2nd Lt David Crow

ENG/20M

CSCE 686 Advanced Algorithms, Homework 1

**Chapter 1, Problem 2**

Consider the set bipartitioning problem. Let be a set of positive integers , where is an even value. The problem consists of partitioning into two subsets and of equal size. How many possible partitions of the set exist?

If we let each element be an object and each of and be a box, then we are placing distinguishable objects into distinguishable boxes. Combinatorics says there are ways to partition as described.

Consider the maximum set bipartitioning problem. The problem consists of maximizing the difference between the sums of the two subsets and . To which complexity class does this problem belong?

This problem is fairly straightforward. We can employ the following algorithm to find the optimal solution:

1 MaximumSetBipartitioning(X)

2 X = sort(X) // assume decreasing order

3 let Y = [], Z = []

4 for i = 0 to n/2

5 Y.append(X[i])

6 Z.append(X[i + n/2]

6 return (Y, Z)

The sort on line 2 runs in time (this is the worst-case performance for standard sorting algorithms), the for-loop on line 4 runs in time, and all other operations run in constant time. Clearly, then, this algorithm runs in polynomial time, and thus the problem is in P.

Consider the minimum set bipartitioning problem. The problem consists of minimizing the difference between the sums of the two subsets and . To which complexity class does this problem belong?

We know we can check whether two sets and are an optimal solution to the minimum set bipartitioning problem in polynomial time. We can do so with the following certifier:

1 MinimumSetBipartitioningCertifier(Y, Z)

2 let sumY = 0, sumZ = 0

3 for i from 0 to n/2

4 sumY = sumY + Y[i]

5 sumZ = sumZ + Z[i]

6 let difference = |sumY – sumZ|

7 for i from 0 to n/2 // Y

8 for j from 0 to n/2 // Z

9 let newSumY = sumY – Y[i] + Z[j]

10 let newSumZ = sumZ – Z[j] + Y[i]

11 if |newSumY – newSumZ| < difference

12 return false

13 return true

This certifier works by swapping every possible pair of numbers (where one is in and the other is in ) and checking whether the new difference in sums is smaller than the old difference in sums. Clearly, this certifier runs in polynomial time. This indicates that the problem is in NP.

We now show that the minimum set partitioning problem (one known to be NP-hard) reduces to the minimum set bipartitioning problem with the following algorithm:

1 MinimumSetPartitioning(X)

2 let X’ = X, q = 0

3 for i from 0 to n

4 X’.append(0)

5 let Y, Z = MinimumSetBipartitioning(X’)

6 while q < length(Y)

7 if Y[q] = 0

8 Y.remove(q)

9 else

10 q = q + 1

11 q = 0

12 while q < length(Z)

13 if Z[q] = 0

14 Z.remove(q)

15 else

16 q = q + 1

17 return (Y, Z)

This algorithm works by adding zeros to . These zeros don’t affect either subset sum, so we can remove them from and after finding the optimal minimum bipartition without affecting the difference in sums. We now have the optimal minimum partition – and it doesn’t matter whether or not (as is required by the minimum set bipartitioning problem).

Line 5, of course, uses a solver for the minimum set bipartitioning problem. All other lines run in polynomial time. Thus, the minimum set partitioning problem is (Cook) reducible to the minimum set bipartitioning problem. In showing that an NP-hard problem reduces to minimum set bipartitioning and that minimum set bipartitioning is in NP, we’ve shown that minimum set bipartitioning is an NP-hard problem.

Consider the following greedy heuristic: sort the set in decreasing order; for each , assign it to the set with the smallest current sum. What is the time complexity of this heuristic?

This heuristic employs the following high-level algorithm:

1 MinimumSetBipartitioning(X)

2 X = sort(X) // assume decreasing order

3 let Y = [], Z = []

4 let sumY = 0, sumZ = 0

5 for i from 0 to n

6 if (sumY <= sumZ and length(Y) < n/2)

or length(Z) = n/2

7 Y.append(X[i])

8 sumY += X[i]

9 else

10 Z.append(X[i])

11 sumZ += X[i]

12 return (Y, Z)

*Note: Line 6 checks whether Y has the smaller current sum. Additionally, it checks whether Y or Z is full. This is to ensure , a constraint given in the first question.*

We can analyze this time complexity. Let’s assume we employ a consistent-time sorting algorithm like Merge Sort (without knowing anything about the input, this seems a reasonable choice). Under this assumption, line 2 runs in . The for-loop on line 5 runs in . All other operations run in constant time. Our total time complexity, then, is .

Of course, employing a sorting algorithm with a different time complexity will change the overall time complexity (e.g. using Bubble Sort gives the heuristic a time complexity). In general, though, is accurate.