**Segment Tree :**

#include <cassert>

#include <cctype>

#include <cmath>

#include <cstdio>

#include <cstdlib>

#include <cstring>

#include <iostream>

#include <sstream>

#include <iomanip>

#include <string>

#include <vector>

#include <list>

#include <set>

#include <map>

#include <stack>

#include <queue>

#include <algorithm>

#include <iterator>

#include <utility>

using namespace std;

template< class T > T abs(T n) { return (n < 0 ? -n : n); }

template< class T > T max(T a, T b) { return (!(a < b) ? a : b); }

template< class T > T min(T a, T b) { return (a < b ? a : b); }

template< class T > T sq(T x) { return x \* x; }

template< class T > T gcd(T a, T b) { return (b != 0 ? gcd<T>(b, a%b) : a); }

template< class T > T lcm(T a, T b) { return (a / gcd<T>(a, b) \* b); }

template< class T > bool inside(T a, T b, T c) { return a<=b && b<=c; }

template< class T > void setmax(T &a, T b) { if(a < b) a = b; }

template< class T > void setmin(T &a, T b) { if(b < a) a = b; }

#define MP(x, y) make\_pair(x, y)

#define REV(s, e) reverse(s, e)

#define SET(p) memset(p, -1, sizeof(p))

#define CLR(p) memset(p, 0, sizeof(p))

#define MEM(p, v) memset(p, v, sizeof(p))

#define CPY(d, s) memcpy(d, s, sizeof(s))

#define READ(f) freopen(f, "r", stdin)

#define WRITE(f) freopen(f, "w", stdout)

#define ALL(c) c.begin(), c.end()

#define SIZE(c) (int)c.size()

#define PB(x) push\_back(x)

#define ff first

#define ss second

#define i64 long long

#define ld long double

#define pii pair< int, int >

#define psi pair< string, int >

const double EPS = 1e-9;

const double BIG = 1e19;

const int INF = 0x7f7f7f7f;

const int MAX = 1 << 18;

i64 Tree[MAX][2]; // 0 sum, 1 add

void update(int Node, int i, int j, int x, int y, int v) {

if(i == x && j == y) {

Tree[Node][1] += v;

Tree[Node][0] += (i64)v \* (y-x+1);

return;

}

int m, l, r;

m = (i + j) >> 1; l = Node << 1; r = l + 1;

if(y <= m) update(l, i, m, x, y, v);

else if(x > m) update(r, m+1, j, x, y, v);

else {

update(l, i, m, x, m, v);

update(r, m+1, j, m+1, y, v);

}

Tree[Node][0] = Tree[l][0] + Tree[r][0] + Tree[Node][1] \* (j-i+1);

}

i64 query(int Node, int i, int j, int x, int y, i64 v) {

if(i == x && j == y) return Tree[Node][0] + v \* (y-x+1);

int m, l, r;

m = (i + j) >> 1; l = Node << 1; r = l + 1;

if(y <= m) return query(l, i, m, x, y, v + Tree[Node][1]);

else if(x > m) return query(r, m+1, j, x, y, v + Tree[Node][1]);

else {

i64 ret = 0;

ret += query(l, i, m, x, m, v + Tree[Node][1]);

ret += query(r, m+1, j, m+1, y, v + Tree[Node][1]);

return ret;

}

}

Binary Indexed Tree :

#include <vector>

using namespace std;

vector<int> create(int n) { return vector<int>(n, 0); }

int query(const vector<int> &tree, int a, int b) {

if (a == 0) {

int sum = 0;

for (; b >= 0; b = (b & (b + 1)) - 1)

sum += tree[b];

return sum;

} else {

return query(tree, 0, b) - query(tree, 0, a-1);

}

}

void increase(vector<int> &tree, int k, int inc) {

for (; k < (int)tree.size(); k |= k + 1)

tree[k] += inc;

}

**KMP String-Matching :**

#include <iostream>

#include <vector>

using namespace std;

vector<int> KMP(string S, string K)

{

vector<int> T(K.size() + 1, -1);

vector<int> matches;

if(K.size() == 0)

{

matches.push\_back(0);

return matches;

}

for(int i = 1; i <= K.size(); i++)

{

int pos = T[i - 1];

while(pos != -1 && K[pos] != K[i - 1]) pos = T[pos];

T[i] = pos + 1;

}

int sp = 0;

int kp = 0;

while(sp < S.size())

{

while(kp != -1 && (kp == K.size() || K[kp] != S[sp])) kp = T[kp];

kp++;

sp++;

if(kp == K.size()) matches.push\_back(sp - K.size());

}

return matches;

}

**0-1 Knapsack :**

#include <stdio.h>

#define MAXWEIGHT 100

int n; /\* The number of objects \*/

int c[10]; /\* c[i] is the \*COST\* of the ith object; i.e. what YOU PAY to take the object \*/

int v[10]; /\* v[i] is the \*VALUE\* of the ith object; i.e. what YOU GET for taking the object \*/

int W = 10; /\* The maximum weight you can take \*/

void fill\_sack() {

int a[MAXWEIGHT]; /\* a[i] holds the maximum value that can be obtained using at most i weight \*/

int last\_added[MAXWEIGHT]; /\* I use this to calculate which object were added \*/

int i, j;

int aux;

for (i = 0; i <= W; ++i) {

a[i] = 0;

last\_added[i] = -1;

}

a[0] = 0;

for (i = 1; i <= W; ++i)

for (j = 0; j < n; ++j)

if ((c[j] <= i) && (a[i] < a[i - c[j]] + v[j])) {

a[i] = a[i - c[j]] + v[j];

last\_added[i] = j;

}

for (i = 0; i <= W; ++i)

if (last\_added[i] != -1)

printf("Weight %d; Benefit: %d; To reach this weight I added object %d (%d$ %dKg) to weight %d.\n", i, a[i], last\_added[i] + 1, v[last\_added[i]], c[last\_added[i]], i - c[last\_added[i]]);

else

printf("Weight %d; Benefit: 0; Can't reach this exact weight.\n", i);

printf("---\n");

aux = W;

while ((aux > 0) && (last\_added[aux] != -1)) {

printf("Added object %d (%d$ %dKg). Space left: %d\n", last\_added[aux] + 1, v[last\_added[aux]], c[last\_added[aux]], aux - c[last\_added[aux]]);

aux -= c[last\_added[aux]];

}

printf("Total value added: %d$\n", a[W]);

}

**Bellman-Ford :**

#include <stdio.h>

typedef struct {

int u, v, w;

} Edge;

int n; /\* the number of nodes \*/

int e; /\* the number of edges \*/

Edge edges[1024]; /\* large enough for n <= 2^5=32 \*/

int d[32]; /\* d[i] is the minimum distance from node s to node i \*/

#define INFINITY 10000

void printDist() {

int i;

printf("Distances:\n");

for (i = 0; i < n; ++i)

printf("to %d\t", i + 1);

printf("\n");

for (i = 0; i < n; ++i)

printf("%d\t", d[i]);

printf("\n\n");

}

void bellman\_ford(int s) {

int i, j;

for (i = 0; i < n; ++i)

d[i] = INFINITY;

d[s] = 0;

for (i = 0; i < n - 1; ++i)

for (j = 0; j < e; ++j)

if (d[edges[j].u] + edges[j].w < d[edges[j].v])

d[edges[j].v] = d[edges[j].u] + edges[j].w;

}

int main(int argc, char \*argv[]) {

int i, j;

int w;

FILE \*fin = fopen("dist.txt", "r");

fscanf(fin, "%d", &n);

e = 0;

for (i = 0; i < n; ++i)

for (j = 0; j < n; ++j) {

fscanf(fin, "%d", &w);

if (w != 0) {

edges[e].u = i;

edges[e].v = j;

edges[e].w = w;

++e;

}

}

fclose(fin);

bellman\_ford(0);

printDist();

return 0;

}

**Dijkstra :**

#include <iostream>

#include <vector>

#include <string>

#include <map>

#include <list>

#include <limits> // for numeric\_limits

#include <set>

#include <utility> // for pair

typedef int vertex\_t;

typedef double weight\_t;

struct edge {

const vertex\_t target;

const weight\_t weight;

edge(vertex\_t arg\_target, weight\_t arg\_weight)

: target(arg\_target), weight(arg\_weight) { }

};

typedef std::map<vertex\_t, std::list<edge> > adjacency\_map\_t;

void DijkstraComputePaths(vertex\_t source,

const adjacency\_map\_t &adjacency\_map,

std::map<vertex\_t, weight\_t> &min\_distance,

std::map<vertex\_t, vertex\_t> &previous)

{

for (adjacency\_map\_t::const\_iterator vertex\_iter = adjacency\_map.begin();

vertex\_iter != adjacency\_map.end();

vertex\_iter++)

{

vertex\_t v = vertex\_iter->first;

min\_distance[v] = std::numeric\_limits< double >::infinity();

for (std::list<edge>::const\_iterator edge\_iter = vertex\_iter->second.begin();

edge\_iter != vertex\_iter->second.end();

edge\_iter++)

{

vertex\_t v2 = edge\_iter->target;

min\_distance[v2] = std::numeric\_limits< double >::infinity();

}

}

min\_distance[source] = 0;

std::set< std::pair<weight\_t, vertex\_t> > vertex\_queue;

vertex\_queue.insert(std::make\_pair(min\_distance[source], source));

while (!vertex\_queue.empty())

{

vertex\_t u = vertex\_queue.begin()->second;

vertex\_queue.erase(vertex\_queue.begin());

// Visit each edge exiting u

const std::list<edge> &edges = adjacency\_map.find(u)->second;

for (std::list<edge>::const\_iterator edge\_iter = edges.begin();

edge\_iter != edges.end();

edge\_iter++)

{

vertex\_t v = edge\_iter->target;

weight\_t weight = edge\_iter->weight;

weight\_t distance\_through\_u = min\_distance[u] + weight;

if (distance\_through\_u < min\_distance[v]) {

vertex\_queue.erase(std::make\_pair(min\_distance[v], v));

min\_distance[v] = distance\_through\_u;

previous[v] = u;

vertex\_queue.insert(std::make\_pair(min\_distance[v], v));

}

}

}

}

std::list<vertex\_t> DijkstraGetShortestPathTo(

vertex\_t target, const std::map<vertex\_t, vertex\_t> &previous)

{

std::list<vertex\_t> path;

std::map<vertex\_t, vertex\_t>::const\_iterator prev;

vertex\_t vertex = target;

path.push\_front(vertex);

while((prev = previous.find(vertex)) != previous.end())

{

vertex = prev->second;

path.push\_front(vertex);

}

return path;

}

int main()

{

adjacency\_map\_t adjacency\_map;

std::vector<std::string> vertex\_names;

vertex\_names.push\_back("Harrisburg"); // 0

vertex\_names.push\_back("Baltimore"); // 1

vertex\_names.push\_back("Washington"); // 2

vertex\_names.push\_back("Philadelphia"); // 3

vertex\_names.push\_back("Binghamton"); // 4

vertex\_names.push\_back("Allentown"); // 5

vertex\_names.push\_back("New York"); // 6

adjacency\_map[0].push\_back(edge(1, 79.83));

adjacency\_map[0].push\_back(edge(5, 81.15));

adjacency\_map[1].push\_back(edge(0, 79.75));

adjacency\_map[1].push\_back(edge(2, 39.42));

adjacency\_map[1].push\_back(edge(3, 103.00));

adjacency\_map[2].push\_back(edge(1, 38.65));

adjacency\_map[3].push\_back(edge(1, 102.53));

adjacency\_map[3].push\_back(edge(5, 61.44));

adjacency\_map[3].push\_back(edge(6, 96.79));

adjacency\_map[4].push\_back(edge(5, 133.04));

adjacency\_map[5].push\_back(edge(0, 81.77));

adjacency\_map[5].push\_back(edge(3, 62.05));

adjacency\_map[5].push\_back(edge(4, 134.47));

adjacency\_map[5].push\_back(edge(6, 91.63));

adjacency\_map[6].push\_back(edge(3, 97.24));

adjacency\_map[6].push\_back(edge(5, 87.94));

std::map<vertex\_t, weight\_t> min\_distance;

std::map<vertex\_t, vertex\_t> previous;

DijkstraComputePaths(0, adjacency\_map, min\_distance, previous);

for (adjacency\_map\_t::const\_iterator vertex\_iter = adjacency\_map.begin();

vertex\_iter != adjacency\_map.end();

vertex\_iter++)

{

vertex\_t v = vertex\_iter->first;

std::cout << "Distance to " << vertex\_names[v] << ": " << min\_distance[v] << std::endl;

std::list<vertex\_t> path =

DijkstraGetShortestPathTo(v, previous);

std::list<vertex\_t>::const\_iterator path\_iter = path.begin();

std::cout << "Path: ";

for( ; path\_iter != path.end(); path\_iter++)

{

std::cout << vertex\_names[\*path\_iter] << " ";

}

std::cout << std::endl;

}

return 0;

}

**Strongly Connected Components :**

#include <iostream>

#include <fstream>

#include <cstdio>

#include <vector>

#define N 875714

using namespace std;

struct node{

bool visited;

int leader;

int finish;

vector<int> linkedVertices;

vector<int> rLinkedVertices;

};

struct node G[N+1];

struct node newG[N+1];

int t=0;

int s=0;

void dfs(node G[], int i, bool reverse){

G[i].visited=true;

G[i].leader=s;

vector<int> next;

if (reverse) next= G[i].rLinkedVertices;

else next= G[i].linkedVertices;

for(int j=0; j<next.size(); j++){

if(!G[next[j]].visited) {

dfs(G, next[j], reverse);

}

}

t++;

G[i].finish=t;

}

void dfs\_loop(node G[], bool reverse){

t=0;

s=0;

for(int i=N;i>0;--i){

if (!G[i].visited){

s=i;

dfs(G,i,reverse);

}

}

}

int main(){

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

G[i].visited=false;

}

FILE\* fp=freopen("SCC.txt","r",stdin);

int head, tail;

while (scanf("%d %d", &head, &tail) > 0) {

G[head].linkedVertices.push\_back(tail);

G[tail].rLinkedVertices.push\_back(head);

}

fclose(fp);

dfs\_loop(G, true);//FIRST LOOP

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

newG[i].visited=false;

newG[i].finish=0;

newG[i].leader=0;

vector<int> temp;

for(int j=0; j< G[i].linkedVertices.size(); j++){

temp.push\_back(G[G[i].linkedVertices[j]].finish);

}

newG[G[i].finish].linkedVertices=temp;

}

dfs\_loop(newG,false);//SECOND LOOP

ofstream ofs("stat.txt", ofstream::out);

for (int k=1;k<=N;k++) ofs<< newG[k].leader<<endl;

ofs.close();

return 0;

}

**Gaussian elimination - Simplex :**

#include <stdio.h>

int n;

float a[10][11];

void forwardSubstitution() {

int i, j, k, max;

float t;

for (i = 0; i < n; ++i) {

max = i;

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

if (a[j][i] > a[max][i])

max = j;

for (j = 0; j < n + 1; ++j) {

t = a[max][j];

a[max][j] = a[i][j];

a[i][j] = t;

}

for (j = n; j >= i; --j)

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

a[k][j] -= a[k][i]/a[i][i] \* a[i][j];

/\* for (k = 0; k < n; ++k) {

for (j = 0; j < n + 1; ++j)

printf("%.2f\t", a[k][j]);

printf("\n");

}\*/

}

}

void reverseElimination() {

int i, j;

for (i = n - 1; i >= 0; --i) {

a[i][n] = a[i][n] / a[i][i];

a[i][i] = 1;

for (j = i - 1; j >= 0; --j) {

a[j][n] -= a[j][i] \* a[i][n];

a[j][i] = 0;

}

}

}

void gauss() {

int i, j;

forwardSubstitution();

reverseElimination();

for (i = 0; i < n; ++i) {

for (j = 0; j < n + 1; ++j)

printf("%.2f\t", a[i][j]);

printf("\n");

}

}

int main(int argc, char \*argv[]) {

int i, j;

FILE \*fin = fopen("gauss.in", "r");

fscanf(fin, "%d", &n);

for (i = 0; i < n; ++i)

for (j = 0; j < n + 1; ++j)

fscanf(fin, "%f", &a[i][j]);

fclose(fin);

gauss();

return 0;

}

Kadane's algorithm :

#include<stdio.h>

int maxSubArraySum(int a[], int size)

{

   int max\_so\_far = 0, max\_ending\_here = 0;

   int i;

   for(i = 0; i < size; i++)

   {

     max\_ending\_here = max\_ending\_here + a[i];

     if(max\_ending\_here < 0)

        max\_ending\_here = 0;

     else if(max\_so\_far < max\_ending\_here)

        max\_so\_far = max\_ending\_here;

    }

    return max\_so\_far;

}

/\*Driver program to test maxSubArraySum\*/

int main()

{

   int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};

   int max\_sum = maxSubArraySum(a, 8);

   printf("Maximum contiguous sum is %d\n", max\_sum);

   getchar();

   return 0;

}

**Prime Sieve :**

#include <iostream>

#include <vector>

#include <string.h>

using namespace std;

vector<unsigned long> get\_primes(unsigned long max){

vector<unsigned long> primes;

char \*sieve;

sieve = new char[max/8+1];

// Fill sieve with 1

memset(sieve, 0xFF, (max/8+1) \* sizeof(char));

for(unsigned long x = 2; x <= max; x++)

if(sieve[x/8] & (0x01 << (x % 8))){

primes.push\_back(x);

// Is prime. Mark multiplicates.

for(unsigned long j = 2\*x; j <= max; j += x)

sieve[j/8] &= ~(0x01 << (j % 8));

}

delete[] sieve;

return primes;

}

int main(void){

vector<unsigned long> primes;

primes = get\_primes(10000000);

vector<unsigned long>::iterator it;

for(it=primes.begin(); it < primes.end(); it++)

cout << \*it << " ";

cout << endl;

return 0;

}

**Fast Exponentiation :**

#include<stdio.h>

int expo(int a, int b,int mod){

int result = 1;

while (b){

if (b&1){

result = (result\*a)%mod;

}

b >>=1 ;

a = (a\*a)%mod;

}

return result;

}