Stacks

What is Data Structure :
Data Structure is a way to store and organize data so that it can be used efficiently.

As per name indicates, it organizes the data in memory.

The data structure is not any programming language like c, c++, Java, etc. It is a set of algorithms that we can use in any programming language to structure data in memory.

1) Linear Data structure:-

The arrangement of data in the sequential manner is known as linear data structure The data structure used for this purpose are Arrays, linked list, Stacks and queues.

In these data structures, one element is connected to only one another element in a linear form.

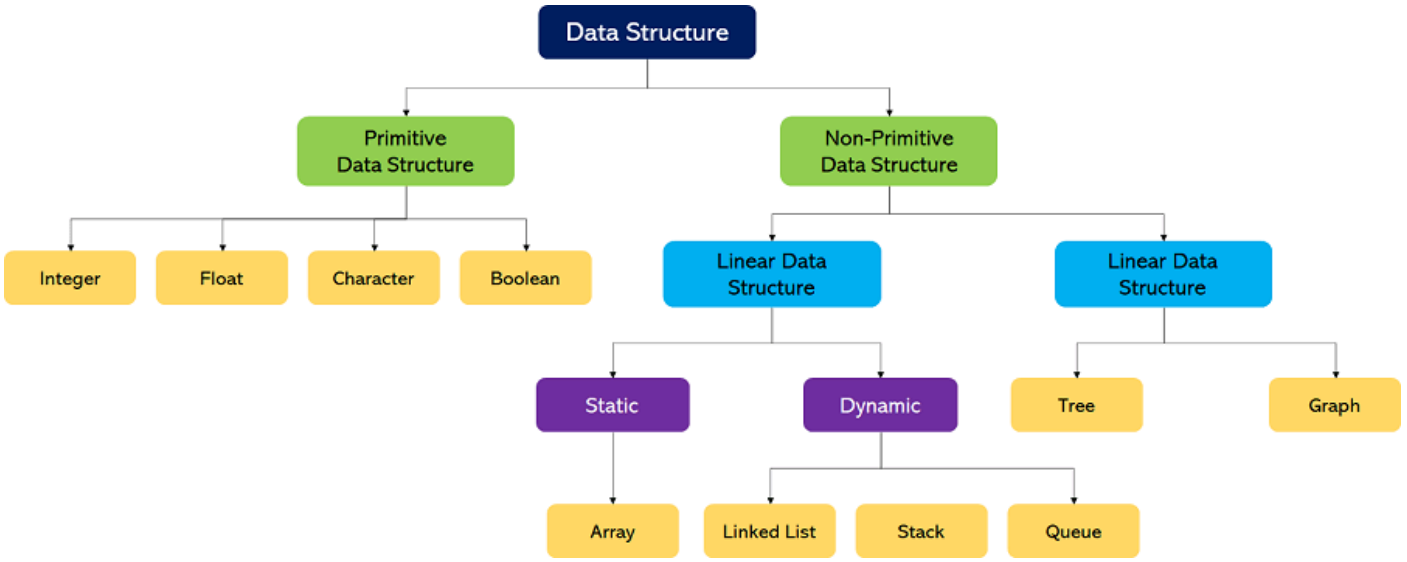
2) Non-Linear data structure:-

When one element is connected to the 'n' number of elements known as Non-Linear data structures. Example trees and graphs.

To structure the data in memory, 'n' number of algorithms are proposed, and all these algorithms are known as Abstract Data Types.

An Abstract Data Type tells what is to be done. and data structure tells how to be done?

ADT gives us the blueprint while data structure provides the implementation part.



What is need of data structures:

As applications are getting complexed and amount of data is increasing day by day, there may arise following problems :-

Processor speed :- As data is growing day by day to the billions of files per entity, processors may fail to deal with enormous amounts of data.

Data Structure :- consider an inventory size of 100 items in store, if our application needs to search for a particular item, it needs to traverse 100 items every time, it results in slowing down process

Multiple requests :- If thousands of users are searching data simultaneously on a web server, then there are chances to be failed to search during that process.

To solve these problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and the required data can be searched instantly.

You can cover the basics of stack from the another Notes given in classroom under Unit 1: Stack
DSA_Module 1_Stack and its Applications (1)

Cover the Application of stack and other side topics from this file

Discuss how stacks can be used for checking the balancing of symbols.
Solution: Stacks can be used to check whether the given expression has balanced symbols. This algorithm is very useful in compilers. Each time the parser reads one character at a time. If the character is an opening delimiter such as (, {, or [- then it is written to the stack. When a closing delimiter is encountered like), }, or]- the stack is popped. The opening and closing delimiters are then compared. If they match, the parsing of the string continues. If they do not match, the parser indicates that there is an error on the line. A linear-time O(n) algorithm based on stack can be given as:

Algorithm
a) Create a stack.
b) while (end of input is not reached)
1) If the character read is not a symbol to be balanced, ignore it.
2) If the character is an opening symbol like (, [, {, push it onto the stack
3) If it is a closing symbol like),],}, then if the stack is empty report an error. Otherwise pop the stack.
4) If the symbol popped is not the corresponding opening symbol, report an Error.
c) At end of input, if the stack is not empty report an error

Input Symbol, A[i]	Operation	Stack	Output
(	Push (	(	
)	Pop (Test if (and A[i] match? YES		
(	Push (	(	
(	Push (	((	
)	Pop (Test if(and A[i] match? YES	(	
[	Push [	([	
(	Push (	([(	
)	Pop (Test if(and A[i] match? YES	([	
]	Pop [Test if [and A[i] match? YES	(	
)	Pop (Test if(and A[i] match? YES		
	Test if stack is Empty? YES		TRUE

Discuss infix to postfix conversion algorithm using stack.
Before discussing the algorithm, first let us see the definitions of infix, prefix and postfix expressions.

Infix: An infix expression is a single letter, or an operator, proceeded by one infix string and followed by another Infix string.

Operand1 Operator Operand 2

A
A+B
(A+B)+ (C-D)

Prefix: A prefix expression is a single letter, or an operator, followed by two prefix strings. Every prefix string longer than a single variable contains an operator, first operand and second operand.
Operator Operand1 Operand 2

A
+AB
++AB-CD

Postfix: A postfix expression (also called Reverse Polish Notation) is a single letter or an operator, preceded by two postfix strings. Every postfix string longer than a single variable contains first and second operands followed by an operator.

Operand1 Operand 2 Operator

A
AB+
AB+CD-+

Prefix and postfix notions are methods of writing mathematical expressions without parentheses. Time to evaluate a postfix and prefix expression is O(n), were n is the number of elements in the array.

Infix	Prefix	Postfix
A+B	+AB	AB+
A+B-C	-+ABC	AB+C-
(A+B)*C-D	-*+ABCD	AB+C*D-

Now, let us focus on the algorithm. In infix expressions, the operator precedence is implicit unless we use parentheses. Therefore, for the infix to postfix conversion algorithm we have to define the operator precedence (or priority) inside the algorithm. The table shows the precedence and their associativity (order of evaluation) among operators.

Important Properties

- Let us consider the infix expression 2 + 3 * 4 and its postfix equivalent 2 3 4 *+. Notice that between infix and postfix the order of the numbers (or operands) is unchanged. It is 2 3 4 in both cases. But the order of the operators * and + is affected in the two expressions.
- Only one stack is enough to convert an infix expression to a postfix expression. The stack that we use in the algorithm will be used to change the order of operators from infix to postfix. The stack we use will only contain operators and the open parenthesis symbol ‘(‘. Postfix expressions do not contain parentheses. We shall not output the parentheses in the postfix output.

Operators	Symbols
Parenthesis	( ), {}, [ ]
Exponents(R->L)	^
Multiplication and Division(L->R)	*, /
Addition and Subtraction(L->R)	+ , -

Algorithm for conversion from infix to postfix expression

1. Create an empty stack
2. for each character t in the input stream repeat steps 3 to 9.
3. Print the operand as they arrive.

4. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
5. If the incoming symbol is '(', push it onto the stack.
6. If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
7. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
8. If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then test the incoming operator against the new top of the stack.
9. If the incoming operator has the same precedence with the top of the stack then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming operator. If the associativity is from right to left then push the incoming operator.
10. At the end of the expression, pop and print all the operators of the stack.

For better understanding let us trace out an example: A * B- (C + D) + E

Input Character	Operation on Stack	Stack	Postfix Expression
A		Empty	A
*	Push	*	A
B		*	AB
-	Check and Push	-	AB*
(	Push	-(	AB*
C		-(	AB*C
+	Check and Push	-(+	AB*C
D			AB*CD
)	Pop and append to postfix till '('	-	AB*CD+
+	Check and Push	+	AB*CD+-
E		+	AB*CD+-E
End of input	Pop till empty		AB*CD+-E+

Practice: Expression
((A + B) – C * (D / E)) + F : AB+CDE/*-F+
a+b*(c^d-e)^(f+g*h)-i : abcd^e-fgh*+^*+i-

Discuss Arithmetic Expression Evaluation using stacks?
The stack organization is very effective in evaluating arithmetic expressions. Expressions are usually represented in what is known as Infix notation, in which each operator is written between two operands (i.e., A + B). With this notation, we must distinguish between (A + B)*C and A + (B * C) by using either parentheses or some operator-precedence convention. Thus, the order of operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.
1. Polish notation (prefix notation) –
It refers to the notation in which the operator is placed before its two operands. Here no parentheses are required, i.e.,
+AB
2. Reverse Polish notation(postfix notation) –
It refers to the analogous notation in which the operator is placed after its two operands. Again, no parentheses is required in Reverse Polish notation, i.e.,
AB+

Stack-organized computers are better suited for post-fix notation than the traditional infix notation. Thus, the infix notation must be converted to the postfix notation. The conversion from infix notation to postfix notation must take into consideration the operational hierarchy.

Algorithm:

1. Convert the expression in Reverse Polish notation(post-fix notation).
2. Scan the Postfix string from left to right.
3. Initialize an empty stack.
4. Repeat steps 4 and 5 till all the characters are scanned.
5. If the scanned character is an operand, push it onto the stack.
6. If the scanned character is an operator, and if the operator is a unary operator, then pop an element from the stack.
   If the operator is a binary operator, then pop two elements from the stack. After popping the elements, apply the operator to those popped elements.
   Let the result of this operation be retVal onto the stack.
6. After all characters are scanned, we will have only one element in the stack.
7. Return top of the stack as result.

E.g. 123*+5-

Given an array of characters formed with a's and b's. The string is marked with special character X which represents the middle of the list (for example: ababa...ababXbabab…..baaa). Check whether the string is palindrome.

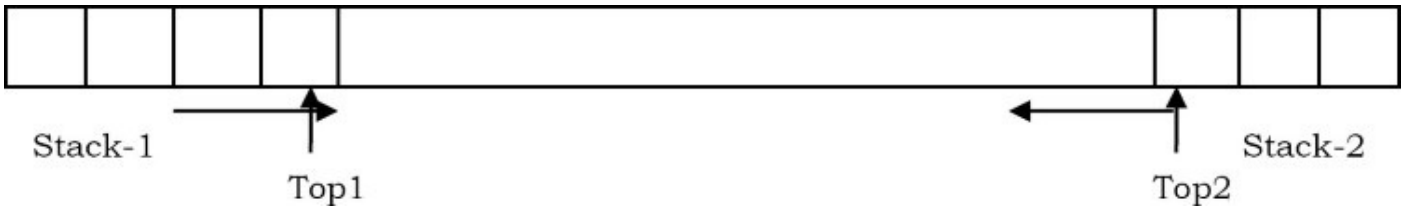
Solution:

This is one of the simplest algorithms. What we do is, start two indexes, one at the beginning of the string and the other at the end of the string. Each time compare whether the values at both the indexes are the same or not. If the values are not the same then we say that the given string is not a palindrome.

If the values are the same then increment the left index and decrement the right index. Continue this process until both the indexes meet at the middle (at X) or if the string is not palindrome.

How do we implement two stacks using only one array? Our stack routines should not indicate an exception unless every slot in the array is used?

Solution:



Algorithm:

- Start two indexes one at the left end and the other at the right end.
- The left index simulates the first stack and the right index simulates the second stack.
- If we want to push an element into the first stack then put the element at the left index.
- Similarly, if we want to push an element into the second stack then put the element at the right index.

- The first stack grows towards the right, and the second stack grows towards the left.
- Stack isFull() condition for both stacks should be top should not cross each other. top1<top2.

Factorial Calculation:

What is Recursion?

Recursion, in the realm of computer science, is a method of solving problems where the solution depends on solutions to smaller instances of the same problem. This involves a function calling itself while a certain condition is true. The process continues until a condition (known as the base case) is met.

The Call Stack and Recursion

To understand recursion, it's necessary to understand the concept of the call stack. The call stack is a data structure that stores information about the active subroutines in a program. In the case of recursion, every time a recursive call is made, the current computation is pushed onto the call stack. When the base case is reached, the computations are popped from the stack and resolved in a last-in, first-out (LIFO) order.

The Base Case in Recursion

The base case in recursion is a condition that determines when the recursive calls should stop. It's crucial to define a reachable base case to prevent infinite recursion.
In our factorial example, the base case is
n == 0, which returns 1 and stops the recursion.

```
    unsigned int factorial(unsigned int n) {
        if (n == 0) {
                return 1;
        }
        else {
                return n * factorial(n - 1);
        }
}
```

While recursion can be a powerful tool, it can also lead to some common issues if not handled correctly. Let's discuss these issues and how to avoid them.

Stack Overflow Errors

One of the most common issues with recursion is the risk of a stack overflow error. This happens when the depth of recursion is too high, exceeding the limit of the call stack.

Avoiding Stack Overflow Errors

To avoid stack overflow errors, you can often refactor your recursive method to an iterative one, as we discussed in the 'Alternative Approaches' section. Alternatively, you can use techniques like tail recursion optimization, which Java unfortunately does not support natively but can be achieved with some workarounds.

Best Practices for Recursive Methods

When writing recursive methods in Java, there are a few best practices to keep in mind:

- Define a clear base case: The base case is what stops the recursion. Without a clear and reachable base case, your recursive method might run indefinitely.
- Ensure progress towards the base case: Each recursive call should progress towards the base case. If the recursive calls do not progress towards the base case, you might end up with infinite recursion.
- Be mindful of the call stack limit: Deep recursion can lead to stack overflow errors. If your problem requires deep recursion, consider using an iterative solution instead or implementing tail recursion optimization.

When and Why to Use Recursion

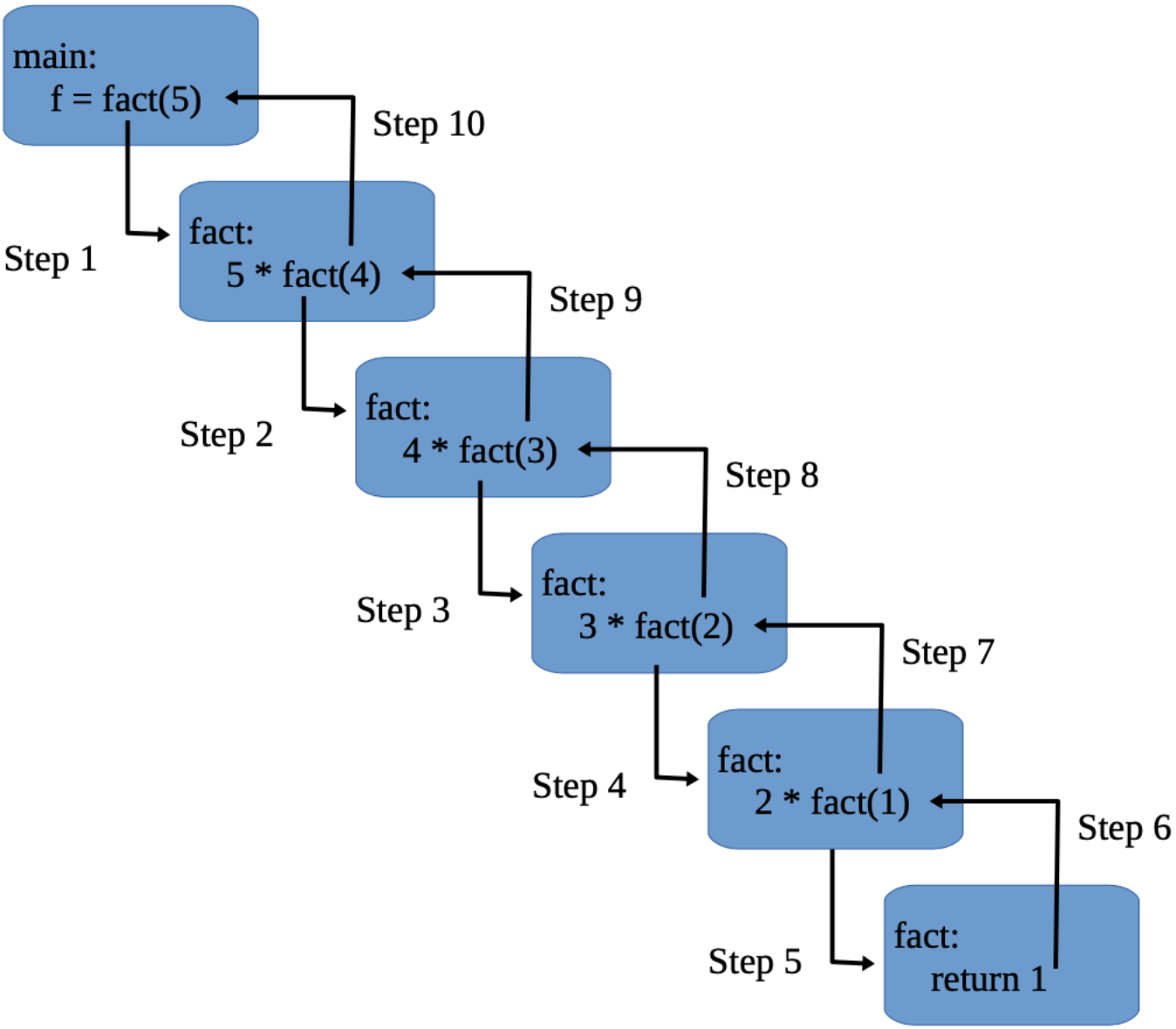
Recursion is a powerful tool in a programmer's toolkit. It's particularly useful when you need to solve problems that can be broken down into similar sub-problems. Recursive algorithms are often simpler and more intuitive to understand than their iterative counterparts.
However, recursion should be used judiciously. It can lead to problems such as stack overflow errors if the depth of recursion is too high. It can also be less efficient than iterative solutions for large inputs due to the overhead of function calls.
Understanding when and why to use recursion, as well as being aware of its potential pitfalls, is an important part of mastering recursion.

```
#include <stdio.h>

// Recursive Function to calculate Factorial of a number
int factorial(int n)
{
   // Base case
   if (n == 0) {
      return 1;
   }


   // Recursive case
   return n * factorial(n - 1);
}


int main()
{
        int n = 5;
printf("Factorial of  %d  is:  %d", n,  factorial(n));
        return 0;
}
```



Given a stack, how to reverse the contents of the stack using only stack operations (push and pop)?
Solution:
Algorithm 1: Follow the steps below to solve the problem:

- Initialize an empty stack called ans.
- Pop the top element of the given stack S and push it into the stack ans.
- Return the stack ans, which will contain the reversed elements of the stack S.

Algorithm 2:

- First pop all the elements of the stack till it becomes empty.
- For each upward step in recursion, insert the element at the bottom of the stack

Follow the steps mentioned below to implement the idea:

- Create a stack and push all the elements in it.

- Call reverse(), which will pop all the elements from the stack and pass the popped element to function putBottom()
- Whenever putBottom() is called it will insert the passed element at the bottom of the stack.
- Print the stack.

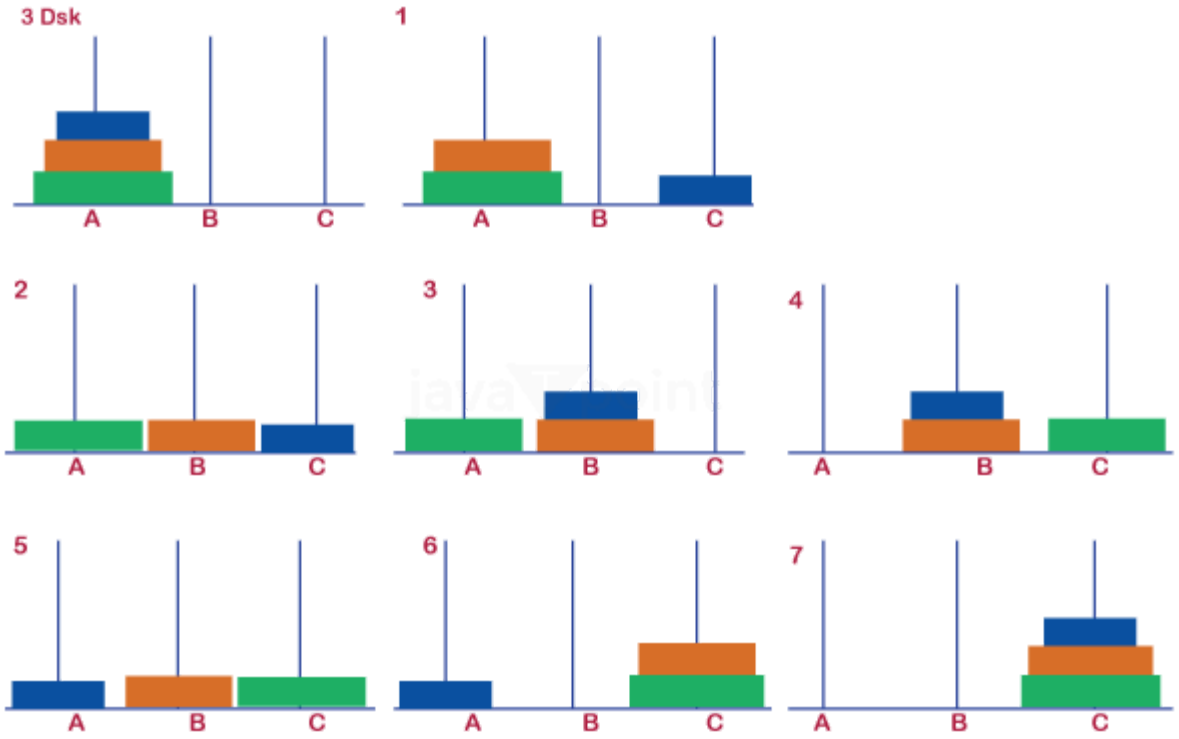
Discuss Towers of Hanoi puzzle.
Solution: The Towers of Hanoi is a mathematical puzzle. It consists of three rods (or pegs or towers) and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks on one rod in ascending order of size, the smallest at the top, thus making a conical shape.
The objective of the puzzle is to move the entire stack to another rod, satisfying the following rules:

1. Only one disk may be moved at a time.
2. Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
3. No disk may be placed on top of a smaller disk.

Algorithm

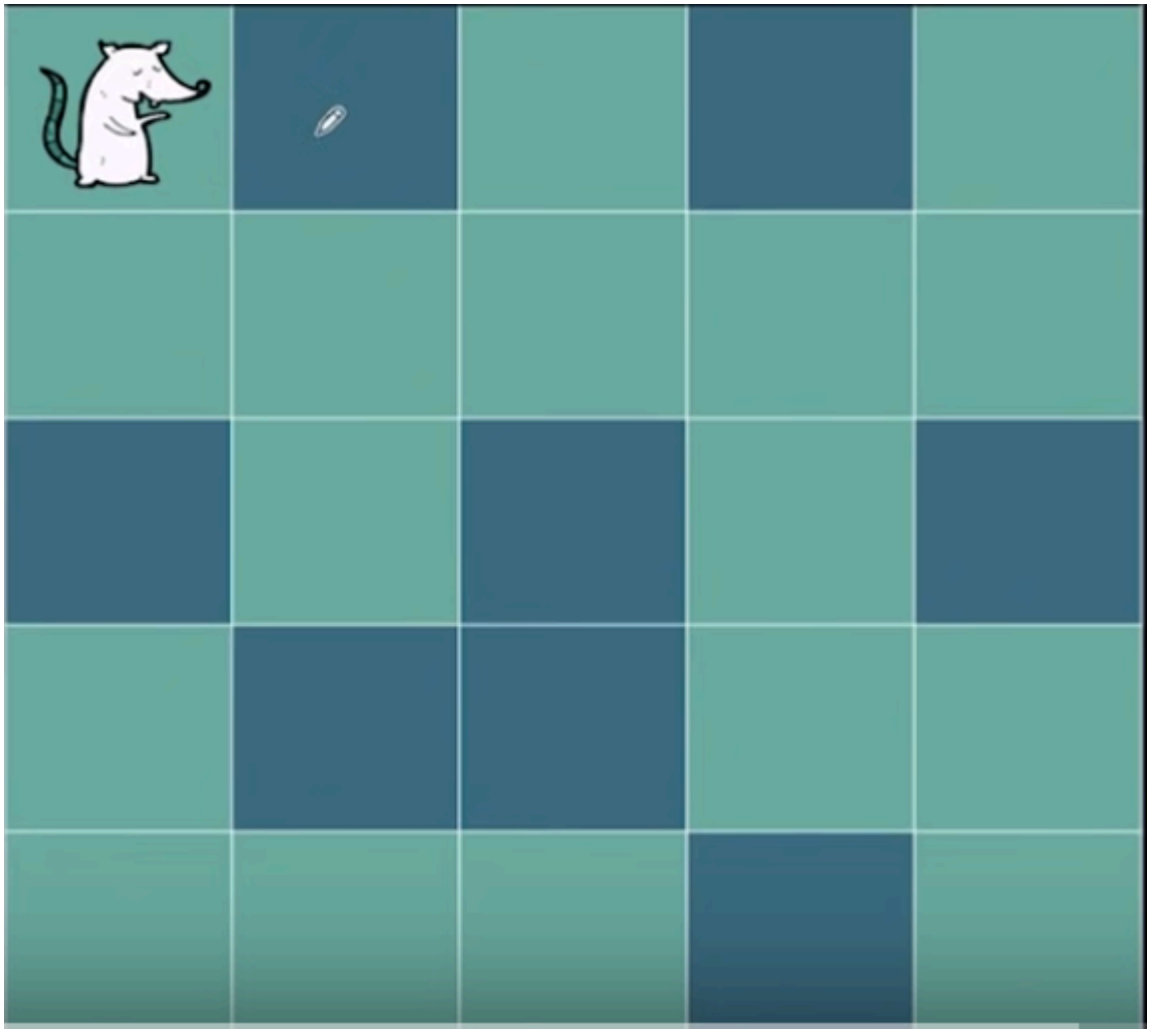
1. Move the top n – 1 disks from Source to Helper tower,
2. Move the nth disk from Source to Destination tower,
3. Move the n – 1 disks from Helper tower to Destination tower.
4. Transferring the top n – 1 disks from Source to Helper tower can again be thought of as a fresh problem and can be solved in the same manner.

Once we solve Towers of Hanoi with three disks, we can solve it with any number of disks with the above algorithm.



To visualize Tower of Hanoi problem solution you can visit website.

Backtracking:
Backtracking is an algorithmic-technique for solving recursive problems by trying to build every possible solution incrementally and removing those solutions that fail to satisfy the constraints of the problem at any point of time.



Algorithms Basics

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain sequence to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Following are some important categories of algorithms −
Search − Algorithm to search an item in a data structure.
Sort − Algorithm to sort items in a certain order.
Insert − Algorithm to insert item in a data structure.
Update − Algorithm to update an existing item in a data structure.
Delete − Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

An algorithm should have the following characteristics−
Unambiguous − Algorithms should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
Input − An algorithm should have 0 or more well-defined inputs.
Output − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
Finiteness − Algorithms must terminate after a finite number of steps.
Feasibility − Should be feasible with the available resources.
Independent − An algorithm should have step-by-step directions, which should be independent of any programming code.

Algorithm Complexity

Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.
Time Factor − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
Space Factor − Space is measured by counting the maximum memory space required by the algorithm.
The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

1. Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm

in its life cycle. The space required by an algorithm is equal to the sum of the following two components −A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.
Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −
Algorithm: SUM(A, B)
Step 1 − START
Step 2 − C ← A + B + 10
Step 3 − Stop
Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

2. Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.
For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

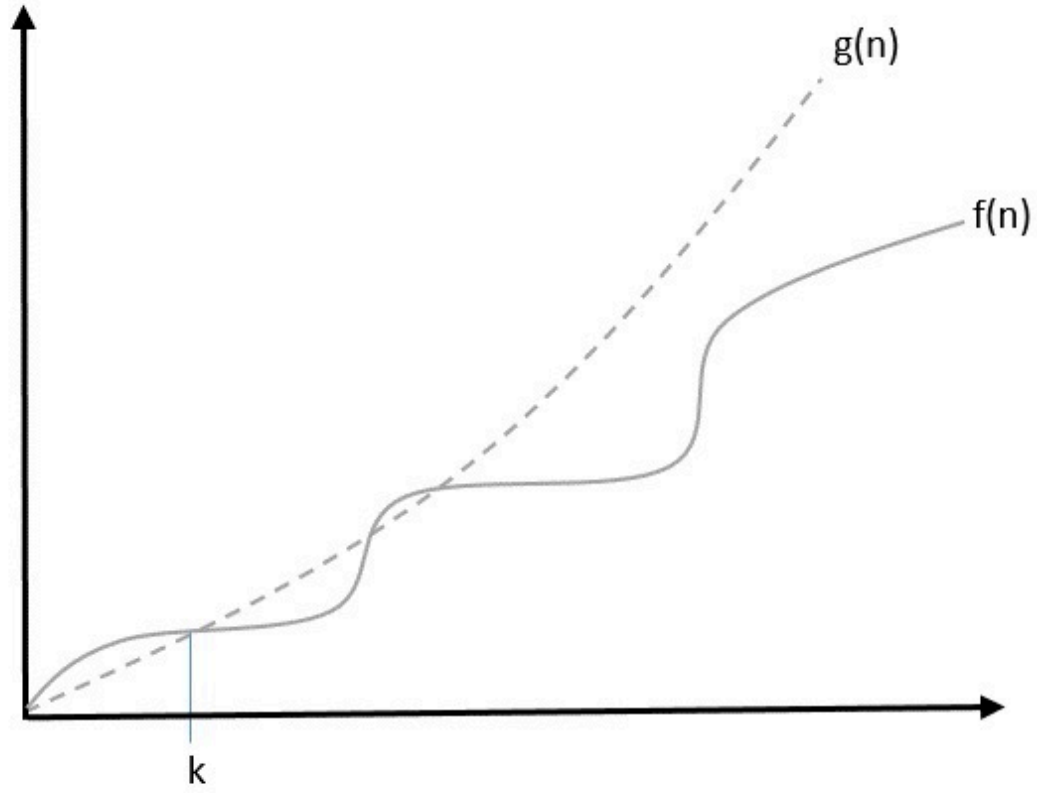
Asymptotic Analysis

Asymptotic analysis of an algorithm refers to defining the mathematical foundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.
Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.
Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as f(n) and may be for another operation it is computed as g(n2). This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is significantly small.
Usually, the time required by an algorithm falls under three types −
Best Case − Minimum time required for program execution.
Average Case − Average time required for program execution.
Worst Case − Maximum time required for program execution.

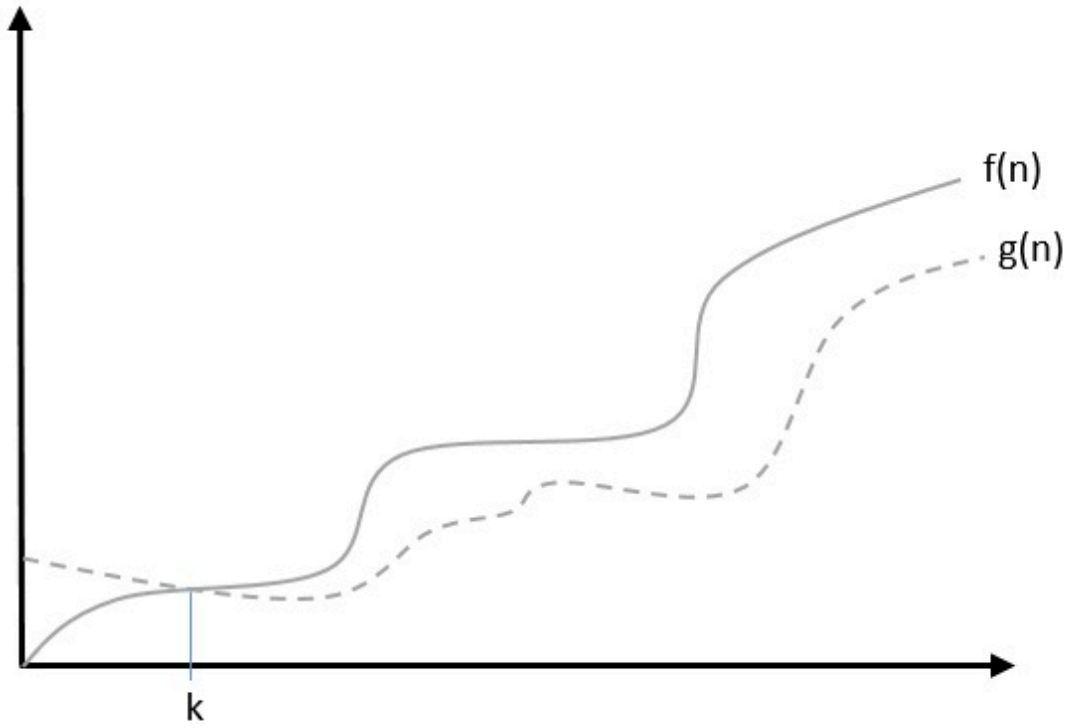
Asymptotic Notations
Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.
Time function of an algorithm is represented by T(n), where n is the input size.
Different types of asymptotic notations are used to represent the complexity of an algorithm.
Following asymptotic notations are used to calculate the running time complexity of an algorithm.
O − Big Oh Notation
Ω − Big omega Notation
θ − Big theta Notation

Big Oh, O: Asymptotic Upper Bound

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. is the most commonly used notation. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.
A function f(n) can be represented is the order of g(n) that is O(g(n)), if there exists a value of positive integer n as n0 and a positive constant c such that −
f(n) ⩽ c. g(n) for n > n0 in all case
Hence, function g(n) is an upper bound for function f(n), as g(n) grows faster than f(n).



Big Omega, Ω: Asymptotic Lower Bound
The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.
We say that f(n) = Ω(g(n)) when there exists constant c that f(n) ⩾ c. g(n) for all sufficiently large value of n. Here n is a positive integer. It means function g is a lower bound for function f ; after a certain value of n, f will never go below g.

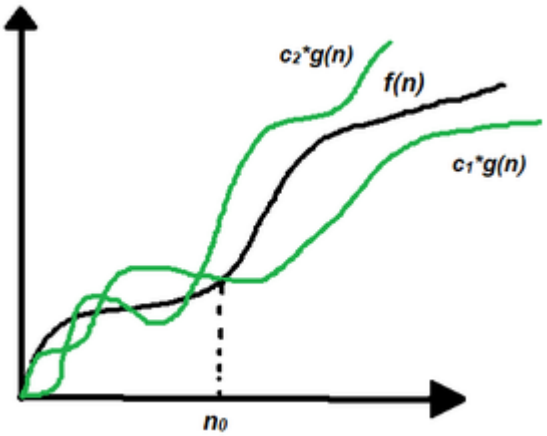


Theta Notation (Θ-Notation):

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

.Theta (Average Case) You add the running times for each possible input combination and take the average in the average case.

Let g and f be the function from the set of natural numbers to itself. The function f is said to be Θ(g), if there are constants c1, c2 > 0 and a natural number n0 such that c1* g(n) ≤ f(n) ≤ c2 * g(n) for all n ≥ n0



Theta notation
Mathematical Representation of Theta notation:

Θ (g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 ≤ c1 * g(n) ≤ f(n) ≤ c2 * g(n) for all n ≥ n0}

Note: Θ(g) is a set

The above expression can be described as if f(n) is theta of g(n), then the value f(n) is always between c1 * g(n) and c2 * g(n) for large values of n (n ≥ n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.
The execution time serves as both a lower and upper bound on the algorithm's time complexity.
It exist as both, most, and least boundaries for a given input value.
A simple way to get the Theta notation of an expression is to drop low-order terms and ignore leading constants. For example, Consider the expression 3n3 + 6n2 + 6000 = Θ(n3), the dropping lower order terms is always fine because there will always be a number(n) after which Θ(n3) has higher values than Θ(n2) irrespective of the constants involved. For a given function g(n), we denote Θ(g(n)) is following set of functions.
Examples :

{ 100 , log (2000) , 10^4 } belongs to Θ(1)
{ (n/4) , (2n+3) , (n/100 + log(n)) } belongs to Θ(n)
{ (n^2+n) , (2n^2) , (n^2+log(n))} belongs to Θ(n2)

Note: Θ provides exact bounds.