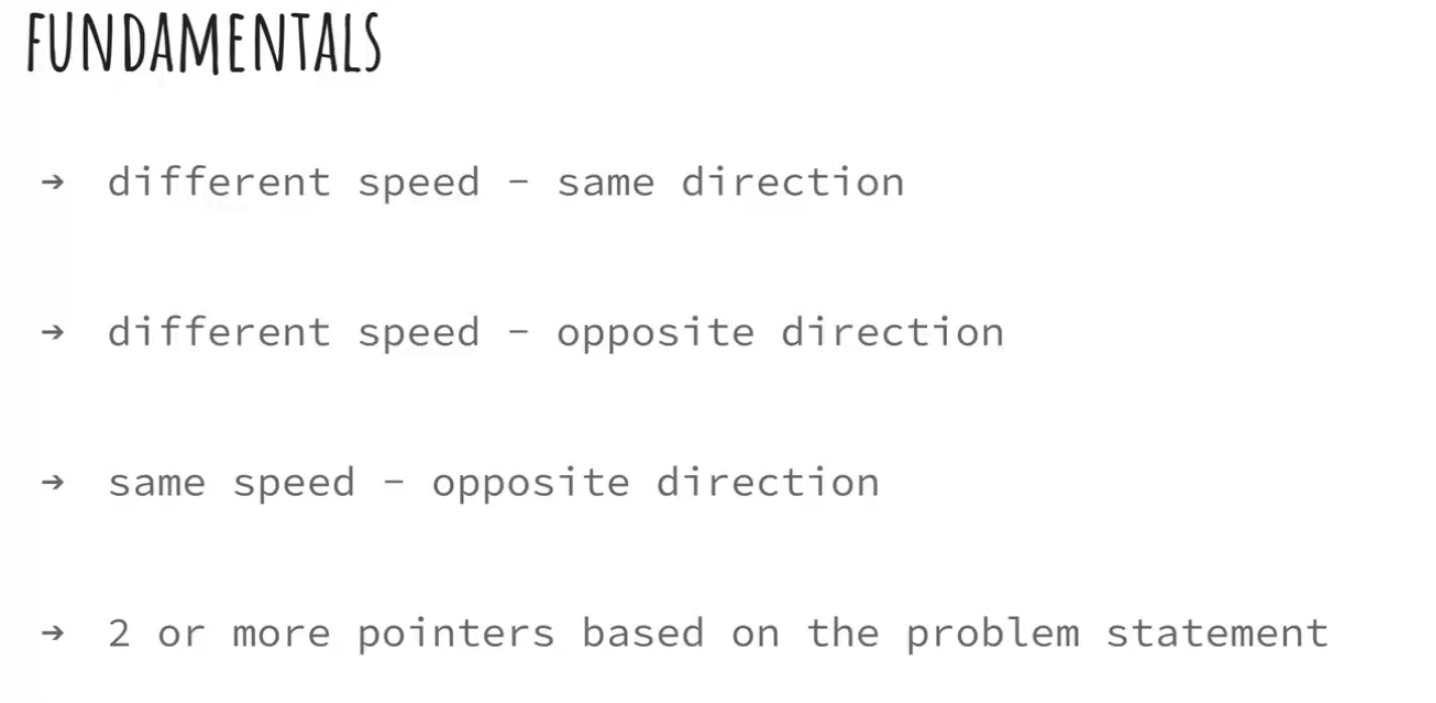
DSA

## *2 pointer Technique:*

In linear DS we use this technique



1. Example of Same direction with different speed (Tortoise and Hare Approach)

A diagram of a diagram

Description automatically generated

1. Example of Opposite directions with different speed

A screenshot of a computer screen

Description automatically generated

1. Example of opposite directions with same speed

Ex: Palindrome

A number on a white background

Description automatically generated with medium confidence

Things to keep in mind:

A screenshot of a computer

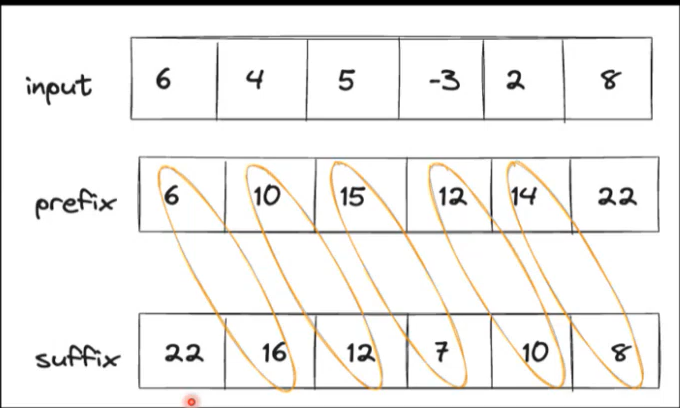
Description automatically generated

Practice:

A close-up of a computer code

Description automatically generated

## *Prefix & Suffix:*



If we have prefix;

Suffix = totalSum – prefix;

If we have suffix;

Prefix = totalSum – suffix;

## *DP:*

(Dynamic Programming) DP is an optimization over plain recursion. In recursion we evaluate same conditions again and again which increases CPE time and memory. So, In DP we store already visited results in memory and avoid evaluating same conditions again.

DP is a powerful technique that helps us solve complex problems by breaking them down into simpler, smaller problems and solving each of those just once. Think of it like solving a big puzzle by first figuring out the smaller pieces and then putting them together.

**1. Recursive Approach:**

* **Concept:** Solve the problem by breaking it down into smaller subproblems and solving each subproblem individually.
* **Pros:** Easy to understand and implement.
* **Cons:** Often inefficient due to redundant calculations and can lead to exponential time complexity.

**2. Top-Down Approach (recursive+ Memoization):**

* **Concept:** Enhance the recursive approach by storing the results of subproblems in a table (memoization) and reusing these results to avoid redundant calculations.
* **Pros:** Significantly improves performance by reducing the number of recursive calls.
* **Cons:** Requires extra space for the memoization table.

**3. bottom-Up Approach (Tabulation):**

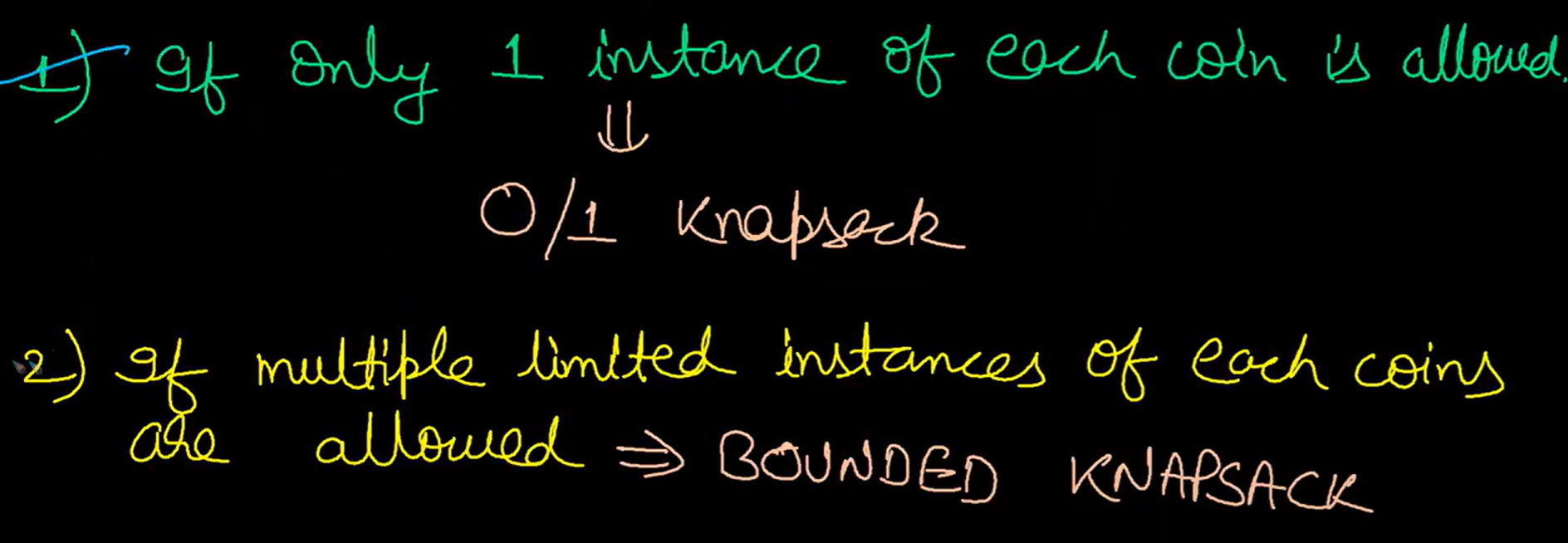
* **Concept:** Use an Iterative method to fill a DP table based on the state transition equations derived from the problem. This approach avoids the overhead of recursive calls.
* **Pros:** Efficient in both time and space, as it avoids stack overflow issues and redundant calculations.
* **Cons:** Can be more complex to implement initially compared to the recursive approach

**Knapsack: Knapsack has three variants;**

Fraction knapsack (it comes under greedy )

0-1 knapsack

Unbounded Knapsack



A diagram of a diagram

Description automatically generated

**Example of Recursion vs Memoization for Kanpsack problem:**

**Recursion:**

int knapsack(int[] wt, int[] val, int W, int n) {

if(n==0 || W==0) return 0;

if(wt[n-1] <= W) {

// Compute the maximum value of including or excluding the nth item

return Math.max(val[n-1] + knapsack(wt, val, W-wt[n-1], n-1),

knapsack(wt, val, W, n-1));

} else if(wt[n-1] > W)

// If weight of the nth item is more than the capacity, skip this item

return knapsack(wt, val, W, n-1);

}

}

**Top-Down:**

int knapsack(int[] wt, int[] val, int W, int n) {

if(n==0 || W==0)

dp[n][W]=0; // Base case: no items left or no capacity left

if( dp[n][W] != -1)

return dp[n][W]; // If the value is already computed, return it

if(wt[n-1] <= W){

// include or exclude and save result

return dp[n][W] = Math.max(val[n-1] + knapsack(wt,val,W-wt[n-1],n-1), knapsack(wt,val,W,n-1));

} else if(wt[n-1] > W)

// skip the weight

return dp[n][W] = knapsack(wt, val, W, n-1);

}

}

**Bottom-Up:**

for(int i=1;i<n+1;i++) {

for(int j=1;j<W+1;j++) {

if(wt[i-1] <= j) {

dp[i][j] = Math.max(val[i-1] + dp[i-1][j-wt[i-1]], dp[i-1][j]);

} else {

dp[i][j] = dp[i-1][j];

}

}

}

Time Complexity=O(n∗W)

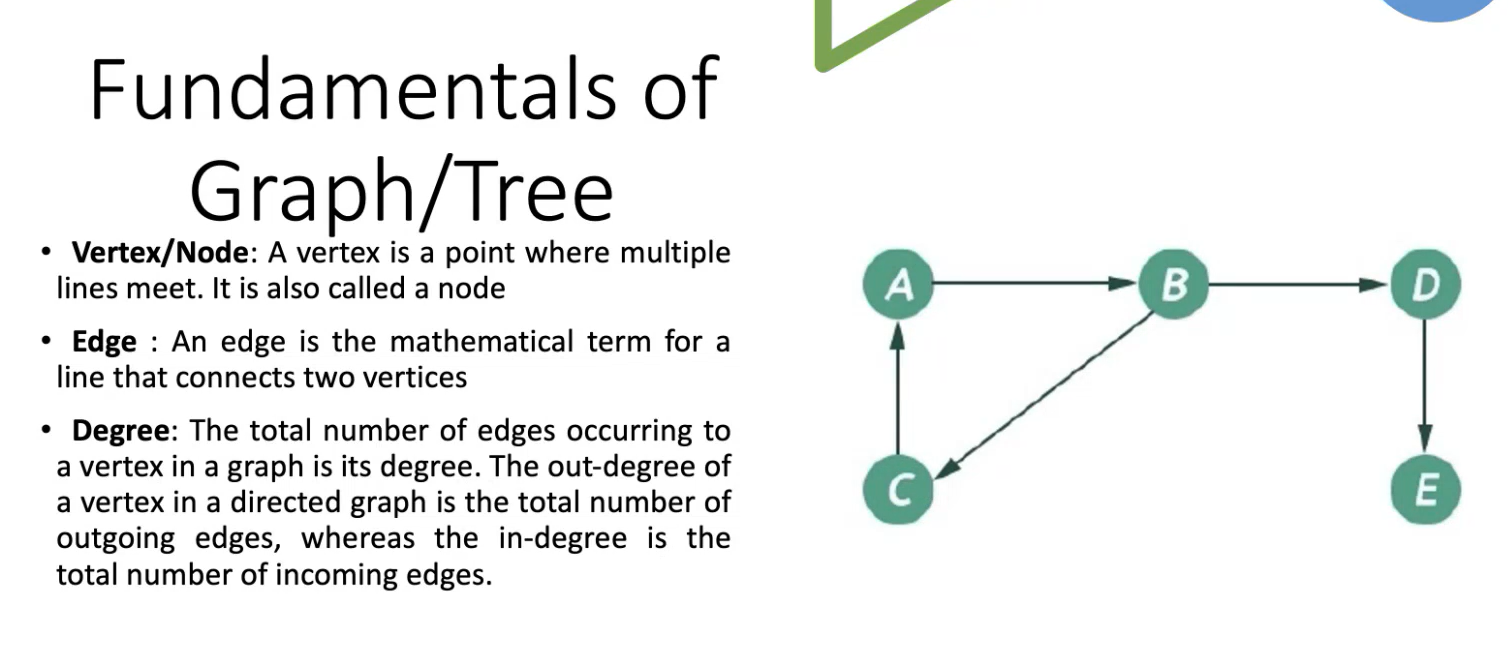
Space Complexity=O(n∗W)

|  |  |
| --- | --- |
| **Memoized** | **Bottom Up** |
| **if(wt[n-1]<=W) {**  **dp[n][W] = max(val[n-1] + knapsack(wt,val,n-1,W-wt[n-1]), knapsack(wt,val,n-1,W))**  **}else if(wt[n-1]>W) {**  **dp[n][W] = knapsack(wt,val,W,n-1);**  **}** | **if(wt[n-1]<=W) {**  **dp[n][W] = max((val[n-1] +**  **dp[n-1][W-wt[n-1]]),**  **dp[n-1][W])**  **}else {**  **dp[n][W]=dp[n-1][W];**  **}** |

Practice Ques:

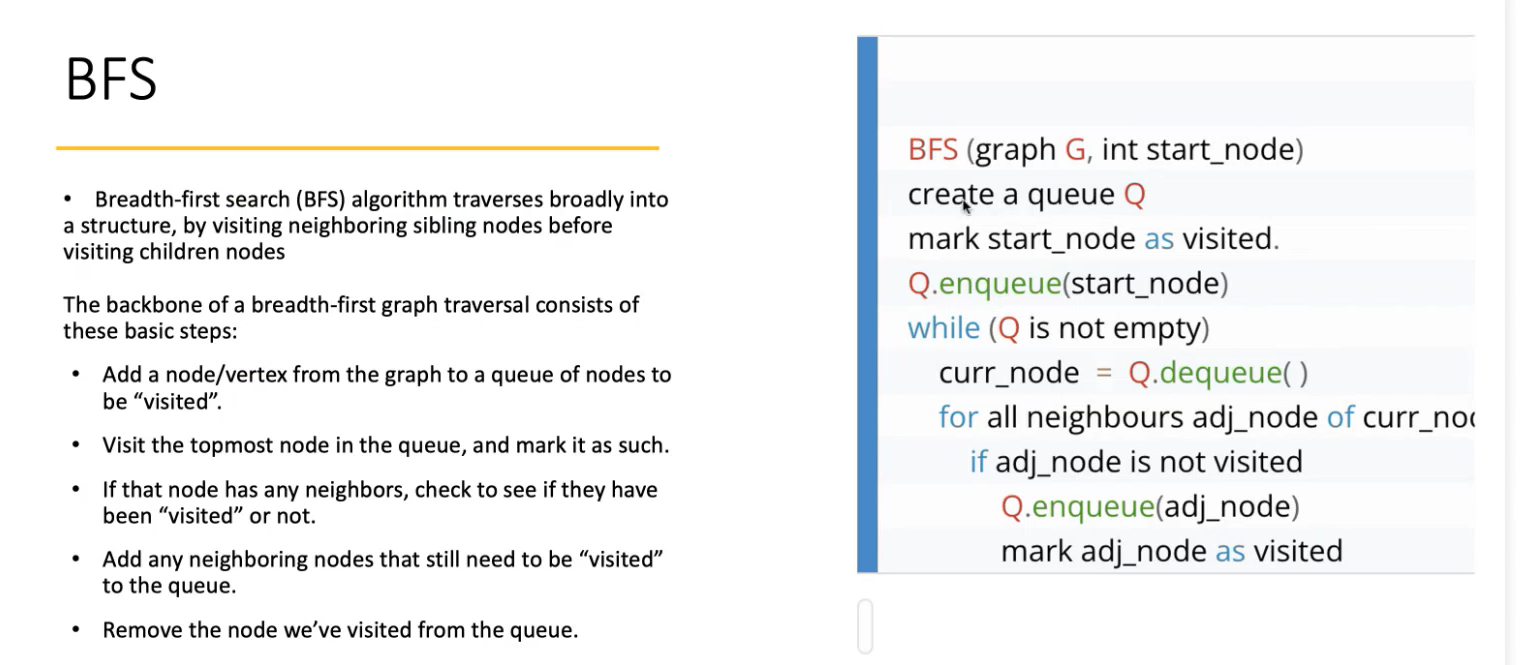
1. https://leetcode.com/problems/partition-equal-subset-sum/
2. https://www.geeksforgeeks.org/subset-sum-problem-dp-25/
3. https://www.geeksforgeeks.org/problems/perfect-sum-problem5633/1
4. https://leetcode.com/problems/partition-array-into-two-arrays-to-minimize-sum-difference/description/
5. https://www.geeksforgeeks.org/count-of-subsets-with-given-difference/
6. https://leetcode.com/problems/target-sum/description/

## *Graph:*



Graph can be traversed eight in BFS or DFS manner.

BFS -> uses Queue



DFS -> uses Stack and then backtrack

