Project Pseudocode and Complexity Analysis

By Brett Huffman
Oct 16, 2021

Overview

For the Gifting project, I have been going through many iterations of different algorithms trying to decide exactly how I was going to attack the problem. As of now, I have decided on a course of action that I think will give me slightly advantageous results.

My goal has been to try to, as quickly as possible, get to the point that I can drastically prune branches. So, in my current application, I am pruning at the same time I am building my search trees. This gives me the advantage of not having to go through the trees multiple times to build and prune.

I am also starting out with a very wide but short tree with just the number of children high, each having exactly the MIN(MedGifts, LrgGifts) number of medium and large gifts. I prune that early tree to get rid of bad combinations early. Next, I build a second tree with only those branches not pruned from the first phase with every combination of left-over gifts. So, if there were 3 more large gifts, I iterate through each branch combination of Phase I with every combination of those three gifts; all with the goal of finding the lowest sum of retail prices

My hope is that this two-phase approach will yield a faster result than just a single scan through one large tree.

Pseudocode and Analysis

There are really three portions of code that must be analyzed: Setup of Phase I, Phase I, and Phase II.

Phase I Setup

In setup for this phase, we will be creating the structures and lookup table that will support the analysis of Phase I.

Pseudocode	Complexity Analysis	Space Complexity Analysis

`// Initialization`
`Vector Children`
`Vector MediumGifts LargeGifts`
`Vector<Vector> ChildBranches`
`Int** childGiftLogicTable // Fast lookup table`
	All O(1)	

Float overallAvgPrice
// Counters
nChildCount = 0
nGiftCount = 0
 	
LoadArrayFromFile(inputFile)
  Foreach Child => Add To Children
  Foreach medGift => Add to MediumGift
  Foreach lrgGift => Add to LargeGifts
 	
If lrgGift.count < child.count ||
medGift.count < child.count
  exit
// From running totals
overallAvgPrice =
      totalPrices/totalChildren
	
	O(C)
	O(MG)
	O(LG)
	
	O(1)
	
	O(1)
	
	Overall: O(C + MG + LG)
		
		O(C)
		O(G)
		
		Overall: O(C+G)

// Create the Logic Table
Init Table to Children.Count *
(LgGift+MedGift).Count
 	
Foreach child c =>
  Foreach totalGifts g =>
    If c.age within gift.age ranges
      Table[g,c] = 1
    Else
      Table[g,c] = 0
	
	O(1)
	
	O(C * totalGifts)
	
	Overall: O(C * totalGifts)
		
		O(C * totalGifts)
		
		Overall: O(C * totalGifts)

Phase I - Generate and Calculate Initial Tree

This phase will be dedicated to actually doing the initial analysis on the first tree. This tree is only children.count deep and children.count$^{MIN(MedGifts, LrgGifts)}$ wide. So, it is very wide and not very deep.

As we build, we will be pruning branches – essentially not adding them to the final Vector – if any gifts in that branch do not meet the age requirements of the respective child.

Pseudocode	Time Complexity Analysis	Space Complexity Analysis

<table>
<tr>
<td>// Initialization
medGiftStart = 0
lrgGiftStart = 0
Vector branch

foreach branch b in 0 to children.size =>
  branch[b].child = children[b]
  foreach g in 0 to children.size
    branch[b].giftMed = (medGiftStart + g)
         % medGift.count
    branch[b].giftLrg = (lrgGiftStart + g)
         % lrgGift.count

    // At this point, we have our branches
    // Next we will mark them for pruning
    If branch[b].giftMed.age not in
       Branch[b].child age range
          Branch[b].prune = true
     Else
          Branch[b].prune = false

    // Finally, for each branch, if not
    // marked to prune, calculate price
    // diff and enter into
    // ChildBranches Vector
    If not Brach[b].prune
      Brach[b].branchDiffTotal =
         BranchRunningSum / children.size
      ChildBranches.add branch</td>
<td>All O(1)

O(C)

O(C)

Overall:
(Children^{Gifts})

O(1)

O(1)

O(1)

Overall:
O(Children^{Gifts})</td>
<td>No Change in size during this operation</td>
</tr>
</table>

Phase II - Generate and Calculate Final Tree and Results

This phase will also iterate through each left-over branch combination of Phase I with every combination of left-over gifts – those gifts that did not match up exactly with children (ie. When there were more gifts than children.)

As with Phase I, we will be pruning branches – essentially not adding them to the final Vector – if any gifts in that branch do not meet the age requirements of the respective child. We will also be keeping a running total for each child and branch so we may determine a winner.

Finally, if there are no left-over gifts, this phase will be omitted.

Pseudocode	Complexity Analysis	Space Complexity Analysis

// Initialization
GiftStart = 0
Vector branch
 	
If giftMed.count > children.count ||
giftLrg.count > children.count
 	
foreach branch b in branches =>
foreach c in children =>
foreach g in giftsLeftOver =>
if gift.age in c age range
create new branch b1
deep copy b to b1
b1.c.price = g.price
 	
// At this point, we have our branches
// Next we will mark them for pruning
If branch[b].giftMed.age not in
Branch[b].child age range
Branch[b].prune = true
Else
Branch[b].prune = false
 	
// Finally, for each branch, if not
// marked to prune, calculate price
// diff and enter into
// ChildBranches Vector
If not Brach[b].prune
Brach[b].branchDiffTotal =
BranchRunningSum / children.size
ChildBranches.add branch
	All O(1)
	
	O(1)
	
	O(C)
	O(C)
	O($\text{Children}^{\text{Gifts}}$)
	
	O(1)
	Overall:
	O($\text{Children}^{\text{Gifts}}$)
	
	O(1)
	
	O(1)
	
	O(1)
	
	Overall:
	O($\text{Children}^{\text{Gifts}}$)
		O($\text{Children}^{\text{Gifts}}$)
		
		Overall:
		O($\text{Children}^{\text{Gifts}}$)

// Iterate through all existing branches
// to find the lowest Sum Differential
Branch winner = branches[0]		

`Foreach branch b in branches[:1]`
`  If b < winner`
`    winner = b`
 	
`print winner`
	O(b) // pruned branches	

The overall time complexity of this operation should be $O(\text{Children}^{\text{Gifts}})$.