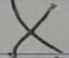
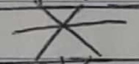
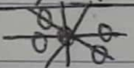
Assignment 2

Question 1:-

Let n be a positive integer and let MaxCrossing(n) be a function that returns the maximum number of line crossings that you can create by drawing n straight lines. Write down a recursive formula for MaxCrossing(n) and analyze the time complexity of the corresponding recursive algorithm. You must write a formal recursive formula including the base case and general recursive step?

Solutions

- Base case: MaxCrossing(n) = 0; if n = 1
- General case: MaxCrossing(n) = MaxCrossing(n-1) + (n-1) that is if n > (greater than 1)

	Max Crossing	
n=2		1
n=3		3
n=4		6

$T(n) = T(n-1) + O(1) = O(n)$

T(n) is the time complexity to compute the MaxCrossing of the n straight lines.

So therefore

$\therefore T(n) = T(n-1) + \underline{O(1) = O(n)}$

Question 2:-

Let $A[1 \ldots n, 1 \ldots n]$ be a 2D array of positive integers with n rows and n columns. Let ArraySum(a,b,c,d) be the sum of all the elements of $A[a \ldots b, c \ldots d]$ (i.e., the submatrix of A with rows a to b and columns c to d). Write down a recursive formula for ArraySum(a,b,c,d) and analyze the time complexity of the corresponding recursive algorithm. You must write a formal including the base case and general recursive step

Solutions sub matrix of rows a to b ; columns = c to d; $a > b$ & $c > d$

- Base case: ArraySum(a,b,c,d) = 0 if $a > b$; or $c > d$
  then return 0

ArraySum(a,b,c,d) will be a is equall $a = b$
if $a = b$ will be the sum of all the elements in rows a that is between c and d

- General case: ArraySum(a,b,c,d) = ArraySum(a,b-1,c,d) + sum of all the elements in row b between c and d {and that is if $a < b$)

- T(n) is the time complexity to compute the sum of all elements of $A[a \ldots b, c \ldots d]$

$$T(n) = T(n-1) + O(n) = O(n^2)$$

Question 3:

Write an efficient algorithm (to the best of your Knowledge) for the following problem, briefly describe why it is a correct algorithm, and analyze the time complexity. If you cannot find any polynomial-time algorithm, then give a backtracking algorithm.

- Problem: Binary Array Core
- Input: Two integers p and q and a binary array A[1...n] i.e each entry contains either a 0 or 1
- Output: Print Yes - if there exists a subarray A[i...K] where 1 <= i < K <= n, with exactly p zeros in A[1...i-1] (there may be 1s) and exactly q ones in A[K+1...n] (there may be 0s). Print No - otherwise.

Solution

The idea of my algorithm is as follows:-

E.g.

Step 1:- if input A[0,1,1,0,1] p=2 & q=1 Output = No
Step 2:- if input A[0,1,1,0,0,1,0,1], p=1 & q=1 Output = Yes
Step 3:- if input A[0] = 0 then A[n-1] = 0
then the Binary Array Core will be (A[2,---n-1], p-1, q)
Step 4:- Input A[0] = 0 and A[n-1] = 0
Binary Array Core (A[2,---n-1], p-1, q-1)
Step 5:- Binary Array Core (A[2,---n-1], p, q)
Step 6:- " " " (A[2,---n-1], p, q-1)

Informal proof idea:- For each A[1...n] if is an even number of all elements; the Binary Array Core will reach length of 2. So when p+q ≥ (greater than or equal to) 2; it then prints NO; otherwise (vice versa) it will print YES. but when p or q is ≠ (not equal to 0) it will print NO.

Informal proof idea:- When each $A[1..n]$ is an odd number of the element, the Binary Array Core will be 1; and that's the lowest length of the subarray to print YES. So when p is not 0 and q is not 0 will need to then find the subarray of the $p-1$ & $q-1$, then the print function is 0 since we can't find ~~the~~ if the subarray is 0 or 1, then will print 0.

Analysis:- Each $A[1..n]$ will need using $O(1)$ to lookup to compute it. Will need the recursive function $(n/2)$ times because of the length of the array. Hence the overall time complexity is

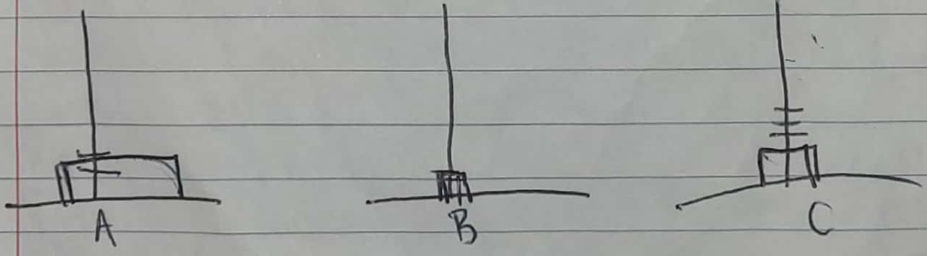
$$O(n/2) \cdot O(1) = O(n)$$

Question 4:

Write an efficient algorithm (to the best of your knowledge) for the following problem, briefly describe why it is a correct algorithm, and analyze the time complexity. If you cannot find any polynomial-time algorithm, then give a backtracking algorithm.

Problem: Tower of Hanoi with Every Tower having some Disks
Input: Each of the three towers contains roughly n/3 disks. Tower A contains floor (n/3) largest disks. Tower C contains ceiling (n/3) smallest disks. Tower B contains the remaining disks. The disks at each tower are already sorted according to the typical tower of Hanoi rule.
Output: A set of moves that transfers all the n disks to C maintaining the Tower of Hanoi that, after each move, the smaller disks at a tower will always be above the larger ones.

Solution
The idea of my algorithms is as follows:
Tower of Hanoi



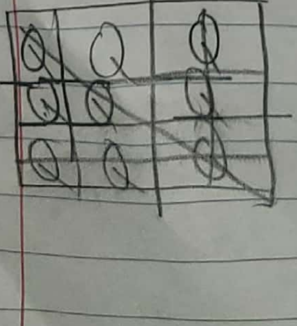
Step 1: Will need to move the disks from disk C to disk B
Step 2: - You can move the floor largest disks from disk A to disk C
Step 3: - We then move the remaining disks from disk A and disk B to disk C
Step 4: - the disks ~~are~~ is successfully moved to disk C with the smaller disk above the larger disk.

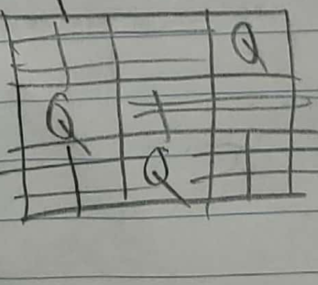
Informal proof idea:- For the 3 disks, the smallest disk is at C while the largest disk is at A, Will need to move the disk from C to the disk B after that will move the disk from disk A to B. The proof idea overall is moving disk from C to B it is n disk problem that has fewer disk left at A

Analysis:- Moving disk from C to disk B is a $O(2^{\frac{n}{3}})$ and $2n/3$ disks is at disk B, and $O(2^{\frac{2n}{3}})$ moving the disks from A to B.
Overall, the time complexity is $O(2^n)$ because the complexity of $O(2^n) - O(2^{2n/3}) + O(2^{n/3})$ is going to be $O(2^n)$

Question 5:-
Write an efficient algorithm (to the best of your knowledge) for the following problem, describe why it is a correct algorithm, and analyze the time complexity. if you cannot find any polynomial time algorithm, then give a backtracking algorithm
Problem: Array Sum
Input: A 2D array A[1...n][1...n] of integers with n rows and n columns. No number is repeated in this array
Output: Return true - if there are exactly n numbers, one from each row, such that the sum of the numbers is 0. Otherwise, return false.

Solution
The idea of my algorithm is as follows:-
Step 1:- Given an n x n grid to be 3x3 grid
~~Step 2:- To check if the n rows does not appear twice would check if the rows~~
Step 2:- Would check the n rows to check if any numbers appears twice
Step 3:- Would check the 3 x 3 grid array using the columns to check no number appears twice
Step 4:- Would check the array diagonally to check if any numbers appears twice. Would check if there are exactly n numbers in the array diagonal
Would find all the possible way to find the array such that it searches it very well.
Using the Queen problem would search the array





~~For Step 4~~:- Applying Randomized polynomial time for a 2D array[1...n][1...n]

Step 1:- It would check the polynomial time of the input size.

Step 2:- Would check the rows and columns to check if the numbers are repeated.

Step 3:- Would loop the rows and columns n times to check if the numbers are repeated.

If there are exactly n numbers from each row or column would return Yes and if there is no number n repeated would be No.

If the exactly n numbers from each row/column is repeated would run n times with the result of each run statistically independent of the others; it would print YES if at least one of the probability appears. So if the array is checked 100 times, then the chance of it giving the wrong answer every time is lower than the chance that cosmic rays corrupted the memory of the computer running the algorithm.

Time Analysis:- The time complexity is $O(n^n)$; the check/determine that there is any conflict with the previous number $O(n^2)$

$T(n) = T(n-1) + O(n^2) = \underline{O(n^n)}$

Question 6:-
Write an efficient algorithm (to the best of your Knowledge) for the following problem, briefly describe why it is a correct algorithm and analyze the time complexity. If you cannot find any polynomial-time algorithm, then give a backtracking algorithm
Problem: Repeated Number
Input: An array A[1...3n] of positive integers
Output: If any number in the array appears at least n times, then print such numbers. Otherwise print 'none'

Solution:-
The idea of my algorithm is as follows:-

Step 1:
Will have to apply Heap/Merge sort to sort the integer
Step 2:- When done sorting, the elements that are equal belong to each other
Step 3:- Would have to loop the positive integers to find the ~~near~~ repeating numbers
Step 4:- Will loop and check if the n positive integer is equal to n, will then will print valid number that repeated; otherwise if the number integer is not equal to n would print none that the number does not apply.

for e.g. A =

1	3	3	3	1	1

Sort A =

1	1	1	3	3	3

num = 3 and 1

<u>Informal proof idea:</u>- For $A[1 \ldots 3n]$ of positive integers would have to sort the numbers in order to loop through the array to find the repeated number. When the number is then equal to n, it would print the number or if is not equal, it would print none. Would just keep searching the array to be sure, same number does not appear twice.

<u>Time Analysis:</u>- The above time complexity would be $O(n \log n)$. if you divide the length of the array it would be $O(3n \log(3n))$ and when checking to compare the two numbers if they are of the same thing would be $2(3n-1)$

So: $O(3n \log(3n)) + 2(3n-1) = O(n \log n)$

So the time complexity is <u>$O(n \log n)$</u>