Jacob Bothell

CS404 – Algorithms and Complexity

Appie Van De Liefvoort

April 10, 2017

# Introduction

This is a study of matrix chains and different heuristics for calculating the most efficient execution tree possible. Most efficient is categorized as the fewer number of simple multiplications; for example when multiplying a 6X4 and a 4X2 there will be 6\*4\*2 or 48 basic multiplication operations. The problem was approached with the language C++ to implement seven different algorithms that allow a window into the understanding of the execution tree efficiency. This is a worthwhile experiment because matrix multiplication is used to create computer graphics and large scientific calculations, namely physics research, and both of these areas would benefit from faster execution for obvious reasons.

# Pre-heuristic decisions and discussion

In all of these experiments there is an assumption that the matrixes and subsequent dimensions cannot be reordered, which makes them an ordered list that has an order of original input. In other words the order is completely unique and could not be expressed in any other way in order to have the same mathematic result. The general idea of dimensions in this context, matrix multiplication, because we are would be calculating the “Dot Product” of the matrixes, means that the second dimension of the first matrix must match the first dimension of the second matrix. This leads to the ability to express the dimensions of n matrixes in a list of numbers n+1 long; for example:

The matrixes can be expressed as

This concept makes the easy transition into the decision of an array as a data structure that holds the dimensions in the way described as above. However due to the in-deterministic nature of the input (we do not know the exact number of elements in the array) the use of resizable arrays or vectors in C++ was the final decision in data structure.

At first implementation and testing there was a problem with numbers not coming out correctly, either negative or wildly incorrect, and this problem was finally mediated with the type long double which is an 8-byte number that can go to approximately 15 digits. The need was found to store the dimensions to be multiplied to calculate the number of base multiplications as a long double before they are multiplied to avoid overflow errors.

Another assumption is made that all of the dimensions are already known and that they are in the best/only possible order. This could influence the output because some of the dimensions could perhaps benefit from being multiplied with some other matrix. This is just an assumption because it is out of the scope of this particular experiment and could warrant further investigation.

## Implementation Details

These tests are done in a testbed designed to simply test the heuristic and how many multiplications would happen if they were to actually do them. There is no actual matrix multiplication going on these tests. The testbed is implemented in C++ for a multitude of reasons; some of which are: familiarity with the language, available control over memory and pointers, basic data structure power, and speed of execution.

The first of these would seem obvious that it would be a large design consideration because the familiarity with the particular language in question not only breeds easier/faster development, but also the ability to manipulate the data very carefully and with much care. With C++ also comes control that allows everything to dance in accordance to what is desired and very precise manipulation and moving of data; pointers are an accidental complexity to using this language and the structure decisions but they are kept in check and carefully watched to ensure proper use. The main data structure used for this testbed is the vector this wonderfully useful dynamic array was used for its dynamic properties in length along with the built in methods provided that make execution of the task at hand easier and much more elegant; these functions include: .push\_back(), .erase(), .size(). Along with the usefulness of iterators that allow moving over the vector and aid in some of the previous functions.

The basic organization of the testbed consists of a heuristics class, a test wrapper class, and the main program. These classes were created to have separation of concerns with regards to the three different needs of this program which are having the heuristic algorithms implemented in executable code, the collection of data, and console and file I/O. The implementation of these classes is detailed as follows.

The heuristics class contains the dimensions, as a vector, all of the functions of the heuristics, of which there are 7, along with getters, setters, and constructors; this class is called matrixHeuristics.

matrixHeuristics is then contained in a wrapper class that makes the heuristic function calls and contains the return values. There is also a function that allows the random heuristic to be run a statistically significant number of times while also keeping the data from all executions. This wrappers main function is information hiding; it allows the main program to make one function call runH() after setting up the dimensions to be used and all of the heuristics are run and data collected.

That leaves the main function to handle console and file I/O for ease of use, and extra functionality on the user side. The extra functionality includes being able to run a ‘standard’ test which collects data from different sized arrays with random integers as dimensions, there is also the ability to enter dimensions from the console and from a file, along with being able to test the heuristics for correct implementation with console I/O dimensions.

# Experimental Design

The seven different heuristics that are under investigation in this experiment are as follows:

* Strategy A: Remove the largest dimension from the group
* Strategy B: Do the most expensive matrix multiplication first
* Strategy C: Remove the smallest dimension first
* Strategy D: Do the least expensive matrix multiplication first
* Strategy E: Random execution
* Strategy F: In order of appearance
* Strategy O: The “Optimal” heuristic

Each of these heuristics will be expanded upon, in detail, with algorithmic, mathematic and implementation focuses.

All heuristic code can be found in Appendix A.

Strategy A has to do with removing the largest dimension from the set of possible selections. The set of possible selections is a subset of all of the dimensions, which includes all but the first and last of them. The actual algorithm is as follows: go through the list of dimensions to find the largest inner dimension, not the first or last in the list, and do that multiplication, which in turn gets rid of the inner dimension then continue in the same manner. This particular algorithm has a simple best, worst, and average case that is common to most of these algorithms and it is as follows:

Best worst average

These are the same because the algorithm says that all of the values must be looked at before the decision can be made about which is the largest. This algorithm has merit since ridding the dimensions of the largest removes it from contention of adding to subsequent multiplications. The algorithm does have its faults because neither the first nor last item in the list of dimensions is considered to carry any weight and thus if a large number were to happen to be in either of those positions it could add a significant amount of multiplications without the ‘knowledge’ of the algorithm.

This algorithmic discussion leads to an implementation of a loop that iterates through the list finding the largest value; after the largest element has been found the calculations are done in-order for the largest value to be considered to be the inner dimension and the dimension is removed from the list of dimensions. There is then an outer while statement that checks if there are only two dimensions left. The iterating is done with an iterator because it is a convenient and fast way to move through, save, and reference the location in a vector for later manipulation. This iterator is simply pointed to the second element and not allowed to go the last element to not allow the selection of those values. The iterator also gives us a convenient way to delete specific items in the vector and not to leave a hole in an array with the use of the .erase() method built into the vector container; so now there is always a way to know the exact number of dimensions left to deal with. This particular functionality is leveraged in almost all of the heuristics.

Strategy B has to do with doing the most expensive matrix multiplication first. This type of algorithm is a bit more expensive in general than the previous type, strategy A, because the number of multiplications must be calculated with each dimension to determine which multiplication would in fact be the most expensive. The actual algorithm goes like this: with each dimension, excluding the first and last, calculate the cost of having that as the common dimension between matrixes and find the one that is the most expensive. After this has been determined save the multiplication by adding it to a sum of the rest of the multiplications and then remove it from the list to be multiplied. Continue in this manner until there exists only two dimensions. This algorithm has the same asymptotic relationship as strategy A but will be more expensive because of the extra multiplications in each iteration. This heuristic has the possibility of success because it may be worthwhile to remove the most costly multiplication as early in the game as possible. This heuristic also works under somewhat of a false assumption that this multiplication could not possibly be avoided entirely, but still there is no harm in seeing how it stacks up against other heuristics in case these thoughts are incorrect.

The implementation to this is self-evident and very similar to the one before it except that the multiplication is moved into the for loop and the result of this calculation is then the basis for choosing a dimension. Since the number is already calculated it can be stored to save the calculation in a sum outside of the loop. Two items must be saved so that not only the number of multiplications can be remembered, allowing the calculation to not have to be done again, but also the inner dimension of that multiplication can be remembered for removing from the vector. All of this is done in a similar to exact same way as heuristic A

Strategy C concerns its self with removing the smallest inner dimension from the list. It is exactly the same as A except that the smallest value is being looked for instead of the largest. This is worthwhile to explore for several reasons: one there is not much work to create this function, as discussed above, and two perhaps if heuristic A does not afford the efficiency perhaps its opposite could. This strategy could possibly be better because we always concern ourselves with the smallest possible value that could be added to the matrix multiplications.

The implementation is also exactly the same except the ‘flipping’ of the inequality in the for loop of heuristic A so that it is finding the smallest instead of the largest.

Strategy D has to do with removing the least expensive matrix multiplication. This is the exact same algorithm as B with the exception of looking for the smallest instead of the largest dot product. This may work as a quick and efficient way to calculate a near optimal matrix chain because it is the basis of how the optimal actually works, discussed a little later, it simply lacks the ability to ‘look back’ and fix incorrect decisions.

The implementation is also exactly the same as B except for the flipping of the inequality in the for loop that allows the smallest to be found instead of the largest.

Strategy E creates a random execution tree. This algorithm is cheap and ‘easy,’ but can lead to some incredibly inefficient decisions as far as multiplying goes. The algorithm goes like this: choose a random inner dimension and do that multiplication; continue like this until there are only two dimensions left. To make the resultant numbers somewhat statistically significant this procedure is repeated 2n times where n is the number of dimensions to be multiplied in the initial set. This is but a small sampling of the possible random execution trees that exist, as they follow the Catalan numbers, but it is significant enough for the purpose of this investigation. This algorithm has a linear complexity but with the introduction of the statistical significance requirement the implementation becomes quadratic in nature. This heuristic, I believe, mimics the reality of matrix multiplication in most applications, not because matrixes are chosen at random but they are most likely given to the multiplication process in a random order and that order is mimicked in this heuristic. That is not to say that there are not ways to circumvent this possible problem in a program, such as a video graphic game, but this may be the way that it is handled in less sophisticated applications.

The implementation is simply a while loop looking for there to only be two dimensions with a random number generator that is given the domain of possible inner dimensions, which is updated during every loop, and the multiplication is done and saved; then the item is deleted from the vector. The method of creating random numbers is as follows: a mersenne twisted engine with a seed of a random hardware generated number, both of which are re-set before each number choosing, is used to choose a number from a uniform distribution that extends from one to n minus one. There is another loop in the container to this heuristic, the container structure will be discussed in more detail later, which continues to execute for 2n times with n being the number of dimensions in the original vector. The container also saves all of the outputs of this heuristic.

Strategy F simply goes in order of index in the vector. This would be the “ignorant” approach or if you did not consider the cost of matrix multiplication or did not care. This algorithm takes the second item and makes that the inner dimension of the multiplication, does the multiplication, deletes the item, and continues until there are only two dimensions. This algorithm is linear in nature. This situation may mimic a system where there would be no way to send the multiplying routine more than one matrix at a time perhaps because of memory constraints in an embedded system or in a real-time application. I believe this could also be the approach if the sizes of the matrixes do not deviate from one to another or they are all very small in size making this approach almost identical to what the optimal would be.

The implementation is a while loop that checks for more than two dimensions and does the calculations with the first inner dimension while also taking into consideration all of the things mentioned in the previous strategies.

Strategy O is the ‘optimal’ approach to calculate the matrix chain. This approach involves dynamic programming to avoid re-calculating the value of matrix multiplication. This lowers the time complexity of the algorithm but does not change the O of the algorithm over all, which is cubic, and also allows us to find the smallest possible number of basic multiplications. The general algorithm for this strategy is looking at levels of multiplication and finding the optimal value as the algorithm progresses. This means first multiplying the basic matrixes… the step of finding an optimal is not chosen on this first step because there are no other paths to choose from at this point. It is easiest to visualize this as a table that is progressively filled out. An example will be evaluated along with the algorithm explanation as to make it easier to understand.

|  |  |  |  |
| --- | --- | --- | --- |
| 4X6  m = 0 | 4X5  m = 120 |  |  |
|  | 6X5  m = 0 | 6X8  m = 240 |  |
|  |  | 5X8  m = 0 | 5X4  m = 160 |
|  |  |  | 8X4  m = 0 |

The next step is to calculate the next level of multiplications. To do this a simple multiplication will not be made alone, but to find a position on the table each corresponding row and column is considered. The smallest value consisting of a sum of the multiplication of those two matrixes and the number it took to get that matrix is found and is filled into that table position. This procedure is followed until the top right cell is calculated which is the final matrix chain and cost for it. For this particular example the solution is shown and the corresponding chain is highlighted.

The possible m values to put in the pink square are:

Multiplying 4X6 and 6X8: (4 \* 6 \* 8) + 0 + 240 = 432

Multiplying 4X5 and 5X8: (4 \* 5 \* 8) + 120 + 0 = 280 🡨 this is the smaller of the two

|  |  |  |  |
| --- | --- | --- | --- |
| 4X6  m = 0 | 4X5  m = 120 | 4X8  m = 280 |  |
|  | 6X5  m = 0 | 6X8  m = 240 |  |
|  |  | 5X8  m = 0 | 5X4  m = 160 |
|  |  |  | 8X4  m = 0 |

The final solution for this particular problem is: (green is the final solution)

|  |  |  |  |
| --- | --- | --- | --- |
| 4X6  m = 0 | 4X5  m = 120 | 4X8  m = 280 | 4X4  m = 360 |
|  | 6X5  m = 0 | 6X8  m = 240 | 6X4  m = 280 |
|  |  | 5X8  m = 0 | 5X4  m = 160 |
|  |  |  | 8X4  m = 0 |

Each ‘level’ is highlighted in a different color, yellow determines pink and pink determines green, and as is evident all of the matrixes are included in the path which helps show that this is a valid algorithm for multiplying n matrixes.

This algorithm can be difficult to understand at first, but the implementation is even more confusing. The implementation is adapted from [www.geeksforgeeks.org](http://www.geeksforgeeks.org) the article is unattributed but pulls its code from pseudo code on Wikipedia. This particular implementation uses something like a Floyd-Warshall search. There are two inner loops that are used to iterate through the diagonals, and each subsequent iteration, which is controlled by an outer loop that knows how long the ultimate chain will be, they work on the next set of diagonals until the chain is complete. This was tricky to implement because during initial testing there was a problem with incorrect output which was found to be: some of the multiplications being, sometimes, magnitudes larger than the added portion which resulted in a rounding error; to fix this problem long doubles were used and the order of operations was carefully considered.

A reasonable effort was made to try and get execution times of each of these heuristics and print them to the output file; however there did not seem to be a solution that would give a non-zero answer. In other words the implementation was too fast for the timing functions available even though some of them had a resolution of hundredths of a second. Attempting to push this to a reasonable limit using large numbers as dimensions and a large number of dimensions still did not yield a non-zero result so it is reasonable to assume the time difference is extremely small and also insignificant.

# Testing the Heuristics

The testing procedures for the heuristics are as follows:

* Pre-determine some set of number of dimensions then assign them values
  + The size of chain and the values assigned to dimensions were chosen to be easy to hand calculate the results of each heuristic… the large numbers were left to the computers.
  + e.x. {4,6,5,8,4}
* hand calculate the output of each heuristic
* use the program to calculate the heuristic
* correctness is established by outputs of both the hand calculation and the computer generated being equal
* This procedure was then done at least 5 times with varying sample sizes and configurations of numbers
  + Typical inputs were
    - {1,2,3,4,5,6}
      * In-order
    - {6,5,4,3,2,1}
      * Reverse order
    - {2,2,2,2,2,2,2,2,2,2,2,2}
      * Same input
    - {10,2,5,3,6,4,2,17}
      * Large numbers on the outside
    - {1,9,10,2,20,5}
      * Large numbers on the inside

The outputs to these exact inputs are given in appendix B along with annotations/observations about the general output. These inputs, including the random input in subsequent tests, provide adequate rigor to testing the functions for their correctness.

The only heuristic that this particular procedure does not work with is the random function; however since this is a derivative of the other functions the only part of it that need be tested is the random selection of numbers; since this is a well-known procedure in C++ I felt no reason to do more with it then a few stack traces to ensure that the engine was properly setup and implemented.

# Data Collection and Interpretation

At this point there are a few things that are known: the heuristics make sense in their own way and have a chance at being close to the optimal solution in some circumstances; the heuristics are implemented correctly based on our sample inputs; and finally that there are some experiments to run and numbers to analyze.

First though there must be a way that the data collection can be useful and return useful statistics that have discernable patters, well actually those will hopefully come after the collection, but first the data collection procedure and creating useful output.

The latter part of that is easy before the numbers are written to the file if they are divided by the optimal solution it can then be known how many times better or worse they are then the optimal. This gives us a basis to now compare the heuristics and their performance.

Now data collection procedure goes something like this: put some numbers in get some numbers out make conclusions about those new numbers and test the hypothesis (standard scientific method). There seems no better place than the initial test cases to begin our findings and with the new ability to get more useful output they will be run again and put into appendix C along with any other test results discussed in this section.

For the first round of test cases it seems that the least expensive heuristic performed the closest to optimal with at most a ~.008 times larger outcome. The worst by far of all of the heuristics is most expensive first this is not a surprise considering the assumtions that this heuristic makes, discussed in the previous section, its worst performance is ~12 times larger than the optimal solution. The smallest dimension first heuristic seems to do well if there is little variance in the sizes of the dimensions as it was performing at optimal level until the larger dimensions were introduced, or perhaps it is the random ness that has begun to interfere with its ability to perform at optimal. This appears to be true with the last two test that it does get worse with larger variance and more items in the dimensions, but it does not get as worse as fast as the other heuristics.

The “standard” test, which puts random numbers into a specified number of dimensions, is next. Here it is also beneficial to calculate the standard deviation, done by Welford’s method, to see the spread of data and perhaps what type of distribution it may be, the first implementation of this was not only inefficient but did not calculate the correct SD so it was rewritten using an online resource(regrettably unattributed) as a model for the current code. These tests do not seem to be optimistic in finding any heuristic that is stable and close to the optimal heuristic. Our standard deviation does point out something interesting though the data is nowhere near a uniform distribution and it also shows us that our and confirms what we already know to be true about our data… it is all over the place. These tests were odd enough, to me at least, that I ran it a second time and rearranged the results so that they can be seen side by side.

So the real question of are there any possible substitutes for the optimal algorithm, yes there are lots of possible options, but are there any good ones it does not seem to be the case. But perhaps gun to my head I would probably choose the least expensive matrix multiplication first algorithm, heuristic 4, if the optimal was not an option. I would choose this because it seems to give the smallest averages thorough the tests. And even though the worst case can be atrociously large I believe the average to be worth the ‘risk’ of the large possible outcome.

# Ending Thoughts

There are further avenues that could be traveled with this information attempting to implement these heuristics such as:

* Cost of subsequent circuitry
  + Perhaps the circuitry cost is higher than the application will allow for
    - Probably more of a business decision
* Cost of calculating heuristic vs. cost of multiplying
  + In some cases there will be no advantage to the calculating the chain to just brute force multiplying the chains together
  + Could the cost of calculating the optimal heuristic not be worth the time in comparison to choosing a faster algorithm but the actual multiplication taking longer
* Code Base
  + The entire code will be viewable on GitHub at
  + I would have liked the time to have refactored my code so that it is a bit easier to read and the execution paths make a little more sense… but it works so get it out the door.

# Appendix A – Heuristic Code

* Strategy A: Remove the largest dimension from the group

long double matrixHeuristics::largestFirst()

{

//used to hold the numbers from the vector

// had to do so so that the math worked out correctly

long double prev;

long double curr;

long double next;

//copy of dimensions

vector<int> D = dimensions;

//number of operations for multiplication

long double operations = 0;

//loops until there are only 2 dimensions

while (D.size() > 2)

{

//used to itterate through

vector<int>::iterator itt = D.begin() + 1;

//used to save the largest position/value

vector<int>::iterator largest = itt;

for (itt; itt < D.end() - 1; itt++)

{

if (\*itt > \*largest)

largest = itt;

}

//assignes items to variables

// did this to make the math work

prev = (\*(largest - 1));

curr = (\*largest);

next = (\*(largest + 1));

operations += prev \* curr \* next;

D.erase(largest);

}

return operations;

}

* Strategy B: Do the most expensive matrix multiplication first

long double matrixHeuristics::mostExpensive()

{

//used to hold the numbers from the vector

// had to do so so that the math worked out correctly

long double prev;

long double curr;

long double next;

//copy of dimensions

vector<int> D = dimensions;

//number of operations for chain

long double numOp = 0;

while (D.size() > 2)

{

//used to iterate through

vector<int>::iterator itt = D.begin() + 1;

//holds the value of the largest value location

vector<int>::iterator largest = itt;

//holds the number of multiplication operations for this iteration

long double maxMult = 0;

//temp for value of

long double value = 0;

for (itt; itt < D.end() - 1; itt++)

{

//assignes items to variables

// did this to make the math work

prev = (\*(itt - 1));

curr = (\*itt);

next = (\*(itt + 1));

value += prev \* curr \* next;

if (value > maxMult)

{

maxMult = value;

largest = itt;

}

}

numOp += maxMult;

D.erase(largest);

}

return numOp;

}

* Strategy C: Remove the smallest dimension first

long double matrixHeuristics::smallestFirst()

{

//used to hold the numbers from the vector

// had to do so so that the math worked out correctly

long double prev;

long double curr;

long double next;

//copy of dimensions

vector<int> D = dimensions;

//number of operations for multiplication

long double operations = 0;

//loops until there are only 2 dimensions

while (D.size() > 2)

{

//used to itterate through

vector<int>::iterator itt = D.begin() + 1;

//used to save the largest position/value

vector<int>::iterator smallest = itt;

for (itt; itt < D.end() - 1; itt++)

{

if (\*itt < \*smallest)

smallest = itt;

}

//assignes items to variables

// did this to make the math work

prev = (\*(smallest - 1));

curr = (\*smallest);

next = (\*(smallest + 1));

operations += prev \* curr \* next;

D.erase(smallest);

}

return operations;

}

* Strategy D: Do the least expensive matrix multiplication first

long double matrixHeuristics::leastExpensive()

{

//copy of dimensions

vector<int> D = dimensions;

//number of operations for chain

long double numOp = 0;

//used to point to the vector

// this will be the middle dimension of the most expensive operation

vector<int>::iterator smallest;

// this is used to look through the array

vector<int>::iterator itt;

//used to hold the numbers from the vector

// had to do so so that the math worked out correctly

long double prev;

long double curr;

long double next;

while (D.size() > 2)

{

//used to iterate through

itt = D.begin() + 1;

//holds the value of the largest value location

smallest = itt;

//holds the number of multiplication operations for this iteration

long double minMult = 0;

for (itt; itt < D.end() - 1; itt++)

{

//assignes items to variables

// did this to make the math work

prev = (\*(itt - 1));

curr = (\*itt);

next = (\*(itt + 1));

long double value = prev \* curr \* next;

if (value < minMult || minMult == 0)

{

minMult = value;

smallest = itt;

}

}

numOp += minMult;

D.erase(smallest);

}

return numOp;

}

* Strategy E: Random execution

long double matrixHeuristics::randomExecution()

{

//copy of dimensions

vector<int> D = dimensions;

//used to iterate through

vector<int>::iterator itt;

//index of location in vector

int item;

//number of operations

long double operations = 0;

//used to hold the numbers from the vector

// had to do so so that the math worked out correctly

long double prev;

long double curr;

long double next;

//sums up the items until there are only 2 dimensions

while (D.size() != 2)

{

random\_device rd;

mt19937 gen(rd());

//creates uniform distribution of numbers

uniform\_int\_distribution<> dist(1, D.size()-2);

//calculating the random index of item

item = dist(gen);

//sets iterator to the appropriate item

itt = D.begin() + item;

//assignes items to variables

// did this to make the math work

prev = (\*(itt - 1));

curr = (\*itt);

next = (\*(itt + 1));

//sums up the number of operations

operations += prev \* curr \* next;

//removes the item from the list

D.erase(itt);

}

return operations;

}

* Strategy F: In order of appearance

long double matrixHeuristics::inOrder()

{

//copy of dimensions

vector<int> D = dimensions;

//used to iterate through

vector<int>::iterator itt = D.begin() + 1;

//used to mark the end

vector<int>::iterator end = D.end() - 1;

//holds the number of operations

long double operations = 0;

long double prev;

long double curr;

long double next;

while (itt != end)

{

prev = (\*(itt - 1));

curr = (\*itt);

next = (\*(itt + 1));

//adding to the number of operations

operations += prev \* curr \* next;

//deletes that item from the list

itt = D.erase(itt);

end = D.end() - 1;

}

return operations;

}

* Strategy O: The “Optimal” heuristic

long double matrixHeuristics::optimal()

{

//number of matrixies

int n = dimensions.size() - 1;

//creates dynamic array

//holds the number of calculations to be done

vector<vector<long double>\*> m;

for (int i = 0; i < n; i++)

{

m.push\_back(new vector<long double>);

for (int j = 0; j < n; j++)

(\*m[i]).push\_back(0);

}

//chain is the length of the chain

for (int chain = 0; chain < n - 1; chain++)

{

//d1 is the first dimension

for (int d1 = 1; d1 < n - chain; d1++)

{

//d2 is the second dimension

int d2 = d1 + chain;

for (int d22 = d1; d22 <= d2; d22++)

{

long double left = (\*m[d1 - 1])[d22 - 1];

long double bottom = (\*m[d22])[d2];

long double dimension1 = dimensions[d1 - 1];

long double dimension2 = dimensions[d22];

long double dimension3 = dimensions[d2 + 1];

long double value = left + bottom;

value += dimension1 \* dimension2 \* dimension3;

if ((\*m[d1])[d2] == 0 || value < (\*m[d1])[d2])

(\*m[d1])[d2] = value;

}

}

}

return (\*m[1])[n - 1];

}

# Appendix B – Heuristic Test Cases

* {1,2,3,4,5,6}

Test Heuristic Results

Largest Dimension First:

240

Most Expensive First:

240

Smallest Dimension First:

68

Least Expensive First:

68

Random Execution:

Best Case: 82

Average Case: 165.726

Worst Case: 216

InOrder:

68

Optimized Raw Value:

68

* {6,5,4,3,2,1}

Test Heuristic Results

Largest Dimension First:

240

Most Expensive First:

240

Smallest Dimension First:

68

Least Expensive First:

68

Random Execution:

Best Case: 104

Average Case: 172.587

Worst Case: 216

InOrder:

240

Optimized Raw Value:

68

* {2,2,2,2,2,2,2,2,2,2,2,2}

Test Heuristic Results

Largest Dimension First:

80

Most Expensive First:

80

Smallest Dimension First:

80

Least Expensive First:

80

Random Execution:

Best Case: 80

Average Case: 80

Worst Case: 80

InOrder:

80

Optimized Raw Value:

80

* {10,2,5,3,6,4,2,17}

Test Heuristic Results

Largest Dimension First:

518

Most Expensive First:

1615

Smallest Dimension First:

2054

Least Expensive First:

510

Random Execution:

Best Case: 712

Average Case: 989.008

Worst Case: 1834

InOrder:

1090

Optimized Raw Value:

506

* {1,9,10,2,20,5}

Test Heuristic Results

Largest Dimension First:

408

Most Expensive First:

3145

Smallest Dimension First:

790

Least Expensive First:

250

Random Execution:

Best Case: 320

Average Case: 1833.21

Worst Case: 3145

InOrder:

250

Optimized Raw Value:

250

## Appendix C – Experiment Output

* {1,2,3,4,5,6}
  + Small multiplication in order small variance

Test Heuristic Results

Largest Dimension First:

3.52941

Most Expensive First:

3.52941

Smallest Dimension First:

1

Least Expensive First:

1

Random Execution:

Best Case: 1

Average Case: 3.06916

Worst Case: 3.52941

InOrder:

1

Optimized Raw Value:

68

* {6,5,4,3,2,1}
  + Small multiplication reverse order small variance

Test Heuristic Results

Largest Dimension First:

3.52941

Most Expensive First:

3.52941

Smallest Dimension First:

1

Least Expensive First:

1

Random Execution:

Best Case: 1

Average Case: 2.34873

Worst Case: 3.17647

InOrder:

3.52941

Optimized Raw Value:

68

* {2,2,2,2,2,2,2,2,2,2,2,2}
  + Large multiplication no variance

Test Heuristic Results

Largest Dimension First:

1

Most Expensive First:

1

Smallest Dimension First:

1

Least Expensive First:

1

Random Execution:

Best Case: 1

Average Case: 1

Worst Case: 1

InOrder:

1

Optimized Raw Value:

80

* {10,2,5,3,6,4,2,17}
  + Small multiplication large variance

Test Heuristic Results

Largest Dimension First:

1.02372

Most Expensive First:

3.1917

Smallest Dimension First:

4.05929

Least Expensive First:

1.00791

Random Execution:

Best Case: 1.53755

Average Case: 3.0057

Worst Case: 4.05929

InOrder:

2.15415

Optimized Raw Value:

506

* {1,9,10,2,20,5}
  + Small multiplication large variance

Test Heuristic Results

Largest Dimension First:

1.632

Most Expensive First:

12.58

Smallest Dimension First:

3.16

Least Expensive First:

1

Random Execution:

Best Case: 1.28

Average Case: 2.8596

Worst Case: 12.58

InOrder:

1

Optimized Raw Value:

250

* {5,2,4,1,3,5,2,4,6,3,2,4,5,2,3,3,2,5}
  + Large multiplication small variance

Test Heuristic Results

Largest Dimension First:

1.5285

Most Expensive First:

3.3057

Smallest Dimension First:

6.31088

Least Expensive First:

2.76166

Random Execution:

Best Case: 2.14508

Average Case: 2.61472

Worst Case: 5.16062

InOrder:

4.27461

Optimized Raw Value:

193

* {3,5,44,2,5,34,78,3,2,22,90,64,77,4,89,12,51,19}
  + Large multiplication large variance

Test Heuristic Results

Largest Dimension First:

4.60959

Most Expensive First:

25.0705

Smallest Dimension First:

44.0017

Least Expensive First:

2.02262

Random Execution:

Best Case: 1.61641

Average Case: 25.0924

Worst Case: 32.3584

InOrder:

1.50496

Optimized Raw Value:

38682

* Test For Random numbers with specified number of dimensions

|  |  |
| --- | --- |
| Run 1 | Run 2 |
| Test Heuristic Results for 10 dimensions  Largest Dimension First:  Best Case: 1  Mean Case: 15.1099  Average Case: 14.6062  Worst Case: 196.821  Standard Deviation: 38.428  Most Expensive First:  Best Case: 1.59542  Mean Case: 29.3958  Average Case: 28.416  Worst Case: 436.438  Standard Deviation: 80.1147  Smallest Dimension First:  Best Case: 1.2988  Mean Case: 36.742  Average Case: 35.5172  Worst Case: 410.959  Standard Deviation: 80.852  Least Expensive First:  Best Case: 1  Mean Case: 6.23089  Average Case: 6.02319  Worst Case: 76.8179  Standard Deviation: 13.6765  Random Execution:  Best Case: 1.04657  Standard Deviation: 19.657  Average Case: 24.6415  Standard Deviation: 61.6262  Worst Case: 500.519  Standard Deviation: 95.2051  InOrder:  Best Case: 1  Mean Case: 15.6175  Average Case: 15.0969  Worst Case: 150.339  Standard Deviation: 32.7655  Optimized Raw Value:  Best Case: 5.98892e+010  Mean Case: 8.55137e+012  Average Case: 8.26632e+012  Worst Case: 4.61381e+013  Standard Deviation: 8.82756e+01 | Test Heuristic Results for 10 dimensions  Largest Dimension First:  Best Case: 1.00701  Mean Case: 13.4381  Average Case: 12.9902  Worst Case: 182.217  Standard Deviation: 35.2045  Most Expensive First:  Best Case: 2.49178  Mean Case: 35.8414  Average Case: 34.6467  Worst Case: 487.245  Standard Deviation: 92.499  Smallest Dimension First:  Best Case: 2.71613  Mean Case: 60.1519  Average Case: 58.1468  Worst Case: 738.333  Standard Deviation: 166.205  Least Expensive First:  Best Case: 1  Mean Case: 30.3348  Average Case: 29.3236  Worst Case: 406.33  Standard Deviation: 99.516  Random Execution:  Best Case: 1.01941  Standard Deviation: 48.9528  Average Case: 28.4441  Standard Deviation: 79.1277  Worst Case: 615.512  Standard Deviation: 145.281  InOrder:  Best Case: 1  Mean Case: 34.544  Average Case: 33.3925  Worst Case: 394.839  Standard Deviation: 95.2687  Optimized Raw Value:  Best Case: 1.05406e+025  Mean Case: 2.00102e+027  Average Case: 1.93432e+027  Worst Case: 9.59914e+027  Standard Deviation: 2.106e+02 |
| Test Heuristic Results for 15 dimensions  Largest Dimension First:  Best Case: 1.15684  Mean Case: 11.631  Average Case: 11.2433  Worst Case: 71.2799  Standard Deviation: 15.2225  Most Expensive First:  Best Case: 4.30684  Mean Case: 32.4716  Average Case: 31.3892  Worst Case: 157.746  Standard Deviation: 40.41  Smallest Dimension First:  Best Case: 3.70176  Mean Case: 53.7993  Average Case: 52.006  Worst Case: 242.023  Standard Deviation: 64.214  Least Expensive First:  Best Case: 1  Mean Case: 15.095  Average Case: 14.5918  Worst Case: 111.038  Standard Deviation: 25.1639  Random Execution:  Best Case: 1.60399  Standard Deviation: 11.2119  Average Case: 25.7019  Standard Deviation: 32.9091  Worst Case: 223.395  Standard Deviation: 56.9637  InOrder:  Best Case: 1  Mean Case: 28.8552  Average Case: 27.8933  Worst Case: 152.234  Standard Deviation: 36.9398  Optimized Raw Value:  Best Case: 4.35453e+011  Mean Case: 7.04579e+012  Average Case: 6.81093e+012  Worst Case: 2.98001e+013  Standard Deviation: 8.17411e+01 | Test Heuristic Results for 15 dimensions  Largest Dimension First:  Best Case: 1.26421  Mean Case: 14.7371  Average Case: 14.2459  Worst Case: 98.8656  Standard Deviation: 26.963  Most Expensive First:  Best Case: 2.54637  Mean Case: 53.0852  Average Case: 51.3157  Worst Case: 492.942  Standard Deviation: 101.329  Smallest Dimension First:  Best Case: 3.58964  Mean Case: 106.255  Average Case: 102.713  Worst Case: 1178.44  Standard Deviation: 235.166  Least Expensive First:  Best Case: 1  Mean Case: 19.4901  Average Case: 18.8404  Worst Case: 223.036  Standard Deviation: 43.9932  Random Execution:  Best Case: 1.29773  Standard Deviation: 22.4446  Average Case: 38.074  Standard Deviation: 71.8292  Worst Case: 817.622  Standard Deviation: 174.225  InOrder:  Best Case: 1  Mean Case: 28.464  Average Case: 27.5152  Worst Case: 305.445  Standard Deviation: 59.4133  Optimized Raw Value:  Best Case: 1.54434e+025  Mean Case: 2.72799e+027  Average Case: 2.63706e+027  Worst Case: 8.83771e+027  Standard Deviation: 2.84841e+02 |
| Test Heuristic Results for 20 dimensions  Largest Dimension First:  Best Case: 1.72604  Mean Case: 23.6334  Average Case: 22.8456  Worst Case: 211.357  Standard Deviation: 43.0739  Most Expensive First:  Best Case: 4.28294  Mean Case: 68.1992  Average Case: 65.9259  Worst Case: 708.46  Standard Deviation: 136.361  Smallest Dimension First:  Best Case: 3.99272  Mean Case: 143.481  Average Case: 138.698  Worst Case: 1241.34  Standard Deviation: 307.907  Least Expensive First:  Best Case: 1.08826  Mean Case: 31.4416  Average Case: 30.3936  Worst Case: 465.155  Standard Deviation: 82.7303  Random Execution:  Best Case: 1.85627  Standard Deviation: 44.3646  Average Case: 57.2347  Standard Deviation: 112.57  Worst Case: 800.756  Standard Deviation: 183.771  InOrder:  Best Case: 1.00848  Mean Case: 67.3623  Average Case: 65.1169  Worst Case: 765.62  Standard Deviation: 148.86  Optimized Raw Value:  Best Case: 9.05641e+010  Mean Case: 9.08753e+012  Average Case: 8.78462e+012  Worst Case: 5.28742e+013  Standard Deviation: 1.12659e+01 | Test Heuristic Results for 20 dimensions  Largest Dimension First:  Best Case: 1.06267  Mean Case: 6.66755  Average Case: 6.4453  Worst Case: 22.2827  Standard Deviation: 5.36023  Most Expensive First:  Best Case: 2.58346  Mean Case: 19.239  Average Case: 18.5977  Worst Case: 86.4109  Standard Deviation: 19.9295  Smallest Dimension First:  Best Case: 3.51911  Mean Case: 33.1964  Average Case: 32.0899  Worst Case: 176.248  Standard Deviation: 36.1931  Least Expensive First:  Best Case: 1  Mean Case: 6.31958  Average Case: 6.10893  Worst Case: 29.731  Standard Deviation: 5.98589  Random Execution:  Best Case: 1.36514  Standard Deviation: 5.70326  Average Case: 14.6518  Standard Deviation: 12.7494  Worst Case: 131.748  Standard Deviation: 28.1357  InOrder:  Best Case: 1  Mean Case: 12.0295  Average Case: 11.6285  Worst Case: 76.4735  Standard Deviation: 15.3201  Optimized Raw Value:  Best Case: 1.65065e+026  Mean Case: 2.7326e+027  Average Case: 2.64152e+027  Worst Case: 1.30689e+028  Standard Deviation: 2.58592e+02 |
| Test Heuristic Results for 25 dimensions  Largest Dimension First:  Best Case: 2.40479  Mean Case: 32.2736  Average Case: 31.1978  Worst Case: 463.128  Standard Deviation: 82.9933  Most Expensive First:  Best Case: 6.08369  Mean Case: 72.3035  Average Case: 69.8934  Worst Case: 818.161  Standard Deviation: 151.656  Smallest Dimension First:  Best Case: 11.4377  Mean Case: 120.376  Average Case: 116.363  Worst Case: 1303.61  Standard Deviation: 242.746  Least Expensive First:  Best Case: 1.54385  Mean Case: 36.3542  Average Case: 35.1424  Worst Case: 692.819  Standard Deviation: 123.241  Random Execution:  Best Case: 2.26536  Standard Deviation: 73.4343  Average Case: 58.543  Standard Deviation: 141.208  Worst Case: 986.896  Standard Deviation: 182.999  InOrder:  Best Case: 1.30271  Mean Case: 44.7657  Average Case: 43.2735  Worst Case: 784.301  Standard Deviation: 139.06  Optimized Raw Value:  Best Case: 1.50214e+011  Mean Case: 8.0415e+012  Average Case: 7.77345e+012  Worst Case: 1.98948e+013  Standard Deviation: 6.4525e+01 | Test Heuristic Results for 25 dimensions  Largest Dimension First:  Best Case: 1.64649  Mean Case: 64.4279  Average Case: 62.2803  Worst Case: 711.331  Standard Deviation: 143.191  Most Expensive First:  Best Case: 2.87171  Mean Case: 175.834  Average Case: 169.972  Worst Case: 2156.67  Standard Deviation: 413.892  Smallest Dimension First:  Best Case: 3.45418  Mean Case: 325.681  Average Case: 314.825  Worst Case: 4024.96  Standard Deviation: 785.122  Least Expensive First:  Best Case: 1.06768  Mean Case: 71.3959  Average Case: 69.0161  Worst Case: 1161.35  Standard Deviation: 212.548  Random Execution:  Best Case: 1.56281  Standard Deviation: 143.765  Average Case: 147.673  Standard Deviation: 464.141  Worst Case: 2984.56  Standard Deviation: 550.666  InOrder:  Best Case: 1  Mean Case: 141.542  Average Case: 136.824  Worst Case: 2590.6  Standard Deviation: 462.844  Optimized Raw Value:  Best Case: 1.61282e+025  Mean Case: 3.25057e+027  Average Case: 3.14221e+027  Worst Case: 2.75236e+028  Standard Deviation: 5.10701e+02 |
| Test Heuristic Results for 30 dimensions  Largest Dimension First:  Best Case: 2.70068  Mean Case: 44.3705  Average Case: 42.8915  Worst Case: 466.739  Standard Deviation: 92.5515  Most Expensive First:  Best Case: 4.03309  Mean Case: 110.405  Average Case: 106.724  Worst Case: 996.678  Standard Deviation: 212.446  Smallest Dimension First:  Best Case: 6.38356  Mean Case: 171.345  Average Case: 165.634  Worst Case: 1140.87  Standard Deviation: 271.481  Least Expensive First:  Best Case: 1  Mean Case: 16.5799  Average Case: 16.0272  Worst Case: 112.34  Standard Deviation: 21.1677  Random Execution:  Best Case: 2.66828  Standard Deviation: 88.8588  Average Case: 72.7313  Standard Deviation: 142.33  Worst Case: 1010.46  Standard Deviation: 208.619  InOrder:  Best Case: 1  Mean Case: 93.0713  Average Case: 89.969  Worst Case: 1050.52  Standard Deviation: 210.93  Optimized Raw Value:  Best Case: 2.07343e+011  Mean Case: 9.10576e+012  Average Case: 8.80224e+012  Worst Case: 6.06352e+013  Standard Deviation: 1.14476e+01 | Test Heuristic Results for 30 dimensions  Largest Dimension First:  Best Case: 2.70452  Mean Case: 95.6849  Average Case: 92.4954  Worst Case: 1571.88  Standard Deviation: 308.721  Most Expensive First:  Best Case: 5.25408  Mean Case: 191.129  Average Case: 184.758  Worst Case: 3116.56  Standard Deviation: 598.463  Smallest Dimension First:  Best Case: 9.1199  Mean Case: 424.704  Average Case: 410.547  Worst Case: 8028.44  Standard Deviation: 1474.98  Least Expensive First:  Best Case: 1.08445  Mean Case: 55.7678  Average Case: 53.9089  Worst Case: 1116.34  Standard Deviation: 198.166  Random Execution:  Best Case: 2.7275  Standard Deviation: 269.374  Average Case: 155.431  Standard Deviation: 464.231  Worst Case: 4636.09  Standard Deviation: 881.899  InOrder:  Best Case: 1  Mean Case: 93.367  Average Case: 90.2548  Worst Case: 1864.35  Standard Deviation: 331.13  Optimized Raw Value:  Best Case: 1.11233e+025  Mean Case: 1.9566e+027  Average Case: 1.89138e+027  Worst Case: 8.61338e+027  Standard Deviation: 1.81852e+02 |