The Azinaa Code Notebook

December 11, 2017

Contents

0.1 Graph

0.1.1 BFS

```cpp
#include <bits/stdc++.h>
using namespace std;
const int SIZE = 100; //Vertex size
int main(){
  int source = 0;
  vector<int> Graph[SIZE];
  bool visited[SIZE];
  int distance[SIZE];
  fill(visited, visited+SIZE, 0);
  fill(distance, distance+SIZE, INT_MAX);
  queue<int> q;
  q.push(source);
  visited[source] = true;
  distance[source] = 0;
  while(!q.empty()){
    int u = q.front();
    q.pop();
    for(int i = 0 ; i < Graph[u].size(); i++){
      if(visited[Graph[u][i]]){
        continue;
      }
      visited[Graph[u][i]] = true;
      distance[Graph[u][i]] = distance[u] + 1;
      q.push(Graph[u][i]);
    }
  }
  return 0;
}
```

0.1.2 Topological-Sorting

```cpp
#include <bits/stdc++.h>
using namespace std;
const int SIZE = 100;//Graph Size Vertex
int visited[SIZE];
vector <int> Graph[SIZE];
stack<int> ans;
int topolSort(int u)
{
  if(visited[u] == 1)
  {
    return -1;//cycle
  }
  if(visited[u] == 2)
  {
    return 0;
  }
  visited[u] = 1;
  for(int i = 0 ; i < Graph[u].size() ; i++)
  {
    topolSort(Graph[u][i]);
  }
    visited[u] = 2;
  ans.push(u);
  return 0;
}

int main(){
  return 0;
}
```

0.1.3 DAG Shortest Path

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

const int Gsize=100000;

int _color[Gsize], _p[Gsize] ,_d[Gsize] ,_f[Gsize] , _time;
vector<int> _V;
void DFS_visit(vector<int> G[] ,int u)
{
  _time++;
  _d[u]=_time;
  _color[u]=1;
  int v;
  for(int i=0;i<G[u].size();i++)
  {
    v=G[u][i];
    if(!_color[v])
    {
      _p[v]=u;
      DFS_visit(G ,v);
    }
  }
```

```
  _color[u]=2;
  _time++;
  _f[u]=_time;
  _V.push_back(u);
}
void DFS(vector<int>G[])
{
  for(int i=0;i<Gsize;i++)
  {
    _color[i]=0; _p[i]=0;
  }
  _time=0;
  for(int i=0;i<Gsize;i++)
    if(!_color[i])
      DFS_visit(G ,i);
}

void Topologival_sort(vector<int> G[], int arr[])
{
  _V.clear();
  DFS(G);
  for(int i=0;i<_V.size();i++)
    arr[_V.size()-1-i] = _V[i];
}

void _Relax(int u, int v, int w)
{
  if(_d[v]>(_d[u]+w))
  {
    _d[v] = _d[u]+w;
    _p[v] = u;
  }
}

void DAG_shortest_paths(vector<int> G[], vector<int> W[], int
    s)
{
  int arr[Gsize];
  Topologival_sort(G, arr);
  for(int i=0;i<Gsize;i++)
    _d[i]=500000000;
  _d[s] = 0;
  for(int i=0;i<Gsize;i++)
  {
    int u=arr[i];
    for(int j=0;j<G[u].size();j++)
      _Relax(u, G[u][j], W[u][j]);
  }
}

int main()
{

  return 0;
}
```

0.1.4 Bellman-Ford

```
// A C++ program for Bellman-Ford's single source
#include <bits/stdc++.h>

struct Edge
{
    int src, dest, weight;
};

struct Graph
{
    int V, E; //Size Of Graph
    struct Edge* edge;
};

struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

void BellmanFord(struct Graph* graph, int src, int* dist)
{
    int V = graph->V;
    int E = graph->E;

    for (int i = 0; i < V; i++)
        dist[i]   = INT_MAX;
    dist[src] = 0;
```

```
    for (int i = 1; i <= V-1; i++)
    {
        for (int j = 0; j < E; j++)
        {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    for (int i = 0; i < E; i++)
    {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            printf("Graph contains negative weight cycle"); // Hint!
    }

    //printArr(dist, V);//Finishing Algo

    return;
}

int main()
{
    int V = 5;  // Number of vertices in graph
    int E = 8;  // Number of edges in graph
    struct Graph* graph = createGraph(V, E);
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = -1;
    int dist[5];
    BellmanFord(graph, 0, dist);
    return 0;
}
```

0.1.5 Djkstra

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define pii pair<int,int>
#define F first
#define S second
#define mp make_pair
#define pb emplace_back

bool vis[100001];
int dis[100001];
vector<pii> a[100001];

class prioritize {
public: bool operator ()(pii &p1 , pii &p2) {
    return p1.S > p2.S;
  }
};

int Dijkstra(int s, int n) {
  for (int i = 0; i <= n; i++) {
    vis[i] = false;
    dis[i] = INT_MAX;
  }
  priority_queue<pii, vector<pii>, prioritize> pq;
  pq.push(mp(s, dis[s] = 0));
  while (!pq.empty()) {
    pii cur = pq.top(); pq.pop();
    int cv = cur.F, cw = cur.S;
    if (vis[cv]) continue;
    vis[cv] = true;
    for (pii x : a[cv]) {
      if (!vis[x.F] && (cw + x.S) < dis[x.F]) {
        pq.push(mp(x.F, dis[x.F] = cw + x.S));
      }
    }
  }
}

int main() {
  int tc;
  cin >> tc;
  while (tc--) {
    int v1, v2, w, n, m;
    cin >> n >> m;
```

```
    for (int i = 0; i <= n; i++) {
      a[i].clear();
    }
    for (int i = 0; i < m; i++) {
      cin >> v1 >> v2 >> w;
      a[v1].pb(mp(v2, w));
    }
    int s;
    cin >> s;
    Dijkstra(s, n);
    for (int i = 1; i <= n; i++) {
      if (dis[i] != INT_MAX) {
        cout << dis[i] << "␣";
      } else {
        cout << "-1␣";
      }
    }
  }
  return 0;
}
```

0.1.6 Floyd Warshall

```
// C Program for Floyd Warshall Algorithm
#include<stdio.h>

#define V 4
#define INF 9999999

void floydWarshall (int graph[][V])
{
  int dist[V][V], i, j, k;
  for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)
      dist[i][j] = graph[i][j];
    for (k = 0; k < V; k++)
    {
      for (i = 0; i < V; i++)
      {
        for (j = 0; j < V; j++)
        {
          if (dist[i][k] + dist[k][j] < dist[i][j])
              dist[i][j] = dist[i][k] + dist[k][j];
        }
      }
    }
```

```
    }
    // All distances -> dist
}

int main()
{
    /* example Graph:
            10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
        \|/        |
       (1)------->(2)
            3           */
    int graph[V][V] =
    {
      {0,   5,  INF, 10},
      {INF, 0,   3, INF},
      {INF, INF, 0,   1},
      {INF, INF, INF, 0}
    };

    // Print the solution
    floydWarshall(graph);
    return 0;
}
```

0.1.7 Strongly Connected Component

```
#include <bits/stdc++.h>
using namespace std;
const int SIZE = 50001;
bool visited[SIZE];
//Input: vector<int> Graph, vector<int> Reverce Graph,
// int Nomber Of vertexces
//Output:
//Strongly connected componnet Graph->
//--->SCC_Graph
//List of Each Componnent -->
//----->ListOfEachSCC
//to see each node blongs to where
//------SCC_list
//SCC returns the size of SCC graph
vector<int> SCC_Graph[SIZE];
```

```
vector<int> ListOfEachSCC[SIZE];
int SCC_List[SIZE];

void TopolSort(vector<int> g[],int u, stack<int>& ans){
    if(visited[u]){
        return;
    }
    visited[u] = true;
    for(int i = 0 ; i < g[u].size() ; i++){
        TopolSort(g,g[u][i],ans);
    }
    ans.push(u);
}

void DFSR(vector<int> g[],int u, int counter){
    if(visited[u]){
        return;
    }
    visited[u] = true;
    for(int i = 0 ; i < g[u].size() ; i++){
        DFSR(g,g[u][i], counter);
    }
    ListOfEachSCC[counter].push_back(u);
    SCC_List[u] = counter;
}

int SCC(vector<int> graph[], vector<int> graph_reverce[],int v){
    fill(visited, visited+v, false);
    for(int i = 0 ; i < v ; i++){
        SCC_Graph[i].clear();
        ListOfEachSCC[i].clear();
        SCC_List[i] = 0;
    }
    stack<int> TopolSorted;
    for(int i = 0 ; i < v ; i++){
        if(visited[i]){
            continue;
        }
        TopolSort(graph,i, TopolSorted);
    }
    fill(visited, visited+v, false);
    fill(SCC_List, SCC_List+v, 0);
    int counter = 0;
    while(!TopolSorted.empty()){
        int u = TopolSorted.top();
        TopolSorted.pop();
        if(visited[u]){
            continue;
        }
        DFSR(graph_reverce, u, counter);
        counter++;
    }
    for(int i = 0 ; i < counter ; i++){
        for(int j = 0 ; j < ListOfEachSCC[i].size() ; j++){
            int u = ListOfEachSCC[i][j];
            for(int k = 0 ; k < graph[u].size() ; k++){
                int w = graph[u][k];
                if(SCC_List[u] != SCC_List[w]){
                    SCC_Graph[SCC_List[u]].push_back(SCC_List[w]);
                }
            }
        }
    }
    return counter;
}
```

0.1.8 Minimum Spanning Tree

```
//C++ program for Prim's Minimum Spanning Tree (MST) algorithm.
#include <stdio.h>
#include <limits.h>

#define V 5 //Graph Size

int minKey(int key[], bool mstSet[])
{
   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++)
     if (mstSet[v] == false && key[v] < min)
         min = key[v], min_index = v;
   return min_index;
}

int printMST(int parent[], int n, int graph[V][V])
{
   printf("Edge   Weight\n");
```

```
    for (int i = 1; i < V; i++)
       printf("%d - %d     %d \n", parent[i], i, graph[i][
          parent[i]]);
}

void primMST(int graph[V][V])
{
     int parent[V];
     int key[V];
     bool mstSet[V];
     for (int i = 0; i < V; i++)
         key[i] = INT_MAX, mstSet[i] = false;
     key[0] = 0;
     parent[0] = -1;
     for (int count = 0; count < V-1; count++)
     {
         int u = minKey(key, mstSet);
         mstSet[u] = true;
         for (int v = 0; v < V; v++)
           if (graph[u][v] && mstSet[v] == false && graph[u][v
              ] <  key[v])
              parent[v]  = u, key[v] = graph[u][v];
     }
     printMST(parent, V, graph);//print Solution
}

int main()
{
    /* Let us create the following graph
            2    3
       (0)--(1)--(2)
        |   / \   |
       6| 8/   \5 |7
        | /     \ |
       (3)-------(4)
             9          */
    int graph[V][V] =
     {{0, 2, 0, 6, 0},
     {2, 0, 3, 8, 5},
     {0, 3, 0, 0, 7},
     {6, 8, 0, 0, 9},
     {0, 5, 7, 9, 0},
     };
     primMST(graph);
     return 0;
}
```

0.1.9 MST Kruskal

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class Disjoint_set
{
  int *id, cnt, *sz;
public:
  Disjoint_set(int N)
  {
    cnt = N;
    id = new int[N];
    sz = new int[N];
    for(int i=0; i<N; i++)
    {
      id[i] = i;
      sz[i] = 1;
    }
  }
  ~Disjoint_set()
  {
    delete [] id;
    delete [] sz;
  }
  int find(int p)
  {
    int root = p;
    while (root != id[root])
      root = id[root];
    while (p != root)
    {
      int newp = id[p];
      id[p] = root;
      p = newp;
    }
    return root;
  }
  void merge(int x, int y)
  {
```

```
    int i = find(x);
    int j = find(y);
    if (i == j) return;
    if   (sz[i] < sz[j])
    {
      id[i] = j;
      sz[j] += sz[i];
    }
    else
    {
      id[j] = i;
      sz[i] += sz[j];
    }
    cnt--;
  }
  bool connected(int x, int y) {return find(x) == find(y);}
  int count() {return cnt;}
};

const int Gsize = 9;

bool cmp(pair<pair<int, int>, int> p1, pair<pair<int, int>,
   int> p2)
{
  return p1.second<p2.second;
}

vector<pair<int, int> > _A;

void MST_kruskal(vector<int> G[], vector<int> w[])
{
  Disjoint_set set(Gsize);
  _A.clear();
  vector<pair<pair<int, int>, int> > E;
  for(int i=0;i<Gsize;i++)
    for(int j=0;j<G[i].size();j++)
      if(i<G[i][j])
        E.push_back(make_pair(make_pair(i, G[i][j]), w[i][j])
            );
  sort(E.begin(), E.end(), cmp);
  for(int i=0;i<E.size();i++)
    if(!set.connected(E[i].first.first, E[i].first.second))
    {
      _A.push_back(make_pair(E[i].first.first, E[i].first.
          second));
      set.merge(E[i].first.first, E[i].first.second);
    }
}

void MST_kruskal(vector<pair<pair<int, int>, int> > E)
{
  Disjoint_set set(Gsize);
  _A.clear();
  sort(E.begin(), E.end(), cmp);
  for(int i=0;i<E.size();i++)
    if(!set.connected(E[i].first.first, E[i].first.second))
    {
      _A.push_back(make_pair(E[i].first.first, E[i].first.
          second));
      set.merge(E[i].first.first, E[i].first.second);
    }
}

int main()
{

  return 0;
}
```

0.1.10 Maximum-BPM

```
// A C++ program to find maximal Bipartite matching.
#include <iostream>
#include <string.h>
using namespace std;

#define M 6 // Size Of Graph M*N
#define N 6

bool bpm(bool bpGraph[M][N], int u, bool seen[], int matchR
   [])
{
    for (int v = 0; v < N; v++)
    {
        if (bpGraph[u][v] && !seen[v])
        {
            seen[v] = true; // Mark v as visited
            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];//List Of Matches

    memset(matchR, -1, sizeof(matchR));

    int result = 0;
    for (int u = 0; u < M; u++)
    {
        bool seen[N];
        memset(seen, 0, sizeof(seen));
        if (bpm(bpGraph, u, seen, matchR))
            result++;
    }
    return result;
}

int main()
{
    bool bpGraph[M][N] = {  {0, 1, 1, 0, 0, 0},
                            {1, 0, 0, 1, 0, 0},
                            {0, 0, 1, 0, 0, 0},
                            {0, 0, 1, 1, 0, 0},
                            {0, 0, 0, 0, 0, 0},
                            {0, 0, 0, 0, 0, 1}
                         };

cout << "Maximum␣Maching␣"<< maxBPM(bpGraph);

return 0;
}
```

0.2 Flow

0.2.1 Max-Flow

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

#define V 6

bool bfs(int rGraph[V][V], int s, int t, int parent[])
{
    bool visited[V];
    memset(visited, 0, sizeof(visited));
    queue <int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;
    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;
            }
        }
    }
    return (visited[t] == true);
}

int fordFulkerson(int graph[V][V], int s, int t)
{
    int u, v;
    int rGraph[V][V];
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
             rGraph[u][v] = graph[u][v];
    int parent[V];
    int max_flow = 0;
```

```
    while (bfs(rGraph, s, t, parent))
    {
        int path_flow = INT_MAX;
        for (v=t; v!=s; v=parent[v])
        {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }

        for (v=t; v != s; v=parent[v])
        {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }
        max_flow += path_flow;
    }
    return max_flow;
}

int main()
{
    int graph[V][V] = { {0, 16, 13, 0, 0, 0},
                        {0, 0, 10, 12, 0, 0},
                        {0, 4, 0, 0, 14, 0},
                        {0, 0, 9, 0, 0, 20},
                        {0, 0, 0, 7, 0, 4},
                        {0, 0, 0, 0, 0, 0}
                      };
    cout << "The maximum possible flow is "
     << fordFulkerson(graph, 0, 5);
    return 0;
}
```

0.2.2 Min-Cut

```
// C++ program for finding minimum cut using Ford-Fulkerson
#include <iostream>
#include <limits.h>
#include <string.h>
#include <queue>
using namespace std;

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


int bfs(int rGraph[V][V], int s, int t, int parent[])
{
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    queue <int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    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;
            }
        }
    }
    return (visited[t] == true);
}

void dfs(int rGraph[V][V], int s, bool visited[])
{
    visited[s] = true;
    for (int i = 0; i < V; i++)
       if (rGraph[s][i] && !visited[i])
           dfs(rGraph, i, visited);
}

void minCut(int graph[V][V], int s, int t)
{
    int u, v;
    int rGraph[V][V];
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
             rGraph[u][v] = graph[u][v];
```

```
    int parent[V];
    while (bfs(rGraph, s, t, parent))
    {
        int path_flow = INT_MAX;
        for (v=t; v!=s; v=parent[v])
        {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }
        for (v=t; v != s; v=parent[v])
        {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }
    }
    //Finishing...
    bool visited[V];
    memset(visited, false, sizeof(visited));
    dfs(rGraph, s, visited);
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
         if (visited[i] && !visited[j] && graph[i][j])
              cout << i << " - " << j << endl;

    return;
}

int main()
{
    int graph[V][V] = { {0, 16, 13, 0, 0, 0},
                        {0, 0, 10, 12, 0, 0},
                        {0, 4, 0, 0, 14, 0},
                        {0, 0, 9, 0, 0, 20},
                        {0, 0, 0, 7, 0, 4},
                        {0, 0, 0, 0, 0, 0}
                      };

    minCut(graph, 0, 5);

    return 0;
}
```

0.3 Geometry

0.3.1 Convex-Hull

```
// A C++ program to find convex hull of a set of points.
   Refer
#include <iostream>
#include <stack>
#include <stdlib.h>
using namespace std;

struct Point
{
    int x, y;
};

Point p0;

Point nextToTop(stack<Point> &S)
{
    Point p = S.top();
    S.pop();
    Point res = S.top();
    S.push(p);
    return res;
}

int swap(Point &p1, Point &p2)
{
    Point temp = p1;
    p1 = p2;
    p2 = temp;
}

int distSq(Point p1, Point p2)
{
    return (p1.x - p2.x)*(p1.x - p2.x) +
          (p1.y - p2.y)*(p1.y - p2.y);
}

int orientation(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - q.x)
        - (q.x - p.x) * (r.y - q.y);
```

```
    if (val == 0) return 0;
    return (val > 0)? 1: 2;
}

int compare(const void *vp1, const void *vp2)
{
   Point *p1 = (Point *)vp1;
   Point *p2 = (Point *)vp2;

   int o = orientation(p0, *p1, *p2);
   if (o == 0)
     return (distSq(p0, *p2) >= distSq(p0, *p1))?
                  -1 : 1;

   return (o == 2)? -1: 1;
}

void convexHull(Point points[], int n)
{
   int ymin = points[0].y, min = 0;
   for (int i = 1; i < n; i++)
   {
     int y = points[i].y;
     if ((y < ymin) || (ymin == y &&
         points[i].x < points[min].x))
        ymin = points[i].y, min = i;
   }

   swap(points[0], points[min]);

   p0 = points[0];
   qsort(&points[1], n-1, sizeof(Point), compare);

   int m = 1; // Initialize size of modified array
   for (int i=1; i<n; i++)
   {
       while (i < n-1 && orientation(p0, points[i], points[i
          +1]) == 0)
          i++;
       points[m] = points[i];
       m++;
   }

   if (m < 3) return;

   stack<Point> S;
   S.push(points[0]);
   S.push(points[1]);
   S.push(points[2]);

   for (int i = 3; i < m; i++)
   {
      while (orientation(nextToTop(S), S.top(), points[i]) !=
          2)
         S.pop();
      S.push(points[i]);
   }

   while (!S.empty()) // List Of The Points In The Convex
      Hull
   {
       Point p = S.top();
       cout << "(" << p.x << ", " << p.y <<")" << endl;
       S.pop();
   }
}

int main()
{
    Point points[] = {{0, 3}, {1, 1}, {2, 2}, {4, 4},
                      {0, 0}, {1, 2}, {3, 1}, {3,3}};
    int n = sizeof(points)/sizeof(points[0]);
    /* N is The Number of Points And points is
    the list Of points */
    convexHull(points, n);
    return 0;
}
```

0.3.2 Shoelace Formula (python)

```
def PolygonArea(corners):
    n = len(corners) # of corners
    area = 0.0
    for i in range(n):
        j = (i + 1) % n
        area += corners[i][0] * corners[j][1]
        area -= corners[j][0] * corners[i][1]
    area = abs(area) / 2.0
    return area
```

```
10
11 # examples
12 corners = [(2.0, 1.0), (4.0, 5.0), (7.0, 8.0)]
13 print (PolygonArea(corners))
```

0.3.3 Swap Line

```
1  #include <bits/stdc++.h>
2  #define px second
3  #define py first
4  typedef pair<long long, long long> pairll;
5  pairll pnts [MAX];
6  int compare(pairll a, pairll b)
7  {
8          return a.px<b.px;
9  }
10 double closest_pair(pairll pnts[],int n)
11 {
12         sort(pnts,pnts+n,compare);
13         double best=INF;
14         set<pairll> box;
15         box.insert(pnts[0]);
16         int left = 0;
17         for (int i=1;i<n;++i)
18         {
19             while (left<i && pnts[i].px-pnts[left].px > best)
20                 box.erase(pnts[left++]);
21             for(typeof(box.begin()) it=box.lower_bound(
                   make_pair(pnts[i].py-best, pnts[i].px-best));
                   it!=box.end() && pnts[i].py+best>=it->py;it
                   ++)
22                  best = min(best, sqrt(pow(pnts[i].py - it->py
                        , 2.0)+pow(pnts[i].px - it->px, 2.0)));
23             box.insert(pnts[i]);
24         }
25         return best;
26 }
```

0.3.4 Union Of Rectangles

```
1  #include <bits/stdc++.h>
2  #define MAX 1000
3  struct event
4  {
5      int ind;    // Index of rectangle in rects
6      bool type; // Type of event: 0 = Lower-left ; 1 = Upper-
           right
7      event(){};
8      event(int ind, int type) : ind(ind), type(type){};
9  };
10 struct point
11 {
12     int x, y;
13 };
14 point rects[MAX][12];
15 // Each rectangle consists of 2 points: [0] = lower-left ;
       [1] = upper-right
16 bool compare_x(event a, event b)
17 {
18     return rects[a.ind][a.type].x < rects[b.ind][b.type].x;
19 }
20 bool compare_y(event a, event b)
21 {
22     return rects[a.ind][a.type].y < rects[b.ind][b.type].y;
23     }
24 int union_area(event events_v[], event events_h[], int n, int
       e)
25 {
26     /*n is the number of rectangles, e=2*n , e is the number
           of
27     points (each rectangle has two points as described in
28     declaration of rects)
29     */
30     bool in_set[MAX] = {0};
31     int area = 0;
32     sort(events_v, events_v + e, compare_x);
33     //Pre-sort of vertical edges
34     sort(events_h, events_h + e, compare_y);
35     // Pre-sort set of horizontal edges
36     in_set[events_v[0].ind] = 1;
37     for (int i = 1; i < e; ++i)
38     {
39         event c = events_v[i];
40         int cnt = 0; // Counter to indicate how many
41         //rectangles are currently overlapping
42         // Delta_x: Distance between current
43         // sweep line and previous sweep line
44         int delta_x = rects[c.ind][c.type].x - rects[events_v
               [i - 1].ind]\
```

```
        [events_v[i - 1].type].x;
        int begin_y;
        if (delta_x == 0)
        {
            in_set[c.ind] = (c.type == 0);
            continue;
        }
        for (int j = 0; j < e; ++j)
            if (in_set[events_h[j].ind] == 1)
            {
                if (events_h[j].type == 0)
                {
                    if (cnt == 0)
                        begin_y = rects[events_h[j].ind][0].y
                            ;
                    ++cnt;
                }
                else
                {
                    --cnt;
                    if (cnt == 0)
                    {
                        int delta_y = (rects[events_h[j].ind
                            ][13].y - begin_y);
                        area += delta_x * delta_y;
                    }
                }
            }
        in_set[c.ind] = (c.type == 0);
    }
    return area;
}
```

0.4 String

0.4.1 LCS

```
/* Dynamic Programming C/C++ implementation of LCS problem */
#include<bits/stdc++.h>

int max(int a, int b)
{
    return (a > b)? a : b;
}

 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];
}

int main()
{
  char X[] = "AGGTAB";
  char Y[] = "GXTXAYB";

  int m = strlen(X);
  int n = strlen(Y);

  printf("Length␣of␣LCS␣is␣%dn", lcs( X, Y, m, n ) );

  return 0;
}
```

0.4.2 LIS

```
typedef vector<int> VI;
typedef pair<int,int> PII;
typedef vector<PII> VPII;

#define STRICTLY_INCREASNG

VI LongestIncreasingSubsequence(VI v) {
```

```
  VPII best;
  VI dad(v.size(), -1);

  for (int i = 0; i < v.size(); i++) {
#ifdef STRICTLY_INCREASNG
    PII item = make_pair(v[i], 0);
    VPII::iterator iter = lower_bound(best.begin(), best.end(), item);
    item.second = i;
#else
    PII item = make_pair(v[i], i);
    VPII::iterator iter = upper_bound(best.begin(), best.end(), item);
#endif
    if (iter == best.end()) {
      dad[i] = (best.size() == 0 ? -1 : best.back().second);
      best.push_back(item);
    } else {
      dad[i] = dad[iter->second];
      *iter = item;
    }
  }

  VI ret;
  for (int i = best.back().second; i >= 0; i = dad[i])
    ret.push_back(v[i]);
  reverse(ret.begin(), ret.end());
  return ret;
}
```

0.4.3 KMP

```
/*
Finds all occurrences of the pattern string p within the
text string t. Running time is O(n + m), where n and m
are the lengths of p and t, respecitvely.
*/

#include <iostream>
#include <string>
#include <vector>

using namespace std;

typedef vector<int> VI;

void buildPi(string& p, VI& pi)
{
  pi = VI(p.length());
  int k = -2;
  for(int i = 0; i < p.length(); i++) {
    while(k >= -1 && p[k+1] != p[i])
      k = (k == -1) ? -2 : pi[k];
    pi[i] = ++k;
  }
}

int KMP(string& t, string& p)
{
  VI pi;
  buildPi(p, pi);
  int k = -1;
  for(int i = 0; i < t.length(); i++) {
    while(k >= -1 && p[k+1] != t[i])
      k = (k == -1) ? -2 : pi[k];
    k++;
    if(k == p.length() - 1) {
      // p matches t[i-m+1, ..., i]
      cout << "matched␣at␣index␣" << i-k << ":␣";
      cout << t.substr(i-k, p.length()) << endl;
      k = (k == -1) ? -2 : pi[k];
    }
  }
  return 0;
}

int main()
{
  string a = "AABAACAADAABAABA", b = "AABA";
  KMP(a, b); // expected matches at: 0, 9, 12
  return 0;
}
```

0.5 Other

0.5.1 Disjoint Set

```
// To represent Disjoint Sets
struct DisjointSets
```

```
{
    int *parent, *rnk;
    int n;

    DisjointSets(int n)
    {
        this->n = n;
        parent = new int[n+1];
        rnk = new int[n+1];
        for (int i = 0; i <= n; i++)
        {
            rnk[i] = 0;

            parent[i] = i;
        }
    }

    int find(int u)
    {
        if (u != parent[u])
            parent[u] = find(parent[u]);
        return parent[u];
    }
    void merge(int x, int y)
    {
        x = find(x), y = find(y);
        if (rnk[x] > rnk[y])
            parent[y] = x;
        else
            parent[x] = y;

        if (rnk[x] == rnk[y])
            rnk[y]++;
    }
};
```

0.5.2 Date Transformation

```
//Dates (C++)

// Routines for performing computations on dates.  In these routines,
// months are exprsesed as integers from 1 to 12, days are expressed
// as integers from 1 to 31, and years are expressed as 4-digit
// integers.

string dayOfWeek[] = {"Mo", "Tu", "We", "Th", "Fr", "Sa", "Su"};

// converts Gregorian date to integer (Julian day number)

int DateToInt (int m, int d, int y){
  return
    1461 * (y + 4800 + (m - 14) / 12) / 4 +
    367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
    3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 +
    d - 32075;
}

// converts integer (Julian day number) to Gregorian date: month/day/year

void IntToDate (int jd, int &m, int &d, int &y){
  int x, n, i, j;

  x = jd + 68569;
  n = 4 * x / 146097;
  x -= (146097 * n + 3) / 4;
  i = (4000 * (x + 1)) / 1461001;
  x -= 1461 * i / 4 - 31;
  j = 80 * x / 2447;
  d = x - 2447 * j / 80;
  x = j / 11;
  m = j + 2 - 12 * x;
  y = 100 * (n - 49) + i + x;
}

// converts integer (Julian day number) to day of week

string IntToDay (int jd){
  return dayOfWeek[jd % 7];
}
```

0.5.3 make CIN and COUT Run Faseter

```
ios_base::sync_with_stdio(false);
```