# Demux Summer Hackathon -2

## Solution

## 1.PAIRS

The first observation we can make is that we don't need to enumerate all N^2 pairs and then check whether the pairs of integers have a difference of K.

What we simply need to do is - for each integer N, check whether the original array contains N-K and N+K. In fact, we can iterate over each number N in the original array and check whether N+K exists in the same array. So basically we need a data structure supporting fast membership inquiry.

Two of the most popular data structures supporting this operation are binary search tree and hash table. The former supports O(logN) time complexity, while the latter supports O(1) time complexity.

Most programming languages already support these data structures in their library functions. For example, if you want to use a binary search tree in C++, you can use STL set; if you want to use a hash table in C++, you can use unordered_set.

It is easy to test the performance of both set and unordered_set. And you can see that unordered_set does run faster than set, which in turn validates the advantage over time complexity.

**Code:**

```
//C++
//Pairs.cpp
//Pairs
//Algorithms - Search

#include<iostream>
#include<unordered_set>
using namespace std;

unordered_set<int> s;

int main()
{
   int n, k, val;
   cin >> n >> k;
   for (int i = 0; i < n; i++)
   {
      cin >> val;
      s.insert(val);
   }
   int ans = 0;
   for (unordered_set<int>::iterator it = s.begin(); it != s.end(); ++it)
      if (s.find(*it + k) != s.end()) ans++;
   cout << ans << endl;
```

```
    return 0;
}
```

---

**#!/bin/python3**

```
import os

# Complete the pairs function below.
def pairs(k, arr):
    a = set(arr)
    # make a set of all a[i] + k
    b = set(x + k for x in arr)
    # return the length of the intersection set
    return len(a&b)

if __name__ == '__main__':
    fptr = open(os.environ['OUTPUT_PATH'], 'w')

    nk = input().split()

    n = int(nk[0])

    k = int(nk[1])

    arr = list(map(int, input().rstrip().split()))

    result = pairs(k, arr)

    fptr.write(str(result) + '\n')

    fptr.close()
```

## 2. The Full Counting Sort

**C++:**

```
#include <cmath>
#include <cstdio>
#include <vector>
#include <iostream>
#include <algorithm>
#include<string>
#include<list>
using namespace std;


vector<string> a[100];
char str[100007];
int n,i,j,x;

int main() {
    /* Enter your code here. Read input from STDIN. Print output to STDOUT */


    cin>>n;
    for(i=0;i<n/2;i++)
    {
        cin>>x;
        cin>>str;
        a[x].push_back("-");

    }

    for(;i<n;i++)
    {
        cin>>x;
        cin>>str;
        a[x].push_back(str);

    }

    for(i=0;i<100;i++)
    {
        x=a[i].size();
        for(j=0;j<x;j++)
            cout<<a[i][j]<<" ";
    }
    return 0;

}
```

---

**JAVA:**

```
import java.io.*;
import java.util.*;
import java.text.*;
import java.math.*;
import java.util.regex.*;

public class Solution {
    public static void main(String[] args) throws Exception {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(in.readLine());
        StringBuffer[] map = new StringBuffer[100];
        for(int i = 0; i < 100; i++) {
            map[i] = new StringBuffer();
        }
        for(int i = 0; i < n; i++) {
            StringTokenizer tok = new StringTokenizer(in.readLine());
            int v = Integer.parseInt(tok.nextToken());
            String s = tok.nextToken();
            map[v].append(i < n / 2 ? "-" : s).append(" ");
        }
        for(int i = 0; i < 100; i++) {
            System.out.print(map[i]);
        }
        System.out.println();
    }
}
```

## 3.Emma’s Super computer

As the grids are limited to 15*15 , a brute force solution will work. Just brute force all centers of pulses and their sizes and check if they intersect.

**C++**

```
#include <bits/stdc++.h>

using namespace std;


const int N = 20;

int n, m, ans;

char c[N][N];


int main() {


  scanf("%d%d", &n, &m);

  for (int i = 1; i <= n; i++) {

    for (int j = 1; j <= m; j++) {

      cin >> c[i][j];

    }

  }


  for (int x = 1; x <= n; x++) {

    for (int y = 1;y <= m; y++){

      int r = 0;

      while (c[x + r][y] == 'G' && c[x - r][y] == 'G' &&

          c[x][y + r] == 'G' && c[x][y - r] == 'G') {

        c[x + r][y] = c[x - r][y] = c[x][y + r] = c[x][y - r] = 'g';

        for (int X = 1; X <= n; X++) {

          for (int Y = 1; Y <= m; Y++){

            int R = 0;

            while (c[X + R][Y] == 'G' && c[X - R][Y] == 'G' &&

                c[X][Y + R] == 'G' && c[X][Y - R] == 'G') {

              ans = max(ans, (1 + 4 * r) * (1 + 4 * R));

              R++;
```

```
                }
              }
            }
            r++;
          }
          r = 0;
          while (c[x + r][y] == 'g' && c[x - r][y] == 'g' &&
                c[x][y + r] == 'g' && c[x][y - r] == 'g') {
            c[x + r][y] = c[x - r][y] = c[x][y + r] = c[x][y - r] = 'G';
            r++;
          }
        }
      }
      cout << ans << endl;
      return 0;
}
```

## 4.Absolute Permutation

**Solution**

If K=0, print the permutation in the order from 1 to N .

If  N is odd, it's easy to prove that the answer does not exist.

If **pos i > 7, pos i-1 = K**; otherwise, posi -1 = -K.

Because N is odd, We can't have a number of K's equal to a number of —K''s. Therefore, we can't have $\sum pos_i - i = 0.$

Because this will hold for any possible permutation, we can say such a permutation does not exist.

If N is even, choose every slot of 2 x K elements and shuffle them to get the lexicographically smallest absolute permutation.

For example: N = 8 and K = 2

- 1,2,3,4,5,6, 7,8
- (1,2, 3,4] first slot --> [3, 4, 1, 2]
- [5, 6, 7, 8] second slot --> [7, 8, 5, 6]

Therefore, the lexicographically smallest absolute permutation is **3 4 1 2 7 8 5 6**. Note that if N is not evenly divisible by

2 x K, we can't have an absolute permutation and thus print -1.

**C++**

```
#include <bits/stdc++.h>

using namespace std ;


const int maxn = 1e5 + 10;

const int mod = 1e9 + 7;


int main () {

    int t;

    cin >> t;

    while (t--) {

        int n, k;

        cin >> n >> k;

        if (!k) {

            int i;

            for (i = 1; i <= n; i++) {

                cout << i << " \n"[ i == n ];

            }

        } else {

            if (n % 2) {

                cout << -1 << "\n";

            } else {

                if ( n % (2 * k) ) {

                    cout << -1 << "\n";

                } else {

                    vector <int> A(n+1);

                    int i;

                    for (i = 1; i <= n; i++) A[i] = i;

                    for (i = 1; i <= n; i += 2*k) {

                        int j;

                        for(j = 1; j <= k; j++) {
```

```
                    swap(A[i + j - 1], A[i + j - 1 + k]);
                  }
                }
                for (i = 1; i <= n; i++) {
                  cout << A[i] << " \n"[ i == n ];
                }
              }
            }
          }
        }
        return 0;
```

## 5.Gridland Metro

There are m X m cells in the city. We must find the number of cells not occupied by a train track.

Observe that &k < $10^3$, so the number of rows we have to check is also < $10^3$

In each row, there can be multiple rail lines.

For each row sort the rail lines by their **l** values.

Take a pointer that iterates through the column of each row and indicates the total distance we've covered so far. In the code below, this pointer is called **p**.

Next, check every rail line of each row. If **l** of the **i**$^{\text{th}}$ rail line is less than **p**, that mean we have already taken this rail line into account and don't need to do anything with it.

Next, if **r** is smaller than **p** but **l** is greater than **p**, that means we haven't yet considered the **l** — p portion of this rail and have to subtract it from the total number of cells.

Finally, if r is greater than p and so is **l,**the whole rail line has not been considered yet and so we subtract **l — r + 1** from the total number of cells.

After each iteration, we have to update **p** to the point we have covered thus far.

### Python

```
n, m, k = map(int, raw_input().split())
grid_points = 0
mapper = {}
for _ in range(k):
    r, c1, c2 = map(int, raw_input().split())
    if r in mapper:
        mapper[r].append((c1, c2))
    else:
        mapper[r] = [(c1, c2)]
for k in mapper:
    temp = mapper[k]
    temp.sort()
    begin = temp[0][0]
    end = temp[0][1]
    points = 0
    for i in range(1, len(temp)):
        if temp[i][0] > end:
            points += end - begin + 1
            begin = temp[i][0]
            end = temp[i][1]
        else:
            end = max(end, temp[i][1])
    points += end - begin + 1
    grid_points += points
print m*n - grid_points
```

## C++

```
#include <bits/stdc++.h>
#include<assert.h>

using namespace std;

vector<pair<int, int> >v[1003];
map<int,int>mp;

void solution() {

    int n, m, k, r, c1, c2, sz1, sz2;
```

```
long long ans, temp, non_emp;

cin>>n>>m>>k;

for(int i=0; i<k; i++){
    cin>>r>>c1>>c2;
    assert(r>0 && r<=n && c1>0 && c1<=m && c2>=c1 && c2<=m);

    if(mp.find(r) == mp.end())
        mp[r] = mp.size();
    r = mp[r];

    v[r].push_back( make_pair(c1,c2) );
}



sz1 = mp.size();
for(int i=1; i<=sz1; i++)
    sort(v[i].begin(), v[i].end());

ans = (long long)n*(long long)m;
non_emp = 0;

for(int i=1,p; i<=sz1; i++)
{
    sz2 = v[i].size();
    p = 0;
    for(int j=0; j<sz2; j++)
    {
        if(v[i][j].first <= p)
        {
            temp = v[i][j].second - p;
            if(temp>0)
                non_emp += temp;
            }
            else if(v[i][j].first > p)
                non_emp += (v[i][j].second - v[i][j].first + 1);

            p = max(p, v[i][j].second);
        }
    }

    ans -= non_emp;
    assert(ans>=0);
    cout<<ans<<"\n";
}

int main () {

    solution();

    return 0;
}
```

## 6.Recursive Digit Sum

We define *super-digit* function as following recursive function:
super-digit(n) =n,n<10

= super-digit(sum-of-digit(n)), otherwise

where,

sum-of-digit(m) =m,m=0

=(m mod 10) + sum-of-digit(m/10)

Note that since initially **n** can be very large, use string representation to represent it for the first time.

```
import Data.List
import Data.Char

main :: IO ()
main = getContents >>= print. (\[n, k] -> superDigit ((read k) * (getSumStr n))). words

superDigit n
  | n < 10 = n
  | otherwise = superDigit (getSum n)

getSum 0 = 0
getSum n = n`rem`10 + getSum (n`div`10)

getSumStr [] = 0
getSumStr (x:xs) = (ord x - ord '0') + getSumStr xs
```

---

**C ++**

```
#include <iostream>

using namespace std;

int super_digit(const string &s) {
   int rem = 0;
   for(char ch : s) {
      rem = (rem + (ch - '0')) % 9;
   }
   return rem;
}


int main() {
   ios_base::sync_with_stdio(false);

   string s;
   cin >> s;

   int k;
   cin >> k;

   k %= 9;

   if(k == 0) {
      cout << "9" << endl;
   } else {
      int remainder = super_digit(s);
      remainder = (remainder * k) % 9;
      cout << ((remainder != 0) ? to_string(remainder) : "9") << endl;
   }

   return 0;
}
```

## 7.Crossword Puzzle

Recursively, try every possible valid position to place a word.
Suppose we have **k** words, then start from first word and try to place it at valid position in the grid, either horizontally or vertically, and then move ahead with the updated grid for the next word. If for any word there is no valid position found, then backtrack and restore the previous grid(i.e, grid before when last word is not placed) and so on.

If we place all the words in the grid for any configuration, then print the grid as an answer and terminate the resursion.

**Code:**

```
#include <bits/stdc++.h>
using namespace std;

vector<string> grid(10);
vector<string> words;
bool f;

void call(int ind)
{
    if(!f) {
        return;
    }
    if(ind == words.size()) {
        if(f) {
            for(auto word: grid) {
                cout<<word<<endl;
            }
            f=false;
        }
        return;
    }
    int i,j,p,q,k;
    for(i=0;i<10;++i) {
        for(j=0;j<10;++j) {
            p=i,q=j;
            for(k=0;k<words[ind].size() && p+k<10;++k) {
                if(grid[p+k][q] != '-' && grid[p+k][q] != words[ind][k]) {
                    break;
                }
            }

            if(k==words[ind].size()) {
                vector<string> temp = grid;
                for(k=0;k<words[ind].size();++k) {
                    grid[p+k][q] = words[ind][k];
                }
                call(ind+1);
                grid = temp;
            }

            for(k=0;k<words[ind].size() && q+k<10;++k) {
                if(grid[p][q+k] != '-' && grid[p][q+k] != words[ind][k]) {
                    break;
                }
            }

            if(k==words[ind].size()) {
                vector<string> temp = grid;
                for(k=0;k<words[ind].size();++k) {
                    grid[p][q+k] = words[ind][k];
                }
                call(ind+1);
                grid = temp;
            }
        }
    }
}

int main()
{
    f=true;

    int i,j;
    for(i=0;i<10;++i) {
        cin>>grid[i];
    }

    string s,w;
    cin>>w;

    for(auto x: w) {
        if(x==';') {
            words.push_back(s);
            s="";
        } else
            s+=x;
    }
    words.push_back(s);
    call(0);

    return 0;
}
```

## 8.Fraudulent Activity Notifications

In this problem, you need to find the running median. There can be several approaches to solving this. Notice that a client can spend at most $200 per day. We can take advantage of this small number.

(2, 3, 2, 5, 7, 6, 6, 7, 4]. The maximum number in the array is 7. If you write down the frequency of each numbers from 0 to 7,

you will get a table like this:

- ¢ freq[0| = 0 .

- freq{1] =0

- freq|2| =2

- freq[3] = 1

- freq(4] =1
- freq[5| =1
- freq|6] = 2
- freq{7| =2

There are 9 elements in the array, so the median is the 5th number in the sorted array. You can loop over the frequency table to find the 5th number.

**Get back to the original problem**

In the original problem, you need to maintain a frequency table for each window of size d in the array. You can do it by keeping track of the starting and ending point of the window. Let's suppose the start point of the current window is s and the end point is **e ande — s + 1 = d.** Also, assume you already have a frequency table for that window. When you go to next window **(s + 1,e +1),** you can update the frequency table by reducing the frequency of the element in index s and by increasing the frequency of the element in index e + 1. Using the frequency table you can find the median and your problem is solved.

The complexity is **O(n x maximum number in the array)**

**Tricky Part**

Note that the median can be a floating point value when the size of the array is even. For example, if the array is [3, 1, 2, 4], the median is 2.5. You will not pass the 2nd sample |/O if you don't handle it properly. .

**Code:**

```
#include <iostream>

#include <cstdio>

#include <algorithm>

#include <cstring>

#include <ctime>

#include <cassert>

using namespace std;

#define SZ(x) ((int)(x.size()))

#define FOR(i,n) for(int (i)=0;(i)<(n);++(i))

#define FOREACH(i,t) for (typeof(t.begin()) i=t.begin(); i!=t.end(); i++)
```

```
#define REP(i,a,b) for(int (i)=(a);(i)<=(b);++i)



typedef long long ll;
const int INF = 1e9;

const int N = 2e5;
const int V = 200;

int a[N];

int cnt[V+1];

int main()
{
   ios_base::sync_with_stdio(0);
   int n, d;
   cin >> n >> d;
   assert(n >= 1 && n <= N);
   assert(d >= 1 && d <= n);
   FOR(i, n) cin >> a[i];
   FOR(i, n) assert(a[i] >= 0 && a[i] <= V);
   int res = 0;

   FOR(i, d) cnt[a[i]]++;
   REP(i, d, n-1)
   {
      //SOLVE HERE
      int acc = 0;
      int low_median = -1, high_median = -1;
      REP(v, 0, V)
```

```
        {
            acc += cnt[v];
            if(low_median == -1 && acc >= int(floor((d+1)/2.0)))
            {
                low_median = v;
            }
            if(high_median == -1 && acc >= int(ceil((d+1)/2.0)))
            {
                high_median = v;
            }
        }
        assert(acc == d);
        int double_median = low_median + high_median;
        //cout << low_median << " " << high_median << " -> " << median << endl;
        if(a[i] >= double_median)
        {
            res++;
        }
        cnt[a[i-d]]--;
        cnt[a[i]]++;
    }
    cout << res << endl;
    return 0;
}
```

# 9..Insertion Sort Advanced Analysis

**Problem Overview**

Given an array of size N, find the number of swaps/shifts of elements which will be done as the array is sorted using insertion sort.

**Insertion sort**

The most obvious way will be to do an insertion sort and record the number of shifts required to sort it. It will take 0($N^2$) time which means 5 x $10^{10}$ in the worst case under our constraints. We need to do better. The obvious answer can be as large as $10^{10}$ so we cannot count each shift one by one. It gives us an idea that we need to shift elements in the same way as insertion sort does, but not one position at a time. There is an algorithm which does just that.

**Merge Sort**

If you know merge sort you must have noticed that when we merge 2 sorted arrays if the element of the 2nd array(on the right) is smaller we put its element in the new sorted array which indirectly means we are shifting that element to the left by the number of elements remaining in the 1st array. We are actually shifting the element but in a higher order. That is why merge sort's complexity is O(N logN), which is the complexity of this problem. We just need to implement merge sort and add the shifts when an element of the 2nd array is less then the element of 1st array.

```
#include<iostream>

#include<cstdio>


using namespace std;


long long ans=0;

void mergei(int a[],int i,int j)

{

   int ni=((i+j)/2)+1,nj=j+1;

   int s=i;

   int * arr = new int [j-i+1];

   j=ni;

   int k=0;

   while(i<ni && j<nj)

   {

      if(a[i]<=a[j])

      {

         arr[k]=a[i];

         i++;

      }

      else

      {

         arr[k]=a[j];
```

```
            ans+=(ni-i);
            j++;
        }
        k++;
    }
    for(;i<ni;i++,k++)
        arr[k]=a[i];
    for(;j<nj;j++,k++)
        arr[k]=a[j];
    for(k=0;s<nj;s++,k++)
        a[s]=arr[k];
    delete [] arr;
}

void m_sort(int a[],int i,int j)
{
    if(i<j)
    {
        m_sort(a,i,(i+j)/2);
        m_sort(a,((i+j)/2)+1,j);
        mergei(a,i,j);
    }
}
int main()
{
    int t;
    scanf("%d",&t);
    while(t--)
    {
        int n;
        ans=0;
        scanf("%d",&n);
        int * a = new int[n];
        for(int i=0;i<n;i++)
        scanf("%d",&a[i]);
      m_sort(a,0,n-1);
      cout<<ans<<endl;
    }
    return 0;
}
```

## 10.Repetitive K-Sums

**Approach**

Firstly, For a given N and K the number of terms in K-Sums sequence will be n+k-1 Choose k.
Let these terms be M and represented by array Sums[] and original array be A[]

After sorting this Sums[] sequence we can say that A[0] = Sums[0]/K. And erase S[0] as it won't have information for any other terms in sequence A[].

Hence for any Sums[i] we note that after erasing all k-sums involving numbers a[0], a[1], ..., a[i-1] the minimal k-sum is a[i] + (k-1) * a[0], hence giving the next a[i] value.

We do the removing part using dynamic programming. Erasing all the terms in S[] that are made using previous K sum elements of A[] till A[i].

Also we don't need to erase k-sums that contain a[n-1] since we have already restored the whole array by then.

**code:**

C++

```
#include <bits/stdc++.h>
using namespace std;

typedef long long LL;

// returns n! / k! / (n-k)! = n * (n-1) * ... * (n-k+1) / 1 / 2 / ... / k
// = n / 1 * (n-1) / 2 * (n-2) / 3 * ... * (n-k+1) / k
int bin(int n, int k) {
    if(k > n - k) {
        k = n - k;
    }
    int p = 1;
    for (int i = 1; i <= k; ++i) {
        p *= n + 1 - i;
        p /= i;
    }
    return p;
}

int n, k;
LL a[100000];
multiset<LL> kSums;

// recursive routine that erase all sums having a[i] as the last element
// @j is the current a[j] we want to add to the sum
// @cnt is the count of numbers in the current sum
// @sum is the value of the sum
void rec(int i, int j, int cnt, LL sum) {
    if (cnt == k) {
        kSums.erase(kSums.find(sum));
    } else {
        rec(i, j, cnt + 1, sum + a[j]);
        if (j < i) {
            rec(i, j + 1, cnt, sum);
        }
    }
}

int main() {
    int T;
    scanf("%d", &T);
    assert ( 1<=T<=100000);
    for (int t = 0; t < T; ++t) {
        // n and k defined globally
        scanf("%d%d", &n, &k);
        assert ( 1<=n<=100000);
        assert ( 1<=k<=100000);
        int m = bin(n + k - 1, k); // the total number of k-sums
        assert ( 1<=m<=100000);

        // input k-sums and insert them into multiset
        kSums.clear();
        for (int i = 0; i < m; ++i) {
            LL kSum;
            scanf("%lld", &kSum);
            assert ( 1<=kSum<=1000000000000000000L);
            kSums.insert(kSum);
        }

        // the minimal k-sum is alsways a[0] * k
        a[0] = *(kSums.begin()) / k;
        kSums.erase(kSums.begin());

        for (int i = 1; i < n; ++i) {

            // after erasing all k-sums involcing numbers a[0], a[1], ..., a[i-1]
            // the minimal k-sum is a[i] + (k-1) * a[0]
            a[i] = *(kSums.begin()) - (k - 1) * a[0];

            // we don't need to erase ksums that contain a[n-1]
            // since we have already restored the whole array
            // and this important in the case n=2 since k could be 99999 in this case
            // which could lead to stack overflow and TL
            if (i < n - 1) {
              rec(i, 0, 1, a[i]);
            }
        }

        for (int i = 0; i < n; ++i) {
            printf("%lld%c", a[i], i < n - 1 ? ' ' : '\n');
        }
    }
    return 0;
```

```
}
```