Algorithm analysis and design

Kurkunda Bhargavi
2020101077
UG2K20

Acknowledgement

I would like to express my special thanks of gratitude to my professor Mr.Srinathan.K as well as our Teaching assistant S Shodasakshari Vidya who gave me the golden opportunity to do this wonderful project on Algorithms, which also helped me in doing a lot of Research and i came to know about so many new things and also a wonderful time to spend on the things I wanted to know more about. I am really thankful to them .

GitHub repository link for all the programs implemented as part of this project

https://github.com/Bhargavi-hash/Anti-Theft-Alarm-IoT.git

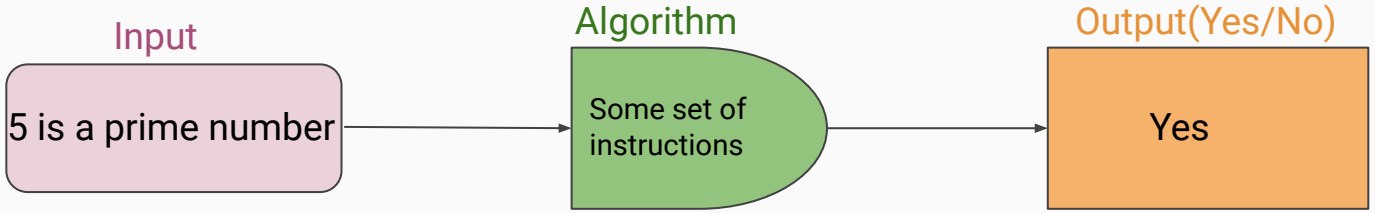
Overview of what we will be studying

1) What is an algorithm?
2) Why are the algorithms necessary?
3) What are the few earliest algorithms?
    - When were they invented?
    - How did that particular algorithm invention help them?
    - What are the better versions of those algorithms?
4) Different categories of algorithms
    - Simple recursive algorithms
    - Backtracking algorithms
    - Divide and conquer algorithms
    - Dynamic programming algorithms
    - Greedy algorithms
    - Branch and bound algorithms
    - Brute force algorithms
    - Randomized algorithms
5) Some widely used algorithms in real life.

What is an algorithm?

An **algorithm** is a procedure or formula for solving a problem or accomplishing a task, based on conducting a sequence of specified actions.

In mathematics and computer science, an algorithm is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of specific problems or to perform a computation.



Why are algorithms necessary ?

Before discussing about the necessities of the computational algorithms let us look at them from the normal day to day life perspective.

For examples, let us say that we want to travel few places but we have very less time. Then the clever thing to do is to travel to the nearest places first and then move to the next nearest place until we have covered all the places.

This journey will take least amount of travel time.

Let us say the time taken for this shortest journey is ‘ T ‘.

Now if we choose any other path except the previous path, then either we will not finish our journey within the given time or we will take more time than ‘ T ‘.

The logic of travelling to the nearest places first helps us not only to save time, but also to travel less distance.

Therefore, we can say that this is an algorithm which makes our journey easier.

Why are algorithms necessary in computer science ?

- Algorithms are used and implemented to give most ideal option for accomplishing a task.
- In computer science algorithms are used to:

    1) Improve efficiency of a computer program

    2) Proper utilization of resources

1. Improve efficiency of a computer program

When it comes to programming, efficiency can be used to mean different things.

One of them is the **accuracy** of the software. With the best algorithm, a computer program will be able to produce very accurate results.

Another way of looking at the efficiency of the software is **speed.**

An algorithm can be used to improve the speed at which a program executes a problem. A single algorithm has the potential of reducing the time that a program takes to solve a problem.

Example:

If we are to search a given element in a sorted list, then we can either start from the starting element and move through the list till we find the given element.(Linear search)

Another algorithm is to check the middle element, and if the given element is greater than the middle element we compare it with the middle element of the second half and continue till we find the given element.(This is binary search which we will be discussing in detailed later)

Clearly the second algorithm is efficient(less time consuming) because it doesn't travel through all the elements.

2. Proper utilization of resources

A typical computer has different resources. One of them is **computer memory**.

During the execution phase, a computer program will require some amount of memory.

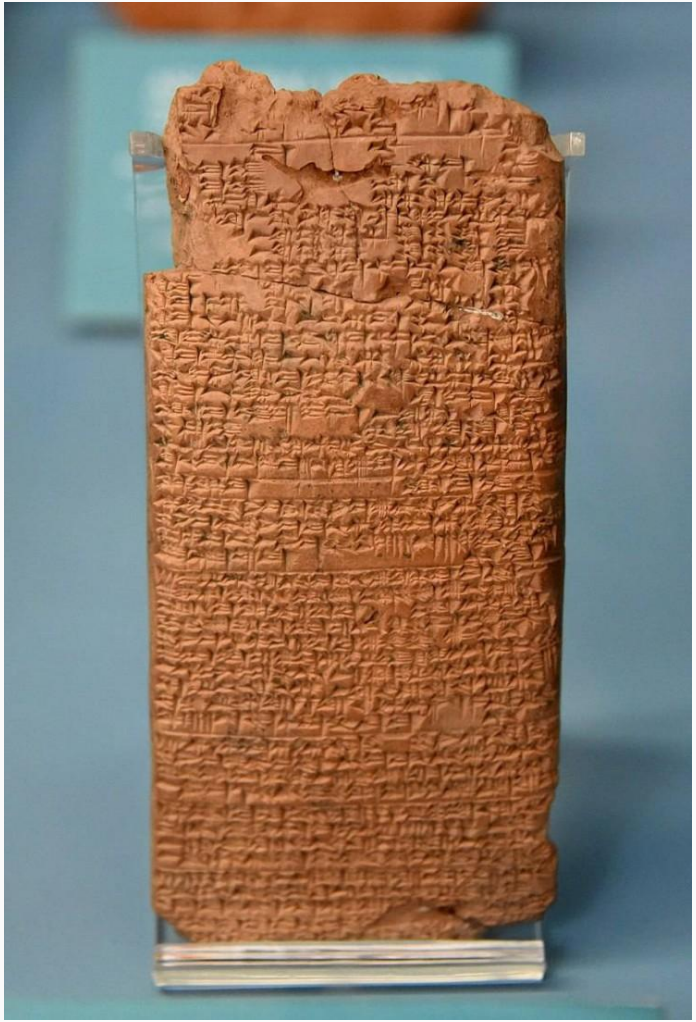
Some programs use more memory space than others. The usage of computer memory depends on the algorithm that has been used.

The right choice of an algorithm will ensure that a program consumes the least amount of memory.

Apart from memory, and the algorithm can determine the amount of **processing power** that is needed by a program.

History of algorithms

Timeline



An 18th Century BC medical tablet

Source: *Osama Shukir Muhammed Amin FRCP/Wikimedia Commons*

Although there is some evidence of early multiplication algorithms in Egypt **(around 2000-1700 BC)**, the oldest written algorithm is widely accepted to have been found on a set of Babylonian clay tablets that date to around **1800-1600 BC.**

Their true significance only came to light around **1972**, when computer scientist and mathematician Donald E. Knuth published the first English translations of various cuneiform mathematical tablets.

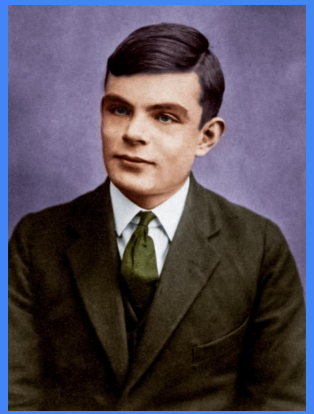
Medieval Period

- 1700 - 2000 BC
- 1600 BC
- 300 BC
- 200 BC
- 820
- 850
- 1025

1700 - 2000 BC:	Egyptians develop earliest known algorithms for multiplying two numbers
1600 BC:	Babylonians introduced earliest known algorithms for factorization and finding square roots
300 BC:	Euclid’s algorithm
200 BC:	The Sieve of Eratosthenes
820:	Algorithms for solving linear and quadratic equations (by Al-Kawarizmi)
850:	Frequency analysis algorithms (by Al-Kindi)
1025:	Algorithm for determining the general formula for the sum of any integral powers, which was fundamental to the development of integral Calculus (by Alhazen)

Before 1940

- 1540
- 1545
- 1614
- 1936
- **Turing machine**



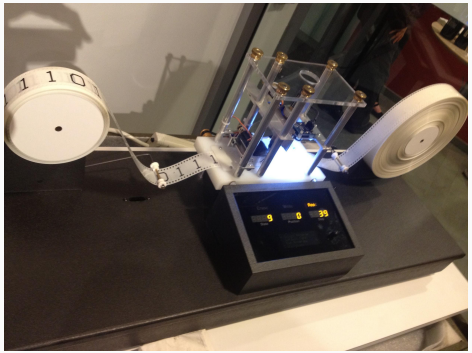
Alan Turing

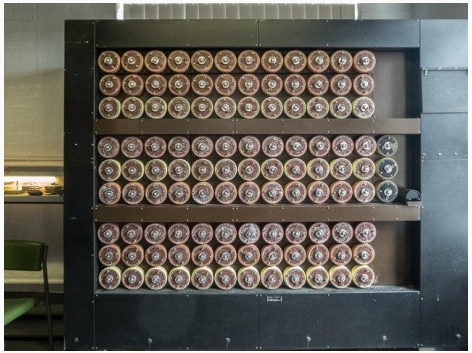
1540: Lodovico Ferrari discovered a method to find the roots of a quadratic polynomial

1545: Gerolamo Cardano published Cardano's method for finding the roots of a cubic polynomial

1614: John Napier develops method for performing calculations using logarithms

1936: **Turing machine**, an abstract machine developed by Alan Turing, with others developed the modern notion of *" algorithm "*.





Turing machine

- Turing machine is a mathematical model of computation that defines an abstract machine that manipulates the symbols on a strip of tape according to the table of rules.

- A Turing machine is a general example of a central processing unit (CPU) that controls all data manipulation done by a computer, with the canonical machine using sequential memory to store data.

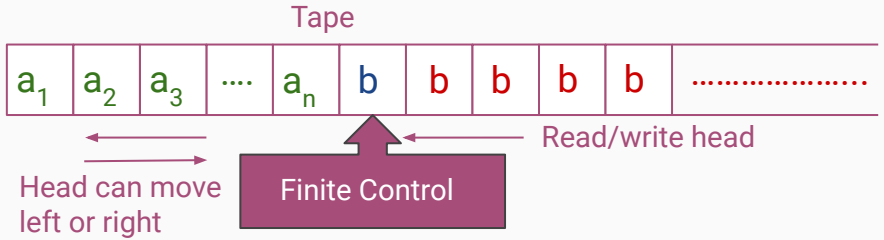
- **Working:**

1. Turing machine operates on infinite memory tape divided into discrete cells.
2. The machine positions it's **"head"** over a cell and **"reads"** or **"scans"** a symbol.
3. Then based on the symbol and machine's own present state in a "finite table" of user-specified instructions the machine:

    a. Writes a symbol in the cell (digit or alphabet)
    b. Then either moves the tape one cell left or one cell right
    c. Then based on the observed symbol and machines own state in the table either proceeds to another instruction or halts the computation.

A turing machine is expressed as a 7-tuple **(Q, T, B, ∑ , δ, q0, F)** where:

- **Q :** Finite set of states
- **T :** Tape alphabet (symbols which can be written on the tape)
- **B :** Blank symbol (every cell is filled with B except the input alphabet initially)
- **∑ :** Input alphabet (symbols which are part of input alphabet)
- **δ :** Transition function. Maps **Q × T → Q × T × {L, R}**
  Depending on the present state and present tape alphabet pointed by head pointer , it will move to a new state, may or may not change the tape symbol and move the head pointer to either left or right.
- **q0:** Initial state
- **F :** Set of final states. If any state of F is reached, input string is accepted.



Let us construct a Turing machine with given specifications

- Q : {q_0, q_1, q_2, q_3} where q_0 is the initial state
- T : {0, 1, X, Y, B} where 'B' represents a blank
- ∑ : {0, 1}
- F : {q_3}
- Transition function **δ** is represented as below:

	0	1	X	Y	B
q_0	(q_1, X, R)			(q_3, Y, R)	
q_1	(q_1, 0, R)	(q_2, Y, L)		(q_1, Y, R)	
q_2	(q_2, 0, L)		(q_0, X, R)	(q_2, Y, L)	
q_3				(q_3, Y, R)	**HALT**

Let us see how the transition works for input string 0011

Initial state of tape

B	0	0	1	1	B

Initial state is q_0 and initially head points to 0

B	**0**	0	1	1	B

Head

Now according to the transition table $\delta(q_0, 0) = (q_1, X, R)$: This means that now the system enters state q_1, replacing the input of the present cell with **X** and moves the head to the **right**.

State q_1 :

B	X	**0**	1	1	B

Head

Now according to the transition table $\delta(q_1, 0) = (q_1, 0, R)$: This means that now the system remains in state q_1, replacing the input of the present cell with **0**(**no change**) and moves the head to the **right**.

State q_1 :

B	X	0	**1**	1	B

Head

Now according to the transition table $\delta(q_1, 1) = (q_2, Y, L)$: This means that now the system enters state q_2, replacing the input of the present cell with **Y** and moves the head to the **left**.

State q_2 :

Now according to the transition table $\delta(q_2, 1)$ has no instruction, so we move to next instruction which is $\delta(q_2, X) = (q_0, X, R)$: This means that now the system enters state q_0, replacing the input of the present cell with **X** and moves the head to the **right**.

State q_0 :

Now according to the transition table $\delta(q_0, X)$ has no instruction, so we move to next instruction which is $\delta(q_0, Y) = (q_3, Y, R)$: This means that now the system enters state q_3, replacing the input of the present cell with **Y**(**remains same**) and moves the head to the **right**.

State q_3 :

Now according to the transition table $\delta(q_3, Y) = (q_3, Y, R)$: This means that now the system remains in state q_3, replacing the input of the present cell with **Y** and moves the head to the **right**.

State q_3 :

B	X	X	Y	Y	B

Head

Now according to the transition table $\delta(q_3, B) = HALT$: This means that the process needs to be stopped. It has reached final state.

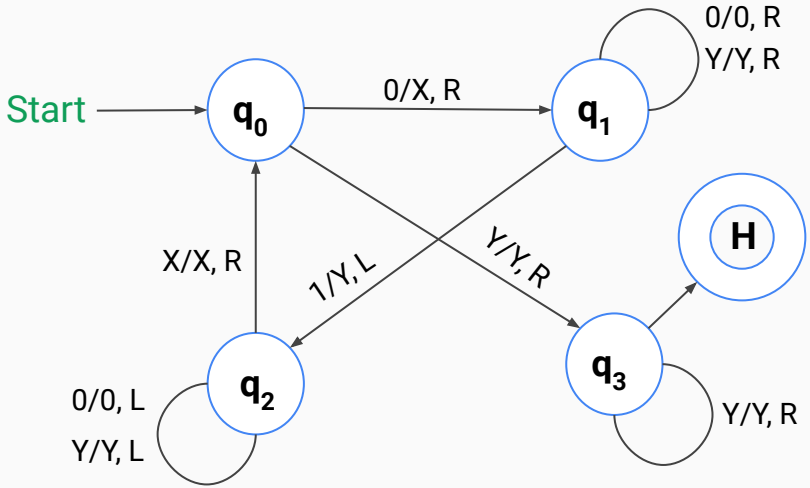
∴ The final output string will be **XXYY**

Let us look at the path we traced in the transition table

	0	1	X	Y	B
q_0	(q_1, X, R)			(q_3, Y, R)	
q_1	$(q_1, 0, R)$	(q_2, Y, L)		(q_1, Y, R)	
q_2	$(q_2, 0, L)$		(q_0, X, R)	(q_2, Y, L)	
q_3				(q_3, Y, R)	HALT

	0	1	X	Y	B
q_0	(q_1, X, R)			(q_3, Y, R)	
q_1	$(q_1, 0, R)$	(q_2, Y, L)		(q_1, Y, R)	
q_2	$(q_2, 0, L)$		(q_0, X, R)	(q_2, Y, L)	
q_3				(q_3, Y, R)	HALT

Finite state representation of the above transition table



One of the Important applications of Turing machine

References:
https://en.wikipedia.org/wiki/Busy_beaver
http://rave.ohiolink.edu/etdc/view?acc_num=osu1486554418657614

Busy Beaver game

The busy beaver game aims at finding a terminating program of given size that produces the most output possible. Since an infinitely looping program produces infinite output, such programs are excluded from the game.

More precisely , the busy beaver game consists of a designing a halting, binary-alphabet **Turing machine** which writes most 1s on the tape, using only a given set of states. Rules of this 2 state game are:

- The machine must have two states in addition to one halting state.
- The tape initially contains 0s only.

- A player should conceive a transition table aiming for the longest output of 1s on the tape while making sure that the machine will halt eventually.
- An n-th state busy beaver, BB-n or simply “busy beaver” is a Turing machine that **wins** the n-th state Busy beaver game, that is it attains the largest number of 1s among all other possible n-state competing turing machine
- Let us look at 2-state busy beaver and number of steps it take to produce maximum output.

2-state 2-symbol busy beaver

	A	B
0	1RB	1LA
1	1LB	1RH

2-state 2-symbol Busy beaver

	A	B
0	1RB	1LA
1	1LB	1R**H**

Initial state of tape: ...0<u>0</u>0000000...

[underlined cell is where the head is present]

- **Starting state:** δ(A, 0) = (1RB) ⇒ Overwrites the present element in the cell to 1, head moves to the right, and the state changes to B.

  After this first step:
  Tape condition: .....00001<u>0</u>000000000.....

- **New state:** δ(B, 0) = (1LA) ⇒ Overwrites the present element in the cell to 1, head moves to the left, and the state changes to A.

  After this second step:
  Tape condition : .....0000<u>1</u>1000000000.....

- **New state:** δ(A, 1) = (1LB) ⇒ Overwrites the present element in the cell to 1, head moves to the left, and the state changes to B.

  After this third step:
  Tape condition: .....000<u>0</u>11000000000.....

- **New state:** δ(B, 0) = (1LA) ⇒ Overwrites the present element in the cell to 1, head moves to the left, and the state changes to A.

  After this fourth step:
  Tape condition: ……00<u>0</u>1111000000000……

- **New state:** δ(A, 0) = (1RB) ⇒ Overwrites the present element in the cell to 1, head moves to the right, and the state changes to B.

  After the fifth step:
  Tape condition: ……001<u>1</u>11000000000……

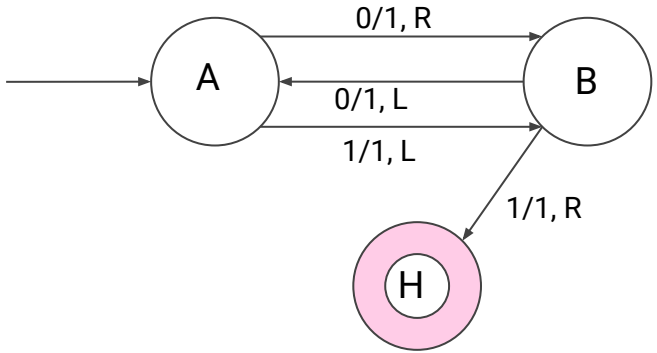
- **New state:** δ(B, 1) = (1R**H**) ⇒ Overwrites the present element in the cell to 1, head moves to the right, and the final state i.e., HALT is achieved.

  After the sixth step: (Final step)
  Tape condition: ……00**1111**000000000……

  Maximum output of four 1s is achieved by the considered 2 state busy beaver in six steps.

Below is the finite state representation of the above 2 state busy beaver.

	A	B
0	1RB	1LA
1	1LB	1RH



To visualize the 3 state 2 symbol busy beaver, run the C-code in the github link here.

Notable instances of Turing machine

A 748-state binary Turing machine has been constructed that halts if and only if ZFC is inconsistent.

A 744-state Turing machine has been constructed that halts if and only if the Riemann hypothesis is false.

A 43-state Turing machine has been constructed that halts if and only if Goldbach's conjecture is false.

A 15-state Turing machine has been constructed that halts iff the following conjecture formulated by Paul Erdős in 1979 is false:

- for all $n > 8$ there is at least one digit 2 in the base 3 representation of 2^n.

Progress of Algorithms

- Present applications of algorithms are found in every single moment of our daily life. For instance, on a smartphone, the colour balance of photographs captured by its camera is defined by a set of algorithms, which identifies colours and balances contrasts based on a scene. With increasing processing power, algorithms have grown in their complexity and level of computational prowess.

- The present generation is gunning for quantum computing and artificial intelligence. Machine learning depends on algorithms to “learn” from usage methods and prepare actions based on personal ways of operations. We are at the verge of time where ‘bots’ are looking to replace apps of our daily use. Algorithms are behind every nascent stage of technology, having evolved in their ability, while keeping the basis of operations constant with Euclid’s basic divisor equation from 300 BC.

Let us now look at different class of algorithms

Important terms and notations

Before we start studying our classifications there are couple of important terms we need to go through:

Asymptotic Notations

1. Big Theta (Θ)
2. Big Oh(O)
3. Big Omega (Ω)

When it comes to analysing the complexity of any algorithm in terms of time and space, we can never provide an exact number to define the time required and the space required by the algorithm, instead we express it using some standard notations, also known as Asymptotic Notations.

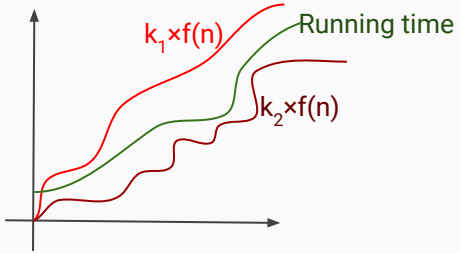
What is asymptotic behaviour?

The word Asymptotic means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken).

1. Big Theta (Θ): [Tight bounds]

Time complexity represented by the Big-Θ notation is like the average value or range within which the actual time of execution of the algorithm will be. For example, if for some algorithm the time complexity is represented by the expression $3n^2 + 5n$, and we use the Big-Θ notation to represent this, then the time complexity would be $\Theta(n^2)$, ignoring the constant coefficient and removing the insignificant part, which is 5n.

Here complexity $\Theta(n^2)$ means that the average time for any input 'n' will remain between $k_1 \times n^2$ and $k_2 \times n^2$, where k_1 and k_2 are constants.



2. Big Oh(O): [Upper bounds]

This notation is known as the upper bound of an algorithm or the worst case of an algorithm.

If we say that the time complexity is O(n), then it implies that the running time will be strictly less than or equal to ‘n’ or simply to say it will never exceed ‘n’.

3. Big Omega(Ω)

Big Omega notation is used to define the lower bound of any algorithm or we can say the best case of any algorithm.

This always indicates the minimum time required for any algorithm for all input values, therefore the best case of any algorithm.

In simple words, when we represent a time complexity for any algorithm in the form of big-Ω, we mean that the algorithm will take at least this much time to complete its execution. It can definitely take more time than this too.

- **Simple recursive algorithms**
- Backtracking algorithms
- Divide and conquer algorithms
- Dynamic programming algorithms
- Greedy algorithms
- Branch and bound algorithms
- Brute force algorithms
- Randomized algorithms

RECURSION

- **Recursion** means "defining a problem in terms of itself". This can be a very powerful tool in writing algorithms.
- In this recursion process the function calls itself directly or indirectly and the corresponding function is called recursive function.
- Recursion comes directly from Mathematics, where there are many examples of expressions written in terms of themselves.
- For example, the Fibonacci sequence is defined as: F(i) = F(i-1) + F(i-2).

Recursion is the process of defining a solution to a problem in terms of (a simpler version of) itself.

For example, we can define the operation "**find your way home**" as:

1. If you are at home, stop moving.
2. Take one step toward home.
3. "find your way home".

What did we do in “find your way home” ?

- Firstly we don’t go to home if we are already at home. So that will be the end of the process.
- Secondly, we make a simple move that makes our situation simpler to solve.
- Finally we redo the entire algorithm.

The above example is called tail recursion. This is where the very last statement is calling the recursive algorithm. Tail recursion can be directly translated into loops

Parts of recursive algorithms

All recursive algorithms must have the following:

1. Base case # When to stop
2. Work towards the base case
3. Recursive call # Call ourselves

The "**work toward base case**" is where we make the problem simpler (e.g., divide list into two parts, each smaller than the original). The **recursive call**, is where we use the same algorithm to solve a simpler version of the problem. The **base case** is the solution to the "simplest" possible problem (For example, the base case in the problem 'find the largest number in a list' would be if the list had only one number... and by definition if there is only one number, it is the largest).

Let us look at some examples of recursions and then understand why recursion works.

Example 1. Adding a list of numbers

Pseudo code

```
function sum( array, length ):
    If length of array = 0
        Then return 0
```

Base case

```
    Decrease the length of the array by 1
    Return array[0] + sum( array + 1, length )
```

Work towards base case

Recursive call

array = List of our numbers to be summed.

length = length of the array passed to the function " sum ".

Loop version of the above recursive code:

Since in the above pseudo code we have our recursive call at the end of the code we can successfully convert it into a Iterative(loop) code.

Let us look at the pseudo code of that program.

Pseudo code

```
function sum( array, length ):
        Declare a variable “index” = 0.
        Declare a variable “result” = 0 to store our sum.
        While index < length of the array follow the below steps:
                Add the value in the present index of the array to the result
        Once out of the loop return the variable result as our final answer.
```

C code for both the above versions

To view the C code for the recursive version click on the link here

To view the C code for the iterative version click on the link here

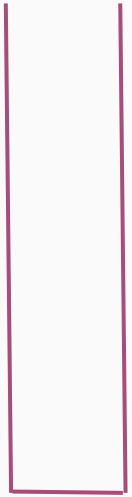
How does the recursion work?

- In recursive algorithms the computer remembers every previous stage of the algorithm. It stores the results in the activation stack. The most recently produced result is held at the top of the stack because it is pushed most recently onto the stack. When we encounter a "**return**" in the program the program starts using the value at the top of the stack to evaluate the previous result and this is further used to evaluate the previous ones.
- The last value which will be evaluated in the stack is our required output.
- This might be not so clear when we read, so let us visualize what's happening actually.

Our recursion code for adding numbers in a list

```
function sum( array, length ):
        If length of array = 0
                Then return 0

        Decrease the length of the array by 1
        Return array[0] + sum( array + 1, length )
```

Let us consider our array to be: [6, 4, 11, 0, 5, 6, 9, 25, 26]



Initial condition of the stack:
Nothing is pushed onto the stack when the program is not started.

[**6**, 4, 11, 0, 5, 6, 9, 25, 26]

Length of the array : 9

Array[0] : 6

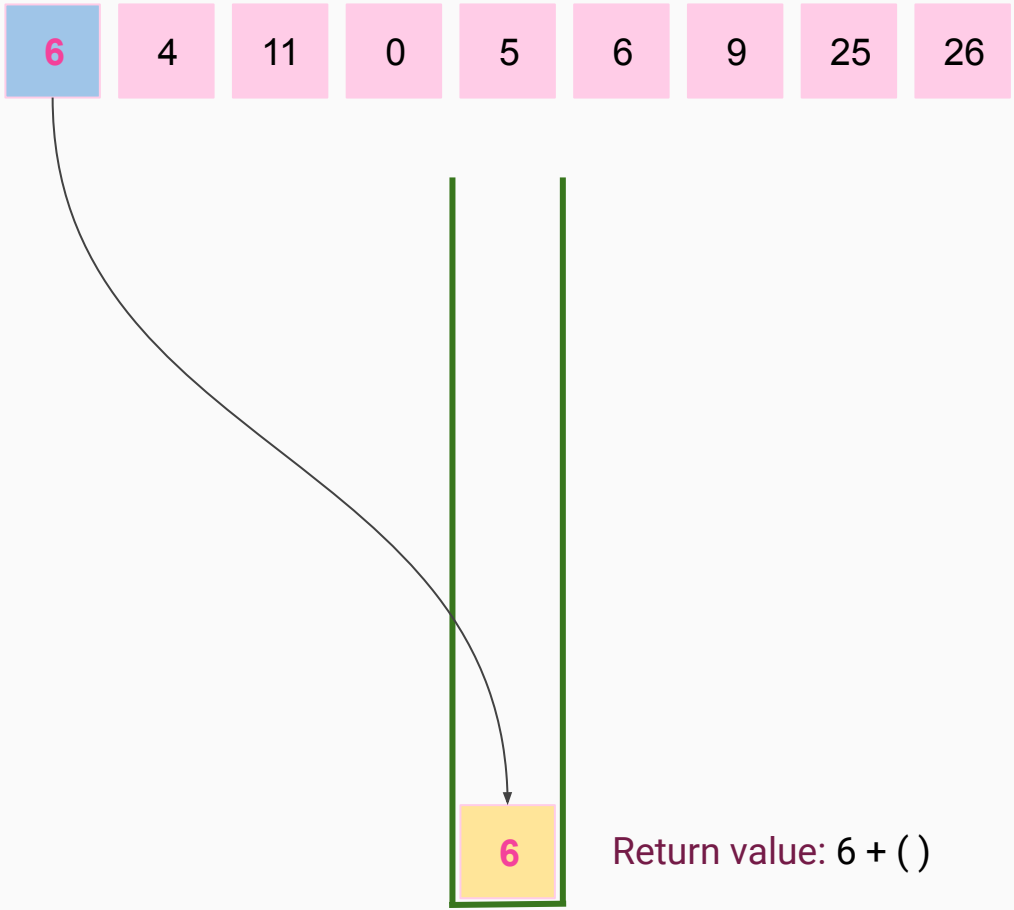
When the program starts it first checks the if clause.

Since our length is not equal to 0 it won't go into the if clause and moves further.

So we will decrease our length now by 1.
So Length will be 9 - 1 = 8.

Now the function will have to return
array[0] + recursive call()

Since there is recursive call the program will push the result on to the stack and proceed executing the recursion call.



[**4**, 11, 0, 5, 6, 9, 25, 26]

Length of the array : 8

Array[0] : 4

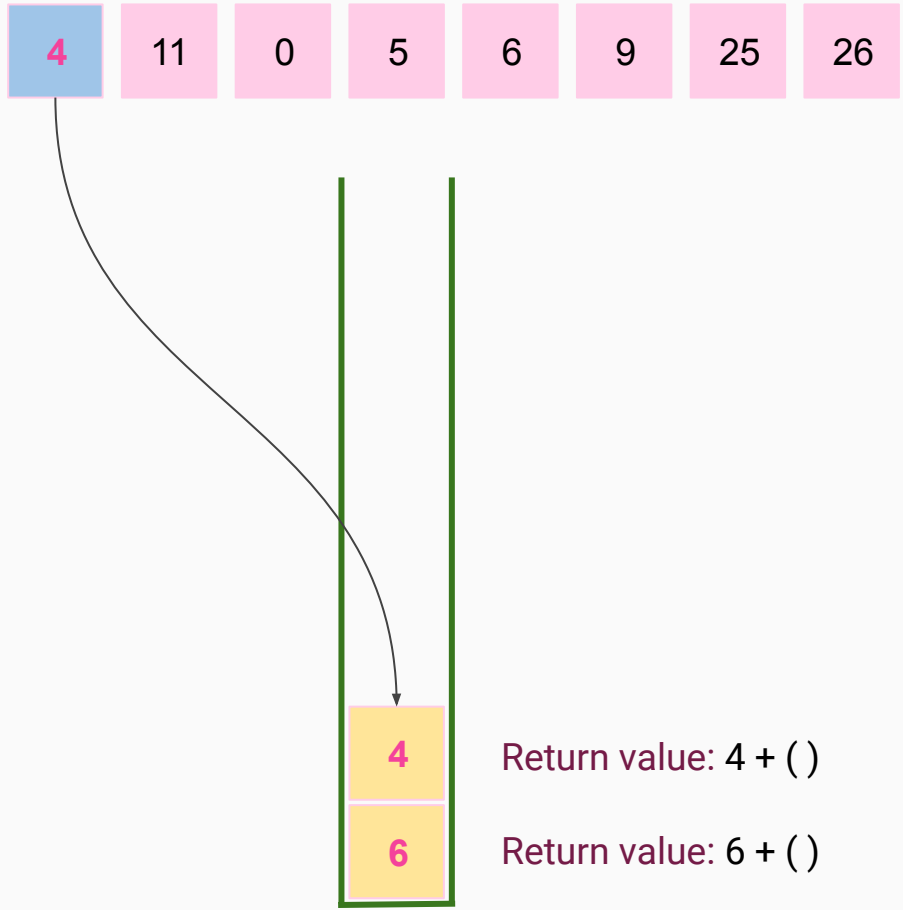
When the program starts it first checks the if clause.

Since our length is not equal to 0 it won't go into the if clause and moves further.

So we will decrease our length now by 1.
So Length will be 8 - 1 = 7.

Now the function will have to return
array[0] + recursive call()

Since there is recursive call the program will push the result on to the stack and proceed executing the recursion call.



[**11**, 0, 5, 6, 9, 25, 26]

Length of the array : 7

Array[0] : 11

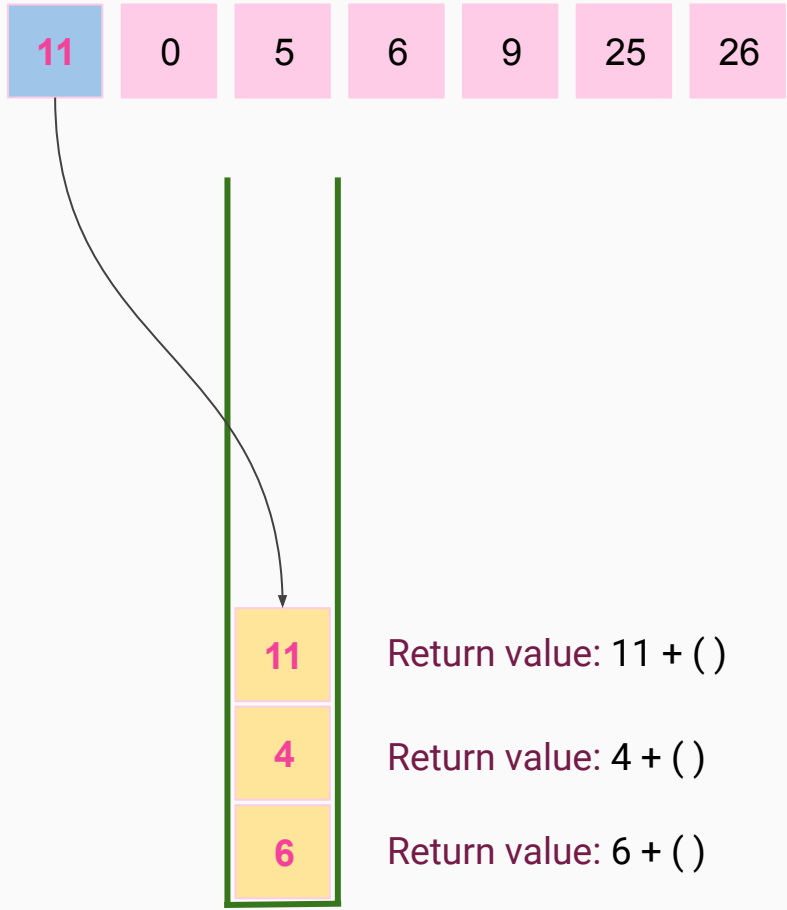
When the program starts it first checks the if clause.

Since our length is not equal to 0 it won't go into the if clause and moves further.

So we will decrease our length now by 1.
So Length will be 7 - 1 = 6.

Now the function will have to return
array[0] + recursive call()

Since there is recursive call the program will push the result on to the stack and proceed executing the recursion call.



[**0**, 5, 6, 9, 25, 26]

Length of the array : 6

Array[0] : 0

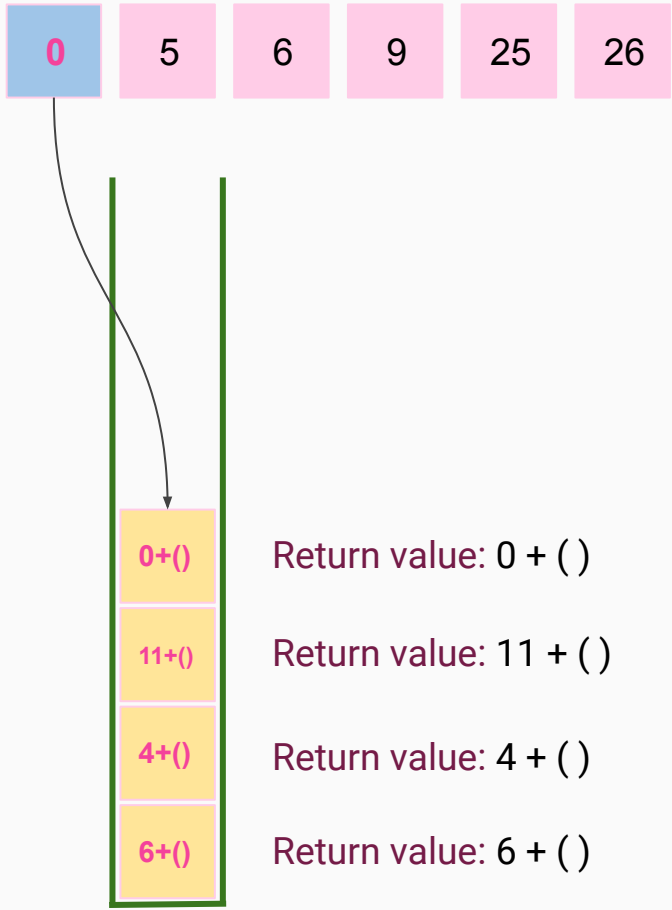
When the program starts it first checks the if clause.

Since our length is not equal to 0 it won't go into the if clause and moves further.

So we will decrease our length now by 1.
So Length will be 6 - 1 = 5.

Now the function will have to return
array[0] + recursive call()

Since there is recursive call the program will push the result on to the stack and proceed executing the recursion call.



[**5**, 6, 9, 25, 26]

Length of the array : 5

Array[0] : 5

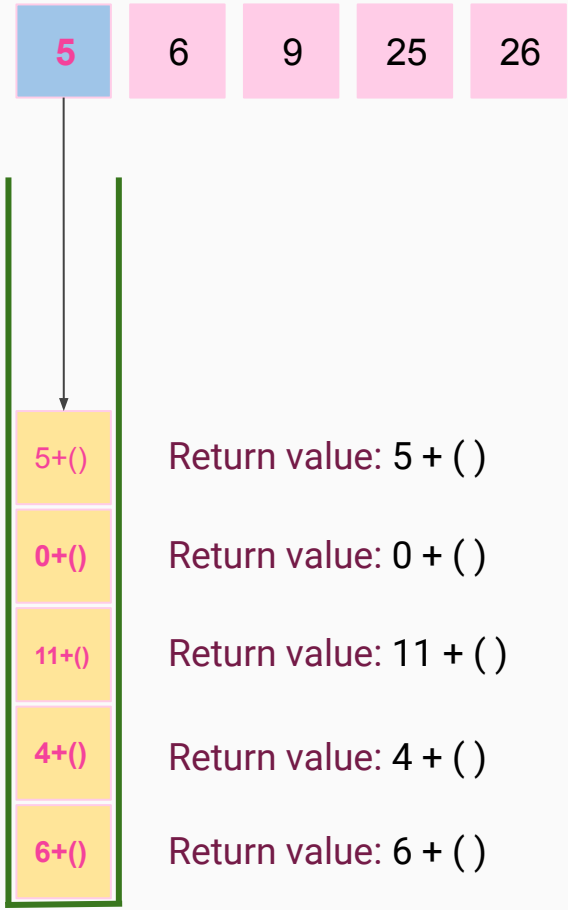
When the program starts it first checks the if clause.

Since our length is not equal to 0 it won't go into the if clause and moves further.

So we will decrease our length now by 1.
So Length will be 5 - 1 = 4.

Now the function will have to return
array[0] + recursive call()

Since there is recursive call the program will push the result on to the stack and proceed executing the recursion call.



[**6**, 9, 25, 26]

Length of the array : 4

Array[0] : 6

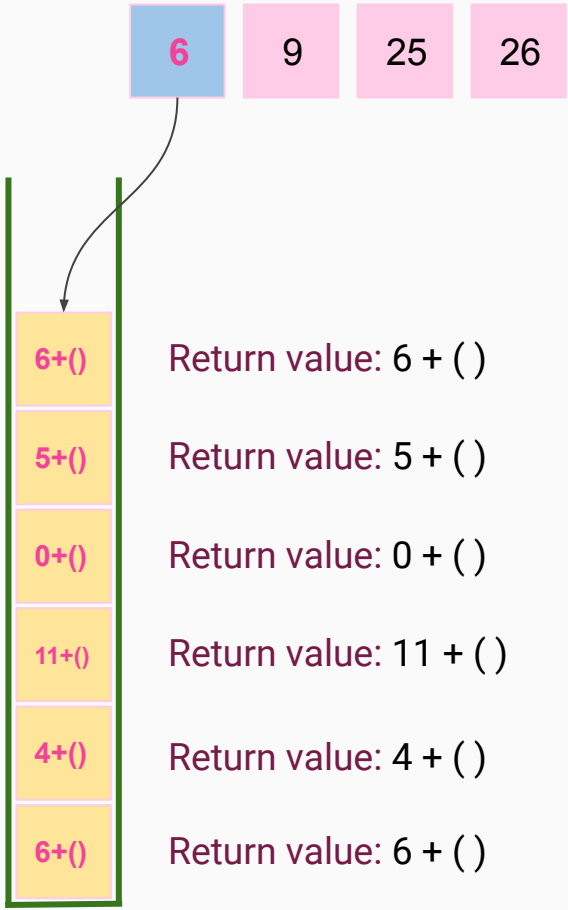
When the program starts it first checks the if clause.

Since our length is not equal to 0 it won't go into the if clause and moves further.

So we will decrease our length now by 1.
So Length will be 4 - 1 = 3.

Now the function will have to return
array[0] + recursive call()

Since there is recursive call the program will push the result on to the stack and proceed executing the recursion call.



[**9**, 25, 26]

Length of the array : 3

Array[0] : 9

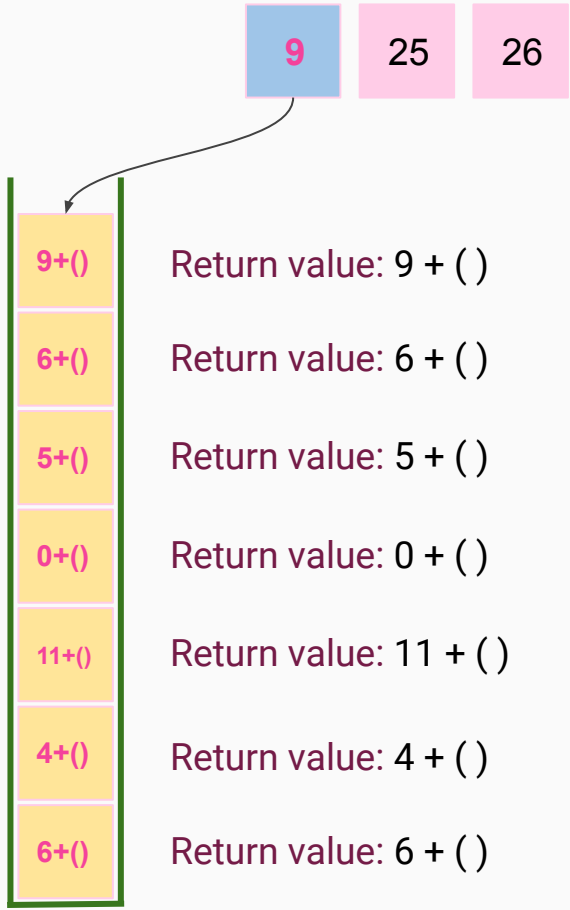
When the program starts it first checks the if clause.

Since our length is not equal to 0 it won't go into the if clause and moves further.

So we will decrease our length now by 1.
So Length will be 3 - 1 = 2.

Now the function will have to return
array[0] + recursive call()

Since there is recursive call the program will push the result on to the stack and proceed executing the recursion call.



[**25**, 26]

Length of the array : 2

Array[0] : 25

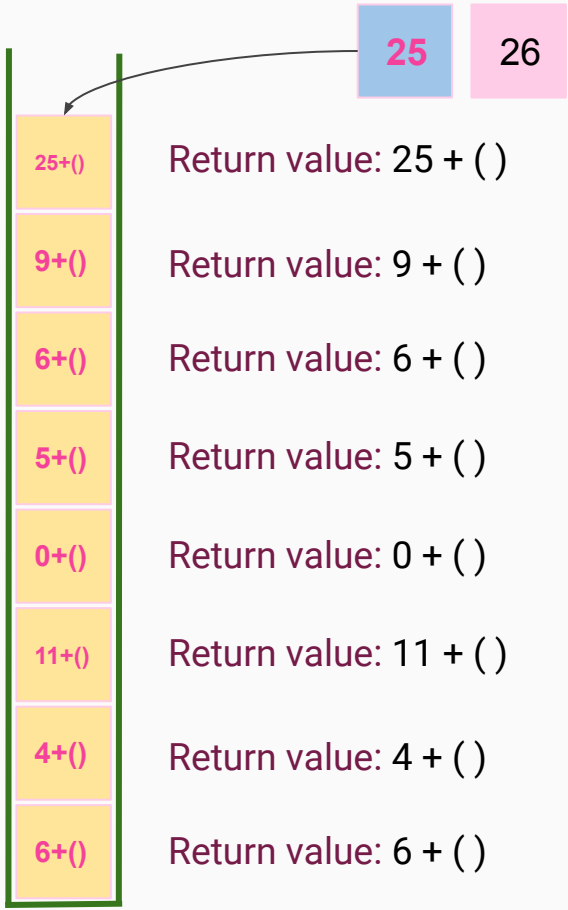
When the program starts it first checks the if clause.

Since our length is not equal to 0 it won't go into the if clause and moves further.

So we will decrease our length now by 1.
So Length will be 2 - 1 = 1.

Now the function will have to return
array[0] + recursive call()

Since there is recursive call the program will push the result on to the stack and proceed executing the recursion call.



[26]

Length of the array : 1

Array[0] : 26

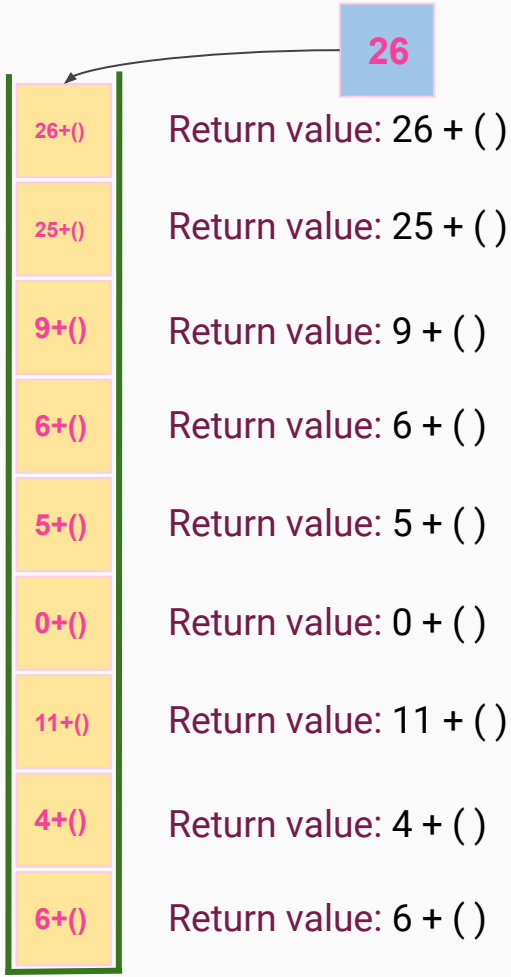
When the program starts it first checks the if clause.

Since our length is not equal to 0 it won't go into the if clause and moves further.

So we will decrease our length now by 1.
So Length will be 1 - 1 = 0.

Now the function will have to return
array[0] + recursive call()

Since there is recursive call the program will push the result on to the stack and proceed executing the recursion call.



[ ]

Length of the array : 0

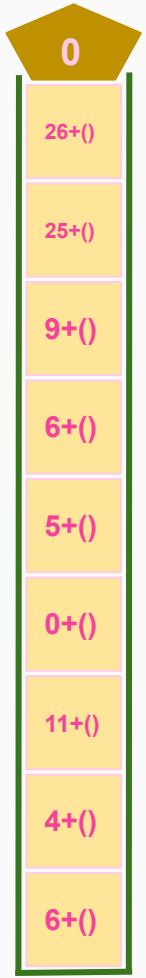
Array[0] : Doesn't exist
Segmentation fault
(Illegal access of memory)

When the program starts it first checks the if clause.

Since our length is equal to 0 it will execute the statements inside if clause. Which means that the program now returns a zero and no further function calls are executed.

Which means we can stop pushing values onto our stack now.

Now the return value '0' is used in evaluating the previous return values in the stack



Return value: 26 + ( )

Return value: 25 + ( )

Return value: 9 + ( )

Return value: 6 + ( )

Return value: 5 + ( )

Return value: 0 + ( )

Return value: 11 + ( )

Return value: 4 + ( )

Return value: 6 + ( )

Now we start moving from the top to the bottom using the most recent returned values for evaluation and popping them out of the stack

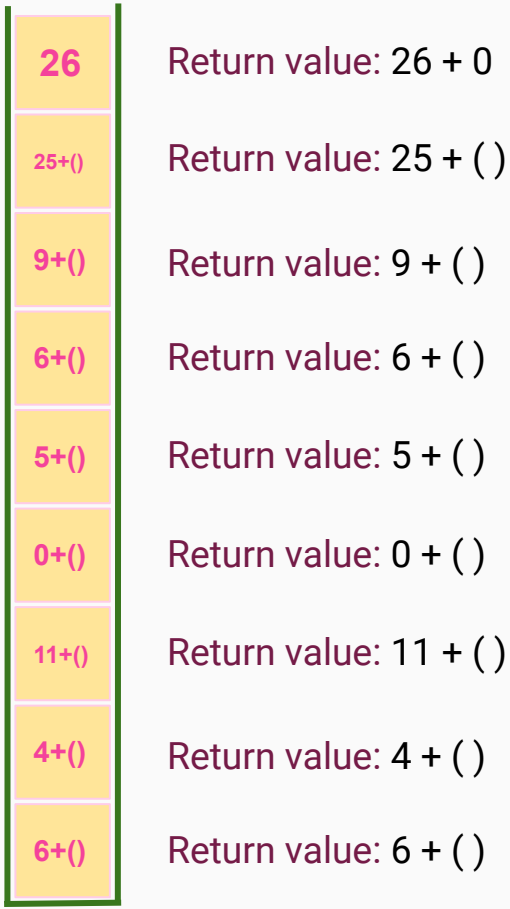
Firstly we use ‘ 0 ‘ in the evaluation of the next recent value on the stack which is

26 + ( ): Here ( ) is the return value of the recursive call executed while the program was executing.

So, ‘0’ will be popped out from the stack and the value will be used in evaluating the present top of the stack which will be 26+().

Now, after popping 0 from the stack the value of the top of the stack will be:

26 + 0 = 26.



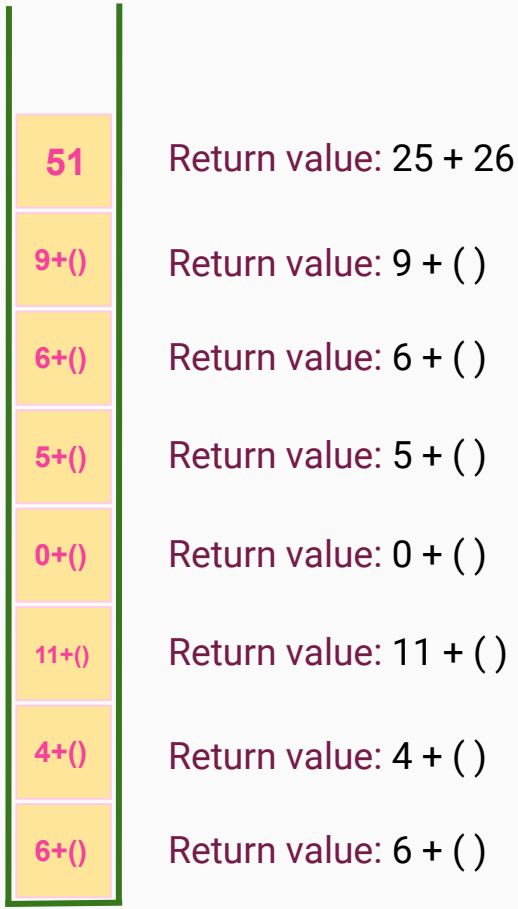
Now we use ‘ 26 ‘ in the evaluation of the next recent value on the stack which is

25 + ( ): Here ( ) is the return value of the recursive call executed while the program was executing.

So, ‘26’ will be popped out from the stack and the value will be used in evaluating the present top of the stack which will be 25+().

Now, after popping 26 from the stack the value of the top of the stack will be:

25 + 26 = 51.



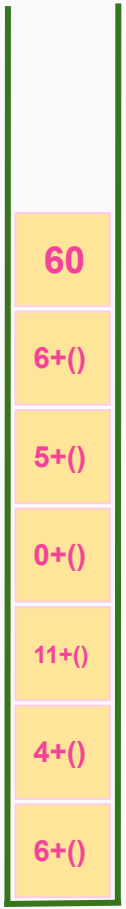
Now we use ‘ 51 ‘ in the evaluation of the next recent value on the stack which is

9 + ( ): Here ( ) is the return value of the recursive call executed while the program was executing.

So, ‘51’ will be popped out from the stack and the value will be used in evaluating the present top of the stack which will be 9+().

Now, after popping 51 from the stack the value of the top of the stack will be:

9 + 51 = 60.



Return value: 9 + 51

Return value: 6 + ( )

Return value: 5 + ( )

Return value: 0 + ( )

Return value: 11 + ( )

Return value: 4 + ( )

Return value: 6 + ( )

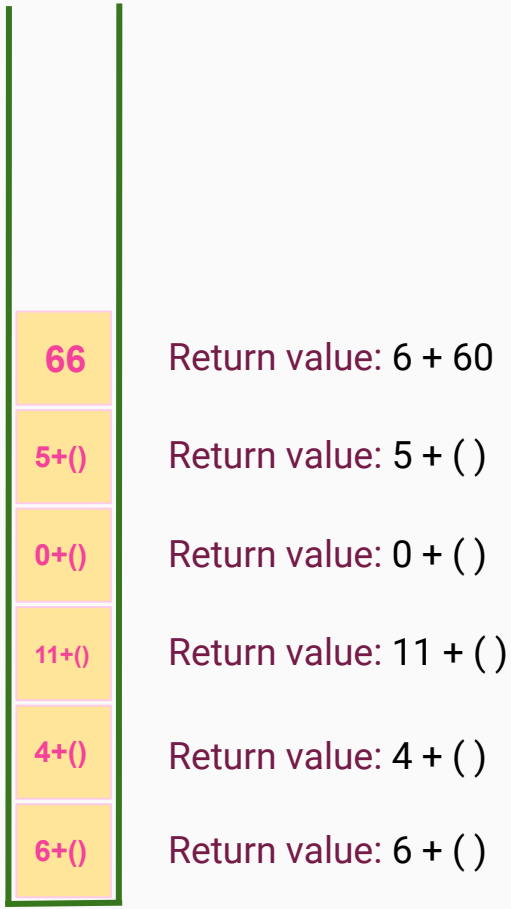
Now we use ‘ 60 ‘ in the evaluation of the next recent value on the stack which is

6 + ( ): Here ( ) is the return value of the recursive call executed while the program was executing.

So, ‘60’ will be popped out from the stack and the value will be used in evaluating the present top of the stack which will be 6+().

Now, after popping 60 from the stack the value of the top of the stack will be:

6 + 60 = 66.



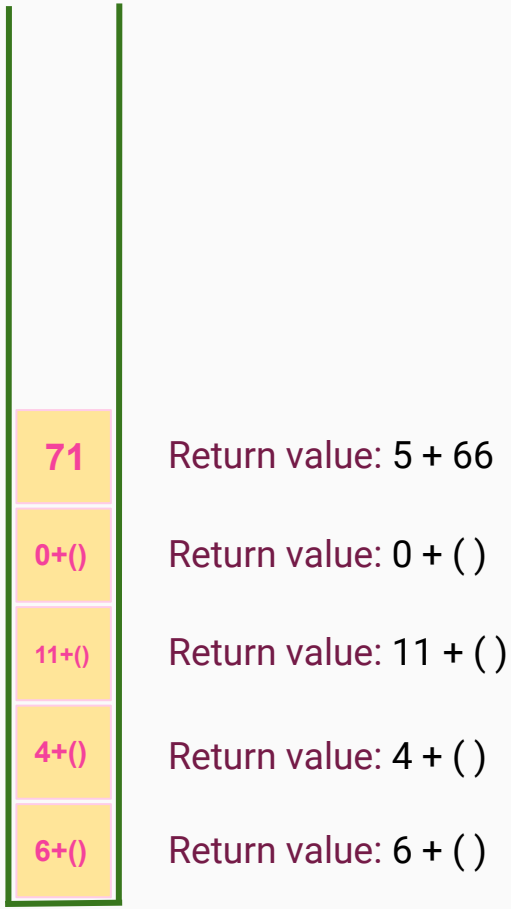
Now we use ‘ 66 ‘ in the evaluation of the next recent value on the stack which is

5 + ( ): Here ( ) is the return value of the recursive call executed while the program was executing.

So, ‘66’ will be popped out from the stack and the value will be used in evaluating the present top of the stack which will be 5+().

Now, after popping 66 from the stack the value of the top of the stack will be:

5 + 66 = 71.



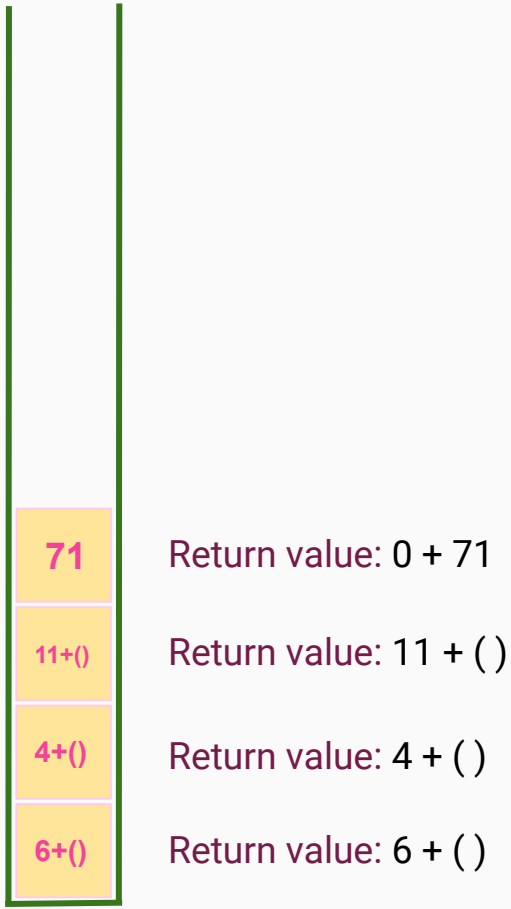
Now we use ‘ 71 ‘ in the evaluation of the next recent value on the stack which is

0 + ( ): Here ( ) is the return value of the recursive call executed while the program was executing.

So, ‘71’ will be popped out from the stack and the value will be used in evaluating the present top of the stack which will be 0+().

Now, after popping 71 from the stack the value of the top of the stack will be:

0 + 71 = 71.



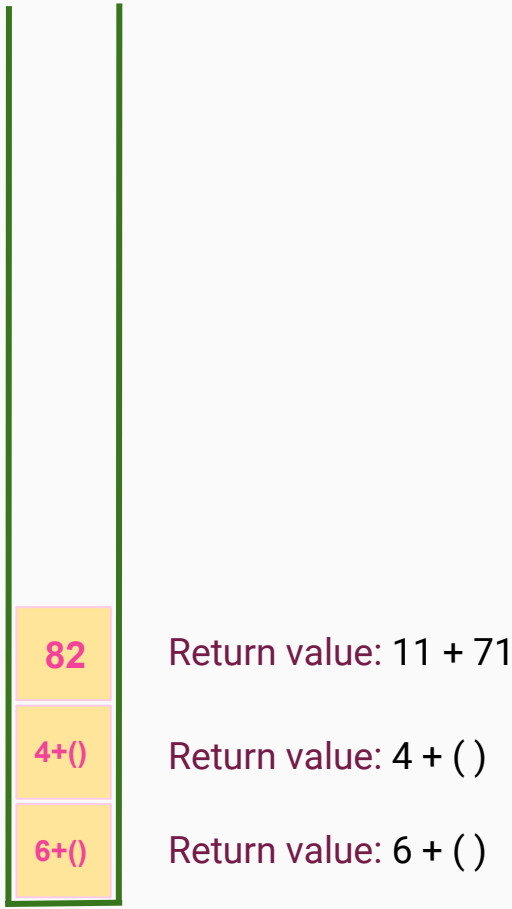
Now we use ‘ 71 ‘ in the evaluation of the next recent value on the stack which is

11 + ( ): Here ( ) is the return value of the recursive call executed while the program was executing.

So, ‘71’ will be popped out from the stack and the value will be used in evaluating the present top of the stack which will be 11+().

Now, after popping 71 from the stack the value of the top of the stack will be:

11 + 71 = 82.



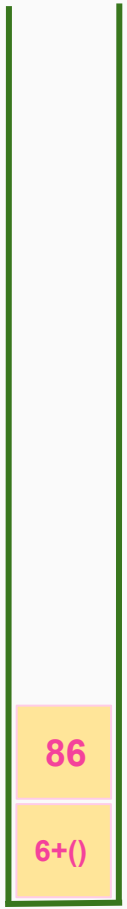
Now we use ‘ 82 ‘ in the evaluation of the next recent value on the stack which is

4 + ( ): Here ( ) is the return value of the recursive call executed while the program was executing.

So, ‘82’ will be popped out from the stack and the value will be used in evaluating the present top of the stack which will be 4+().

Now, after popping 82 from the stack the value of the top of the stack will be:

4 + 82 = 86.



Return value: 4 + 82

Return value: 6 + ( )

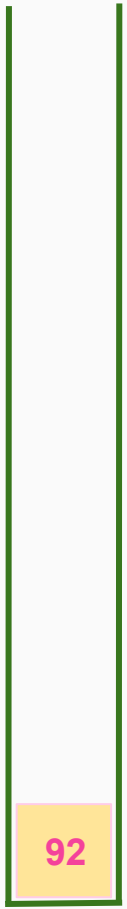
Now we use ‘ 86 ‘ in the evaluation of the next recent value on the stack which is

6 + ( ): Here ( ) is the return value of the recursive call executed while the program was executing.

So, ‘86’ will be popped out from the stack and the value will be used in evaluating the present top of the stack which will be 6+().

Now, after popping 86 from the stack the value of the top of the stack will be:

6 + 86 = 92.

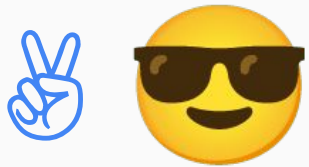


Return value: 6 + 86

Now, the final value popped out of the stack will be the final answer and in our case it is 92

The sum of the array [6, 4, 11, 0, 5, 6, 9, 25, 26] = 6+4+11+0+5+6+9+25+26 = **92**

This is how recursion works



Time complexity

- Recursive approach

Let us see what is the time complexity of the above addition function and how to calculate it.

Outline of our function goes as below:

```
Function sum(n):
        Base case
        {
          ……..
        }
        Recursive call: sum(n - 1)
```

Let T(n) denote the number of elementary operations performed by the function sum(n).

We can identify two important properties:

- Since sum(1) is calculated using fixed number of operations say k_1, we say **$T(1) = k_1$**
- If n>1, then the function will perform a fixed number of operations say k_2 and make a recursive call to sum(n-1). This recursive call will make **T(n-1)** operations. In total we get, **$T(n) = k_2 + T(n-1)$.**

For an asymptotic estimate we don't need the exact values of k_1 and k_2. Instead we consider $k_1 = k_2 = 1$. Now we can find the time complexity using the following recurrence relation.

$T(n) = t$
$1 + T(n-1) = T(n) = t$
$1 + (1 + T(n-2)) = 2 + T(n-2) = T(n) = t$
$2 + (1 + T(n-3)) = 3 + T(n-3) = T(n) = t$
⋮
$k + T(n-k) = T(n) = t$
⋮
$(n-1) + T(1) = T(n) = t$
$(n-1) + 1 = \Theta(n)$

Example 2. Fibonacci Sequence

What is a Fibonacci number ?

A nth Fibonacci number is defined as below:

$$F(n) = \begin{cases} F(n-1) + F(n-2) & ; \text{ if } n > 1 \\ 1 & ; \text{ if } n = 1 \\ 0 & ; \text{ if } n = 0 \end{cases}$$

What is a Fibonacci sequence?

The sequence of all F(n)s is called a Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ……

Fibonacci numbers are named after the Italian mathematician Leonardo of Pisa, later known as Fibonacci.

Applications of Fibonacci numbers include computer algorithms such as the **Fibonacci search technique** and the **Fibonacci heap data structure.**

They also appear in biological settings, such as branching in trees, the arrangement of leaves on a stem, the fruit sprouts of a pineapple, the flowering of an artichoke etc.

We will discuss about the above applications and observations later in this presentation.

Pseudo Code

Recursive version:

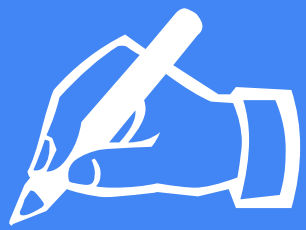
```
Function fibo(n):
        Begin
                if n <= 1 then
                        Return n;
                else
                        Return sum of Fibo(n-1), Fibo(n-2);

        End
```

Base case

Recursive call

Notice that we again have the Recursive call part at the end of the function.
So we can write the Iterative (loop) version of this.

Analysis



- By using Recursion to solve this problem we get a cleanly written function, that checks. If the given number is equal to 0 and 1 we return both given numbers.

- If we pass a number that is greater than 0 and 1. Then we make two recursive calls where we add both calls with the nth Number minus 1 and 2 in both calls.

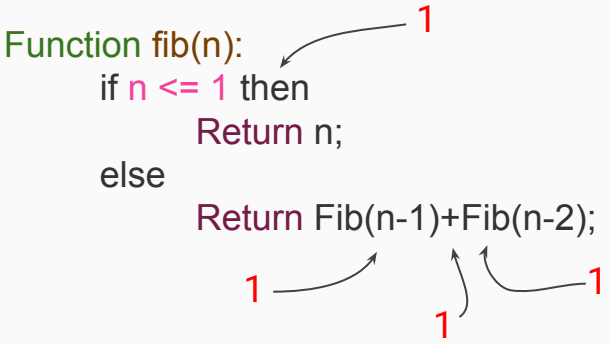
- Passing these Integers 0, 1, 2, 3, 4, 5, Will most likely give us --> 0, 1, 1, 2, 3, 5 in a timely fashion.

- But what happens if we pass higher numbers like 50, 67, 100. Well if we try to run the function with the given numbers in our IDE we will begin to notice how much longer it takes for this method gives us our Fibonacci number(More than minutes). Now trying to run a Space Complexity analysis will be a tricky thing to do because of a lot of things are happening behind the scenes of this recursive function. The reason for the poor performance is heavy push-pop of the stack memory in each recursive call.

Time complexity

- Recursive approach

Let us have a look at our code:



While computation, the conditional statement will take a constant amount of time to get executed. Let us consider that it takes 1 unit of time.

The same conditional statement gets executed again and again whenever we recursively call children of fib(n). Fib(n) has two children (fib(n-1) and fib(n-2)). Hence the two children will cost 1 unit of time each. So, we add up extra 2 units of time.

The addition operation between the children will take again a constant amount of time. So we add another 1 unit of time.

So, totally a constant of 4 will be present when we try to calculate T(n).

Lower Bound of time complexity

$T(n) = T(n-1) + T(n-2) + c$; $c = 4$

We consider that T(n-1) and T(n-2) are approximately equal and for lower bound we replace T(n-1) with T(n-2).

$$T(n-1) \cong T(n-2)$$

So, $T(n) = T(n-2) + T(n-2) + c$

$T(n) = 2T(n-2) + c$

$= 2(2T(n-4) + c) + c$

$= 4T(n-4) + 3c$

$= 4(2T(n-6) + c) + 3c$

$= 8T(n-6) + 7c$

$= 16T(n-8) + 15c$ .....

General form: $2^kT(n-2k) + (2^k-1)c$

This series goes till T(n-2k) = T(0). Which means it goes till $n-2k = 0 \Rightarrow k = n/2$

$T(n) = 2^{n/2}T(0) + (2^{n/2}-1)c$

$= (1+c)2^{n/2} - c$ [Since T(0) =T(1) = 1]

Lower bound $\propto 2^{n/2}$

Upper Bound of time complexity

$T(n) = T(n-1) + T(n-2) + c$; $c = 4$

We consider that T(n-1) and T(n-2) are approximately equal and for upper bound we replace T(n-2) with T(n-1).

$$T(n-2) \cong T(n-1)$$

So, $T(n) = T(n-1) + T(n-1) + c$

$T(n) = 2T(n-1) + c$

$= 2(2T(n-2) + c) + c$

$= 4T(n-2) + 3c$

$= 4(2T(n-3) + c) + 3c$

$= 8T(n-3) + 7c$

$= 16T(n-4) + 15c$ .....

General form: $2^kT(n-k) + (2^k-1)c$

This series goes till T(n-k) = T(0). Which means it goes till $n-k = 0 \Rightarrow k = n$

$T(n) = 2^nT(0) + (2^n-1)c$

$= (1+c)2^n - c$ [Since T(0) =T(1) = 1]

Upper bound $\propto 2^n$

So, the lower bound time complexity is **Ω($2^{n/2}$)** and the upper bound is **O(2^{n})**.

Which means the time taken will be between the lower bound and upper bound.

Do you think this is efficient? 🤔

If we want to calculate the fibonacci number of 100, i.e., fib(100), how much time will it take?

It is not tough to answer since we have already calculated the lower and upper bounds

Lower bound = $2^{(100/2)} = 2^{50}$
Upper bound = 2^{100}

Time taken (t): $2^{50} < t < 2^{100}$

∴ The time taken will be no less than 2^{50} = **1,125,899,906,842,624**.
and in the worst case it might nearly reach 2^{100} = **1.2676506e+30**

This will take years to get calculated 😨😩

Well, the 2^n is not exactly the tight upper bound

Recall that fibonacci sequence is a recursive sequence where each term is defined as $F_n = F_{(n-1)} + F_{(n+2)}$
For $n > 1$ with base cases as $F_0 = 0$ and $F_1 = 1$.

It is useful to find a closed-form formula that gives the n^{th} term of the fibonacci series so that we don't have to recursively generate every term prior to the term we want to calculate.

We do this by solving a second order linear differential equation.

We bring all the terms onto left hand side:

$$F_n - F_{(n-1)} - F_{(n-2)} = 0$$

We consider that r^n solves the equation:

$$r^n - r^{(n-1)} - r^{(n-2)} = 0$$

$$r^{(n-2)}[\, r^2 - r - 1 \,] = 0$$

$$r^2 - r - 1 = 0$$

It is a simple quadratic equation whose roots are given by the formula $(-b \pm \sqrt{(b^2 - 4ac)})/2a$

Let the roots be p and q.

$p = (-(-1) + \sqrt{((-1)^2 - 4(1)(-1))})/2(1) = (1 + \sqrt{(1 + 4)})/2 = (1 + \sqrt{5})/2$

$q = 1 - p = (1 - \sqrt{5})/2$

For any c_1 substituting $c_1 p^n$ for F_n in $F_n - F_{n-1} - F_{n-2}$ yields zero and for any c_2 substituting $c_2 q^n$ for F_n in $F_n - F_{n-1} - F_{n-2}$ yields zero.

This suggests our solution has the form: $F_n = c_1 p^n + c_2 q^n$

We can simply substitute $q = 1 - p$

$$F_n = c_1 p^n + c_2 (1 - p)^n$$

Considering n = 0 and substituting $F_0 = 0$ in the above equation:

$$F_0 = c_1 p^0 + c_2 (1 - p)^0$$

$$0 = c_1 (1) + c_2 (1)$$

$$\rightarrow \quad c_1 = -c_2$$

$$\therefore \quad F_n = c_1 (p^n - (1 - p)^n)$$

Incorporating the initial condition $F_1 = 1$:

$$F_1 = 1 = c_1 (p^1 - (1 - p)^1)$$

$$1 = c_1 (2p - 1)$$

$$c_1 = 1/(2p-1) = 1/\sqrt{5}$$

$$F_n = (p^n - (1 - p)^n)/\sqrt{5}$$

Now while applying recursion:

$$F_n = F_{(n-1)} + F_{(n+2)}$$

$$F_n = ((1 + \sqrt{5})/2)^n + ((1 - \sqrt{5})/2)^n$$

$$T(n) = O(((1 + \sqrt{5})/2)^n + ((1 - \sqrt{5})/2)^n)$$

Using the property of Big O notation we can drop the lower order terms.

$$T(n) = O(((1 + \sqrt{5})/2)^n)$$

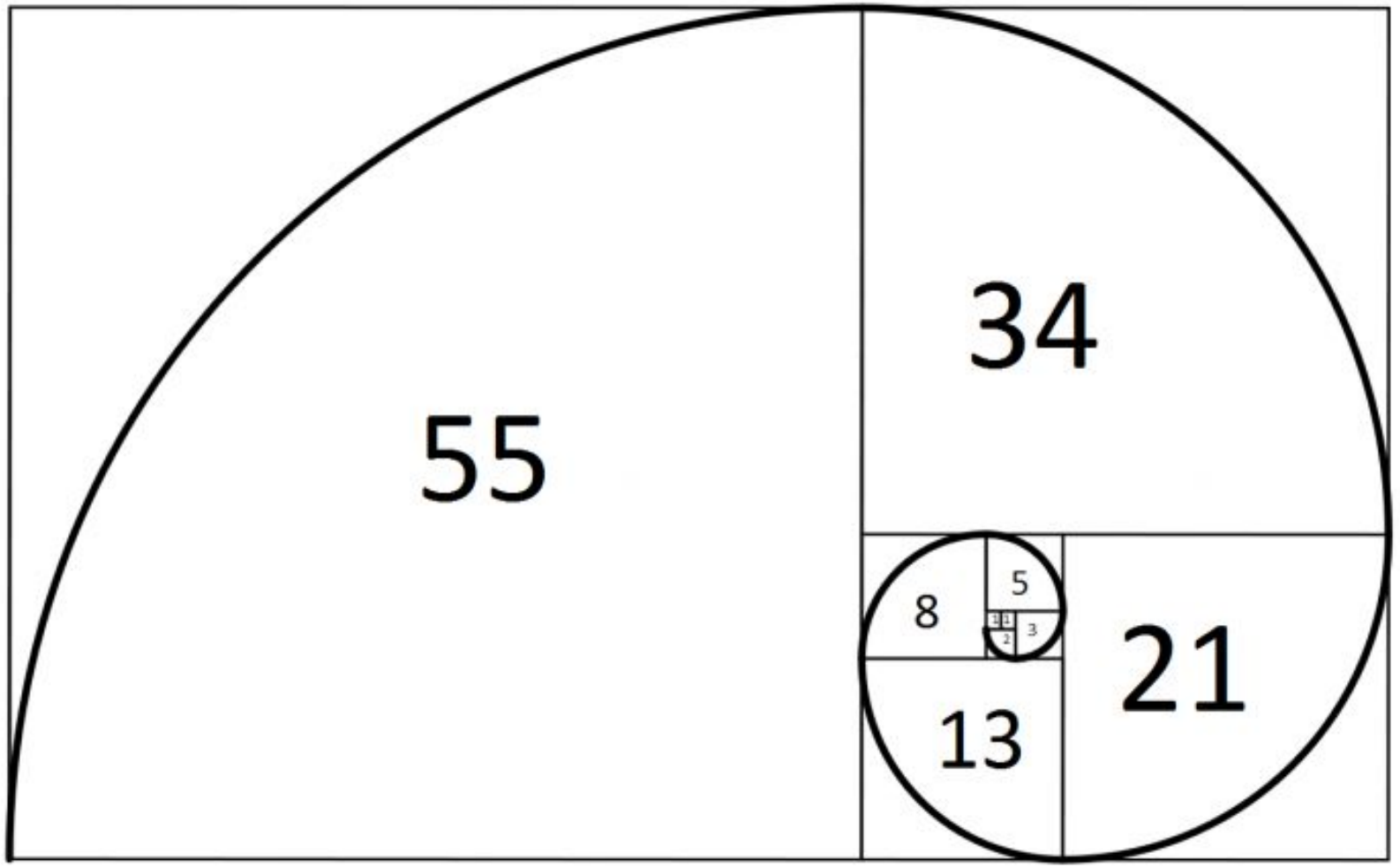
$$T(n) = O(1.6180)^n$$

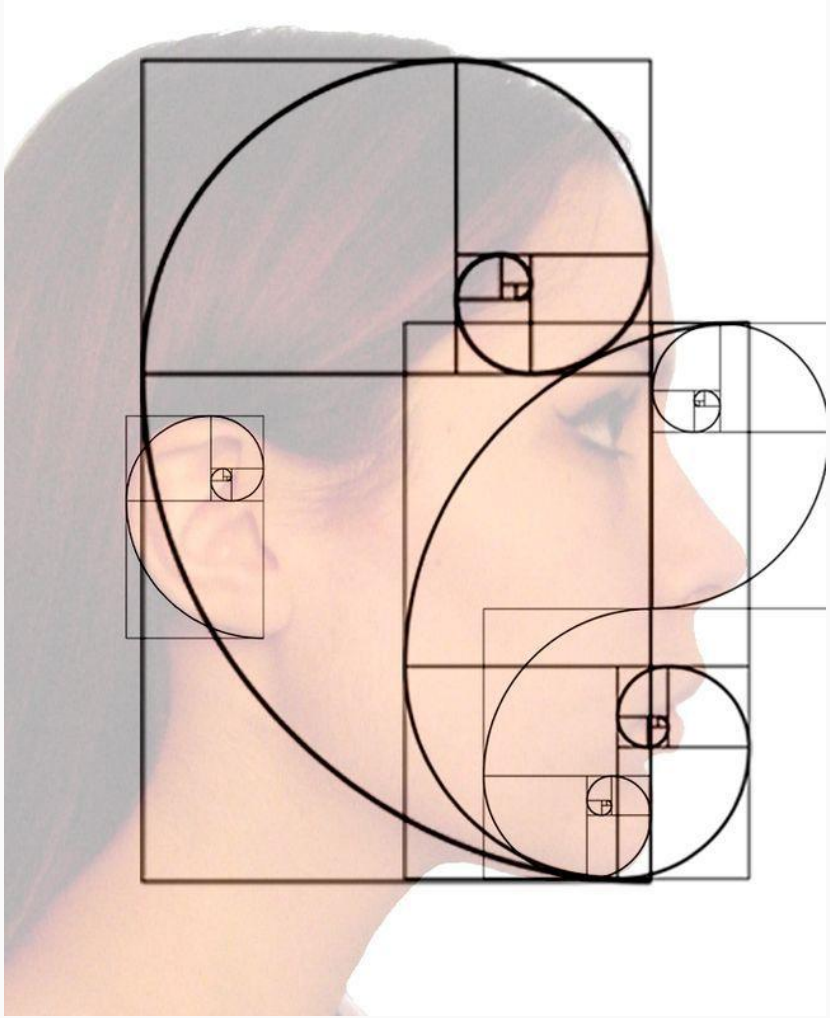
The value 1.6180 is called Golden ratio and as some pretty interesting applications

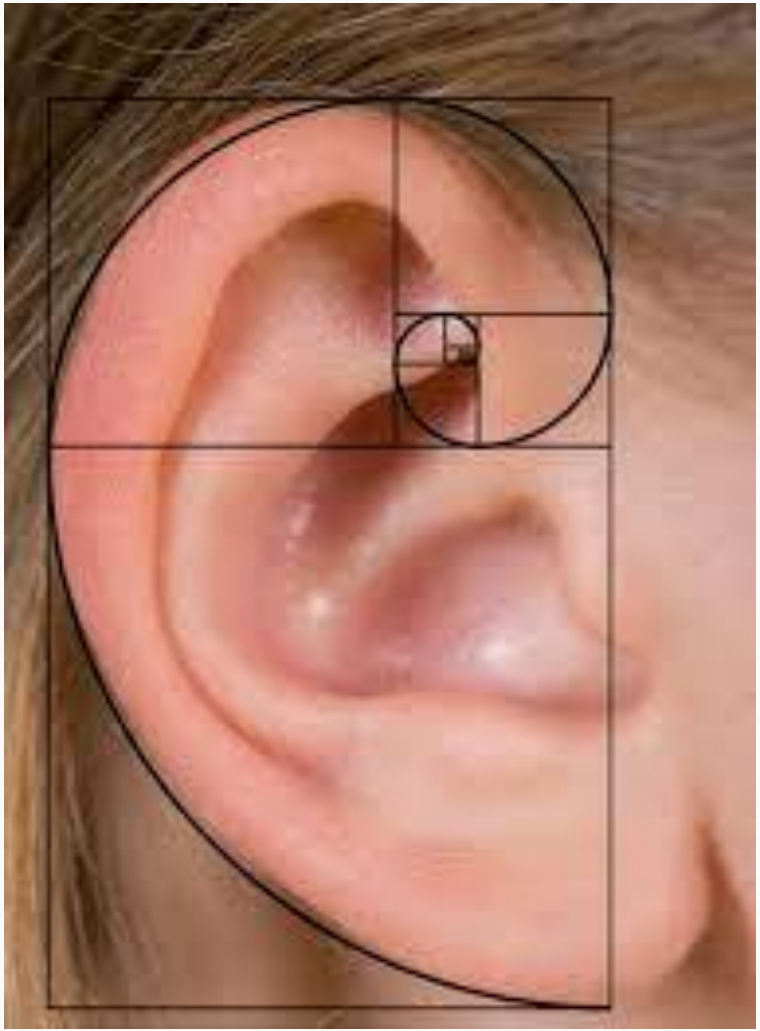
Golden ratio - 1.6180...

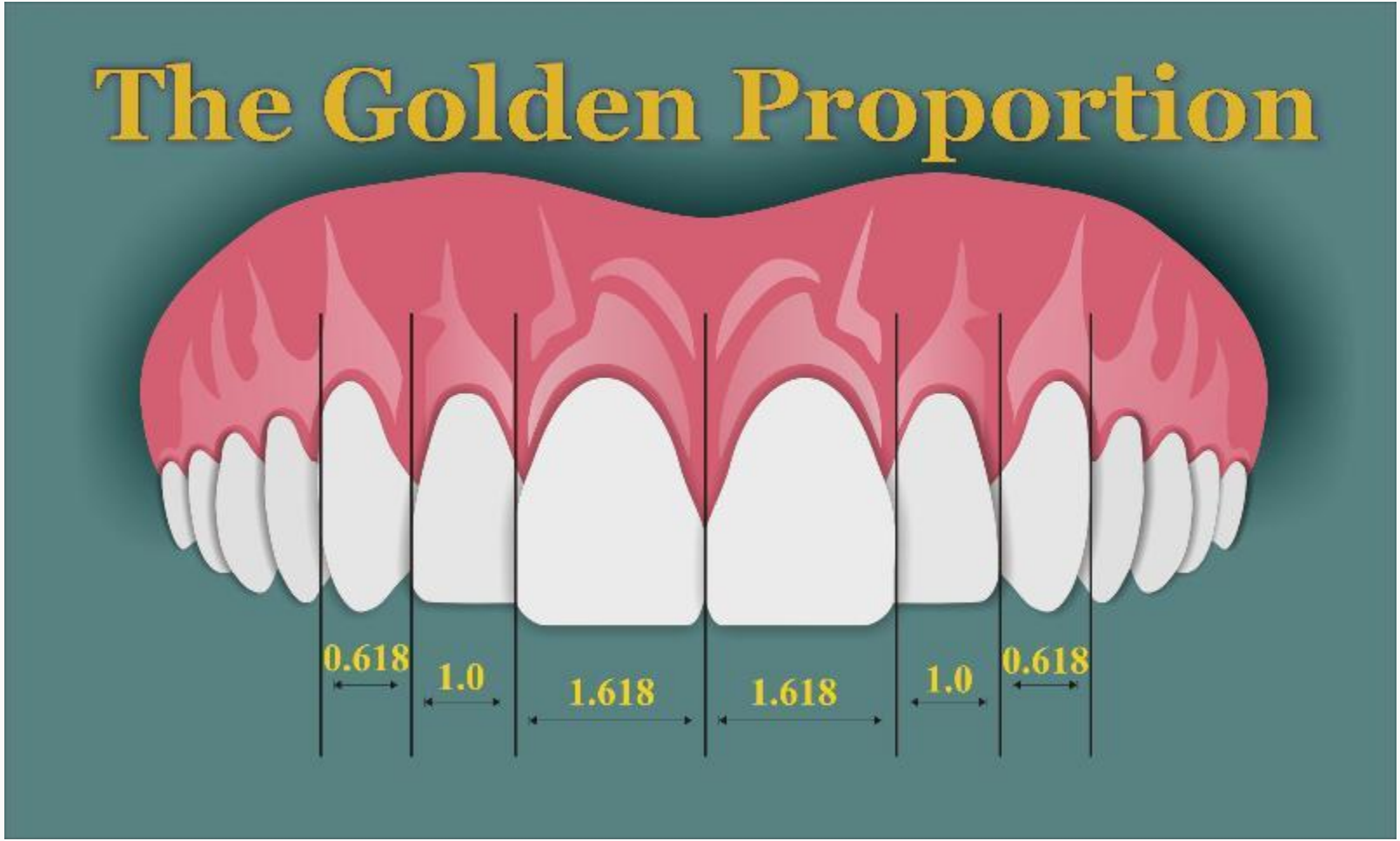
A magical number

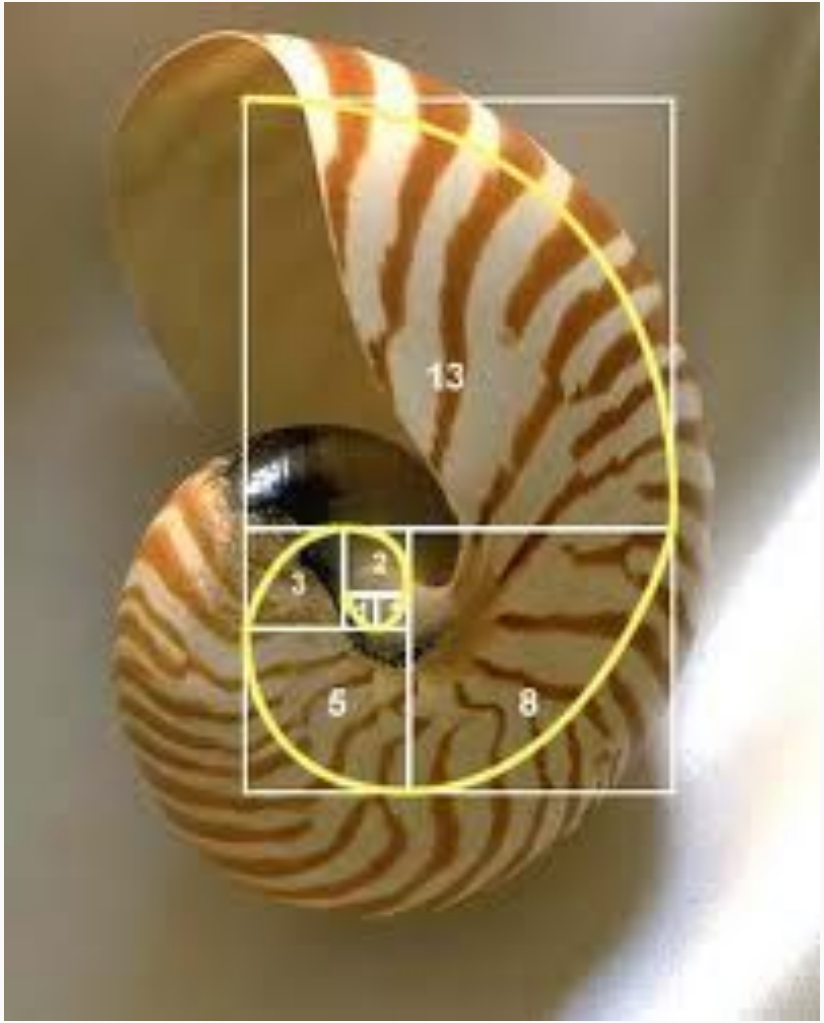

0,1+1

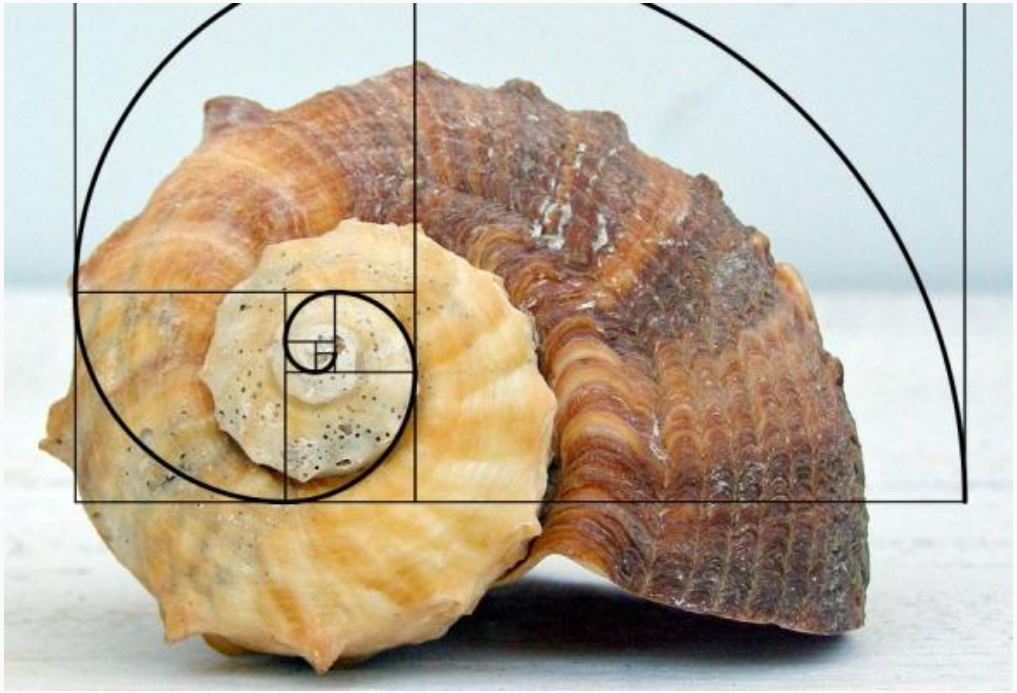

55
34
21
13
8
5
3
1
1
2
3

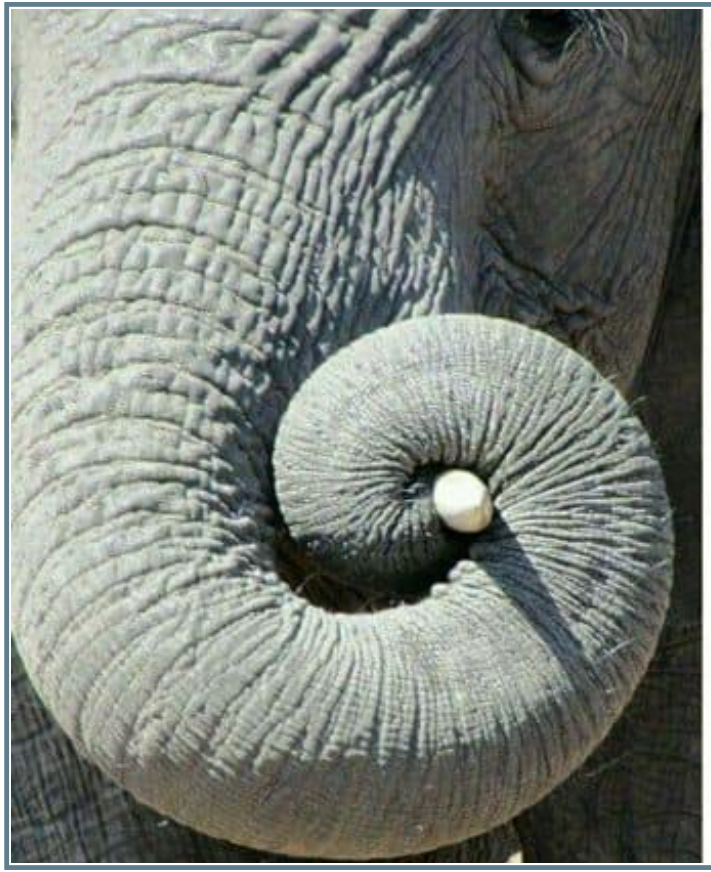


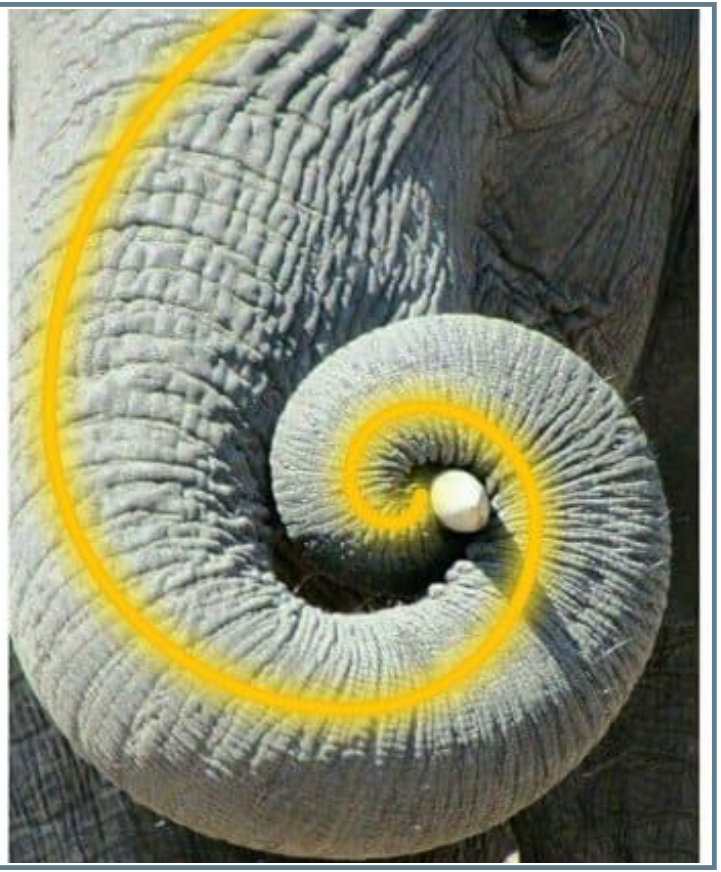


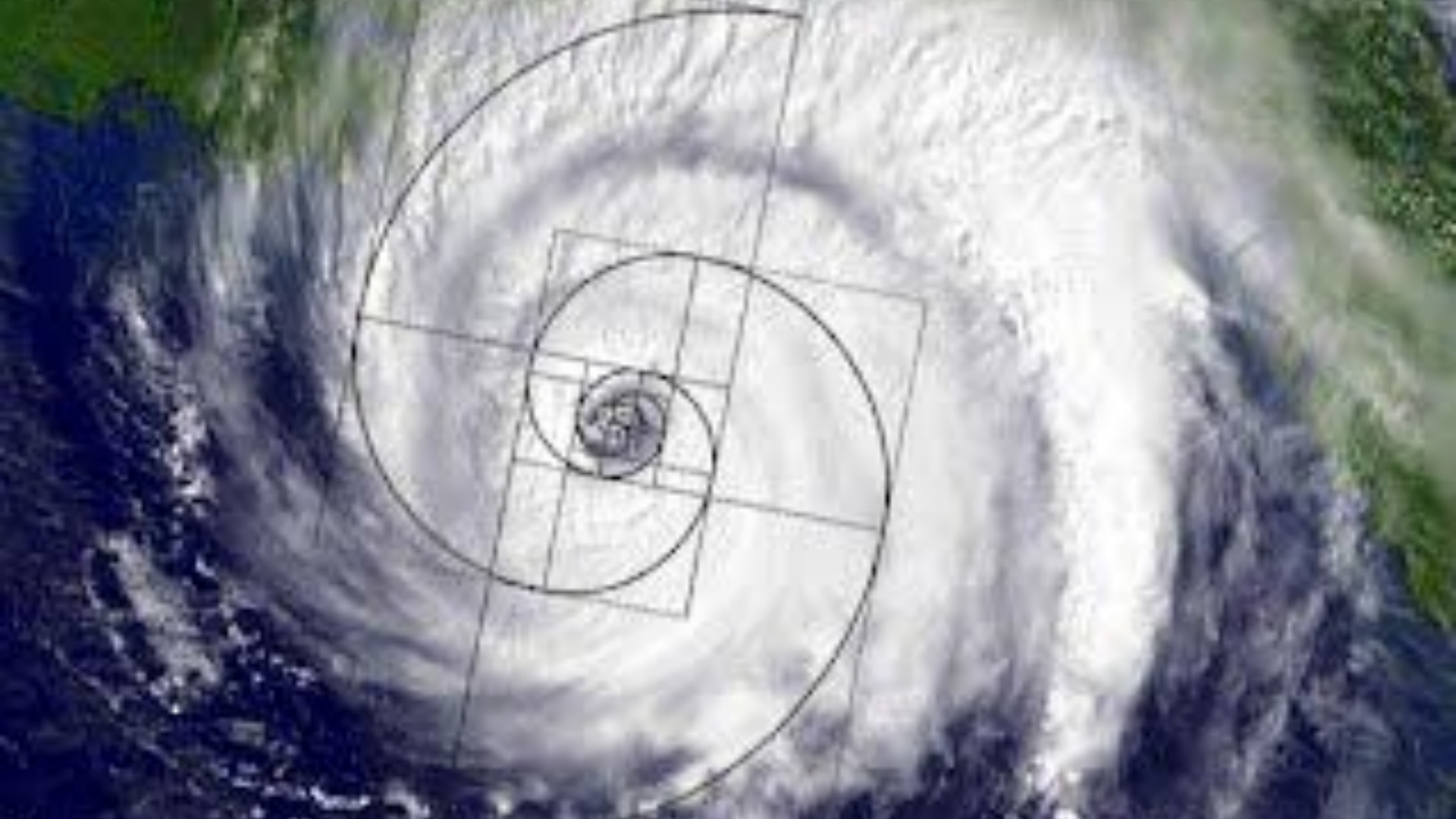

The Golden Proportion
0.618
1.0
1.618
1.618
1.0
0.618


13
3
2
5
8

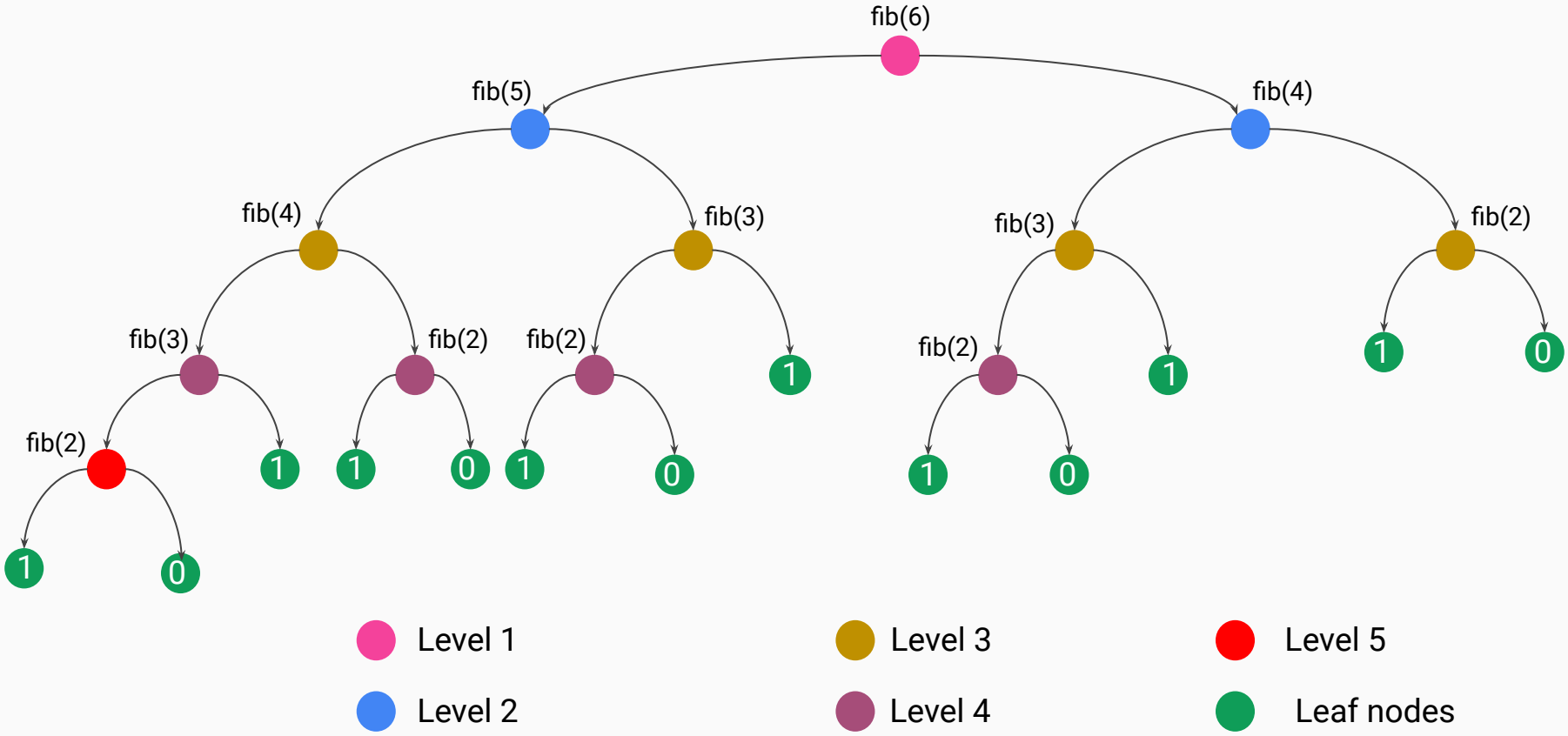








Graphical view



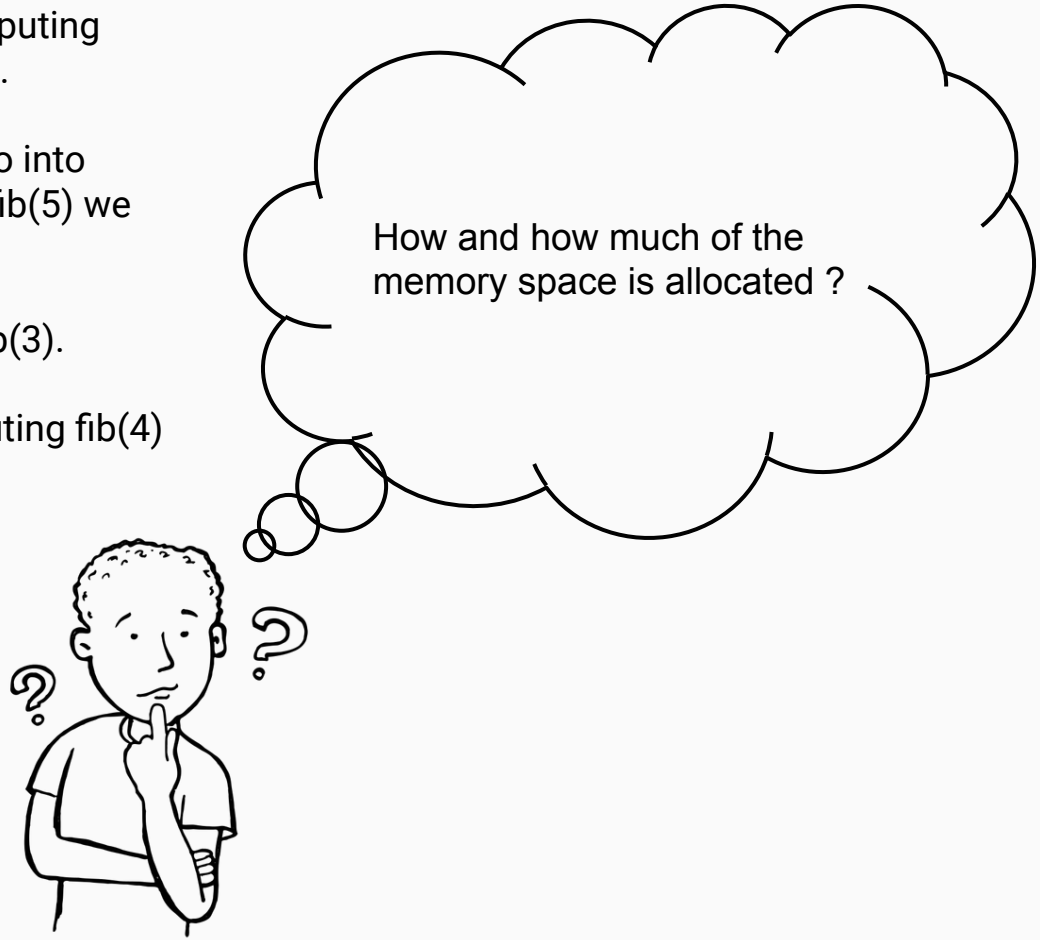
What might be the reason for such a huge time ?

- Firstly, it will be calculating same values repeatedly.
- It will calculate fib(k) multiple times as seen in the graphical representation

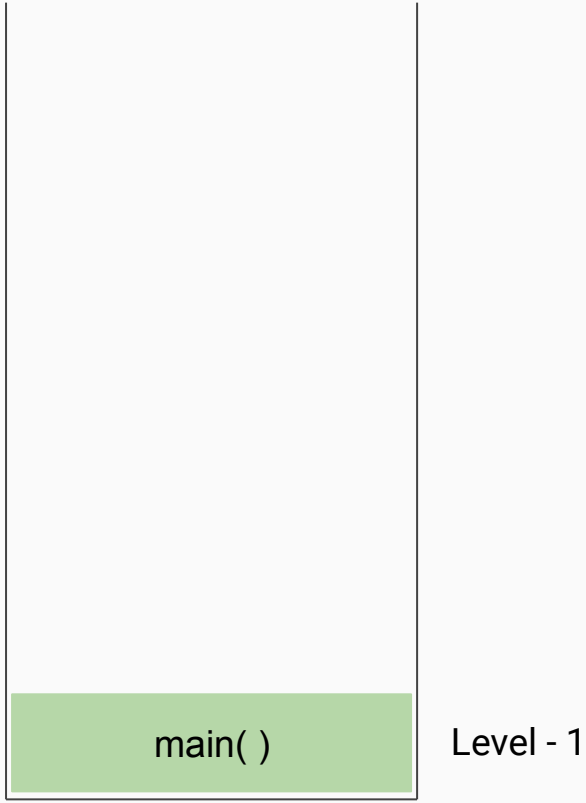
- Secondly it will have to push all these values onto stack and then pop them all.
- Firstly, the values corresponding to the first recursive call will be pushed on to the stack.
- Next the values of the second recursive call will be pushed onto the stack and when it reaches the base case they start adding up with the previous values pushed onto the stack and gets popped.
- These push and pop operations will cost lot of time.

Space complexity

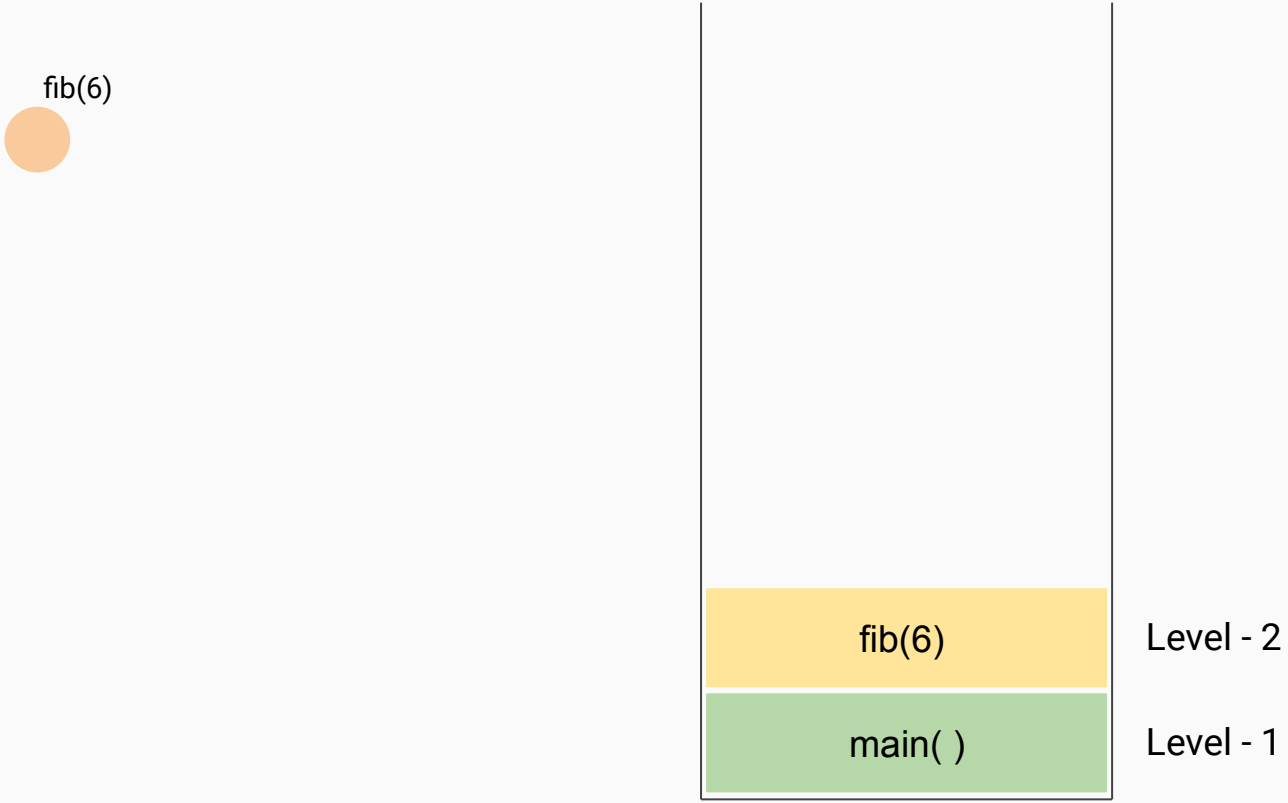
- As depicted in the graphical view of computing fib(6), we recursively call fib(5) and fib(4).
- While computing fib(5) + fib(4) we first go into fib(5) and only after the computation of fib(5) we move on with the computation of fib(4).
- fib(5) again recursively calls fib(4) and fib(3).
- Since fib(4) is called first we start computing fib(4) before fib(3).
- And this process goes on…..



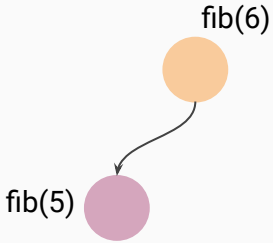
Allocation of system memory

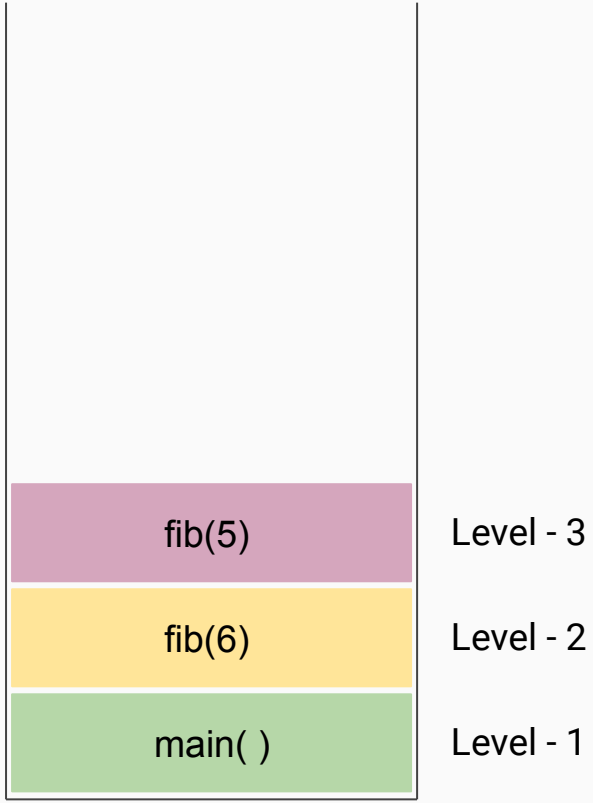


main( ) function calls fib(6)

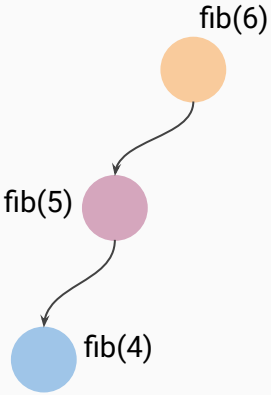


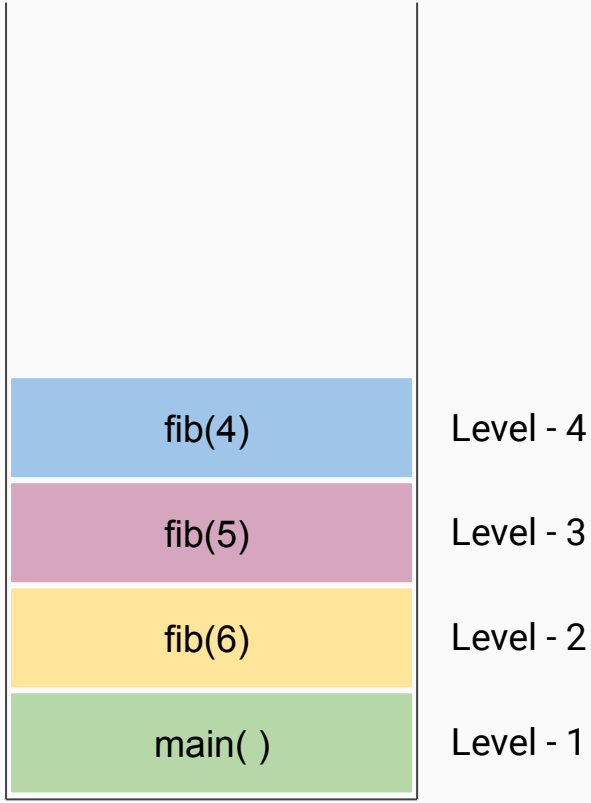
fib(6) = fib(5) + fib(4)



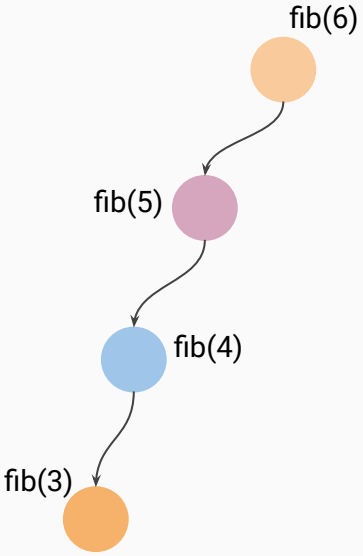


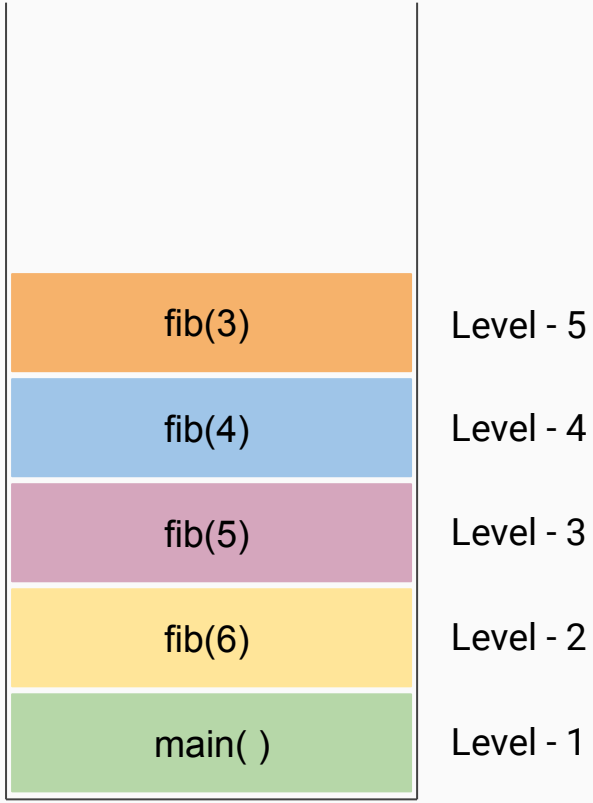
fib(5) = fib(4) + fib(3)



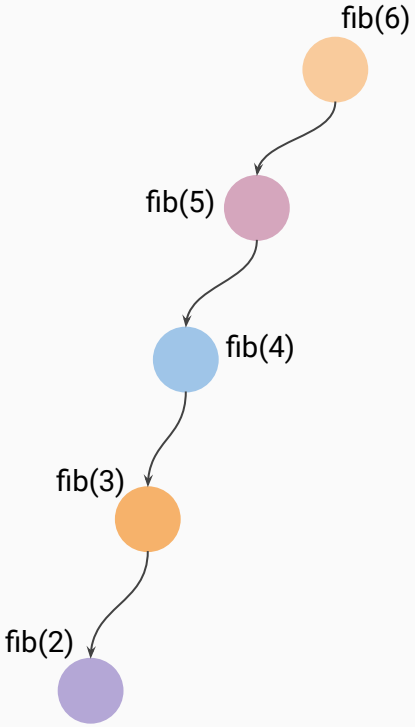


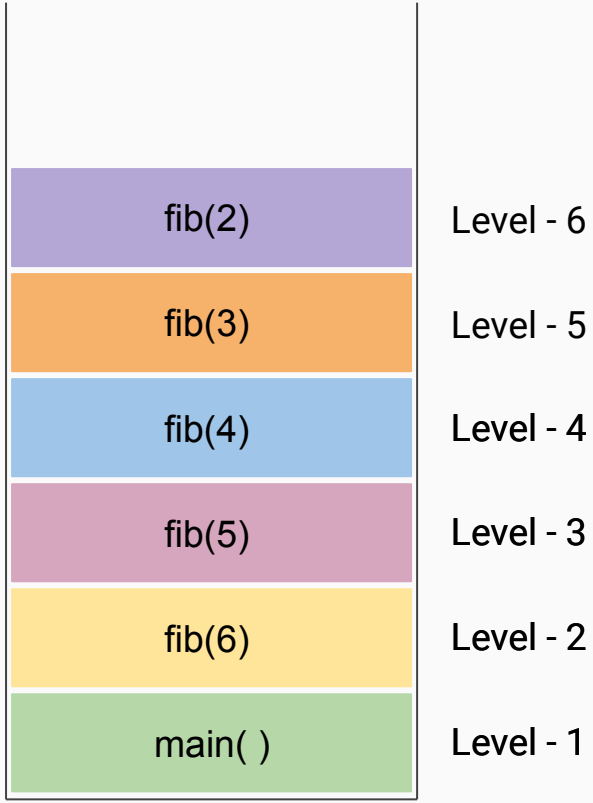
fib(5) = fib(4) + fib(3)



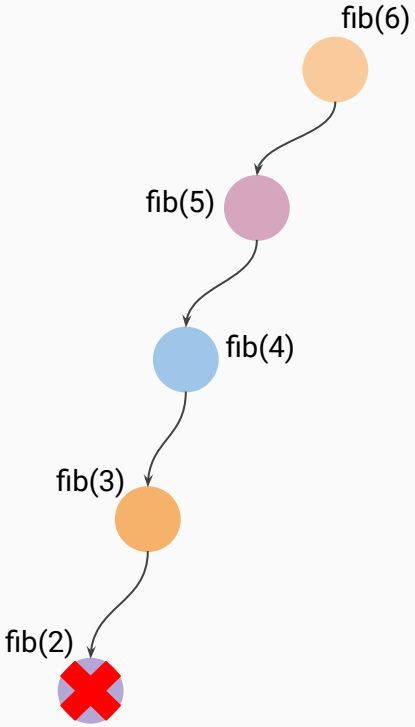


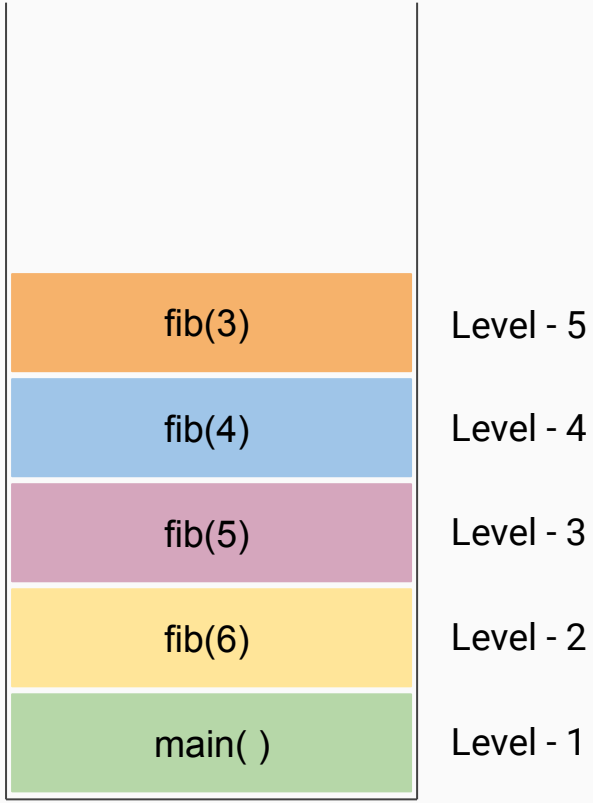
fib(4) = fib(3) + fib(2)



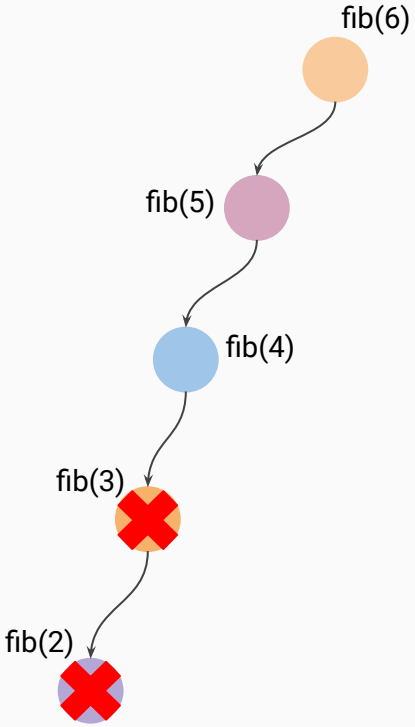


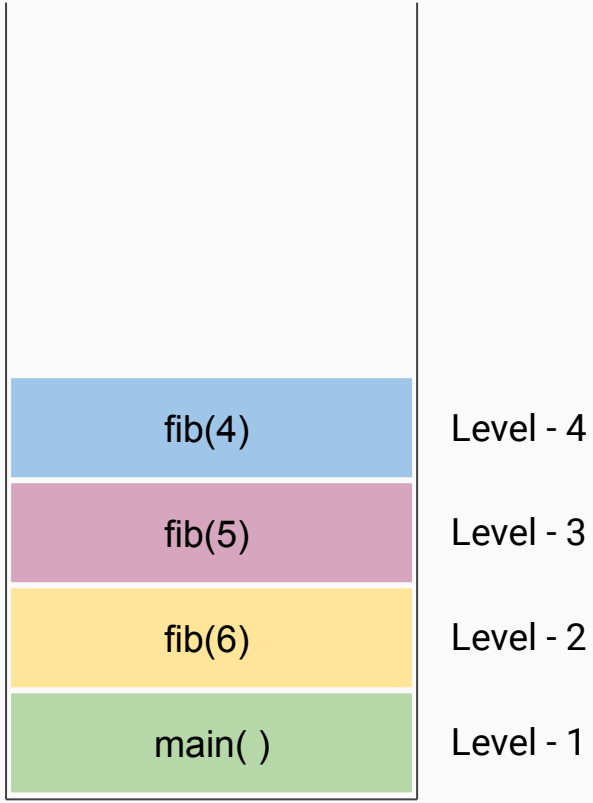
fib(3) = fib(2) + fib(1)



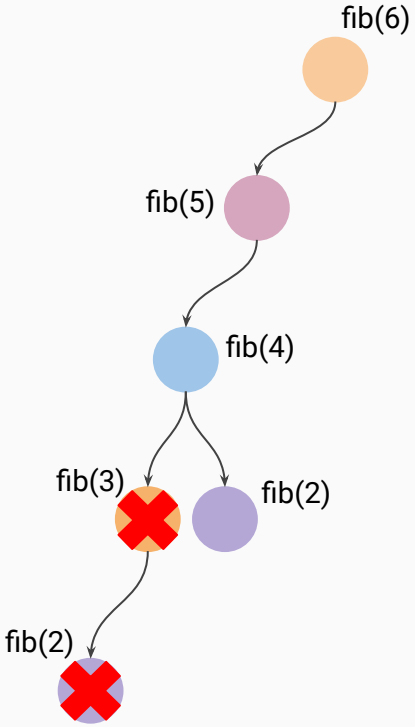


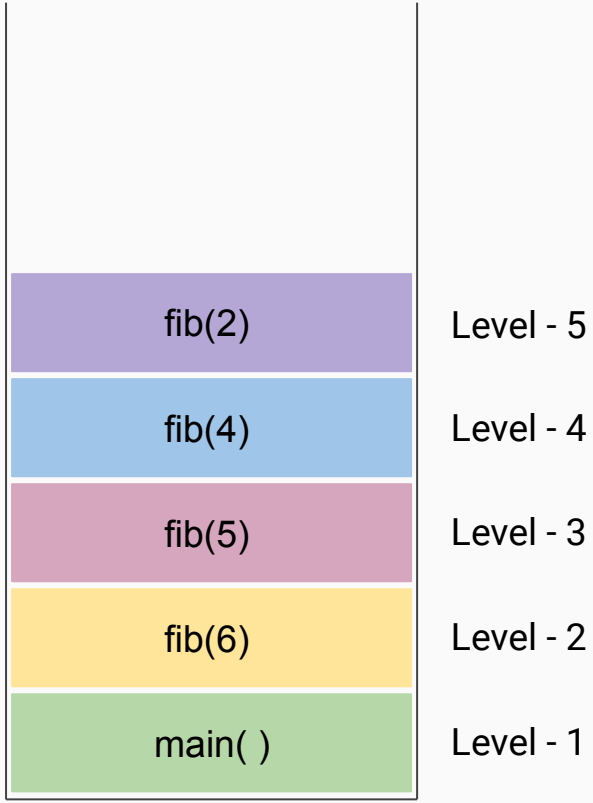
fib(3) = fib(2) + fib(1)



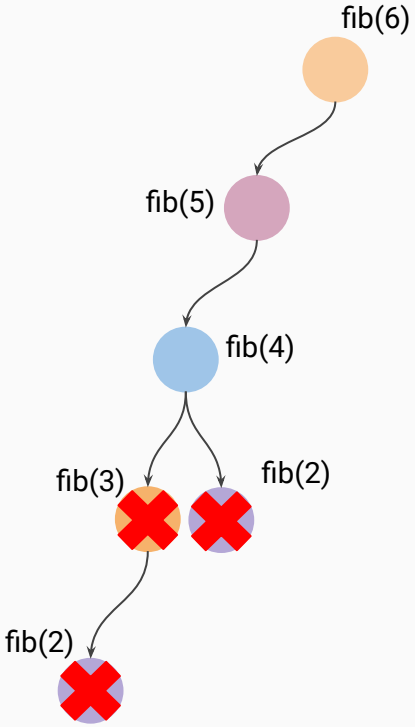


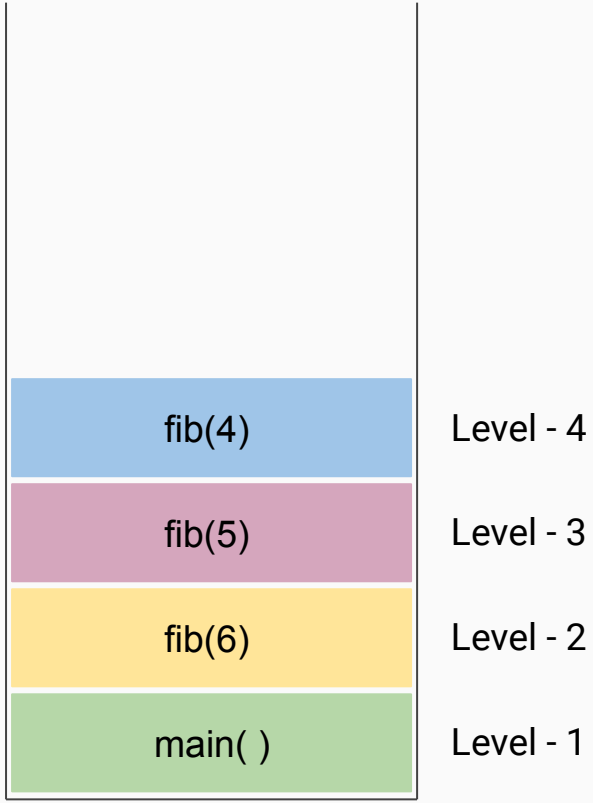
fib(4) = fib(3) + fib(2)



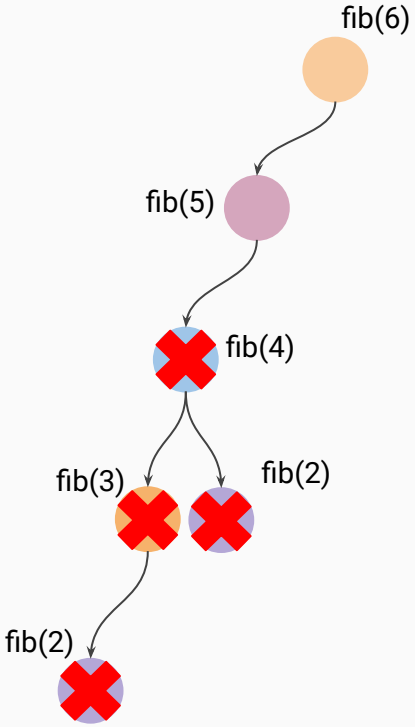


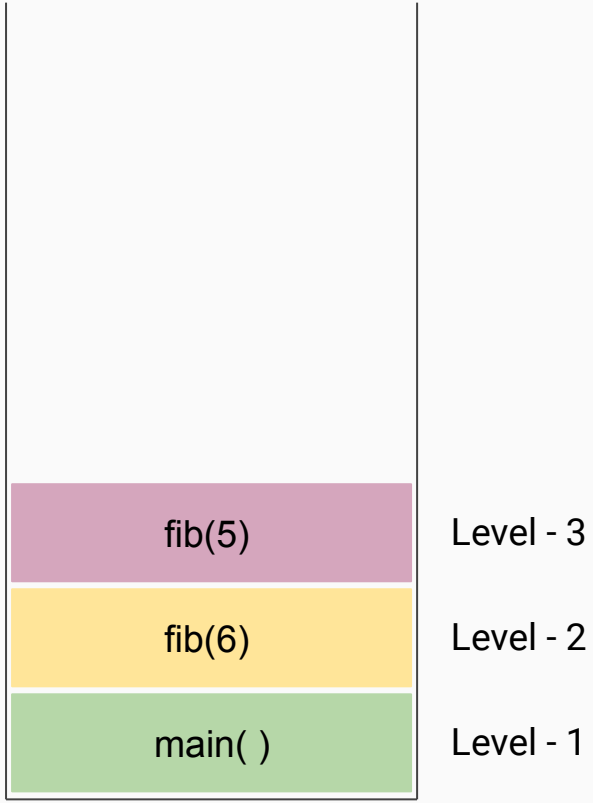
fib(4) = fib(3) + fib(2)



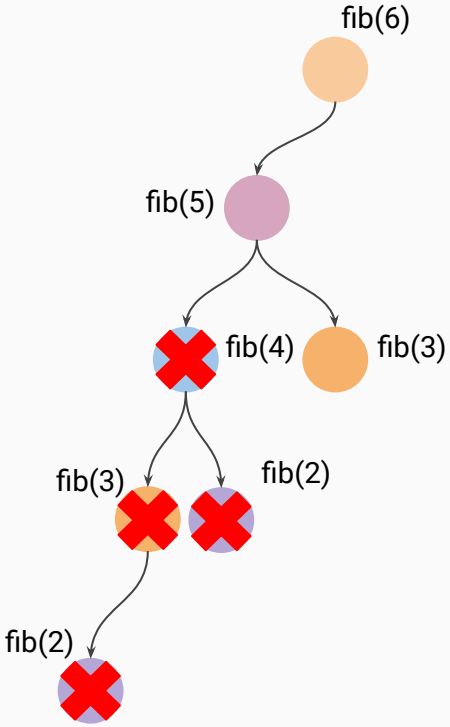


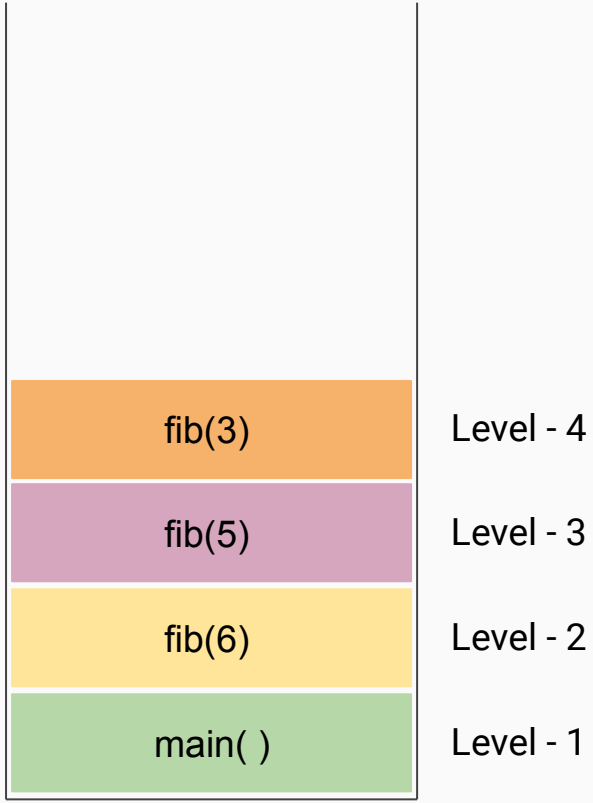
fib(4) = fib(3) + fib(2)



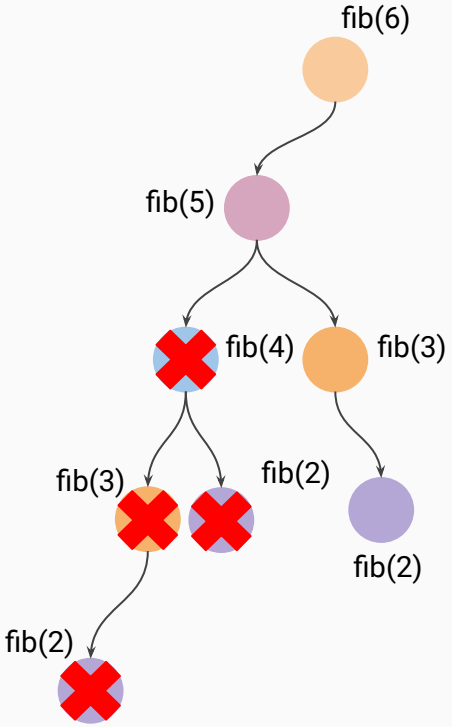


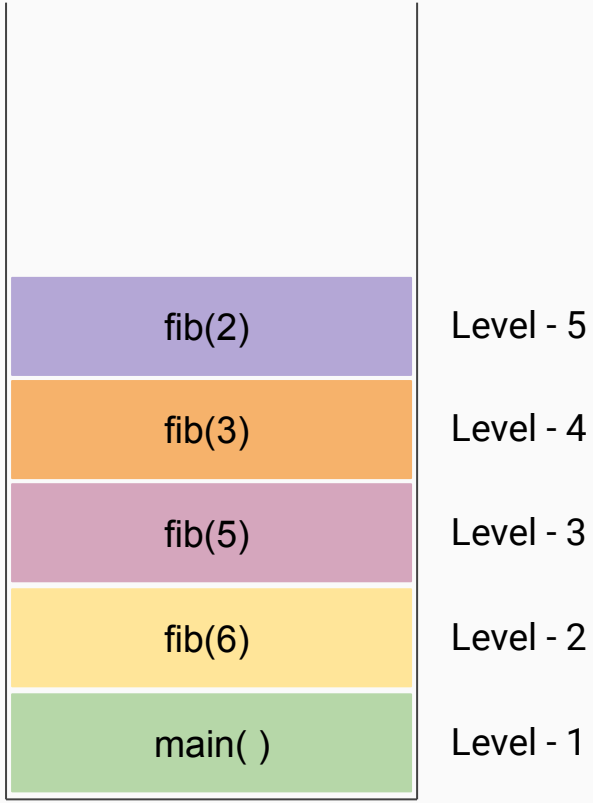
fib(5) = fib(4) + fib(3)



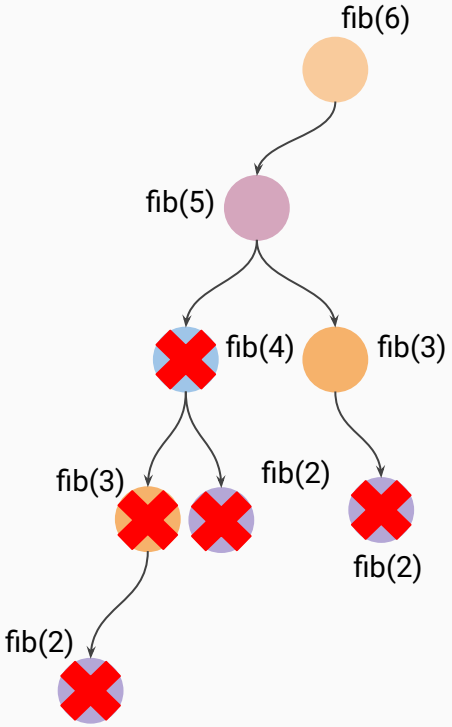


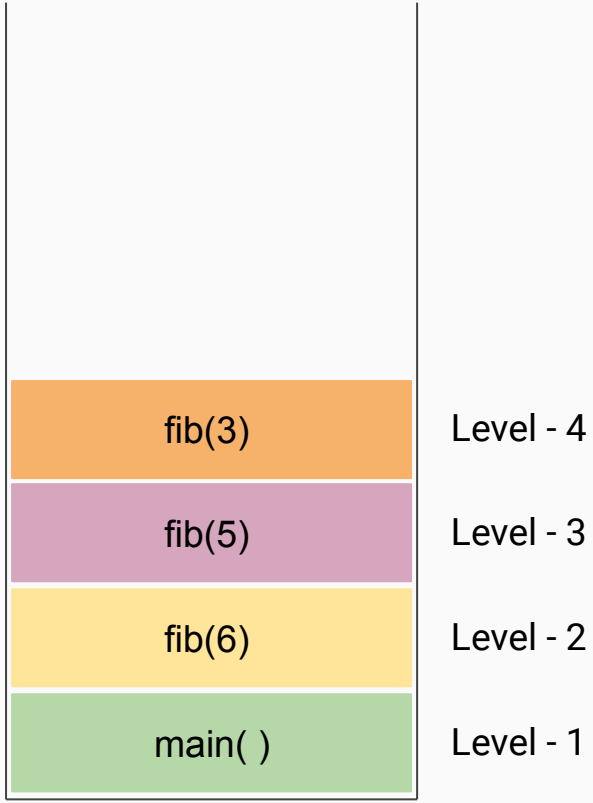
fib(3) = fib(2) + fib(1)



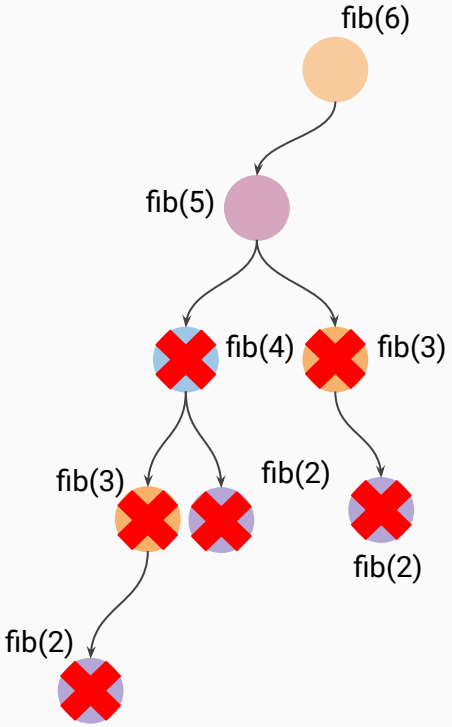


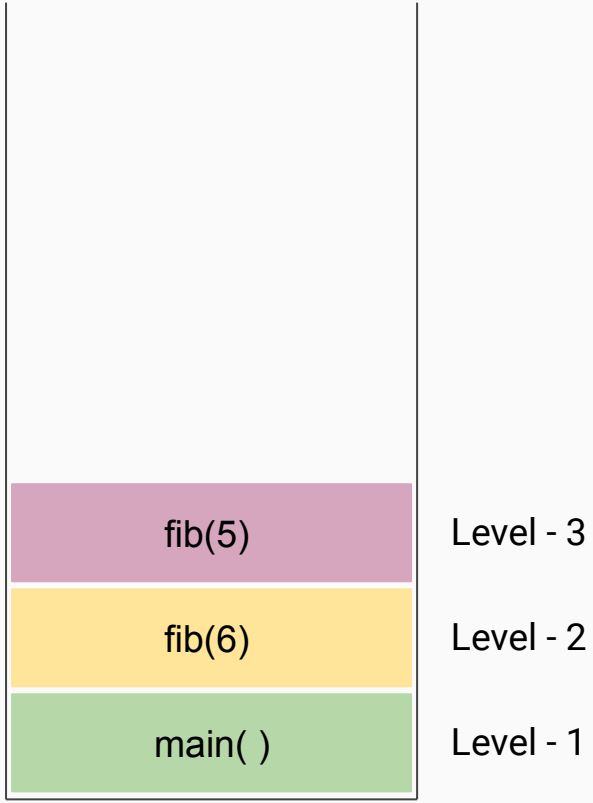
fib(5) = fib(4) + fib(3)



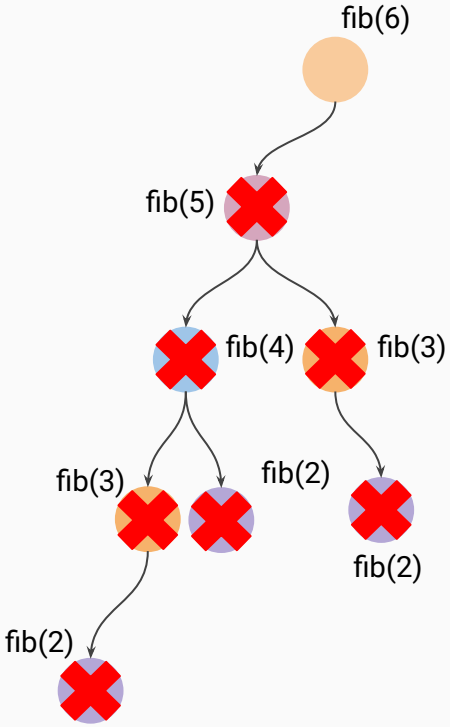


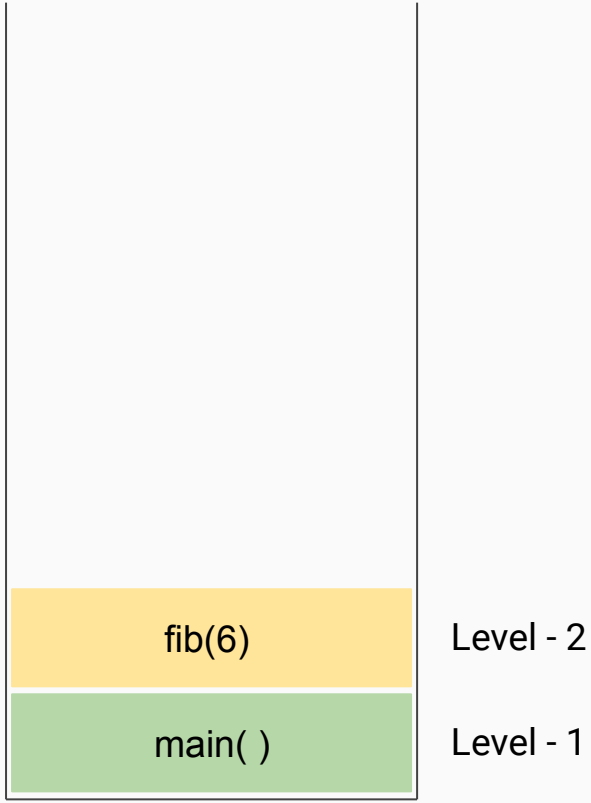
fib(5) = fib(4) + fib(3)



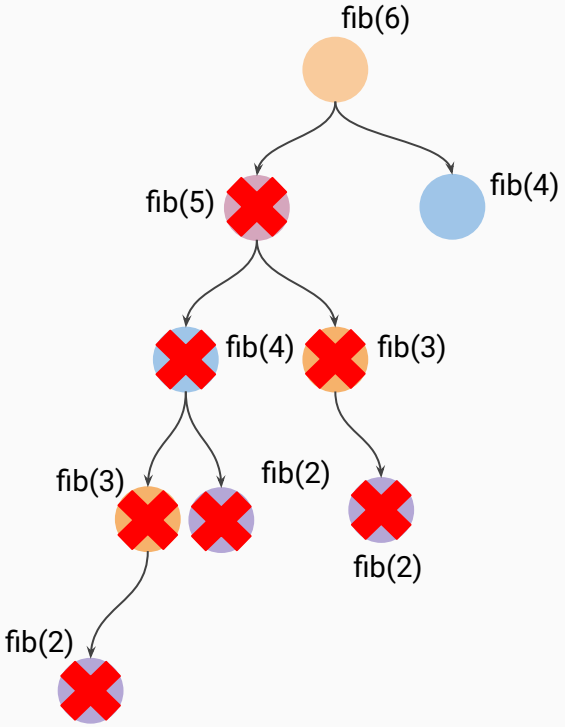


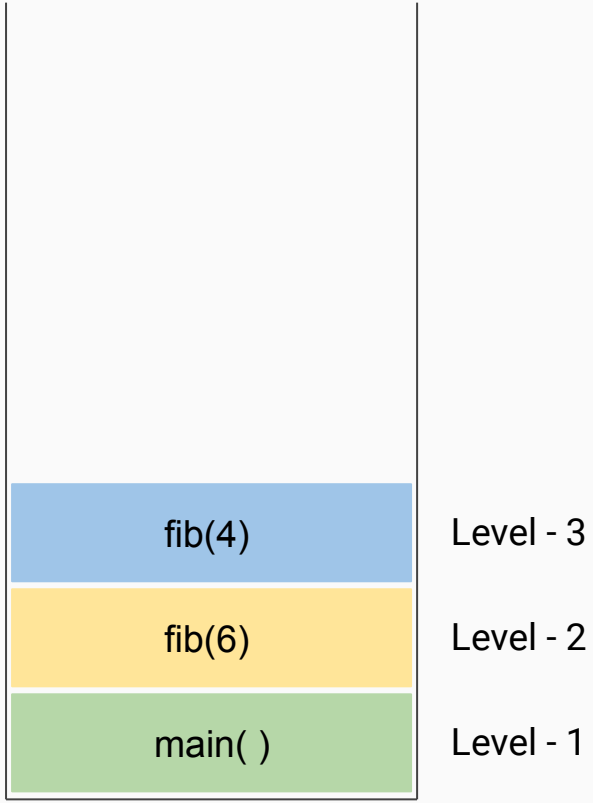
fib(6) = fib(5) + fib(4)



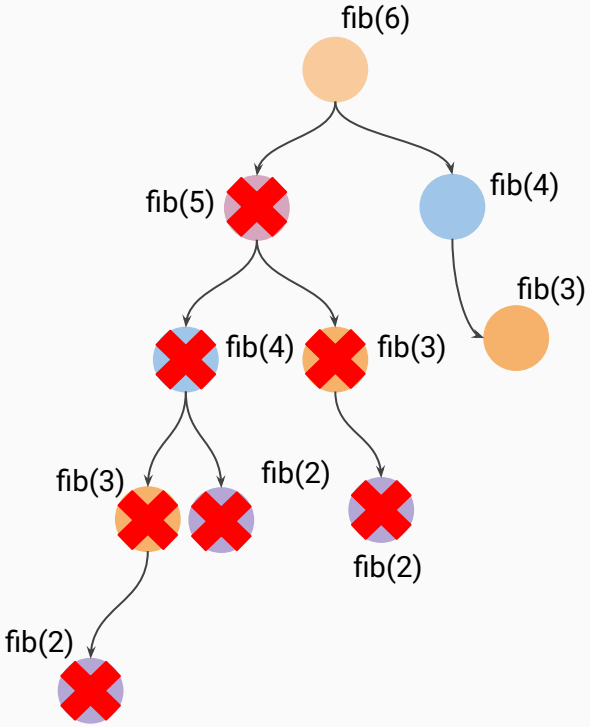


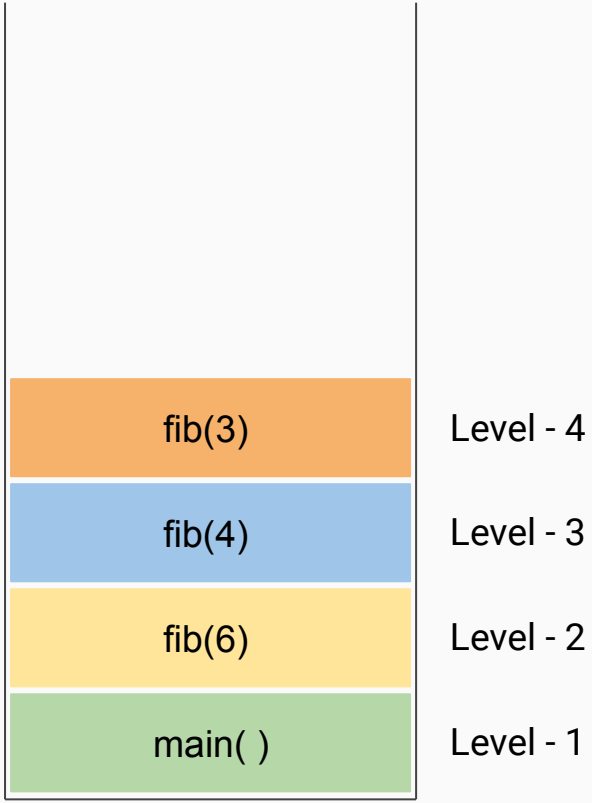
fib(6) = fib(5) + fib(4)



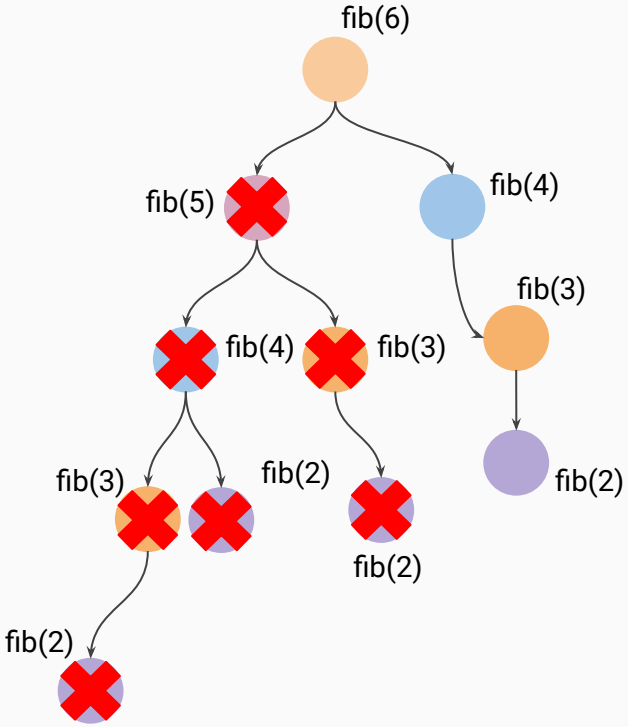


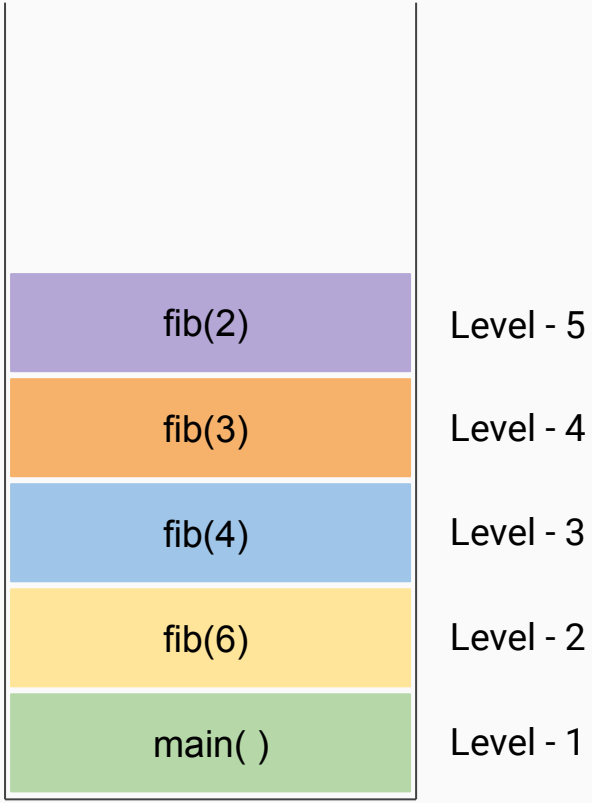
fib(4) = fib(3) + fib(2)



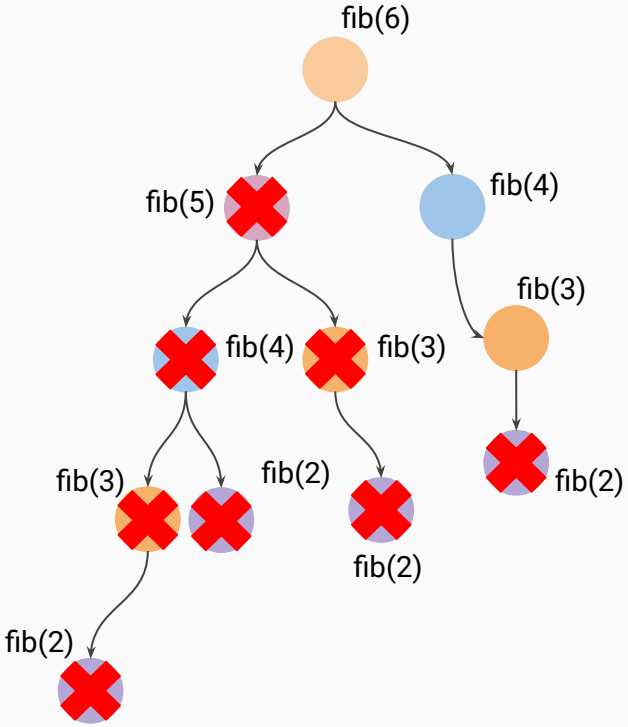


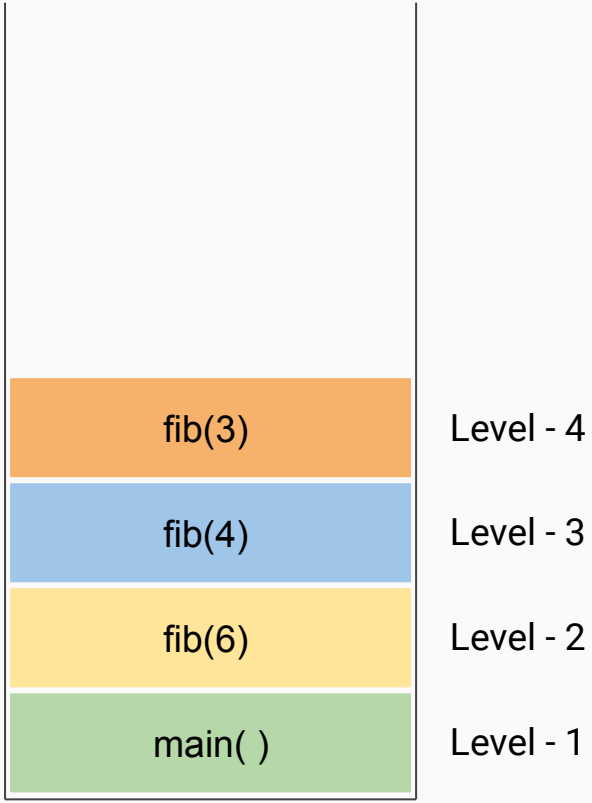
fib(3) = fib(2) + fib(1)



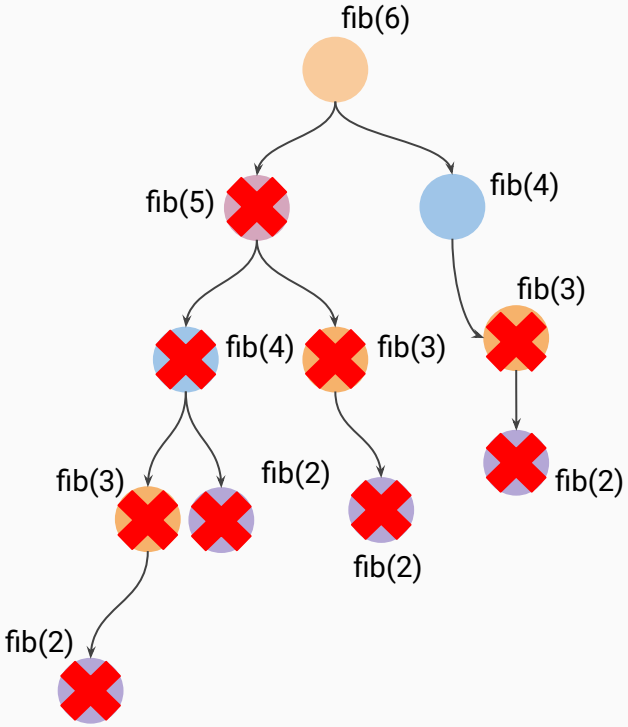


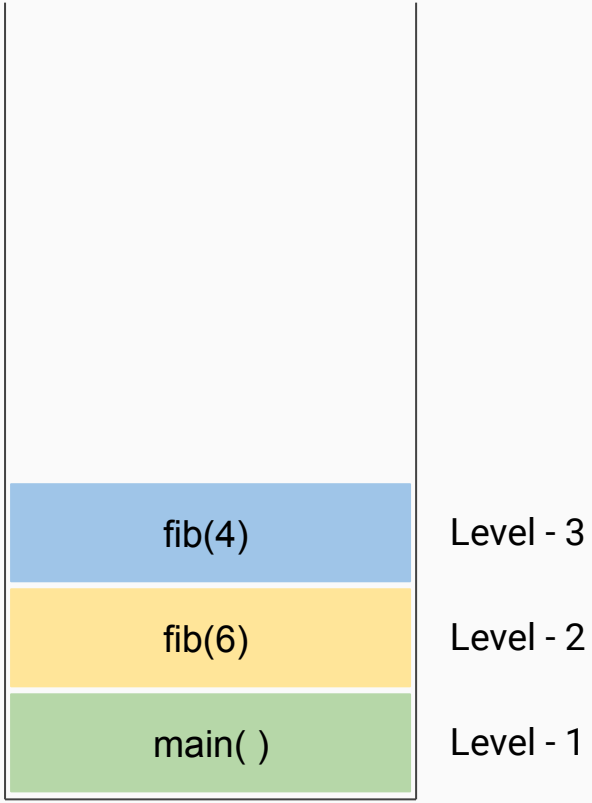
fib(3) = fib(2) + fib(1)



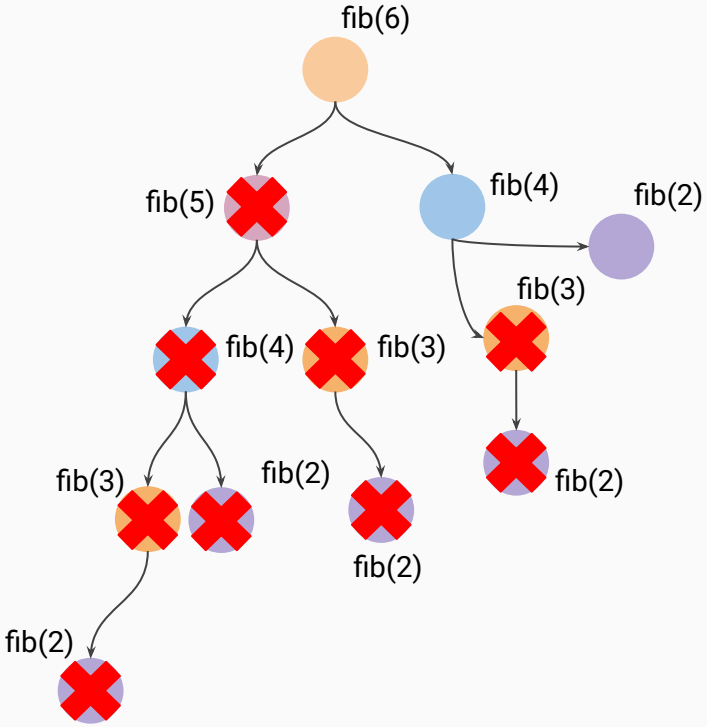


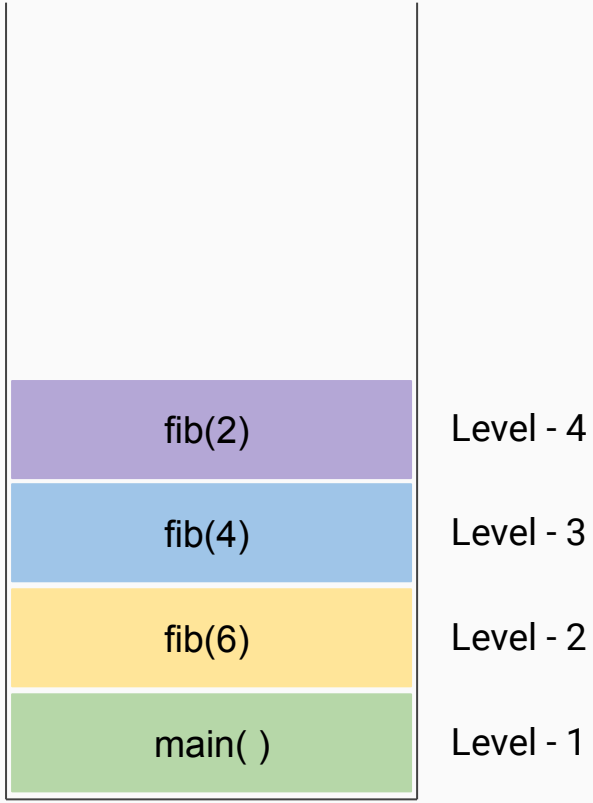
fib(3) = fib(2) + fib(1)



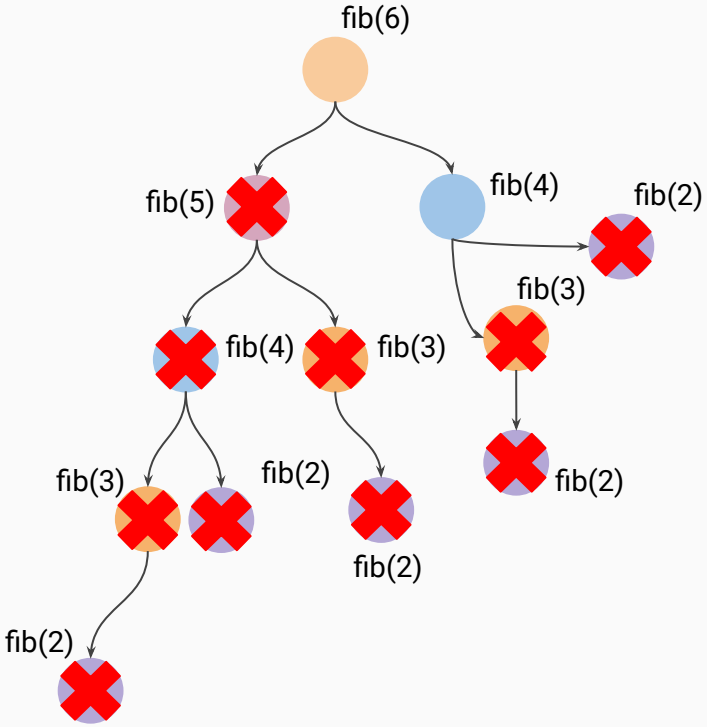


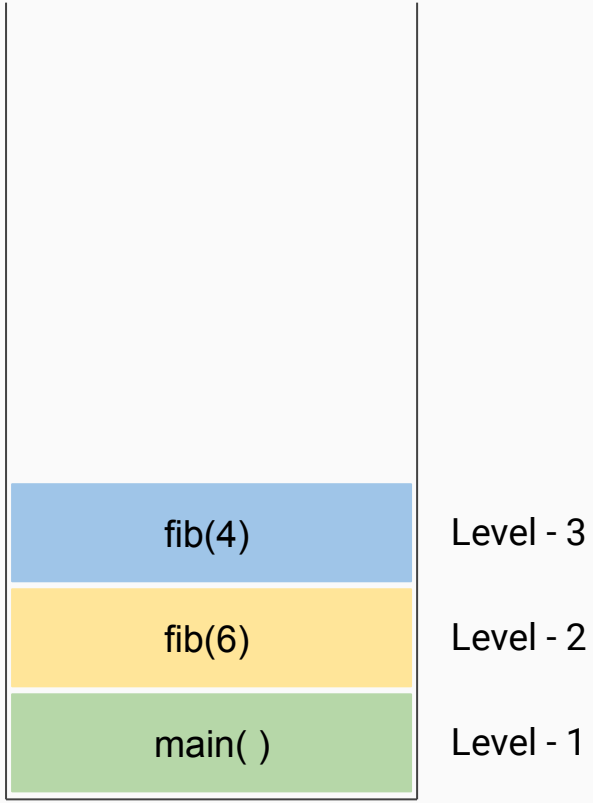
fib(4) = fib(3) + fib(2)



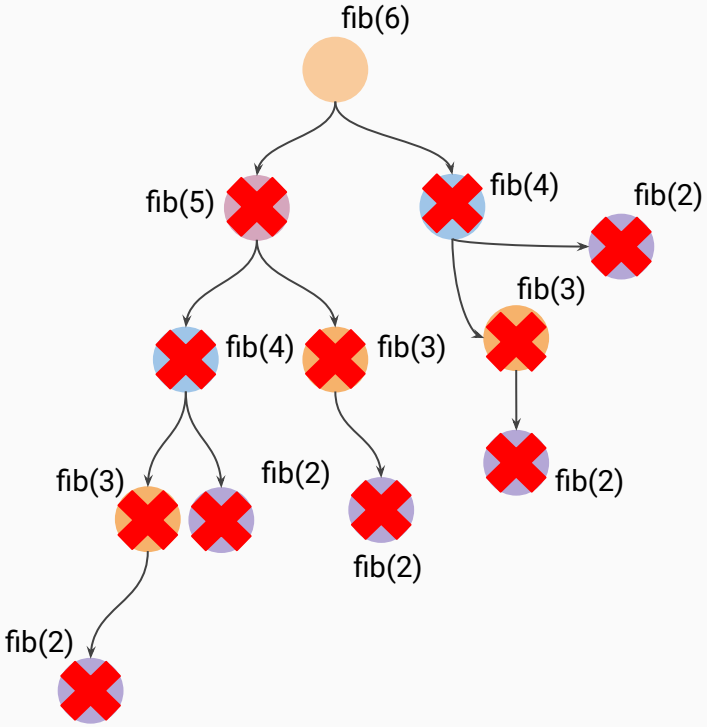


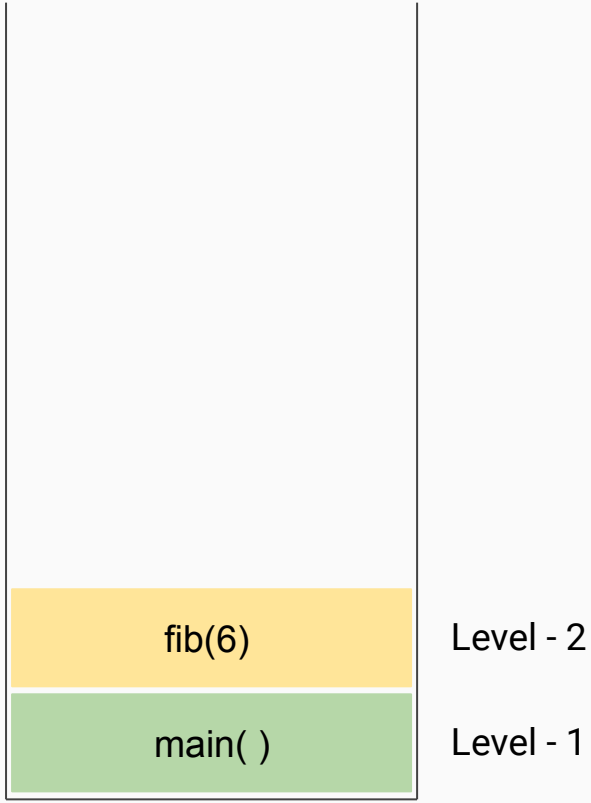
fib(2) = fib(1) + fib(0) = 1



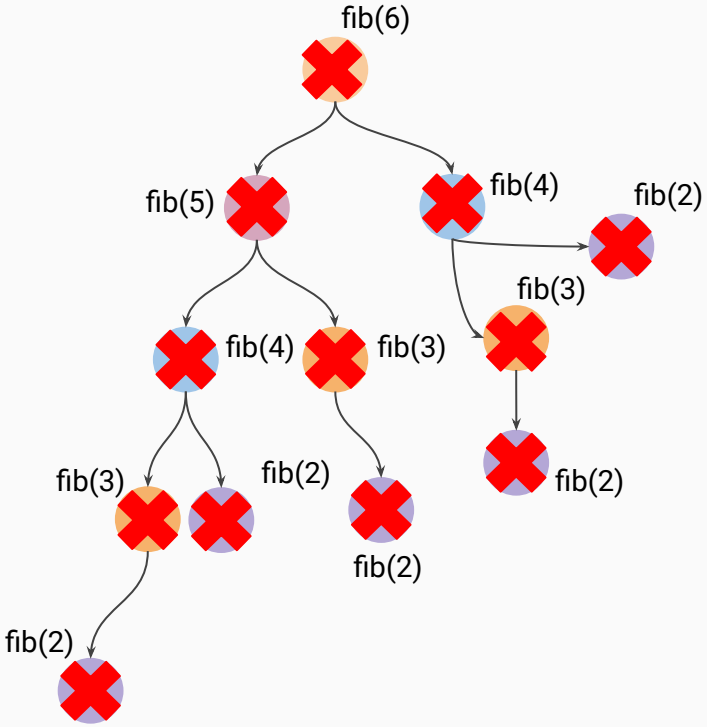


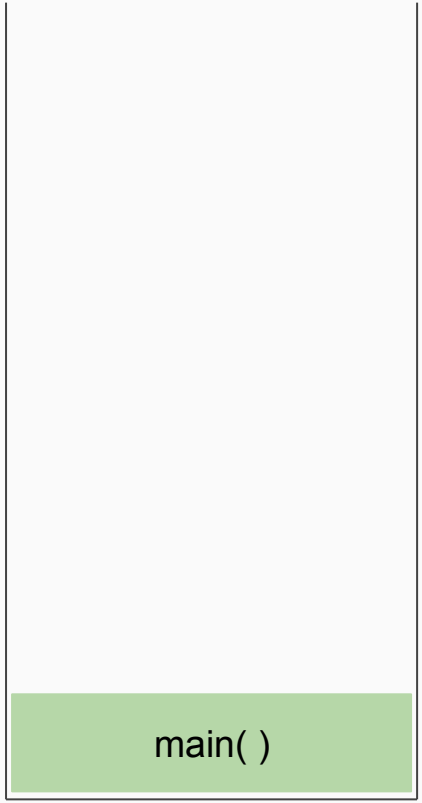
fib(4) = fib(3) + fib(2)





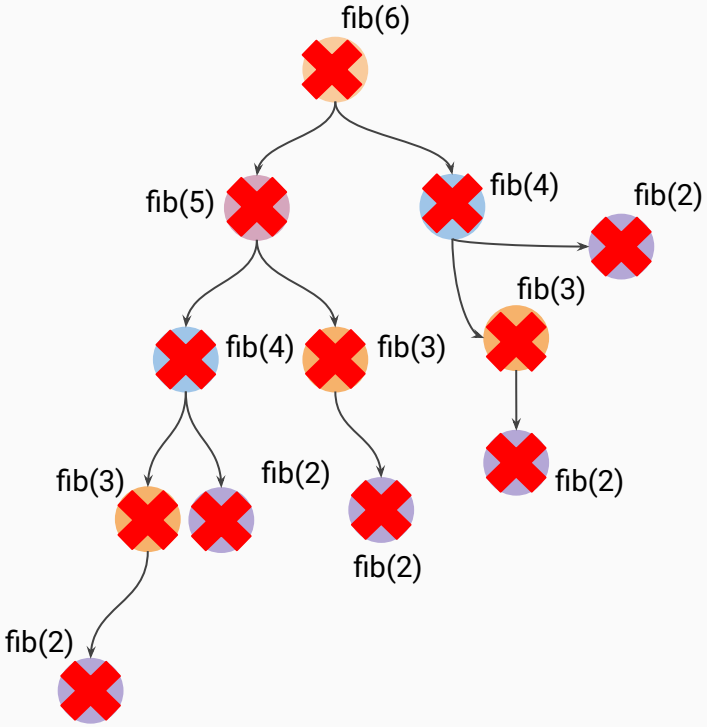
fib(6) = fib(5) + fib(4)

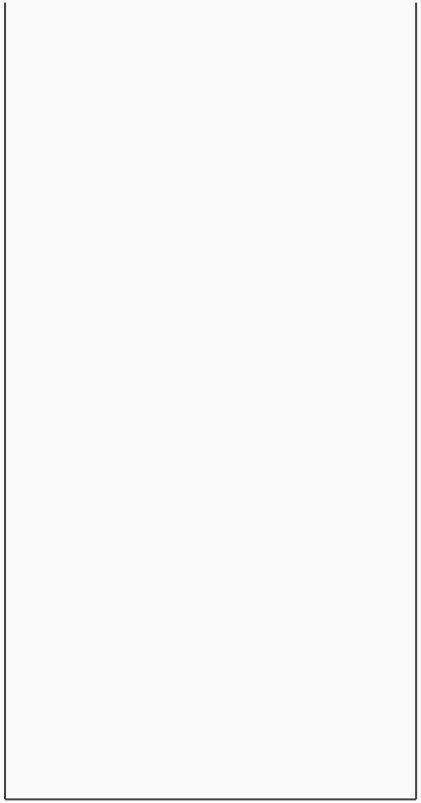




Level - 1

main( ) returns the value of fib(6) and gets popped





Maximum number of levels in the stack frame = 6 = height of the fibonacci tree or the maximum depth of the fibonacci tree.

$\therefore$ The space complexity = O(h); where h = height of the fibonacci tree

A better approach

- Iterative version

Pseudo Code

Iterative version:

```
Function fibo(n):
    Begin
        If n < 2
            Return n
        Declare 3 variables to store the values of
            → Previous Previous Number
            → Previous Number
            → Current number
        As we iterate through all the numbers from 1 to n we will store
        updated the values of declared values.

        Once the loop ends we return the present value of current number.
    End
```

Analysis

- The Iteration method would be the preferable and faster approach to solving our problem because we are storing the first two of our Fibonacci numbers in two variables (previous previous Number, previous Number) and using "Current Number" to store our Fibonacci number. Storing these values prevent us from constantly using memory space in the Stack. Thus giving us a time complexity of **O(n)**.

- All the space we use is to store 3 variables.

- This reduces the usage of STACKS which are implemented using nodes and pointers and even prevents all the push and pop business.

- Both the space and time complexity are optimized in this method.

- This method belongs to the dynamic programming class and called memoization.

C code for both the above versions

To view the C code for the recursive version click on the link here

To view the C code for the iterative version click on the link here

Backtracking Algorithms

Krishna has recently shifted into a new home and is not familiar with the routes of the colony. Now he wants to visit a bakery but he doesn't know the way. What can he do?

Well, a simple solution is to go and ask the neighbours about any nearby bakery. But let us say that all the neighbours are unfriendly and are not willing to tell him the route.

So, now he have to explore all the routes from his home and check which route leads him to a bakery. The routes which doesn't lead him to a bakery will be marked read and routes which will lead him to a bakery will be marked green.

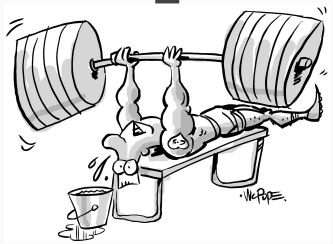
Let us visualize this.


publicdomainvectors.org


SCHOOL

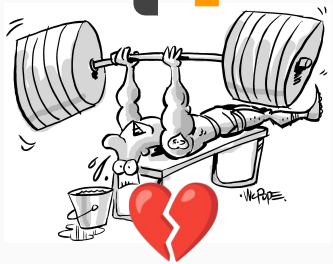

publicdomainvectors.org


SCHOOL




publicdomainvectors.org


SCHOOL

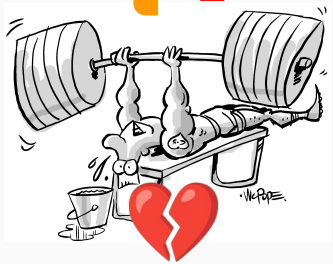



publicdomainvectors.org


SCHOOL

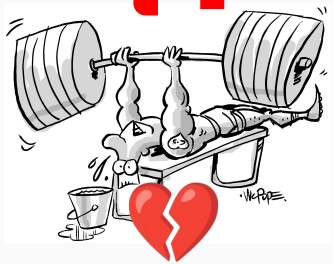

publicdomainvectors.org


SCHOOL



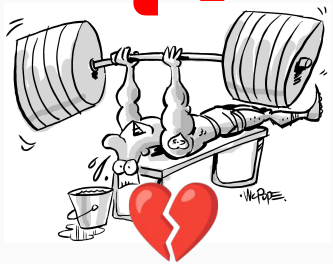

publicdomainvectors.org


SCHOOL



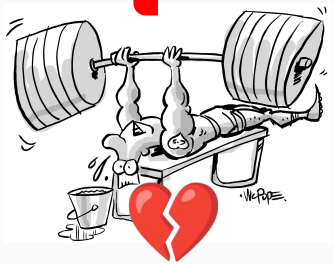

publicdomainvectors.org


SCHOOL




publicdomainvectors.org


SCHOOL

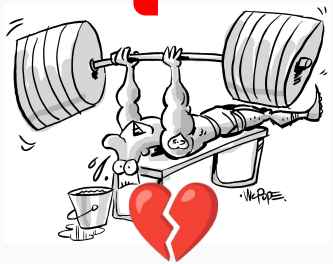



publicdomainvectors.org


SCHOOL

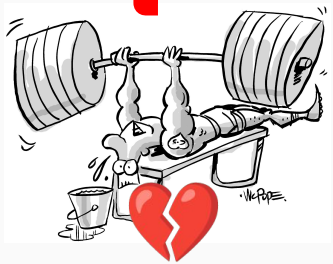

publicdomainvectors.org


SCHOOL




publicdomainvectors.org


SCHOOL



The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach.

This approach is used to solve problems that have multiple solutions. If you want an optimal solution, you must go for dynamic programming.

Backtracking algorithm

```
Backtrack(x)
    if x is not a solution
        return false
    if x is a new solution
        add to list of solutions
     backtrack(expand x)
```

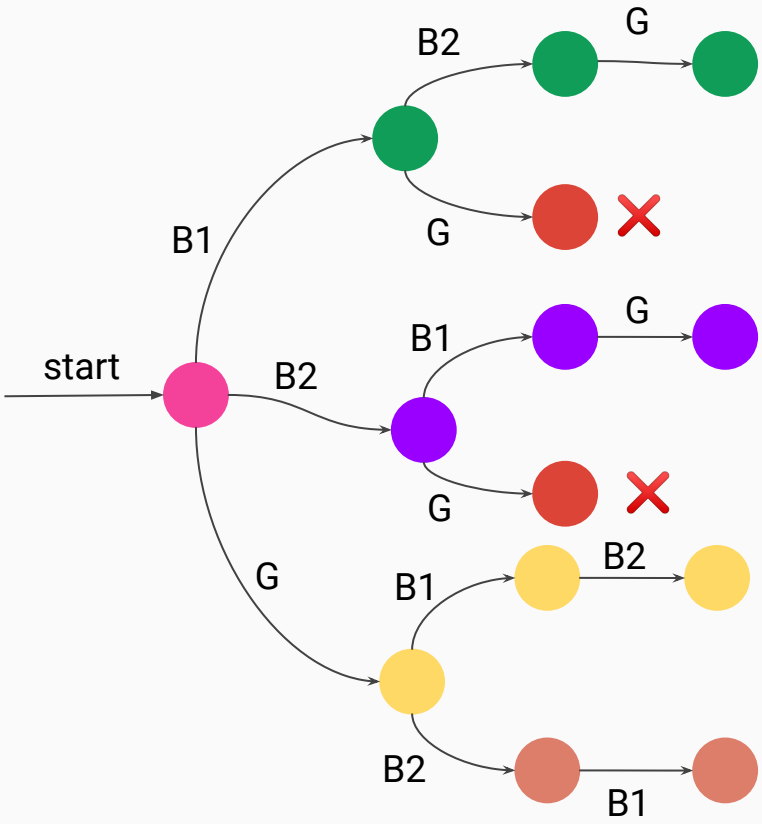
Let’s go with a simple example

Problem: You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches.

Constraint: Girl should not be in the middle of the bench

Possible outcomes

Considering our constraints



Correct outcomes

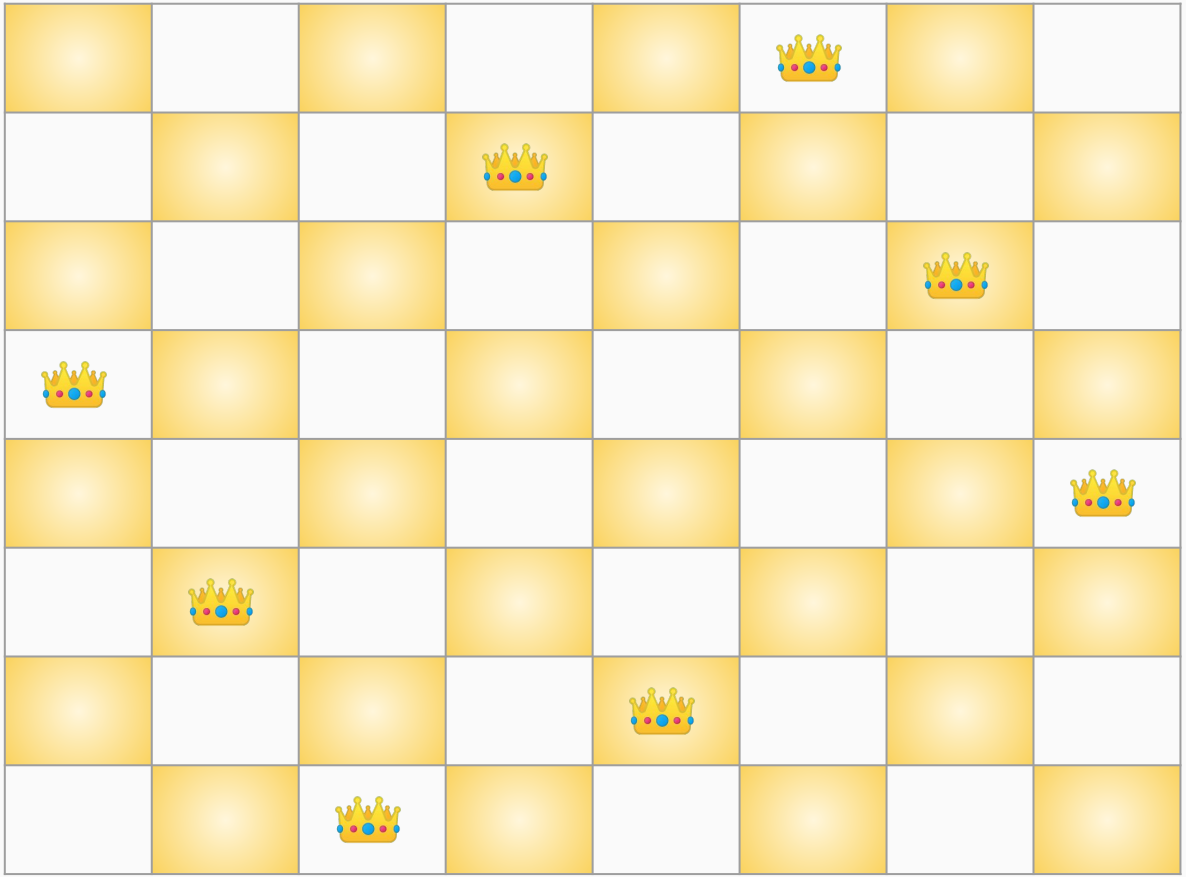
N - Queen problem 👑

What is N - Queen problem ?

N-Queen is the game of placing N chess queens on a N×N chess board such that none of the attack each other.

This kind of arrangement is possible only for N > 3

One of the possible arrangements for 8-Queen



Naive approach

```
while there is a unchecked configuration
{
    generate the next configuration:
    if queens don't attack in this configuration
    then
    {
        print this configuration;
    }
}
```

Is that the best approach ?

Total number of configurations generated to test on the board (8*8):

$$^{64}C_{8} = 4{,}426{,}165{,}368$$

Successful configurations : 92

Backtracking approach

- The idea is to place queens one by one in different columns, starting from the leftmost column.

- When we place a queen in a column, we check for clashes with already placed queens.

- In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution.

- If we do not find such a row due to clashes then we backtrack and return false.

Start in the leftmost column
2) If all queens are placed
    return true
3) Try all rows in the current column.
   Do following for every tried row.
    a) If the queen can be placed safely in this row
       then mark this [row, column] as part of the
       solution and recursively check if placing
       queen here leads to a solution.
    b) If placing the queen in [row, column] leads to
       a solution then return true.
    c) If placing queen doesn't lead to a solution then
       unmark this [row, column] (Backtrack) and go to
       step (a) to try other rows.
3) If all rows have been tried and nothing worked,
    return false to trigger backtracking.

Divide and conquer algorithms

A divide and conquer algorithm is a strategy of solving a large problem by

1. Breaking the problem into smaller sub problems -- **Divide**
2. Solving the sub-problems, and -- **Conquer**
3. Combining them to get the desired output. -- **Combine**

To use the divide and conquer algorithm, recursion is used.

Merge Sort

- A popular application of divide and conquer approach
- Sorting algorithm

Our unsorted array: A

Our task is to sort the above unsorted array.

Q. What can a subproblem of this ?

A subproblem would be to sort a sub section of this array starting at index ‘p’ and ending at index ‘r’, denoted as A[p……r].

Let us solve this step by step using all the three of the parts : Divide, conquer and combine

Divide:

Let index ‘q’ is the half-way point between index ‘p’ and index ‘r’. Now we will divide the unsorted array into two parts.

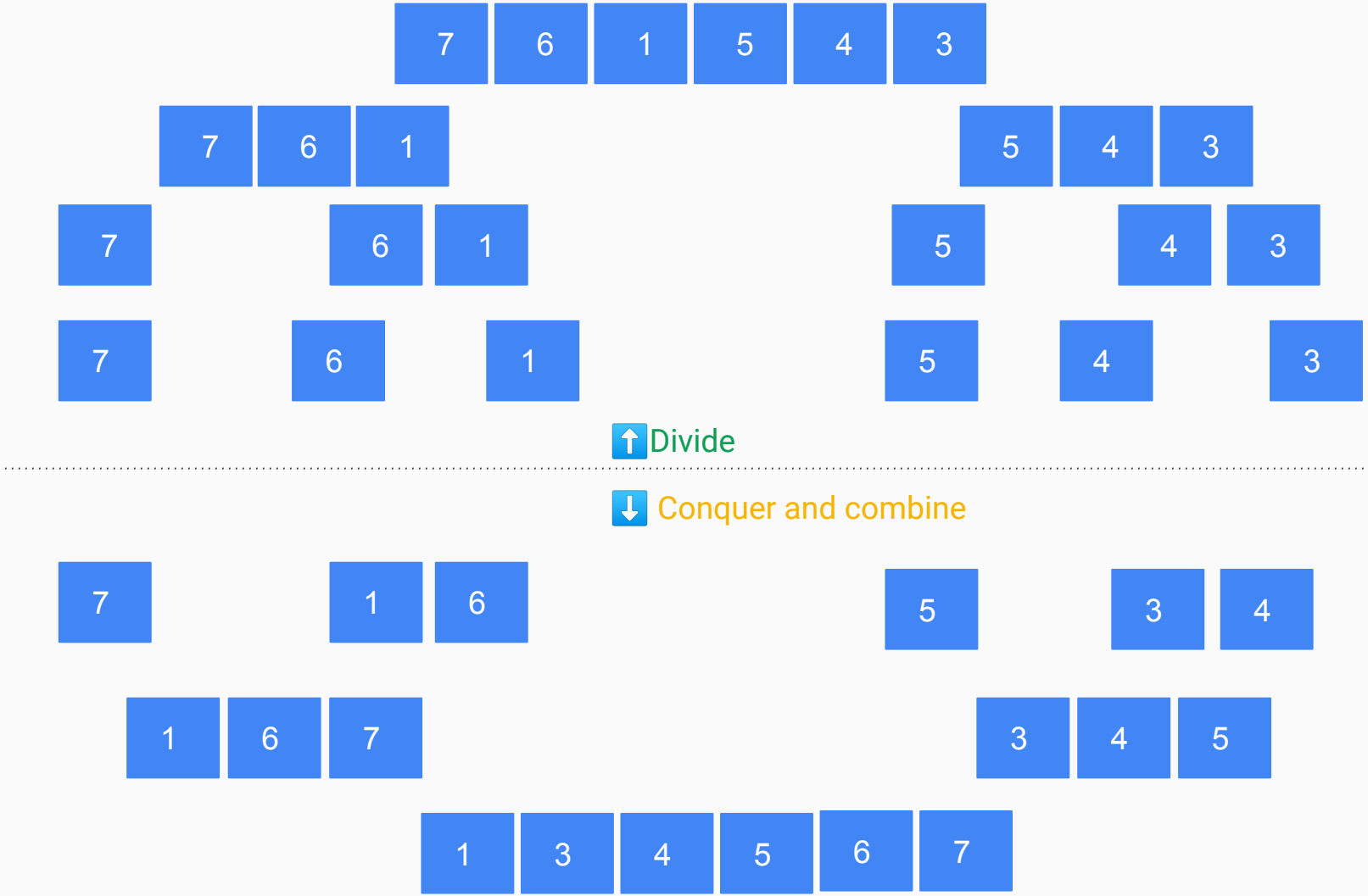
One of the part as: A[p……q]
Other part: A[(q+1).....r]

Conquer:

- In the conquer step, we try to sort both the subarrays A[p……q] and A[(q+1).......r].
- If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine:

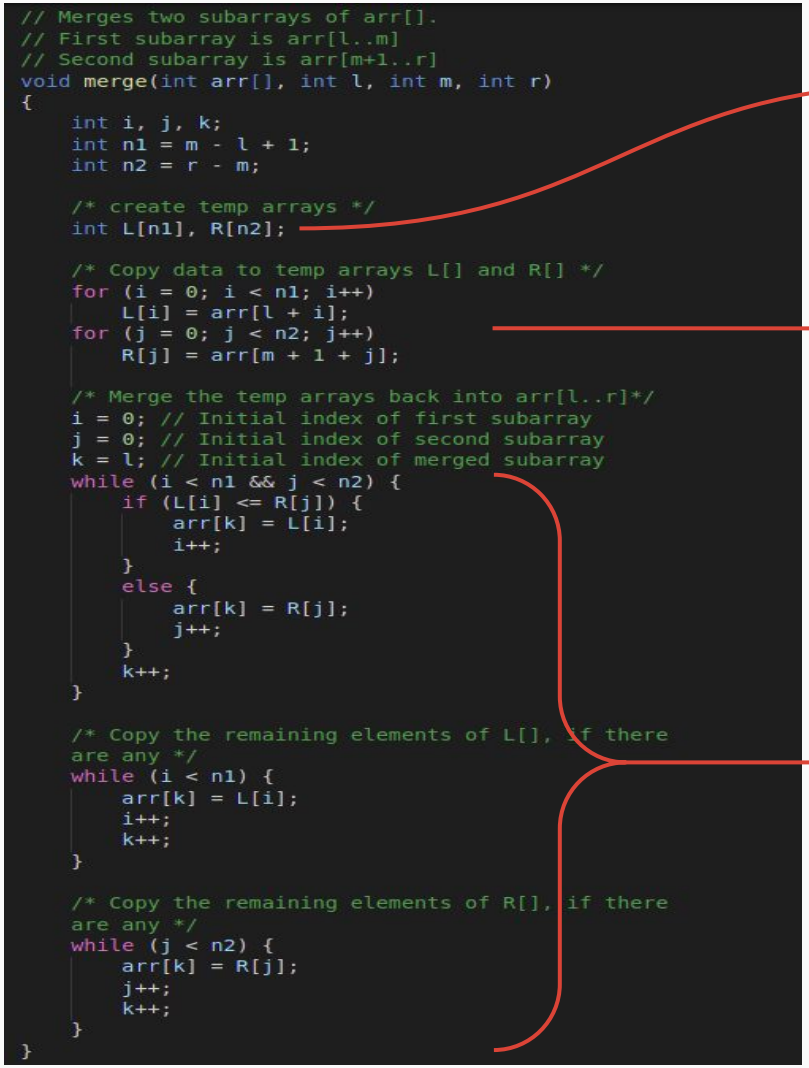
When the conquer step reaches the base step and we get two sorted subarrays A[p….q] and A[(q+1)..r] for array A[p……r], we combine the results by creating a sorted array A[p……r] from two sorted subarrays A[p….q] and A[(q+1)..r].


7 6 1 5 4 3
7 6 1
5 4 3
7
6 1
5
4 3
7
6
1
5
4
3
⬆️Divide
⬇️ Conquer and combine
7
1 6
5
3 4
1 6 7
3 4 5
1 3 4 5 6 7

Let us have a look at the code

```
/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and r
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

This part of the code selects the middle index of the array or the subarray and start dividing the subparts further till the starting and the ending index become the same i.e., l = r



```
// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
    are any */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
    are any */
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

1

Creating two subarrays L and R of size n1 and n2 respectively where n1 equals to the number of elements from the starting index provided to the middle index.

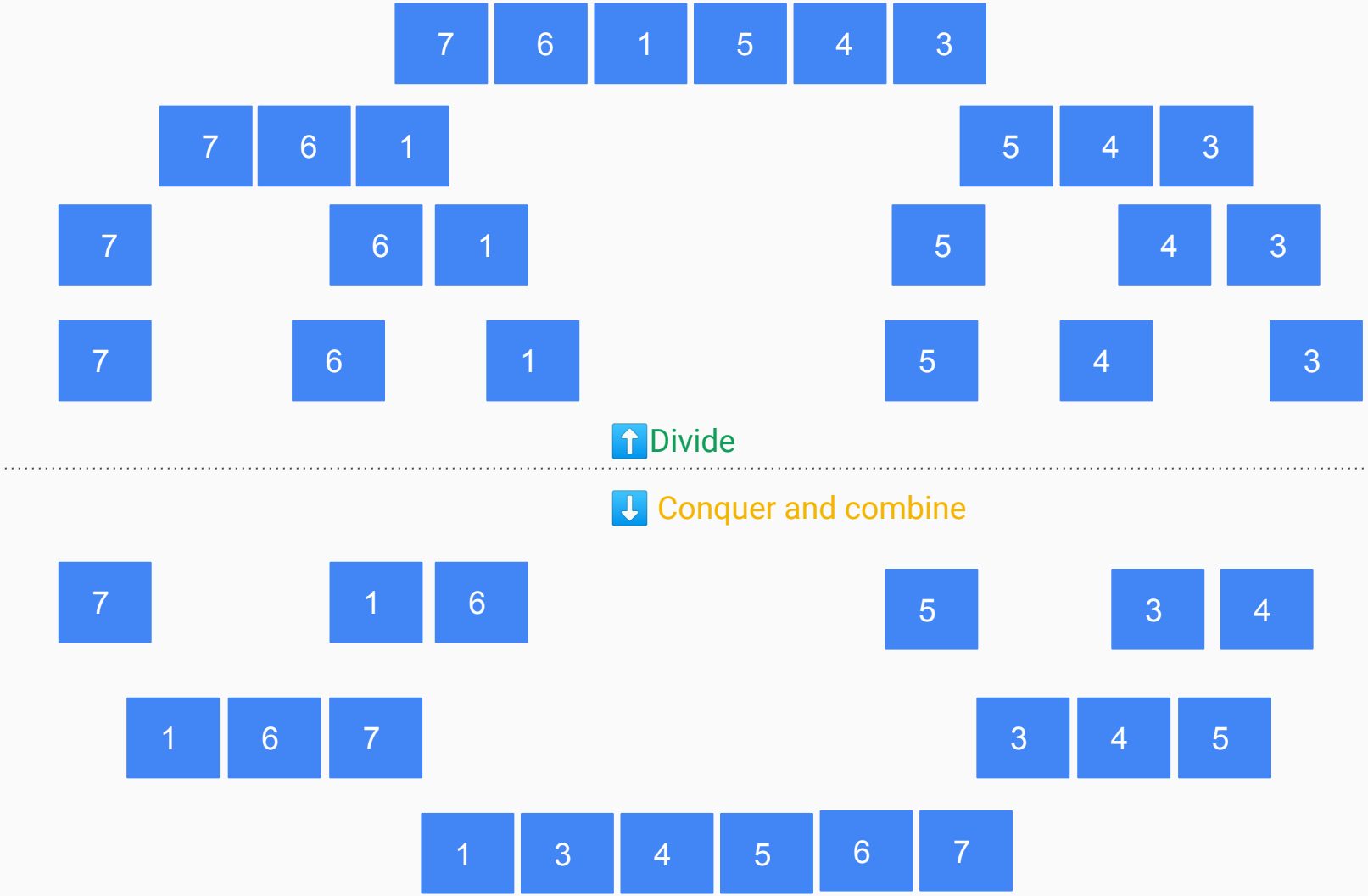
2

Copying the elements from the original array into the subarrays

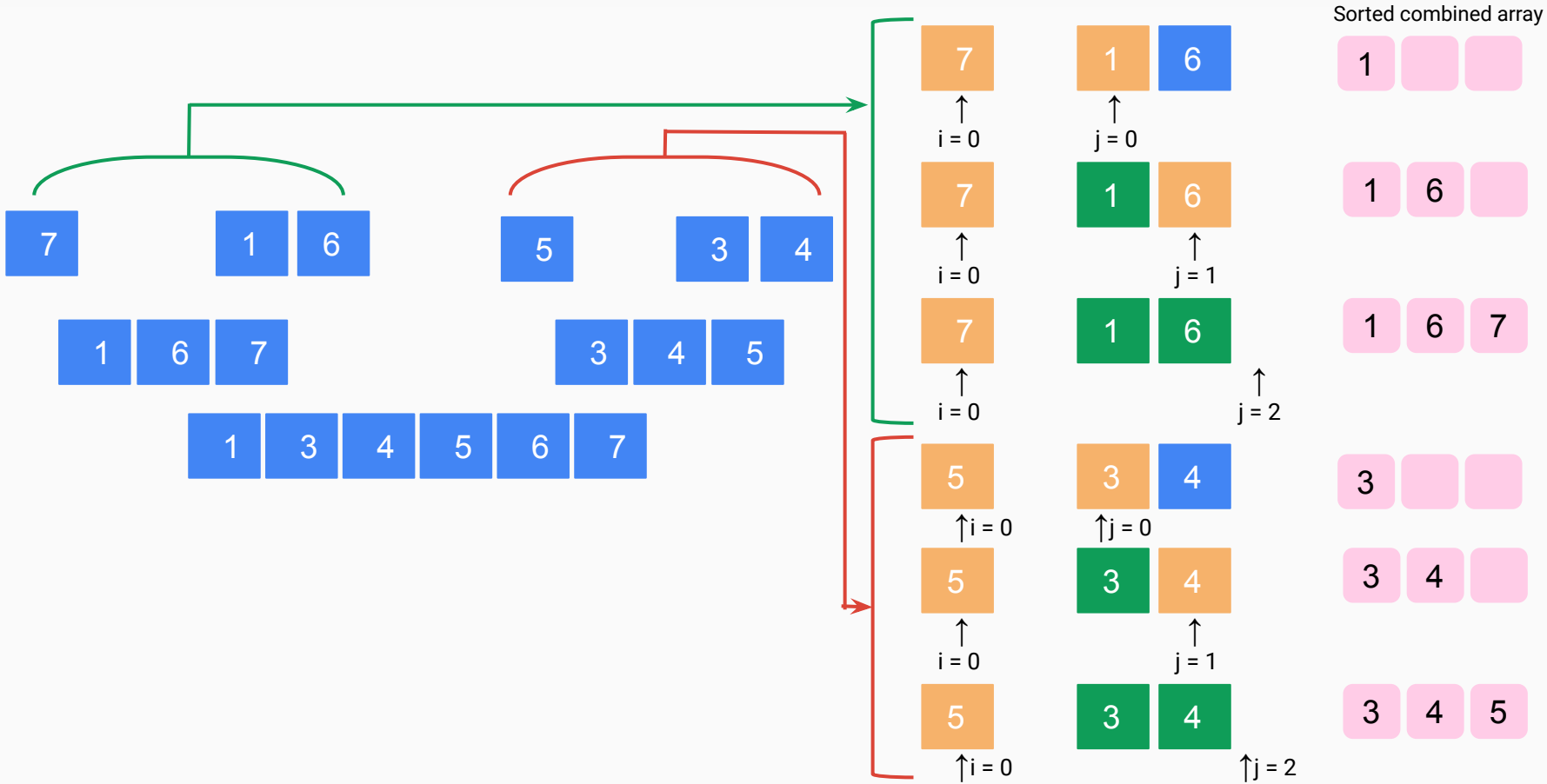
3

If the element at ‘i’th index of the subarray is less than the element at ‘j’ th index of the subarray then we copy the value of L[i] into the merged array and increment the value of ‘i’ by 1 while keeping the value of ‘j’ same. If R[j] is less than L[i] then we copy L[i] into the merged array and increment the value ‘j’ while keeping the value of ‘i’ the same.

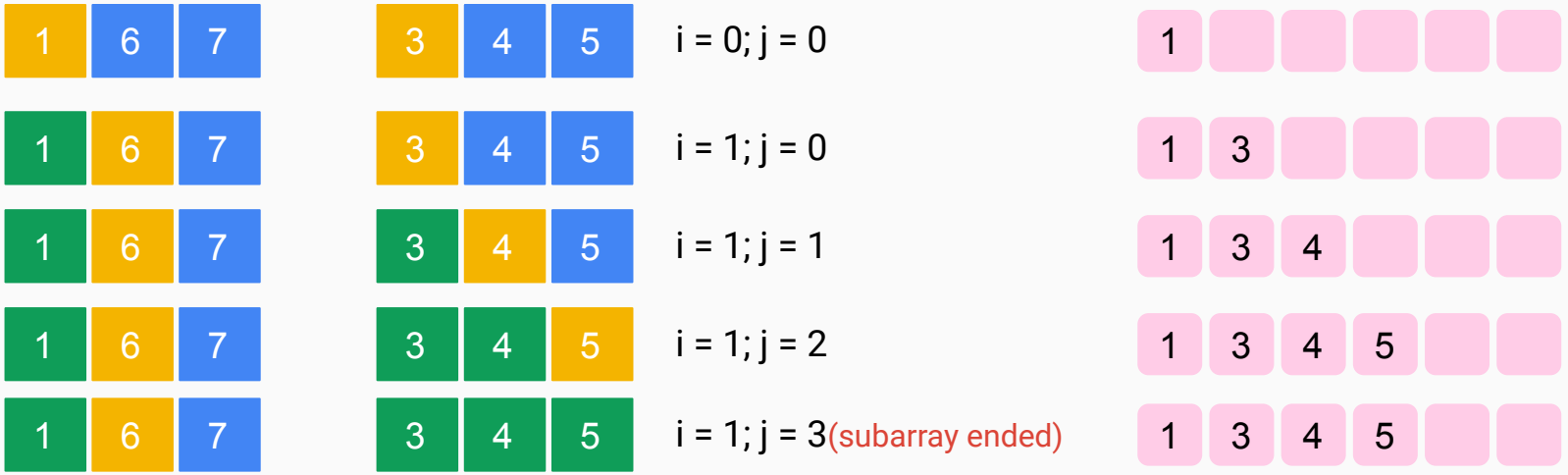
If there are any remaining elements in any of the subarrays we append them to the merged array directly since these parts are already sorted during previous recursions.


7 6 1 5 4 3
7 6 1
5 4 3
7
6 1
5
4 3
7
6
1
5
4
3
⬆️Divide
⬇️ Conquer and combine
7
1 6
5
3 4
1 6 7
3 4 5
1 3 4 5 6 7

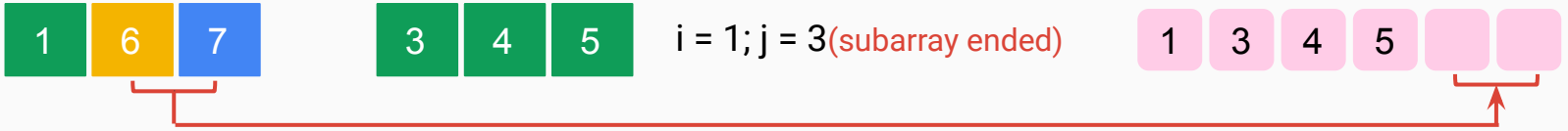
Let us visualize what exactly is happening in the 3rd part of the code.



Now our task is to combine the two sorted subarrays. We again follow the same previous procedure.



Since iterating through one of the subarray has been completed we can just append the remaining elements of the another subarray to the sorted array without reading through each cell since the subarray is already sorted.



Final sorted array: 1 3 4 5 6 7

Master’s theorem

Master theorem is for solving recurrence relation of the form:

$T(n) = aT(n/b) + f(n)$,
where,
n = size of input
a = number of subproblems in the recursion
n/b = size of each subproblem. All subproblems are assumed
to have the same size.
f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem
and cost of merging the solutions

Here, $a \geq 1$ and $b > 1$ are constants, and f(n) is an asymptotically positive function.

If a >= 1 and b > 1 are constants and f(n) is an asymptotically positive function, then the time complexity of recursive relation is given by:

$T(n) = aT(n/b) + f(n)$

where, T(n) has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} * \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$, then $T(n) = \Theta(f(n))$.

$\epsilon > 0$ is a constant.

Time complexity

$T(n) = aT(n/b) + f(n)$
$= 2T(n/2) + O(n)$

Where,
a = 2 (each time, a problem is divided into 2 subproblems)
n/b = n/2 (size of each sub problem is half of the input)

f(n) = time taken to divide the problem and merging the subproblems
T(n/2) = O(n log n) (To understand this, please refer to the master theorem in the previous slides.)

Now, $T(n) = 2T(n \log n) + O(n)$
$\approx O(n \log n)$

Quick Sort

- This is another sorting algorithm which follows divide and conquer approach

Quicksort is again a sorting algorithm based on the divide and conquer approach where

1. An array is divided into subarrays by **selecting** a pivot element (element selected from the array).

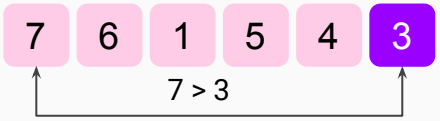
   While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Let us understand this by considering an example.

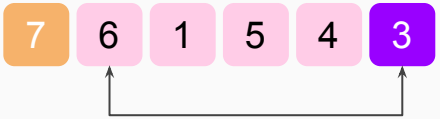
Our unsorted array :

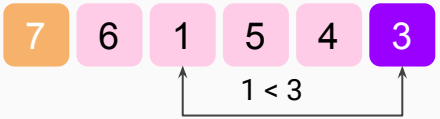
Since we can choose any index as the pivot I will choose the rightmost index to be my pivot.

Now we need to rearrange or position the array before we divide it.

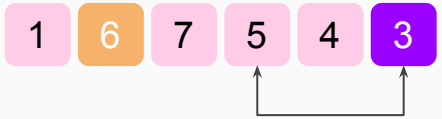


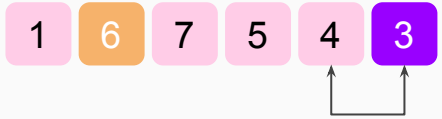
We compare the elements in the array starting from the first index with the pivot. If the element is greater than the pivot element, a second pointer is set for that element.



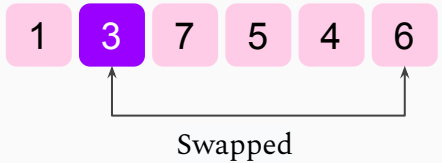


Since we have found an element which is smaller than the pivot we swap it with the second pointer and make the second largest element as the new second pointer

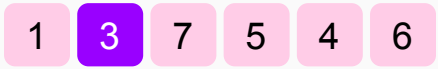




Finally, the pivot element is swapped with the second pointer.



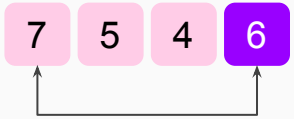
Now we consider the left and right subparts to the pivot and follow the same procedure to position them.



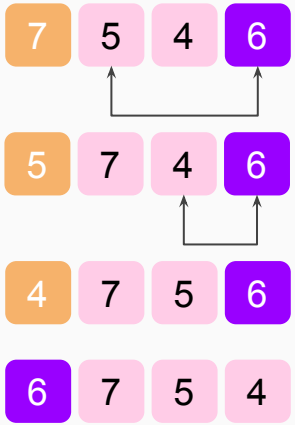
Left subpart

Since that is the only element in the left subpart we can just place it in the array

Right subpart



Since 7 > 6 we declare 7 as the second pointer and move on



New Right subpart

After following the steps

New Right subpart

New left subpart

New Right subpart

We have divided our unsorted into subparts till we have only one element in each subpart.

Now we can simply append them together since they are already sorted.

Sorted array: 1 3 7 5 4 6

Let us have a look at the code

```
// function to find the partition position
int partition(int array[], int low, int high) {

  // select the rightmost element as pivot
  int pivot = array[high];

  // pointer for greater element
  int i = (low - 1);

  // traverse each element of the array
  // compare them with the pivot
  for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {

      // if element smaller than pivot is found
      // swap it with the greater element pointed by i
      i++;

      // swap element at i with element at j
      swap(&array[i], &array[j]);
    }
  }

  // swap the pivot element with the greater element at i
  swap(&array[i + 1], &array[high]);

  // return the partition point
  return (i + 1);
}
```

This part of the code is to select the pivot and start positioning the array.

```
void quickSort(int array[], int low, int high) {
  if (low < high) {

    // find the pivot element such that
    // elements smaller than pivot are on left of pivot
    // elements greater than pivot are on right of pivot
    int pi = partition(array, low, high);

    // recursive call on the left of pivot
    quickSort(array, low, pi - 1);

    // recursive call on the right of pivot
    quickSort(array, pi + 1, high);
  }
}
```

This part of the code is to recursively call the subparts

Time and space complexity analysis

- Worst Case Complexity [Big-O]: $O(n^2)$

  It occurs when the pivot element picked is either the greatest or the smallest element.

  This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains $n - 1$ elements. Thus, quicksort is called only on this sub-array.

  However, the quicksort algorithm has better performance for scattered pivots.

- Best Case Complexity [Big-omega]: $O(n*\log n)$

  It occurs when the pivot element is always the middle element or near to the middle element.

- Average Case Complexity [Big-theta]: $O(n*\log n)$

  It occurs when the above conditions do not occur.

Strassen’s matrix multiplication

- Multiplication of two n*n matrices

Naive approach

It is a simple method we solve in mathematics for matrix multiplication

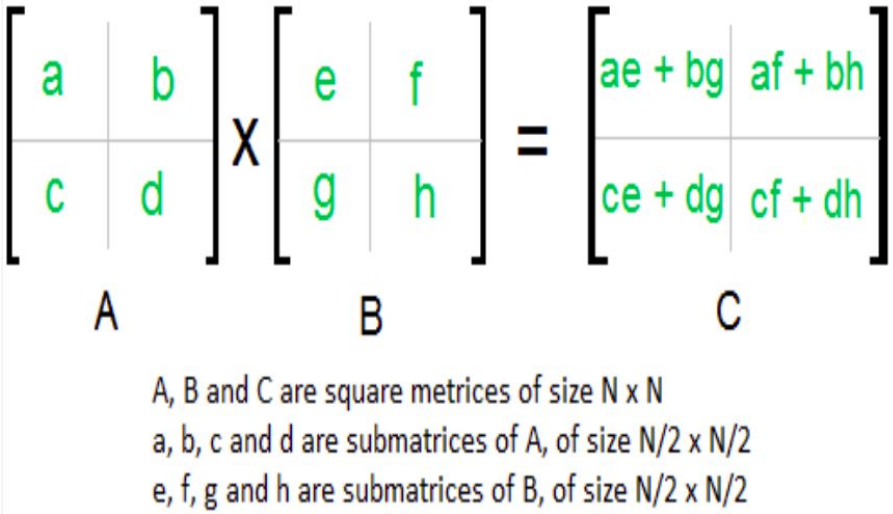
```
void multiply( int A[ ][N], int B[ ][N], int C[ ][N] )
{
        for (int i = 0; i < N; i++)
        {
                for (int j = 0; j < N; j++)
                {
                        C[i][j] = 0;
                        for (int k = 0; k < N; k++)
                        {
                                C[i][j] += A[i][k] * B[k][j];
                        }
                }
        }
}
```

Time complexity: O(n^3)

Following simple divide and conquer approach

Following is simple Divide and Conquer method to multiply two square matrices.

- Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
- Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.



In the above method we need to perform 8 multiplications and 4 additions.

Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as

$$\mathbf{T(N) = 8T(N/2) + O(N^2)}$$

This is again $O(N^3)$, which is same as the naive approach

----- From master's theorem

Better approach ?

In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of **Strassen's method** is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

$$p1 = a(f - h) \qquad p2 = (a + b)h$$
$$p3 = (c + d)e \qquad p4 = d(g - e)$$
$$p5 = (a + d)(e + h) \qquad p6 = (b - d)(g + h)$$
$$p7 = (a - c)(e + f)$$

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Time complexity

Addition and Subtraction of two matrices takes $O(N^2)$ time.

Time complexity can be written as

$$\mathbf{T(N) = 7T(N/2) + O(N^2) = O(N^{\log 7}) = O(N^{2.8074})}$$

Generally Strassen's Method is not preferred for practical applications for following reasons.

- The constants used in Strassen's method are high and for a typical application Naive method works better.
- For Sparse matrices, there are better methods especially designed for them.
- The submatrices in recursion take extra space.
- Because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassen's algorithm than in Naive Method

Dynamic programming algorithms

Those who cannot remember the past are condemned to repeat it.

- Dynamic programming

The image above says a lot about Dynamic Programming. So, is repeating the things for which you already have the answer, a good thing ? A programmer would disagree. That's what Dynamic Programming is about. To *always* remember answers to the sub-problems you've already solved.

The core idea of Dynamic Programming is to avoid repeated work by remembering partial results and this concept finds it application in a lot of real life situations.

In programming, Dynamic Programming is a powerful technique that allows one to solve different types of problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time.

Here is an amazing quora answer about what is dynamic programming?

> Writes down "1+1+1+1+1+1+1+1 =" on a sheet of paper.
> "What's that equal to?"
> Counting "Eight!"
> Writes down another "1+" on the left.
> "What about that?"
> "Nine!" " How'd you know it was nine so fast?"
> "You just added one more!"
> "So you didn't need to recount because you remembered there were eight! Dynamic Programming is just a fancy way to say remembering stuff to save time later!"

Dynamic Programming and Recursion:

Dynamic programming is basically, recursion plus using common sense. What it means is that recursion allows you to express the value of a function in terms of other values of that function. Where the common sense tells you that if you implement your function in a way that the recursive calls are done in advance, and stored for easy access, it will make your program faster. This is what we call Memoization - it is memorizing the results of some specific states, which can then be later accessed to solve other sub-problems.

The intuition behind dynamic programming is that we trade space for time, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

One of the best and simple example of this is fibonacci series which we have already discussed before.

Majority of the Dynamic Programming problems can be categorized into two types:

1. Optimization problems.

2. Combinatorial problems.

The optimization problems expect you to select a feasible solution, so that the value of the required function is minimized or maximized. Combinatorial problems expect you to figure out the number of ways to do something, or the probability of some event happening.

Every Dynamic Programming problem has a schema to be followed:

- Show that the problem can be broken down into optimal sub-problems.
- Recursively define the value of the solution by expressing it in terms of optimal solutions for smaller sub-problems.
- Compute the value of the optimal solution in bottom-up fashion.
- Construct an optimal solution from the computed information.

Some famous Dynamic Programming algorithms are:

- Longest common subsequence
- Bellman-Ford for shortest path routing in networks
- Real world examples

Longest common subsequence

LCS Problem Statement:

Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”.

In order to find out the complexity of brute force approach, we need to first know the number of possible different subsequences of a string with length n, i.e., find the number of subsequences with lengths ranging from 1,2,..n-1.

Number of combinations with 1 element are ${}^nC_1$. Number of combinations with 2 elements are ${}^nC_2$ and so forth and so on. We know that ${}^nC_0 + {}^nC_1 + {}^nC_2 + \ldots {}^nC_n = 2^n$. So a string of length n has 2^n-1 different possible subsequences since we do not consider the subsequence with length 0.

This implies that the time complexity of the brute force approach will be $O(n * 2^n)$. Note that it takes O(n) time to check if a subsequence is common to both the strings. This time complexity can be improved using dynamic programming.

It is a classic computer science problem, the basis of diff (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem.

Optimal substructure:

Let the input sequences be X[0..m-1] and Y[0..n-1] of lengths m and n respectively.

And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y.

Following is the recursive definition of L(X[0..m-1], Y[0..n-1]).

If last characters of both sequences match (or X[m-1] == Y[n-1]) then

L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

If last characters of both sequences do not match (or X[m-1] != Y[n-1]) then
L(X[0..m-1], Y[0..n-1]) = MAX (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))

1) Consider the input strings “AGGTAB” and “GXTXAYB”. Last characters match for the strings. So length of LCS can be written as:

L(“AGGTAB”, “GXTXAYB”) = 1 + L(“AGGTA”, “GXTXAY”)

	A	G	G	T	A	B
G			4			
X						
T				3		
X						
A					2	
Y						
B						1

2) Consider the input strings “ABCDGH” and “AEDFHR. Last characters do not match for the strings. So length of LCS can be written as:

L(“ABCDGH”, “AEDFHR”) = MAX (L(“ABCDG”, “AEDFH**R**”), L(“ABCDG**H**”, “AEDFH”))

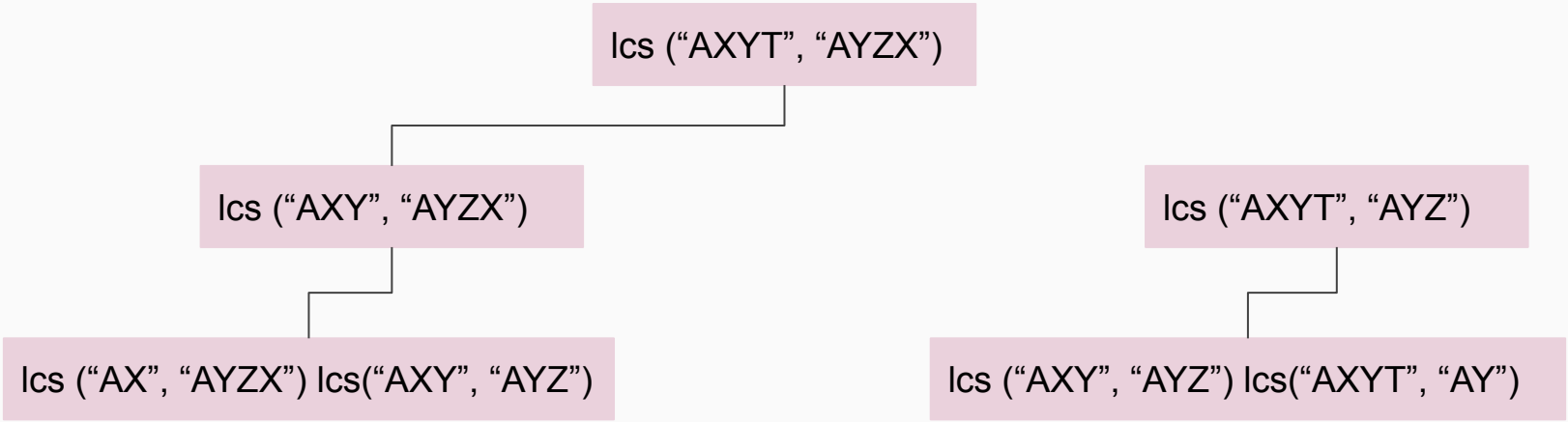
So the LCS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.

```
#include <bits/stdc++.h>
int max(int a, int b)
{
	return (a > b)? a : b;
}
int lcs( char *X, char *Y, int m, int n )
{
	if (m == 0 || n == 0)
		return 0;
	if (X[m-1] == Y[n-1])
		return 1 + lcs(X, Y, m-1, n-1);
	else
		return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}
int main()
{
	char X[] = "AGGTAB";
	char Y[] = "GXTXAYB";

	int m = strlen(X);
	int n = strlen(Y);

	printf("Length of LCS is %d", lcs( X, Y, m, n ) );

	return 0;
}
```

2) Overlapping Subproblems:
Following is simple recursive implementation of the LCS problem. The implementation simply follows the recursive structure mentioned above.

Time complexity of the above naive recursive approach is O(2^n) in worst case and worst case happens when all characters of X and Y mismatch i.e., length of LCS is 0.

Considering the above implementation, following is a partial recursion tree for input strings "AXYT" and "AYZX"



In the above partial recursion tree, lcs("AXY", "AYZ") is being solved twice. If we draw the complete recursion tree, then we can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation.

```
#include<bits/stdc++.h>
int max(int a, int b);

int lcs( char *X, char *Y, int m, int n )
{
	int L[m+1][n+1];
	int i, j;

	for (i=0; i<=m; i++)
	{
		for (j=0; j<=n; j++)
		{
			If (i == 0 || j == 0)
				L[i][j] = 0;
			else if (X[i-1] == Y[j-1])
				L[i][j] = L[i-1][j-1] + 1;
			else
				L[i][j] = max(L[i-1][j], L[i][j-1]);
		}
	}
	return L[m][n];
}
int max(int a, int b)
{
	return (a > b)? a : b;
}
int main()
{
	char X[] = "AGGTAB";
	char Y[] = "GXTXAYB";

	int m = strlen(X);
	int n = strlen(Y);

	printf("Length of LCS is %d", lcs( X, Y, m, n ) );

	return 0;
}
```

Time Complexity of the above implementation is O(mn) which is much better than the worst-case time complexity of Naive Recursive implementation.

Bellman-Ford for shortest path routing in networks

Problem statement:

Given a graph and a source vertex ***src*** in graph, find shortest paths from ***src*** to all vertices in the given graph. The graph may contain negative weight edges.

Algorithm

Input: Graph and a source vertex *src*
Output: Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1) This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.
a) Do following for each edge u-v
If dist[v] > dist[u] + weight of edge uv, then update dist[v]
dist[v] = dist[u] + weight of edge uv
3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
……If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn’t contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle.

How does this work?

- Like other Dynamic Programming Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path.

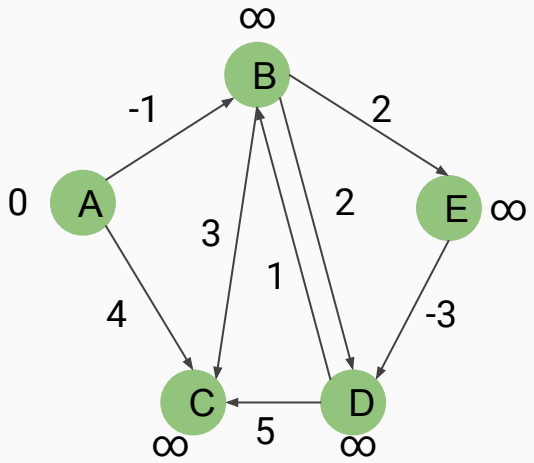
- Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most i edges are calculated.

- There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times.

- The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges

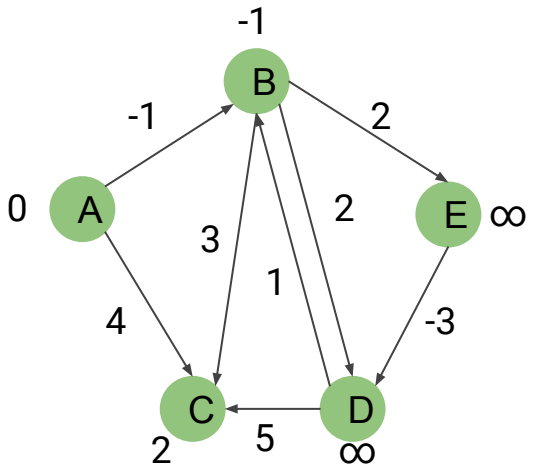
Let us understand the algorithm with following example graph.

Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times.*



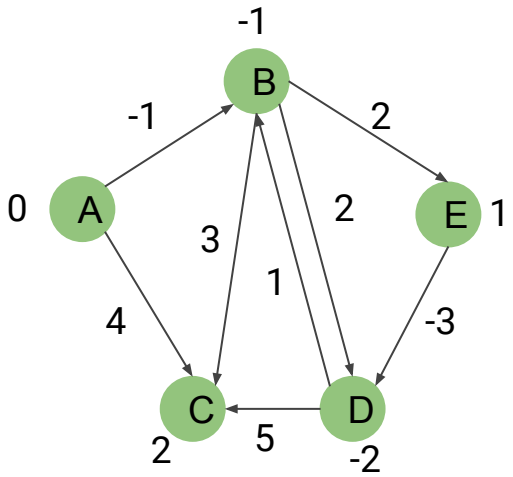
Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed.

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed second time (The last row shows final values).

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don’t update the distances.

```c
void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Step 1: Initialize distances from src to all other vertices
    // as INFINITE
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;

    // Step 2: Relax all edges |V| - 1 times. A simple shortest
    // path from src to any other vertex can have at-most |V| - 1
    // edges
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    // Step 3: check for negative-weight cycles. The above step
    // guarantees shortest distances if graph doesn't contain
    // negative weight cycle. If we get a shorter path, then there
    // is a cycle.
    for (int i = 0; i < E; i++) {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            printf("Graph contains negative weight cycle");
            return; // If negative cycle is detected, simply return
        }
    }

    printArr(dist, V);

    return;
}
```

The links edges and their respective weights are created in a struct named graph

As we can see in the above code, the time complexity of the Bellman-Ford algorithm is O(|V||E|), where V = total number of vertices in the graph and E = Total number of edges.

Real world examples of dynamic programming

In spacecraft communications, the radio transmissions are often sent using error correcting codes. When the spacecraft is out at the edge of the solar system, it would just take too long to ask for a repeat of a mangled packet, and anyway, the spacecraft probably doesn't have enough memory to buffer everything.

The codes used are often "convolutional codes" which use linear feedback shift registers to make transmitted bits be a function of a number of data bits.

The decoders for such things often use the "Viterbi Algorithm" which is a variant of dynamic programming.

In gene sequencing, the sequencing machines produce multiple redundant “reads” of random sections of the genome. the error rates can be quite high. The alignment software has to compare string which may have insertions, deletions, and plain old errors. The algorithms that do this are variations of dynamic programming.

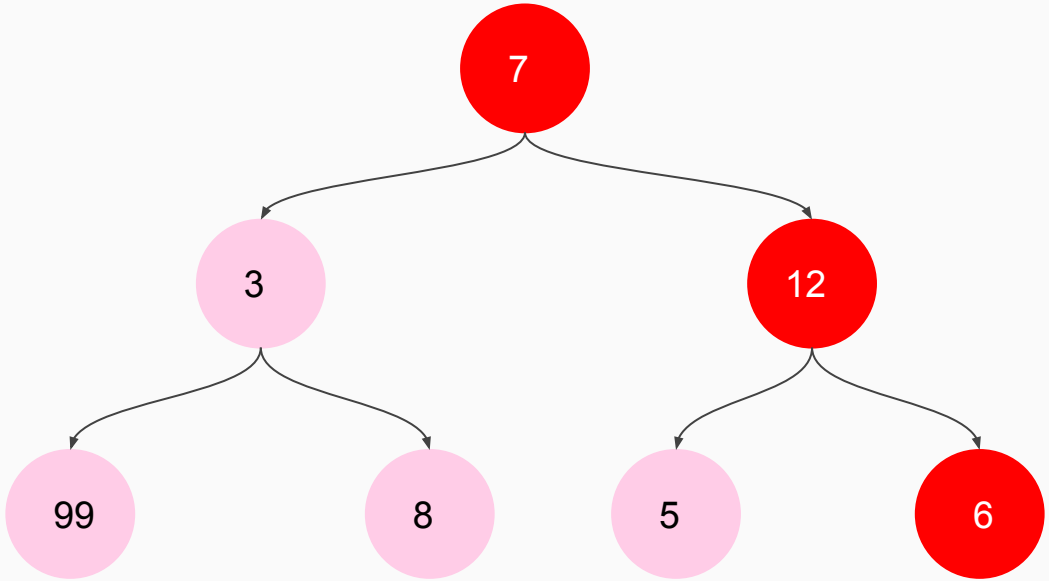
In speech recognition, the recognizer has to match possible phoneme sequences restricted by the speech grammar against speech waveform and spectral data. This used to be done by dynamic programming, but things might be different now with machine learning.

Greedy algorithms

A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem. Greedy algorithms are quite successful in some problems, such as Huffman encoding which is used to compress data, or Dijkstra's algorithm, which is used to find the shortest path through a graph.

However, in many problems, a greedy strategy does not produce an optimal solution. For example, in the graph below, the greedy algorithm seeks to find the path with the largest sum. It does this by selecting the largest available number at each step. The greedy algorithm fails to find the largest sum, however, because it makes decisions based only on the information it has at any one step, without regard to the overall problem.

Below is the path followed by greedy algorithm to get the largest sum path.

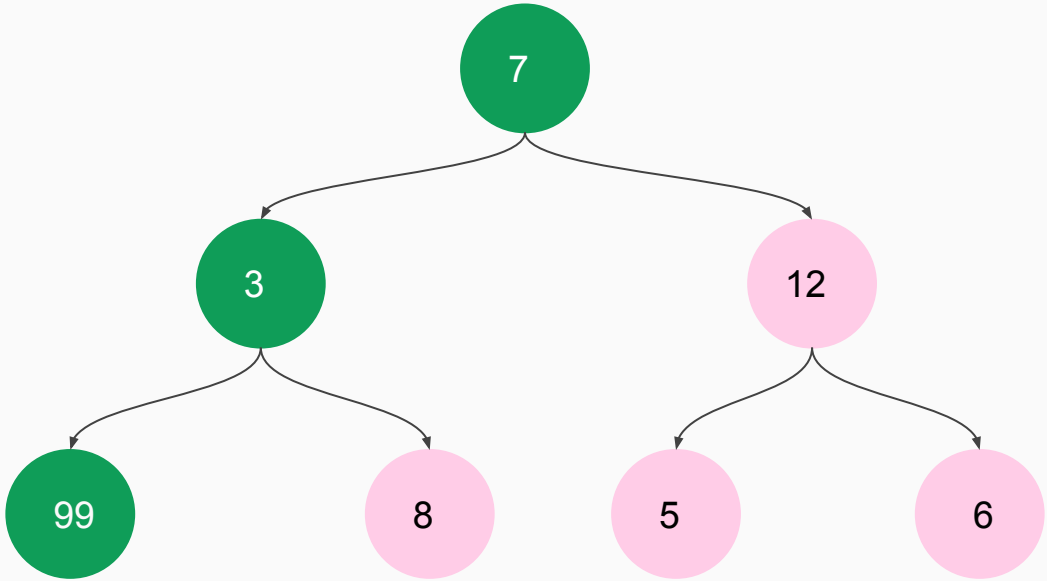


Sum of the path followed: 7 + 12 + 6 = 25

But is that the largest sum possible in the graph ?

Ans: NO !

Below is the actual largest sum path.



Sum of the path followed: 7 + 3 + 99 = 109

With a goal of reaching the largest sum, at each step, the greedy algorithm will choose what appears to be the optimal immediate choice, so it will choose 12 instead of 3 at the second step and will not reach the best solution, which contains 99.

Structure of greedy algorithm

Greedy algorithms take all of the data in a particular problem, and then set a rule for which elements to add to the solution at each step of the algorithm. In the animation above, the set of data is all of the numbers in the graph, and the rule was to select the largest number available at each level of the graph. The solution that the algorithm builds is the sum of all of those choices.

If both of the properties below are true, a greedy algorithm can be used to solve the problem.

- **Greedy choice property:** A global (overall) optimal solution can be reached by choosing the optimal choice at each step.
- **Optimal substructure:** A problem has an optimal substructure if an optimal solution to the entire problem contains the optimal solutions to the sub-problems.

In other words, greedy algorithms work on problems for which it is true that, at every step, there is a choice that is optimal for the problem up to that step, and after the last step, the algorithm produces the optimal solution of the complete problem.

To make a greedy algorithm, identify an optimal substructure or subproblem in the problem. Then, determine what the solution will include (for example, the largest sum, the shortest path, etc.). Create some sort of iterative way to go through all of the subproblems and build a solution.

Sometimes greedy algorithms fail to find the globally optimal solution because they do not consider all the data. The choice made by a greedy algorithm may depend on choices it has made so far, but it is not aware of future choices it could make.

Here, we will look at one form of the knapsack problem. The knapsack problem involves deciding which subset of items you should take from a set of items if you want to optimize some value: perhaps the worth of the items, the size of the items, or the ratio of worth to size.

In this problem, we will assume that we can either take an item or leave it (we cannot take a fractional part of an item). We will also assume that there is only one of each item. Our knapsack has a fixed size, and we want to optimize the worth of the items we take, so we must choose the items we take with care.

Our knapsack can hold at most 25 units of space. Here is the list of items and their worths.

Item	**Size**	**Price**
Laptop	22	12
Playstation	10	9
Text book	9	9
Basketball	7	6

Which items do we choose to optimize for price?

There are two greedy algorithms we could propose to solve this. One has a rule that selects the item with the largest price at each step, and the other has a rule that selects the smallest sized item at each step.

- Largest-price Algorithm: At the first step, we take the laptop. We gain 12 units of worth, but can now only carry $25-22=3$ units of additional space in the knapsack. Since no items that remain will fit into the bag, we can only take the laptop and have a total of 12 units of worth.
- Smallest-sized-item Algorithm: At the first step, we will take the smallest-sized item: the basketball. This gives us 6 units of worth, and leaves us with $25-7=18$ units of space in our bag. Next, we select the next smallest item, the textbook. This gives us a total of $6+9=15$ units of worth, and leaves us with $18-9=9$ units of space. Since no remaining items are 9 units of space or less, we can take no more items.

The greedy algorithms yield solutions that give us 12 units of worth and 15 units of worth. But neither of these are the optimal solution. Inspect the table yourself and see if you can determine a better selection of items. Taking the textbook and the PlayStation yields $9+9=18$ units of worth and takes up $10+9=19$ units of space. This is the optimal answer, and we can see that a greedy algorithm will not solve the knapsack problem since the greedy choice and optimal substructure properties do not hold.

In problems where greedy algorithms fail, dynamic programming might be a better approach.

Applications

Dijkstra’s algorithm

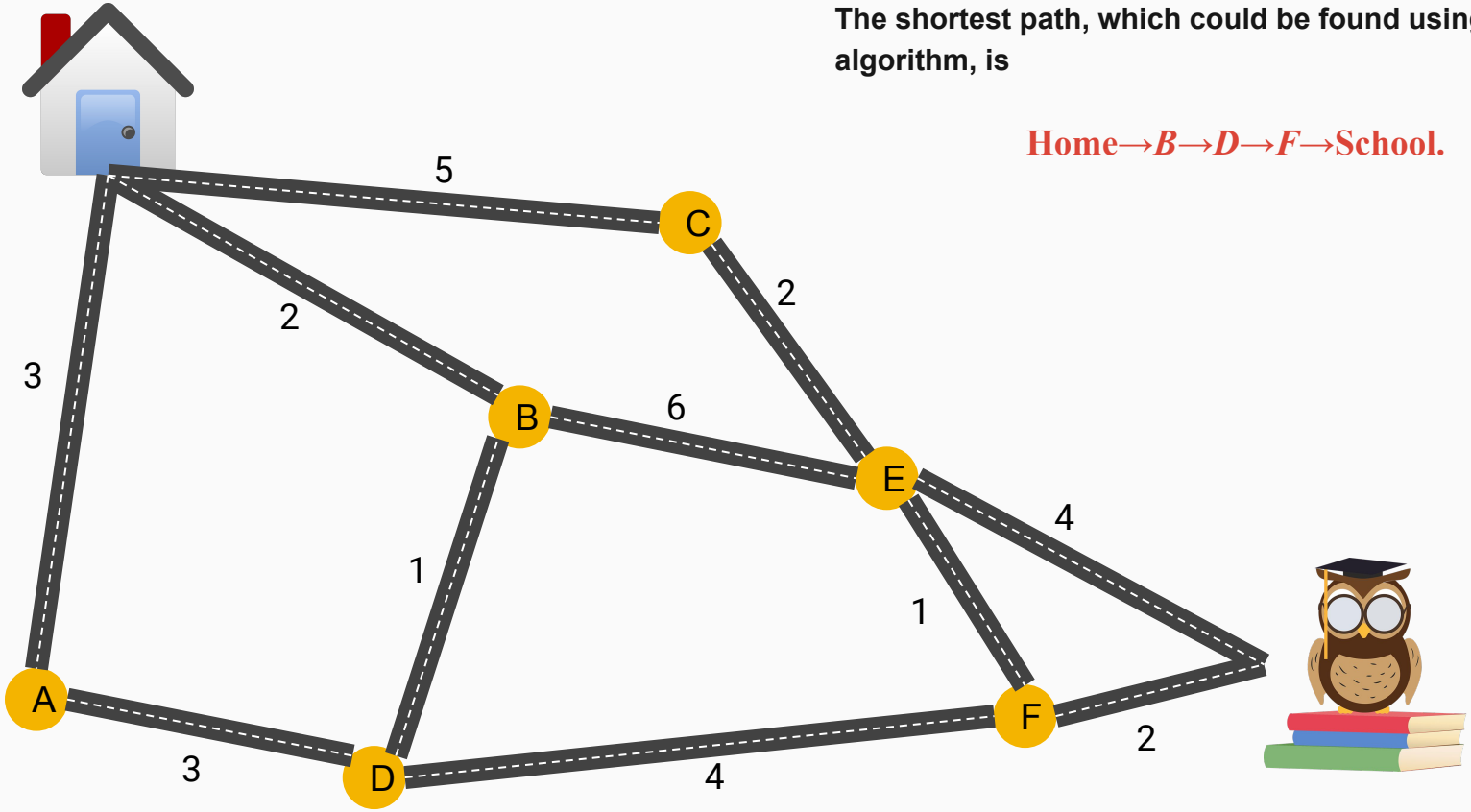
One algorithm for finding the shortest path from a starting node to a target node in a weighted graph is Dijkstra’s algorithm. The algorithm creates a tree of shortest paths from the starting vertex, the source, to all other points in the graph.

Dijkstra’s algorithm, published in 1959 and named after its creator Dutch computer scientist Edsger Dijkstra, can be applied on a weighted graph. The graph can either be directed or undirected. One stipulation to using the algorithm is that the graph needs to have a nonnegative weight on every edge.

Suppose a student wants to go from home to school in the shortest possible way. She knows some roads are heavily congested and difficult to use. In Dijkstra's algorithm, this means the edge has a large weight--the shortest path tree found by the algorithm will try to avoid edges with larger weights. If the student looks up directions using a map service, it is likely they may use Dijkstra's algorithm, as well as others.

The shortest path, which could be found using Dijkstra's algorithm, is

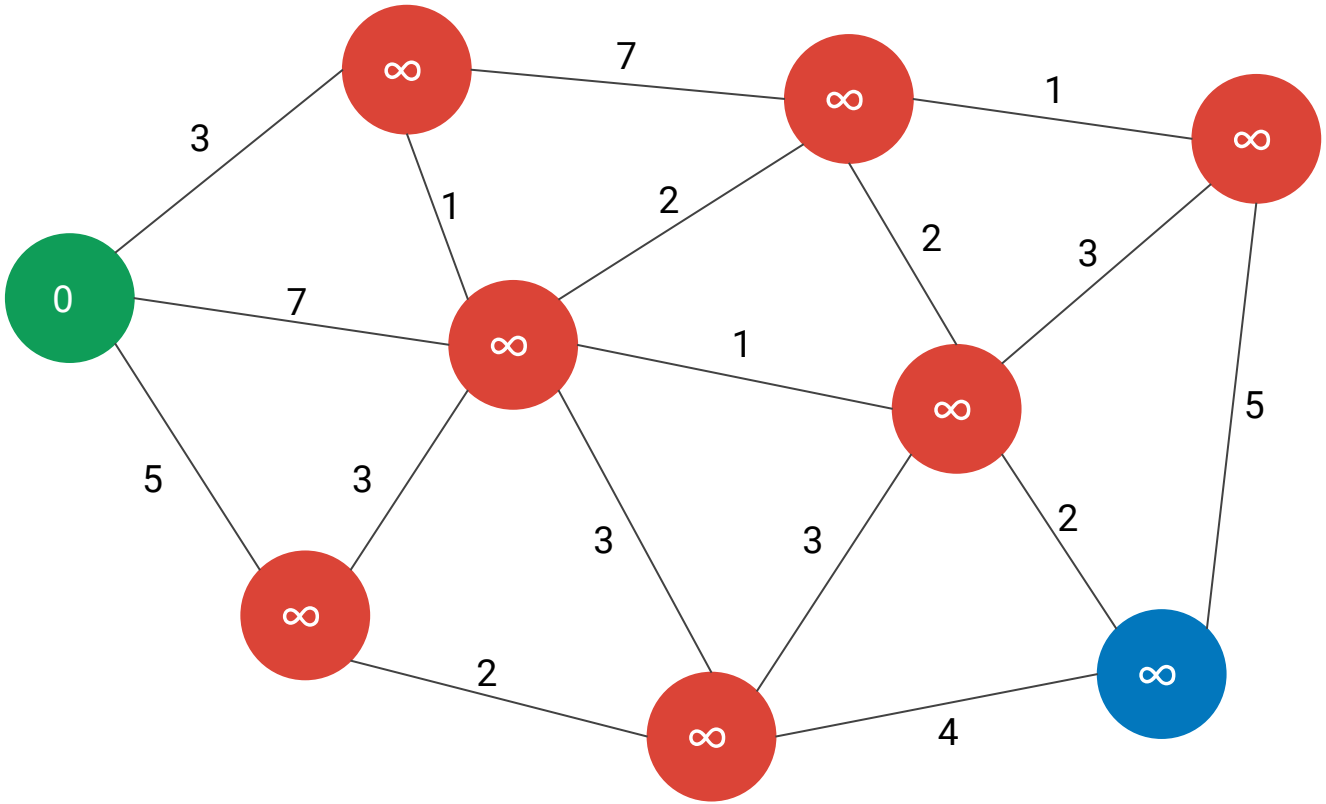
Home→*B*→*D*→*F*→School.



Dijkstra's algorithm finds a shortest path tree from a single source node, by building a set of nodes that have minimum distance from the source.

The graph has the following:

- vertices, or nodes, denoted in the algorithm by v or u;
- weighted edges that connect two nodes: (u,v) denotes an edge, and $w(u,v)$ denotes its weight. In the diagram on the right, the weight for each edge is written in gray.This is done by initializing three values:
- $dist$, an array of distances from the source node
- s to each node in the graph, initialized the following way:
- $dist(s) = 0$; and for all other nodes v, $dist(v) = \infty$. This is done at the beginning because as the algorithm proceeds, the $dist$ from the source to each node v in the graph will be recalculated and finalized when the shortest distance to v is found.
- Q, a queue of all nodes in the graph. At the end of the algorithm's progress, Q will be empty.
- S, an empty set, to indicate which nodes the algorithm has visited. At the end of the algorithm's run, S will contain all the nodes of the graph.


0
∞
∞
∞
∞
∞
∞
∞
3
7
1
2
7
1
2
3
5
3
3
3
2
5
2
4

The algorithm proceeds as follows:

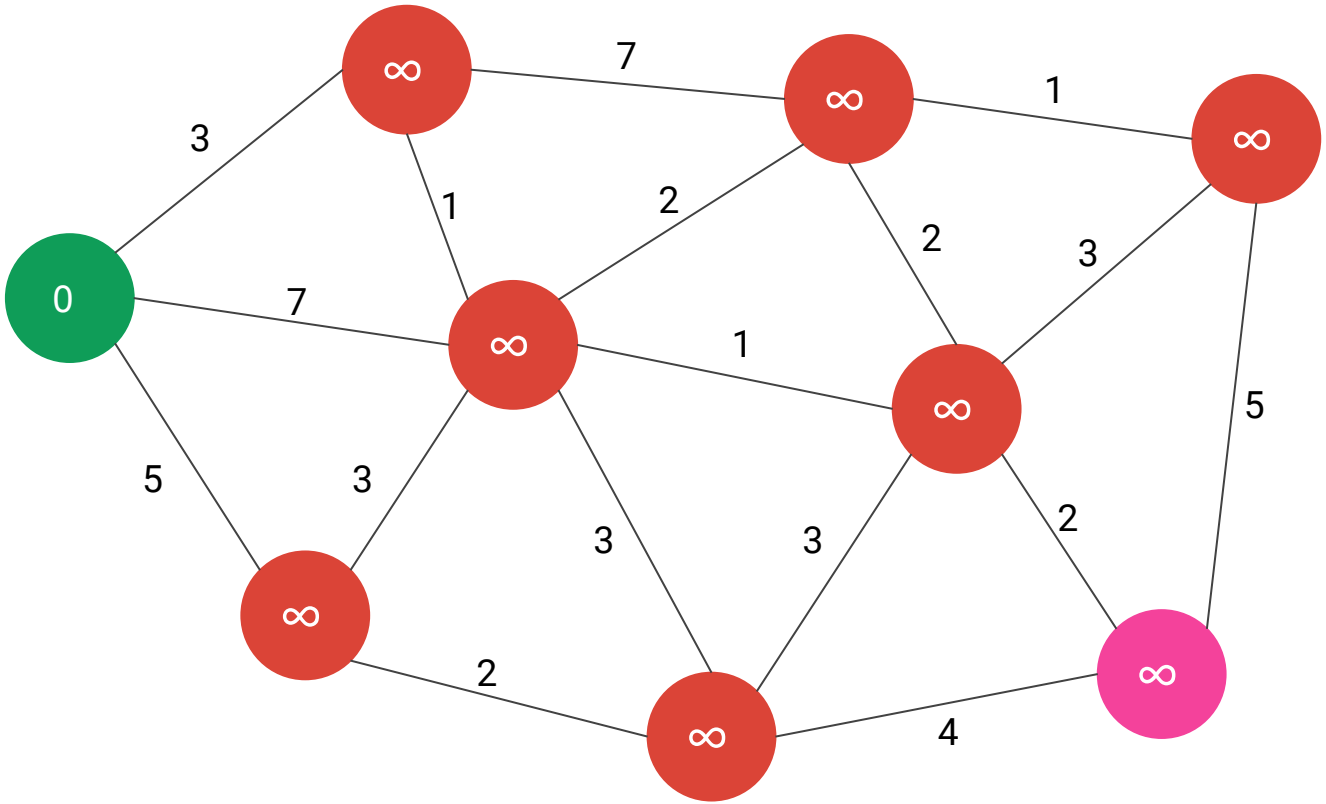
1. While Q is not empty, pop the node v, that is not already in S, from Q with the smallest $dist\ (v)$. In the first run, source node s will be chosen because $dist(s)$ was initialized to 0. In the next run, the next node with the smallest $dist$ value is chosen.
2. Add node v to S, to indicate that v has been visited
3. Update $dist$ values of adjacent nodes of the current node v as follows: for each new adjacent node u,

- if $dist\ (v) + weight(u,v) < dist\ (u)$, there is a new minimal distance found for u, so update $dist\ (u)$ to the new minimal distance value;
- otherwise, no updates are made to $dist\ (u)$.

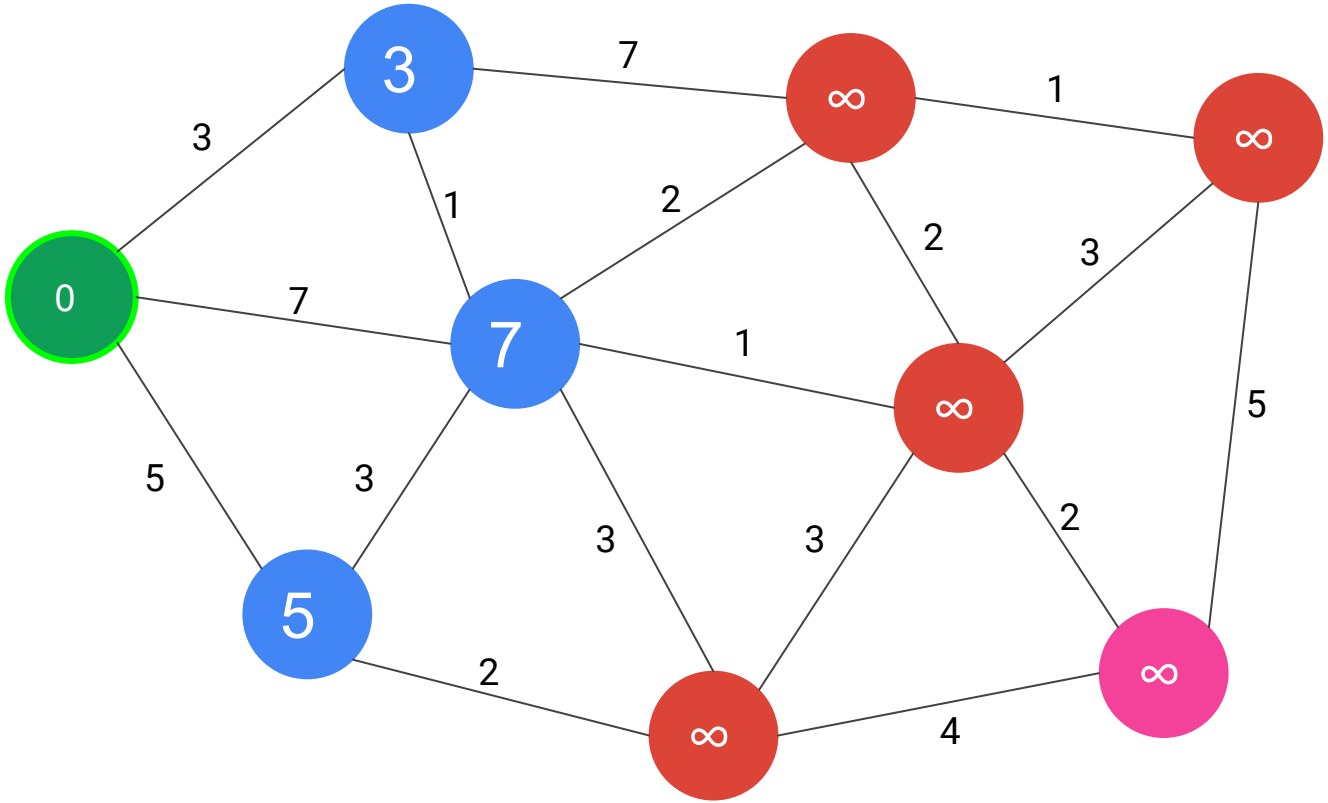
The algorithm has visited all nodes in the graph and found the smallest distance to each node. $dist$ now contains the shortest path tree from source s.

Note: The weight of an edge (u,v) is taken from the value associated with (u,v) on the graph.

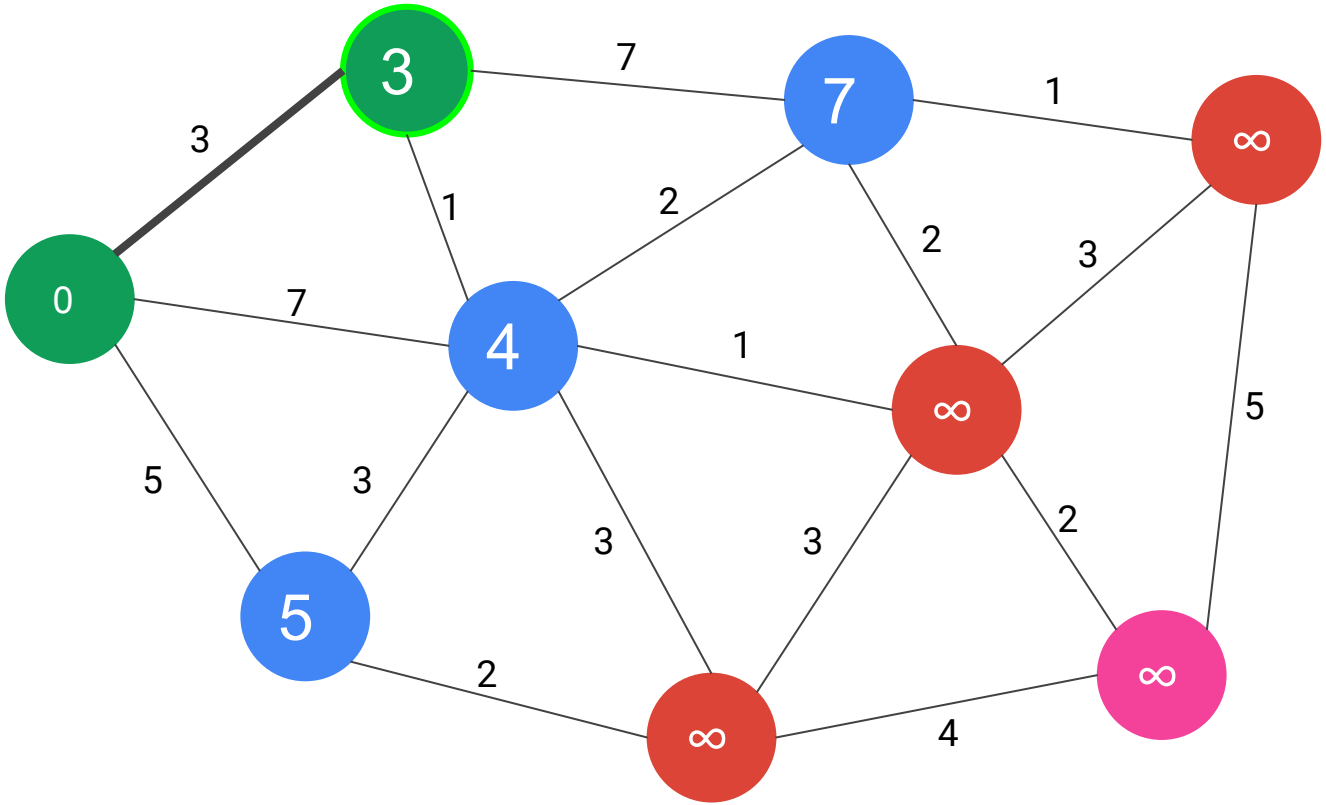
Initialize distance according to the algorithm



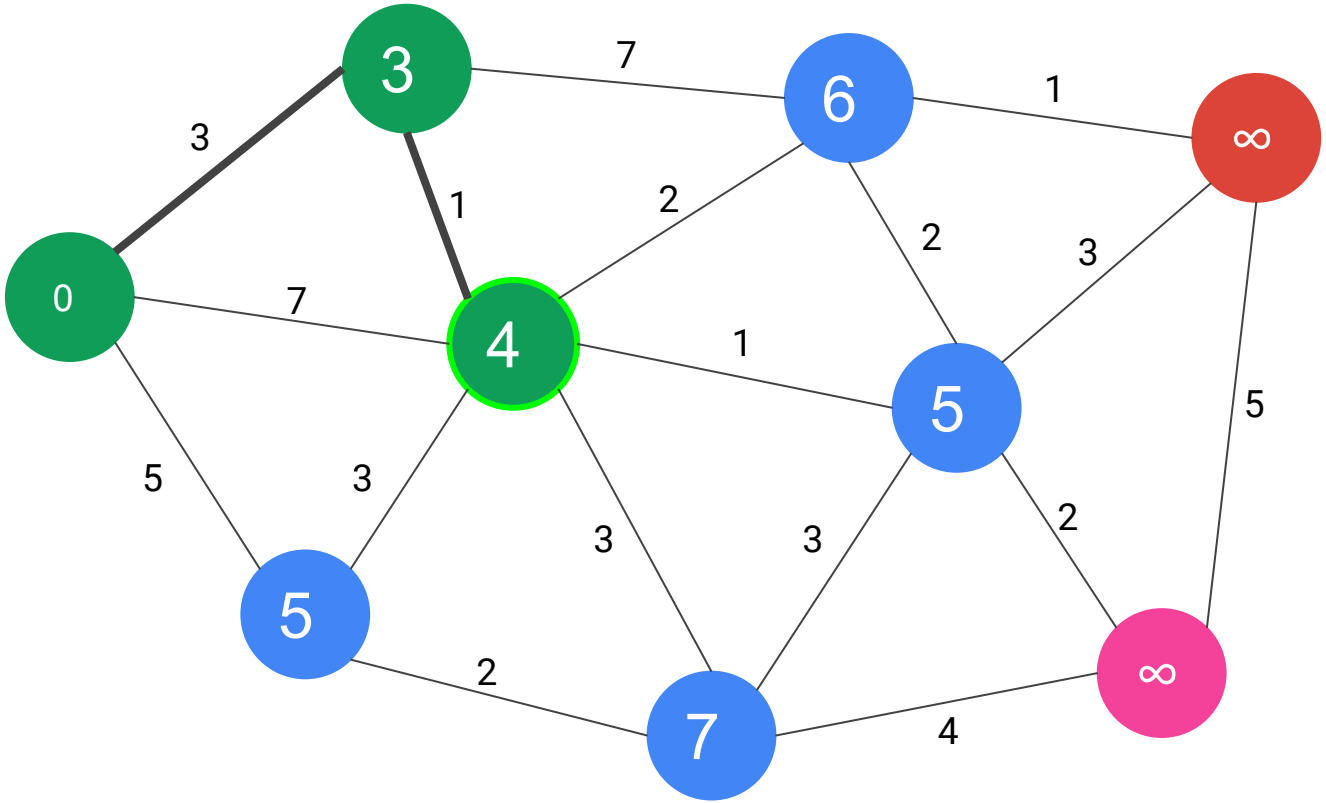
Pick first node and find distance to adjacent nodes



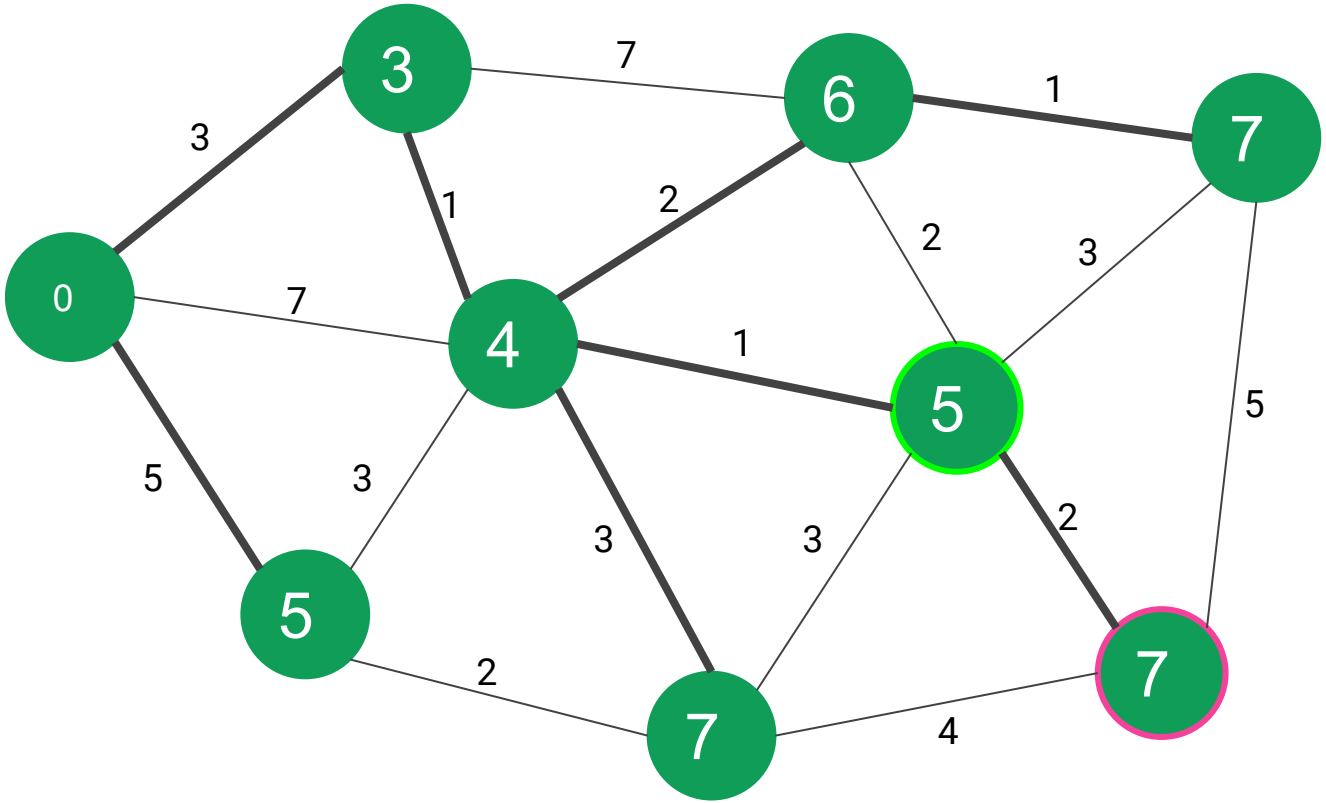
Pick next node with minimal distance and repeat the adjacent node distance calculation.



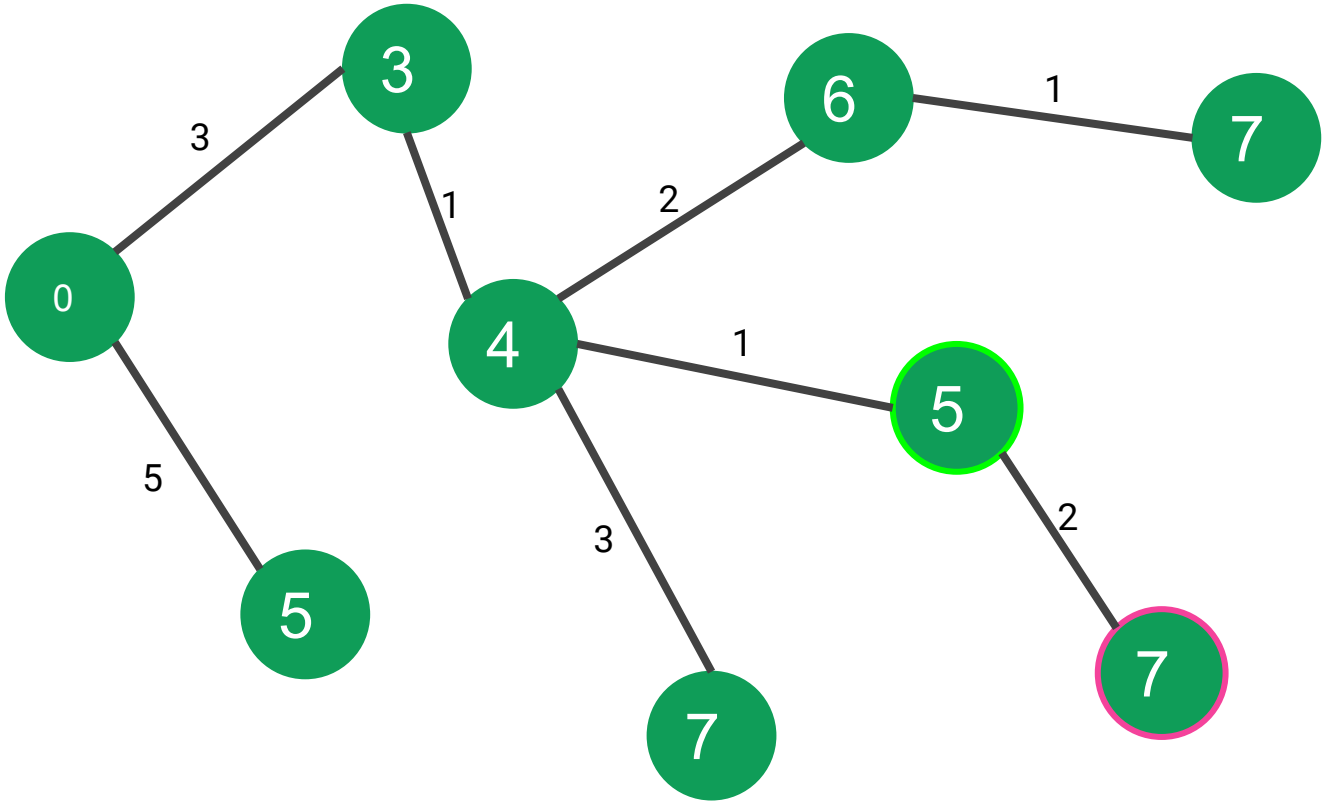
Pick next node with minimal distance and repeat the adjacent node distance calculation.



Pick next node with minimal distance and repeat the adjacent node distance calculation.



So, here is the shortest path tree.



Let us have a look at the code

```c
void dijkstra(int graph[V][V], int src)
{
    int dist[V]; // The output array. dist[i] will hold the shortest
                 // distance from src to i
    bool sptSet[V]; // sptSet[i] will be true if vertex i is included in shortest
                    // path tree or shortest distance from src to i is finalized
    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;
    // Distance of source vertex from itself is always 0
    dist[src] = 0;
    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum distance vertex from the set of vertices not
        // yet processed. u is always equal to src in the first iteration.
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = true;

        // Update dist value of the adjacent vertices of the picked vertex.
        for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet, there is an edge from
            // u to v, and total weight of path from src to v through u is
            // smaller than current value of dist[v]
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
    // print the constructed distance array
    printSolution(dist);
}
```

This function is used to pick the minimum distance adjacent vertex which is implemented in next slide.

```
int minDistance(int dist[], bool sptSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}
```

Time Complexity of the implementation is $O(V^2)$. If the input graph is represented using adjacency list, it can be reduced to O(E log V) with the help of a binary heap.

Huffman encoding algorithm

- **Applications of greedy algorithm**

Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by **David Huffman**.

Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

How does huffman coding work ?

Suppose the string below is sent over a network:

Each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of `8 * 15 = 120` bits are required to send this string.

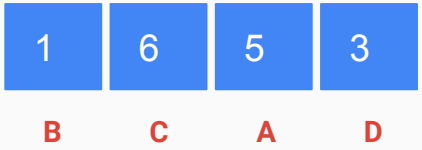
Using the Huffman Coding technique, we can compress the string to a smaller size.

Huffman coding first creates a tree using the frequencies of the character and then generates code for each character. Once the data is encoded, it has to be decoded. Decoding is done using the same tree.

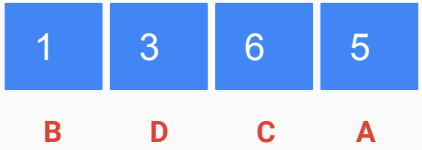
Huffman Coding prevents any ambiguity in the decoding process using the concept of prefix code ie. a code associated with a character should not be present in the prefix of any other code. The tree created above helps in maintaining the property.

Huffman coding is done using the following steps:

1. Calculate the frequency of each character in the string.

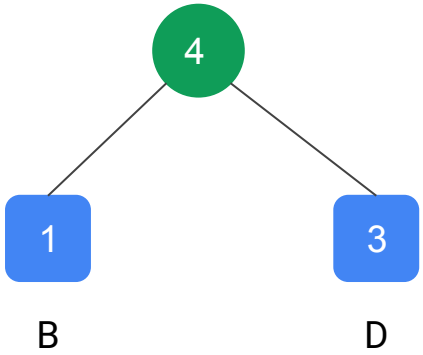


2. Sort the characters in the increasing order of their frequency. These are stored in a priority queue.



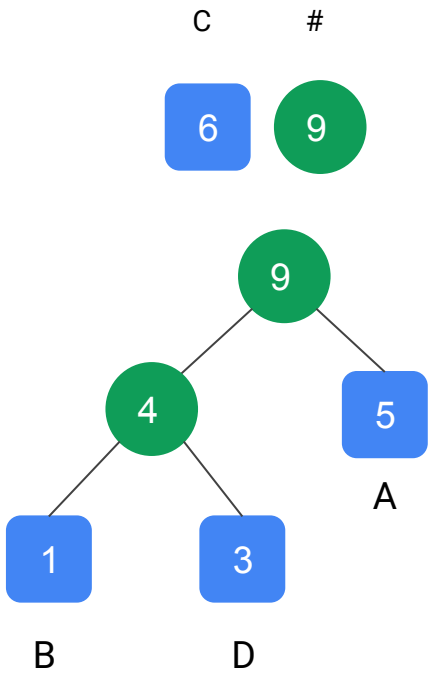
3. Make each unique character as a leaf node.

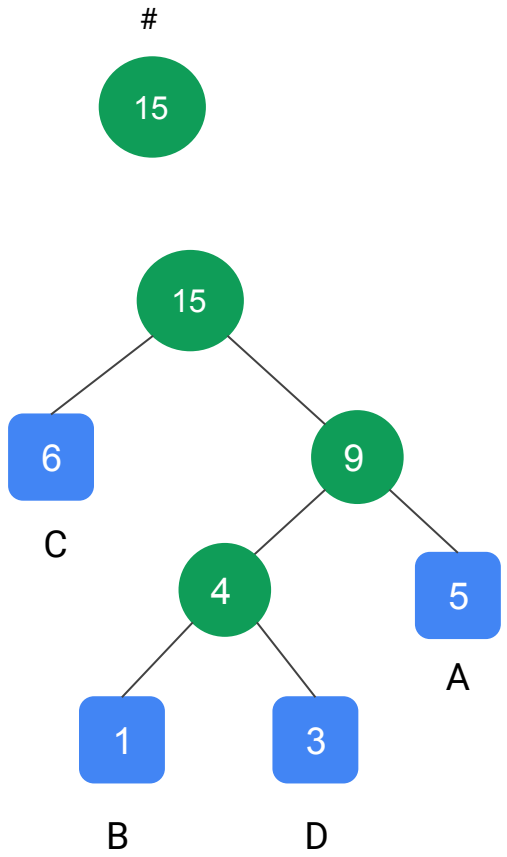
4. Create a empty node “z”. Assign the minimum frequency to the left child of “z” and the value of the second minimum frequency to the right child of “z”. Set the value of “z” as the sum of the above two minimum frequencies.



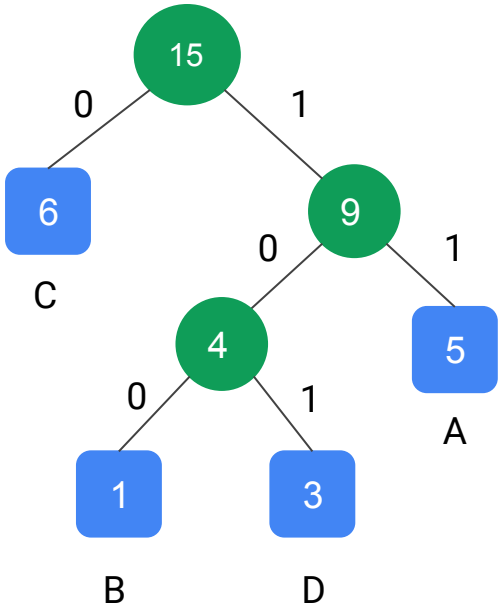
5. Remove these two minimum frequency nodes from the queue and add the sum into the list of frequencies (# denotes the internal nodes in the figure above.).

6. Insert node “z” into the tree.
7. Repeat steps 3 to 5 for all the characters.


C
#
6
9
9
4
5
A
1
3
B
D


#
15
15
6
9
C
4
5
A
1
3
B
D

For each non-leaf node, assign 0 to the left edge and 1 to the right edge.



For sending the above string over a network, we have to send the tree as well as the compressed code. The total size is given in the below table.

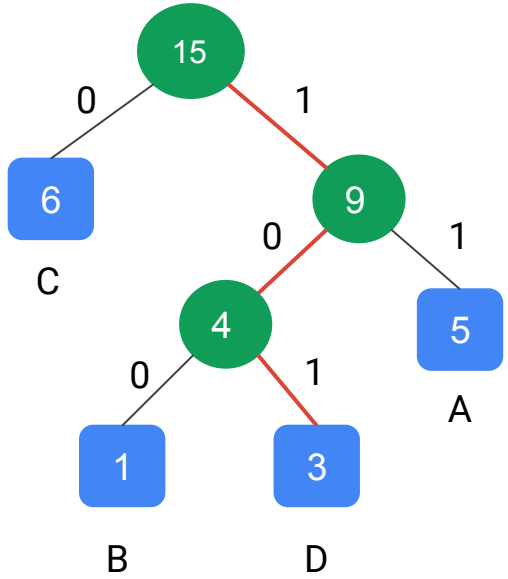
Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4*8 = 32bits	15 bits		28 bits

Without encoding the total size of the string was 120 bits. After encoding the size reduced to 32 + 15 + 28 = 75

Decoding the code

For decoding the code, we can take the code and traverse through the tree to find the character.

Let **101** is to be decoded, we can traverse from the root as in the figure below.



Check out the C code for Huffman-coding here

Time complexity:

The time complexity for encoding each unique character based on its frequency is `O(nlog n)`.

Extracting minimum frequency from the priority queue takes place `2*(n-1)` times and its complexity is `O(log n)`. Thus the overall complexity is `O(nlog n)`.

Applications:

- Huffman coding is used in conventional compression formats like GZIP, BZIP2, PKZIP, etc.
- For text and fax transmissions.

Brute force algorithms

“Data is the new oil” this is the new mantra that is ruling the global economy. We live in the digital world, and every business revolves around data, which translates into profits and helps the industries stay ahead of their competition. With the rapid digitization, an exponential increase in the app-based business model, cyber-crimes is a constant threat. One such common activity that hackers perform is Brute force.

Brute Force is a trial and error approach where attackers use programs to try out various combinations to break into any websites or systems. They use automated software to repetitively generate the User id and passwords combinations until it eventually generates the right combination.

Brute force search

Brute force search is the most common search algorithm as it does not require any domain knowledge; all that is required is a state description, legal operators, the initial state and the description of a goal state. It does not improve the performance and completely relies on the computing power to try out possible combinations.

The brute force algorithm searches all the positions in the text between 0 and n-m, whether the occurrence of the pattern starts there or not. After each attempt, it shifts the pattern to the right by exactly 1 position. The time complexity of this algorithm is O(m*n). If we are searching for n characters in a string of m characters, then it will take n*m tries.

Let’s see a classic example of a travelling salesman to understand the algorithm in an easy manner.

Suppose a salesman needs to travel 10 different cities in a country, and he wants to determine the shortest possible routes out of all the possible combinations. Here brute force algorithm simply calculates the distance between all the cities and selects the shortest one.

Another example is to make an attempt to break the 5 digit password; then brute force may take up to 10^5 attempts to crack the code.

Each blank space of the password can hold any one of the digits from 0 to 9. Which means it has 10 possibilities.

We have 5 such blanks to fill and each blank have 10 possibilities.

Which means that total number of possibilities of this 5 digit password is $10*10*10*10*10 = 10^5$.

Brute force sort

In the brute force sort technique, the data list is scanned multiple times to find the smallest element in the list.

After each iteration over the list, it replaces the smallest element to the top of the stack and starts the next

iteration from the second smallest data in the list.

The above statement can be written in pseudo-code as follows.

```
Algorithm selectionSort(A[0...n-1])
        For i = 0 to i = n-2 do
                Min = i
                For j = i+1 to n-1 do
                        If A[j] < A[min] then Min = j
                Swap A[i] and A[Min]
```

Here the problem is of size ‘n’, and the basic operation is ‘if’ test where the data items are being compared in each iteration. The will be no difference between the worst and best case as the no of swap is always n-1.

Brute force string matching

If all the characters in the pattern are unique, then Brute force string matching can be applied with the complexity of Big O(n) where n is the string's length. Brute force String matching compares the pattern with the substring of a text character by character until it gets a mismatched character. As soon as a mismatch is found, the substring's remaining character is dropped, and the algorithm moves to the next substring.

The pseudo-codes explain the string matching logic. Here the algorithm is trying to search for a pattern of P[0…m-1] in the text T[0….n-1].

Here the worst case would be when a shift to another substring is not made until M^{th} comparison.

```
Algorithm BruteForceStringSearch(T[0….n-1], P[0….m-1])
        For i=0 to n-m do
                J = 0
                While j < m and P[j] = T[i+j] do
                        J++
                If j == m return m
        Return  -1
```

Closest pair

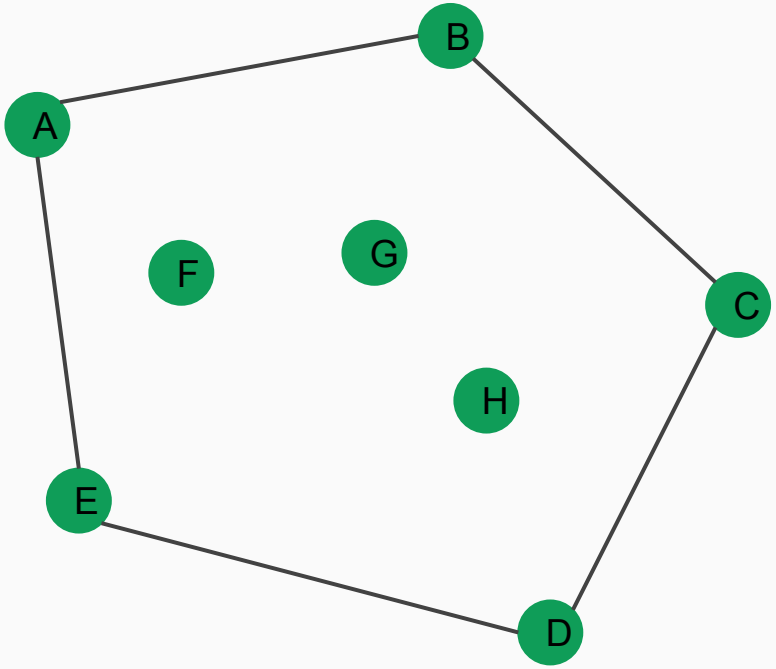
Problem statement: To find out the two closest points in a set of n points in the two-dimensional cartesian plane. There is n number of scenarios where this problem arises. A real-life example would be in an air traffic control system where you have to monitor the planes flying near to each other, and you have to find out the safest minimum distance these planes should maintain.

The brute force algorithm computes the distance between every distinct set of points and returns the point's indexes for which the distance is the smallest. Brute force solves this problem with the time complexity of [O(n2)] where n is the number of points. Below the pseudo-code uses the brute force algorithm to find the closest point.

```
Algorithm BruteForceClosestPoints(P)
    // P is list of points
    dmin ← ∞
    for i ← 1 to n-1 do
        for j ← i+1 to n do
            d ← sqrt((x_i-x_j)^2 + (y_i-y_j)^2)
            if d < dmin then
                dmin ← d; index1 ← i; index2 ← j
    return index1, index2
```

Convex Hull

Problem statement: A convex Hull is the smallest polygon that contains all the points. The convex hull of a set s of the points is the smallest convex polygon containing s.



The convex hull is the set of points A, B, C, D, E.
A line segment “A” and An of a set of n points is a part of the convex hull if and only if all the other points of the set lies inside the polygon boundary formed by the line segment.
Let us relate it with a rubber band.
Point (x1, y1) and (x2, y2) make the line ax+by=c.
When a=y2-y1, b=x2-x1 and c=x1*y2-x2*y1 and divides the plane by ax+by-c<0 and ax+by-c>0.
So we need to check ax+by-c for the other points.

Brute force solve this problem with the time complexity of $O(n^3)$

Exhaustive search

For discrete problems in which there is no known efficient solution, it becomes necessary to test each and every possible solution sequentially. Exhaustive search is an activity to find out all the possible solutions to a problem in a systematic manner. Let's try to solve the Travelling salesman problem (TSP) using a Brute exhaustive search algorithm.

Problem Statement: There are n cities that salesmen need to travel, he wants to find out the shortest route covering all the cities.

We are considering Hamilton Circuit to solve this problem. If a circuit exists, then any point can start vertices and end vertices. Once the start vertices are selected, then we only need the order for the remaining vertices, i.e. n-1. Then there would be (n-1)! Possible combinations and the total cost for calculating the path would be O(n). thus the total time complexity would be O(n!).

Randomized algorithms

A randomized algorithm is a technique that uses a source of randomness as part of its logic. It is typically used to reduce either the running time, or time complexity; or the memory used, or space complexity, in a standard algorithm. The algorithm works by generating a random number, *r*, within a specified range of numbers, and making decisions based on *r*'s value.

A randomized algorithm could help in a situation of doubt by flipping a coin or a drawing a card from a deck in order to make a decision. Similarly, this kind of algorithm could help speed up a brute force process by randomly sampling the input in order to obtain a solution that may not be totally optimal, but will be good enough for the specified purposes.

Example:

A superintendent is attempting to score a high school based on several metrics, and she wants to do so from information gathered by confidentially interviewing students. However, the superintendent has to do this with all the schools in the district, so interviewing every single student would take a time she cannot afford. What should she do?

The superintendent should employ a randomized algorithm, where, without knowing any of the kids, she'd select a few at random and interview them, hoping that she gets a wide variety of students. This technique is more commonly known as random sampling, which is a kind of randomized algorithm. Of course, she knows that there are diminishing returns from each additional interview, and should stop when the quantity of data collected measures what she was trying to measure to an acceptable degree of accuracy. The way that the superintendent is determining the score of the school can be thought of as a randomized algorithm.

Definitions of a Randomized Algorithm

Randomized algorithms are used when presented with a time or memory constraint, and an average case solution is an acceptable output. Due to the potential erroneous output of the algorithm, an algorithm known as amplification is used in order to boost the probability of correctness by sacrificing runtime. Amplification works by repeating the randomized algorithm several times with different random subsamples of the input, and comparing their results. It is common for randomized algorithms to amplify just parts of the process, as too much amplification may increase the running time beyond the given constraints.

Randomized algorithms are usually designed in one of two common forms: as a **Las Vegas** or as a **Monte Carlo algorithm**. A Las Vegas algorithm runs within a specified amount of time. If it finds a solution within that time frame, the solution will be exactly correct; however, it is possible that it runs out of time and does not find any solutions. On the other hand, a Monte Carlo algorithm is a probabilistic algorithm which, depending on the input, has a slight probability of producing an incorrect result or failing to produce a result altogether.

Practically speaking, computers cannot generate completely random numbers, so randomized algorithms in computer science are approximated using a pseudorandom number generator in place of a true source of random number, such as the drawing of a card.

Monte Carlo Algorithms

Intuitive explanation

The game, **Wheel of Fortune**, can be played using a Monte Carlo randomized algorithm. Instead of mindfully choosing letters, a player (or computer) picks randomly letters to obtain a solution, as shown in the image below. The more letters a player reveals, the more confident a player becomes in their solution. However, if a player does not guess quickly, the chance that other players will guess the solution also increases. Therefore, a Monte Carlo algorithm is given a deterministic amount of time, in which it must come up with a "guess" based on the information revealed; the best solution it can come up with. This allows for the possibility of being wrong, maybe even a large probability of being wrong if the Monte Carlo algorithm did not have sufficient time to reveal enough useful letters. But providing it with a time limit controls the amount of time the algorithm will take, thereby decreasing the risk of another player guessing and getting the prize. It is important to note, however, that this game differs from a standard Monte Carlo algorithm as the game has one correct solution, whereas for a Monte Carlo algorithm, the 'good enough' solution is an acceptable output.

Technical Example: Approximating π

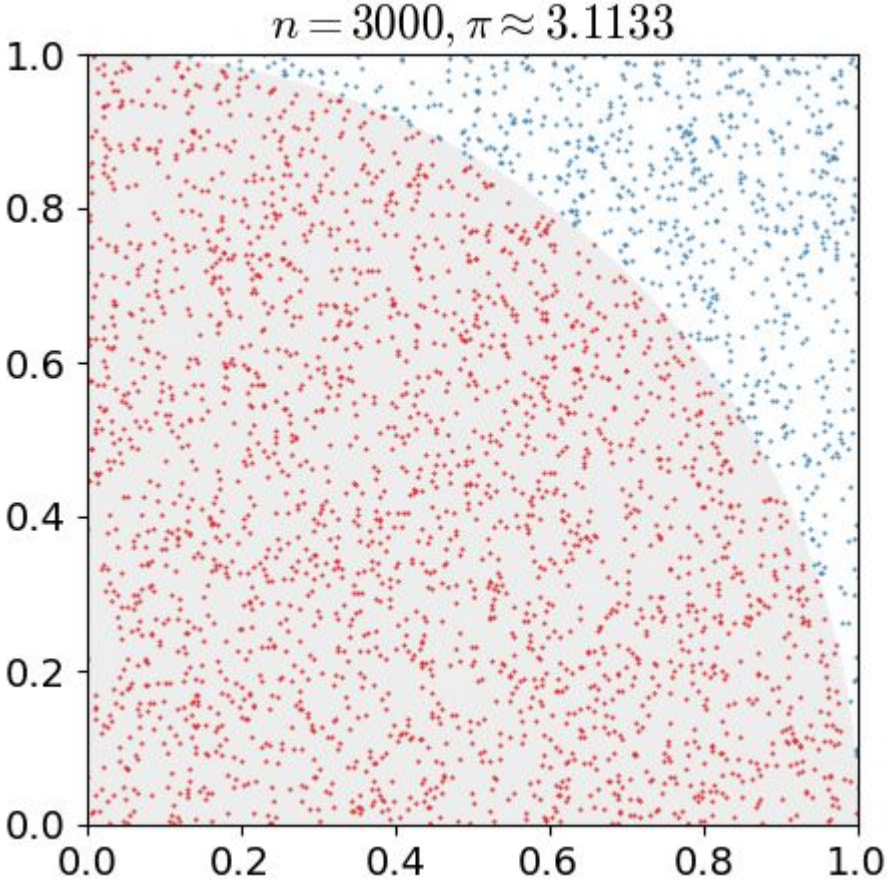
A classic example of a more technical Monte Carlo algorithm lies in the solution to approximating π, the ratio of a circle's circumference to its diameter. This approximation problem is so common that it's actually asked in job interviews for programmer positions at big banks and other mathematically rigorous companies.

Imagine you're blind and someone asks you to find the shape of an object. Intuitively, the solution would be to touch the object in as many places as possible until a familiar object come to mind, at which point you make a guess. The same strategy is applied when approximating π.

Start with a Cartesian plane (x,y coordinates) with an x-axis from -1 to 1, and a y-axis from -1 to 1. This will be a 2×2 box. Then generate many random points on this grid. Count the number of points, *C*, that fall within a distance of 1 from the origin $(0,0)$, and the number of points, *T*, that don't. For instance, $(0.5,0)$ is less than 1 from the origin, but $(-0.9,0.9)$ is $\sqrt{(1.62)}$ from the origin, by the pythagorean theorem.

We can now approximate π by knowing that the ratio of the area of a circle to the area of the rectangular bounds should be equal to the ratio of points inside the circle to those outside the circle; namely,

$$(\pi\, r^2)/(4r^2) \cong C/T$$


$n = 3000, \pi \approx 3.1133$
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

Las Vegas Algorithms

Intuition:

A Las Vegas algorithm is a randomized algorithm that always produces a correct result, or simply doesn't find one, yet it cannot not guarantee a time constraint; the time complexity varies on the input. It does, however, guarantee an upper bound in the worst-case scenario.

Las Vegas algorithms occur almost every time people look something up. Think of a Las Vegas algorithm as a task that when the solution is found, it has full confidence that it is the correct solution, yet the path to get there can be murky. As an extremely generalized and simplified example, a Las Vegas algorithm could be thought of as the strategy used by a user who searches for something online. Since searching every single website online is extremely inefficient, the user will generally use a search engine to get started. The user will then surf the web until a website is found which contains exactly what the user is looking for. Since clicking through links is a decently randomized process, assuming the user does not know exactly what's contained on the website at the other end of the, the time complexity ranges from getting lucky and reaching the target website on the first link, to being unlucky and spending countless hours to no avail. What makes this a Las Vegas algorithm is that the user knows exactly what she is looking for, so once the website is found, there is no probability of error. Similarly, if the user's allotted time to surf the web is exceeded, she will terminate the process knowing that the solution was not found.

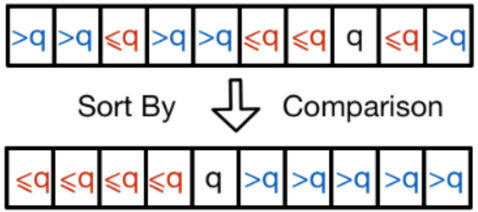
The term for this algorithm, *Las Vegas*, is attributed to mathematician Laszlo Babai, who coined it in 1979 simply as a parallel to the much older ***Monte Carlo* algorithm**, as both are major world gambling centers. However, the gambling styles of the two have nothing to do with the styles of the algorithms, as it cannot be said that gambling in Las Vegas always gives a correct, or even positive, turnout.

Randomized quick sort

A common Las Vegas randomized algorithm is quicksort, a sorting algorithm that sorts elements in place, using no extra memory. Since this is a comparison based algorithm, the worst case scenario will occur when performing pairwise comparison, taking $O(n2)$, where the time taken grows as a square of the number of digits to be sorted grows. However, through randomization, the runtime of this algorithm can be reduced up to $O(n\ log(n))$.

Quicksort applies a divide-and-conquer paradigm in order to sort an array of numbers, A. It works in three steps: it first picks a pivot element, $A[q]$, using a random number generator (hence a randomized algorithm); then rearranges the array into two subarrays $A[p\ldots q-1]$ and $A[q+1\ldots r]$, where the elements in the first and second arrays are smaller and greater than $A[q]$, respectively.

The algorithm then recursively applies the above steps of quicksort on the two independent arrays, thereby outputting a fully sorted array.



Pseudo code:

```
INPUT:
    Array A of n elements
def Randomized_quicksort(A):
    If n = 1:
        return A # A is sorted.
    Else:
        i = Random number in range(1, n)
        X = A[i]     # the pivot element
        Partition A into elements < x, x, and >x  # as shown in the figure above.
    Execute Quicksort on A[1 to i-1] and A[i+1 to n].
     Combine the responses in order to obtain a sorted array.
```

In the worst case scenario, this algorithm takes O(n^2) time to sort n digits in case the pivot element chosen at random is the first or last element in the array.

The following is a **time complexity** analysis for the worst-case scenario of quicksort. Since each insertion and removal takes $O(1)$ time, and partitioning takes $O(n)$ time, as every item must be iterated over to create both subarrays, if the smallest or the largest elements in the array are picked, the partition will result in two arrays: one of size 0, since there will be no items at one extreme of the array, and an array of size $O(n-1)$; excluding the pivot element chosen. Therefore, the runtime analysis is the following:

$$T(n) = T(0) + T(n-1) + \Theta(n)$$

$$T(n) = \Theta(1) + T(n-1) + \Theta(n)$$

$$T(n) = T(n-1) + \Theta(n)$$

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

Where the last step follows from the arithmetic series.

Now, using randomized quicksort, where the pivot is chosen at random, and neither the smallest or the largest number in the array are selected, the expected runtime is $O(n\ log(n))$. If luck is on the algorithm's side and middle valued elements are picked each time, the array will constantly be partitioned in half, obtaining the following time complexity:

$$T(n) \leq 2*T(n/2) + \Theta(n) = O(n\ log(n))$$

Atlantic city algorithms

Monte Carlo algorithms are always fast and probably correct, whereas Las Vegas algorithms are sometimes fast but always correct. There is a type of algorithm that lies right in the middle of these two, and it is called the Atlantic City algorithm. This type of algorithm meets the other two halfway: it is almost always fast, and almost always correct. However, designing these algorithms is an extremely complex process, so very few of them are in existence.

Computational complexity

Randomized algorithms have a complexity class of their own, since they operate under Probabilistic Turing Machines. The most basic probabilistic complexity class is called RP, randomized polynomial time algorithms, which encompass problems with an efficient randomized algorithm, taking polynomial time, and recognizes bad solutions with absolute certainty, and correct solutions with probability of at least ½.

Amplification can be used with RP algorithms in order to increase the probability that the correct answer is recognized. Since said recognition occurs with probability ½, the algorithm can be repeated n times in order to obtain the correct solution with probability 1- $(1/2)^n$ times. Therefore, if an RP algorithm is executed 100 times, the chance of obtaining a wrong answer every time is lower than the chance that cosmic rays corrupted the memory of the computer running the algorithm! This makes RP algorithms extremely practical.

The complement of RP class is called co-RP, implying that correct solutions are accepted with absolute certainty, but if the answer is incorrect, the algorithm has ½ probability of outputting *no* .

Randomized algorithms with polynomial time runtime complexity, whose output is always correct, are said to be in ZPP, or zero-error probabilistic polynomial time algorithms. ZPP, then, are Las Vegas algorithms which are both in RP and in co-RP. Lastly, the class of problems for which both YES and NO-instances are allowed to be identified with some error, commonly known as Monte Carlo algorithms, are in the complexity class called BPP, bounded-error probabilistic polynomial time.

Derandomization

It is worth mentioning that a considerable amount of mathematicians and theoreticians are working on producing an efficient derandomization process, whereby all the randomness is removed from an algorithm while keeping runtimes the same. Although a few techniques have been developed, whether $P=BPP$, where P is a polynomial-time algorithm, is still an open problem. Most theoreticians believe that $P!=BPP$, but that has not been formally proven yet. For a more technical reference on the question of the implications involving $P=BPP$, refer to the paper In a World of P=BPP, by Oded Goldreich's, professor of Computer Science at the Faculty of Mathematics and Computer Science of Weizmann Institute of Science, Israel.

Real World Applications

Quick glance at algorithms discussed so far

Applications of array:

- Arranging particular data type in a sequential arrangement: storing contacts on our phone, storing speech signals in speech processing etc.
- Implementing stack and queue, adjacency matrix in graphs, Implementing hash tables and heaps.

Applications of linked lists:

- **Singly-linked lists:** Maintaining a directory of names, Performing arithmetic operations on long integers, Manipulation of polynomials, Implementing stack and queue, Adjacency list in graphs, Hashing by chaining method, Dynamic memory allocation.
- **Doubly-linked lists:** Implementing LRU cache, Implementing fibonacci heap, Thread scheduler in operating systems, representing deck of cards game, implement undo and redo functionalities, previous and next page in web browsers.
- **Circular linked lists:** Process scheduling in operating, keep track of turn in multiplayer game

Applications of stacks:

- Storing browser history, UNDO/REDO operations in text editor.
- Process scheduling, static memory allocation
- Storing function calls in recursion
- In IDE or compiler to know missing braces.
- Expression evaluation, syntax parsing.

Applications of queue:

- Process scheduling in operating systems
- Interprocess communication
- Waiting lists in applications
- Customer care calls management

Applications of hashing:

- Accessing website using keywords in search engine
- Find phone numbers in mobile devices, Employee information system
- Spelling checkers in word processing software, symbol table in a compiler
- Encrypting critical data (Cryptography)

Applications of tree data structure:

- **Binary tree:** Store hierarchical data, like folder structure, organization structure, File systems in Linux, Document object model to represent documents with a logical tree.

- **Binary search tree:** Used in applications where continuous insertions and deletion of data is happening, to implement map and set objects in language libraries.

- **Trie data structure:** Auto completion in google search, spell checking in word processing, swipe features in your mobile keypad, Network browser history of the URL's, longest prefix matching used by routers in internet protocols, predictive text technology (T9 dictionary in mobiles)

- **Heap data structure:** Used in implementing efficient priority queues. Scheduling processes in operating systems, dynamic memory allocation, dynamic memory allocation, order statistics

- **Advanced data structures:** Compiler design for parsing syntax, Wireless networking and memory allocation, Range query processing on a data set, finding nearest neighbour for a particular point(k-dimensional tree), to store data on drive(B-tree), fast full text search (Suffix tree).

Application of graph data structure:

- **Shortest path algorithm:** Network routing protocols, Google maps, Shipment routing in e-commerce, robot path planning.
- **Breadth-first search(BFS):** Social network websites to find people with a given distance k from the person, Web crawling in google search, finding all neighbour nodes in peer to peer network like BitTorrent, Network broadcasting, GPS navigation
- **Depth-first search(DFS):** Detecting cycle in a graph, Bipartiteness test in a graph, finding strongly connected components, Solving maze puzzles, Topological sorting, path-finding.
- **Topological sorting:** Used in many different scenarios that involve scheduling several tasks with inter-dependencies, for example: **1)** generates a dependency graph of dependent modules IDE-build systems **2)** Determining order to create complex database tables with inter dependencies. Determining order to take courses and their prerequisites to complete a degree.
- **Other important-applications:** Assigning fastest pick-ups to Uber drivers(Hungarian algorithm), Facebook's friend suggestion algorithm, **Google page ranking algorithm** where webpages are considered to be vertices, Resource allocation graph(Operating systems), Transaction graphs in cryptocurrency(Blockchain, which is a large graph), Artificial neural networks, Product recommendation graphs(Amazon recommendation system).

Applications of dynamic programming

- Sequence alignment, document diffing algorithms, Document distance algorithm(edit distance), Plagiarism detection, a type setting system.
- Duckworth lewis method in cricket, Flight control.
- Speech recognition, Image processing, Machine learning algorithms.
- Economics, financial trading, Bioinformatics, operations research.

Applications of greedy algorithms

- Loss-less data compression of .png and .mp3 file formats (Huffman coding).
- Shortest path algorithms(Dijkstra), Minimum spanning tree(Kruskal's and prim's algorithm), approximation algorithms for NP-hard problems.
- Solving activity selection and other optimization problems

References

https://www.enjoyalgorithms.com/blog/introduction-to-algorithms

https://brilliant.org/wiki/greedy-algorithm/#:~:text=A%20greedy%20algorithm%20is%20a,to%20solve%20the%20entire%20problem.&text=However%2C%20in%20many%20problems%2C%20a,not%20produce%20an%20optimal%20solution.

https://en.wikipedia.org/wiki/Fibonacci_number

https://www.hackerearth.com/practice/algorithms/dynamic-programming/introduction-to-dynamic-programming-1/tutorial/

http://theory.stanford.edu/people/pragh/amstalk.pdf

Thank you