ICPC Cookbook *perocoders*

Table of contents:

*** Modular Exponentiation (a^k)%mod**

```
ll mpow(ll a, ll k){
    ll ans = 1;
    while(k){
       if(k&1)ans = ans*a%mod;
       a = a*a%mod;
       k >>= 1;
    }
    return ans;
}
```

*** nCr w/o mod - O(n)(only particular)**

```
ll ncr(ll n, ll r) {
    ll res = 1;
    r = min(n-r,r);
    for (ll i = 0; i < r; i++) {
        res = res * (n-i);
        res = res / (i+1);
    }
    return res;
}
```

*** nCr w/ mod Time- O(n), Space- O(n)**

(only particular)

// make sure "mod" is prime

```
ll nCr(ll n, ll r)
{
   vector<ll> f(n + 1,1);
   for (ll i=2; i<=n;i++)
       f[i]= (f[i-1]*i) % mod;
    ll a = mpow(f[r], mod-2);
    ll b =  mpow(f[n-r], mod-2);
    ll c = f[n];
   return (c*((a * b) % mod)) % mod;
}
```

*** Modular Invers**

```
// Useful for large n in nCr w/ mod
ll modular_inverse(ll n, ll mod){
   return mpow(n, mod-2);
}
```

*** nCr w/ mod Time- O(n*k) Space- O(k)**
(only particular)

```
ll nck(ll n, ll k)
{
   ll C[k+1];
   memset(C, 0, sizeof(C));
    C[0] = 1 % mod;
   for (ll i = 1; i <= n; i++)
   {
       for (ll j = min(i, k); j > 0; j--)
           C[j] = (C[j] + C[j-1]) % mod;
   }
   return C[k];
}
```

*** nCr w/ mod Time- O(n*k) Space- O(n*k)**
(all intermediates are calculated)

```
ll nck(ll n, ll k)
{
   ll C[n+1][k+1];
   ll i, j;
   for (i = 0; i <= n; i++)
   {
       for (j = 0; j <= min(i, k); j++)
       {
           if (j == 0 || j == i)
               C[i][j] = 1 % mod;
           else
               C[i][j] = (C[i-1][j-1] +
C[i-1][j]) % mod;
       }
   }
   return C[n][k];
}
```

*** To check power of 2**

```
bool isPower2(int v){
    return (v & (v - 1)) == 0;
}
```

*** Sieve for Primes**

```cpp
void SieveOfEratosthenes(int n)
{
    bool prime[n+1];
    memset(prime, true, sizeof(prime));
    for (int p=2; p*p<=n; p++) {
        if (prime[p] == true) {
            for (int i=p*p; i<=n; i += p)
                prime[i] = false;
        }
    }
```

*** Sieve in O(n)**

```cpp
const long long MAX_SIZE = 1000001;
vector<long long >isprime(MAX_SIZE ,
true);
vector<long long >prime;
vector<long long >SPF(MAX_SIZE);
void manipulated_seive(int N) {
    isprime[0] = isprime[1] = false ;
    for (long long int i=2; i<N ; i++) {
        if (isprime[i]) {
            prime.push_back(i);
            SPF[i] = i;
        }
        for (long long int j=0;
            j < (int)prime.size() &&
            i*prime[j] < N && prime[j] <=
SPF[i];j++) {
            isprime[i*prime[j]]=false;
            SPF[i*prime[j]] = prime[j] ;
        }
    }
}
```

*** Dijkstra's Algorithm SSSP O(V + E*logV)**
// Initialize dist[] with INT_MAX
// P[x] is list of pair of vertices Adjacent to x with weight. {e,w}

```cpp
void Dijkstra(ll src,ll dist[],bool
vis[]){
    dist[src] = 0;
    multiset<pair<ll,ll> > s;
    s.insert(make_pair(0,src));
    while(!s.empty()){
        pair<ll,ll> p = *s.begin();
        s.erase(s.begin());
        ll x = p.second; ll wei = p.first;
        if(vis[x]) continue;
        vis[x] = true;
        auto itr = P[x].begin();
        while(itr!=P[x].end()){
            ll e = (*itr).first;
            ll w = (*itr).second;
            if(dist[x] + w < dist[e]){
                dist[e] = dist[x] + w;

s.insert(make_pair(dist[e],e));
            }
            itr++;
        }
    }
}
```

*** Bellman Ford's Algorithm SSSP O(V*E)**

// Initialize dist[] with INT_MAX

```
void BellmanFord(int src,int dist[]){
    dist[src] = 0;
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            auto itr = P[j].begin();
            while(itr!=P[j].end()){
                int u = j;
                int v = (*itr).first;
                int weight = (*itr).second;
                if (dist[u] != INF &&
                    dist[u] + weight < dist[v])
                    dist[v] = dist[u] + weight;
                itr++;
            }
        }
    }
   for(int j=0;j<n;j++){
        auto itr = P[j].begin();
        while(itr!=P[j].end()){
            int u = j;
            int v = (*itr).first;
            int weight = (*itr).second;
            if (dist[u] != INF &&
                dist[u] + weight < dist[v])
                printf("negative weight cycle");
                itr++;
        }
    }
}
```

*** Floyd-Warshall's Algorithm APSP O(V^3)**

```
// Don't forget to fill dist[][] before calling this procedure
void FloydWarshall(){
    for(int k = 0; k < n; k++)
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                dist[i][j] = min( dist[i][j], dist[i][k] + dist[k][j] );
}
```

*** Kruskal's Algorithm MST O(E*logV)**

```
// Sort edges in preprocessing
int id[MAX], nodes, edges;
pair <long long, pair<int, int>> p[MAX];
void initialize(){
    for(int i = 0;i < MAX;++i)
        id[i] = i;
}
int root(int x){
    while(id[x] != x){
        id[x] = id[id[x]]; x = id[x];
    }
    return x;
}
void union1(int x, int y){
    int p = root(x);
    int q = root(y);
    id[p] = id[q];
}
bool find(int A,int B) {
    if( root(A)==root(B) )
       return true;
    else
         return false;
}
long long kruskal(pair<long long,
pair<int, int> > p[]){
    int x, y;
    long long cost, minimumCost = 0;
    for(int i = 0;i < edges;++i){
        x = p[i].second.first;
        y = p[i].second.second;
        cost = p[i].first;
        if(root(x) != root(y)){
            minimumCost += cost;
            union1(x, y);
        }
    }
    return minimumCost;
}
```

*** Prim's Algorithm MST O(V + E*logV)**

```
int prim(int s)
{
   multiset<pair<int,int>> pq;
   pq.insert(make_pair(0,s));
   int ans=0;
   while(!pq.empty())
   {
    pair <int,int> p = *pq.begin();
    pq.erase(pq.begin());
    if(vis[p.second]==true)
      continue;
    vis[p.second]=true;
      auto itr = P[x].begin();
      while(itr!=P[x].end()) {
      int e = (*itr).first;
      int w = (*itr).second;
      pq.insert(make_pair(w,e));
      itr++;
      }
    ans += p.first;
   }
   return ans;
}
```

*** Trie**

```
const int ALPHABET_SIZE = 26;
struct TrieNode
{
   struct TrieNode
*children[ALPHABET_SIZE];
   // isEndOfWord is true if the node
represents
   // end of a word
   bool isEndOfWord;
};
// Returns new trie node (initialized to
NULLs)
struct TrieNode *getNode(void)
{
   struct TrieNode *pNode =  new TrieNode;
   pNode->isEndOfWord = false;
   for (int i = 0; i < ALPHABET_SIZE; i++)
       pNode->children[i] = NULL;
   return pNode;
}
// If not present, inserts key into trie
// If the key is prefix of trie node, just
// marks leaf node
void insert(struct TrieNode *root, string
key)
{
   struct TrieNode *pCrawl = root;
   for (int i = 0; i < key.length(); i++)
   {
       int index = key[i] - 'a';
       if (!pCrawl->children[index])
           pCrawl->children[index] =
getNode();
       pCrawl = pCrawl->children[index];
   }
   // mark last node as leaf
   pCrawl->isEndOfWord = true;
}
 // Returns true if key presents in trie,
else
// false
bool search(struct TrieNode *root, string
key)
{
   struct TrieNode *pCrawl = root;
   for (int i = 0; i < key.length(); i++)
   {
       int index = key[i] - 'a';
       if (!pCrawl->children[index])
           return false;
       pCrawl = pCrawl->children[index];
   }
   return (pCrawl != NULL &&
pCrawl->isEndOfWord);
}
int main()
{
   // Input keys (use only 'a' through 'z'
   // and lower case)
   string keys[] = {"the", "a", "there",
                   "answer", "any", "by",
                    "bye", "their" };
   int n = sizeof(keys)/sizeof(keys[0]);
    struct TrieNode *root = getNode();
    // Construct trie
   for (int i = 0; i < n; i++)
       insert(root, keys[i]);
    // Search for different keys
   search(root, "the")? cout << "Yes\n" :
                        cout << "No\n";
   search(root, "these")? cout << "Yes\n"
:
                          cout << "No\n";
   return 0;
}
```

*** Bipartite matching algorithms**

```
#define M
#define N
bool bpm(bool bpGraph[M][N], int u,
        bool seen[], int matchR[])
{
   // Try every job one by one
   for (int v = 0; v < N; v++)
   {
       // If applicant u is interested in
       // job v and v is not visited
       if (bpGraph[u][v] && !seen[v])
       {
           // Mark v as visited
           seen[v] = true;
           if (matchR[v] < 0 || bpm(bpGraph, matchR[v],
                                    seen, matchR))
           {
               matchR[v] = u;
               return true;
           }
       }
   }
   return false;
}
int maxBPM(bool bpGraph[M][N])
{
   int matchR[N];
   memset(matchR, -1, sizeof(matchR));
   int result = 0;
   for (int u = 0; u < M; u++)
   {
       // Mark all jobs as not seen
       // for next applicant.
       bool seen[N];
       memset(seen, 0, sizeof(seen));
       // Find if the applicant 'u' can get a job
       if (bpm(bpGraph, u, seen, matchR))
           result++;
```

```
    }
    return result;
}
int main()
{
    bool bpGraph[M][N]’
     cout << maxBPM(bpGraph);
    return 0;
}
```

*** Segment Tree**

```
int tree[2 * N];
void build( int arr[])
{
    for (int i=0; i<n; i++)
        tree[n+i] = arr[i];
    for (int i = n - 1; i > 0; --i)
        tree[i] = tree[i<<1] + tree[i<<1 | 1];
}
void updateTreeNode(int p, int value)
{
    tree[p+n] = value;
    p = p+n;
    for (int i=p; i > 1; i >>= 1)
        tree[i>>1] = tree[i] + tree[i^1];
}
int query(int l, int r)
{
    int res = 0;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1)
    {
        if (l&1)
            res += tree[l++];
        if (r&1)
            res += tree[--r];
    }
    return res;
}
```

*** Longest Common Subsequence**

```
#include<bits/stdc++.h>
using namespace std;
int main()
int lcs( char *X, char *Y, int m, int n )
{
  int L[m+1][n+1];
  int i, j;
     for (i=0; i<=m; i++)
  {
    for (j=0; j<=n; j++)
    {
      if (i == 0 || j == 0)
        L[i][j] = 0;

      else if (X[i-1] == Y[j-1])
        L[i][j] = L[i-1][j-1] + 1;

      else
        L[i][j] = max(L[i-1][j],
L[i][j-1]);
    }
  }
  return L[m][n];
}
```

*** Longest increasing subsequence**

```
#include<bits/stdc++.h>
using namespace std;
void search(int a[],int start,int end,int
key)
{
   if(start+1==end)
   {
    if(a[start]==key || a[end]==key)
      return;
    a[end] = key;
   }
else
   {
    int mid = (start+end)/2;
    if(a[mid]==key)
    {   return;  }
    else
      if(a[mid]<key)
            search(a,mid,end,key);
      else
            search(a,start,mid,key);
   }
}
int main()
{
   ios_base::sync_with_stdio(false);
   cin.tie(NULL);
   int n;
   cin>>n;
   int arr[n];
   for(int i=0;i<n;i++)
    cin>>arr[i];
   int a[n],s=0;
   a[0] = arr[0],s=1;
   for(int i=1;i<n;i++)
   {
    if(arr[i] < a[0])
      a[0] = arr[i];
    else if(arr[i] > a[s-1])
      a[s++] = arr[i];
    else
    {    search(a,0,s-1,arr[i]);  }
   }
   cout<<s;
   return 0;
}
```

```
* Polygon Related Algorithms
const double PI = 4*atan(1);
struct point{
    double x; double y;
};
double dist(point P0, point P1){
    return sqrt((P0.x-P1.x)*(P0.x-P1.x) + (P0.y-P1.y)*(P0.y-P1.y));
}
double angle(point pointA, point pointB, point pointC){
    double a = pointB.x - pointA.x; double b = pointB.y - pointA.y;
    double c = pointB.x - pointC.x; double d = pointB.y - pointC.y;
    double atanA = atan2(a, b); double atanB = atan2(c, d);
    return atanB - atanA;
}
int ccw (point P0, point P1, point P2) {
    double dx1 = P1.x - P0.x; double dx2 = P2.x - P0.x;
    double dy1 = P1.y - P0.y; double dy2 = P1.y - P0.y;
    if (dy1 * dx2 > dy2 * dx1) return -1;
    if (dx1 * dy2 > dy1 * dx2) return 1;
    if ((dx1 * dx2 < 0) || (dy1 * dy2 < 0)) return 1;
    if ((dx1 * dx1 + dy1 * dy1) < (dx2 * dx2 + dy2 * dy2)) return -1;
    return 0;
}
double perimeter(const vector<point> &P){
    double result = 0.0;
    for(int i = 0; i < (int)P.size()-1; i++) // remember that P[0] = P[n-1]
        result += dist(P[i], P[i+1]);
    return result;
}
double area(const vector<point> &P){
    double result = 0.0, x1, y1, x2, y2;
    for(int i = 0; i < (int)P.size()-1; i++){
        x1 = P[i].x; x2 = P[i+1].x; y1 = P[i].y; y2 = P[i+1].y;
        result += (x1 * y2 - x2 * y1);
    }
    return fabs(result) / 2.0;
}
bool isConvex(const vector<point> &P){ // returns true if all three
    int sz = (int)P.size(); // consecutive vertices of P form the same turns
    if (sz <= 3) return false;
```

```
    bool isLeft = ccw(P[0], P[1], P[2]); // remember one result
    for (int i = 1; i < sz-1; i++) // then compare with the others
        if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)
            return false; // different sign -> this polygon is concave
    return true;
}
// returns true if point p is in either convex/concave polygon P
bool inPolygon(point pt, const vector<point> &P){
    if((int)P.size() == 0) return false;
    double sum = 0; // assume the first vertex is equal to the last vertex
    for (int i = 0; i < (int)P.size()-1; i++) {
        if (ccw(pt, P[i], P[i+1])) sum += angle(P[i], pt, P[i+1]); // left turn/ccw
        else sum -= angle(P[i], pt, P[i+1]); } // right turn/cw
    double EPS = 0.1;
    return fabs(fabs(sum) - 2*PI) < EPS;
}
```

*** Fast Fourier Transform (FFT) O(n*logn)**

```
void init_fft(long long n){
    FFT_N  = n; omega.resize(n); double angle = 2 * PI / n;
    for(int i = 0; i < n; i++)
        omega[i] = base( cos(i * angle), sin(i * angle));
}
void fft (vector<base> & a){
    long long n = (long long) a.size();
    if (n == 1)  return;
    long long half = n >> 1;
    vector<base> even (half),  odd (half);
    for (int i=0, j=0; i<n; i+=2, ++j){
        even[j] = a[i]; odd[j] = a[i+1];
    }
   fft (even), fft (odd);
   for (int i=0, fact = FFT_N/n; i < half; ++i){
        base twiddle =  odd[i] * omega[i * fact] ;
        a[i] =  even[i] + twiddle; a[i+half] = even[i] - twiddle;
   }
}
void multiply (const vector<long long> & a, const vector<long long> & b, vector<long long>
& res){
    vector<base> fa (a.begin(), a.end()),  fb (b.begin(), b.end());
    long long n = 1;
```

```
    while (n < 2*max (a.size(), b.size()))  n <<= 1;
    fa.resize (n),  fb.resize (n);
    init_fft(n);
    fft (fa),  fft (fb);
    for (size_t i=0; i<n; ++i) fa[i] = conj( fa[i] * fb[i]);
    fft (fa);
    res.resize (n);
    for(size_t i=0; i<n; ++i){
        res[i] = (long long) (fa[i].real() / n + 0.5); res[i]%=mod;
    }
}
```

*** Lowest Common Ancestor (LCA) O(n)**

```
struct Node
{
    int key;
    struct Node *left, *right;
};
// Utility function creates a new binary tree node with given key
Node * newNode(int k)
{
    Node *temp = new Node;
    temp->key = k;
    temp->left = temp->right = NULL;
    return temp;
}
// Finds the path from root node to given root of the tree, Stores the
// path in a vector path[], returns true if path exists otherwise false
bool findPath(Node *root, vector<int> &path, int k)
{
    // base case
    if (root == NULL) return false;

    // Store this node in path vector. The node will be removed if
    // not in path from root to k
    path.push_back(root->key);

    // See if the k is same as root's key
    if (root->key == k)
        return true;
```

```
    // Check if k is found in left or right sub-tree
    if ( (root->left && findPath(root->left, path, k)) ||
        (root->right && findPath(root->right, path, k)) )
        return true;

    // If not present in subtree rooted with root, remove root from
    // path[] and return false
    path.pop_back();
    return false;
}
// Returns LCA if node n1, n2 are present in the given binary tree,
// otherwise return -1
int findLCA(Node *root, int n1, int n2)
{
    // to store paths to n1 and n2 from the root
    vector<int> path1, path2;

    // Find paths from root to n1 and root to n1. If either n1 or n2
    // is not present, return -1
    if ( !findPath(root, path1, n1) || !findPath(root, path2, n2))
        return -1;

    /* Compare the paths to get the first different value */
    int i;
    for (i = 0; i < path1.size() && i < path2.size() ; i++)
        if (path1[i] != path2[i])
            break;
    return path1[i-1];
}
// Driver program to test above functions
int main()
{
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    cout << "LCA(4, 5) = " << findLCA(root, 4, 5);
    return 0;
}
```

*** Extended Euclidean Algorithm. Calculates X,Y s.t aX+bY = gcd(a,b)**

```
int gcdExtended(int a, int b, int *x, int *y){
   if (a == 0)    // Base Case
   {
       *x = 0;
       *y = 1;
       return b;
   }
   int x1, y1; // To store results of recursive call
   int gcd = gcdExtended(b%a, a, &x1, &y1);
   // Update x and y using results of recursive
   // call
   *x = y1 - (b/a) * x1;
   *y = x1;
   return gcd;
}
```

*** Modular inverse of a, modulo m**

```
void modInverse(int a, int m)
{
   int x, y;
   int g = gcdExtended(a, m, &x, &y);
   if (g != 1)
       cout << "Inverse doesn't exist";
   else
   {
       // m is added to handle negative x
       int res = (x%m + m) % m;
       cout << "Modular multiplicative inverse is " << res;
       return res;
   }
}
```

*** CHINESE REMAINDER THEOREM**

```
// Given a1,a2 ….ak the remainders and n1,n2 ….nk
// Find min sol x which satisfies : X = a1(mod n1) …. X = ak(mod nk)
// N = n1*n2….nk, yi = N/ni , zi = inv(yi)mod(ni)
// X = SUMMATION for(i = 1:k) ai*yi*zi

// k is size of num[] and rem[].  Returns the smallest (a1, a2 .. ak in rem[] and n1,n2 ..
nk in num[])
// number x such that:
```

```
//  x % num[0] = rem[0],
//  x % num[1] = rem[1],
//  ..................
//  x % num[k-2] = rem[k-1]
// Assumption: Numbers in num[] are pairwise coprime
// (gcd for every pair is 1)
int findMinX(int num[], int rem[], int k)
{
   // Compute product of all numbers
   int prod = 1;
   for (int i = 0; i < k; i++)
       prod *= num[i];
   // Initialize result
   int result = 0;
   // Apply above formula
   for (int i = 0; i < k; i++)
   {
       int pp = prod / num[i];
       result += rem[i] * inv(pp, num[i]) * pp;
   }
   return result % prod;
}
```

*** EULER TOTIENT SIEVE**

```
#define SIZE 10000
int phi[SIZE];
for(int i = 0;i<SIZE;i++)
    phi[i] = i;
int EulerTotientFiller(){
   for(int i = 2;i<SIZE;i++){
    if(phi[i] == i){    // its a prime
      for(j = 2*i;j<SIZE;j+= i){
            phi[j] = phi[j] - (phi[j]/i);    // i is prime factor of j
      }
    }
    else{
      // its not prime skip it.
    }
   }
}
```

*** KMP Pattern Matching O(n+k)**

```
#include <bits/stdc++.h>
void computeLPSArray(char* pat, int M,
int* lps);
void KMPSearch(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int lps[M];
    computeLPSArray(pat, M, lps);
    int i = 0; // index for txt[]
    int j = 0; // index for pat[]
    while (i < N) {
       if (pat[j] == txt[i]) {
           j++;
           i++;
       }
       if (j == M) {
            printf("Found pattern at index
%d ", i - j);
            j = lps[j - 1];
       }
       else if (i < N && pat[j] != txt[i])
{
           if (j != 0)
               j = lps[j - 1];
           else
               i = i + 1;
       }
    }
}
void computeLPSArray(char* pat, int M,
int* lps)
{   int len = 0;
   lps[0] = 0; // lps[0] is always 0
   int i = 1;
    while (i < M) {
       if (pat[i] == pat[len]) {
           len++;
           lps[i] = len;
           i++;
       }
       else // (pat[i] != pat[len])
       {
           if (len != 0) {
               len = lps[len - 1];
           }
           else // if (len == 0)
           {  lps[i] = 0;  i++;   }
}
}
}
int main()
{
    char txt[] = "ABABDABACDABABCABAB";
    char pat[] = "ABABCABAB";
    KMPSearch(pat, txt);
    return 0;
}
```

*** 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
    return max_flow;
}
// Driver program to test above functions
int main()
{
    int graph[V][V]

    cout << "The maximum possible flow is " << fordFulkerson(graph, 0, 5);
    return 0;
}
```

*** Coordinate Compression 1D**

```
int main() {
   int t, n, q, x1, x2, a, b, c, m;
   cin >> t;
   for (int c = 1; c <= t; c++) {
       cin >> n >> q;
       vector<int> x(n);
       scan(x, n);
       vector<int> y(n);
       scan(y, n);
       vector<int> z(q);
       scan(z, q);
       vector<int> l(n), r(n), k(q);
       for (int i = 0; i < n; i++) {
           l[i] = min(x[i], y[i]) + 1;
           r[i] = max(x[i], y[i]) + 1;
           r[i]++;
       }
       for (int i = 0; i < q; i++) {
           k[i] = z[i] + 1;
       }
       set <int> s;
       for (int i = 0; i < n; i++) {
           s.insert(l[i]);
           s.insert(r[i]);
```

```
        }
        vector<int> pos;
        set <int> :: iterator itr;
        for (itr = s.begin(); itr != s.end(); ++itr) {
            pos.push_back(*itr);
        }
        int p = pos.size();
        vector<int> m(p+1, 0);
        for (int i = 0; i < n; i++) {
            m[lower_bound(pos.begin(), pos.end(), l[i]) - pos.begin()]++;
            m[lower_bound(pos.begin(), pos.end(), r[i]) - pos.begin()]--;
        }
        for (int i = 0; i < p-1; i++) {
            m[i+1] = m[i+1] + m[i];
        }
        vector<ll> sf(p, 0);
        sf[p-1] = m[p-1];
        for (int i = p-1; i > 0; i--) {
            sf[i-1] = sf[i] + m[i-1] * 1LL * (pos[i]-pos[i-1]);
        }
        ll ans = 0;
        for (int i = 0; i < q; i++) {
            if (k[i] <= sf[0]) {
                int x1 = upper_bound(sf.begin(), sf.end(), k[i], greater<long long>()) -
sf.begin();
                if (sf[x1] == k[i]) {
                    ans += (1LL * pos[x1] * (i+1));
                }
                else {
                    X1--;
                    long long d = sf[x1] - k[i];
                    d = d / m[x1];
                    ll term = pos[x1] + d;
                    ans += term * (i+1);
                }
            }
        }
        cout << "Case #" << c << ": " << ans << endl;
    }
}
```

*** Bridge in Graph**

```
void Graph::bridgeUtil(int u, bool visited[], int disc[],
                               int low[], int parent[])
{
   // A static variable is used for simplicity, we can
   // avoid use of static variable by passing a pointer.
   static int time = 0;
   // Mark the current node as visited
   visited[u] = true;
   // Initialize discovery time and low value
   disc[u] = low[u] = ++time;
   // Go through all vertices aadjacent to this
   list<int>::iterator i;
   for (i = adj[u].begin(); i != adj[u].end(); ++i)
   {
       int v = *i; // v is current adjacent of u

       // If v is not visited yet, then recur for it
       if (!visited[v])
       {
           parent[v] = u;
           bridgeUtil(v, visited, disc, low, parent);

           // Check if the subtree rooted with v has a
           // connection to one of the ancestors of u
           low[u] = min(low[u], low[v]);

           // If the lowest vertex reachable from subtree
           // under v is below u in DFS tree, then u-v
           // is a bridge
           if (low[v] > disc[u])
           cout << u <<" " << v << endl;
       }

       // Update low value of u for parent function calls.
       else if (v != parent[u])
           low[u] = min(low[u], disc[v]);
   }
}
```

*** Articulation Point**

```
void Graph::APUtil(int u, bool visited[], int disc[],
                                    int low[], int parent[], bool ap[])
{
   // A static variable is used for simplicity, we can avoid use of static
   // variable by passing a pointer.
   static int time = 0;
   // Count of children in DFS Tree
   int children = 0;
   // Mark the current node as visited
   visited[u] = true;
   // Initialize discovery time and low value
   disc[u] = low[u] = ++time;
   // Go through all vertices aadjacent to this
   list<int>::iterator i;
   for (i = adj[u].begin(); i != adj[u].end(); ++i) {
       int v = *i; // v is current adjacent of u
       // If v is not visited yet, then make it a child of u
       // in DFS tree and recur for it
       if (!visited[v])  {
           children++;
           parent[v] = u;
           APUtil(v, visited, disc, low, parent, ap);
           // Check if the subtree rooted with v has a connection to
           // one of the ancestors of u
           low[u] = min(low[u], low[v]);
           // u is an articulation point in following cases
           // (1) u is root of DFS tree and has two or more chilren.
           if (parent[u] == NIL && children > 1)
           ap[u] = true;
           // (2) If u is not root and low value of one of its child is more
           // than discovery value of u.
           if (parent[u] != NIL && low[v] >= disc[u])
           ap[u] = true;
       }
       // Update low value of u for parent function calls.
       else if (v != parent[u])
           low[u] = min(low[u], disc[v]);
   }
}
```

*** Segment Tree with Lazy Propogation build: O(n*logn), query: O(logn + k)**

```
int tree[MAX] = {0};int lazy[MAX] = {0};
void updateRangeUtil(int si,int ss,int se,int us,int ue,int diff){
   if (lazy[si] != 0){
      tree[si] += (se-ss+1)*lazy[si];
      if (ss != se){
          lazy[si*2 + 1] += lazy[si]; lazy[si*2 + 2] += lazy[si];
      }
      lazy[si] = 0;
  }
  if (ss>se || ss>ue || se<us) return ; // out of range
  if (ss>=us && se<=ue){
      tree[si] += (se-ss+1)*diff;
      if (ss != se){
          lazy[si*2 + 1] += diff; lazy[si*2 + 2] += diff;
      }
      return;
  }
  int mid = (ss+se)/2;
  updateRangeUtil(si*2+1, ss, mid, us, ue, diff);
  updateRangeUtil(si*2+2, mid+1, se, us, ue, diff);
  tree[si] = tree[si*2+1] + tree[si*2+2];
}
void updateRange(int n, int us, int ue, int diff){updateRangeUtil(0, 0, n-1, us, ue,
diff);}
int getSumUtil(int ss, int se, int qs, int qe, int si){
  if (lazy[si] != 0){
      tree[si] += (se-ss+1)*lazy[si];
       if (ss != se){lazy[si*2+1] += lazy[si]; lazy[si*2+2] += lazy[si];}
      lazy[si] = 0;
  }
  if (ss>se || ss>qe || se<qs) return 0;
  if (ss>=qs && se<=qe) return tree[si];
  int mid = (ss + se)/2;
  return getSumUtil(ss,mid,qs,qe,2*si+1)+getSumUtil(mid+1,se,qs,qe,2*si+2);
}
int getSum(int n, int qs, int qe){
  if (qs < 0 || qe > n-1 || qs > qe) return -1;
  return getSumUtil(0, n-1, qs, qe, 0);
}
```

```
void constructSTUtil(int arr[], int ss, int se, int si){
  if (ss > se) return ;
  if (ss == se){ tree[si] = arr[ss]; return;}
  int mid = (ss + se)/2;constructSTUtil(arr, ss, mid, si*2+1);
  constructSTUtil(arr, mid+1, se, si*2+2);
  tree[si] = tree[si*2 + 1] + tree[si*2 + 2];
}
void constructST(int arr[], int n){constructSTUtil(arr, 0, n-1, 0);}
int main(){
  int arr[] = {1, 3, 5, 7, 9, 11}; int n = sizeof(arr)/sizeof(arr[0]);
  constructST(arr, n);
  printf("Sum of values in given range = %d\n",getSum(n, 1, 3));
  updateRange(n, 1, 5, 10); // Add 10 to all nodes at indexes from 1 to 5.
  printf("Updated sum of values in given range = %d\n",getSum( n, 1, 3));
  return 0;
}
```