Contents

Report — HomeWork

24120409 –Lê Thanh Phong

March 2025

A Introduction

Doing homework is a must-do job of a student after each lesson on the class because there are many benefits to that can help both the student who did it correctly and the teacher guiding.

If there is any problem with this report, please contact with me by 24120409@student.hcmus.edu.vn

B About The Homework

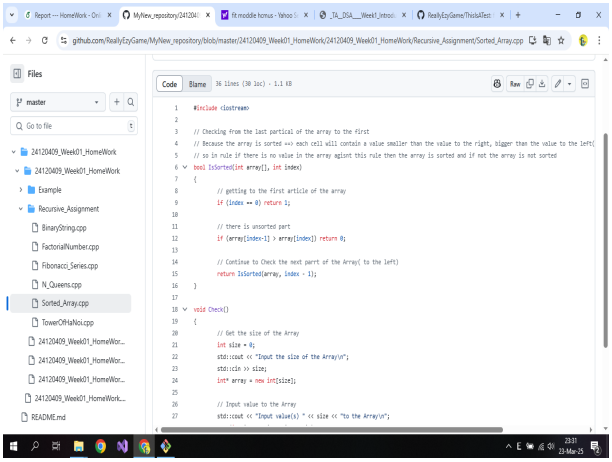
In this report, we talking about homework of Week 01(practice data structure and algorithm course), the report is an essential part of the homework because not only it mirror the understand of the student after each class, but also what they have learned outside the program teaching in school so that the guide teacher can give the best evaluation students and prepares for next lecture so that it can help the students to develop skills in many crucial parts of programming.

This report will give the teacher to have a vision about the homework of week 1 on some faculty that I have decided to include:

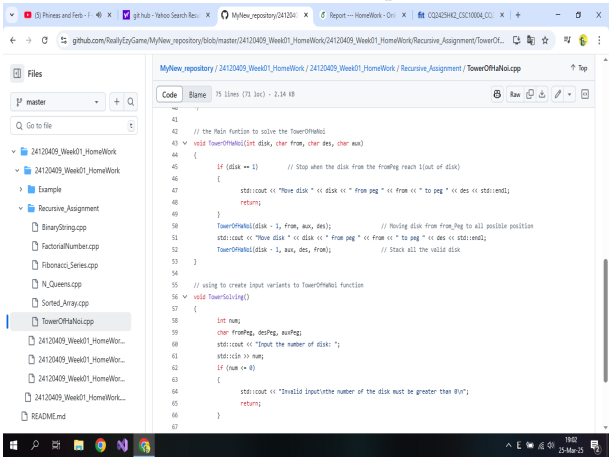
1. Problem
2. Purpose of the function
3. The Idea Approaching the Solution to the Problem
4. Pseudo Code of the algorithm or diagram
5. Test case of the function
6. The space and time complexity of the function

This is the link to the github of the week 1 requirement: Click This

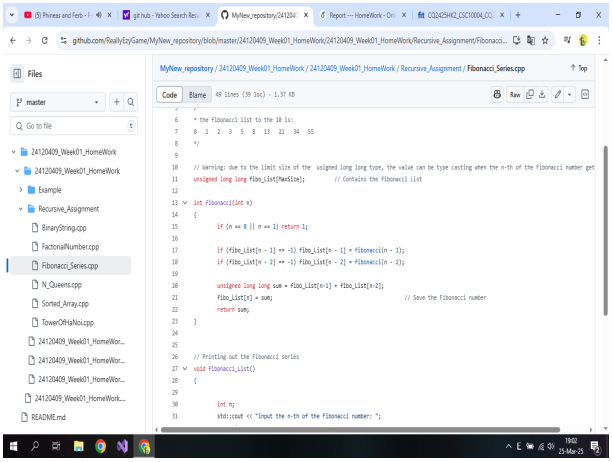
The picture of the Github upload file, I have put it in the same .zip file.



Picture 01 of Github practice



Picture 02 of Github practice



Picture 03 of Github practice

C Tower Of Ha Noi

C.1 Problem

Tower of Ha Noi, a classic problem of the recursion that every programmer who have learned about programming at least heard one time. The problem is a simple: given 3 rod and a number of n disk(s) on a rod(usually the left rod), our mission is to move each time a disk from a rod to another rod until the other rod(usually the right rod) contains all the disk(s), each disk can stack on the other disk if and only if the disk is smaller than the below disk, providing the middle rod as the auxiliary rod to move the disk

The puzzle was invented by the French mathematician Édouard Lucas, first presented in 1883 as a game discovered by "N. Claus (de Siam)" (an anagram of "Lucas d'Amiens"),and later published as a booklet in 1889 and in a posthumously-published volume of Lucas' Récréations mathématiques. Accompanying the game was an instruction booklet, describing the game's purported origins in Tonkin, and claiming that according to legend Brahmins at a temple in Benares have been carrying out the movement of the "Sacred Tower of Brahma", consisting of sixty-four golden disks, according to the same rules as in the game, and that the completion of the tower would lead to the end of the world. Numerous variations on this legend exist, regarding the ancient and mystical nature of the puzzle.

If the legend were true, and if the priests were able to move disks at a rate of one per second, using the smallest number of moves, it would take them $2^{64} - 1$ seconds or roughly 585 billion years to finish, which is about 42 times the estimated current age of the universe.

C.2 Purpose of the Function

Solving tower of Ha Noi can be tedious, especially when the number of disk(s) to move is large, but luckily we have a thing that can run our problem really fast just by given it some command told it to do something. The computer can do such task that with human speed, it will consider impossible to accomplish and in this case is the tower of Ha Noi.

C.3 The Idea Approaching the Problem

When diving into this problem, we can try to solve it, as we usually play around as we were kids. The first thing came up in my mind was to solve the problem by moving the disk from the from rod to the destination rod first, if it is invalid then moving it to the auxiliary rod, if both of them is invalid then I change the rod from to the auxiliary rod then repeat the process, if it lead to dead-end then backtracking to where we start and do it with other choice. Although the plan worked and I could solve the puzzle but when implemented, the algorithm got really complicated too bad to be call a good algorithm in any way. On top of that, if we use that plan, the algorithm is literally a brute-force solving puzzle

and that is very bad when we want our program to have the best space and time complexity when solving a question. That is when I thought, is there any way to help us solving the question by only move the right disk and put it to the right rod? And turn out there is. We take the easiest case which we only need to move 1 disk from from rod to destination rod, then we increase the number of disk by 1 the problem become 2 disks puzzle. We can solve it by put the first disk to the auxiliary rod then put the second disk to the destination rod and put the first disk on top of the second disk, the puzzle is finished. I realized a pattern, if the number of disk is 3, then we need to move 2 first disk to the auxiliary rod then moved the last disk to the destination rod, then we can solve the problem of 3 disks become the problem of 2 disks, so we can solve it easily. The problem of 4 disks is still the same, we try to put all the first 3 disks to the auxiliary rod then move the last disk to the destination disk, the problem becomes 3 disk, then 2 disks, then 1 last disk. The problem of 5 disks, 6 disks,... n disks is still the same.

Following that reason, when we facing a puzzle of n disks, the first thing to do is to move n-1 first disks to the auxiliary rod, since it is still a complicated case to solve, we break the problem again by move n-2 disks to the auxiliary rod, then n-3, n-4,... until the problem is only 1 disks to move left, since we can solve 1 disk, it's means that the second disk(or we can say 2 disks problem) can be solved, since thee first 2 disk is now stack on each other, the third disk(3 disks problem), can be solved and so on.

This new method worked well while planing and even the implement part is much better than the old method by using recursion.

C.4 Pseudo Code

The algorithm for solving the problem can be describe in the form of Pseudo code that I represent below:

Algorithm TowerOfHaNoi(disk, from, des, aux)

INPUT: number of disk(s) on the from rod, the from rod contains the disk moving, the des rod(destination rod), the auxiliary rod temporary contains the rod.

OUTPUT: each step/move disk to accomplish the problem in the fewest step.

```
if (disk = 1) then
        display(move disk disk from rod from to rod des)
        return (void)

TowerOfHaNoi(disk-1, from, aux, des)
display(move disk disk from rod from to rod des)
TowerOfHaNoi(disk-1, aux, des, from)
return (void)
```

Explanation:

Sometimes, solving a problem as we see it a one big problem can be a hard thing to do, tower of Ha Noi is one of the case but by using recursion as an alternative way to solve, it is easier even though it can be hard to understand at first. Let me explain the algorithm: we know that the number of disk 1 means that we can solve the problem by simply taking the disk from the rod from and placing it to the rod destination, when we call the function TowerOfHaNoi() check if the number of disk is 1(solvable), but if the number of disk is a different number then we need to prove it can be solved.

Note: every display command is consider as one move to the disk.

Theory: for all Tower of Ha Noi puzzles with number of disk greater than 1, then it can be solved.

Proof: giving that the case which the number of disk is minimum (n = 1) can be solved as the reason I gave in the explanation, consider that the puzzle with the number of disks is n can be solved.

Now we need to prove that the puzzle of n+1 disk can be solved as well.

We can recognize the common that both of the n disks and n+1 disks have is the first n disks of the puzzle. It's means that we only need to prove that we can move the (n+1)-th disk to the destination then all the n first disks can be solved too. First we ignore the (n+1)-th disk, we move the n first disks but we will move them to the auxiliary rod instead of the destination rod, now the puzzle become moving n disks, since the puzzle of n disks is true(in the theory) then we really can move n disks to the auxiliary rod by using destination rod as an other auxiliary rod, after that we can simply move the (n+1)-th disk from from rod to destination rod, finally we move n disks from auxiliary rod to destination rod by using the from rod as an auxiliary rod. So in conclude, we can solve puzzle n+1 disks. By using induction, since the puzzle 1,n and n+1 disks puzzle can be solved we can conclude that the Tower Of HaNoi can be solved.

We have solved the tricky part, now we only need to implement by using programming language, pseudo code, etc... we realized that if the puzzle contains n disk then it need to break to smaller case just like how it works in the Proof part above, then it needs to be broken into n-1, then n-2,... until the case is 1.

TowerOfHaNoi(n-1,from,aux,des)

That's the most important and the reason why we need to have the first recursion in the algorithm. It's main part is to break the problem into the smaller one until the size of the problem is 1(can be solved without using recursion). The reason behind the change of between the auxiliary and destination rod is that we wanted our n first disks to move to the auxiliary rod not the destination rod as begin. When the number of disk is 1 then

‘

if (n = 1) then
display(move disk **disk** from rod **from** to rod **des**)
return (void)

The code inside the if condition is to solve the smallest problem when there only 1 disk left. Our next thing to do is move the next disk to the remain rod then stack them up together

display(move disk n from rod from to rod des)
TowerOfHaNoi(disk-1, aux, des, from)

The act of moving disk x from rod from to rod destination is taking the next disk and add to the des rod, then stacking the disk(moved before) to the current disk until we stack all n-1 disk to the auxiliary rod then we move the n-th disk to the destination and repeat the process until all the disk have been moved to the destination rod.

C.5 Test Case of The Function

In this part, I will give the result the algorithm with the input is 5 disks
Move disk 1 from peg A to peg C
Move disk 2 from peg A to peg B
Move disk 1 from peg C to peg B
Move disk 3 from peg A to peg C
Move disk 1 from peg B to peg A
Move disk 2 from peg B to peg C
Move disk 1 from peg A to peg C
Move disk 4 from peg A to peg B
Move disk 1 from peg C to peg B
Move disk 2 from peg C to peg A
Move disk 1 from peg B to peg A
Move disk 3 from peg C to peg B
Move disk 1 from peg A to peg C
Move disk 2 from peg A to peg B
Move disk 1 from peg C to peg B
Move disk 5 from peg A to peg C
Move disk 1 from peg B to peg A
Move disk 2 from peg B to peg C
Move disk 1 from peg A to peg C
Move disk 3 from peg B to peg A
Move disk 1 from peg C to peg B
Move disk 2 from peg C to peg A
Move disk 1 from peg B to peg A
Move disk 4 from peg B to peg C
Move disk 1 from peg A to peg C
Move disk 2 from peg A to peg B

Move disk 1 from peg C to peg B
Move disk 3 from peg A to peg C
Move disk 1 from peg B to peg A
Move disk 2 from peg B to peg C
Move disk 1 from peg A to peg C
If we follow the output, we can in fact get the solution with the fewest step

C.6 Space and Time Complexity

C.6.1 Space Complexity

Space for parameter for each call is independent of n i.e., constant. Let it be k. When we do the 2nd recursive call 1st recursive call is over . So, we can reuse the space of 1st call for 2nd call . Hence ,

T(n) = T(n-1) + k
T(0) = k
T(1) = 2k
T(2) = 3k
:
:
So the space complexity is O(n) as n is the number of the disk

C.6.2 Time Complexity

Let the time to run this algorithm is T(n). Since there are 2 recursive call n-1 disks, one constant time used for for condition checking and moving a disk from from rod to des rod. Let that constant time operator is k1.

T(n) = 2T(n-1) + k1
T(0) = k2, the constant time to move the smallest disk
T(1) = 2k2 + k1
T(2) = 4k2 + 2k1 + k1
T(3) = 8k2 + 4k1 + 2k1 + k1

we realize there is a pattern that the time complexity follows for every time the number of disk increase by 1 the time require to complete the puzzle will growth 2 times plus and plus k1. This is because every time the number of disk increase by 1, we need to do 3 things:

1. Moving all the above disks to the auxiliary rod.
2. Moving the last disk to the destination rod.
3. Moving the remain disks from auxiliary rod to the destination rod.

The first action requires T(n-1) time to complete, the second action is a constant time action, the third action need T(n-1) time to complete. Since, T(n-1) is also call recursive to T(n-2), then T(n-3)... until T(0) so time require to run the algorithm double each time we add 1 more disk

We guess that the time complexity is: $1 + 2 + 4... + 2^n = \sum_{i=0}^{n}(k2^i)$

Proof: Given the case of n disk has the time to solve is T(n), we have:

$$\begin{aligned}
&\text{T(n)} = \text{2T(n-1)} + 1 \\
&= 2(\text{2T(n-2)} + 1) + 1 = \text{4T(n-2)} + 2 + 1 \\
&= 4(\text{T(n-3)} + 1) + 2 + 1 = \text{4T(n-3)} + 4 + 2 + 1 \\
&\quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \\
&= 2^n T(0) + 2^{n-1} + ... + 4 + 2 + 1 \\
&= 2^n + 2^{n-1} + 2^{n-2}... + 4 + 2 + 1 \\
&= \sum_{i=0}^{n}(2^i)
\end{aligned}$$

Using the big-Oh notation we get the asymtonic time complexity for the worst case of the algorithm is: $\text{O}(\sum_{i=0}^{n}(2^i)) = \text{O}(2^n)$

Similarly, we can proved the best case of the algorithm is $\Omega(\sum_{i=0}^{n}(2^i) = \Omega(2^n)$

Since the worst case and the best case have the running time of 2^n it's means that for all case running, the algorithm will always run in the time of 2^n or we can say the algorithm has the big-Theta is $\Theta(2^n)$.

D N-Queens Puzzle

D.1 Problem

The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal. There are 92 solutions. The problem was first posed in the mid-19th century. In the modern era, it is often used as an example problem for various computer programming techniques.

the eight-queens puzzle is a special case of the more general n queens problem of placing n non-attacking queens on an nxn chessboard. Solutions exist for all natural number n with the exception of n = 2 and n = 3. Although the exact number of solution is only know for $n \leq 27$, the asymptotic growth rate of the number of solutions is approximately $(0.143n)^n$ by using the fastest algorithm. But in this article, we only talk about the normal N Queen Solving algorithm has the time complexity of O(n!)

D.2 Purpose of the function

Like the problem of Ha Noi Tower, the N-Queens puzzle can take a long time to complete, especially when the number of queen is very large. That's led to the

time we need to solve the riddle can growth aggressively that any mere mortal human cannot solve it. But luckily we have computer to solve it.

D.3 The Idea Approaching the Solution

The first time I saw the N Queens Problem was when I read "Introduction to Artificial Intelligence"(author: Le Hoai Bac), I though this would be a great practice for one of my first experience coding. But I quickly realized that this is the hard task that I couldn't do at that moment so I chose to put it away for a time. Recently I have met again in the homework given by our practice teacher. He told us to follow the way of recursion and break the problem into smaller one for easier to solve. Thanks to him, I can finally made the algorithm by myself even though I still need a lot of help.

Approaching the N Queens problem shares many things that like the Tower of Ha Noi puzzle, one of them is to break the problem into the smaller one. First given the chess table has the size of NxN and the total numbers of queen is N the first time we put a Queen, we limit the problem by eliminate the row and the column that has the Queen on, by that way we break the problem into the smaller on which has the size of N-1, then N-2... then 1. If the the total number of the queen on the board is the same with the number of the row column then it's a solution to the puzzle.

D.4 Pseudo Code of the Algorithm

in this section, I represent the pseudo code of the algorithm solving the N queen problem as N is the number of queens need to be placed on the chessboard has the size of NxN:

```
        const int N: the size of the chessboard, number of queens
              need to be placed.
        int table[N][N]: the chessboard
        solution: count total solution for the riddle
    Algorithm IsSafe(row, col)
        Input: the row and the column placing the queen
        Output: that position in the chess board can or cannot place the
queen
        r <-- 0
        for(r < row; row <-- row + 1)
            if (board[i][col] = 1) then
                return false
            if ((col - (row -i) ≥ 0) and borad[i][col - (row - i)] = 1) then
                return false
            if ((col + (row - i) < N) and board[i][col + (row - i)] = 1) then
                    return false
        return true
```

```
Algorithm NQueenSolve(row)
    Input: the row place the first queen( usually the first row)
    Output: the number of total solution for the N queens problem
    if (row = N) then
        solution <-- solution + 1
        return (void)
    col <-- 0
    for (col < N, ++col) do
        if( IsSafe(row,col)) then
            board[row][column] = 1
            row <-- row + 1
            NQueenSolve(row)
            board[row][column] = 0
```

Explanation:

Although the algorithm IsSafe() represent first befor the NQueenSolve() but it's the NQueenSolve() is the first function we counter in the algorithm for solving the N Queen puzzle.

The main purpose of the function NQueenSolve() is to decide wether or not put a queen in a position and checking around the chessboard to find the safe sport. We can see it through the pseudo in the function:

```
col <-- 0
for (col < N, ++col) do something
```

the purpose of this 2 lines are begin to search for a safe sport by moving to the next column since the position is occupied by other queen or the position is unsafe to place a queen. Then we have the condition scope in this loop:

```
if( IsSafe(row,col)) then
    board[row][column] = 1
    row <-- row + 1
    NQueenSolve(row)
    board[row][column] = 0
```

By checking a position inside the chessboard through the function IsSafe() (which I will talk later), there are 2 cases can happen here: first is the position is safe to place a queen, second is the opposite, the position is not safe to place a queen.

In the first case, the function will put a queen to that position and recursively call itself, the reason why it calls itself is because we intend to break the problem into the smaller one like the previous problem is that: instead of solving the N queens on and NxN chessboard, we want to reduce this problem by shrinking it one by one for each time we put the a queen. This way, the problem becomes smaller and smaller when we put a queen. From N queen on an NxN chessboard, the problem now is N-1 queen on (N-1)x(N-1) chessboard, then N-2, N-3 and so on with one condition: the smaller problem must place the queen but some place is occupied by the previous queen (because of the diagonal moves). Because the

queen has been placed so there is no valid position in the same row, that's why we need to increase the row by one before break the problem to smaller one. The final line in the if condition scope is call "back tracking" part of the function, the reason why such command exist because for each time we mark the position of the queen by value 1, it didn't disappear by it self and such thing is a bad thing for us when we want to find all the valid position, if we don't find a way to fix it to 0 then if there is a case that is satisfy the condition of the N queens problem all the case behind it will also satisfy and that is wrong(because all the case share the same chessboard table)! So we need to find out a way to change it back to 0. This is when the "Back-Tracking" technique comes in handy. With the back-tracking technique we can change value 1(queen) back to 0 (no queen) after the algorithm finished finding a solution or not to make sure in the next finding, there is no queen on the board to mess up our finding.

The second case is where the place is not a safe place to place a queen, if that happens the function will move to the right by one column.

In conclude, the function purpose is to move around the board and finding a place to set the queen and erase the. the algorithm act like this: if the position is a safe place to put the queen then it will of course put the queen and breaks problem into the smaller one by making a recursion and repeat the process until it find a solution(which is row equal to N number of queen because the algorithm only move one by one when there is a queen put onto the table) or the set of the queens on the table is a wrong solution, no matter which happens, both of them will lead to the algorithm to move to the right by one column and find a new solution. If the position is unsafe then it simply move to the right next column in repeat the process.

The second function we want to talk about is the IsSafe() function, like its name the reason why this function exist is to determine wether a position in the chessboard is valid to put a queen or not. We've known that the set of move for the queen is include 3 parts: move anywhere in the same column, move anywhere in the same row and move diagonally. This is what the function tend to do, it will check if there is any queen is placed on the pattern and if there is, the position will consider as unsafe to put a queen on it.

$$\begin{aligned}&\mathrm{r} < -- \ 0\\&\text{for}(\mathrm{r} < \mathrm{row};\ \mathrm{row} < -- \ \mathrm{row} + 1)\end{aligned}$$

Since the rule of putting the queen is to put from right to left, from up to below so in this case queen(s)(if exist) can only appear in the left of the current position we are checking. This is why we only need to check to the current row instead of checking all the row.

$$\begin{aligned}&\text{if } (\mathrm{board[i][col]} = 1) \text{ then}\\&\quad \text{return false}\end{aligned}$$

The first condition in the loop is to check wether there is any queen in the

same row or not

if ((col - (row -i) $\geq$ 0) and borad[i][col - (row - i)] = 1) then
return false

The second if condition in the loop is to check wether there is any queen in the diagonal left or not(we won't count the case which the row is out of the chessboard)

if ((col + (row - i) < N) and board[i][col + (row - i)] = 1) then
return false

the final if condition in the loop is to check wether there is any queen in the diagonal right or not(we won't count the case which the row is out of the chessboard as well)

return true

This is the case where there is no queen that can attack the current position, so we can put the queen onto this position.

D.5 Test Case of the Algorithm

In this section we will use the test case of N = 8 means that the table is an 8x8 chessboard and the number of queens that we need to put on the table is also 8. Input: 8 Output: Number of solution: 92

The result 92 is proof to be correct in the case

And for the 10 queens puzzle the number of solutions is 724

D.6 The Space and Time complexity

D.6.1 Space Complexity

Using this algorithm, we need to make an 2D array of size NxN to contains all possible position of the queen so the algorithm always has the space complexity O(n^2).

D.6.2 Time Complexity

The first thing we need to notice is that for every time the algorithm reach a new position, it will try to check if the spot is safe or not through calling IsSafe() function. The IsSafe() function will check all row in the table, we can easily evaluate the worst case of function, which is when the function check all row without finding a queen position that is invalid this lead to the total loop time is n if the the riddle has the size of n.

$\forall n \geq 1$, the cost of the function: n (1)

We also notice that: if the problem has the size of n, it can generate n problems with the size of n-1, this happen because for each 1 square in n squares can put queen, it will call itself(recursion) to solve the smaller part of the problem, then to the n-1, it can generate n-1 problems has the size of n-2 and so on. We can give the mathematic formula for this scenario

$\forall n \geq 1$, the cost of the recursion part is:

$$n * (n-1) * (n-2)...1 = \prod_{i=1}^{n}(i) = n! \ (2)$$

Since each time the algorithm find out a new position, it always check if the place is safe to put a queen. Following that for all time recursion in (2) the cost for each to check is (1) and base on that we have the formula:

$$n * n!$$

Since the algorithm run the same for all cases and doesn't have anything to tell it to stop at some condition, the algorithm run with the same cost for all cases. Base on that reasoning, we use the Big-Theta notation to give the conclude(in mathematic) all the case are limit in the same bound.

Cost of the NQueenSovle(): $\Theta(n * n!)$

E Fibonacci Series

E.1 Problem

the Fibonacci sequence is a sequence in which each element is the sum of the two elements that precede it. Numbers that are part of the Fibonacci sequence are known as Fibonacci numbers, commonly denoted F_n. Many writers begin the sequence with 0 and 1, although some authors start it from 1 and 1 and some (as did Fibonacci) from 1 and 2. Starting from 0 and 1, the sequence begins.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ... (sequence A000045 in the OEIS)

The Fibonacci numbers were first described in Indian mathematics as early as 200 BC in work by Pingala on enumerating possible patterns of Sanskrit poetry formed from syllables of two lengths. They are named after the Italian mathematician Leonardo of Pisa, also known as Fibonacci, who introduced the sequence to Western European mathematics in his 1202 book Liber Abaci.

This is one of the first and famous recursion formula for many student first time learn discrete mathematic, computer science, etc...

E.2 Purpose of the Function

Like many problem in this report, finding a value in the Fibonacci series can can be tedious and consuming time especially when the number we need to check is a large one. For that reason, we need an algorithm to find the Fibonacci number and run on the computer to do such boring task.

E.3 The Idea Approaching the Solution

The first thing I notice about the problem is the formula of the Fibonacci number(since I had leaned it in the discrete mathematic course before), so base one that I implement algorithm they call the naive Fibonacci(the $O(2^n)$) which is very bad, then until one day I leaned about the optimization for recursion which is a new approach to me, instead of just blindlessly calculate all the Fibonacci we can create an array to contain the Fibonacci we have calculate before so we don't need to calculate with it again by that way the algorithm reduce from 2^n to simply n which is very good.

E.4 Pseudo Code

In this report, I will use the algorithm of the recursion optimization Fibonacci recursion instead of the naive one.

MaxSize: an positive integer

FiboList[MaxSize] // default as -1, contains all the value in the Fibonacci series

Algorithm Fibonacci(n)

INPUT: the ordinal N in the Fibonacci series

OUTPUT: the number of No N in the Fibonacci series

if (n = 0) then return 0

if (n = 1) then return 1

if (FiboList[n-1] = -1) then FiboList[n-1] = Fibonacci(n-1)

if (FiboList[n-2] = -1) then FiboList[n-2] = Fibonacci(n-2)

FiboList[n] ← FiboList[n-1] + FiboList[n-2]

return FiboList[n]

Explanation:

We knew that the formula of the Fibonacci number is the next number is the sum of the two number before it we can show it in the mathematic notation:

$$F(n) = F(n-1) + F(n-2)$$

This is one pure mathematic problem, we can use the formula as the main part of the algorithm. Since F(n-1) and F(n-2) is Fibonacci numbers, so we still need to calculate it before calculating the result of F(n), that's why we check F(n-1) and F(n-2) first

if (FiboList[n-1] = -1) then FiboList[n-1] = Fibonacci(n-1)

if (FiboList[n-2] = -1) then FiboList[n-2] = Fibonacci(n-2)

The recursion will going until n is equal to 0 or 1 the function will return 0 or 1 as the result of the first two Fibonacci numbers.

if (n = 0) then return 1
if (n = 1) then return 1

E.5 Test Case

In the Test Case section, I will use only the valid case where the n is the natural number

Input: 5
Output: 1 1 2 3 5

The algorithm show the first 5 Fibonacci number correctly
Now we try with 10

Input: 10
Output: 1 1 2 3 5 8 13 21 34 55

E.6 Space and Time Complexity

E.6.1 Space Complexity

The recursion optimization Fibonacci using the space for 2 critical things: first of all it needs an array contain the result of the Fibonacci number before for getting it fast when it's needed, second is the recursion part which is the back bone of the Fibonacci algorithm.

With the problem calculating the Fibonacci has the size of n requires us to have an array of number can also contains n numbers which is require n space.

The recursion part, when we run the algorithm to find the ordinal n of the Fibonacci. The first time the algorithm call, we don't have any Fibonacci number in the array, so we need to run to the case where n = 0 and n = 1, so the depth of the recursion is n which cost n space.

Combine two reasoning, we have the worst case of the algorithm is 2n. Using the Big-Oh we get

$$O(2n) = O(n)$$

Since the algorithm runs the same for all value of n belongs to the natural number set so the Big-Theta of the algorithm is:

$$\Theta(2n) = \Theta(n)$$

E.6.2 Time Complexity

This is where the optimization really shine. When using the naive algorithm ,the time complexity of the algorithm can growth massively. That because when we find the value n the function recursively call 2 of itself with the number of n-1 and n-2 and both of them recursively call n-2, n-3 and n-3, n-4 again and again. That's lead to the time complexity in the form of:

$$T(n) = T(n\text{-}1) + T(n\text{-}2) + C$$

As C is the constant for some operator in the algorithm.

We can prove that the running time of the algorithm is $\Theta(2^n)$ which is not great. Using the optimization way, instead of finding each Fibonacci numbers discretely by finding one value by one value, we save all the value we have calculated to the array so for the next time we only need to take the value in the array instead of calculate it over, over and over and over again. Therefore, for any n value greater than 0, we only need to recursive 1 time for each number and save it to the array for the later use. By that way, the time complexity for any case is:

$$T(n) = T(n\text{-}1) + C$$

For C is the constant time to do some operator in the function. Solving it we get

$$T(n) = T(n\text{-}1) + C = T(n\text{-}2) + C + C = T(0) + (n\text{-}1)C = nC$$

Since the algorithm runs the same for all case, so Big-Theta of the algorithm is $\Theta(nC) = \Theta(n)$ which is linear time(much better than the old one).

F Binary String

F.1 Problem

Binary strings are fundamental of computer science and have a wide range of application like memory storage, digital communication, data compression, etc... but the problem arises when we need to know how many binary strings there are when the length of the string is n. Of course we can use the formula 2^n to give the result but we can do it in the other way by printing out all the binary strings have the length of n.

F.2 Purpose of the Function

Giving the string of n characters, our mission is to find all the combination string of 0 and 1.

F.3 The Idea Approaching the Solution

The first thing that came to mind when solving the problem was to try the bitwise operator notation, since the binary string problem and the bits almost identical. The first thing I tried was to convert the string type to represent it in

the binary form, then add 1 to it, converse it into string again. The good side of this is that is all the operators are bitwise notation, because of that, we can eliminate the loop in the main function so the run time of the algorithm can drop down to $\Theta(2^n)$ instead of $\Theta(n * 2^n)$. But this way has a lot of downside, first of all, the length of the string is limit by the numbers of bits we have, sadly the value with the greatest bits I could find was long long type which is 64 bits; it's means that we can only get the string with 64 char if we do this. the second thing is the converse task, it can reach out to the cost $\Theta(n)$ when we converse the from string to bit and from bit to string(because the function only give us string type as input). With that downside, I decided if we kept the bitwise operator notation but using it on the string instead of bits and turned out it worked, even though there are many function to do this(even with the non-recursion one with the better performance) but still this is of the algorithm that can find all the string of binary.

F.4 Pseudo Code

The pseudo code below is the algorithm printing out all the combination of the binary string.

Algorithm BinaryString(str,n)
INPUT: the string in the previous recursion, the length of the string
OUTPUT: a binary string in 2^n combination
if (the first call) then
PreCondition(str)
display(str)
flag ← true
pos ← n - 1
for (;pos ≥ 0; $pos \leftarrow pos + 1$) do
if (str[pos] = 0) then
str[pos] ← 1
flag ← 0
break
else then
str[pos] = 0
if (flag = true) then
return (void)
return BinaryString(str,n)

Explanation:

The algorithm above generates all the 2^n combinations of the binary string and stops when it encounters the string of only 0(after the loop).

The first thing we need to notice here is the first if condition of the algorithm

if (the first call) then

PreCondition(str)

The Precondition() main purpose is to create the string full of 0 to begin the algorithm when we calling the function for the first time

display(str)

The purpose of this function is to print out the result. The result can be the str from the previous recursion or the str of 0 gets from the PreCondition() function.

```
flag ← true
pos ← n - 1
for (;pos ≥ 0; pos ← pos + 1) do
    if (str[pos] = 0) then
        str[pos] ← 1
        flag ← 0
        break
    else then
        str[pos] = 0
```

Before I start to explain, we must know the rule of adding operator in binary. Just like adding in the normal number, the binary adding follow the same rule " 1 plus 0 is 1, 0 plus 1 is 1, 0 plus 0 is 0, 1 plus 1 is 0 the value next to the left adding 1". Using this rule, we try to create the string by using the previous string add 1 to it. The for loop exist because we want to add 1 for the current string to create the next binary string. The first condition tells that: "I found 0, then I add 1 to it, now it becomes 1 which is a new string so I completed" then the loop stop because it complete the operator adding 1. The second condition of the loop is to remember add 1 for the next to the left number in the string like: "Since you're 1, I'm 1, 1 plus 1 means 0 then add 1 to the next left value".

But the function seem to run endlessly without any sign of stop, that's when the 'flag' comes in handy. The flag first thing is to check the there is any 'number' 0 in the string, if there is, the string can still be add 1, on the other hand, if it found no 0 exist in the string which means the string is full of 1, so we can stop now.

```
if (flag = true) then
    return (void)
return BinaryString(str,n)
```

The condition is to check whether the string of 1 or else. When the flag is true which is the function cannot found any 0 'number' in the string, then it will stop the algorithm because it have done its job, it is to display all the binary string has the length of n. If the flag is false then the string is not the string of 1 so we can adding more or we can say there is more string to find.

F.5 Test Case

In this test case, I will show you the test case for the string that has the length of 5.

00000	00001	00010	00011
00100	00101	00110	00111
01000	01001	01010	01011
01100	01101	01110	01111
10000	10001	10010	10011
10100	10101	10110	10111
11000	11001	11010	11011
11100	11101	11110	11111

We can see the Output string increasing 1 by 1 until the string is full of 1

F.6 Space and Time Complexity

F.6.1 Space Complexity

The part has the most impact on the space complexity is the recursion part. for each recursion, our program requires position in the CPU for the process, we consider the size of the function is some constant C. Since the algorithm runs and calls recursion 2^n(including the first time call which consume n space for the length of the first string) for 2^n combinations of the binary string, we get the total of space require by the algorithm:

$$n + (C + C + C... + C)_{2^n-1} = \sum_{i=0}^{2^n-1} (C) + n$$
$$= C * (2^n - 1) + n = C * 2^n + n - c$$

Because the algorithm runs the same for any case of n greater than 0. We can conclude that the algorithm has the space complexity of:

$$\Theta(C * 2^n + n - c) = \Theta(C * 2^n) = \Theta(2^n)$$

F.6.2 Time Complexity

Using the same reasoning as we used for the space complexity, we can still pointing out the time complexity of the algorithm. The function itself without the recursion part consist of 2 condition operator, 1 display command, 2 initialize times, all of them cost a constant time we call C to run, a for loop condition with 1 condition check, in the condition check there are some operator which cost a constant C1 time to run. We got the cost of each time we call the function:

$$C + (C1 + C1 + ... + C1)_{n/2} = C + \sum_{i=0}^{(n-1)/2} (C1) = C + n/2 * C1 \qquad (1)$$

The reason why the program only run n/2 in the for loop because the break condition will stop the function if the condition is satisfy, but most of the time the function doesn't run n time but less than. So I take the average case of the loop which is n/2. Since each function will give one binary string cost (1) time. There are 2^n strings that we need to display. So, the cost of the algorithm is:

$$(C+n/2*C1)*2^n \tag{2}$$

Now we using the big-Theta notation for (2) since for all cases of n, the algorithm runs the same. So the running time of the algorithm is:

$$\Theta((C + (n/2) * C1) * 2^n) = \Theta(C * 2^n + (n/2) * C1 * 2^n)$$
$$= \Theta((n/2) * C1 * 2^n) = \Theta((n/2) * 2^n) = \Theta(n * 2^n).$$

G Factorial Number

G.1 Problem

Factorial has many applications in our life, especially when it comes to statistics and probability, machine learning, bunnies, ...

G.2 Purpose of the Function

Because of the need to find a factorial. I represent an algorithm that help to find it by using recursion.

G.3 The Idea Approaching the Solution

When saw the problem in the homework, I knew the problem is nearly mathematic problem so I immediately checked the rule of the factorial through the Internet. After knew the rule, I jump right into the problem. My first solution was to use a loop to get the factorial number, but because the homework required us to use the recursion version of it not the for loop, so I tried to changed the loop to the recursion but I still decided to keep some important that a factorial must have in the loop(stop when the number is 0). That's why the algorithm is most likely a recursion but act like a loop.

G.4 Pseudo Code

The pseudo code below is represent the algorithm finding the factorial numbers using recursion notation.

Algorithm Factorial(n)
INPUT: the factorial number
OUTPUT: the result of the factorial
if (n = 0) then return 1

return $n * factorial(n-1)$

Explanation:

We've known that the factorial is mean multiply a the natural numbers from that number to 1:

$$\forall n \geq 0, n! = \prod_{i=1}^{n}(1) = n * (n-1) * (n-2)... * 1$$

We notice that the part behind n is actually factorial of (n-1):

$$n! = n * (n-1)!$$

The process continues until we come to 1 which is the end of the algorithm(note that factorial of 0 is 1). The if condition is to make sure the algorithm will stop when it reach 0 instead of endlessly recursion and get stack overflow. Every time it call recursion, it will multiply with the result got before, the process continue until we get 0 then the algorithm return to us the result of the factorial number.

G.5 Test Case

We assume that all of the test cases is valid(input is natural number of $n \geq 0$)

We take the input is: 5

The Output is: 120

This is true because:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

We take the input is: 30

The Output is: 1409286144

G.6 Space and Time Complexity

G.6.1 Space Complexity

Like many other recursion algorithms I have talked about before, the recursion version of the algorithm finding the factorial number depends on the depth of the recursion that it can have. Given he factorial number is n, the function will recursively call the function recursion with n-1, then n-2, again and again until it reaches 0 then the algorithm returns 1. For each recursion call, the space required for storage tasks is a constant C. And because the number of recursion is n, we have that the space required by the algorithm is:

$$C * n$$

Using the big-Theta notation we can give the asymptotic space complexity of the algorithm is $\Theta(C * n) = \Theta(n)$

G.6.2 Time Complexity

For each time we call the function in the recursion, the function do 3 things: check if the factorial number is 0 to return 1 and the return n*factorial(n-1) meaning that when be called, so the total is 3 (check if condition, return, multiply operator) those operator require a constant time C to finish. Since the algorithm of the factorial be called by n times so the algorithm needs to run n times of C to finish.

$$\forall n \geq 0, C \geq 1, \text{ The Run-Time of the function is: } C * n$$

Because for every n in the natural number set, the algorithm run time are still C*n with n is the factorial number. Using the Big-Theta notation we get the run time of the algorithm for all case: $\Theta(C * n) = \Theta(n)$.

H Sorted Array

H.1 Problem

When getting an array of integer, float number, char, string, etc... because of the benefit that an sorted array brings to us in many ways, one of those benefit is that we can use the binary searching algorithm that can helps us to find a value with time complexity almost like a constant function when the number of particles in the list increases. We need to know that we use searching in our daily life more frequently than sorting, one example is that in the supermarket, number of people trying to find a good is greater than the employee sorting and place the good in the supermarket. So an searching algorithm with time complexity of O(lg2) is definitely more useful than the naive searching O(n) running time, but the down side of it is that it require the array to be sorted before searching task is executed. So our need right now is to find the answer that the array we are talking about is sorted or not. Knowing an array is sorted or not might help us saving the CPU for other task rather than just "if I see an array, I sort it anyways" because this is not a great idea, especially when an array can have the number of member up to a million or more, if we sorting without knowing the array is need to sort or not, we can accidentally sorting an array which has been sorted and it's not a great way to deal with it.

H.2 Purpose of the Function

For all the reasons mentioned above, it is necessary to have an algorithm to find out if the array is sorted. In this section, I will represent the checking function that can determine whether the array is sorted or not by using recursion(although we can just use the loop notation for a greater space coefficient).

H.3 The Idea Approaching the Solution

The first time I encountered the problem, my first thought was to check all the array if there is any position in the array that breaks the rule(since all the member increasing/ decreasing array always bigger/ smaller than the member in front of it). If not we return the array is sorted; otherwise we return the array is not sorted. Turn out it's worked, even though I have done the check using the loop notation before, but this time I using the recursion notation and it's a big difference. I using the recursion of the function replace for the loop and it can work well even though it no match with the loop notation at the space complexity(the loop notation only cost around O(1) space).

H.4 Pseudo Code

The Pseudo code I represent in the following is the algorithm for the function to decide whether the array is sorted or not using recursion.

Algorithm IsSorted(array, index)
INPUT: an array contains information that needs to be checked whether is sorter or not, the index for the current position of the array. Start from the last position of the array.
OUTPUT: decide whether the array is sorted or not.
if (index = 0) then return the array is sorted.
if (array[index-1] > array[index]) then return the array is not sorted
return IsSorted(array, index-1)

Explanation:

Checking whether the array is sorting or not means that we need to check if there is any position does not follow our rule. Like this algorithm, it tries to find if there is any position in the increasing array that has the front value greater than the value behind it; if that happens, the rule of increasing array is broken, and the array is unsorted. That is what the second-if condition tells us. The main idea of the recursion is to move to the left by 1 cell and check that the cell satisfies the condition until we encounter the head of the array or we find that there is a value that breaks the rule. When we encounter the head of the array, it means that there is no position in the array that breaks the rule of the increasing array. So we can conclude that the array is sorted.

H.5 Test Case

In this example, I will use the array of 10 value to show you the output

Input:
[1,2,3,4,5,6,7,8,9,10]

Output: the array is sorted

When the algorithm begins, the first position it check is the last value. Since the last value has the index different from 0 then it will compare the value of 10 with the value on the left of it which is 9, because $9 < 10$ so the rule is true then it move to the next position. The process repeat until the it encounter the first value which has the index of 0 then the array is sorted so it return the result: the array is sorted.

Input:
[1,2,3,4,2,5,6,7,8,9]
Output: the array is not sorted

Familiar with that, the process begins at the last value in the array, it check through one by one and move to the left. But when the function reaches the value of 2, it compares with the value in front of it which is 4, the case happens when $4 > 2$, the rule has been broken, so it return the array is not sorted.

H.6 Space and Time Complexity

H.6.1 Space Complexity

The main part that has the greatest affect on space complexity of this algorithm is the recursion part. In any case, we recursively call the function itself until it comes to the end of the array or encounters a position where the value does not follow the rule. And by that we can conclude that: given the array has n value, the worst case of this algorithm is O(n) due to the array being sorted so it need to recursively call itself n times. The best case is for the first time calls the function, it finds out the (n-1)-th value breaks the rule, in this case the space complexity is O(1). For the average case, the function requires O(n/2) which is O(n) space complexity.

H.6.2 Time Complexity

The time complexity of the algorithm is identical with space complexity, since the algorithm each time it moves to the new position, it needs to do some operator(2 if conditions most, 1 recursion call or return command) which cost a constant time. We can use the same reasoning for the space to conclude that the time complexity of the algorithm for the worst case is O(n), best case O(1), average case O(n/2) which is O(n).