Time Complexity Analysis of Bottom-Up, Top-Down, and Brute Force Methods: Case of Job Scheduling

Finnian N.G. Wengerd

Eastern Mennonite University

I. Abstract

This is a look at the different possible dynamic approaches available to solve the job scheduling problem presented in chapter six of Tardos's book *Algorithm Design*. Two dynamic programming styles, Top-Down and Bottom-Up, were timed against each other with a Brute-Force algorithm as the control group. The Brute-Force algorithm took the longest($O(3^n)$), surprisingly followed by Bottom-Up ($O(n)$) and Top-Down ($O(n)$).

II. Background

In today's increasing demand for faster programs with high level operations, the Brute-Force programs we relied on are now stepping aside for a smarter programming style called "Dynamic Programming." Dynamic programming was first introduced in 1950 by mathematical Richard Bellman who named it as such even though, at the time, it had little to do with programming or computers at all. In fact, he was trying to shield his research from the prying eyes of US Secretary of Defense Charles Wilson. Although there is no formula to find an optimal solution, a dynamic program should have some of the following; recursive calls to an optimum value, a matrix to remember each optimum value (usually a cache), filling a matrix in a bottom-up style, and a traceback to the matrix to find the needed optimum value[1].

Dynamic programming has become useful outside of the realm of computer science. It has shown up in marketing, finance, economics, and operation management[2]. For example, in

Tardos and Kleinberg's problem of scheduling a plan of action for a company's operation scheduling shown in chapter six of his book *Algorithms Design*, he is working dynamic programming into operation management[3].

A company has the option every week to tackle a high stress or low stress operation that will bring in revenue equal to the operation's description. Both types of operations are required to be completed in a week. That being said, a high stress operation requires all hands on deck. Every employee is required to rest for a week in order to maintain a functional and focused crew. The basic question is whether to choose a low or high stress operation each week.

Given a sequence of n weeks, the manager must come up with a plan of action with the options: low-stress, high-stress, or rest resulting in the optimum sequence to bring in the most revenue.

The problem: Given sets of values $l_1, l_2,....l_n$ and $h_1, h_2,....,h_n$, find a plan of maximum value (revenue). (Such a plan will be called optimal).[3]

Example:

	WEEK 1	WEEK 2	WEEK 3	WEEK 4
l_i	10	1	10	10
h_i	5	50	5	1

Optimal plan : $(r, h, l, l) = 0 + 50 + 10 + 10 = 70$

Methods

For illustrative purposes, the following test case will be solved.

Given the test case:

	$WEEK_1$	$WEEK_2$	$WEEK_3$	$WEEK_4$
l_i	1	30	60	5
h_i	80	20	70	10

Find the optimal solution.

Using the top-down approach, a formula can be found to find an optimal solution for the nth week. In building this formula, $WEEK_1$ is used as the starting position. The preset values $OPT(-1) = 0$ and $OPT(0) = 0$ are given.

If the optimal solution for $WEEK_1$ is needed, the value would simply be the maximum value of l_0 and h_0 given that l and h are arrays. So:

$$OPT(1) = max(l_0, h_0) = max(1, 80) = 80$$

$$Plan = (h_0)$$

WEEK	OPT Value	Plan
1	80	h_0

Finding the solution for weeks beyond $WEEK_1$ requires knowledge of the previous weeks. The formula for $WEEK_i$ should then chooses if the $WEEK_{i-1}$ should be skipped. For example: in $WEEK_2$ the plan is found by taking the maximum value of taking a low stress job (l_1)plus the optimal value from the previous week (OPT(1)), and the value of taking a high stress job (h_1) and taking a rest the week before.

$$OPT(2) = max(l_1 + OPT(1), h_1) = max((30 + 80), 20) = 110$$

$$Plan = (h_0, l_1)$$

WEEK	OPT Value	Plan
1	80	h_0
2	110	l_1

$WEEK_3$ requires the information from $WEEK_1$ and $WEEK_2$ where $WEEK_2$ will be used as OPT(2) and $WEEK_1$ will be added to the value of h_2 as the result of adding the optimum value of $WEEK_1$, taking a rest in $WEEK_2$, and then choosing a high-stress job in $WEEK_3$:

WEEK	OPT Value	Plan
1	80	h_0
2	110	h_0, l_1
3	170	h_0, l_1, l_2

$$OPT(3) = max(l_2 + OPT(2),\ h_2 + OPT(1) =$$

$$max((60 + 110),\ (70 + 80)) = 170$$

$$Plan = (h_0,\ l_1,\ l_2)$$

Now that all of the elements are present, a formula can be determined and applied to $WEEK_4$

$$OPT(n) = max(l_{n-1} + OPT(n-1),\ h_{n-1} + OPT(n-2)\ )$$

$$OPT(4) = max(l_3 + OPT(3),\ h_3 + OPT(2) =$$

$$max((5 + 170), (10 + 110)) = 175$$

$$Plan = (h_0,\ l_1,\ l_2, l_3)$$

WEEK	OPT Value	Plan
1	80	h_0
2	110	h_0, l_1
3	170	h_0, l_1, l_2
4	175	h_0, l_1, l_2, l_3

The experiment that took place consisted of three different algorithms; Top-Down, Bottom-Up, and Brute force. The Top-Down method parses through both of the arrays l and h starting at the last element and comparing the two values as shown above. This algorithm queues OPT Values as needed contrary to the Bottom-Up algorithm approach.

The Bottom-Up approach, like the Top-Down method, parses through both of the arrays l and h. However, starting with the first element of the array, it moves through the values of n and solves every OPT Value until the arrays have ended.

The Brute-Force combs through l and h and creates every possible combination of paths and finds the value of each path. The algorithm, then, searches through all the plans available and rids itself of any invalid path patterns (for example $[h_1,h_2]$). Finally, it returns the path and value of the highest valid path.

Each algorithm will be running increasing n values until the Brute-Force algorithm becomes unusable. Test cases will be made randomly within the code before the start of the datetime module. A test case will consist of a set n amount of values for both l and h with values ranging from 0 to 1,000 inclusive using the random module. Each trial of n will be run 1,000 times to get an average runtime.

III. Complexity

The top-down algorithm starts at the nth value of the total weeks. The algorithm then checks the value for n-1 recursively until it stops at n_0. From here, it finds the OPT Values from n_0 up to the final value. These recursive calls give the Top-Down algorithm a final Big-O of 2n or, for all intents and purposes, n.

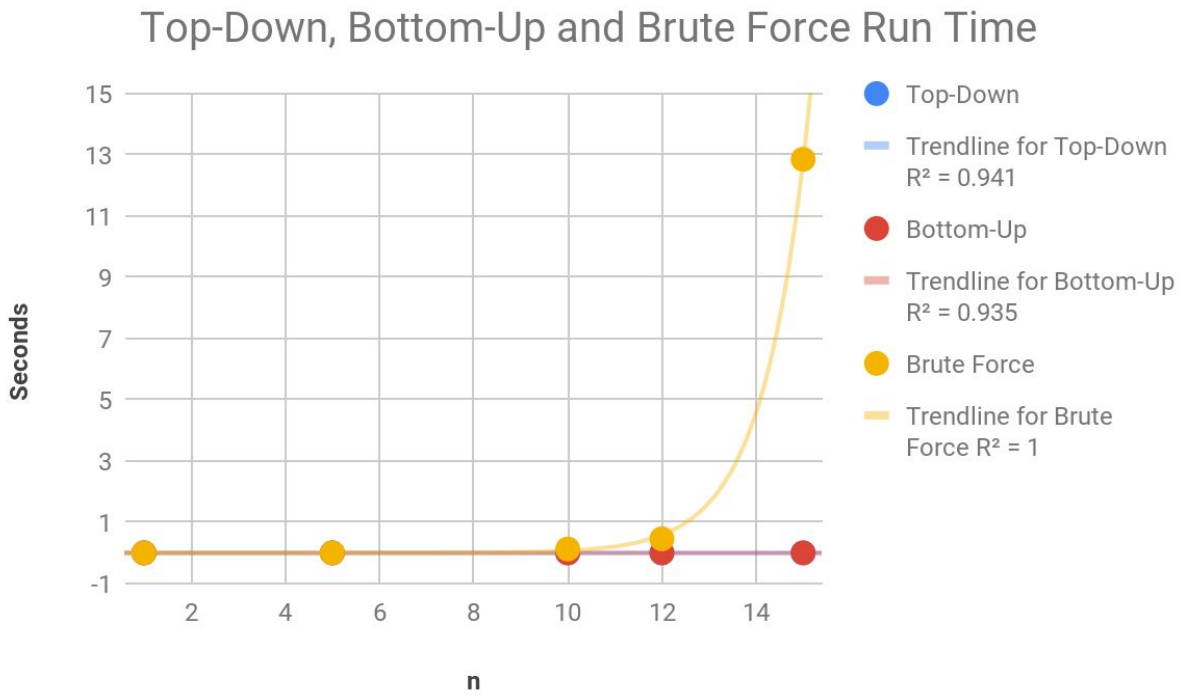
The Bottom-Up algorithms starts at n_0 working its way towards the nth value, solving each OPT Value of the current week. Working in a straight line through the weeks will give the Bottom-Up algorithm a final Big-O of n.

As the Brute-Force algorithm finds all the combinations possible of rest, low-stress, and high-stress choices it has the largest Big-O value. It starts with three possible solutions for the

first value of n; rest, low-stress, or high-stress. From each of these spring another three values of what the next possible value would be resulting in cubing the previous value. If there are n values in the array, the value of three must be multiplied n times resulting in a Big-O value of 3^n.

IV. Empirical Experiments

The Brute-Force algorithm ran for the longest amount of time which grew exponentially as the value of n increased [4].



The differences between Top-Down and Bottom-Up were negligible, yet still consistent. Top-Down ran slightly faster than Bottom-Up in every case [5].

V. Conclusion

As expected, the Brute-Force algorithm took an exeptionaly longer amount of time that Top-Down and Bottom-Up. This discrepancy was logical yet surprising when played out. While running the Top-Down and Bottom-Up algorithms with a n of fifteen, 1,000 times took less than a second, the Brute-Force algorithm took about four hours to complete.

Unexpectedly, the run time of the Top-Down algorithm took less than the Bottom-Up algorithm. Theoretically, O(BU) = O(TD), yet in this experiment, the Top-Down algorithm out performed the Bottom-Up algorithm. Possible discrepancies may lie in the fact that the test cases were randomized for each individual run of each individual algorithm so the test cases were not globally set across all algorithms.

VI. References

Eddy, Sean R. *What Is Dynamic Programming*. Nature Publishing Group, 2004AD,www.marcottelab.org/users/CH391L/Handouts/Lecture5-NBT-primer-dynamic programming.pdf.

Denardo, Eric V. "Dynamic Programming." *Google Books*, Dover Publications, 15 Dec. 2012, books.google.com/books/about/Dynamic_Programming.html?id=KKY1R_eVRyoC.

Kleinberg, Jon, and Eva Tardos. *Algorithm Design*. Pearson Education, 2014.

VII. Appendix A-Code

```
'''
Finnian Wengerd
Algorithms
Daniel Showalter
Eastern Mennonite University
Job Scheduling: Stress Level and Optimum Value (TOP-DOWN)
'''
'''
.....
Given a sequence of n weeks, the manager must come up with a plan of action with the options:
low-stress, high-stress, or rest resulting in the optimum sequence to bring in the most revenue.

The problem: Given sets of values l1, l2,....ln and h1, h2,....,hn, find a plan of maximum value
(revenue).
(Such a plan will be called optimal).
.....
'''
from datetime import datetime
import random

opt_cache = {
        -1:0,           #This begins the cache with pre-determined values for indexes of i less
than 1
        0:0,
        }


'''Recursive function that returns the optimum value of the given parameter i'''
def OPT(i):
   #print(i)
   optimum_i = 0
   #print(i, opt_cache)
   if (opt_cache.get(i)!= None):
      return opt_cache[i], plan
   else:
      optimum_i = max((l[i-1]+OPT(i-1)[0]),(h[i-1]+OPT(i-2)[0]))
      opt_cache[i] = optimum_i
      if optimum_i == (l[i-1]+OPT(i-1)[0]):
         plan[i-1]=("l")
```

```
        elif optimum_i == (h[i-1]+OPT(i-2)[0]):
            plan[i-2] = "r"
            plan[i-1]=("h")
        return optimum_i, plan



'''Starting function that focuses on integrating each test case and deals with printing out the results (True if the expected result is equal to the optimum value)'''

def My_Print(l,h, expected):
    plan = [0]*len(h)
    starttime = datetime.now()
    return_value = OPT(len(l))
    optimum_value = return_value[0]
    plan = return_value[1]
    #print(optimum_value)
    #print(expected == optimum_value)
    endtime = datetime.now()
    deltaT = endtime-starttime
    totalTime.append(deltaT.total_seconds())
    return deltaT



'''
Start Code
'''

totalTime = []
deltaT = 0
l = []
h=[]
for i in range(12):
    i = random.randint(0,1000)
    l.append(i)

for i in range(12):
    i = random.randint(0,1000)
    h.append(i)
expected = 375477
plan = [0]*len(h)
for i in range(1000):
    My_Print(l,h, expected)
```

```
totalseconds = 0
averageTime = 0
for i in totalTime:
   totalseconds +=i
averageTime = (totalseconds/len(totalTime))

print("Average runtime: %f seconds." %(averageTime))




'''
Finnian Wengerd
Algorithms
Daniel Showalter
Eastern Mennonite University
Job Scheduling: Stress Level and Optimum Value (BOTTOM-UP)
'''
'''
.....
Given a sequence of n weeks, the manager must come up with a plan of action with the options:
low-stress, high-stress, or rest resulting in the optimum sequence to bring in the most revenue.

The problem: Given sets of values l1, l2,....ln and h1, h2,....,hn, find a plan of maximum value
(revenue).
(Such a plan will be called optimal).
.....
'''
from datetime import datetime
import random

opt_cache = {
        -1:0,           #This begins the cache with pre-determined values for indexes of i less
than 1
        0:0,
        }


'''Recursive function that returns the optimum value of the given parameter i'''
def OPT(i):
   optimum_i = 0
   #print(i, opt_cache)
   if (opt_cache.get(i)!= None):
      return opt_cache[i], plan
   else:
```

```
        optimum_i = max((l[i-1]+OPT(i-1)[0]),(h[i-1]+OPT(i-2)[0]))
        opt_cache[i] = optimum_i
        if optimum_i == (l[i-1]+OPT(i-1)[0]):
            plan[i-1]=("l")
        elif optimum_i == (h[i-1]+OPT(i-2)[0]):
            plan[i-2] = "r"
            plan[i-1]=("h")
        return optimum_i, plan



'''Starting function that focuses on integrating each test case and deals with printing out the
results (True if the expected result is equal to the optimum value)'''

def My_Print(l,h, expected):
    plan = [0]*len(h)
    starttime = datetime.now()
    for i in range(len(l)):
        return_value = OPT(i)
    optimum_value = return_value[0]
    plan = return_value[1]
    #print(expected == optimum_value)
    endtime = datetime.now()
    deltaT = endtime-starttime
    totalTime.append(deltaT.total_seconds())
    return deltaT



'''
Start Code
'''

totalTime = []
deltaT = 0
l = []
h=[]
for i in range(12):
    i = random.randint(0,1000)
    l.append(i)

for i in range(12):
    i = random.randint(0,1000)
    h.append(i)
expected = 16969
```

```
plan = [0]*len(h)
for i in range(1000):
    My_Print(l,h, expected)

totalseconds = 0
averageTime = 0
for i in totalTime:
    totalseconds +=i
averageTime = (totalseconds/len(totalTime))

print("Average runtime: %f seconds." %(averageTime))


'''
Finnian Wengerd
Algorithms
Daniel Showalter
Eastern Mennonite University
Job Scheduling: Stress Level and Optimum Value (Brute-Force)
'''
'''
.....
Given a sequence of n weeks, the manager must come up with a plan of action with the options:
low-stress, high-stress, or rest resulting in the optimum sequence to bring in the most revenue.

The problem: Given sets of values l1, l2,....ln and h1, h2,....,hn, find a plan of maximum value
(revenue).
(Such a plan will be called optimal).
.....
'''
#This code was provided by Professor Daniel Showalter
from itertools import product
from datetime import datetime
import random


def score(path):
    path = ''.join(path)
    total = 0
    for i,v in enumerate(path):
        if v == 'l':
            total += l[i]
        elif v == 'h':
            total += h[i]
        if 'lh' in path or 'hh' in path: #If another job before a high stress job
```

```
            return 0                    #remove the path from the list

    return total



l = []
h = []
totalTime = []

for i in range(12):
    i = random.randint(0,1000)
    l.append(i)

for i in range(12):
    i = random.randint(0,1000)
    h.append(i)

for i in range(1000):
    best = 0
    starttime = datetime.now()
    for path in product("rlh", repeat = len(h)): #1 is low, 2 is high, 0 is none
        best = max(best,score(path))
        #print(path,best)

    endtime = datetime.now()
    deltaT = endtime-starttime
    totalTime.append(deltaT.total_seconds())

totalseconds = 0
averageTime = 0
for i in totalTime:
    totalseconds +=i

averageTime = (totalseconds/len(totalTime))
print("Average runtime: %f ." %(averageTime))
```

Appendix B-Representative Sample of Timed Results

Length of n	Top-Down Average Time of 10,000 Runs (s)	Bottom-Up Average Time of 10,000 Runs (s)	Brute-Force Average Time of 10,000 Runs (s)
1	0.000002	0.0000011	0.000011
5	0.000002	0.000007	0.000966
10	0.000005	0.000009	0.126192
12	0.000006	0.000018	0.458773
15	0.000007	0.000023	12.85436

Length of n	Top-Down Average Time of 10,000 Runs (s)	Bottom-Up Average Time of 10,000 Runs (s)
100	0.000006	0.000093
200	0.000008	0.000273
400	0.000009	0.000428
991*	0.000012	0.001074

*991 was used as it is the maximum recursion depth in python 3.7.2