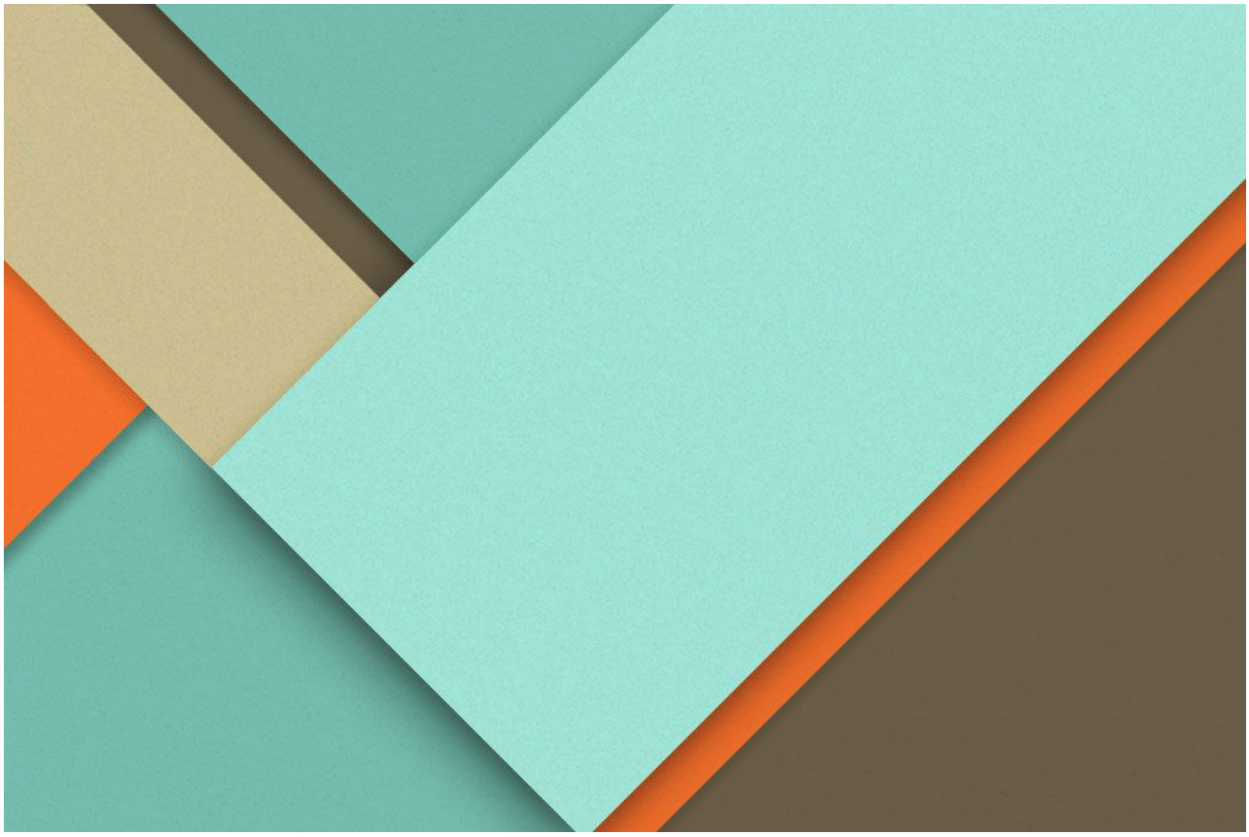

Dynamic Programming

Analysis of Algorithms Project 2
14 November 2016

Matthew Kramer

Lott Lalime

Junior Recinos

Table of Contents

Overview

"For this project, you will solve a variant of the integer partition problem described in Section 8.5 of the required text. For this problem, you will be given an array of positive integers that you need to divide as evenly as possible without rearranging the array. Unlike the problem described in the textbook, you are given a limit on how large these partitions can be, rather than how many you can use. Specifically, you should divide the integers to minimize the 'inequality score,' calculated as the sum of the squares of the unused capacity; i.e.,

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

where t is the maximum size of a partition and $p_1, p_2, \ldots, p_k$ are the sums of the values in each of the k partitions." [1]

Divide and Conquer

To divide this problem into smaller subproblems, we must first be sure that each subproblem we are solving is uniquely defined, allowing us to store the value for future lookup. With uniquely defined subproblems (in our case, array partitions), we can store these values in a two-dimensional array for lookup if we come across the same problem again. These two dimensions can be the length of the subarray and the location of an index that divides this subarray into a partition that is left alone and a partition that is yet to be minimized for total inequality score. In this way, we can ensure that we cover all subarrays of the problem and that each is uniquely identified within the data structure for lookup.

Defining a Recurrence

bestPartition(array, t, index, n) = 0, **if index > n**
infinity, **if sum > t**
bestPartition(array[i + 1 to n], t, index + 1, n)
if sum ≤ t and table[index][1 to n] are infinity
minimum of table[index][1 to n]
if sum ≤ t and table[index][1 to n] are not infinity

The base case of this recurrence is the last of the preceding piecewise function, when we encounter a case where the sum is below the maximum partition size and where the values memoized for that index are all valid (finite). The first two cases - when the index exceeds the length of the partition being examined and when the current sum exceeds the maximum partition size - could also be considered base cases, as their values can

[1] Project 2: Dynamic Programming, Dr. William Hendrix. 2016. University of South Florida.

immediately be calculated upon execution.

Developing a Solution

1. Memoized

```
Input: data: an array of integers (could be sub-array),
       n: size of data, t: maximum size of a partition,
       index: index from which the algorithm starts
Output: minimum inequality score
Algorithm: MemoPartition
1  int newindex = index
2  int sum = 0
3  if index is less than or equal to n
4     for int i = 0 to n
5        sum = sum + data[i]
6        if sum is less than or equal to t
7           if elements dyn[index][0 to n] do not equal infinity
8              return minimum of dyn[index][0 to n]
9           else
10             dyn[index][newindex] = (t - sum(data[index to
               newindex]))^2 + MemoPartition(data[(i + 1) to n],
               n, t, index + 1)
11          end
12       end
13       newindex = newindex + 1
14    end
15 else
16    return 0
17 end
18 return minimum of dyn[index][newindex...n]
```

```
Input: data: an array of integers, n: the size of data
Output: dyn: the dynamic programming structure
Algorithm: MemoWrapper
1  int dyn[n][n] // Two-dimensional array of integers
2  for int i = 0 to n
3     for int j = 0 to n
4        dyn[i][j] = infinity // Initialized to infinity
5     end
6  end
```

2. Iterative

Input: *data*: an array of integers, *n*: the size of *data*, *t:* the maximum size of a partition
Output: *dyn* - the dynamic programming table
Algorithm: IterativeDyn

```
1  int dyn[n][n]
2  Initialize all elements of dyn to infinity
3  for int i = 0 to n
4     int sum = 0
5     for int j = i to n
6        sum = sum + array[j]
7        if sum is less than or equal to t
8           int temp_min = infinity
9           if i does not equal 0 then
10             for int k = 0 to n
11                if temp_min is greater than dyn[k][i -1]
12                   temp_min = dyn[k][i-1]
13                else
14                   temp_min = 0
15                end
16                dyn[i][j] = (t - sum)^2 + min
17             end
18          end
19       end
20    end
21 end
```

3. Extended Iterative

Input: *data*: an array of integers, *n*: the size of *data*, *t:* the maximum size of a partition
Output: *dyn* - the dynamic programming table
Algorithm: IterativeDynExt

```
1  int dyn[n][n] // O(1)
2  Initialize all elements of dyn to infinity // O(n^2)
3  for int i = 0 to n // n iter. × O(n lg n) = O(n^2 lg n)
4     int sum = 0 // O(1)
5     for int j = i to n // lg n iter. × O(n) = O(n lg n)
6        sum = sum + array[j] // O(1)
7        if sum is less than or equal to t // O(1)
8           int min = infinity // O(1)
9           if i does not equal 0 // O(1)
10             for int k = 0 to n // n iter. × O(1) = O(n)
11                if min is greater than dyn[k][i -1] // O(1)
12                   min = dyn[k][i-1] // O(1)
13                else
14                   min = 0 // O(1)
15                end
16                dyn[i][j] = (t - sum)^2 + min // O(1)
17             end
18          end
19       end
20    end
21 end
22 int min = infinity // O(1)
23 for int i = 0 to n // n iter. × O(1) = O(n)
24    if min is greater than dyn[i][n] // O(1)
25       min = dyn[i][n] // O(1)
26    end
27 end
28 int col = n // O(1)
29 int row = 0 // O(1)
30 int count = 0 // O(1)
31 while column is greater than or equal to 0 // n iter. × O(n)
   = O(n^2)
32    min = infinity // O(1)
33    for int i = 0 to n // n iter. × O(1) = O(n)
34       if min > dyn[i][col] // O(1)
```

```
35          min = dyn[i][col] // O(1)
36          row = i // O(1)
37        end
38     end
39     vector.push(col - row + 1) // O(1)
40     col = row - 1 // O(1)
41     count++ // O(1)
42 end
43 Output count to file // O(1)
44 while the stack is not empty // n iter. × O(1) = O(n)
45     Pop the vector and output this value to the file // O(1)
46 end
```

Analyzing the Solution

To improve the space complexity of our algorithm, instead of storing all subproblems in the dynamic programming structure (in our case, a two-dimensional array), we can just store the minimum cost for each subarray of size 1 to n, where n is the size of the original data array. Instead of using a two-dimensional array as seen above - which would have $O(n^2)$ space complexity with an n-by-n matrix - we could just use a one-dimensional arrays to store the minimum inequality scores, for a space complexity of just O(n). For line-by-line time complexity, see the comments in the code above. Overall, these complexities are outweighed by the complexity of the segment of code where the algorithm is executed, a total running time of $\mathbf{O(n^2 \times \log_2 n)}$.

Closing Comments

The first step to take was to understand the problem, so we decided to go over the partition problem described in section 8.5 of the textbook. By studying this problem, we were able to draw parallels to the project. For instance, we were able to understand the recursive nature of the problem and how to split the input data into smaller partitions. After stepping through the recursion, we were able to get an insight on overlapping subproblems. By identifying these overlapping subproblems we were able to think of an appropriate method to memoize and refine the recursive algorithm.

The most difficult part of project was to derive pseudo-code that could be easily translated into a programming language. Once the pseudo code was completed, the algorithm implementation in C++ was relatively simple. Instead of using simple arrays, we decided that vectors would allow us easier access to the array size in addition to being able to double as a stack when we needed for storing the optimal partition sizes.