] = SF[j - 1] + min(MNY[j - 1] + M, MSF[j - 1])

    return min(MNY[-1], MSF[-1])

*#tabulatedOptNYSF()*

*#\* Brute Force Solution*

*#\* --------------------*

def bruteForceNYSF():

    def optNY(j):

        if j == 0:

            return j

        else:

            return NY[j - 1] + min(optSF(j - 1) + M, optNY(j - 1))

    def optSF(j):

        if j == 0:

            return j

        else:

            return SF[j - 1] + min(optNY(j - 1) + M, optSF(j - 1))

    return min(optNY(len(NY)), optSF(len(SF)))

*#bruteForceNYSF()*

*#\* Credit to Brandon Chupp for figuring out the timeit wrapper function*

*#\* --------------------------------------------------------------------*

def wrapper(func, \*args):

    def wrapped():

        return func(\*args)

    return wrapped

memoized = wrapper(optNYSF)

tabulated = wrapper(tabulatedOptNYSF)

bruteForce = wrapper(bruteForceNYSF)

print("size:", N, "memoized:", timeit(memoized, number = 10000)/10000)

print("size:", N, "tabulated:", timeit(tabulated, number = 10000)/10000)

print("size:", N, "brute force:", timeit(bruteForce, number = 1000)/1000)