GREEDY PROBLEMS

KAPIL YADAV

Greedy Problems

~ Kapil Yadav

17 May 2022 20:20

Index

Greedy Algorithm:

25 January 2022 19:35

LinkedIn

~ By Kapil Yadav

- While playing something, in some cases making a decision that looks right at that moment gives the best solution (Greedy), but in other cases it doesn't.
- Greedy algorithms work in stages. In each stage, a decision is made that is good at that moment, without thinking about the future. That means some Local best is chosen.

Do greedy always work?

Greedy algorithms will not always give optimize solutions.

Advantages of Greedy Methods:

It is straightforward.
It is easy to understand and implement.
Analysing the time complexity is generally easier than other techniques.

Disadvantages of Greedy:

We can not apply Greedy algorithms for many problems as it gives only local optimal solution, that does not always guaranteed to be optimal global solution.
There are some correctness issues related to the greedy algorithms, even if our algorithm is correct, it is difficult to prove its correctness,

Applications of Greedy Algorithm:

Some problems can be solve optimally using Greedy technique:

- Sorting algorithms : Selection Sort, Topological Sort
- Priority queues, heap sort
- Huffman coding Compression algorithm
- Minimum Spanning tree problem
    - Prim's algorithm
    - Kruskal's algorithm
- Single Source
- Shortest Path in weighted graph-Dijkstra's algorithm.
    - Dijkstra's algorithm
    - Bellman-Ford algorithm
- Coin Change Problem
- Fractional Knapsack problem
- Disjoint sets-UNION by size and UNION by height(or rank)
- Job scheduling algorithm

1. Activity Selection Problem (Greedy):

12 April 2022 18:23

LinkedIn

Problem 1. Activity Selection Problem

Problem Statement :

Given **N** activities with their start and finish day given in array **start[ ]** and **end[ ]**. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a given day.
Note : Duration of the activity includes both starting and ending day.

Solution :

Example 1: Consider the following 6 activities.

```
start[]  =  {0, 1, 3, 5, 8, 5}
end[]    =  {6, 2, 4, 7, 9, 9}
```

- We will take a pair which will store start, end of an activity. To store the pair of all the activities, we will use a vector of pairs.
- We need to choose the activities in such a way that, we can select maximum activities, So we will sort the end time of all the activities in increasing order and select the activity with minimum end time as first activity.
- Before selecting another activity, we need to compare its start time with the end time of previously selected activity.
  - If end time of previous selected activity is less than start time of current activity, we will increment the count (i.e. We can select the activity).
  - Else we will check for further activities and compare with them.

```
start[]  =  {0, 1, 3, 5, 8, 5}
end[]    =  {6, 2, 4, 7, 9, 9}
            1  2  3  4  5  6
```

(1, 2, 2)
(3, 4, 3)
(0, 6, 1)
(5, 7, 4)
(8, 9, 5)
(5, 9, 6)

→ we can perform 2nd activity from 1-2, then 3rd activity from 3-4, then, 4th activity from 5-7, & 5th activity from 8-9.
So, 4 will be our answer.

Code: for above approach :

TC = O(Nlogn) if input activities are not be sorted,
= O(n) if input activities are always sorted.
SC = O(N)

```cpp
int activitySelection(vector<int> start, vector<int> end, int n)
    {
        pair<int,int> p[n];
        for(int i=0;i<n;i++)
        {
            p[i].first = end[i];
            p[i].second = start[i];
        }
        sort(p,p+n);
        int count=1;
        int prev = 0;
        for(int j=1;j<n;j++)
        {
            if(p[j].second>p[prev].first)
           {  count++;
              prev=j;
           }
        }
       return count;
    }
```

NOTE :

- Above program will return the number of activities we can select, If we need to return, which activities are getting selected, declare a vector, and inside the if condition, insert p[j].second (i.e. Start of next activity).
- Sort function by default sorts the first parameter of the pair, if we want to Sort using other parameter, we can use comparator function.

```cpp
static bool comparator(pair<int, int> p1, pair<int ,int> p2) {
    if (p1.second < p2.second) return 1;
    return 0;
}
```

SAME PROBLEM: **N meetings in one room** (GFG)

2. Fractional Knapsack (Greedy)

12 April 2022 23:00

LinkedIn

Problem 2. Fractional Knapsack (Greedy)

Problem Statement :

Given there are **N** items and each object have *weight and profit* , we need to put these items in a knapsack of capacity **W** to get the *maximum* total Profit in the knapsack.

Note: Unlike 0/1 knapsack, you are allowed to break the item.

Input:	N = 3, W = 50 Profit[] = {60,100,120} weight[] = {10,20,30}
Output:	240.00

Solution : In 01 Knapsack, we either can take complete item or we cannot take it.
But in Fractional Knapsack, we can take break the items, and take some part of it.
Our Goal is to take weights in such a way that we get maximum total Profit.

We can solve this problem using different strategies:

1. We can arrange the items by their profit values.
   (Items having maximum profit will be selected first).
2. We can arrange the items by their weights
   (Items having minimum weight will be selected first).
3. We can calculate the profit/weight ratio for each item.
   Sort items based on their ratio.
   Take the item having highest ratio completely and add them to knapsack untill we cannot add the next item as whole.
   At last, add the next item as much fraction as we can.

Object	1	2	3	4	5	6	7
Profit	9	7	25	15	12	2	8
Weight	1	2	10	3	4	1	3

Weight of Knapsack is : 16 units.

Strategy 1: Select the item with maximum profit

Object	**Profit**	**Weights**	**Remaining weights**
3	25	10	16-10=6
4	15	4	6-4 = 2
5	12*(2/4) = 6	2	2-2 = 0

Total profit = **46**

Strategy 2: Select the item with minimum weights

Object	Profit	Weights	Remaining weights
1	9	1	16-1=15
6	2	1	15-1 = 14
2	7	2	14-2 = 12
4	15	3	12-3 = 9
7	8	3	9-3 = 6
5	12	4	6-4 = 2
3	25*(2/10) = 5	2	2-2=0

Total profit = **58**

Strategy 3: Select the item higher profit/weight ratio

Object	1	2	3	4	5	6	7
Profit	9	7	25	15	12	2	8
Weight	1	2	10	3	4	1	3
P/W	9	3.5	2.5	5	3	2	2.6667

Weight of Knapsack is : **16 units**.

Object	Profit	Weights	Remaining weights
1	9	1	16-1=15
4	15	3	15-3=12
2	7	2	12-2 = 10
5	12	4	10-4 = 6
7	8	3	6-3 = 3
3	25*(3/10) =7.5	3	3-3 = 0

Total profit = **58.5**

In all the above strategies, Strategy 3 gives more profit.

Code: for above approach (Strategy 3):

TC = O(Nlogn), SC = O(1)

```cpp
struct Item{
    int value;
    int weight;
};

class Solution
{   public:

    static bool com(Item a, Item b)
    {  double a1 = (double)a.value/a.weight;
       double a2 = (double)b.value/b.weight;
       return a1>a2;
    }

    double fractionalKnapsack(int capacity, Item arr[], int n)
    {   int curweight = 0;
        double result = 0.0;

        sort(arr,arr+n,com);

        for(int i = 0;i<n;i++)
        {   int remain = capacity - curweight;
            int curritem_weight = arr[i].weight;
            int curritem_profit = arr[i].value;
            if(arr[i].weight<remain)
            { curweight+=curritem_weight;
              result += curritem_profit;
            }
            else
            { result+= (curritem_profit/(double)curritem_weight)*
                       (double)remain;
              break;
            }
        }
        return result;
    }
};
```

3. Water Connection Problem

30 April 2022 19:13

LinkedIn

Problem 3. Water Connection Problem:

Problem Statement :

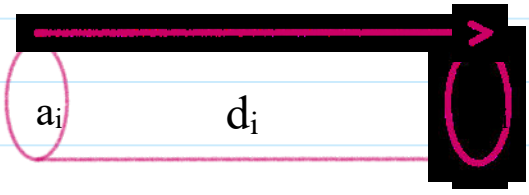
There are n houses and p water pipes in Geek Colony. Every house has at most one pipe going into it and at most one pipe going out of it. Geek needs to install pairs of tanks and taps in the colony according to the following guidelines.
1. Every house with one outgoing pipe but no incoming pipe gets a tank on its roof.
2. Every house with only one incoming and no outgoing pipe gets a tap.
The Geek council has proposed a network of pipes where connections are denoted by three input values: ai, bi, di denoting the pipe of diameter di from house ai to house bi.
Find a more efficient way for the construction of this network of pipes. Minimize the diameter of pipes wherever possible.

Input:	n=9 , p = 6 a[] = {7,5,4,2,9,3} b[] = {4,9,6,8,7,1} d[] = {98,72,10,22,17,66}
Output:	3 2 8 22 3 1 66 5 6 10

Solution : For a network of houses, There can be 3 cases:

1. A House have both incoming and outgoing pipe.
2. A House have only incoming pipe.
3. A House have only outcoming pipe.

- The House with only incoming pipe will gets a tap.
- The House with only outgoing pipe will gets a tank.



- We Need to arrange houses and pipes in efficient manner, such that we can place tank and tap in proper position of house.

- If we observe carefully, we can relate it with Directed graph problem, where we need to find connected components in the network of houses, where each component's starting node carries a tank and the ending node carries a tap.

For example: n=9 , p = 6

a[] = {7,5,4,2,9,3}

b[] = {4,9,6,8,7,1}

d[] = {98,72,10,22,17,66}

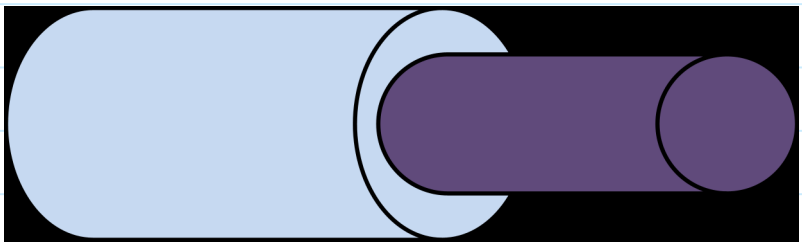
7 -> 4
5 -> 9
4 -> 6
2 -> 8
9 -> 7
3 -> 1

- If we draw a directed graph using the incoming and outgoing edges, our problem will be mostly solve:

5 -> 9 -> 7 -> 4->6
2-> 8
3-> 1

There will be three connected components, starting node in each component will store tank and ending node in each component will store tap.

Now, we have to minimize the diameter of overall network.
If the water is flowing through the pipes with different diameters, we know that the water will be flow with the same rate of the pipe with minimum diameter.
So, we will take minimum of the diameters of all connecting pipes in a component.



So, this will be nothing, but our answer.

STEPS:

- We will create two vectors for storing the information of incoming and outcoming edges of the Houses and initialize them with 0.
- If there is an incoming edge towards the node, we will mark IN as 1, and
- If there is an outgoing edge from the node, we will mark OUT as 1.
- We will create the graph, using the incoming vector and diameter of each vector.
- Now, we will run dfs from 1 to number of houses and update minweight of the diameter.
- And finally will store start, end and weight in the answer.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
IN	0	1	0	0	1	0	1	1	1	1
OUT	0	0	1	1	1	1	0	1	0	1

Code: for above approach :

TC = O(), SC = O()

```cpp
void dfs(int node, vector<pair<int,int>> graph[], int &minweight, int &end,vector<int>& vis)
    {
        vis[node] =1;
        for(auto edge: graph[node])
        {
            if(vis[edge.first]==0) {
                minweight = min(minweight, edge.second);
                end = edge.first;
                dfs(edge.first,graph,minweight,end,vis);
            }
        }
    }
    vector<vector<int>> solve(int numofhouses,int numofpipes,vector<int> a,vector<int> b,vector<int> d)
    {
        vector<int> vis(numofhouses+1,0);
        vector<vector<int>> ans;
        vector<int> in(numofhouses+1,0);
        vector<int> out(numofhouses+1,0);
        vector<pair<int,int>> graph[numofhouses+1];
        for(int i =0;i<numofpipes;i++)
        {
            out[a[i]] = 1;
            in[b[i]] =1;
            graph[a[i]].push_back({b[i],d[i]});
        }
        for(int i =1;i<=numofhouses;i++)
        {
            if(in[i]==0 && out[i]==1 && vis[i]==0)
            {
                int start = i;
                int end;
                int minweight = INT_MAX;
                dfs(i,graph,minweight,end,vis);
                ans.push_back({start,end,minweight});
            }
        }
        return ans;
    }
```

4. Minimum Platforms Problem

30 April 2022 19:18

LinkedIn

Problem 4. Minimum Platforms Problem

Problem Statement : Given arrival and departure times of all trains that reach a railway station. Find the minimum number of platforms required for the railway station so that no train is kept waiting.
Consider that all the trains arrive on the same day and leave on the same day. Arrival and departure time can never be the same for a train but we can have arrival time of one train equal to departure time of the other. At any given instance of time, same platform can not be used for both departure of a train and arrival of another train. In such cases, we need different platforms.

Input : n = 6
arr[] = { 0900, 0940, 0950, 1100, 1500, 1800 }
dep[] = { 0910, 1200, 1120, 1130, 1900, 2000 }
Output : 3

Solution : We will sort the arrival time and departure time in ascending order, after sorting both the arrays, arrival time and departure time for particular train might be at different index.

- In this problem, we are not bothering about the pair of arrival time and departure time of particular train. We just want to calculate number of platform for that particular time.

arr[] = { 0900, 0940, 0950, 1100, 1500, 1800 }
dep[] = { 0910, 1120, 1130, 1200, 1900, 2000 }

i	j	Arrival Time at index i	Departure time at index j	Number of current platforms required	No of Maximum Platforms Required
0	0	0900	0910	1	1
1	0	0940	0910	0	1
1	1	0940	1120	1	1
2	1	0950	1120	2	2
3	1	1100	1120	3	3
4	1	1500	1120	2	3
4	2	1500	1130	1	3
4	3	1500	1200	0	3
4	4	1500	1900	1	3
5	4	1800	1900	2	3

Minimum Number of Platforms required : **3**

NOTE: at i = 1, j = 0 and
at i =4, j = 3 , the number of current platforms required is 0, because before that time, only one platform was in used, and now the train departure time is less than train arrival time, Which means, the platform was idle during that time frame.

Code: for above approach :

TC = O(nlogn)+ O(nlogn) + O(n) + O(n),
SC = O(1)

```
int findPlatform(int arr[], int departure[], int n)
    {
        int result = 0;
        int platform = 0;

        sort(arr,arr+n);
        sort(departure,departure+n);

        int i =0;
        int j=0;

        while(i<n && j<n)
        {   if(arr[i]<=departure[j])
            {   platform++;
                i++;
            }
            else
            {   platform--;
                j++;
            }

           result=platform>result? platform: result;
        }
        return result;
    }
```

OR

We can increase the platform count by 1 and start comparing the arrival time of 2nd train, with the departure time of first train, and so on.

```
int findPlatform(int arr[], int departure[], int n)
    {
        int result = 1;
        int platform = 1;

        sort(arr,arr+n);
        sort(departure,departure+n);

        int i =1;
        int j=0;

        while(i<n && j<n)
        {   if(arr[i]<=departure[j])
            {   platform++;
                i++;
            }
            else
            {   platform--;
                j++;
            }

           result=platform>result? platform: result;
        }
        return result;
    }
```

5.Minimum number of coins

30 April 2022 19:19

LinkedIn

Problem 5. Minimum number of coins

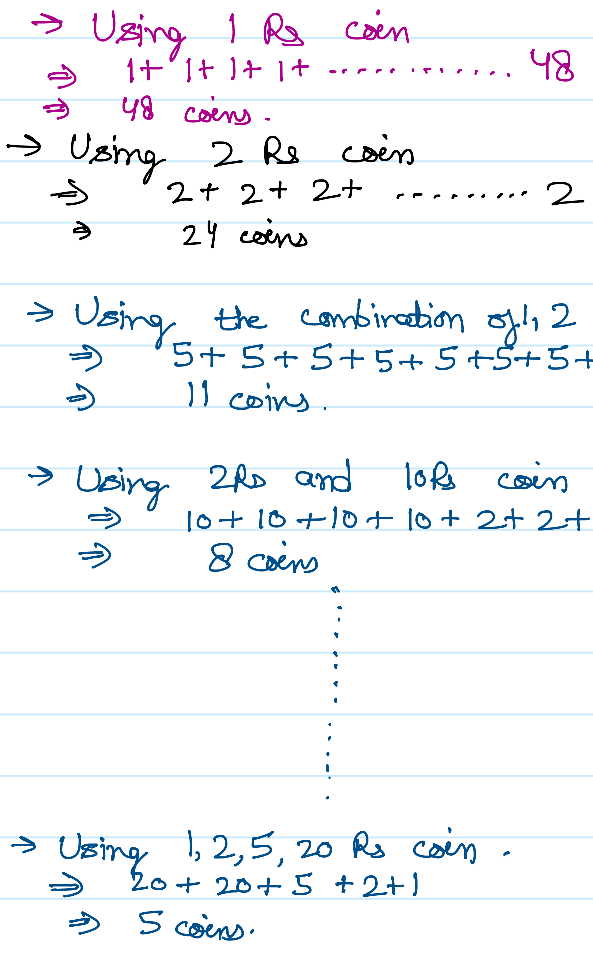
Problem Statement :

Given a value **V** and array **coins[]** of size **M**, the task is to make the change for **V** cents, given that you have an infinite supply of each of coins{$coins_1$, $coins_2$, ... $coins_m$} valued coins. Find the minimum number of coins to make the change. If not possible to make change then return -1.

Input :	V = 48, M = 7, coins[] = {1,2,5,10,20,50,100}
Output :	

Solution :

We have given a coin array, in which there are coins of coin[i] value.
We have given a particular amount.
Let say the amount is 48, To make the change of 48,



- There can be multiple ways to make the amount V, but we need to find out the minimum number of coins required to make the amount.
- In the above example, we need minimum 5 coins to make the amount 48.
- But How to get minimum coins?

Greedy Approach: :

- If coin array is not sorted, sorted it in ascending order.
- Let's be greedy, first add a coin with maximum value<amount.
- Subtract the value of coin[maxvalue] from amount .
  amount-=coin[[i]
- So, add that coin untill it's less than remaining amount, and so on.

Code: for above approach :

TC = O(V + Nlogn), if coin array is unsorted
= O(V), if coin array is sorted
SC = O(1)

```
int coinChange(vector<int>& coins, int amount) {
        //greedy approach
        int M = coins.size();
       sort(coins.begin(),coins.end());
        int count = 0;
        for(int i = M-1;i>=0;i--)
        {
            while(amount>=coins[i])
            {
                amount-=coins[i];
                count++;
            }
        }
        //fails for denominations, where sum of two coins> next greater coins  (3+5>6,5+6>9,6+9>11)
        // {1,3,5,6,9,11} amount = 15
        // minimum coins using greedy = 3 (11 + 3 + 1 = 15)
        // minimum coins using dp = 2 (9 + 6 = 15)
        //cout<<amount;
        if(amount>0)
            return -1;
        return count;
    }
```

NOTE: Greedy algorithms works only, when sum of 2 coins in a sorted order is not greater than the next greater coin.
For example: coin[] = {1,2,5,10,20,50,100,200,500,2000}

- 1+2<5
- 2+5<10
- 10+20<50
- 20+50<100
- 50+100<200
- 100+200<500
- 200+500<2000

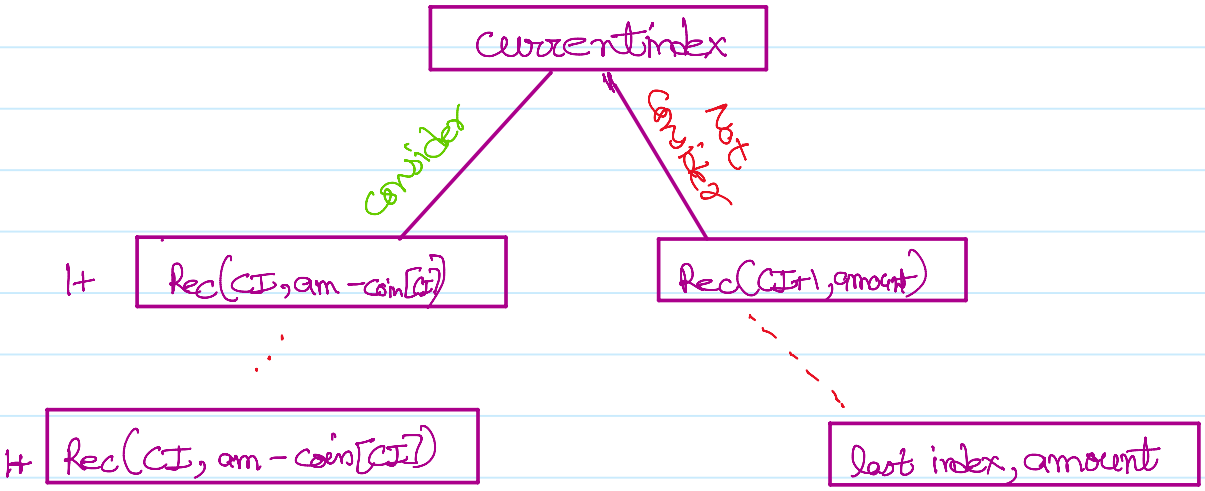
It will not work for the below example:
Coin[] = {1,3,5,6,9,11}, amount = 15
Using Greedy approach = 11 + 3 + 1 =15 (3 coins)

Coin[] = {1,3,5,6,9,11}, amount = 15
Using Greedy approach = 11 + 3 + 1 =15 (3 coins)
But Using DP = 9+6 = 15 (2 coins)

Because
5+6>9, 6+9>11 (sum of two coins are greater than the next coin)

Using Recursion :

- We will start iterating from 0th index, at every index, we will have two option, whether to consider the coin to make the amount, or whether to not consider it.
- If we consider it, then we will add 1 and the pass the same index for next recursive call, because there can be possibility that we can use it…like to make 15, we can use 1 rs coin for 15 times.
- If we are not considering it, we can increase the current index and call the recursive function for further indices. Since we are not considering it, so we will not add 1, to the count.
- We will find every possible solution and return the minimum out of them.
- Since. There can be many overlapping recursive calls, so it will take a lot of time.
- We can use Dynamic Programming, to store the overlapping recursive calls.



Dynamic Programming:

- We will use a 2 d vector to store the current index, amount value.

Code: for above approach :

TC = O(mV), where V is the amount and m is the number of available coins.
SC = O(1)

```cpp
int coinChange(vector<int>& coins, int amount) {
       vector<vector<int>>dp(coins.size()+1,vector<int>(amount+1,-1));
     int ans= minCoins(coins,0,amount,dp);
       if(ans==100005)
           return -1;
      else
          return ans;
   }

  int minCoins(vector<int>& coins,int currentindex, int amount,vector<vector<int>>&dp)
  {  int consider=100005;
      if(amount==0)
          return 0;
      if(currentindex>=coins.size())
      {
          return 100005;
      }
     if(dp[currentindex][amount]!=-1)
     {
         return dp[currentindex][amount];
     }

      if(coins[currentindex]<=amount)
      {   consider = 1 + minCoins(coins,currentindex,amount-coins[currentindex],dp);

      }
       int notconsider = minCoins(coins, currentindex+1, amount,dp);
       return dp[currentindex][amount]= min(consider,notconsider);
  }
```

 LinkedIn

6. Egyptian Fraction

30 April 2022 19:19

Problem 6. Egyptian Fraction:

Problem Statement : Find out the Egyptian Fraction of a given number.

A positive fraction number can be represented as the sum of unique unit fractions. A fraction is said to be a unit fraction if its numerator is one and the denominator some positive number. For example, 1/10 is said to be a unit fraction.

- Egyptian Fraction representation of 5/6 is 2/3 + 1/2.
- Egyptian Fraction representation of 8/15 is 1/3 + 1/5.

Solution :

Steps to follow:

1. Let the numerator and denominator of the fraction as **num** and **denom**.
2. Check the corner case when **denom** is equal to zero or num is equal to zero.
3. Check if denom is divisible by **num** then print 1/(**denom**/**num**).
4. If num is divisible by denom then print (**num**/**denom**).
5. Now compare **num** with **denom**, if **num** is greater than **denom**
   print **num**/**denom**
   recursively call the egyptianFraction() function : egyptianFraction(**num**%**denom**,**denom**);
6. If **denom** is greater than **num**
   print **denom**/**num** + 1
   recursively call the egyptianFraction() function : egyptianFraction(**num***x-**denom**, **denom***x);

Code: for above approach :

```cpp
#include <iostream>
using namespace std;

void egyptianFraction(int num, int denom){
    if (denom == 0 || num == 0)
    return;
    if(denom%num == 0){
        cout<<"1/"<<denom/num;
        return;
    }
    if(num%denom == 0){
        cout<<num/denom;
        return;
    }
    if (num>denom){
        cout<<num/denom<<" + ";
        egyptianFraction(num%denom, denom);
        return;
    }
    int x = denom/num+1;
    cout<<"1/"<<x<<" + ";
    egyptianFraction(num*x-denom, denom*x);
}
int main(){
    int num, denom;
    cout<<"Emter the numerator of Fraction: ";
    cin>>num;
    cout<<"Enter the denominator of Fraction: ";
    cin>>denom;
    cout<<"egyptian Fraction representation of "<<num<<"/"<<denom<<" is"<<endl;
    egyptianFraction(num,denom);
    return 0;
}

```

LinkedIn/kapilyadav22

7. Job Sequencing Problem

30 April 2022 19:20

LinkedIn/kapilyadav22

Problem 7. Job Sequencing Problem

Problem Statement :

Given a set of N jobs where each job_i has a deadline and profit associated with it.
Each job takes 1 unit of time to complete and only one job can be scheduled at a time. We earn the profit associated with job if and only if the job is completed by its deadline.
Find the number of jobs done and the maximum profit.

Input :	N = 6 Jobs = {(1,5,60) (2,4,20) (3,4,40) (4,1,10) (5,1,30) (6,3,15)}
Output :	5 165

Solution :

Job Id	Deadline	Profit
1	5	60
2	4	20
3	4	40
4	1	10
5	1	30
6	3	15

- We will get maximum overall profit, by choosing a good combination of jobs with maximum profit.
- So, we can think greedy, Let's sort the profit in decreasing order.

Job Id	Deadline	Profit
1	5	60
3	4	40
5	1	30
2	4	20
6	3	15
4	1	10

Strategy 1 : Let's select the job according to it's profit,

Job Id	Time Unit	Overall Profit
1	1	60
3	2	= 60+40 =100
5	3	=100+30

		= 130
2	4	= 130+ 20 = 150
6	It exceeds the deadline, we cannot select it	= 150
4	It exceeds the deadline, we cannot select it	= 150

Overall Profit is 150

Strategy 2 :

- We can perform any job till its deadline, so if we perform Job 1, in the last time unit of its deadline, i.e. At time =5 and so on, we can get better overall profit.

Job Id	Deadline	Overall Profit
1	5	60
3	4	= 60+40 =100
5	1	=100+30 = 130
2	Since, job 3 is already scheduled at time 4, We can check for time unit from 3 to 1. T = 3 is free, so schedule job 2 at T = 3.	= 130+ 20 = 150
6	Since, job 2 is already scheduled at time 3, We can check for time unit from 2 to 1. T = 2 is free, so schedule job 6 at T = 2.	= 150+15 = 165
4	Since, job 6 is already scheduled at time 1, We can not perform Job 4.	= 165

Overall Profit is 165

Code: for above approach :

TC = O(NlogN) + O(N*M),
SC = O(M)

Where N is the number of Jobs, and M is the maximum deadline that a process can have.

- First sort the jobs with profit in descending order, and then we need to find the maximum deadline of any job, it will take O(N).
- Then we have to take an array and initialize it with -1.
- Iterate every job and perform it as late as we can.
- If the time slot is not free, check for previous time slots till t = 1.
- If we don't get any of free time slot till time t =1, then that process will not get chance to perform.

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

    static bool com(Job a, Job b)
    { int a1 = a.profit;
        int a2 = b.profit;
        return a1>a2;
    }

    vector<int> JobScheduling(Job arr[], int n)
    {   vector<int> result;
        sort(arr,arr+n,com);
        // maximum deadline
        int maxi = arr[0].dead;
        for(int i=1;i<n;i++)
        { maxi = max(maxi,arr[i].dead);
        }

        //initializing the array with -1;
        vector<int> slot(maxi+1,-1);
        int noofjobs = 0;
        int totalprofit = 0;
        for(int i=0;i<n;i++)
        {
            for(int j=arr[i].dead;j>0;j--)
            {
                if(slot[j]==-1)
                {
                    slot[j]=arr[i].id;
                    noofjobs++;
                    //result.push_back(arr[i].id);
                    totalprofit+=arr[i].profit;
                    break;

                }
            }
        }
        result.push_back(noofjobs);
        result.push_back(totalprofit);
        return result;
    }
```

8. Lemonade change

30 April 2022 19:21

LinkedIn/kapilyadav22

Problem 8. Lemonade change

Problem Statement :

At a lemonade stand, each lemonade costs $5. Customers are standing in a queue to buy from you and order one at a time (in the order specified by bills). Each customer will only buy one lemonade and pay with either a $5, $10, or $20 bill. You must provide the correct change to each customer so that the net transaction is that the customer pays $5.

Note that you do not have any change in hand at first.

Given an integer array bills where bills[i] is the bill the i^{th} customer pays, return true if you can provide every customer with the correct change, or false otherwise.

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

Solution : We need to pay the correct change to every customer.

So, The possible cases can be:

- If a customer has paid 5 dollar, you don't need to return anything.
- If a customer has paid 10 dollar, then you have to pay him/her 5 dollar in return, or we can say we should have atleast one coin of 5 dollar.
- If a customer has paid 20, dollar then there can be 2 cases,
  a. We should have atleast one coin of 10 dollar and atleast one coin of 5 dollar or
  b. We should have atleast 3 coins of 5 dollar.
- If any of the above condition fails, we have to return false, else return true.

Code: for above approach :

TC = O(M),

where M is the number of customers.

SC = O(1),

we are taking an extra space of size 3, which can be consider as constant or we can take 3 integers to store the count of coins (5,10,15)

```cpp
bool lemonadeChange(vector<int>& bills) {
        vector<int> coin(3,0);
        // 0 index for 5 dollar
        // 1 index for 10 dollar
        // 2 index for 20 dollar
        int m = bills.size();

        for(int i=0;i<m;i++)
        { if(bills[i]==5)
            {  coin[0]++;
                continue;
            }
            if(bills[i]==10)
            {  if(coin[0]>0)
                {   coin[1]++;
                    coin[0]--;
                }
                else return false;
            }
            else if(bills[i]==20)
            {
                if((coin[1]<=0 && coin[0]<3) || (coin[1]>0 && coin[0]<=0))
                    return false;
                { coin[2]++;
                    if(coin[1])
                    { coin[1]--;
                      coin[0]--;
                    }
                    else coin[0]-=3;
                }
            }
        }

        return true;
    }
```

9. [Minimum Cost to move chips to the same position]

30 April 2022 19:21

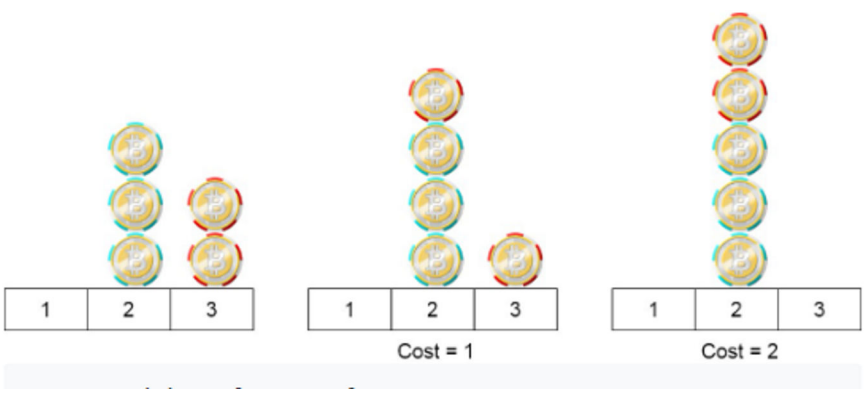
Problem 9. Minimum Cost to move chips to the same position

Problem Statement :

We have n chips, where the position of the ith chip is position[i].
We need to move all the chips to the same position. In one step, we can change the position of the ith chip from position[i] to:

- position[i] + 2 or position[i] - 2 with cost = 0.
- position[i] + 1 or position[i] - 1 with cost = 1.

Return the minimum cost needed to move all the chips to the same position.



Input :	position = [2,2,2,3,3]
Output :	2

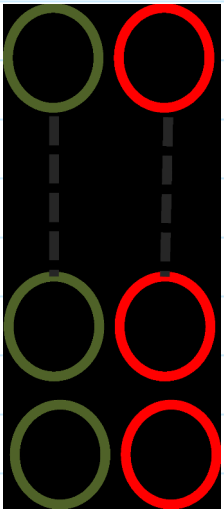
Solution :

We can move any chip 2 steps forward or backward with 0 cost.
We can move any chip 1 step forward or backward with cost =1.

- Let's say we have m chips at even indices and n chips at odd indices.

0	1	2	3	…	…	…	…	…	….

- We can combine all even chips at a particular index with cost 0, and all odd chips at any another particular index with cost 0.



0	1

- Now to move any chip one step, cost =1 is required, so we will move minimum of m chips or n chips to combine with other chips.

Let's say m = 5 at index = 0, and n = 4 at index = 1, so moving 4 chips required cost = 4 *1 = **4** .

Code: for above approach :

TC = O(n), SC = O(1)

```
1 int minCostToMoveChips(vector<int>& position) {
2      int  n = position.size();
3         int even = 0;
4         int odd = 0;
5
6         for(int i =0;i<n;i++)
7         {
8         if(((position[i]%2)))
9               odd++;
10        else
11             even++;
12        }
13
14     return min(odd,even);
15    }
```

LinkedIn/kapilyadav22

10. Task Scheduler

30 April 2022 19:22

LinkedIn/kapilyadav22

Problem 10. Task Scheduler

Problem Statement :

Given a characters array tasks, representing the tasks a CPU needs to do, where each letter represents a different task. Tasks could be done in any order. Each task is done in one unit of time. For each unit of time, the CPU could complete either one task or just be idle.
However, there is a non-negative integer n that represents the cooldown period between two same tasks (the same letter in the array), that is that there must be at least n units of time between any two same tasks.

Return the least number of units of times that the CPU will take to finish all the given tasks.

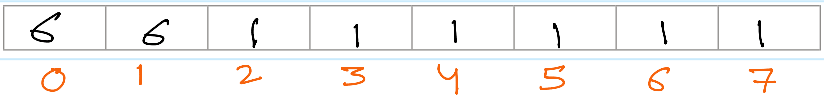
Input :	tasks = ["A","A","A","B","B","B"], n = 2
Output :	8

Solution : Method 1: Greedy Method

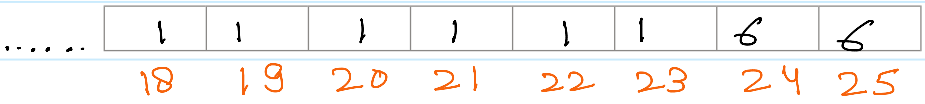
Example 1 :

Input :	tasks = ["A","A","A","A","A","A","B","B","B","B","B","B","C","D","E","F","G","H"], n = 2
Output :	18

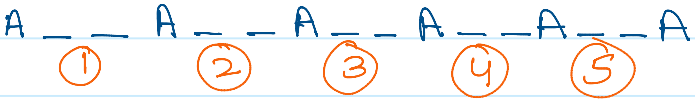
Take a frequency array of size 26, (I am showing index only which are non zero)



- After Sorting indices will be



- Schedule Task A :



- **Calculate Idle time :**
  As we can count above , the idle time is : 10 units,
  to convert it into a formula we can say, the idle timeslot is divided into A's frequency -1 =5,
  and n = 2, so total idle time will be = **(maxfrequency-1) * n**

- Schedule Task B :

Idle time = 10 - 5, because last b was scheduled after last A

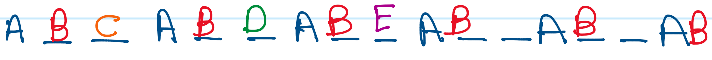
- Schedule Task C :

Idle time = 5-1 =4

- Schedule Task D :

Idle time = 4-1 = 3

- Schedule Task E :



Idle time = 3-1 = 2

- Schedule Task F :

Idle time = 2-1 = 1

- Schedule Task G :

Idle time = 1-1=0

- Schedule Task H :

Idle time = 0-1 = -1

- Since, we needed one slot to perform task "H", and there were no idle slot, that's why idle time became -1, so we will update it to 0, and our total minimum time will be **task.size = 18.**
- If in any case, the idle time might be in positive, we need to consider both task size + idle time.

Algorithm:

- Initialize an array of size 26 to store the frequency of each task.
- Calculate the frequency of each task.
- Sort the frequency array.
- After sorting, last index task will have maximum frequency, store it in maxfrequency.
- Idle time will be = **(maxfrequency-1) * n**
- Iterate the array from second last index and start assigning the task at idle slots, and keep

task.

- If in case idle time will be negative, update it with 0.
- Answer will be task size + idle time

Code: for above approach :

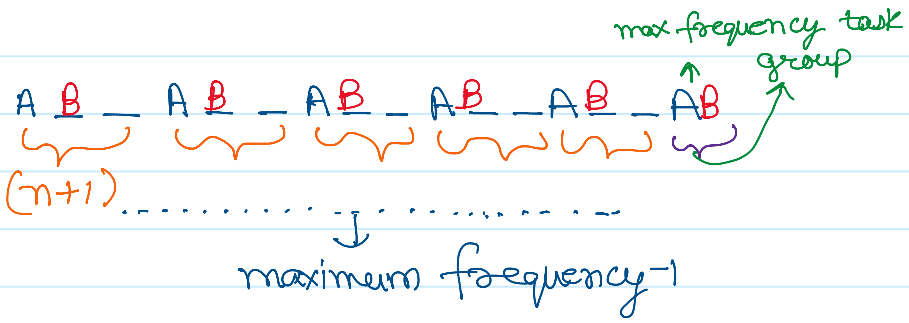
TC = O(NlogN), SC = O(26) = O(1)

```cpp
int leastInterval(vector<char>& tasks, int n) {
        vector<int> frequency(26, 0);
        for(auto task: tasks)
             frequency[task-'A']++;

        sort(frequency.begin(),frequency.end());

        int maxfrequency = frequency[25];
        int idletime = (maxfrequency - 1) * n;

        for(int i = 24; i >= 0; i--)
        {
            if(frequency[i]!=0)
            idletime -= min(frequency[i], maxfrequency-1);
        }
        idletime = max(0,idletime);
        return tasks.size() + idletime;
    }
```

Method 2: Mathematical Method

Example 2 :

Input :	tasks = ["A","A","A","A","A","A","B","B","B","B","B","B","C","D","E","F","G","H"], n = 2
Output :	18



$\Rightarrow$ ans = (n+1) x (maximum frequency - 1) + group size

Code: for above approach :

TC = O(N), SC = O(N)

```cpp
int leastInterval(vector<char>& tasks, int n) {
        unordered_map<char,int> frequency;

       int maxfrequency = 0;
        int groupsize = 0;

        for(auto it: tasks){
            frequency[it]++;
            maxfrequency  = max(maxfrequency ,frequency[it]);
        }

        for(auto it:frequency)
        {     if(it.second==maxfrequency)
                  groupsize++;
        }
        int answer = (maxfrequency - 1) * (n+1) + groupsize;
        int size = tasks.size();
        return max(answer,size);
    }
```

11. Car Pooling

30 April 2022 19:22

LinkedIn/kapilyadav22

Problem 11. Car Pooling

Problem Statement :

There is a car with capacity empty seats. The vehicle only drives east (i.e., it cannot turn around and drive west). You are given the integer capacity and an array trips where trips[i] = [numPassengers$_i$, from$_i$, to$_i$] indicates that the i^{th} trip has numPassengers$_i$ passengers and the locations to pick them up and drop them off are from$_i$ and to$_i$ respectively. The locations are given as the number of kilometers due east from the car's initial location.

Return true if it is possible to pick up and drop off all passengers for all the given trips, or false otherwise.

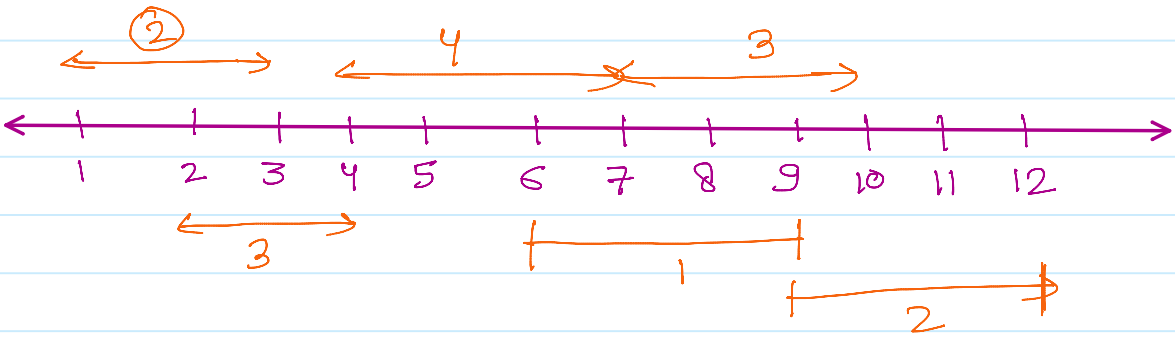
Input :	trips = [[2,1,3],[3,2,4],[4,4,7],[1,6,9],[3,7,10],[2,9,12]] capacity = 5
Output :	True

Solution :

We can Understand this problem by plotting a graph.

- First We have to calculate the maximum drop location, in above example it is 12.

Number of Passengers	Source	Destination
2	1	3
3	2	4
4	4	7
1	6	9
3	7	10
2	9	12



- Now take an vector **dist** of size same as maximum drop location + 1 (for 0 index) and initialize them with 0.
- Add current number of passengers at **Source** index
- Subtract current number of passengers at **Destination** index..

0	2	3	-2	-3+4	0	1	-4+3	0	2-1	-3	0	-2
0	1	2	3	4	5	6	7	8	9	10	11	12

- After adding and subtraction, the dist vector is :

0	2	3	-2	1	0	1	-1	0	1	-3	0	-2
0	1	2	3	4	5	6	7	8	9	10	11	12

- Now Take cumulative sum from first index till last index.
- Cumulative sum is showing the number of total passengers at any particular location.

0	2	5	3	4	4	5	4	4	5	2	2	0
0	1	2	3	4	5	6	7	8	9	10	11	12

- Now the question is, how does cumulative sum work in our problem.
- Basically what we are doing, we are adding the number of passengers at the index where we are picking them, and by doing cumulative sum, we will get the number of previous passengers in the car + the number of passengers sitting right now.
- We are subtracting the number of passengers at the index where they are dropping the car, so we are simply excluding them.

NOTE : If car capacity is 5 and number of passengers are 5 at location x, and they all drop the car at location y, we can pick other passengers from location y itself.

- Now we will again run a loop from starting location and check in our dist vector, if at any location, the number of passengers are greater than car capacity, we will return false, else after iterating complete dist vector, we will return true.

Code: for above approach :

TC = O(N) + O(N) + O(lastlocation) =O(N)
SC = O(1)
Since trip length in given question is 1000, so O(1000) = O(1) or O(Maximum Last Location).

```cpp
bool carPooling(vector<vector<int>>& trips, int capacity) {
        int size = trips.size();
        int maxdist =0;

        for(int i =0;i<size;i++)
        { if(trips[i][2]>maxdist)
                maxdist=trips[i][2];
        }

        vector<int> dist(maxdist+1,0);

        for(int i =0;i<size;i++)
        {    int from = trips[i][1];
             int to = trips[i][2];
             int currpass = trips[i][0];
         //we are ading in dist[from] and dist[to], because at distance d, some passengers can drop and some can sit in the car
         // [2,1,5][3,5,7] = at dist 5, 2 passengers are droping and 3 are sitting so, dist[5]=-2+3=1,
         // -2+2+3 = 3, 3 passengers at dist 5
             dist[from]+= currpass;
             dist[to]-= currpass;
        }
        //cumulative sum
        for(int i= 0;i<maxdist+1;i++)
        { if(i>0)
            dist[i]+=dist[i-1];
         if(dist[i]>capacity)
             return false;
        }
        return true;
    }
```

12. Divide Array in Sets of K Consecutive Numbers

30 April 2022 19:22

Problem 12. Divide Array in Sets of K Consecutive Numbers

Problem Statement :

Given an array of integers nums and a positive integer k, check whether it is possible to divide this array into sets of k consecutive numbers.

Return true if it is possible. Otherwise, return false.

Input :	nums = [1,2,3,3,4,4,5,6], k = 4
Output :	true

Solution : As we have to work with count/frequency, we can take map…

- Either we can take ordered map or (we can sort the array and use unordered map).
- We want to divide our array in the set of k numbers, that means the number of elements in the nums array should be divisible by k.
- If nums size is not divisible by k, then there is no way to divide the array in the set of K consecutive numbers, so will return false.
- Now, We will store the count of each element in ordered map.

⇒ curr = 1,
⇒ search for 2, 3, 4 in map, if not exist in map, return false.
⇒ decrease the count of curr, curr+1, curr+2, curr+3.
⇒ delete the element from the map, whose count is 0

nums[i]	Frequency
~~1~~	~~1~~ 0
~~2~~	~~1~~ 0
3	~~2~~ 1
4	~~2~~ 1
5	1
6	1

⇒ Map is not empty, enter while loop again:-
⇒ This time curr = 3.
⇒ Search for 4, 5, 6 in map,
⇒ decrease the count of 3, 4, 5, 6.
⇒ Count of 3, 4, 5, 6 is 0 now, so delete them from map.

nums[i]	Frequency
3	~~1~~ 0
4	~~1~~ 0
5	~~1~~ 0
6	~~1~~ 0

⇒ Now map is empty, so came out of while loop.

Code: for above approach :

TC = O(NlogN) : ordered map takes logn time to search one item, for N items, it will take NlogN

SC = O(N): For storing values in map.

```cpp
bool isPossibleDivide(vector<int>& nums, int k) {
        int m = nums.size();

         if((m%k)!=0)
             return false;

         map<int,int> mp;

         for(int i =0;i<m;i++)
         { mp[nums[i]]++;
         }

        while(!mp.empty())
         {
             int curr = (*mp.begin()).first;
             for (int i = 0; i < k; i++) {
                 if (mp.find(curr + i) == mp.end())
                     return false;

                 mp[curr + i]--;
                 if (mp[curr + i] == 0) mp.erase(mp.find(curr + i));
             }
         }
         return true;

    }
```

13. Jump Game

30 April 2022 19:22

LinkedIn/kapilyadav22

Problem 13 . Jump Game

Problem Statement :

You are given an integer array nums. You are initially positioned at the array's first index, and each element in the array represents your maximum jump length at that position.

Return true if you can reach the last index, or false otherwise.

Input :	nums = [1,2,2,0,3,1,1,2]
Output :	true

Solution : We need to reach to the last index of the nums array.
We cannot move further, if the value at any index is 0, that means, if we jump 0 step, if we can not reach to further positions.

There can be multiple test cases, some of them are:

1.

Input :	nums = [0,2,1,0,3,1,1,2]
Output :	False.

If the starting index value itself is 0, we can not jump further, so return false immediately.

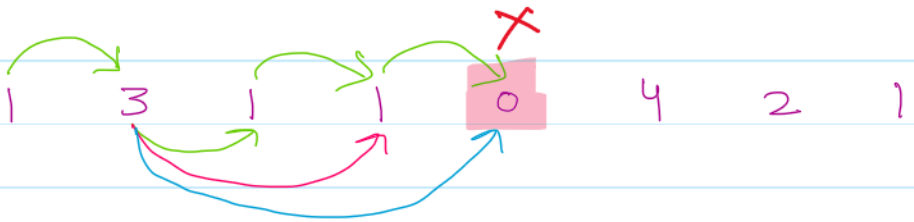
2.

Input :	nums = [0]
Output :	True

If we are at 0th index, and we want to reach 0th index, the number of jumps we need is 0, so we will return true.

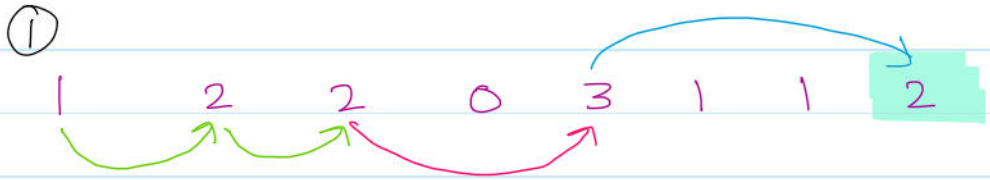
3.

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

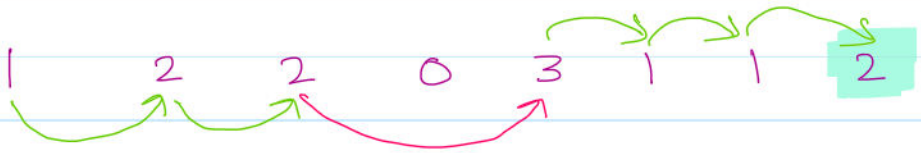


4.

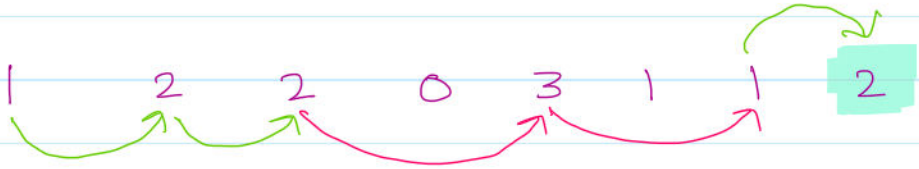
Input :	nums = [1,2,2,0,3,1,1,2]
Output :	true



2nd way

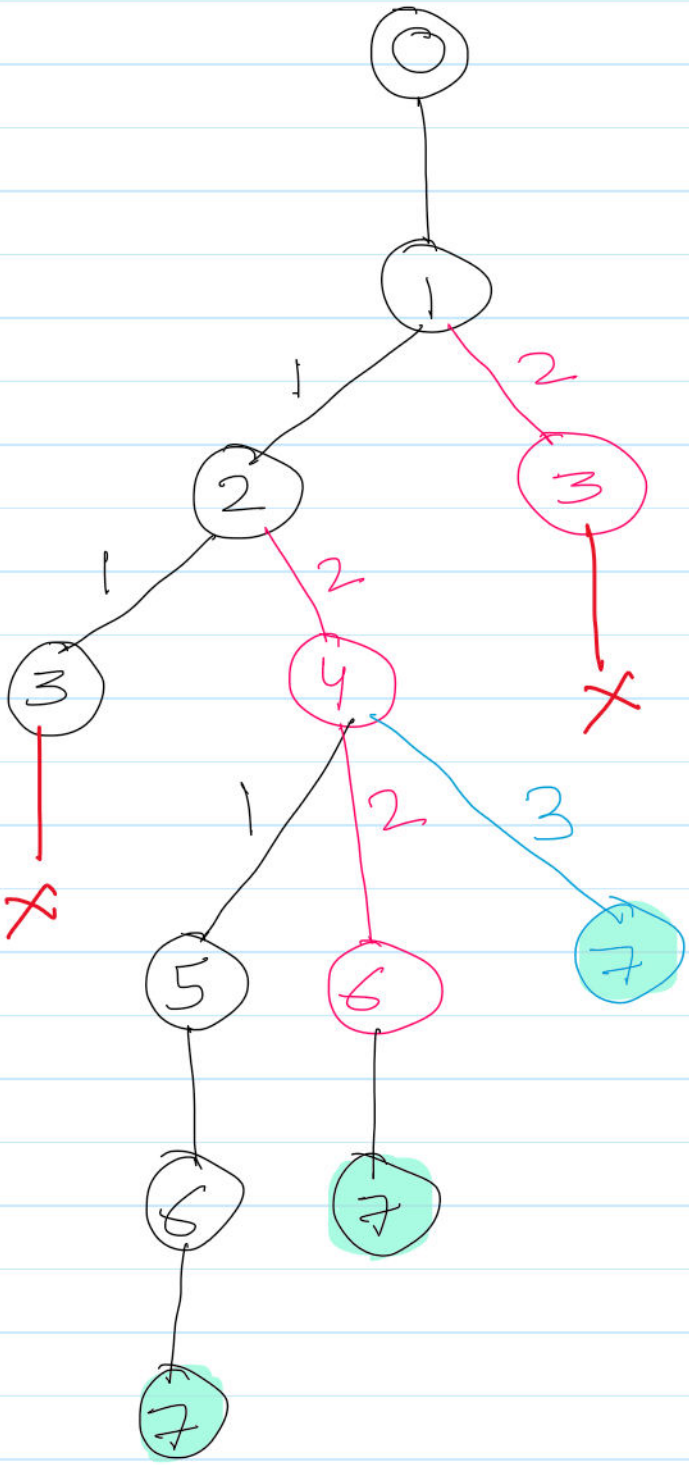


3rd way



- There can be mutiple paths to reach end, even if only one path exists, we will return true.
- If nums array doesn't have 0 value, then we can always reach to the end.

Recursive Approach:



- There can be overlapping recursive calls, so, we can use Dynamic programming to store the results.

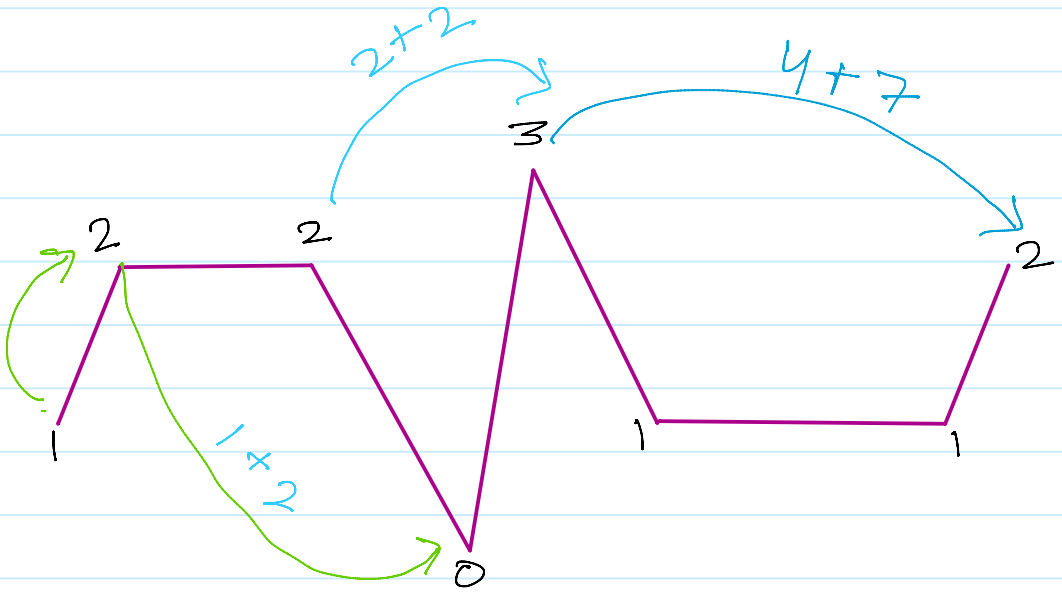
Code: for above approach :

TC = O(max(nums[i] * N) = O(N²),
SC = O(N)

```cpp
bool Jump(vector<int>& nums) {
        vector<int> v(nums.size()+1,-1);
       return canjump(0,nums,v);
    }

bool canjump(int currind, vector<int>& nums,vector<int>& v)
{
    if(currind>=nums.size()-1)
        return true;

   if(v[currind]!=-1)
        return v[currind];

     bool ans= false;
    for(int i=1;i<=nums[currind];i++)
    {

    bool jump = canjump(currind+i,nums,v);
      ans = ans || jump;
    if(ans)
        {  return v[currind]= ans;
        }
    }
     return v[currind]=false;
}
```

Greedy Approach: :

- Let's declare a variable name Reachability = 0.
- From the 0th index itself, we will keep on adding the nums[currentindex] with currentindex, and store the maximum in reachability.
- In any case, if reachability is less than our current index, that means there is no way to reach further, so will return false.



Reachability = ~~0~~ ~~1~~ ~~3~~ ~~4~~ 7.

Code: for above approach :

TC = O(N), SC = O(1)

```cpp
bool canJump(vector<int>& nums) {
        //by using Greedy
         int m = nums.size();
         int reachable=0;
         for(int i= 0;i<m;i++)
         {
             if(reachable<i)
                 return false;

             reachable = max(reachable,i+nums[i]);
         }
         return true;
    }
```

14. Jump Game 2

LinkedIn/kapilyadav22

30 April 2022 19:22

Problem 14. Jump Game 2

Problem Statement :

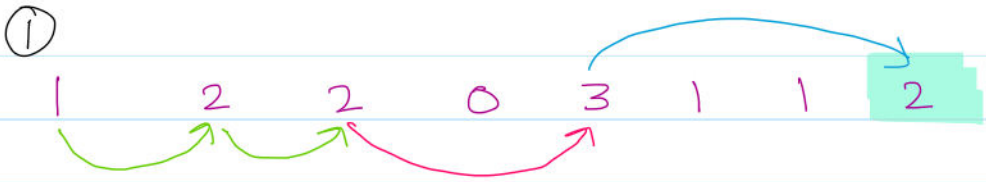
Given an array of non-negative integers nums, you are initially positioned at the first index of the array.
Each element in the array represents your maximum jump length at that position.
Your goal is to reach the last index in the minimum number of jumps.
You can assume that you can always reach the last index.

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

Solution : It is a variation of Jump game.

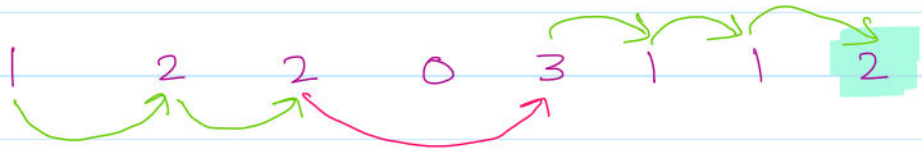
As we have seen in the previous question that there can be multiple paths to reach the end,
Here we need to find out the path in which we can reach to the end in minimum number of jumps.

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



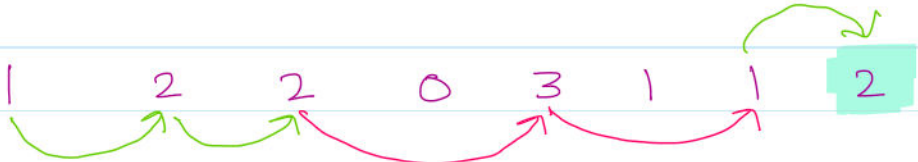
Number of Jumps required to reach the end = 4

2nd way



Number of Jumps required to reach the end = 6

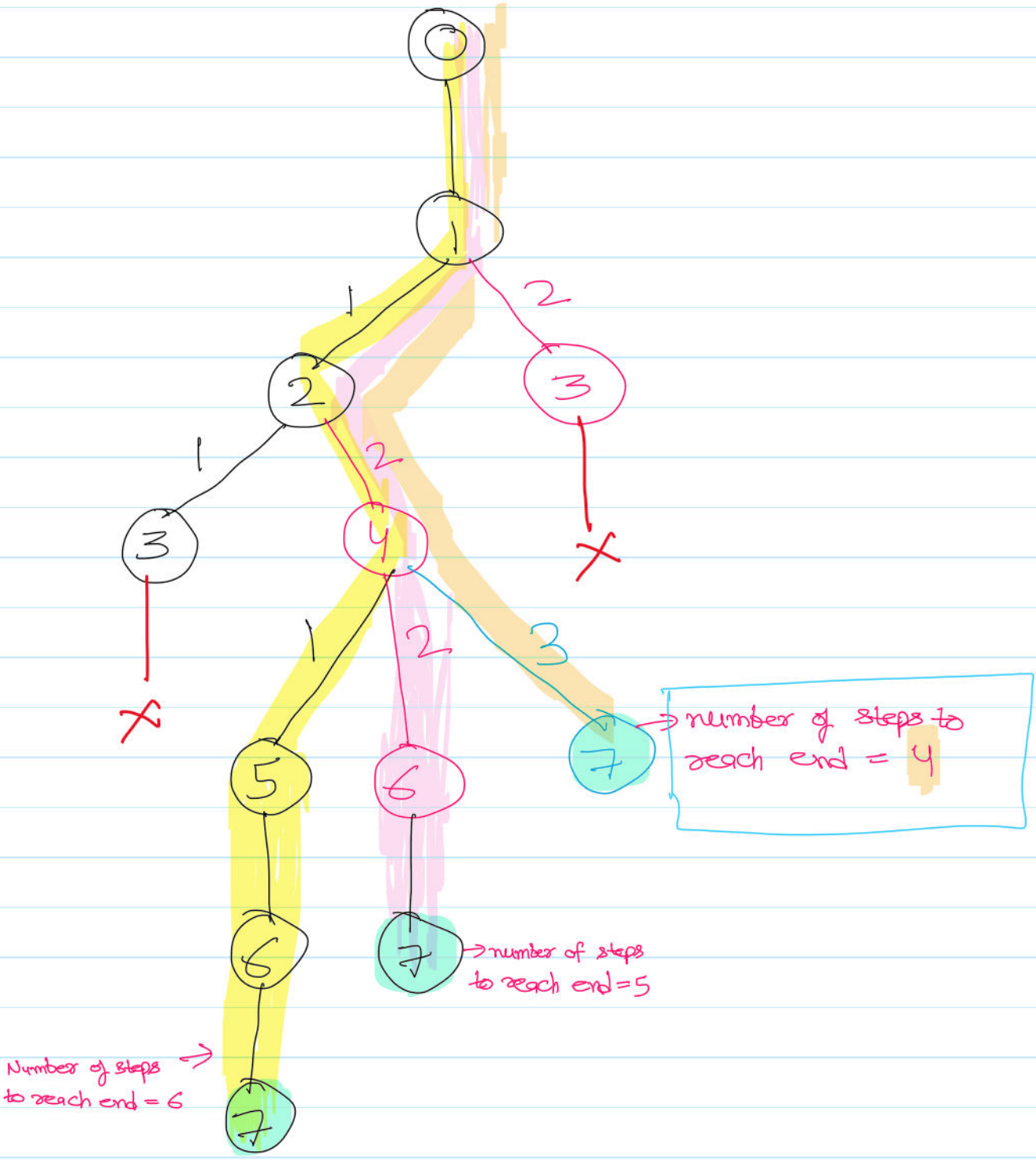
3rd way



Number of Jumps required to reach the end = 5

- So minimum number of jumps in this example is 4. Now, we can use multiple approaches to solve the problem.

Recursive Approach:



- There can be overlapping recursive calls, so, we can use Dynamic programming to store the results.

Code: for above approach :

TC = O(max(nums[i] * N) = O(N),

N represents the maximum length of nums. In this question, N <= 1000, so we are taking 10001 size additional storage.

```cpp
int jump(vector<int>& nums) {
        int memo[10001];
        memset(memo,-1,sizeof(memo));
        return minjump(nums,0,memo);
    }

    int minjump(vector<int>& nums, int currentindex, int memo[])
    {
        if(currentindex>=nums.size()-1)
        { return 0;
        }

         int currentjump= nums[currentindex];
         int currentkey = currentindex;
         int ans= 10001;

         if(memo[currentkey]!=-1)
            return memo[currentkey];

        for(int i=1;i<= currentjump;i++)
        {
            int tempans= 1+ minjump(nums,currentindex+i,memo);
            ans = min(ans, tempans);

        }
      memo[currentkey]= ans;
      return ans;
    }
```

Greedy Approach: :

- Declare a variable name result = 0 to store the number of min steps required to reach the end.
- Basically, we are going to use a BFS kind of approach.
- At every index, we have a choice to jump the steps from 1 to the nums[currentindex], so we will try to make windows of these jumps as farthest as possible.
- Take two variable Left and Right and initialize them with 0. Left is pointing to the starting point of window, and right is pointing to the ending point of the window.
- First we will calculate the farthest point in our window, by iterating left till right.
- After calculating the farthest point, we will shift our window by updating left and right point of window…
  left = right + 1
  right = farthest.
- The number of windows are nothing but the number of minimum steps we need to take to reach the end point of the array.

- Every time by shifting the window, we will update the result by 1 as one window shift means one step is taken.

i=0 1 2 2 0 3 1 1 2

left
right

⇒ farthest = max(0, 0+1) = 1
⇒ result = 0+1 = 1.
⇒ left = right + 1 = 1.
⇒ Right = farthest.

i=1 1 2 2 0 3 1 1 2

left
right

⇒ farthest = max(1, 1+2) = 3
⇒ result = 1+1 = 2
⇒ left = 2
⇒ right = 3

i=2 1 2 2 0 3 1 1 2

left right

⇒ farthest = max(3, 2+2) = 4

i=3 1 2 2 0 3 1 1 2

left
right

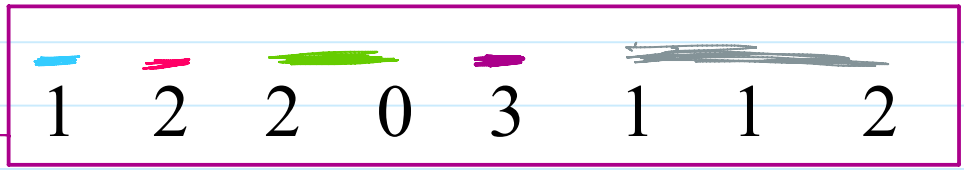
⇒ farthest = max(4, 3+0) = 4
⇒ left = 4
⇒ right = 4
⇒ result = 2+1 = 3

i=4 1 2 2 0 3 1 1 2

left
right

⇒ farthest = max(4, 4+3) = 7

⇒ farthest = max(4, 4+3) = 7
⇒ left = 5
⇒ right = 8 → if right >= n-1, exit loop.
⇒ result = 3+1 = 4



- Since we are starting from 0th index, we will not count it as a step, rest all windows will be count as 1 step.

Code: for above approach :

TC = O(N), SC = O(1)

```cpp
int jump(vector<int>& nums) {
     //Greedy Solution O(N)
     int result = 0;
     int left = 0;
     int right =0;
      int n = nums.size();

     while(right<n-1)
     {   int farthest = 0;
      for(int i =left;i<=right;i++)
      {
        farthest = max(farthest,i+nums[i]);
      }
         left = right+1;
         right = farthest;
         result+=1;
     }
     return result;
    }
```

15. Gas station

30 April 2022 19:22

LinkedIn/kapilyadav22

Problem 15. Gas Station

Problem Statement :

There are n gas stations along a circular route, where the amount of gas at the ith station is gas[i].
You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from the ith station to its next $(i + 1)^{th}$ station. You begin the journey with an empty tank at one of the gas stations.
Given two integer arrays gas and cost, return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1. If there exists a solution, it is guaranteed to be unique

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

Solution :
We need to find the index from where we can start, so that we can reach it again along a circular route, If there we can not find any index, we will return -1.

- First thing to do is, to check whether there can be any route possible or not, so simply, we can calculate the total gas we have and subtract the total cost we required to complete the route.
- If total gas< total cost, there will be no such route, so we will return -1.
- If total gas > = total cost, then there will surely a route possible and given in the question that, the solution will be unique, so we just have to find that index, from where we can start.

Brute Force Approach: :

We can check at every index, if we start from one index, is there any way to reach it again or not. Time complexity of this approach will be $O(N^2)$.

- But the input constraints are high, $N <= 10^5$, so we need to optimize our solution.

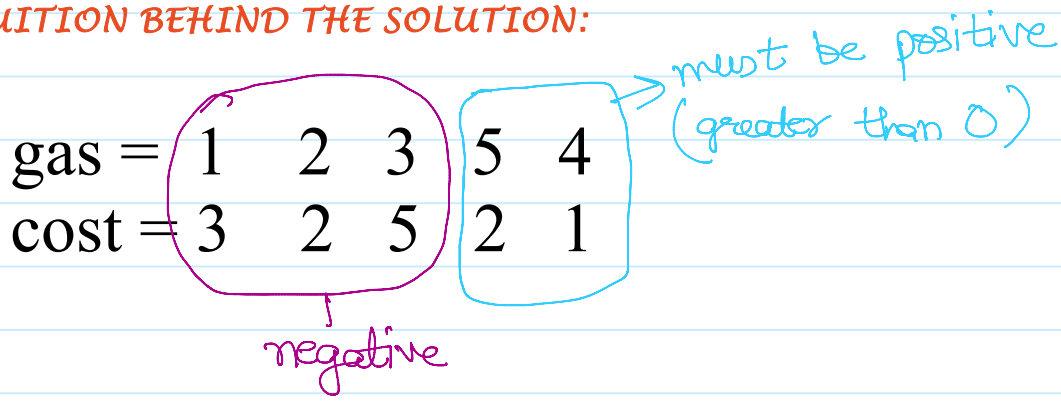
Greedy Approach: :

- Declare a variable currentgas to store the amount of gas till now, initialize it with 0.
- Take a variable start, which will use to store the index of the gas station from which we can travel around the circuit once in the clockwise direction, initialize it with 0.
- Iterate from first index, and calculate the current gas :

currentgas+=gas[i] - cost[i]

- If at any point currentgas<0, we will update our start to next index (i+1) and set currentgas =0.
- We will iterate the complete gas array.
- Basically we are trying to find the index, from where the value of currentgas till the last index will be in positive.

INTUITION BEHIND THE SOLUTION:



- As we have calculated above, there will any solution if and only if, the totalgas>=totalcost,
- So, if one part of the solution gives negative currentgas and (totalgas-totalcost) is positive, so another part of it should be positive.

  (Negative value + X) = positive ,

  so X should be positive.
- So, Our goal is to find the **first** index, from where we can partition it in positive currentgas and negative currentgas, in our example, till index 2 currentgas is in negative, from index 3 currentgas will be in postive till last index.
- As, there will be only unique solution, so we can say index 3 will be our answer.

Q. Why are we finding the first index of positive partition?

Ans. If we add all the positive values first, where gas[i]>cost[i], our currentgas will be in positive, and we only left with the stations, which will give negative currentgas, so we will be able to travel the whole circle in clockwise direction.

NOTE: **Take any problem, if there is a solution exists, we can partition it into negative and positive currentgas.**

Code: for above approach :

TC = O(N), SC = O(1)

```cpp
int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {

    int totalgas = 0;
    int totalcost=0;
    int currentgas =0;
    int start =0;

    int gs = gas.size();
    int cs = cost.size();
    for(int i =0;i<gs;i++)
    {    totalgas+=gas[i];
    }
    for(int i =0;i<cs;i++)
    {  totalcost+=cost[i];
    }
    if(totalcost>totalgas)
    return -1;

    for(int i =0;i<gs;i++)
    {
        currentgas+=(gas[i]-cost[i]);
        if(currentgas<0)
        {
            start = i+1;
            currentgas =0;
        }
    }
    return start;
}
```

16. Candy

30 April 2022 19:22

Problem 16. Candy

Problem Statement :

There are n children standing in a line. Each child is assigned a rating value given in the integer array ratings.

You are giving candies to these children subjected to the following requirements:

- Each child must have at least one candy.
- Children with a higher rating get more candies than their neighbours.

Return the minimum number of candies you need to have to distribute the candies to the children.

Input :	ratings = [1,0,2]
Output :	5

Solution :

We have to follow the rules:

- Each child must have at least one candy.
- Children with a higher rating get more candies than their neighbours.

- If there is only one child, we can give him only 1 candy.
- Otherwise, we can create vectors name **left and right** of ratings.size() and initially we will give 1 candy to each child.

Approach 1 :

In a Left vector, take second index as child(i= 1) and compare ranking of the current child with its left neighbour(i= 0),

- If **(ranking[currentchild] < ranking[currentchild - 1])**, then
  left[currentchild] = 1
- If **(ranking[currentchild] > ranking[currentchild - 1])**, then
  left[currentchild] = left[currentchild-1] + 1, and so on till ratings.size().

Similarly,
In a Right vector, take second last index as child(ratings.size()-2) and compare ranking of the current child with its Right neighbour(ratings.size()-3),

- If **(ranking[currentchild] < ranking[currentchild + 1])**, then
  left[currentchild] = 1
- If **(ranking[currentchild] > ranking[currentchild + 1])**, then
  lright[currentchild] = right[currentchild-1] + 1, and so on till child = 0.

After iterating both left and right array, we will calculate the total candies required to distribute:

Total candies += max(left,right)

Example 1 :

ratings = [4,2,3,1,6,2,1,2,4,6]
Output : 21

Ranking	4	2	3	1	6	2	1	2	4	6
Left	1	1	2	1	2	1	1	2	3	4
Right	2	1	2	1	3	2	1	1	1	1
Total Candies	2	1	2	1	3	2	1	2	3	4

max(left, right)

Code: for above approach :

TC = O(N), SC = O(N)

```cpp
int candy(vector<int>& ratings) {
        int n = ratings.size();

      vector<int> left(n, 1);
      vector<int> right(n,1);
        if(n == 1){
            return 1;
        }

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

        for(int i = n-2; i >= 0; i--){
            if(ratings[i] > ratings[i+1]){
                right[i] = right[i+1] + 1;
            }
        }

        int totalCandies = 0;
        for(int i = 0; i < n; i++){
            totalCandies += max(left[i],right[i]);
        }
       return totalCandies;
    }
```

Small Optimization : We don't need two left and right array, we can take only one array, and can update that array in second loop.

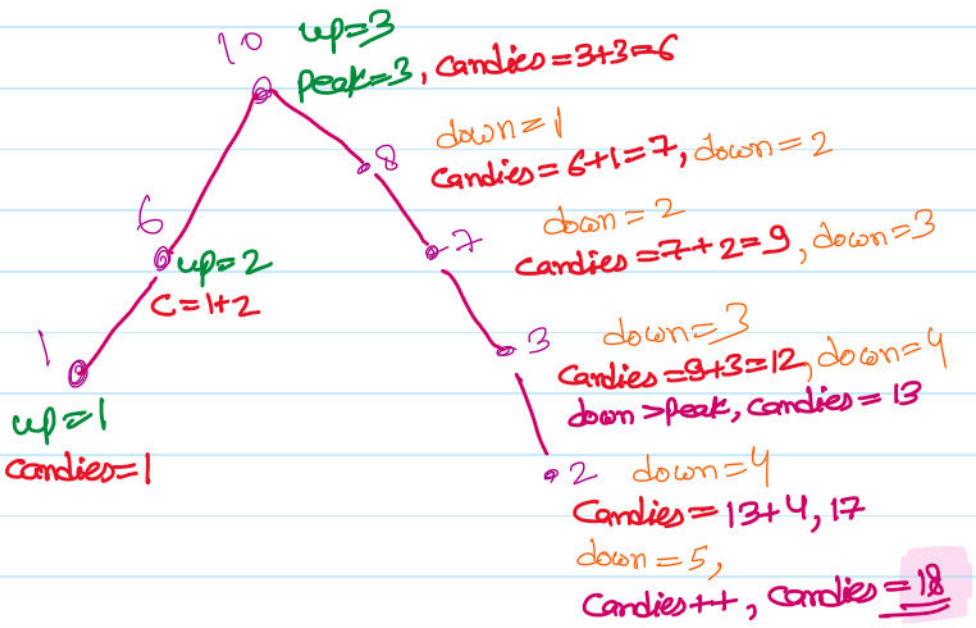
Code: for this approach :

TC = O(N), SC = O(N)

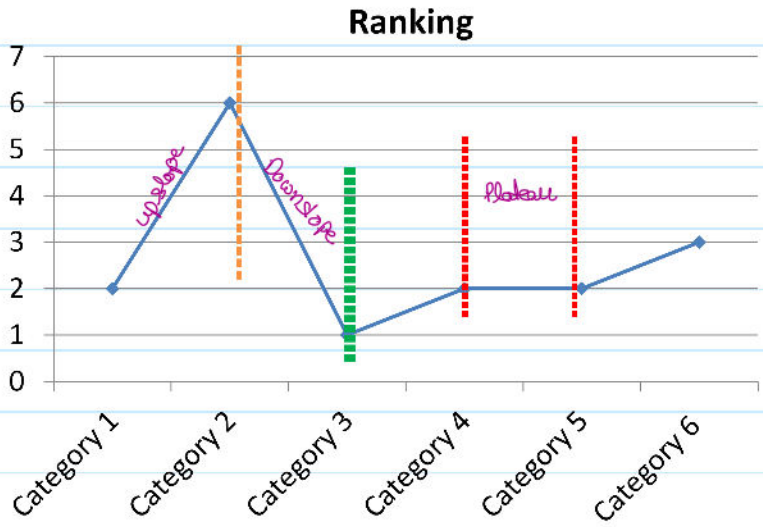
```cpp
int candy(vector<int>& ratings) {
        int n = ratings.size();
        vector<int> candies(n, 1);
        if(n == 1){
            return 1;
        }

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

        for(int i = n-2; i >= 0; i--){
            if(ratings[i] > ratings[i+1] && candies[i] <= candies[i+1]){
                candies[i] = candies[i+1] + 1;
           }
        }


        int totalCandies = 0;
        for(int i = 0; i < n; i++){
            totalCandies += candies[i];
        }
       return totalCandies;

    }
```

Approach 2 : We can solve this problem in TC = O(N), and SC = O(1).

Example : ratings = [1,6,10,8,7,3,2]
Output : 18



- If we draw the graph of any problem, It will look like a hill.
- If We divide the hill into parts, There can be 3 cases:
  a. The Hill can be go upwards (peek)
  b. The Hill can go downwards.
  c. The Hill can be at plateau.



- In case of upslope, i.e. ratings[i]> ratings[i-1], reset the down counter, increament the up variable and set last_peak = up, update the candies.

- In case of downslope, i.e. ratings[i]< ratings[i-1],update the candies first, increament the down variable and reset the up.
  NOTE : if the down> peak, that means slope goes down more than its peak, then, we will increament the candies by 1, because, we have to give atleast 1 candy to every child, so if we don't update the candies, we will not able to assign the candies according to the rules (specially rule 2)
- In case of plateau, we just have to increament the candy by 1.

rCode: for this approach :

TC = O(N), SC = O(1)

```cpp
int candy(vector<int>& ratings)
    {    if (ratings.size() == 0)
        return 0;
            int candies = 1;
            int up = 1, down = 1, last_peak = 0;
            for (int i = 1; i < ratings.size(); i++) {
                //increasing slope
                if (ratings[i - 1] < ratings[i]) {
                    //reset the down counter
                    down = 1;
                    up++;
                    last_peak = up;
                    candies+=up;
                }
                //decreasing slope
                else if(ratings[i - 1] > ratings[i])
                {   //reset the up counter
                    candies+=down;
                    up = 1;
                    down++;
                    // Compensate, if needed, for the lack of candies on the rising slope/plateau.
                    //[1,6,10,8,7,3,2]
                    if(down > last_peak)
                    candies++;
                }
                // plateau
                else  {
                    up = down = 1;
                    last_peak = 0;
                    candies++;
                }

            }

            return candies;
    }
```

LinkedIn/kapilyadav22

17. Remove k digits

30 April 2022 19:22

Problem 17 . Remove k digits

Problem Statement :

Given string num representing a non-negative integer num, and an integer k, return the smallest possible integer after removing k digits from num.

Input :	num = "1432219", k = 3
Output :	"1219"

Solution : Let's say there is a number 1546,

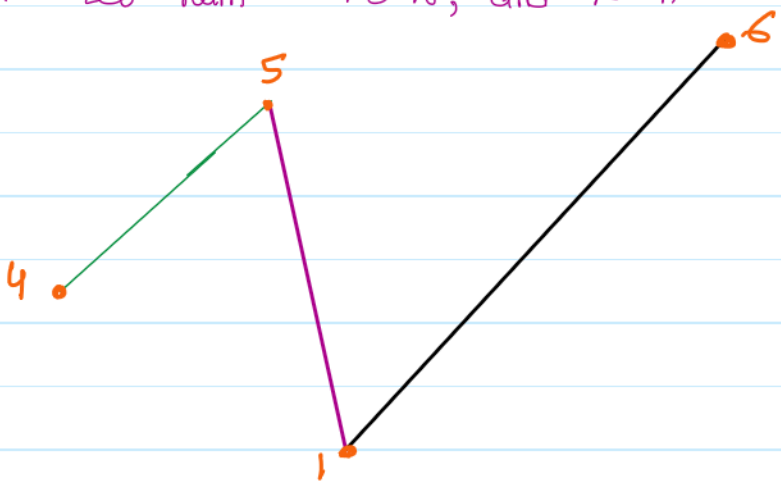
The digits of number can be arrange in different ways, It can be :

1456	4156	5146	6145
1546	4516	5416	6415
1645	4561	5461	6451

The total number of ways can be 4 * 3 * 2 * 1 = 4!. But out of all the ways,
The number which is in ascending order produces the least number possible.

- Since we can not interchange the number after removing k digits, but we can use this logic To find out the minimun number possible.

Ex → Let num = 4516, and K=1.



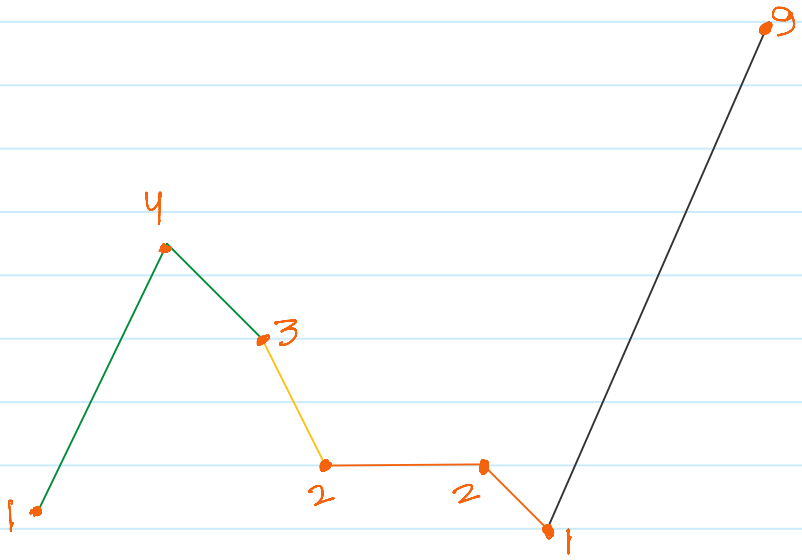
→ We have to remove 1 digits, so if we remove first peak element from the number, we will get minimum number.

→ Here 5 is the first peak element, so removing its will give – 416.

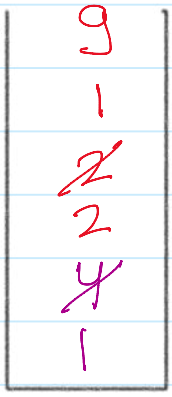
There can be various cases :
Case 1 : When number doesn't contain any 0.
Case 2 : When number contains 0.
Case 3 : When number is in non decreasing order (123456)

Case 1: When number doesn't contain any 0.

Example 1 : Num = 1432219 and k = 3



- We will use stack to store the elements, we will initially push first digit and from second digit onwards, we will start comparing,
- As soon as we get our first peak element, we will pop it untill the digits are in decreasing order or k = 0 or stack becomes empty.
- Other than above condition, insert the next digits in stack.
- After iterating the complete number, stack will contains the digits of minimum number in ascending order (smallest number at bottom, largest at top).
- Start popping the stack and store it in result.
- Reverse the result, since it contains number in descending order.



Case 2: When number contains any 0.

- Everything will be same as case 1, the only difference is:
  If we get trailing zeroes, we will not store it in stack. *(Line 18,19)*
- After removing K elements, If the number is completely zero

We will directly return 0. *(Line 27,28 in code)*

- Example : 100100100, K = 3 , after removing 3 digits, Number will be : 000000, so no need to store it.

Case 3: When number is in non decreasing order (123456)

- If number is in non decreasing order, we will simply store it stack, and start Popping the stack untill k =0. *(Line 23 to 25 in code)*

Code: for above approach :

TC = O(N), SC = O(N)

```
string removeKdigits(string num, int k) {
        if(num.size()==k)
        return "0";

        stack<char> st;
        st.push(num[0]);
        int n = num.size();
        string result = "";

       for(int i =1;i<n;i++)
        {   //40040, stack will be empty after one while loop, so add !st.empty() condition
           while(!st.empty() && k>0  && num[i]<st.top())
             {
               st.pop();
                k--;
             }
           //if stack is empty and num[i] is 0, that means, its a trialing zero, so we will not push it stack, rest others will push in stack.
           if(st.empty() && num[i]=='0')
             continue;
            st.push(num[i]);
        }
       // for non decreasing elements like 12345 or 55555 and k still is not equal to  0, we will pop larger elements from stack.
        while(!st.empty() && k--)
        { st.pop();
        }
        // like 1010010000, and k = 3, so after removing 3 ones, stack will be empty, that means, our number after removing k digits is 0, so return 0.
        if(st.empty())
            return "0";

        while(!st.empty()){
              result+=st.top();
            st.pop();
        }

         reverse(result.begin(),result.end());
         return result;
    }
```

Congrats,

You Have studied Greedy

Kapil Yadav