Peyton Scherschel

Algorithms

4 April 2020

LP1 and LP2 Equivalence for K = 2

LP1 aims to minimize the maximum sum of any range where as LP2 aims to maximize the minimum sum within a range. For K = 2, we are only dividing the set in one place. Using either LP1 or LP2 will result in the same subranges as one range will have the minimum of the maximum sum of all ranges and the other range will be the maximum minimum sum of all ranges. Because each method solves for one of the two ranges the other range must be the same one that the other algorithm solves for. Therefore, regardless of the algorithm we use we will always have the same result.

Example of how LP1 and LP2 may differ

LP1 and LP2 will often yield different results. In general, the solution that minimizes the maximum sum and the solution that maximizes the minimum sum will not be the same. The different solutions can be show by this simple example. If we have the numbers:

1 5 3 7 1 2

Then a correct result for LP1 would be the ranges {1,5,3} {7,1} {2}. This solution gives the minimum max sum of 9. LP2 on the other hand, would give {1,5,3} {7} {1,2} which has the maximum minimum sum of 3. As we can see both answers satisfy their goals, but the result is two different ranges. This is because when we have more than 2 subdivisions, once the max min sum or min max sum is found the other ranges can be different as long as they do not change our goal number. This results in different ranges as LP1 and LP2 have different goal numbers.

Size of the Solution Space for Linear Partition Problem

The solution space for the linear partition problem will be:

$$\binom{n + k\text{-}1}{n}$$

This is the number of ways to divide a partition S of size n into k ranges.

Recurrence Relation for LP2

$$M_2[n,k] = \max_{i=1}^{n}\{\min(M_2[i,k\text{-}1], \sum_{j=i+1}^{n} s_j)\}$$

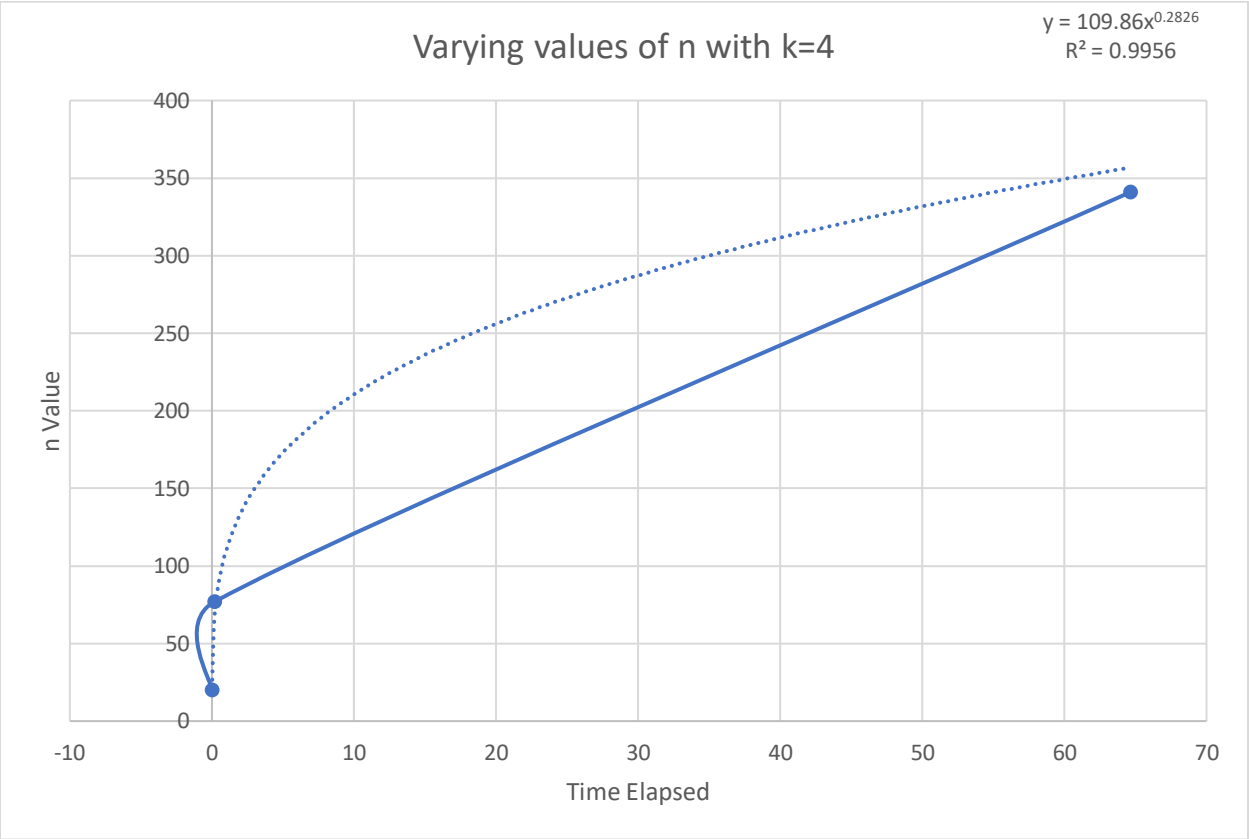
With the basis case of

$$M_2[1,k] = s_1 \text{ for all } k > 0 \text{ and } M_2[j,1] = \sum_{i=1}^{j} s_i$$

Recursive Algorithm for LP2

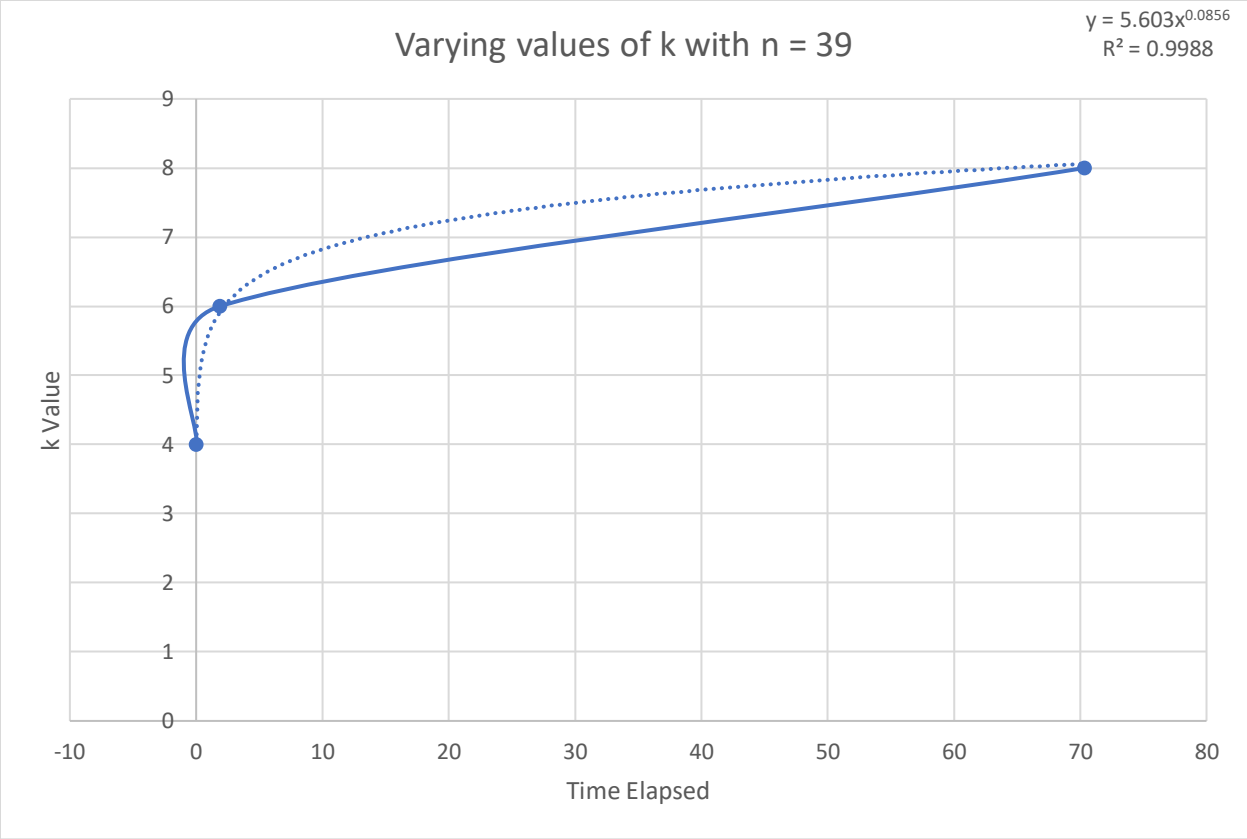
```python
def LP2_recurse(arr, n, k):
    #if n = 1, there is only 1 element left at the end of arr
    if n == 1:
        return arr[0]
    #if k == 1 then we are on our last segment
    #then return the cost of everything remaining
    if k == 1:
        sum = 0
        for i in range(0,n):
            sum += arr[i]
        return sum
    #array to keep track of all of our mins we find
    #eventually we will take the max of this as per LP2
    mins = []
    for i in range(1,n):
        #calculate the min
        sum = 0
        for j in range(i, n):
            sum += arr[j]
        #append to mins
        mins.append(min(LP2_recurse(arr,i,k-1),  sum  ))
        #print(mins)

    return max(mins)
```

Recursive Implementation of LP2 – Growth of Runtime



time elapsed	n value
0.00301528	20
0.193509579	77
64.65779781	341

As we can see, increasing the values of n greatly increases our runtime. The three data points above yield an r^2 value of .9956 for the equation y=109.86*x^(0.282). This shows polynomial growth for increasing values of n.



time elapsed	k value
0.020945	4
1.889946	6
70.31871	8

With increasing values of k we can likewise see polynomial growth. We get an r^2 value of .9988 for the equation 5.603x^(.0856). It is worth noting that both n and k have very similar growths when we increase either of them.

Dynamic Programing Algorithm for LP2

```
def LP2_dynamic(n,k,arr):
    #initialize total array
    total = [0] * (n+1)
    #initialize matrix / table
    M = [[0 for x in range(k+1)] for y in range(n+1)]
    #initialize traceback table
    Traceback = [[0 for x in range(k+1)] for y in range(n+1)]
    #first total will always be 0
    total[0] = 0
    #fill in preliminary totals
    for i in range(1,n+1):
        total[i] = arr[i-1] + total[i-1]

    #fill in remainder of matrix
    for i in range(1,n+1):
            M[i][1] = total[i]
    for i in range(1,k+1):
        M[1][i] = arr[0]


    for i in range (2,n+1):
        for j in range(2,k+1):
            M[i][j] = -1

            for x in range(1,i):
                p = min(M[x][j-1],total[i]-total[x])
                if M[i][j] < p:
                    M[i][j] = p
                    #keep track of traceback
                    Traceback[i][j] = x
```

Traceback Step

```
#traceback step
temp = k
#trace for end result
trace = []
temp = Traceback[n][k]
trace.append(temp)
i = 1
while temp > 0:
    temp = Traceback[trace[i-1]][k-i]
    if temp == 0:
        break
    trace.append(temp)
    i = i+1


#formatted output
for i in range(0,len(arr)):
    if i in trace:
        print("D",arr[i],end=' ')
    else:
        print(arr[i],end=' ')
print()
#return fairness index
return M[n][k]
```

Code Demonstration

Given the following input:

S = (10 6 7 3 8 5 7 9 11 7 15 10 12 6 19 7 3 12 14 6); k = 4

Our algorithm produces the following output:

10 6 7 3 8 5 7 D 9 11 7 15 D 10 12 6 19 D 7 3 12 14 6

42