Pre-Requesties:

1. Custom Comparator
2. Custom Data Structure

Pattern #1: Sorting/Iteration

1. **Assign Cookies -** Sort and Compare the Entire Arrays
2. **Lemonade Change -** Sometimes Just Iterating Works Best

Pattern #2: Knapsack

1. **Bounded: Fractional Knapsack -** Take One Only Once and a Custom Comparator
2. **Unbounded: Coin Change -** Sort and Take One Any Number of Times

Pattern #3: Range

1. **Jump Game -** Can you Reach the End?
2. **Jump Game 2 -** Min Jumps to Reach the End? Calculate the Range Every Time
3. **Valid Parenthesis -** Reduce Recursion to Greedy Using Range

Pattern #4: Intervals

1. **Insert Interval -** Find the Left Intervals, the Common Intervals and then the Right
2. **Merge Intervals -** Sort the Intervals First, Then Check for Merging Cases
3. **Non-overlapping Intervals -** Basic Interval Check Problem With Sorting

Pattern #5: Scheduling

1. **Job Sequencing -** Do the Max Profit Jobs First on Their Last Possible Day
2. **Shortest Job First -** Same as Operating System
3. **N Meetings -** Do the Early Ending Meetings First With a Custom Comparator
4. **Min Platforms -** Sort the ARR and DEP Arrays Independently, ++ for A, -- for D
5. **Least Recently Used -** Same as Operating System

Pattern #6: Slope

1. **Candy -** Solve it as a Slope, Will Explain in Detail Below

Pre-Requesties

1. Custom Data Structure
2. Custom Comparator

Custom Data Structure
Let's say you have a data type with three elements. This cannot be stored in a normal array or pair. You need to make a custom data structure to store three elements at once.

```
vector<Job> jobs = {
    {1, 4, 20},
    {2, 1, 10},
    {3, 2, 40},
    {4, 2, 30}
};

struct Job {
    int id;
    int dead;
    int profit;
};
```

Custom Comparator
Let's say you don't have to sort a vector in either ascending or descending order. You want to follow an approach that can not be directly done with STL. In this case, you use a custom comparator using a bool function.

```
pair <int.int> a[] = {{1, 2}, {2, 1}, {4, 1}};
sort(a, a+n, comp);

// sort in ascending order
// sort it according to the 2nd element, if they are the same
// sort it according to the 1st element in descending order

bool comp(pair<int,int> p1, pair<int,int> p2) {
    if(p1.second < p2.second) return true;
    if(p1.second > p2.second) return false;

    // the only case left is that the 2nd elements are the same
    if(p1.first > p2.first) return true;
    return false;
}
```

If the order you want is already there, just return true. Otherwise, return false. The swapping part will be taken care of by the function. In the first line of the comp function, the element in front, "p1," is already smaller than the second element, "p2". That's what we want, so return true. Otherwise, return false.

Pattern #1 - Greedy Sort/Iteration

Idea - The concept here is to normally sort the inputs given to you, or just iterate through the inputs and take the greediest possible step. Nothing fancy, just a normal sorting/iteration while being as greedy as possible.

Assign Cookies ([LINK])
You have some children and some cookies. Each child has a greed factor (the size of the cookie that s/he wants). How many children will get a big enough cookie?

Input: children_greed = [2, 3, 1, 4], cookie_size = [3, 1, 2]
Output: 3

Pattern Matching - The idea will be to first satisfy the greed of the child who wants the smallest cookie size. If you cannot satisfy his/her greed, you won't be able to satisfy any other child, as s/he would want a bigger cookie.

To start with the smallest greed, you need to sort the children_greed vector. You would also want to start with the smallest cookie, so that you are left with the bigger cookies for the more greedy children. So, you also need to sort the cookie_size vector.

Now, children_greed = [1, 2, 3, 4], cookie_size = [1, 2, 3]

Use two pointers and simply compare the two vectors. You will notice that only the greed of 3 children can be satisfied.

Lemonade Change ([LINK])
You have a shop with lemonade for $5, but you have no change (chutthe) when the shop opens. Will you be able to handle every transaction with no change?

Input: bills = [5, 5, 5, 10, 20]
Output: true

Pattern Matching - Here, you cannot sort the bills vector because the customers come when they want; they won't come in order. You cannot make any changes to the input here. Thus, you can only iterate.

Customer 1: $5
Customer 2: $5
Customer 3: $5 (all good till here, no change needed, you have $15)
Customer 4: $10 (you need to return $5, you still have (15 + 10) - 5 = $20)
Customer 5: $20 (you need to return $15, you still have (20 + 20) - 15 = $25)

This is simple math code. If your money goes below 0, then return false. Otherwise, true.

Pattern #2 - Greedy Knapsack

Idea - The concept here is to pick up values till you reach the given target. This is of two types: bounded and unbounded. In bounded, you can pick up an item only once. In unbounded, you can pick up an item an unlimited number of times.

Fractional Knapsack (LINK)
You have some objects with their weights and values. You need to fill a bag of capacity W with these objects and get the maximum value. You can partially pick an item.

Input: value = [60, 100, 120], weight = [10, 20, 30], capacity = 50
Output: 240

Pattern Matching - You have to pick up values till you reach the given target. You can pick an item only once, but you can also break it. This is screaming "Greedy Bounded Knapsack".

First, sort the objects in the order of "most value per unit weight".

Object 1 ⇒ Weight = 10, Value = 60, Value/Weight = 6
Object 2 ⇒ Weight = 20, Value = 100, Value/Weight = 5
Object 3 ⇒ Weight = 30, Value = 120, Value/Weight = 4

Fill object 1 first: Value = 60, Weight = 10, Capacity Left = 40
Now object 2: Value = 160, Weight = 30, Capacity Left = 20

Now in object 3, you cannot take the entire object. So, take only 20 units (left capacity).
Value of 20kg of object 3 = 120/30 x 20 = 80

Final after object 3: Value = 240, Weight = 50, Capacity Left = 0

Coin Change (LINK)
You have coins of some denomination and a target value. Reach that target by picking up the least number of coins. If not possible, return -1.

Input: coins = [1, 2, 5], amount = 11
Output: 3 [5 + 5 + 1]

Pattern Matching - You have to pick up coins till you reach a given target. You can pick a coin an unlimited times. You have to start from the highest denomination coin so that you use the least number of coins.

Pick up coins till you get a target, pick up one coin unlimited times = Unbounded Knapsack
Start from the highest denomination coin = Sort in descending order
MIC DROP.

Pattern #3 - Greedy Range

Idea - The concept here is to realise that your next step or your overall solution will always be part of a range. You need to identify this range and work accordingly.

Jump Game I (LINK)
You are at the first index of the array. You can jump as long as the element you are currently on. Can you reach the end of the array?

Input: nums = [2, 3, 1, 1, 4]
Output: True

Pattern Matching - You have to calculate what is the maximum index that you can reach if you take the longest jump from the current index. If this becomes >= the last index, return true. Otherwise, return false. The maximum index you can reach will be your current index + the jump from this index.

Index 0 ⇒ Jump = 2, Max_Index = 0 + 2 = 2
Index 1 ⇒ Jump = 3, Max_Index = 1 + 3 = 4 (4 is the last index, return true)

Jump Game II (LINK)
You need to return the minimum number of jumps you need to reach the end. It is confirmed that you will always reach the end.

Input: nums = [2, 3, 0, 1, 4]
Output: 2 [2 -> 3 -> end]

Pattern Matching - You calculate the nearest and furthest point you can reach from the given index. Then, you iterate in this range to further find the further and nearest point. Repeat this until the furthest point becomes >= the last index.

Step 1: [2, 3, 0, 1, 4]
Standing at 2, you can reach 3 and 0. Nearest Index = 1, Furthest Index = 2. Now, iterate through this and find the nearest and furthest.

Step 2 : [2, 3, 0, 1, 4]
Standing at 3, you can reach 0, 1 and 4. Nearest Index = 2, Furthest Index = 4. The last index is also 4, so return the number of steps, that is 2.

```
for (int i = near; i <= far; i++) farthest = max(farthest, i + nums[i]);
near = far + 1;
far = farthest;
jumps++;
```

Valid Parenthesis String (LINK)
You have a string with “(”, “)” or “x”. Here, “x” can be treated as “(”, “)” or an empty string. Will this combination result in a valid combination of brackets?

Input: s = "(x))"
Output: true [(())]

Pattern Matching - The open bracket will always be +1, the close bracket will always be -1. But, the “x” can be both + and -. So, take two variables, max and min. When you get an “x”, increase max and decrease min. This covers both the cases of “x” becoming an open and a closed bracket.

During this increment-decrement, if the min ever becomes less than 0, then make it zero. If the max ever becomes less than zero, then return false, as this means the brackets are not balanced. At the end, return true if min == 0, otherwise false.

Step 1: (⇒ Min = 1, Max = 1
Step 2: x ⇒ Min = 0, Max = 2
Step 3:) ⇒ Min = -1 becomes 0, Max = 1
Step 4:) ⇒ Min = -1 becomes 0, Max = 0

(⇒ Increases both min and max
) ⇒ Decreases both min and max
X ⇒ Decrease min and increase max

While calculating this range, max should never be less than 0, and if min ever becomes less than 0, then make it 0. At the end, min should be 0.

Pattern #4 - Greedy Intervals

Idea - This pattern covers all the greedy problems that include intervals. They are mostly solved by comparing the first and second elements of an interval with some other interval.

Insert Intervals (LINK)
You have some intervals and an extra interval. You need to insert this extra interval into the already existing intervals.

Input: intervals = [[1,3], [6,9]], newInterval = [2,5]
Output: [[1,5], [6,9]]

Pattern Matching - To insert this interval, you need to compare the values of the existing intervals with the new interval. Yes, “Greedy Intervals”. If newInterval[1] >= intervals[i][0] and newInterval[0] > intervals[i][1] that is when you insert. Very intuitive.

Merge Intervals (LINK)
You have to merge the overlapping intervals so that no overlapping intervals are left.

Input: intervals = [[1,3], [2,6], [8,10], [15,18]]
Output: [[1,6], [8,10], [15,18]]

Pattern Matching - To merge the intervals, you need to compare the values of the current interval with the next interval. “Greedy Intervals!”. If cur_interval[1] > next_interval[0], that is when you merge. Linear and straightforward solution.

Non-Overlapping Intervals (LINK)
You have to remove the intervals which are causing the intervals to overlap. Remove the least number of intervals to achieve this.

Input: intervals = [[1,2], [2,3], [3,4], [1,3]]
Output: 1

Pattern Matching - To check if there is an overlap, you need to compare the values of the current interval with the previous or next interval. First, sort the intervals based on end time, so that the intervals that end first are counted first. You need to check in order. Use a custom comparator for this.

Intervals = [[1,2], [1,3], [2,3], [3,4]]
🟩 should be greater than or equal to 🟥 to avoid overlap. Whenever this is not followed, add 1 to the count and remove that interval.

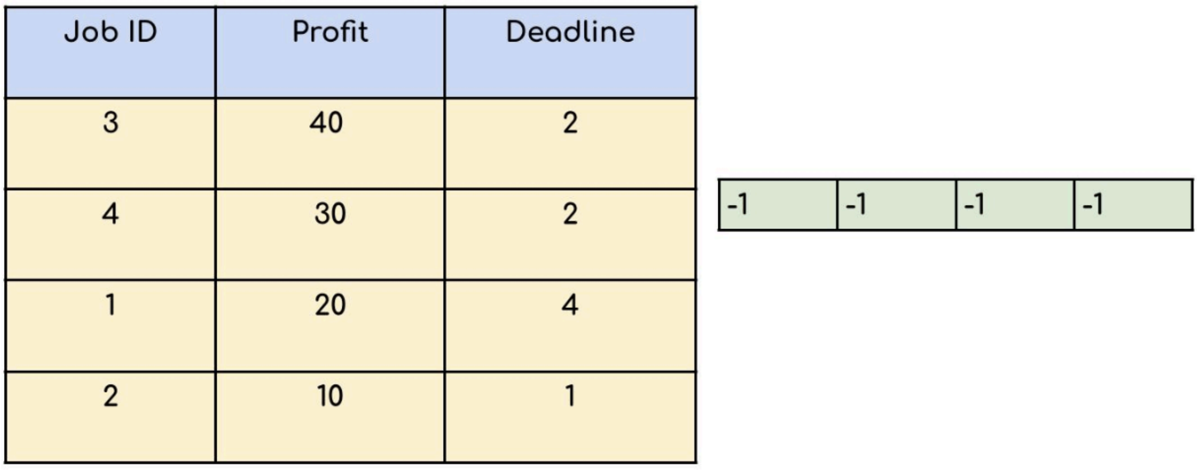
Pattern #5 - Greedy Scheduling

Idea - This pattern covers all the greedy problems that have some kind of scheduling or allotment. Scheduling algorithms of the operating system are also included in this.

Job Sequencing Problem (LINK)
You need to do N jobs within their deadlines to achieve maximum profit.

Input: Jobs = {(1,4,20), (2,1,10), (3,2,40), (4,2,30)}
Output: 2 60

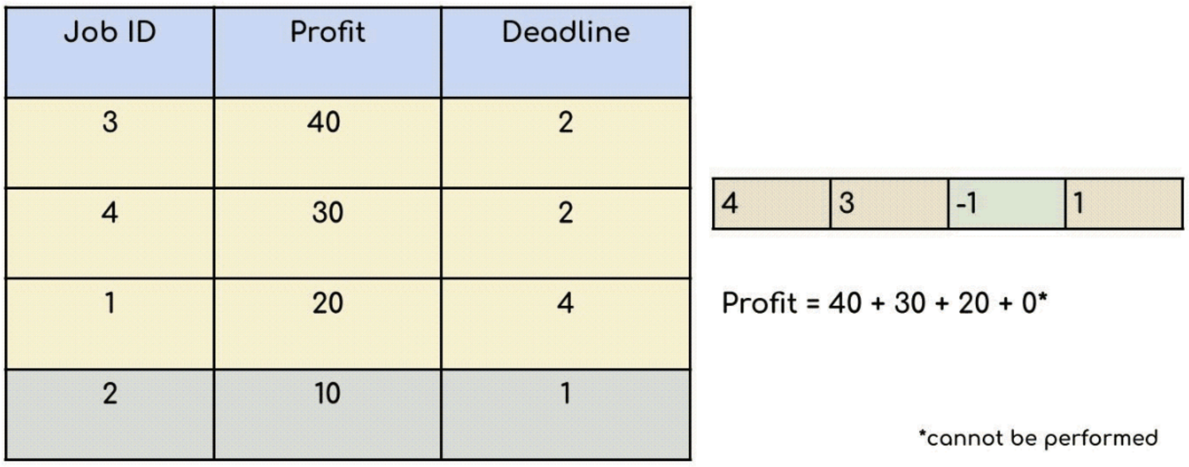
Pattern Matching - You do the maximum profit jobs first, which will ensure the highest profit at the end. To make sure deadlines are taken care of, you perform all these jobs on the last possible day. Make a custom JOB data structure, {ID, Deadline, Profit}.



Job ID	Profit	Deadline
3	40	2
4	30	2
1	20	4
2	10	1

-1	-1	-1	-1

This is your starting point. Do the ID-3 on day 2, ID-4 on day 1 and ID-1 on day 4. Total profit will be 90, and 3 jobs will be done. Use nested for loops to get this done.



Job ID	Profit	Deadline
3	40	2
4	30	2
1	20	4
2	10	1

4	3	-1	1

Profit = 40 + 30 + 20 + 0*
*cannot be performed

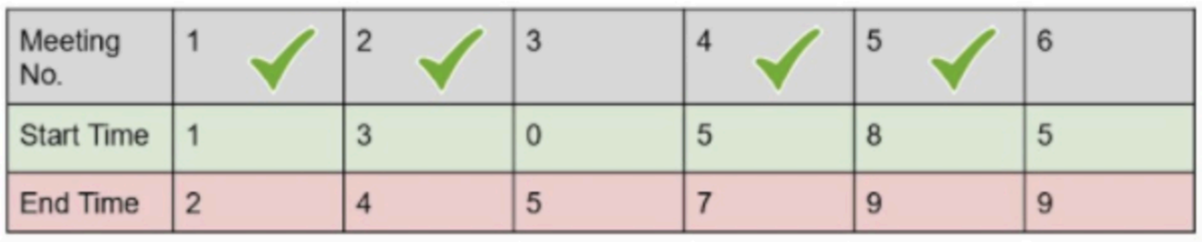
Shortest Job First (LINK)
You have a list of jobs; do the shortest jobs first and calculate the total waiting time.

Input: Jobs = [3, 1, 4, 2, 5]
Output: 4

Pattern Matching - This is the same algorithm as the operating system. Firstly, sort the jobs by their length. Jobs = [1, 2, 3, 4, 5].

Waiting Time for Jobs: 0, 1, 1 + 2, 1 + 2 + 3, 1 + 2 + 3 + 4 = 0, 1, 3, 6, 10 = 20
Average Waiting Time = 20/5 = 4

N Meetings in One Room (LINK)
You have the start and end times of N meetings. You have to schedule the maximum number of meetings possible in one meeting room. No overlapping.



Meeting No.	1 ✓	2 ✓	3	4 ✓	5 ✓	6
Start Time	1	3	0	5	8	5
End Time	2	4	5	7	9	9

Pattern Matching - First, you need to sort the meetings, keeping the meeting with the earlier end time first. Now, you perform the first meeting in every case. From here, if the end time of the first meeting is less than the start time of the next meeting, that's when the next meeting happens. Otherwise, skip that next meeting.

Minimum Number of Platforms Required for a Railway Station (LINK)
N trains are arriving and departing using a railway station. You need to calculate the least number of stations we need to keep the operations running smoothly.

Input:
arr[] = {9:00, 9:45, 9:55, 11:00, 15:00, 18:00}
dep[] = {9:20, 12:00, 11:30, 11:50, 19:00, 20:00}

Output:3

Pattern Matching - The order of these trains doesn't matter. The only thing that matters is the maximum number of arrivals and departures at one instance of time. So, sort both the arrays given.

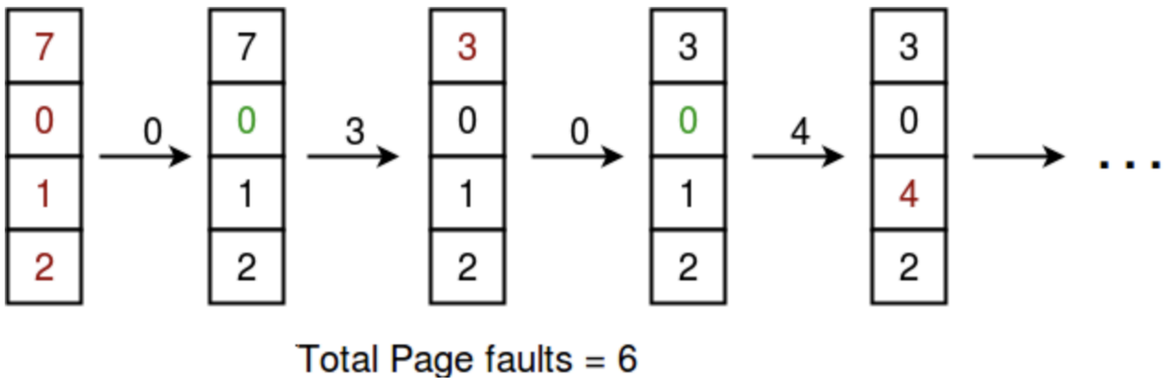
arr[] = {9:00, 9:45, 9:55, 11:00, 15:00, 18:00}
dep[] = {9:20, 11:30, 11:50, 12:00, 19:00, 20:00}

Now, traverse through these two arrays in ascending order with the help of two pointers. Whenever there is an arrival, add to the count; when there is a departure, reduce the count. The maximum value of count will be the answer.

The order will be:
900A, 920D, 945A, 955A, 1100A, 1130D, 1150D, 1200D, 1500A, 1800A, 1900D, 2000D

Count will be:
1, 0, 1, 2, 3, 2, 1, 0, 1, 2, 1, 0

Least Recently Used - Page Replacement Algorithm (LINK)
You are given a data structure with a given capacity, and you need to store the values in that data structure in such a way that the number of page faults is the least. Use the least recently used algorithm from the operating system to get this done.



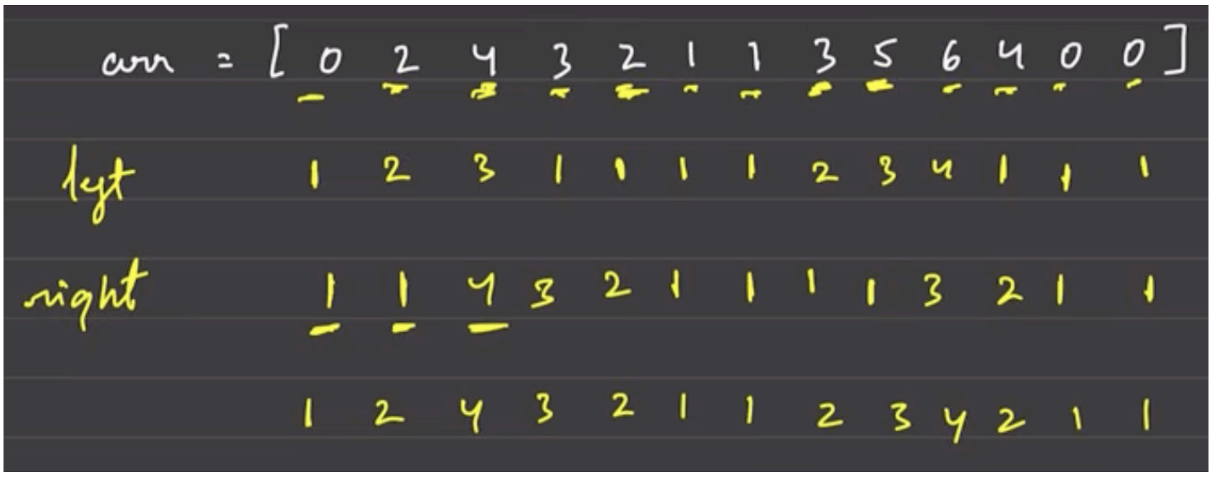
Pattern Matching - The direct approach is obviously to use a map to store the values and search the elements when you need to find them.

Pattern #6 - Greedy Slope

Idea - In these problems, you take care of increasing, decreasing or plateau slope. The outputs are based on the slope direction and may or may not depend on the immediate greedy solution.

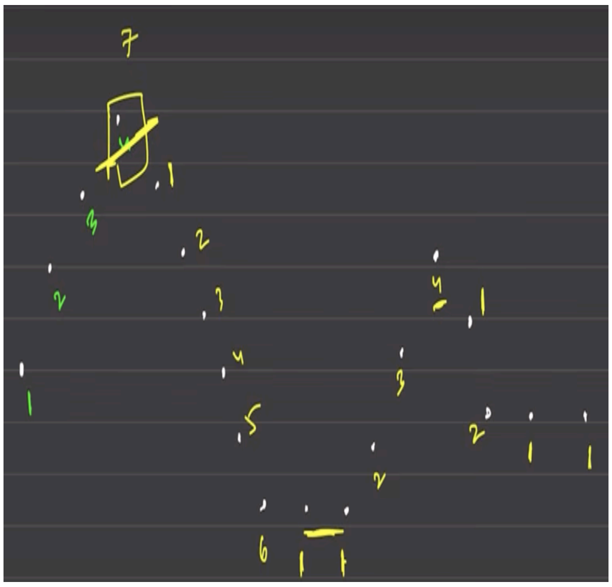
Candy ([LINK](#))
There are N children, and you have to give at least 1 candy to each child. Each child also has a greed factor, and you need to make sure that children with higher greed get more candies than their neighbours.

Input: greed = [1,0,2]
Output: 5 [candies = 2, 1, 2]



arr =	[0	2	4	3	2	1	1	3	5	6	4	0	0]
left	1	2	3	1	1	1	1	2	3	4	1	1	1
right	1	1	4	3	2	1	1	1	1	3	2	1	1
	1	2	4	3	2	1	1	2	3	4	2	1	1

Pattern Matching - A naive approach will be to create a left and right array. The left array satisfies the greed taking into account only the left neighbour. The right array satisfies the greed, taking into account only the right neighbour. The final array has the maximum of these two arrays. The sum of this final array will be the answer.



You need to picture a slope like this, which shows all the increasing, decreasing and plateau slopes of the array. You sequentially give a numerical value to each element of one slope. The point where an increasing and decreasing slope overlap, you will find the maximum value out of the two.

In the first overlapping point, 7 > 4, so the value at that point becomes 7. In the second overlapping point, 4 > 3, so the value at that point becomes 4.

The sum is the answer.