-42

The hitchhiker's guide to staying alive at `Wethinkcode_` *South Africa*

Written by: Luyanda Mncube

"The answer to all things in Bootcamp is -42"

-e5r8p4

This is a notebook on the C Programming language I wrote whilst attending the `Wethinkcode_` *bootcamp in Johannesburg South Africa in preparation for daily exercises to help fellow bootcampers who want to avoid the dreaded "0/100" Fail screen. All improvements/suggestions welcome at:*

`lmncube@student.wethinkcode.co.za`

Lesson 1

- C does not support object oriented programming
- C does not support information hiding (no support for polymorphism, encapsulation, and inheritance)
- C does not allow functions to be defined inside structures.
- C uses functions for input/output. For example `scanf` and `printf` .
  (FORBIDDEN by 42 Norm)
- C provides `malloc` and `calloc` functions for dynamic memory allocation, and `calloc` for memory de-allocation.
  (FORBIDDEN by 42 Norm)
- C does not provide String or Boolean data types. It supports primitive & built-in data types.
- C supports only Pointers (no references)

HOW TO COMPILE IN UNIX (using gcc)

Moulinette uses `-Wall -Wextra Werror` when compiling, and uses `gcc`. All warnings in the compiler basically turn to errors
So anything that even looks like it will crash will cause you to get a fail grade ☹

Example 1: Compile command

```
$> gcc -Wall -Wextra -Werror test.c
$> ./a.out
```

You can also compile multiple C files for the same program.

```
gcc -O0 -Wall -Wextra -Werror -ansi -o my_executable my_code.c
```

PRO TIP: Make an alias

To avoid having to type this long command at each compilation, make yourself an alias!
Open the ~ / SHELLrc file (replacing SHELL with sh, csh, tcsh, bash, 42sh, ...)
Add a line of type:
`alias mycc 'gcc -O0 -Wall -Wextra -Werror -ansi'`
Then type
`source ~ / SHELLrc`
And finally, to compile, do
`mycc -o my_executable my_code.c`

Lesson 2 – Variables & Character handling

Variables are little boxes that help us store stuff in programming.

To know the ASCII values of each letter, we use the ASCII table that can be accessed through `man ASCII`.

```
48 - 57 | '0' - '9'
65 - 90 | 'A' - 'Z'
97 - 122 | 'a' - 'z'
```

Example: The letter 'F' is set to ASCII 70 while the letter 'f' is ASCII 102.
Other non-visible characters that may be useful:

```
00 | '\ 0' | NULL, 0              //Usually at the end of strings, so our
                                  compiler knows when to stop reading a
                                  string
07 | '\ a' | bell, beep
08 | '\ b' | backspace
09 | '\ t' | tabulation
10 | '\ n' | Return to the line      //Skips to next line
11 | '\ v' | Vertical tabulation
13 | '\ r' | Come back to the beginning of the line
```

To display a letter, we will use this small function:

```
#include      <unistd.h>

void    ft_putchar(char  c)
{
  write (1, &c, 1);
}
```

You can send a string to a function by using double quotes (“ ”) my string of characters \ n ".
Single quotes (') are used for single characters.

PRO TIP: Understanding NULL or ‘\0’

NB: NULL in c languages is for NOTHING. It is not the same as zero. If you return 0 in a function it has a meaning but if you return NULL it can crash your program. NULL is typically more useful in things like arrays and memory allocation. (i.e. set all values of an array to NULL before use so that if one result is stored as ‘0’ you can be aware of the fact that a value went in that index, the value just happened to be zero)

Lesson 3 - Useful (And Useless) Functions

Write

- call to system
- uses the `#include <unistd.h>` directive
  Remember, numbers still fall on the asci table. So don't try convert, treat them the SAME as you would a normal char.

in the form

```
write(fd, buf, nbytes);
```

fd	It is the file descriptor which has been obtained from the call to open. It is an integer value. The values 0, 1, 2 can also be given, for standard input, standard output & standard error, respectively .
buf	It points to a character array, which can be used to store content obtained from the file pointed to by fd
nbytes	It specifies the number of bytes (essentially characters) to be written from the file into the character array.

To display words to screen

```
#include <unistd.h>

int main(void)
{
    if (write(1, "Hello World!\n", 13) != 14)
    {
        write(2, "There was an error writing to standard out\n", 44);
        return -1;
    }

    return 0;
}
```

PRO TIP: Some useless information

What is a descriptor? (Used for Manipulation of files)
In simple words, when you open a file, the operating system creates an entry to represent that file and store the information about that opened file. So if there are 100 files opened in your OS then there will be 100 entries in OS (somewhere in kernel). These entries are represented by integers like (...100, 101, 102....). This entry number is the file descriptor. So it is just an integer number that uniquely represents an opened file in operating system. If your process opens 10 files then your Process table will have 10 entries for file descriptors.

Printf

(FORBIDDEN by 42 Norm but… It's super useful for testing)

- Provides FORMATTED output
- Used with a number of specifiers, use them when you know what type of variable you are outputting
- uses the `#include <stdio.h>` directive
- In the const char part, use `\n` if you need to output stuff after printing

in the form

```
#include <stdio.h>

int main()
{
   printf ("Characters: %c %c \n", 'a', 65);
   printf ("Decimals: %d %ld\n", 1977, 650000L);
   printf ("Width trick: %*d \n", 5, 10);
   printf ("%s \n", "A string");
}
```

A list of some specifiers

d or i	Signed decimal integer	392
u	Unsigned decimal integer	7235
f	Decimal floating point, lowercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed.	

Scanf

(FORBIDDEN by 42 Norm)

- The scanf function allows you to accept input from standard in, which for us is generally the keyboard.
- uses the `#include <stdio.h>` directive
- The scanf function uses the same placeholders as printf:

int uses **%d**
float uses **%f**
char uses **%c**
character strings (discussed later) use **%s**

Void type

- is a type, goes for BOTH function parameters & function returns
- So, a void pointer in c is one without a type, but this actually doesn't matter (in the example, a VP function with a VP variable is created to act as a place holder to convert a char to an int)
- Therefore, void pointers are POLYMORPHIC
- In c++, when void is used as a parameter, it's basically casting a variable (i.e void void_one) but in c it's much simpler, just use void (i.e void) as a parameter

Ternary operators *(FORBIDDEN by 42 Norm)*

- Different from Boolean operators because they have three sides instead of two.
- shortens and replaces need for if else
- if condition is true to left of question mark, returns first operand, if condition is false, returns second operand

In the form of:
(condition ? return_true : return_false)

Example 1 – Shortening and if else statement using ternary operators:

```
int opening_time = (day == SUNDAY) ? 12 : 9;
```

Instead of:

```
int opening_time;

if (day == SUNDAY)
    opening_time = 12;
else
    opening_time = 9;
```

Ternary operators can also be parameters for other functions (since ternary operators have specified output)

```
#include <stdio.h>

main()
{
   int a , b;
   a = 10;
   printf( "Value of b is %d\n", (a == 1) ? 20 : 30 );
   printf( "Value of b is %d\n", (a == 10) ? 20 : 30 );
}
```

Typedef (Useful for when we create classes in C)

alias for a type name
signified with _t (i.e. cards_t)
Not much to it, just saves you time cause pointer definitions or even variables can take a while to type

Lesson 4 – Arrays

Arrays

- Little bookshelves that help you store information. More like badass variables
- Same as in C++, same logic and use
- Difference here is that the functions new and delete for allocating and de-allocating memory are now malloc and delete
  (FORBIDDEN by 42 Norm)

Process for allocation
1. Declare pointer variable to point to allocated heap space
2. Call to allocate the correct number of space in memory
3. Initialize array (If you can)
4.

Example 1 – How to dynamically allocate in C

```
//declare a pointer variable to point to allocated heap space
int    *p_array;
double *d_array;

//call malloc to allocate that appropriate number of bytes for the array

p_array = (int *)malloc(sizeof(int)*50);       // allocate 50 ints
d_array = (int *)malloc(sizeof(double)*100);  // allocate 100 doubles

// always CHECK RETURN VALUE of functions and HANDLE ERROR return values

if(p_array == NULL)
{
  printf("malloc of size %d failed!\n", 50  exit(1);
}
```

Passing array as a parameter
When passing an array as a parameter, this:

```
      void arraytest(int a[])
```

means exactly the same as:

```
      void arraytest(int *a)
```

Dynamically changing the size of an array
(FORBIDDEN by 42 Norm… For Now)

```
   /* Initial memory allocation */
   str = (char *) malloc(15);

   /* Reallocating memory */
   str = (char *) realloc(str, 25);
   free(str);
```

Lesson 5 – Pointers

- when you declare a pointer, it's not "a pointer of the type int" but rather "a pointer of the type point to int"
  So a pointer is Geff pointing to the box he will put your body in.
  A Variable is Geff storing your body in the box.
- So it's technically not a variable. Instead of making space in memory (stack), it just waits and points to it until you decide to use it

key to understanding pointers?

1. The address of "&" operator changes what the pointer is pointing to
   int *ip;
2. the pointer ip only points to something (point to the same thing that variable x is pointing to)
   ip = &x;

   OR

2. you assign the pointers value to something it should point at
   *ip = *x;

- For arrays, *ip=2; assigns 2 to the first element, because the pointer points to the first element of the array
- If you increment the pointer using ++ip; , it will point to the second element
- after incrementing, you can use ip=3; to assign to the second element

PRO TIP: Increment AND use a pointer at the same time

you can increment AND use a pointer at the same time using *(++ip) = 4;
This is useful for when you want to quickly store something in a pointer array

```
int ia[5];              *ip =2;
ia[0] =1;               ++p;
int *ip = ia;           *ip =3;
                        *(++ip) = 4;
```

C strings

- in the c language, strings are a special case of an array (unlike in c++ where they are objects)
- they are an array of characters terminated by a zero value
- char s[ ] = {'s','t','r','i','n','g',0}
  is the same as
  char s[ ] = "string".
  Use the second, it's more common

You can use a pointer as the i iterator in a loop when outputting an array. The pointer actually even replaces the need for s[i] entirely (example: use a character pointer as an iterator/i)

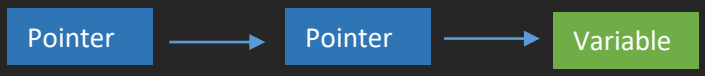
```
#include <stdio.h>

int main(int argc, char ** argv)
{
   Char s[ ] = “string”;
   Printf(“String is: %s\n”,s);
   For (char * cp = s; *cp; ++cp)
   {
         Printf(“Char is: %c\n”, *cp);

   }
   Return (0);
}
```

Pointer to a pointer (Pass by reference)

A pointer to a pointer is a form of multiple indirection, or a chain of pointers




```
#include <stdio.h>
int main(void)
{
      int **a;
      **a = 42;
      Printf(“%i\n”,**a);
}
```

The code above actually prints “42

PRO TIP: So why use pointer to a pointer?

In C, double pointers are more common, since C does not have a reference type. Therefore, you also use a pointer to *"pass by reference."*. Where “ & ” is used in a parameter type in the C++, “ * ” would be used in C (and dereferenced when manipulating the parameter). A function is not able to change the actual parameter’s value.One example:

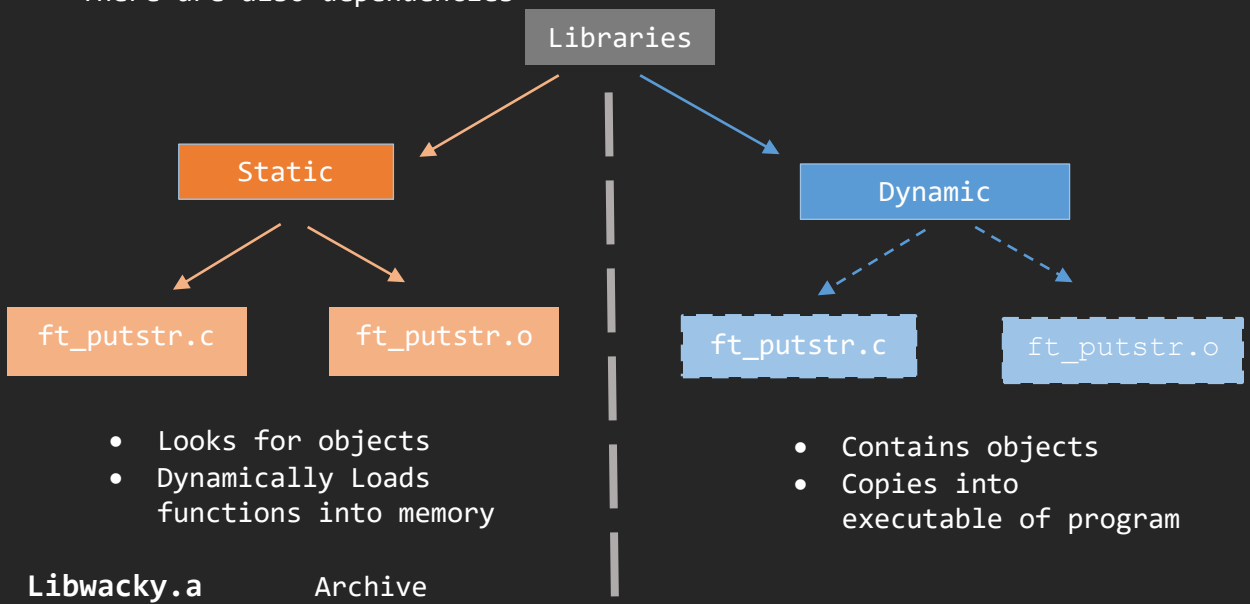
```
void SomeFun1(int *);
void SomeFun2(int **);
int i;int *ptr = &I;          //when you want value of ptr should remain unchanged,
                              but you want to change only value of i, use,

SomeFun1(int *)

SomeFun2(int **);             //when you want value of ptr should change. i.e, it
                              should point to some other memory other than i, use,
```

Lesson 6: Libraries

- Libraries essentially bundle code/functions/objects together in order to reuse and access everything easily
- Archives describe how everything is housed
- These are further separated into Static & Dynamic/Shared Libraries
- Dependencies are created between the Program and the archive. There are also dependencies



- Looks for objects
- Dynamically Loads functions into memory

- Contains objects
- Copies into executable of program

Libwacky.a Archive
- Wacky.o Object of Archive
- Wackier.c Function of Archive

How to create Objects

- These are usually created from the .c extension, .c files which are put through the compiler are used to create object files.
- The command line or shell scripts are used to execute the different calls we will need to create object files, creating a library, creating an index in the library and even removing all the object files once they have been created

Example – Some shell script to help build up a library (with .c files located in current directory)

`lib_creator.sh`

```
#!/bin/sh
gcc . -c * c                    //Create object files from ALL .c files in current directory
ar rc libft.a *o                //Create a library by name
ranlib libft.a                  //Creates index in library
rm. * o                         //remove/delete all object files
```

- The ar tool can also help us list the files inside a library
  `ar rc libft.a *o`

Lesson 7: Arguments

Stuff for nerds.

Stuff for nerds I : Space & Time Complexity of an algorithm

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Similarly, ***Space*** complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input. This usually shows the number of times an algorithm will have to execute in the worst case scenario.
There are some common running times when analyzing an algorithm:

Big O Notation

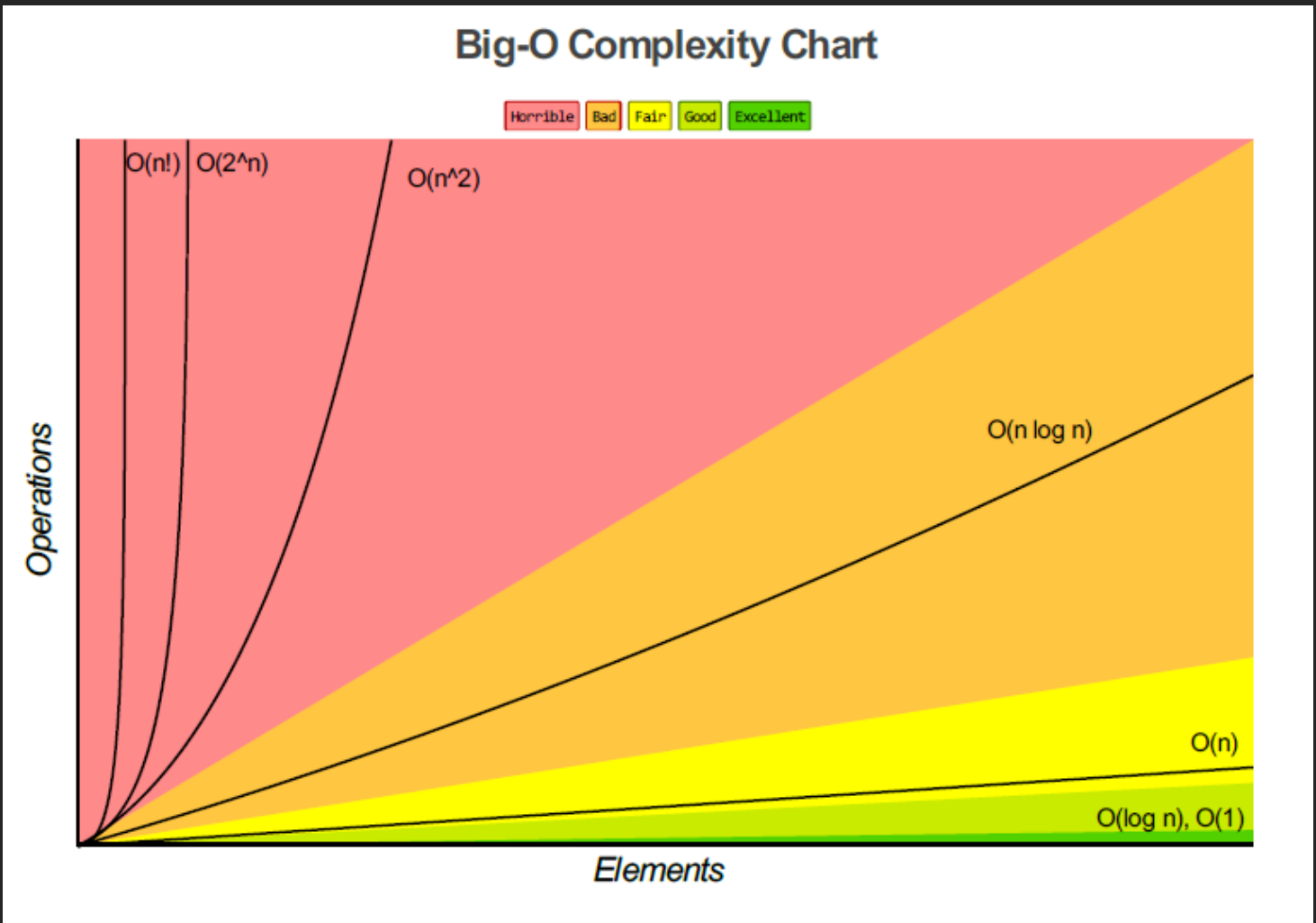
(In order of speed)

$O(1)$ *"Order of 1"*	Constant	Running time is constant, it's not affected by the input size. Example: Adding item to array.
$O(logN)$	Logarithmic	Algorithm that has running time O(log n) is slightly faster than O(n). Commonly, algorithm divides the problem into sub problems with the same size. Example: Binary search algorithm, binary conversion algorithm.
$O(N)$	Linear	When an algorithm accepts n input size, it would perform n operations as well. Example: Linear search
$O(n\ log_n)$	Linearithmic	This running time is often found in "divide & conquer algorithms" which divide the problem into sub problems recursively and then merge them in n time. Example: Merge Sort algorithm.
$O(N^2)$	Quadratic	Look Bubble Sort algorithm! Example: Bubble Sort algorithm (nested loops, swap elements)
$O(N^3)$	Cubic	Same as above
$O(2^n)$	Exponential	Time It is very slow as input get larger, if n = 1000.000, T(n) would be 21000.000. Brute Force algorithm has this running time.
$O(N!)$	Factorial	THE SLOWEST !!! Example : Travel Salesman Problem (TSP)

So if you did ft_fibonacci.c from day04 and tested it, you might have noted that it was REALLY slow, especially at higher values. This is because it has a $O(2^n)$ Time complexity (SUPER inefficient).

So typically, the more data we have, the "N-times" or "N-squared times" etc the algorithm will have to execute in the worst case scenario.

Remember, we always look at the part of the algorithm with the BIGGEST effect on the performance of the algorithm.



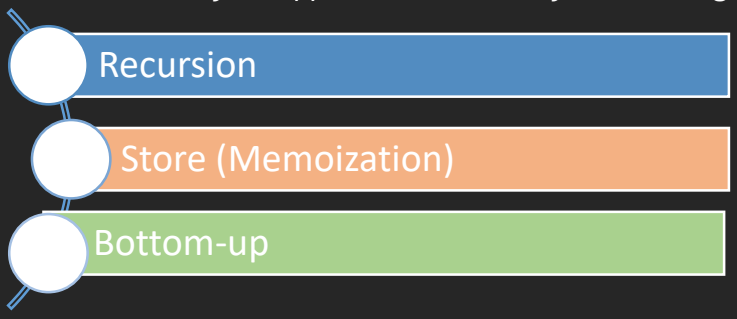
PRO TIP: TSP for nerds

- A salesman must travel between N cities (in any order) and finishes where he started
- This is an NP-hard problem
- For n cities, you have N-1 Factorial solutions
  (for only 10 cities you have over 180 000 combinations)
- Therefore this has the complexity of $\boldsymbol{O}(\boldsymbol{n}!)$

Stuff for nerds II: Dynamic Programming

(Stuff to know for when we do searches)

- Dynamic programming can make an algorithm more efficient by storing some of the intermediate results (where there are repetitive computations)
- Seems like this is used where we have super slow/inefficient algorithms (like ones that compute in exponential $O(2^n)$ or even factorial $O(n!)$ time)
- Normally 3 approaches to a Dynamic Programming problem:



Recursion

- In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.
- But while using recursion, programmers need to be careful to define an exit condition from the function; otherwise it will go into an infinite loop.

The base case = exit condition for function

Steps for writing recursive functions:

1. Define what the recursive function does in English (i.e. the prototype)
2. Think about solving the next smaller sized problem
   Recursion solves a big problem (of size n, say) by solving one or more smaller problems, and using the solutions of the smaller problems, to solve the bigger problem
3. Finally, you should deal with the base case, which is the smallest problem

```
int sum( int arr[], int n )              //1) Function prototype
{
  if ( n == 0 )                          //3) base case
    return 0 ;   // no recursive call
  else
    {
      int small = sum( arr, n - 1 ) ;    //2) solve smaller problem
      return small + arr[ n - 1 ] ;
    }
}
```

Recursion will help solve complex scenarios elegantly and most importantly, will result in readable/understandable code.

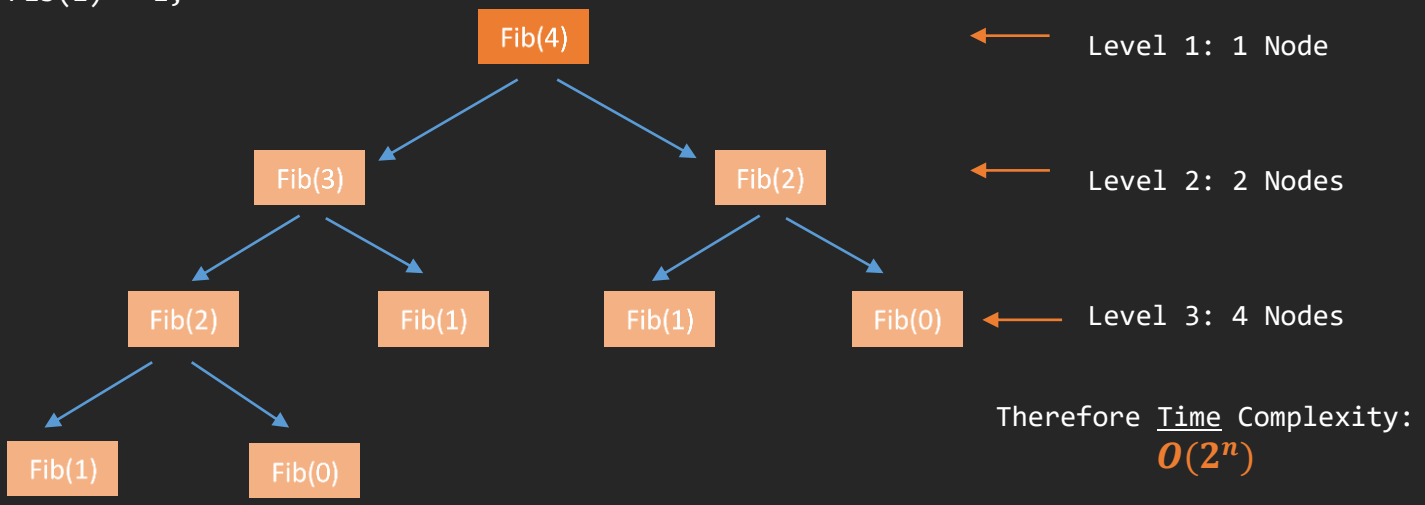
Memoization

- Simply storing the results obtained from a recursive solution
- Best example would be the Fibonacci sequence program

Example:

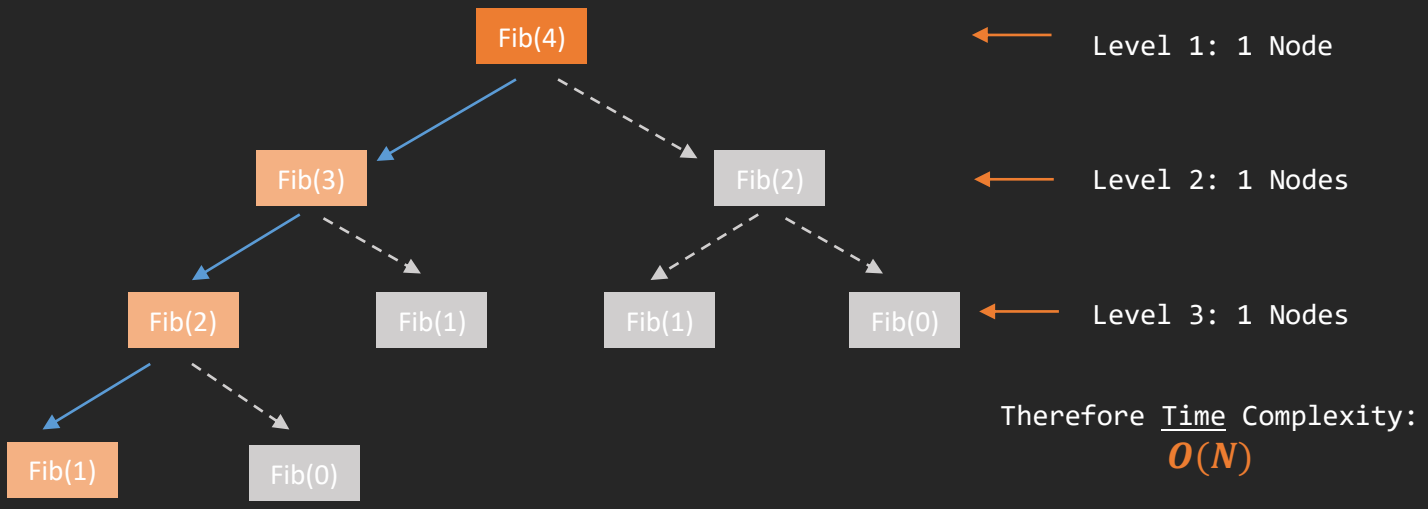
Fib(n) = Fib(n-1)+ Fib(n-2)

With the starting conditions of:
Fib(0) = 0;
Fib(1) = 1;



Therefore Time Complexity:
$O(2^n)$

A treemap of the fib(4) function (WITHOUT Memoization)



Therefore Time Complexity:
$O(N)$

A treemap of the fib(4) function (WITH Memoization)

Bottom-up

- Better logical approach, instead of recursively trying to find a solution, figure out a relationship on the sequence and store indexes in array

Algorithms for fibonacci using all 3 methods:

Recursion

```
int fib (int n)
{
      int result = NULL;
      if (n == 1)||(n == 2)
            result 1;
      else
            result = fib(n-1) + fib(n-2);
      return result;
}
```

Therefore Time Complexity:

$$O(2^n)$$

Memoization

```
int fib (int n, int *memo)
{
      int result = NULL;
      if (memo[n] != NULL)
      return memo[n]
      if (n == 1)||(n == 2)
            result 1;
      else
            result = fib(n-1) + fib(n-2);
      memo[n]= result;
      return result;
}
```

Therefore Time Complexity:

$$T \leq (2n + 1).O(1)$$

$$= O(N)$$

Bottom-up

```
int fib (int n){
int result = NULL;
      if (n == 1)||(n == 2)
            result 1;
bottom = new int[n+1];            //Define an array
bottom[1]= 1;
bottom[2]= 1;
for (int i=3; i <= n; i++)
            bottom[i]= bottom[i-1]+ bottom[i-2];
      return bottom[n];
}
```

Therefore Time Complexity:

$$= O(N)$$

Stuff for nerds III: Other algorithms

Binary search

- This is an interesting search method that relies on the property that a set is sorted
- This algorithm divides a set in two, compares the search item to the middle of the set, and decides whether it should recursively search the top or bottom of the set.
- Each search level has half the number of results therefore has a logarithmic time complexity $O(log_n)$
- This search can be iterative or recursive (prefer iterative as you can use a single parameter for a function)

Steps:

1. *While left <= right…*
2. *mid = (left + right)/2… OR mid = (left + right)/2 to prevent integer overflow*
3. *Check if x < mid (then adjust right boundary, right = mid-1)*
4. *Else adjust left boundary (left = mid+1)*

```
int binsearch(int nb)
{
      int * arr[nb] , left = 0, right = nb-1, x, mid;
      char * result;
      while (left <= right){
            mid = (left+right)/2;
            if (arr[mid] == x)
                  result = mid ;
            else if (x < arr[mid])              //Check left, adjust right
                  right = mid-1;
            else if (x > arr[mid])              //Check right, adjust left

                  left = mid+1;
      }
      result = '\0'
}
```