

Data Structures & Algorithms

SoS End Term Report 2021



Name: Jayanth Dosapati

Roll No: 200260016

Mentor: Ritik Mandal

Preface:

I was assigned Data Structures and algorithms in Summer of Science 2020-21. After beginning this I realized this is as hard topic . I tried my level best to grasp as much as I can in this short period that too in this difficult situation and I did not find much time to implement the stuff so mainly I dealt with examples and definitions.

Acknowledgment:

I would like to thank Maths and Physics Club, IIT Bombay for this project. I thank my mentor Ritik Mandal for guiding me in this difficult project. I also thank my parents and sisters for cooperating me to complete this project in this difficult time of covid.

References:

I learned mostly from the stuff of www.geeksforgeeks.org, www.topcoder.com, and I have used recourses provided by www.codechef.com.

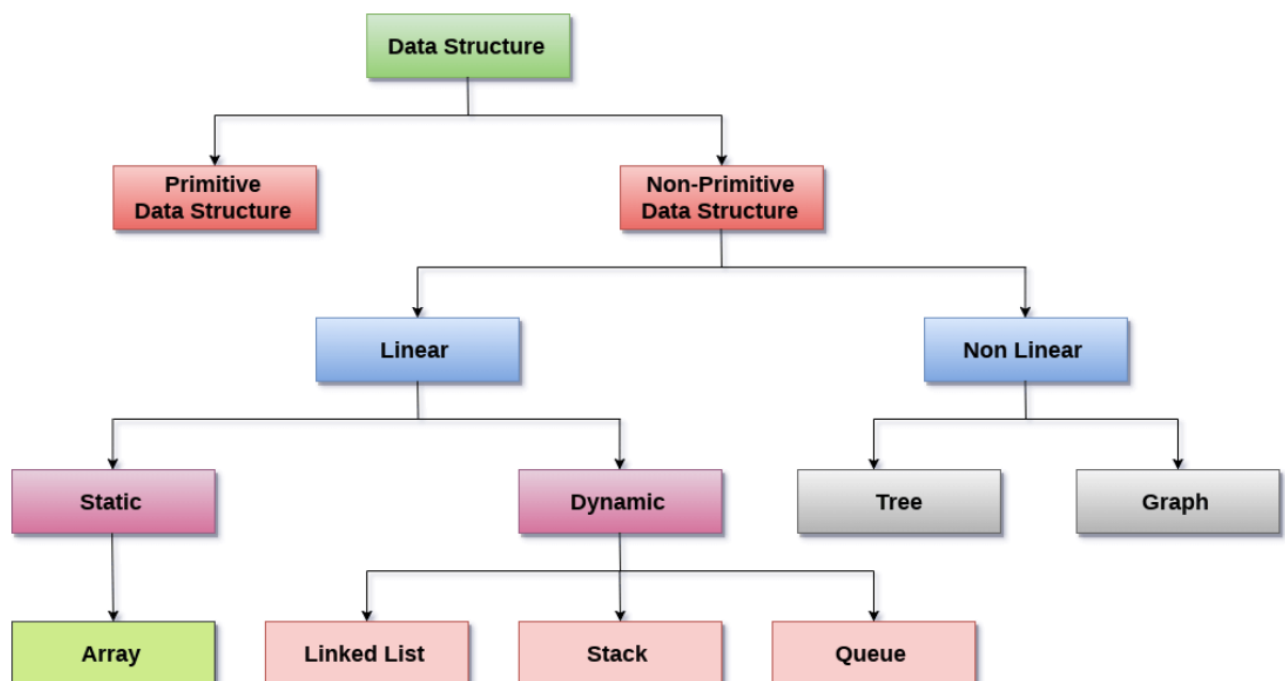
INTRODUCTION:

What is a Data Structure?

We know that doing a work in a systemic and organized way makes the work easier. Data Structure is a data organization management, and storage format that enables efficient Data structure is a data organization, management, and storage format that enables efficient access and modification. It is a collection of data values, the relationships among them and functions or operation that can be applied to data.

Example: Arrays, Linked list, Record, Union, Tagged Union.

Classification of Data Structures:



Advantages of Data Structures:

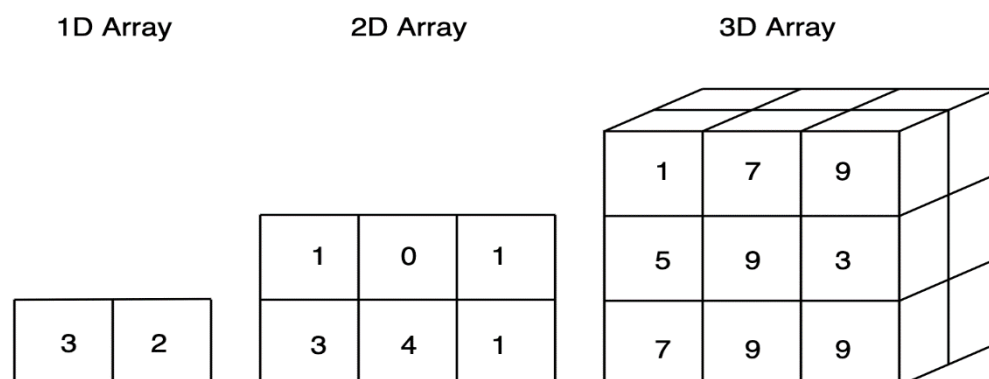
Efficiency

Reusability

Abstraction

Arrays:

An array is a Collection of similar data type of data items, and each item is called an element of the array. The data type can be int, float, double, Char etc. These can be 1 dimensional, 2 dimensional or many dimensional.



String:

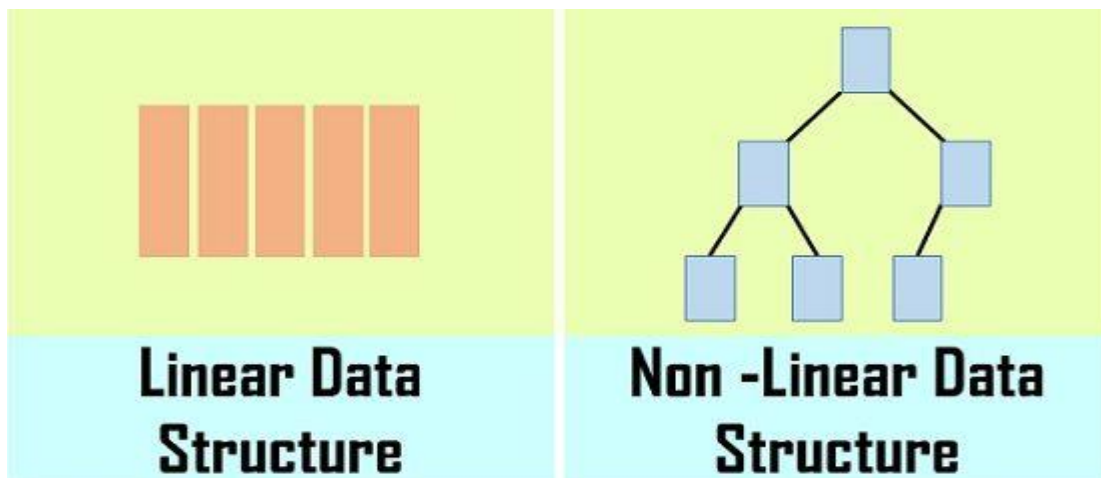
String is defined as an array of characters.

Difference between an array and string is string is terminated with null character “/0”.

Pointer:

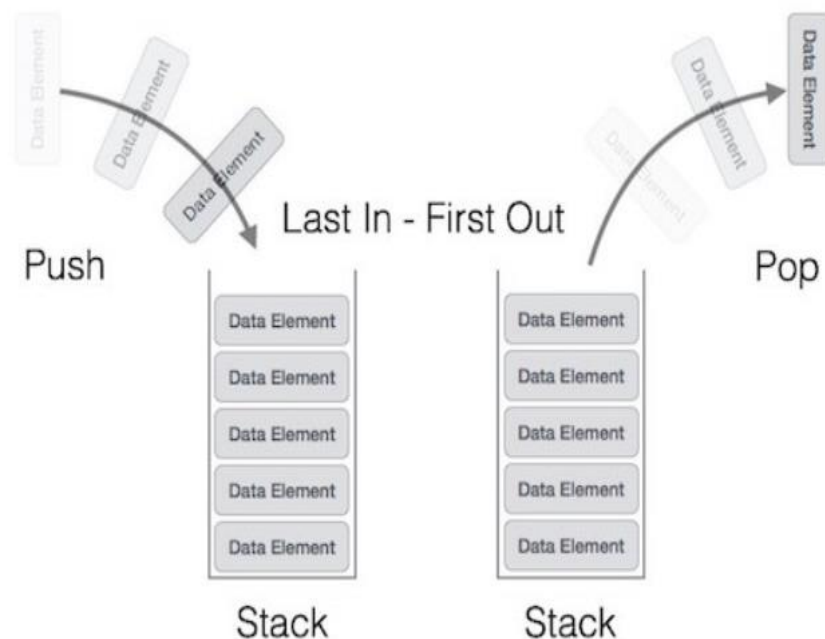
In C++, a pointer refers to a variable that holds the address of another variable.

Linear Data Structure:

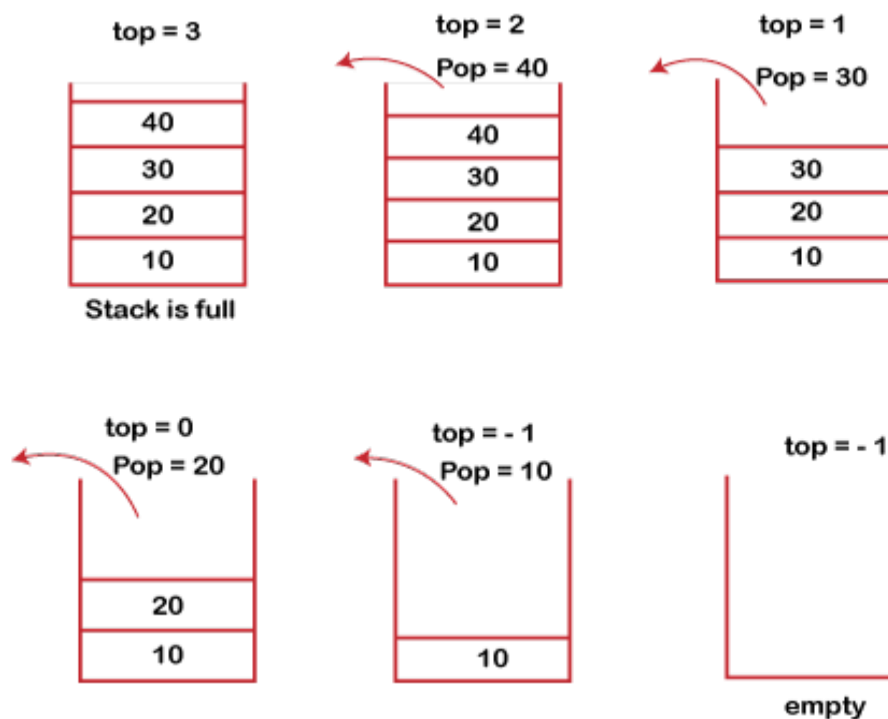


A Linear data structure have data elements arranged in sequential manner and each member element is connected to its previous and next element. Such data structures are easy to implement as computer memory is also sequential. Examples of linear data structures are List, Queue, Stack, Array etc.

Stack:



Stack is a linear data structure which follows a particular order in which operations are performed. The order is last in first out. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.



Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it.

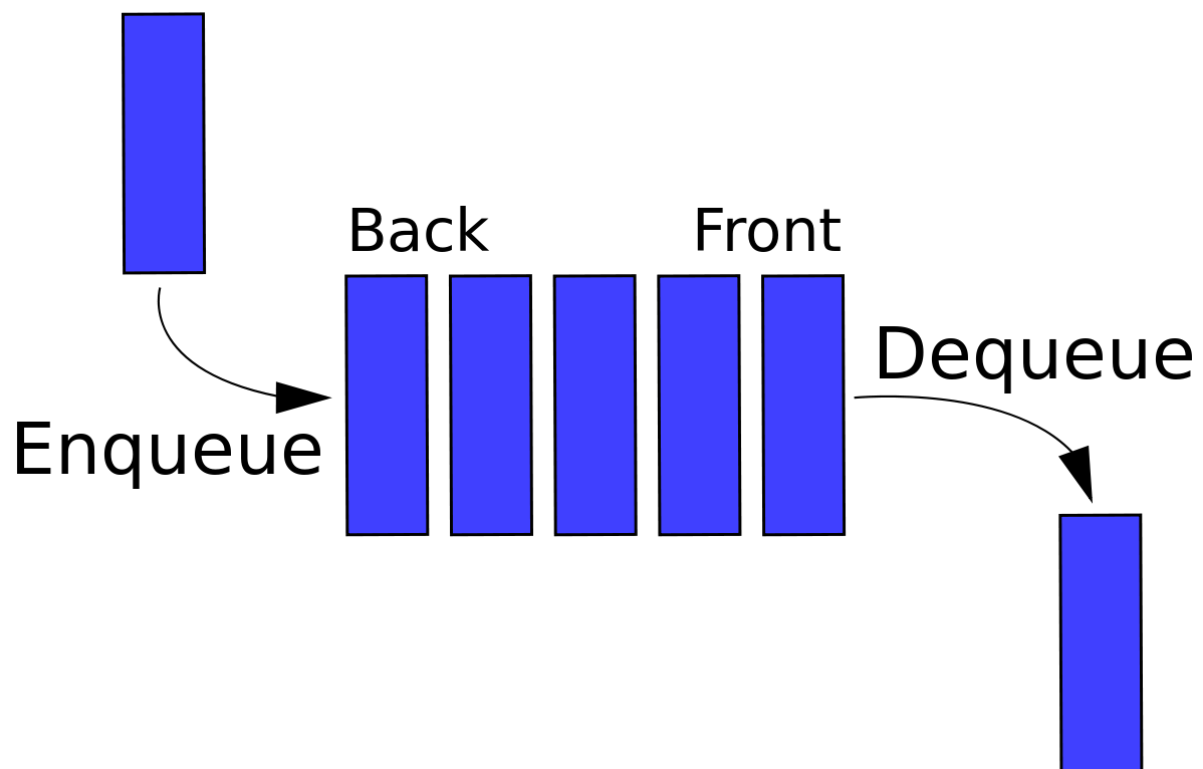
- **push()** – Pushing (storing) an element on the stack.

- **pop()** – Removing (accessing) an element from the stack.
- **isFull()** – check if stack is full.
- **isEmpty()** – check if stack is empty.

Queues:



Queue is a linear data structure which follows a particular order in which operations are performed. The order is first in first out.



Basic Operations

- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue.
- **peek()** – Gets the element at the front of the queue without removing it.
- **isfull()** – Checks if the queue is full.
- **isempty()** – Checks if the queue is empty.

Big 'O' Notation:

It is used in Computer science to describe the performance or complexity of an algorithm. Big O specifically describe the worst scenario and can be used to describe the execution time required or the space used by an algorithm. It is also called Landau's symbol. For example $f(x) = 8x^3 - 2x^2 + 3$. If we ignore the constant and slower growing terms, we could say that $f(x)$ grows at the order of x^3 .

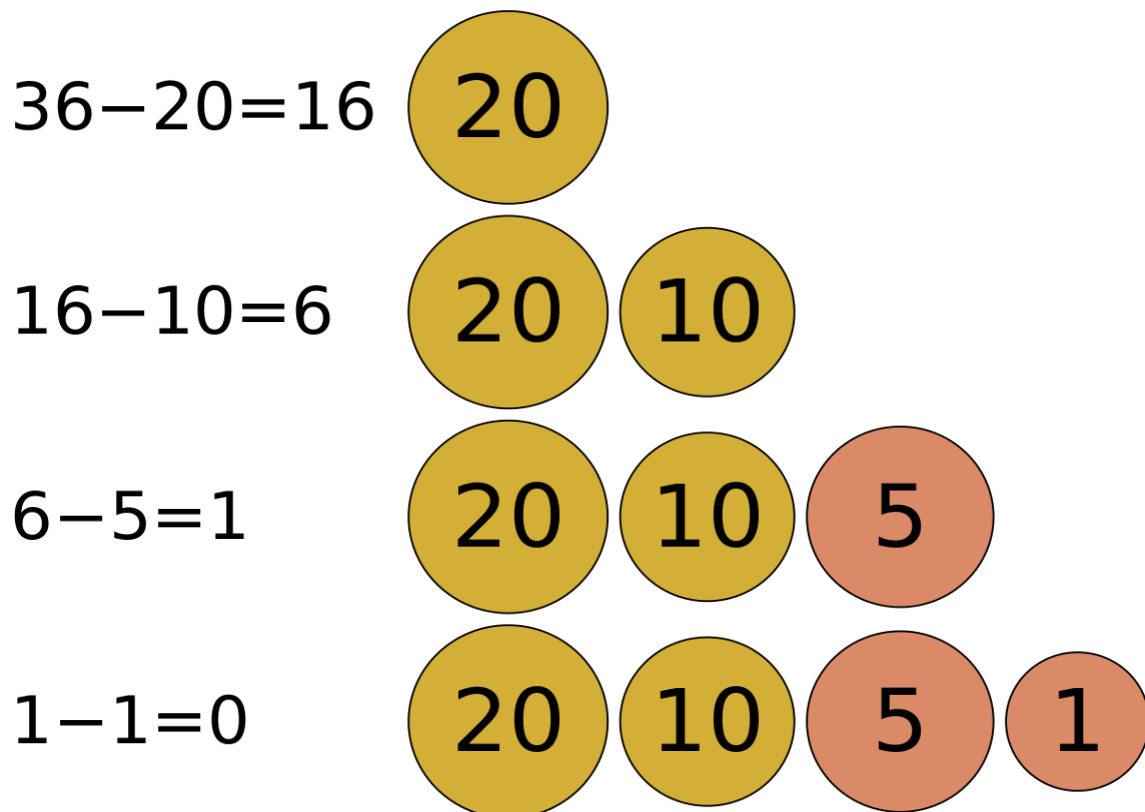
$$f(x) = O(x^3).$$

notation	name
$O(1)$	constant
$O(\log(n))$	logarithmic
$O((\log(n))^c)$	polylogarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential

Greedy algorithms:

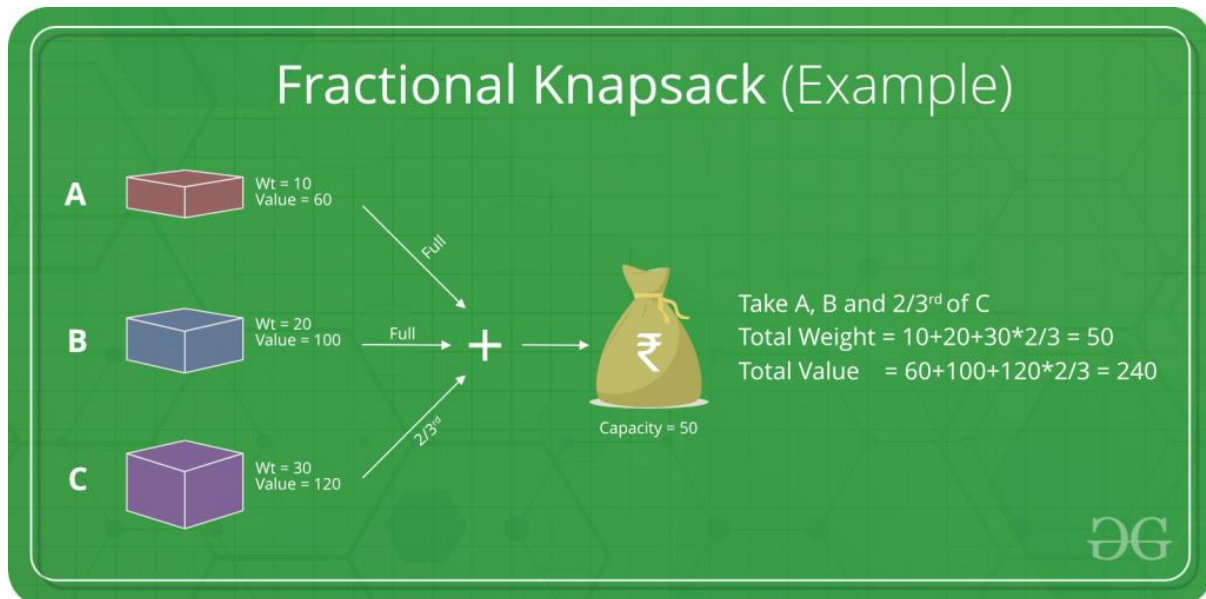
An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.

Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.



For example consider the [Fractional Knapsack Problem](#). The local optimal strategy is to choose the item that has maximum value vs weight ratio. This strategy also leads

to global optimal solution because we allowed to take fractions of an item.



Source: <https://www.geeksforgeeks.org/greedy-algorithms/>

Linked lists:

In classification of data structures we said that in linear-dynamic kind there are stack, Queue, Linked-list. As we already learned about stack and queue now let us discuss about Linked list.

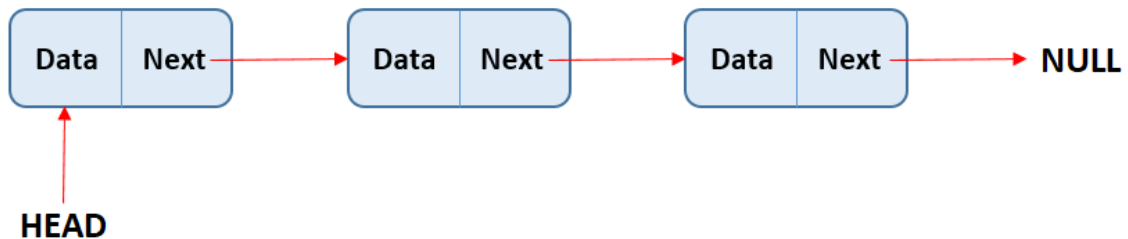
Linked-list is a linear data structure consisting of group of nodes in sequence. In this addresses of each element is not in sequence as arrays.

Classification of Linked-lists:

1. Singly linked-list.
2. Doubly linked-list.
3. Circular linked-list.

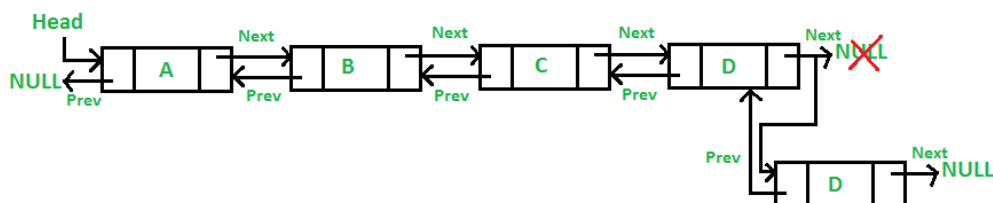
1. Singly Linked-list:

Singly Linked-list contain nodes which have data part as well as address part that is the address of the next node.



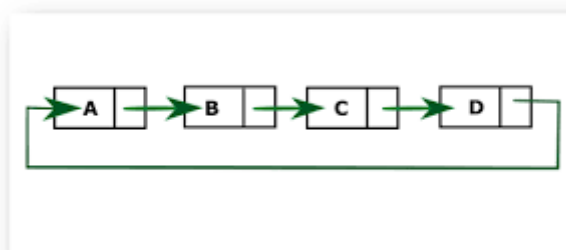
2. Doubly Linked-list:

Each node of doubly Linked-list contain one data part and two address parts, one for address of previous node and other for next node.



3. Circular Linked-list:

The only difference between circular Linked-list and Singly Linked-list is the address part in last node. In Singly Linked-



list the last node contain null as address but in this it contain address of first node.

Applications of Linked-lists:

- Linked-lists are used to implement stacks, queues, graphs, etc.
- Linked-lists let us insert elements at the beginning and end of list.
- In Linked-list we don't need to know the size on advance.

Dynamic Programming:

A Dynamic Programming is an algorithmic technique usually based on a recurrent formula and one(or more) starting states. A sub-solution of a problem is constructed from previously found ones. DP solution have a polynomial complexity which assures a much faster running time than other techniques like backtracking, brute-force etc.

1. Invented by Richard Bellman
2. 1950's was not good years for mathematical research.
3. So he choose this name.
4. It is wonderful, Isn't it?

$O(n \log n)$ sorting:

There are several types of algorithms, each one has a different best and worst-case time complexity and is optimal for a particular type of data structure.

1. Merge sort.
2. Heap sort.
3. Quick sort.

Merge Sort:

It is an classical example of divide-and-conquer algorithm. It has running complexity of $n(\log n)$ in the worst case also. It has two steps.

1. Divide the unsorted list into N -sub lists with one element each.
2. Merge the sub-lists two at a time to produce a sorted sub-list; repeat this until all the elements are included in a single list.

Heap Sort:

Heapsort uses the heap data structure for sorting. The largest (or smallest) element is extracted from the heap (in $O(1)$ time), and the rest of the heap is re-arranged such that the next largest (or smallest) element takes $O(\log n)$ time. Repeating this over n elements makes the overall time complexity of a heap sort $O(n \log(n))$.

Quick Sort:

It is also like merge sort but it has three steps.

1. Select an element that is designated as the pivot from array to be sorted.
2. Move smaller elements to left of the pivot and larger to right side of the pivot.

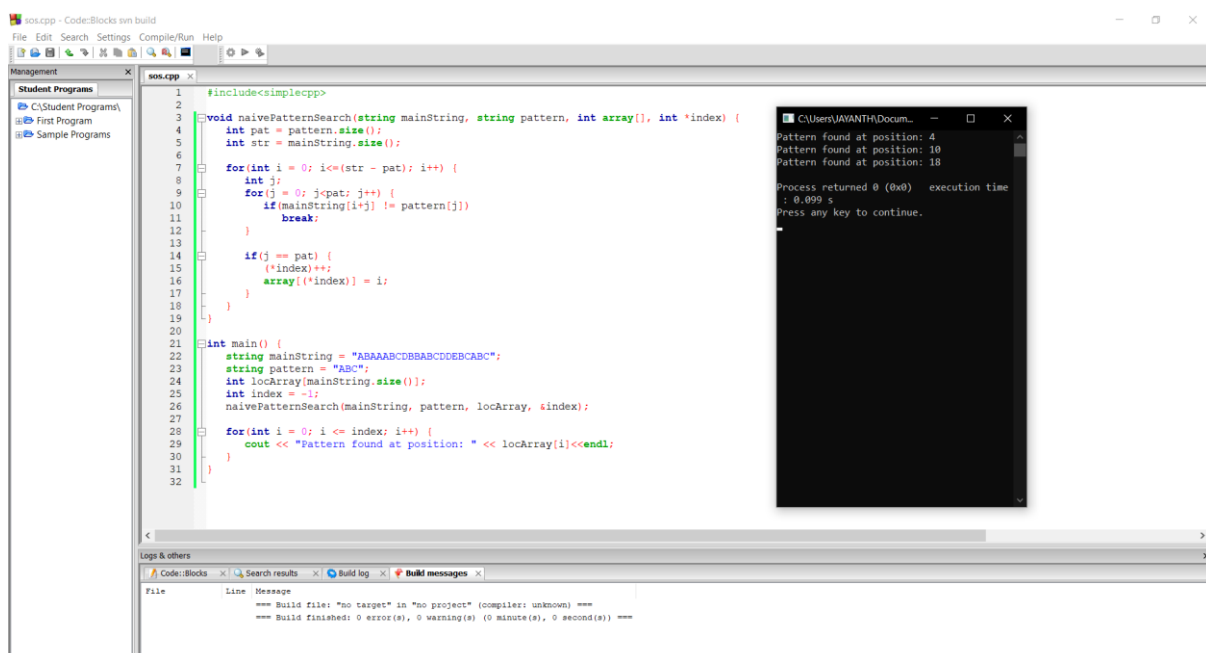
3. Recursively apply step 1 and step 2 of sub arrays.

I have not done the proper implementations here.

Naïve pattern searching:

Naïve pattern searching is the simplest method among other pattern searching algorithms. It checks for all character of the main string to the pattern. This algorithm is helpful for smaller texts. It does not need any pre-processing phases. We can find substring by checking once for the string. It also does not occupy extra space to perform the operation.

Sample code in c++:



The screenshot shows a C++ code editor with a file named 'sos.cpp'. The code implements a naive pattern search algorithm. It defines a function 'naivePatternSearch' that takes a main string, a pattern, an array to store positions, and a reference to an index. The function iterates over the main string, and for each position, it checks if the pattern matches. If it does, it increments the index and stores the position in the array. The 'main' function initializes the main string as 'ABAAABCDDBABCDDEBCABC', the pattern as 'ABC', and calls the 'naivePatternSearch' function. It then prints the positions found. The output window shows the pattern found at positions 4, 10, and 18. The execution time is 0.099 s.

```
#include<string>
using namespace std;

void naivePatternSearch(string mainString, string pattern, int array[], int *index) {
    int pat = pattern.size();
    int str = mainString.size();
    for(int i = 0; i <= (str - pat); i++) {
        int j;
        for(j = 0; j < pat; j++) {
            if(mainString[i+j] != pattern[j])
                break;
        }
        if(j == pat) {
            (*index)++;
            array[*index] = i;
        }
    }
}

int main() {
    string mainString = "ABAAABCDDBABCDDEBCABC";
    string pattern = "ABC";
    int locArray[mainString.size()];
    int index = -1;
    naivePatternSearch(mainString, pattern, locArray, &index);
    for(int i = 0; i <= index; i++) {
        cout << "Pattern found at position: " << locArray[i] << endl;
    }
}
```

Pattern found at position: 4
Pattern found at position: 10
Pattern found at position: 18
Process returned 0 (0x0) execution time : 0.099 s
Press any key to continue.

The time complexity of Naïve Pattern Search method is $O(m*n)$. The m is the size of pattern and n is the size of the main string.

Binary Search:

Binary search is used to quickly find the value of sorted sequence. We call the sought value as target value for clarity. Binary search maintains a contiguous subsequence of the starting sequence where the target value is surely located. This is called the search space. The search space is initially the entire sequence. At each step, the algorithm compares the median value in the search space to the target value. Based on the comparison and because the sequence is sorted, it can then eliminate half of the search space. By doing this repeatedly, it will eventually be left with a search space consisting of a single element, the target value.



Since each comparison binary search uses halves the search space, we can assert and easily prove that binary search will never use more than (in big-oh notation) $O(\log N)$ comparisons to find the target value.

THANK YOU