Project 2: Dynamic programming

COT 4400, Fall 2016

Authors: Aaron Daniel, John Astfanous, and Walter Soulliere

1. Intro:

In this project, we solve a dynamic programing problem and write a program to solve it. The problem is to find partitions of an array that minimize the wasted space in each partition. To do this we use an inequality score that is calculated by adding all the squares of the unused capacity. The formula for calculating the inequality score is a as follows:

$$\sum_{i=1}^{k} (t - p_i)^2$$

The maximum size of each partition is t, the partitions are p and there are k partitions. The optimal partitioning for each array will have the lowest inequality score.

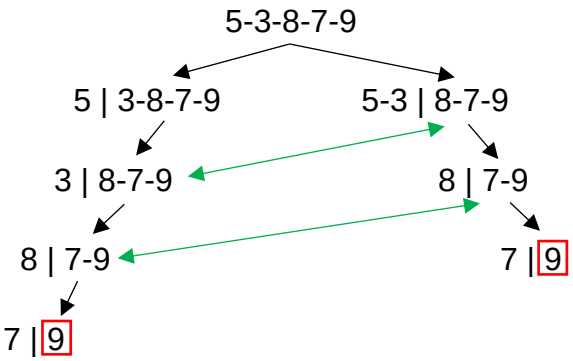
2. Method:

How the problem is broken up:

The problem is broken up into all the possible candidates for the first partition and recursively calls the function on the remaining array. The returned value is a data structure containing a partitioning list and its inequality score. The partitioning list is a list consisting of the number of values in each partition. An example of a partitioning list and inequality score is shown for the following partitions and t = 10.

5-1-2 | 5-4 | 9 | 7 = { partitioning = [3, 2, 1, 1] , score = 15 }

You can see how these values are calculated in the pseudocode below. An example of how its broken up is shown below. Given the list [5,3,8,7,9] and t = 10 it will be broken up as seen below.



: Base cases

: Recursive Calls

: Memoizable Values

Recurrence used to model the problem using dynamic programming:

A recursive function is used to find the minimum possible score for possible partitions. The score is minimum of the possible first Partitions + The remaining arrays minimum possible score. The model is shown below for the MPS(minimum possible score) function:

MPS(n, t) = $\min_{x=0}^{\text{sum(n[0:x])} \le t}$(Score(n[0:x]) + MPS(n[x:n.size], t))

Score() = $\sum_{i=0}^{x}(t - n[i])^2$

n[0:x] = possible Partitions with a sum less than or equal to t

n[x:n.size] = the remainder of the array.

Base cases:

The base cases (highlighted in red above) is when the partitions can be created with a sum less then t and no remainder. When this is met, it will return the start of the partitioning list (the size of the base case partition) and the inequality score of that partition.

Pseudocode:

The following Pseudocode describes how the recursive solution functions, how Score is defined and how base cases are handled. It is a recursive function that uses memorization to memoize the minimum possible score for a case that has already been calculated. It returns the minimum possible score and an append partitioning list of the sizes of each partition.

```
The following Algorithms utilizes the Score object
class Score
{
	score = 0
	path<list>
}
```

```
Algorithm: RecursivePartition
Input: file - A file in the correct format containing a series of problems

Description: Returns the minimum Inequality Value of the array based on a maximum
capacity value

instances = first line of file

for 1...instances
	problem = next line of file

	n = problem[1]
	t = problem[2]
	Map<Scores> map = new Map<Scores>
	Generate array A based on the next line

	RecursivePartition(A, start, n, t)
end
```

```
Algorithm: RecursivePartitionSolver
Input: A - an array of integers
	start - the starting index to work from
	n - the number of elements in array A
	t - the maximum capacity of a partition

Description: Find the optimal inequality value of array A through recursion

Score cur= new Score

if(map[start] != NIL) return map[start]

if(sum(A[start:n]) <= t)
	cur.score +=(t - sum(A[start:n]))^2
	cur.partitioning append size_of_partition + 1

size_of_partition = 0
map_value = 0
min = infinity

while size_of_partition < (n - start) and map_value < t
	if(map[start + size_of_partition + 1] = NIL)
		map[start + size_of_partition + 1] = RecursivePartition(A,
                                                start + size_of_partition + 1, n, t)
	if map_value <= t
		cur = map[start + size_of_partition + 1]

		cur.score += (t - sum(A[start:start + size_of_partition + 1]))^2
		cur.partitioning append size_of_partition + 1

		if(cur.score < min) min = cur
	end

	size_of_partition += 1
```

```
end

map[start] = min
return min
```

Iterative version:

If we want to be able to partition the array Iteratively we must start from the base cases and work our way to the solution. A very simple way for doing this is starting from the right and making a partition every time the partition exceeds t. This would give us an adequate answer but not the optimal solution. We can take this basic concept and improve it.

Modified and extended iterative algorithm to identify the optimal partition:

The iterative algorithm can then be modified to find the optimal solution. Rather than iterating through the array partitioning based on t to find a value the algorithm can be altered to instead consider all possible options for partitioning an element in the array. This can be combined with memoization to find the optimal value without severely impacting the time complexity of the algorithm.

As for an algorithm that finds the optimal solution, we can alter the Algorithm to begin iterating through the array at the last element, checking to determine if any of the previous values can be added into the same partition with the current element. The analysis of this would then be stored into a map so they can be used further along during the iteration. When moving forward to the previous element in the array the minimum value for all partitions including later elements would already be calculated. Thus, the algorithm would only need to determine the results of the element in a partition by itself, or with other previous values in the array. When the algorithm iterates through to the first element in the algorithm will end and the minimum possible value will have been stored within the map. This is because the map will have stored the optimal partitioning at for every index after the iterator has passed it. For an example of how this algorithm finds the answer, there is an example at the end of the report.

3. Analysis:

Time and space complexity of the improved iterative algorithm:

The time complexity of the IterativePartitionSolver is largely dependent on the input. The inner most while loop will continue to run if the maximum capacity of the partition has not yet been met, or until the entirety of the array is in a single partition. In the worst-case scenario, this can iterate n times. The inner workings of the while loop are mostly static with a time complexity of Θ(1). The only exception to this is the call to the getInequality function, which is called inside of an if statement as well as the else statement which ensure it will always run with every iteration of the while loop. The getInequality function iterates pending on the number of elements in the partition – which means at most can iterate up to n times. As all the actions performed by the

function require a complexity of Θ(1) this will always have an overall complexity of n iterations of Θ(1) which is equivalent to a time complexity of Θ(n).

This brings the complexity of the while loop to be at worst n iterations of Θ(n) for a worst-case complexity of O(n2). As we know the complexity of the while loop – this allows us to determine the complexity of the main for loop of the Algorithm. The for loop iterates n times regardless of input, with the contained while loop accounting for almost the entirety of the complexity resulting in a worst-case complexity of O(n3).

Taking into account added complexity of initializing the map this brings the algorithm overall to have a complexity of O(n3 + n). The best-case scenario is significantly reduced complexity – and can be attained only when each number in the array can be placed in its own partition. When this happens the complexity of the getInequality function is a much improved Ω(1). In addition the containing while loop will only iterate once which leaves the best case scenario for the algorithm to be Ω(n + n). Space Complexity on the other hand is increased as to be expected with the use of memoization. The map in the algorithm contributes to an additional array of size n which is used to prevent repeating the same sub-problems more than once. This adds an additional space complexity to the algorithm of Θ(n). However, at the expense of added space complexity this drastically reduces the amount of time complexity taken by the algorithm.

space complexity improvement relative to the memorized algorithm:

In this case the space complexity introduced by memoization can't be offset. This is due to the process the algorithm takes when solving as well as the input being resolved. As the Algorithm iterates through the input array it's constantly updating the map with additional solution possibilities. As the map is filled what may appear to be an optimal solution may be incorrect based on some of the later elements to be analyzed. For this reason, the map needs to contain every possible solution to truly find the optimal combination of partitions. While this does have a negative impact on space complexity that can't be reduced – the improvements to time complexity are significantly more impactful.

Iteration algorithm example:

The arrow is where the Iterator is and the bolded numbers is the partition it's looking at

0	1	2	3	4	5	6
5	2	3	4	10	5	**3**

index	Score	Partitions
0		
1		
2		
3		
4		
5		
6	49	1

0	1	2	3	4	5	6
5	2	3	4	10	**5**	**3**

index	Score	Partitions
0		
1		
2		
3		
4		
5	4	2
6	49	1

0	1	2	3	4	5	6
5	2	3	4	10	**5**	3

index	Score	Partitions
0		
1		
2		
3		
4		
5	4	2
6	49	1

0	1	2	3	4	5	6
5	2	3	4	**10**	5	3

index	Score	Partitions
0		
1		
2		
3		
4	4	1,2
5	4	2
6	49	1

0	1	2	3	4	5	6
5	2	3	**4**	10	5	3

index	Score	Partitions
0		
1		
2		
3	40	1,1,2
4	4	1,2
5	4	2
6	49	1

0	1	2	3	4	5	6
5	2	**3**	**4**	10	5	3

index	Score	Partitions
0		
1		
2	13	2,1,2
3	40	1,1,2
4	4	1,2
5	4	2
6	49	1

0	1	2	3	4	5	6
5	**2**	**3**	**4**	10	5	3

index	Score	Partitions
0		
1	4	3,1,2
2	13	2,1,2
3	40	1,1,2
4	4	1,2
5	4	2
6	49	1

0	1	2	3	4	5	6
5	2	**3**	4	10	5	3

index	Score	Partitions
0		
1	4	3,1,2
2	13	2,1,2
3	40	1,1,2
4	4	1,2
5	4	2
6	49	1

0	1	2	3	4	5	6
5	**2**	**3**	4	10	5	3

index	Score	Partitions
0		
1	4	3,1,2
2	13	2,1,2
3	40	1,1,2
4	4	1,2
5	4	2
6	49	1

0	1	2	3	4	5	6
5	**2**	3	4	10	5	3

index	Score	Partitions
0		
1	4	3,1,2
2	13	2,1,2
3	40	1,1,2
4	4	1,2
5	4	2
6	49	1

0	1	2	3	4	5	6
5	**2**	3	4	10	5	3

index	Score	Partitions
0	22	2,2,1,2
1	4	3,1,2
2	13	2,1,2
3	40	1,1,2
4	4	1,2
5	4	2
6	49	1

0	1	2	3	4	5	6
5	2	3	4	10	5	3

index	Score	Partitions
0	22	2,2,1,2
1	4	3,1,2
2	13	2,1,2
3	40	1,1,2
4	4	1,2
5	4	2
6	49	1

The solution is stored in index 0 in the map after the iteration completes.