WashU ICPC Notebook

Contents

1 Techniques

1.1 Various algorithm techniques

```
Recursion
Divide and conquer
        Finding interesting points in N log N
Greedy algorithm
        Scheduling
        Max contigous subvector sum
        Invariants
        Huffman encoding
Graph theory
        Dynamic graphs (extra book-keeping)
        Breadth first search
        Depth first search
        * Normal trees / DFS trees
        Dijkstra's algoritm
        MST: Prim's algoritm
        Bellman-Ford
        Konig's theorem and vertex cover
        Min-cost max flow
        Lovasz toggle
        Matrix tree theorem
        Maximal matching, general graphs
        Hopcroft-Karp
        Hall's marriage theorem
        Graphical sequences
        Floyd-Warshall
        Eulercykler
        Flow networks
        * Augumenting paths
        * Edmonds-Karp
        Bipartite matching
        Min. path cover
        Topological sorting
        Strongly connected components
        2-SAT
        Cutvertices, cutedges och biconnected components
        Edge coloring
        * Trees
        Vertex coloring
        * Bipartite graphs (=> trees)
        * 3^n (special case of set cover)
        Diameter and centroid
        K'th shortest path
        Shortest cycle
Dynamic programmering
        Knapsack
        Coin change
        Longest common subsequence
        Longest increasing subsequence
        Number of paths in a dag
        Shortest path in a dag
        Dynprog over intervals
        Dynprog over subsets
        Dynprog over probabilities
        Dynprog over trees
        3^n set cover
        Divide and conquer
        Knuth optimization
        Convex hull optimizations
        RMQ (sparse table a.k.a 2^k-jumps)
        Bitonic cycle
        Log partitioning (loop over most restricted)
Combinatorics
        Computation of binomial coefficients
        Pigeon-hole principle
        Inclusion/exclusion
        Catalan number
        Pick's theorem
Number theory
        Integer parts
        Divisibility
        Euklidean algorithm
        Modular arithmetic
        * Modular multiplication
        * Modular inverses
        * Modular exponentiation by squaring
        Chinese remainder theorem
        Fermat's small theorem
        Euler's theorem
        Phi function
        Frobenius number
```

```
        Quadratic reciprocity
        Pollard-Rho
        Miller-Rabin
        Hensel lifting
        Vieta root jumping
Game theory
        Combinatorial games
        Game trees
        Mini-max
        Nim
        Games on graphs
        Games on graphs with loops
        Grundy numbers
        Bipartite games without repetition
        General games without repetition
        Alpha-beta pruning
Probability theory
Optimization
        Binary search
        Ternary search
        Unimodality and convex functions
        Binary search on derivative
Numerical methods
        Numeric integration
        Newton's method
        Root-finding with binary/ternary search
        Golden section search
Matrices
        Gaussian elimination
        Exponentiation by squaring
Sorting
        Radix sort
Geometry
        Coordinates and vectors
        * Cross product
        * Scalar product
        Convex hull
        Polygon cut
        Closest pair
        Coordinate-compression
        Quadtrees
        KD-trees
        All segment-segment intersection
Sweeping
        Discretization (convert to events and sweep)
        Angle sweeping
        Line sweeping
        Discrete second derivatives
Strings
        Longest common substring
        Palindrome subsequences
        Knuth-Morris-Pratt
        Tries
        Rolling polynom hashes
        Suffix array
        Suffix tree
        Aho-Corasick
        Manacher's algorithm
        Letter position lists
Combinatorial search
        Meet in the middle
        Brute-force with pruning
        Best-first (A*)
        Bidirectional search
        Iterative deepening DFS / A*
Data structures
        LCA (2^k-jumps in trees in general)
        Pull/push-technique on trees
        Heavy-light decomposition
        Centroid decomposition
        Lazy propagation
        Self-balancing trees
        Convex hull trick (wcipeg.com/wiki/Convex_hull_trick)
        Monotone queues / monotone stacks / sliding queues
        Sliding queue using 2 stacks
        Persistent segment tree
```

2 Dynamic Programming

2.1 Max Sum Subarray (Kadane's Algorithm)

```python
def maxSubArraySum(a,size):
    max_so_far = 0
    max_ending_here = 0

    for i in range(0, size):
        max_ending_here = max_ending_here + a[i]
        if max_ending_here < 0:
            max_ending_here = 0
        elif (max_so_far < max_ending_here):
            max_so_far = max_ending_here
    return max_so_far
```

2.2 Longest Common Subsequence

```python
def lcs(X , Y):
    # find the length of the strings
    m = len(X)
    n = len(Y)

    # declaring the array for storing the dp values
    L = [[None]*(n+1) for i in xrange(m+1)]

    """Following steps build L[m+1][n+1] in bottom up fashion
    Note: L[i][j] contains length of LCS of X[0..i-1]
    and Y[0..j-1]"""
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                L[i][j] = 0
            elif 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])

    # L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]
    return L[m][n]
```

2.3 Levenshtein Distance

```python
def levenshtein(s1, s2):
    if len(s1) < len(s2):
        return levenshtein(s2, s1)

    # len(s1) >= len(s2)
    if len(s2) == 0:
        return len(s1)

    previous_row = range(len(s2) + 1)
    for i, c1 in enumerate(s1):
        current_row = [i + 1]
        for j, c2 in enumerate(s2):
            insertions = previous_row[j + 1] + 1 # j+1 instead of j since previous_row and current_row
                    are one character longer
            deletions = current_row[j] + 1       # than s2
            substitutions = previous_row[j] + (c1 != c2)
            current_row.append(min(insertions, deletions, substitutions))
        previous_row = current_row
    return previous_row[-1]
```

2.4 Longest Increasing Subsequence

```python
def lis(arr):
    n = len(arr)

    # Declare the list (array) for LIS and initialize LIS
    # values for all indexes
    lis = [1]*n

    # Compute optimized LIS values in bottom up manner
    for i in range (1 , n):
        for j in range(0 , i):
            if arr[i] > arr[j] and lis[i]< lis[j] + 1 :
                lis[i] = lis[j]+1

    # Initialize maximum to 0 to get the maximum of all
    # LIS
    maximum = 0

    # Pick maximum of all LIS values
    for i in range(n):
        maximum = max(maximum , lis[i])
    return maximum
```

3 Geometry

3.1 Convex Hull Algorithm

```
bool compare(PT a,PT b){    return a.y<b.y || (a.y==b.y && a.x<b.x); }
double cross(PT o,PT a, PT b)
{
        return (a.x-o.x)*(b.y-o.y)-(a.y-o.y)*(b.x-o.x);
}

vector<PT> ConvexHull(vector<PT> p) {    int n=p.size();    int k=0;
    vector<PT> h(2*n);
    sort(p.begin(),p.end(),compare);
        //build lower hull
        for(int i=0;i<n;++i)
        {
                while(k>=2 && cross(h[k-2],h[k-1],p[i])<=0) k--;
                h[k++]=p[i];
        }
        //build top hull
        for(int i=n-2,t=k+1;i>=0;--i)
        {
                while(k>=t && cross(h[k-2],h[k-1],p[i])<=0) k--;
                h[k++]=p[i];
        }
        h.resize(k);
        return h;
}
```

3.2 Convex Hull (Python)

```
def convex_hull(points):
    """Computes the convex hull of a set of 2D points.

    Input: an iterable sequence of (x, y) pairs representing the points.
    Output: a list of vertices of the convex hull in counter-clockwise order,
      starting from the vertex with the lexicographically smallest coordinates.
    Implements Andrew's monotone chain algorithm. O(n log n) complexity.
    """

    # Sort the points lexicographically (tuples are compared lexicographically).
    # Remove duplicates to detect the case we have just one unique point.
    points = sorted(set(points))

    # Boring case: no points or a single point, possibly repeated multiple times.
    if len(points) <= 1:
        return points

    # 2D cross product of OA and OB vectors, i.e. z-component of their 3D cross product.
    # Returns a positive value, if OAB makes a counter-clockwise turn,
    # negative for clockwise turn, and zero if the points are collinear.
    def cross(o, a, b):
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

    # Build lower hull
    lower = []
    for p in points:
        while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:
            lower.pop()
        lower.append(p)

    # Build upper hull
    upper = []
    for p in reversed(points):
        while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
            upper.pop()
        upper.append(p)

    # Concatenation of the lower and upper hulls gives the convex hull.
    # Last point of each list is omitted because it is repeated at the beginning of the other list.
    return lower[:-1] + upper[:-1]

# Example: convex hull of a 10-by-10 grid.
assert convex_hull([(i//10, i%10) for i in range(100)]) == [(0, 0), (9, 0), (9, 9), (0, 9)]
```

3.3 Delaunay Triangulation

```
/*
Stanford notebook
-----------------------------------------------------
Delaunay Algorithm Does not handle degenerate cases
Running time: O(n^4)
INPUT: x[] = x-coordinates
          y[] = y-coordinates
OUTPUT: triples = a vector containing m triples
(indices corresponding to triangle vertices)
-----------------------------------------------------
*/

typedef double T;

struct triple {
        int i, j, k;
        triple() {}
        triple(int i, int j, int k) : i(i), j(j), k(k) {}
};

vector<triple> delaunayTriangulation(vector<T>& x, vector<T>& y)
{
        int n = x.size();
        vector<T> z(n);
        vector<triple> ret;

    for (int i = 0; i < n; i++)
        z[i] = x[i] * x[i] + y[i] * y[i];
    for (int i = 0; i < n-2; i++)
    {
        for (int j = i+1; j < n; j++)
        {
                for (int k = i+1; k < n; k++)
                {
                        if (j == k) continue;
                        double xn = (y[j]-y[i])*(z[k]-z[i]) - (y[k]-y[i])*(z[j]-z[i]);
                double yn = (x[k]-x[i])*(z[j]-z[i]) - (x[j]-x[i])*(z[k]-z[i]);
                double zn = (x[j]-x[i])*(y[k]-y[i]) - (x[k]-x[i])*(y[j]-y[i]);
                bool flag = zn < 0;
                for (int m = 0; flag && m < n; m++)
                        flag = flag && ((x[m]-x[i])*xn + (y[m]-y[i])*yn + (z[m]-z[i])*zn <= 0);
                if (flag) ret.push_back(triple(i, j, k));
            }
        }
    }
    return ret;
}

int main() {
        T xs[]={0, 0, 1, 0.9};
        T ys[]={0, 1, 0, 0.9};
        vector<T> x(&xs[0], &xs[4]), y(&ys[0], &ys[4]);
        vector<triple> tri = delaunayTriangulation(x, y);
        //expected: 0 1 3
        //          0 3 2
        int i;
        for(i = 0; i < tri.size(); i++)
                printf("%d %d %d\n", tri[i].i, tri[i].j, tri[i].k);
        return 0;
}
```

3.4 Various Geometry Functions

```
// Stanford Notebook
double INF = 1e100;
double EPS = 1e-12;


struct PT {
        double x, y;
        PT() {}
        PT(double x, double y) : x(x), y(y) {}
        PT(const PT &p) : x(p.x), y(p.y)    {}
        PT operator + (const PT &p)  const { return PT(x+p.x, y+p.y); }
        PT operator - (const PT &p)  const { return PT(x-p.x, y-p.y); }
        PT operator * (double c)     const { return PT(x*c,   y*c  ); }
        PT operator / (double c)     const { return PT(x/c,   y/c  ); }
};

double dot(PT p, PT q)     { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q)   { return dot(p-q,p-q); }
double cross(PT p, PT q)   { return p.x*q.y-p.y*q.x; }

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p)   { return PT(-p.y,p.x); }
PT RotateCW90(PT p)    { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {     return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t)); }
```

```
// project point c onto line through a and b // assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) { return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a); }
// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c)
{
        double r = dot(b-a,b-a);
        if (fabs(r) < EPS) return a;
        r = dot(c-a, b-a)/r;
        if (r < 0) return a;
        if (r > 1) return b;
        return a + (b-a)*r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c)
{
        return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z, double a, double b, double c, double d)
{
    return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {  return fabs(cross(b-a, c-d)) < EPS; }

bool LinesCollinear(PT a, PT b, PT c, PT d)
{
        return LinesParallel(a, b, c, d)
        && fabs(cross(a-b, a-c)) < EPS
        && fabs(cross(c-d, c-a)) < EPS;
}


// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d)
{
        if (LinesCollinear(a, b, c, d))
        {
                if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
                dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
                if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
                        return false;
                return true;
        }
        if (cross(d-a, b-a) * cross(c-a, b-a) > 0)
                return false;
        if (cross(a-c, d-c) * cross(b-c, d-c) > 0)
                return false;
        return true;
}


// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d)
{
        b=b-a; d=c-d; c=c-a;
        assert(dot(b, b) > EPS && dot(d, d) > EPS);
        return a + b*cross(c, d)/cross(b, d);
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c)
{
        b=(a+b)/2;
        c=(a+c)/2;
        return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-c));
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q)
{
        bool c = 0;
        for (int i = 0; i < p.size(); i++)
        {
                int j = (i+1)%p.size();
                if ((p[i].y <= q.y && q.y < p[j].y ||
                p[j].y <= q.y && q.y < p[i].y) &&
                q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
                        c = !c;
        }
        return c;
}


// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q)
{
        for (int i = 0; i < p.size(); i++)
                if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
                        return true;
        return false;
}

// compute intersection of line through points a and b with
 // circle centered at c with radius r > 0
 vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r)
 {
        vector<PT> ret;
        b = b-a; a = a-c;
        double A = dot(b, b);
        double B = dot(a, b);
        double C = dot(a, a) - r*r;
        double D = B*B - A*C;
        if (D < -EPS) return ret;
        ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
        if (D > EPS)
                ret.push_back(c+a+b*(-B-sqrt(D))/A);
        return ret;
}


// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R)
{
        vector<PT> ret;
        double d = sqrt(dist2(a, b));
        if (d > r+R || d+min(r, R) < max(r, R)) return ret;
        double x = (d*d-R*R+r*r)/(2*d);
        double y = sqrt(r*r-x*x);
        PT v = (b-a)/d;
        ret.push_back(a+v*x + RotateCCW90(v)*y);
        if (y > 0) ret.push_back(a+v*x - RotateCCW90(v)*y);
        return ret;
}

// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion.  Note that the centroid is often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p)
{
        double area = 0;
        for(int i = 0; i < p.size(); i++)
        {
                int j = (i+1) % p.size();
                area += p[i].x*p[j].y - p[j].x*p[i].y;
        }
        return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {    return fabs(ComputeSignedArea(p)); }

PT ComputeCentroid(const vector<PT> &p)
{
        PT c(0,0);
        double scale = 6.0 * ComputeSignedArea(p);
        for (int i = 0; i < p.size(); i++)
        {
                int j = (i+1) % p.size();
                c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
        }
        return c / scale;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p)
{
        for (int i = 0; i < p.size(); i++)
        {
                for (int k = i+1; k < p.size(); k++)
                {
                        int j = (i+1) % p.size();
                        int l = (k+1) % p.size();
                        if (i == l || j == k) continue;
                        if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
                                return false;
                }
        }
        return true;
```

```
	}
```

4 Graphs

4.1 Dijkstra's Algorithm

```
//Dijkstra Algorithm
int t,n,m,s,e;
vector<ii> edges[N];//pair<NodeEnd,dist>
int distances[N]; // =INF=0x3f3f3f3f
int parent[N]; // =-1

int Dijkstra()
{
	vector<ii> :: iterator it;
	priority_queue< ii, vector<ii>, greater<ii> > pq;
	distances[s]=0;
	pq.push(ii(distances[s],s));
	while(!pq.empty())
	{
		ii p = pq.top();
		pq.pop();
		int d=p.first;
		int a=p.second;
		for(it=edges[a].begin();it!=edges[a].end();++it)
		{
			if(distances[it->first]>distances[a]+it->second)
			{
			    distances[it->first]=distances[a]+it->second;
			    parent[it->first]=a;
			    pq.push(ii(distances[it->first],it->first));
			}
		}
	}
	return distances[e];
}
```

4.2 Max Flow (Dinic's Algorithm)

```
// Stanford Notebook
typedef long long LL;

struct Edge
{
	int from, to, cap, flow, index;
	Edge(int from, int to, int cap, int flow, int index) :
	from(from), to(to), cap(cap), flow(flow), index(index) {}
	LL rcap() { return cap - flow; }
};

struct Dinic
{
	int N;
	vector<vector<Edge> > G;
	vector<vector<Edge *> > Lf;
	vector<int> layer;
	vector<int> Q;
	Dinic(int N) : N(N), G(N), Q(N) {}
	void AddEdge(int from, int to, int cap)
	{
		if (from == to) return;
		G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
		G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
	}

	LL BlockingFlow(int s, int t)
	{
		layer.clear();
		layer.resize(N, -1);
		layer[s] = 0;
		Lf.clear(); Lf.resize(N);

	int head = 0, tail = 0;
		Q[tail++] = s;
	while (head < tail)
		{
	    int x = Q[head++];
			for (int i = 0; i < G[x].size(); i++)
			{
				Edge &e = G[x][i]; if (e.rcap() <= 0) continue;
				if (layer[e.to] == -1)
				{
					layer[e.to] = layer[e.from] + 1;
					Q[tail++] = e.to;
				}
				if (layer[e.to] > layer[e.from])
				{
					Lf[e.from].push_back(&e);
				}
			}
		}
	if (layer[t] == -1) return 0;
	LL totflow = 0;
		vector<Edge *> P;
		while (!Lf[s].empty())
		{
			int curr = P.empty() ? s : P.back()->to;
			if (curr == t)
			{
				// Augment
				LL amt = P.front()->rcap();
				for (int i = 0; i < P.size(); ++i)
				{
					amt = min(amt, P[i]->rcap());
				}
				totflow += amt;
				for (int i = P.size() - 1; i >= 0; --i)
				{
					P[i]->flow += amt;
					G[P[i]->to][P[i]->index].flow -= amt;
					if (P[i]->rcap() <= 0)
					{
						Lf[P[i]->from].pop_back();
						P.resize(i);
					}
				}
			}
			else if (Lf[curr].empty())
			{
				// Retreat
				P.pop_back();
				for (int i = 0; i < N; ++i)
					for (int j = 0; j < Lf[i].size(); ++j)
						if (Lf[i][j]->to == curr)
							Lf[i].erase(Lf[i].begin() + j);
			}
			else
			{
				// Advance
				P.push_back(Lf[curr].back());
			}
		}
		return totflow;
	}

	LL GetMaxFlow(int s, int t)
	{
		LL totflow = 0;
		while (LL flow = BlockingFlow(s, t))
			totflow += flow;
		return totflow;
	}
};
```

4.3 Max Flow (Edmonds-Karp Algorithm)

```
/*
Stanford Notebook
MinCostMaxFlow (adjacency matrix, Edmonds and Karp 1972)
This implementation keeps track of forward and reverse edges separately
(so you can set cap[i][j] != cap[j][i]). For a regular max flow, set all edge costs to 0.
Running time, O(|V|^2) cost per augmentation
	max flow:           O(|V|^3) augmentations
	min cost max flow:  O(|V|^4 * MAX_EDGE_COST) augmentations
INPUT: - graph, constructed using AddEdge()
	     - source
	     - sink
OUTPUT: - (maximum flow value, minimum cost value)
	        - To obtain the actual flow, look at positive values only.
*/

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef long long L;
typedef vector<L> VL;
typedef vector<VL> VVL;
typedef pair<int, int> PII;
```

```
typedef vector<PII> VPII;

const L INF = numeric_limits<L>::max() / 4;

struct MinCostMaxFlow {
	int N;
	VVL cap, flow, cost;
	VI found;
	VL dist, pi, width;
	VPII dad;

    MinCostMaxFlow(int N) :
    N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),
    found(N), dist(N), pi(N), width(N), dad(N) {}

    void AddEdge(int from, int to, L cap, L cost)
    {
        this->cap[from][to] = cap;
        this->cost[from][to] = cost;
    }

    void Relax(int s, int k, L cap, L cost, int dir)
    {
        L val = dist[s] + pi[s] - pi[k] + cost;
        if (cap && val < dist[k])
        {
                dist[k] = val;
                dad[k] = make_pair(s, dir);
                width[k] = min(cap, width[s]);
        }
    }

    L Dijkstra(int s, int t)
    {
        fill(found.begin(), found.end(), false);
        fill(dist.begin(), dist.end(), INF);
        fill(width.begin(), width.end(), 0);
        dist[s] = 0;
        width[s] = INF;

        while (s != -1)
        {
                int best = -1;
                found[s] = true;
                for (int k = 0; k < N; k++)
                {
                        if (found[k]) continue;
                        Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);
                        Relax(s, k, flow[k][s], -cost[k][s], -1);
                        if (best == -1 || dist[k] < dist[best]) best = k;
                }
            s = best;
        }

        for (int k = 0; k < N; k++)
                pi[k] = min(pi[k] + dist[k], INF);
        return width[t];
    }

    pair<L, L> GetMaxFlow(int s, int t)
    {
        L totflow = 0, totcost = 0;
        while (L amt = Dijkstra(s, t))
        {
                totflow += amt;
                for (int x = t; x != s; x = dad[x].first)
                {
                        if (dad[x].second == 1)
                        {
                                flow[dad[x].first][x] += amt;
                                totcost += amt * cost[dad[x].first][x];
                        }
                        else
                        {
                                flow[x][dad[x].first] -= amt;
                                totcost -= amt * cost[x][dad[x].first];
                        }
                }
        }
        return make_pair(totflow, totcost);
    }
};
```

4.4 Eulerian Path

```
struct Edge;
typedef list<Edge>::iterator iter;
struct Edge
{
        int next_vertex;
        iter reverse_edge;
    Edge(int next_vertex) :next_vertex(next_vertex) { }
};

const int max_vertices = ;
int num_vertices;
list<Edge> adj[max_vertices]; // adjacency list
vector<int> path;

void find_path(int v)
{
        while(adj[v].size() > 0)
        {
                int vn = adj[v].front().next_vertex;
        adj[vn].erase(adj[v].front().reverse_edge);
        adj[v].pop_front();
        find_path(vn);
    }
    path.push_back(v);
}

void add_edge(int a, int b)
{
        adj[a].push_front(Edge(b));
        iter ita = adj[a].begin();
        adj[b].push_front(Edge(a));
        iter itb = adj[b].begin();
        ita->reverse_edge = itb;
        itb->reverse_edge = ita;
}
```

4.5 Hopcroft-Karp Algorithm

```
#include <vector>

vector<int> g[N];
int r[N], l[N], n, m, e, a, b;

// generate g;

bool dfs(int v)
{
        if(vis[v]) return false;
        vis[v] = true;
        for(int u=0; u<g[v].size(); ++u)
        {
                if(!r[g[v][u]])
                {
                        l[v]=g[v][u];
                        r[g[v][u]]=v;
                        return true;
                }
        }
        for(int u=0; u<g[v].size(); ++u)
        {
                if(dfs(r[g[v][u]]))
                {
                        l[v]=g[v][u];
                        r[g[v][u]]=v;
                        return true;
                }
        }
        return false;
}

void hopcroft_karp()
{
        bool change = true;
        while(change)
        {
                change = false;
                fill(vis, vis+n+1, false);
                for(int i=1; i<=n; ++i)
                        if(!l[i])
                                change|= dfs(i);
        }
}
```

4.6 Lowest Common Ancestor

```
const int max_nodes, log_max_nodes;
```

```
int num_nodes, log_num_nodes, root;
vector<int> children[max_nodes];
// children[i] contains the children of node i int A[max_nodes][log_max_nodes+1];      // A[i][j] is
      the 2^j-th ancestor of node i, or -1 if that ancestor does not exist int L[max_nodes];
                             // L[i] is the distance between node i and the root
// floor of the binary logarithm of n
int lb(unsigned int n)
{
        if(n==0)  return -1;
        int p = 0;
        if (n >= 1<<16) { n >>= 16; p += 16; }
        if (n >= 1<< 8) { n >>=  8; p +=  8; }
        if (n >= 1<< 4) { n >>=  4; p +=  4; }
        if (n >= 1<< 2) { n >>=  2; p +=  2; }
        if (n >= 1<< 1) {           p +=  1; }
        return p;
}

void DFS(int i, int l)
{
        L[i] = l;
        for(int j = 0; j < children[i].size(); j++)
                DFS(children[i][j], l+1);
}

int LCA(int p, int q) {
        // ensure node p is at least as deep as node q
        if(L[p] < L[q])  swap(p, q);

        // "binary search" for the ancestor of node p situated on the same level as q
        for(int i = log_num_nodes; i >= 0; i--)
                if(L[p] - (1<<i) >= L[q])
        p = A[p][i];
        if(p == q)   return p;

         // "binary search" for the LCA
        for(int i = log_num_nodes; i >= 0; i--)
        {
                if(A[p][i] != -1 && A[p][i] != A[q][i])
                {
                        p = A[p][i];
                        q = A[q][i];
                }
         }
    return A[p][0];
}

int main(int argc,char* argv[])
{
        // read num_nodes, the total number of nodes
        log_num_nodes=lb(num_nodes);
        for(int i = 0; i < num_nodes; i++)
        {
                int p;
                // read p, the parent of node i or -1 if node i is the root
        A[i][0] = p;
        if(p != -1)  children[p].push_back(i);
        else  root = i;
    }
    // precompute A using dynamic programming
    for(int j = 1; j <= log_num_nodes; j++)
        for(int i = 0; i < num_nodes; i++)
                if(A[i][j-1] != -1)  A[i][j] = A[A[i][j-1]][j-1];
                else  A[i][j] = -1;
    // precompute L
        DFS(root, 0);
        return 0;
  }
```

4.7 Strongly Connected Components

```
#include <memory.h>

struct edge
{
        int e, nxt;
};

int V, E;
edge e[MAXE], er[MAXE];
int sp[MAXV], spr[MAXV];
int group_cnt, group_num[MAXV];
bool v[MAXV];
int stk[MAXV];

void fill_forward(int x)
{
        int i;
        v[x]=true;
        for(i=sp[x];i;i=e[i].nxt)
                if(!v[e[i].e]) fill_forward(e[i].e);
        stk[++stk[0]]=x;
}

void fill_backward(int x)
{
        int i;
        v[x]=false;
        group_num[x]=group_cnt;
        for(i=spr[x];i;i=er[i].nxt)
                if(v[er[i].e])
                        fill_backward(er[i].e);
}

void add_edge(int v1, int v2) //add edge v1->v2
{
        e[++E].e=v2;
        e[E].nxt=sp [v1];
        sp[v1]=E;
        er[E].e=v1;
        er[E].nxt=spr[v2];
        spr[v2]=E;
}

void SCC()
{
        int i;
        stk[0]=0;
        memset(v, false, sizeof(v));
        for(i=1;i<=V;i++)
                if(!v[i])
                        fill_forward(i);
        group_cnt=0;
        for(i=stk[0];i>=1;i--)
                if(v[stk[i]]){group_cnt++; fill_backward(stk[i]);}
}
```

4.8 Union-Find Set

```
struct Edge // MST
{
        int a,b,d;
        bool operator < (const Edge &E) const{
        return this->d <E.d;
        }
};
int ranks[M]; int c[N];

int Find(int x)
{
        int y=x;
        while(y!=c[y])
                y=c[y];
        while(x!=c[x])
        {
                int aux=c[x];
                c[x]=y;
                x=aux;
        }
        return y;
}

void Union(int x,int y)
{
        if(ranks[x]>ranks[y])
                c[x]=y;
        else
                c[y]=x;
        if(ranks[x]==ranks[y])
                ranks[y]++;
}
```

5 Tree

5.1 Cartesian Tree

```
struct Tr
{
	Tr *l,*r;
	int key,pr,cnt,val,rev;
	long long sum;
	Tr(int new_key,int new_pr,int new_val)
	{
		rev=0;
		key=new_key;
		cnt=1;
		l=r=NULL;
		pr=new_pr;
		val=new_val;
		sum=new_val;
	}
};


#define T Tr*
T R=NULL;

int cnt(T t)
{
	if(!t) return 0;
	return t->cnt;
}

void upd_cnt(T &t)
{
	if(t) t->cnt=cnt(t->l)+cnt(t->r)+1;
}
long long sum(T t)
{
	if(!t) return 0;
	return t->sum;
}
void upd_sum(T &t)
{
    if(t) t->sum=sum(t->l)+sum(t->r)+t->val;
}
void push(T &t)
{
	if(t && t->rev)
	{
		t->rev=0;
		swap(t->l,t->r);
		upd_sum(t);
		if(t->l) t->l->rev^=1;
		if(t->r) t->r->rev^=1;
	}
}


void split(T t,T &l,T &r,int key,int add)
{
	if(!t)
		return void(l=r=NULL);
	push(t);
	upd_cnt(t);
	int current_key=add+cnt(t->l)+1;
	if(key<=current_key)
		split(t->l,l,t->l,key,add),r=t;
    else
		split(t->r,t->r,r,key,current_key),l=t;
	upd_cnt(t);
	upd_sum(t);
}


void merge(T &t,T l,T r)
{
	push(l);
	push(r);
	if(!l || !r)
		t=l?l:r;
	else if(l->pr>r->pr)
		merge(l->r,l->r,r), t=l;
	else
		merge(r->l,l,r->l), t=r;
	upd_cnt(t);
	upd_sum(t);
}


void insert(T &t,T it,int add)
{
	push(t);
	if(!t)
	{
		t=it;
		upd_cnt(t);
		return;
	}
	upd_sum(t);
	if(it->pr > t->pr)
		split(t,it->l,it->r,it->key,add),t=it;
	else if(it->key>add+cnt(t->l)+1)
		insert(t->r,it,add+cnt(t->l)+1);
	else
		insert(t->l,it,add);
	upd_sum(t);
	upd_cnt(t);
}

void print(T t)
{
	if(!t) return;
	print(t->l);
	cout<<t->val<<" ";
	print(t->r);
}

void reverse(int left,int right)
{
	Tr *t1,*t2,*t3;
	t1=t2=t3=NULL;
	split(R,t1,t2,left,0);
	split(t2,t2,t3,right-left+2,0);
	t2->rev^=1;
	merge(R,t1,t2);
	merge(R,R,t3);
}

void get_sum(int left,int right)
{
	Tr *t1,*t2,*t3;
	t1=t2=t3=NULL;
	split(R,t1,t2,left,0);
	split(t2,t2,t3,right-left+2,0);
	cout<<t2->sum<<"\n";
	merge(R,t1,t2);
	merge(R,R,t3);
}


int n,m, q,a,b;
void example()
{
	ios_base::sync_with_stdio(0);
	cin.tie(0);
	freopen("reverse.in","r",stdin);
	freopen("reverse.out","w",stdout);
	srand(time(0));
	cin>>n>>m;
	for(int i=1;i<=n;++i)
	{
		cin>>a;
		T it=new Tr(i,rand()+1,a);
		insert(R,it,0);
	}
	for(int i=0;i<m;++i)
	{
		cin>>q>>a>>b;
		if(q) reverse(a,b);
		else  get_sum(a,b);
	}
	return 0;
}
```

5.2 Segment Tree

```
//Segment Tree
#include <iostream>
#define N (1<<18)

using namespace std;
//int find(vector <int>& C, int x){return (C[x]==x) ? x : C[x]=find(C, C[x]);} C++
//int find(int x){return (C[x]==x)?x:C[x]=find(C[x]);}
typedef pair<int,int> ii;
ii arb[N]={{0,0}};
int n,m,a,b,v;
char type;

void update(int node,int l,int r,int a,int b,int val,int p)
{
	if(a<=l && r<=b)
	{
		arb[node].first=val;
```

```
                        arb[node].second=p;
                }
                else
                {
                        int mid=(l+r)/2;
                        if(a<=mid)
                                update(node*2,l,mid,a,b,val,p);
                        if(b>mid)
                                update(2*node+1,mid+1,r,a,b,val,p);
                }
        }

        pair<int,int> search(int node,int l,int r,int a)
        {
                if(a==l && a==r)
                {
                        return arb[node];
                }
                else
                {
                        int mid=(l+r)/2;
                        ii cur;
                        if(a<=mid)
                                cur=search(2*node,l,mid,a);
                        else
                                cur=search(2*node+1,mid+1,r,a);
                        if(cur.second<arb[node].second)
                                return arb[node];
                        else
                                return cur;
                }
        }
```

6 Python Graphs/Trees

6.1 Graph structure example for our DFS and BFS algorithms

```
graph = {'A': set(['B', 'C']),
         'B': set(['A', 'D', 'E']),
         'C': set(['A', 'F']),
         'D': set(['B']),
         'E': set(['B', 'F']),
         'F': set(['C', 'E'])}
```

6.2 Breadth-First Search

```
def bfs(graph, start):
    visited, queue = set(), [start]
    while queue:
        vertex = queue.pop(0)
        if vertex not in visited:
            visited.add(vertex)
            queue.extend(graph[vertex] - visited)
    return visited

bfs(graph, 'A') # {'B', 'C', 'A', 'F', 'D', 'E'}
```

6.3 Breadth-First Search Paths

```
def bfs_paths(graph, start, goal):
    queue = [(start, [start])]
    while queue:
        (vertex, path) = queue.pop(0)
        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]
            else:
                queue.append((next, path + [next]))

list(bfs_paths(graph, 'A', 'F')) # [['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
```

6.4 Breadth-First Search Shortest Path

```
def shortest_path(graph, start, goal):
    try:
        return next(bfs_paths(graph, start, goal))
    except StopIteration:
        return None

shortest_path(graph, 'A', 'F') # ['A', 'C', 'F']
```

6.5 Depth-First Search

```
def dfs(graph, start):
    visited, stack = set(), [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            stack.extend(graph[vertex] - visited)
    return visited

dfs(graph, 'A') # {'E', 'D', 'F', 'A', 'C', 'B'}
```

6.6 Depth-First Search Paths

```
#Returns all paths from start to goal
def dfs_paths(graph, start, goal):
    stack = [(start, [start])]
    while stack:
        (vertex, path) = stack.pop()
        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]
            else:
                stack.append((next, path + [next]))

list(dfs_paths(graph, 'A', 'F')) # [['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
```

6.7 Dijkstra's Algorithm

```
from collections import defaultdict
from heapq import *

def dijkstra(edges, f, t):
   g = defaultdict(list)
   for l,r,c in edges:
       g[l].append((c,r))

   q, seen = [(0,f,())], set()
   while q:
        (cost,v1,path) = heappop(q)
        if v1 not in seen:
            seen.add(v1)
            path = (v1, path)
            if v1 == t: return (cost, path)
            for c, v2 in g.get(v1, ()):
                if v2 not in seen:
                    heappush(q, (cost+c, v2, path))
   return float("inf")

#Code example
edges = [("A", "B", 7), ("A", "D", 5), ("B", "C", 8),
         ("B", "D", 9), ("B", "E", 7), ("C", "E", 5)]
print "A -> E:"
print dijkstra(edges, "A", "E")   #(14, ('E', ('B', ('A', ()))))
```

6.8 Kruskal's Algorithm (including Merge-Find set)

```python
parent = dict()
rank = dict()

def make_set(vertice):
    parent[vertice] = vertice
    rank[vertice] = 0

def find(vertice):
    if parent[vertice] != vertice:
        parent[vertice] = find(parent[vertice])
    return parent[vertice]

def union(vertice1, vertice2):
    root1 = find(vertice1)
    root2 = find(vertice2)
    if root1 != root2:
        if rank[root1] > rank[root2]:
            parent[root2] = root1
        else:
            parent[root1] = root2
        if rank[root1] == rank[root2]: rank[root2] += 1

def kruskal(graph):
    for vertice in graph['vertices']:
        make_set(vertice)
        minimum_spanning_tree = set()
        edges = list(graph['edges'])
        edges.sort()
        #print edges
    for edge in edges:
        weight, vertice1, vertice2 = edge
        if find(vertice1) != find(vertice2):
            union(vertice1, vertice2)
            minimum_spanning_tree.add(edge)

    return sorted(minimum_spanning_tree)
```

6.9 Bellman-Ford Algorithm

```python
# Step 1: For each node prepare the destination and predecessor
def initialize(graph, source):
    d = {} # Stands for destination
    p = {} # Stands for predecessor
    for node in graph:
        d[node] = float('Inf') # We start admiting that the rest of nodes are very very far
        p[node] = None
    d[source] = 0 # For the source we know how to reach
    return d, p

def relax(node, neighbour, graph, d, p):
    # If the distance between the node and the neighbour is lower than the one I have now
    if d[neighbour] > d[node] + graph[node][neighbour]:
        # Record this lower distance
        d[neighbour]  = d[node] + graph[node][neighbour]
        p[neighbour] = node

def bellman_ford(graph, source):
    d, p = initialize(graph, source)
    for i in range(len(graph)-1): #Run this until is converges
        for u in graph:
            for v in graph[u]: #For each neighbour of u
                relax(u, v, graph, d, p) #Lets relax it

    # Step 3: check for negative-weight cycles
    for u in graph:
        for v in graph[u]:
            assert d[v] <= d[u] + graph[u][v]

    return d, p

def test():
    graph = {
        'a': {'b': -1, 'c':  4},
        'b': {'c':  3, 'd':  2, 'e':  2},
        'c': {},
        'd': {'b':  1, 'c':  5},
        'e': {'d': -3}
        }
    d, p = bellman_ford(graph, 'a')
        # d = {'a':0, 'b':-1, 'c':2, 'd':-2, 'e':1},
        # p = {'a':None, 'b':'a', 'c':'b', 'd':'e', 'e':'b'}
```

6.10 Floyd-Warshall Algorithm

```python
# Number of vertices in the graph
V = 4

# Define infinity as the large enough value. This value will be
# used for vertices not connected to each other
INF  = 99999

# Solves all pair shortest path via Floyd Warshall Algorithm
def floydWarshall(graph):

    """ dist[][] will be the output matrix that will finally
        have the shortest distances between every pair of vertices """
    """ initializing the solution matrix same as input graph matrix
    OR we can say that the initial values of shortest distances
    are based on shortest paths considerting no
    intermediate vertices """
    dist = map(lambda i : map(lambda j : j , i) , graph)

    """ Add all vertices one by one to the set of intermediate
     vertices.
     ---> Before start of a iteration, we have shortest distances
     between all pairs of vertices such that the shortest
     distances consider only the vertices in set
    {0, 1, 2, .. k-1} as intermediate vertices.
      ----> After the end of a iteration, vertex no. k is
     added to the set of intermediate vertices and the
    set becomes {0, 1, 2, .. k}
    """
    for k in range(V):

        # pick all vertices as source one by one
        for i in range(V):

            # Pick all vertices as destination for the
            # above picked source
            for j in range(V):

                # If vertex k is on the shortest path from
                # i to j, then update the value of dist[i][j]
                dist[i][j] = min(dist[i][j] ,
                                  dist[i][k]+ dist[k][j]
                                )
    printSolution(dist)

"""
            10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)------->(2)
            3           """
graph = [[0,5,INF,10],
             [INF,0,3,INF],
             [INF, INF, 0,   1],
             [INF, INF, INF, 0]
        ]

floydWarshall(graph) # [[0,5,8,9],[INF,0,3,4],[INF,INF,0,1],[INF,INF,INF,0]]
```

6.11 Max Flow (Ford-Fulkerson Algorithm)

```python
from collections import defaultdict

#This class represents a directed graph using adjacency matrix representation
class Graph:

    def __init__(self,graph):
        self.graph = graph # residual graph
        self.ROW = len(graph)
        #self.COL = len(gr[0])


    '''Returns true if there is a path from source 's' to sink 't' in
    residual graph. Also fills parent[] to store the path '''
    def BFS(self,s, t, parent):

        # Mark all the vertices as not visited
        visited =[False]*(self.ROW)

        # Create a queue for BFS
        queue=[]

        # Mark the source node as visited and enqueue it
        queue.append(s)
        visited[s] = True
```

```
        # Standard BFS Loop
        while queue:

            #Dequeue a vertex from queue and print it
            u = queue.pop(0)

            # Get all adjacent vertices of the dequeued vertex u
            # If a adjacent has not been visited, then mark it
            # visited and enqueue it
            for ind, val in enumerate(self.graph[u]):
                if visited[ind] == False and val > 0 :
                    queue.append(ind)
                    visited[ind] = True
                    parent[ind] = u

        # If we reached sink in BFS starting from source, then return
        # true, else false
        return True if visited[t] else False


    # Returns tne maximum flow from s to t in the given graph
    def FordFulkerson(self, source, sink):

        # This array is filled by BFS and to store path
        parent = [-1]*(self.ROW)

        max_flow = 0 # There is no flow initially

        # Augment the flow while there is path from source to sink
        while self.BFS(source, sink, 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.
            path_flow = float("Inf")
            s = sink
            while(s !=  source):
                path_flow = min (path_flow, self.graph[parent[s]][s])
                s = parent[s]

            # Add path flow to overall flow
            max_flow +=  path_flow

            # update residual capacities of the edges and reverse edges
            # along the path
            v = sink
            while(v !=  source):
                u = parent[v]
                self.graph[u][v] -= path_flow
                self.graph[v][u] += path_flow
                v = parent[v]

        return max_flow


# Create a graph given in the above diagram

graph = [[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]]

g = Graph(graph)

source = 0; sink = 5

print ("The maximum possible flow is %d " % g.FordFulkerson(source, sink))
```

6.12 Segment Tree

```
#encoding:utf-8
class SegmentTree(object):
    def __init__(self, start, end):
        self.start = start
        self.end = end
        self.max_value = {}
        self.sum_value = {}
        self.len_value = {}
        self._init(start, end)

    def add(self, start, end, weight=1):
        start = max(start, self.start)
        end = min(end, self.end)
        self._add(start, end, weight, self.start, self.end)
        return True

    def query_max(self, start, end):
        return self._query_max(start, end, self.start, self.end)

    def query_sum(self, start, end):
        return self._query_sum(start, end, self.start, self.end)

    def query_len(self, start, end):
        return self._query_len(start, end, self.start, self.end)

    """"""
    def _init(self, start, end):
        self.max_value[(start, end)] = 0
        self.sum_value[(start, end)] = 0
        self.len_value[(start, end)] = 0
        if start < end:
            mid = start + int((end - start) / 2)
            self._init(start, mid)
            self._init(mid+1, end)

    def _add(self, start, end, weight, in_start, in_end):
        key = (in_start, in_end)
        if in_start == in_end:
            self.max_value[key] += weight
            self.sum_value[key] += weight
            self.len_value[key] = 1 if self.sum_value[key] > 0 else 0
            return

        mid = in_start + int((in_end - in_start) / 2)
        if mid >= end:
            self._add(start, end, weight, in_start, mid)
        elif mid+1 <= start:
            self._add(start, end, weight, mid+1, in_end)
        else:
            self._add(start, mid, weight, in_start, mid)
            self._add(mid+1, end, weight, mid+1, in_end)
        self.max_value[key] = max(self.max_value[(in_start, mid)], self.max_value[(mid+1, in_end)])
        self.sum_value[key] = self.sum_value[(in_start, mid)] + self.sum_value[(mid+1, in_end)]
        self.len_value[key] = self.len_value[(in_start, mid)] + self.len_value[(mid+1, in_end)]

    def _query_max(self, start, end, in_start, in_end):
        if start == in_start and end == in_end:
            ans = self.max_value[(start, end)]
        else:
            mid = in_start + int((in_end - in_start) / 2)
            if mid >= end:
                ans = self._query_max(start, end, in_start, mid)
            elif mid+1 <= start:
                ans = self._query_max(start, end, mid+1, in_end)
            else:
                ans = max(self._query_max(start, mid, in_start, mid),
                        self._query_max(mid+1, end, mid+1, in_end))
        #print start, end, in_start, in_end, ans
        return ans

    def _query_sum(self, start, end, in_start, in_end):
        if start == in_start and end == in_end:
            ans = self.sum_value[(start, end)]
        else:
            mid = in_start + int((in_end - in_start) / 2)
            if mid >= end:
                ans = self._query_sum(start, end, in_start, mid)
            elif mid+1 <= start:
                ans = self._query_sum(start, end, mid+1, in_end)
            else:
                ans = self._query_sum(start, mid, in_start, mid) + self._query_sum(mid+1, end, mid+1,
                      in_end)
        return ans

    def _query_len(self, start, end, in_start, in_end):
        if start == in_start and end == in_end:
            ans = self.len_value[(start, end)]
        else:
            mid = in_start + int((in_end - in_start) / 2)
            if mid >= end:
                ans = self._query_len(start, end, in_start, mid)
            elif mid+1 <= start:
                ans = self._query_len(start, end, mid+1, in_end)
            else:
                ans = self._query_len(start, mid, in_start, mid) + self._query_len(mid+1, end, mid+1,
                      in_end)

        #print start, end, in_start, in_end, ans
        return ans
```

7 Math with Numbers

7.1 Extended Euclid's Algorithm

```
#include "GcdLcm.h"

// returns d = gcd(a,b); find x, y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y){
        int current_x = y = 0;
        int current_y = x = 1;
        while(b)
        {
                int q = a/b;
                int t = b;
                b = a%b;
                a = t;
                t = current_x; current_x = x-q*current_x; x = t;
                t = current_y; current_y = y-q*current_y; y = t;
        }
        return a;
}
```

7.2 Fast Prime Number Sieve

```
// compute prime numbers to N really fast (Sieve)
#define N 10000100
vector<int> primes;
char p[N/8];

void GeneratePrimes()
{
        primes.push_back(2);
        int i,j;
        for(i=1;((i*i)<<1)+(i<<1)<N;++i)
        {
                if((p[i>>3]&(1<<(i&7)))==0)
                {
                        primes.push_back((i<<1)+1);
                        for(j=((i*i)<<1)+(i<<1),k=(i<<1)+1;(j<<1)+1<N;j+=k)
                        {
                                p[j>>3] |=(1<<(j&7));
                        }
                }
        }

        for(;(i<<1)+1<N;++i)
        {
                if((p[i>>3]&(1<<(i&7)))==0)
                {
                        primes.push_back((i<<1)+1);
                }
        }
}
```

7.3 Prime Number Sieve (generator)

```
from itertools import count

def postponed_sieve():                  # postponed sieve, by Will Ness
    yield 2; yield 3; yield 5; yield 7;  # original code David Eppstein,
    sieve = {}                           #   Alex Martelli, ActiveState Recipe 2002
    ps = postponed_sieve()               # a separate base Primes Supply:
    p = next(ps) and next(ps)            # (3) a Prime to add to dict
    q = p*p                              # (9) its sQuare
    for c in count(9,2):                 # the Candidate
        if c in sieve:               # c's a multiple of some base prime
            s = sieve.pop(c)         #     i.e. a composite ; or
        elif c < q:
             yield c                 # a prime
             continue
        else:   # (c==q):            # or the next base prime's square:
            s=count(q+2*p,2*p)       #    (9+6, by 6 : 15,21,27,33,...)
            p=next(ps)               #    (5)
            q=p*p                    #    (25)
        for m in s:                  # the next multiple
            if m not in sieve:       # no duplicates
                break
        sieve[m] = s                 # original test entry: ideone.com/WFv4f
```

7.4 GCD and LCM

```
// return a%b
int mod(int a, int b)
{
        return ((a%b)+b)%b;
}

// computes gcd(a,b)
int gcd(int a, int b)
{
        int tmp;
        while(b)
        {
                a %= b;
                tmp = a;
                a = b;
                b = tmp;
        }
        return a;
}

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

7.5 GCD and Euler's Totient Function

```
# Function to return gcd of a and b
def gcd(a, b):
    if a == 0:
        return b
    return gcd(b%a, a)

# A simple method to evaluate Euler Totient Function
def phi(n):
    result = 1
    for i in range(2, n):
        if gcd(i, n) == 1:
            result = result + 1
    return result
```

7.6 Miller-Rabin Primality Test

```
def miller_rabin(n, k):
    # The optimal number of rounds (k) for this test is 40
    # for justification

    if n == 2:
        return True
    if n % 2 == 0:
        return False
    r, s = 0, n - 1
    while s % 2 == 0:
        r += 1
        s //= 2
    for _ in xrange(k):
        a = random.randrange(2, n - 1)
        x = pow(a, s, n)
        if x == 1 or x == n - 1:
            continue
        for _ in xrange(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True
```

8 Math with Matrices?

8.1 Modular Linear Equation Solver

```cpp
// Stanford Notebook
#include <vector>
#include "ExtendedEuclid.h"

// find all solutions to ax = b (mod n)
vector<int> modular_linear_equation_solver(int a, int b, int n)
{
        int x,y;
        vector<int> sol;
        int d = extended_euclid(a, n, x, y);
        if(!(b%d)) {
                x = mod(x%(b/d), n);
                for(int i=0; i < d; ++i)
                        sol.push_back(mod(x + i*(n/d),n));
        }
        return sol;
}

// computes b such tath ab = 1(mod n), returs -1 on failure
int mod_inverse(int a, int n)
{
        int x, y;
        int d = extended_euclid(a, n, x, y);
        if(d>1) return -1;
        return mod(x,n);
}

// computes x and y such that ax + by =c; on failure, x = y =-1
void linear_diophantine(int a,int b, int c,int &x, int &y)
{
        int d = gcd(a,b);
        if(c%d) x = y = -1;
        else
        {
                x = c/d * mod_inverse(a/d,b/d);
                y = (c-a*x)/b;
        }
}
```

8.2 Fast Fourier Transform

```cpp
struct cpx
{
        cpx(){}
        cpx(double aa): a(aa){}
        cpx(double aa, double bb): a(aa), b(bb){}
        double a;
        double b;
        double modsq(void) const{ return a*a+b*b; }
        cpx bar(void) const {return cpx(a,-b);}
};

cpx b[N+100],c[N+100],B[N+100],C[N+100];
int a[N+100], int x[N+100];
double coss[N+100], sins[N+100];
int n,m,p;

cpx operator +(cpx a,cpx b) {    return cpx(a.a+b.a,a.b+b.b); }
cpx operator *(cpx a,cpx b) {    return cpx(a.a*b.a-a.b*b.b,a.a*b.b+a.b*b.a); }
cpx operator /(cpx a,cpx b) {    cpx r = a*b.bar();    return cpx(r.a/b.modsq(),r.b/b.modsq()); }
cpx EXP(int i,int dir) {    return cpx(coss[i],sins[i]*dir); }

const double two_pi = 4 * acos(0);

void FFT(cpx *in,cpx *out,int step,int size,int dir)
{
        if(size<1) return;
        if(size==1)
        {
                out[0]=in[0];
                return;
        }
        FFT(in,out,step*2,size/2,dir);
        FFT(in+step,out+size/2,step*2,size/2,dir);
        for(int i=0;i<size/2;++i)
        {
                cpx even=out[i];
                cpx odd=out[i+size/2];
                out[i] = even+EXP(i*step,dir)*odd;
                out[i+size/2]=even+EXP((i+size/2)*step,dir)*odd;
        }
}



void exemple
{
        for(int i=0;i<=N;++i)
        {
                coss[i]=cos(two_pi*i/N);
                sins[i]=sin(two_pi*i/N);
        }
        while(scanf("%d",&n)==1)
        {
                fill(x,x+N+100,0);
                fill(a,a+N+100,0);
                for(int i=0;i<n;++i)
                {
                        scanf("%d",&p);
                        x[p]=1;
                }
                for(int i=0;i<N+100;++i)
                {
                        b[i]=cpx(x[i],0);
                }
                scanf("%d",&m);
                for(int i=0;i<m;++i)
                {
                        scanf("%d",&a[i]);
                }
                FFT(b,B,1,N,1);
                for(int i=0;i<N;++i)
                        C[i]=B[i]*B[i];
                FFT(C,c,1,N,-1);
                for(int i=0;i<N;++i)
                        c[i]=c[i]/N;
                int cnt=0;
                for(int i=0;i<16;++i)
                        cout<<c[i].a<<" ";
                for(int i=0;i<m;++i)
                        if(c[a[i]].a>0.5 || x[a[i]])
                                cnt++;
                printf("%d\n",cnt);
        }
}
```

8.3 Gauss-Jordan Elimination (Matrix inversion and linear system solving)

```python
def gauss_jordan(m, eps = 1.0/(10**10)):
  """Puts given matrix (2D array) into the Reduced Row Echelon Form.
     Returns True if successful, False if 'm' is singular.
     NOTE: make sure all the matrix items support fractions! Int matrix will NOT work!
     Written by Jarno Elonen in April 2005, released into Public Domain"""
  (h, w) = (len(m), len(m[0]))
  for y in range(0,h):
    maxrow = y
    for y2 in range(y+1, h):    # Find max pivot
      if abs(m[y2][y]) > abs(m[maxrow][y]):
        maxrow = y2
    (m[y], m[maxrow]) = (m[maxrow], m[y])
    if abs(m[y][y]) <= eps:     # Singular?
      return False
    for y2 in range(y+1, h):    # Eliminate column y
      c = m[y2][y] / m[y][y]
      for x in range(y, w):
        m[y2][x] -= m[y][x] * c
  for y in range(h-1, 0-1, -1): # Backsubstitute
    c  = m[y][y]
    for y2 in range(0,y):
      for x in range(w-1, y-1, -1):
        m[y2][x] -=  m[y][x] * m[y2][y] / c
    m[y][y] /= c
    for x in range(h, w):       # Normalize row y
      m[y][x] /= c
  return True

def solve(M, b):
  """
  solves M*x = b
  return vector x so that M*x = b
  :param M: a matrix in the form of a list of list
  :param b: a vector in the form of a simple list of scalars
  """
  m2 = [row[:]+[right] for row,right in zip(M,b) ]
  return [row[-1] for row in m2] if gauss_jordan(m2) else None

def inv(M):
  """
  return the inv of the matrix M
  """
  #clone the matrix and append the identity matrix
  # [int(i==j) for j in range_M] is nothing but the i(th row of the identity matrix
  m2 = [row[:]+[int(i==j) for j in range(len(M) )] for i,row in enumerate(M) ]
```

```
    # extract the appended matrix (kind of m2[m:,...]
    return [row[len(M[0]):] for row in m2] if gauss_jordan(m2) else None

def zeros( s , zero=0):
    """
    return a matrix of size `size`
    :param size: a tuple containing dimensions of the matrix
    :param zero: the value to use to fill the matrix (by default it's zero )
    """
    return [zeros(s[1:] ) for i in range(s[0] ) ] if not len(s) else zero
```

8.4 Gauss-Jordan Elimination

```
/*
Stanford notebook
-------------------------------------------------------------
GaussJordan Algorithm (elimination with full pivoting)
Uses:
(1) solving systems of linear equations (AX=B)
(2) inverting matrices (AX=I)
(3) computing determinants of square matrices
Running time: O(n^3)
INPUT:    a[][] = an nxn matrix
          b[][] = an nxm matrix
OUTPUT:   X      = an nxm matrix (stored in b[][])
              A^{-1} = an nxn matrix (stored in a[][])
              returns determinant of a[][]
-------------------------------------------------------------
*/

const double EPS = 1e-10;
typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

T GaussJordan(VVT &a, VVT &b)
{
        const int n = a.size();
        const int m = b[0].size();
        VI irow(n), icol(n), ipiv(n);
        T det = 1;

    for (int i = 0; i < n; i++)
    {
        int pj = -1, pk = -1;
                for (int j = 0; j < n; j++)
                        if (!ipiv[j])
                                for (int k = 0; k < n; k++)
                                        if (!ipiv[k])
                                                if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj
                                                      = j; pk = k; }
                if (fabs(a[pj][pk]) < EPS)
                {
                        cerr << "Matrix is singular." << endl;
                        exit(0);
                }
        ipiv[pk]++;
        swap(a[pj], a[pk]);
        swap(b[pj], b[pk]);
        if (pj != pk) det *= -1;
        irow[i] = pj;
        icol[i] = pk;

        T c = 1.0 / a[pk][pk];
        det *= a[pk][pk];
        a[pk][pk] = 1.0;

        for (int p = 0; p < n; p++)
                a[pk][p] *= c;
        for (int p = 0; p < m; p++)
                b[pk][p] *= c;
        for (int p = 0; p < n; p++)
        {
                if (p != pk)
                {
                        c = a[p][pk];
                        a[p][pk] = 0;
                        for (int q = 0; q < n; q++)
                                a[p][q] -= a[pk][q] * c;
                        for (int q = 0; q < m; q++)
                                b[p][q] -= b[pk][q] * c;
                }
        }
    }
    for (int p = n-1; p >= 0; p--)
        {
                if (irow[p] != icol[p])
                {
                        for (int k = 0; k < n; k++)
                                swap(a[k][irow[p]], a[k][icol[p]]);
                }
        }
        return det;
}


int main()
{
        const int n = 4;
        const int m = 2;
        double A[n][n] = { {1,2,3,4},{1,0,1,0},{5,3,2,4},{6,1,4,6} };
        double B[n][m] = { {1,2},{4,3},{5,6},{8,7} };
        VVT a(n), b(n);
        for (int i = 0; i < n; i++)
        {
                a[i] = VT(A[i], A[i] + n);
                b[i] = VT(B[i], B[i] + m);
        }
        double det = GaussJordan(a, b);
        // expected: 60
        cout << "Determinant: " << det << endl;
        // expected: -0.233333 0.166667 0.133333 0.0666667
        //           0.166667 0.166667 0.333333 -0.333333
        //           0.233333 0.833333 -0.133333 -0.0666667
        //           0.05 -0.75 -0.1 0.2
        cout << "Inverse: " << endl;
        for (int i = 0; i < n; i++)
        {
                for (int j = 0; j < n; j++)
                        cout << a[i][j] << ' ';
                cout << endl;
        }
        // expected: 1.63333 1.3
        //           -0.166667 0.5
        //           2.36667 1.7
        //           -1.85 -1.35
        cout << "Solution: " << endl;
        for (int i = 0; i < n; i++)
        {
                for (int j = 0; j < m; j++)
                        cout << b[i][j] << ' ';
                cout << endl;
        }
}
```

9 Strings

9.1 Aho-Corasick Algorithm

```
/*
Implementation - Benoit Chabod
------------------------------
Aho Corasick algorithm
------------------------------
*/

struct node
{
        int f;
        map<char, int> g;
        vector<short> out;
        node(int fail = -1): f(fail) {}
};

vector<node> nodes;

void add_str(const string & s, int num)
{
        int cur = 0;
        int n = s.size();
        for(int i = 0; i < n; i++)
        {
                auto it = nodes[cur].g.find(s[i]);
                if(it == nodes[cur].g.end())
                {
                        nodes[cur].g[s[i]] = nodes.size();
                        cur = nodes.size();
                        nodes.push_back(node());
                }
                else
                {
```

```
                        cur = it->second;
                }
        }
        nodes[cur].out.push_back(num);
}

void init_fail()
{
        int cur = 0;
        queue<int> q;
        q.push(cur);

    while( !q.empty() )
    {
        cur = q.front();
        map<char, int>::iterator it;
        for(it = nodes[cur].g.begin(); it != nodes[cur].g.end(); it++)
        {
                int child = it->second;
                int pfail = nodes[cur].f;
                char ch = it->first;
                map<char, int>::iterator f;
                while( pfail != -1 && ((f = nodes[pfail].g.find(ch)) == nodes[pfail].g.end()) )
                {
                        pfail = nodes[pfail].f;
                }
                nodes[child].f = (pfail == -1)? 0 : f->second;
                pfail = nodes[child].f;
                nodes[child].out.insert(nodes[child].out.end(),nodes[pfail].out.begin(),nodes[pfail].
                      out.end());
                q.push(child);
        }
        q.pop();
    }
}

// Usage
void usage()
{
        nodes.push_back(node());
        for [each word] add_str(word,i)
                init_fail();
        for [each letter]
        {
                map<char, int>::iterator f;
                while( cur != -1 && ((f = nodes[cur].g.find(letter)) == nodes[cur].g.end()) )
                        cur = nodes[cur].f;
                if( cur == -1 )
                {
                        cur = 0;
                        continue;
        }
        cur = f->second;
        for(auto v : nodes[cur].out)
        {
                // Word v was found
        }
    }
}
```

9.2 Knuth-Morris-Pratt Algorithm (fast pattern matching)

```
def KnuthMorrisPratt(text, pattern):

    '''Yields all starting positions of copies of the pattern in the text.
Calling conventions are similar to string.find, but its arguments can be
lists or iterators, not just strings, it returns all matches, not just
the first one, and it does not need the whole text in memory at once.
Whenever it yields, it will have read the text exactly up to and including
the match that caused the yield.'''

    # allow indexing into pattern and protect against change during yield
    pattern = list(pattern)

    # build table of shift amounts
    shifts = [1] * (len(pattern) + 1)
    shift = 1
    for pos in range(len(pattern)):
        while shift <= pos and pattern[pos] != pattern[pos-shift]:
            shift += shifts[pos-shift]
        shifts[pos+1] = shift

    # do the actual search
    startPos = 0
    matchLen = 0
    for c in text:
        while matchLen == len(pattern) or \
              matchLen >= 0 and pattern[matchLen] != c:
            startPos += shifts[matchLen]
            matchLen -= shifts[matchLen]
        matchLen += 1
        if matchLen == len(pattern):
            yield startPos
```

9.3 Knuth-Morris-Pratt Algorithm

```
/*
---------------
KMP/Pi function
Note : cin>>(s+1) (the operations in the pi-function start at 1)
---------------
*/
void preKmp()
{
        int k;
        k=kmpNext[1]=0;
        for(int i=2;i<=n;++i)
        {
                while(k && p[k+1]!=p[i]) k=kmpNext[k];
                if(p[k+1]==p[i])
                        k++;
                kmpNext[i]=k;
        }
}

void KMP()
{
        preKmp();
        int k=0;

        for(int i=1;i<=m;++i)
        {
                while(k && p[k+1]!=s[i])
                        k=kmpNext[k];
                if(p[k+1]==s[i])
                        k++;
                if(k==n)
                {
                        // here we have a match
                        k=kmpNext[k];
                }
        }
}
```

9.4 Rabin-Karp Algorithm (multiple pattern matching)

```
# d is the number of characters in input alphabet
d = 256

# pat  -> pattern
# txt  -> text
# q    -> A prime number

def search(pat, txt, q):
    M = len(pat)
    N = len(txt)
    i = 0
    j = 0
    p = 0    # hash value for pattern
    t = 0    # hash value for txt
    h = 1

    # The value of h would be "pow(d, M-1)%q"
    for i in xrange(M-1):
        h = (h*d)%q

    # Calculate the hash value of pattern and first window
    # of text
    for i in xrange(M):
        p = (d*p + ord(pat[i]))%q
        t = (d*t + ord(txt[i]))%q

    # Slide the pattern over text one by one
    for i in xrange(N-M+1):
        # Check the hash values of current window of text and
        # pattern if the hash values match then only check
        # for characters on by one
        if p==t:
            # Check for characters one by one
            for j in xrange(M):
```

```
            if txt[i+j] != pat[j]:
                break

            j+=1
            # if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
            if j==M:
                print "Pattern found at index " + str(i)

        # Calculate hash value for next window of text: Remove
        # leading digit, add trailing digit
        if i < N-M:
            t = (d*(t-ord(txt[i])*h) + ord(txt[i+M]))%q

            # We might get negative values of t, converting it to
            # positive
            if t < 0:
                t = t+q

# Driver program to test the above function
txt = "GEEKS FOR GEEKS"
pat = "GEEK"
q = 101 # A prime number
search(pat,txt,q)
```

9.5 Suffix Array

```
//Suffix Array

struct SuffixArray
{
        const int L;
        string s;
        vector<vector<int> > P;
        vector<pair<pair<int,int>,int> > M;
        SuffixArray(const string &s) : L(s.length()), s(s), P(1, vector<int>(L, 0)), M(L)
        {
                for (int i = 0; i < L; i++)
                        P[0][i] = int(s[i]);
                for (int skip = 1, level = 1; skip < L; skip *= 2, level++)
                {
                        P.push_back(vector<int>(L, 0));
                        for (int i = 0; i < L; i++)
                                M[i] = make_pair(make_pair(P[level-1][i], i + skip < L ? P[level-1][i
                                        + skip] : -1000), i);
                        sort(M.begin(), M.end());
                        for (int i = 0; i < L; i++)
                                P[level][M[i].second] = (i > 0 && M[i].first == M[i-1].first) ? P[
                                        level] [M[i-1].second] : i;
                }
        }

    vector<int> GetSuffixArray()
    {
        return P.back();
        }

        // returns the length of the longest common prefix of s[i...L-1] and s[j...L-1]
        int LongestCommonPrefix(int i, int j)
        {
                int len = 0;
                if (i == j) return L - i;
                for (int k = P.size() - 1; k >= 0 && i < L && j < L; k--)
                {
                        if (P[k][i] == P[k][j])
                        {
                            i += 1 << k;
                            j += 1 << k;
                            len += 1 << k;
                        }
                }
                return len;
        }
};


int main()
{
        // bobocel is the 0'th suffix
        //  obocel is the 5'th suffix
        //   bocel is the 1'st suffix
        //    ocel is the 6'th suffix
        //     cel is the 2'nd suffix
        //      el is the 3'rd suffix
        //       l is the 4'th suffix
        SuffixArray suffix("bobocel");
        vector<int> v = suffix.GetSuffixArray();
        // Expected output: 0 5 1 6 2 3 4
        //                  2
        for (int i = 0; i < v.size(); i++)
                cout << v[i] << " ";
        cout << endl;
        cout << suffix.LongestCommonPrefix(0, 2) << endl;
}
```