* Fractional Knapsack problem

→ PS :- We have to rob some items which each of them have some wealth attached to it and the weight that particular item takes.
→ We only have a bag with limited capacity, hence we need to choose items that help us maximise wealth.

→ Unlike 0/1 Knapsack, here the items can be broken down.

→ Input :- [(60, 10), (100, 20), (120, 30)].
(val, wt)

capacity = 50

→ Output :- 240 [60, 100, 80] * (120/30) * 20.

→ Approach:

To iterate over array to find maxProfit
↑
:- O(N + NlogN).
↑
Sorting based on ratio

(i) Sort the array based on profit/weight ratio (desc).

(ii) Be greedy and take the items with highest ratio
↳ iff our remaining capacity >= item's capacity

(iii) if capacity of item exceeds our capacity, pickup the fraction of it

Ex:- (120, 30).

if for 30 units profit = 120 (30 → 120)
for 1 unit profit = 120/30 = 4. (1 → 120/30).

∴ if rem cap = 5
updated profit = (120/30) * 5

(120/30) ← profit for 1 unit
5 → our max capacity

(Q) Minimum Coins problem.

→ PS:- Return the minimum number of coins required to form the given amount

→ Note that we have ∞ supply of each coin

→ Input :- N = 43 , output = 5 (20+20+2+1).

↳ Sample Space : {1, 2, 5, 10, 20, 100, 500, 1000} (INR currency).

→ Approach

(i) for each N, find the max closest coin, note
Ex:- for N=43, closest = 20 (50 > 43).

(ii) Subtract it from N, add to coins list

(iii) Repeat till N == 0.

(Q) Lemonade change

→ PS:- At Lemonade stand, users want to buy lemonade, each lemonade costs 5 $
→ for whatever amount user give, we need to deduct 5 $ and return back the remaining amount.
→ we only have 5 $, 10 $ and 20 $
→ Initially we dont have any balance.
→ Return true if we sell Lemonade to all users with current balance.

→ Input :- bills = [5, 5, 5, 10, 20].
→ output :- True
→ 5 (nothing to return)
→ 5 " "
→ 5 " "
→ 10 (return 5) Balance = 15 + 10 - 5
→ 20 (return 15) = 20
>> 5 + 10

☆ Approach

(i) Count the numbers of each currencies

↳ if 5

five++, nothing to return

↳ if 10

if 5 == 0 → false

5--; 10++;

↳ if 20

↳ if we have 3 5's → 5-=3

↳ if we have one 10, one 5

↳ 10--; 5--;

↳ else false;

→ return True.

we need to first check with 10's & 5's. because if we give away 3 5's, it could cause problem in future.

[5,5,5,10,20,10].

giveaway for

left [10,10] ✓

(Q) Valid Paranthesis

→ PS:- Given a String containing '(', ')', '*' return true if we have a valid paranthesis match, else return false.

→ A paranthesis match is valid iff for every opening we have a closing bracket.
→ * can act as '(' or ')' or ''.

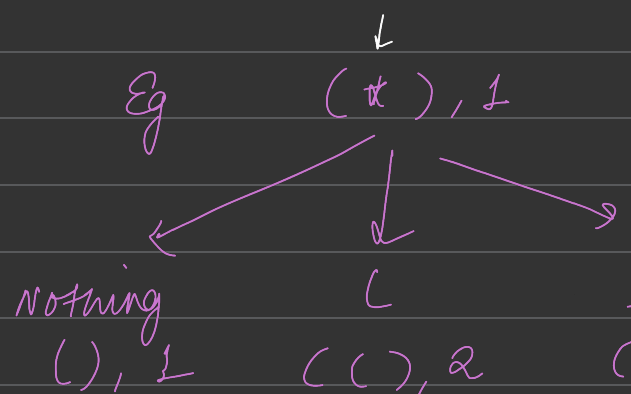
→ Input :- '((*)' output :- True
((())).

→ Input :- '(**(' output :- false

★ Recursion Based Approach

isValid(ind, s, count).

→ In place of '*' try out all possible different combinations



→ if s[ind] = '(', move to next, count++;
→ if s[ind] = ')', move to next count--;
→ if s[ind] = '*'
- Try '(' case
- Try ')' case
- Try Nothing case

} Return OR of all

★ TC :- $O(3^N)$ Memo $O(N^2)$
★ SC :- $O(N)$ → $O(N^2)$.

☆ Greedy Approach

→ Still try out all possible combinations of * but in O(N).
(→ max)

→ min = min. number of unmatched '('

→ To do above maintain a range [min, max].

(i) if '(' min++; max++;

(ii) if ')' min--; max--;

(iii) if * min--; max++;

Trying out all combos

-1 0 +1

★ Dry-Run.

$s = "((*)"$

(i) (
- min = 1
- max = 1

(ii) (
- min = 2
- max = 2

(iii) *

(2-1) (2-0) (2+1).

∴ min = 1

max = 3.

(This is why -- and ++).

(iv))
- min = 0
- max = 2.

```
bool isValid(s){
    min = max = 0;

    for (i = 0 → N){

        if (s[i] == '('){
            min++;
            max++;
        }
        else if (s[i] == ')'){
            min--;
            max--;
        }
        else {
            min--;
            max++;
        }
        if (min < 0) min = 0;
    }  if (max < 0) return false;
    return True;
}
```

* TC :- O(N)
* SC :- O(1)

-ve not possible

(Q) N meetings in one Room

→ PS:- Given 'N' meetings, their start & end Time, return the max number of meetings in one room.

→ Note that we cannot start a new meeting until the old meeting has ended.

→ Input :- start : [1, 3, 0, 5, 8, 5]
end : [2, 4, 6, 7, 9, 9].

→ output :- 4

(1, 2) (5, 7)
(3, 4) (8, 9)

☆ Approach

→ Be Greedy and choose the meetings that end early.

↳ Sort the meetingTime array acc. to endingTime ← {end[i], start[i]}.

→ maxMeetings = 1 for sure.

↳ roomfreeTime = firstMeeting.end.

→ we can only pick a meeting if

if meeting[i].start > lastMeeting.End

↳ true : count++; roomfreeTime

(Q) Jump Game.

→ PS:- We are given integer array nums, we are placed at firstIndex, each index represents the maximum jump we can take.

→ Return True if we can reach the lastIndex

→ Input :- nums :- [2,3,1,1,4] output :- True

2 → 3 → 4.
0 1 4

→ Input :- nums : [3,2,1,0,4] Output :- false

☆ Approach

(i) Each index doesn't represent the actual jump, it represents the max jump we can take (3 => we can take 1, 2, 3 Jumps).

(ii) Keep track at each index what is the maximum jump we can take

```
maxJump = max(maxJump, i + arr[i]);
```

(iii) If at any instance we get

```
i > maxJump → false
```

↳ This can only occur if we are not able to reach lastIndex

[2,3,1,1,4]

(Q) Jump Game - 2

→ PS :- Same question as above, but here we have to return the minJumps required to reach n-1.

→ Input :- nums: [2,3,1,1,4]. → output :- 2

2 → 3 → 4.
0 1 4.

✱ Recursion (DP) approach

```
int minJumps (ind, jumps) {
    if (ind >= n-1) return jumps;
    mini = INT_MAX;
    for (i = 1 → arr[ind]) {
        mini = min (mini, f(ind+i, jumps+1))
    }
    return mini;
}
```

✱ TC :- $O(N^N)$

↓ optimised

memoization

☆ Greedy Approach

→ Instead of computing all branches, get them as range.

(i) Similar to valid parantheses, maintain a left, right Range we can reach

(ii) leftRange = rightRange + 1

```
rightRange = farthestInd;
while (r < n-1)
{   for (ind = leftRange → rightRange) {
        farthestInd = max (farthestInd, ind + arr[ind]);
    l = r+1;
    }  r = farthestInd;
    jumps++;
}
```

(Q) Minimum platforms required by Railway station

→ Ps:- we are given arrival & departure times of 'n' trains, we need to find the max number of platforms required to accomodate these n trains.

→ Input :- arr : [900, 940, 950, 1100, 1500, 1800]
dep : [910, 1200, 1120, 1130, 1900, 2000]

→ output :- 3

☆ Bruteforce

→ we will try to find at one instance, what is the maximum number of platforms that were required to handle everything.

→ A new platform is required iff

OG Arrival	OG Departure
(i) n Arrival	n Departure
(ii) n Arrival	n Departure
(iii)	n Arrival n Departure
(iv)	n Arrival n Departure.

```
for (i = 0 → N) {
    platforms = 0;
    for (j = 0 → i) {
        if ((i) || (ii) || (iii) || (iv)) {
            platforms++;
        }
    }
    maxP = max(maxP, platforms);
}
```

☆ TC :- $O(N^2)$

☆ Greedy Approach

sort
- $O(N \log N)$ +
- $O(N \log N)$ +
- $O(2N)$ → traversal

→ Stand at a station, as time passes by observe what action occurs
- Arrival
- Departure

(i) Sort both arr and dep array

(ii) if arr[i] <= dep[j]
↳ arrival will happen first
∴ platforms++;
i++;

(iii) if (arr[i] > dep[j])
platforms--;
j++;
↳ we can re-use current platform

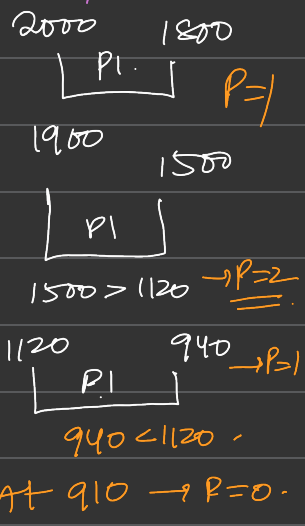
(iv) while observing these actions, at an instance the more platforms required is our answer.

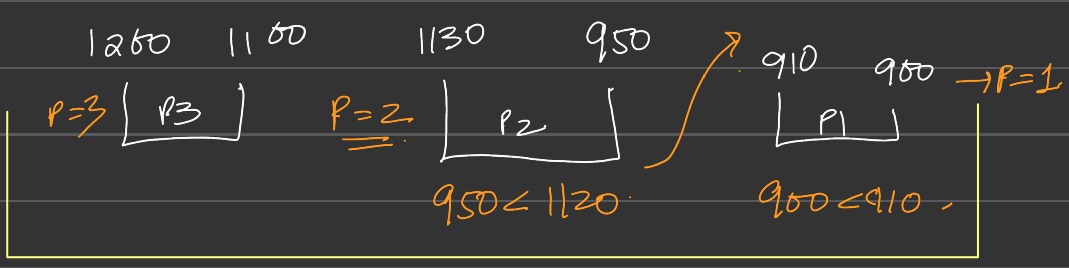
maxPlatforms = max (maxPlatforms, Platforms);

Eg :-

Input : arr : [900, 940, 950, 1100, 1500, 1800]

dep : [910, 1120, 1130, 1200, 1900, 2000].





max P = 1 1 2. 3.

(Q) Job Sequencing Problem

→ PS:- We are given profits and deadlines of 'n' jobs, return the total jobs that can be performed within deadline and the max profit associated with it

→ Input : deadline : [4, 1, 1, 1]
profit : [20, 10, 40, 30].

NOTE :- Here deadline = 4 means Job 1 can be performed anytime from 1 → 4

→ output :- {2, 60}.

Job 1 at d = 4, p = 20
Job 3 at d = 1 p = 60

TC :- O(N log N) +
O(N * maxDeadline).

☆ Approach:

(i) Be Greedy sort based on profit

(ii) Maintain a deadline visited array
↓ (min - max deadline)

(iii) for each job, try to fit in the job in as far deadline as possible

if (!vis[d]) : d = dead[i] → 1.

why far?
→ consider we filled at 1
→ There is a chance the next max profit Job might have deadline only as 1.

(iv) Repeat till no deadline are remaining

(Q) Min Candy problem

→ PS :- Given 'n' childrens and they individually have some ratings denoted by ratings[]
→ We need to distribute candies such that
(i) Each child at min gets 1 candy
(ii) children with higher ratings get more candies than their neighbours

→ Input :- ratings [1,3,2,1]

→ Output :- 7 (1,3,2,1).

TC SC
→ O(3N), O(2N)

☆ Brute force (i) Each child gets 1 candy

(iii) find min candies for left & right neighbours individually
(iv) max(left[i], right[i]) gives required value

☆ Better

(i) Remove right array from above approach

(ii) Compute right neighbour rating on the fly

↳ right = 1, curr = 1. sum = max(1, left[N-1]); → (N-1 ele).

```
if (rating[i] > rating[i+1]) {
    curr = right + 1;
    right = curr; → for next Iteration
}
sum += max(left[i], curr);
```

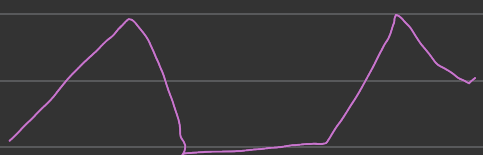
(iii) return sum;

TC :- O(2N)

SC :- O(N).

☆ Greedy (Slope approach)

(i) prepare a slope like trend from given candies

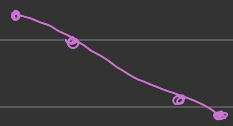


(ii) if slope is constant each gets 1 candy

```
candiesReq++; ind++;
```

(iii) if slope is increasing, greater element gets more candies, here we need to maintain Peak uPeak = 1.

```
while (ratings[i] > ratings[i-1]){
    uPeak++;
    candiesReq += uPeak; ind++;
}
```

(iii) If Slope is decreasing 

→ ideally values should be assigned from bottom to top

→ and if we get a newPeak > uPeak
candiesReq += (newPeak - uPeak);

→ But since we dont care about distribution we simulate this behaviour from bottom to top -

```
dPeak = 1;
while (ratings[i] < ratings[i-1]) {
    candiesReq += dPeak;
    dPeak++; i++;
}
```

(iv) return candiesReq.

→ Because we visit each element only once

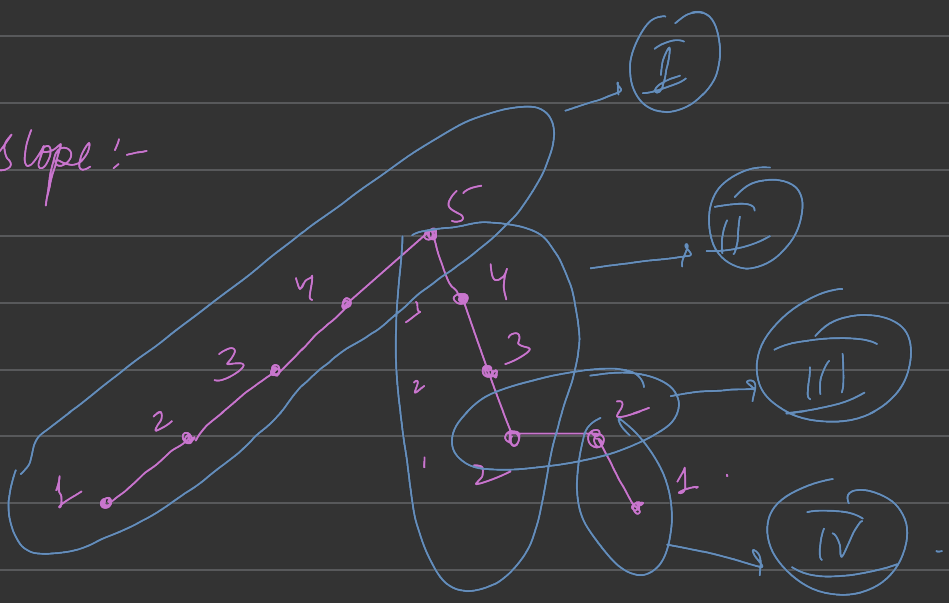
→ TC :- O(N)

→ SC :- O(1).

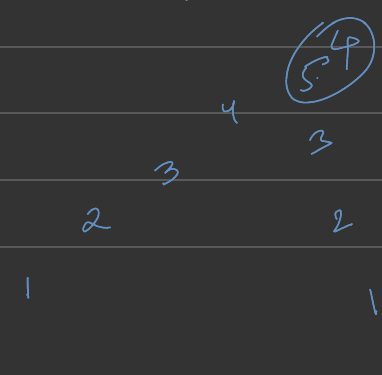
☆ Dry-run

Ex :- [1, 2, 3, 4, 5, 4, 3, 2, 2, 1] - Output :- 24.

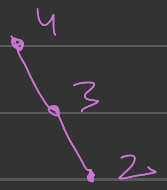
→ Slope :-



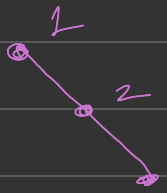
(I) → An increasing slope
- After assigning candies Peak = 5
candiesReq = 15



(II) → Decreasing slope



→ Assign candies



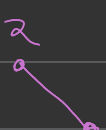
∴ dPeak = 4 ⇒ dPeak < uPeak ⇒ No change

CandiesReq = 15 + 6 = 21

(III) constant Slope.

CandiesReq = 22.

(IV) Downward slope



dPeak = 2. CandiesReq = 23.

dPeak > uPeak

↓

To satisfy neighbour greater condition. //

CandiesReq = 24

(Q) Insert Intervals

→ Ps:- Given a list of intervals [start_i, end_i] and a new Interval to be inserted
(i) Insert the new Interval to existing
(ii) Merge overlapping Intervals

→ Input :- [[1,2], [3,5], [6,7], [8,10], [12,16]]
new Interval : [4,8]

→ Output :- [[1,2], [3,10], [12,16]]

↓

([1,2], [3,5), [4,8], [6,7], [8,10], [12,16]]

merge

[[1,2], [3,8], [6,7], [8,10], [12,16]]

[[1,2], [3,8), [8,10], [12,16]]

merge

[[1,2], [3,10], [12,16]].

☆ Approach (DIY) :-

→ TC :- O(N)
SC :- O(N).

(i) find the unaffected left part

```
while ( ans[i][1] < newInterval[0] ).
```

(ii) find and Merge overlapping interval

```
while ( ans[i][0] <= newInt[1] ) {
  // when merging
      start = min of all starts
      end = max of all ends
                                  //
  newInt[0] = min (newInt[0], ans[i][0]);
  newInt[1] = max (newInt[1], ans[i][1]);
```

(iii) Add right unaffected part to the answer

(Q) Merge Intervals

→ PS:- Given intervals, merge them if they are overlapping.

→ Input :- [[1,3], [2,6], [8,10], [15,18]]

→ Output :- [[1,6], [8,10], [15,18]].

☆ Approach

TC :- O(N log N) + O(N).
SC :- O(N)

(i) Sort array if not sorted

(ii) follow a bit similar approach as that of above question

(iii) create a new Array :- mergeInterval

(iv) Check overlapping case :-

```
if (!isNotEmpty(mergedIntervals) &&
        mergedInterval.back()[1]
              >=
        intervals[i][0]   )

    // Candidate to merge
    mInt.back()[0] = min( , int[i][0]);

    mInt.back()[1] = max( , int[i][1]);
```

(v) else // No overlapping

```
    mInt.push_back({int[i][0], [i][1]});
```

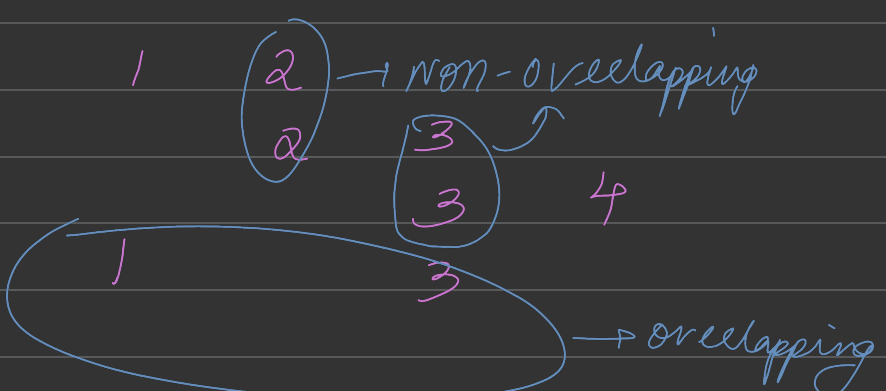
(Q) Non-Overlapping Intervals.

→ PS :- Given intervals, return the minimum number of intervals required to be removed in order to make the intervals non-overlapping.

Note :- Here [1,2], [2,3] are non-overlapping

→ Input :- [[1,2], [2,3], [3,4], [1,3]].

→ ouput :- 1 (remove [1,3]).



TC :- O(NlogN) + O(N)
SC :- O(1).

☆ Approach

(i) Sort the array based on end time

Why endTime ?

→ Removing min number of intervals is Equivalent to Keeping the max number of non-overlapping intervals

→ when we sort by endTime we keep max non-overlapping intervals

(ii) find Max No. of Non-overlapping intervals.

```
cTime = int[0][1];
of ( int[ind][0] >= cTime) {
        maxNP ++;
        cTime = int[i][1];
}.
```

(iii) ans = N - max Non Overlapping

☆ Alternate :-

```
lTime = int[i][1];

for (i = 1 → n) {

    if ( int[i][0] < lTime) {
        removals++;
    }
    else {
            lTime = int[i][1];
    }
```