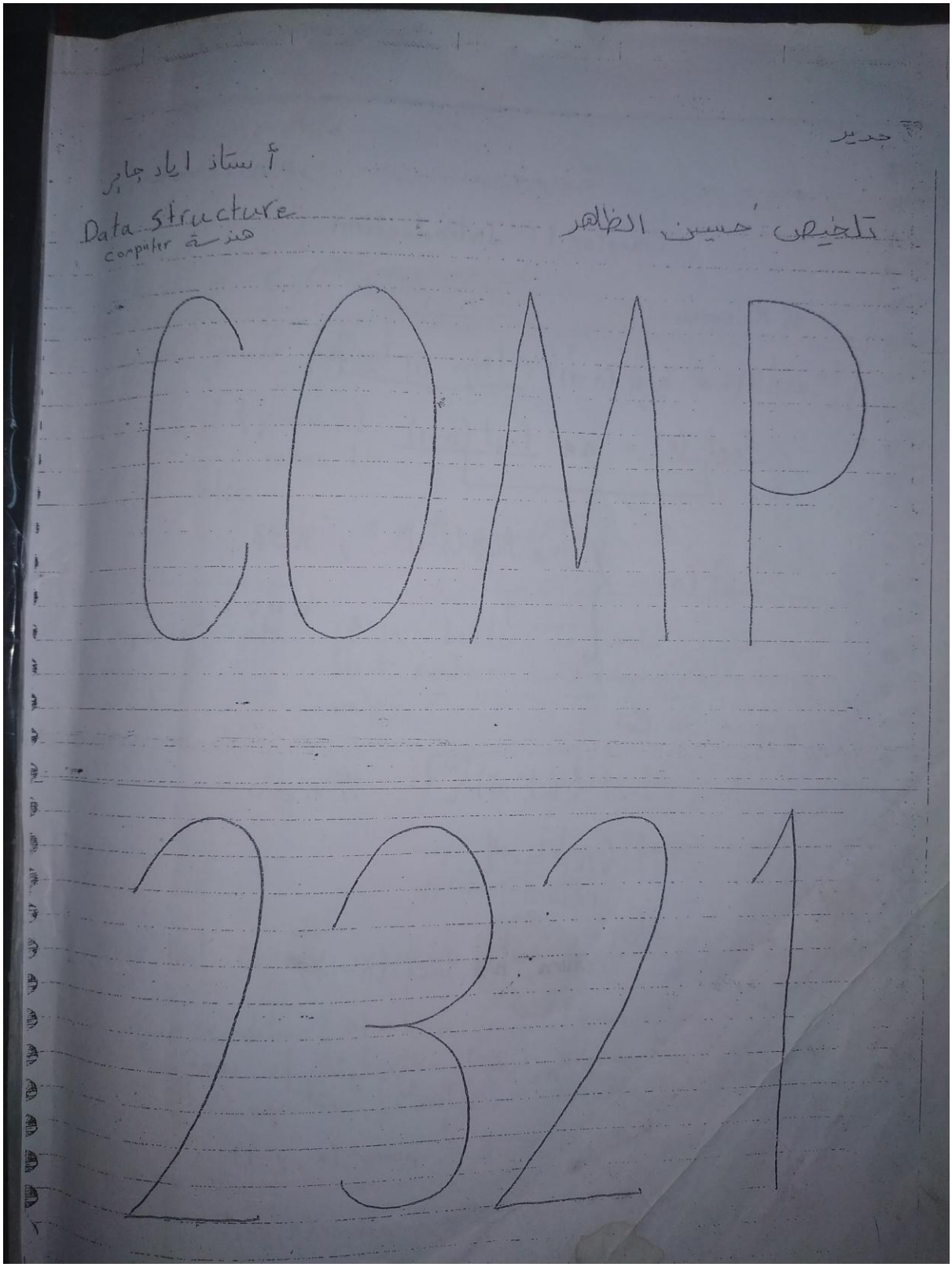


Data structure

Chapter 1: Introduction

* Recursion:

ex1: $n! = n * (n-1)!$; $0! = 1$

fact (n) = n * fact (n-1)

$$fact(n) = \begin{cases} n * fact(n-1) & , n > 0 \\ 1 & , n = 0 \end{cases}$$

program in C:

```
long fact (int n){      // n >= 0
    if (n==0)
        return 1;
    else
        return n * fact (n-1);
}
```

ex2: $x^y = x * x^{y-1}$; $x^0 = 1$

power (x, y) = x * power (x, y-1)

$$power(x,y) = \begin{cases} x * power(x,y-1) & , y > 0 \\ 1 & , y = 0 \end{cases}$$

program in C:

```
double power (float x , int y){   // y is posttive int
    if (y==0)
        return 1;
    else
        return x * power (x , y-1) ;
```

important note: it is better to avoid comparing float numbers and compare int numbers only.

ex: if (x == pow (sqrt(x) , 2))

may be false beacause of approximations!

Note that in any recursion , two conditions must be satisfied:

1) Stop condition (0! = 1 in factorial example).
2) The name of the function must be in the both sides of the equation (fact(n) = n * fact(n-1)).

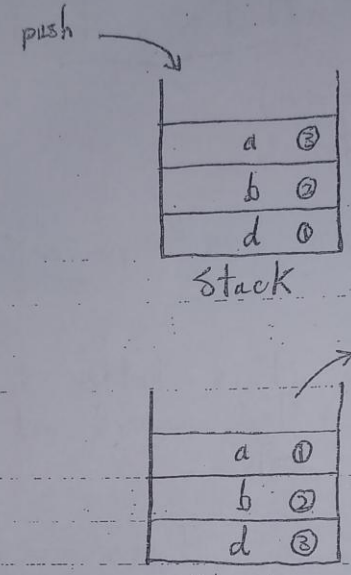
* How Functions Acually Executed in the Computer?

```
main () {
   D ();
}

D () {
   B ();
}

B () {
   A ();
}

A () {
  // code
}
```



when a function is called, its addr will be pushed to the stack, so th will be pushed respec (d then b then a

after all functions is called and their addresses are in the stack, the addresses will be popped one by one and the fu tions will be execu (a then b then d

FILO: First In Last Out

where a is the address of function A, b is the address of function B and d is the address of function D.

This can describe how recursions executed.

note: a stack overflow error will occur, if:

1) The stopping condition in a recursion is wrong.
2) The stopping condition isn't wrong, but it is too far and the stack filled before reaching it.

Note that we should use recursions only if we can write an optimal code that includes all possible cases.

Types of Variables:

1) Global 2) Local 3) Scope

Note that we should use global variables as less as we can and only if needed, since they reserve memory from the beginning of program execution until its terminate.

note: #include < x.cpp >
#include "C:\\ Folder1 \\ Folder2 \\ x.cpp"

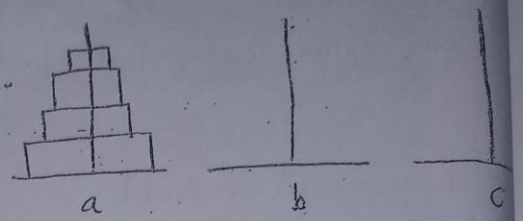
the difference between those two sentences is that the first one will search for the file "x.cpp" in the include directory, while the second one will search for it in the given directory (or in the current directory if only the file name was given).

note: the character reserves 1 byte in C, while it reserves 2 bytes in Java.

1 byte = 8 bits (ASCII), one for parity and 7 for the character, that it we got 2^7 characters (0-127) (english only).

2 bytes = 16 bits (Unicode), 2^{16} characters (first 128 for english and the other bits are for other languages).

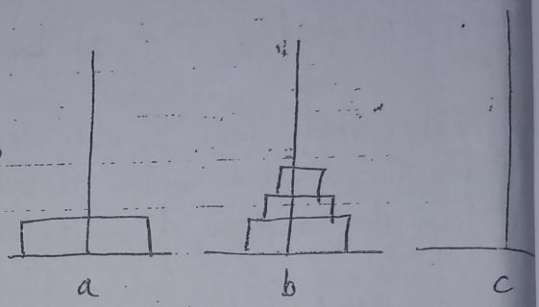
* Towers of Hanoy:



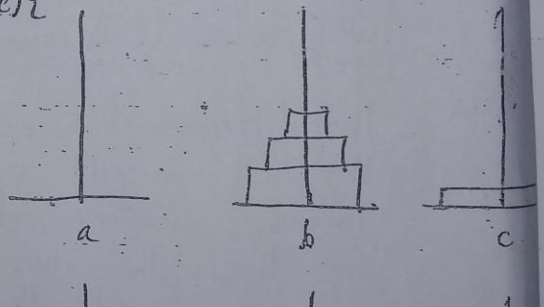
Hanoi (n, a, b, c)

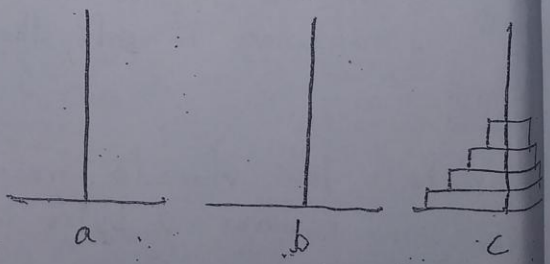
$$= \begin{cases} \text{Hanoi}(n-1, a, c, b) \\ a \longrightarrow c \\ \text{Hanoi}(n-1, b, a, c) \end{cases}, \quad n \neq 0$$

nothing , $n = 0$




```
void Hanoi (int n, int a, int b, int c){
 if (n > 0){
    Hanoi (n-1, a, c, b);
    printf ("%d → %d", a, c);
    Hanoi (n-1, b, a, c);
 }
}
```





note: 1) number of moves $= 2^n - 1$

2) number of calls $= 2^{n+1} - 1$

Can number of calls = number of moves? answer: yes, see this code

```
void Hanoi (int n, int a, int b, int c){   // n>0
    if (n==1)
        printf ("%d → %d", a, c);
    else{
        Hanoi (n-1, a, c, b);
        printf ("%d → %d", a, c);
        Hanoi (n-1, b, a, c);
    }
}
```

This code will make number of calls = number of moves = $2^n - 1$, because it will not call the function when n=0 and it will stop when it reaches n=1.