Alex's Anthology of Algorithms:

Common Code for Contests in Concise C++

(Draft, as of March 16th, 2014)

Compiled, Edited and Written by

Alex Li

Woburn Collegiate Institute

Programming Enrichment Group

Email: alextrovert@gmail.com

Contents

Contents

This page is intentionally left blank.

Preface

This document does not intend to explain algorithms. You will only find examples and brief descriptions to help you better understand the given implementations. Thus, you are expected to already have knowledge of algorithmic programming paradigms, as well as the actual algorithms being discussed. In many cases, it is possible to learn an algorithm by examining its implementation. However, memorizing another person's code is generally a less efficient way to learn than researching the idea and trying to implement it by yourself. Treat this as a "cheat sheet", if you will, and use this as a last resort when you simply require a working implementation.

If you seek to become a better contest programmer, I suggest you follow through with the USACO training pages (ace.delos.com/usacogate), read up on informatics books and online algorithm tutorials (e.g. "Introduction to Algorithms" by Cormen et al., "The Algorithm Design Manual" by Skiena, "The Art of Computer Programming" by Knuth, and Topcoder tutorials: www.topcoder.com/tc?d1=tutorials&d2=alg_index&module=Static). Practice on sites such as the Sphere Online Judge (www.spoj.com), Codeforces (www.codeforces.com), the UVa Online Judge (uva.onlinejudge.org), and the PEG judge (www.wcipeg.com). Perhaps along the way, you will infrequently refer here for insight on ways to implement any newly-learned concepts.

C++ is my language of choice because of its predominance in competitions. The International Olympiad in Informatics (and practically every other programming contest) accepts solutions to tasks in C, C++ and Pascal. C++ is a fast, flexible language with a sizable Standard Template Library and support for useful features like built-in memory management and object-oriented programming. In an attempt to focus less on reinventing the wheel and more on the algorithms themselves, the implementations here will often try to take advantage of useful features of the C++ language. The programs are tested with version 4.7.3 of the GNU Compiler Collection (GCC) on a 32-bit system (that means, for instance, `bool` and `char` are assumed to be 8-bits, `int` and `float` are assumed to be 32-bits, `double` and `long long` are assumed to be 64-bits, and `long double` is assumed to be 96-bits).

All the information in descriptions come from Wikipedia and other online sources. Some programs you will find are direct implementations of pseudocode found online, while others are re-adaptated from informatics books and journals. If references for a program are not listed in the endnotes section, you may assume that I have written them from scratch. You are free to use, modify, and distribute these programs for personal or educational purposes provided you have examined their origins , though I strongly recommend making an effort to understand any code you did not write on your own. Once you are well acquainted with the underlying algorithms, you will become better at adapting them for the problem at hand.

This anthology started as a personal project to implement classic algorithms in the most concise and "vanilla" way possible while minimizing code obfuscation. I tried to determine the most appropriate trade-off between clarity, flexibility, efficiency, and code length. Short descriptions of what the programs do along with their time (and space, where necessary) complexities are typically included. For further clarification, diagrams may be given for the examples. If these are insufficient to help you understand, then you should be able to find extra information by referring to the endnotes and looking online. Lastly, I do not guarantee that the programs are bug-free - it is always best to read the endnotes, especially when you are modifying a program. Best of luck!

Alex Li
2013-2015 Student Leader
Woburn Collegiate Institute
Programming Enrichment Group
September, 2013

Section 1 - Graph Theory

Section Notes: Some programs in this section internally operate on 1-based vertices, and others 0-based. However, all of the example inputs are 1-based. You can assume that the algorithm operates on 0-based labelling if nodes are immediately decremented when they're inputted (e.g. "`cin >> a >> b;    a--; b--;`"). It is your job to be cautious with shifting indices, as well as properly initializing arrays when readapting these programs for specific problems.

1.1 - Depth First Search[1]

Description: Given an unweighted graph, traverse all reachable nodes from a source node. Each branch is explored as deep as possible before more branches are visited. DFS only uses as much space as the length of the longest branch. When DFS'ing recursively, the internal call-stack could overflow, so sometimes it is safer to use an explicit stack data structure.

Complexity: O(number of edges) for explicit graphs traversed without repetition. $O(b^d)$ for implicit graphs with a branch factor of b, searched to depth d.[1]

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

const int MAX_N = 101;
int nodes, edges, start, a, b;
bool visit[MAX_N] = {0};
vector<int> adj[MAX_N];

void DFS(int n) {
    visit[n] = true;
    cout << " " << n;
    for (int j = 0; j < adj[n].size(); j++)
        if (!visit[adj[n][j]]) DFS(adj[n][j]);
}

int main() {
    cin >> nodes >> edges >> start;
    for (int i = 0; i < edges; i++) {
        cin >> a >> b;
        adj[a].push_back(b);
    }
    cout << "Nodes visited:\n";  DFS(start);
    return 0;
}
```

1.2 - Breadth First Search[2]

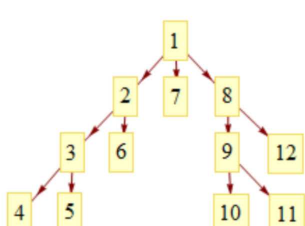
Description: Given an unweighted graph, traverse all reachable nodes from a source node. All nodes of a certain depth are explored before nodes of one depth greater are examined. Therefore, BFS will reach a given destination in the shortest path possible. More auxiliary memory than DFS is required.

Complexity: Same as depth first search.

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

const int MAX_N = 101;
int nodes, edges, start, a, b;
bool visit[MAX_N] = {0};
vector<int> adj[MAX_N];

int main() {
    cin >> nodes >> edges >> start;
    for (int i = 0; i < edges; i++) {
        cin >> a >> b;
        adj[a].push_back(b);
    }
    cout << "Nodes visited:\n";
    queue<int> q;
    for (q.push(start); !q.empty(); q.pop()) {
        int n = q.front();
        visit[n] = true;
        cout << " " << n;
        for (int j = 0; j < adj[n].size(); j++)
           if (!visit[adj[n][j]]) q.push(adj[n][j]);
    }
    return 0;
}
```

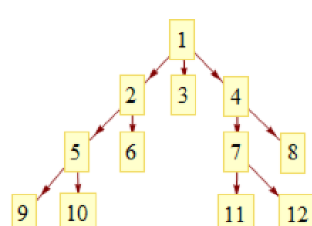
Example Input (DFS):

```
12 11 1
1 2
1 7
1 8
2 3
2 6
3 4
3 5
8 9
8 12
9 10
9 11
```



Example Input (BFS):

```
12 11 1
1 2
1 3
1 4
2 5
2 6
4 7
4 8
5 9
5 10
7 11
7 12
```



Output for Both Examples:

```
Nodes visited:
1 2 3 4 5 6 7 8 9 10 11 12
```

Note: In the BFS program, the line "`for (q.push(start); !q.empty(); q.pop())`" is simply a mnemonic for searching with a FIFO queue. This will *not* work as intended with a priority queue, such as in Dijkstra's algorithm.

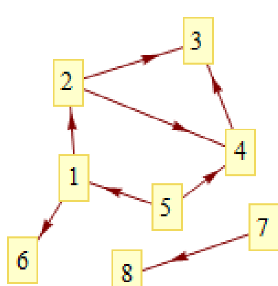
1.3 - Floodfill

Description: Given a directed graph and a source node, traverse to all reachable nodes from the source and determine the total area traveled. Logically, the order that nodes are visited in a floodfill should resemble a BFS. However, if the objective is simply to visit all reachable nodes without regard for the order (as is the case with most applications of floodfill in contests), it is much simpler to DFS because an extra queue is not needed. The graph is stored in an adjacency list.
Complexity: O(V) on the number of vertices.

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

const int MAX_N = 101;
int nodes, edges, a, b, source;
bool visit[MAX_N] = {0};
vector<int> adj[MAX_N];

int DFS(int node) {
    if (visit[node]) return 0;
    visit[node] = 1;
    cout << ", " << node;
    int area = 1;
    for (int j = 0; j < adj[node].size(); j++)
        area += DFS(adj[node][j]);
    return area;
}

int main() {
    cin >> nodes >> edges;
    for (int i = 0; i < edges; i++) {
        cin >> a >> b;
        adj[a].push_back(b);
    }
    cin >> source;
    cout << "Visited " << DFS(source);
    cout << " nodes, starting from " << source;
    return 0;
}
```

Example Input:

```
8 8
1 2
1 6
2 3
2 4
4 3
5 1
5 4
7 8
```



Output:

```
Visiting 1, 2, 3, 4, 6
Visited 5 nodes, starting from 1.
```

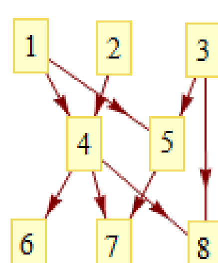
1.4 - Topological Sorting

Description: Given a directed acyclic graph (DAG), order the nodes such that for every edge from a to b, a precedes b in the ordering. Usually, there is more than one possible valid ordering. The following program uses DFS to produce one possible ordering. This can also be used to detect whether the graph is a DAG. Note that the DFS algorithm here produces a reversed topological ordering, so the output must be printed backwards. The graph is stored in an adjacency list.
Complexity: O(V) on the number of vertices.

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

const int MAX_N = 101;
int nodes, edges, a, b;
bool done[MAX_N] = {0}, visit[MAX_N] = {0};
vector<int> adj[MAX_N], sorted;

void DFS(int node) {
    if (visit[node]) {
        cout << "Error: Graph is not a DAG!\n";
        return;
    }
    if (done[node]) return;
    visit[node] = 1;
    for (int j = 0; j < adj[node].size(); j++)
        DFS(adj[node][j]);
    visit[node] = 0;
    done[node] = 1;
    sorted.push_back(node);
}

int main() {
    cin >> nodes >> edges;
    for (int i = 0; i < edges; i++) {
        cin >> a >> b;
        adj[a].push_back(b);
    }
    for (int i = 1; i <= nodes; i++)
        if (!done[i]) DFS(i);
    for (int i = sorted.size() - 1; i >= 0; i--)
        cout << " " << sorted[i];
    return 0;
}
```

Example Input:

```
8 9
1 4
1 5
2 4
3 5
3 8
4 6
4 7
4 8
5 7
```



Output:

```
The topological order:
3 2 1 5 4 8 7 6
```

1.5 - Shortest Paths

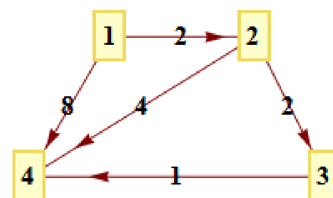
Description: Shortest path problems mainly fall into two categories: single-source, or all-pairs. Dijkstra's and Bellman-Ford's algorithms can be used to solve the former while the Floyd-Warshall algorithm can be used to solve the latter. BFS is a special case of Dijkstra's algorithm where the priority queue becomes a FIFO queue on unweighted graphs. Dijkstra's algorithm is a special case of A* search, where the heuristic is zero. The all-pairs shortest path on sparse graphs is best computed with Johnson's algorithm (a combination of Bellman-Ford and Dijkstra's). However, Johnson's algorithm and A* search are both rare in contests, and thus are ommitted.

1.5.1 - Dijkstra's Algorithm:

Complexity: The simplest version runs in $O(E+V^2)$ where V is the number of vertices and E is the number of edges. Using an adjacency list and priority queue (internally a binary heap), the implementation here is $\Theta((E+V)\log(V))$, dominated by $\Theta(E\log(V))$ if the graph is connected.[3]

Example Input:

```
4 5
1 2 2
1 4 8
2 3 2
2 4 4
3 4 1
1 4
```



Output:

```
The shortest distance from 1 to 4 is 5.
Take the path 1->2->3->4.
```

Implementation Notes: The graph is stored using an adjacency list. This implementation negates distances before adding them to the priority queue, since the container is a max-heap by default. This method is suggested in contests because it is easier than defining special comparators. An alternative would be declaring the queue with template parameters `priority_queue<pair<int,int>, vector<pair<int,int> >, greater<pair<int,int> > > pq;`" If the path is to be computed for only a single pair of nodes, one may break out of the loop as soon as the destination is reached, by inserting the line "`if (a == dest) break;`" after the line "`pq.pop();`".

Shortest Path Faster Algorithm: The code for Dijkstra's algorithm here can be easily modified to become the Shortest Path Faster Algorithm (SPFA) by simply commenting out "`visit[a] = true;`" and changing the priority queue to a FIFO queue like BFS. SPFA is a faster version of the Bellman-Ford algorithm, working on negative path lengths (whereas Dijkstra's cannot). However, certain graphs can be crafted to make the SPFA very slow.[4]

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

const int MAX_N = 101, INF = 1<<28;
int nodes, edges, a, b, weight, start, dest;
int dist[MAX_N], pred[MAX_N];
bool visit[MAX_N] = {0};
vector< pair<int, int> > adj[MAX_N];

int main() {
    cin >> nodes >> edges;
    for (int i = 0; i < edges; i++) {
        cin >> a >> b >> weight;
        adj[a].push_back(make_pair(b, weight));
    }
    cin >> start >> dest;

    for (int i = 0; i < nodes; i++) {
        dist[i] = INF;
        pred[i] = -1;
        visit[i] = false;
    }
    dist[start] = 0;
    priority_queue< pair<int, int> > pq;
    pq.push(make_pair(0, start));
    while (!pq.empty()) {
        a = pq.top().second;
        pq.pop();
        visit[a] = true;
        for (int j = 0; j < adj[a].size(); j++) {
            b = adj[a][j].first;
            if (visit[b]) continue;
            if (dist[b] > dist[a] + adj[a][j].second) {
                dist[b] = dist[a] + adj[a][j].second;
                pred[b] = a;
                pq.push(make_pair(-dist[b], b));
            }
        }
    }
    cout << "The shortest distance from " << start;
    cout << " to " << dest << " is " << dist[dest] << ".\n";

    /* Use pred[] to backtrack and print the path */
    int i = 0, j = dest, path[MAX_N];
    while (pred[j] != -1) j = path[++i] = pred[j];
    cout << "Take the path: ";
    while (i > 0) cout << path[i--] << "->";
    cout << dest << ".\n";
    return 0;
}
```

1.5.2 - Bellman-Ford Algorithm

Description: Given a directed graph with positive or negative weights but no negative cycles, find the shortest distance to all nodes from a single starting node. The input graph is stored using an edge list.
Complexity: O(V×E) on the number of vertices and edges, respectively.

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

const int MAX_N=101, MAX_E = MAX_N*MAX_N, INF=1<<28;
int nodes, edges, a, b, weight, start, dest;
int E[MAX_E][3], dist[MAX_N], pred[MAX_N];

int main() {
    cin >> nodes >> edges;
    for (int i = 0; i < edges; i++)
        cin >> E[i][0] >> E[i][1] >> E[i][2];
    cin >> start >> dest;

    for (int i = 1; i <= nodes; i++) {
        dist[i] = INF;
        pred[i] = -1;
    }
    dist[start] = 0;
    for (int i = 1; i <= nodes; i++)
      for (int j = 0; j < edges; j++)
        if (dist[E[j][1]] > dist[E[j][0]] + E[j][2]) {
            dist[E[j][1]] = dist[E[j][0]] + E[j][2];
            pred[E[j][1]] = E[j][0];
        }
    cout << "The shortest path from " << start;
    cout << " to " << dest << " is ";
    cout << dist[dest] << "." << endl;

    /* Optional: Report negative-weight cycles */
    for (int i = 0; i < edges; i++)
      if (dist[E[i][0]] + E[i][2] < dist[E[i][1]])
        cout << "Negative-weight cycle detected!\n";

    /* Use pred[] to backtrack and print the path */
    int i = 0, j = dest, path[MAX_N + 1];
    while (pred[j] != -1) j = path[++i] = pred[j];
    cout << "Take the path: ";
    while (i > 0) cout << path[i--] << "->";
    cout << dest << ".\n";
    return 0;
}
```

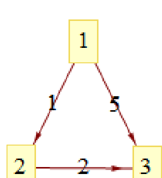
1.5.3 - Floyd-Warshall Algorithm

Description: Given a directed graph with positive or negative weights but no negative cycles, find the shortest distance between all pairs of nodes. The input graph is stored using an adjacency matrix.
Complexity: O(V^3) on the number of vertices.

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

const int MAX_N = 101, INF = 1<<28;
int nodes, edges, a, b, weight, start, dest;
int dist[MAX_N][MAX_N], next[MAX_N][MAX_N];

void print_path(int i, int j) {
    if (next[i][j] != -1) {
        print_path(i, next[i][j]);
        cout << next[i][j];
        print_path(next[i][j], j);
    } else cout << "->";
}

int main() {
    cin >> nodes >> edges;
    for (int i = 0; i < edges; i++) {
        cin >> a >> b >> weight;
        dist[a][b] = weight;
    }
    cin >> start >> dest;
    for (int i = 1; i <= nodes; i++)
        for (int j = 1; j <= nodes; j++) {
            dist[i][j] = (i == j) ? 0 : INF;
            next[i][j] = -1;
        }
    for (int k = 1; k <= nodes; k++)
     for (int i = 1; i <= nodes; i++)
      for (int j = 1; j <= nodes; j++)
        if (dist[i][j] > dist[i][k] + dist[k][j]) {
            dist[i][j] = dist[i][k] + dist[k][j];
            next[i][j] = k;
        }
    cout << "The shortest path from " << start;
    cout << " to " << dest << " is ";
    cout << dist[start][dest] << ".\n";

    /* Use next[][] to recursively print the path */
    cout << "Take the path " << start;
    print_path(start, dest);
    cout << dest << ".\n";
    return 0;
}
```

Example Input:

```
3 3
1 2 1
2 3 2
1 3 5
1 3
```



Output:

```
The shortest path from 1 to 3 is 3.
Take the path 1->2->3.
```

Implementation Note: Infinity is arbitrarily defined as `2^28=268,435,456`. Defining it closer to `INT_MAX` (`2,147,483,647`) will cause overflow errors during intermediate steps of the algorithms where addition with infinity may be performed.

1.6 - Minimum Spanning Trees

Description: Given an undirected graph, its minimum spanning tree (MST) is a tree connecting all nodes with a subset of its edges such that their total weight is minimized. Both Prim's and Kruskal's algorithms use a greedy algorithm to find the solution. Prim's algorithm requires a graph to be connected. Kruskal's algorithm uses a disjoint-set data structure, and will find the minimum spanning *forest* if the graph is not connected, thus making it more versatile than Prim's. Though a Fibonacci Heap would make Prim's faster than Kruskal's, it is unfeasible to implement during contests. Using a linear sorting algorithm like radix sort or bucket sort alongside a disjoint-set data structure that uses union by rank and path compression, Kruskal's can be optimized to run in $O(E \times \alpha(V))$, where α is the extremely slow growing inverse of the Ackermann function (see section 3.1 for an implementation of disjoint-set forests).

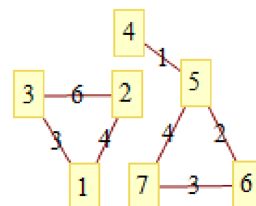
1.6.1 - Kruskal's Algorithm

Description: Given an undirected graph, its minimum spanning tree (MST) is a tree connecting all nodes with a subset of its edges such that their total weight is minimized. The input graph is stored in an edge list.
Complexity: O(E log(V)), where E is the number of edges and V is the number of vertices.[5]

```
#include <algorithm> /* std::sort() */
#include <iostream>
#include <vector>
using namespace std;

const int MAX_N = 101;
int nodes, edges, a, b, weight, root[MAX_N];
vector< pair<int, pair<int, int> > > E;
vector< pair<int, int> > MST;

int find_root(int x) {
    if (root[x] != x) root[x] = find_root(root[x]);
    return root[x];
}

int main() {
    cin >> nodes >> edges;
    for (int i = 0; i < edges; i++) {
        cin >> a >> b >> weight;
        E.push_back(make_pair(weight, make_pair(a, b)));
    }

    sort(E.begin(), E.end());
    for (int i = 1; i <= nodes; i++) root[i] = i;
    int totalDistance = 0;
    for (int i = 0; i < E.size(); i++) {
        a = find_root(E[i].second.first);
        b = find_root(E[i].second.second);
        if (a != b) {
            MST.push_back(E[i].second);
            totalDistance += E[i].first;
            root[a] = root[b];
        }
    }

    for (int i = 0; i < MST.size(); i++) {
        cout << MST[i].first << "<->";
        cout << MST[i].second << endl;
    }
    cout << "Total distance: " << totalDistance;
    return 0;
}
```

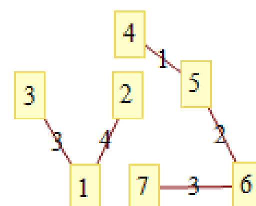
Example Input:

```
7 7
1 2 4
2 3 6
3 1 3
4 5 1
5 6 2
6 7 3
7 5 4
```



Output:

```
4<->5
5<->6
3<->1
6<->7
1<->2
Total distance: 13
```



Equivalent Code using an Explicitly Defined Disjoint Set Forest Data Structure (see 3.1)**:**

```
disjoint_set_forest<int> F;
sort(E.begin(), E.end());
for (int i = 1; i <= nodes; i++) F.make_set(i);
int totalDistance = 0;
for (int i = 0; i < E.size(); i++) {
    a = E[i].second.first;
    b = E[i].second.second;
    if (!F.is_united(a, b)) {
        MST.push_back(E[i].second);
        totalDistance += E[i].first;
        F.unite(a, b);
    }
}
```

Note: If the auxiliary disjoint set forest data structure is used, then both the `find_root()` function and the `root[]` array are not needed.

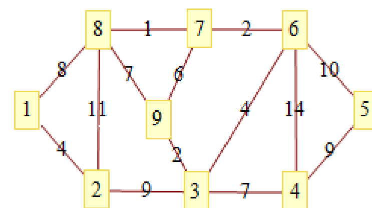
1.6.2 - Prim's Algorithm

Complexity: This implementation uses an adjacency list and priority queue (internally a binary heap) and has a complexity of $O((E+V)\log(V)) = O(E\log(V))$. The priority queue and adjacency list improves the simplest $O(V^2)$ version of the algorithm, which uses looping and an adjacency matrix. If the priority queue is implemented as a more sophisticated Fibonacci heap, the complexity reduces to $O(E+V\log(V))$.[6]

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

const int MAX_N = 101;
int nodes, edges, a, b, weight, pred[MAX_N];
bool visit[MAX_N] = {0};
vector< pair<int, int> > adj[MAX_N];

int main() {
    cin >> nodes >> edges;
    for (int i = 0; i < edges; i++) {
        cin >> a >> b >> weight;
        adj[a].push_back(make_pair(b, weight));
        adj[b].push_back(make_pair(a, weight));
    }

    for (int i = 1; i <= nodes; i++) {
        pred[i] = -1;
        visit[i] = false;
    }
    int start = 1; /* arbitrarily pick 1 as a starting node */
    visit[start] = true;
    priority_queue< pair<int, pair<int, int> > > pq;
    for (int j = 0; j < adj[start].size(); j++)
        pq.push(make_pair(-adj[start][j].second,
                 make_pair(start, adj[start][j].first)));
    int MSTnodes = 1, totalDistance = 0;
    while (MSTnodes < nodes) {
        if (pq.empty()) {
            cout << "Error: Graph is not connected!\n";
            return 0;
        }
        weight = -pq.top().first;
        a = pq.top().second.first;
        b = pq.top().second.second;
        pq.pop();
        if (visit[a] && !visit[b]) {
            visit[b] = true;
            MSTnodes++;
            pred[b] = a;
            totalDistance += weight;
            for (int j = 0; j < adj[b].size(); j++)
                pq.push(make_pair(-adj[b][j].second,
                         make_pair(b, adj[b][j].first)));
        }
    }
    for (int i = 2; i <= nodes; i++)
        cout << i << "<->" << pred[i] << "\n";
    cout << "Total distance: " << totalDistance << "\n";
    return 0;
}
```

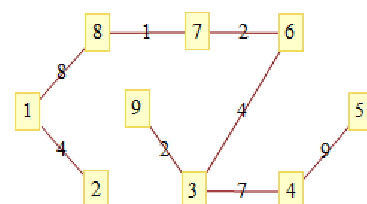
Example Input:

```
9 14
1 2 4
1 8 8
2 3 9
2 8 11
3 4 7
3 9 2
3 6 4
4 5 9
4 6 14
5 6 10
6 7 2
7 8 1
7 9 6
8 9 7
```



Output:

```
2<->1
3<->6
4<->3
5<->4
6<->7
7<->8
8<->1
9<->3
Total distance: 37
```



Implementation Notes:

The input graph is stored in an adjacency list. Similar to the implementation of Dijkstra's algorithm in 1.5.1, weights are negated before they are added to the priority queue (and negated once again when they are retrieved). To find the maximum spanning tree, simply skip the two negation steps and the highest weighted edges will be prioritized.

Prim's algorithm greedily selects edges from a priority queue, and is similar to Dijkstra's algorithm, where instead of processing nodes, we process individual edges. Note that the concept of the minimum spanning tree makes Prim's algorithm work with edge weights of an arbitrary sign. In fact, a big positive constant added to all of the edge weights of the graph will not change the resulting spanning tree.

1.7 - Maximum Flow

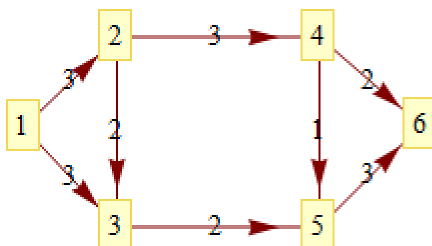
Description: Given a flow network, find a flow from a single source node to a single sink node that is maximized.

1.7.1 - Edmonds-Karp Algorithm

Complexity: O(V×E²), where V is the number of vertices and E is the number of edges. This improves the original Ford-Fulkerson algorithm, which has a complexity of O(E×|F|), where F is the max-flow of the graph.

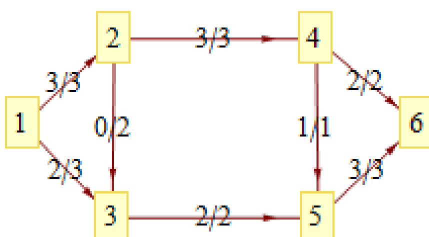
Example Input:

```
6 8
1 2 3
1 3 3
2 3 2
2 4 3
3 5 2
4 5 1
4 6 2
5 6 3
1 6
```



Output:

```
5
```



Implementation Notes: The flow capacities are stored in an adjacency matrix. The edges themselves are stored in an adjacency list to optimize the BFS. The function `edmonds_karp()` takes in three parameters: the number of nodes, the source node, and the sink node. Nodes are zero-based. That is, values in `adj[][]` and `flow[][]` must describe a graph with nodes labeled in the range `[0..nodes-1]`.

Despites its worst case complexity of O(V×E²), this algorithm appears to be relatively efficient in practice, and will even run in time for many cases where V=1000 and E=10000. For a faster algorithm, see Dinic's algorithm in section 1.7.2, which optimizes the Edmonds-Karp algorithm even further to a complexity of O(V²E).

Comparison with Ford-Fulkerson Algorithm: The Ford-Fulkerson algorithm is only optimal on graphs with integer capacities; there exists certain real-valued inputs for which it will never terminate. The Edmonds-Karp algorithm here also works on real-valued capacities.

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

const int MAX_N = 1000, INF = 1<<28;
int cap[MAX_N][MAX_N];
vector<int> adj[MAX_N];

int edmonds_karp(int nodes, int source, int sink) {
    int max_flow = 0, a, b, best[nodes], pred[nodes];
    bool visit[nodes];
    for (int i = 0; i < nodes; i++) best[i] = 0;
    while (true) {
        for (int i = 0; i < nodes; i++) visit[i] = false;
        visit[source] = true;
        best[source] = INF;
        pred[sink] = -1;
        queue<int> q;
        for (q.push(source); !q.empty(); q.pop()) {
            a = q.front();
            if (a == sink) break;
            for (int j = 0; j < adj[a].size(); j++) {
                b = adj[a][j];
                if (!visit[b] && cap[a][b] > 0) {
                    visit[b] = true;
                    pred[b] = a;
                    if (best[a] < cap[a][b])
                        best[b] = best[a];
                    else
                        best[b] = cap[a][b];
                    q.push(b);
                }
            }
        }
        if (pred[sink] == -1) break;
        for (int i = sink; i != source; i = pred[i]) {
            cap[pred[i]][i] -= best[sink];
            cap[i][pred[i]] += best[sink];
        }
        max_flow += best[sink];
    }
    return max_flow;
}

int main() {
    int nodes, edges, a, b, capacity, source, sink;
    cin >> nodes >> edges;
    for (int i = 0; i < edges; i++) {
        cin >> a >> b >> capacity; a--; b--;
        adj[a].push_back(b);
        cap[a][b] = capacity;
    }
    cin >> source >> sink;
    cout << edmonds_karp(nodes, source-1, sink-1);
    return 0;
}
```

```
#include <cstdio>
#include <vector>

const int MAX_N = 2005, INF = 1<<30;
struct edge { int to, rev, f; };

int nodes, source, sink;
std::vector<edge> adj[MAX_N];

void add_edge(int s, int t, int cap) {
    adj[s].push_back((edge){t, (int)adj[t].size(), cap});
    adj[t].push_back((edge){s, (int)adj[s].size()-1, 0});
}

int dist[MAX_N], queue[MAX_N], work[MAX_N];

bool dinic_bfs() {
    for (int i = 0; i < nodes; i++) dist[i] = -1;
    dist[source] = 0;
    int qh = 0, qt = 0;                /* queue<int> q; */
    queue[qt++] = source;              /* q.push(source); */
    while (qh < qt) {              /* while (!q.empty()) { */
        int u = queue[qh++];  /* u = q.front(); q.pop(); */
        for (int j = 0; j < adj[u].size(); j++) {
            edge &e = adj[u][j];
            int v = e.to;
            if (dist[v] < 0 && e.f) {
                dist[v] = dist[u] + 1;
                queue[qt++] = v;           /* q.push(v) */
            }
        }
    }
    return dist[sink] >= 0;
}

int dinic_dfs(int u, int f) {
    if (u == sink) return f;
    for (int &i = work[u]; i < adj[u].size(); i++) {
        edge &e = adj[u][i];
        if (!e.f) continue;
        int v = e.to, df;
        if (dist[v] == dist[u] + 1 &&
           (df = dinic_dfs(v, std::min(f, e.f))) > 0) {
            e.f -= df;
            adj[v][e.rev].f += df;
            return df;
        }
    }
    return 0;
}

int dinic() {
    int result = 0;
    while (dinic_bfs()) {
        for (int i = 0; i < nodes; i++) work[i] = 0;
        while (int delta = dinic_dfs(source, INF))
            result += delta;
    }
    return result;
}
```

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

int main() {
    int edges, a, b, capacity;
    cin >> nodes >> edges;
    for (int i = 0; i < edges; i++) {
        cin >> a >> b >> capacity;
        add_edge(a-1, b-1, capacity);
    }
    cin >> source >> sink;
    source--; sink--;
    cout << dinic() << endl;
    return 0;
}
```

1.7.2 - Dinic's Algorithm

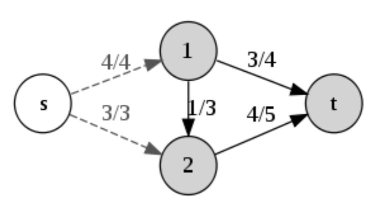
Complexity: $O(V^2 \times E)$ on the number of vertices and edges.

Example Input/Output: See previous section.

Comparison with Edmonds-Karp Algorithm: Dinic's algorithm is similar to the Edmonds-Karp algorithm in that it uses the shortest augmenting path. The introduction of the concepts of the level graph and blocking flow enable Dinic's algorithm to achieve its better performance. Hence, Dinic's algorithm is also called Dinic's blocking flow algorithm.

Max-flow Min-cut Theorem: The max-flow min-cut thorem states that in a flow network, the maximum amount of flow passing from the source to the sink is equal to the minimum capacity that, when removed in a specific way from the network, causes the situation that no flow can pass from the source to the sink.

Max-flow Min-cut Example:
The figure below is a network with a max-flow of 7. We want to block off all flow from the source node s, to the sink node t by removing a subset of the graph's edges, while minimizing the total capacity of the edges that are removed. The best way to do this is by removing the edges $s \to 1$ and $s \to 2$, which has a total capacity of $4 + 3 = 7$, the same value as the maxflow. Another way to block off the flow from s to t is by removing the edges $1 \to t$ and $2 \to t$. However the capacity removed would be $4 + 5 = 9$, and this is not optimal.



1.8 - Strongly Connected Components

Description: Given a directed graph, its strongly connected components (SCC) are its maximal strongly connected sub-graphs. A graph is strongly connected if there is a path from each node to every other node. Condensing the strongly connected components of a graph into single nodes will result in a directed acyclic graph

```
#include <algorithm> /* std::reverse() */
#include <iostream>
#include <vector>
using namespace std;

const int MAX_N = 101;
int nodes, edges, a, b;
bool visit[MAX_N] = {0};
vector<int> adj[MAX_N], rev[MAX_N], order;
vector< vector<int> > SCC;

void DFS(vector<int> graph[], vector<int> &res, int i) {
    visit[i] = true;
    for (int j = 0; j < graph[i].size(); j++)
        if (!visit[graph[i][j]])
            DFS(graph, res, graph[i][j]);
    res.push_back(i);
}

int main() {
    cin >> nodes >> edges;
    for (int i = 0; i < edges; i++) {
        cin >> a >> b;    a--, b--;
        adj[a].push_back(b);
    }

    for (int i = 0; i < nodes; i++) visit[i] = false;
    for (int i = 0; i < nodes; i++)
        if (!visit[i]) DFS(adj, order, i);
    for (int i = 0; i < nodes; i++)
        for (int j = 0; j < adj[i].size(); j++)
            rev[adj[i][j]].push_back(i);
    for (int i = 0; i < nodes; i++) visit[i] = false;
    reverse(order.begin(), order.end());
    for (int i = 0; i < order.size(); i++)
        if (!visit[order[i]]) {
            vector<int> component;
            DFS(rev, component, order[i]);
            SCC.push_back(component);
        }

    for (int i = 0; i < SCC.size(); i++) {
        cout << "Component " << i + 1 << ":";
        for (int j = 0; j < SCC[i].size(); j++)
            cout << " " << (SCC[i][j] + 1);
        cout << endl;
    }
    return 0;
}
```

1.8.1 - Kosaraju's Algorithm[7]

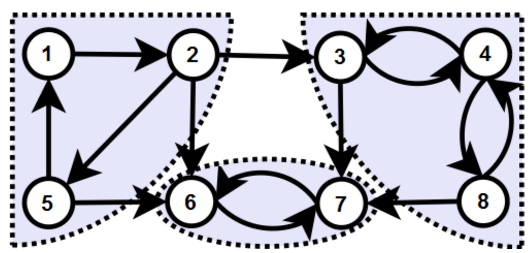
Complexity: Θ(V+E) on the number of vertices and edges.

Example Input:

```
8 14
1 2
2 3
2 5
2 6
3 4
3 7
4 3
4 8
5 1
5 6
6 7
7 6
8 4
8 7
```

Output:

```
Component 1: 2 5 1
Component 2: 8 4 3
Component 3: 6 7
```



Comparison with other SCC algorithms:
The strongly connected components of a graph can be efficiently computed using Kosaraju's algorithm, Tarjan's algorithm, or the path-based strong component algorithm. Tarjan's algorithm can be seen as an improved version of Kosaraju's because it performs a single DFS rather than two. Though they both have the same complexity, Tarjan's algorithm is much more efficient in practice. However, Kosaraju's algorithm is conceptually simpler.

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

const int MAX_N = 101;
int nodes, edges, a, b, counter;
int num[MAX_N], low[MAX_N];
bool visit[MAX_N] = {0};
vector<int> adj[MAX_N], stack;
vector< vector<int> > SCC;

void DFS(int a) {
    int b;
    low[a] = num[a] = ++counter;
    stack.push_back(a);
    for (int j = 0; j < adj[a].size(); j++) {
        b = adj[a][j];
        if (visit[b]) continue;
        if (num[b] == -1) {
            DFS(b);
            low[a] = min(low[a], low[b]);
        } else
            low[a] = min(low[a], num[b]);
    }
    if (num[a] != low[a]) return;
    vector<int> component;
    do {
        visit[b = stack.back()] = true;
        stack.pop_back();
        component.push_back(b);
    } while (a != b);
    SCC.push_back(component);
}

int main() {
    cin >> nodes >> edges;
    for (int i = 0; i < edges; i++) {
      cin >> a >> b;    a--, b--;
      adj[a].push_back(b);
    }
    for (int i = 0; i < nodes; i++) {
      num[i] = low[i] = -1;
      visit[i] = false;
    }
    counter = 0;
    for (int i = 0; i < nodes; i++)
      if (num[i] == -1) DFS(i);
    for (int i = 0; i < SCC.size(); i++) {
      cout << "Component " << i + 1 << ":";
      for (int j = 0; j < SCC[i].size(); j++)
         cout << " " << (SCC[i][j] + 1);
      cout << endl;
    }
    return 0;
}
```

1.8.2 - Tarjan's Algorithm[8]

Complexity: O(V+E) on the number of vertices and edges.

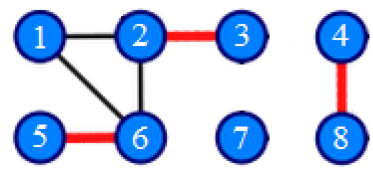
Example Input/Output: See previous section.

Implementation Notes: In this implementation, a vector is used to emulate a stack for the sake of simplicity. One useful property of Tarjan's algorithm is that, while there is nothing special about the ordering of nodes within each component, the resulting DAG is produced in reverse topological order.

1.8.3 - Tarjan's Bridge-Finding Algorithm

Description: Given an *undirected* graph, a bridge is an edge, when deleted, increases the number of connected components. An edge is a bridge iff if it is not contained in any cycle.

Complexity: O(V+E) on the number of vertices and edges.




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

const int MAX_N = 101;
int nodes, edges, a, b, counter, num[MAX_N], low[MAX_N];
bool visit[MAX_N] = {0};
vector<int> adj[MAX_N];

void DFS(int a, int c) {
    int b;
    low[a] = num[a] = ++counter;
    for (int j = 0; j < adj[a].size(); j++) {
        if (num[b = adj[a][j]] == -1) {
            DFS(b, a);
            low[a] = min(low[a], low[b]);
        } else if (b != c)
            low[a] = min(low[a], num[b]);
    }
    for (int j = 0; j < adj[a].size(); j++)
        if (low[b = adj[a][j]] > num[a])
            cout << a+1 << "-" << b+1 << " is a bridge\n";
}

int main() {
    cin >> nodes >> edges;
    for (int i = 0; i < edges; i++) {
        cin >> a >> b;    a--, b--;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }
    for (int i = 0; i < nodes; i++) {
        num[i] = low[i] = -1;
        visit[i] = false;
    }
    counter = 0;
    for (int i = 0; i < nodes; i++)
        if (num[i] == -1) DFS(i, -1);
    return 0;
}
```

Example Input:

```
8 6
1 2
1 6
2 3
2 6
4 8
5 6
```

Output:

```
6-5 is a bridge.
2-3 is a bridge.
4-8 is a bridge.
```

Section 2 – Sorting and Searching

Section Notes: The sorting functions in this section are included for novelty purposes, perhaps as demonstrations for ad hoc contest problems. With the exception of radix sort, there is generally no reason to use these sorts over C++'s built-in `std::sort()` and `std::stable_sort()` in an actual contest. The functions are implemented like `std::sort()`, taking in two `RandomAccessIterators` as the range to be sorted. To use with special comparators, either overload the < operator, or change every use of the < operator with array elements to a binary comparison function. E.g. to use `quicksort()` with a custom `comp()` function, change "`(*i < *lo)`" to "`(comp(*i, *lo))`".

```
#include <algorithm> /* std::swap(), std::(stable_)partition */
#include <iterator>  /* std::iterator_traits<T> */

template<class RAI> void quicksort(RAI lo, RAI hi) {
    if (lo + 1 >= hi) return;
    std::swap(*lo, *(lo + (hi - lo) / 2));
    RAI x = lo;
    for (RAI i = lo + 1; i < hi; i++)
        if (*i < *lo) std::swap(*(++x), *i);
    std::swap(*lo, *x);
    quicksort(lo, x);
    quicksort(x + 1, hi);
}

template<class RAI> void mergesort(RAI lo, RAI hi) {
    if (lo >= hi - 1) return;
    RAI mid = lo + (hi - lo - 1) / 2, a = lo, c = mid + 1;
    mergesort(lo, mid + 1);
    mergesort(mid + 1, hi);
    typedef typename std::iterator_traits<RAI>::value_type T;
    T *buff = new T[hi - lo], *b = buff;
    while (a <= mid && c < hi)
        *(b++) = (*c < *a) ? *(c++) : *(a++);
    if (a > mid)
        for (RAI k = c; k < hi; k++) *(b++) = *k;
    else
        for (RAI k = a; k <= mid; k++) *(b++) = *k;
    for (int i = hi - lo - 1; i >= 0; i--)
        *(lo + i) = buff[i];
}

template<class RAI> void heapsort(RAI lo, RAI hi) {
    typename std::iterator_traits<RAI>::value_type t;
    RAI n = hi, i = (n - lo) / 2 + lo, parent, child;
    while (true) {
        if (i <= lo) {
            if (--n == lo) return;
            t = *n, *n = *lo;
        } else t = *(--i);
        parent = i, child = lo + (i - lo) * 2 + 1;
        while (child < n) {
            if (child+1 < n && *child < *(child+1)) child++;
            if (!(t < *child)) break;
            *parent = *child, parent = child;
            child = lo + (parent - lo) * 2 + 1;
        }
        *(lo + (parent - lo)) = t;
    }
}
```

2.1 - Quicksort
Best: O(N log(N))
Average: O(N log(N))
Worst: O(N^2)
Auxiliary Space: O(log(N))
Stable?: No

Notes: The pivot value chosen here is always half way between lo and hi. Choosing random pivot values reduces the chances of encountering the worst case performance of O(N^2). Quicksort is faster in practice than other O(N log(N)) algorithms.

2.2 - Merge Sort
Best: O(N log(N))
Average: O(N log(N))
Worst: O(N log(N))
Auxiliary Space: O(N)
Stable?: Yes

Notes: Merge Sort has a better worse case complexity than quicksort and is stable (i.e. it maintains the relative order of equivalent values after the sort). This is different from `std::stable_sort()` in that it will simply fail if extra memory is not available. Meanwhile, `std::stable_sort()` will fall back to a run time of O(N $\log^2$(N)) if out of memory.

2.3 - Heap Sort
Best: O(N log(N))
Average: O(N log(N))
Worst: O(N log(N))
Auxiliary Space: O(1)
Stable?: No

Notes: Heap Sort has a better worst case complexity than quicksort, but also a better space complexity than merge sort. On average, this will very likely run slower than a well implemented Quicksort.

2.4 - Comb Sort[9]

Best: O(N)
Average: $O(N^2/2^p)$ for p increments of gap[10].
Worst: $O(N^2)$
Auxiliary Space: O(1)
Stable?: No

Notes: An improved bubble sort that can often be used to even replace O(N log(N)) sorts because it is so simple to memorize.

```
template<class RAI> void combsort(RAI lo, RAI hi) {
    int gap = hi - lo, swapped = 1;
    while (gap > 1 || swapped) {
        if (gap > 1) gap = (int)((float)gap/1.3f);
        swapped = 0;
        for (RAI i = lo; i + gap < hi; i++)
            if (*(i + gap) < *i) {
                std::swap(*i, *(i + gap));
                swapped = 1;
            }
    }
}
```

2.5 - Radix Sort (32-bit Signed Integers Only)[11]

Worst: O(d×N), where d is the number of digits in each of the N values. Since it is constant, one can say that radix sort runs in linear time.
Auxiliary Space: O(d)
Stable?: The LSD (least significant digit) radix sort is stable, but because `std::stable_partition()` is slower than the unstable `std::partition()`, the MSD (most significant digit) radix sort is much faster. Both are given here, with the MSD version implemented recursively.

Notes: Radix sort usually employs a bucket sort or counting sort. Here, we take advantage of the partition functions found in the C++ library.

```
struct radix_comp { //UnaryPredicate for std::partition
    const int bit;  //bit position [0..31] to examine
    radix_comp(int offset) : bit(offset) { }
    bool operator()(int value) const {
        return (bit==31) ? (value<0) : !(value&(1<<bit));
    }
};

void lsd_radix_sort(int *lo, int *hi) {
    for (int lsb = 0; lsb < 32; ++lsb)
        std::stable_partition(lo, hi, radix_comp(lsb));
}

void msd_radix_sort(int *lo, int *hi, int msb = 31) {
    if (lo == hi || msb < 0) return;
    int *mid = std::partition(lo, hi, radix_comp(msb--));
    msd_radix_sort(lo, mid, msb); //sort left partition
    msd_radix_sort(mid, hi, msb); //sort right partition
}
```

2.6 - Binary Search Query[12]

`BSQ_min()` returns the smallest value x in the range `[lo, hi)`, i.e. including lo, but not including hi, for which the boolean function `query(x)` is true. It can be used if and only if for all x in the queried range, `query(x)` implies `query(y)` for all `y>x`.

`BSQ_max()` returns the largest value x in the range `[lo, hi)` for which the boolean function `query(x)` returns true. It can be used if and only if for all x in the range, `query(x)` implies `query(y)` for all `y<x`.

For both of the functions: If all values in the range return false when queried, then `hi` is returned.

Examples:

x:	0 1 2 3 4 5 6	BSQ_min(0,7,query)
query(x):	0 0 0 1 1 1 1	>> returns 3
x:	0 1 2 3 4 5 6	BSQ_min(0,7,query)
query(x):	0 0 0 0 0 0 0	>> returns 7
x:	0 1 2 3 4 5 6	BSQ_max(0,7,query)
query(x):	1 1 1 1 1 1 0	>> returns 5
x:	0 1 2 3 4 5 6	BSQ_max(0,7,query)
query(x):	1 0 0 0 0 0 0	>> returns 0

```
template<class T, class UnaryPredicate>
T BSQ_min(T lo, T hi, UnaryPredicate query) {
    T mid;
    while (lo < hi) {
        mid = lo + (hi - lo)/2;
        if (query(mid)) hi = mid;
        else lo = mid + 1;
    }
    if (!query(lo)) return hi;
    return lo;
}

template<class T, class UnaryPredicate>
T BSQ_max(T lo, T hi, UnaryPredicate query) {
    T mid;
    while (lo < hi) {
        mid = lo + (hi - lo + 1)/2;
        if (query(mid)) lo = mid;
        else hi = mid - 1;
    }
    if (!query(lo)) return hi;
    return lo;
}
```

Section 3 – Data Structures

Section Notes: This section contains useful data structures *not* found in the C++ Standard Library. Garbage collection is support for all containers, but you may want to remove their destructors for more speed in contests.

3.1.1 - Disjoint Set Forest (Simple Version for Ints)

Description: This data structure dynamically keeps track of items partitioned into non-overlapping sets.
Time Complexity: Every function below is O(α(N)) amortized on the number of items in the set, due to the optimizations of union by rank and path compression.[13] α(N) is the extremely slow growing inverse of the Ackermann function. $\alpha(n) < 5$ for all practical values of n.
Space Complexity: O(N) total.

```
const int MAX_N = 1001;
int num_sets = 0, root[MAX_N], rank[MAX_N];

int find_root(int x) {
  if (root[x] != x) root[x] = find_root(root[x]);
  return root[x];
}
void make_set(int x) {
  root[i] = i; rank[i] = 0; num_sets++;
}
bool is_united(int x, int y) {
  return find_root(x) == find_root(y);
}
void unite(int x, int y) {
  int X = find_root(x), Y = find_root(y);
  if (X == Y) return;
  num_sets--;
  if (rank[X] < rank[Y]) root[X] = Y;
  else if (rank[X] > rank[Y]) root[Y] = X;
  else rank[root[Y] = X]++;
}
```

Example Usage (for OOP Version):

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

int main() {
  disjoint_set_forest<char> d;
  for (char c='a'; c<='g'; c++) d.make_set(c);
  d.unite('a', 'b'); d.unite('b', 'f');
  d.unite('d', 'e'); d.unite('e', 'g');
  cout << "Elements: " << d.elements();
  cout << ", Sets: " << d.sets() << endl;
  vector< vector<char> > V = d.get_all_sets();
  for (int i = 0; i < V.size(); i++) {
    cout << "{ ";
    for (int j = 0; j < V[i].size(); j++)
      cout << V[i][j] << " ";
    cout << "}";
  }
  return 0;
}
```

Output:

```
Elements: 7, Sets: 3
{ a b f }{ c }{ d e g }
```

3.1.2 - Disjoint Set Forest (OOP Version with Compression)

Description: This template version uses an `std::map` for built in storage and coordinate compression.
Time Complexity: `make_set()`, `unite()` and `is_united()` are O(log(N)) on the number of elements in the disjoint set forest. `get_all_sets()` is O(N). `find()` is O(α(N)) amortized.
Space Complexity: O(N) total.

```
#include <map>
#include <vector>

template<class T> class disjoint_set_forest {
  int num_elements, num_sets;
  std::map<T, int> ID;
  std::vector<int> root, rank;

  int find_root(int x) {
      if (root[x] != x) root[x] = find_root(root[x]);
      return root[x];
  }

 public:
  disjoint_set_forest(): num_elements(0), num_sets(0) {}
  int elements() { return num_elements; }
  int sets() { return num_sets; }

  bool is_united(const T &x, const T &y) {
      return find_root(ID[x]) == find_root(ID[y]);
  }
  void make_set(const T &x) {
      if (ID.find(x) != ID.end()) return;
      root.push_back(ID[x] = num_elements++);
      rank.push_back(0);
      num_sets++;
  }
  void unite(const T &x, const T &y) {
      int X = find_root(ID[x]), Y = find_root(ID[y]);
      if (X == Y) return;
      num_sets--;
      if (rank[X] < rank[Y]) root[X] = Y;
      else if (rank[X] > rank[Y]) root[Y] = X;
      else rank[root[Y] = X]++;
  }
  std::vector< std::vector<T> > get_all_sets() {
      std::map< int, std::vector<T> > tmp;
      for (typename std::map<T, int>::iterator
        it = ID.begin(); it != ID.end(); it++)
          tmp[find_root(it->second)].push_back(it->first);
      std::vector< std::vector<T> > ret;
      for (typename std::map<int, std::vector<T> >::
        iterator it = tmp.begin(); it != tmp.end(); it++)
          ret.push_back(it->second);
      return ret;
  }
};
```

3.2.1 - Binary Indexed Tree[14] (Point Update with Range Query)

Description: A binary indexed tree (a.k.a. Fenwick Tree or BIT) is a data structure that allows for the sum of an arbitrary range of values in an array to be dynamically queried in logarithmic time.
Time Complexity: `query()` and `update()` are O(log(N)). All other functions are O(1).
Space Complexity: O(N) storage and O(N) auxiliary on the number of elements in the array.

```
template<class T> class binary_indexed_tree {
    int SIZE;
    T *data, *bits;

    T _query(int hi) {
        T sum = 0;
        for (; hi > 0; hi -= hi & -hi)
            sum += bits[hi];
        return sum;
    }

  public:
    binary_indexed_tree(int N): SIZE(N+1) {
        data = new T[SIZE];
        bits = new T[SIZE];
        for (int i = 0; i < SIZE; i++)
            data[i] = bits[i] = 0;
    }

    ~binary_indexed_tree() {
        delete[] data;
        delete[] bits;
    }

    void update(int i, const T &newval) {
        T inc = newval - data[++i];
        data[i] = newval;
        for (; i < SIZE; i += i & -i)
            bits[i] += inc;
    }

    int size()  { return SIZE - 1; }
    T at(int i) { return data[i + 1]; }
    T query(int lo = 0, int hi = 0) {
        return _query(hi + 1) - _query(lo);
    }
};
```

Example Usage:

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

int main() {
    int a[] = {10, 1, 2, 3, 4};
    binary_indexed_tree<int> BIT(5);
    for (int i=0; i<5; i++) BIT.update(i,a[i]);
    cout << "BIT values: ";
    for (int i=0; i<BIT.size(); i++)
        cout << BIT.at(i) << " ";
    cout << "\nSum of range [1,3] is ";
    cout << BIT.query(1, 3) << ".\n";
    return 0;
}
```

Output:

```
BIT values: 10 1 2 3 4
Sum of range [1,3] is 6.
```

3.2.2 - 2D BIT (Point Update with Range Query)

Description: A 2D BIT is abstractly a 2D array which also supports efficient queries for the sum of values in the rectangle with bottom-left (0,0) and top-right (x,y). The 2D BIT implemented below has indices accessible in the range `[0...rows-1][0...cols-1]`.
Time Complexity: `query()` and `update()` are both O(log(rows)*log(cols)). All other functions are O(1).
Space Complexity: O(rows*cols) storage and auxiliary.

```
#include <vector>

template<class T> class BIT2D {
    std::vector< std::vector<int> > data, sums;

    T _query(int r, int c) {
        T sum = 0;
        for ( ; r > 0; r -= r & -r)
            for (int C = c; C > 0; C -= C & -C)
                sum = sum + sums[r][C];
        return sum;
    }

  public:
    BIT2D(int R, int C): sums(R+1), data(R+1) {
        for (int i = R; i >= 0; i--) {
            sums[i].resize(C+1);
            data[i].resize(C+1);
        }
    }

    void update(int r, int c, const T &newval) {
        T inc = newval - data[++r][++c];
        data[r][c] = newval;
        for ( ; r < sums.size(); r += r & -r)
            for (int C = c; C < sums[r].size(); ) {
                sums[r][C] = sums[r][C] + inc;
                C += C & -C;
            }
    }

    int rows() { return data.size() - 1; }
    int cols() { return data[0].size() - 1; }
    T at(int r, int c)    { return   data[r+1][c+1]; }
    T query(int r, int c) { return _query(r+1, c+1); }

  /* sum of cells in rectangle with top left corner  *
     at (r1, c2) and bottom right corner at (r2, c2) */
    T query(int r1, int c1, int r2, int c2) {
        return query(r2, c2) + query(r1-1, c1-1) -
               query(r1-1, c2) - query(r2, c1-1);
    }
};
```

3.2.3 - Binary Indexed Tree (Range Update with Range Query, without Co-ordinate Compression)

Description: Using two arrays, a BIT can be made to support range updates and range queries simultaneously.

```
template<class T> class binary_indexed_tree {
  int SIZE;
  T *B1, *B2;

  T Pquery(T* B, int i) {
    T sum = 0;
    for (; i != 0; i -= i & -i) sum += B[i];
    return sum;
  }

  T Rquery(int i) {
    return Pquery(B1, i)*i - Pquery(B2, i);
  }

  T Rquery(int L, int H) {
    return Rquery(H) - Rquery(L-1);
  }

  void Pupdate(T* B, int i, const T &inc) {
    for (; i <= SIZE; i += i & -i) B[i] += inc;
  }

  void Rupdate(int L, int H, T inc) {
    Pupdate(B1, L, inc);
    Pupdate(B1, H+1, -inc);
    Pupdate(B2, L, inc*(L-1));
    Pupdate(B2, H+1, -inc*H);
  }

 public:
  binary_indexed_tree(int N): SIZE(N+1) {
    B1 = new T[SIZE+1];
    B2 = new T[SIZE+1];
    for (int i = 0; i <= SIZE; i++)
      B1[i] = B2[i] = 0;
  }

 ~binary_indexed_tree() {
     delete[] B1; delete[] B2;
  }

  int size() { return SIZE - 1; }
  T at(int idx) { return Rquery(idx+1, idx+1); }
  T query(int L, int H) { return Rquery(L+1, H+1); }

  void update(int i, const T &newval) {
    Rupdate(i+1, i+1, newval - at(i + 1));
  } /* sets index value at index i to newval */

  void inc_range(int L, int H, const T &inc) {
    Rupdate(L+1, H+1, inc);
  } /* adds inc to every value in range [L,H] */
};
```

3.2.4 - Binary Indexed Tree, Simplified (Range Update with Point Query, without Co-ordinate Compression)

```
const int SIZE = 10001;
int BIT[SIZE];

void internal_update(int i, int inc) {
  for (i++; i <= SIZE; i += i & -i) BIT[i] += inc;
}

void update(int L, int H, int inc) {
  internal_update(L, inc);
  internal_update(H+1, -inc);
} /* add inc to each value in range [L,H] */

int query(int i) {
  int sum = 0;
  for (i++; i > 0; i -= i & -i) sum += BIT[i];
  return sum;
} /* returns value at index i, NOT the sum [0,i] */
```

3.2.5 - Binary Indexed Tree, Simplified (Range Update with Range Query, with Co-ordinate Compression)[15]

```
#include <map>

const int SIZE = 1<<30;
std::map<int, int> dataMul, dataAdd;

void internal_update(int at, int mul, int add) {
  for (int i = at; i < SIZE; i = (i | (i+1))) {
    dataMul[i] += mul;
    dataAdd[i] += add;
  }
}

void update(int L, int H, int inc) {
  internal_update(L, inc, -inc*(L-1));
  internal_update(H, -inc, inc*H);
} /* add inc to each value in range [L,H] */

int query(int x) {
  int mul = 0, add = 0, start = x;
  for (int i = x; i >= 0; i = (i & (i+1)) - 1) {
    if (dataMul.find(i) != dataMul.end())
      mul += dataMul[i];
    if (dataAdd.find(i) != dataAdd.end())
      add += dataAdd[i];
  }
  return mul*start + add;
} /* returns sum of range [0,x] */

int query(int L, int H) {
  return query(H) - query(L-1);
} /* returns sum of range [L,H] */
```

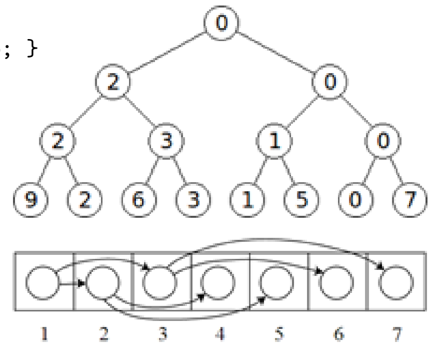
3.3.1 - 1D Segment Tree

Description: A segment tree is a data structure used for solving the dynamic range query problem, which asks to determine the maximum (or minimum) value in any given range in an array that is constantly being updated.

Time Complexity: `at()`, `query()` and `update()` are O(log(N)). All other functions are O(1).

Space Complexity: O(N) on the size of the array. A segment tree needs an array of size 2×2^$(\log_2(N)+1) = 4N$.

```
#include <limits> /* std::numeric_limits<T>::min() */

template<class T> class segment_tree {
    int SIZE;
    T *data;

    static inline const T MAX(const T &a, const T &b) { return a > b ? a : b; }

    void internal_update(int node, int lo, int hi, int idx, const T &val) {
        if (idx < lo || idx > hi || lo > hi) return;
        if (lo == hi) { data[node] = val; return; }
        internal_update(node*2 + 1, lo, (lo + hi)/2, idx, val);
        internal_update(node*2 + 2, (lo + hi)/2 + 1, hi, idx, val);
        data[node] = MAX(data[node*2 + 1], data[node*2 + 2]);
    }

    T internal_query(int node, int lo, int hi, int LO, int HI) {
        if (lo > HI || hi < LO || lo > hi) return std::numeric_limits<T>::min();
        if (lo >= LO && hi <= HI) return data[node];
        return MAX(internal_query(node*2 + 1, lo, (lo + hi)/2, LO, HI),
                   internal_query(node*2 + 2, (lo + hi)/2 + 1, hi, LO, HI));
    }

  public:
    segment_tree(int N): SIZE(N) { data = new T[4*N]; }
    ~segment_tree() { delete[] data; }
    int size() { return SIZE; }
    void update(int idx, const T &val) { internal_update(0, 0, SIZE - 1, idx, val); }
    T query(int lo, int hi) { return internal_query(0, 0, SIZE - 1, lo, hi); }
    T at(int idx) { return internal_query(0, 0, SIZE - 1, idx, idx); }
};
```



Example Usage:

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

int main() {
    int arr[5] = {6, 4, 1, 8, 10};
    segment_tree<int> T(5);
    for (int i = 0; i < 5; i++) T.update(i, arr[i]);
    cout << "Array contains:";
    for (int i = 0; i < T.size(); i++)
        cout << " " << T.at(i);
    cout << "\nThe max value in the range [0, 3] is ";
    cout << T.query(0, 3) << ".\n";
    return 0;
}
```

Output:

```
Array contains: 6 4 1 8 10
The max value in the range [0, 3] is 8
```

Implementation Notes:

See section 3.3.2 for an implementation of `build()`. The value retrieved by `query()` is always the max value in the inclusive range. Use one of the two ways below to make the tree handle minimum values:

1. Call `update()` with negated values and negate once again upon retrieval.
2. Redefine `MAX()` and change `numeric_limits<T>::min()` in `internal_query()` to a custom `null_value`, defined such that for any value `x` stored in the tree, `MAX(x, null_value)` always returns `x`.

3.3.2 - 1D Segment Tree (with Lazy Propagation)

Description: Lazy propagation is a technique applied to segment trees that allows range updates to be carried out in O(log(N)) complexity.
Complexity: Same as 3.3.1. Updating ranges takes only one O(log(N)) operation. `build()` is O(N).

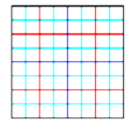
```
template<class T> class segment_tree {

  int len;
  T *tree, *lazy, NEG_INF;

  /* temporary variables to speed up recursion */
  int I, J;         T VAL;          const T *A;

 public:
  static inline const T MAX(const T& a, const T& b);

  segment_tree(int len, const T& neginf,
               const T* const array = 0) {
    this->len = len;
    this->NEG_INF = neginf;
    tree = new int[4*len];
    lazy = new int[4*len];
    for (int i = 0; i<4*len; i++) lazy[i] = neginf;
    if (array != 0) {
      A = array;
      build(0, 0, len-1);
    }
  }

 ~segment_tree() { delete[] tree; delete[] lazy; }
  int size() { return len; }

  T at(int i) {
      I = i; J = i; return _query(0, 0, len-1);
  } /* returns value at array index i */

  T query(int i, int j) {
      I = i; J = j; return _query(0, 0, len-1);
  } /* returns max value in range[i, j] */

  void update(int i, const T &v) {
      I = i; J = i; VAL = v; _update(0, 0, len-1);
  } /* sets value at array index i to v */

  void update(int i, int j, const T &v) {
      I = i; J = j; VAL = v; _update(0, 0, len-1);
  } /* sets all values in range [i, j] to v */

 private:
  void build(int node, int lo, int hi) {
    if (lo == hi) {
      tree[node] = A[lo];
      return;
    }
    build(node*2 + 1, lo, (lo + hi)/2);
    build(node*2 + 2, (lo + hi)/2 + 1, hi);
    tree[node] = MAX(tree[node*2+1],tree[node*2+2]);
  }

  void _update(int node, int lo, int hi) {
    if (I > hi || J < lo) return;
    if (lo == hi) {
      tree[node] = VAL;
      return;
    }
    if (I <= lo && hi <= J) {
      lazy[node] = MAX(lazy[node], VAL);
      tree[node] = lazy[node];
      return;
    }
    int mid = (lo + hi)/2;
    int Lchild = node*2 + 1, Rchild = node*2 + 2;
    if (lazy[node] != NEG_INF) {
      lazy[Lchild] = lazy[Rchild] = lazy[node];
      lazy[node] = NEG_INF;
    }
    _update(Lchild, lo, mid);
    _update(Rchild, mid+1, hi);
    tree[node] = MAX(tree[Lchild], tree[Rchild]);
  }

  T _query(int node, int lo, int hi) {
    if (I > hi || J < lo) return NEG_INF;
    if (I <= lo && hi <= J) {
      if (lazy[node] == NEG_INF) return tree[node];
      tree[node] = lazy[node];
      return tree[node];
    }
    int mid = (lo + hi)/2;
    int Lchild = node*2 + 1, Rchild = node*2 + 2;
    if (lazy[node] != NEG_INF) {
      lazy[Lchild] = lazy[Rchild] = lazy[node];
      lazy[node] = NEG_INF;
    }
    return MAX(_query(Lchild, lo, mid),
               _query(Rchild, mid+1, hi));
  }
};
```

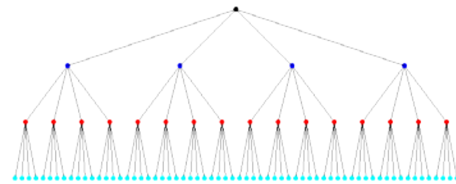
Example Usage:

```
/* define MAX() so MAX(NEG_INF, x) returns x for all x */

template<class T> inline const T segment_tree<T>
::MAX(const T& a, const T& b) { return a>b?a:b; }

#include <iostream>
using namespace std;

int main() {
  int arr[5] = {6, 4, 1, 8, 10};
  segment_tree<int> T(5, -100000000, arr);
  T.update(1, 3, 5);             // arr: 6 5 5 5 10
  T.update(3, 7);                // arr: 6 5 5 7 10
  cout << T.query(0, 3) << endl; // prints 7
  T.update(1, 9);                // arr: 6 9 5 7 10
  cout << T.query(1, 3) << endl; // prints 9
  return 0;
}
```

3.3.3 - Quadtree

Description: A quadtree can be used to dynamically query values of rectangules in a 2D array. In a quadtree, every node has exactly 4 children. The following uses a statically allocated array to store the nodes.
Time Complexity: For `update()`, `query()` and `at()`: O(log(N*M)) on average and O(sqrt(N*M)) in the worst case.
Space Complexity: O(N log(N))





```
#include <cmath>  /* ceil(), log2() to calculate array size */
#include <limits> /* std::numeric_limits<T>::min() */

template<class T> class quadtree {
    int XMAX, YMAX; /* indices are [0..XMAX-1][0..YMAX-1] */
    T *data, tempval;

    static inline const T MAX(const T& a, const T& b) { return a > b ? a : b; }

    void _update(int p, int xlo, int xhi, int ylo, int yhi, int X, int Y) {
        if (X < xlo || X > xhi || Y < ylo || Y > yhi) return;
        if (xlo == xhi && ylo == yhi) { data[p] = tempval; return; }
        _update(p*4+1, xlo, (xlo+xhi)/2, ylo, (ylo+yhi)/2, X, Y);
        _update(p*4+2, xlo, (xlo+xhi)/2, (ylo+yhi)/2+1, yhi, X, Y);
        _update(p*4+3, (xlo+xhi)/2+1, xhi, ylo, (ylo+yhi)/2, X, Y);
        _update(p*4+4, (xlo+xhi)/2+1, xhi, (ylo+yhi)/2+1, yhi, X, Y);
        data[p] = MAX(MAX(data[p*4+1], data[p*4+2]), MAX(data[p*4+3], data[p*4+4]));
    }

    void _query(int p, int xlo, int xhi, int ylo, int yhi, int XLO, int XHI, int YLO, int YHI) {
        if (xlo > XHI || xhi < XLO || yhi < YLO || ylo > YHI || MAX(data[p], tempval) == tempval) return;
        if (xlo >= XLO && xhi <= XHI && ylo >= YLO && yhi <= YHI) { tempval = MAX(data[p], tempval); return; }
        _query(p*4+1, xlo, (xlo+xhi)/2, ylo, (ylo+yhi)/2, XLO, XHI, YLO, YHI);
        _query(p*4+2, xlo, (xlo+xhi)/2, (ylo+yhi)/2+1, yhi, XLO, XHI, YLO, YHI);
        _query(p*4+3, (xlo+xhi)/2+1, xhi, ylo, (ylo+yhi)/2, XLO, XHI, YLO, YHI);
        _query(p*4+4, (xlo+xhi)/2+1, xhi, (ylo+yhi)/2+1, yhi, XLO, XHI, YLO, YHI);
    }

   public:
    quadtree(int N, int M): XMAX(N), YMAX(M) {
        data = new T[(int)ceil(4*N*M*log2(4*N*M))];
    }

   ~quadtree() { delete[] data; }

    int xmax() { return XMAX; }
    int ymax() { return YMAX; }

    void update(int X, int Y, T val) {
        tempval = val;
        _update(0, 0, XMAX-1, 0, YMAX-1, X, Y);
    }

    T query(int xlo, int ylo, int xhi, int yhi) {
        tempval = std::numeric_limits<T>::min();
        _query(0, 0, XMAX-1, 0, YMAX-1, xlo, xhi, ylo, yhi);
        return tempval;
    }

    T at(int X, int Y) {
        tempval = std::numeric_limits<T>::min();
        _query(0, 0, XMAX-1, 0, YMAX-1, X, X, Y, Y);
        return tempval;
    }
};
```

Implementation Notes:
See the note in section 3.3.1 on how to define `MAX()` and why `std::numeric_limits`. is used as the negative infinity. All indices are zero-based. All queried and updated ranges are inclusive (i.e. `[xlo..xhi][ylo..yhi]`).

Memory Optimization:
Use a map instead of an array to store `data` to support updates and queries of larger indices (say, for N, M < 1 billion). Alternatively, dynamically allocate nodes, where nodes are implemented as a value and 4 pointers to the 4 children.

Example Usage:

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

int main() {
    int arr[5][5] = {{1, 2, 3, 4, 5},
                     {5, 4, 3, 2, 1},
                     {6, 7, 8, 0, 0},
                     {0, 1, 2, 3, 4},
                     {5, 9, 9, 1, 2}};

    quadtree<int> T(5, 5);
    for (int r = 0; r < T.xmax(); r++)
      for (int c = 0; c < T.ymax(); c++)
        T.update(r, c, arr[r][c]);

    cout << "The maximum value in the rectangle with ";
    cout << "upper left (0,2) and lower right (3,4) is ";
    cout << T.query(0, 2, 3, 4) << ".\n";
    return 0;
}
```

Output:

```
The maximum value in the rectangle with upper left (0,2)
and lower right (3,4) is 8.
```

3.3.4 - 2D Segment Tree[16]

Description: The statically allocated quadtree in 3.3.3 is inefficient on large indicies. The following implementation is a 2D segment tree with various optimizations

Time Complexity: O(log(XMAX)×log(YMAX))

Space Complexity: Left as an exercise for the reader.

```
template<class T> class segment_tree_2D {
#define XMAX 1000000000 /* indices are [0..XMAX][0..YMAX] */
#define YMAX 1000000000 /* XMAX, YMAX can be very large!  */

  struct layer2_node {
    int lo, hi;
    layer2_node *L, *R;
    T value;
    layer2_node(int l, int h) : lo(l), hi(h), L(0), R(0) {}
  };

  struct layer1_node {
    layer1_node *L, *R;
    layer2_node l2;
    layer1_node() : L(0), R(0), l2(0, YMAX) {}
  } *root;

  T NEG_INF;    /* MAX(NEG_INF, x) must return x for all x */
  inline const T& MAX(const T& a, const T& b);
  /* e.g. { return a>b?a:b; } - see example in 3.3.2 */

  void update2(layer2_node* node, int Q, const T &K) {
    int lo = node->lo, hi = node->hi, mid = (lo + hi) / 2;
    if (lo + 1 == hi) {
      node->value = K;
      return;
    }
    layer2_node*& tgt = Q < mid ? node->L : node->R;
    if (tgt == 0) {
      tgt = new layer2_node(Q, Q + 1);
      tgt->value = K;
    } else if (tgt->lo <= Q && Q < tgt->hi) {
      update2(tgt, Q, K);
    } else {
      do {
        (Q < mid ? hi : lo) = mid;
        mid = (lo + hi) / 2;
      } while ((Q < mid) == (tgt->lo < mid));
      layer2_node *nnode = new layer2_node(lo, hi);
      (tgt->lo < mid ? nnode->L : nnode->R) = tgt;
      tgt = nnode;
      update2(nnode, Q, K);
    }
    node->value = MAX(node->L ? node->L->value : NEG_INF,
                      node->R ? node->R->value : NEG_INF);
  }

  T query2(layer2_node* nd, int A, int B) {
    if (nd == 0 || B <= nd->lo || nd->hi <= A) return NEG_INF;
    if (A <= nd->lo && nd->hi <= B) return nd->value;
    return MAX(query2(nd->L, A, B), query2(nd->R, A, B));
  }

  void update1(layer1_node* node, int lo, int hi,
               int x, int y, T val) {
    if (lo + 1 == hi) update2(&node->l2, y, val);
    else {
      int mid = (lo + hi) / 2;
      layer1_node*& nnode = x < mid ? node->L : node->R;
      (x < mid ? hi : lo) = mid;
      if (nnode == 0) nnode = new layer1_node();
      update1(nnode, lo, hi, x, y, val);
      val = MAX(
        node->L ? query2(&node->L->l2, y, y+1) : NEG_INF,
        node->R ? query2(&node->R->l2, y, y+1) : NEG_INF);
      update2(&node->l2, y, val);
    }
  }

  T query1(layer1_node* nd, int lo, int hi,
           int A1, int B1, int A2, int B2) {
    if (nd == 0 || B1 <= lo || hi <= A1) return NEG_INF;
    if (A1 <= lo && hi <= B1)
      return query2(&nd->l2, A2, B2);
    int mid = (lo + hi) / 2;
    return MAX(query1(nd->L, lo, mid, A1, B1, A2, B2),
               query1(nd->R, mid, hi, A1, B1, A2, B2));
  }

  void clean_up2(layer2_node* n) {
    if (n == 0) return;
    clean_up2(n->L); clean_up2(n->R);
    delete n;
  }

  void clean_up1(layer1_node* n) {
    if (n == 0) return;
    clean_up1(n->L); clean_up2(n->l2.L);
    clean_up1(n->R); clean_up2(n->l2.R);
    delete n;
  }

 public:
  segment_tree_2D(const T& neginf): NEG_INF(neginf) {
     root = new layer1_node();
  }
  ~segment_tree_2D() { clean_up1(root); }

  void update(int x, int y, const T &val) {
    update1(root, 0, XMAX, x, y, val);
  }
  T query(int x1, int y1, int x2, int y2) {
    return query1(root, 0, XMAX, x1, x2 + 1, y1, y2 + 1);
  }
  T at(int x, int y) { return query(x, y, x, y); }

};
```

Example Usage: Same as 3.3.3, but the constructor only requires `NEG_INF` since indices can be very large. `segment_tree_2D::MAX()` must be defined (see 3.3.2).

3.4 - Binary Search Tree

Description: A binary search tree (BST) is a node-based binary tree data structure where the left sub-tree of every node has keys less than the node's key and the right sub-tree of every node has keys greater (greater or equal in this implementation) than the node's key. A BST may be come degenerate like a linked list resulting in an O(N) running time per operation. A self-balancing binary search tree such as a randomized Treap, or a more complicated AVL Tree solves this problem with a worst case of O(log(N)).

3.4.1 - Simple Binary Search Tree

Complexity: `insert()`, `remove()` and `find()` are O(log(N)) on average, but O(N) at worst if the tree becomes degenerate. Speed can be improved by randomizing insertion order. `walk()` is O(N). All other functions are O(1).

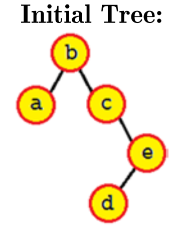
```
template<class key_t, class val_t> class binary_search_tree {
    struct node_t {
        key_t key; val_t val;
        node_t *left, *right;
    } *root;

    int _size;

    void _insert(node_t *&node, const key_t &k, const val_t &v) {
        if (node == 0) {
            node = new node_t();
            node->key = k;
            node->val = v;
            _size++;
        } else if (k < node->key) {
            _insert(node->left, k, v);
        } else
            _insert(node->right, k, v);
    }

    bool _remove(node_t *&ptr, const key_t &key) {
        if (ptr == 0) return false;
        if (key < ptr->key) return _remove(ptr->left, key);
        if (ptr->key < key) return _remove(ptr->right, key);
        if (ptr->left == 0) {
            node_t *temp = ptr->right;
            delete ptr;
            ptr = temp;
        } else if (ptr->right == 0) {
            node_t *temp = ptr->left;
            delete ptr;
            ptr = temp;
        } else {
            node_t *temp = ptr->right, *parent = 0;
            while (temp->left != 0) {
                parent = temp;
                temp = temp->left;
            }
            ptr->key = temp->key;
            ptr->val = temp->val;
            if (parent != 0)
                return _remove(parent->left, parent->left->key);
            return _remove(ptr->right, ptr->right->key);
        }
        _size--;
        return true;
    } /* Returns whether the key was successfully removed */

    void clean_up(node_t *&n) {
        if (n == 0) return;
        clean_up(n->left);
        clean_up(n->right);
        delete n;
    }

    template<class Func>
     void _walk(node_t *p, void(*f)(Func), int order) {
          if (p == 0) return;
          if (order <  0) (*f)(p->val);
          if (p->left) _walk(p->left, f, order);
          if (order == 0) (*f)(p->val);
          if (p->right) _walk(p->right, f, order);
          if (order >  0) (*f)(p->val);
      }

  public:
    binary_search_tree(): root(0), _size(0) {}
    ~binary_search_tree() { clean_up(root); }
    int size() const { return _size; }
    bool empty() const { return root == 0; }
    void insert(const key_t &key, const val_t &val) {
        _insert(root, key, val);
    }
    bool remove(key_t key) {
        return _remove(root, key);
    }
   template<class Func>
    void walk(void (*f)(Func), int order = 0) {
        _walk(root, f, order);
    }
    val_t* find(const key_t &key) {
        for (node_t *n = root; n != 0; ) {
          if (n->key == key) return &(n->val);
          if (key < n->key) n = n->left;
          else n = n->right;
        }
        return 0; /* key not found */
    }
};
```

Output:
```
In-order: abcde
Removed node with key 3
Pre-order: baed
Post-order: adeb
```

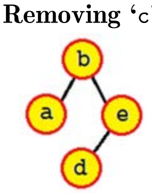
Example Usage:

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

void printch(char c) { cout << c; }

int main() {
    binary_search_tree<int, char> T;
    T.insert(2, 'b'); T.insert(1, 'a');
    T.insert(3, 'c'); T.insert(5, 'e');
    T.insert(4, 'x'); *T.find(4) = 'd';
    cout << "In-order: "; T.walk(printch, 0);
    cout << "\nRemoved node with key 3";
    T.remove(3);
    cout << "\nPre-order: "; T.walk(printch, -1);
    cout << "\nPost-order: "; T.walk(printch, 1);
    return 0;
}
```

Initial Tree:



Removing 'c':

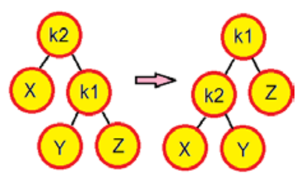


3.4.2 - Treap[17]

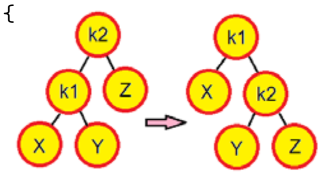
Notes: Treaps use randomly generated priorities to reduce the height of the tree. We assume that the `rand()` function in `<cstdlib>` is 16-bits, and call it twice to generate a 32-bit number. For the treap to be effective, the range of the randomly generated numbers should be between 0 and around the number of elements in the treap.

Complexity: `insert()`, `remove()`, and `find()` are O(log(N)) amortized in the worst case. `walk()` is O(N).

```
#include <cstdlib> /* srand(), rand() */
#include <ctime>   /* time() */

template<class key_t, class val_t> class treap {

  struct node_t {
    key_t key; val_t val;
    int priority;
    node_t *L, *R;
    static inline int rand_int(int l, int h) {
      return l+(((int)rand()<<16)|rand())%(h-l+1);
    }
    node_t(const key_t&k, const val_t&v): key(k), val(v),
      L(0), R(0), priority(rand_int(0, 1<<30)) {}
  } *root;

  int _size;

  void rotateL(node_t *&k2) {
    node_t *k1 = k2->R;
    k2->R = k1->L;
    k1->L = k2;
    k2 = k1;
  }
```




```
  void rotateR(node_t *&k2) {
    node_t *k1 = k2->L;
    k2->L = k1->R;
    k1->R = k2;
    k2 = k1;
  }
```




```
template<class Func>
  void _walk(node_t *p, void (*f)(Func), int order) {
    if (p == 0) return;
    if (order <  0) (*f)(p->val);
    if (p->L) _walk(p->L, f, order);
    if (order == 0) (*f)(p->val);
    if (p->R) _walk(p->R, f, order);
    if (order >  0) (*f)(p->val);
  }

  void _insert(node_t*&n, const key_t&k, const val_t&v) {
    node_t *p_node = new node_t(k, v);
    if (n == 0) { n = p_node; _size++; return; }
    if (k < n->key) {
      _insert(n->L, k, v);
      if (n->L->priority < n->priority) rotateR(n);
    } else {
      _insert(n->R, k, v);
      if (n->R->priority < n->priority) rotateL(n);
    }
  }

  bool _remove(node_t *&n, const key_t &k) {
    if (n == 0) return false;
    if (k < n->key) return _remove(n->L, k);
    if (k > n->key) return _remove(n->R, k);
    if (n->L == 0 || n->R == 0) {
      node_t *temp = n;
      n = (n->L != 0) ? n->L : n->R;
      delete temp;
      _size--;
      return true;
    }
    if (n->L->priority < n->R->priority) {
      rotateR(n);
      return _remove(n->R, k);
    }
    rotateL(n);
    return _remove(n->L, k);
  }

  void clean_up(node_t *&n) {
    if (n == 0) return;
    clean_up(n->L); clean_up(n->R);
    delete n;
  }

 public:
  treap(): root(0), _size(0) { srand(time(0)); }
  ~treap() { clean_up(root); }
  int size() const { return _size; }
  bool empty() const { return root == 0; }

  void insert(const key_t &k, const val_t &v) {
    _insert(root, k, v);
  }

  bool remove(const key_t &k) {
    return _remove(root, k);
  }

 template<class Func>
  void walk(void (*f)(Func), int order = 0) {
    _walk(root, f, order);
  }

  val_t* find(const key_t &k) {
    for (node_t *n = root; n != 0; ) {
      if (n->key == k) return &(n->val);
      if (k < n->key) n = n->L;
      else n = n->R;
    }
    return 0; /* key not found, return NULL */
  }
};
```

3.4.3 - AVL Tree[18]

Notes: Other alternatives to the AVL Tree includes red-black trees (used internally by C++ to implement `std::map` and `std::set`), splay trees, weight-balanced trees and Tango trees.
Complexity: `insert()`, `remove()`, `find()`, and `subtree()` are at worst O(log(N)). `walk()` is O(N).

```
template<class key_t, class val_t> class avl_tree {

  struct node {
    key_t key; val_t val;
    node *parent, *L, *R;
    int height;

    node(const key_t &k, const val_t &v):
    key(k), val(v), height(0), parent(0), L(0), R(0) {}

    int updateHeight() {
      if (L != 0 && R != 0) {
        if (L->height > R->height)
          return height = L->height + 1;
        return height = R->height + 1;
      }
      if (L != 0) return height = L->height + 1;
      if (R != 0) return height = R->height + 1;
      return height = 0;
    }

    int getBalance() {
      node *n = this;
      if (n->L != 0 && n->R != 0)
        return n->L->height - n->R->height;
      if (n->L != 0) return n->L->height + 1;
      if (n->R != 0) return -(n->R->height + 1);
      return 0;
    }

    node* setL(node *n) {
      if (n != 0) n->parent = this;
      L = n; updateHeight(); return L;
    }

    node* setR(node *n) {
      if (n != 0) n->parent = this;
      R = n; updateHeight(); return R;
    }
  } *root;

  int _size;

  void setRoot(node *n) {
    if ((root = n) != 0) root->parent = (node*)0;
  }

  node* find_node(const key_t& key) {
    for (node* n = root; n != 0; ) {
      if (key < n->key) n = n->L;
      else if (n->key < key) n = n->R;
      else return n;
    }
    return (node*)0;
  }

  void rotateL(node *n) {
    node *p = n->parent; enum { L, R } side;
    if (p != 0) side = (p->L == n) ? L : R;
    node *temp = n->R;
    n->setR(temp->L);
    temp->setL(n);
    if (p == 0) setRoot(temp);
    else if (side == L) p->setL(temp);
    else if (side == R) p->setR(temp);
  }

  void rotateR(node *n) {
    node *p = n->parent; enum { L, R } side;
    if (p != 0) side = (p->L == n) ? L : R;
    node *temp = n->L;
    n->setL(temp->R);
    temp->setR(n);
    if (p == 0) setRoot(temp);
    else if (side == L) p->setL(temp);
    else if (side == R) p->setR(temp);
  }

  void balance(node *n) {
    int bal = n->getBalance();
    if (bal > 1) {
      if (n->L->getBalance() < 0) rotateL(n->L);
      rotateR(n);
    } else if (bal < -1) {
      if (n->R->getBalance() > 0) rotateR(n->R);
      rotateL(n);
    }
  }

 template <class Func>
  void _walk(node *p, void (*f)(Func), char order) {
    if (p == 0) return;
    if (order == -1) (*f)(p->val);
    _walk(p->L, f, order);
    if (order == 0) (*f)(p->val);
    _walk(p->R, f, order);
    if (order == 1) (*f)(p->val);
  }

  void clean_up(node *n) {
    if (n == 0) return;
    clean_up(n->L); clean_up(n->R);
    delete n;
  }

 public:
  avl_tree(): _size(0) { root = (node*)0; }
  avl_tree(const key_t &k, const val_t &v):
     _size(1), root(new node(k, v)) {}
  ~avl_tree() { clean_up(root); }
  int height() { return root->height; }
  int size() { return _size; }
```

```
avl_tree* subtree(const key_t &k) {
  node *target = find_node(k);
  if (target == 0) return (avl_tree*)0;
  avl_tree *subtree = new avl_tree();
  subtree->root = target;
  return subtree;
} /* subtree(k) */

val_t* find(const key_t &k) {
  for (node *n = root; n != 0; ) {
    if (k < n->key) n = n->L;
    else if (n->key < k) n = n->R;
    else return &(n->val);
  }
  return 0;
} /* find(k) */

template <class Func>
void walk(void (*f)(Func), char order = 0) {
  _walk(root, f, order);
} /* walk(f, order) */

bool insert(const key_t &k, const val_t &v) {
  if (root == 0) {
    root = new node(k, v);
    _size++;
    return true;
  }
  node *tmp = root, *added;
  while (true) {
    if (k < tmp->key) {
      if ((tmp->L) == 0) {
        added = tmp->setL(new node(k, v));
        break;
      } else tmp = tmp->L;
    } else if (tmp->key < k) {
      if ((tmp->R) == 0) {
        added = tmp->setR(new node(k, v));
        break;
      } else tmp = tmp->R;
    } else return false;
  }
  for (tmp = added; tmp != 0; tmp = tmp->parent) {
    tmp->updateHeight();
    balance(tmp);
  }
  _size++;  return true;
} /* insert(k, v) */

bool remove(const key_t &k) {
  if (root == 0) return false;
  node *new_n, *new_p, *tmp, *n = find_node(k), *p;
  if (n == 0) return false;
  int bal = n->getBalance(); enum {L, R} side;
  if ((p = n->parent) != 0) side = (p->L == n) ? L : R;
  if (n->L == 0 && n->R == 0) {
    if (p != 0) {
      if (side == L) p->setL((node*)0);
      else p->setR((node*)0);
      delete n;
      p->updateHeight();
      balance(p);
    } else {
      setRoot((node*)0);
      delete n;
    }
  } else if (n->R == 0) {
    if (p != 0) {
      if (side == L) p->setL(n->L); else p->setR(n->L);
      delete n;
      p->updateHeight();
      balance(p);
    } else {
      setRoot(n->L);
      delete n;
    }
  } else if (n->L == 0) {
    if (p != 0) {
      if (side == L) p->setL(n->R); else p->setR(n->R);
      delete n;
      p->updateHeight();
      balance(p);
    } else {
      setRoot(n->R);
      delete n;
    }
  } else {
    if (bal > 0) {
      if (n->L->R == 0) {
        tmp = new_n = n->L;
        new_n->setR(n->R);
      } else {
        new_n = n->L->R;
        while (new_n->R != 0) new_n = new_n->R;
        tmp = new_p = new_n->parent;
        new_p->setR(new_n->L);
        new_n->setL(n->L);
        new_n->setR(n->R);
      }
    } else {
      if (n->R->L == 0) {
        tmp = new_n = n->R;
        new_n->setL(n->L);
      } else {
        new_n = n->R->L;
        while (new_n->L != 0) new_n = new_n->L;
        tmp = new_p = new_n->parent;
        new_p->setL(new_n->R);
        new_n->setL(n->L);
        new_n->setR(n->R);
      }
    }
    if (p != 0) {
      if (side == L) p->setL(new_n);
      else p->setR(new_n);
    } else setRoot(new_n);
    delete n;
    balance(tmp);
  }
  _size--;
  return true;
} /* remove(k) */

};
```

3.5 -Hashmap

Description: A hashmap is an alternative to binary search trees. Hashmaps use more memory than BSTs, but are usually more efficient. In C++11, the built in hashmap is known as `std::unordered_map`. This implementation uses chaining to deal with collisions. Three integer hash algorithms and one string hash algorithm are presented here.[19]

Complexity: `insert()`, `remove()`, `find()`, are O(1) amortized. `rehash()` is O(N).

```
#include <list>

template<class key_t, class val_t, class Hash> class hashmap {
  struct entry_t {
    key_t key;
    val_t val;
    entry_t(const key_t& k, const val_t& v): key(k), val(v) {}
  };

  std::list<entry_t> *table;
  int table_size, map_size;

  /**
   * This doubles the table size, then rehashes every entry.
   * Rehashing is expensive; it is strongly suggested for the
   * table to be constructed with a large size to avoid rehashing.
   */
  void rehash() {
    std::list<entry_t> *old = table;
    int old_size = table_size;
    table_size = 2*table_size;
    table = new std::list<entry_t>[table_size];
    map_size = 0;
    typename std::list<entry_t>::iterator it;
    for (int i = 0; i < old_size; i++)
      for (it = old[i].begin(); it != old[i].end(); ++it)
        insert(it->key, it->val);
    delete[] old;
  }

 public:
  hashmap(int SZ = 1024): table_size(SZ), map_size(0) {
    table = new std::list<entry_t>[table_size];
  }
  ~hashmap() { delete[] table; }
  int size() const { return map_size; }

  void insert(const key_t &key, const val_t &val) {
    if (find(key) != 0) return;
    if (map_size >= table_size) rehash();
    unsigned int i = Hash()(key) % table_size;
    table[i].push_back(entry_t(key, val));
    map_size++;
  }

  void remove(const key_t &key) {
    unsigned int i = Hash()(key) % table_size;
    typename std::list<entry_t>::iterator it = table[i].begin();
    while (it != table[i].end() && it->key != key) ++it;
    if (it == table[i].end()) return;
    table[i].erase(it);
    map_size--;
  }

  val_t* find(const key_t &key) {
    unsigned int i = Hash()(key) % table_size;
    typename std::list<entry_t>::iterator it = table[i].begin();
    while (it != table[i].end() && it->key != key) ++it;
    if (it == table[i].end()) return 0;
    return &(it->val);
  }

  inline val_t& operator[] (const key_t &key) {
    val_t *ret = find(key);
    if (ret != 0) return *ret;
    insert(key, val_t());
    return *find(key);
  }
};
```

Example Usage:

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

struct class_hash {
  unsigned int operator() (int key) {
    /* Knuth's multiplicative method (one-to-one) */
    return key * 2654435761u;
  }
  unsigned int operator() (unsigned int key) {
    /* Robert Jenkins' 32-bit mix (one-to-one) */
    key = ~key + (key << 15);
    key = key ^ (key >> 12);
    key = key + (key << 2);
    key = (key ^ (key >> 4)) * 2057;
    return key ^ (key >> 16);
  }
  unsigned int operator() (unsigned long long key) {
    key = (~key) + (key << 18);
    key = (key ^ (key >> 31)) * 21;
    key = key ^ (key >> 11);
    key = key + (key << 6);
    return key ^ (key >> 22);
  }
  unsigned int operator() (const std::string &key) {
    /* Jenkins' one-at-a-time hash */
    unsigned int hash = 0;
    for (unsigned int i = 0; i < key.size(); i++) {
      hash += ((hash += key[i]) << 10);
      hash ^= (hash >> 6);
    }
    hash ^= ((hash += (hash << 3)) >> 11);
    return hash + (hash << 15);
  }
};

int main() {
  hashmap<string, int, class_hash> M;
  M["foo"] = 1; M.insert("bar", 2);
  cout << M["foo"] << M["bar"] << endl;  //prints 12
  cout << M["baz"] << M["qux"] << endl;  //prints 00
  M.remove("foo");
  cout << M.size() << endl;              //prints 3
  cout << M["foo"] << M["bar"] << endl;  //prints 02
  return 0;
}
```

Section 4 - Mathematics

Section Notes: The running times of many algorithms in this section may be harder to determine due to micro-optimizations. The reader is responsible for assessing their performances by inspection/benchmarking. If an algorithm is too slow for a problem, you are likely not approaching it correctly.

4.1 - Rounding Real Values: Given a real number x, this returns x rounded half-up to N decimal places in O(1).

```
#include <cmath> /* modf(), floor(), ceil(), pow() */

double round(double x, unsigned int N = 0) {
    double discard;
    if (modf(x *= pow(10, N), &discard) >= 0.5) return (x < 0 ? floor(x) : ceil(x)) / pow(10,N);
    return (x < 0 ? ceil(x) : floor(x)) / pow(10,N);
}
```

4.2 - Generating Combinations[20]

Given a range of N elements defined by iterators lo and hi, this function rearranges the elements such that the elements in [lo,K) (i.e. including lo but not including K) are the next combination chosen from [lo,hi) that is lexographically greater than the current elements in that range. It returns 0 if there are no lexicographically greater combinations in [lo,K).

Complexity: Left as an exercise for the reader. The follow benchmarks on an array populated with values from 0 to N-1 should give an idea of its performance on a typical CPU.

N	k	N choose k	Time
20	10	184,756	0.00s
50	45	2,118,760	0.09s
2000	2	1,999,000	2.96s
105	100	96,560,646	4.31s
3000	2	4,498,500	9.95s

```
#include <algorithm> /* rotate(), iter_swap(), */
                     /* swap(), swap_ranges()  */

template<typename Value, typename Iterator>
void disjoint_rotate(Iterator L1, size_t sz1,
      Iterator L2, size_t sz2, Value *type)
{
  const size_t total = sz1 + sz2;
  size_t gcd = total;
  for (size_t div = sz1; div != 0; )
     std::swap(gcd %= div, div);
  const size_t skip = total / gcd - 1;
  for (size_t i = 0; i < gcd; i++) {
    Iterator curr((i<sz1)?L1+i:L2+(i-sz1));
    size_t k = i;
    const Value v(*curr);
    for (size_t j = 0; j < skip; ++j) {
      k = (k + sz1) % total;
      Iterator next((k<sz1)?L1+k:L2+(k-sz1));
      *curr = *next;
      curr = next;
    }
    *curr = v;
  }
}
```

```
template<typename Iterator>
bool next_combination(Iterator lo, Iterator K, Iterator hi)
{
  if (lo == K || K == hi) return false;
  Iterator Lpos(K), Rpos(hi);
  --Lpos; --Rpos;
  size_t Llen = 1;
  while (Lpos != lo && !(*Lpos < *Rpos)) --Lpos, ++Llen;
  if (Lpos == lo && !(*Lpos < *Rpos)) {
    std::rotate(lo, K, hi);
    return false;
  }
  size_t Rlen = 1;
  while (Rpos > K) {
    --Rpos; ++Rlen;
    if (!(*Rpos > *Lpos)) {
      ++Rpos; --Rlen;
      break;
    }
  }
  if (Llen == 1 || Rlen == 1) {
    std::iter_swap(Lpos, Rpos);
    return true;
  }
  if (Llen == Rlen) {
    std::swap_ranges(Lpos, K, Rpos);
    return true;
  }
  std::iter_swap(Lpos, Rpos);
  disjoint_rotate(Lpos+1, Llen-1, Rpos+1, Rlen-1, &*Lpos);
  return true;
}
```

Example Usage (5 choose 3 with the string "11234"):

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

int main() {
    string s = "11234";
    do {
        cout << s.substr(0, 3) << " ";
    } while(next_combination(s.begin(), s.begin()+3, s.end()));
    return 0;
}
```

Output: `112 113 114 123 124 134 234`

4.3 - Number Theory

4.3.1 - GCD and LCM: These uses Euclid's algorithm to compute the GCD and LCM in O(log(a + b)).

```
int gcd(int a, int b) { return b ? gcd(b,a%b) : a; }
int lcm(int a, int b) {
    if (a == 0 || b == 0) return 0;
    return a / gcd(a, b) * b;
}
```

4.3.2 - Primality Testing (Deterministic): This optimizes trial division using the fact that all primes greater than 3 take the form 6n±1. Complexity: O(sqrt(N)).

```
bool is_prime(int N) {
    if (N == 2 || N == 3) return true;
    if (N < 2 || !(N % 2) || !(N % 3)) return false;
    for (int i = 5, w = 4; i*i <= N; i += (w = 6 - w))
        if (N % i == 0) return false;
    return true;
}
```

4.3.3 - Primality Testing (Probabilistic)[21]
The Miller-Rabin primality test checks whether a number p is probably prime. If p is prime, the function is guaranteed to return 1. If p is composite, the function returns 1 with a probability of $(1/4)^k$, where k is the number of iterations. With k=1, the probability of a composite being falsely predicted to be a prime is 25%. If k=5, the probability for this error is just less than 0.1%. Thus, k=18-20 is accurate enough for most applications. All values of p less than 2^63 is supported.
Complexity: O(k $\log^3$(p)). In comparison to trial division, the Miller-Rabin algorithm on 32-bit integers take ~45 operations for k=10 iterations (~0.0001% error), while the former takes ~10,000.

Implementation of `randULL()` (remember to call `srand(time(0))` beforehand):

```
inline unsigned long long randULL() {
return ((unsigned long long)rand() << 48) |
       ((unsigned long long)rand() << 32) |
       ((unsigned long long)rand() << 16) |
       ((unsigned long long)rand());
}
```

```
long long mulmod(long long a, long long b, long long c) {
    unsigned long long x = 0, y = a % c;
    for (; b > 0; b /= 2) {
        if (b % 2 == 1) x = (x + y) % c;
        y = (y * 2) % c;
    }
    return x % c;
}
long long powmod(long long a, long long b, long long c) {
    unsigned long long x = 1, y = a;
    for (; b > 0; b /= 2) {
        if (b % 2 == 1) x = mulmod(x, y, c);
        y = mulmod(y, y, c);
    }
    return x % c;
}
bool is_probable_prime(long long p, int k = 20) {
    if (p < 2 || (p != 2 && p % 2 == 0)) return 0;
    unsigned long long s = p - 1, x, mod;
    while (s % 2 == 0) s /= 2;
    for (int i = 0; i < k; i++) {
        mod = powmod(randULL() % (p - 1) + 1, s, p);
        for (x = s; x != p-1 && mod != 1 && mod != p-1; x *= 2)
            mod = mulmod(mod, mod, p);
        if (mod != p-1 && x % 2 == 0) return 0;
    }
    return 1;
}
```

4.3.4 - Prime Generation (Sieve of Eratosthenes): This fills the array `primes` with all primes up to N and returns the number of primes generated. Complexity: O(N log(log(N))).

```
int generate_primes(int N, int P[]) {
  int cnt = 0, sqrt_limit = sqrt(N);
  std::vector<bool> prime(N, true);
  for (int n = 2; n <= sqrt_limit; ++n) {
    if (!prime[n]) continue;
    P[cnt++] = n;
    for (unsigned k = n*n; k < N; k += n) prime[k] = 0;
  }
  for (int n = sqrt_limit + 1; n < N; n++)
    if (prime[n]) P[cnt++] = n;
  return cnt;
}
```

4.3.5 - Prime Factorization (Trial Division): This fills the array `factors` with the prime factorization of N, then returns the number of factors. E.g. For N=120, the function gets {2,2,2,3,5} and returns 5.
Complexity: O(sqrt(N))

```
int prime_factorize(int N, int factors[]) {
    int i = 2, numfactors = 0;
    while (i * i <= N)
        if (N % i == 0) {
            factors[numfactors++] = i;
            N /= i;
        } else i++;
    if (N > 1) factors[numfactors++] = N;
    return numfactors;
}
```

4.4 - Rational Number Class

Description: Operations on exact rational values. The sign is stored in the numerator. It is recommended to use `long long` for the template type since intermediate operations may overflow a normal `int`.

```
template<class Int> struct rational {
    Int num, den;

    rational(Int N = 0, Int D = 1) {
        if (D == 0) throw "Denominator cannot be 0";
        num = D < 0 ? -N : N;
        den = D < 0 ? -D : D;
        Int a = (num<0 ? -num : num), b = den, tmp;
        while (a != 0 && b != 0) {
            tmp = a % b; a = b; b = tmp;
        }
        Int gcd = (b == 0) ? a : b;
        num /= gcd; den /= gcd;
    }

    bool operator <  (const rational& r) const {
        return num*r.den < r.num*den;
    }

    bool operator >  (const rational& r) const {
        return num*r.den > r.num*den;
    }

    bool operator <= (const rational& r) const {
        return !((*this) > r);
    }

    bool operator >= (const rational& r) const {
        return !((*this) < r);
    }

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

    bool operator != (const rational& r) const {
        return num != r.num || den != r.den;
    }

    rational operator + (const rational& r) const {
        return  rational(num*r.den+r.num*den, den*r.den);
    }

    rational operator - (const rational& r) const {
        return  rational(num*r.den-r.num*den, r.den*den);
    }

    rational operator * (const rational& r) const {
        return rational(num*r.num, r.den*den);
    }

    rational operator / (const rational& r) const {
        return rational(num*r.den, den*r.num);
    }

    operator double() { return (double)num / den; }
};
```

4.5 - Base Conversion

Description: Given the digits of a number X in base A as a vector of its digits from most to least significant, this returns a vector containing the digits of X expressed in base B. The value of the number being converted must be small enough to be stored in an unsigned 64-bit integer.

Complexity: O(N+M), where N is the number of elements in vector X and M is the number of elements in the vector that is returned.

Output: `1 1 0 2`

```
#include <cmath>  /* pow(), ceil(), log() */
#include <vector>

std::vector<int> convert_base(const std::vector<int> &X, int A, int B) {
    int base10 = 0;
    for (int i=0; i<X.size(); i++) base10 += pow(A, X.size()-i-1)*X[i];
    unsigned long long N = ceil(log(base10 + 1) / log(B));
    std::vector<int> baseB;
    for (int i=1; i<=N; i++) baseB.push_back(int(base10/pow(B,N-i))%B);
    return baseB;
}
```

Example (convert 123_5 to base 3):

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

int main() {
    int digits[] = {1, 2, 3};
    vector<int> ans = convert_base(vector<int>(digits, digits+3), 5, 3);
    for (int i = 0; i < ans.size(); i++) cout << ans[i] << " ";
    return 0;
}
```

4.6 - Arbitrary-Precision Arithmetic[22]

Description: Integer arbitrary precision functions. To use, pass BigInts to the functions by addresses. E.g. calling "`add(&a, &b, &c)`" will store the sum of a and b into c.
Complexity: `comp()`, `to_string()`, `digit_shift()`, `add()` and `sub()` are O(N) on the number of digits. `mul()` and `div()` are O(N^2). `zero_justify()` is amortized constant.

```
#include <string>

struct BigInt {
#define MAXDIGITS 100
    char dig[MAXDIGITS], sign;
    int last;
    BigInt(long long x = 0): sign(x < 0 ? -1 : 1) {
        for (int i = 0; i < MAXDIGITS; i++) dig[i] = 0;
        if (x == 0) { last = 0; return; }
        if (x < 0) x = -x;
        for (last = -1; x > 0; x /= 10) dig[++last] = x%10;
    }
    BigInt(const std::string &s): sign(s[0] == '-' ? -1 : 1) {
        for (int i = 0; i < MAXDIGITS; i++) dig[i] = 0;
        last = -1;
        for (int i = s.size() - 1; i >= 0; i--)
            dig[++last] = (s[i] - '0');
        if (dig[last]+'0' == '-') dig[last--] = 0;
    }
};

void add(BigInt *a, BigInt *b, BigInt *c);
void sub(BigInt *a, BigInt *b, BigInt *c);

void zero_justify(BigInt *x) {
    while (x->last > 0 && !x->dig[x->last]) x->last--;
    if (x->last == 0 && x->dig[0] == 0) x->sign = 1;
}

int comp(BigInt *a, BigInt *b) {
    if (a->sign != b->sign) return b->sign;
    if (b->last > a->last) return  a->sign;
    if (a->last > b->last) return -a->sign;
    for (int i = a->last; i >= 0; i--) {
        if (a->dig[i] > b->dig[i]) return -a->sign;
        if (b->dig[i] > a->dig[i]) return  a->sign;
    }
    return 0;
} /* returns: -1 if a < b,  0 if a == b,  or 1 if a > b */

void add(BigInt *a, BigInt *b, BigInt *c) {
    if (a->sign != b->sign) {
        if (a->sign == -1)
            a->sign = 1, sub(b, a, c), a->sign = -1;
        else
            b->sign = 1, sub(a, b, c), b->sign = -1;
        return;
    }
    c->sign = a->sign;
    c->last = (a->last > b->last ? a->last : b->last) + 1;
    for (int i = 0, carry = 0; i <= c->last; i++) {
        c->dig[i] = (carry + a->dig[i] + b->dig[i]) % 10;
        carry = (carry + a->dig[i] + b->dig[i]) / 10;
    }
    zero_justify(c);
}

void sub(BigInt *a, BigInt *b, BigInt *c) {
    if (a->sign == -1 || b->sign == -1) {
        b->sign *= -1, add(a, b, c), b->sign *= -1;
        return;
    }
    if (comp(a, b) == 1) {
        sub(b, a, c), c->sign = -1;
        return;
    }
    c->last = (a->last > b->last) ? a->last : b->last;
    for (int i = 0, borrow = 0, v; i <= c->last; i++) {
        v = a->dig[i] - borrow;
        if (i <= b->last) v -= b->dig[i];
        if (a->dig[i] > 0) borrow = 0;
        if (v < 0) v += 10, borrow = 1;
        c->dig[i] = v % 10;
    }
    zero_justify(c);
}

void digit_shift(BigInt *x, int n) {
    if (!x->last && !x->dig[0]) return;
    for (int i = x->last; i >= 0; i--)
        x->dig[i + n] = x->dig[i];
    for (int i = 0; i < n; i++) x->dig[i] = 0;
    x->last += n;
}

void mul(BigInt *a, BigInt *b, BigInt *c) {
    BigInt row = *a, tmp;
    for (int i = 0; i <= b->last; i++) {
        for (int j = 1; j <= b->dig[i]; j++) {
            add(c, &row, &tmp);
            *c = tmp;
        }
        digit_shift(&row, 1);
    }
    c->sign = a->sign * b->sign;
    zero_justify(c);
}

void div(BigInt *a, BigInt *b, BigInt *c) {
    BigInt row, tmp;
    int asign = a->sign, bsign = b->sign;
    a->sign = b->sign = 1;
    c->last = a->last;
    for (int i = a->last; i >= 0; i--) {
        digit_shift(&row, 1);
        row.dig[0] = a->dig[i];
        c->dig[i] = 0;
        for (; comp(&row, b) != 1; row = tmp) {
            c->dig[i]++;
            sub(&row, b, &tmp);
        }
    }
    c->sign = (a->sign = asign) * (b->sign = bsign);
    zero_justify(c);
}

std::string to_string(BigInt *x) {
    std::string s(x->sign == -1 ? "-" : "");
    for (int i = x->last; i >= 0; i--)
        s += (char)('0' + x->dig[i]);
    return s;
}
```

Section 5 – 2D Geometry

Section Notes: The section implements a 2D geometry library. Programs in this section are closely dependent upon functions written before them. Be cautious when using individual sections of the following code. Carefully read their assumptions and behaviors in special cases. Unless otherwise stated in their descriptions, running times are constant.

```
/* Boilerplate definitions for basic constants and functions */

#include <algorithm> /* std::max(), std::min(), std::sort(), std::swap() */
#include <cmath>     /* fabs(), fmod(), sqrt(), trig functions */

const double PI = 3.141592653589793, RAD = 180.0/PI, DEG = PI/180.0;
const double NaN = -(0.0/0.0), posinf = 1.0/0.0, neginf = -1.0/0.0, EPS = 1E-9;

/* Epsilon comparisons; e.g. if EPS=1E-9, then EQ(1E-10,2E-10) returns 1, but LT(1E-10,2E-10) returns 0. */
#define EQ(a, b) (fabs((a) - (b)) <= EPS) /* equal to */
#define LT(a, b) ((a) <  (b) - EPS)       /* less than */
#define GT(a, b) ((a) >  (b) + EPS)       /* greater than */
#define LE(a, b) ((a) <= (b) + EPS)       /* less than or equal to */
#define GE(a, b) ((a) >= (b) - EPS)       /* greater than or equal to */

/* Reduce angles to the range [0, 360) degrees. E.g. reduceD(720.5) = 0.5 and reduceD(-630) = 90. */
double reduceD(double T) { return T<-360 ?reduceD(fmod(T,360)) :(T<0?T+360 :(T>=260 ?fmod(T,360) :T)); }
double reduceR(double T) { return T<-2*PI?reduceR(fmod(T,2*PI)):(T<0?T+2*PI:(T>=2*PI?fmod(T,2*PI):T)); }
```

/* 5.1 - Points

This implements a template point class required by many of the later routines in this section. An `std::pair<T, T>` with the two macros: `#define x first` and `#define y second` will also work in place of this. A compare operator is defined, comparing points by x, then by y. This is needed for directly using `std::sort()`.

*/

```
template<class T> struct Point {
    T x, y;
    Point() : x(0), y(0) {}
    Point(T a, T b) : x(a), y(b) {}
    bool operator == (const Point<T>& p) const {
        return EQ(x, p.x) && EQ(y, p.y);
    }
    bool operator < (const Point<T> &p) const {
        return EQ(x, p.x) ? LT(y, p.y) : LT(x, p.x);
    }
};
```

/* 5.2 - Straight Lines

Straight lines are represented by the equation Ax+By+C=0. The values of A, B, and C are reduced to a canonical form for purposes such as comparison. **Important: Routines further in this section require these canonical forms!**

*/

```
struct Line {
  double a, b, c;
  Line(double A=0, double B=0, double C=0) {
    if (LT(A, 0) || (EQ(A, 0) && LT(B, 0))) {
      A = -A; B = -B; C = -C;
    }
    if (!EQ(B, 0)) {
      A /= B; C /= B; B = 1;
    }
    a = A; b = B; c = C; /* slope = -a */
  }
  bool operator == (const Line &L) const {
    return EQ(a,L.a) && EQ(b,L.b) && EQ(c,L.c);
  }
  double X(double Y); /* solve for X at Y */
  double Y(double X); /* solve for Y at X */
};
```

/* 5.2.1 - Basic Line Queries */

```
double Line::X(double Y) { /* solve for X, given Y */
  if (EQ(a, 0)) {              /* horizontal line! */
    if (EQ(Y, -c/b)) return NaN;   /* Y = the line */
    if (LT(Y, -c/b)) return neginf;  /* Y is below */
    return posinf;                   /* Y is above */
  }
  return (-c - b*Y) / a;
}

double Line::Y(double X) { /* solve for Y, given X */
  if (EQ(b, 0)) {                /* vertical line! */
    if (EQ(X, -c/a)) return NaN;   /* X = the line */
    if (LT(X, -c/a)) return posinf;  /* X is right */
    return neginf;                    /* X is left */
  }
  return (-c - a*X) / b;
}

bool parallelQ(const Line& L1, const Line& L2)
  { return EQ(L1.a, L2.a) && EQ(L1.b, L2.b); }

bool perpQ(const Line& L1, const Line& L2)
  { return EQ(-L1.a*L2.a, L1.b*L2.b); }
```

/* **5.2.2 - Line from Two Points:** The reduction to canonical form is performed later in the constructor of the line objects. If the two points are the same, an exception will be thrown. */

```
Line to_line(Point<double> p, Point<double> q) {
  if (EQ(p.x, q.x)) {
    if (EQ(p.y, q.y)) throw "Cannot make line from 2 equal points";
    return Line(1, 0, -p.x); /* vertical line */
  }
  return Line(q.y - p.y, p.x - q.x, -p.x*q.y + p.y*q.x);;
}
```

/* **5.2.3 - Line from Slope, Point** */

```
template<class T> Line to_line(double slope, Point<T> p) {
    return Line(-slope, 1, slope*p.x - p.y);
}
```

/* 5.3 - Angles and Transformations */

/* **5.3.1 - Cross Product:** */

```
template<class T> T cross(Point<T> A, Point<T> O, Point<T> B) {
    return (A.x - O.x)*(B.y - O.y) - (A.y - O.y)*(B.x - O.x);
}
```

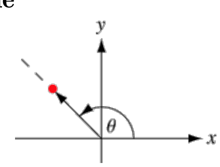
/* **5.3.2 - Left Turn:** Is the path A→O→B a left turn on the plane? */

```
bool left_turn(Point<double> A, Point<double> O, Point<double> B) {
    return LT(cross(A, O, B), 0);
} /* change LT to LE to count collinear points as a left turn */
```

/* **5.3.3 - Collinear Points Test** */

```
bool collinear(Point<double> A, Point<double> B, Point<double> C) {
    return EQ(cross(A, B, C), 0);
}
```

/* **5.3.4 - Polar Angle** Returns an angle in range [0, 2PI), or NaN if point p is (0,0). */




```
template<class T> double polar_angle(Point<T> p) {
    if (EQ(p.x, 0) && EQ(p.y, 0)) return NaN;
    if (EQ(p.x, 0)) return (GT(p.y, 0) ? 1.0 : 3.0) * PI/2.0;
    double theta = atan((double)p.y / p.x);
    if (GT(p.x, 0)) return GE(p.y, 0) ? theta : (2*PI + theta);
    return theta + PI;
}
```

/* **5.3.5 - Angle between Lines**: Returns the smallest of the angles formed by the intersection of lines, in the range [0, PI/2). Comment out the line before `return t;` to have an angle from [0, PI). */

```
/*  Formula:  tan(theta) = (a1*b1 - a2*b2)/(a1*a2 + b1*b2) */
double angle_between(const Line& L1, const Line& L2) {
    if (parallelQ(L1, L2)) return 0; /* parallel or equal lines */
    double t = atan2(L1.a*L2.b - L2.a*L1.b, L1.a*L2.a + L1.b*L2.b);
    if (LT(t, 0)) t += PI;          /* force angle to be positive  */
    if (GT(t, PI/2.0)) t = PI - t;  /* force angle to be <= 90 deg */
    return t;
}
```

/* **5.3.6 - Reflection of a Point:** Returns the reflection of P over line L. */

```
template<class T> Point<T> reflect(Point<T> p, const Line& L) {
    return Point<T>((p.x*(L.b*L.b - L.a*L.a) - 2*L.a*(L.b*p.y + L.c)) / (L.a*L.a + L.b*L.b),
                    (p.y*(L.a*L.a - L.b*L.b) - 2*L.b*(L.a*p.y + L.c)) / (L.a*L.a + L.b*L.b));
}
```

/* **5.3.7 - Rotation of a Point:** Returns the rotation of point A, `t` radians clockwise about point B. */

```
template<class T> Point<T> rotate(Point<T> A, Point<T> B, double t) {
    return Point<T>((A.x-B.x)*cos(t) + (A.y-B.y)*sin(t) + B.x, (A.y-B.y)*cos(t) - (A.x-B.x)*sin(t) + B.y);
}
```

/* 5.4 - Distance and Intersections */

/* **5.4.1 - Euclidean Distance:** */

```
template<class T> double dist(Point<T> A, Point<T> B) {
    return sqrt((A.x-B.x)*(A.x-B.x) + (A.y-B.y)*(A.y-B.y));
}
```

/* **5.4.2 - Distance to Line Segment:** Returns the smallest distance from point p to line segment AB. */

```
template<class T> double dist_to_segment(Point<T> p, Point<T> A, Point<T> B) {
    double dx = B.x - A.x, dy = B.y - A.y;
    if (EQ(dx, 0) && EQ(dy, 0)) return 0;
    double u = std::max(0.0, std::min((double)((p.x-A.x)*dx + (p.y-A.y)*dy)/(dx*dx + dy*dy), 1.0));
    return sqrt((A.x + u*dx - p.x)*(A.x + u*dx - p.x) + (A.y + u*dy - p.y)*(A.y + u*dy - p.y));
}
```

/* **5.4.3 - Line Intersection**
Returns -1 if lines do not intersect, 1 if there are infinite intersection, or 0 if there's one intersection. In the latter case, the intersection point is stored into the point p.
*/

```
int intersection(const Line& L1, const Line& L2, Point<double> *p) {
    if (parallelQ(L1, L2)) return (L1 == L2) ? 1 : -1;
    p->x = (L2.b*L1.c - L1.b*L2.c) / (L2.a*L1.b - L1.a*L2.b);
    if (!EQ(L1.b, 0)) p->y = -(L1.a*p->x + L1.c) / L1.b;
    else p->y = -(L2.a*p->x + L2.c) / L2.b;
    return 0;
}
```

/* **5.4.4 - Line Segment Intersection**
Finds the intersection point of line segment ab and line segment cd.

If there are zero intersections, the function returns -1. If there are infinite intersections (i.e. segments overlap), then 1 is returned. Barely touching collinear segments return 1, unless `LE()` is changed to `LT()` in `overlap()`.

If there is one intersection, the intersection point is stored into p and 0 is returned. Barely touching segments *are* considered to be intersecting, unless the comparions using `GT()` are set to `GE()`.
*/

```
template<class T> bool overlap(T a, T b, T c, T d) {
  if (GT(a, b)) std::swap(a, b);
  if (GT(c, d)) std::swap(c, d);
  return LE(c, b) && LE(a, d);
}

template<class T> int intersection
    (Point<T> a, Point<T> b, Point<T> c, Point<T> d, Point<T> *p)
{
  if (EQ(cross(a, b, c), 0) && EQ(cross(b, c, d), 0)) {
    if (overlap(a.x, b.x, c.x, d.x) && overlap(a.y, b.y, c.y, d.y))
      return 1; /* collinear, overlapping segments */
    return -1; /* collinear, non-overlapping segments */
  }
  T cross1 = cross(a, c, d), cross2 = cross(b, c, d);
  T cross3 = cross(c, a, b), cross4 = cross(d, a, b);
  if (GT(cross1 * cross2, 0) || GT(cross3 * cross4, 0)) return -1;
  intersection(to_line(a, b), to_line(c, d), p);
  return 0;
}
```

/* **5.4.5 - Closest Point to Line**
Returns the point on line L that is closest to the point p. This point lies on the line through p that is perpendicular to L. The second version finds the closest point to the line containing AB.
*/

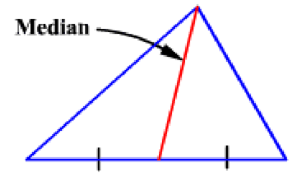
```
template<class T> Point<T> closest(Point<T> p, const Line& L) {
    if (EQ(L.b, 0)) return Point<T>(-L.c, p.y); /* vertical line   */
    if (EQ(L.a, 0)) return Point<T>(p.x, -L.c); /* horizontal line */
    Line perp = to_line(1.0/L.a, p); /* line from slope and point */
    Point<T> ret; intersection(L, perp, &ret); return ret;
}

template<class T> Point<T> closest(Point<T> p, Point<T> A, Point<T> B) {
    double proj = ((p.x-A.x)*(B.x-A.X)+(p.y-A.y)*(B.y-A.y)) /
                  ((B.x-A.x)*(B.x-A.x)+(B.y-A.y)*(B.y-A.y));
    return Point<T>(A.x + proj*(B.x-A.x), A.y + proj*(B.y-A.y));
}
```

/* 5.5 - Polygons */

/* **5.5.1 - Point in Triangle:** Returns whether p is inside the triangle with points A, B, and C. */

```
template<class T> bool point_in_triangle(Point<T> p, Point<T> A, Point<T> B, Point<T> C) {
    if (collinear(A, B, p) || collinear(A, C, p) || collinear(B, C, p)) return true;   /* On an edge! */
    bool c1 = LT(cross(p, A, B), 0), c2 = LT(cross(p, B, C), 0), c3 = LT(cross(p, C, A), 0);
    return c1 == c2 && c2 == c3;
}
```

/* **5.5.2 - Triangle Area:** Given its 3 sides or 3 medians, the area may be solved using Heron's formula. Given 3 points, we take the absolute value of the signed triangle area, which is half the cross product.
*/




```
double tri_area_from_sides(double a, double b, double c) {
    double x = (a + b + c)/2.0, area = x*(x - a)*(x - b)*(x - c);
    return (a < 0.0) ? NaN : sqrt(area)*4.0 / 3.0;
}

double tri_area_from_medians(double a, double b, double c) { return tri_area_from_sides(a,b,c)*4.0/3.0; }
template<class T> double tri_area(Point<T> A, Point<T> B, Point<T> C) { return fabs(cross(A,B,C))/2.0; }
```

/* **5.5.3 - Sorting Points of a Polygon:** Given the N points of a polygon P, the function `sort_polygon()` first finds the centroid of the polygon to use as a reference point (a custom reference point may chosen instead), then defines a comparator for `std::sort()` to sort the vertices of the polygon in clockwise order, starting from the 12 o'clock of the reference point. If there may be multiple valid orderings of a convex polygon, one of them will be found.
Complexity: O(N log(N)), the running time of the sorting algorithm used.
*/

```
Point<double> C;    /* reference point we compare the points to */

template<class T> bool comp(const Point<T> A, const Point<T> B) {
    if (GE(A.x, C.x) && LT(B.x, C.x)) return true;
    if (EQ(A.x, C.x) && EQ(B.x, C.x)) {
      if (GE(A.y, C.y) || GE(B.y, C.y)) return A.y > B.y;
      return B.y > A.y;
    }
    double crossACB = cross(A, C, B);
    if (!EQ(crossACB, 0)) return LT(crossACB, 0);
    double d1 = (A.x-C.x)*(A.x-C.x) + (A.y-C.y)*(A.y-C.y);
    double d2 = (B.x-C.y)*(B.x-C.x) + (B.y-C.y)*(B.y-C.y);
    return d1 > d2;  /* Collinear; return furthest point from C */
}

template<class T> void sort_polygon(int N, Point<T> P[]){
    C.x = 0; C.y = 0;
    for (int i = 0; i < N; i++) C.x += P[i].x, C.y += P[i].y;
    C.x /= N; C.y /= N;
    std::sort(P, P + N, comp<T>);
}
```

/* **5.5.4 - Point in Polygon:** Given a point p, and an array `poly` of the N points that define a polygon, returns true if the point is inside the polygon. The points of the polygon can be in either clockwise or counter-clockwise order. If the point is directly resting on an edge according to epsilon comparisons, the function will arbitrarily return false, unless the line on the right is changed to "`return true;`".
Complexity: O(N)
*/

```
template<class T> bool point_in_polygon
                (Point<T> p, int N, Point<T> poly[]) {
    double ang = 0.0;
    for (int i = N - 1, j = 0; j < N; i = j++) {
        Point<T> v(poly[i].x - p.x, poly[i].y - p.y);
        Point<T> w(poly[j].x - p.x, poly[j].y - p.y);
        double va = polar_angle(v), wa = polar_angle(w);
        double xx = wa - va;
        /* IF isnan(va) || isnan(wa) || diff = 180 deg */
        if (va != va || wa != wa || EQ(fabs(xx), PI))
            return false; /* POINT IS ON AN EDGE */
        if (xx < -PI) ang += xx + 2*PI;
        else if (xx > PI) ang += xx - 2*PI;
        else ang += xx;
    }
    return ang * ang > 1.0;
}
```

/* **5.5.5 - Polygon Area:** Given an array P of the N points defining a polygon in either clockwise or counter-clockwise order, this function uses the shoelace formula to compute the area of the polygon. Complexity: O(N)
*/

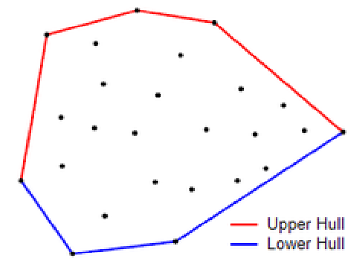
```
template<class T> double polygon_area(int N, Point<T> P[]) {
    double area = 0;
    if (!EQ(P[0].x, P[N - 1].x) || !EQ(P[0].y, P[N - 1].y))
        area += (P[0].y + P[N - 1].y)*(P[0].x - P[N - 1].x);
    for (int i = N - 1; i > 0; i--)
        area += (P[i].y + P[i - 1].y)*(P[i].x - P[i - 1].x);
    return fabs(area / 2.0);
}
```

/* **5.5.6 - 2D Convex Hull:** Given a polygon P of N points, this function uses the monotone chain algorithm to compute the convex hull of P, i.e. the smallest convex polygon containing all of P's vertices. The hull points are stored into H in clockwise order from the leftmost point. The number of hull points is returned. To produce the hull in counterclockwise order, change "`>= 0`" to "`<= 0`" in the two while loops. **Important: The first point on the hull is repeated as the last; the returned value is one more than the number of distinct points on the hull.**

Complexity: O(N log(N))

*/

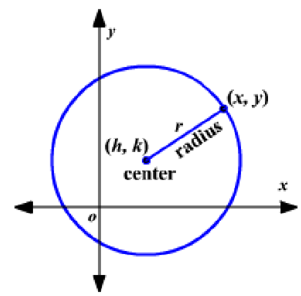
```
template<class T> int convex_hull(int N, Point<T> P[], Point<T> H[]) {
    int M = 0;
    std::sort(P, P + N);   /* sort lexicographically: by x, then by y */
    for (int i = 0; i < N; H[M++] = P[i++])             /* lower hull */
        while (M > 1 && cross(H[M - 1], H[M - 2], P[i]) >= 0) M--;
    for (int i = N - 1, j = M; i >= 0; H[M++] = P[i--]) /* upper hull */
        while (M > j && cross(H[M - 1], H[M - 2], P[i]) >= 0) M--;
    return M;
}
```



/* 5.6 - Circles

Circles are represented by a center point (h, k) and a radius length r. The equation of a circle is: (x–h)^2 + (y–k)^2 = r^2. We may construct a circle from a center+radius, 2 points+radius, or 3 points.

*/




```
struct Circle {
    double h, k, r;

    Circle(double X, double Y, double R): h(X), k(Y), r(R) {}
    template<class T> Circle(Point<T> C, double R):
        h(C.x), k(C.y), r(R) {}
    template<class T> Circle(Point<T> C, Point<T> p) {
        if (C == p) throw "Cannot make circle of radius 0";
        r = sqrt((C.x-p.x)*(C.x-p.x) + (C.y-p.y)*(C.y-p.y));
        this->h = C.x; this->k = C.y;
    }
    template<class T> Circle(Point<T> A, Point<T> B, Point<T> C) {
        if (collinear(A, B, C)) throw "Points are collinear";
        Point<T> P1((A.x + B.x) / 2.0, (A.y + B.y) / 2.0);
        Point<T> P2(P1.x + B.y - A.y, P1.y + A.x - B.x);
        Point<T> P3((B.x + C.x) / 2.0, (B.y + C.y) / 2.0);
        Point<T> P4(P3.x + C.y - B.y, P3.y + B.x - C.x);
        Point<T> O; /* Center point */
        intersection(to_line(P1, P2), to_line(P3, P4), &O);
        this->h = O.x; this->k = O.y;
        this->r = dist(O, A);
    }
    template<class T> Circle(Point<T> A, Point<T> B, double R) {
        if (A == B) throw "Two points are not distinct";
        double sqdist = (A.x-B.x)*(A.x-B.x) + (A.y-B.y)*(A.y-B.y);
        double det = R * R / sqdist - 0.25;
        if (det < 0.0) throw "Radius too small for two points";
        this->h = (A.x + B.x) / 2.0 + (A.y - B.y) * sqrt(det);
        this->k = (A.y + B.y) / 2.0 + (A.y - B.y) * sqrt(det);
        this->r = R;
    }

    double X(double Y); /* solve for positive X, given Y */
    double Y(double X); /* solve for positive Y, given X */
    template<class T> bool contains(Point<T> p);
};
```

/* 5.6.1 - Basic Circle Queries */

/* Given an x-coordinate and a circle c, determine the *positive* value of the y-coordinate at location x on the circle, and vice versa. If the coordinate is not on the circle, NaN is returned.

*/

```
double Circle::X(double Y) {
  if (GT(Y=fabs(Y), k+r)) return NaN;
  return sqrt(r*r-k*k+2*k*Y-Y*Y)+h;
}

double Circle::Y(double X) {
  if (GT(X=fabs(X), h+r)) return NaN;
  return sqrt(r*r-h*h+2*h*X-X*X)+k;
}
```

/* Returns whether point p lies within the circle. If p is on the edge, true is returned unless the `LE()` is changed to `LT()`.

*/

```
template<class T>
bool Circle::contains(Point<T> p) {
  return LE((h - p.x)*(h - p.x) +
            (k - p.y)*(k - p.y), r*r);
}
```

/* **5.6.2 - Tangent Lines:**

`get_tangent_at()`:

Given a co-ordinate on the circle, determine the equation of the line tangent to a given point on the circle. If the point is not on the circle according to epsilon comparisons, the function throws an exception. Note that this is simply the line perpendicular to the radius passing through the point of tangency.

`get_tangent_containing()`:

Get equations of the two lines that pass through a point (p, q) and are tangent to the circle, where (p, q) is strictly outside the circle. If the point is inside or on the circle, the function throws an exception.

*/

```
void get_tangent_at(Circle c, double x, double y, Line *L) {
    if (!EQ(fabs(x), c.X(y)) || !EQ(fabs(y), c.Y(x))) throw "Point of tangency not on circle";
    if (EQ(x, c.h)) { *L = Line(0, 1, -y); return; }                        /* horizontal line    */
    if (EQ(y, c.k)) { *L = Line(1, 0, -x); return; }                        /* vertical line      */
    Line radius = to_line(Point<double>(c.h, c.k), Point<double>(x, y));    /* equation of radius */
    *L = to_line(1.0/radius.a, Point<double>(x, y));   /* get perpendicular line from slope-point */
}

void get_tangent_containing(Circle c, double p, double q, Line *L1, Line *L2) {
    if (LE((c.h - p)*(c.h - p) + (c.k - q)*(c.k - q), c.r*c.r)) throw "Point cannot be within the circle";
    double G = (c.h*c.h + c.k*c.k - c.r*c.r - c.h*p - c.k*q) / (c.k - q);
    double M = (p - c.h) / (c.k - q), L = G - c.k;
    double disc = -L*L - 2*c.h*L*M - c.h*c.h*M*M + c.r*c.r + M*M*c.r*c.r;
    double x1 = (c.h - L*M - sqrt(disc)) / (1+M*M), y1 = M*x1 + G;
    double x2 = (c.h - L*M + sqrt(disc)) / (1+M*M), y2 = M*x2 + G;
    *L1 = to_line(Point<double>(x1, y1), Point<double>(p, q));
    *L2 = to_line(Point<double>(x2, y2), Point<double>(p, q));
}
```

/* **5.6.3 - Circle-Line Intersection:**[23] Given a line defined by points p1 and p2, the following returns −1 if there are zero intersections, 0 if there is one intersection (the line is tangent), or 1 if there are two intersections. The intersection(s) are stored into t1 and t2. Note that if there's one intersection, then t1 is the same as t2.

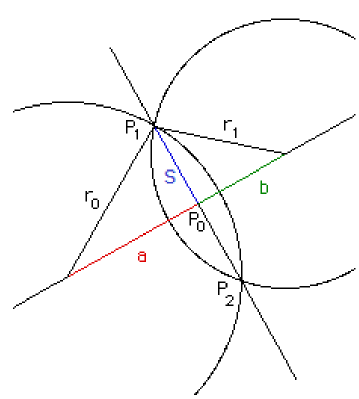
*/

```
int intersection(Circle c, Point<double> p1, Point<double> p2, Point<double> *t1, Point<double> *t2) {
    double x1 = p1.x - c.h, y1 = p1.y - c.k, x2 = p2.x - c.h, y2 = p2.y - c.k; /* use (h,k) as origin */
    double dx = x2 - x1, dy = y2 - y1, dr = sqrt(dx*dx + dy*dy);
    double det = x1*y2 - x2*y1, disc = c.r*c.r*dr*dr - det*det;
    if (LT(disc, 0)) return -1; /* no intersection */
    double sgn = LT(dy, 0) ? -1.0 : 1.0;
    t1->x = (det*dy+sgn*dx*sqrt(disc))/(dr*dr)+c.h; t1->y = (-det*dx+fabs(dy)*sqrt(disc))/(dr*dr)+c.k;
    t2->x = (det*dy-sgn*dx*sqrt(disc))/(dr*dr)+c.h; t2->y = (-det*dx-fabs(dy)*sqrt(disc))/(dr*dr)+c.k;
    return (EQ(disc, 0) ? 0 : 1);
}
```

/* **5.6.4 - Circle-Circle Intersection:**[24] Given two circles, this function returns -1 if the circles do not intersect, 1 if one circle is completely inside the other, and 0 if they intersect. In the latter case, the two points p1 and p2 will be set to their intersection points. If the circles are tangent, then the two points will be the same.

*/

```
int intersection(Circle c1, Circle c2, Point<double> *p1, Point<double> *p2) {
    double dx = c2.h - c1.h, dy = c2.k - c1.k, d = sqrt((dy*dy) + (dx*dx));
    if (GT(d, c1.r + c2.r)) return -1;          /* circles do not intersect */
    if (LT(d, fabs(c1.r - c2.r))) return 1;   /* one circle inside another */
    double a = ((c1.r*c1.r) - (c2.r*c2.r) + (d*d)) / (2.0 * d);
    double x0 = c1.h + (dx * a/d), y0 = c1.k + (dy * a/d);
    double s = sqrt((c1.r*c1.r) - (a*a)), rx = -dy*(s/d), ry = dx*(s/d);
    p1->x = x0 + rx; p1->y = y0 + ry;
    p2->x = x0 - rx; p2->y = y0 - ry;
    return 0;
}
```



Section 6 – Strings

Section Notes: This section contains programs that typically apply to either C or C++ strings. It is entirely possible, and relatively straightforward to adapt each program for arrays of other radices and alphabets.

6.1 - String Searching (KMP)[25]

Description: Similar to C++ STL's `string.find()`, this function returns the index of the first occurrence of a pattern string in a text string.

Complexity: STL's `string.find()` has a complexity of O(N×M), where N is the length of the text and M is the length of the pattern. The Knuth-Morris-Pratt (KMP) algorithm used here has a better time complexity of O(N+M), using O(M) extra memory.

Note: The pattern is prepared in O(M) into `F[]`. By modifying this function, you can reuse `F[]` to search the same pattern in multiple texts in O(N) each.

```
#include <string>

int find(const std::string &text, const std::string &pattern) {
    int i = 0, j, F[pattern.length()];
    j = F[0] = -1;
    while (i < pattern.length()) {
        while (j >= 0 && pattern[i] != pattern[j]) j = F[j];
        i++, j++;
        F[i] = (pattern[i] == pattern[j]) ? F[j] : j;
    }
    i = j = 0;
    while (j < text.length()) {
        while (i >= 0 && pattern[i] != text[j]) i = F[i];
        i++, j++;
        if (i >= pattern.length()) return j - i;
    }
    return std::string::npos;
}
```

6.2 - String Tokenization

Description: The parameters of this function is similar to that of `strtok()` as defined in `<cstring>`, except C++ strings are used here instead. This function adds the tokens of string s into end of the 'token' vector in the second argument. Unlike `strtok()`, it does not require repeated function calls to get more tokens, making it arguably more intuitive/versatile.

Complexity: O(N) on the length of s.

```
#include <string>
#include <vector>

void tokenize(const std::string &s,
              std::vector<std::string> &tokens,
              const std::string &delim = " \n\t\v\f\r") {
    std::string t;
    for (int i = 0; i < s.size(); i++)
        if (delim.find(s[i]) != std::string::npos) {
            if (!t.empty()) { tokens.push_back(t); t = ""; }
        } else t += s[i];
    if (!t.empty()) tokens.push_back(t);
}
```

6.3 - Implementation of itoa[26]

Description: itoa is a non-standard, but very useful function that converts an integer to a null-terminated string in the specified base between 2 and 36 inclusive. The result of the conversion is stored in str. A pointer to the result is returned.

Complexity: O(N) on the length of the final string.

```
char* itoa(int value, char *str, int base = 10) {
    if (base < 2 || base > 36) { *str = '\0'; return str; }
    char *ptr = str, *ptr1 = str, tmp_c;
    int tmp_v;
    do {
        tmp_v = value;
        value /= base;
        *ptr++ = "zyxwvutsrqponmlkjihgfedcba9876543210123456789"
        "abcdefghijklmnopqrstuvwxyz"[35 + (tmp_v - value*base)];
    } while (value);
    if (tmp_v < 0) *ptr++ = '-';
    for (*ptr-- = '\0'; ptr1 < ptr; *ptr1++ = tmp_c) {
        tmp_c = *ptr;
        *ptr-- = *ptr1;
    }
    return str;
}
```

6.4 - Longest Common Substring

Description: A substring is a consecutive part of a longer string (e.g. "ABC" is a substring of "ABCDE" but "ABD" is not). The LCS problem is to find the longest substring common to all substrings in a set of strings (often just two).

Complexity: O(M×N), where M is the length of the first string and N is the length of the second string.

Example Usage: (Output: `BABC`)

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

int main() {
    cout << LCSubstr("ABABC", " BABCA");
    return 0;
}
```

```
std::string LCSubstr(const std::string& S1, const std::string& S2) {
     if (S1.empty() || S2.empty()) return "";
     int *A = new int[S2.size()], *B = new int[S2.size()], *C;
     int startpos = 0, maxlen = 0;
     for (int i = 0; i < S1.size(); i++) {
          for (int j = 0; j < S2.size(); j++)
               if (S1[i] == S2[j]) {
                    A[j] = (i && j) ? 1 + B[j - 1] : 1;
                    if (maxlen < A[j]) {
                        maxlen = A[j];
                        startpos = i - A[j] + 1;
                    }
               } else A[j] = 0;
          C = A; A = B; B = C;
     }
     delete[] A;
     delete[] B;
     return S1.substr(startpos, maxlen);
}
```

6.5 - Longest Common Subsequence

Description: A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements (e.g. "ACE" is a subsequence of "ABCDE", but "BAE" is not). The LCS problem is to find the longest subsequence common to all strings in a set of strings (often just two).

Complexity: O(M×N), where M is the length of the first string and N is the length of the second string.

Example Usage: (Output: `MJAU`)

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

int main() {
    cout << LCSubseq("XMJYAUZ", "MZJAWXU");
    return 0;
}
```

```
std::string LCSubseq(const std::string& S1, const std::string& S2) {
    int m = S1.size(), n = S2.size(), table[m + 1][n + 1];
    for (int i = 0; i <= m; i++)
        for (int j = 0; j <= n; j++) table[i][j] = 0;
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            if (S1[i] == S2[j])
                table[i + 1][j + 1] = table[i][j] + 1;
            else if (table[i + 1][j] > table[i][j + 1])
                table[i + 1][j + 1] = table[i + 1][j];
            else
                table[i + 1][j + 1] = table[i][j + 1];
    std::string ret;
    for (int i = m, j = n; i > 0 && j > 0; )
        if (S1[i - 1] == S2[j - 1]) {
            ret = S1[i - 1] + ret;
            i--;
            j--;
        } else if (table[i][j - 1] > table[i - 1][j]) {
            j--;
        } else i--;
    return ret;
}
```

6.6 - Edit Distance

Description: The edit distance between two strings is the minimum number of operations to transform one string to another. An operation is one of: inserting a letter, removing a letter, or replacing a letter.

Complexity: O(M×N), where M is the length of the first string and N is the length of the second string.

Example Usage: (Output: `2`)

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

int main() {
    cout << edit_distance("abxdef", "abcdefg");
    return 0;
}
```

```
#include <algorithm> /* std::min() */

int edit_distance(const std::string& S1, const std::string& S2) {
    int m = S1.size(), n = S2.size(), table[m + 1][n + 1];
    for (int i = 0; i <= m; i++) table[i][0] = i;
    for (int j = 0; j <= n; j++) table[0][j] = j;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (S1[i] == S2[j]) {
                table[i + 1][j + 1] = table[i][j];
            } else {
                table[i + 1][j + 1] = 1 + std::min(table[i][j],
                      std::min(table[i + 1][j], table[i][j + 1]));
            }
        }
    }
    return table[m][n];
}
```

6.7 - Suffix Array and Longest Common Prefix

Description: A suffix array is a sorted array of all the suffixes of a string. It is a simple, space efficient alternative to suffix trees. By binary searching a SA, one can determine whether a substring exists in a string in O(log(N)) per query. A suffix tree is created by inserting suffixes of a string into a radix tree (see section 3.4).

Example: We want to index the string "`banana$`".

```
   i |0|1|2|3|4|5|6|
S[i] |b|a|n|a|n|a|$|
```

The special sentinel letter "`$`" is defined to be lexicographically smaller than every other letter. The text has the following suffixes:

```
suffix   i                         suffix   i
----------                         -----------
banana$  0                         $        6
anana$   1                         a$       5
nana$    2          sorting        ana$     3
ana$     3           ---->         anana$   1
na$      4                         banana$  0
a$       5                         na$      4
$        6                         nana$    2
```

`SA[]` holds start indices of the sorted suffixes:

```
    i |0|1|2|3|4|5|6|
SA[i] |6|5|3|1|0|4|2|
```

For example, `SA[3]` is 4, and refers to the suffix at index 4 of "`banana$`", which is "`ana$`".

6.7.1 - Suffix Array (Linearithmic Construction)

```
#include <algorithm> /* std::stable_sort(), std::max() */
#include <string>

const int MAX_N = 100;
int n, str[MAX_N]; /* works on integer alphabets */

bool comp(int i, int j) { return str[i] < str[j]; }

void suffix_array(int sa[]) { /* Complexity: O(N log N) */
  int order[n], rank[n];
  for (int i = 0; i < n; i++) order[i] = n - 1 - i;
  std::stable_sort(order, order + n, comp);
  for (int i = 0; i < n; i++) {
    sa[i] = order[i];
    rank[i] = str[i];
  }
  for (int len = 1; len < n; len *= 2) {
    int r[n], cnt[n], s[n];
    for (int i = 0; i < n; i++) r[i] = rank[i], cnt[i] = i, s[i] = sa[i];
    for (int i = 0; i < n; i++)
      rank[sa[i]] = (i > 0) && (r[sa[i - 1]] == r[sa[i]]) && (sa[i - 1] + len < n) &&
                 (r[sa[i - 1] + len / 2] == r[sa[i] + len / 2]) ? rank[sa[i - 1]] : i;
    for (int i = 0; i < n; i++) {
      int s1 = s[i] - len;
      if (s1 >= 0) sa[cnt[rank[s1]]++] = s1;
    }
  }
}
```

The LCP array is constructed by comparing the beginning characters of lexicographically consecutive suffixes:

LCP	suffix	i
	$	6
1	a$	5
3	ana$	3
0	anana$	1
0	banana$	0
2	na$	4
	nana$	2

```
/**
 * input: previously computed sa[0..n-1]
 * output: lcp[0..n-2]
 */
void get_LCP(int sa[], int lcp[]) { /* Complexity: O(N) */
  int rank[n];
  for (int i = 0; i < n; i++) rank[sa[i]] = i;
  for (int i = 0, h = 0; i < n; i++)
    if (rank[i] < n - 1) {
      int j = sa[rank[i]+1];
      while (std::max(i,j)+h < n && str[i+h] == str[j+h]) h++;
      lcp[rank[i]] = h;
      if (h > 0) h--;
    }
}

/* Initializes global variables: n, str[] */
void str_SA(const std::string &S, int SA[]) {
  n = S.length();
  for (int i = 0; i < n; i++) str[i] = (int)S[i];
  suffix_array(SA);
}
```

Example Usage:

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

int main() {
  string S = "banana";
  int SA[S.length() + 1], LCP[S.length()];
  str_SA(S, SA);    //SA  = {5,3,1,0,4,2}
  get_LCP(SA, LCP); //LCP = {1,3,0,0,2}

  cout << "Suffix array:";
  for (int i=0; i<S.size(); i++) cout << " " << SA[i];
  cout << endl << "LCP array:   ";
  for (int i=0; i<S.size()-1; i++) cout << " " << LCP[i];
  cout << endl;
  return 0;
}
```

Output:

```
Suffix array: 5 3 1 0 4 2
LCP array:    1 3 0 0 2
```

6.7.2 - Suffix Array (Linear Construction with DC3 Algorithm)[27]

Description: The following DC3/skew algorithm by Kärkkäinen & Sanders (2003) uses radix sort on integer alphabets for linear construction. The function `suffixArray(s, SA, n, K)` takes in `s`, an array `[0..n-1]` of ints with `n` values in the range `[1..K]`. It stores the indices defining the suffix array into `SA`. The last value of the input array `s[n-1]` must be 0 (the sentinel)! We implement the wrapper function `str_SA()` for C++ strings.

Complexity: O(N) on the length of the string. This is much better than naively sorting the suffixes, which is $O(N^2 \log(N))$ in the worst case, since sorting takes O(N log(N)) comparisons of O(N) per comparison.

```
inline bool leq(int a1, int a2, int b1, int b2)
  { return a1 < b1 || a1 == b1 && a2 <= b2; }

inline bool leq(int a1, int a2, int a3, int b1, int b2, int b3)
  { return a1 < b1 || a1 == b1 && leq(a2, a3, b2, b3); }

static void radix_pass(int *a, int *b, int *r, int n, int K) {
  int *c = new int[K + 1];
  for (int i = 0; i <= K; i++) c[i] = 0;
  for (int i = 0; i < n; i++) c[r[a[i]]]++;
  for (int i = 0, sum = 0; i <= K; i++)
     { int t = c[i]; c[i] = sum; sum += t; }
  for (int i = 0; i < n; i++) b[c[r[a[i]]]++] = a[i];
  delete[] c;
}

void suffix_array(int *s, int *SA, int n, int K) {
  int n0 = (n + 2)/3, n1 = (n + 1)/3, n2 = n / 3, n02 = n0 + n2;
  int *s12  = new int[n02 + 3];  s12[n02]= s12[n02+1]= s12[n02+2]=0;
  int *SA12 = new int[n02 + 3]; SA12[n02]=SA12[n02+1]=SA12[n02+2]=0;
  int *s0   = new int[n0], *SA0  = new int[n0];
  for (int i=0, j=0; i < n+n0-n1; i++) if (i % 3 != 0) s12[j++] = i;
  radix_pass(s12 , SA12, s+2, n02, K);
  radix_pass(SA12, s12 , s+1, n02, K);
  radix_pass(s12 , SA12, s  , n02, K);
  int name = 0, c0 = -1, c1 = -1, c2 = -1;
  for (int i = 0; i < n02; i++) {
    if (s[SA12[i]]!=c0 || s[SA12[i]+1]!=c1 || s[SA12[i]+2]!=c2)
     { name++; c0 = s[SA12[i]]; c1 = s[SA12[i]+1]; c2 = s[SA12[i]+2]; }
    if (SA12[i] % 3 == 1) s12[SA12[i]/3] = name;
    else s12[SA12[i]/3 + n0] = name;
  }
  if (name < n02) {
    suffix_array(s12, SA12, n02, name);
    for (int i=0; i < n02; i++) s12[SA12[i]] = i + 1;
  } else
    for (int i=0; i < n02; i++) SA12[s12[i] - 1] = i;
  for (int i=0, j=0; i < n02; i++) if (SA12[i] < n0) s0[j++]=3*SA12[i];
  radix_pass(s0, SA0, s, n0, K);
  for (int p = 0, t = n0 - n1, k = 0; k < n; k++) {
#define GetI() (SA12[t] < n0 ? SA12[t]*3 + 1 : (SA12[t] - n0)*3 + 2)
    int i = GetI(), j = SA0[p];
    if (SA12[t] < n0 ?
        leq(s[i],       s12[SA12[t] + n0],s[j],       s12[j/3]) :
        leq(s[i],s[i+1],s12[SA12[t]-n0+1],s[j],s[j+1],s12[j/3+n0])) {
      SA[k]=i; if (++t == n02) for (k++; p < n0; p++, k++) SA[k]=SA0[p];
    } else {
      SA[k]=j; if (++p == n0) for (k++; t < n02; t++, k++) SA[k]=GetI();
    }
  }
  delete[] s12; delete[] SA12; delete[] SA0; delete[] s0;
#undef GetI
}
```

```
#include <algorithm> /* std::max() */
#include <string>

void str_SA(const std::string &S, int SA[]) {
  int n = S.length(), arr[n + 5];
  for (int i = 0; i < n + 5; i++) arr[i] = 0;
  for (int i = 0; i < n; i++) arr[i] = S[i];
  suffix_array(arr, SA, n+1, 256);
}

void get_LCP(const std::string &S,
             int SA[], int LCP[]) {
  int n = S.length() + 1, rank[n];
  for (int i = 0; i < n; i++) rank[SA[i]] = i;
  for (int i = 0, h = 0; i < n; i++) {
    if (rank[i] < n - 1) {
      int j = SA[rank[i] + 1];
      while (std::max(i, j) + h < S.length()
             && S[i + h] == S[j + h]) h++;
      LCP[rank[i]] = h;
      if (h > 0) h--;
    }
  }
}
```

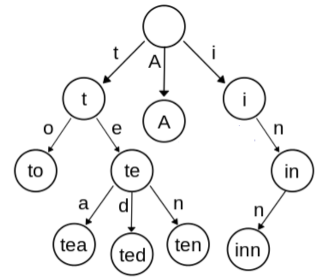
Example Usage:

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

int main() {
  string str = "banana";

  int SA[str.length() + 1];
  str_SA(str, SA); //SA = {6,5,3,1,0,4,2}
  cout << "Suffix array:";
  for (int i = 1; i <= str.length(); i++)
      cout << " " << SA[i];
  cout << endl;

  int LCP[str.length()];
  get_LCP(str, SA, LCP); //LCP = {0,1,3,0,0,2}
  cout << "LCP array:   ";
  for (int i = 1; i < str.length(); i++)
      cout << " " << LCP[i];
  cout << endl;
  return 0;
}
```

Output:

```
Suffix array: 5 3 1 0 4 2
LCP array:    1 3 0 0 2
```

6.8 - Tries



A trie for keys “a”, “to”, “tea”, “ted”, “ten”, “i”, “in”, and “inn”.

Description: A trie is an ordered tree data structure used to a store a dynamic set of strings. Each edge represents a character of one or more strings that is part of the trie. The data structure thus allows for the querying of whether a string has been inserted in O(k), where k is the length of the string. Tries are faster than binary search trees because even though k is usually greater than log(N), where N is the number of elements in the binary search tree, every comparison is O(k) in a BST operation. So in reality, BST operations on strings run in O(k log(N)), which is worse than trie operations.

6.8.1 - Simple Trie

```
#include <string>

class trie {
/* number of symbols (a-z by default); */
#define ALPHABET_SIZE 26    /* use 256 for general chars */
/* convert to 0...ALPHABET_SIZE-1 */
#define INDEX(c) ((c)-'a')  /* (c+128) for unsigned char */

  struct node {
    bool isleaf; /* can be used to store nonzero values! */
    node *children[ALPHABET_SIZE];

    node(): isleaf(0) {
      for (int i = 0; i < ALPHABET_SIZE; i++)
        children[i] = 0; /* NULL */
    }
  } *root;

  static bool is_free_node(node *n) {
    for (int i = 0; i < ALPHABET_SIZE; i++)
      if (n->children[i] != 0) return true;
    return false;
  }

  bool _remove(const std::string& s, node *n, int level) {
    if (n != 0) {
      if (level == s.size()) {
        if (n->isleaf) {
          n->isleaf = 0;
          return is_free_node(n);
        }
      } else {
        int idx = INDEX(s[level]);
        if (_remove(s, n->children[idx], level+1)) {
          delete n->children[idx];
          return !n->isleaf && is_free_node(n);
        }
      }
    }
    return false;
  }

  static void clean_up(node *n) {
    if (n == 0 || n->isleaf) return;
    for (int i = 0; i < ALPHABET_SIZE; i++)
      clean_up(n->children[i]);
    delete n;
  }

public:
  trie() { root = new node(); }
  ~trie() { clean_up(root); }

  void insert(const std::string& s) {
    node *n = root;
    for (int level = 0; level < s.size(); level++) {
      int idx = INDEX(s[level]);
      if (n->children[idx] != 0) {
        n = n->children[idx];
      } else {
        n->children[idx] = new node();
        n = n->children[idx];
      }
    }
    n->isleaf = 1; /* or any non-zero value */
  }

  bool exists(const std::string& s) {
    node *n = root;
    for (int level = 0; level < s.size(); level++) {
      int idx = INDEX(s[level]);
      if (n->children[idx] == 0) return 0;
      n = n->children[idx];
    }
    return (n != 0 && n->isleaf);
  }

  bool remove(const std::string& s) {
    return _remove(s, root, 0);
  } /* returns whether remove was successful */
};
```

Example Usage:[28]

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

int main() {
  string s[8] = {"a","to","tea","ted","ten","i","in","inn"};
  trie T;
  for (int i = 0; i < 8; i++) T.insert(s[i]);
  cout << T.exists("ten") << endl; /* prints 1 */
  T.remove("tea");
  cout << T.exists("tea") << endl; /* prints 0 */
  return 0;
}
```

6.8.2 - Radix Trie

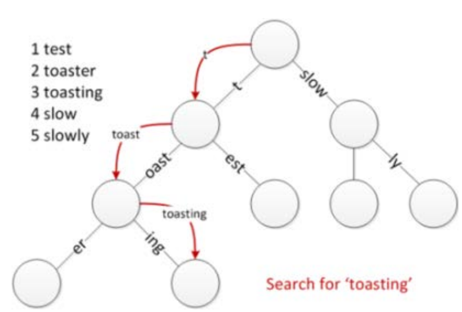
Description: A radix trie (a.k.a. radix tree, patricia tree, or compact prefix tree) is an trie optimized for memory, obtained by merging nodes so that internal nodes have ≥2 children.

Complexity: `insert()`, `remove()` and `exists()` are O(k), where k is the maximum length of all the strings that have been inserted into the trie. `walk()` is O(N) on the number of nodes in the trie.

```
#include <string>
#include <vector>

class radix_trie {

  struct node {
    std::string label;
    std::vector<node*> children;
    node(std::string s = ""): label(s) { }
  } *root;

  int match(const std::string &s, const std::string &t) {
    int X = 0, minlen;
    minlen = (t.size()<s.size()) ? t.size() : s.size();
    if (minlen == 0) return 0;
    for (int i = 0; i < minlen && s[i]==t[i]; i++) X++;
    return X;
  }

  void _insert(const std::string &s, node *n) {
    int X = match(s, n->label);
    if (X == 0 || n == root || (X > 0 &&
        X < s.size() && X >= n->label.size())) {
      bool inserted = false;
      std::string newstr = s.substr(X, s.size() - X);
      for (int i = 0; i < n->children.size(); i++)
        if (n->children[i]->label[0] == newstr[0]) {
            inserted = true;
            _insert(newstr, n->children[i]);
        }
      if (!inserted)
        n->children.push_back(new node(newstr));
    } else if (X < s.size()) {
      node *t = new node();
      t->label = n->label.substr(X, n->label.size()-X);
      t->children.assign(
        n->children.begin(), n->children.end());
      n->label = s.substr(0, X);
      n->children.assign(1, t);
      n->children.push_back(new node(
        s.substr(X, s.size() - X)));
    }
  }

  void _remove(const std::string &s, node *n) {
    int X = match(s, n->label);
    if (X == 0 || n == root || (X > 0 &&
        X < s.size() && X >= n->label.size())) {
      std::string newstr = s.substr(X, s.size() - X);
      for (int i = 0; i < n->children.size(); i++)
        if (n->children[i]->label[0] == newstr[0]) {
          if (newstr == n->children[i]->label &&
              n->children[i]->children.empty()) {
            n->children.erase(n->children.begin()+i);
            return;
          }
          _remove(newstr, n->children[i]);
        }
    }
  }

  bool _exists(const std::string &s, node *n) {
    int X = match(s, n->label);
    if (X == 0 || n == root || (X > 0 &&
        X < s.size() && X >= n->label.size())) {
      std::string newstr = s.substr(X, s.size() - X);
      for (int i = 0; i < n->children.size(); i++)
        if (n->children[i]->label[0] == newstr[0])
          return _exists(newstr, n->children[i]);
      return false;
    }
    return X == n->label.size();
  }

  static void clean_up(node *n) {
    if (n == 0) return;
    for (int i = 0; i < n->children.size(); i++)
      clean_up(n->children[i]);
    delete n;
  }

 template<class Func>
  static void _walk(node *n, void(*f)(Func)) {
    if (n == 0) return;
    (*f)(n->label);
    for (int i = 0; i < n->children.size(); i++)
      _walk(n->children[i], f);
  }

 public:
  radix_trie() { root = new node(); }
  ~radix_trie() { clean_up(root); }
  void insert(const std::string& s) { _insert(s, root); }
  void remove(const std::string& s) { _remove(s, root); }
  bool exists(const std::string& s) { return _exists(s,root); }
 template<class Func>
  void walk(void (*f)(Func)) { _walk(root, f); }
};
```

Example Usage: [29]

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

void printstr(const string &s) { cout << s << " "; }

int main() {
    radix_trie T;
    T.insert("test");
    T.insert("toaster");
    T.insert("toasting");
    T.insert("slow");
    T.insert("slowly");
    cout << "Preorder:";
    T.walk(printstr);
    return 0;
}
```



Output:

```
Preorder: t est oast er ing slow ly
```

Section 7 - Miscellaneous

7.1 - Fast Integer Input: The behaviour of `read(x)` is almost the same as `scanf("%d", &x)` in the 1st version, and `scanf("%u", &x)` in the 2nd version. The 3rd version is even more optimized, but stricter on format and should only be used for unsigned integers separated by spaces. It discards the character from the input stream right after the value read. These functions are very simple, and do not have error checks. On fully POSIX compliant compilers (GCC on windows will not work, but many contest systems use Linux, which will), `getchar_unlocked()` can be used instead of `getchar()` for even more speed in situations where thread-safety is unimportant, such as contests.

```
#include <cstdio> /* getchar(_unlocked)(), ungetc() */
#ifndef _WIN32    /* unlocked I/O is for POSIX only */
  #define getchar getchar_unlocked
#endif
template<class T> inline void read(T &a) {  //v1
    char c, neg = 0;
    while ((a = getchar()-'0') < 0 && a!=-3 || a > 9);
    if (a == -3) a = getchar()-'0', neg = 1;
    while ((c = getchar()-'0') >= 0 && c < 10)
        a = (a << 3) + (a << 1) + c;
    if (neg) a = -a;
    ungetc(c + '0', stdin);
}
```

```
template<class T> inline void read(T &a) { //v2
    char c;
    while ((a = getchar()-'0') < 0 || a > 9);
    while ((c = getchar()-'0') >= 0 && c < 10)
        a = (a << 3) + (a << 1) + c;
    ungetc(c + '0', stdin);
}

template<class T> inline void read(T &a) { //v3
    while ((a = getchar()-'0') < 0);
    for (char c; (c = getchar()-'0') >= 0;)
        a = (a << 3) + (a << 1) + c;
}
```

7.2 - Integer to Roman Numerals: Given an integer x, this function returns the Roman numeral representation of x as a C++ string. More 'M's are appended to the front of the output when x is greater than 1000.

```
std::string toRoman(int x) {
    static std::string H[] = {"","C","CC","CCC","CD","D","DC","DCC","DCCC","CM"};
    static std::string T[] = {"","X","XX","XXX","XL","L","LX","LXX","LXXX","XC"};
    static std::string O[] = {"","I","II","III","IV","V","VI","VII","VIII","IX"};
    return std::string(x / 1000, 'M') + H[(x %= 1000) / 100] + T[x / 10 % 10] + O[x % 10];
}
```

7.3 - Bitwise Operations: Interesting bit twiddling hacks. `get(x,N)`, `set1(x,N)`, `set0(x,N)` and `flip(x,N)` apply to the Nth bit from the right of x. `__builtin_popcount()` on GCC counts the number of 1 bits. `CHAR_BIT` is equal to 8.

```
inline int get(int x, int N) { return x & (1 << N); }
inline void set1(int &x, int N) { x |=  (1 << N); }
inline void set0(int &x, int N) { x &= ~(1 << N); }
inline void flip(int &x, int N) { x ^=  (1 << N); }
inline void swap(int &x, int &y) { x = (y ^= (x ^= y)) ^ x; }
inline int abs(int x) { const int mask = x >> sizeof(int)*CHAR_BIT - 1; return (x + mask) ^ mask; }
/* fast min, max functions without branching. They require the precondition: INT_MIN <= x - y <= INT_MAX */
inline int min(int x, int y) { return y + ((x - y) & ((x - y) >> (sizeof(int)*CHAR_BIT - 1))); }
inline int max(int x, int y) { return x - ((x - y) & ((x - y) >> (sizeof(int)*CHAR_BIT - 1))); }
```

7.4 - Coordinate Compression:
Description: Given an array of N values, this function reassigns values to the array elements such that the magnitude of each new value is no more than N, while preserving the relative order of each element from the original array.
Complexity: O(N log(N)).

```
#include <map>
void compress(int N, int A[]) {
    std::map<int, int> m;
    for (int i = 0; i < N; i++) m[A[i]] = 0;
    std::map<int, int>::iterator x = m.begin();
    for (int i = 0; x != m.end(); x++) x->second = i++;
    for (int i = 0; i < N; i++) A[i] = m[A[i]];
}
```

Example List: `{1, 30, 30, 7, 9, 8, 99, 99}`

After Compression: `{0, 4, 4, 1, 3, 2, 5, 5}`

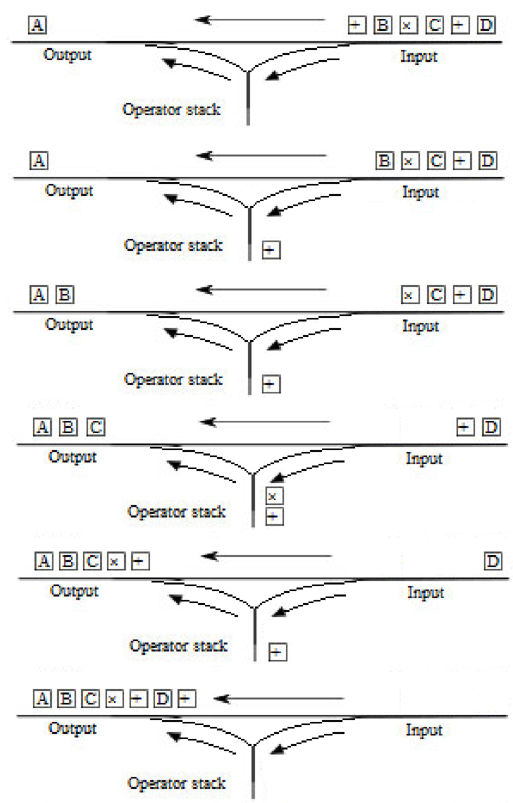
7.5 - Expression Evaluation (Shunting-yard Algorithm)[30]

Description: The shunting-yard algorithm parses mathematical expressions specified in infix notation.
Complexity: O(N) on the total number of operators and operands.

```
#include <cstdlib> /* strtol() */
#include <stack>
#include <string>
#include <vector>

/* We use strings for operators so we can even define
 * things like "sqrt" and "mod" as operators by
 * changing prec() and split_expr() accordingly.
 */
inline int prec(const std::string &op, bool unary) {
  if (unary) {
    if (op == "+" || op == "-") return 3;
    return 0; /* not a unary operator */
  }
  if (op == "*" || op == "/") return 2;
  if (op == "+" || op == "-") return 1;
  return 0;  /* not a binary operator */
}

inline int calc1(const std::string &op, int val) {
  if (op == "+") return +val;
  if (op == "-") return -val;
}

inline int calc2(const std::string &op, int L, int R) {
  if (op == "+") return L + R;
  if (op == "-") return L - R;
  if (op == "*") return L * R;
  if (op == "/") return L / R;
}

inline bool is_operand(const std::string &s) {
  return s!="(" && s!=")" && !prec(s,0) && !prec(s,1);
}
```

```
int eval(std::vector<std::string> E) { /* E stores the tokens */
  E.insert(E.begin(), "("); E.push_back(")");
  std::stack<std::pair<std::string, bool> > ops;
  std::stack<int> vals;
  for (int i = 0; i < E.size(); i++) {
    if (is_operand(E[i])) {
      vals.push(strtol(E[i].c_str(), 0, 10)); // convert to int
      continue;
    }
    if (E[i] == "(") {
      ops.push(std::make_pair("(", 0));
      continue;
    }
    if (prec(E[i],1)&&(i==0||E[i-1]=="("||prec(E[i-1],0))) {
      ops.push(std::make_pair(E[i], 1));
      continue;
    }
    while(prec(ops.top().first,ops.top().second)>=prec(E[i],0)) {
      std::string op = ops.top().first;
      bool is_unary = ops.top().second;
      ops.pop();
      if (op == "(") break;
      int y = vals.top(); vals.pop();
      if (is_unary) {
        vals.push(calc1(op, y));
      } else {
        int x = vals.top(); vals.pop();
        vals.push(calc2(op, x, y));
      }
    }
    if (E[i] != ")") ops.push(std::make_pair(E[i], 0));
  }
  return vals.top();
}
```

```
/**
 * Split a string expression to tokens, ignoring whitespace delimiters.
 * A vector of tokens is a more flexible format since you can decide to
 * parse the expression however you wish just by modifying this function.
 * e.g. "1+(51 * -100)" converts to {"1","+","(","51","*","-","100",")"}
 */
std::vector<std::string> split_expr(const std::string &s,
                const std::string &delim = " \n\t\v\f\r") {
  std::vector<std::string> ret;
  std::string acc = "";
  for (int i = 0; i < s.size(); i++)
    if (s[i] >= '0' && s[i] <= '9') acc = acc + s[i];
    else {
      if (i > 0 && s[i-1] >= '0' && s[i-1] <= '9') ret.push_back(acc);
      acc = "";
      if (delim.find(s[i]) != std::string::npos) continue;
      ret.push_back(std::string("") + s[i]);
    }
  if (s[s.size()-1] >= '0' && s[s.size()-1] <= '9') ret.push_back(acc);
  return ret;
}

#include <iostream>
using namespace std;

int main() {
  cout << eval(split_expr("1+(51 * -100)")) << endl; /* prints -5099 */
  return 0;
}
```



7.6 - Linear Programming (Simplex Algorithm)[31]

Description: The canonical form of a linear programming problem is to maximize $c^{T}x$, subject to $Ax \leq b$, and $x \geq 0$. x represents the vector of variables (to be determined), c and b are vectors of (known) coefficients, A is a (known) matrix of coefficients, and $(.)^{T}$ is the matrix transpose.

Complexity: The simplex method is remarkably efficient in practice, usually taking 2m or 3m iterations (where m is the number of constraints), converging in expected polynomial time for certain distributions of random inputs. However, its worst-case complexity is exponential, and can be demonstrated with carefully constructed examples.

```
#include <cmath> /* fabs() */
#include <vector>

double simplex_solve(std::vector<double> &results,
                     const std::vector<double> &function,
                     const std::vector<std::vector<double> > &mat,
                     bool maximize = true) {
    int P1 = 0, P2 = 0, DONE, XERR = 0;
    int NV = function.size() - 1, NC = mat.size();
    double TS[NC + 2][NV + 2], R = maximize ? 1 : -1;
    for (int i = 0; i < NC + 2; i++)
        for (int j = 0; j < NV + 2; j++) TS[i][j] = 0;
    for (int j = 1; j <= NV; j++) TS[1][j+1] = function[j-1] * R;
    TS[1][1] = function[NV] * R;
    for (int i = 1; i <= NC; i++) {
        for (int j = 1; j <= NV; j++)
            TS[i+1][j+1] = -mat[i-1][j-1];
        TS[i+1][1] = mat[i-1][NV];
    }
    for (int j = 1; j <= NV; j++) TS[0][j+1] = j;
    for (int i = NV + 1; i <= NV + NC; i++) TS[i-NV+1][0] = i;
    do {
        double M = 1E+36, V, XMAX = 0.0;
        for (int j = 2; j <= NV + 1; j++)
            if (TS[1][j] > 0.0 && TS[1][j] > XMAX)
                XMAX = TS[1][P2 = j];
        for (int i = 2; i <= NC + 1; i++)
            if (TS[i][P2] < 0 && (V=fabs(TS[i][1]/TS[i][P2])) < M)
                M = V, P1 = i;
        V = TS[0][P2];
        TS[0][P2] = TS[P1][0];
        TS[P1][0] = V;
        for (int i = 1; i <= NC + 1; i++) {
            if (i == P1) continue;
            for (int j = 1; j <= NV + 1; j++)
               if (j != P2)
                   TS[i][j] -= TS[P1][j] * TS[i][P2] / TS[P1][P2];
        }
        TS[P1][P2] = 1.0 / TS[P1][P2];
        for (int j = 1; j <= NV + 1; j++)
            if (j != P2) TS[P1][j] *= fabs(TS[P1][P2]);
        for (int i = 1; i <= NC + 1; i++)
            if (i != P1) TS[i][P2] *= TS[P1][P2];
        for (int i = 2; i <= NC + 1; i++)
            if (TS[i][1] < 0.0) XERR = 1;
        if (XERR == 1) return -(0.0/0.0); /* no solution; return NaN */
        DONE = 1;
        for (int j = 2; j <= NV + 1; j++)
            if (TS[1][j] > 0) DONE = 0;
    } while (!DONE);
    results.clear();
    for (int i = 1; i <= NV; i++)
        for (int j = 2; j <= NC + 1; j++)
            if (TS[j][0] == (double)i) results.push_back(TS[j][1]);
    return TS[1][1];
}
```

Example Usage:

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

int main()  {
    int vars, cons; cin >> vars >> cons;
    vector<double> F(vars+1);
    vector< vector<double> >
      M(cons, vector<double>(vars+1));
    for (int i = 0; i < vars+1; i++)
        cin >> F[i];
    for (int i = 0; i < cons; i++)
        for (int j = 0; j < vars+1; j++)
            cin >> M[i][j];
    vector<double> res;
    double ans = simplex_solve(res, F, M);
    cout << "Solution=" << ans;
    cout << " at: " << "(" << res[0];
    for (int i = 1; i < res.size(); i++)
        cout << ", " << res[i];
    cout << ")." << endl;
    return 0;
}
```

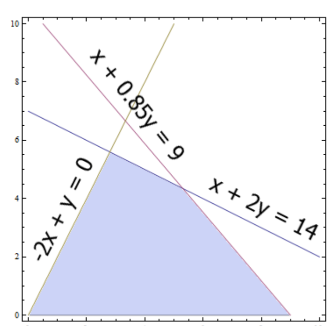
Example Input:

```
2 3
3 4  5
-2 1 0
1 0.85 9
1 2 14
```

Output: `Solution=38.3043 at (5.30435, 4.34783).`

Explanation:

```
Maximize 3x + 4y + 5, subject to x,y >= 0 and:
             -2x +     y <=  0
               x + 0.85y <=  9
               x +    2y <= 14
```



Endnotes

Section 1

[1] Depth-first search. (2013, August 29). In *Wikipedia, The Free Encyclopedia*. Retrieved from http://en.wikipedia.org/w/index.php?title=Depth-first_search&oldid=570671753

[2] Breadth-first search. (2013, August 29). In *Wikipedia, The Free Encyclopedia*. Retrieved from http://en.wikipedia.org/w/index.php?title=Breadth-first_search&oldid=570694689

[3] Dijkstra's algorithm. (2013, August 26). In *Wikipedia, The Free Encyclopedia*. Retrieved from http://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=570309136

[4] Shortest Path Faster Algorithm. (2013, July 7). In *Wikipedia, The Free Encyclopedia*. Retrieved 04:25, September 1, 2013, from http://en.wikipedia.org/w/index.php?title=Shortest_Path_Faster_Algorithm&oldid=563209123

[5] Kruskal's algorithm. (2013, June 20). In *Wikipedia, The Free Encyclopedia*. Retrieved from http://en.wikipedia.org/w/index.php?title=Kruskal%27s_algorithm&oldid=560792262

[6] Prim's algorithm. (2013, August 28). In *Wikipedia, The Free Encyclopedia*. Retrieved from: http://en.wikipedia.org/w/index.php?title=Prim%27s_algorithm&oldid=570579798

[7] Kosaraju's algorithm. (2013, December 24). In *Wikipedia, The Free Encyclopedia*. Retrieved from http://en.wikipedia.org/w/index.php?title=Kosaraju%27s_algorithm&oldid=587504275

[8] Tarjan's strongly connected components algorithm. (2013, August 31). In *Wikipedia, The Free Encyclopedia*. Retrieved from http://en.wikipedia.org/w/index.php?title=Tarjan%27s_strongly_connected_components_algorithm&oldid=570990581

Section 2

[9] Comb sort. (2013, October 8). In *Wikipedia, The Free Encyclopedia*. Retrieved from http://en.wikipedia.org/w/index.php?title=Comb_sort&oldid=576237100

[10] Brejová, B. (2001). Analyzing variants of shellsort.*Information Processing Letters*, *79*(5), 223–227. doi: 10.1016/S0020-0190(00)00223-4

[11] Implementation adapted from: http://rosettacode.org/wiki/Sorting_algorithms/Radix_sort#C.2B.2B

[12] For further reading: http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binarySearch

Section 3

[13] Disjoint-set data structure. (2014, January 25). In *Wikipedia, The Free Encyclopedia*. Retrieved from http://en.wikipedia.org/w/index.php?title=Disjoint-set_data_structure&oldid=592338143

[14] For further reading: http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees

[15] Implementation adapted from: http://petr-mitrichev.blogspot.ca/2013/05/fenwick-tree-range-updates.html

[16] Implementation adapted from Mark Gordon’s solution to IOI 2013 Day 2 Problem 3 - “Game”, found here: https://gist.github.com/msg555/6025939

[17] Implementation adapted from: https://github.com/jonnyhsy/treap

[18] Implementation adapted from: http://www.oopweb.com/Algorithms/Documents/AvlTrees/Volume/AvlTrees.htm

[19] For more integer hash functions: https://gist.github.com/badboy/6267743

Section 4

[20] Implementation adapted from: https://sites.google.com/site/hannuhelminen/next_combination

[21] Implementation adapted from:
http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=primalityTesting

[22] Skiena, S. (2003). High-Precision Integers. *Programming challenges*. (pp. 103-111). New York: Springer-Verlag New York, Inc.

Section 5

[23] Weisstein, Eric W. "Circle-Line Intersection." From *MathWorld* - A Wolfram Web Resource.
http://mathworld.wolfram.com/Circle-LineIntersection.html

[24] Bourke, P. (1997, April). *Circles and spheres: Intersection of two circles*. Retrieved from
http://paulbourke.net/geometry/circlesphere/

Section 6

[25] For further reading: http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=stringSearching

[26] Implementation adapted from: http://www.jb.man.ac.uk/~slowe/cpp/itoa.html

[27] Kärkkäinen, J., Sanders, P., & Burkhard, S. (2006). Linear work suffix array construction. *Journal of the ACM (JACM)*, *53*(6), 19-20. doi: 10.1145/1217856.1217858. Retrieved from http://algo2.iti.kit.edu/documents/jacm05-revised.pdf

[28] Example and image taken from: http://en.wikipedia.org/wiki/File:Trie_example.svg

[29] Example and image taken from:
http://en.wikipedia.org/wiki/File:An_example_of_how_to_find_a_string_in_a_Patricia_trie.png

Section 7

[30] Shunting-yard algorithm. (2014, January 5). In *Wikipedia, The Free Encyclopedia*. Retrieved from
http://en.wikipedia.org/w/index.php?title=Shunting-yard_algorithm&oldid=589343402

[31] Implementation adapted from: http://jean-pierre.moreau.pagesperso-orange.fr/Cplus/simplex_cpp.txt