CS 404: Algorithms & Complexity

MCA Project Report

4/6/15

William Freeman - 12293195

BS Computer Science/Physics

Minor in Mathematics

I understand and have adhered to the rules regarding student conduct. In particular, any and all material, including algorithms and programs, have been produced and written by myself. Any outside sources that I have consulted are free, publicly available, and have been appropriately cited. I understand that a violation of the code of conduct will result in a zero (0) for this assignment, and that the situation will be discussed and forwarded to the Academic Dean of the School for any follow up action. It could result in being expelled from the university.

//William Freeman// //4/6/2015//

Algorithms and Data Structures

For this project I took a look back at my solution for Assignment 9, when we were told to construct an algorithm to optimize the computational effort for multiplying a chain of matrices. Since this project is very similar to that assignment I choose to use my algorithm design from Assignment 9 as a starting point for this more advanced problem. This was a logical choice since I could use this basic algorithm, examine the algorithm further and modify accordingly for finding the top 5 optimal expression trees. The original algorithm only found the first optimal expression tree so I would need to account for this in the new algorithm.

The original algorithm used in assignment 9 was the following:

Initial Algorithm:

BEGIN ALGORITHM

```
//Algorithm takes in an integer array of matrix dimensions that are to be multiplied. Will build the MC
//from the passed dims[] values.

BuildMCA(int dims[]){

    //Store the optimal break-points in an n-1 x n-1 matrix, where n is the size of dims.

    matrix pValues[n-1][n-1];

    //Store the optimal cost for computing the chain in an n-1 x n-1 matrix.

    matrix chain[n-1][n-1];

    //Initialize the diagonal of chain to 0, no computation required.

    for(index i = 0; i < n -1; i++){

        chain[i][i] = 0;

    }

    //Visit each upper triangle location for chain and pValues to calculate and store information.

    for(index j = 0; j < n -2; j++){

        for(index i = 0; i < (n-2 – j); i++){

            array tempValues[];

            tempValues = FindMin(i, i + j, chain, dims);

            chain[i, i + j] = tempValues[0]; //Store the min cost.

            pValues[i, i + j] = tempValues[1]; //Store the optimal breakpoint.

        }

    }
```

```
}

//FindMin() returns an array of size two that contains the information for building the MC.

//Index 0 value is the computation cost, index 1 value is the optimal break point.

//index i is the current row value of the MC to be calculated.

//index j is the current column value of the MC to be calculated.

//matrix cost, is the MatrixChain.

//int dims[] is the array of matrix dimensions to be computed.

int[] FindMin(index i, index j, matrix cost, int dims[]){

        int tempArr[j - i];

        int k = i;

        while(k < j){

                tempArr[k] = cost[i,k] + cost[k+1,j] + dims[i-1] * dims[k] * dims[j];

                k++;

        }

        int x = tempArr.minIndex();

        return toArrary(tempArr[x], x);

}
```

END ALGORITHM

My initial thought for storing the top 5 MCs involved running through all possible MC combinations and storing each separate MC that was created then examining each ones associated cost and disregard all but the top 5 computational costs. Immediately I realized that this is very brute force and would not only take a long time to compute but would also be very high in storage cost as well. I had to come up with a way to dynamically keep less MCs to save in storage cost and to find a solution that would be able to compute the top 5 MCs without having to create each and every possible combination.

I noticed that by examination of the MCA, the main diagonal is obviously always zero, and there is no decision to be made to determine the min on the second diagonal as well. I could use this information to modify the algorithm to calculate the first two diagonals and they will always be the same for all matrix chains. Upon further examination I realized that if we just take the minimum values for all other diagonals except the last one we would end up with several choices that would be near minimal cost if our chain was long enough. This would be true always because if we have minimal subtrees then adding another node would involve rearranging all the combinations of subtrees to find the minimal cost of these minimal subtree arrangements. The algorithm would then assume that there would be a matrix chain long enough that would always provide us with a way to determine the top 5 optimal

combinations in this manner. It would be very redundant to run this algorithm for a small matrix chain anyways.

I needed to come up with a way to handle the last section of the matrix chain that would iterate through all possible combination arrangements of the sub expression trees. For finding the top 5, I first considered iterating through and using the FindMin() partition of the algorithm and calling that 5 times. The algorithm would then search for the next lowest if the p-value has already been found. This would work but it is a fairly complex implementation when there is a much easier solution that is more understandable. If we were to sort the array of cost values found and take the first 5 values, we would have a much easier algorithm to understand. Also, we would not lose any time complexity and even perhaps gain some time complexity depending on what type of sort is implemented. Using the search method we would have to perform the search at least 5 times and this would take an average time complexity of n/2 each time. Therefore we would be looking at a time complexity of 5(n/2) on average. Analyzing the average case for a sort is rather difficult but we can take note that the worst case would be nlg(n) if we implement a proper sort method. I choose not to implement my own sorting algorithm and just use a programs sorting implementation. The new algorithm would look like the following:

Adding the Top5 Search:

```
BEGIN ALGORITHM

//Algorithm takes in an integer array of matrix dimensions that are to be multiplied. Will build the MC
//from the passed dims[] values.

BuildMCA(int dims[]){

    //Store the optimal break-points in an n-1 x n-1 matrix, where n is the size of dims.

    matrix pValues[n-1][n-1];

    //Store the optimal cost for computing the chain in an n-1 x n-1 matrix.

    matrix chain[n-1][n-1];

    //Initialize the diagonal of chain to 0, no computation required.

    for(index i = 0; i < n -1; i++){

        chain[i][i] = 0;

    }

    //Visit each upper triangle location for chain and pValues to calculate and store information.

    for(index j = 0; j < n - 3; j++){

        for(index i = 0; i < (n-3 – j); i++){

            array tempValues[];

            tempValues = FindMin(i, i + j, chain, dims);

            chain[i, i + j] = tempValues[0]; //Store the min cost.
```

```
                    pValues[i, i + j] = tempValues[1]; //Store the optimal breakpoint.
                }
        }
        //Top5 is an array pairing p-values and matrix computation cost.
        pair Top5[] = FindTop5(1, chain.size(), chain, dims);
}
//FindMin() returns an array of size two that contains the information for building the MC.
//Index 0 value is the computation cost, index 1 value is the optimal break point.
//index i is the current row value of the MC to be calculated.
//index j is the current column value of the MC to be calculated.
//matrix cost, is the MatrixChain.
//int dims[] is the array of matrix dimensions to be computed.
int[] FindMin(index i, index j, matrix cost, int dims[]){
        int tempArr[j - i];
        int k = i;
        while(k < j){
                tempArr[k] = cost[i,k] + cost[k+1,j] + dims[i-1] * dims[k] * dims[j];
                k++;
        }
        int x = tempArr.minIndex();
        return toArrary(tempArr[x], x);
}
//FindTop5() returns a pairing of the top 5 computation costs and the breakpoint associated with each
// breakpoint.
//The parameters are all the same as FindMin().
pair[] FindTop5(index i, index j, matrix cost, int dims[]){
        pair tempArr[j - i];
        int k = i;
        while(k < j){
```

```
            tempArr.cost[k] = cost[i,k] + cost[k+1,j] + dims[i-1] * dims[k] * dims[j];
            tempArr.break[k] = k;
            k++;
        }
        tempArr.sort();
        return tempArr[0,4]; //Return the first 5 elements.
}
```

END ALGORITHM

Now that I have the top 5 computation cost and their associated breakpoint values, I can now focus on creating the fully parenthesized expression tree. Going back to assignment 9, we were asked to implement this algorithm so I just inserted into the entire algorithm after creating 5 separate matrix chains for the associated top 5 optimal costs. Using each one of these matrices, I used my algorithm from assignment 9 to print the expression trees. The section of the algorithm to print the expression tree is as follows:

Expression Tree:

BEGIN ALGORITHM

```
//Express() stores the expression tree in a queue by iterating through the matrix chain.
//index i represents the row of the matrix, j is the column.
//result is the expression tree.
Express(matrix chain, index i, index j, queue& result){
        break = chain[i,j].pValue;
        if(break == NIL){
            result.add(chain[i,j].Name);
            return;
        }
        result.add("(");
        Express(chain, 1, break, result);
        result.add("*");
        Express(chain, break, break+1, result);
        result.add(")");
```

```
}
```

END ALGORITHM

The end result would be a fully parenthesized expression that we can use to do analyze to determine the widths and depths of the generated expression trees for the associated optimal trees discovered. Removing the functions used in the algorithm the overall algorithm would now look like the following.

Base Algorithm with Expression Build:

BEGIN ALGORITHM

```
//Algorithm takes in an integer array of matrix dimensions that are to be multiplied. Will build the MC
//from the passed dims[] values.
BuildMCA(int dims[]){
        //Store the optimal break-points in an n-1 x n-1 matrix, where n is the size of dims.
        matrix pValues[n-1][n-1];
        //Store the optimal cost for computing the chain in an n-1 x n-1 matrix.
        matrix chain[n-1][n-1];
        //Initialize the diagonal of chain to 0, no computation required.
        for(index i = 0; i < n -1; i++){
                chain[i][i] = 0;
        }
        //Visit each upper triangle location for chain and pValues to calculate and store information.
        for(index j = 0; j < n - 3; j++){
                for(index i = 0; i < (n-3 – j); i++){
                        array tempValues[];
                        tempValues = FindMin(i, i + j, chain, dims);
                        chain[i, i + j] = tempValues[0]; //Store the min cost.
                        pValues[i, i + j] = tempValues[1]; //Store the optimal breakpoint.
                }
        }
        //Top5 is an array pairing p-values and matrix computation cost.
        pair Top5[] = FindTop5(1, chain.size(), chain, dims);
        for(index i = 0; i < 5; i++){
```

```
                pValues[pValues.size(),pValues.size()] = top5[i].pValue;
                //results in a queue that holds the expression tree.
                Express(pValues, pValues.size(), pValues.size(), results);
                pair expressions.expression = results.emptyContainer();
                expressions.cost = top5[i].cost;
        }
}
```

END ALGORITHM

We would now have the expression trees paired with their associated costs. Now that we have the expressions, we can analyze the depth and width of the tree that would be formed. For the depth I noticed that if you count the number of open parentheses and their associated closed parentheses, we would obtain the depth if we just added 1 to the max of the two sides of the expression tree. Using this knowledge I was able to come up with an algorithm to analyze the depth of an expression tree. I also noticed that for the width of the tree if we counted the number of multiplication signs and added 2 to that value we would obtain the tree's width. I used these two pieces of information to create the next two portions of the algorithm.

Depth & Width Analysis:

BEGIN ALGORITHM

```
//DepthAndWidth() returns the depth of the expression tree by examining and counting the number of
// brackets. The algorithm also computes the width of the expression tree by counting the number of *
// found and adding 2 to the value.
//expression is the expression tree being examined.
//Width is the width of the tree, defaulted as zero.
int DepthAndWidth(char[] expression, int& width = 0){
        int count = 0;
        int leftCount = 0;
        int rightCount = 0;
        bool closeFound = false;
        for(index i = 0; i < expression.size(); i++){
                if(char[i] == ")“ and !closeFound){
                        closeFound = true;
                        count++;
```

```
            }else if(char[i] == "(" and closeFound){
                closeFound = false;
                leftCount = count;
                count = 0;
            }else if (char[i] == "*"){
                width++;
            }
        }
        rightCount = count;
        width += 2;
        return max(leftCount, rightCount) + 1;
}
```

END ALGORITHM

Initially I was going to separate the two algorithms from one another which would make this operation an n^2 operation but I soon realized I can combine the two and kill two birds with one stone by simply having width be a parameter argument. This would simplify the complexity to be a value of n. This would save on the overall time complexity. This would allow me to finish the entire program and begin the implementation phase. The overall complexity of this algorithm would roughly be n^3 for the chain, nlgn + c, where c is come constant for finding the top 5, and 5n for the expression analysis. There is a 5 constant in for the expression analysis because we are finding the top 5 optimal trees. The final algorithm is below:

Final Algorithm:

BEGIN ALGORITHM

```
//Algorithm takes in an integer array of matrix dimensions that are to be multiplied. Will build the MC
//from the passed dims[] values.
BuildMCA(int dims[]){
    //Store the optimal break-points in an n-1 x n-1 matrix, where n is the size of dims.
    matrix pValues[n-1][n-1];
    //Store the optimal cost for computing the chain in an n-1 x n-1 matrix.
    matrix chain[n-1][n-1];
    //Initialize the diagonal of chain to 0, no computation required.
    for(index i = 0; i < n -1; i++){
```

```
            chain[i][i] = 0;
        }
        //Visit each upper triangle location for chain and pValues to calculate and store information.
        for(index j = 0; j < n - 3; j++){
            for(index i = 0; i < (n-3 – j); i++){
                array tempValues[];
                tempValues = FindMin(i, i + j, chain, dims);
                chain[i, i + j] = tempValues[0]; //Store the min cost.
                pValues[i, i + j] = tempValues[1]; //Store the optimal breakpoint.
            }
        }
        //Top5 is an array pairing p-values and matrix computation cost.
        pair Top5[] = FindTop5(1, chain.size(), chain, dims);
        for(index i = 0; i < 5; i++){
            pValues[pValues.size(),pValues.size()] = top5[i].pValue;
            //results in a queue that holds the expression tree.
            Express(pValues, pValues.size(), pValues.size(), results);
            pair expressions.expression = results.emptyContainer();
            expressions.cost = top5[i].cost;
        }
        for(index i  = 0; i < 5; i++){
            depth = DepthAndWidth(expressions[i], width);
        }
}
```

END ALGORITHM

This would be the final algorithm that I would use in my implementation process. I decided that I would use matrices and lists for my data structures because of my programming language choice. I stayed pretty abstract in my algorithm because of this fact. I wanted the implementer to have a choice of data structures that may be used when it comes to implementation. I could have used a heap in some areas, particularly when storing the costs, but because I am not comfortable with the implementation of heaps

in R, I choose to stay away in order to complete the implementation. Also, I chose to write the expression tree out as a string instead of building a tree, I chose this because of my programming language implementation choice. I know that I should not have done that for the algorithm but I had to make a choice for the implementation and that was my deciding factor between building the actual tree and using the string method.

Program Implementation and Correctness

I chose to implement my algorithm in the language R. I used this language because of its ability to display program data structures easily in a visual manner. I also chose this language over similar languages such as Matlab or Maple because I was more familiar with R than the other languages. I also chose R over faster languages such as C or C++ because R is a more readable language and is not as syntax heavy. To implement and run my program I used RStudio v0.98.1091, which uses R v3.1.2.

The initial issue I ran into was running my first test case, I was having problems with the read in data converting my matrix into an array. After some debugging and research I found out that this was caused by the fact that the read in data was not in the proper format to be used with matrices. I found some information in the R documentation that would convert my read in data to a matrix format which allowed me to finish the initial algorithm test and insure that the algorithm was working properly.

When implementing the portion of the algorithm that computed the top 5 optimal costs. I ran into some language barriers. R does not easily support a paring map that I am aware of, so I had to use the index value as my breakpoint. I just had to create a temp array which added to the space complexity but since the array was the same size of the dimensions array there is not much to be added. I ended up having to then go back to the original array and search for the computation value to find the breakpoint value. This also led to an increase in time complexity during program implementation.

The hardest part of the algorithm implementation was the portion that would print the fully parenthesize the expression trees. Since R is a computation language, it does not work well with strings so I ran into a few issues during this implementation portion. I decided to do the expression implementation using a large list in R and appended the proper expression results as needed. I then examined the entire array as though it was a string expression. The way I treated the array was essentially like a queue by appending the proper character to the front I did not need to use a queue specifically during the implementation.

While I tried to implement my algorithm I ran into a language barrier, I was not able to complete the algorithm implementation in time due to the fact that R does not handle recursion very well. This is a program language issue and it is my fault for not allotting enough time to do research on these particular issues. I had to scrap the recursive method and come up with a new method that did not use recursion for my program implementation. It made the implementation very difficult for this part. I learned that I should allow myself more time to research the language barriers I may run into during programming implementation. I initially thought that it would be difficult to do this part of the implementation but I had no idea it would be this much trouble.

Upon my initial implementation of the expression tree algorithm, I tested it out with smaller matrix chains and it appeared to be working correctly from my reference sheet that was handed out in class. When I went to test with the larger matrices, it seemed that the implementation was not working as should. I was getting shorter expression trees than I was with the reference cases. I am not sure as to why this was happening but I had to tweak my implementation even more if I wanted to obtain a correct expression tree. I decided to move on for the time being in order to get my implementation of the width and depth calculation correct. I could easily implement that algorithm while I work out the issues involving my expression tree implementation. Upon further evaluation I noticed that the expression will produce the correct expression if we only need to walk down the edges of the chain, for expressions

that need to go inside the chain the expression is incorrect. I would then attempt to correct this issue after implementing the width and depth calculation.

Analysis of Results

The program ran to completion during each dataset test, up until it came time to express the tree. I did not have any errors when running the algorithm implementation for any of the datasets other than where I was unable to complete the algorithm implementation due to a programming language error. Most of this project involved a lot of thought. I spent most of my time sitting and thinking without actually doing any work. There were a lot of times were I thought I had the algorithm figured out and would have to start over because of some oversight on my part. Overall I feel that the algorithm does run to completion and is correct in my analysis. I am not sure about the depth or width calculation. I did not have as much time as I would have liked to test those algorithms on the larger datasets as I would have liked to of. The results did seem pretty typical leading to a very deep but narrow tree. I was also somewhat confused about the meaning of the width of the tree but since I waited too long I was not able to get a clarification on this matter. Some of the width results were very high and that did not seem correct. This may just be from my lack of understanding of the definition of tree width.

For the expression trees I had to manually examine the breakPoints and look at what might become the expression tree. I unfortunately was not able to implement this in my language because of lack of time and barriers with the programming language understanding. Next time I will allot more time to research and actual implementation of the program. I thought that once I had the algorithm it would be easy to implement using any language but I was wrong on that part. While the costs of the chains seems to make sense, with the longer chains having a higher cost, I have tested the algorithm on several smaller examples where I have produced the top minimal costs for those chains. There were no results for any of the test cases that appeared strange or out of the ordinary.

Obviously the expression algorithm that was implemented is not correct. I have tinkered with the implementation as much as I could, but does not handle this type of implementation easily. It was very difficult to reproduce these results as you can see from the implementation. I believe my algorithm is correct for reproducing the expression, I just was not able to reproduce the algorithm for the implementation. This threw off my results for the depth and width calculations because of this.

Epilogue

The main thing I learned from this experience is that if you think something will take x amount of time, you should probably double or even triple that time. Especially when developing algorithms. That being said, I would have started on this project next sooner if I had to do it all over again. I felt like I did not have enough time to plan out and fully develop and analyze an algorithm nearly as much as I wanted to. I was still in the process of thinking about whether or not having minimal subtrees will lead to the upper corner or the matrix chain being the only section that would affect the total outcome for the top 5 optimal costs. I had considered the possibility of having some non-optimal combination of subtrees that, when arranged in a specific order would lead to a top 5 optimal cost for the entire chain. I did not have enough time to fully develop and test an algorithm implementation for this theory.

With all this in mind, I do believe that having these optimal subtrees does lead to a minimal cost when adding in the final computation arrangement. If I were to try and find the suboptimal combinations of the subtrees I would have to modify the algorithm to compute all possible combinations of expression trees to truly analyze the possibility of the above theory. This would cause the algorithm to be very expensive. Not only space, but especially in time. There may be a way to reduce the cost but I was not able to explore that option due to time constraints.

Implementation Issues:

During the implementation process I had some issues with R. R is a computational language so it does not handle string manipulation very well. The hardest part of this process was figuring out how to implement the algorithm portion that would print the fully parenthesized expression trees. I had many issues with this portion of the project. It took several trial and error debugging and tweaking in order to get the implementation correct. Overall I had to scrap my initial algorithm for the expression trees and start a new with an algorithm that did not use recursion. Coming up with an iterative version was very difficult due to the fact that we had to retrain several pieces of information from the chain. In the end my implementation of the expression tree algorithm was a mess. I learned that there is sometimes a HUGE gap between algorithms and program implementation at times. Especially when you are dealing with a programming language that does not handle recursion very well. In doing this project I now understand the importance of doing good research on your program implementation before jumping right in and going past the point of no return.

Closing Thoughts:

Overall this experience was a great learning experience. I had fun tweaking and breaking down my algorithm piece by piece in order to gain an understanding of how dynamic programming can be used. While this was my first implementation of a dynamic programming algorithm, it was littered with errors and problems. I feel that as I implement more dynamic programming algorithms I will have a better understanding of how to implement them better.

Appendix A: Program Listing

```
# Finds the min computation value and returns the cost along with the
# breakpoint index.
FindMin <- function(i, j, mat, dimMat){
  temp = matrix(NA,1,j - i)
  k = i
  while(k < j){
    # Modify function based on a 1-index array.
    temp[k] = mat[i,k] + mat[k+1,j] + dimMat[i]*dimMat[k+1]*dimMat[j+1]
    k = k+1
  }
  x = which.min(temp)
  return(list(M = temp[x], p = x))
}

# Finds the top 5 computation cost and returns a list containing the top 5 costs,
# along with their breakpoints.
FindTop5 <- function(i, j, mat, dimMat){
  temp = matrix(NA,1,j - i)
  k = i
  while(k < j){
    # Modify function based on a 1-index array.
    temp[k] = mat[i,k] + mat[k+1,j] + dimMat[i]*dimMat[k+1]*dimMat[j+1]
    k = k+1
  }
  tempSort = sort(temp, decreasing=FALSE)
  # Obtain the optimal 5 computation costs.
  tempPValues = list()
  index = 1
  for(i in 1:5){
    x <- which(temp == tempSort[i])
    # Store the pValues for the top 5 costs.
    for(j in 1:length(x)){
      tempPValues[index] = x[j];
      index = index + 1
    }
  }
  pValues = unique(tempPValues)
  costs = list()
  top5pValues = list()
  for(i in 1:5){
    pValue = as.numeric(pValues[i])
    costs[i] <- temp[pValue]
    top5pValues[i] <- pValue
  }
  return(list(M=costs,p=top5pValues))
}

# Traverses down the pValue chain and returns the expression tree for the
# resulting chain. Works on some chain expressions but not all.
ExpressTree <- function(pValues, i, j){
  expIndex = 1;
  expression = list();
  initialpValue = as.numeric(pValues[i,j])
  breakValue = initialpValue
  matrixNums = list()
  numIndex = 1
  # Traverse across the chain.
```

```
  # Traverse across the chain.
  while(breakValue != 0){
    matrixNums[numIndex] = breakValue
    numIndex = numIndex + 1
    breakValue = as.numeric(pValues[1, breakValue])
    if(breakValue != 0){
      expression[expIndex] = "("
      expIndex = expIndex + 1
    }
  }
  expression[expIndex] = toString(matrixNums[length(matrixNums)])
  expIndex = expIndex + 1
  if(length(matrixNums) != 1){
    for(index in (length(matrixNums) - 1):1){
      expression[expIndex] = "*"
      expIndex = expIndex + 1
      expression[expIndex] = toString(matrixNums[index])
      expIndex = expIndex + 1
      expression[expIndex] = ")"
      expIndex = expIndex + 1
    }
  }

  expression[expIndex] = "*"
  expIndex = expIndex + 1
  matrixNums = list()
  numIndex = 1
  breakValue = initialpValue
  closeBraceCount = 0

  # Traverse down the chain.
  while(breakValue != 0){
    matrixNums[numIndex] = breakValue + 1
    numIndex = numIndex + 1
    breakValue = as.numeric(pValues[breakValue+1, j])
  }
  for(index in 1:(length(matrixNums) - 1)){
    expression[expIndex] = "("
    expIndex = expIndex + 1
    closeBraceCount = closeBraceCount + 1
    expression[expIndex] = toString(matrixNums[index])
    expIndex = expIndex + 1
    expression[expIndex] = "*"
    expIndex = expIndex + 1
  }
  expression[expIndex] = toString(matrixNums[length(matrixNums)])
  expIndex = expIndex + 1

  for(i in 1:closeBraceCount){
    expression[expIndex] = ")"
    expIndex = expIndex + 1
  }
  return(expression)
}
```

```
111
112 # Returns the depth and width of a tree expression
113 DepthAndWidth <- function(expression){
114   count = 0
115   leftCount = 0
116   rightCount = 0
117   width = 0
118   closeFound = FALSE
119   for(i in 1:length(expression)){
120     if(expression[i] == ")"){
121       closeFound = TRUE
122       count = count + 1
123     }else if(expression[i] == "(" && closeFound){
124       closeFound = FALSE
125       leftCount = count
126       count = 0
127     }else if(expression[i] == "*"){
128       width = width + 1
129     }
130   }
131   rightCount = count
132   width = width + 2
133   depth = max(rightCount, leftCount)
134   return(list(w = width, d = depth))
135 }
136
137 # Store Data paths, must change file path to where the data file is stored.
138 paths = list(F100 = "E:/Google Drive Sync Folder/Save Folder/Comp Sci 404/Project/Test Data/MatrixChainF100.csv",
139              B100 = "E:/Google Drive Sync Folder/Save Folder/Comp Sci 404/Project/Test Data/RandB100.csv",
140              C100 = "E:/Google Drive Sync Folder/Save Folder/Comp Sci 404/Project/Test Data/RandC100.csv",
141              D100 = "E:/Google Drive Sync Folder/Save Folder/Comp Sci 404/Project/Test Data/RandD100.csv",
142              S100 = "E:/Google Drive Sync Folder/Save Folder/Comp Sci 404/Project/Test Data/RandS100.csv",
143              S200 = "E:/Google Drive Sync Folder/Save Folder/Comp Sci 404/Project/Test Data/RandS200.csv",
144              S300 = "E:/Google Drive Sync Folder/Save Folder/Comp Sci 404/Project/Test Data/RandS300.csv",
145              T100 = "E:/Google Drive Sync Folder/Save Folder/Comp Sci 404/Project/Test Data/RandT100.csv",
146              T100b = "E:/Google Drive Sync Folder/Save Folder/Comp Sci 404/Project/Test Data/RandT100b.csv")
147
148 # Store dimension infomation
149 dims <- read.csv(paths$T100b, header=FALSE)
150 dims = data.matrix(dims) # Convert to proper format for data manipulation.
151
152 size = dim(dims)[2] - 1
153
154 pValues = matrix(NA,size,size) # Store P-Value information
155 chain = matrix(NA,size,size) # Store M-Value information
156
157 for(i in 1:dim(chain)[1]){
158   chain[i,i] = 0
159   pValues[i,i] = 0
160 }
161
162 # Calculate the matrix chain, except the last combination.
163 for(j in 1:(dim(chain)[1] - 2)){
164   for(i in 1:(dim(chain)[1] - j)){
165     r = FindMin(i, i+j, chain, dims)
166     m = r$M
167     chain[i,i+j] = r$M[1]
168     pValues[i,i+j] = r$p
169   }
95:28  ExpressTree
```

```
171
172 # Return the top 5 possible execution trees.
173 lastIndex = dim(chain)[1]
174 top5 = FindTop5(1, lastIndex, chain, dims)
175 topCost = as.numeric(top5$M)
176 topBreakPoints = as.numeric(top5$p)
177
178 # Evaluate the express tree
179 for(i in 1:5){
180   pValues[1,size] = topBreakPoints[i]
181   # Run into issues. I am not sure how well R handles Recursion or if I have
182   # a logical error
183   tree = ExpressTree(pValues, 1, size)
184   wd = DepthAndWidth(tree)
185   # Print the expression trees.
186   print(paste("Tree",i,":",toString(tree),sep=" "))
187   print(paste("Width",":",toString(wd$w),sep=" "))
188   print(paste("Depth",":",toString(wd$d),sep=" "))
189 }
166:12  (Top Level)
```

Appendix B: Output

F100

BreakPoint	98	28	91	1	44
Cost	243890	244115	244115	244178	244236
Width	6	5	5	4	5
Depth	2	1	1	1	1

```
[1] "Tree 1 : (, 1, *, 98, ), *, (, 99, *, (, , *, 99, ), )"
[1] "Tree 2 : (, 1, *, 28, ), *, (, 29, *, 99, )"
[1] "Tree 3 : (, 1, *, 91, ), *, (, 92, *, 99, )"
[1] "Tree 4 : 1, *, (, 2, *, 99, )"
[1] "Tree 5 : (, 1, *, 44, ), *, (, 45, *, 99, )"
```

B100

BreakPoint	98	97	96	95	94
Cost	38623662	38750354	38892945	39056965	39242081
Width	102	100	100	100	100
Depth	97	96	95	94	93

```
[1] "Tree 1 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, 1, *,
2, ), *, 3, ), *, 4, ), *, 5, ), *, 6, ), *, 7, ), *, 8, ), *, 9, ), *, 10, )
, *, 11, ), *, 12, ), *, 13, ), *, 14, ), *, 15, ), *, 16, ), *, 17, ), *, 18
, ), *, 19, ), *, 20, ), *, 21, ), *, 22, ), *, 23, ), *, 24, ), *, 25, ), *,
26, ), *, 27, ), *, 28, ), *, 29, ), *, 30, ), *, 31, ), *, 32, ), *, 33, ),
*, 34, ), *, 35, ), *, 36, ), *, 37, ), *, 38, ), *, 39, ), *, 40, ), *, 41,
), *, 42, ), *, 43, ), *, 44, ), *, 45, ), *, 46, ), *, 47, ), *, 48, ), *, 4
9, ), *, 50, ), *, 51, ), *, 52, ), *, 53, ), *, 54, ), *, 55, ), *, 56, ), *
, 57, ), *, 58, ), *, 59, ), *, 60, ), *, 61, ), *, 62, ), *, 63, ), *, 64, )
, *, 65, ), *, 66, ), *, 67, ), *, 68, ), *, 69, ), *, 70, ), *, 71, ), *, 72
, ), *, 73, ), *, 74, ), *, 75, ), *, 76, ), *, 77, ), *, 78, ), *, 79, ), *,
80, ), *, 81, ), *, 82, ), *, 83, ), *, 84, ), *, 85, ), *, 86, ), *, 87, ),
*, 88, ), *, 89, ), *, 90, ), *, 91, ), *, 92, ), *, 93, ), *, 94, ), *, 95,
), *, 96, ), *, 97, ), *, 98, ), *, (, 99, *, (, , *, 99, ), )"

[1] "Tree 2 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, 1, *, 2,
), *, 3, ), *, 4, ), *, 5, ), *, 6, ), *, 7, ), *, 8, ), *, 9, ), *, 10, ), *
, 11, ), *, 12, ), *, 13, ), *, 14, ), *, 15, ), *, 16, ), *, 17, ), *, 18, )
, *, 19, ), *, 20, ), *, 21, ), *, 22, ), *, 23, ), *, 24, ), *, 25, ), *, 26
, ), *, 27, ), *, 28, ), *, 29, ), *, 30, ), *, 31, ), *, 32, ), *, 33, ), *,
34, ), *, 35, ), *, 36, ), *, 37, ), *, 38, ), *, 39, ), *, 40, ), *, 41, ),
*, 42, ), *, 43, ), *, 44, ), *, 45, ), *, 46, ), *, 47, ), *, 48, ), *, 49,
), *, 50, ), *, 51, ), *, 52, ), *, 53, ), *, 54, ), *, 55, ), *, 56, ), *, 5
7, ), *, 58, ), *, 59, ), *, 60, ), *, 61, ), *, 62, ), *, 63, ), *, 64, ), *
```

```
, 65, ), *, 66, ), *, 67, ), *, 68, ), *, 69, ), *, 70, ), *, 71, ), *, 72, )
, *, 73, ), *, 74, ), *, 75, ), *, 76, ), *, 77, ), *, 78, ), *, 79, ), *, 80
, ), *, 81, ), *, 82, ), *, 83, ), *, 84, ), *, 85, ), *, 86, ), *, 87, ), *,
88, ), *, 89, ), *, 90, ), *, 91, ), *, 92, ), *, 93, ), *, 94, ), *, 95, ),
*, 96, ), *, 97, ), *, (, 98, *, 99, )"

[1] "Tree 3 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, 1, *, 2, ),
*, 3, ), *, 4, ), *, 5, ), *, 6, ), *, 7, ), *, 8, ), *, 9, ), *, 10, ), *, 1
1, ), *, 12, ), *, 13, ), *, 14, ), *, 15, ), *, 16, ), *, 17, ), *, 18, ), *
, 19, ), *, 20, ), *, 21, ), *, 22, ), *, 23, ), *, 24, ), *, 25, ), *, 26, )
, *, 27, ), *, 28, ), *, 29, ), *, 30, ), *, 31, ), *, 32, ), *, 33, ), *, 34
, ), *, 35, ), *, 36, ), *, 37, ), *, 38, ), *, 39, ), *, 40, ), *, 41, ), *,
42, ), *, 43, ), *, 44, ), *, 45, ), *, 46, ), *, 47, ), *, 48, ), *, 49, ),
*, 50, ), *, 51, ), *, 52, ), *, 53, ), *, 54, ), *, 55, ), *, 56, ), *, 57,
), *, 58, ), *, 59, ), *, 60, ), *, 61, ), *, 62, ), *, 63, ), *, 64, ), *, 6
5, ), *, 66, ), *, 67, ), *, 68, ), *, 69, ), *, 70, ), *, 71, ), *, 72, ), *
, 73, ), *, 74, ), *, 75, ), *, 76, ), *, 77, ), *, 78, ), *, 79, ), *, 80, )
, *, 81, ), *, 82, ), *, 83, ), *, 84, ), *, 85, ), *, 86, ), *, 87, ), *, 88
, ), *, 89, ), *, 90, ), *, 91, ), *, 92, ), *, 93, ), *, 94, ), *, 95, ), *,
96, ), *, (, 97, *, (, 98, *, 99, ), )"

[1] "Tree 4 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, 1, *, 2, ), *,
3, ), *, 4, ), *, 5, ), *, 6, ), *, 7, ), *, 8, ), *, 9, ), *, 10, ), *, 11,
), *, 12, ), *, 13, ), *, 14, ), *, 15, ), *, 16, ), *, 17, ), *, 18, ), *, 1
9, ), *, 20, ), *, 21, ), *, 22, ), *, 23, ), *, 24, ), *, 25, ), *, 26, ), *
, 27, ), *, 28, ), *, 29, ), *, 30, ), *, 31, ), *, 32, ), *, 33, ), *, 34, )
, *, 35, ), *, 36, ), *, 37, ), *, 38, ), *, 39, ), *, 40, ), *, 41, ), *, 42
, ), *, 43, ), *, 44, ), *, 45, ), *, 46, ), *, 47, ), *, 48, ), *, 49, ), *,
50, ), *, 51, ), *, 52, ), *, 53, ), *, 54, ), *, 55, ), *, 56, ), *, 57, ),
*, 58, ), *, 59, ), *, 60, ), *, 61, ), *, 62, ), *, 63, ), *, 64, ), *, 65,
), *, 66, ), *, 67, ), *, 68, ), *, 69, ), *, 70, ), *, 71, ), *, 72, ), *, 7
3, ), *, 74, ), *, 75, ), *, 76, ), *, 77, ), *, 78, ), *, 79, ), *, 80, ), *
, 81, ), *, 82, ), *, 83, ), *, 84, ), *, 85, ), *, 86, ), *, 87, ), *, 88, )
, *, 89, ), *, 90, ), *, 91, ), *, 92, ), *, 93, ), *, 94, ), *, 95, ), *, (,
96, *, (, 97, *, (, 98, *, 99, ), ), )"

[1] "Tree 5 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, 1, *, 2, ), *, 3,
), *, 4, ), *, 5, ), *, 6, ), *, 7, ), *, 8, ), *, 9, ), *, 10, ), *, 11, ),
*, 12, ), *, 13, ), *, 14, ), *, 15, ), *, 16, ), *, 17, ), *, 18, ), *, 19,
), *, 20, ), *, 21, ), *, 22, ), *, 23, ), *, 24, ), *, 25, ), *, 26, ), *, 2
7, ), *, 28, ), *, 29, ), *, 30, ), *, 31, ), *, 32, ), *, 33, ), *, 34, ), *
, 35, ), *, 36, ), *, 37, ), *, 38, ), *, 39, ), *, 40, ), *, 41, ), *, 42, )
, *, 43, ), *, 44, ), *, 45, ), *, 46, ), *, 47, ), *, 48, ), *, 49, ), *, 50
, ), *, 51, ), *, 52, ), *, 53, ), *, 54, ), *, 55, ), *, 56, ), *, 57, ), *,
58, ), *, 59, ), *, 60, ), *, 61, ), *, 62, ), *, 63, ), *, 64, ), *, 65, ),
*, 66, ), *, 67, ), *, 68, ), *, 69, ), *, 70, ), *, 71, ), *, 72, ), *, 73,
), *, 74, ), *, 75, ), *, 76, ), *, 77, ), *, 78, ), *, 79, ), *, 80, ), *, 8
1, ), *, 82, ), *, 83, ), *, 84, ), *, 85, ), *, 86, ), *, 87, ), *, 88, ), *
, 89, ), *, 90, ), *, 91, ), *, 92, ), *, 93, ), *, 94, ), *, (, 95, *, (, 96
, *, (, 97, *, (, 98, *, 99, ), ), ), ), )"
```

C100

BreakPoint	118	117	116	115	114
Cost	61060644	61189104	61336140	61498392	61678503
Width	122	120	120	120	120
Depth	117	116	114	115	113

```
[1] "Tree 1 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, 1, *, 2, ), *, 3, ), *,
4, ), *, 5, ), *, 6, ), *, 7, ), *, 8, ), *, 9, ), *, 10, ), *, 11, ), *, 12,
), *, 13, ), *, 14, ), *, 15, ), *, 16, ), *, 17, ), *, 18, ), *, 19, ), *, 2
0, ), *, 21, ), *, 22, ), *, 23, ), *, 24, ), *, 25, ), *, 26, ), *, 27, ), *
, 28, ), *, 29, ), *, 30, ), *, 31, ), *, 32, ), *, 33, ), *, 34, ), *, 35, )
, *, 36, ), *, 37, ), *, 38, ), *, 39, ), *, 40, ), *, 41, ), *, 42, ), *, 43
, ), *, 44, ), *, 45, ), *, 46, ), *, 47, ), *, 48, ), *, 49, ), *, 50, ), *,
51, ), *, 52, ), *, 53, ), *, 54, ), *, 55, ), *, 56, ), *, 57, ), *, 58, ),
*, 59, ), *, 60, ), *, 61, ), *, 62, ), *, 63, ), *, 64, ), *, 65, ), *, 66,
), *, 67, ), *, 68, ), *, 69, ), *, 70, ), *, 71, ), *, 72, ), *, 73, ), *, 7
4, ), *, 75, ), *, 76, ), *, 77, ), *, 78, ), *, 79, ), *, 80, ), *, 81, ), *
, 82, ), *, 83, ), *, 84, ), *, 85, ), *, 86, ), *, 87, ), *, 88, ), *, 89, )
, *, 90, ), *, 91, ), *, 92, ), *, 93, ), *, 94, ), *, 95, ), *, 96, ), *, 97
, ), *, 98, ), *, 99, ), *, 100, ), *, 101, ), *, 102, ), *, 103, ), *, 104,
), *, 105, ), *, 106, ), *, 107, ), *, 108, ), *, 109, ), *, 110, ), *, 111,
), *, 112, ), *, 113, ), *, 114, ), *, 115, ), *, 116, ), *, 117, ), *, 118,
), *, (, 119, *, (, , *, 119, ), )"
```

```
[1] "Tree 2 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, 1, *, 2, ), *, 3, ), *, 4,
), *, 5, ), *, 6, ), *, 7, ), *, 8, ), *, 9, ), *, 10, ), *, 11, ), *, 12, ),
*, 13, ), *, 14, ), *, 15, ), *, 16, ), *, 17, ), *, 18, ), *, 19, ), *, 20,
), *, 21, ), *, 22, ), *, 23, ), *, 24, ), *, 25, ), *, 26, ), *, 27, ), *, 2
8, ), *, 29, ), *, 30, ), *, 31, ), *, 32, ), *, 33, ), *, 34, ), *, 35, ), *
, 36, ), *, 37, ), *, 38, ), *, 39, ), *, 40, ), *, 41, ), *, 42, ), *, 43, )
, *, 44, ), *, 45, ), *, 46, ), *, 47, ), *, 48, ), *, 49, ), *, 50, ), *, 51
, ), *, 52, ), *, 53, ), *, 54, ), *, 55, ), *, 56, ), *, 57, ), *, 58, ), *,
59, ), *, 60, ), *, 61, ), *, 62, ), *, 63, ), *, 64, ), *, 65, ), *, 66, ),
*, 67, ), *, 68, ), *, 69, ), *, 70, ), *, 71, ), *, 72, ), *, 73, ), *, 74,
), *, 75, ), *, 76, ), *, 77, ), *, 78, ), *, 79, ), *, 80, ), *, 81, ), *, 8
2, ), *, 83, ), *, 84, ), *, 85, ), *, 86, ), *, 87, ), *, 88, ), *, 89, ), *
, 90, ), *, 91, ), *, 92, ), *, 93, ), *, 94, ), *, 95, ), *, 96, ), *, 97, )
, *, 98, ), *, 99, ), *, 100, ), *, 101, ), *, 102, ), *, 103, ), *, 104, ),
*, 105, ), *, 106, ), *, 107, ), *, 108, ), *, 109, ), *, 110, ), *, 111, ),
*, 112, ), *, 113, ), *, 114, ), *, 115, ), *, 116, ), *, 117, ), *, (, 118,
*, 119, )"
```

```
[1] "Tree 3 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, 1, *, 2, ), *, 3, ), *, 4, ),
*, 5, ), *, 6, ), *, 7, ), *, 8, ), *, 9, ), *, 10, ), *, 11, ), *, 12, ), *,
13, ), *, 14, ), *, 15, ), *, 16, ), *, 17, ), *, 18, ), *, 19, ), *, 20, ),
*, 21, ), *, 22, ), *, 23, ), *, 24, ), *, 25, ), *, 26, ), *, 27, ), *, 28,
), *, 29, ), *, 30, ), *, 31, ), *, 32, ), *, 33, ), *, 34, ), *, 35, ), *, 3
6, ), *, 37, ), *, 38, ), *, 39, ), *, 40, ), *, 41, ), *, 42, ), *, 43, ), *
```

```
, 44, ), *, 45, ), *, 46, ), *, 47, ), *, 48, ), *, 49, ), *, 50, ), *, 51, )
, *, 52, ), *, 53, ), *, 54, ), *, 55, ), *, 56, ), *, 57, ), *, 58, ), *, 59
, ), *, 60, ), *, 61, ), *, 62, ), *, 63, ), *, 64, ), *, 65, ), *, 66, ), *,
67, ), *, 68, ), *, 69, ), *, 70, ), *, 71, ), *, 72, ), *, 73, ), *, 74, ),
*, 75, ), *, 76, ), *, 77, ), *, 78, ), *, 79, ), *, 80, ), *, 81, ), *, 82,
), *, 83, ), *, 84, ), *, 85, ), *, 86, ), *, 87, ), *, 88, ), *, 89, ), *, 9
0, ), *, 91, ), *, 92, ), *, 93, ), *, 94, ), *, 95, ), *, 96, ), *, 97, ), *
, 98, ), *, 99, ), *, 100, ), *, 101, ), *, 102, ), *, 103, ), *, 104, ), *,
105, ), *, 106, ), *, 107, ), *, 108, ), *, 109, ), *, 110, ), *, 111, ), *,
112, ), *, 113, ), *, 114, ), *, 115, ), *, 116, ), *, (, 117, *, (, 118, *,
119, ), )"
```

```
[1] "Tree 4 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, 1, *, 2, ), *, 3, ), *, 4, ), *,
5, ), *, 6, ), *, 7, ), *, 8, ), *, 9, ), *, 10, ), *, 11, ), *, 12, ), *, 13
, ), *, 14, ), *, 15, ), *, 16, ), *, 17, ), *, 18, ), *, 19, ), *, 20, ), *,
21, ), *, 22, ), *, 23, ), *, 24, ), *, 25, ), *, 26, ), *, 27, ), *, 28, ),
*, 29, ), *, 30, ), *, 31, ), *, 32, ), *, 33, ), *, 34, ), *, 35, ), *, 36,
), *, 37, ), *, 38, ), *, 39, ), *, 40, ), *, 41, ), *, 42, ), *, 43, ), *, 4
4, ), *, 45, ), *, 46, ), *, 47, ), *, 48, ), *, 49, ), *, 50, ), *, 51, ), *
, 52, ), *, 53, ), *, 54, ), *, 55, ), *, 56, ), *, 57, ), *, 58, ), *, 59, )
, *, 60, ), *, 61, ), *, 62, ), *, 63, ), *, 64, ), *, 65, ), *, 66, ), *, 67
, ), *, 68, ), *, 69, ), *, 70, ), *, 71, ), *, 72, ), *, 73, ), *, 74, ), *,
75, ), *, 76, ), *, 77, ), *, 78, ), *, 79, ), *, 80, ), *, 81, ), *, 82, ),
*, 83, ), *, 84, ), *, 85, ), *, 86, ), *, 87, ), *, 88, ), *, 89, ), *, 90,
), *, 91, ), *, 92, ), *, 93, ), *, 94, ), *, 95, ), *, 96, ), *, 97, ), *, 9
8, ), *, 99, ), *, 100, ), *, 101, ), *, 102, ), *, 103, ), *, 104, ), *, 105
, ), *, 106, ), *, 107, ), *, 108, ), *, 109, ), *, 110, ), *, 111, ), *, 112
, ), *, 113, ), *, 114, ), *, 115, ), *, (, 116, *, (, 117, *, (, 118, *, 119
, ), ), )"
```

```
[1] "Tree 5 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, 1, *, 2, ), *, 3, ), *, 4, ), *, 5,
), *, 6, ), *, 7, ), *, 8, ), *, 9, ), *, 10, ), *, 11, ), *, 12, ), *, 13, )
, *, 14, ), *, 15, ), *, 16, ), *, 17, ), *, 18, ), *, 19, ), *, 20, ), *, 21
, ), *, 22, ), *, 23, ), *, 24, ), *, 25, ), *, 26, ), *, 27, ), *, 28, ), *,
29, ), *, 30, ), *, 31, ), *, 32, ), *, 33, ), *, 34, ), *, 35, ), *, 36, ),
*, 37, ), *, 38, ), *, 39, ), *, 40, ), *, 41, ), *, 42, ), *, 43, ), *, 44,
), *, 45, ), *, 46, ), *, 47, ), *, 48, ), *, 49, ), *, 50, ), *, 51, ), *, 5
2, ), *, 53, ), *, 54, ), *, 55, ), *, 56, ), *, 57, ), *, 58, ), *, 59, ), *
, 60, ), *, 61, ), *, 62, ), *, 63, ), *, 64, ), *, 65, ), *, 66, ), *, 67, )
, *, 68, ), *, 69, ), *, 70, ), *, 71, ), *, 72, ), *, 73, ), *, 74, ), *, 75
, ), *, 76, ), *, 77, ), *, 78, ), *, 79, ), *, 80, ), *, 81, ), *, 82, ), *,
83, ), *, 84, ), *, 85, ), *, 86, ), *, 87, ), *, 88, ), *, 89, ), *, 90, ),
*, 91, ), *, 92, ), *, 93, ), *, 94, ), *, 95, ), *, 96, ), *, 97, ), *, 98,
), *, 99, ), *, 100, ), *, 101, ), *, 102, ), *, 103, ), *, 104, ), *, 105, )
, *, 106, ), *, 107, ), *, 108, ), *, 109, ), *, 110, ), *, 111, ), *, 112, )
, *, 113, ), *, 114, ), *, (, 115, *, (, 116, *, (, 117, *, (, 118, *, 119, )
, ), ), )"
```

D100

BreakPoint	78	77	76	75	74
Cost	10204866	10232850	10264926	10302654	10346654
Width	82	80	80	80	80
Depth	77	76	75	74	73

```
[1] "Tree 1 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, 1, *, 2, ), *, 3, ), *, 4, ), *, 5, ), *, 6, ), *, 7, ), *, 8, ),
*, 9, ), *, 10, ), *, 11, ), *, 12, ), *, 13, ), *, 14, ), *, 15, ), *, 16, )
, *, 17, ), *, 18, ), *, 19, ), *, 20, ), *, 21, ), *, 22, ), *, 23, ), *, 24
, ), *, 25, ), *, 26, ), *, 27, ), *, 28, ), *, 29, ), *, 30, ), *, 31, ), *,
32, ), *, 33, ), *, 34, ), *, 35, ), *, 36, ), *, 37, ), *, 38, ), *, 39, ),
*, 40, ), *, 41, ), *, 42, ), *, 43, ), *, 44, ), *, 45, ), *, 46, ), *, 47,
), *, 48, ), *, 49, ), *, 50, ), *, 51, ), *, 52, ), *, 53, ), *, 54, ), *, 5
5, ), *, 56, ), *, 57, ), *, 58, ), *, 59, ), *, 60, ), *, 61, ), *, 62, ), *
, 63, ), *, 64, ), *, 65, ), *, 66, ), *, 67, ), *, 68, ), *, 69, ), *, 70, )
, *, 71, ), *, 72, ), *, 73, ), *, 74, ), *, 75, ), *, 76, ), *, 77, ), *, 78
, ), *, (, 79, *, (, , *, 79, ), )"
```

```
[1] "Tree 2 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, 1, *, 2, ), *, 3, ), *, 4, ), *, 5, ), *, 6, ), *, 7, ), *, 8, ), *,
9, ), *, 10, ), *, 11, ), *, 12, ), *, 13, ), *, 14, ), *, 15, ), *, 16, ), *
, 17, ), *, 18, ), *, 19, ), *, 20, ), *, 21, ), *, 22, ), *, 23, ), *, 24, )
, *, 25, ), *, 26, ), *, 27, ), *, 28, ), *, 29, ), *, 30, ), *, 31, ), *, 32
, ), *, 33, ), *, 34, ), *, 35, ), *, 36, ), *, 37, ), *, 38, ), *, 39, ), *,
40, ), *, 41, ), *, 42, ), *, 43, ), *, 44, ), *, 45, ), *, 46, ), *, 47, ),
*, 48, ), *, 49, ), *, 50, ), *, 51, ), *, 52, ), *, 53, ), *, 54, ), *, 55,
), *, 56, ), *, 57, ), *, 58, ), *, 59, ), *, 60, ), *, 61, ), *, 62, ), *, 6
3, ), *, 64, ), *, 65, ), *, 66, ), *, 67, ), *, 68, ), *, 69, ), *, 70, ), *
, 71, ), *, 72, ), *, 73, ), *, 74, ), *, 75, ), *, 76, ), *, 77, ), *, (, 78
, *, 79, )"
```

```
[1] "Tree 3 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, 1, *, 2, ), *, 3, ), *, 4, ), *, 5, ), *, 6, ), *, 7, ), *, 8, ), *, 9,
), *, 10, ), *, 11, ), *, 12, ), *, 13, ), *, 14, ), *, 15, ), *, 16, ), *, 1
7, ), *, 18, ), *, 19, ), *, 20, ), *, 21, ), *, 22, ), *, 23, ), *, 24, ), *
, 25, ), *, 26, ), *, 27, ), *, 28, ), *, 29, ), *, 30, ), *, 31, ), *, 32, )
, *, 33, ), *, 34, ), *, 35, ), *, 36, ), *, 37, ), *, 38, ), *, 39, ), *, 40
, ), *, 41, ), *, 42, ), *, 43, ), *, 44, ), *, 45, ), *, 46, ), *, 47, ), *,
48, ), *, 49, ), *, 50, ), *, 51, ), *, 52, ), *, 53, ), *, 54, ), *, 55, ),
*, 56, ), *, 57, ), *, 58, ), *, 59, ), *, 60, ), *, 61, ), *, 62, ), *, 63,
), *, 64, ), *, 65, ), *, 66, ), *, 67, ), *, 68, ), *, 69, ), *, 70, ), *, 7
1, ), *, 72, ), *, 73, ), *, 74, ), *, 75, ), *, 76, ), *, (, 77, *, (, 78, *
, 79, ), )"
```

```
[1] "Tree 4 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, 1, *, 2, ), *, 3, ), *, 4, ), *, 5, ), *, 6, ), *, 7, ), *, 8, ), *, 9, ),
*, 10, ), *, 11, ), *, 12, ), *, 13, ), *, 14, ), *, 15, ), *, 16, ), *, 17,
), *, 18, ), *, 19, ), *, 20, ), *, 21, ), *, 22, ), *, 23, ), *, 24, ), *, 2
5, ), *, 26, ), *, 27, ), *, 28, ), *, 29, ), *, 30, ), *, 31, ), *, 32, ), *
, 33, ), *, 34, ), *, 35, ), *, 36, ), *, 37, ), *, 38, ), *, 39, ), *, 40, )
, *, 41, ), *, 42, ), *, 43, ), *, 44, ), *, 45, ), *, 46, ), *, 47, ), *, 48
```

```
, ), *, 49, ), *, 50, ), *, 51, ), *, 52, ), *, 53, ), *, 54, ), *, 55, ), *,
56, ), *, 57, ), *, 58, ), *, 59, ), *, 60, ), *, 61, ), *, 62, ), *, 63, ),
*, 64, ), *, 65, ), *, 66, ), *, 67, ), *, 68, ), *, 69, ), *, 70, ), *, 71,
), *, 72, ), *, 73, ), *, 74, ), *, 75, ), *, (, 76, *, (, 77, *, (, 78, *, 7
9, ), ), )"
```

```
[1] "Tree 5 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
1, *, 2, ), *, 3, ), *, 4, ), *, 5, ), *, 6, ), *, 7, ), *, 8, ), *, 9, ), *,
10, ), *, 11, ), *, 12, ), *, 13, ), *, 14, ), *, 15, ), *, 16, ), *, 17, ),
*, 18, ), *, 19, ), *, 20, ), *, 21, ), *, 22, ), *, 23, ), *, 24, ), *, 25,
), *, 26, ), *, 27, ), *, 28, ), *, 29, ), *, 30, ), *, 31, ), *, 32, ), *, 3
3, ), *, 34, ), *, 35, ), *, 36, ), *, 37, ), *, 38, ), *, 39, ), *, 40, ), *
, 41, ), *, 42, ), *, 43, ), *, 44, ), *, 45, ), *, 46, ), *, 47, ), *, 48, )
, *, 49, ), *, 50, ), *, 51, ), *, 52, ), *, 53, ), *, 54, ), *, 55, ), *, 56
, ), *, 57, ), *, 58, ), *, 59, ), *, 60, ), *, 61, ), *, 62, ), *, 63, ), *,
64, ), *, 65, ), *, 66, ), *, 67, ), *, 68, ), *, 69, ), *, 70, ), *, 71, ),
*, 72, ), *, 73, ), *, 74, ), *, (, 75, *, (, 76, *, (, 77, *, (, 78, *, 79,
), ), ), )"
```

S100

BreakPoint	77	96	1	95	2
Cost	1289745	1301548	1302813	1303900	1304177
Width	5	6	5	7	6
Depth	1	2	2	2	2

```
[1] "Tree 1 : (, 1, *, 77, ), *, (, 78, *, 99, )"
```

```
[1] "Tree 2 : (, (, 1, *, 77, ), *, 96, ), *, (, 97, *, 99, )"
```

```
[1] "Tree 3 : 1, *, (, 2, *, (, 78, *, 99, ), )"
```

```
[1] "Tree 4 : (, (, 1, *, 77, ), *, 95, ), *, (, 96, *, (, 97, *, 99, ), )"
```

```
[1] "Tree 5 : (, 1, *, 2, ), *, (, 3, *, (, 78, *, 99, ), )"
```

S200

BreakPoint	193	36	43	46	127
Cost	2595394	2596362	2596483	2596483	2596483
Width	5	5	5	5	5
Depth	1	1	1	1	1

```
[1] "Tree 1 : (, 1, *, 193, ), *, (, 194, *, 199, )"
```

```
[1] "Tree 2 : (, 1, *, 36, ), *, (, 37, *, 199, )"
```

```
[1] "Tree 3 : (, 1, *, 43, ), *, (, 44, *, 199, )"
```

```
[1] "Tree 4 : (, 1, *, 46, ), *, (, 47, *, 199, )"
```

```
[1] "Tree 5 : (, 1, *, 127, ), *, (, 128, *, 199, )"
```

S300

BreakPoint	298	256	45	60	82
Cost	4170096	4171669	4171911	4171911	4171911
Width	50	24	23	24	24
Depth	45	20	19	20	20

```
[1] "Tree 1 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (,
(, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, 1, *,
2, ), *, 3, ), *, 4, ), *, 5, ), *, 6, ), *, 7, ), *, 8, ), *, 9, ), *, 10, )
, *, 11, ), *, 12, ), *, 13, ), *, 14, ), *, 15, ), *, 16, ), *, 17, ), *, 18
, ), *, 22, ), *, 45, ), *, 256, ), *, 273, ), *, 274, ), *, 275, ), *, 276,
), *, 277, ), *, 278, ), *, 279, ), *, 281, ), *, 282, ), *, 283, ), *, 284,
), *, 285, ), *, 286, ), *, 287, ), *, 288, ), *, 289, ), *, 290, ), *, 291,
), *, 292, ), *, 293, ), *, 294, ), *, 295, ), *, 296, ), *, 297, ), *, 298,
), *, (, 299, *, (, , *, 299, ), )"
```

```
[1] "Tree 2 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, 1,
*, 2, ), *, 3, ), *, 4, ), *, 5, ), *, 6, ), *, 7, ), *, 8, ), *, 9, ), *, 10
, ), *, 11, ), *, 12, ), *, 13, ), *, 14, ), *, 15, ), *, 16, ), *, 17, ), *,
18, ), *, 22, ), *, 45, ), *, 256, ), *, (, 257, *, 299, )"
```

```
[1] "Tree 3 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, 1, *,
2, ), *, 3, ), *, 4, ), *, 5, ), *, 6, ), *, 7, ), *, 8, ), *, 9, ), *, 10, )
, *, 11, ), *, 12, ), *, 13, ), *, 14, ), *, 15, ), *, 16, ), *, 17, ), *, 18
, ), *, 22, ), *, 45, ), *, (, 46, *, 299, )"
```

```
[1] "Tree 4 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, 1,
*, 2, ), *, 3, ), *, 4, ), *, 5, ), *, 6, ), *, 7, ), *, 8, ), *, 9, ), *, 10
, ), *, 11, ), *, 12, ), *, 13, ), *, 14, ), *, 15, ), *, 16, ), *, 17, ), *,
18, ), *, 22, ), *, 45, ), *, 60, ), *, (, 61, *, 299, )"
```

```
[1] "Tree 5 : (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, (, 1,
*, 2, ), *, 3, ), *, 4, ), *, 5, ), *, 6, ), *, 7, ), *, 8, ), *, 9, ), *, 10
, ), *, 11, ), *, 12, ), *, 13, ), *, 14, ), *, 15, ), *, 16, ), *, 17, ), *,
18, ), *, 22, ), *, 45, ), *, 82, ), *, (, 83, *, 299, )"
```

T100

BreakPoint	1	97	98	5	6
Cost	3267822	3267822	3270338	3270705	3270705
Width	4	5	6	5	5
Depth	1	1	2	1	1

```
[1] "Tree 1 : 1, *, (, 2, *, 99, )"
```

```
[1] "Tree 2 : (, 1, *, 97, ), *, (, 98, *, 99, )"
```

```
[1] "Tree 3 : (, 1, *, 98, ), *, (, 99, *, (, , *, 99, ), )"
```

```
[1] "Tree 4 : (, 1, *, 5, ), *, (, 6, *, 99, )"
```

```
[1] "Tree 5 : (, 1, *, 6, ), *, (, 7, *, 99, )"
```

T100b

BreakPoint	1	6	8	9	10
Cost	3683559	3684648	3684648	3684648	3684648
Width	39	39	38	37	36
Depth	36	35	34	33	32


```
[1] "Tree 1 : 1, *, (, 2, *, (, 7, *, (, 9, *, (, 10, *, (, 11, *, (, 15, *,
(, 25, *, (, 26, *, (, 28, *, (, 29, *, (, 41, *, (, 43, *, (, 44, *, (, 47,
*, (, 51, *, (, 53, *, (, 54, *, (, 55, *, (, 58, *, (, 62, *, (, 66, *, (, 6
9, *, (, 73, *, (, 75, *, (, 77, *, (, 78, *, (, 79, *, (, 80, *, (, 82, *, (
, 83, *, (, 85, *, (, 90, *, (, 92, *, (, 95, *, (, 97, *, (, 98, *, 99, ), )
, ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ),
), ), ), ), ), ), ), ), )"

[1] "Tree 2 : (, 1, *, 6, ), *, (, 7, *, (, 9, *, (, 10, *, (, 11, *, (, 15,
*, (, 25, *, (, 26, *, (, 28, *, (, 29, *, (, 41, *, (, 43, *, (, 44, *, (, 4
7, *, (, 51, *, (, 53, *, (, 54, *, (, 55, *, (, 58, *, (, 62, *, (, 66, *, (
, 69, *, (, 73, *, (, 75, *, (, 77, *, (, 78, *, (, 79, *, (, 80, *, (, 82, *
, (, 83, *, (, 85, *, (, 90, *, (, 92, *, (, 95, *, (, 97, *, (, 98, *, 99, )
, ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ),
), ), ), ), ), ), ), ), )"

[1] "Tree 3 : (, 1, *, 8, ), *, (, 9, *, (, 10, *, (, 11, *, (, 15, *, (, 25,
*, (, 26, *, (, 28, *, (, 29, *, (, 41, *, (, 43, *, (, 44, *, (, 47, *, (, 5
1, *, (, 53, *, (, 54, *, (, 55, *, (, 58, *, (, 62, *, (, 66, *, (, 69, *, (
, 73, *, (, 75, *, (, 77, *, (, 78, *, (, 79, *, (, 80, *, (, 82, *, (, 83, *
, (, 85, *, (, 90, *, (, 92, *, (, 95, *, (, 97, *, (, 98, *, 99, ), ), ), ),
), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ),
), ), ), )"

[1] "Tree 4 : (, 1, *, 9, ), *, (, 10, *, (, 11, *, (, 15, *, (, 25, *, (, 26
, *, (, 28, *, (, 29, *, (, 41, *, (, 43, *, (, 44, *, (, 47, *, (, 51, *, (,
53, *, (, 54, *, (, 55, *, (, 58, *, (, 62, *, (, 66, *, (, 69, *, (, 73, *,
(, 75, *, (, 77, *, (, 78, *, (, 79, *, (, 80, *, (, 82, *, (, 83, *, (, 85,
*, (, 90, *, (, 92, *, (, 95, *, (, 97, *, (, 98, *, 99, ), ), ), ), ), ), ),
), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), )"

[1] "Tree 5 : (, 1, *, 10, ), *, (, 11, *, (, 15, *, (, 25, *, (, 26, *, (, 2
8, *, (, 29, *, (, 41, *, (, 43, *, (, 44, *, (, 47, *, (, 51, *, (, 53, *, (
, 54, *, (, 55, *, (, 58, *, (, 62, *, (, 66, *, (, 69, *, (, 73, *, (, 75, *
, (, 77, *, (, 78, *, (, 79, *, (, 80, *, (, 82, *, (, 83, *, (, 85, *, (, 90
, *, (, 92, *, (, 95, *, (, 97, *, (, 98, *, 99, ), ), ), ), ), ), ), ), ), )
, ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), ), )"
```

References

https://stat.ethz.ch/R-manual/R-devel/library/base/html/paste.html - Used for string concentration in R. Was used to reprint the fully parenthesized expression tree.

https://stat.ethz.ch/R-manual/R-devel/library/base/html/data.matrix.html - Used to manipulate the read in data into a matrix form.

https://stat.ethz.ch/R-manual/R-devel/library/base/html/sort.html - Used for sorting of arrays.

https://stat.ethz.ch/R-manual/R-devel/library/base/html/unique.html - Used to create unique sets for the pValues found.