Assignment 2 Algorithms

By Calvin Hicks

P1 – using strong induction our base cases are 1 and 2.

Base Case: 1 => 2-1 = 1 => TRUE

Base Case: 2 => 4-1 = 3 and 1+ 2 = 3 => TRUE

Inductive Case: assume T(n − 1) + · · · + T(1) + n = 2^n – 1 for all T less than n. Show T(n) + …+T(1) + n + 1 = 2^(n+1) – 1. From this we can subtract T(n − 1) + · · · + T(1) + n and 2^n – 1 from our equation and we get

=> T(n) + 1 = (2^(n+1) – 1) – (2^n – 1)

=> T(n) = 2^(n+1) – 2^n – 1. From our base case we kno0w T(n) = 2^n – 1 so we plug that in and get …

=> 2^n – 1 = 2^(n+1) – 2^n – 1

=> 2^n + 2^n – 1 = 2^(n+1) – 1

=> 2^(n+1) – 1 = 2^(n+1) – 1 and the inductive case is proven. QED

P2

1. 1) k = 20

|  |  |  |  |
| --- | --- | --- | --- |
| Loop Iteration | I , J | A[i][j] | Comparison w/ K |
| 0 | 5, 0 | 13 | A[5][0] < k |
| 1 | 5, 1 | 17 | A[5][1] < k |
| 2 | 5, 2 | 22 | A[5][2] > k |
| 3 | 4, 2 | 19 | A[4][2] > k |
| 4 | 3, 2 | 9 | A[3][2] > k |
| 5 | 2, 2 | 8 | A[2][2] > k |
| 6 | 1, 2 | 7 | A[1][2] > k |
| 7 | 0,2 | 5 | A[0][2] > k (return False) |

1. k = 10

|  |  |  |  |
| --- | --- | --- | --- |
| Loop Iteration | I , J | A[i][j] | Comparison w/ K |
| 0 | 5, 0 | 13 | A[5][0] > k |
| 1 | 4, 0 | 11 | A[4][0] > k |
| 2 | 3, 0 | 5 | A[3][0] < k |
| 3 | 3, 1 | 8 | A[4][2] < k |
| 4 | 3, 2 | 9 | A[3][2] < k |
| 5 | 3, 3 | 18 | A[3][3] > k |
| 6 | 2, 3 | 17 | A[2][3] > k |
| 7 | 1,3 | 11 | A[1][3] > k |
| 8 | 0,3 | 7 | A[0][3] > k (return False) |

b) Prove this algorithm will produce proper results. Proof by weak induction:

Base Case: the starting point of this function includes the whole matrix. If k exists, it has to be in there.

Inductive Case: Assume k is in the original matrix and that the algorithm will work for loop n-1. We must prove that it works for loop iteration n. Now there are three possible sub cases:

Subcase 1:A[i][j] = k This will return True

Subcase 2: A[i][j] > k. Since we start searching in the lower right hand corner, we know that all numbers above [i] will be smaller than A[i][j] and all the numbers to the right of [j] will be larger than A[i][j]. Since A[i][j] is greater than the value we are searching for, we know that all the values [j] in the matrix at row [i] are greater than the value we are searching for and can be eliminated from the search. The search is called recursively on this new submatrix which still may have k inside it.

Subcase 3: A[i][j] < k. By the same logic we can eliminate all values [i] in collum [k] and recall the function on a sub matrix which still may include k.

Also, if the loop condition does not hold we know k couldn’t be in this matrix because we have traversed through all options and are currently at an [i] or [j] value outside of the matrix. Since these three subcases will lead us to a correct result or a reduced matrix that still must contain k if it exists, we can consider this algorithm functional by weak induction. QED

c) worse case complexity is O(2n -2). If an item is in the upper right hand corner and you search starting at the lower left hand corner it will take O(2n-2) steps to travel diagonally across the matrix. More simply, it is O(n) complexity because it is sorting through a matrix linearly and reducing the size of the search by at least one unit every time it loops.

d) 1) if m < k prove A1 can not contain k. This is true because m has to be the largest value in matrix A1. By both the columns and rows being sorted you can say that m is larger than everything to the left or above it. Each of those elements will also have to be larger than the elemnets to the left and above it and this will inductively make m the largest element in A1. Therefore, if m < k A1 can not contain k.

2) if m > k then A4 cannot contain k as well. Using the same logic, we can say that m is the smallest element in A4 because everything in the column and row below and to the right of m will be larger than m and each of this will hold true inductively for each of those elements as well, making m the smallest element in A4 and making k not exist in A4 if it is smaller than m.

e) pseudocode for divide and conquer:

def sattleBack(matrix, k)

n = matrix.size()

m = n/2

if m == 1

if A[1][1] = k| A[2][1] = k | A[1][2] = k| A[2][2] = k

return True

else

return False

else if m < k

return saddleback(A2, k) | saddleback(A3,k) | saddleback(A4,k)

else if m < k

return saddleback(A1, k) | saddleback(A2,k) | saddleback(A3,k)

f) This problem is broken into 3 parts every time a loop is called and each of those 3 parts takes 3 operations to check therefore the recurrence relationship is T(n) = 3T(n/2) + cn. Using masters theorem Alpha = log2(3). Since this is greater than 1 it falls under Case #1 of the theorem and T(n) = O(n^Alpha) = O(n^ log2(3))

P3

1. T(n) = 4T(n/2) + n (T(1) = 1)

= 4(4T(n/2^2) + n/2) + n

= 4^2T(n/2^2) + 2n + n

=4^3T(n/2^3) + 4n + 2n + n

This continues till n/2^i = 1 => i = log(n) so the last term will be 4^log(n)

leading to the Summation from 0 to log n-1 of (2^k)n + 2^(2logn). This summation solves down to n\*O(n) = O(n^2)

1. T(n) = 8T(n/2) + n^3 (T(1) = 1)

=8(8T(n/2^2) + 8(n^3)/8) + n^3

=8^2T(n/2^2) + 8n^3 + n^3

=8^2(8T(n/2^3) + 8n^3/2^6)

=8^3T(n/2^3) + 8n^3 + 8n^3 +n^3

This continues till n/2^i = 1 => i = log(n) so the last term will be 8^log(n)

leading to the Summation from 0 to log n-1 of 8n^3 + 8^(2logn). This summation solves down to O(n^3log(n))

3) T(n) = T(n^.5) + c (T(2) = 1)

= T(n^.25) + c + c

=T(n^.125) +c + c + c

This continues till n^i = 2 => i = log base n of 2. The last term will be 1 because no exponent is added in front of T each iteration.

leading to the Summation from 0 to log n of 2 of (k)c. This summation solves down to O(1) since log base n of 2 can be converted to a constant value.