Competitive Programming Cheatsheet

September 16, 2021

Contents

1 Quick Notes

1.1 Fast Input

```
ios::sync_with_stio(false);
cin.tie(0);
```

1.2 vimrc

```
set tabstop=4
set shiftwidth=4
set expandtab
set smarttab
set wrap
set number
syntax enable
```

2 Strings

2.1 Suffix Arrays

A suffix array is a sorted array of all suffixes of a given string.

Some interesting observations regarding suffix arrays:

1. Prefix of a suffix is a unique (not distinct) substring of the given string.
2. All the suffixes are unique.
3. If 2 different suffixes, say 'ana' and 'anana', have a common prefix, here 'ana', it is NOT the same substring in the given string. The first 'a' in both the suffixes occurs at different positions in the given string.

Construction in $O(nlognlogn)$:

```
struct suffix {
    int index;
    int rank[2];
};

bool cmp(struct suffix a, struct suffix b) {
  if(a.rank[0] == b.rank[0])
        return a.rank[1] < b.rank[1];
  return a.rank[0] < b.rank[0];
}

vector<int> suffixify(string a) {
  int n = a.size();
  vector<suffix> suffixes(n);
  for (int i = 0; i < n; ++i) {
    suffixes[i].index = i;
    suffixes[i].rank[0] = a[i] - 'a';
    suffixes[i].rank[1] = i+1<n ? a[i+1]-'a' : -1;
  }
  sort(suffixes.begin(), suffixes.end(), cmp);

  vector<int> indices(n);
  for (int k = 4; k < 2*n; k *= 2) {
    int rank = 0;

    int prev_rank = suffixes[0].rank[0];
    suffixes[0].rank[0] = rank;
    indices[suffixes[0].index] = 0;

    for (int i = 1; i < n; ++i)
    {
      if(suffixes[i].rank[0] == prev_rank &&
         suffixes[i].rank[1] == suffixes[i-1].rank[1]) {
        prev_rank = suffixes[i].rank[0];
        suffixes[i].rank[0] = rank;
      }
      else {
        rank++;
        prev_rank = suffixes[i].rank[0];
        suffixes[i].rank[0] = rank;
      }
      indices[suffixes[i].index] = i;
    }

    for (int i = 0; i < n; ++i) {
      int next = suffixes[i].index + k/2;
      suffixes[i].rank[1] = next < n ? suffixes[indices[next]].rank[0] : -1;
```

```
    }

    sort(suffixes.begin(),suffixes.end(), cmp);
  }

  vector<int> suffixArray(n);
  for (int i = 0; i < n; ++i) suffixArray[i] = suffixes[i].index;
  return suffixArray;
}
```

2.2 Longest Common Prefix

LCP Array is an array of size n (like Suffix Array). A value lcp[i] indicates length of the longest common prefix of the suffixes indexed by suffix[i] and suffix[i+1]. suffix[n-1] is not defined as there is no suffix after it.

Interesting observation(s) regarding LCP Arrays:

Sum of all elements in LCP array is the total number of duplicate substrings in the given string. This is because the common prefix between, say "aba" and "abaccd" has length 3 (lcp[i]) which implies 3 duplicate substrings "a", "ab", "aba".

Implementation in O(n):

```
vector<int> lcp(string a, vector<int> &suffixArray) {
    int n = a.length();
    vector<int> LCP(n,0);
    vector<int> indexInSA(n);
    for (int i = 0; i < n; ++i)
        indexInSA[suffixArray[i]] = i;

    int k = 0;
    for (int i = 0; i < n; ++i) {
        if(indexInSA[i]==n-1) {
            k = 0;
            continue;
        }

        int j = suffixArray[indexInSA[i]+1];
        while(i+k<n && j+k<n && a[i+k]==a[j+k]) k++;

        LCP[indexInSA[i]] = k;
        k = max(0,k-1);
    }
    return LCP;
}
```

2.3 Palindromic Subsequences

Calculate the number of palindromic subsequences in a string.
E.g. aab = 4 (a,a,b,aa) aba = 5 (a,b,a,aba,aa) abca = 7 (a,b,c,a,aba,aca,aa)
Note: NOT substrings BUT subsequences (not necessarily contiguous)

```
string input;
int ps(int l, int r) {
    if(l==r)
        return 1;
    else if(l+1==r)
        return 2 + (input[l]==input[r]);
    else if(l<r)
        return ps(l+1,r) + ps(l,r-1) - ps(l+1,r-1) + (input[l]==input[r])*(ps(l+1,r-1)+1);
```

```
    else
        return 0;
}
```

The answer is $ps(0, n-1)$ where n = length of the string.

2.4 Longest Palindromic Substring

Manacher's Algorithm: O(n)

```
void findLongestPalindromicString()
{
    int N = strlen(text);
    if(N == 0)
        return;
    N = 2*N + 1; //Position count
    int L[N]; //LPS Length Array
    L[0] = 0;
    L[1] = 1;
    int C = 1; //centerPosition
    int R = 2; //centerRightPosition
    int i = 0; //currentRightPosition
    int iMirror; //currentLeftPosition
    int maxLPSLength = 0;
    int maxLPSCenterPosition = 0;
    int start = -1;
    int end = -1;
    int diff = -1;

    //Uncomment it to print LPS Length array
    //printf("%d %d ", L[0], L[1]);
    for (i = 2; i < N; i++)
    {
        //get currentLeftPosition iMirror for currentRightPosition i
        iMirror  = 2*C-i;
        L[i] = 0;
        diff = R - i;
        //If currentRightPosition i is within centerRightPosition R
        if(diff > 0)
            L[i] = min(L[iMirror], diff);

        //Attempt to expand palindrome centered at currentRightPosition i
        //Here for odd positions, we compare characters and
        //if match then increment LPS Length by ONE
        //If even position, we just increment LPS by ONE without
        //any character comparison
        while ( ((i + L[i]) < N && (i - L[i]) > 0) &&
            ( ((i + L[i] + 1) % 2 == 0) ||
            (text[(i + L[i] + 1)/2] == text[(i - L[i] - 1)/2] )))
        {
            L[i]++;
        }

        if(L[i] > maxLPSLength)  // Track maxLPSLength
        {
            maxLPSLength = L[i];
            maxLPSCenterPosition = i;
        }
```

```
        //If palindrome centered at currentRightPosition i
        //expand beyond centerRightPosition R,
        //adjust centerPosition C based on expanded palindrome.
        if (i + L[i] > R)
        {
            C = i;
            R = i + L[i];
        }
        //Uncomment it to print LPS Length array
        //printf("%d ", L[i]);
    }
    //printf("\n");
    start = (maxLPSCenterPosition - maxLPSLength)/2;
    end = start + maxLPSLength - 1;
    printf("LPS of string is %s : ", text);
    for(i=start; i<=end; i++)
        printf("%c", text[i]);
    printf("\n");
}
```

2.5 Z Algorithm

The Z-array z of a string s of length n contains for each k = 0,1,...,n − 1 the length of the longest substring of s that begins at position k and is a prefix of s. Thus, z[k] = p tells us that s[0...p−1] equals s[k...k + p − 1].

Construct Z-Array:

```
vector<int> z(const string& s) {
  int n = s.size();
  vector<int> z(n);
  int x = 0, y = 0;
  for (int i = 1; i < n; i++) {
    z[i] = max(0, min(z[i-x], y-i+1));
    while (i+z[i] < n && s[z[i]] == s[i+z[i]]) {
        x = i; y = i+z[i]; z[i]++;
    }
  }
  return z;
}
```

3 Graphs

3.1 Shortest Paths

3.1.1 Dijkstra's

Dijkstra's algorithm finds shortest paths from the starting node to all nodes of the graph. The benefit of Dijsktra's algorithm is that it is more efficient and can be used for processing large graphs. However, the algorithm requires that there are no negative weight edges in the graph.

Complexity is implementation dependant: Using adj list + pq - O(v+eloge), using adj matrix - O(v*v). Note: Simply swapping out adj list with matrix in the implementation below will result in O(v*v+eloge). O(v*v) implementation is different.

Implementation:

```
for (int i = 1; i <= n; i++) distance[i] = INF;

distance[x] = 0;
q.emplace(0, x);
while (!q.empty()) {
  int a = q.top().second; q.pop();
  if (processed[a]) continue;
  processed[a] = true;

  for (auto u : adj[a]) {
    int b = u.first, w = u.second;
    if (distance[a] + w < distance[b]) {
      distance[b] = distance[a] + w;
      q.emplace(-distance[b], b);
    }
  }
}
```

3.1.2 Bellman–Ford

The Bellman–Ford algorithm1 finds shortest paths from a starting node to all nodes of the graph. The algorithm can process all kinds of graphs, provided that the graph does not contain a cycle with negative length. If the graph contains a negative cycle, the algorithm can detect this.

Implementation:

```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++){
  for (auto e : edges){
    int a, b, w;
    tie(a, b, w) = e;
    distance[b] = min(distance[b], distance[a]+w);
  }
}
```

A negative cycle can be detected using the Bellman–Ford algorithm by running the algorithm for n rounds. If the last round reduces any distance, the graph contains a negative cycle. Note that this algorithm can be used to search for a negative cycle in the whole graph regardless of the starting node.

3.1.3 Floyd–Warshall

Computes the shortest path from every node to every other node in $O(n^3)$.

Implementation:

```
for (int i = 1; i <= n; i++) {
  for (int j = 1; j <= n; j++) {
    if (i == j) distance[i][j] = 0;
    else if (adj[i][j]) distance[i][j] = adj[i][j];
    else distance[i][j] = INF;
  }
}

for (int k = 1; k <= n; k++) {
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      distance[i][j] = min(distance[i][j],
      distance[i][k]+distance[k][j]);
    }
  }
}
```

3.2 Disjoint Set (Union-Find)

```
struct UFDS {
    UFDS(int n) : rank(n, 0), parent(n, -1) { }

    int Find(int x) {
        if(parent[x] == -1) return x;
        return parent[x] = Find(parent[x]);
    }

    void Union(int x, int y) {
        int r1 = Find(x);
        int r2 = Find(y);
        if(r1 == r2) return;
        if(rank[r1] < rank[r2]) { parent[r1] = r2; }
        else if(rank[r1] > rank[r2]) { parent[r2] = r1; }
        else {
            parent[r1] = r2;
            rank[r2]++;
        }
    }

private:
    vector<int> rank;
    vector<int> parent;
};
```

3.3 Minimum Spanning Tree

Both Kruskal's and Prim's algorithm incrementally construct an MST from an undirected graph.

3.3.1 Kruskal's

Sort edges based on weight (in ascending order). Go through each edge, add the edge to the MST if it does not introduce a cycle. Use UFDS to determine if the edge will create a cycle, `if Find(edge.from) == Find(edge.to)` => edge introduces a cycle. Otherwise add the edge and perform a union.

Kruskal's used more in practice.

```
using Edge = tuple<int, int, int>;
pair<vector<Edge>, ll> mst(int n, vector<Edge>& edges) {
    sort(edges.begin(), edges.end(), [](Edge& e1, Edge& e2) {
```

```
        return get<2>(e1) < get<2>(e2);
    });

    UFDS uf(n);
    ll weight = 0;
    vector<Edge> mst;
    for(auto& edge : edges) {
        int u, v, c;
        tie(u, v, c) = edge;

        if(uf.Find(u) == uf.Find(v))
            continue;

        uf.Union(u, v);
        weight += c;
        mst.push_back(edge);
    }
    return make_pair(mst, weight);
}
```

3.3.2 Prim's

Similar to dijktra's algorithm. Choose a root node arbitarily as the start. Add a node to the MST via an edge that is closest to the start node that does not cause a cycle (use UDFS). Continue until you have added each node. TODO: Don't think this is quite right

3.4 Topological Sorting

A topological sort is an ordering of the nodes of a directed graph such that if there is a path from node a to node b, then node a appears before node b in the ordering. This requires that there are no cycles in the graph.

Algorithm: push to a stack in post-order DFS, your stack defines the ordering. Alternatively, reverse all the edges and do a post-order DFS.

3.5 Cycle detection

Directed graph

```
enum Colour {
        WHITE,
    GREY,
    BLACK
};

bool detectCycle(vector<vector<int>> &graph, vector<int> &visited, int node) {
    if(visited[node]==GREY) return true;
    if(visited[node]==BLACK) return false;

    visited[node] = GREY;
    for (int c = 0; c < int(graph.size()); ++c)
        if(graph[node][c] && detectCycle(graph,visited,c))
            return true;
    visited[node] = BLACK;
    return false;
}

bool hasCycle(vector<vector<int>> &graph) {
    vector<int> visited(n,WHITE);
```

```
    for (int r = 0; r < n; ++r)
        if(detectCycle(graph, visited, r))
            return true;
    return false;
}
```

Undirected graph

Given a list of edges (for UFDS see 3.2):

```
bool hasCycle(vector<pair<int,int>> &edges, int n) {
        UFDS uf(n);
    for (auto& e : edges) {
        int u = e.first;
        int v = e.second;

        if(uf.Find(u) == Find(v))
            return true;

        uf.Union(u, v);
    }
    return false;
}
```

3.6 Strongly Connected Components

A strongly connected component is defined by a set of vertices that have the property of: Any two vertices within the set are reachable by one another (i.e., you can go from v1 to v2 and from v2 to v1).

In an undirected graph, strongly connected components = connected components.

3.6.1 Kosaraju's algorithm

The idea behind this algorithm is the realisation that within a strongly connected component, a vertex u must have a path from/to other nodes by both forward and backward edges. e.g. if for some vertex u there exists a path to vertex v by only forward edge(s) (via an incoming edge(s) to v), then u and v must not be in the same strongly connected component as v cannot reach u.

Phase 1: perform a postorder traversal on the graph, for each vertex you push it against a stack which defines your ordering (or at the beginning of a list). This is similar to topological ordering, except it's not because there are potentially cycles. Note BFS could be used, but it's easier to do a post-order traversal with a DFS.

Phase 2: reverse the edges. When performing a DFS from some vertex v, the vertices you visit within this DFS will define your strongly connected component.

TODO: implementation

3.7 Flows

3.7.1 Ford Fulkerson algorithm

```
// C++ program for implementation of Ford Fulkerson algorithm
#include <iostream>
#include <limits.h>
#include <string.h>
#include <queue>
using namespace std;
```

```
// Number of vertices in given graph
#define V 6

/* Returns true if there is a path from source 's' to sink 't' in
  residual graph. Also fills parent[] to store the path */
bool bfs(int rGraph[V][V], int s, int t, int parent[])
{
    // Create a visited array and mark all vertices as not visited
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    // Create a queue, enqueue source vertex and mark source vertex
    // as visited
    queue <int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    // Standard BFS Loop
    while (!q.empty())
    {
        int u = q.front();
        q.pop();

        for (int v=0; v<V; v++)
        {
            if (visited[v]==false && rGraph[u][v] > 0)
            {
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }

    // If we reached sink in BFS starting from source, then return
    // true, else false
    return (visited[t] == true);
}

// Returns the maximum flow from s to t in the given graph
int fordFulkerson(int graph[V][V], int s, int t)
{
    int u, v;

    // Create a residual graph and fill the residual graph with
    // given capacities in the original graph as residual capacities
    // in residual graph
    int rGraph[V][V]; // Residual graph where rGraph[i][j] indicates
                    // residual capacity of edge from i to j (if there
                    // is an edge. If rGraph[i][j] is 0, then there is not)
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
             rGraph[u][v] = graph[u][v];

    int parent[V]; // This array is filled by BFS and to store path

    int max_flow = 0; // There is no flow initially

    // Augment the flow while tere is path from source to sink
    while (bfs(rGraph, s, t, parent))
    {
        // Find minimum residual capacity of the edges along the
```

```
        // path filled by BFS. Or we can say find the maximum flow
        // through the path found.
        int path_flow = INT_MAX;
        for (v=t; v!=s; v=parent[v])
        {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }

        // update residual capacities of the edges and reverse edges
        // along the path
        for (v=t; v != s; v=parent[v])
        {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }

        // Add path flow to overall flow
        max_flow += path_flow;
    }

    // Return the overall flow v
    return max_flow;
}

// Driver program to test above functions
int main()
{
    // Let us create a graph shown in the above example
    cout << "The maximum possible flow is " << fordFulkerson(graph, 0, 5) << endl;

    return 0;
}
```

4 Trees

4.1 Order statistic tree

Variant of a BST, where we can perform the following in $O(lgn)$:

- Order(i) => i'th smallest element
- Rank(x) => Number of elements smaller than x (conversely, larger)

We can use a GCC extension:

```
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
using namespace std;
template <typename T>
using ordered_set = tree<T,
                     null_type,
                     less<T>,
                     rb_tree_tag,
                     tree_order_statistics_node_update>;

ordered_set<int> s;
s.insert(1); s.insert(2);

// rank(x)
cout << s.order_of_key(2) << endl; // ouputs 1

// order(i) -> returns iterator
cout << *s.find_by_order(0) << endl; // outputs 1
```

4.2 Diameter

An efficient way to calculate the diameter of a tree is based on two depth first searches. First, we choose an arbitrary node a in the tree and find the farthest node b from a. Then, we find the farthest node c from b. The diameter of the tree is the distance between b and c.

5 Decomposition

5.1 Segment Tree

Simple implementation (no range updates, can update single values):

```
int nearest_pow2(int x) {
    int p = 2;
    while(x > p) p *= 2;
    return p;
}

struct RangeTree {
    RangeTree(const vector<int>& arr) {
        n = nearest_pow2((int)arr.size());

        tree.resize(2*n, 0); // (1), default value, e.g. for min use INT_MAX
        for(int i = 0; i < arr.size(); ++i) {
            tree[i + n] = abs(arr[i]); // (2)
        }

        for(int i = n - 1; i > 0; --i) {
            tree[i] = combine(left(i), right(i));
        }
    }

    int combine(int l, int r) { return gcd(l, r); }
    void update(int i, int x) {
        i += n;
        tree[i] = abs(x); // (3)

        i /= 2;
        while(i >= 1) {
            tree[i] = combine(left(i), right(i));
            i /= 2; // walk up tree
        }
    }
    int query(int i, int j) {
        i += n; j += n;
        int g = tree[i];
        while(i <= j) {
            if(i % 2 == 1) g = combine(g, tree[i++]); // right child
            if(j % 2 == 0) g = combine(g, tree[j--]); // left child
            i /= 2;
            j /= 2;
        }
        return g;
    }

private:
    int left(int i) { return tree[2*i]; }
    int right(int i) { return tree[2*i+1]; }

    vector<int> tree; int n; // (4), e.g. use ll
};
```

To modify for functions other than GCD, please see all lines with numbers associated with them.

TODO: figure out how to allow for range updates (later chapters on Range Tree)

5.2 SQRT Decomposition (Bucketting)

Answer Q queries for a range in an array, where each query asks to evaluate a function, e.g. min, max, GCD.

Simple bucketting is where we segment the array into buckets of size $\sqrt{n}$ (this is the optimal choice). Requires your function to have properties:

- $f(x_1, x_2, ..., x_n) = f(x_1, f(x_2, ..., x_n))$
- $f(x, y) = f(y, x)$ (for below implementation, can modify to not require)

i.e. you need to be able to combine a bucket with a single value. Example with GCD:

Construction:

```
vector<int> arr(n); // ...
const int SQRT_N = sqrt(n);

vector<int> buckets;
for(int i = 0; i < n; ++i) {
  int b = i / SQRT_N;
  if(buckets.size() <= b) {
       buckets.emplace_back(abs(arr[i]));
  }
  buckets[b] = gcd(buckets[b], arr[i]);
}
```

Query $[i, j]$ for $i \leq j$:

```
int bI = i / SQRT_N;
int bJ = j / SQRT_N;

// first bucket
int g = abs(arr[i]);
i++;
while(i < n && i <= j && i / SQRT_N == bI) {
  g = gcd(g, abs(arr[i]));
  ++i;
}

// last bucket
while(j >= i && j / SQRT_N == bJ) {
  g = gcd(g, abs(arr[j]));
  --j;
}

// in-between
for(int b = bI + 1; b < bJ; ++b) {
  g = gcd(buckets[b], g);
}
```

Update value at index i, note we can also update from $[i, j]$ using similar logic as above and below combined:

```
int b = i / SQRT_N;
arr[i] = x;

buckets[b] = abs(arr[b*SQRT_N]);
for(int j = b*SQRT_N; j < min(n, b*SQRT_N + SQRT_N); ++j) {
    buckets[b] = gcd(arr[j], buckets[b]);
}
```

5.3 Mo's Algorithm

Offline algorithm (no updates). Requires to be able to maintain a solution by adding or removing one element from the range $[l, r]$, i.e. go from $[l, r]$ to: $[l-1, r]$, $[l+1, r]$, $[l, r-1]$, or $[l, r+1]$.

Basic idea is to perform SQRT decomposition (bucketting) on the queries. Sort queries, $[l, r]$, based off which bucket l falls in and for each l in the same bucket order r as increasing.

Complexity is $O(N + Qf + Q\sqrt{Q}))$. Where f is the complexity of moving from one solution to the other.

```
vector<int> a(n);
for(auto& x : a) cin >> x;

vector<tuple<int, int, int>> queries;
queries.reserve(q);
for(int i = 0; i < q; ++i) {
  int l, r;
  cin >> l >> r;
  if(r < l) swap(l, r);
  --l; --r;
  queries.emplace_back(l, r, i);
}

const int SQRT_N = sqrt(n);

sort(queries.begin(), queries.end(), [&SQRT_N](auto& a, auto& b) {
    int bI = get<0>(a) / SQRT_N;
    int bJ = get<0>(b) / SQRT_N;
    if(bI == bJ) return get<1>(a) < get<1>(b);
    return bI < bJ;
});

ll sum = 0; // current answer

// helper code to maintain solution
vector<ll> counts(1e6, 0);
auto update = [&sum, &counts](int element, int a) {
    ll& c = counts[element - 1];
    sum += ((c+a)*(c+a) - c*c)*element;
    c += a;
};

// process queries, start with empty range
int lo = 0, hi = -1;

vector<ll> answers(q);
for(auto& query : queries) {
    int start, end, idx;
    tie(start, end, idx) = query;

    while(hi < end) {
        // right increment
        update(a[hi + 1], 1);
        hi++;
    }
    while(hi > end) {
        // right decrement
        update(a[hi], -1);
        --hi;
    }
    while(lo < start) {
      // left increment
      update(a[lo], -1);
```

```
      lo++;
    }
    while(lo > start) {
      // left decrement
      update(a[lo - 1], 1);
      lo--;
    }

    answers[idx] = sum;
}

for(ll a : answers) {
    cout << a << '\n';
}
```

5.4 Functional Graphs (k'th successor)

Functional graph (or successor graph) is a graph where each node has an out-degree of at most 1. There may be cycles. We wish to ask what the k'th successor of a node is, we can decompose k into powers of 2 (binary) and solve it with $lg(n)$ steps, e.g.

$$s(x, 15) = s(s(s(s(x, 8), 4), 2), 1)$$

Where $s(x, k)$ returns the kth successor of x. If you pre-compute $s(x, 2^i)$, for each possible i, we have $lg(n)$ complexity for any (x, k). Pre-computation can be done in $O(NlogK)$ steps where K is the largest path ($K = N - 1$ if the graph has no cycles) and N is the number of nodes in the graph. This is basically fast exponentation, but for the succesor of a node.

Assuming `next` contains the next element for node with index i, also $2^{32} - 1$ is the largest possible value for k (sum of all 2^i for $0 \leq i \leq 31$; should be using unsigned).

Pre-compute:

```
vector<vector<int>> succ;
succ.emplace_back(next);
for(int p = 1; p < 32; ++p) {
  succ.emplace_back();
  succ.back().resize(n);
    for(int i = 0; i < n; ++i) {
    int halfWay = succ[p - 1][i];
    succ[p][i] = succ[p - 1][halfWay];
  }
}
```

Answer query:

```
int j = /* some node */
int k = /* number of steps */;

int loc = j;
for(int p = 0; p < 32; ++p) {
  int pow = 1 << p;
  if(pow & k) {
       loc = succ[p][loc];
  }
}
```

Note, in the above code `succ[p][x]` is the 2^pth successor for node x. Query loops through each possible bit in k in order to decompose it (could use bitset).

5.5 Range Minimum Query (RMQ)

Requirements:

- Static data (not updating values)
- Function requires to have the following properties:
  - $f(x, y) = f(y, x)$
  - $f(x, f(y, z)) = f(x, y, z)$
  - Overlap is fine: $f(x_1, x_2, ..., x_n) = f(f(x_1, x_2, x_3, ..., x_i), f(x_{i-j}, x_{i+2}, ..., x_n)$ for any i, j This overlapping property is required to make queries $O(1)$, otherwise queries will be $O(lgn)$ (see below).

For every sub-array with length that is a power of two, we pre-compute the function, i.e. all sub-arrays of the form $[i, i + 2^x - 1]$ for all valid i and x.

This way, we can simply do an $O(1)$ computation for the query $[l, r]$ by decomposing it into two ranges: $[l, l + 2^i]$ and $[r - 2^i, r]$ where 2^i is the largest power of two not bigger than the length of the range $[l, r]$ $(r - l + 1)$. This is assuming we can overlap the function, example implementation:

```
int lg2(int x) { // 2^(p-1) <= x
    int p = 0;
    while(x > 0) {
        p++; x /= 2;
    }
    return p;
}

template <class F>
vector<vector<ll>> create_rmq(vector<ll>& input, F combine) {
    const int N = input.size();

    vector<vector<ll>> rmq;
    rmq.emplace_back(input);

    const int max_p = lg2(N);
    for(int p = 1; p < max_p; ++p) {
        const int offset = 1 << (p - 1);
        const int num_elements = N - offset;

        rmq.emplace_back();
        rmq.back().resize(num_elements);
        for(int i = 0; i < num_elements; ++i) {
            rmq[p][i] = combine(rmq[p - 1][i], rmq[p - 1][i + offset]);
        }
    }
    return rmq;
}

template <class F>
ll query(vector<vector<ll>>& rmq, int lo, int hi, F combine) {
    const int p = lg2(hi - lo + 1) - 1;
    return combine(rmq[p][lo], rmq[p][hi - (1 << p) + 1]);
}
```

Alternatively, to not require overlapping property, can split it up like so:

```
template <class F>
ll query(vector<vector<ll>>& rmq, int lo, int hi, F combine) {
    const int L = hi - lo + 1;
    ll value = rmq[0][lo]; // (1)
    for(int p = 0; p < 32; ++p) {
        if(L & (1 << p)) {
```

```
            value = combine(value, rmq[p][lo]);
            lo += (1 << p);
        }
    }
    return value;
}
```

If $f(x, x, y) \neq f(x, y)$, then consider changing (1) in the code to some default value, such as `INT_MAX` for min.

6 Dynamic Programming

6.1 Digit DP

There are many types of problems that ask to count the number of integers x between two integers say l and h such that x satisfies a specific property that can be related to its digits.

```
ll f(l, h, i, bool isG = false, bool isS = false) {
    if(base case) { return some_value; }

    int a = !isG ? l[i] : lowest_digit (typically 0);
    int b = !isS ? h[i] : highest_digit (typically 9);
    ll ans = 0;
    for (int y = a; y <= b; ++y)
        ans += f(l, h, i+1, (isG||y!=a), (isS||y!=b));
    return ans;
}

where
l = array of digits in the lower number
h = array of digits in the upper number
i = index of the current digit (starts at most/least significant)
```

6.2 Edit Distance

The edit distance or Levenshtein distance is the minimum number of editing operations needed to transform a string into another string. The allowed editing operations are as follows:

• insert a character (e.g. ABC → ABCA)

• remove a character (e.g. ABC → AC)

• modify a character (e.g. ABC → ADC)

Suppose that we are given a string x of length n and a string y of length m, and we want to calculate the edit distance between x and y. To solve the problem, we define a function distance(a,b) that gives the edit distance between prefixes x[0...a] and y[0...b]. Thus, using this function, the edit distance between x and y equals distance(n−1,m−1).

We can calculate values of distance as follows:

`distance(a,b) = min(distance(a,b −1)+1, distance(a−1,b)+1, distance(a−1,b −1)+cost(a,b))`

Here cost(a,b) = 0 if x[a] = y[b], and otherwise cost(a,b) = 1. The formula considers the following ways to edit the string x:

• distance(a,b −1): insert a character at the end of x

• distance(a−1,b): remove the last character from x

• distance(a−1,b −1): match or modify the last character of x

In the two first cases, one editing operation is needed (insert or remove). In the last case, if x[a] = y[b], we can match the last characters without editing, and otherwise one editing operation is needed (modify).

6.3 Coin change

Given a set of coin values coins = c1, c2,..., ck and a target sum of money n, our task is to form the sum n using as few coins as possible.

```
bool ready[N];
int value[N];

//Time complexity: O(nk), where n is the target sum and k is the number of coins
int solve(int n) {
    if (n < 0) return INF;
    if (n == 0) return 0;
    if (ready[n]) return value[n];
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(n-c)+1);
    }
    value[n] = best;
    ready[n] = true;
    return best;
}
```

Constructing the solution:

```
int first[N];

value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0 && value[x-c]+1 < value[x]) {
            value[x] = value[x-c]+1;
            first[x] = c;
        }
    }
}

while (n > 0) {
    cout << first[n] << "\n";
    n -= first[n];
}
```

6.4 Longest Increasing Subsequence

```
/* Finds longest strictly increasing subsequence. O(n log k) algorithm. */
void find_lis(vector<int> &a, vector<int> &b) {
vector<int> p(a.size());
int u, v;

if (a.empty()) return;

b.push_back(0);

for (size_t i = 1; i < a.size(); i++) {
// If next element a[i] is greater than last element of current longest
// subsequence a[b.back()], just push it at back of "b" and continue
if (a[b.back()] < a[i]) {
p[i] = b.back();
b.push_back(i);
continue;
}
```

```
// Binary search to find the smallest element referenced by b which is
// just bigger than a[i]
// Note : Binary search is performed on b (and not a).
// Size of b is always <=k and hence contributes O(log k) to complexity.
for (u = 0, v = b.size()-1; u < v;) {
int c = (u + v) / 2;
if (a[b[c]] < a[i]) u=c+1; else v=c;
}

// Update b if new value is smaller then previously referenced value
if (a[i] < a[b[u]]) {
if (u > 0) p[i] = b[u-1];
b[u] = i;
}
}

for (u = b.size(), v = b.back(); u--; v = p[v]) b[u] = v;
}
```

7 Mathematics

7.1 Series and Sequences

7.1.1 Faulhaber's

In mathematics, Faulhaber's formula, expresses the sum of the k-th powers of the first n positive integers i.e.

$F(k, n) = 1^k + 2^k + \ldots + n^k$

Implementation:

```
#define rep(i, s, e) for(int i = (s);i < (e);i++)
#define Rep(i, e) for(int i = 0;i < (e);i++)

typedef long long ll;
typedef unsigned long long ull;
typedef pair<ll, ll> P;

const int INF = (int)2e9;
const int MOD = (int)1e9 + 9;
const double EPS = 1e-10;

#define MAX_N 1005

ll combi[MAX_N+2][MAX_N+2]; //aCb = combi[a][b]

void makeCombiMod(){
  rep(i, 0, MAX_N+1) combi[i][0] = 1;
  rep(i, 1, MAX_N+1) combi[0][i] = 0;
  rep(i, 1, MAX_N+1){
        rep(j, 1, i+1) combi[i][j] = (combi[i-1][j-1] + combi[i-1][j]) % MOD;
  }
}

// a x + b y = gcd(a, b)
int extgcd(int a, int b, int &x, int &y) {
  int g = a; x = 1; y = 0;
  if (b != 0) g = extgcd(b, a % b, y, x), y -= (a / b) * x;
  return g;
}
// 1/a mod m
int mod_inverse(int a){
  int x, y;
  extgcd(a, MOD, x, y);
  return (MOD + x % MOD) % MOD;
}

ll B[MAX_N+2];

void initBernoulliMod(){
  makeCombiMod();
  Rep(i, MAX_N+1){
    if(i % 2) B[i] = 0;
    else B[i] = -INF;
  }
  B[0] = 1;
  B[1] = -mod_inverse(2) + MOD;
}

int BernoulliMod(int n){
  if(B[n] == -INF){
```

```
    ll sum = (1 + combi[n+1][1] * B[1]) % MOD;
    for(int i = 2; i < n; i+= 2) sum = (sum + (combi[n+1][i] * BernoulliMod(i) % MOD)) %
        MOD;
    B[n] = sum * mod_inverse(n+1) % MOD;
    B[n] *= -1;
    while(B[n] < 0) B[n] += MOD;
  }
  return B[n];
}

//n^p % m
int powMod(ll n, int p){
  ll ans = 1, ln = n % MOD;
  if(p <= 0) return 1;
  while(p != 0){
        if((p & 1) == 1) ans = (ans*ln) % MOD;
        ln = (ln * ln) % MOD;
        p /= 2;
  }
  return (int)ans;
}

//Faulhaber's formula
//S(k, n) = 1^k + 2^k + .. + n^k
int faulhaberMod(int k, ll n){
  ll ans = 0;
  Rep(j, k+1){
    if(j % 2) ans = (ans - ((combi[k+1][j] * BernoulliMod(j) % MOD) * powMod(n, k+1-j) %
        MOD) + MOD) % MOD;
    else ans = (ans + ((combi[k+1][j] * BernoulliMod(j) % MOD) * powMod(n, k+1-j) % MOD))
        % MOD;
  }
  ans = ans * mod_inverse(k+1) % MOD;
  return (int)ans;
}

int main() {
ll n; int k;
       cin >> n >> k;
     int result = faulhaberMod(k,n);
     cout << result << endl;
   }
   return 0;
}
```

7.2 Number Theory

7.2.1 Prime Numbers

A number is said to be prime if it is divisible by 1 or itself.

Checking if a number if prime in $O(\sqrt{n})$:

```
bool isPrime (int n) {
  if (n<=1) return false;
  if (n==2) return true;
  if (n%2==0) return false;
  int m = sqrt(n);

  for (int i=3; i<=m; i+=2) {
       if (n%i==0) return false;
  }
  return true;
}
```

If multiple numbers (up to n) are to be checked, maintain a list using the Sieve of Eratosthenes. Can also be used to count the number of factors a number has.

Complexity is $O(\sqrt{n}log(\sqrt{n}))$.

```
vector<bool> sieve(int n) {
  vector<bool> prime(n+1, true);
  prime[0] = prime[1] = false;
  for (int i=2; i*i<=m; i++) {
    if (prime[i]) {
      for (int k=i*i; k<=n; k+=i) {
          prime[k]=false;
      }
        }
  }
  return prime;
}
```

Modified Sieve's for counting number of factors ($O(nlogn)$):

```
const int N = 1000000;
vector<int> sieve(N + 1, 0);
for(int i = 1; i <= N; ++i) {
  for(int j = i; j <= N; j += i) {
    sieve[j] += 1;
  }
}
```

7.2.2 Prime Factorization

For example, $24 = 2 * 2 * 2 * 3$, so the result of the function is [2,2,2,3].

```
vector<int> factors(int n) {
  vector<int> f;
  for (int x = 2; x*x <= n; x++) {
    while (n%x == 0) {
      f.push_back(x);
      n /= x;
    }
  }
  if (n > 1) f.push_back(n);
  return f;
}
```

7.2.3 Factors

Prime factorization of a number:

$n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$

The number of factors of n:

$$f(n) = \prod_{i=1}^{k} (a_i + 1)$$

The sum of factors of n:

$$\begin{aligned} \mu(n) &= \prod_{i=1}^{k} (1 + p_i + \ldots + p_i^{a_i}) \\ &= \prod_{i=1}^{k} \frac{p_i^{a_i+1} - 1}{p_i - 1} \end{aligned}$$

The product of factors of n:

$$p(n) = n^{\frac{f(n)}{2}}$$

7.2.4 Co-Primes

Euler's totient function gives the number of co-prime numbers to n between 1 and n.

$$t(n) = \prod_{i=1}^{k} p_i^{a_i - 1}(p_i - 1)$$

Note, if n is prime $t(n) = n - 1$.

7.2.5 GCD/HCF

The greatest common divisor (GCD) of two numbers a and b is the greatest number that divides evenly into both a and b.

In most practical cases GCD can be computed using the inbuilt `__gcd(x,y)` function (note there are 2 underscores).

Alternatively, GCD can be calculated using Euclidean algorithm as follows:

Euclidean Algorithm

```
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }
```

7.2.6 LCM

Lowest common multiple of 2 numbers can be computed using following:

```
int lcm(int a, int b) { return (a*b)/gcd(a,b); }
```

7.2.7 Modular Arithmetic

$$\begin{aligned}(x+y) \mod m &= (x \mod M + y \mod m) \mod m \\ (x-y) \mod m &= (x \mod M - y \mod m) \mod m \\ (xy) \mod m &= (x \mod m)(y \mod m) \mod m \\ x^n \mod m &= (x \mod m)^n \mod m\end{aligned}$$

The following function calculates the value of $x^n \mod M$ in $O(logn)$:

```
int modPow(int x, int n, int m) {
  if (n == 0) return 1;
  int u = modPow(x, n/2, m);
  u *= u;
  if (n % 2 == 1) u = (u%m) * (x%m);
  return u % m;
}
```

Fermat's Theorem

$x^{m-1} \mod m = 1$ when m is prime and x and m are co-prime.
This also yields $x^k \mod m = x^{k \mod (m-1)} \mod m$.

Euler's theorem

$x^{f(m)} \mod m = 1$ when x and m are co-prime.
Fermat's theorem follows from Euler's theorem, because if m is a prime, then $f(m) = m - 1$.

Modular Multiplicative Inverse

Multiplicative inverse of $A \mod M$ is a number X (A^{-1}) such that

$(A * X) \equiv 1 \mod M$
$(A * X) \mod M = 1$

Only the numbers co-prime to M have a modular inverse (mod M).

If M is prime, then the inverse is defined as $modPow(x, M-2, M)$, otherwise you must use Extended Euler's Algorithm.

Extended Euclidean Algorithm

TODO: convert to recursive for readability?

```
int modInverse(int a, int m) {
    int m0 = m; int y = 0, x = 1;
    if (m == 1) return 0;
    while (a > 1) {
      int q = a / m; int t = m;
      m = a % m, a = t;
      t = y;
      y = x - q * y;
      x = t;
    }
    if (x < 0) x += m0;
    return x;
}
```

Mod inverse of a is x in the above code.

7.3 Solving Equations

7.3.1 Linear Recurrences

A linear recurrence is when each term of a sequence is a linear function of earlier terms in the sequence.

1. Find a particular solution

   Consider the example:

   $f(x) = f(x-1) + 3, f(0) = 2$

   Try replacing f(x) with a constant k and solve for k.

   $k = k + 3$ $0 = 3$

   Well that did not work! Now try $f(x) = kx$

   $kx = k(x-1) + 3$, $kx = kx - k + 3$, $k = 3$

   The particular solution is 3. If $f(x) = kx$ does not work, try $f(x) = k(x^2)$ and so on.

2. Find a general solution to the homogeneous equation using characteristic equation

   Consider the example:

   $f(x) = 2.f(x-1) + 24.f(x-3) + 5$

   Homogeneous equation: $f(x) = 2.f(x-1)+24.f(x-3)$ Characteristic equation: $n^3 = 2n^2+24$ Solve for n. Let's say n = a,a,b in this case.

   General solution to homogeneous equation: $(qx + r)(a^x) + s(b^x)$

   where q,r,s are some unknown constants. Note that if the number of occurrences of a in the solution were 3, the coefficient would be $(qx^2 + rx + m)$ and so on.

3. Add the particular solution to general solution found in step 2

   $f(x) = (equation : step2) + (particular solution : Step1)$

   E.g. $f(x) = (qx + r)(a^x) + s(b^x) + 3$

4. Find general solution to non-homogeneous equation using the base case.

7.3.2 Linear Diophantine

A linear Diophantine equation is of the form ax + by = c. A Diophantine equation can be solved if c is divisible by gcd(a,b), and otherwise it cannot be solved.

r	u	v	q
$r1 = a$	$u1 = 1$	$v1 = 0$	$-$
$r2 = b$	$u2 = 0$	$v2 = 1$	$q2 = r1/r2$
$r3 = r1\%r2$	$u3 = u1 - u2 = 1$	$v3 = v1 - v2 = 1$	$q3 = r2/r3$
$r4 = r2\%r3$	$u4 = u2 - u3$	$v4 = v2 - v3$	$q4 = r3/r4$
.	.	.	.
.	.	.	.
$rn = gcd(a, b)$	un	vn	qn

$rn = gcd(x, y) = un.a + vn.b$

Therefore, a particular solution is:

$x = un, y = vn$

General solution:

$x = un + \frac{kb}{gcd(a,b)}, y = vn + \frac{ka}{gcd(a,b)}$

for all integers k.

7.3.3 Gaussian Elimination

Solves system of linear equations of the form AX=B.

Input: Matrix A|B i.e.

a00	a01	.	b0
a10	a11	.	b1
.	.	.	.
ar0	.	.	br

Output: Vector X

```
vector<rational> gaussian_elimination(vector<vector<rational>> A) {
    int n = A.size();

    for (int i=0; i<n; i++) {
        rational maxEl = A[i][i].mag();
        int maxRow = i;
        for (int k=i+1; k<n; k++) {
            if (A[k][i].mag() > maxEl) {
                maxEl = A[k][i].mag();
                maxRow = k;
            }
        }
        for (int k=i; k<n+1;k++) {
            rational tmp = A[maxRow][k];
            A[maxRow][k] = A[i][k];
            A[i][k] = tmp;
        }

        for (int k=i+1; k<n; k++) {
            rational c = -A[k][i]/A[i][i];
            for (int j=i; j<n+1; j++) {
                if (i==j) {
                    A[k][j] = 0;
                } else {
                    A[k][j] = A[k][j] + (rational(c) * A[i][j]);//can cause overflow
                }
            }
        }
    }

    vector<rational> x(n);
    for (int i=n-1; i>=0; i--) {
        x[i] = A[i][n]/A[i][i];
        for (int k=i-1;k>=0; k--) {
            A[k][n] = A[k][n] - (A[k][i] * x[i]);
        }
    }
    return x;
}
```

7.4 Matrix Multiplication/Exponentation

```
constexpr int64_t kModuloValue = 1e9 + 7;

template<typename T>
```

```
struct Mat {
  T &operator()(size_t i, size_t j) {
    return m[i * cols + j];
  }
  const T &operator()(size_t i, size_t j) const {
    return m[i * cols + j];
  }
  Mat(size_t r, size_t c, T val = 0)
    : cols(c), rows(r), m(c * r, val) { }

  Mat operator*(const Mat &that) const {
    if (cols != that.rows) {
      std::cerr << "Error: incompatible dims." << std::endl;
      exit(1);
    }
    Mat next(rows, that.cols);
    for (size_t i = 0; i < rows; i++) {
      for (size_t j = 0; j < that.cols; j++) {
        for (size_t k = 0; k < cols; k++) {
          next(i,j) += (((*this)(i, k) * that(k, j)) % kModuloValue);
          next(i,j) %= kModuloValue;
        }
      }
    }
    return next;
  }
  size_t cols, rows;
  std::vector<T> m;
};

template<typename T>
Mat<T> Pow(const Mat<T> &m, size_t n) {
  if (m.cols != m.rows) {
    std::cerr << "Error: not square." << std::endl;
    exit(1);
  }
  if (n == 0) {
    Mat<T> eye(m.cols, m.cols);
    for (size_t i = 0; i < m.cols; i++) {
      eye(i,i) = 1;
    }
    return eye;
  }
  if (n == 1) return m;
  Mat<T> next = Pow(m, n/2);
  next = next * next;
  if (n%2) return next * m;
  return next;
}
```

7.5 Simplex

```
// Two-phase simplex algorithm for solving linear programs of the form
//
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
```

```
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
//
// To use this code, create an SimplexSolver object with A, b, and c as
// arguments.  Then, call Solve(x).
struct SimplexSolver {
int m, n;
vi B, N;
vve D;
SimplexSolver(const vve &A, const ve &b, const ve &c) : m(b.size()), n(c.size()), N(n + 1), B(m),
for (int i = 0; i < m; ++i) for (int j = 0; j < n; ++j) D[i][j] = A[i][j];
for (int i = 0; i < m; ++i) { B[i] = n + i; D[i][n] = -1; D[i][n + 1] = b[i]; }
for (int j = 0; j < n; ++j) { N[j] = j; D[m][j] = -c[j]; }
N[n] = -1; D[m + 1][n] = 1;
}
void pivot(int r, int s) {
for (int i = 0; i < m + 2; ++i) if (i != r)
for (int j = 0; j < n + 2; ++j) if (j != s)
D[i][j] -= D[r][j] * D[i][s] / D[r][s];
for (int j = 0; j < n + 2; ++j) if (j != s) D[r][j] /= D[r][s];
for (int i = 0; i < m + 2; ++i) if (i != r) D[i][s] /= -D[r][s];
D[r][s] = el_t(1) / D[r][s];
swap(B[r], N[s]);
}
bool simplex(int phase) {
int x = phase == 1 ? m + 1 : m;
while (true) {
int s = -1;
for (int j = 0; j <= n; ++j) {
if (phase == 2 && N[j] == -1) continue;
if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s]) s = j;
}
if (-EPS < D[x][s]) return true;
int r = -1;
for (int i = 0; i < m; ++i) {
if (D[i][s] < EPS) continue;
if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] || (D[i][n + 1] / D[i][s]) == (D[r][n
}
if (r == -1) return false;
pivot(r, s);
}
}
el_t solve(ve &x) {
int r = 0;
for (int i = 1; i < m; ++i) if (D[i][n + 1] < D[r][n + 1]) r = i;
if (D[r][n + 1] < -EPS) {
pivot(r, n);
if (!simplex(1) || D[m + 1][n + 1] < -EPS) return -numeric_limits<el_t>::infinity();
for (int i = 0; i < m; ++i) if (B[i] == -1) {
int s = -1;
for (int j = 0; j <= n; ++j)
if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s]) s = j;
```

```
pivot(i, s);
}
}
if (!simplex(2)) return numeric_limits<el_t>::infinity();
x = ve(n);
for (int i = 0; i < m; ++i) if (B[i] < n) x[B[i]] = D[i][n + 1];
return D[m][n + 1];
}
};
```

7.6 Numerics

7.6.1 Base Conversion

Converting numbers in base 'from' to base 'to':

```
int convertBase(int n, int from, int to) {
  int result = 0;
  int multiplier = 1;

  while(n > 0) {
    result += (n % to) * multiplier;
    multiplier *= from;
    n /= to;
  }
  return result;
}
```

e.g. 100 (in decimal) to base 3: convertBase(100, 10, 3) = 10201

For bases > 10, require to input a set of characters to represent a number.

7.6.2 Fractions

Fractions can be represented as a struct with overloaded operators. If using modular arithmetic, remove all the gcd calls and add mod.

```
struct rational {
    long long num, den;
    rational() : num(0), den(1) {}
    rational(long long num) : num(num), den(1) {}
    rational(long long num, long long den) : num(num), den(den) {
        if (den == 0) throw "Division by zero"; // might wanna change this to den = 1
        if (num == 0) den = 1;
        if (den < 0) {
            num *= -1;
            den *= -1;
        }
        reduce();
    }
    rational(const rational& cpy) : rational(cpy.num, cpy.den) {}
    void reduce() {
        long long div = abs(__gcd(num, den));
        num /= div;
        den /= div;
        if (den < 0) {
            num *= -1;
            den *= -1;
        }
    }
    rational operator+(const rational& rhs) const {
```

```
        long long n, d, div = __gcd(den, rhs.den);
        n = num*(rhs.den/div)+rhs.num*(den/div);
        d = den/div*rhs.den;
        return rational(n, d);
    }
    rational operator-() const {
        return rational(-num, den);
    }
    rational operator-(const rational& rhs) const {
        return (*this)+(-rhs);
    }
    rational operator*(const rational& rhs) const {
        if (num == 0 || rhs.num == 0) return rational(0,1);
        long long n, d, diva = __gcd(num, rhs.den), divb = __gcd(rhs.num, den);
        n = (num/diva)*(rhs.num/divb);
        d = (den/divb)*(rhs.den/diva);
        return rational(n, d);
    }
    rational operator/(const rational& rhs) const {
        return (*this)*rational(rhs.den, rhs.num);
    }
    bool operator<(const rational& rhs) const {
        long long div = __gcd(den, rhs.den);
        return rhs.den/div*num < den/div*rhs.num;
    }

    bool operator>(const rational& rhs) const {
        long long div = __gcd(den, rhs.den);
        return rhs.den/div*num > den/div*rhs.num;
    }

    bool operator==(const rational& rhs) const {
        return num==rhs.num && den==rhs.den;
    }

    rational mag() {
        return rational(abs(num), abs(den));
    }

    friend ostream& operator<<(ostream& os, const rational& rhs) {
        rational out(rhs);
        out.reduce();
        os << out.num << '/' << out.den;
        return os;
    }
};
```

7.7 Permutation and Combination

The number of ways to select r objects from a set of size n if the objects are:

	No Repetition	Repetition
Ordered	$P_r^n = \frac{n!}{(n-r)!}$	n^r
Unrdered	$C_r^n = \frac{n!}{r!(n-r)!}$	$C_r^{r+(n-1)}$

The number of ways to allocate r objects into n containers:

Objects are	Containers are	Restriction on containers	Number of ways
Distinct	Distinct	none	n^r
		non-empty	$A(r,n)$
Identical	Distinct	none	$C_r^{r+(n-1)}$
		non-empty	C_{n-1}^{r-1}
Identical	Identical	none	$S(r+n,n)$
		non-empty	$S(r,n)$

where

$$A(r,n) = \begin{cases} 0 & r < n \\ 1 & n = 1 \\ 1 & n = r \\ 2^r - 2 & n = 2 \\ \sum_{m=1}^{r-n+1} C_m^r A(r-m, n-1) & otherwise \end{cases}$$

$$S(r,n) = \begin{cases} 0 & r < n \\ 1 & n = 1 \\ 1 & n = r \\ 1 & n = r-1 \\ \left\lfloor \frac{r}{2} \right\rfloor & n = 2 \\ \sum_{i=1}^{n} S(r-n, i) & otherwise \end{cases}$$

Also,

$C_r^n = C_{r-1}^{n-1} + C_r^{n-1}, C_0^n = C_n^n = 1$

$C_r^n = C_{n-r}^n$

8 Computational Geometry

8.1 Geometry

8.1.1 Convex Hull

In mathematics, the convex hull or convex envelope or convex closure of a set X of points in the Euclidean plane or in a Euclidean space (or, more generally, in an affine space over the reals) is the smallest convex set that contains X.

Implementation of Andrew's monotone chain 2D convex hull algorithm. Asymptotic complexity: O(n log n).

```
typedef double coord_t;      // coordinate type
typedef double coord2_t; // must hold 2*max(|coordinate|)^2

struct Point {
    coord_t x, y;

    bool operator <(const Point &p) const {
        return x < p.x || (x == p.x && y < p.y);
    }
};

/* 2D cross product of OA and OB vectors, i.e. z-component of their 3D cross product.
    Returns a +ve value, if OAB makes a counter-clockwise turn, -ve for clockwise turn,
    and zero if the points are collinear. */
coord2_t cross(const Point &O, const Point &A, const Point &B) {
    return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

// Returns a list of points on the convex hull in counter-clockwise order (first and last
    point in the list are the same)
vector<Point> convex_hull(vector<Point> P) {
    size_t n = P.size(), k = 0;
    if (n <= 3) return P;
    vector<Point> H(2*n);
    sort(P.begin(), P.end());

    // Build lower hull
    for (size_t i = 0; i < n; ++i) {
        while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
        H[k++] = P[i];
    }

    // Build upper hull
    for (size_t i = n-1, t = k+1; i > 0; --i) {
        while (k >= t && cross(H[k-2], H[k-1], P[i-1]) <= 0) k--;
        H[k++] = P[i-1];
    }

    H.resize(k-1);
    return H;
}
```

8.1.2 Polygons

Area of Lattice Polygon

Polygons whose vertices have integer coordinates are known as lattice polygons. Area of such polygon is given by:

$$\frac{1}{2}|\sum_{i=1}^{n-1} x_i(y_i+1) - (x_i+1)y_i|$$

The points are supposed to be in an order such that vertices are adjacent and $p_1 = p_n$. Area can also be given by:

$$B/2 + I - 1$$

where B = number of lattice points on the boundary of the polygon and I = number of lattice points in the interior of the polygon.

Euler's Formula for Polygonal Nets

Planar graph: Look

A polygonal net is a simple polygon divided into smaller polygons. The smaller polygons are called faces, the sides of the faces are called edges and the vertices of the faces are called vertices. Euler's Formula then states:

$$V - E + F = 2$$

where V = number of vertices, E = number of edges and F = number of faces.

For example, consider a square with both diagonals drawn. We have $V = 5$, $E = 8$ and $F = 5$ (the outside of the square is also a face) and so $V \check{} E + F = 2$.

Intersecting Rectangles

Suppose we have two rectangles R1 and R2. Let (x1, y1) be the location of the bottom-left corner of R1 and (x2, y2) be the location of its top-right corner. Similarly, let (x3, y3) and (x4, y4) be the respective corner locations for R2.

The intersection of R1 and R2 will be a rectangle R3 whose bottom-left corner is at (max(x1, x3), max(y1, y3)) and top-right corner at (min(x2, x4), min(y2, y4)).

If max(x1, x3) > min(x2, x4) or max(y1, y3) > min(y2, y4) then R3 does not exist, ie R1 and R2 do not intersect.

This method can be extended to intersection in more than 2 dimensions.

Useful code

```
typedef long double coord_t;

struct Point {
    coord_t x, y;
    Point() : x(0), y(0) {}
    Point(coord_t x, coord_t y) : x(x), y(y) {}
    Point operator+(const Point& rhs) const {
        return Point(x + rhs.x, y + rhs.y);
    }
    Point operator-(const Point& rhs) const {
        return Point(x - rhs.x, y - rhs.y);
    }
    Point operator*(const coord_t rhs) const {
        return Point(x * rhs, y * rhs);
    }
    Point operator/(const coord_t rhs) const {
```

```
        return Point(x / rhs, y / rhs);
    }
    coord_t operator*(const Point& rhs) const {
        return (x * rhs.x) + (y * rhs.y);
    }
    coord_t cross(const Point& rhs) const {
        return (x * rhs.y) - (y * rhs.x);
    }
    Point norm() {
        return Point(-y, x);
    }
    coord_t sqmag() {
        return x*x + y*y;
    }
    coord_t mag() {
        return sqrt(sqmag());
    }
    Point unit() {
        coord_t m = mag();
        return Point(x/m, y/m);
    }
    bool operator<(const Point& rhs) const {
        if (x == rhs.x) return y < rhs.y;
        else return x < rhs.x;
    }
    bool operator>(const Point& rhs) const {
        if (x == rhs.x) return y > rhs.y;
        else return x > rhs.x;
    }
    bool operator==(const Point& rhs) const {
        return x == rhs.x && y == rhs.y;
    }
    bool operator!=(const Point& rhs) const {
        return x != rhs.x || y != rhs.y;
    }
    friend ostream& operator<<(ostream& os, const Point& rhs) {
        os << "(" << rhs.x << ", " << rhs.y << ")";
        return os;
    }
};

typedef vector<Point> Polygon;

coord_t poly_twice_area(const Polygon& poly) {
    coord_t twice_area = 0;
    Point prev = poly.back();
    for (Point curr : poly) {
        twice_area += prev.cross(curr);
        prev = curr;
    }
    return twice_area;
}
coord_t poly_area(const Polygon& poly) { return poly_twice_area(poly) / 2; }

pair<int, bool> poly_winding(const Polygon& poly, Point pt) {
    int wn = 0;
    Point prev = poly.back() - pt;
    for (Point curr : poly) {
        curr = curr - pt;
        if (prev.cross(curr) == 0 && prev * curr <= 0) return pair<int, bool>(0, true);
        else if (prev.y <= 0 && 0 < curr.y && prev.cross(curr) > 0) ++wn;
        else if (curr.y <= 0 && 0 < prev.y && prev.cross(curr) < 0) --wn;
        prev = curr;
```

```
    }
    return pair<int, bool>(wn, false);
}

pair<int, bool> poly_curl(const Polygon& poly) {
    int curl = 0;
    Point ref = poly[1] - poly[0];
    Point srt = poly.back(), end = poly.front();
    for (Point cur : poly) {
        if (cur == end) continue;
        if ((cur-end).cross(end-srt) == 0 && (cur-end) * (end-srt) < 0) return pair<int,
            bool>(0, true);
        else if (0 < ref.cross(cur-end) && ref.cross(end-srt) <= 0 && 0 < (end-srt).cross(
            cur-end)) ++curl;
        else if (0 < ref.cross(end-srt) && ref.cross(cur-end) <= 0 && 0 < (cur-end).cross(
            end-srt)) --curl;
        srt = end; end = cur;
    }
    return pair<int, bool>(curl, false);
}

Point poly_centroid(const Polygon& poly) {
    Point num;
    coord_t den = 0;
    Point prev = poly.back();
    for (Point curr : poly) {
        num = num + (prev + curr) * prev.cross(curr);
        den += prev.cross(curr);
        prev = curr;
    }
    return num / den / 3;
}

Polygon convex_hull(vector<Point> pts) {
    sort(pts.begin(), pts.end());
    Polygon upr, lwr;
    for (Point pt : pts) {
        while (upr.size() >= 2 && (upr.back() - upr[upr.size()-2]).cross(pt - upr.back())
            >= 0)
           upr.pop_back();
        upr.push_back(pt);
        while (lwr.size() >= 2 && (lwr.back() - lwr[lwr.size()-2]).cross(pt - lwr.back())
            <= 0)
           lwr.pop_back();
        lwr.push_back(pt);
    }
    reverse(upr.begin(), upr.end());
    upr.pop_back(); lwr.pop_back();
    lwr.insert(lwr.end(), upr.begin(), upr.end());
    return lwr;
}
```

9 Greedy Algorithms

9.1 Scheduling

Many scheduling problems can be solved using greedy algorithms. A classic problem is as follows: Given n events with their starting and ending times, find a schedule that includes as many events as possible. It is not possible to select an event partially

The idea is to always select the next possible event that ends as early as possible. It turns out that this algorithm always produces an optimal solution. The reason for this is that it is always an optimal choice to first select an event that ends as early as possible. After this, it is an optimal choice to select the next event using the same strategy, etc., until we cannot select any more events

9.2 Tasks And Deadlines

Consider a problem where we are given n tasks with durations and deadlines and our task is to choose an order to perform the tasks. For each task, we earn d − x points where d is the task's deadline and x is the moment when we finish the task. What is the largest possible total score we can obtain?

Surprisingly, the optimal solution to the problem does not depend on the deadlines at all, but a correct greedy strategy is to simply perform the tasks sorted by their durations in increasing order. The reason for this is that if we ever perform two tasks one after another such that the first task takes longer than the second task, we can obtain a better solution if we swap the tasks.

9.3 Minimising Sums

We next consider a problem where we are given n numbers a1,a2,...,an and our task is to find a value x that minimizes the sum

$|a_1 - x|^c + |a_2 - x|^c + ... + |a_n - x|^c$

Case c = 1

The best choice for x is the median of the numbers.

Case c = 2

The best choice for x is the average of the numbers.

10 Game Theory

10.1 Sprague–Grundy theorem

If a game meets the following criteria:

• There are two players who move alternately.

• The game consists of states, and the possible moves in a state do not depend on whose turn it is i.e. an impartial game. Classic games like chess, tic-tac-toe etc. are NOT impartial (they are partisan games).

• The game ends when a player cannot make a move.

• The game surely ends sooner or later and there is a clear outcome win or lose (and maybe draw).

• The players have complete information about the states and allowed moves, and there is no randomness in the game.

then, assuming both the players play optimally, we can predict who is going to win without simulating the game.

1. Break the composite game into sub-games.
2. Then for each sub-game, calculate the Grundy Number for the current state.
3. Then calculate the XOR of all the calculated Grundy Numbers.
4. If the XOR value is non-zero, then the player who is going to make the turn (first player) will win else he is destined to lose, no matter what.

Grundy Numbers

The Grundy Number is equal to

• 0 for a game that is lost immediately by the first player, otherwise

• Mex of the Grundy Numbers of all possible next states for any other game.

Mex

Minimum excludant a.k.a Mex of a set of numbers is the smallest non-negative number not present in the set.

```
int calcMex(unordered_set<int> &Set) {
    int Mex = 0;
    while (Set.find(Mex) != Set.end())
        Mex++;
    return (Mex);
}
```

11 Miscellaneous

11.1 Sliding window

A sliding window is a constant-size subarray that moves from left to right through the array. At each window position, we want to calculate some information about the elements inside the window.

In example below, we focus on the problem of maintaining the sliding window minimum, which means that we should report the smallest value inside each window.

```
int n;            // size of array
int w;            //window size
deque<int> window; //stores indices

//initialising the window
for (int i = 0; i < w; ++i) {
    while(!window.empty() && d[window.back()] > array[i]) window.pop_back();
    window.push_back(i);
}

//the first element contains the minimum for the initial window

//sliding
for (int i = 1; i+w < n; ++i) {
    while(!window.empty() && array[window.back()] > array[i+w-1]) window.pop_back();
    while(!window.empty() && window.front() < i) window.pop_front();
    window.push_back(i+w-1);

    //the first element i.e. array[window.front()] contains the minimum for this range
}
```