Greedy Algorithm :

Given two vectors, one which has greed values and the other size of cookies.

if size of cookie ≥ greed then child is satisfied. return how many children are satisfied ?

```
int findContentChildren(std::vector<int>& g, std::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]) {
      // If satisfied move to next child and cookie
          child++;
      }
      cookie++;
      // Else move to the next child for the same child
  }
  return child;
}
```

Customers pay you 5, 10 or 20 for a 5 rupees product. Can you return change to everyone ?

```
bool lemonadeChange(vector<int>& bills) {
  int n = bills.size();
  int five = 0;
  int ten = 0;
  // I NEED NOT RETURN ANY 20

  for(int i = 0;i < n;i++) {
      if(bills[i] == 5) {
          five++;
          continue; // NO CHANGE
      }
      else if(bills[i] == 10) {
          ten++;
          if(five) {
              five--;
              continue;
          }
          else return false;
      }
      else {
```

```
            if(five && ten) { // CAN PAY 5, 10
                five--;
                ten--;
            }
            else if(five >= 3) { // OR 5, 5, 5
                five -= 3;
            }
            else return false;
        }
    }
    return true;
}
```

Given an array of lengths of tasks that are to be executed, the one with less time takes priority.

return the average waiting time for all tasks :

```
int solve(vector<int>& bt) {
    int n = bt.size();
    sort(bt.begin(), bt.end());
    int wt = 0; // waiting time for 1st task is 0
    int net_wt = 0;
    for(int i = 0; i < n; i++) {
        net_wt += wt;
        wt += bt[i];
    }
    return net_wt / n;
}
```

Given an array where each entry specifies how many jumps we can make, determine if we can cross the array ?

- if all the entries are positive it is always true, the issue comes when there are zeroes.

```
bool canJump(vector<int>& nums) {
    int n  = nums.size();
    int maxi = 0;

    for(int i = 0;i < n;i++) {
        if(i <= maxi) {
            maxi = max(maxi, i + nums[i]);
            // KEEP TRACK OF WHAT INDEX YOU COULD HAVE REACHED,
            // {This is independet of from where you have come to that index}
        }
        else return false;
        // IF BY CHANCE WE COME TO AN INDEX WHICH IS LESS THAN MAX_INDEX,
        // IT => THERE WERE 0'S IN BETWEEN
    }
```

```
  return true;
}
```

What if in the previous question it is guaranteed that we can reach the end. return min number of steps to do so.

```
int jump(vector<int>& nums) {
    int n = nums.size();
    if (n <= 1) return 0; // no jumping required

    int jumps = 0;
    int l = 0;
    int r = 0;
    while (r < n - 1) { // as if we reach n-1, its crossed
        int farthest = 0; // cover the max range reached in each jump
        for (int i = l; i <= r; i++) {
            farthest = max(farthest, i + nums[i]);
        }
        l = r + 1;
        r = farthest;
        jumps++;
    }
    return jumps;
}
```

Given jobs with deadlines and profit, return max_profit earned.

```
struct Job
{
    int id;    // Job Id
    int dead;  // Deadline of job
    int profit; // Profit if job is over before or on deadline
};

class Solution
{
    public:
    static bool cus_com(Job a, Job b) {
        return a.profit > b.profit;
    }

    vector<int> JobScheduling(Job arr[], int n)
    {
        // Sorting jobs by profit in descending order using custom comparator
        sort(arr, arr + n, cus_com); ARRAY DECOMPOSES TO A POINTER

        // Find the maximum deadline TO DETERMINE THE SLOTS SIZE
```

```
        int max_deadline = 0;
        for (int i = 0; i < n; i++) {
            max_deadline = max(max_deadline, arr[i].dead);
        }

        // Create a slot array to keep track of free time slots
        vector<int> slot(max_deadline + 1, -1);
        int countJobs = 0, maxProfit = 0;

        // Iterate through all given jobs
        for (int i = 0; i < n; i++) {
            // Find a free slot for this job
            //(Note that we start from the last possible slot)
            // AS THERE IS TIME WE CAN DELAY IT IF WANTED
            for (int j = arr[i].dead; j > 0; j--) {
                // Free slot found
                if (slot[j] == -1) {
                    slot[j] = i; // Assign the job to this slot
                    countJobs++;
                    maxProfit += arr[i].profit;
                    break;
                }
            }
        }
        return {countJobs, maxProfit};
    }
};
```

Given two vectors one having starting times and the other ending of meetings. There is only one available meeting room. return max meetings possible

```
vector<pair<int, int>> meetings;
  for (int i = 0; i < n; i++) {
      meetings.push_back({end[i], start[i]});
  }

  // Sort meetings based on their end time
  // IF THERE ARE MULTIPLE MEETINGS ENDING AT SAME TIME
  // WE CAN CONSIDER ONLY ONE OF THOSE ALL
  sort(meetings.begin(), meetings.end());

  int count = 1; ATLEAST ONE MEETING IS SURELY POSSIBLE
  int last_end_time = meetings[0].first;

  for (int i = 1; i < n; i++) {
      if (meetings[i].second > last_end_time) {
          count++;
```

```
            last_end_time = meetings[i].first;
        }
    }
return count;
```

Given few intervals, return how many min intervals we need to remove to make the remaining intervals non overlapping

```
int n = intervals.size();
vector<pair<int, int>> mp;
for(auto interval : intervals) {
    mp.push_back({interval[1], interval[0]});
}
sort(mp.begin(), mp.end());

int count = 1;
int last_interval_end = mp[0].first;

for(int i = 1;i < n;i++) {
    if(mp[i].second >= last_interval_end) {
        count++;
        last_interval_end = mp[i].first;
    }
}
return n - count;
// IT IS JUST THE NUMBER OF MEETINGS WHICH WERE NOT POSSIBLE
```

Given some existing intervals, we have to insert a new interval in it. If there is an overlapping consider merging and making one big interval.

```
vector<vector<int>> addInterval(vector<vector<int>> &intervals, int n,
vector<int> &newInterval)
{
   vector<vector<int>> ans;
   int i = 0;
   for(;i < n;i++) {
       if(newInterval[0] > intervals[i][1]) {
           ans.push_back(intervals[i]);
       }
       else break;
       // IT IS MENTIONED THEY ARE SORTED BASED ON THEIR START TIME
   }

   for(;i < n;i++) {
       if(newInterval[1] >= intervals[i][0]) {
           newInterval[0] = min(newInterval[0], intervals[i][0]);
           newInterval[1] = max(newInterval[1], intervals[i][1]);
```

```
        }
        else break;
    }
    ans.push_back(newInterval);
    // WE ARE PUSHING AT THE END BEACAUSE THERE COULD BE
    // MORE THAN ONE OVERLAPPING INTERVALS

    for(;i < n;i++) {
        if(newInterval[1] < intervals[i][0]) {
            ans.push_back(intervals[i]);
        }
    }
    return ans;
}
```

Given two arrays in which there are arrival and departure time, return minimum number of platform required to make sure trains don’t clash.

```
int calculateMinPatforms(int at[], int dt[], int n) {
  sort(at, at+n);
  sort(dt, dt+n);

  int ans = 0, maxi = 0;
  int i = 0, j = 0;
  while(i < n) {
  // WHEN ARRIVALS ARE DONE THEN WE CAN TERMINATE AS ONLY DEPARTURE REMAINS
      if(at[i] <= dt[j]) {
      // WHEN TWO TRAINS ARRIVE AND DEPART AT SAME TIME WE
      // CONSIDER THEM TO BE DIFFERENT
          maxi++; // DEPENDS ON NUMBER OF ARRIVALS
          ans = max(ans, maxi);
          i++;
      }
      else {
          maxi--;
          j++;
      }
  }
  return ans;
}
```

Valid Parenthesis :

- when we are just given a string of braces, if we encounter a open brace then +1, closed one -1 . At the end if the final result is 0 good. But make sure you don’t get a -ve anywhere that means }{.
- Now say we are given * along with {, } and * can be {, }, or ‘’ now tell ?
- we can just do a recursion trying all different possibilities with *

```
bool checkValidString(std::string s) {
  int minOpen = 0; // Minimum number of open parentheses
  int maxOpen = 0; // Maximum number of open parentheses

  for (char c : s) {
      if (c == '(') {
          minOpen++;
          maxOpen++;
      } else if (c == ')') {
          minOpen = max(minOpen - 1, 0);
          maxOpen--;
          if (maxOpen < 0) return false; // More ')' than '(' or '*'
      } else { // '*'
          minOpen = max(minOpen - 1, 0); // Treat '*' as ')'
          maxOpen++; // Treat '*' as '('
      }
  }

  return minOpen == 0; // All '(' are matched
}
```

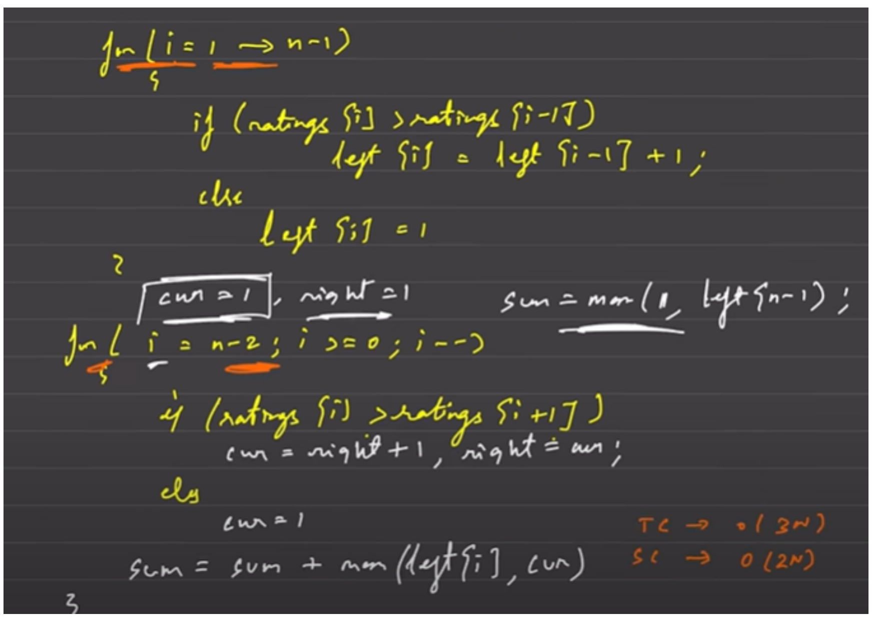
Candy : (Min number required)

- Each child must have at least one candy.
- A child with a higher rating than their adjacent children must get more candies than their adjacent children.

```
int minCandies(const std::vector<int>& ratings) {
    int n = ratings.size();
    if (n == 0) return 0;

    // Step 1: Initialize two arrays to store the number of candies
    vector<int> left2right(n, 1);
    vector<int> right2left(n, 1);

    // Step 2: Traverse the ratings from left to right
    for (int i = 1; i < n; ++i) {
        if (ratings[i] > ratings[i - 1]) {
        // For sure takes care of left constraint
            left2right[i] = left2right[i - 1] + 1;
        }
    }

    // Step 3: Traverse the ratings from right to left
    for (int i = n - 2; i >= 0; --i) {
        if (ratings[i] > ratings[i + 1]) {
        // For sure takes care of right constraint
```

```
            right2left[i] = right2left[i + 1] + 1;
        }
    }

    // Step 4: Calculate the total number of candies
    int totalCandies = 0;
    for (int i = 0; i < n; ++i) {
        totalCandies += max(left2right[i], right2left[i]);
        // For sure takes care of both constraints
    }

    return totalCandies;
}
```

- An extra space of O(2n) is being used here. We can completely omit the right array and keep integers curr and just_right and while traversing back we can keep adding sum.



- Space → O(n) and Time → O(2n)
- Most optimal one is

```
int function(vector<int> &ratings) {
    int n = ratings.size();
    int sum = 1, i = 1;
    while(i < n) {
        if(ratings[i] == ratings[i-1]) {
            sum += 1;
            i++;
            continue;
        }
```

```
        peak = 1;
        while(i < n && ratings[i] > ratings[i-1]) {
            peak += 1;
            sum += peak;
            i++; // If we continue here peak will be made 1 so don't
        }
        down = 1;
        while(i < n && ratings[i] < ratings[i-1]) {
            sum += down;
            i++;
            down += 1;
        }
        if(down > peak) sum += (down - peak);
    }
    return sum;
}
```

Fractional Knap-Sack :

Here you can even take fraction of the item

1. Calculate the **value-to-weight ratio** for each item.
2. Sort the items in **descending order** of this ratio.
3. Take **as much** of the item with the highest ratio **as possible**.
4. Move to the next item with the highest ratio and repeat **until** the knapsack is **full**.

```
bool compare(Item a, Item b) {
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

// Function to calculate the maximum value we can get
double fractionalKnapsack(int W, stdvector<Item>& items) {
    // Sort items by value-to-weight ratio in descending order
    stdsort(items.begin(), items.end(), compare);

    double totalValue = 0.0;  // Variable to store the total value

    for (auto& item : items) {
        if (W == 0) break;  // If the knapsack is full, break the loop

        if (item.weight <= W) {
            // If the item can be completely added
            W -= item.weight;
            totalValue += item.value;
        } else {
```

```
            // If only a fraction of the item can be added
            totalValue += item.value * ((double)W / item.weight);
            W = 0;
        }
    }

    return totalValue;
}
```