



Problems in Algorithms

Table of Contents

1. Greedy
 - 1.1 Grouping Elements Problem
 - 1.2 Minimize Error Problem
 - 1.3 Room Allocation(Selection Problem)
 - 1.4 Running Gold Problem(Nice modification to finding max problem)
 - 1.5 Greedily Knapsack Problem
2. Binary Search
 - 2.1 Introduction to Binary Search
 - 2.2 Sum of two k problem
 - 2.3 Deletion of two values problem
3. Dynamic Programming
 - 3.1 Coin Problem
 - 3.2 Longest Increasing subsequence Problem(in $O(n \log n)$ complexity)
 - 3.3 LCS for Permutations Problems
 - 3.4 Counting number of arrays problems(3 problems)
 - 3.5 Subsequence Sum Problem
4. Sub algorithms / Techniques
 - 4.1 Subalgorithm-1/Technique-1
 - 4.2 Subalgorithm-2/Technique-2
5. Some Implementation Tricks and Techniques



A **greedy** algorithm constructs a solution to the problem by always making a choice that looks the best at the moment. However, it is quite difficult in designing greedy algorithms is to find a greedy strategy that always produces an optimal solution to the problem. Sometimes it is quite tough to prove that the greedy solution actually works

Grouping elements

Problem Statement

Suppose we are given an array A which has n elements and our task is to make the minimum no of groups possible from the elements of an array in such a way that each group can take one or two elements from the array and the sum of elements in every group should not exceed x . We have to find the no of such groups.

Example
if the array is 7 2 3 9 and $x=10$
Minimum no of groups : 3
Groups can be {7,2},{3},{9}.
Notice that in no group sum of the element/elements exceed 10.

Solution

As we have to minimize the no of groups, the sum of values of elements in every group is as large as possible but should not exceed x . Suppose if we place an element p in a group say G , can you think of the best possible element (let us say q) which can be placed in the same group. As $p + q \leq x$, q should be as close as possible to $x - p$ but should not exceed it. So we have to pick every element and choose the best element which with it can pair up according to the condition described in the previous line. If there is no such element in the array which with it can pair up then the element will remain alone in the group. This is a greedy approach as we are making a choice that is the best at the moment.

Implementation

If we follow the naive approach, it will take $O(n^2)$ complexity but can be done in $O(n \log n)$ by making use of binary search. First, let us sort the array and start iterating from the first element. In every iteration, we can choose the best element with which the current element can pair up using binary search. Suppose if the current element is p then the lower bound of $x - p$ will be the best element to pair up as mentioned in the solution which can be done in $O(n \log n)$ complexity.

We can use one more approach using two pointer method. Suppose we have two pointers such that one pointer initially points to the starting element and another pointer points to the last element. let them be $ptr1$ and $ptr2$. if suppose $a[ptr1] + a[ptr2]$ is greater than x we are sure that $a[ptr2]$ can't make a pair with any element available as $a[ptr1]$ is the minimum among all elements and if $a[ptr1] + a[ptr2]$ is less than or equal to x then $a[ptr2]$ is the best element to pair up with $a[ptr1]$. So we can make a group in any case. This can be implemented as

Pseudo Code

```
sort(v.begin(),v.end());
long long int count=0,ptr1=0,ptr2=n-1;
while(ptr1<ptr2)
{
    if(v[ptr1]+v[ptr2]>k)
    {
        ptr2--;
        count++;
    }
    else
    {
        count++;
        ptr1++;
        ptr2--;
    }
}
if(ptr1==ptr2&&v[ptr1]<=k)
{
    count++;
}
cout<<count<<endl;
```

Minimize Error

Problem Statement

Suppose if we are given two arrays a and b of length n and let the elements be $a[1], a[2], \dots, a[n]$ and $b[1], b[2], \dots, b[n]$ respectively. We can do k_1 operations on array a and k_2 operations on the array b . In one operation, we have to choose one element of the array and increase or decrease it by 1. We can choose any element in both the arrays. Our task is to minimize $\sum |a[i] - b[i]|^2$, where $1 \leq i \leq n$.

Solution

We know that in every operation we can either increase any element or decrease any element. The primary observation is if $x^2 < y^2$ then obviously $(x - a)^2 - x^2 < (y - a)^2 - y^2$. So it is optimal to pick the largest element among $|a[1] - b[1]|, |a[2] - b[2]|, \dots, |a[n] - b[n]|$ and decrease it in order to minimize the whole sum in a greater extent, as all the terms are squared. We can decrease every term by 1 in every operation by either decreasing $a[i]$ by 1 or increasing $b[i]$ by 1. This is a greedy approach as we have to pick the maximum among all the terms and decrease it by 1 in every iteration until all the operations are exhausted. We have a total of $k_1 + k_2$ operations. But if suppose the sum becomes zero before all the operations are exhausted then we should alternatively decrease and increase the term. That is if all terms are zero, then we pick one term and alternately decrease and increase the term.

We can use heaps to solve the problem in $O(N \log N)$ complexity. As we have to pick the maximum among all the differences in every iteration, the heap ensures that the topmost element is the maximum element.

Pseudo Code

```
priority_queue<long long int> differences;
k1=k1+k2;
for(int i=0; i<n; i++)
{
    cin>>arr1[i];
}
for(int i=0; i<n; i++)
{
    cin>>arr2[i];
}
for(int i=0; i<n; i++)
{
    differences.push(abs(arr1[i]-arr2[i]));
}
while(k1-->0)
{
    long long int x=differences.top();
    // cout<<x<<endl;
    differences.pop();
    differences.push(abs(x-1));
}
while(!differences.empty())
{
    long long int x=differences.top();
    ans+=x*x;
    differences.pop();
}
cout<<ans<<endl;
```

Optimal Room Allocation Problem

Problem Statement

Suppose there is a large hotel and n customers who can reside in the hotel. Each customer wants to have a single room.

We are given each customer's arrival and departure day. Two customers can stay in the same room if the departure day of the first customer is earlier than the arrival day of the second customer.

We have to find the minimum number of rooms that are needed to accommodate all customers.

Solution

Suppose let us say some k rooms were occupied at a particular instant. So now the question is which room was to allocate next and which customer has to reside in that. Obviously, among the k rooms, the room occupied by the customer with minimum departure day is freed up first. Among all the customers who haven't arrived yet, the customer with a minimum arrival day is to be allocated next. Suppose if this minimum arrival day is greater than the minimum departure day among all customers who are currently residing in k rooms we allocate that room which was occupied by customer having minimum departure day or else we allocate a new room. This is a greedy approach as we are making a choice that is the best at the moment.

Implementation

We can sort all the customers according to their arrival time which ensures our second condition that the customer with minimum arrival day is to be allocated next.

Now, we can iterate through the customers while maintaining a minimum priority queue that stores the departure times of customers we've already processed. For each customer, we check to see if the minimum element in the priority queue is less than the arrival time of the new customer.

- If this is true, that means a room previously occupied has opened up, so we remove the minimum element from the priority queue and replace it with the new customer's departure time. The new customer will be allocated to the same room as the customer who departed.
- Otherwise, all the rooms are full, so we need to allocate another room for the customer and add it to the priority queue.

This ensures our first condition,

Pseudo Code

```
priority_queue<pair<int, int>> pq;
for (int i = 0; i < N; i++) {
    if (pq.empty()) {
        last_room++;
    }
```

```

        pq.push(make_pair(-v[i].first.second, last_room));
        ans[v[i].second] = last_room;
    }
    else {
        pair<int, int> minimum = pq.top();
        if (-minimum.first < v[i].first.first) {
            pq.pop();
            pq.push(make_pair(-v[i].first.second, minimum.second));
            ans[v[i].second] = minimum.second;
        }
        else {
            last_room++;
            pq.push(make_pair(-v[i].first.second, last_room));
            ans[v[i].second] = last_room;
        }
    }
    }
    rooms = max(rooms, pq.size());
}

```

Running Gold

This is a kind of problem which involves good algorithmic thinking

Problem Statement

There are n athletes, numbered from 1 to n , competing in the marathon, and all of them have taken part in 5 important marathons, numbered from 1 to 5, in the past. Suppose if we consider a matrix A with n rows and 5 columns such that $A[i][j]$ represents the rank of athlete i in marathon j .

An athlete x superior to athlete y if athlete x ranked better than athlete y in at least 3 past marathons, that is $A[x][j] < A[y][j]$ for at least 3 distinct values of j . An athlete is considered to get a gold medal if he is superior to all other athletes.

Our task is to determine the athlete who is likely to get the gold medal (that is, an athlete who is superior to all other athletes), or determine that there is no such athlete.

The constraint is to solve this problem in $O(n)$ complexity.

Solution

The approach to solve this problem is almost similar to the approach we follow in calculating maximum element in the array.

To find the maximum element in a array , we store the current best element in every iteration in a variable and return it after traversing through all the elements

```

int max=a[0];
for(int i=0;i<n;i++)
{
    if(a[i]>max)
    {
        max=a[i];
    }
}

```

```

    }
}

```

We follow the exact same approach in this problem as well. The primary observation is that we are looking for the athlete who is superior of all other athletes. Suppose for any two athletes x and y , there can be only two cases either x is superior to y or vice versa. If x is superior to y , then y can be eliminated from the competition as we can say that y is not likely to get a gold medal as it is already defeated to x (remember that athlete capable of winning a gold must be superior to all other athletes). Basically in every comparison one athlete is rejected. In finding the maximum element in an array as well, we reject one element in every iteration. This is a greedy approach as we are selecting the athlete who is best at that moment.

```

vector<long long int>v[n+5]; // v[i] represent a vector which stores the ranks of athlete i in
for(int i=0; i<n; i++)
{
    for(int j=0; j<5; j++)
    {
        long long int a;
        cin>>a;
        v[i].push_back(a);
    }
}
vector<long long int>best; // Current best athlete , similar to max in previous pseduo code
best=v[0];
long long int index=1;
for(int i=1; i<n; i++)
{
    long long int count=0;
    for(int j=0; j<5; j++)
    {
        if(best[j]<v[i][j])
        {
            count++;
        }
    }
    if(count<3)
    {
        best=v[i];
        index=i+1;
    }
}
}

```

After iterating the whole array we get the best athlete, where all other athletes are rejected. But now the point is to check whether the best athlete is capable of getting a gold medal or not. For this we again have to check with all other athletes.

```

for(int i=0; i<n; i++)
{
    if(index!=i)
    {
        long long int count=0;
        for(int j=0; j<5; j++)
        {
            if(best[j]<v[i][j])
            {
                count++;
            }
        }
    }
}

```

```

    }
    }
    if(count<3)
    {
        cout<<-1;
        return;
    }
}
cout<<index;

```

If suppose, some athlete is superior to our best athlete, then we are sure that there is no athlete in the given list who is likely to get the gold medal or else the best athlete is likely to get the gold medal.

The problem which looks like DP but actually greedy

Knapsack Problem with a slight modification

Knapsack Problem is very special, it gives a feeling of a greedy problem if we look at it for the first time but actually can be solved using dynamic programming.

This is the problem that gives a feeling of dp problem but can be solved using greedy

Problem Statement

Suppose we are given N items, whether each item has some cost and weight. Our task is to pick the elements with the maximum total cost with a total weight of at most C. But the constraint is that the weight of every item can be either 1 or 2.

This is a knapsack problem but here weight can take only 1 or 2.

Solution

At every step let us try to pick the element with maximum cost but minimum weight. There are two cases, either the final cost(W) can be even or odd

If C is even

Let us sort all the elements having weight 1 and elements having weight 2 in two different arrays.

So as the c is even in every step let us try to pick a weight of 2. So we can choose either one item having weight two or two items having weight 1. If we want to pick the item having weight 2, then it is optimal to pick the one with maximum cost(let it be x) and suppose if we choose to pick two items having weight 1 then we pick the maximum weighted two items(let the sum of these costs be y). So on a whole, it is optimal to pick the maximum of x and y in every step. This is a greedy approach as we are making a choice that is the best at the moment. If all the items of weight 1 are exhausted then we pick only the items having weight 2 and vice-versa.

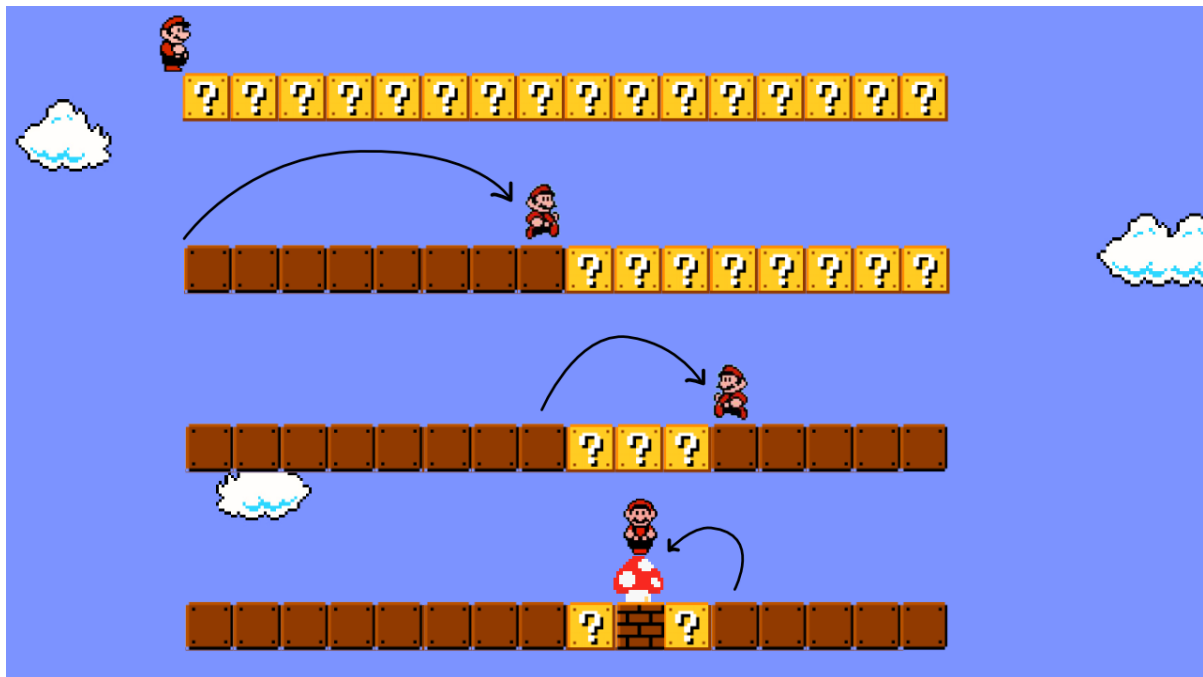
If C is odd

If C is odd, then there should be at least one item having weight 1. So in the first step, we pick the item with weight 1 having the maximum cost and then repeat the same as mentioned above.

Pseudo Code



Left as an exercise to the reader, this problem can be solved with either a heap or by sorting



Binary Search

Suppose if an array is sorted then if we are asked to find the position of some element in the array then it can be found by traversing the whole array which takes $O(n)$ complexity but can be efficiently solved using binary search in $O(n \log n)$ time complexity.

The usual way to implement binary search resembles looking for a word in a dictionary.

At each step, we check the middle element of the array. If the middle element is the target element, the search terminates. Otherwise, the search recursively continues to the left or right half of the region, depending on the value of the middle element. If the middle element is greater than the target element we search only the right half of the array(as we are sure that all the elements in the left half are smaller than the target element) and vice-versa. It can be proved that this kind of search will take $O(n \log n)$ complexity.

This can be implemented as

```
int a = 0, b = n-1;
while (a <= b)
{
    int k = (a+b)/2;
    if (array[k] == x) {
        // x found at index k
    }
    if (array[k] > x){
        b = k-1;
    }
    else
    {
        a = k+1;
    }
}
```

However, in C++ there is a standard library to find lower bound and upper bound for an array.

Suppose for vector we can do like



`lower_bound (v.begin() , v.end() , element)`



`upper_bound(v.begin() , v.end() , element)`

Lower bound function returns an iterator which points to the first occurrence of the specified element . If suppose the specified element is not present in the vector it returns a iterator pointing to the element which is just greater than the specified element and if there is no such element then it returns -1 .

Similarly Upper bound function returns an iterator which points to the last occurrence of the specified element. If suppose the specified element is not present in the vector it returns a iterator pointing to the element which is just greater than the specified element and if there is no such element then it returns -1

Sum of k values

Suppose if we are given an array which consists of n values. We have to determine the number of subsequence of size k with a sum S .

Case-1, $k = 2$

So if $k=2$, the problem can be restated as "Calculate the no of pairs with sum equal to k ".

This can be solved using binary search with time complexity $O(n \log n)$.

Let the array be a , so $a[i]$ represents the element that is present at i th position. Our task is to find no of pairs i, j such that $a[i] + a[j] = S$.

This can be also written as $A[i] = S - A[j]$. So basically we can traverse through all the elements of the array and for every element $A[i]$ we find the no of elements in the array with value $S - A[i]$. The naive approach is to traverse through all the elements in the array for every i and calculate the no of elements with value $k - A[i]$ and then sum them up. This will take a time complexity of $O(n^2)$.

As the final answer has nothing to do with the positions of elements we can sort the array initially which can be done in $O(n \log n)$ using merge sort.

So as the array is sorted we can find the lower bound of $S - A[i]$ (first occurrence of $S - A[i]$) and upper bound (last occurrence of $S - A[i]$) using binary search and **upper bound - lower bound + 1** gives the no of elements with a value $S - A[i]$. We have to do this for every value of i to get the total number of pairs with sum equal to S .



Note : We have to avoid the case where $A[i] = S - A[i]$ or $S = A[i]/2$ as it violates the constraint $k=2$.

Pseudo Code

```
sort(arr, arr + n);
for (int i = 0; i < n; i++)
{
    long long int first = lower_bound(arr, arr + n, sum - arr[i]) - arr;
    long long int last = upper_bound(arr, arr + n, sum - arr[i]) - arr - 1;
    if (first == n)
    {
        continue;
    }
    else if (arr[i] * 2 == find)
    {
        count = count + last - first;
    }
    else
    {
        count = count + last - first + 1;
    }
}
cout << count << endl;
```

Case-2, $k > 2$

If k is greater than 2, the problem can't be solved in $O(n \log n)$ however will take $O(n^2)$ complexity. This problem can be solved using dynamic programming which we will discuss in the next section

Delete two elements such that mean remains constant

This problem can be derived from the sum of k values problem with a slight modification.

Problem Statement

We are given an array A which consists of n elements and our task is to calculate no of pairs of positions $[i, j]$ such that if the elements of these positions are deleted, the mean of the array still remains the same.

Mean can be given as the sum of all elements present in the array divided by the no of elements (it can be fractional as well). Let the sum of all elements be S and no of elements be n . So mean can be given as S/n . Suppose let us remove two elements $A[i]$ and $A[j]$ from the array A . Mean of the new array after deletion of these two elements can be given as

$$\text{Mean} = (S - A[i] - A[j]) / (n - 2)$$

As Mean can be written as S/n .

This in turn results in $A[i] + A[j] = S/n$

The primary observation is that as $A[i]$ and $A[j]$ are integers, S/n should be an integer, otherwise there are no two elements which on deletion gives the same mean.

This problem is exactly similar to the previous problem (Sum of two values) which can be solved by binary search.

Pseudo Code

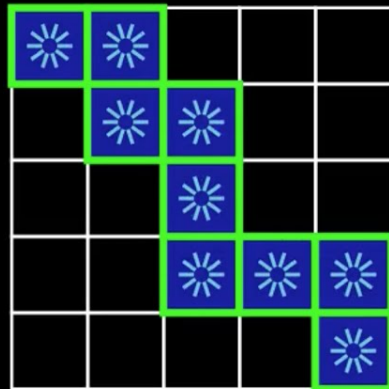
```
sum = sum * 2;
if (sum % n != 0)
{
    cout << 0 << endl;
    return;
}
long long int count = 0, find = sum / n;
sort(arr, arr + n);
for (int i = 0; i < n; i++)
{
    long long int first = lower_bound(arr, arr + n, find - arr[i]) - arr;
    long long int last = upper_bound(arr, arr + n, find - arr[i]) - arr - 1;
    if (first == last)
    {
        continue;
    }
    else if (arr[i] * 2 == find)
    {
        count = count + last - first;
    }
}
```

```

    }
    else
    {
        count = count + last - first + 1;
    }
}
cout << count / 2 << endl;

```

Dynamic Programming



Dynamic Programming

Dynamic Programming can be used to find an optimal solution if the problem can be divided into overlapping subproblems.

Dynamic Programming is an optimized technique of recursion that can be used to solve the problem in polynomial time rather than exponential time. It is efficient because it uses memorization and calculates the answer to each subproblem only once.

Generally, dynamic programming is used to find an optimal solution or for counting the no of possible solutions

There are two main steps in every Dynamic programming problem

- To find the recursive equation
- To find the start state

Coin Problem(DP from one previous state)

Problem Statement

We are given a set of coins consisting of different value coins and our task is to produce a sum of money x using the available coins in such a way that the number of coins is minimal.

If the available set of coins are $\{1,3,4\}$ and the sum of money we have to produce is 6 then the minimum no of coins are 2 which are two 3 rupee coins $\{3,3\}$

This can be solved efficiently using the technique of dynamic programming.

Suppose let the total money be x and we know the optimal answer for all the i , where $i < x$. Let $dp[i]$ be the minimum no of coins with which we can make money i .

We have to reach money x through any of these $i < x$. So as we know the answers for all i we can make use of it. Let the coins we have are $a[0], a[1], a[2], \dots, a[n-1]$, we can represent $dp[x]$ as



$$dp[x] = \min(dp[x - a[0]], dp[x - a[1]], dp[x - a[2]], \dots) + 1$$

This can be done because if we reach $x - a[i]$, then we can add the coin $a[i]$ to make a total of x (this last coin corresponds to $+1$ in the above equation). As we can reach any of $x - a[0], x - a[1], \dots, x - a[n-1]$ in the previous step it is optimal to pick minimum among these $dp[0]$ is the start state and its value is zero, as no coins are required to make zero money.

Pseudo Code

```
vector<long long int>dp(x+1,1e7);
dp[0]=0;
for(int i=0; i<=x; i++)
{
    for(int j=0; j<n; j++)
    {
        if(i>=arr[j])
        {
            dp[i]=min(dp[i],dp[i-arr[j]]+1);
        }
    }
}
if(dp[x]!=1e7)
{
    cout<<dp[x]<<endl;
}
else
{
    cout<<-1<<endl;
}
```

Longest Increasing Subsequence

Suppose if we are given an array and the task is to find out the longest increasing subsequence of the array. This is a maximum-length sequence of array elements that go from left to right, and each element in the sequence is larger than the previous element.

Example:

Let the array be [2,4,6,1,9], then the longest increasing subsequence is {2,4,6,9}

Solution in $O(n \log n)$ time complexity

Let us consider a new array dp which keeps on updating on every iteration. We update dp array in such a way that it is always increasing. The no of elements in dp traversing up to $a[i]$ gives the length of longest increasing subsequence till index i . But note that this dp array do not contain the longest increasing subsequence but its length is equal to the length of the longest increasing subsequence at any point.



$dp[x]$ = minimum ending value of an increasing subsequence of length $x+1$, using the elements considered so far.

We add elements one by one from left to right. Say we want to add a new value v . For this to be part of an increasing subsequence, the previous value in the subsequence must be lower than v . We want to find the rightmost element in the dp array (as the position corresponds to the length of the subsequence), with a value less than v . Say it is at position x . We can put v at a position $x+1$, replacing the previous element which is currently present (since we have an increasing subsequence of length $x+1$, which ends on v). This ensures that in every iteration we get the longest increasing subsequence up to the corresponding element. As dp array is always increasing we can find the position of any element at which the element is to be placed using binary search.

```
for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    auto it = lower_bound(dp.begin(), dp.end(), x);
    if (it == dp.end()) {
        dp.push_back(x);
    } else {
        *it = x;
    }
}
cout << dp.size() << endl;
```

Longest Common Subsequence

This problem is derived from the longest increasing subsequence problem

Problem Statement

We are given two permutations of numbers from 1 to n and the task is to output the longest common subsequence of these two permutations

Example

```
Permutation-1 : 1 2 3 4
Permutation-2 : 2 3 4 1
```

In this example, we are provided with two permutations of numbers from 1 to 4 and the longest common subsequence is 3,2,1 as 3,2,1 occurs in the same order in both the permutations and it is the longest among all the common subsequences.

This problem can be framed into the longest increasing subsequence problem with a slight modification. The primary observation is that every number from 1 to n should be present exactly once in both permutations. Let us form a new array of length n in which the i th element represents the position of the i th element of the first array in the second array. If the given permutations can be represented as arrays a and b respectively, let the new array be c .



$c[i] = \text{position of } a[i] \text{ in array } b, \text{ for every } i \leq n$

If we observe carefully the length of the **longest increasing subsequence** of the newly formed array is the **longest common subsequence** of arrays a and b .

Proof

Let us consider an increasing subsequence of newly created array c . If suppose let there be two elements x and y in the respective order in this subsequence. Let the positions of x and y be i and j respectively in the array c . This implies $y \geq x$ and $j \geq i$ as the subsequence is increasing. Here x represents the position of $a[i]$ in b and similarly, y represents the position of $a[j]$ in b .

As $y \geq x$, $a[i]$ and $a[j]$ should appear in the respective order in array b as well. This implies that $a[i]$ and $a[j]$ form a common subsequence.

On the other hand, let us consider any decreasing subsequence of array c and let x and y appear in the respective order in this subsequence. If the positions of x and y are i and j respectively in the array c then $y \leq x$ and $j \geq i$. As per the rule array c was being framed, x represents the position of $a[i]$ in b , and y represents the position of $a[j]$ in b . As $y \leq x$, $a[j]$ should appear before $a[i]$ in the array b which implies that $a[i]$ and $a[j]$ do not form a common subsequence as the order they appear is different in the both the arrays

From the above discussion, we can infer that for every common subsequence in arrays a and b there should exist an increasing subsequence in array c . Similarly, for every increasing subsequence in array c there should be a common subsequence in arrays a and b . So, the length

of the longest common subsequence in array c is equal to the length of common subsequence in arrays a and b.

Pseudo Code

```
for(int i=0; i<n; i++)
{
    cin>>a[i];
    m1[a[i]]=i;
}
for(int i=0; i<n; i++)
{
    cin>>b[i];
    m2[b[i]]=i;
}
vector<long long int>v,dp;
for(int i=0; i<n; i++)
{
    v.push_back(m2[a[i]]);
}
for (int i = 0; i < n; i++)
{
    auto it = lower_bound(dp.begin(), dp.end(), v[i]);
    if (it == dp.end())
    {
        dp.push_back(v[i]);
    }
    else
    {
        *it = v[i];
    }
}
cout << dp.size() << endl;
```

Problems to count no of arrays with a given condition

Total number of arrays of length n where every element should be in between low and high with total sum even

Problem Statement

Given three numbers n, low, and high our task is to determine the no of arrays of length n where all elements should lie between low and high and total sum is even.

Example
Suppose n=3, low=1 and high=2
The arrays are {1,1,2}, {1,2,1}, {2,1,1}, {2,2,2}

Solution

Let us try to build a recursive approach. If we know the no of such arrays consisting of $i - 1$ elements, can we get the answer for i elements. Every element of the array can be either odd or even. If the i th element is even, then the sum of all elements up to the $(i - 1)$ th element should be even and if i th element is odd, then sum of all elements up to the $(i - 1)$ th element should be odd to make the total sum even. If suppose the sum of all elements up to $(i - 1)$ th element is even then we can choose any even number which lies between low and high and the similar job for choosing odd element as well.

Let us say $f[i - 1][0]$ and $f[i - 1][1]$ represents the no of arrays where every element lies between low and high and total sum even and odd respectively and let count1 and count2 be the no of even numbers and odd numbers between low and high respectively. So we can say



$$f[i][0] = f[i - 1][0] * (\text{count1}) + f[i - 1][1] * (\text{count2})$$

As per the approach mentioned above.

Similarly we have to calculate $f[i][1]$ which can be calculated as



$$f[i][1] = f[i - 1][0] * (\text{count2}) + f[i - 1][1] * (\text{count1})$$

We can say that $f[1][0]$ is equal to count1 and $f[1][1]$ is equal to count2 as if there is only one element and if it should be even then it can take any even number between low and high which is count1 and if it should be odd then it can take any odd number between low and high which is count2.

Pseudo Code

```
for(int i=low; i<=high; i++)
{
    if(i%2==0)
    {
        count_1++;
    }
    else
    {
        count_2++;
    }
}
f[1][0]=count_1;
f[1][1]=count_2;
for(int i=2; i<=n; i++)
{
    f[i][0]= (f[i-1][0]*count_1) + (f[i-1][1]*count_2);
    f[i][1]= (f[i-1][0]*count_2) + (f[i-1][1]*count_1);
}
cout<<f[n][0]<<endl;
```

Total number of increasing arrays of length n ending with integer m

Problem Statement

Given two integers n and m and our task is to determine the no of arrays with length n and ending m.

Example

Suppose n=3 and m=2

Non-decreasing arrays of length 3 ending with 2 are {1,1,2}, {1,2,2}, {2,2,2}

So the answer is 3

Solution

Let $f[n][m]$ be the no of arrays of length n ending with m.

We can say that this number is finite as the array is non-decreasing. We can clearly state that every integer in the array should be less than m.

Let us consider the last but one element which is $a[n-1]$ if we consider 1 based indexing.

This element can take any integer between 1 and m, so taking every case $f[n][m]$ can be expressed as


$$f[n][m] = f[n-1][1] + f[n-1][2] + f[n-1][3] + \dots + f[n-1][m]$$

This equation will cover all the possible arrays with last element m and total no of elements n.

Similarly we can write $f[n-1][m]$ as

$$f[n-1][m] = f[n-2][1] + f[n-2][2] + f[n-2][3] + \dots + f[n-2][m-1]$$

Combining these two equations we get


$$f[n][m] = f[n][m-1] + f[n-1][m]$$

We can represent this in a 2D grid in which row no represents the no of elements in the array and column number represents the last element of the array. Every element in the first row is 1 as no of elements itself is 1 so array has only one element with value j(column number). Similarly, every element in the first column is 1 as the array has to end with 1 and as it is non-decreasing every element of the array should be 1.

If we think diagrammatically we can compute every element in the grid by the pattern of recursive relation. Following this process we can eventually reach the last element in the grid which is $f[n][m]$

[m].

Pseudo Code

```
int dp[n + 5][m + 5];
for (int i = 0; i < m; i++)
{
    dp[0][i] = 1;
}
for (int i = 0; i < n; i++)
{
    dp[i][0] = 1;
}
for (int i = 1; i < n; i++)
{
    for (int j = 1; j < m; j++)
    {
        dp[i][j] = dp[i][j - 1] + dp[i - 1][j];
    }
}
cout << dp[n - 1][m - 1] << endl;
```

Total number of increasing arrays of length n with every integer between 1 to m

Problem Statement

Given two integers n and m and our task is to determine the no of arrays with length n and every integer is between 1 and m. This problem is a modification of the previous problem.

Solution

Using the approach discussed in the previous problem we can compute the no of arrays of length n ending with m . In this problem there is no constraint on the last element so we can consider every case. As every element has to in-between 1 and m , last element can be 1,2.....m . Let the no of non-decreasing arrays with every element between 1 to m be $g[n][m]$ and let the no of non-decreasing arrays with last element m be $f[n][m]$. So $g[n][m]$ can be represented as

$$g[n][m] = f[n][1] + f[n][2] +f[n][m]$$

And similarly $g[n][m - 1]$ can be represented as

$$g[n][m - 1] = f[n][1] + f[n][2].....f[n][m - 1]$$

Combining these two equations



$$g[n][m] = g[n][m - 1] + f[n][m]$$

We already know how to compute $f[n][m]$ so this can be converted into 1D dynamic programming problem which can be computed easily

Pseudo Code

```
int f[n + 5][m + 5], g[m + 5];
for (int i = 0; i < m; i++)
{
    f[0][i] = 1;
}
for (int i = 0; i < n; i++)
{
    f[i][0] = 1;
}
for (int i = 1; i < n; i++)
{
    for (int j = 1; j < m; j++)
    {
        f[i][j] = f[i][j - 1] + f[i - 1][j];
    }
}
g[0] = 1;
for (int i = 1; i < m; i++)
{
    g[i] = g[i - 1] + f[n - 1][m - 1];
}
cout << g[m - 1] << endl;
```

No of Subsequences with a given sum S

Problem Statement

Given an array A , we have to count the no of subsequences of the array with sum S.

Example

Suppose if the array is {1,2,3,4} and S=6

Answer is 2 as there were two subsequences {1,2,3} and {2,4} with the given sum 6

Solution

Let us think of recursive approach. Let $f[i][N]$ be the no of subsequences with a sum N upto the i th element of the array. Can this be computed if we know $f[i - 1][0], f[i - 1][1], \dots, f[i - 1][N - 1]$ and $f[i - 1][N]$. The answer is yes , we can compute

There are two kinds of subsequences in $f[i][N]$, there may be some subsequences that contain i th element and some which do not contain i th element.

- The no of subsequences that do not contain i th element is $f[i - 1][N]$

- The no of subsequences which contain *i*th element are $f[i - 1][N - arr[i]]$, as the total sum of elements in the subsequence is N and *i*th element should be included , the sum of elements upto $(i - 1)$ th element is $N - arr[i]$

So the recursive equation is

$$f[i][N] = f[i - 1][N] + f[i - 1][N - arr[i]]$$

We can represent this in a 2D grid in which row no represents the no of elements of the array up to *i*th element and column number represents the corresponding sum. First row represents $i = 0$, which implies that no elements were chosen , so all the values in the first row are zero. The first column represents $sum=0$, so as all the elements in the array are positive ideally all values should be zero in the first column as well, but we fill all of them with 1 (think on this 🤔). Finally $f[N][S]$ will be our answer.

Pseudo code

```
for (int i = 1; i <= sum; i++)
{
    dp[1][i] = 0;
}
for (int i = 1; i <= n; i++)
{
    dp[i][0] = 1;
}

for (int i = 1; i <= n; i++)
{
    for (int j = 1; j <= sum; j++)
    {
        if (a[i - 1] > j)
        {
            dp[i][j] = dp[i - 1][j];
        }
        else
        {
            dp[i][j] = dp[i - 1][j] + dp[i - 1][j - a[i - 1]];
        }
    }
}
cout << dp[n][sum] << endl;
```

Algorithm Design Techniques

Sub-algorithms / Techniques

Subalgorithm-1

Let there be two arrays A and B where each array has n elements in it and are sorted. Let the elements in array A be $a[1], a[2], a[3], \dots, a[n]$ and in array B be $b[1], b[2], b[3], \dots, b[n]$. Let us make a one-one mapping between two arrays in such a way that every element of A is mapped to exactly one element in B and vice versa. The distance between the two arrays corresponding to a particular one-one relation is defined as $\sum |a[i] - b[j]|$, for every $a[i]$ and $b[j]$ which are mapped to each other in this relation. For ex if $n=3$, let a one-one relation be $a[1] \rightarrow b[2]$, $a[2] \rightarrow b[3]$ and $a[3] \rightarrow b[1]$ so the distance will be $|a[1] - b[2]| + |a[2] - b[3]| + |a[3] - b[1]|$.

The one-one relation with the minimum distance will be $a[i] \rightarrow b[i]$, for all $i \leq n$, that is $a[1] \rightarrow b[1]$, $a[2] \rightarrow b[2]$, $a[3] \rightarrow b[3]$ $a[n] \rightarrow b[n]$

This technique is used in some problems.

Sample Problem

Problem Statement

We are given an array A. Let the elements in A be $a[0], a[1], a[2], \dots, a[n]$. We can swap any consecutive elements in the array. The problem is finding out the no of swaps required so that the array does not contain two neighboring items with the same parity. That is every odd number should have even numbers in its adjacent positions and every even number should have odd numbers in its adjacent positions.

Let us represent odd element as o and even element as e. So the final array be either in the form

e o e o e o or o e o e o e

Let the swaps required to convert the given array to the form e o e o e o be ans1 and similarly swaps required to convert it into o e o e o e be ans2. Then our final answer will be a minimum of ans1 and ans2.

Let us try to convert the given array into the form e o e o with the minimum swaps required. Here in the final array, all odd numbers should be in even positions and all even numbers should be in odd positions. The primary observation is the no of odd numbers and no of even numbers in the given array should be the same otherwise conversion to the required form is not possible

```
If the array is e e e o o o initially and we have to convert it to e o e o e o
then the minimum no of swaps required is 3.
First we consider the even numbers which are in even positions and odd numbers
which are in odd positions as mentioned above.
Clearly we can see that in the 2nd position there is an even number and in
the 5th position there is an odd number.
This can be converted to e o e o e o as

e e e o o o ----> e e o e o o ----> e o e e o o ----> e o e o e o
The steps are
1) swap elements which are at positions {3,4}
2) swap elements which are at positions {2,3}
3) swap elements which are at positions {4,5}
So this involves 3 swaps.
So if there is
```

If we check for some examples we can get to the conclusion that to swap an even number at position x with an odd number at position y we require $|x-y|$ swaps. So let us push all the even number positions which are even in one array say a and push all the odd numbers which are odd in another array b (we are trying to convert the array in the form o e o e as mentioned above). If we think carefully this results in the above-mentioned sub-algorithm. We have to minimize $\sum |a[i] - b[j]|$, in the two arrays a and b get the minimum no of swaps. let this be ans1.

We have to do the same for converting the given array into e o e o e o Let the swaps required for this be ans2. So our final answer is the minimum of ans1 and ans2.

```
vector<long long int>num,prob1_even,prob1_odd,prob2_even,prob2_odd;
long long int ans1=1e9,ans2=1e9;
for(int i=0; i<n; i++)
{
    long long int x;
    cin>>x;
    num.push_back(x);
}
```



```

for(int i=0; i<n; i++)
{
    if(num[i]%2==0&& i%2!=0)
    {
        prob1_even.push_back(i);
    }
    if(num[i]%2!=0&& i%2==0)
    {
        prob1_odd.push_back(i);
    }
}
for(int i=0; i<n; i++)
{
    if(num[i]%2==0&& i%2==0)
    {
        prob2_even.push_back(i);
    }
    if(num[i]%2!=0&& i%2!=0)
    {
        prob2_odd.push_back(i);
    }
}
if(prob1_even.size()!=prob1_odd.size()&& prob2_even.size()!=prob2_odd.size())
{
    cout<<-1<<endl;
}
else
{
    if(prob1_even.size()==prob1_odd.size())
    {
        ans1=0;
        for(int i=0; i<prob1_even.size(); i++)
        {
            ans1+=abs(prob1_even[i]-prob1_odd[i]);
        }
    }
    if(prob2_even.size()==prob2_odd.size())
    {
        ans2=0;
        for(int i=0; i<prob2_even.size(); i++)
        {
            ans2+=abs(prob2_even[i]-prob2_odd[i]);
        }
    }
    cout<<min(ans1,ans2)<<endl;
}
}

```

Subalgorithm-2

Suppose we are given an array which has n elements, where i^{th} element can be represented as $a[i]$, our task is to find a integer x such that $|a[0] - x| + |a[1] - x| + |a[2] - x| + \dots + |a[n - 1] - x|$ is as minimum as possible.

The optimal x in such a case is the median of all elements.

If n is odd then there x can take only one value which is the median,

But if n is even then x can take any value between the two medians (think on this 🤔)

Intuition

Let us pick a number y which is smaller than the median, so by increasing y the sum decreases as there were more elements in the array to right of the y than to the left of y . Similarly if y is greater than median, by increasing y the sum increases as there are more elements to the left of y . This implies the sum should be minimum at the median of the array.

Sample Problem

Suppose if we are given n coordinates in 2D space and distance between any points can be given as $|x - x'| + |y - y'|$. Our task is to find no of points so that sum of distance between every point to all the n coordinates is as minimum as possible.

Example

Suppose if the coordinates are $\{0,2\}, \{1,0\}, \{2,3\}, \{3,1\}$

Then no of points with sum of distance between every point to all these 4 coordinates are 4.

They are $\{1,1\}, \{2,2\}, \{1,2\}, \{2,1\}$

Note that the total distance from each of these points to the coordinates mentioned above is 8 and it is the minimum.

Solution

In first glance, this looks like a 2D problem but it is actually a 1D problem. The primary observation is that if we change the x coordinate the sum of distances by y is not changed at all. This implies that we can pick x -coordinate and y -coordinate independently. So if the no of x -coordinates x' for which $|x' - x_1| + |x' - x_2| \dots |x' - x_n|$ is minimum are $ans1$ and similarly the no of y -coordinates y for which $|y' - y_1| + |y' - y_2| \dots |y' - y_n|$ be $ans2$ then our final answer is $ans1 \times ans2$.

Using the above-mentioned technique we know that x' should be the median of x_1, x_2, \dots, x_n in order to make $|x' - x_1| + |x' - x_2| \dots |x' - x_n|$ minimum. If n is odd there exists only one median so the answer will be 1. But if n is even as mentioned in the above technique, the answer should be the count of all numbers that are in between two medians.

Pseudo Code

```
for (int i=0;i<n;i++)
{
    cin>>a[i]>>b[i];
}
sort(a,a+n);
sort(b,b+n);
cout<<(a[n/2]-a[(n-1)/2+1])*(b[n/2]-b[(n-1)/2+1])<<endl
```

Few handy Implementation Tricks

- In greedy problems, sometimes it is very difficult to bring a mathematical proof of correctness to solve the problem. So in such cases just get intuition and check with a few test cases or if the problem is given in some contest then submit it and check. If the approach is wrong then try to think of a dynamic programming approach.
- In some greedy problems where some kind of sorting is involved, it is always optimal to solve it using a heap or priority queue (Room Allocation Problem and minimize sum problems we discussed in the greedy section).

Priority queue can be of two types, min-heap or max-heap. min-heap stores the minimum element at its top while max-heap has the maximum element

In c++ priority queue can be declared as



```
priority_queue< int > pq; , for max-heap
```



```
priority_queue< int , vector< int >, greater< int > > pq for min-heap
```

We can do some operations like emptying the heap using *empty()* function , get the size of heap using *size()* , get the topmost element using *top()* , push and pop from the heap using *push()* and *pop()*.

- If we encounter a problem in which we have to count the no of arrays satisfying a certain property, first think of any method in which it can be solved using permutations and combinations. If there is no such approach we can think of then start thinking in dynamic programming.
- Generally, most of the problems which are designed in such a way to count the no of subsequences that satisfies a condition, think of dynamic programming approach.