Greedy Algorithm notes:-

1️⃣ Interval / Activity Selection Problems

Problem Description:

- Given a set of intervals, remove the minimum number of intervals so that the remaining intervals **do not overlap**.
- Input: `intervals = [[1,2],[2,3],[1,3]]`
- Output: `1`

  Explanation: Remove `[1,3]` to make intervals non-overlapping.

Approach:

1. Sort intervals by **end time ascending**.
2. Initialize `prevEnd = intervals[0][1]`.
3. Traverse intervals from 1 to n-1:
   - If `interval[i].start >= prevEnd` → pick it → `prevEnd = interval[i].end`.
   - Else → remove it.
4. Return total removed intervals.

```
class Solution {
public:
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        if(intervals.empty()) return 0;
        sort(intervals.begin(), intervals.end(), [](vector<int>& a, vector<int>& b){
            return a[1] < b[1];
        });

        int prevEnd = intervals[0][1];
        int remove = 0;
```

```
        for(int i = 1; i < intervals.size(); i++){
            if(intervals[i][0] >= prevEnd)
                prevEnd = intervals[i][1];
            else
                remove++;
        }
        return remove;
    }
};
```

Key Intuition:

- Pick interval that ends earliest to maximize future picks.

Problem 2: Minimum Platforms

GeeksForGeeks Classic

Problem Description:

- Given arrival and departure times of trains at a station, find **minimum number of platforms** needed so that no train waits.
- Input: `arr = [9:00, 9:40, 9:50], dep = [9:10, 12:00, 11:20]`
- Output: `2`

Approach:

1. Sort arrival and departure arrays separately.
2. Initialize `platforms = 0, maxPlatforms = 0`.
3. Use two pointers `i,j` for arrivals & departures.
4. Traverse:
    - If `arr[i] <= dep[j]` → `platforms++`, `i++`.
    - Else → `platforms--`, `j++`.
    - Update `maxPlatforms`.

5. Return `maxPlatforms`.

```
class Solution {
public:
    int minPlatforms(vector<int>& arr, vector<int>& dep) {
        sort(arr.begin(), arr.end());
        sort(dep.begin(), dep.end());
        int platform = 0, maxPlatform = 0;
        int i = 0, j = 0;

        while(i < arr.size() && j < dep.size()){
            if(arr[i] <= dep[j]){
                platform++;
                i++;
            }
            else{
                platform--;
                j++;
            }
            maxPlatform = max(maxPlatform, platform);
        }
        return maxPlatform;
    }
};
```

Key Intuition:

- Track **overlaps** efficiently with sorted times.

Problem 3: Minimum Arrows to Burst Balloons

LeetCode 452

Problem Description:

- Given points representing balloons on a 1D line `[start,end]`, find **minimum number of arrows** to burst all balloons.

- Input: `points = [[10,16],[2,8],[1,6],[7,12]]`
- Output: `2`

Approach:

1. Sort balloons by **end coordinate** ascending.
2. Initialize `arrows = 1`, `end = points[0][1]`.
3. Traverse balloons:
    - If `start > end` → new arrow, `arrows++`, `end = current end`.
    - Else → same arrow.
4. Return `arrows`.

```
class Solution {
public:
    int findMinArrowShots(vector<vector<int>>& points) {
        if(points.empty()) return 0;
        sort(points.begin(), points.end(), [](vector<int>& a, vector<int>& b){
            return a[1] < b[1];
        });
        int arrows = 1, end = points[0][1];
        for(int i = 1; i < points.size(); i++){
            if(points[i][0] > end){
                arrows++;
                end = points[i][1];
            }
        }
        return arrows;
    }
};
```

Key Intuition:

- Overlap balloons → reuse arrow; gap → new arrow.

Problem 4: Job Sequencing Problem

GeeksForGeeks Classic

Problem Description:

- Given jobs with `(profit, deadline)`, schedule to **maximize total profit**.
- Input: `jobs = [(100,2),(19,1),(27,2),(25,1),(15,3)]`
- Output: `142`

  Explanation: Schedule jobs `[19,100,15]`.

Approach:

1. Sort jobs by **profit descending**.
2. Initialize `slots[n] = -1` (all empty).
3. For each job:
   - Place in latest available slot `<= deadline`.
4. Add profit for assigned jobs.

```
class Solution {
public:
    int jobScheduling(vector<int>& profit, vector<int>& deadline) {
        int n = profit.size();
        vector<pair<int,int>> jobs;
        for(int i = 0; i < n; i++)
            jobs.push_back({profit[i], deadline[i]});

        sort(jobs.rbegin(), jobs.rend());

        vector<int> slot(n+1, -1);
        int totalProfit = 0;

        for(auto job : jobs){
            for(int j = min(n, job.second); j > 0; j--){
                if(slot[j] == -1){
```

```
                slot[j] = 1;
                totalProfit += job.first;
                break;
            }
        }
    }
    return totalProfit;
  }
};
```

Key Intuition:

- Maximize profit by scheduling **high-profit jobs first**.

2️⃣ Assign / Candy Type Problems

Problem 1: Assign Cookies

LeetCode 455

Problem Description:

- Children have greed factors `g[i]`, cookies have sizes `s[j]`.
- Assign cookies so that a child is satisfied if `s[j] >= g[i]`. Maximize **number of satisfied children**.
- Input: `g = [1,2], s = [1,2,3]`
- Output: `2`

Approach:

1. Sort `g` and `s`.
2. Use two pointers: `child = 0, cookie = 0`.
3. Traverse:
    - If `s[cookie] >= g[child]` → satisfy child → `child++, cookie++`.
    - Else → `cookie++`.

4. Return `child`.

```
class Solution {
public:
    int findContentChildren(vector<int>& g, vector<int>& s) {
        sort(g.begin(), g.end());
        sort(s.begin(), s.end());
        int child = 0, cookie = 0;
        while(child < g.size() && cookie < s.size()){
            if(s[cookie] >= g[child]){
                child++;
            }
            cookie++;
        }
        return child;
    }
};
```

Key Intuition:

- Minimum resource first maximizes satisfied children.

Problem 2: Candy

LeetCode 135

Problem Description:

- Each child has `rating[i]`.
- Rules:
    1. Each child ≥ 1 candy
    2. Child with higher rating than neighbor gets more candy
- Return **minimum candies total**.
- Input: `[1,0,2]` → Output: `5` (`[2,1,2]`)

Approach:

1. Initialize `candies[n] = 1`.
2. **Left → Right pass**: if `ratings[i] > ratings[i-1]` → `candies[i] = candies[i-1]+1`.
3. **Right → Left pass**: if `ratings[i] > ratings[i+1]` → `candies[i] = max(candies[i], candies[i+1]+1)`.
4. Sum candies.

```
class Solution {
public:
    int candy(vector<int>& ratings) {
        int n = ratings.size();
        vector<int> candies(n, 1);

        for(int i = 1; i < n; i++)
            if(ratings[i] > ratings[i-1])
                candies[i] = candies[i-1] + 1;

        for(int i = n-2; i >= 0; i--)
            if(ratings[i] > ratings[i+1])
                candies[i] = max(candies[i], candies[i+1]+1);

        int total = 0;
        for(int c : candies) total += c;
        return total;
    }
};
```

Key Intuition:

- Two passes because each child has **2 neighbors**.

3️⃣ Jump / Range Expansion Problems

Problem 1: Jump Game II

LeetCode 45

Problem Description:

- `nums[i]` = max jump from index i
- Find **minimum jumps to reach last index**
- Input: `[2,3,1,1,4]` → Output: `2`

Approach:

1. `jumps=0, currentEnd=0, farthest=0`.
2. Loop `i=0` to `n-2`:
    - `farthest = max(farthest, i+nums[i])`
    - If `i == currentEnd` → jump → `jumps++`, `currentEnd=farthest`
3. Return `jumps`.

```
class Solution {
public:
    int jump(vector<int>& nums) {
        int jumps = 0, currentEnd = 0, farthest = 0;
        for(int i = 0; i < nums.size()-1; i++){
            farthest = max(farthest, i + nums[i]);
            if(i == currentEnd){
                jumps++;
                currentEnd = farthest;
            }
        }
        return jumps;
    }
};
```

Key Intuition:

- Each jump = BFS level, expand reach greedily.

Problem 2: Jump Game I

LeetCode 55

Problem Description:

- Can you reach last index from first?
- Input: `[2,3,1,1,4]` → Output: `true`

Approach:

1. `maxReach = 0`
2. Traverse `i = 0 → n-1`:
   - If `i > maxReach` → false
   - Else `maxReach = max(maxReach, i+nums[i])`
3. Return true if reachable.

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int maxReach = 0;
        for(int i = 0; i < nums.size(); i++){
            if(i > maxReach) return false;
            maxReach = max(maxReach, i + nums[i]);
        }
        return true;
    }
};
```

4️⃣ Transactions / Change / Profit

Problem 1: Lemonade Change

LeetCode 860

Problem Description:

- Bill sequence: `[5,5,5,10,20]`
- Give change using previous bills
- Return true if possible, else false.

Approach:

1. Track counts: `five=0, ten=0`
2. For each bill:
   - $5 → `five++`
   - $10 → give 5 → `five--, ten++`
   - $20 → prefer 10+5 → else 3×5 → else false
3. Return true if all bills handled.

```
class Solution {
public:
    bool lemonadeChange(vector<int>& bills) {
        int five = 0, ten = 0;
        for(int b : bills){
            if(b == 5) five++;
            else if(b == 10){
                if(five == 0) return false;
                five--; ten++;
            }
            else{ // 20
                if(ten > 0 && five > 0){
                    ten--; five--;
                } else if(five >= 3){
                    five -= 3;
                } else return false;
            }
        }
        return true;
```

```
    }
};
```

Problem 2: Gas Station

LeetCode 134

Problem Description:

- Stations: `gas[i]`, `cost[i]`
- Find **start index** to complete circuit.
- Input: `gas=[1,2,3,4,5], cost=[3,4,5,1,2]` → Output: `3`

Approach:

1. `totalCost=0, currentCost=0, start=0`
2. Traverse stations:
    - `diff = gas[i]-cost[i]`
    - `totalCost += diff, currentCost += diff`
    - If `currentCost<0` → start=i+1, currentCost=0
3. If `totalCost>=0` → return start, else -1.

```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int total = 0, curr = 0, start = 0;
        for(int i = 0; i < gas.size(); i++){
            int diff = gas[i] - cost[i];
            total += diff;
            curr += diff;
            if(curr < 0){
                start = i+1;
                curr = 0;
            }
```

```
        }
        return total >= 0 ? start : -1;
    }
};
```