

Desenho e Analise de Algoritmos

2017/2018

Projeto 1

Autor

Francisco João Guimarães Coimbra de Almeida Araújo, n.º 45701

I start this report by presenting a gist of the issue and explaining the logic behind the solution and what kind of problems we can solve with it.

The problem can be resumed as such, we have n weeks labeled 1… n with each week ‘i’ having a weight associated to the cargo needing transport. Each week i now has a weight Wi. Each week can have two companies transporting it A or B, being that A has a rate R where we need to multiply the weight to get the cost, and B has a constant value C where we must multiply by 4 since we need to use it 4 consecutively times. The goal is to minimize the cost of transportation for the all the n weeks.

This algorithm can solve problems where we need to minimize the costs associated with an associated linear sequence of events where we need to pass through all of them.

The logic behind the implementation of the solution was as such:

Given an instance of the problem, and an optional solution O to that problem. I knew that either the week 1 (the first one) can either be a week where we choose A or a week where we choose B to transport the supplies. When I explored both sides of this coin I knew I could reach a conclusion.

If week 1 is associated with A in the optional solution then the optimal cost will be the optimal cost of the next week , since we know all weeks are associated, plus the rate for the current week R multiplied by its weight Wi.

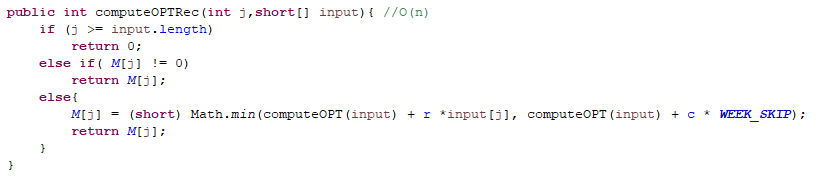
Or it’s associated with B then the optional cost will be the optimal cost of 3 weeks from now, since we need to skip the next 3, plus the constant cost associated with B C times the weeks we need to mandatory use them 4.

All of this suggests that finding the optimal solution for the weeks {1,2,…,n} involves looking at the solution of the form {j,…,n} and let OPT(j) denote the value of this solution. The optional solution we are seeking is *O1*, with value OPT(1). For the optional solution Oj on {j,..,,n}, using the reasoning above and knowing that either j is associated with A or with B in Oj. Since these are precisely the two possible choices, it follows that

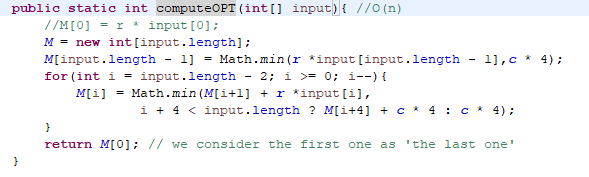
(***1,1) The Optimal Cost formula***

The problem now is how to decide if week j belongs to A or B in the Oj. It belongs to A if and only if the first of the options above is at least as good as the second.

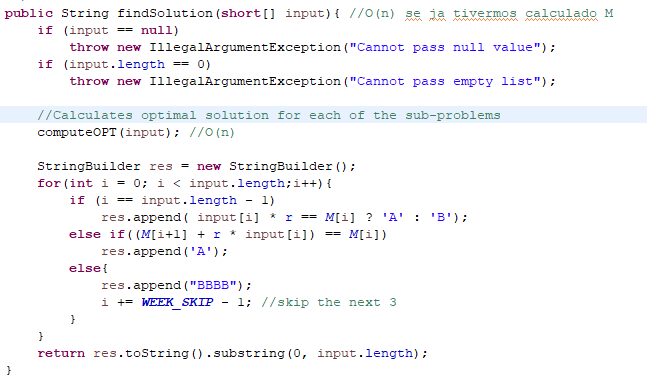
From there I reached this recursive function.



Since I wanted the algorithm to not suffer from StackOverflowErrors I decided to implement an Iterative algorithm to the same problem. I realized the key was the array M since it encodes the notion of the optimal solution to the sub problems, and use ***(1,1)*** to define the value of M[j] based on the values that come earlier in array (that are bigger than ours). Once we have the array M, the problem is solved. M[0] contains the optimal solution to the full instance, since we start from the end, and findSolution can be used to trace back through M efficiently and return the optimal solution.

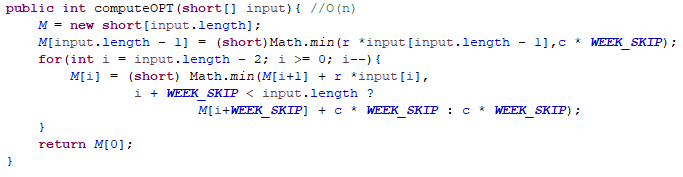


I then pass through all the values of M to check if the corresponding index corresponds to the first half of **(1,1)** or the second part. If it corresponds to the first part then I know it’s part of A otherwise I know it’s part of B and skip the next 3.



# Detailed mathematical analysis of its correctness, space and time usage

# Correctness



Following the logic presented before we can then start to conduct the proof.

We start by proving that the recursive function works. By definition OPT(>n) = 0. If we take a 0 < j <= n, and assume that computeOPT(j) correctly computes OPT(i) for all i < j. By the induction hypothesis, we know that Compute-OPT(j+1) = OPT(j+1) ; and from ***(1,1)*** it follows that

Since the Iterative algorithm uses **(1,1)** to find OPT(j) for M[j], if the previous statements holds then we can say with confidence that the Iterative version of the recursive algorithm works

# Space Usage

By analyzing the code we can see that the variables that the two variables that occupy the most space are M (short[n]), input (short[n]). So as we can see we know that the space occupied should grow according to n itself Space = 2n \* short = 2n \* 16 = 32n which follows a linear growth.

# Time Usage

Since my solution to the problem first need to identify the optimal cost for each sub solution it calls method computeOPT before it can reach a conclusion about the solution itself. So we need to analyze both of these methods, computeOPT and ShippingCostSolution itself.

The running time of computeOPT is clearly O (n) since it explicitly runs for n iterations and spends constant time in each and since shippingCostSolution passes through all values of M (memory) it makes a total of O(n) checks to the memory, and since it spends constant time per call we find the solution in O(n) time given an array M of the optimal values of the sub-problems.

So the total becomes O(n) + O(n) = O(2n) = O(n).

So we can say with confidence that my solution finds the solution to the problem in O (n) time.

# Identification and explanation of inputs

Since we will always need to pass through all the values of every week n when finding the sub optimal solutions for M it will not be taken into account when considering the worst, average and best-case performance. What will be taken is when finding the solution itself, since we might have to iterate through all the values of M or we may be able to skip a few if the optimal solution so allows.

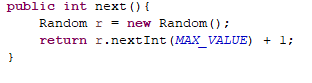
The worst case scenario obviously will happen when we have to iterate through all the values of M, since then we will have to pass through n values. This case happens if the best solution for every week is company A. So the worst-case performance will be O (n)

The best-case scenario will occur when we do not have to pass through almost any value in n, if the optimal solution consists of every single week being supplied by company B then we will only need to pass through n/4 values, since we skip 4 weeks when choosing B. Which is much better than n values previously seen. So the best-case scenario is O (n/4).

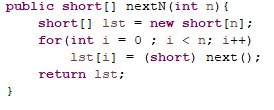
The Average-case performance is one where we will not need to pass through all the values of the n weeks, but we also have to pass through more than n/4 of the weeks So if P is the number of weeks we will need to verify we know that n/4 < P < n. And so we know that the average-case performance will be O (P)

# Numeric Characterization of the algorithm worst- and average-case space and time usage

The generator implemented in the project is a simple input generator called InputGenerator that generates input to the problem by generating the next value when needed.



It generates and array of Integers in a range of 0 to 100, and can be used with a fixed length.



Using the previous demonstrated generator we proceed to the numerical tests, each test with different size was run 1000 times with inputs generated from the previous generator.

**(3,1) Average Time tests 1000 times for each N (note it takes around 2 min to run with 1000 tests)**

|  |  |  |
| --- | --- | --- |
| **SIZE** | **TIME** | **CATEGORY** |
| N = 10 | 800 nanoseconds | < 1 sec |
| N = 100 | 1879 nanoseconds | <1 sec |
| N = 1000 | 7447 nanoseconds | < 1sec |
| N = 10000 | 87767 nanoseconds | <1 sec |
| N = 100000 | 1010141 nanoseconds | <1 sec |
| N = 1000000 | 9889219 nanoseconds | <1 sec |

Following the results of the numerical testing we can see it follows the expected values in all categories, since it’s always <1 second, we can also see it has a growth similar to the one we would expect since for each increase in \*10 of N we have an approximate increment in Time.

**(3.1) Average Size tests 1000 times for each N**

|  |  |  |
| --- | --- | --- |
| **SIZE** | **BYTES** | **CATEGORY** |
| N = 10 | 367987 | 0 MB |
| N = 100 | 369559 | 0 MB |
| N = 1000 | 378543 | 0 MB |
| N = 10000 | 505270 | 0 MB |
| N = 100000 | 1877902 | 1 MB |
| N = 1000000 | 11893575 | 11 MB |

**//TODO**