Calculating Running Time

Running Time: Example

SO FAR…….

RUNNING TIME OF FUNCTION DEPENDS ON INPUT SIZE.

RUNNING TIME OF AN ALGORITHM GROWS LINEARLY WITH N (INPUT SIZE).

$T(N) = C_1N^2+N$

Running time: Input size + input type

- Running time depends on input data size and nature of input.
- For this we will have different running time for different cases.

Running Time: Input size

FIND MAXIMUM

```
function max(A)
    max=0
    for 1 <= i < N
        if (A[i]>A[max])
            max=i
    return max
```

FIND MAXIMUM

```
function max(A)
    max=0
    for 1 <= i < N
      if (A[i]>A[max])
          max=i
    return max
```

C_0

C_1*N+C_2

C_3*N+C_4

C_5

C_6

$$T(N)= C_7*N+C_8$$

It grows linearly with N.

Running Time:

In this example running time depends on input size and nature of the input.

LINEAR SEARCH

```
function L_Search(A,x)
    for 0 <= i < N
        if(A[i]==x)
            return i
    return -1
```

Example: Linear Search

LINEAR SEARCH

```
function L_Search(A,x)        (A,7)
    for 0 <= i < N  ------>  C1*N+C2
      if(A[i]==x)   ------>  C3*N
        return i
    return -1       ------>  C4
```

$T(N)=C_5*N+C_6$

Array A

13	8	2	24	5	17	6	9
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

It grows linearly with N.

LINEAR SEARCH

```
function L_Search(A,x)
    for 0 <= i < N
        if(A[i]==x)
            return i
    return -1
```

Array A

13	8	2	24	5	17	6	9
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Function call	Running time
L_search(A,7)	T(N) α N

LINEAR SEARCH

```
function L_Search(A,x)        (A,13)
    for 0 <= i < N   ------> C1
      if(A[i]==x)    ------> C2
        return i     ------> C3
    return -1
```

$T(N)=C_4$

It has a constant running time.

Array A

13	8	2	24	5	17	6	9
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

LINEAR SEARCH

```
function L_Search(A,x)
    for 0 <= i < N
        if(A[i]==x)
            return i
    return -1
```

Array A

13	8	2	24	5	17	6	9
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

WORST CASE FOR LINEAR SEARCH
$T_W(N) \alpha N$
It grows linearly with N.

BEST CASE FOR LINEAR SEARCH
$T_B(N) \alpha 1$
Constant running time.

Function Call	Running Time
L_search(A,7)	T(N) α N
L_search(A,13)	T(N) α 1

Recursion

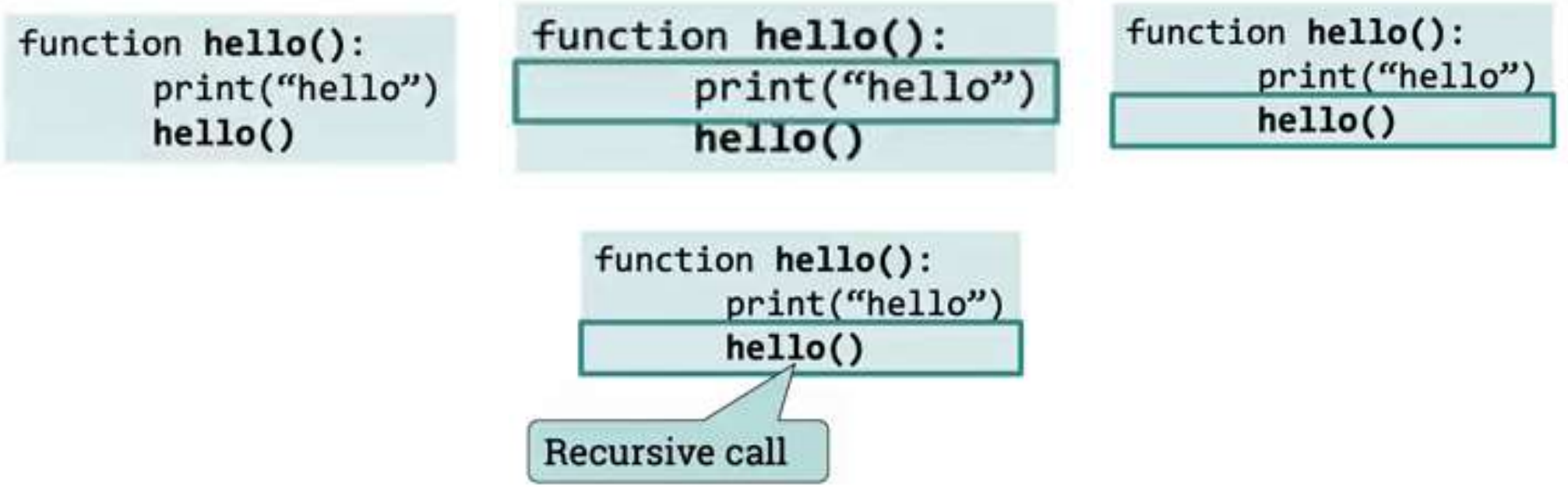
Understanding and writing recursive algorithms

Recursion

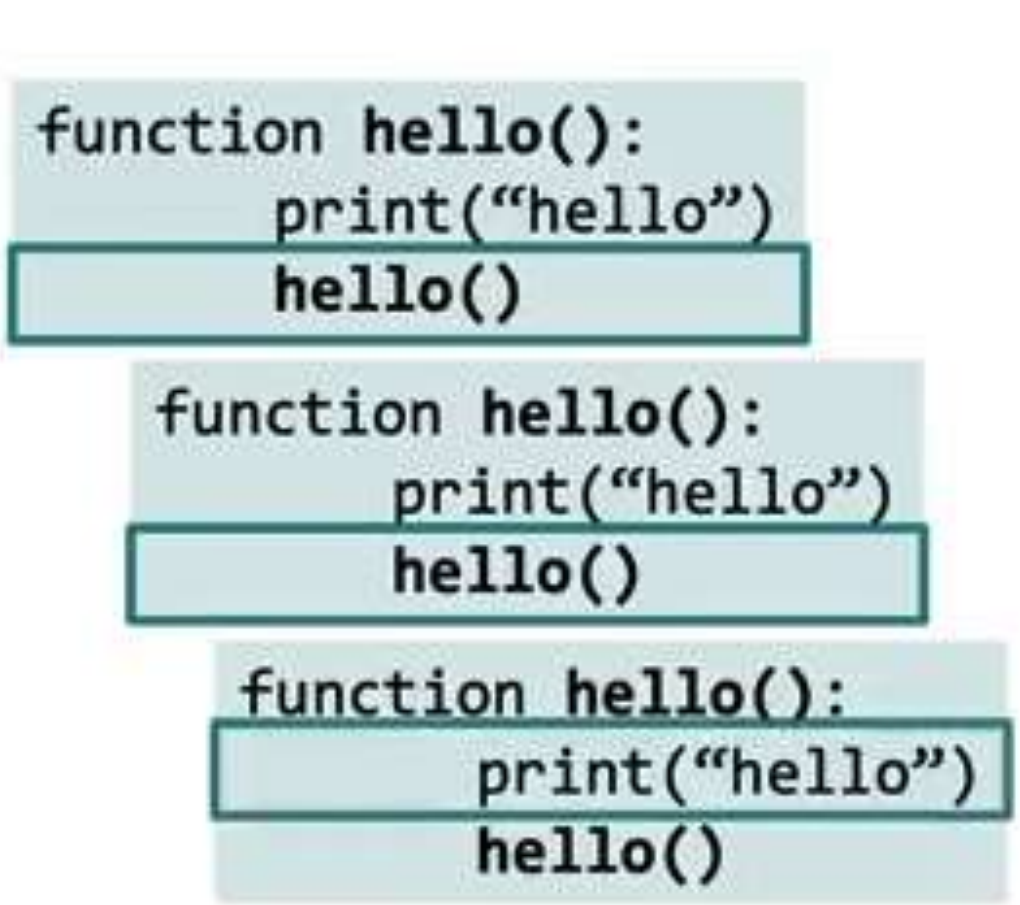
- Understanding
  - How recursion is used in algorithms
  - How recursion works
- Creating
  - Creating own recursive algorithms
- Analyzing
  - Analyzing recursive algorithms to find their running time/time complexity.

Recursive algorithm:

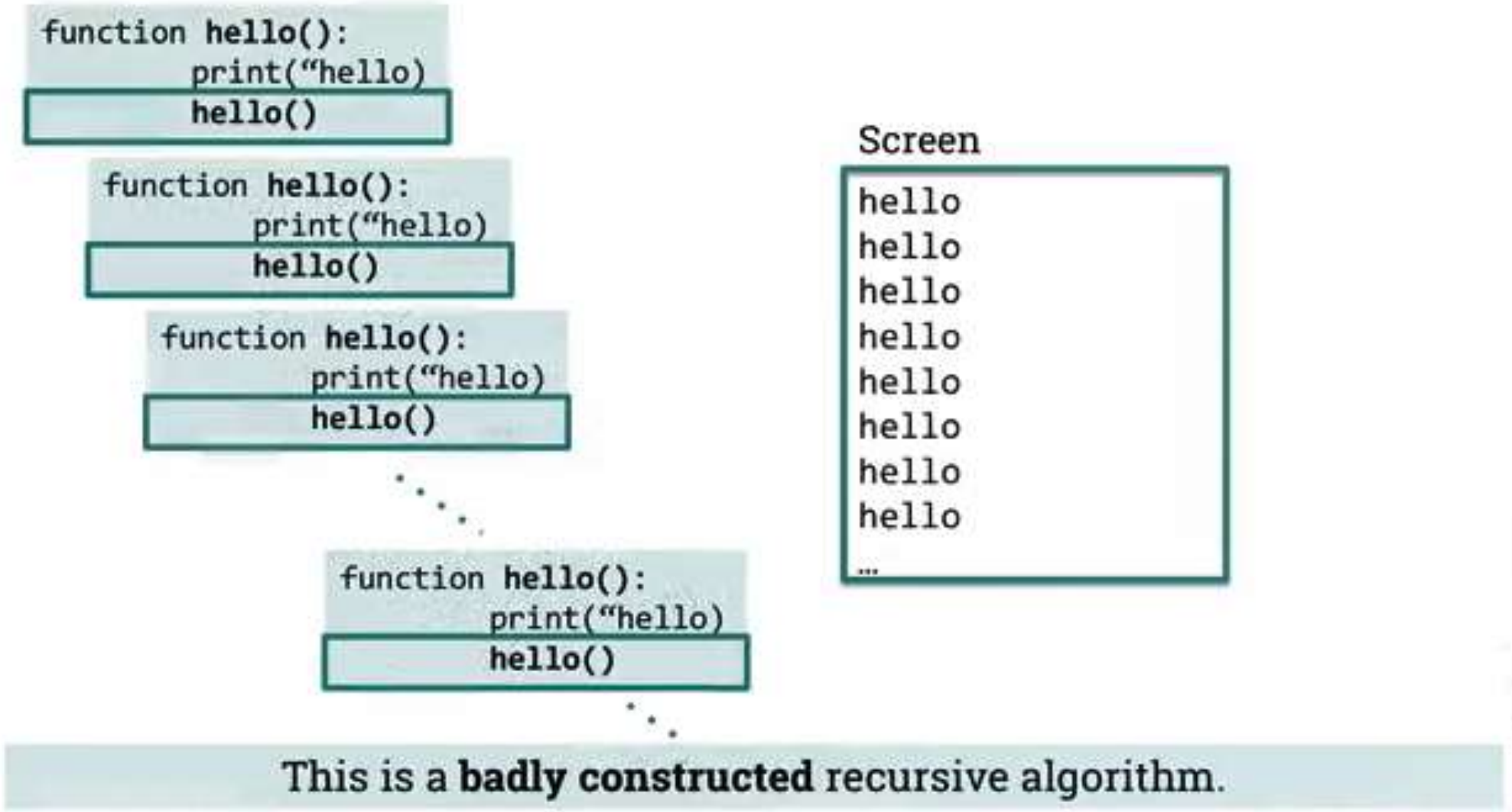
- An algorithm that calls itself.
- For example:



Simulation of algorithm: Example



Simulation of Algorithm:



Simulation of Algorithm: Example

- The above algorithm never ends as shown in last slide.
- That, it will be executed for infinite time.
- It has no condition that terminates the execution of algorithm.

Simulation of Algorithm: Example

- Algorithm with finite time execution.
- Finite number of recursive calls.
- Algorithm with several changes as shown below:

n: integer number

```
function hello(n):
    if (n==0)
        return
    print("hello")
    hello(n-1)
```

n: integer number

```
function hello(n):
    if (n==0)
        return
    print("hello")
    hello(n-1)
```

n: integer number

```
function hello(n):
    if (n==0)
        return
    print("hello")
    hello(n-1)
```

Base case: to stop the recursion

n: integer number

```
function hello(n):
    if (n==0)
        return
    print("hello")
    hello(n-1)
```

Recursive call: getting closer to the base case

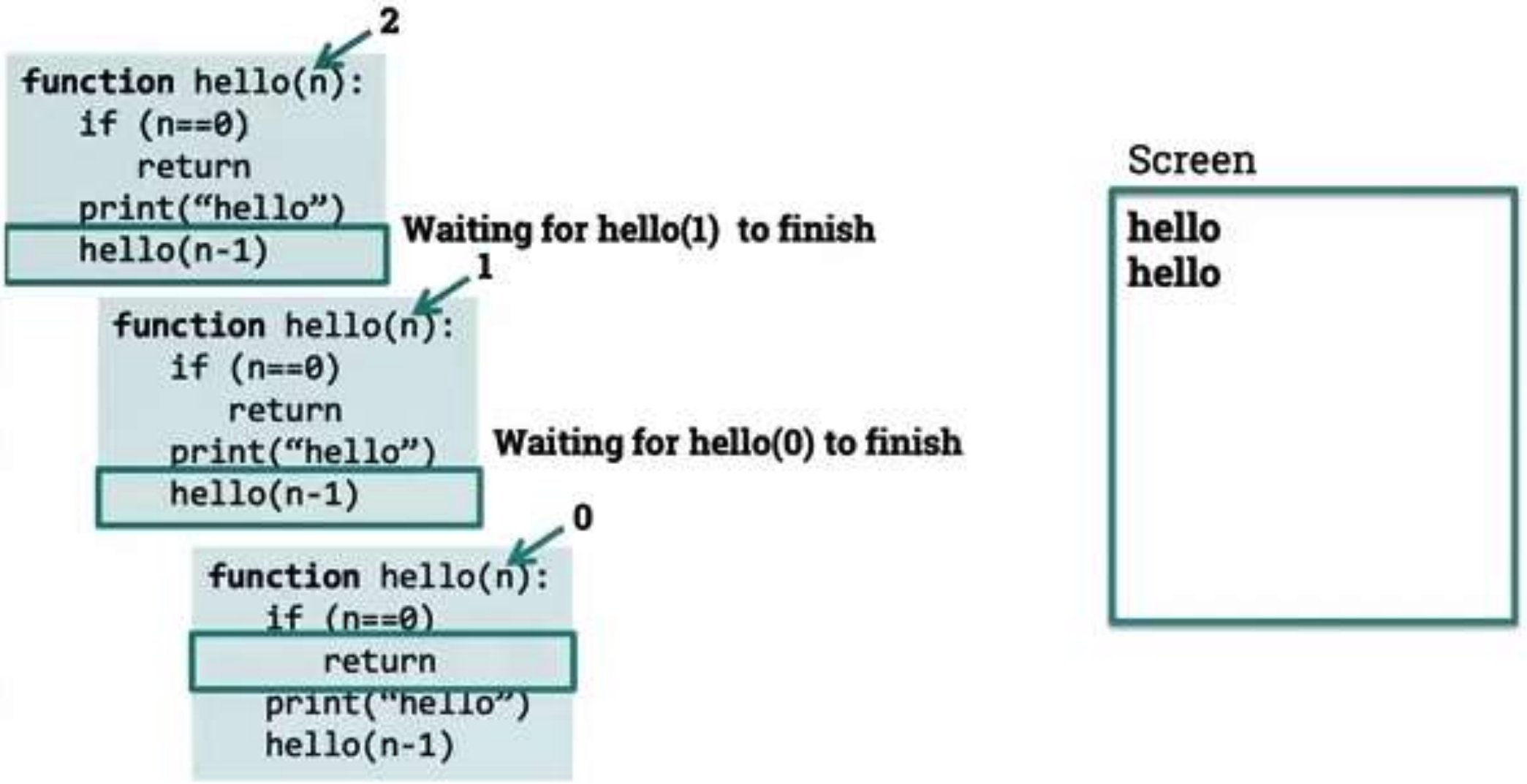
n: integer number

```
function hello(n):
    if (n==0)
        return
    print("hello")
    hello(n-1)
```

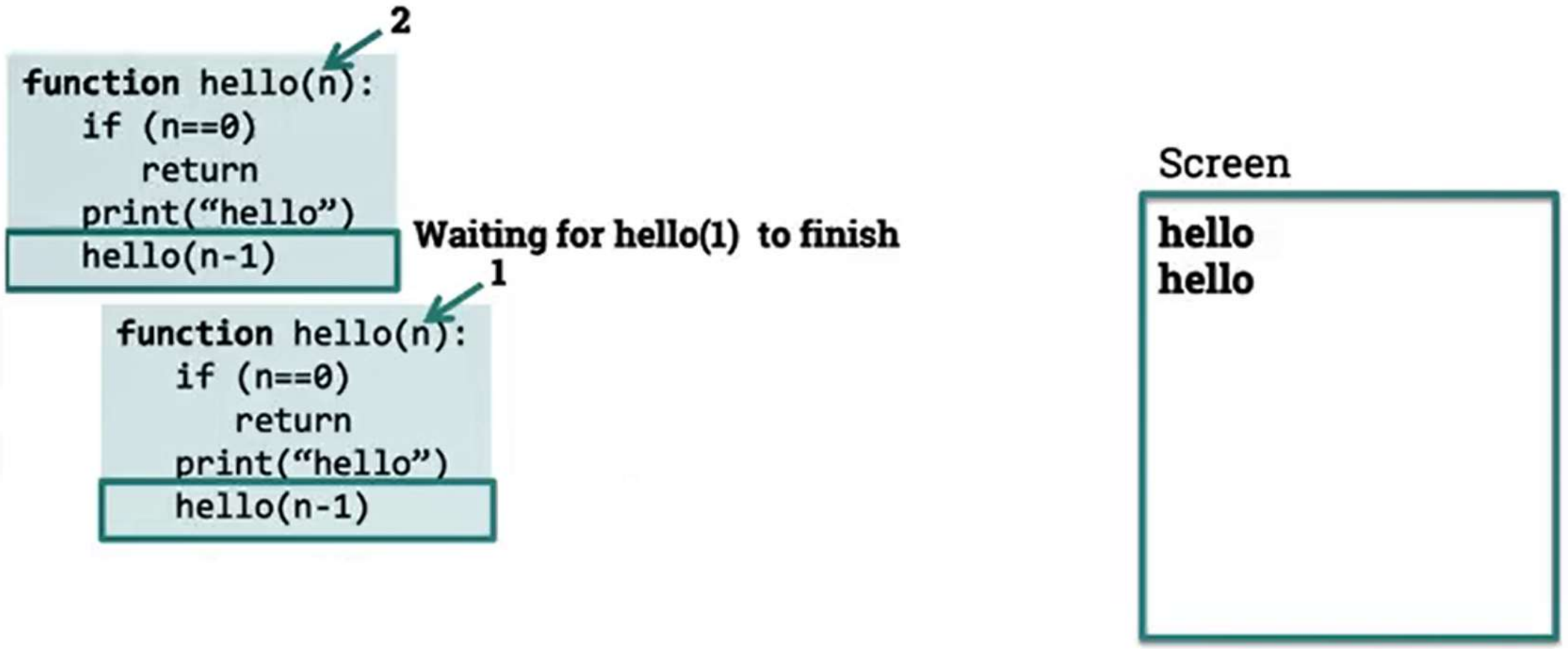
Base case

Recursive call

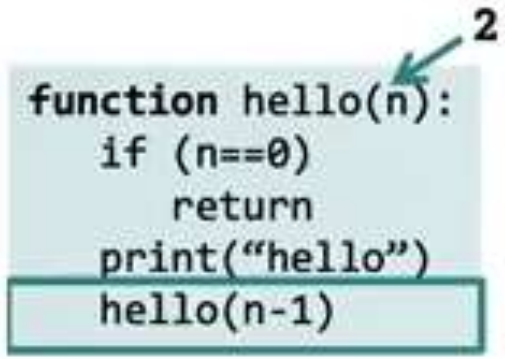
Simulation of the algorithm



Simulation of Algorithm:



Simulation of Algorithm

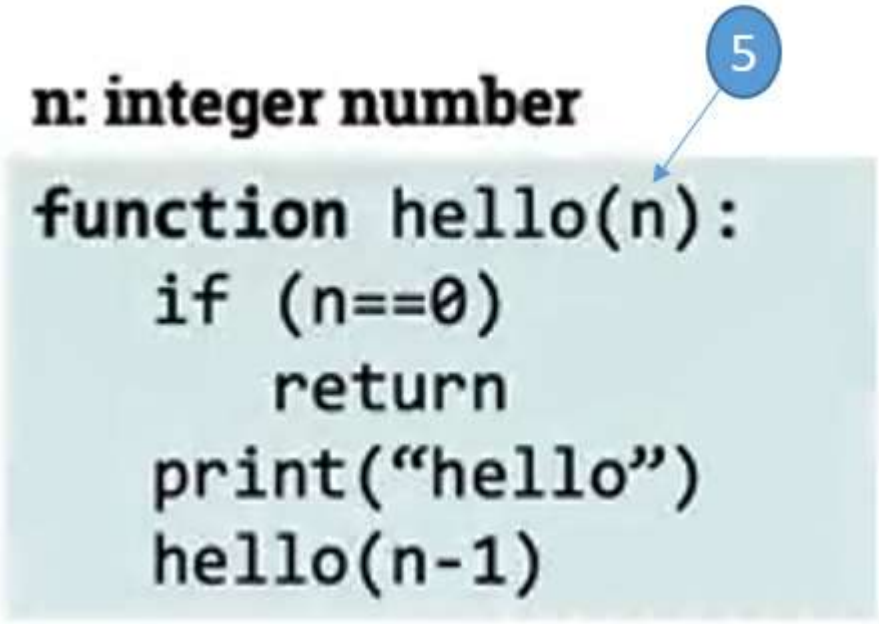


Simulation of algorithm.

- At the end of the algorithm the output will be as:

Conclusion:

- If we pass 5 to the following algorithm then it will print “hello” five times on the screen.



Tracing a recursive algorithm

- Instruction by instruction execution.
- Example: as shown
- Question: What this algorithm is doing?
- Answer: ?

a,b: positive integer numbers

```
function F(a,b):
    if(b==0):
        return a
    return F(a+1,b-1)
```

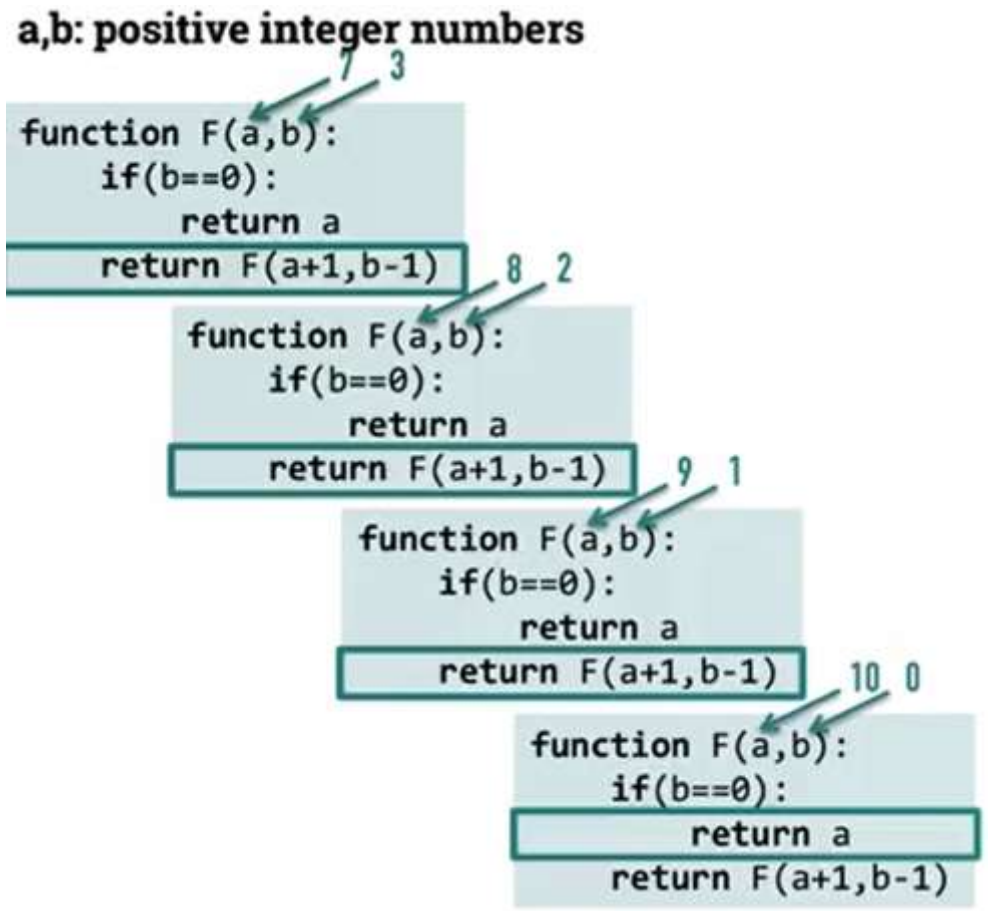
Base Case

a,b: positive integer numbers

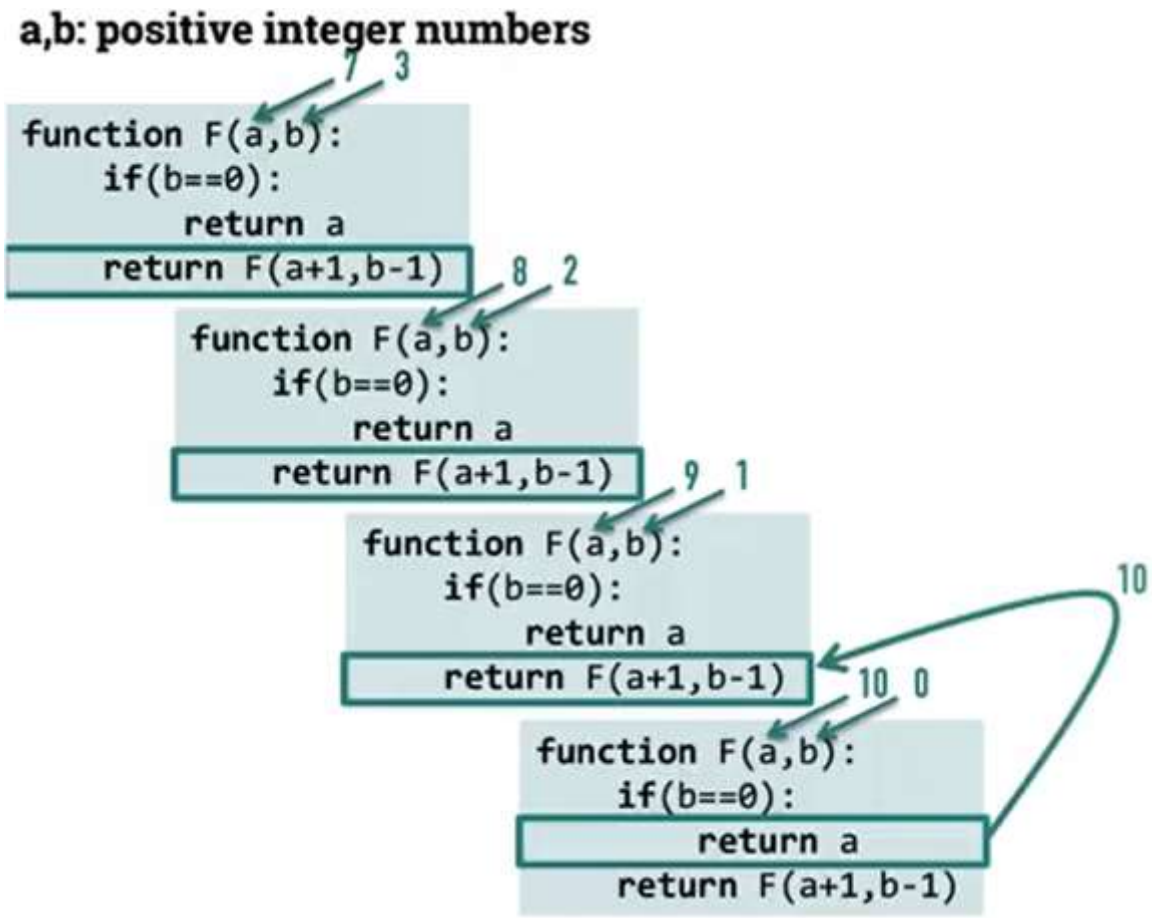
```
function F(a,b):
    if(b==0):
        return a
    return F(a+1,b-1)
```

Recursive Call

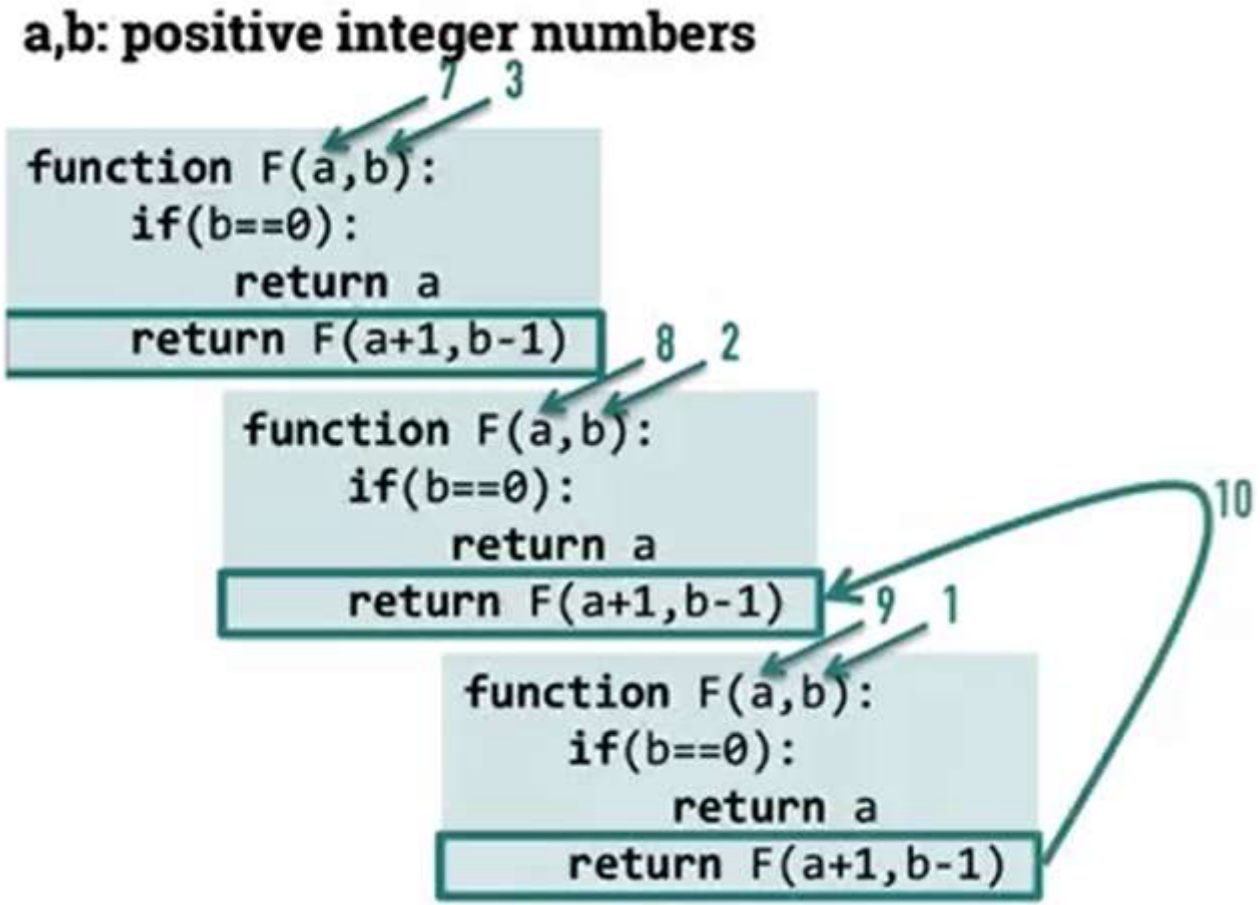
Tracing a recursive algorithm



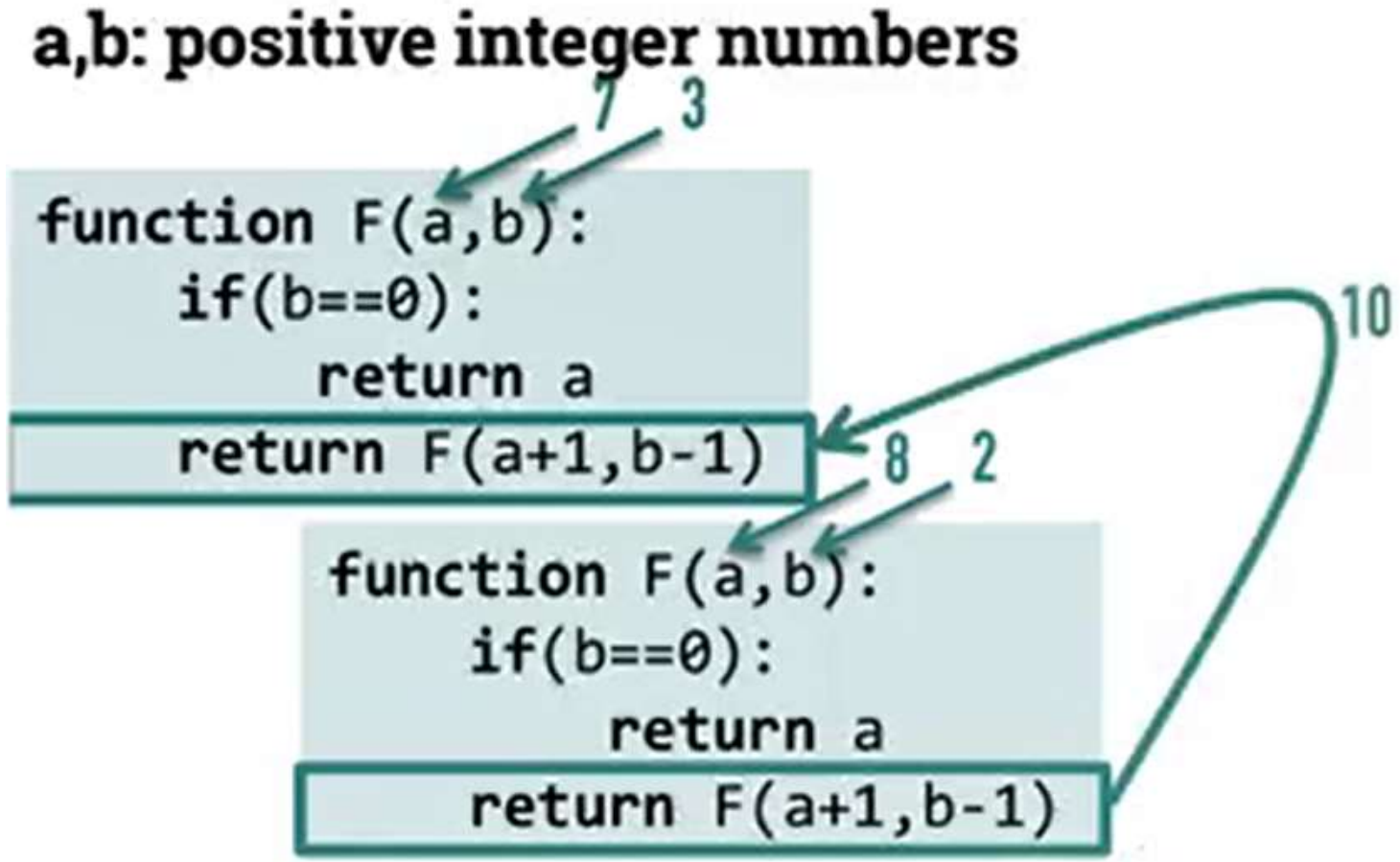
Tracing a recursive algorithm



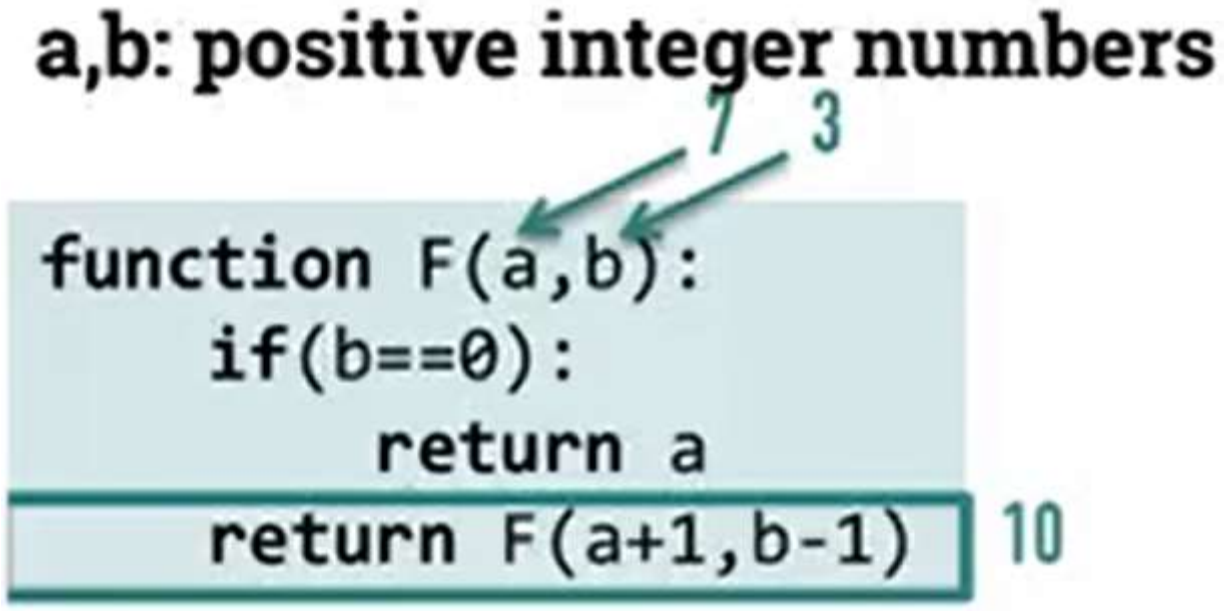
Tracing a recursive algorithm



Tracing a recursive algorithm



Tracing a recursive algorithm



Tracing a recursive algorithm

- Question: What this algorithm is doing?
- Answer: Adding the values of a and b.

Difference between iterative and recursive algorithms

- Iterative algorithm:
  - An iterative algorithm **executes steps in iterations**.
  - Uses loops
  - Example: Printing numbers from n to 0

n: positive integer number

```
function iterCountDown(n):
    for (i=n; i>=0; i--)
        print(i)
```

Difference between iterative and recursive algorithms

- Recursive algorithm:
  - A recursive algorithm is an **algorithm which calls itself with "smaller (or simpler)" input values.**
  - Example: Printing numbers from n to 0.

n: positive integer number

```
function recCountDown(n):
    if(n<0)
        return
    print(n)
    recCountDown(n-1)
```

Working of algorithms.

- Let us check how following algorithms work.
- Both algorithms use “print” statement to print the value of variable i/n as shown in the diagram.

n: positive integer number

```
function iterCountDown(n):
   for (i=n; i>=0; i--)
      print(i)
```

n: positive integer number

```
function recCountDown(n):
   if(n<0)
      return
   print(n)
   recCountDown(n-1)
```

Working of algorithms.

- Issue is to check that how many time information will be displayed on screen and how the information being displayed changes.
- For this, both algorithms must determine the initial value.

n: positive integer number

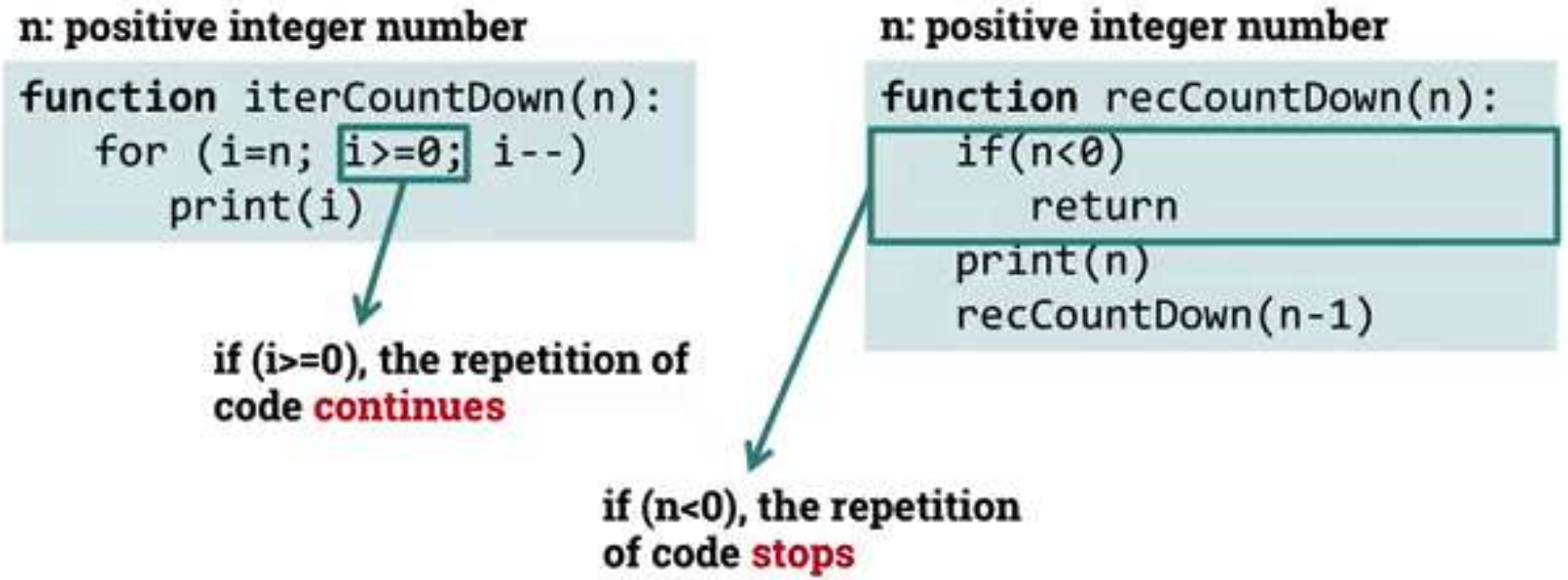
```
function iterCountDown(n):
   for (i=n; i>=0; i--)
      print(i)
```

n: positive integer number

```
function recCountDown(n):
   if(n<0)
      return
   print(n)
   recCountDown(n-1)
```

Working of algorithms.

- Algorithms should also determine when to stop printing.



Working of algorithms.

- Finally, the change in value of variable done to continue or stop the execution.

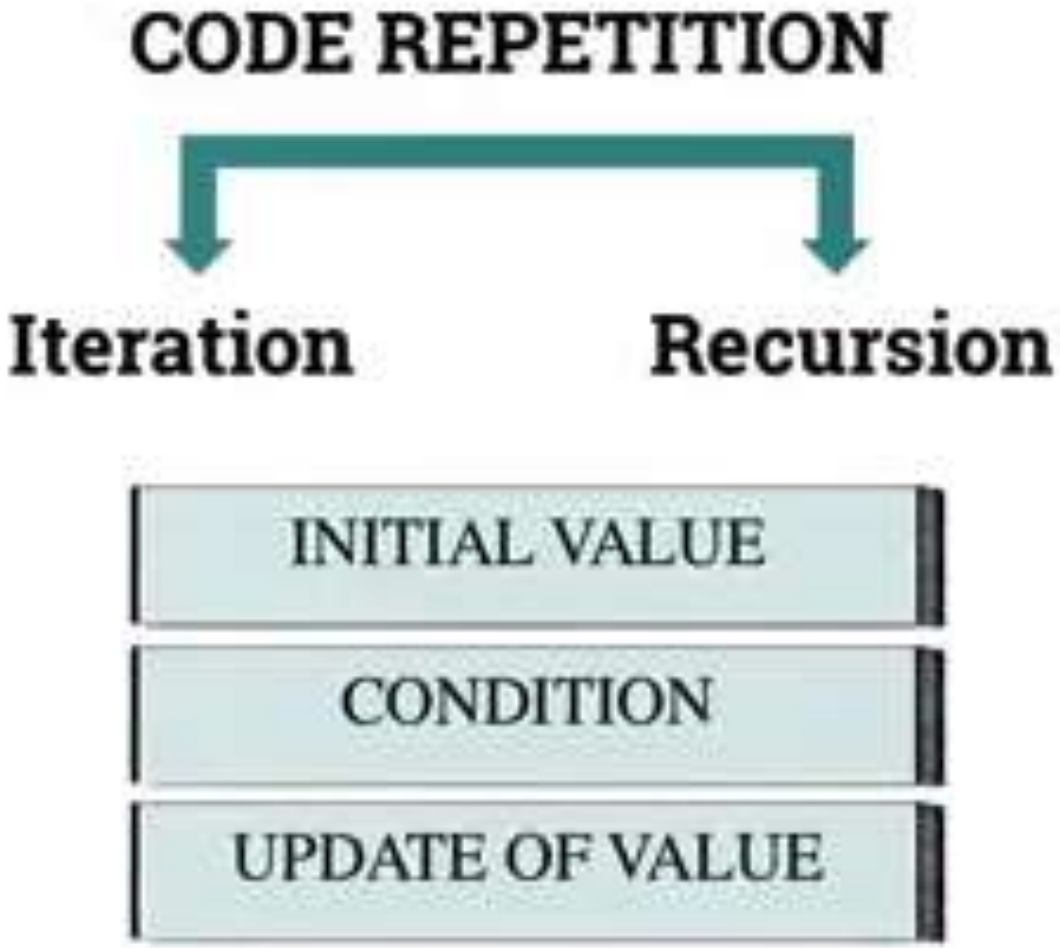
n: positive integer number

```
function iterCountDown(n):
    for (i=n; i>=0; i--)
        print(i)
```

n: positive integer number

```
function recCountDown(n):
    if(n<0)
        return
    print(n)
    recCountDown(n-1)
```

Summary:


CODE REPETITION
Iteration
Recursion
INITIAL VALUE
CONDITION
UPDATE OF VALUE

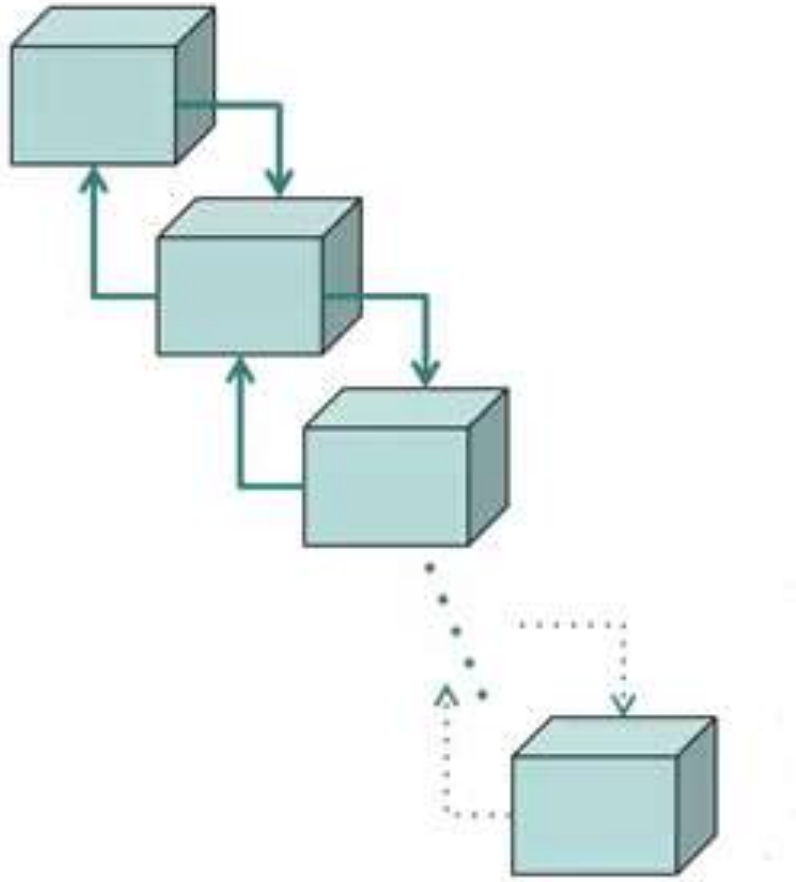
Practicing the recursion:

```
function F(a,b):
    if(b==0):
        return a
    return F(a+1,b-1)
```

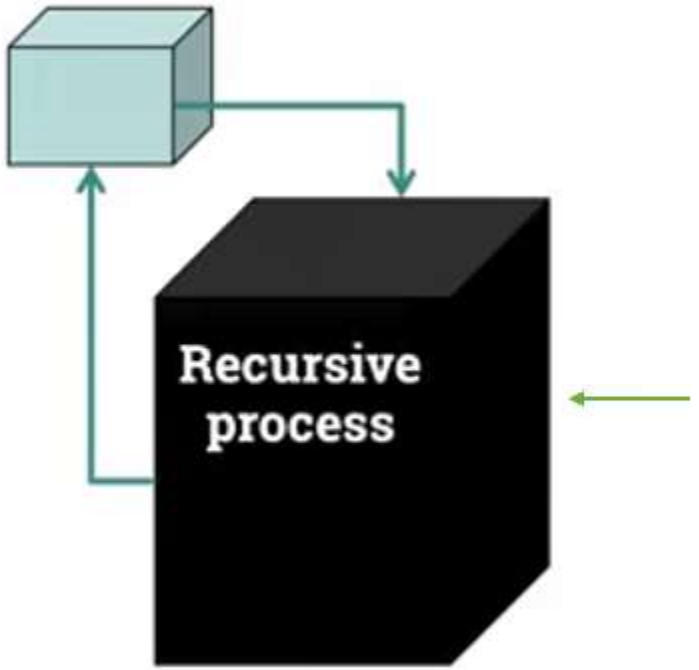
stand by

```
function F(a,b):
    if(b==0):
        return a
    return F(a+1,b-1)
```

Practicing the recursion:

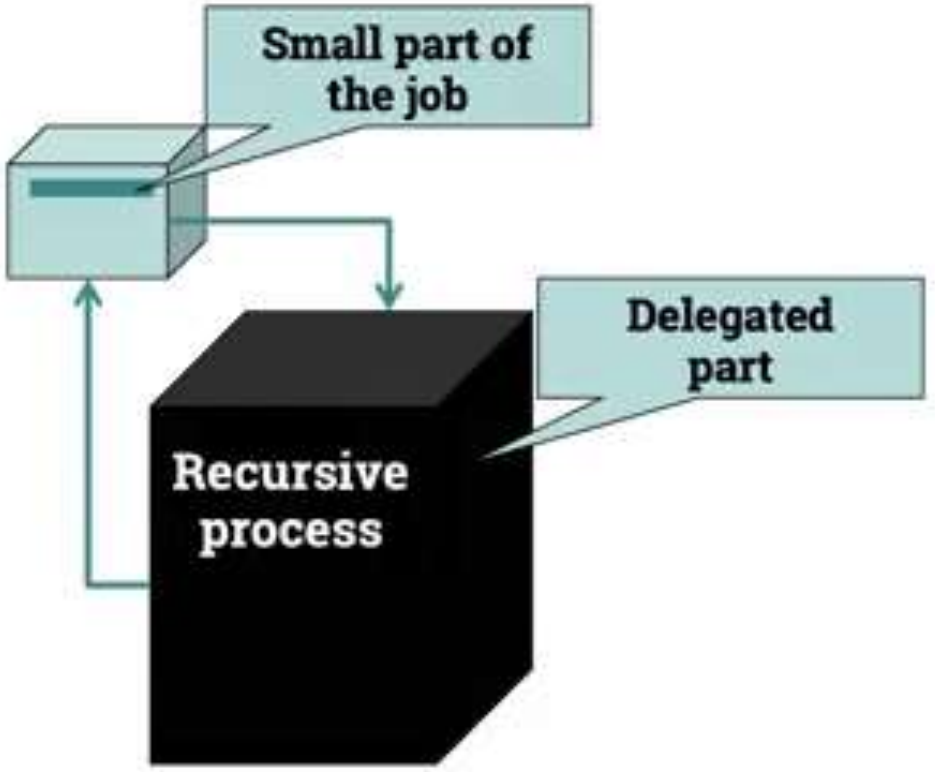


Writing a recursive algorithm

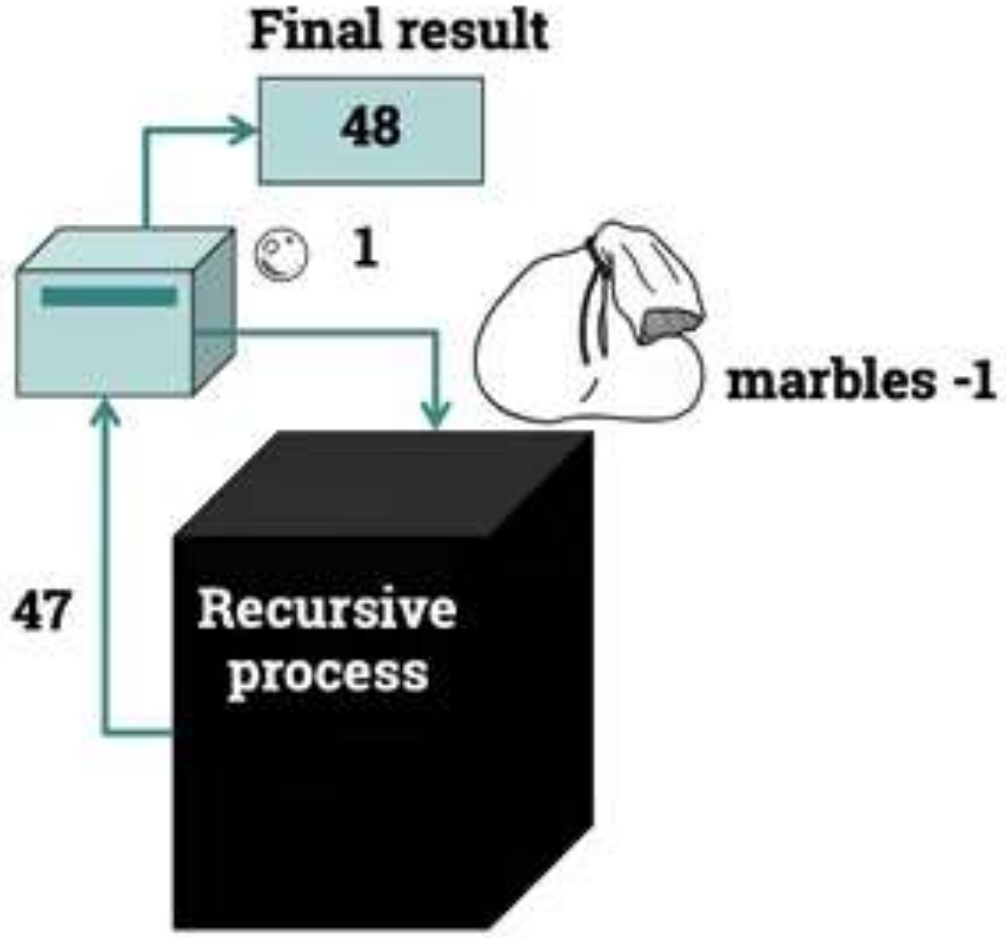


Black box:
We don’t know what will be performed inside this

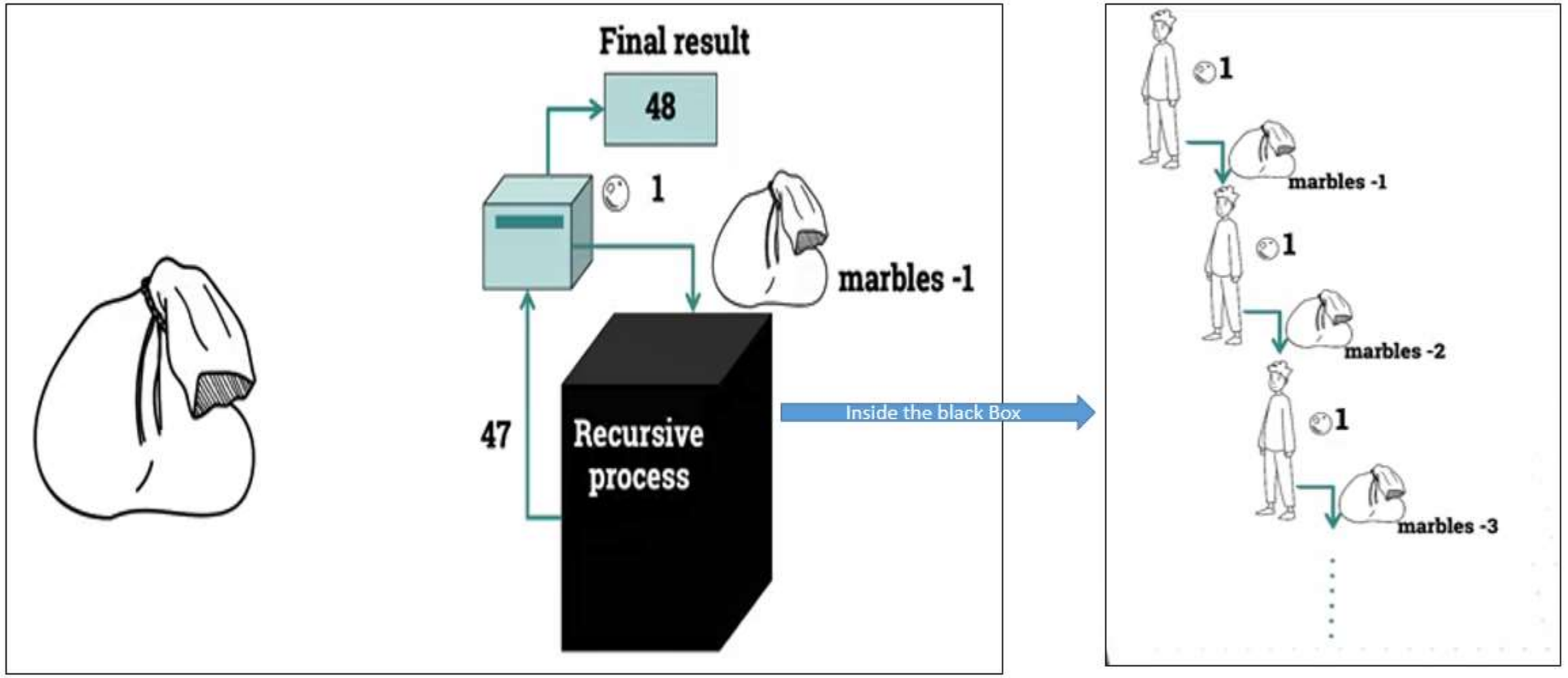
Understanding the Recursive Process:



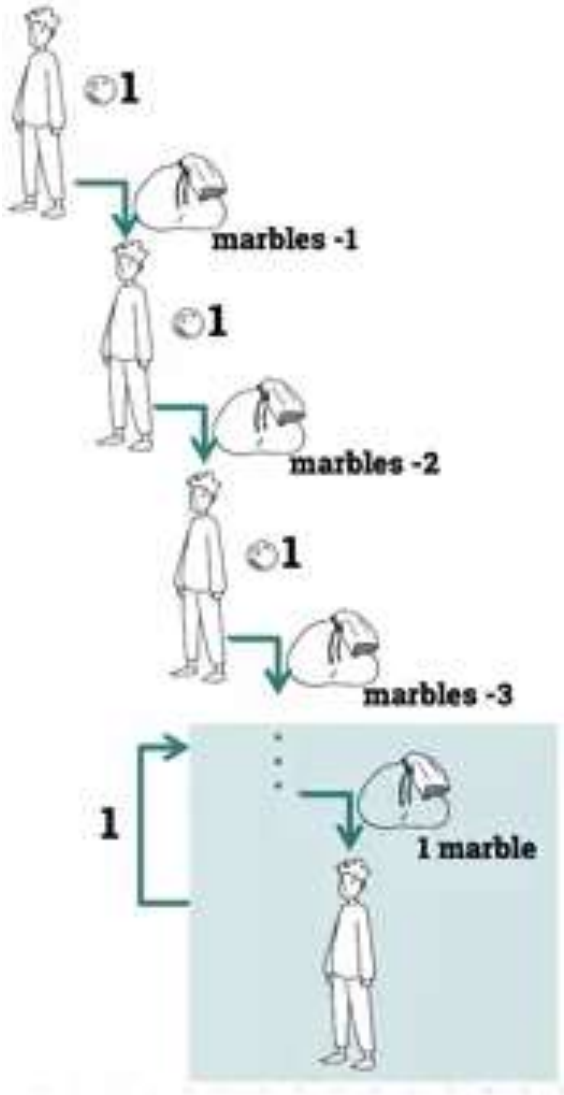
Understanding the Recursive Process:

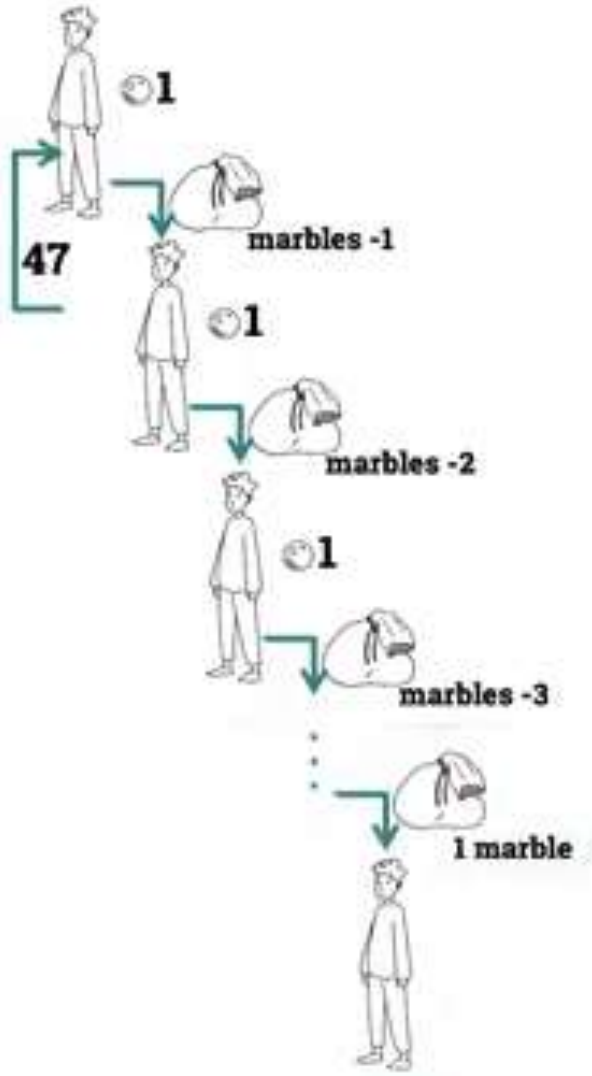


Understanding the Recursive Process: Inside the Black box

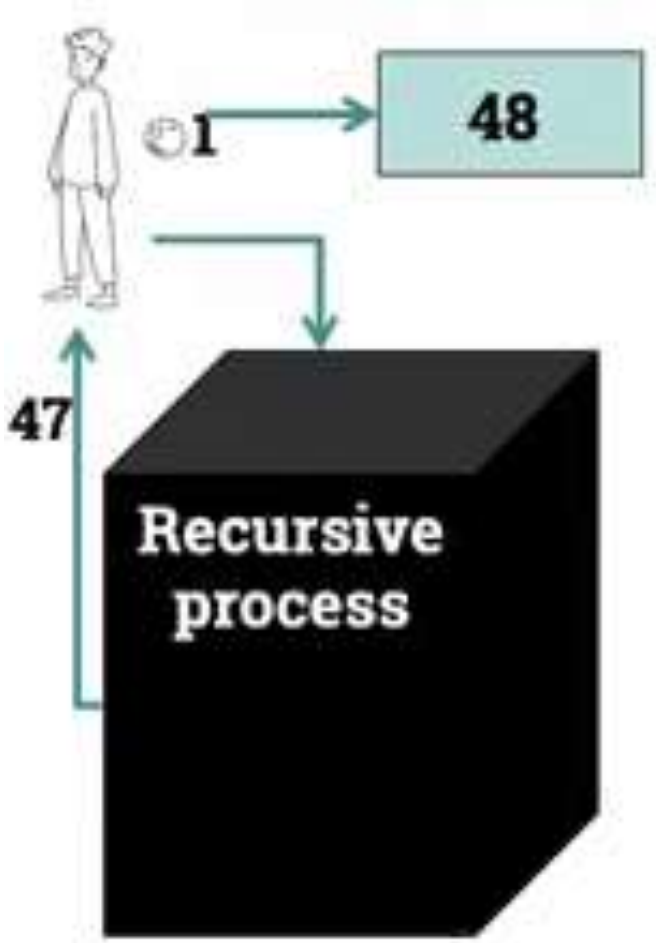


Understanding the Recursive Process: Inside the Black box



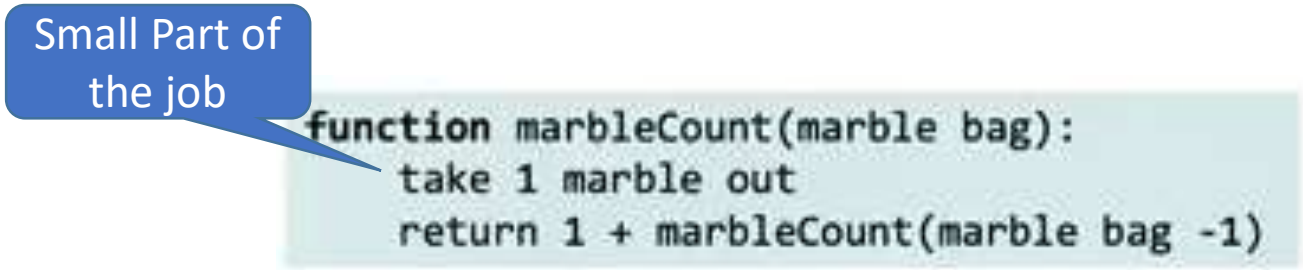


Understanding the Recursive Process: Final Result

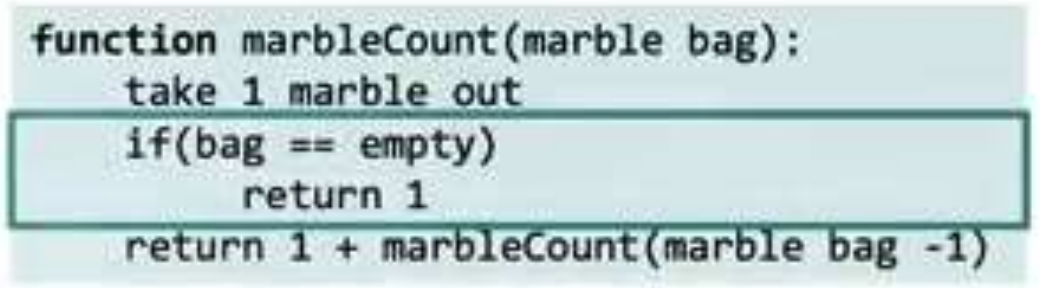


Algorithm of Marble example

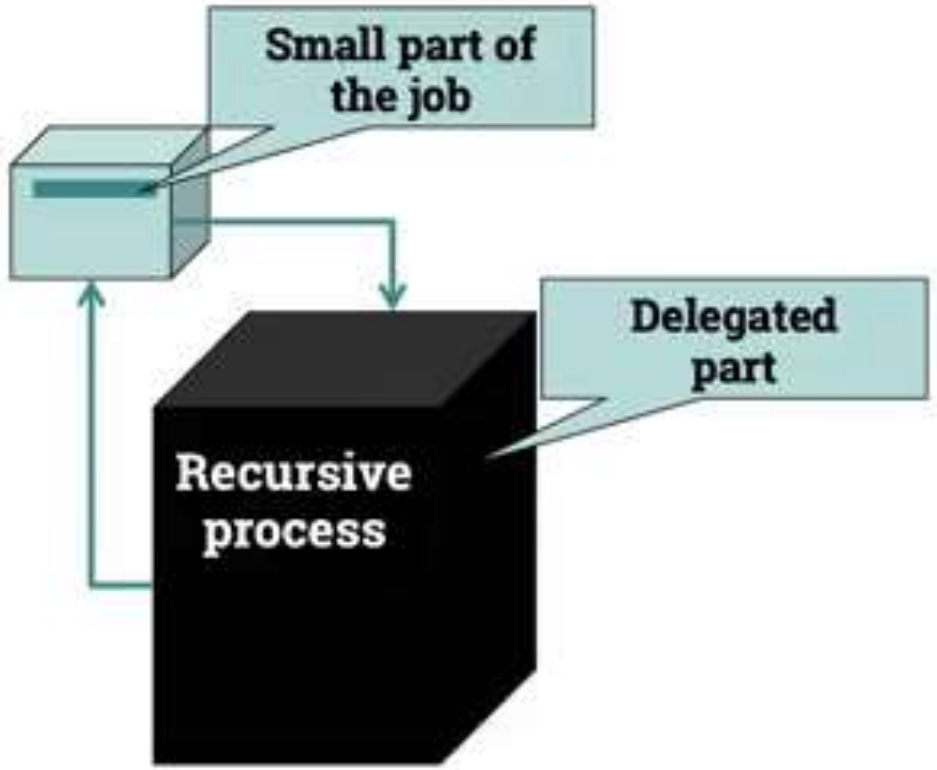
- Algorithm of above example without base case is:



- Algorithms of above example with base case is:

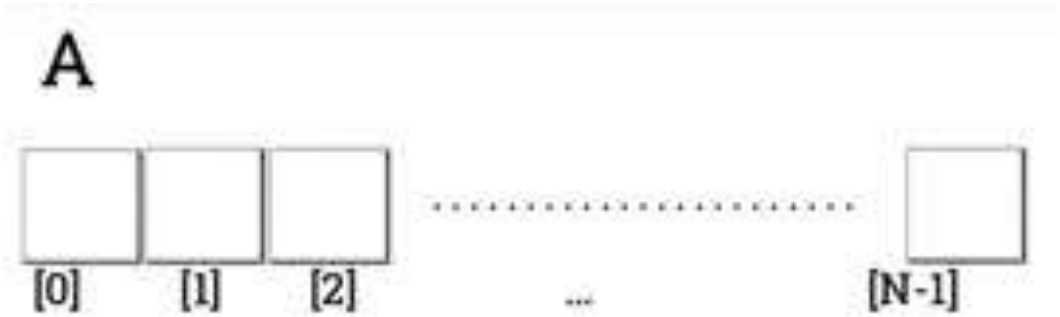


Writing the Recursive Algorithm:



Recursive Linear Search Algorithm

- This will determine whether the number is present in an array or not.
- If value found then return “True”.
- If value not found then return “False”.



Recursive Linear Search Algorithm

```
function recLinSearch(A,N,x)
    if(N<0)
        return FALSE
    if(A[N-1]==x)
        return TRUE
    return recLinSearch(A,N-1,x)
```

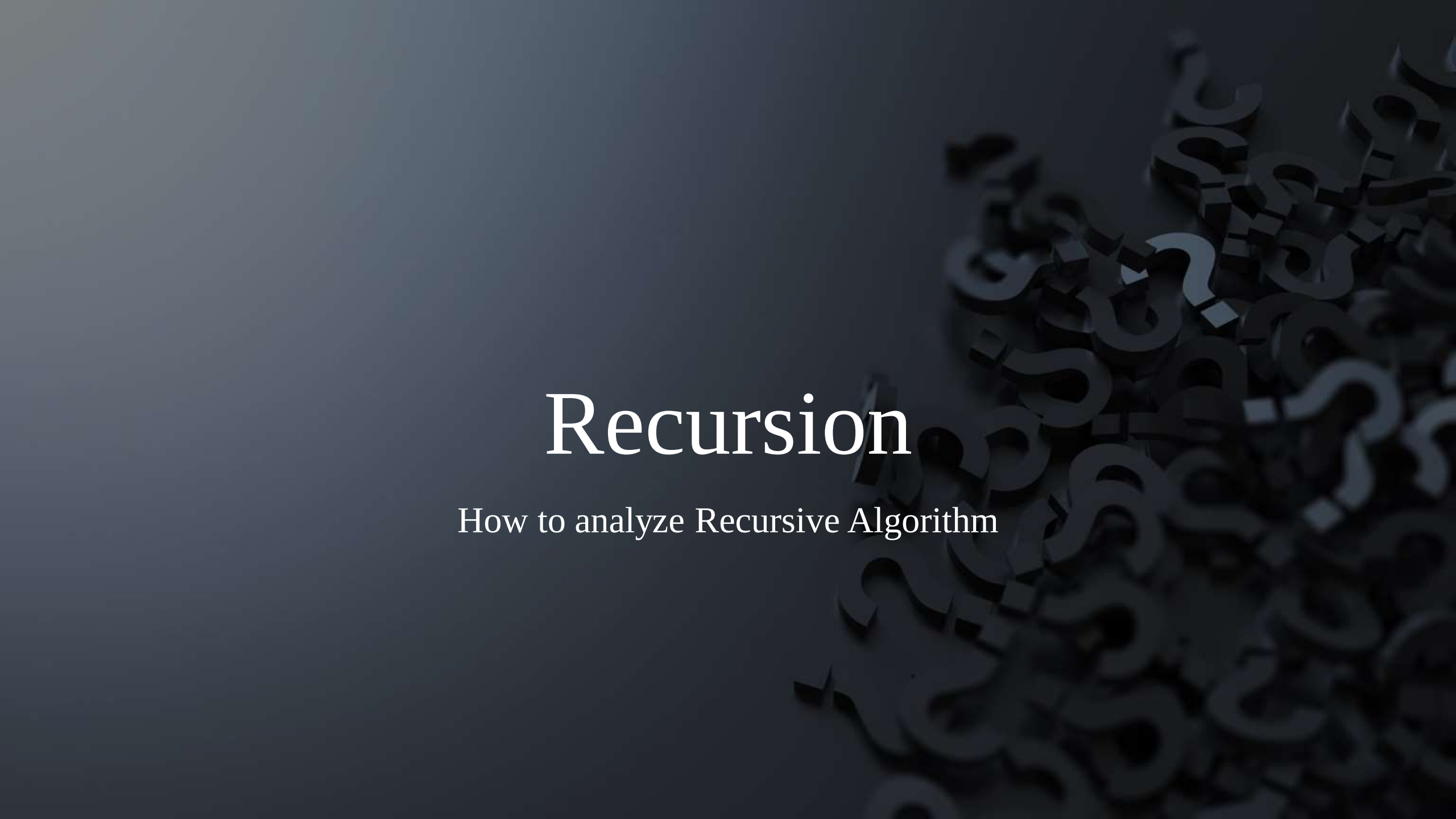
Base Case

```
function recLinSearch(A,N,x)
    if(N<0)
        return FALSE
    if(A[N-1]==x)
        return TRUE
    return recLinSearch(A,N-1,x)
```

Small part of the job

```
function recLinSearch(A,N,x)
    if(N<0)
        return FALSE
    if(A[N-1]==x)
        return TRUE
    return recLinSearch(A,N-1,x)
```

Recursive call



Recursion

How to analyze Recursive Algorithm

Recursion

Understanding

- How recursion is used in algorithms
- How recursion works

Creating

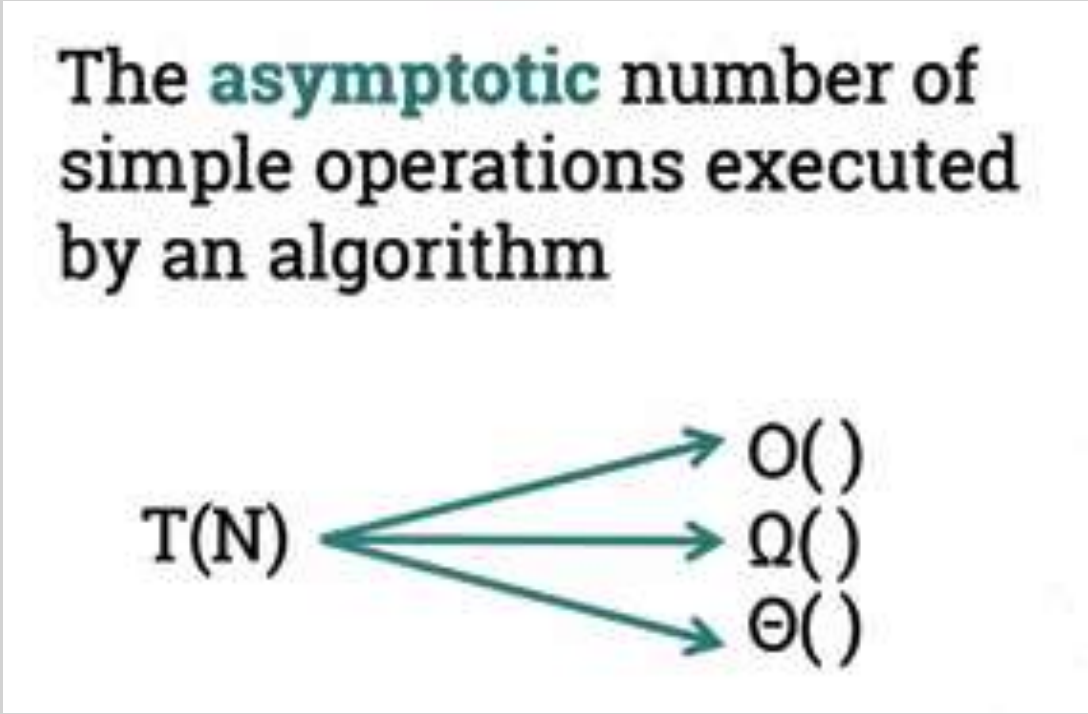
- Creating own recursive algorithms

Analyzing

- Analyzing recursive algorithms to find their running time/time complexity.

Time Complexity

- The time complexity is how long a program takes to process a given input.
- As input changes (amount/contents) the time changes.



Mathematical notation of factorial (Recursive)

Factorial of N

$$N! = \begin{cases} N*(N-1)! & \text{, if } N>0 \\ 1 & \text{, if } N=0 \end{cases}$$

$3! = 3*2!$

$= 3*2*1!$

$= 3*2*1 = 6$

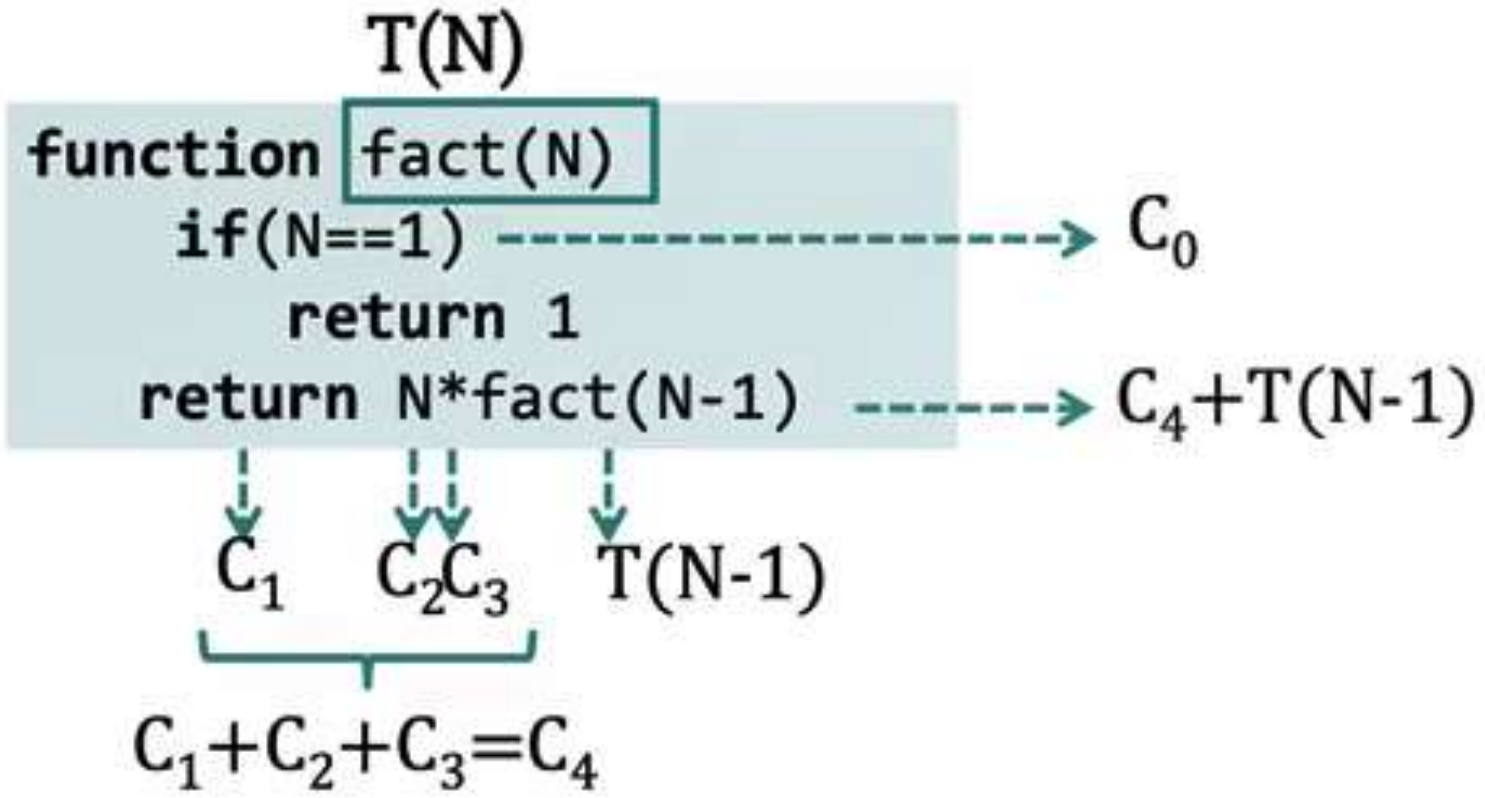
Mathematical notation of factorial (Recursive)

Factorial of N

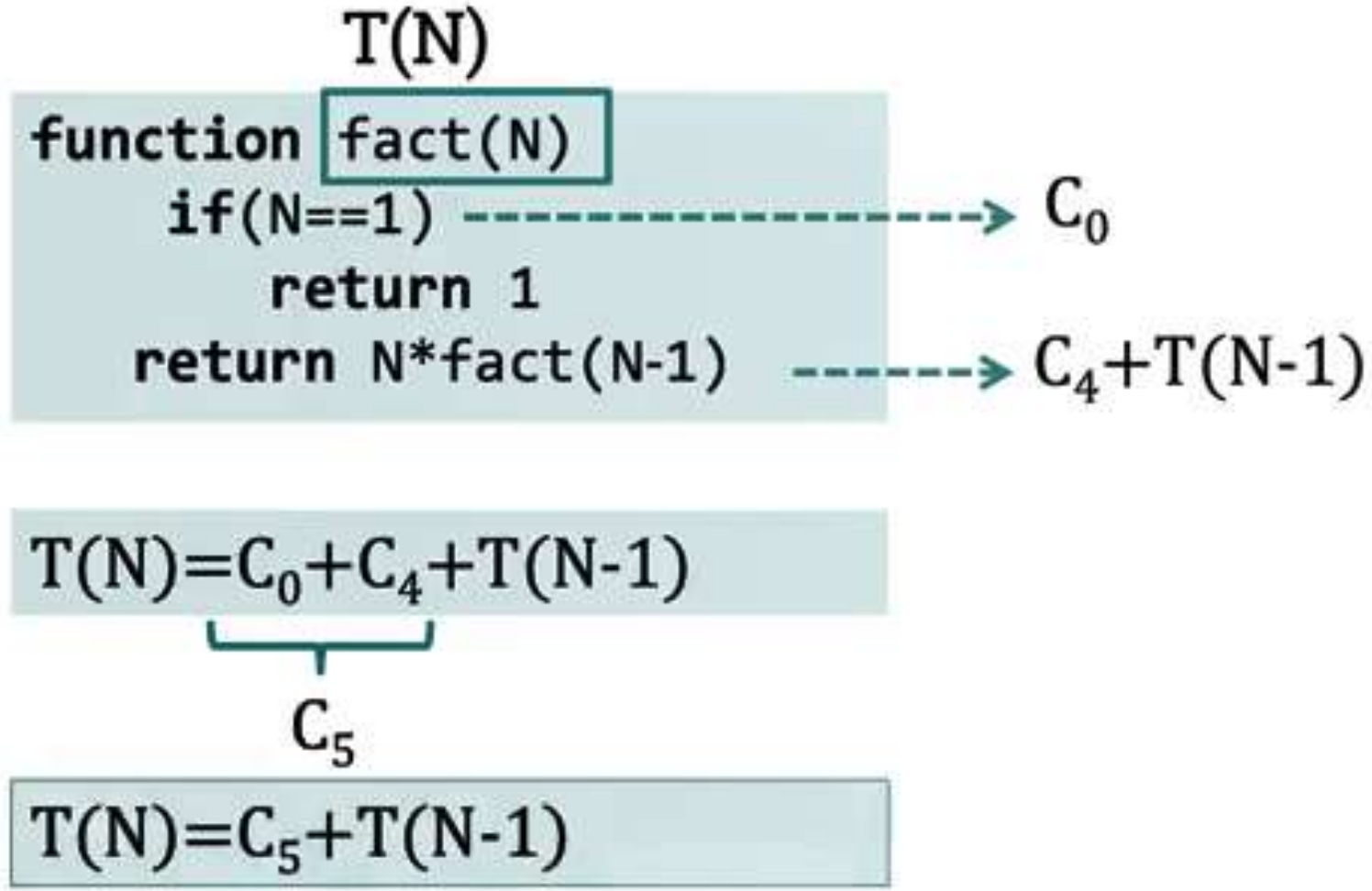
$$N! = \begin{cases} N*(N-1)! & \text{, if } N>0 \\ 1 & \text{, if } N=0 \end{cases}$$

```
function fact(N)
    if(N==1)
        return 1
    return N*fact(N-1)
```

Running time calculation of recursive algorithm: Worst Case



Running time calculation of recursive algorithm: Worst Case



Recurrence Equation

- A recurrence equation defines a sequence based on a rule that gives the next term as a function of the previous term(s).
- We can say as “running time of N depends on the running time of N-1”
- For example:

$$T(N)=C_5+T(N-1)$$

Recurrence equation

T(N) in terms of the running time of smaller inputs

Recurrence Equation

- In the recurrence equation we do not have any explicit expression so we couldn’t block this.
- Q: How to solve?
- A: Two steps.

$$T(N)=C_5+T(N-1)$$

Recurrence equation

Method to solve “Recurrence Equation”.

- Find the value of N for which T(N) is known.
  - This is easy to do for the running time in the best case.
- Expand the right side of the recurrence equation.

$$T(N)=C_5+T(N-1)$$

Recurrence equation

Solving "Recurrence Equation".

1. Find a value of N for which T(N) is known

$$T(N)=C_5+T(N-1)$$

```
function fact(N)
    if(N==1)
        return 1
    return N*fact(N-1)
```

C_0, C_1 $\}$ C

$$T(1)=C$$

Solving "Recurrence Equation".

2. Expand the right side of the recurrence equation

$T(N)=C_5+T(N-1)$

$T(N)=C_5+(\ C_5+T(N-2)\ )$

$T(N)=C_5+(\ C_5+(C_5+T(N-3))\ )$

$\vdots$

$T(N)=k*C_5+T(N-k)$

$\vdots$ $\quad T(1)=C$

$T(N)=(N-1)*C_5+T(N-(N-1))$

$T(N)=(N-1)*C_5+C$

$T(N)=C_5N+C_6$

Grows Linearly with N

Asymptotic Analysis

$$T(N) = C_5 N + C_6$$

$T(N)$ is $O(N)$, $O(N^2)$, $O(N^3)$...

$T(N)$ is $\Omega(N)$, $\Omega(\lg N)$, $\Omega(1)$...

$T(N)$ is $\Theta(N)$

Asymptotic Analysis of Recursive algorithm: Steps

In summary:

- Find its recurrence equation.
- Solve the recurrence equation.
- Asymptotic analysis.

The Master Theorem

- Previously we discussed how to find recurrence equation that defines the running time of the recursive algorithm and how to solve that recurrence equation.
- That is solving a recurrence equation includes:
  - Finding the known value for running time T(N) in the best case.
  - Expanding the right side of the recurrence equation until the known value found in the previous step could be replaced in it.
- This way of solving recurrence equation is effective but very time consuming.
- So……”The Master Theorem” can be used in some cases.

The Master Theorem

- Master theorem makes asymptotic analysis of recursive algorithm much easier.
- It can only be applied for the following structure of recurrence equation.

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

where $a \geq 1$ and $b > 1$

The Master Theorem

- For Example:

$$T(n) = T(n/2) + n$$ ✓

$$T(n) = 2 * T(n) + n$$ ✗

The Master Theorem

- The master theorem classifies recurrence equation in three cases.

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

Case 1:

$f(n) < n^{\log_b a}$

Case 2:

$f(n) = n^{\log_b a}$

Case 3:

$f(n) > n^{\log_b a}$ *and* $a*f(n/b) \leq c\, f(n)$, $c<1$ *and n large*

The Master Theorem

- Once you defined that in which case your recurrence equation lies then you can use Master Theorem to define "Θ" notation.
- So the running time for all these cases of give recurrence equation is:

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

Case 1:

$f(n) < n^{\log_b a}$ $\Longrightarrow$ $T(n) = \Theta(n^{\log_b a})$

Case 2:

$f(n) = n^{\log_b a}$ $\Longrightarrow$ $T(n) = \Theta(n^{\log_b a} \lg n)$

Case 3:

$f(n) > n^{\log_b a}$ $\Longrightarrow$ $T(n) = \Theta(f(n))$

and $a*f(n/b) \leq c\, f(n)$, $c<1$ and n large

The Master Theorem: How to apply

- Firstly, check the values of coefficients a and b.
- If a>=1 and b>1, then master theorem can be applied.
- So…….

$$T(n) = aT(n/b) + f(n)$$
$$a \geq 1 \text{ and } b > 1$$

Case 1:
$f(n) < n^{\log_b a} \Longrightarrow T(n) = \Theta(n^{\log_b a})$

Case 2:
$f(n) = n^{\log_b a} \Longrightarrow T(n) = \Theta(n^{\log_b a} \lg n)$

Case 3:
$f(n) > n^{\log_b a}$ and $a*f(n/b) \leq c\,f(n)$, $c<1$ and n large $\Longrightarrow T(n) = \Theta(f(n))$

$$T(n) = 2T(n/2) + n$$
$$a = 2 \geq 1,\ b = 2 > 1$$
$$\lg_b a = \lg_2 2 = 1$$

Case 1?
$n < n$ ✗

The Master Theorem: How to apply

- Firstly, check the values of coefficients a and b.
- If a>=1 and b>1, then master theorem can be applied.
- So…….

$$T(n) = aT(n/b) + f(n)$$
$$a \geq 1 \text{ and } b > 1$$

Case 1:
$f(n) < n^{\log_b a} \Longrightarrow T(n) = \Theta(n^{\log_b a})$

Case 2:
$f(n) = n^{\log_b a} \Longrightarrow T(n) = \Theta(n^{\log_b a} \lg n)$

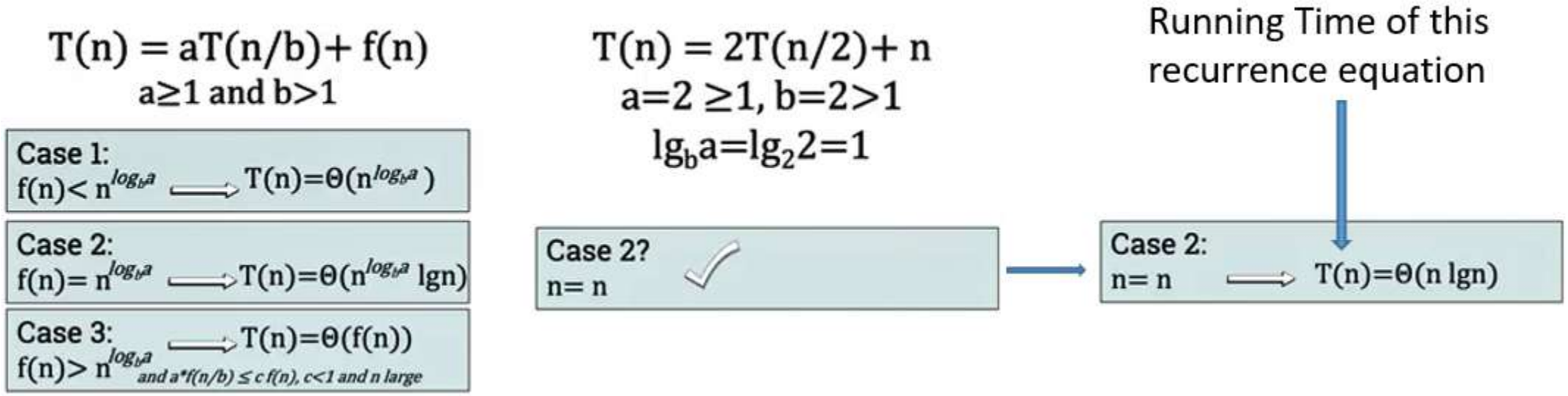
Case 3:
$f(n) > n^{\log_b a}$ and $a*f(n/b) \leq c\, f(n)$, $c<1$ and n large $\Longrightarrow T(n) = \Theta(f(n))$

$$T(n) = 2T(n/2) + n$$
$$a = 2 \geq 1,\ b = 2 > 1$$
$$\lg_b a = \lg_2 2 = 1$$

Case 2?
$n = n$ ✓

The Master Theorem: How to apply

- Firstly, check the values of coefficients a and b.
- If a>=1 and b>1, then master theorem can be applied.
- So…….



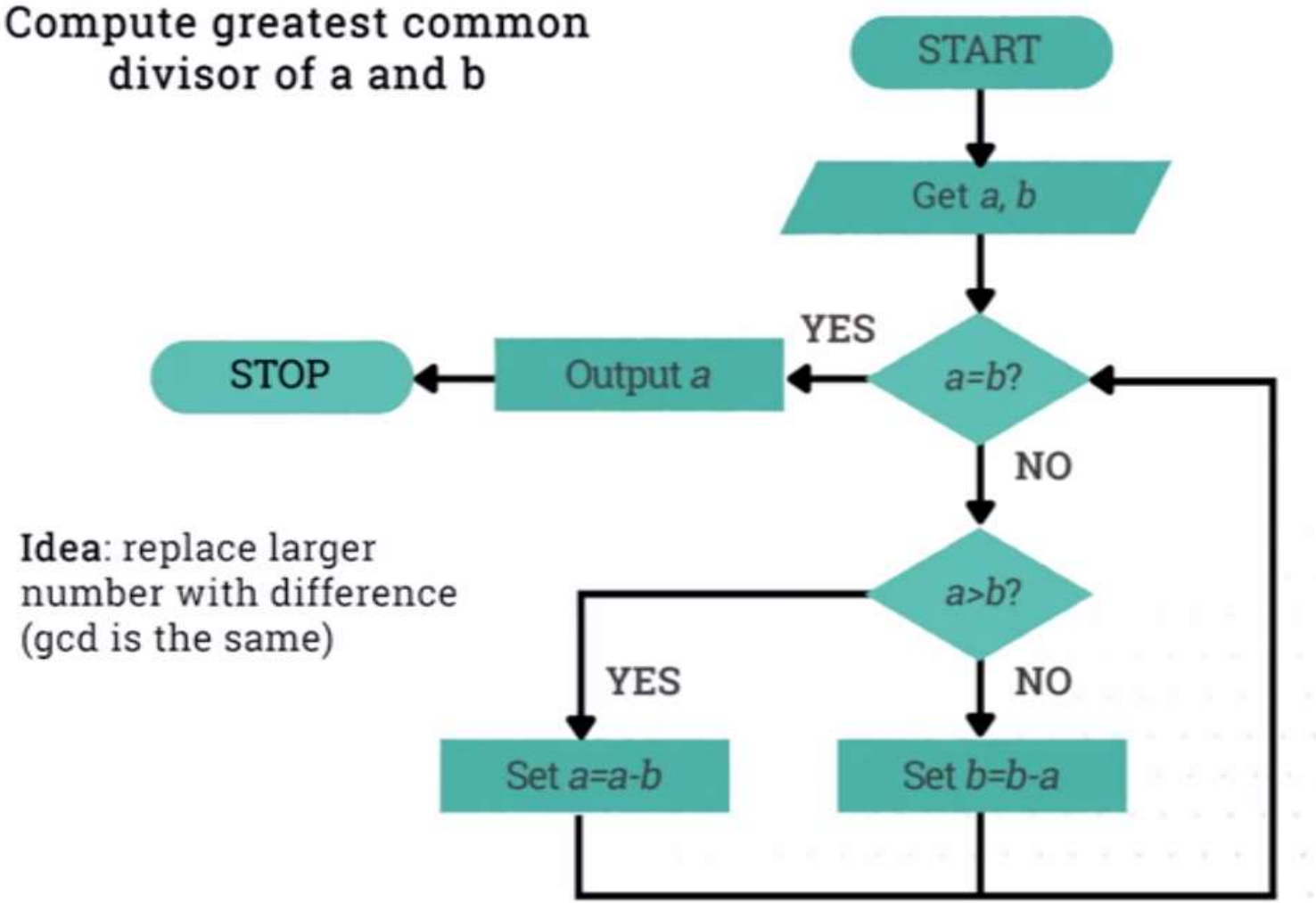
Summary

- Firstly, check the values of coefficients a and b.
  - If a>=1 and b>1, then master theorem can be applied.
- Secondly, check the case in which your equation lies.
- Apply the master theorem.

Recursive Algorithms: Examples

Recursive Euclidean Algorithm

Flowchart of Euclidean algorithm



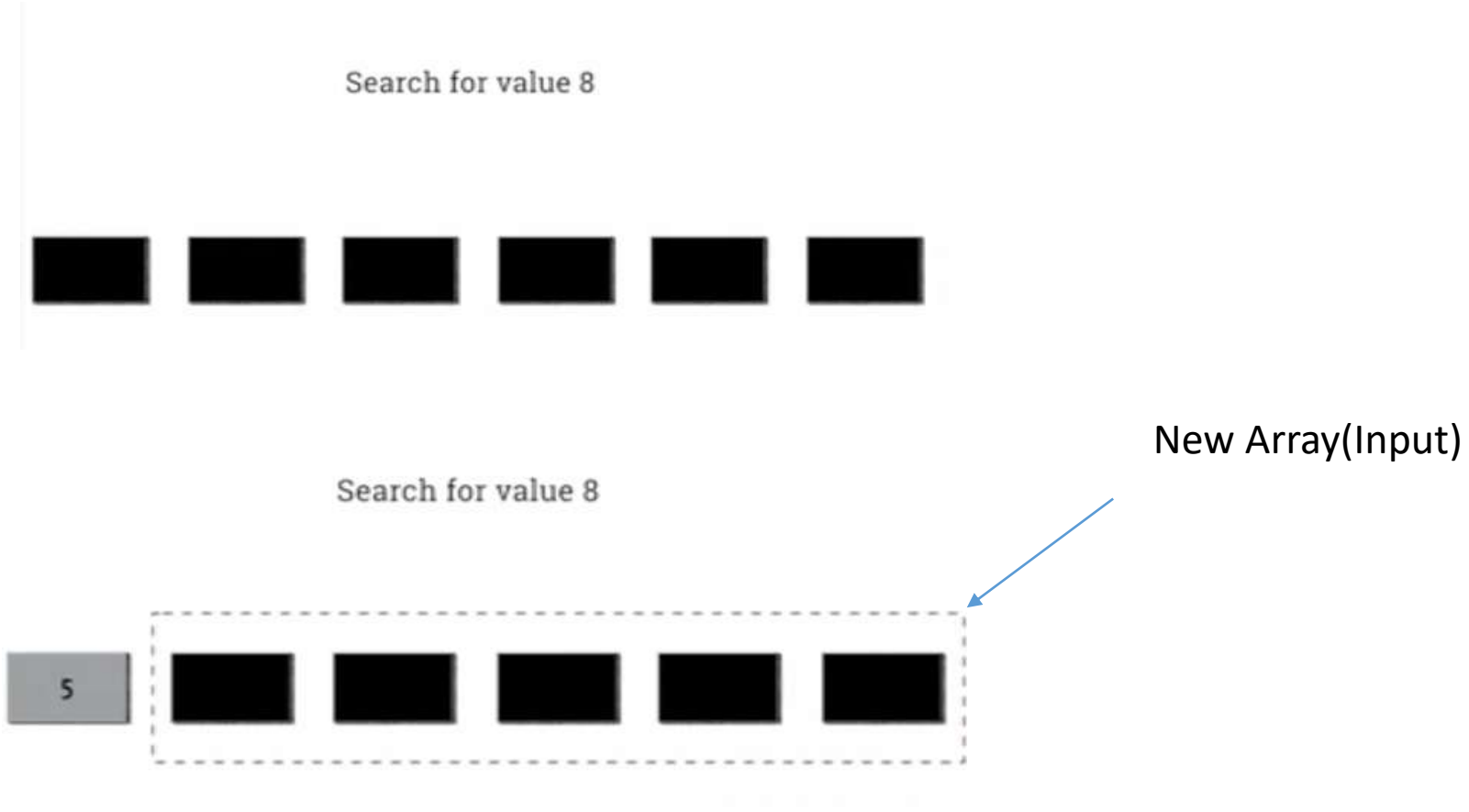
Pseudocode of Euclidean Algorithm

```
function GreatestCommonDivisor(a, b)
    while a ≠ b do
        if a > b then
            a ← a − b
        else
            b ← b − a
        end if
    end while
    return a
end function
```

Recursive Pseudocode of Euclidean Algorithm

```
function GreatestCommonDivisor(a, b)
    if a = b then                                   <-- Base Case
        return a
    else if a > b then
        return GreatestCommonDivisor(a − b, b)
    else
        return GreatestCommonDivisor(a, b − a)
    end if
end function
```

Recursive Linear Search



Recursive Linear Search

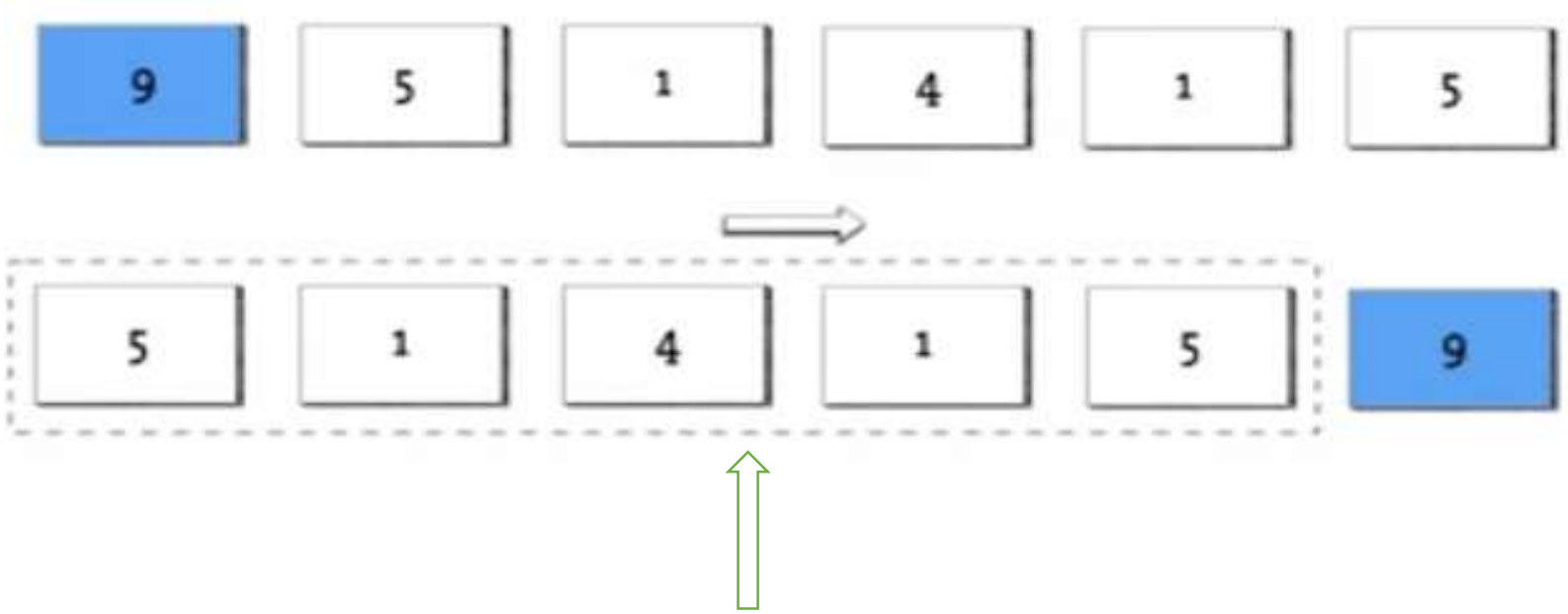
l is just going to tell us where the first element of the vector is that we care about.

```
function Search(v, l, item)
    n ← LENGTH[v]
    if l > n then
        return FALSE
    else if v[l] = item then
        return TRUE
    end if
    return Search(v, l + 1, item)
end function

function LinearSearch(v, item)
    return Search(v, 1, item)
end function
```

Recursive Bubble Sort

Q: How to reduce the size of the input to the problem using recursion?



Recursive Bubble Sort: Swap function

function Swap($vector, i, j$)

$x \leftarrow vector[j]$

$vector[j] \leftarrow vector[i]$

$vector[i] \leftarrow x$

return $vector$

end function

Recursive Bubble Sort: Pseudocode

```
function Sort(vector, r)
    if r ≤ 1 then
        return vector
    end if
    for 1 ≤ j ≤ r − 1 do
        if vector[j + 1] < vector[j] then
            Swap(vector, j, j + 1)
        end if
    end for
    Sort(vector, r − 1)
    return vector
end function
function BubbleSort(vector)
    n ← LENGTH[vector]
    return Sort(vector, n)
end function
```

Base Case

Recursive Insertion Sort: Shift Function

```
function Shift(vector, i, j)
    if i ≤ j then
        return vector
    end if
    store ← vector[i]
    for 0 ≤ k ≤ (i − j − 1) do
        vector[i − k] ← vector[i − k − 1]
    end for
    vector[j] ← store
    return vector
end function
```

Recursive Insertion Sort: Pseudocode

- Mixture of recursion and iteration to give a simple implementation.

```
function Sort(vector, r)
    if r ≤ 1 then
        return vector
    end if
    Sort(vector, r − 1)
    j ← r    i ← r
    while (vector[i] < vector[j − 1]) ∧ (j > 1) do
        j ← j − 1
    end while
    Shift(vector, i, j)
    return vector
end function
function InsertionSort(vector)
    n ← LENGTH[vector]
    return Sort(vector, n)
end function
```

Iterative Binary Search: Pseudocode

```
function BinarySearch(v, item)
    n ← LENGTH[v]
    R ← n
    L ← 1
    while R ≥ L do
        M ← ⌊(L + R)/2⌋
        if v[M] = item then
            return TRUE
        else if v[M] > item then
            R ← M − 1
        else
            L ← M + 1
        end if
    end while
    return FALSE
end function
```

Recursive Binary Search: Pseudocode

```
function Search(v, item, L, R)
        if L > R then
                return FALSE
        end if
        M ← ⌊(L + R)/2⌋
        if v[M] = item then
                return TRUE
        else if v[M] > item then
                R ← M − 1
        else
                L ← M + 1
        end if
        return Search(v, item, L, R)
end function

function BinarySearch(v, item)
        n ← LENGTH[v]
        R ← n
        L ← 1
        return Search(v, item, L, R)
end function
```

Analysis of Algorithms

Analysis of Algorithms

- Analysis:
- Predict the cost of an algorithm in terms of resources and performance
- Analysis enables us to select best algorithms.
- Which algorithm is best to use for specific task for example there are multiple sorting algorithms.

Aspects of algorithm analysis

- Correctness:
  - Performing specific task according to the given specifications
  - Does the input/output relation match algorithm requirement?
- Ease of Understanding:
  - Easy algorithms are easy to update and maintain accordingly.
- Resource Consumption:
  - Processor
  - Memory
  - Algorithms with less consumption are better than the others.

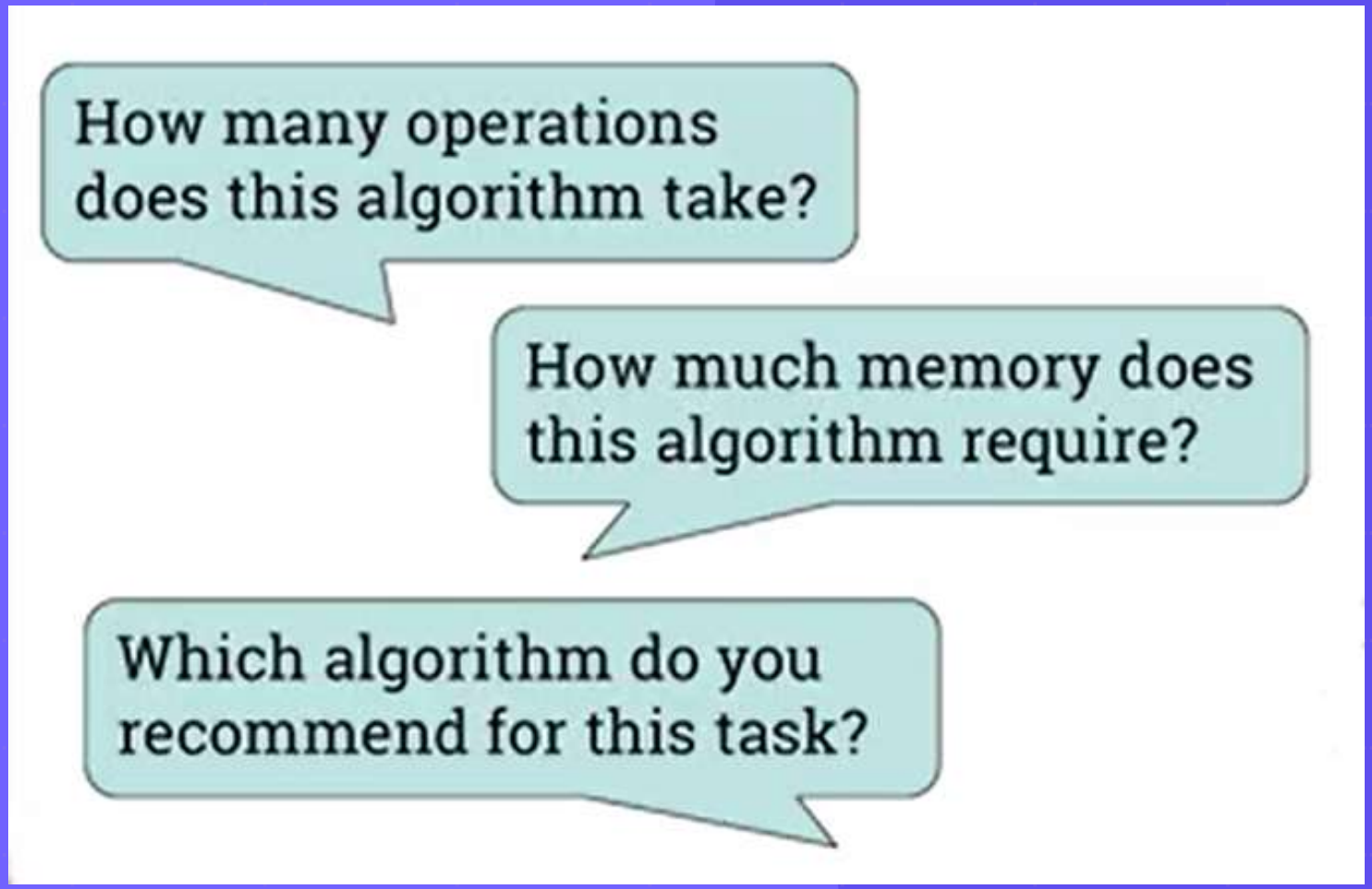
Processing:

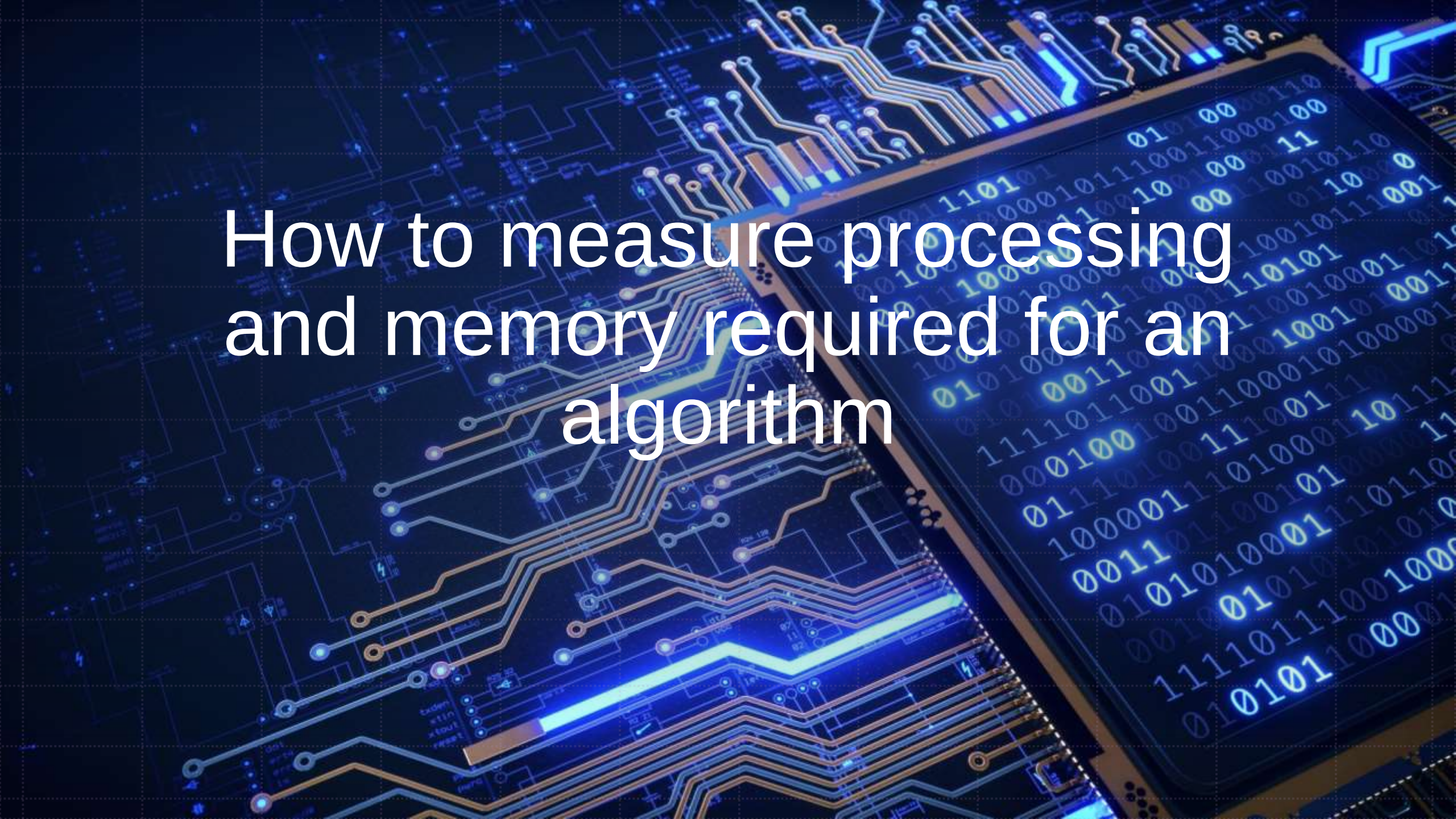
- Processing requirements are usually measured in number of operations a processor performs.
- No of operations are very important.
- Algorithms with less number of operations are faster.

Memory

- Memory requirement is measured in amount of memory required to execute the algorithm.
- Memory is very important as per the system.

Questions to answer:




How to measure processing and memory required for an algorithm

How much time and memory it will consume?

- Question: What Procedure you would follow to determine the time and space?
- Ans: ??

```
function F(array s)
    for 1 ≤ j < length(s)  do
        key ← s[j]
        i ← j-1
        while i ≥ 0 ∧ s[i] > key  do
            s[i+1] ← s[i]
            i ← i - 1
        end while
        s[i+1] ← key
    end for
end function
```

Approaches to analyze an algorithm

Empirical

Theoretical

Empirical approach (estimation):

1

Implementing the algorithm in specific language

2

Executing the algorithm on a specific machine

3

Measuring the time and space of the algorithm

Empirical Analysis

- In practice, we will often need to resort to empirical rather than theoretical analysis to compare algorithms.
  - We may want to know something about performance of the algorithm "on average" for real instances.
  - Our model of computation may not capture important effects of the hardware architecture that arise in practice.
  - There may be implementational details that affect constant factors and are not captured by asymptotic analysis.
- For this purpose, we need a methodology for comparing algorithms based on real-world performance.

Theoretical Approach (estimation):

Making some assumptions about number of CPU operations for every instructions of an algorithm.

Multiply by time required for each operation.

For Memory: Calculate the memory used by the variables used during execution.

Advantages of Empirical Approach

- Real/Exact Results
- No need for calculations.

Drawbacks/disadvantages

- Machine dependent results
- Implementation effort

Issues to consider

- Empirical analysis introduces many more factors that need to be controlled for in some way.
  - Test platform (hardware, language, compiler)
  - Measures of performance (what to compare)
  - Benchmark test set (what instances to test on)
  - Algorithmic parameters
  - Implementational details
- It is much less obvious how to perform a rigorous analysis in the presence of so many factors.
- Practical considerations prevent complete testing.

Theoretical approach:

ADVANTAGES	DRAWBACKS
Universal results	Approximate results
No implementation effort	Calculations effort

Theoretical Estimation:

Theoretical approach

- Learn about the machine model 
- Know the assumptions 
- Do the calculations