1. Grafos

1.1. Shortest paths

1.1.1. Dijkstra

Dijkstra's algorithm, is a graph search algorithm that solves the single-source shortest path problem for a graph with nonnegative edge path costs, producing a shortest path tree.

For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities.

Pseudocode:

```
function Dijkstra(Graph, source):
      for each vertex v in Graph:
            v.dist := infinity
            v.previous := undefined
            // Previous node in optimal path from source
       source.dist := 0
       Q := Priority queue with the set of all nodes in Graph
       while not Q.isEmpty():
           u := Q.deleteMin()
           if u.dist = infinity:
               break
          for each neighbor v of u:
          // where v has not yet been removed from Q.
               alt := u.dist + dist_between(u, v)
               if alt < v.dist:             // Relax (u,v,a)
                   v.dist := alt
                   v.previous := u
```

```
                    Q.decreaseKey(v)
```

An upper bound of the running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of | E | and | V | using the Big-O notation.

For any implementation of set Q the running time is O(E * dkQ + V * emQ), where dkQ and emQ are times needed to perform decrease key and extract minimum operations in set Q, respectively.

The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and extract minimum from Q is simply a linear search through all vertices in Q. In this case, the running time is O(| V | 2 + | E |) = O(| V | 2).

For sparse graphs, that is, graphs with far fewer than O(| V | 2) edges, Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a binary heap, pairing heap, or Fibonacci heap as a priority queue to implement extracting minimum efficiently. With a binary heap, the algorithm requires O((| E | + | V |)log | V |) time (which is dominated by O(| E | log | V |), assuming the graph is connected), and the Fibonacci heap improves this to O(| E | + | V | log | V |).

1.1.2. Bellman-Ford (also longest and negative)

The **Bellman-Ford algorithm** computes single-source shortest paths in a weighted digraph. For graphs with only non-negative edge weights, the faster Dijkstra's algorithm also solves the problem. Thus, Bellman-Ford is used primarily for graphs with negative edge weights. The algorithm is named after its developers, Richard Bellman and Lester Ford, Jr.

If a graph contains a "negative cycle", i.e., a cycle whose edges sum to a negative value, then walks of arbitrarily low weight can be constructed, i.e., there can be no *shortest* path. Bellman-Ford can detect negative cycles and report their existence, but it cannot produce a correct answer if a negative cycle is reachable from the source.

Para solucionar el longest path, simplemente se aplica bellman-ford con los pesos de los caminos negativos.

Pseudocodigo:

```
procedure BellmanFord(list vertices, list edges, vertex source)
   // This implementation takes in a graph, represented as lists of vertices
   // and edges, and modifies the vertices so that their distance and
   // predecessor attributes store the shortest paths.

   // Step 1: initialize graph
   for each vertex v in vertices:
       if v is source then v.distance := 0
       else v.distance := infinity
       v.predecessor := null

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1:
       for each edge uv in edges: // uv is the edge from u to v
           u := uv.source
           v := uv.destination
           if u.distance + uv.weight < v.distance:
               v.distance := u.distance + uv.weight
               v.predecessor := u

   // Step 3: check for negative-weight cycles
   for each edge uv in edges:
       u := uv.source
       v := uv.destination
       if u.distance + uv.weight < v.distance:
           error "Graph contains a negative-weight cycle"
```

1.1.3. Edge disjoint shortest paths

Edge disjoint shortest pair algorithm is an algorithm in computer network routing.[1] The algorithm is used for generating the shortest pair of edge disjoint paths between a given pair of vertices as follows:

- Run the shortest pair algorithm for the given pair of vertices
- Replace each edge of the shortest path (equivalent to two oppositely directed arcs) by a single arc directed towards the source vertex
- Make the length of each of the above arcs negative
- Run the shortest path algorithm *(Note: the algorithm should accept negative costs)*
- Erase the overlapping edges of the two paths found, and reverse the direction of the remaining arcs on the first shortest path such that each arc on it is directed towards the sink vertex now. The desired pair of paths results.

1.1.3.1. Suurballe (completely disjoint)

In theoretical computer science and network routing, **Suurballe's algorithm** is an algorithm for finding two disjoint paths in a nonnegatively-weighted directed graph, so that both paths connect the same pair of vertices and have minimum total length. The algorithm was conceived by J. W. Suurballe and published in 1974.[1][2][3] The main idea of Surballe's algorithm is to use Dijkstra's algorithm to find one path, to modify the weights of the graph edges, and then to run Dijkstra's algorithm a second time. The modification to the weights is similar to the weight modification in Johnson's algorithm, and preserves the non-negativity of the weights while allowing the second instance of Dijkstra's algorithm to find the correct second path.

Suurballe's algorithm performs the following steps:

1. Find the shortest path tree *T* rooted at node *s* by running Dijkstra's algorithm. This tree contains for every vertex *u*, a shortest path from *s* to *u*. Let *F*1 be the shortest cost path froms to *t*. The edges in *T* are called *tree edges* and the remaining edges are called *non tree edges*.
2. Modify the cost of each edge in the graph by replacing the cost $w(u,v)$ of every edge (u,v) by $w'(u,v) = w(u,v) - d(s,v) + d(s,u)$. According to the resulting modified cost function, all tree edges have a cost of 0, and non tree edges have a non negative cost.
3. Create a residual graph *Gt* formed from *G* by removing the edges of *G* that are directed into *s* and by reversing the direction of the zero length edges along path *F*1.
4. Find the shortest path *F*2 in the residual graph *Gt* by running Dijkstra's algorithm.
5. Discard the reversed edges of *F*2 from both paths. The remaining edges of *F*1 and *F*2 form a subgraph with two outgoing edges at *s*, two incoming edges at *t*, and one incoming and one outgoing edge at each remaining vertex. Therefore, this subgraph consists of two edge-disjoint paths from *s* to *t* and possibly some additional (zero-length) cycles. Return the two disjoint paths from the subgraph.

1.2. All-pairs shortest paths

Es una versión del shortest path problem donde hay que hayar la menor distancia entre todos los nodos.

Existen dos algoritmos para solucionarlo, Floyd-Warshall y Jhonson. Floyd-Warshall lo hace en tiempo cubico, mientras que Jhonson en cuadratico, pero, sin embargo, muchas veces el overhead del Jhonson puede hacer que sea más lento que el Floyd-Warshall para Nes medianamente grandes (porque hay que implementar Dijkstra, y por ende una Priority Queue)

1.2.1. Floyd-Warshall

In computer science, the **Floyd–Warshall algorithm** (sometimes known as the **WFI Algorithm**[*clarification needed*] or **Roy–Floyd algorithm**) is a graph analysis algorithm for finding shortest paths in a weighted graph (with positive or negative edge weights). A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between *all* pairs of vertices. The algorithm is an example of dynamic programming.

Pseudocodigo:

Normal

```
/* Assume a function edgeCost(i,j) which returns the cost of the edge from i to j
   (infinity if there is none).
   Also assume that n is the number of vertices and edgeCost(i,i) = 0
*/

int path[][];
/* A 2-dimensional matrix. At each step in the algorithm, path[i][j] is the shortest path
   from i to j using intermediate vertices (1..k−1).  Each path[i][j] is initialized to
   edgeCost(i,j) or infinity if there is no edge between i and j.
*/

procedure FloydWarshall ()
   for k := 1 to n
      for i := 1 to n
         for j := 1 to n
            path[i][j] = min ( path[i][j], path[i][k]+path[k][j] );
```

Con reconstrucción de path

```
procedure FloydWarshallWithPathReconstruction ()
   for k := 1 to n
      for i := 1 to n
         for j := 1 to n
            if path[i][k] + path[k][j] < path[i][j] then
               path[i][j] := path[i][k]+path[k][j];
               next[i][j] := k;
```

```

procedure GetPath (i,j)
   if path[i][j] equals infinity then
     return "no path";
   int intermediate := next[i][j];
   if intermediate equals 'null' then
     return " ";   /* there is an edge from i to j, with no vertices between */
  else
     return GetPath(i,intermediate) + intermediate + GetPath(intermediate,j);
```

1.2.2. Johnson

Johnson's algorithm is a way to find the shortest paths between all pairs of vertices in a sparse directed graph. It allows some of the edge weights to be negative numbers, but no negative-weight cycles may exist. It works by using the Bellman-Ford algorithm to compute a transformation of the input graph that removes all negative weights, allowing Dijkstra's algorithm to be used on the transformed graph.

Pseudocodigo:

Johnson's algorithm consists of the following steps:

1. First, a new node *q* is added to the graph, connected by zero-weight edges to each other node.
2. Second, the Bellman-Ford algorithm is used, starting from the new vertex *q*, to find for each vertex *v* the least weight $h(v)$ of a path from *q* to *v*. If this step detects a negative cycle, the algorithm is terminated.
3. Next the edges of the original graph are reweighted using the values computed by the Bellman-Ford algorithm: an edge from *u* to *v*, having length$w(u,v)$, is given the new length $w(u,v) + h(u) - h(v)$.
4. Finally, *q* is removed, and Dijkstra's algorithm is used to find the shortest paths from each node *s* to every other vertex in the reweighted graph.

In the reweighted graph, all paths between a pair *s* and *t* of nodes have the same quantity $h(s) - h(t)$ added to them, so a path that is shortest in the original graph remains shortest in the modified graph and vice versa. However, due to the way the values $h(v)$ were computed, all modified edge lengths are non-negative, ensuring the optimality of the paths found by Dijkstra's algorithm. The distances in the original graph may be calculated from the distances calculated by Dijkstra's algorithm in the reweighted graph by reversing the reweighting transformation.

The time complexity of this algorithm, using Fibonacci heaps in the implementation of Dijkstra's algorithm, is O(*V*2log *V* + *VE*): the algorithm uses O(*VE*) time for the Bellman–Ford stage of the algorithm, and O(*V* log *V* + *E*) for each of *V* instantiations of Dijkstra's algorithm. Thus, when the graph is sparse, the total time can be faster than the Floyd–Warshall algorithm, which solves the same problem in time O(*V*3).

1.3. MST

Given a connected, undirected graph, a spanning tree of that graph is a subgraph which is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a *weight* to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A minimum spanning tree (MST) or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of minimum spanning trees for its connected components.
One example would be a cable TV company laying cable to a new neighborhood. If it is constrained to bury the cable only along certain paths, then there would be a graph representing which points are connected by those paths. Some of those paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. A *spanning tree* for that graph would be a subset of those paths that has no cycles but still connects to every house. There might be several spanning trees possible. A *minimum spanning tree* would be one with the lowest total cost.

Hay dos algoritmos para solucionar MST, Kruskal y Prim. Cada uno tiene sus ventajas y desventajas.

1.3.1. Kruskal

Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

Where E is the number of edges in the graph and V is the number of vertices, Kruskal's algorithm can be shown to run in $O(E \log E)$ time, or equivalently, $O(E \log V)$ time, all with simple data structures. These running times are equivalent because:

- E is at most $V2$ and $\log V2 = 2\log V$ is $O(\log V)$.
- If we ignore isolated vertices, which will each be their own component of the minimum spanning forest, $V \leq E+1$, so $\log V$ is $O(\log E)$.

We can achieve this bound as follows: first sort the edges by weight using a comparison sort in $O(E \log E)$ time; this allows the step "remove an edge with minimum weight from S" to operate in constant time. Next, we use a disjoint-set data structure (Union&Find) to keep track of which vertices are in which components. We need to perform $O(E)$ operations, two 'find' operations and possibly one union for each edge. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform $O(E)$ operations in $O(E \log V)$ time. Thus the total time is $O(E \log E) = O(E \log V)$.

Provided that the edges are either already sorted or can be sorted in linear time (for example with counting sort or radix sort), the algorithm can use more sophisticated disjoint-set data structure to run in $O(E\, \alpha(V))$ time, where α is the extremely slowly-growing inverse of the single-valued Ackermann function.

Pseudocodigo:

```
function Kruskal(G = <N, A>: graph; length: A → R+): set of edges
    Define an elementary cluster C(v) ← {v}.
    Initialize a priority queue Q to contain all edges in G, using the weights as keys.
    Define a forest T ← Ø       //T will ultimately contain the edges of the MST
     // n is total number of vertices
    while T has fewer than n-1 edges do
      // edge u,v is the minimum weighted route from u to v
      (u,v) ← Q.removeMin()
      // prevent cycles in T. add u,v only if T does not already contain a path between u and v.
      // the vertices has been added to the tree.
      Let C(v) be the cluster containing v, and let C(u) be the cluster containing u.
      if C(v) ≠ C(u) then
        Add edge (v,u) to T.
        Merge C(v) and C(u) into one cluster, that is, union C(v) and C(u).
    return tree T
```

Este algoritmo se usa con la estructura de datos disjoint-set que esta en el ultimo capitulo de este resumen.

1.3.2. Prim

Prim's algorithm is an algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. Prim's algorithm is an example of a greedy algorithm.

El algoritmo de Prim tiene exactamente el mismo tiempo de ejecución asimptotico que el de Dijkstra y usar una Fibonnaci Heap o una Binary Heap tienen los mismos beneficios.

Pseudocodigo:

```
Min-heap
Initialization
    inputs: A graph, a function returning edge weights weight-function, and
    an initial vertex
Initial placement of all vertices in the 'not yet seen' set, set initial
vertex to be added to the tree, and place all vertices in a min-heap to allow
for removal of the min distance from the minimum graph.
for each vertex in graph
   set min_distance of vertex to ∞
   set parent of vertex to null
   set minimum_adjacency_list of vertex to empty list
   set is_in_Q of vertex to true
set min_distance of initial vertex to zero
add to minimum-heap Q all vertices in graph, keyed by min_distance

/*

Algorithm
In the algorithm description above,
    nearest vertex is Q[0], now latest addition
    fringe is v in Q where distance of v < ∞ after nearest vertex is removed
    not seen is v in Q where distance of v = ∞ after nearest vertex is
    removed
The while loop will fail when remove minimum returns null. The adjacency list
is set to allow a directional graph to be returned.
    time complexity: V for loop, log(V) for the remove function

*/
```

```
while latest_addition = remove minimum in Q
    set is_in_Q of latest_addition to false
    add latest_addition to (minimum_adjacency_list of (parent of
latest_addition))
    add (parent of latest_addition) to (minimum_adjacency_list of
latest_addition)
   for each adjacent of latest_addition
    if (is_in_Q of adjacent) and (weight-function(latest_addition, adjacent)
< min_distance of adjacent)
        set parent of adjacent to latest_addition
        set min_distance of adjacent to weight-function(latest_addition,
adjacent)
       update adjacent in Q, order by min_distance
```

Implementación:

```
public class Prim
{
	static class Nodo
	{
		Nodo parent = null;
		boolean is_in_q = true;
		int min_distance = Integer.MAX_VALUE;
		ArrayList <Arista> aristas = new ArrayList <Arista> ();
	}

	static class Arista
	{
		Nodo b;
		int cost;

		public Arista(Nodo b, int cost)
		{
			this.b = b;
			this.cost = cost;
		}
	}

	static class EntradaPQ
	{
		int costo;
		Nodo c;

		public EntradaPQ(Nodo c, int costo)
		{
			this.c = c;
```

```
                this.costo = costo;
            }
        }

        public static int mstPrim(ArrayList <Nodo> nodos)
        {
            Nodo inicial = nodos.get(0); // <- establecer inicial
            PriorityQueue <EntradaPQ> q = new PriorityQueue <EntradaPQ> ();
            q.add(new EntradaPQ(inicial, 0));
            int totalCost = 0;
            while(!q.isEmpty())
            {
                Nodo latest = q.poll().c;
                if(!latest.is_in_q)
                    continue;
                totalCost += latest.min_distance;
                latest.is_in_q = false;
                for(Arista a : latest.aristas)
                {
                    Nodo adj = a.b;
                    if(adj.is_in_q && a.cost < adj.min_distance)
                    {
                        adj.parent = latest;
                        adj.min_distance = a.cost;
                        q.add(new EntradaPQ(adj, adj.min_distance));
                    }
                }
            }
            return totalCost;
        }
    }
```

1.4. Graph search

1.4.1. BFS

In graph theory, **breadth-first search** (**BFS**) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

1. Enqueue the root node.
2. Dequeue a node and examine it.

    - If the element sought is found in this node, quit the search and return a result.
    - Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
4. Repeat from Step 2.

Note: Using a stack instead of a queue would turn this algorithm into a depth-first search.

1.4.2 DFS

Depth-first search (**DFS**) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

1.5. Connected components

In graph theory, a **connected component** of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and to which no more vertices or edges (from the larger graph) can be added while preserving its connectivity. That is, it is a maximalconnected subgraph. For example, the graph shown in the illustration on the right has three connected components. A graph that is itself connected has exactly one connected component, consisting of the whole graph.

It is straightforward to compute the connected components of a graph in linear time using either breadth-first search or depth-first search. In either case, a search that begins at some particular vertex *v* will find the entire connected component containing *v* (and no more) before returning. To find all the connected components of a graph, loop through its vertices, starting a new breadth first or depth first search whenever the loop reaches a vertex that has not already been included in a previously found connected component. Hopcroft and Tarjan (1973)[1]describe essentially this algorithm, and state that at that point it was "well known".

There are also efficient algorithms to dynamically track the connected components of a graph as vertices and edges are added, as a straightforward application of disjoint-set data structures. These algorithms require amortized $O(\alpha(n))$ time per operation, where adding vertices and edges and determining the connected component in which a vertex falls are both operations, and α(*n*) is a very slow-growing inverse of the very quickly-growing Ackermann function. A related problem is tracking connected components as all edges are deleted from a graph, one by one; an algorithm exists to solve this with constant time per query, and O(|V||E|) time to maintain the data structure; this is an amortized cost of O(|V|) per edge deletion. For forests, the cost can be reduced to O(q + |V| log |V|), or O(log |V|) amortized cost per edge deletion.[2]

Researchers have also studied algorithms for finding connected components in more limited models of computation, such as programs in which the working memory is limited to alogarithmic number of bits (defined by the complexity class L). Lewis & Papadimitriou (1982) asked whether it is possible to test in logspace whether two vertices belong to the same connected component of an undirected graph, and defined a complexity class SL of problems logspace-equivalent to connectivity. Finally Reingold (2008) succeeded in finding an algorithm for solving this connectivity problem in logarithmic space, showing that L = SL.

1.5.1. Strongly connected components and cycle detection

A directed graph is called *strongly connected* if there is a path from each vertex in the graph to every other vertex. In particular, this means paths in each direction; a path from *a* to *b* and also a path from *b* to *a*.
The **strongly connected components** of a directed graph *G* are its maximal strongly connected subgraphs. If each strongly connected component is contracted to a single vertex, the resulting graph is a directed acyclic graph, the **condensation** of *G*. A directed graph is acyclic if and only if it has no (nontrivial) strongly connected subgraphs (because a cycle is strongly connected, and every strongly connected graph contains at least one cycle).
Kosaraju's algorithm, Tarjan's algorithm and Gabow's algorithm all efficiently compute the strongly connected components of a directed graph, but Tarjan's and Gabow's are favoured in practice since they require only one depth-first search rather than two.
Algorithms for finding strongly connected components may be used to solve 2-satisfiability problems (systems of Boolean variables with constraints on the values of pairs of variables): asAspvall, Plass & Tarjan (1979) showed, a 2-satisfiability instance is unsatisfiable if and only if there is a variable *v* such that *v* and its complement are both contained in the same strongly connected component of the implication graph of the instance.
According to Robbins theorem, an undirected graph may be oriented in such a way that it becomes strongly connected, if and only if it is 2-edge-connected.

```
public class SCC
{
	static class Node implements Comparable <Node>
	{
```

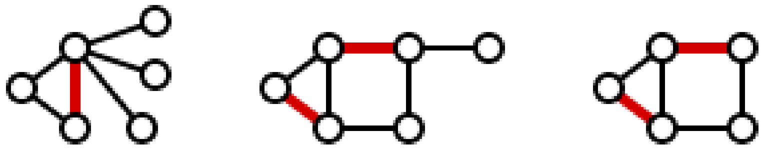
```
            final int name;
            ArrayList <Node> adjacents = new ArrayList <Node> (100);
            boolean visited = false;
            int lowlink = -1;
            int index = -1;

            public Node(final int argName)
            {
                name = argName;
            }

            public int compareTo(final Node argNode)
            {
                return argNode == this ? 0 : -1;
            }
      }

      static int index = 0;
      static ArrayDeque <Node> stack = new ArrayDeque <Node> ();
      static ArrayList <ArrayList <Node> > SCC = new ArrayList <ArrayList
<Node> > ();

      public static ArrayList < ArrayList <Node> > tarjanSCC(ArrayList <Node>
nodes)
      {
            index = 0;
            SCC.clear();
            stack.clear();
            for(Node n : nodes)
                  if(n.index == -1)
                        tarjan(n);
            return SCC;
      }

      public static void tarjan(Node v)
      {
            v.index = index;
            v.lowlink = index;
            index++;
            stack.push(v);
            for(Node n : v.adjacents)
            {
                  if(n.index == -1)
                  {
                        tarjan(n);
                        v.lowlink = Math.min(v.lowlink, n.lowlink);
                  }
```

```
                        else if(stack.contains(n))
                              v.lowlink = Math.min(v.lowlink, n.index);
                  }
                  if(v.lowlink == v.index)
                  {
                        Node n;
                        ArrayList <Node> component = new ArrayList <Node> ();
                        do
                        {
                              n = stack.pop();
                              component.add(n);
                        }
                        while(n != v);
                        SCC.add(component);
                  }
            }
      }
```

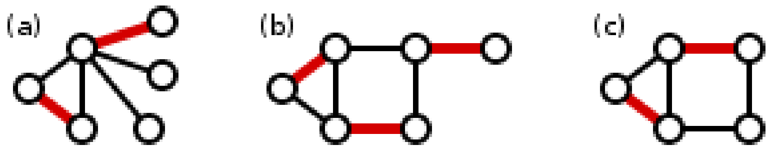
1.6. Matching

In the mathematical discipline of graph theory, a **matching** or **independent edge set** in a graph is a set of edges without common vertices. It may also be an entire graph consisting of edges without common vertices.

Given a graph *G* = (*V*,*E*), a **matching** *M* in *G* is a set of pairwise non-adjacent edges; that is, no two edges share a common vertex.
A vertex is **matched** (or **saturated**) if it is incident to an edge in the matching. Otherwise the vertex is **unmatched**.
A **maximal matching** is a matching *M* of a graph *G* with the property that if any edge not in *M* is added to *M*, it is no longer a matching, that is, *M* is maximal if it is not a proper subset of any other matching in graph *G*. In other words, a matching *M* of a graph *G* is maximal if every edge in *G* has a non-empty intersection with at least one edge in *M*. The following figure shows examples of maximal matchings (red) in three graphs.



A **maximum matching** is a matching that contains the largest possible number of edges. There may be many maximum matchings. The **matching number** ν(*G*) of a graph *G* is the size of a maximum matching. Note that every maximum matching is maximal, but not every maximal matching is a maximum matching. The following figure shows examples of maximum matchings in three graphs.



A **perfect matching** (a.k.a. 1-factor) is a matching which matches all vertices of the graph. That is, every vertex of the graph is incident to exactly one edge of the matching. Figure (b) above is an example of a perfect matching. Every perfect matching is maximum and hence maximal. In some literature, the term **complete matching** is used. In the above figure, only part (b) shows a perfect matching. A perfect matching is also a minimum-size edge cover. Thus, $\nu(G) \leq \rho(G)$, that is, the size of a maximum matching is no larger than the size of a minimum edge cover.

A **near-perfect matching** is one in which exactly one vertex is unmatched. This can only occur when the graph has an odd number of vertices, and such a matching must be maximum. In the above figure, part (c) shows a near-perfect matching. If, for every vertex in a graph, there is a near-perfect matching that omits only that vertex, the graph is also called factor-critical.

Given a matching *M*,

- an **alternating path** is a path in which the edges belong alternatively to the matching and not to the matching.
- an **augmenting path** is an alternating path that starts from and ends on free (unmatched) vertices.

One can prove that a matching is maximum if and only if it does not have any augmenting path. (This result is sometimes called Berge's lemma.)

1.6.1. Maximum matching

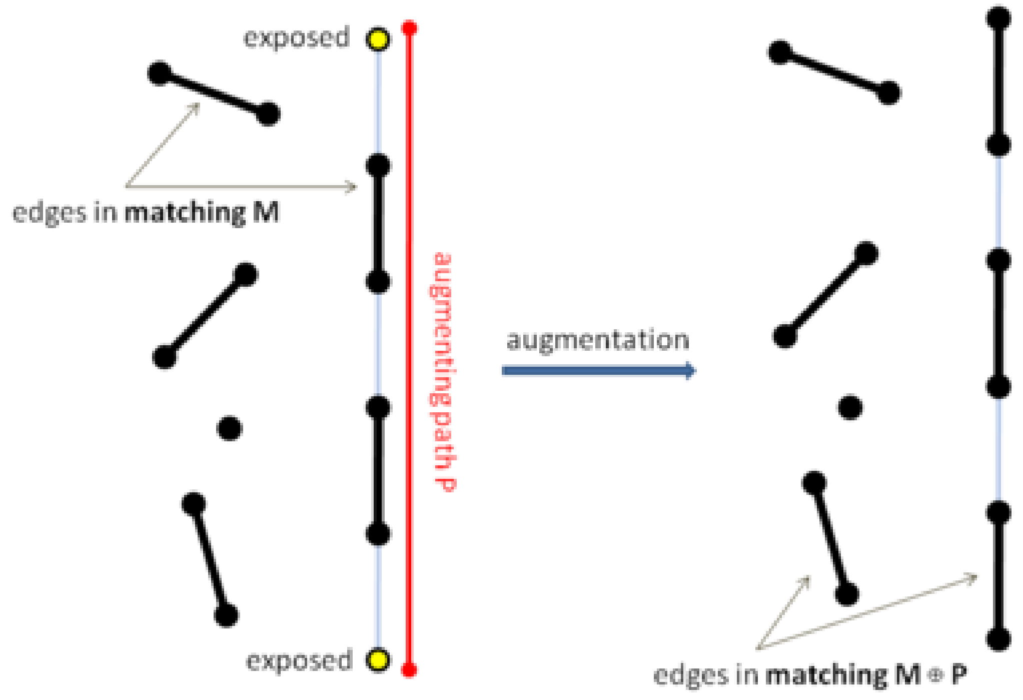
1.6.1.1. Edmonds

Edmonds's matching algorithm [1] is an algorithm in graph theory for constructing maximum matchings on graphs. The algorithm was discovered by Jack Edmonds in 1965. Given a general graph *G* = (*V*, *E*), the algorithm finds a matching *M* such that each vertex in *V* is incident with at most one edge in *M* and |*M*| is maximized. The matching is constructed by iteratively improving an initial empty matching along augmenting paths in the graph. To search for augmenting paths, some odd-length cycles in the graph (blossoms) are contracted to single vertices and the search continues recursively in the contracted graphs.

Augmenting paths

Given *G* = (*V*, *E*) and a matching *M* of *G*, a vertex *v* is **exposed**, if no edge of *M* is incident with *v*. A path in *G* is an **alternating path**, if its edges are alternately not in *M* and in *M* (or in *M* and not in *M*). An **augmenting path** *P* is an alternating path that starts and ends at two distinct exposed vertices. A

matching augmentation along an augmenting path *P* is the operation of replacing *M* with a new matching $M_1 = M \oplus P = (M \setminus P) \cup (P \setminus M)$.



The algorithm uses the following fact:

- Matching *M* is not maximum if and only if there exists an *M*-augmenting path in *G*,.[2][3]

Here is the high-level algorithm.

```
  INPUT:  Graph G, initial matching M on G
   OUTPUT: maximum matching M* on G
A1 function find_maximum_matching( G, M ) : M*
A2     P ← find_augmenting_path( G, M )
A3     if P is non-empty then
A4          return find_maximum_matching( G, augment M along P )
A5     else
A6          return M
A7     end if
A8 end function
```

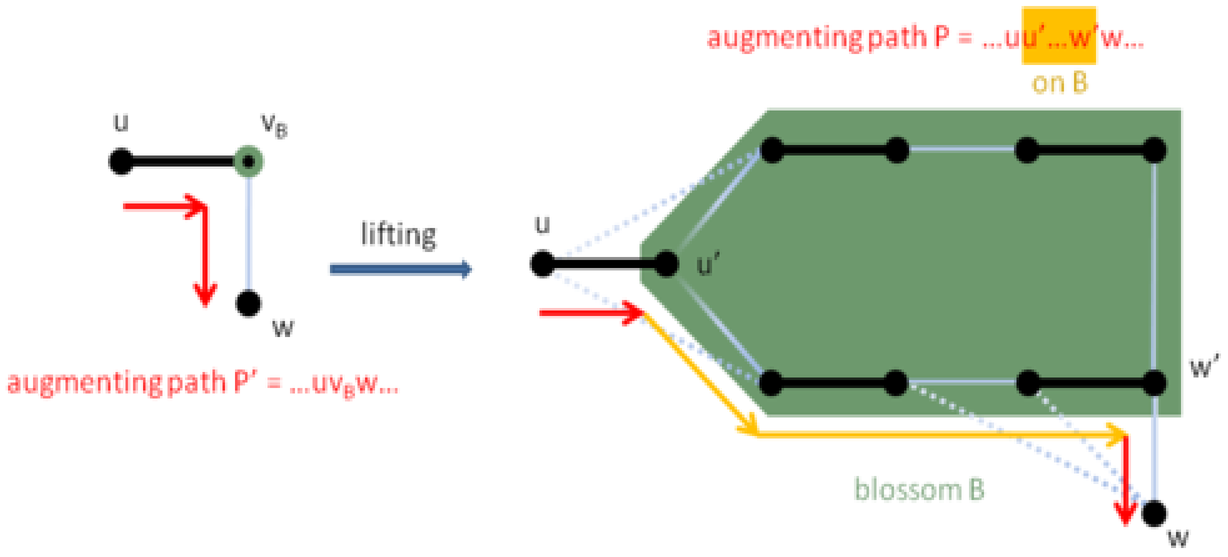
The subroutine to find an augmenting path uses blossoms and contractions.

Blossoms and contractions

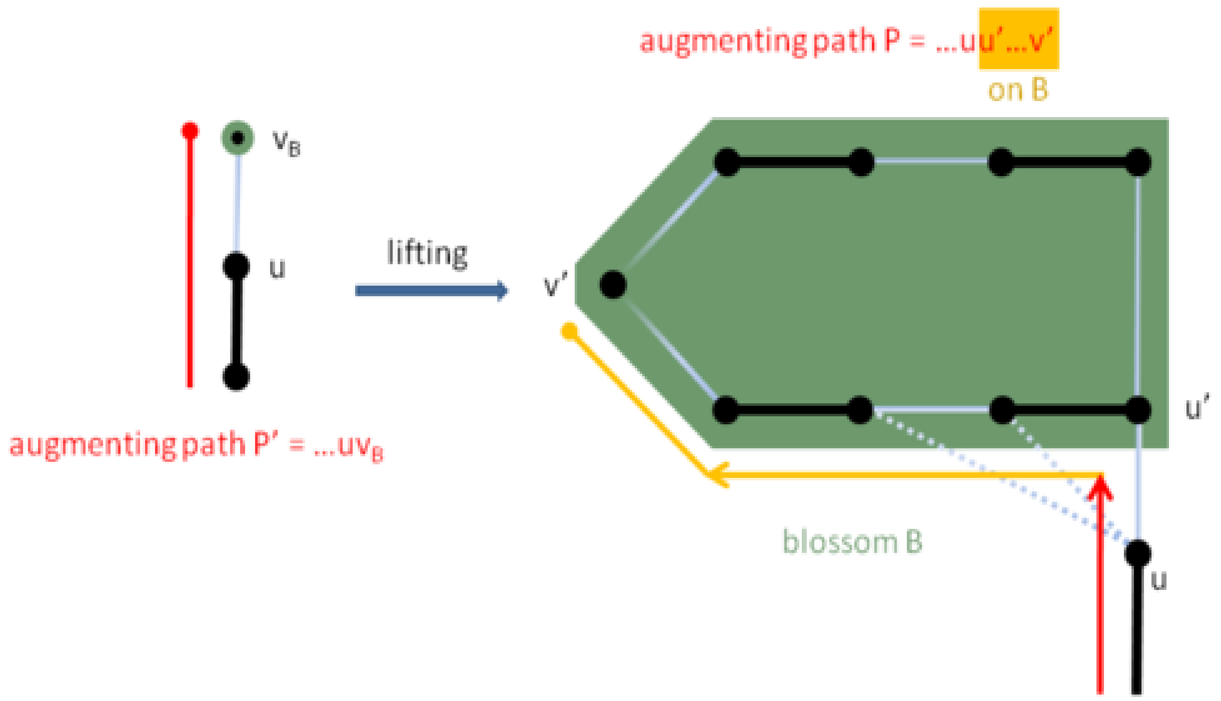
Given *G* = (*V*, *E*) and a matching *M* of *G*, a *blossom B* is a cycle in *G* consisting of *2k + 1* edges of which exactly *k* belong to *M*. We use *G'*, the **contracted graph**, to denote the graph obtained from *G* by contracting every edge of *B*. We use *M'*, the **contracted matching**, to denote the corresponding matching of *G'*.

If P' is a M'-augmenting path in G' then P' can be **lifted**to a M-augmenting path in G by undoing the contraction by B so that the segment of P' (if any) traversing through vB is replaced by an appropriate segment traversing through B. In more detail:

- if P' traverses through a segment $u \to vB \to w$ in G', then this segment is replaced with the segment $u \to (u' \to ... \to w') \to w$ in G, where blossom vertices u' and w' and the side of B, $(u' \to ... \to w')$, going from u' tow' are chosen to ensure that the new path is still alternating (u' is exposed with respect to $M \cap B$, $\{w', w\} \in E \setminus M$).


augmenting path P = ...uu'...w'w...
on B
u
v_B
w
lifting
augmenting path P' = ...uv_Bw...
u
u'
w'
w
blossom B

- if P' has an endpoint vB, then the path segment $u \to vB$ in G' is replaced with the segment $u \to (u' \to ... \to v')$ in G, where blossom vertices u' and v' and the side of B, $(u' \to ... \to v')$, going from u' to v' are chosen to ensure that the path is alternating (v' is exposed, $\{u', u\} \in E \setminus M$).


augmenting path P = ...uu'...v'
on B
v_B
u
lifting
v'
u'
augmenting path P' = ...uv_B
blossom B
u

The algorithm uses the following fact (using the notations from above):

- If *B* is a blossom, then *G′* contains a *M′*-augmenting path if and only if *G* contains a *M*-augmenting path,.[4] In addition, *M′*-augmenting paths in *G′* correspond to *M*-augmenting paths in*G*.

Thus blossoms can be contracted and search performed in the contracted graphs. This reduction is at the heart of Edmonds's algorithm.

[edit]Finding an augmenting path

The search for augmenting path uses an auxiliary data structure consisting of a forest *F* whose individual trees correspond to specific portions of the graph *G*. Using the structure, the algorithm either (1) finds an augmenting path or (2) finds a blossom and recurses onto the corresponding contracted graph or (3) concludes there are no augmenting paths. The auxiliary structure is built by an incremental procedure discussed next,.[4]

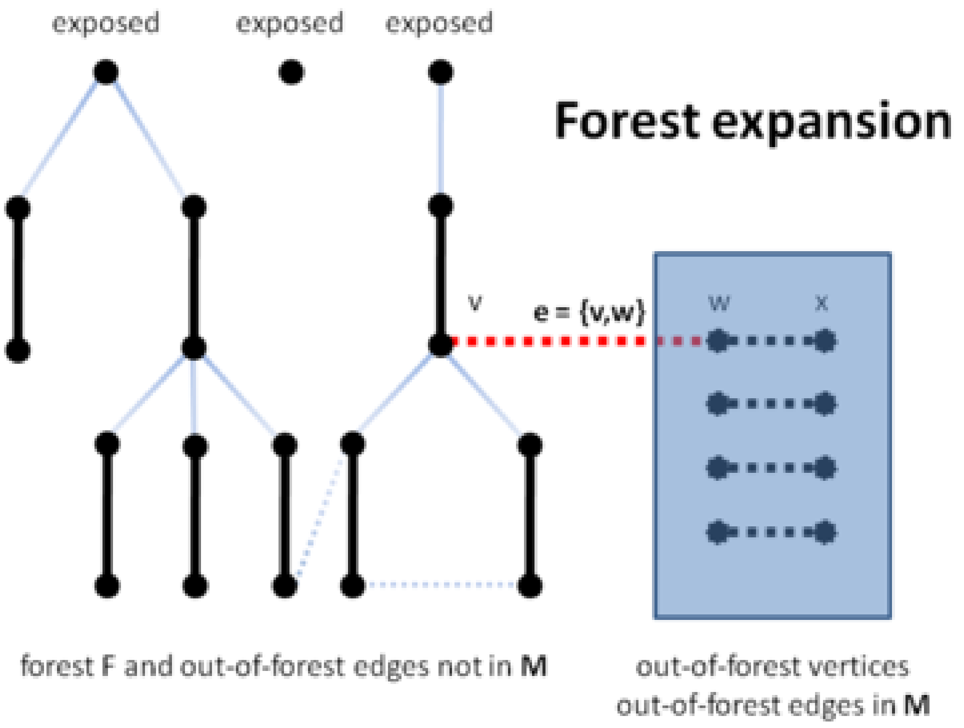
The construction procedure considers vertices *v* and edges *e* in *G* and incrementally updates *F* as appropriate. If *v* is in a tree *T* of the forest, we let *root(v)* denote the root of *T*. If both *u*and *v* are in the same tree *T* in *F*, we let *distance(u,v)* denote the length of the unique path from *u* to *v* in *T*.

```
   INPUT:  Graph G, matching M on G
    OUTPUT: augmenting path P in G or empty path if none found
B01 function find_augmenting_path( G, M ) : P
B02    F ← empty forest
B03    unmark all vertices and edges in G, mark all edges of M
B05    for each exposed vertex v do
B06        create a singleton tree { v } and add the tree to F
```

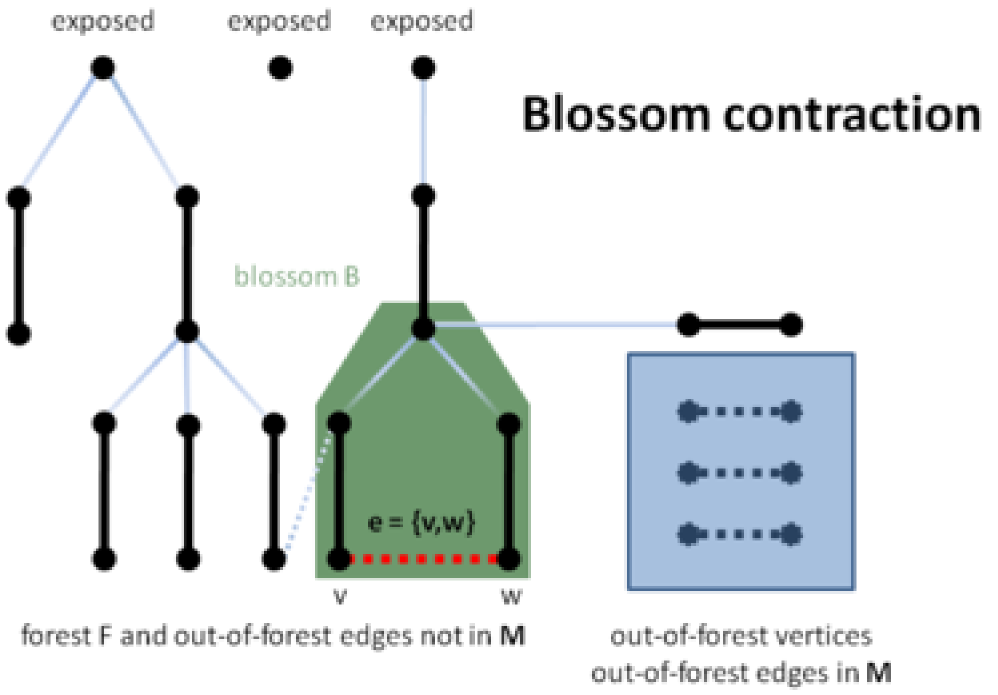
```
B07    end for
B08    while there is an unmarked vertex v in F with distance( v, root( v ) ) even do
B09        while there exists an unmarked edge e = { v, w } do
B10            if w is not in F then
                   // Update F.
B11                x ← vertex matched to w in M
B12                add edges { v, w } and { w, x } to the tree of v
B13            else
B14            if distance( w, root( w ) ) is odd then
B15                do nothing
B16            else
B17            if root( v ) ≠ root( w ) then
                   // Report an augmenting path in F ⋃ { e }.
B18                P ← path ( root( v ) → ... → v ) → ( w → ... → root( w ) )
B19                return P
B20            else
                   // Contract a blossom in G and look for the path in the contracted graph.
B21                B ← blossom formed by e and edges on the path v → w in T
B22                G’, M’ ← contract G and M by B
B23                P’ ← find_augmenting_path( G’, M’ )
B24                P ← lift P’ to G
B25                return P
B26            end if
B27            mark edge e
B28        end while
B29        mark vertex v
B30    end while
B31    return empty path
B32 end function
```

[edit]Examples

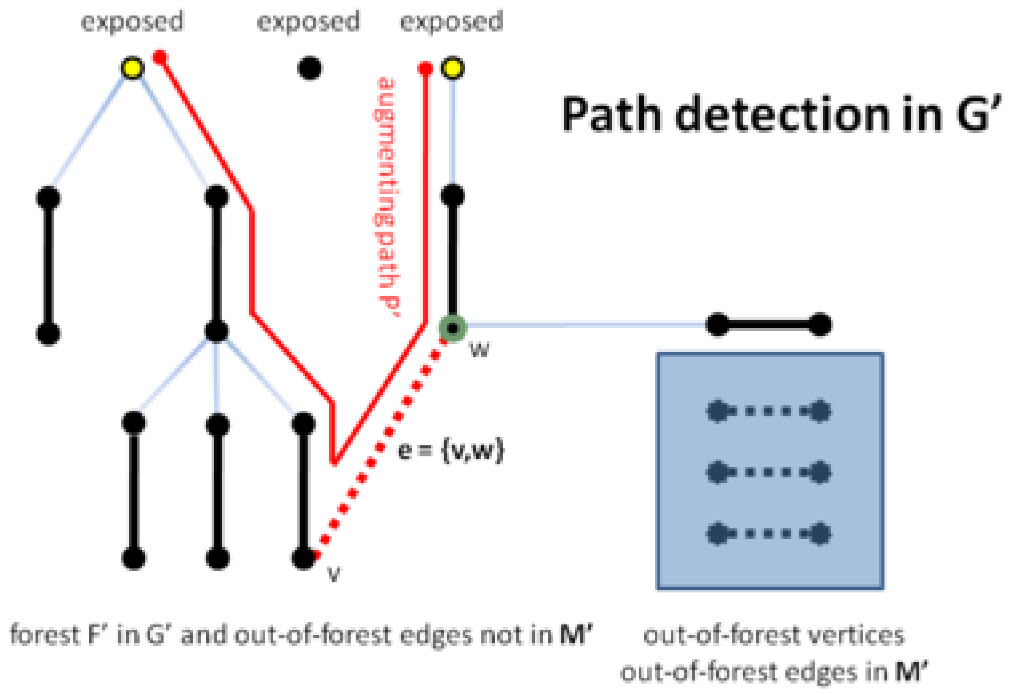
The following four figures illustrate the execution of the algorithm. We use dashed lines to indicate edges that are currently not present in the forest. First, the algorithm processes an out-of-forest edge that causes the expansion of the current forest (lines B10 – B12).

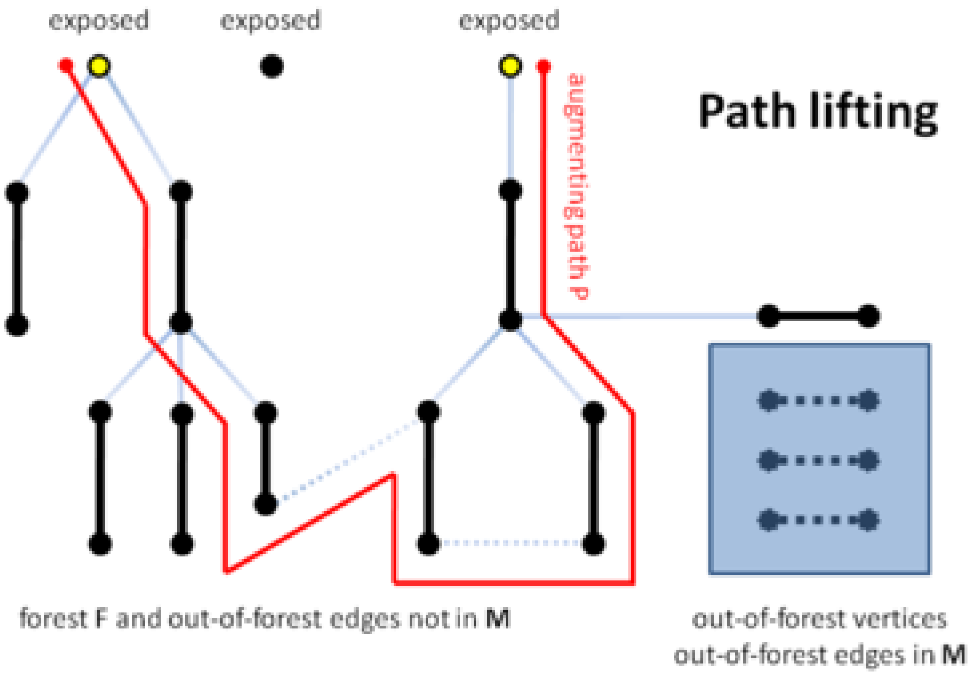


Next, it detects a blossom and contracts the graph (lines B20 – B22).



Finally, it locates an augmenting path P′ in the contracted graph (line B23) and lifts it to the original graph (line B24). Note that the ability of the algorithm to contract blossoms is crucial here; the algorithm can not find *P* in the original graph directly because only out-of-forest edges between vertices at even distances from the roots are considered on line B17 of the algorithm.





[edit]Analysis

The forest *F* constructed by the *find_augmenting_path()* function is an alternating forest, .[5]

- a tree *T* in *G* is an **alternating tree** with respect to *M*, if
    - *T contains exactly one exposed vertex r called the tree root*
    - every vertex at an odd distance from the root has exactly two incident edges in *T*, and
    - all paths from *r* to leaves in *T* have even lengths, their odd edges are not in *M* and their even edges are in *M*.
- a forest *F* in *G* is an **alternating forest** with respect to *M*, if
    - its connected components are alternating trees, and
    - every exposed vertex in *G* is a root of an alternating tree in *F*.

Each iteration of the loop starting at line B09 either adds to a tree *T* in *F* (line B10) or finds an augmenting path (line B17) or finds a blossom (line B21). It is easy to see that the running time is $O(|V|4)$. Micali and Vazirani [6] show an algorithm that constructs maximum matching in $O(|E||V|1/2)$ time.

[edit]Bipartite matching

The algorithm reduces to the standard algorithm for matching in bipartite graphs [3] when *G* is bipartite. As there are no odd cycles in *G* in that case, blossoms will never be found and one can simply remove lines B21 – B29 of the algorithm.

[edit]Weighted matching

The matching problem can be generalized by assigning weights to edges in *G* and asking for a set *M* that produces a matching of maximum (minimum) total weight. The weighted matching problem can be solved by a combinatorial algorithm that uses the unweighted Edmonds's algorithm as a subroutine.[2] Kolmogorov provides an efficient C++ implementation of this.[7]

1.6.1.2. Maximum cardinality bipartite matching

In the mathematical field of graph theory, a bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V; that is, U and V are independent sets. Equivalently, a bipartite graph is a graph that does not contain any odd-length cycles.

The two sets U and V may be thought of as a coloring of the graph with two colors: if we color all nodes in U blue, and all nodes in V green, each edge has endpoints of differing colors, as is required in the graph coloring problem. In contrast, such a coloring is impossible in the case of a nonbipartite graph, such as a triangle: after one node is colored blue and

another green, the third vertex of the triangle is connected to vertices of both colors, preventing it from being assigned either color.
One often writes G = (U, V, E) to denote a bipartite graph whose partition has the parts U and V. If |U| =|V|, that is, if the two subsets have equal cardinality, then G is called a balanced bipartite graph.

Any graph with no odd cycles is bipartite. As a consequence of this:
Every tree is bipartite.
Cycle graphs with an even number of vertices are bipartite.
Any planar graph where all the faces in its planar representation consist of an even number of edges is bipartite. Special cases of this are grid graphs and squaregraphs, in which every inner face consists of 4 edges.

If a bipartite graph is connected, its bipartition can be defined by the parity of the distances from any arbitrarily chosen vertex v: one subset consists of the vertices at even distance to v and the other subset consists of the vertices at odd distance to v.
Thus, one may efficiently test whether a graph is bipartite by using this parity technique to assign vertices to the two subsets U and V, separately within each connected component of the graph, and then examine each edge to verify that it has endpoints assigned to different subsets.

Matching problems are often concerned with bipartite graphs. Finding a **maximum bipartite matching**[2] (often called a **maximum cardinality bipartite matching**) in a bipartite graph$G = (V = (X,Y),E)$ is perhaps the simplest problem. The **augmenting path algorithm** finds it by finding an augmenting path from each $x \in X$ to Y and adding it to the matching if it exists. As each path can be found in $O(E)$ time, the running time is $O(VE)$. This solution is equivalent to adding a *super source* s with edges to all vertices in X, and a *super sink* t with edges from all vertices in Y, and finding a maximal flow from s to t. All edges with flow from X to Y then constitute a maximum matching. An improvement over this is the Hopcroft-Karp algorithm, which runs in $O(\sqrt{V}E)$ time.
Another approach is based on the fast matrix multiplication algorithm and gives $O(V2.376)$ complexity,[3] which is better in theory for sufficientlydense graphs, but in practice the algorithm is slower.
In a weighted bipartite graph, each edge has an associated value. A **maximum weighted bipartite matching**[2] is defined as a perfect matching where the sum of the values of the edges in the matching have a maximal value. If the graph is not complete bipartite, missing edges are inserted with value zero. Finding such a matching is known as the assignment problem. It can be solved by using a modified shortest path search in the augmenting path algorithm. If the Bellman-Ford algorithm is used, the running time becomes $O(V2E)$, or the edge cost can be shifted with a potential to achieve $O(V2\log V + VE)$ running time with the Dijkstra algorithm and Fibonacci heap. The remarkable Hungarian algorithm solves the assignment problem and it was one of the beginnings of

combinatorial optimization algorithms. The original approach of this algorithm needs $O(V2E)$ running time, but it could be improved to $O(V2\log V + VE)$time with extensive use of priority queues.

Implementación:

```
typedef vector<int> VI;
typedef vector<VI> VVI;

bool FindMatch(int i, const VVI &w, VI &mr, VI &mc, VI &seen) {
  for (int j = 0; j < w[i].size(); j++) {
    if (w[i][j] && !seen[j]) {
      seen[j] = true;
      if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)) {
        mr[i] = j;
        mc[j] = i;
        return true;
      }
    }
  }
  return false;
}

int BipartiteMatching(const VVI &w, VI &mr, VI &mc) {
  mr = VI(w.size(), -1);
  mc = VI(w[0].size(), -1);

  int ct = 0;
  for (int i = 0; i < w.size(); i++) {
    VI seen(w[0].size());
    if (FindMatch(i, w, mr, mc, seen)) ct++;
  }
  return ct;
}
```

1.6.2. Stable marriage problem

In mathematics, the **stable marriage problem** (**SMP**) is the problem of finding a **stable matching** between two sets of elements given a set of preferences for each element. A matching is a mapping from the elements of one set to the elements of the other set. A matching is stable whenever it is *not* the case that both:

1. some given element *A* of the first matched set prefers some given element *B* of the second matched set over the element to which A is already matched, and

2. *B also prefers A over the element to which B is already matched*

In other words, a matching is stable when there does not exist any alternative pairing (A, B) in which both A and B are individually better off than they would be with the element to which they are currently matched.

The problem is commonly stated as:

> Given *n* men and *n* women, where each person has ranked all members of the opposite sex with a unique number between 1 and *n* in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. If there are no such people, all the marriages are "stable".

In 1962, David Gale and Lloyd Shapley proved that, for any equal number of men and women, it is always possible to solve the SMP and make all marriages stable. They presented analgorithm to do so.[1][2]

The **Gale-Shapley algorithm** involves a number of "rounds" (or "iterations") where each unengaged man "proposes" to the most-preferred woman to whom he has not yet proposed. Each woman then considers all her suitors and tells the one she most prefers "Maybe" and all the rest of them "No". She is then provisionally "engaged" to the suitor she most prefers so far, and that suitor is likewise provisionally engaged to her. In the first round, first *a*) each unengaged man proposes to the woman he prefers most, and then *b*) each woman replies "maybe" to her suitor she most prefers and "no" to all other suitors. In each subsequent round, first *a*) each unengaged man proposes to the most-preferred woman to whom he has not yet proposed (regardless of whether the woman is already engaged), and then *b*) each woman replies "maybe" to her suitor she most prefers (whether her existing provisional partner or someone else) and rejects the rest (again, perhaps including her current provisional partner). The provisional nature of engagements preserves the right of an already-engaged woman to "trade up" (and, in the process, to "jilt" her until-then partner).

This algorithm guarantees that:

Everyone gets married

> Once a woman becomes engaged, she is always engaged to someone. So, at the end, there cannot be a man and a woman both unengaged, as he must have proposed to her at some point (since a man will eventually propose to everyone, if necessary) and, being unengaged, she would have to have said yes.

The marriages are stable

> Let Alice be a woman and Bob be a man who are both engaged, but not to each other. Upon completion of the algorithm, it is not possible for both Alice and Bob to prefer each other over their current partners. If Bob prefers Alice to his current partner, he must have proposed to Alice before he

proposed to his current partner. If Alice accepted his proposal, yet is not married to him at the end, she must have dumped him for someone she likes more, and therefore doesn't like Bob more than her current partner. If Alice rejected his proposal, she was already with someone she liked more than Bob.

Algorithm

```
function stableMatching {
    Initialize all m ∈ M and w ∈ W to free
    while ∃ free man m who still has a woman w to propose to {
       w = m's highest ranked such woman who he has not proposed to yet
       if w is free
         (m, w) become engaged
       else some pair (m', w) already exists
         if w prefers m to m'
           (m, w) become engaged
           m' becomes free
         else
           (m', w) remain engaged
    }
}
```

Optimality of the Solution

While the solution is stable, it is not necessarily optimal from all individuals' points of view. The traditional form of the algorithm is optimal for the initiator of the proposals and the stable, suitor-optimal solution may or may not be optimal for the reviewer of the proposals. An informal proof by example is as follows:
There are three suitors (A,B,C) and three reviewers (X,Y,Z) which have preferences of:

```
A: YXZ  B: ZYX  C: XZY  X: BAC  Y: CBA  Z: ACB
```

There are 3 stable solutions to this matching arrangement:

```
suitors get their first choice and reviewers their third (AY, BZ, CX)
 all participants get their second choice (AX, BY, CZ)
 reviewers get their first choice and suitors their third (AZ, BX, CY)
```

All three are stable because instability requires both participants to be happier with an alternative match. Giving one group their first choices ensures that the matches are stable because they would be unhappy with any other proposed match. Giving everyone their second choice ensures that any other match would be disliked by one of the parties. The algorithm converges in a single round on the suitor-optimal solution because each reviewer receives exactly one proposal, and therefore selects that proposal as its best choice, ensuring that each suitor has an accepted offer, ending the match.

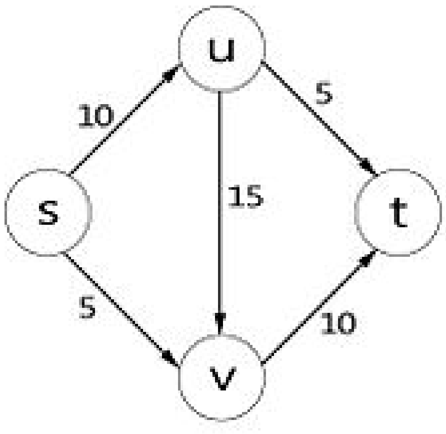
This asymmetry of optimality is driven by the fact that the suitors have the entire set to choose from, but reviewers choose between a limited subset of the suitors at any one time.

1.7. Max-flow min-cut

In optimization theory, the **maximum flow problem** is to find a feasible flow through a single-source, single-sink flow network that is maximum.

The maximum flow problem can be seen as a special case of more complex network flow problems, such as the circulation problem. The maximum value of an s-t flow is equal to the minimum capacity of an s-t cut in the network, as stated in the max-flow min-cut theorem.

Definition



A flow network, with source s and sink t. The numbers next to the edge are the capacities.

Let $N=(V,E)$ be a network with $s,t\in V$ being the source and the sink of N respectively.

The **capacity** of an edge is a mapping $c:E\to\mathbb{R}^+$, denoted by c_{uv} or $c(u,v)$. It represents the maximum amount of flow that can pass through an edge.

A **flow** is a mapping $f:E\to\mathbb{R}^+$, denoted by f_{uv} or $f(u,v)$, subject to the following two constraints:

1. $f_{uv}\le c_{uv}$, for each $(u,v)\in E$ (capacity constraint: the flow of an edge cannot exceed its capacity)
2. $\sum_{u:(u,v)\in E} f_{uv}=\sum_{u:(v,u)\in E} f_{vu}$, for each $v\in V\setminus\{s,t\}$ (conservation of flows: the sum of the flows entering a node must equal the sum of the flows exiting a node, except for the source and the sink nodes)

The **value of flow** is defined by $|f|=\sum_{v\in V} f_{sv}$, where s is the source of N. It represents the amount of flow passing from the source to the sink.

The **maximum flow problem** is to maximize $|f|$, that is, to route as much flow as possible from s to t.

Application

[edit]Multi-source multi-sink maximum flow problem

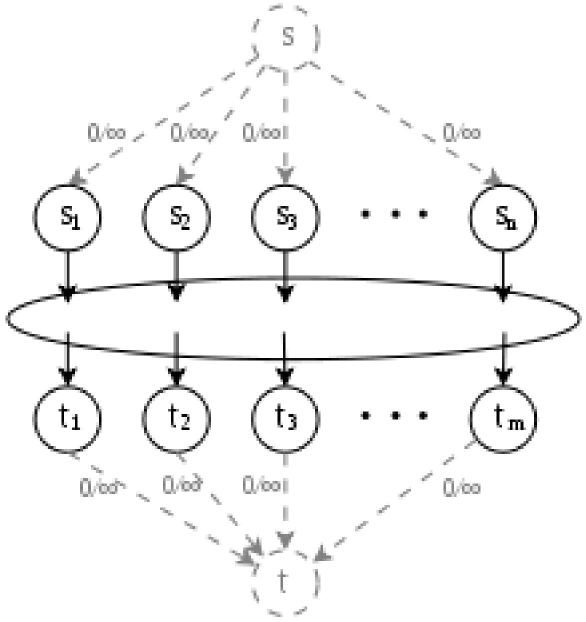


Fig. 4.1.1. Transformation of a multi-source multi-sink maximum flow problem into a single-source single-sink maximum flow problem

Given a network *N* = (*V*,*E*) with a set of sources *S* = {*s*1, ..., *s*n} and a set of sinks *T* = {*t*1, ..., *t*m} instead of only one source and one sink, we are to find the maximum flow across *N*. We can transform the multi-source multi-sink problem into a maximum flow problem by adding a*consolidated source* connecting to each vertex in *S* and a *consolidated sink* connected by each vertex in *T* with infinite capacity on each edge (See Fig. 4.1.1.).

[edit]Minimum path cover in directed acyclic graph

Given a directed acyclic graph *G* = (*V*, *E*), we are to find the minimum number of paths to cover each vertex in *V*. We can construct a bipartite graph *G*' = (*Vout*∪*Vin*, *E*') from *G*, where

1. *Vout = {v∈V: v has positive out-degree}.*
2. *Vin = {v∈V: v has positive in-degree}.*
3. *E' = {(u,v)∈(Vout,Vin): (u,v)∈E}.*

Then it can be shown that *G*' has a matching of size *m* if and only if there exists *n*-*m* paths that cover each vertex in *G*, where *n* is the number of vertices in *G*. Therefore, the problem can be solved by finding the maximum cardinality matching in *G'* instead.

[edit]Maximum cardinality bipartite matching

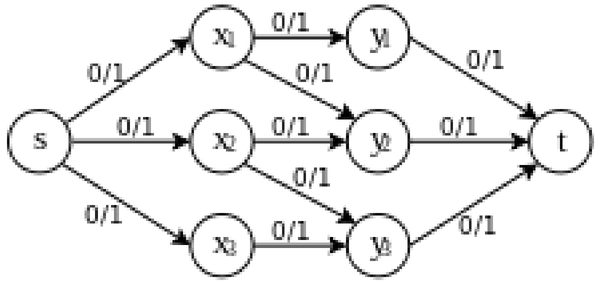


Fig. 4.3.1. Transformation of a maximum bipartite matching problem into a maximum flow problem

Given a bipartite graph *G* = (*X*∪*Y*, *E*), we are to find a maximum cardinality matching in *G*, that is a matching that contains the largest possible number of edges. This problem can be transformed into a maximum flow problem by constructing a network *N* = (*X*∪*Y*∪{*s*,*t*}, *E*' }, where

1. *E' contains the edges in G directed from X to Y.*
2. (*s*,*x*)∈*E*' for each *x*∈*X* and (*y*,*t*)∈*E*' for each *y*∈*Y*.
3. *c(e) = 1 for each e∈E' (See Fig. 4.3.1).*

Then the value of the maximum flow in *N* is equal to the size of the maximum matching in *G*.

[edit]Maximum flow problem with vertex capacities

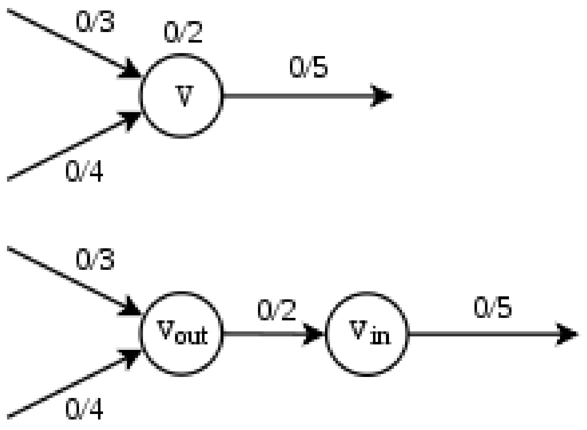


Fig. 4.4.1. Transformation of a maximum flow problem with vertex capacities constraint into the original maximum flow problem by node splitting

Given a network *N* = (*V*, *E*), in which there is capacity at each node in addition to edge capacity, that is, a mapping *c*: *V*→*R*+, denoted by *c*(*v*), such that the flow *f* has to satisfy not only the capacity constraint and the conservation of flows, but also the vertex capacity constraint

$\Sigma_{i \in V} f_{iv} \leq c(v)$ for each $v \in V \setminus \{s,t\}$.
In other words, the amount of flow passing through a vertex cannot exceed its capacity.
To find the maximum flow across N, we can transform the problem into the maximum flow problem in the original sense by expanding N. First, each $v \in V$ is replaced by v_{in} and v_{out}, where v_{in} is connected by edges going into v and v_{out} is connected to edges coming out from v, then assign capacity $c(v)$ to the edge connecting v_{in} and v_{out} (See Fig. 4.4.1). In this expanded network, the vertex capacity constraint is removed and therefore the problem can be treated as the original maximum flow problem.

[edit]Maximum independent path

Given a directed graph $G = (V, E)$ and two vertices s and t, we are to find the maximum number of independent paths from s to t. Two paths are said to be independent if they do not have a vertex in common apart from s and t. We can construct a network $N = (V, E)$ from G with vertex capacities, where

1. *s and t are the source and the sink of N respectively.*
2. *$c(v) = 1$ for each $v \in V$.*
3. *$c(e) = 1$ for each $e \in E$.*

Then the value of the maximum flow is equal to the maximum number of independent paths from s to t.

[edit]Maximum edge-disjoint path

Given a directed graph $G = (V, E)$ and two vertices s and t, we are to find the maximum number of edge-disjoint paths from s to t. This problem can be transformed to a maximum flow problem by constructing a network $N = (V, E)$ from G with s and t being the source and the sink of N respectively and assign each edge with unit capacity.

Implementación:

```
public class MaxFlow
{
      static class Edge
      {
            int from, to, cap, flow, index;

            Edge(int fromi, int toi, int capi, int flowi, int indexi)
            {
                  from = fromi;
                  to = toi;
                  cap = capi;
                  flow = flowi;
                  index = indexi;
            }
      }
```

```
	static class PushRelabel
	{
		int N;
		ArrayList < ArrayList <Edge> > G;
		long[] excess;
		int[] dist, count;
		boolean[] active;
		LinkedList <Integer> Q = new LinkedList <Integer> ();

		PushRelabel(int N1)
		{
			N = N1;
			G = new ArrayList < ArrayList <Edge> > (N);
			for(int i = 0; i < N; i++)
				G.add(new ArrayList <Edge> ());
			excess = new long[N];
			dist = new int[N];
			active = new boolean[N];
			count = new int[2 * N];
		}

		void AddEdge(int from, int to, int cap)
		{
			int cambio = from == to ? 1 : 0;
			G.get(from).add(new Edge(from, to, cap, 0, G.get(to).size()
+ cambio));
			G.get(to).add(new Edge(to, from, 0, 0, G.get(from).size() -
1));
		}

		void Enqueue(int v)
		{
			if (!active[v] && excess[v] > 0)
			{
				active[v] = true;
				Q.add(v);
			}
		}

		void Push(Edge e)
		{
			long amt = Math.min(excess[e.from], e.cap - e.flow);
			if(dist[e.from] <= dist[e.to] || amt == 0)
				return;
			e.flow += amt;
			G.get(e.to).get(e.index).flow -= amt;
			excess[e.to] += amt;
		excess[e.from] -= amt;
		Enqueue(e.to);
	}

	void Gap(int k)
	{
		for(int v = 0; v < N; v++)
		{
			if(dist[v] < k)
				continue;
			count[dist[v]]--;
			dist[v] = Math.max(dist[v], N + 1);
			count[dist[v]]++;
			Enqueue(v);
		}
	}

	void Relabel(int v)
	{
		count[dist[v]]--;
		dist[v] = 2 * N;
		for (Edge e : G.get(v))
			if (e.cap - e.flow > 0)
				dist[v] = Math.min(dist[v], dist[e.to] + 1);
		count[dist[v]]++;
		Enqueue(v);
	}

	void Discharge(int v)
	{
		for(Edge e : G.get(v))
		{
			if(excess[v] <= 0)
				break;
			Push(e);
		}
		if(excess[v] > 0)
		{
			if(count[dist[v]] == 1)
				Gap(dist[v]);
			else
				Relabel(v);
		}
	}

	long GetMaxFlow(int s, int t)
	{
```

```
                  count[0] = N - 1;
                  count[N] = 1;
                  dist[s] = N;
                  active[s] = active[t] = true;
                  for (Edge e : G.get(s))
                  {
                        excess[s] += e.cap;
                        Push(e);
                  }
                  while (!Q.isEmpty())
                  {
                        int v = Q.poll();
                        active[v] = false;
                        Discharge(v);
                  }
                  long totflow = 0;
                  for (Edge e : G.get(s))
                        totflow += e.flow;
                              return totflow;
            }
      }
}
```

Ford-fulkerson

```
#include<memory.h>


#define min(a, b) (((a)<(b))?(a):(b))
#define INF 1000000000

//global variables
int n;
int cap[MAXN][MAXN];
bool v[MAXN];
int source, sink;

//returns the capacity of the path. zero if there isn't one
int augment(int x, int minedge)
{
  if(x==sink) return minedge;
  v[x]=true;
  for(int i=0;i<n;i++)
  {
    if(!v[i] && cap[x][i])
    {
      int ret=augment(i, min(minedge, cap[x][i]));
      if(ret){cap[i][x]-=ret; cap[x][i]+=ret; return ret;}
    }
  }
  return 0;
}

//returns the maximum flow
int maxFlow()
{
  int ret=0;
  while(true)
  {
    memset(v, false, sizeof(v));
    int flow=augment(source, INF);
    if(!flow) break;
    ret+=flow;
  }
  return ret;
}
```

1.8. Eulerian path

In graph theory, an **Eulerian path** is a path in a graph which visits each edge exactly once. Similarly, an **Eulerian circuit** is an Eulerian path which starts and ends on the same vertex. They were first discussed by Leonhard Euler while solving the famous Seven Bridges of Königsberg problem in 1736. Mathematically the problem can be stated like this:

> Given the graph on the right, is it possible to construct a path (or a cycle, i.e. a path starting and ending on the same vertex) which visits each edge exactly once?

Euler proved that a necessary condition for the existence of Eulerian circuits is that all vertices in the graph have an even degree, and stated without proof that connected graphs with all vertices of even degree have an Eulerian circuit. The first complete proof of this latter claim was published in 1873 byCarl Hierholzer.[1]

The term **Eulerian graph** has two common meanings in graph theory. One meaning is a graph with an Eulerian circuit, and the other is a graph with every vertex of even degree. These definitions coincide for connected graphs.[2]

For the existence of Eulerian paths it is necessary that no more than two vertices have an odd degree; this means the Königsberg graph is *not* Eulerian. If there are no vertices of odd degree, all Eulerian paths are circuits. If there are exactly two vertices of odd degree, all Eulerian paths start at one of them and end at the other. Sometimes a graph that has an Eulerian path, but not an Eulerian circuit (in other words, it is an open path, and does not start and end at the same vertex) is called **semi-Eulerian**.

Properties

- A connected undirected graph is Eulerian if and only if every graph vertex has an even degree.
- An undirected graph is Eulerian if it is connected and can be decomposed into edge-disjoint cycles.
- If an undirected graph *G* is Eulerian then its line graph *L*(*G*) is Eulerian too.
- A directed graph is Eulerian if it is strongly connected and every vertex has equal in degree and out degree.
- A directed graph is Eulerian if it is strongly connected and can be decomposed into edge-disjoint directed cycles.
- An Eulerian path exists in a directed graph if and only if the graph's underlying undirected graph is connected, at most one vertex has out degree-in degree=1, at most one vertex has in degree-out degree=1 and every other vertex has equal in degree and out degree.
- An undirected graph is **traversable** if it is connected and at most two vertices in the graph are of odd degree.

[edit]Constructing Eulerian paths and circuits
Consider a graph known to have all edges in the same component and at most two vertices of odd degree. We can construct an Eulerian path out of this graph by using Fleury's algorithm, which dates to 1883. We start with a vertex of odd degree—if the graph has none, then start with any vertex. At each step we move across an edge whose deletion would not disconnect the graph, unless we have no choice, then we delete that edge. At the end of the algorithm there are no edges left, and the sequence of edges we moved across forms an Eulerian cycle if the graph has no vertices of odd degree; or an Eulerian path if there are exactly two vertices of odd degree.

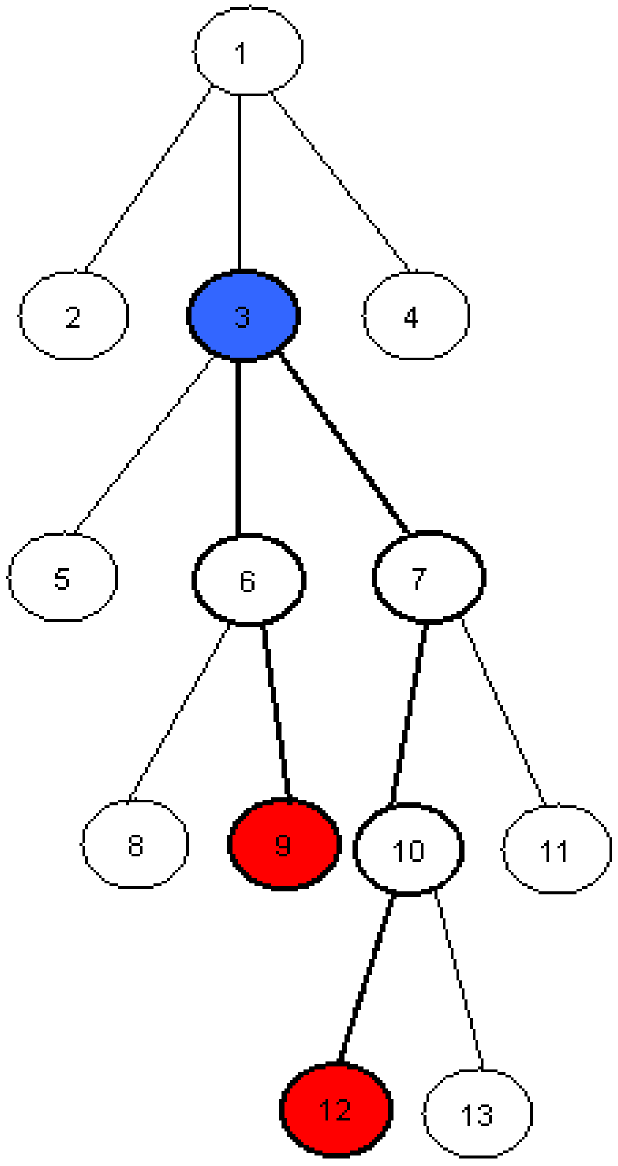
1.9. Lowest common ancestor

The **lowest common ancestor** (**LCA**) is a concept in graph theory and computer science. Let T be a rooted tree with n nodes. The lowest common ancestor is defined between two nodes *v* and *w* as the lowest node in T that has both *v* and *w* as descendants (where we allow a node to be a descendant of itself).
The LCA of *v* and *w* in T is the shared ancestor of *v* and *w* that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from *v* to *w* can be computed as the

distance from the root to *v*, plus the distance from the root to *w*, minus twice the distance from the root to their lowest common ancestor.
In a tree data structure where each node points to its parent, the lowest common ancestor can be easily determined by finding the first intersection of the paths from *v* and *w* to the root. In general, the computational time required for this algorithm is O(*h*) where *h* is the height of the tree (length of longest path from a leaf to the root). However, there exist several algorithms for processing trees so that lowest common ancestors may be found more quickly, in constant time per query after a linear time preprocessing stage.
The problem of finding the Lowest Common Ancestor (LCA) of a pair of nodes in a rooted tree has been studied more carefully in the second part of the 20th century and now is fairly basic in algorithmic graph theory. This problem is interesting not only for the tricky algorithms that can be used to solve it, but for its numerous applications in string processing and computational biology, for example, where LCA is used with suffix trees or other tree-like structures. Harel and Tarjan were the first to study this problem more attentively and they showed that after linear preprocessing of the input tree LCA, queries can be answered in constant time. Their work has since been extended, and this tutorial will present many interesting approaches that can be used in other kinds of problems as well.

Let's consider a less abstract example of LCA: the tree of life. It's a well-known fact that the current habitants of Earth evolved from other species. This evolving structure can be represented as a tree, in which nodes represent species, and the sons of some node represent the directly evolved species. Now species with similar characteristics are divided into groups. By finding the LCA of some nodes in this tree we can actually find the common parent of two species, and we can determine that the similar characteristics they share are inherited from that parent.

Lowest Common Ancestor (LCA)
Given a rooted tree **T** and two nodes **u** and **v**, find the furthest node from the root that is an ancestor for both **u** and **v**. Here is an example (the root of the tree will be node 1 for all examples in this editorial):



$LCA_T(9, 12) = 3$

1.9.1. Tarjan

In computer science, Tarjan's off-line least common ancestors algorithm (more precisely, least should actually be lowest) is an algorithm for computing lowest common ancestorsfor pairs of nodes in a tree, based on the union-find data structure. The lowest common ancestor of two nodes *d* and *e* in a rooted tree *T* is the node *g* that is an ancestor of both *d* and *e*and that has the greatest depth in *T*. It is named after Robert Tarjan, who discovered the technique in 1979. Tarjan's algorithm is *offline*; that is, unlike other lowest common ancestor algorithms, it requires that all pairs of nodes for which the lowest common ancestor is desired must be specified in advance. The simplest version of the algorithm uses the union find data structure, which unlike other lowest common ancestor data structures can take more than constant time per operation when the number of pairs of nodes is similar in magnitude to the number of nodes. A later refinement by Gabow & Tarjan (1983) speeds the algorithm up to linear time.

[edit]Pseudocode

The pseudocode below determines the lowest common ancestor of each pair in *P*, given the root *r* of a tree in which the children of node *n* are in the set *n.children*. For this offline algorithm, the set *P* must be specified in advance. It uses the *MakeSet*, *Find*, and *Union* functions of a disjoint-set forest. *MakeSet(u)* removes *u* to a singleton set, *Find(u)* returns the standard representative of the set containing *u*, and *Union(u,v)* merges the set containing *u* with the set containing *v*. TarjanOLCA(*r*) is first called on the root *r*.

```
function TarjanOLCA(u)
     MakeSet(u);
     u.ancestor := u;
     for each v in u.children do
         TarjanOLCA(v);
         Union(u,v);
         Find(u).ancestor := u;
     u.colour := black;
     for each v such that {u,v} in P do
         if v.colour == black
             print "Tarjan's Least Common Ancestor of " + u +
                   " and " + v + " is " + Find(v).ancestor + ".";
```

Each node is initially white, and is colored black after it and all its children have been visited. The lowest common ancestor of the pair *{u,v}* is available as *Find(v).ancestor* immediately (and only immediately) after *u* is colored black, provided *v* is already black. Otherwise, it will be available later as *Find(u).ancestor*, immediately after *v* is colored black.

Implementación:

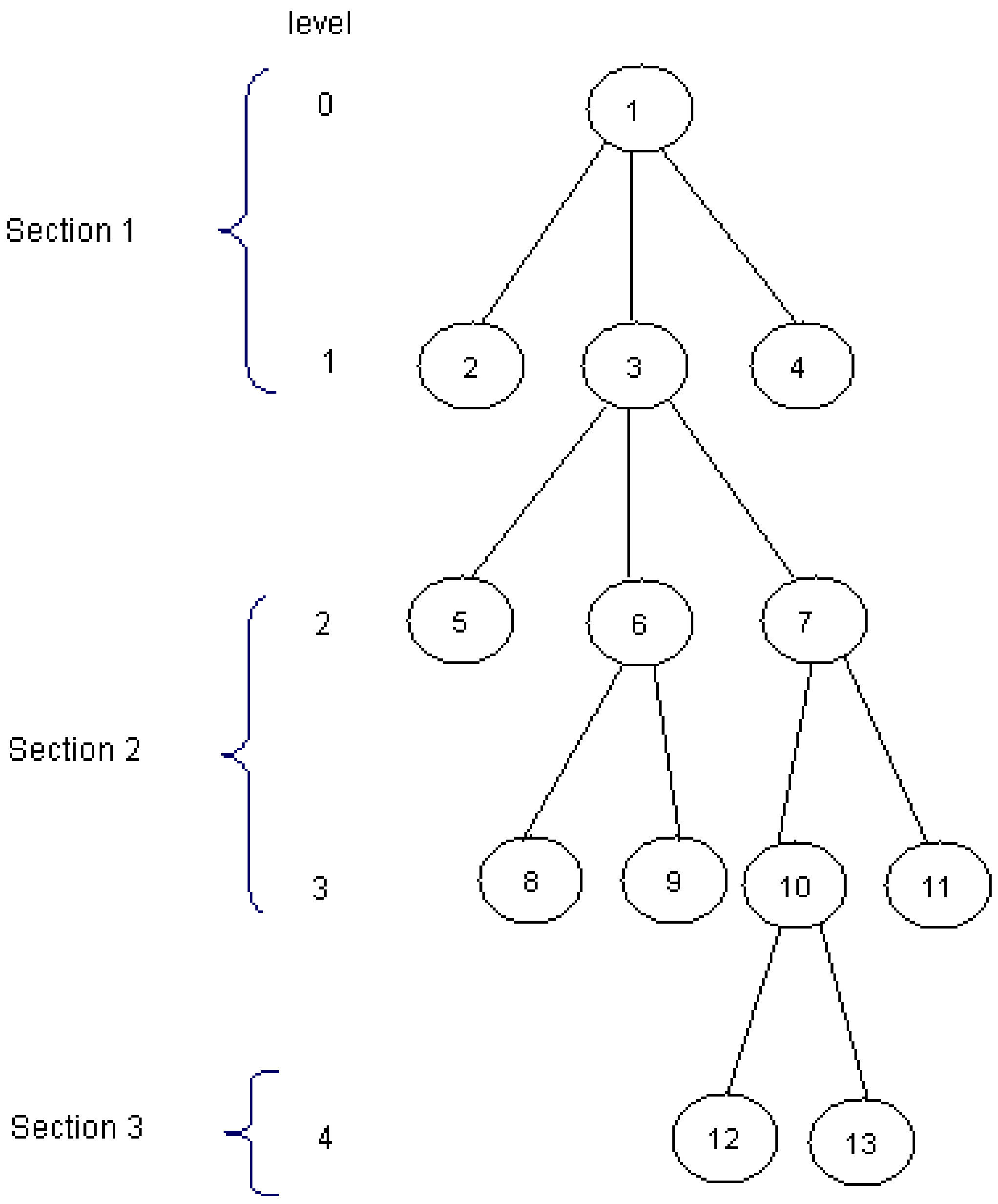
```
struct Query;

struct Nodo
{
    vector <Nodo*> adjacentes;
    vector <Query*> queries;
    int numero;
    long long distanciaCero;
    Nodo *parent;
    Nodo *ancestor;
    int rank;
    bool encontrado;

    void clear(int i)
    {
        numero = i;
        distanciaCero = 0;
        adjacentes.clear();
        queries.clear();
        encontrado = false;
    }
};

struct Query
{
    Nodo *a, *b;
    long long respuesta;
};

Nodo nodos[100001];
Query queries[100001];

void makeSet(Nodo *x)
{
     x->parent = x;
     x->rank   = 0;
}

Nodo* find(Nodo *x)
{
     if(x->parent == x)
        return x;
     else
     {
        x->parent = find(x->parent);
        return x->parent;
     }
}

int unir(Nodo *x, Nodo *y)
{
     Nodo *xRoot = find(x);
     Nodo *yRoot = find(y);
     if(xRoot->rank > yRoot->rank)
```

```
            yRoot->parent = xRoot;
        else if(xRoot->rank < yRoot->rank)
            xRoot->parent = yRoot;
        else if(xRoot != yRoot)
        {
            yRoot->parent = xRoot;
            xRoot->rank = xRoot->rank + 1;
        }
}

void tarjanOLCA(Nodo *u)
{
    makeSet(u);
    u->ancestor = u;
    for(int i = 0; i < u->adjacentes.size(); i++)
    {
        tarjanOLCA(u->adjacentes[i]);
        unir(u, u->adjacentes[i]);
        find(u)->ancestor = u;
    }
    u->encontrado = true;
    for(int i = 0; i < u->queries.size(); i++)
    {
        Query *actual = u->queries[i];
        Nodo *v;
        if(u == actual->a)
            v = actual->b;
        else
            v = actual->a;
        if(v->encontrado)
        {
            Nodo *ancestro = find(v)->ancestor;
            actual->respuesta = … procesar query aqui
        }
    }
}
```

1.9.2. Lingas

An <O(N), O(sqrt(N))> solution
Dividing our input into equal-sized parts proves to be an interesting way to solve the RMQ problem. This method can be adapted for the LCA problem as well. The idea is to split the tree in **sqrt(H)** parts, were **H** is the height of the tree. Thus, the first section will contain the levels numbered from **0 to sqrt(H) - 1**, the second will contain the levels numbered from **sqrt(H) to 2 ***

sqrt(H) - 1, and so on. Here is how the tree in the example should be divided:



Now, for each node, we should know the ancestor that is situated on the last level of the upper next section. We will preprocess this values in an array **P[1, MAXN]**. Here is how **P** should look like for the tree in the example (for simplity, for every node **i** in the first section let **P[i] = 1**):

P[1]	P[2]	P[3]	P[4]	P[5]	P[6]	P[7]	P[8]	P[9]	P[10]	P[11]	P[12]	P[13]
1	1	1	1	3	3	3	3	3	3	3	10	10

Notice that for the nodes situated on the levels that are the first ones in some sections, **P[i] = T[i]**. We can preprocess **P** using a depth first search (**T[i]** is the father of node **i** in the tree, **nr** is **[sqrt(H)]** and **L[i]** is the level of the node **i**):

```
 void dfs(int node, int T[MAXN], int N, int P[MAXN], int L[MAXN], int
nr)  {
     int k;

  //if node is situated in the first
  //section then P[node] = 1
  //if node is situated at the beginning
  //of some section then P[node] = T[node]
  //if none of those two cases occurs, then
  //P[node] = P[T[node]]
     if (L[node] < nr)
         P[node] = 1;
     else
        if(!(L[node] % nr))
             P[node] = T[node];
        else
             P[node] = P[T[node]];

     for each son k of node
        dfs(k, T, N, P, L, nr);
  }
```

Now, we can easily make queries. For finding **LCA(x, y)** we we will first find in what section it lays, and then trivially compute it. Here is the code:

```
 int LCA(int T[MAXN], int P[MAXN], int L[MAXN], int x, int y)
  {
  //as long as the node in the next section of
  //x and y is not one common ancestor
  //we get the node situated on the smaller
  //lever closer
     while (P[x] != P[y])
       if (L[x] > L[y])
            x = P[x];
       else
            y = P[y];

  //now they are in the same section, so we trivially compute the LCA
```

```
      while (x != y)
        if (L[x] > L[y])
              x = T[x];
        else
              y = T[y];
        return x;
  }
```

This function makes at most **2 * sqrt(H)** operations. Using this approach we get an **<O(N), O(sqrt(H))>** algorithm, where **H** is the height of the tree. In the worst case **H = N**, so the overall complexity is **<O(N), O(sqrt(N))>**. The main advantage of this algorithm is quick coding (an average Division 1 coder shouldn't need more than 15 minutes to code it).

Another easy solution in <O(N logN, O(logN)>
If we need a faster solution for this problem we could use dynamic programming. First, let's compute a table P[1,N][1,logN] where P[i][j] is the 2j-th ancestor of i. For computing this value we may use the following recursion:

The preprocessing function should look like this:

```
 void process3(int N, int T[MAXN], int P[MAXN][LOGMAXN])
  {
      int i, j;

  //we initialize every element in P with -1
      for (i = 0; i < N; i++)
            for (j = 0; 1 << j < N; j++)
             P[i][j] = -1;

  //the first ancestor of every node i is T[i]
      for (i = 0; i < N; i++)
            P[i][0] = T[i];

  //bottom up dynamic programing
      for (j = 1; 1 << j < N; j++)
         for (i = 0; i < N; i++)
             if (P[i][j - 1] != -1)
                 P[i][j] = P[P[i][j - 1]][j - 1];
  }
```

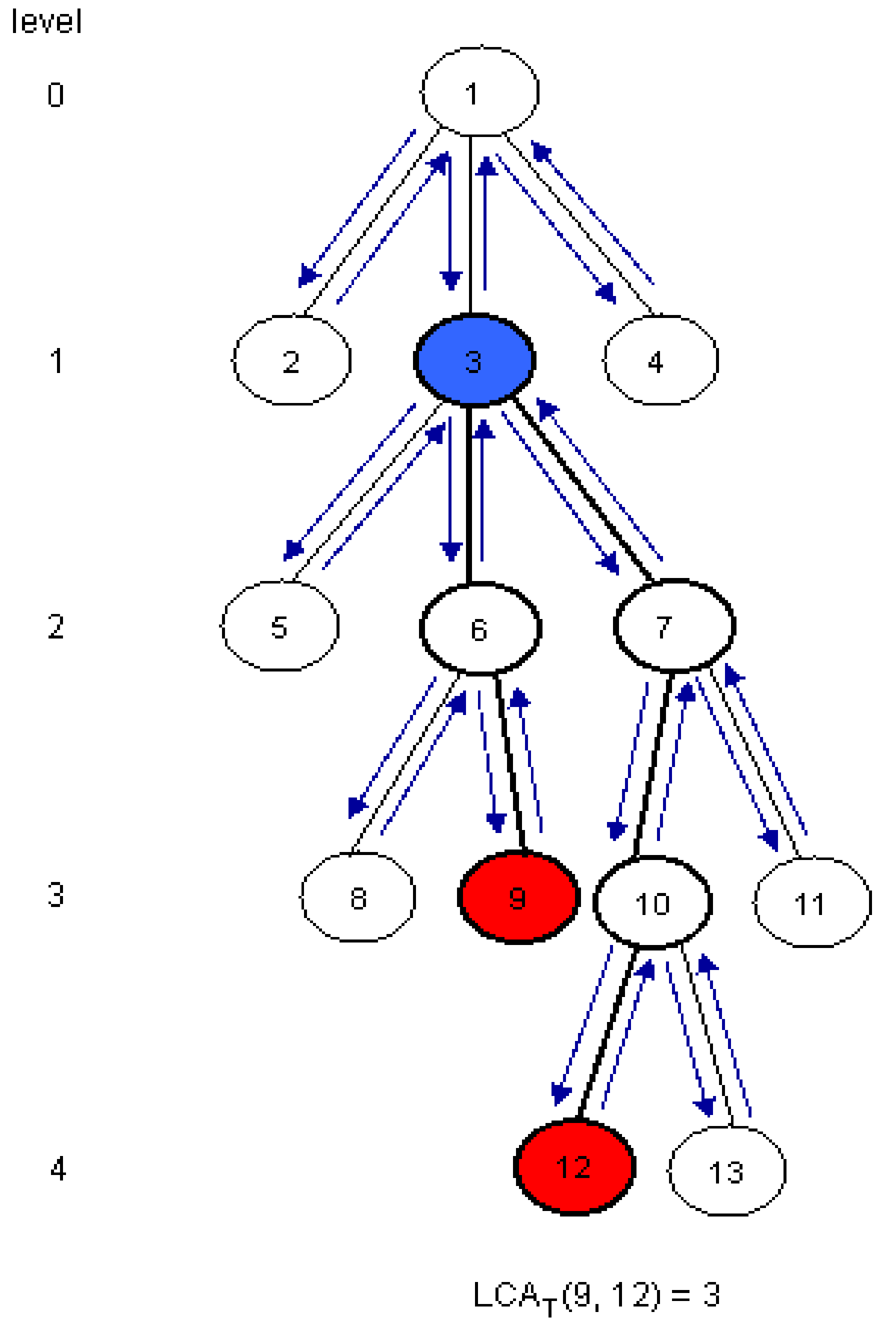
This takes **O(N logN)** time and space. Now let's see how we can make queries. Let **L[i]** be the level of node **i** in the tree. We must observe that if **p** and **q** are on the same level in the tree we can compute **LCA(p, q)** using a meta-binary search. So, for every power **j** of **2** (between **log(L[p])** and **0**, in descending order), if **P[p][j] != P[q][j]** then we know that **LCA(p, q)** is on a higher level and we will continue searching for **LCA(p = P[p][j], q = P[q][j])**. At the end, both **p** and **q** will have the same father, so return **T[p]**. Let's see what happens if **L[p] != L[q]**. Assume, without loss of generality, that **L[p] < L[q]**. We can use the same meta-binary search for finding the ancestor of **p** situated on the same level with **q**, and then we can compute the **LCA** as described below. Here is how the query function should look:

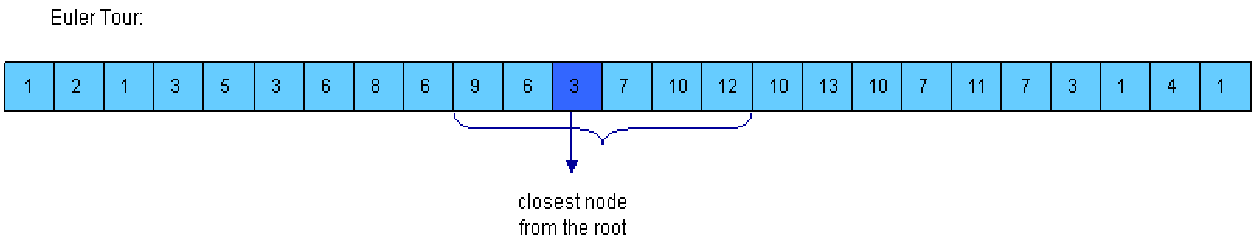
```
 int query(int N, int P[MAXN][LOGMAXN], int T[MAXN],
  int L[MAXN], int p, int q)
  {
      int tmp, log, i;

  //if p is situated on a higher level than q then we swap them
      if (L[p] < L[q])
          tmp = p, p = q, q = tmp;

  //we compute the value of [log(L[p)]
      for (log = 1; 1 << log <= L[p]; log++);
      log--;

  //we find the ancestor of node p situated on the same level
  //with q using the values in P
      for (i = log; i >= 0; i--)
         if (L[p] - (1 << i) >= L[q])
               p = P[p][i];

      if (p == q)
            return p;

  //we compute LCA(p, q) using the values in P
      for (i = log; i >= 0; i--)
         if (P[p][i] != -1 && P[p][i] != P[q][i])
                p = P[p][i], q = P[q][i];

      return T[p];
  }
```

Now, we can see that this function makes at most **2 * log(H)** operations, where **H** is the height of the tree. In the worst case **H = N**, so the overall complexity of this algorithm is **<O(N logN), O(logN)>**. This solution is easy to code too, and it's faster than the previous one.

1.9.3. LCA with RMQ

Now, let's show how we can use RMQ for computing LCA queries. Actually, we will reduce the LCA problem to RMQ in linear time, so every algorithm that solves the RMQ problem will solve the LCA problem too. Let's show how this reduction can be done using an example:



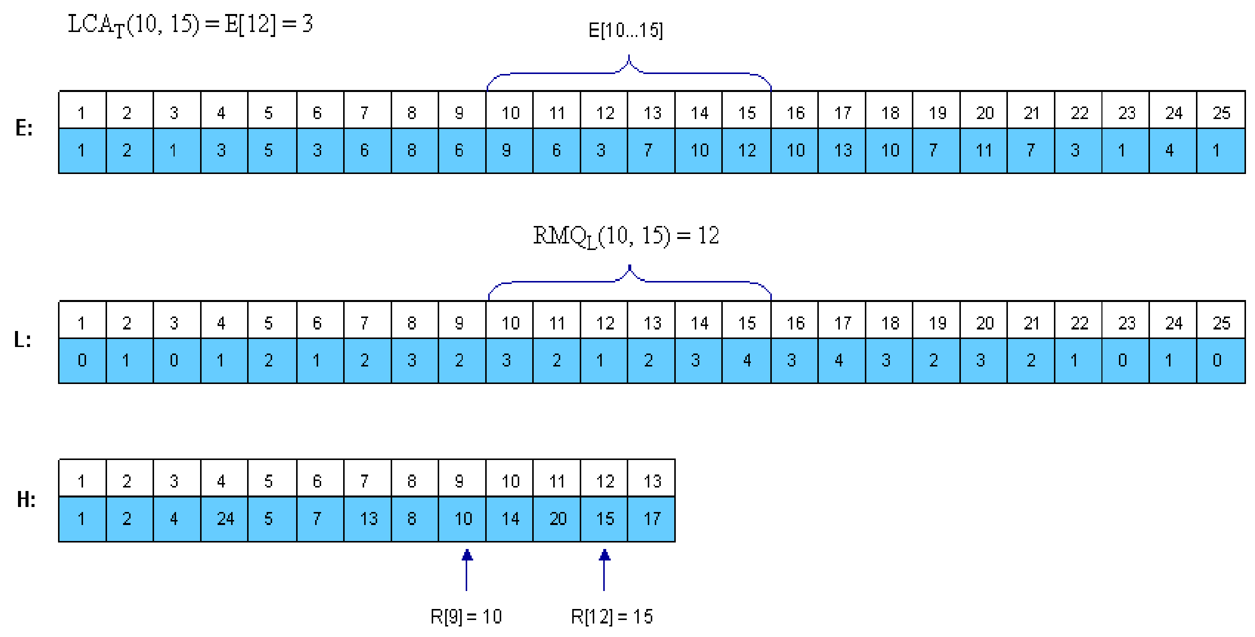


click to enlarge image

Notice that **LCAT(u, v)** is the closest node from the root encountered between the visits of **u** and **v** during a depth first search of **T**. So, we can consider all nodes between any two indices of **u** and **v** in the Euler Tour of the tree and then find the node situated on the smallest level between them. For this, we must build three arrays:

- **E[1, 2*N-1] - the nodes visited in an Euler Tour of T; E[i] is the label of i-th visited node in the tour**
- **L[1, 2*N-1] - the levels of the nodes visited in the Euler Tour; L[i] is the level of node E[i]**
- **H[1, N] - H[i] is the index of the first occurrence of node i in E (any occurrence would be good, so it's not bad if we consider the first one)**

Assume that **H[u] < H[v]** (otherwise you must swap **u** and **v**). It's easy to see that the nodes between the first occurrence of **u**and the first occurrence of **v** are **E[H[u]...H[v]]**. Now, we must find the node situated on the smallest level. For this, we can use**RMQ**. So, **LCAT(u, v) = E[RMQL(H[u], H[v])]** (remember that RMQ returns the index). Here is how **E**, **L** and **H** should look for the example:



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
E:	1	2	1	3	5	3	6	8	6	9	6	3	7	10	12	10	13	10	7	11	7	3	1	4	1

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
L:	0	1	0	1	2	1	2	3	2	3	2	1	2	3	4	3	4	3	2	3	2	1	0	1	0

	1	2	3	4	5	6	7	8	9	10	11	12	13
H:	1	2	4	24	5	7	13	8	10	14	20	15	17

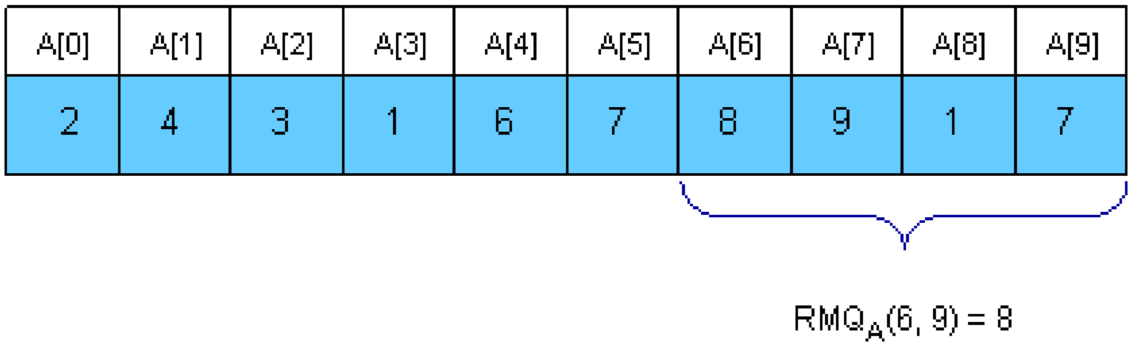
click to enlarge image

Notice that consecutive elements in L differ by exactly 1.

From RMQ to LCA
We have shown that the LCA problem can be reduced to RMQ in linear time. Here we will show how we can reduce the RMQ problem to LCA. This means that we actually can reduce the general RMQ to the restricted version of the problem (where consecutive elements in the array differ by exactly 1). For this we should use cartesian trees.

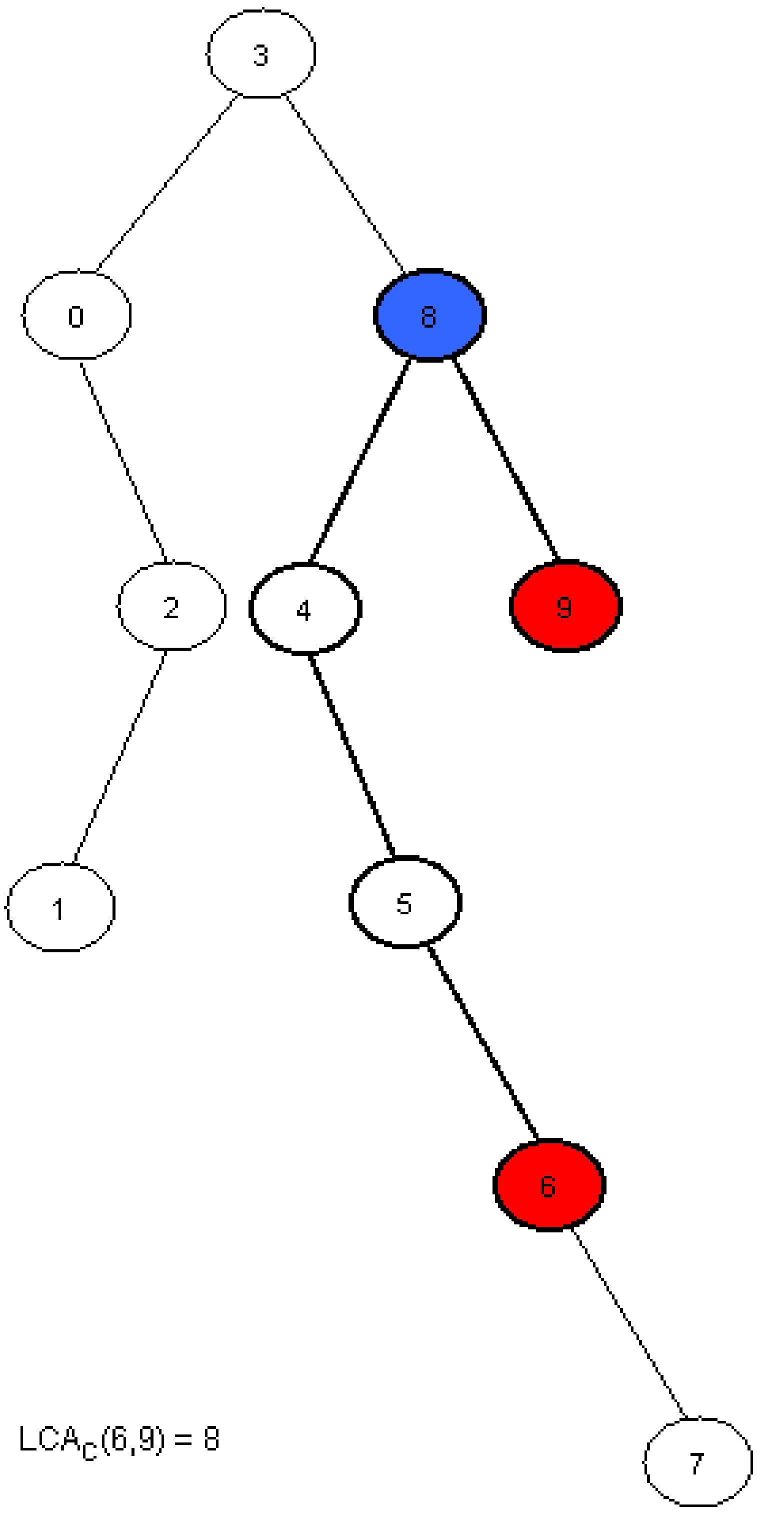
A Cartesian Tree of an array **A[0, N - 1]** is a binary tree **C(A)** whose root is a minimum element of **A**, labeled with the position **i**of this minimum. The left child of the root is the Cartesian Tree of **A[0, i - 1]** if **i > 0**, otherwise there's no child. The right child is defined similary for **A[i + 1, N - 1]**. Note that the Cartesian Tree is not necessarily unique if **A** contains equal elements. In this tutorial the first appearance of the minimum value will be used, thus the Cartesian Tree will be unique. It's easy to see now that **RMQA(i, j) = LCAC(i, j).**

Here is an example:



A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
2	4	3	1	6	7	8	9	1	7

$RMQ_A(6, 9) = 8$



Now we only have to compute **C(A)** in linear time. This can be done using a stack. At the beginning the stack is empty. We will then insert the elements of **A** in the stack. At the **i-th** step **A[i]** will be added next to the last element in the stack that has a smaller or equal value to **A[i]**, and all the greater elements will be removed. The element that was in the stack on the position of**A[i]** before the insertion was done will become the left son of **i**, and **A[i]** will become the right son of the smaller element behind him. At every step the first element in the stack is the root of the cartesian tree. It's easier to build the tree if the stack will hold the indexes of the elements, and not their value.

Here is how the stack will look at each step for the example above:

Step	Stack	Modifications made in the tree
0	0	0 is the only node in the tree.
1	0 1	1 is added at the end of the stack. Now, 1 is the right son of 0.
2	0 2	2 is added next to 0, and 1 is removed (A[2] < A[1]). Now, 2 is the right son of 0 and the left son of 2 is 1.
3	3	A[3] is the smallest element in the vector so far, so all elements in the stack will be removed and 3 will become the root of the tree. The left child of 3 is 0.
4	3 4	4 is added next to 3, and the right son of 3 is 4.
5	3 4 5	5 is added next to 4, and the right son of 4 is 5.
6	3 4 5 6	6 is added next to 5, and the right son of 5 is 6.
7	3 4 5 6 7	7 is added next to 6, and the right son of 6 is 7.

8	3 8	8 is added next to 3, and all greater elements are removed. 8 is now the right child of 3 and the left child of 8 is 4.
9	3 8 9	9 is added next to 8, and the right son of 8 is 9.

Note that every element in **A** is only added once and removed at most once, so the complexity of this algorithm is **O(N)**. Here is how the tree-processing function will look:

```
void computeTree(int A[MAXN], int N, int T[MAXN])
{
      int st[MAXN], i, k, top = -1;

 //we start with an empty stack
 //at step i we insert A[i] in the stack
      for (i = 0; i < N; i++)
      {
 //compute the position of the first element that is
 //equal or smaller than A[i]
            k = top;
            while (k >= 0 && A[st[k]] > A[i])
            k--;
 //we modify the tree as explained above
          if (k != -1)
            T[i] = st[k];
          if (k < top)
            T[st[k + 1]] = i;
 //we insert A[i] in the stack and remove
 //any bigger elements
            st[++k] = i;
            top = k;
      }
 //the first element in the stack is the root of
 //the tree, so it has no father
      T[st[0]] = -1;
 }
```

An<O(N), O(1)> algorithm for the restricted RMQ

Now we know that the general RMQ problem can be reduced to the restricted version using LCA. Here, consecutive elements in the array differ by exactly 1. We can use this and give a fast **<O(N), O(1)>** algorithm. From now we will solve the RMQ problem for an array **A[0, N - 1]** where **|A[i] - A[i + 1]| = 1, i = [1, N - 1]**. We transform **A** in a binary array with **N-1** elements, where **A[i]**

= A[i] - A[i + 1]. It's obvious that elements in **A** can be just **+1** or **-1.** Notice that the old value of **A[i]** is now the sum of **A[1], A[2] .. A[i]** plus the old **A[0].** However, we won't need the old values from now on.

To solve this restricted version of the problem we need to partition **A** into blocks of size **l = [(log N) / 2].** Let **A'[i]** be the minimum value for the **i-th** block in **A** and **B[i]** be the position of this minimum value in **A.** Both **A** and **B** are **N/l** long. Now, we preprocess **A'** using the ST algorithm described in Section1. This will take **O(N/l * log(N/l)) = O(N)** time and space. After this preprocessing we can make queries that span over several blocks in **O(1).** It remains now to show how the in-block queries can be made. Note that the length of a block is **l = [(log N) / 2],** which is quite small. Also, note that **A** is a binary array. The total number of binary arrays of size **l** is **2l=sqrt(N).** So, for each binary block of size **l** we need to lock up in a table **P** the value for RMQ between every pair of indices. This can be trivially computed in **O(sqrt(N)*l2)=O(N)** time and space. To index table **P,** preprocess the type of each block in **A** and store it in array **T[1, N/l].** The block type is a binary number obtained by replacing -1 with 0 and +1 with 1.

Now, to answer **RMQA(i, j)** we have two cases:

- **i and j are in the same block, so we use the value computed in P and T**
- **i and j are in different blocks, so we compute three values: the minimum from i to the end of i's block using P and T, the minimum of all blocks between i's and j's block using precomputed queries on A' and the minimum from the begining of j'sblock to j, again using T and P; finally return the position where the overall minimum is using the three values you just computed.**

1.10. Topological sort

In graph theory, a topological sort or topological ordering of a directed acyclic graph (DAG) is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more topological sorts.
More formally, define the reachability relation *R* over the nodes of the DAG such that *xRy* if and only if there is a directed path from *x* to *y*. Then, *R* is a partial order, and a topological sort is a linear extension of this partial order, that is, a total order compatible with the partial order.
The usual algorithms for topological sorting have running time linear in the number of nodes plus the number of edges (O(|V|+|E|)).
One of these algorithms, first described by Kahn (1962), works by choosing vertices in the same order as the eventual topological sort. First, find a list of "start nodes" which have no incoming edges and insert them into a set S; at least one such node must exist if graph is acyclic. Then:

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
    remove a node n from S
    insert n into L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    output error message (graph has at least one cycle)
else
    output message (proposed topologically sorted order: L)
```

If the graph was a DAG, a solution is contained in the list L (the solution is not unique). Otherwise, the graph has at least one cycle and therefore a topological sorting is impossible.
Note that, reflecting the non-uniqueness of the resulting sort, the structure S can be simply a set or a queue or a stack. Depending on the order that nodes n are removed from set S, a different solution is created.
An alternative algorithm for topological sorting is based on depth-first search. For this algorithm, edges point in the opposite direction as the previous algorithm (and the opposite direction to that shown in the diagram in the Examples section above). There is an edge from *x* to *y* if job *x* depends on job *y* (in other words, if job *y* must be completed before job *x* can be started). The algorithm loops through each node of the graph, in an arbitrary order, initiating a depth-first search that terminates when it hits any node that has already been visited since the beginning of the topological sort:

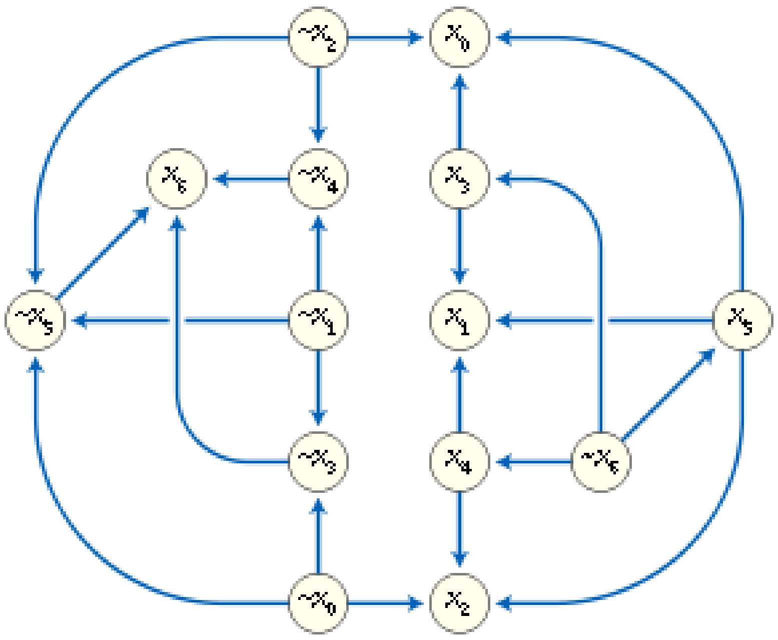
```
L ← Empty list that will contain the sorted nodes
S ← Set of all nodes with no incoming edges

function visit(node n)
    if n has not been visited yet then
        mark n as visited
        for each node m with an edge from n to m do
            visit(m)
        add n to L

for each node n in S do
    visit(n)
```

Note that each node *n* gets added to the output list L only after considering all other nodes on which *n* depends (all descendant nodes of *n* in the graph). Specifically, when the algorithm adds node *n*, we are guaranteed that all nodes on which *n* depends are already in the output list L: they were added to L either by the preceding recursive call to visit(), or by an earlier call to visit(). Since each edge and node is visited once, the algorithm runs in

linear time. Note that the simple pseudocode above cannot detect the error case where the input graph contains cycles. The algorithm can be refined to detect cycles by watching for nodes which are visited more than once during any nested sequence of recursive calls to visit() (e.g., by passing a list down as an extra argument to visit(), indicating which nodes have already been visited in the current call stack). This depth-first-search-based algorithm is the one described by Cormen, Leiserson & Rivest (1990); it seems to have been first described in print by Tarjan (1976).
[edit]Uniqueness
If a topological sort has the property that all pairs of consecutive vertices in the sorted order are connected by edges, then these edges form a directed Hamiltonian path in the DAG. If a Hamiltonian path exists, the topological sort order is unique; no other order respects the edges of the path. Conversely, if a topological sort does not form a Hamiltonian path, the DAG will have two or more valid topological orderings, for in this case it is always possible to form a second valid ordering by swapping two consecutive vertices that are not connected by an edge to each other. Therefore, it is possible to test in polynomial time whether a unique ordering exists, and whether a Hamiltonian path exists, despite the NP-hardness of the Hamiltonian path problem for more general directed graphs (Vernet & Markenzon 1997).

1.11. Two-sat

In computer science, **2-satisfiability** (abbreviated as 2-SAT or just 2SAT) is the problem of determining whether a collection of two-valued (Boolean or binary) variables with constraints on pairs of variables can be assigned values satisfying all the constraints. It is a special case of the general Boolean satisfiability problem, which can involve constraints on more than two variables, and of constraint satisfaction problems, which can allow more than two choices for the value of each variable. But in contrast to those problems, which are NP-complete, it has a known polynomial time solution. Instances of the 2-satisfiability problem are typically expressed as 2-CNF or **Krom formulas**.



The implication graph for the example 2-SAT instance shown in this section.

A 2-SAT problem may be described using a Boolean expression with a special restricted form: a conjunction of disjunctions (and of ors), where each disjunction (or operation) has two arguments that may either be variables or the negations of variables. The variables or their negations appearing in this formula are known as terms and the disjunctions of pairs of terms are known as clauses. For example, the following formula is in conjunctive normal form, with seven variables and eleven clauses:

$$(x_0 \lor x_2) \land (x_0 \lor \lnot x_3) \land (x_1 \lor \lnot x_3) \land (x_1 \lor \lnot x_4) \land (x_2 \lor \lnot x_4) \land$$

$$(x_0 \lor \lnot x_5) \land (x_1 \lor \lnot x_5) \land (x_2 \lor \lnot x_5) \land (x_3 \lor x_6) \land (x_4 \lor x_6) \land (x_5 \lor x_6).$$

The 2-satisfiability problem is to find a truth assignment to these variables that makes a formula of this type true: we must choose whether to make each of the variables true or false, so that every clause has at least one term that becomes true. For the expression shown above, one possible satisfying assignment is the one that sets all seven of the variables to true. There are also 15 other ways of setting all the variables so that the formula becomes true. Therefore, the 2-SAT instance represented by this expression is satisfiable.

Formulas with the form described above are known as 2-CNF formulas; the "2" in this name stands for the number of terms per clause, and "CNF" stands for conjunctive normal form, a type of Boolean expression in the form of a conjunction of disjunctions. They are also called Krom formulas, after the work of UC Davis mathematician Melven R. Krom, whose 1967 paper was one of the earliest works on the 2-satisfiability problem.[1]

Each clause in a 2-CNF formula is logically equivalent to an implication from one variable or negated variable to the other. For example,

$$(x_0 \lor \lnot x_3) \equiv (\lnot x_0 \Rightarrow \lnot x_3) \equiv (x_3 \Rightarrow x_0).$$

Because of this equivalence between these different types of operation, a 2-satisfiability instance may also be written in implicative normal form, in which we replace each or operation in the conjunctive normal form by both of the two implications to which it is equivalent.
A third, more graphical way of describing a 2-satisfiability instance is as an implication graph. An implication graph is a directed graph in which there is one vertex per variable or negated variable, and an edge connecting one vertex to another whenever the corresponding variables are related by an implication in the implicative normal form of the instance. An implication graph must be a skew-symmetric graph, meaning that the undirected graph formed by forgetting the orientations of its edges has a symmetry that takes each variable to its negation and reverses the orientations of all of the edges.[2]

Algorithms
Several algorithms are known for solving the 2-satisfiability problem; the most efficient of them take linear time.[1][2][3]

Strongly connected components
Aspvall, Plass & Tarjan (1979) found a simpler linear time procedure for solving 2-satisfiability instances, based on the notion of strongly connected components from graph theory.[2]
Two vertices in a directed graph are said to be strongly connected to each other if there is a directed path from one to the other and vice versa. This is an equivalence relation, and the vertices of the graph may be partitioned into strongly connected components, subsets within which every two vertices are strongly connected. There are several efficient linear time algorithms for finding the strongly connected components of a graph, based on depth first search: Tarjan's strongly connected components algorithm[5] and Gabow's algorithm[6] each perform a single depth first search. Kosaraju's algorithm performs two depth first searches, but is very simple: the first search is used only to order the vertices, in the reverse of a postorder depth-first traversal. Then, the second pass of the algorithm loops through the vertices in this order, and for each vertex that has not already been assigned to a component, it performs a depth-first search of the transpose graph starting from that vertex, backtracking when the search reaches a previously-assigned vertices; the unsassigned vertices reached by this search form a new component.
In terms of the implication graph, two terms belong to the same strongly connected component whenever there exist chains of implications from one term to the other and vice versa. Therefore, the two terms must have the same value in any satisfying assignment to the given 2-satisfiability instance. In particular, if a variable and its negation both belong to the same strongly connected component, the instance cannot be satisfied, because it is impossible to assign both of these terms the same value. As Aspvall et al. showed, this is a necessary and sufficient condition: a 2-CNF formula is satisfiable if and only if there is no variable that belongs to the same strongly connected component as its negation.[2]

This immediately leads to a linear time algorithm for testing satisfiability of 2-CNF formulae: simply perform a strong connectivity analysis on the implication graph and check that each variable and its negation belong to different components. However, as Aspvall et al. also showed, it also leads to a linear time algorithm for finding a satisfying assignment, when one exists. Their algorithm performs the following steps:

- Construct the implication graph of the instance, and find its strongly connected components using any of the known linear-time algorithms for strong connectivity analysis.
- Check whether any strongly connected component contains both a variable and its negation. If so, report that the instance is not satisfiable and halt.
- Construct the condensation of the implication graph, a smaller graph that has one vertex for each strongly connected component, and an edge from component *i* to component *j* whenever the implication graph contains an edge *uv* such that *u* belongs to component *i* and *v* belongs to component *j*. The condensation is automatically a directed acyclic graph and, like the implication graph from which it was formed, it is skew-symmetric.
- Topologically order the vertices of the condensation; the order in which the components are generated by Kosaraju's algorithm is automatically a topological ordering.
- For each component in this order, if its variables do not already have truth assignments, set all the terms in the component to be false. This also causes all of the terms in the complementary component to be set to true.

Due to the topological ordering, when a term *x* is set to false, all terms that lead to it via a chain of implications will themselves already have been set to false. Symmetrically, when a term is set to true, all terms that can be reached from it via a chain of implications will already have been set to true. Therefore, the truth assignment constructed by this procedure satisfies the given formula, which also completes the proof of correctness of the necessary and sufficient condition identified by Aspvall et al.[2]
As Aspvall et al. show, a similar procedure involving topologically ordering the strongly connected components of the implication graph may also be used to evaluate fully quantified Boolean formulae in which the formula being quantified is a 2-CNF formula.[2]

1.12. Assigment problem

Introduction
Are you familiar with the following situation? You open the Div I Medium and don't know how to approach it, while a lot of people in your room submitted it in less than 10 minutes. Then, after the contest, you find out in the

editorial that this problem can be simply reduced to a classical one. If yes, then this tutorial will surely be useful for you.

Problem statement

In this article we'll deal with one optimization problem, which can be informally defined as:

Assume that we have ***N*** *workers and* ***N*** *jobs that should be done. For each pair (worker, job) we know salary that should be paid to worker for him to perform the job. Our goal is to complete all jobs minimizing total inputs, while assigning each worker to exactly one job and vice versa.*

Converting this problem to a formal mathematical definition we can form the following equations:

$\{c_{ij}\}_{N \times N}$ - cost matrix, where ***cij*** - cost of worker ***i*** to perform job ***j***.

$\{x_{ij}\}_{N \times N}$ - resulting binary matrix, where ***xij*** = 1 if and only if ***ith*** worker is assigned to ***jth*** job.

$$\sum_{j=1}^{N} x_{ij} = 1, \qquad \forall i \in \overline{1,N}$$

- one worker to one job assignment.

$$\sum_{i=1}^{N} x_{ij} = 1, \qquad \forall j \in \overline{1,N}$$

- one job to one worker assignment.

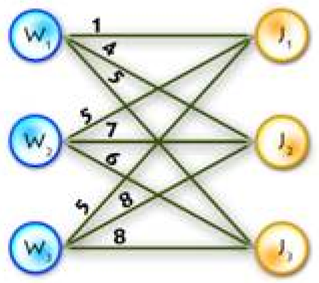
$$\sum_{i=1}^{N} \sum_{j=1}^{N} c_{ij} x_{ij} \rightarrow min$$

- total cost function.

We can also rephrase this problem in terms of graph theory. Let's look at the job and workers as if they were a bipartite graph, where each edge between the***ith*** worker and ***jth*** job has weight of ***cij***. Then our task is to find minimum-weight matching in the graph (the matching will consists of **N** edges, because our bipartite graph is complete).

Small example just to make things clearer:

$$\begin{pmatrix} 1 & 4 & 5 \\ 5 & 7 & 6 \\ 5 & 8 & 8 \end{pmatrix} \Leftrightarrow$$



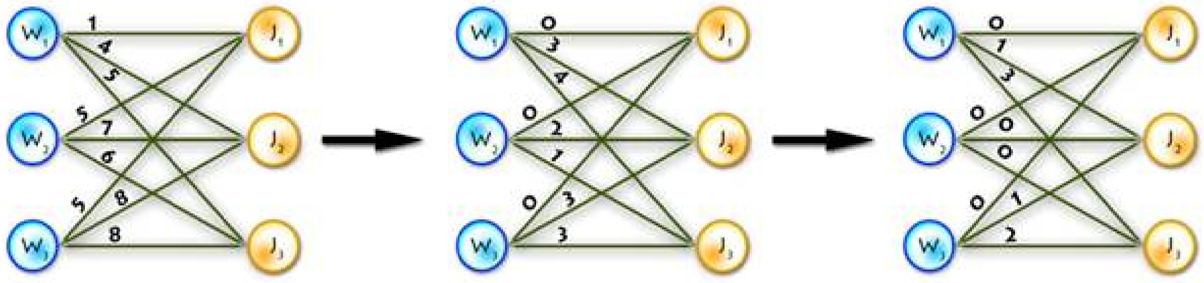
General description of the algorithm

This problem is known as the assignment problem. The assignment problem is a special case of the transportation problem, which in turn is a special case of the [min-cost flow]() problem, so it can be solved using algorithms that solve the more general cases. Also, our problem is a special case of binary integer linear programming problem (which is NP-hard). But, due to the specifics of

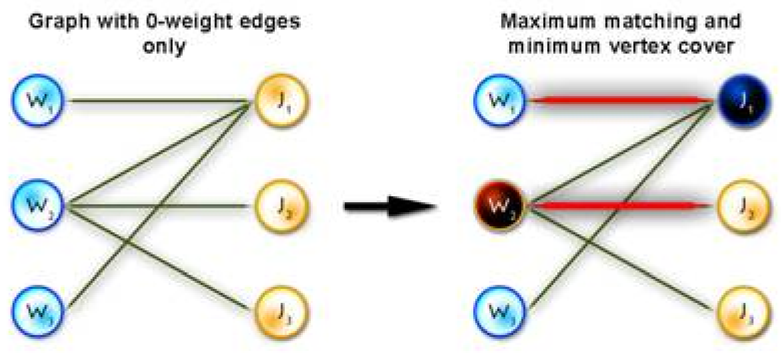
the problem, there are more efficient algorithms to solve it. We'll handle the assignment problem with the Hungarian algorithm (or Kuhn-Munkres algorithm). I'll illustrate two different implementations of this algorithm, both graph theoretic, one easy and fast to implement with **O(n4)** complexity, and the other one with **O(n3)** complexity, but harder to implement.
There are also implementations of Hungarian algorithm that do not use graph theory. Rather, they just operate with cost matrix, making different transformation of it (see [1] for clear explanation). We'll not touch these approaches, because it's less practical for TopCoder needs.

O(n4) algorithm explanation

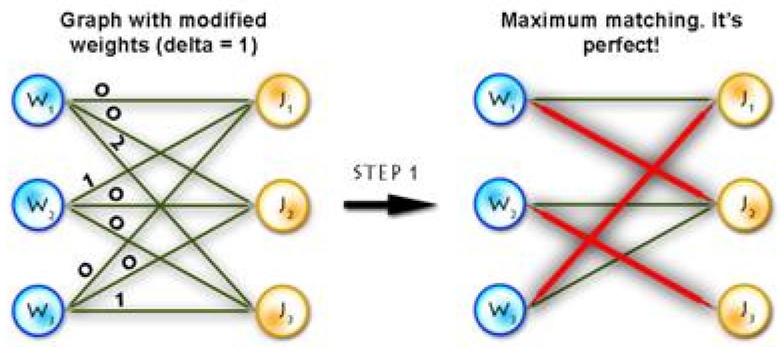
As mentioned above, we are dealing with a bipartite graph. The main idea of the method is the following: consider we've found the perfect matching using only edges of weight 0 (hereinafter called "0-weight edges"). Obviously, these edges will be the solution of the assignment problem. If we can't find perfect matching on the current step, then the Hungarian algorithm changes weights of the available edges in such a way that the new 0-weight edges appear and these changes do not influence the optimal solution.
To clarify, let's look at the step-by-step overview:
Step 0)
A. For each vertex from left part (workers) find the minimal outgoing edge and subtract its weight from all weights connected with this vertex. This will introduce 0-weight edges (at least one).
B. Apply the same procedure for the vertices in the right part (jobs).



Actually, this step is not necessary, but it decreases the number of main cycle iterations.
Step 1)
A. Find the maximum matching using only 0-weight edges (for this purpose you can use max-flow algorithm, augmenting path algorithm, etc.).
B. If it is perfect, then the problem is solved. Otherwise find the minimum vertex cover ***V*** (for the subgraph with 0-weight edges only), the best way to do this is to use Köning's graph theorem.


Graph with 0-weight edges only
Maximum matching and minimum vertex cover
W_1
W_2
W_3
J_1
J_2
J_3

Step 2) Let   and adjust the weights using the following rule:


Graph with modified weights (delta = 1)
Maximum matching. It's perfect!
STEP 1
W_1
W_2
W_3
J_1
J_2
J_3
0
0
2
1
0
0
0
0
1

Step 3) Repeat Step 1 until solved.
But there is a nuance here; finding the maximum matching in step 1 on each iteration will cause the algorithm to become **O(n5)**. In order to avoid this, on each step we can just modify the matching from the previous step, which only takes **O(n2)** operations.
It's easy to see that no more than n2 iterations will occur, because every time at least one edge becomes 0-weight. Therefore, the overall complexity is **O(n4)**.

O(n3) algorithm explanation

Warning! In this section we will deal with the maximum-weighted matching problem. It's obviously easy to transform minimum problem to the maximum one, just by setting:

or

.

Before discussing the algorithm, let's take a look at some of the theoretical ideas. Let's start off by considering we have a complete bipartite graph

G=(V,E) where and , ***w(x,y)*** - weight of edge ***(x,y)***.

Vertex and set neighborhood

Let . Then is **v's** neighborhood, or all vertices that share an edge with **v**.

Let . Then is ***S*'s** neighborhood, or all vertices that share an edge with a vertex in **S**.

Vertex labeling

This is simply a function (for each vertex we assign some number called a label). Let's call this labeling feasible if it satisfies the following condition: . In other words, the sum of the labels of the vertices on both sides of a given edge are greater than or equal to the weight of that edge.

Equality subgraph

Let ***Gl=(V,El)*** be a spanning subgraph of **G** (in other words, it includes all vertices from **G**). If **G** only those edges **(x,y)** which satisfy the following condition: , then it is an equality subgraph. In other words, it only includes those edges from the bipartite matching which allow the vertices to be perfectly feasible.

Now we're ready for the theorem which provides the connection between equality subgraphs and maximum-weighted matching:

*If **M*** is a perfect matching in the equality subgraph **Gl**, then **M*** is a maximum-weighted matching in **G**.*

The proof is rather straightforward, but if you want you can do it for practice. Let's continue with a few final definitions:

Alternating path and alternating tree

Consider we have a matching ***M* ()**.

Vertex is called matched if , otherwise it is called *exposed (free, unmatched)*.

(In the diagram below, **W1**, **W2**, **W3**, **J1**, **J3**, **J4** are matched, **W4**, **J2** are exposed)

Path ***P*** is called alternating if its edges alternate between **M** and **E\M**. (For example, (**W4**, **J4**, **W3**, **J3**, **W2**, **J2**) and (**W4**, **J1**, **W1**) are alternating paths)

If the first and last vertices in alternating path are exposed, it is called *augmenting* (because we can increment the size of the matching by inverting edges along this path, therefore matching unmatched edges and vice versa).

((**W4**, **J4**, **W3**, **J3**, **W2**, **J2**) - augmenting alternating path)

A tree which has a root in some exposed vertex, and a property that every path starting in the root is alternating, is called an *alternating tree*. (Example on the picture above, with root in **W4**)

That's all for the theory, now let's look at the algorithm:

First let's have a look on the scheme of the Hungarian algorithm:
Step 0. Find some initial feasible vertex labeling and some initial matching.
Step 1. If **M** is perfect, then it's optimal, so problem is solved. Otherwise, some exposed exists; set . (**x** - is a root of the alternating tree we're going to build). Go to step 2.
Step 2. If go to step 3, else . Find

	(1)

and replace existing labeling with the next one:

	(2)

Now replace with
Step 3. Find some vertex . If ***y*** is exposed then an alternating path from ***x*** (root of the tree) to ***y*** exists, augment matching along this path and go to step 1. If ***y*** is matched in **M** with some vertex ***z*** add ***(z,y)*** to the alternating tree and set , go to step 2.
And now let's illustrate these steps by considering an example and writing some code.
As an example we'll use the previous one, but first let's transform it to the maximum-weighted matching problem, using the second method from the two described above. (See Picture 1)

Picture 1

Here are the global variables that will be used in the code:

```
#define N 55             //max number of vertices in one part
#define INF 100000000    //just infinity

int cost[N][N];          //cost matrix
int n, max_match;        //n workers and n jobs
int lx[N], ly[N];        //labels of X and Y parts
int xy[N];               //xy[x] - vertex that is matched with x,
int yx[N];               //yx[y] - vertex that is matched with y
bool S[N], T[N];         //sets S and T in algorithm
int slack[N];            //as in the algorithm description
int slackx[N];           //slackx[y] such a vertex, that
                         // l(slackx[y]) + l(y) - w(slackx[y],y) = slack[y]
int prev[N];             //array for memorizing alternating paths
```

Step 0:
It's easy to see that next initial labeling will be feasible:

And as an initial matching we'll use an empty one. So we'll get equality subgraph as on Picture 2. The code for initializing is quite easy, but I'll paste it for completeness:

```
void init_labels()
{
    memset(lx, 0, sizeof(lx));
    memset(ly, 0, sizeof(ly));
    for (int x = 0; x < n; x++)
        for (int y = 0; y < n; y++)
            lx[x] = max(lx[x], cost[x][y]);
}
```

The next three steps will be implemented in one function, which will correspond to a single iteration of the algorithm. When the algorithm halts, we will have a perfect matching, that's why we'll have n iterations of the algorithm and therefore **(n+1)** calls of the function.

Step 1

According to this step we need to check whether the matching is already perfect, if the answer is positive we just stop algorithm, otherwise we need to clear ***S***,***T*** and alternating tree and then find some exposed vertex from the ***X*** part. Also, in this step we are initializing a *slack* array, I'll describe it on the next step.

```
void augment()                          //main function of the algorithm
{
    if (max_match == n) return;         //check wether matching is already
perfect
    int x, y, root;                     //just counters and root vertex
    int q[N], wr = 0, rd = 0;           //q - queue for bfs, wr,rd - write and
read
                                        //pos in queue
    memset(S, false, sizeof(S));        //init set S
    memset(T, false, sizeof(T));        //init set T
    memset(prev, -1, sizeof(prev));     //init set prev - for the alternating
tree
    for (x = 0; x < n; x++)             //finding root of the tree
        if (xy[x] == -1)
        {
            q[wr++] = root = x;
            prev[x] = -2;
            S[x] = true;
            break;
        }

    for (y = 0; y < n; y++)             //initializing slack array
    {
        slack[y] = lx[root] + ly[y] - cost[root][y];
        slackx[y] = root;
    }
```

Step 2

On this step, the alternating tree is completely built for the current labeling, but the augmenting path hasn't been found yet, so we need to improve the labeling. It will add new edges to the equality subgraph, giving an opportunity to expand the alternating tree. This is the main idea of the method; *we are improving the labeling until we find an augmenting path in the equality graph corresponding to the current labeling*. Let's turn back to step 2. There we just change labels using formulas **(1)** and **(2)**, but using them in an obvious manner will cause the algorithm to have **O(n4)** time. So, in order to avoid this we use a*slack* array initialized in **O(n)** time because we only augment the array created in step 1:

Then we just need O(n) to calculate a delta Δ (see (1)):

Updating slack:
1) On **step 3**, when vertex ***x*** moves from ***X\S*** to ***S***, this takes **O(n)**.
2) On **step 2**, when updating labeling, it's also takes **O(n)**, because:

So we get **O(n)** instead of **O(n2)** as in the straightforward approach.
Here's code for the label updating function:
```
void update_labels()
{
    int x, y, delta = INF;             //init delta as infinity
    for (y = 0; y < n; y++)            //calculate delta using slack
        if (!T[y])
            delta = min(delta, slack[y]);
    for (x = 0; x < n; x++)            //update X labels
        if (S[x]) lx[x] -= delta;
    for (y = 0; y < n; y++)            //update Y labels
        if (T[y]) ly[y] += delta;
    for (y = 0; y < n; y++)            //update slack array
        if (!T[y])
            slack[y] -= delta;
}
```

Step 3
In step 3, first we build an alternating tree starting from some exposed vertex, chosen at the beginning of each iteration. We will do this using breadth-first search algorithm. If on some step we meet an exposed vertex from the ***Y*** part, then finally we can augment our path, finishing up with a call to the main function of the algorithm. So the code will be the following:
1) Here's the function that adds new edges to the alternating tree:
```
void add_to_tree(int x, int prevx)
//x - current vertex,prevx - vertex from X before x in the alternating path,
//so we add edges (prevx, xy[x]), (xy[x], x)
{
    S[x] = true;                    //add x to S
    prev[x] = prevx;                //we need this when augmenting
    for (int y = 0; y < n; y++)    //update slacks, because we add new vertex
to S
```

```
        if (lx[x] + ly[y] - cost[x][y] < slack[y])
        {
            slack[y] = lx[x] + ly[y] - cost[x][y];
            slackx[y] = x;
        }
}
```

3) And now, the end of the ***augment()*** function:

```
//second part of augment() function
    while (true)                                                       //main
cycle
    {
        while (rd <
wr)                                                   //building tree with bfs
cycle
        {
            x = q[rd+
+];                                                  //current vertex from X
part
            for (y = 0; y < n; y+
+)                                      //iterate through all edges in equality
graph
                if (cost[x][y] == lx[x] + ly[y] &&  !T[y])
                {
                    if (yx[y] == -1) break;                            //an
exposed vertex in Y found, so
                                                                       //aug
menting path exists!
                    T[y] = true;                                       //else
just add y to T,
                    q[wr++] = yx[y];                                   //add
vertex yx[y], which is matched
                                                                       //with
y, to the queue
                    add_to_tree(yx[y], x);                             //add
edges (x,y) and (y,yx[y]) to the tree
                }
            if (y < n)
break;                                                //augmenting path found!
        }
        if (y < n)
break;                                                    //augmenting path found!

        update_labels();                                               /
/augmenting path not found, so improve labeling
        wr = rd = 0;
        for (y = 0; y < n; y++)
        //in this cycle we add edges that were added to the equality graph as
a
        //result of improving the labeling, we add edge (slackx[y], y) to the
tree if
        //and only if !T[y] &&  slack[y] == 0, also with this edge we add
another one
        //(y, yx[y]) or augment the matching, if y was exposed
            if (!T[y] &&  slack[y] == 0)
            {
                if (yx[y] ==
-1)                                       //exposed vertex in Y found -
augmenting path exists!
                {
                    x = slackx[y];
                    break;
                }
                else
                {
                    T[y] = true;                                        //else
just add y to T,
                    if (!S[yx[y]])
                    {
                        q[wr++] = yx[y];                                //add
vertex yx[y], which is matched with
                                                                        //y,
to the queue
                        add_to_tree(yx[y], slackx[y]);                  //and
add edges (x,y) and (y,
                                                                        //yx[y
]) to the tree
                    }
                }
            }
        if (y < n)
break;                                                 //augmenting path found!
    }

    if (y < n)                                                          //we
found augmenting path!
    {
        max_match+
+;                                                      //increment matching
        //in this cycle we inverse edges along augmenting path
        for (int cx = x, cy = y, ty; cx != -2; cx = prev[cx], cy = ty)
        {
            ty = xy[cx];
            yx[cy] = cx;
            xy[cx] = cy;
        }
        augment();                                                  /
/recall function, go to step 1 of the algorithm
    }
}//end of augment() function
```

The only thing in code that hasn't been explained yet is the procedure that goes after labels are updated. Say we've updated labels and now we need to complete our alternating tree; to do this and to keep algorithm in **O(n3)** time (it's only possible if we use each edge no more than one time per iteration) we need to know what edges should be added without iterating through all of them, and the answer for this question is to use BFS to add edges only from those vertices in **Y**, that are not in **T** and for which **slack[y] = 0** (it's easy to prove that in such way we'll add all edges and keep algorithm to be **O(n3)**). See picture below for explanation:

At last, here's the function that implements Hungarian algorithm:

```
int hungarian()
{
    int ret = 0;                      //weight of the optimal matching
    max_match = 0;                    //number of vertices in current matching
    memset(xy, -1, sizeof(xy));
    memset(yx, -1, sizeof(yx));
    init_labels();                    //step 0
    augment();                        //steps 1-3
    for (int x = 0; x < n; x++)       //forming answer there
        ret += cost[x][xy[x]];
    return ret;
}
```

To see all this in practice let's complete the example started on step 0.

	Build alternating tree		Augmenting path found		Build alternating tree
	→		→		→
	Update labels (Δ=1)		Build alternating tree		Update labels (Δ=1)
	→		→		→
	Build alternating tree		Augmenting path found		Build alternating tree
	→		→		→
	Update labels (Δ=2)		Build alternating tree		Update labels (Δ=1)

	→		→		→
	Build alternating tree		Augmenting path found		Optimal matching found
	→		→		

Finally, let's talk about the complexity of this algorithm. On each iteration we increment matching so we have **n** iterations. On each iterations each edge of the graph is used no more than one time when finding augmenting path, so we've got **O(n2)** operations. Concerning labeling we update *slack* array each time when we insert vertex from ***X*** into ***S***, so this happens no more than n times per iteration, updating *slack* takes **O(n)** operations, so again we've got **O(n2).** Updating labels happens no more than n time per iterations (because we add at least one vertex from ***Y*** to ***T*** per iteration), it takes **O(n)** operations - again**O(n2)**. So total complexity of this implementation is **O(n3)**.

2. Math and number theory

2.1. Combinations and permutations

```
Implementación:

/**
 * Receives a vector of n integers and does all possible permutations
 * of r elements. Once a new permutation if found, it calls the received
 * function func with the computed vector and its respective size (r)
 */
void permutations(int *vector, int n, int r, void (*func) (int*,int));

/**
 * Receives a vector of n integers and does all possible combinations
 * of r elements. Once a new combination if found, it calls the received
 * function func with the computed vector and its respective size (r)
 */
void combinations(int *vector, int n, int r, void (*func) (int*,int));


char *alg_mark;
int *alg_lista;

//r most be less or equal than n, index=0
void perm(int* vector, int n, int r, int index, void (*func) (int*,int)) {
    if (r==0) {
        func(alg_lista,index);
        return;
    }
    for (int i=0; i<n; i++) {
        if (! alg_mark[i]) {
            alg_mark[i]=1;
            alg_lista[index]=vector[i];
            perm(vector, n, r-1, index+1, func);
            alg_mark[i]=0;
        }
    }
}

//r most be less or equal than n, index=0
void comb(int* vector, int n, int r, int index, void (*func) (int*,int)) {
    if (r==0) {
        func(alg_lista,index);
        return;
    }
    int *nvect = vector;
    for (int i=0; i<=(n-r); i++) {
        alg_lista[index]=vector[i];
        comb(++nvect, n-i-1, r-1, index+1, func);
    }
}

void permutations(int* vector, int n, int r, void (*func) (int*,int)) {
    if ((r<0) || (r>n)) return;
    alg_mark = new char[n];
    alg_lista = new int[r];
    memset(alg_mark,0,n);
    perm(vector,n,r,0,func);
    delete [] alg_mark;
    delete [] alg_lista;
}

void combinations(int* vector, int n, int r, void (*func) (int*,int)) {
    if ((r<0) || (r>n)) return;
    alg_lista = new int[r];
    comb(vector,n,r,0,func);
    delete [] alg_lista;
}

public class CombinationGenerator {

  private int[] a;
  private int n;
  private int r;
  private BigInteger numLeft;
  private BigInteger total;

  //------------
  // Constructor
  //------------

  public CombinationGenerator (int n, int r) {
    if (r > n) {
      throw new IllegalArgumentException ();
    }
    if (n < 1) {
      throw new IllegalArgumentException ();
    }
    this.n = n;
    this.r = r;
    a = new int[r];
    BigInteger nFact = getFactorial (n);
    BigInteger rFact = getFactorial (r);
    BigInteger nminusrFact = getFactorial (n - r);
    total = nFact.divide (rFact.multiply (nminusrFact));
    reset ();
  }

  //------
  // Reset
  //------

  public void reset () {
    for (int i = 0; i < a.length; i++) {
      a[i] = i;
    }
    numLeft = new BigInteger (total.toString ());
  }

  //--------------------------------------------------
  // Return number of combinations not yet generated
  //--------------------------------------------------

  public BigInteger getNumLeft () {
    return numLeft;
  }

  //-----------------------------
  // Are there more combinations?
  //-----------------------------

  public boolean hasMore () {
    return numLeft.compareTo (BigInteger.ZERO) == 1;
  }

  //------------------------------------
  // Return total number of combinations
  //------------------------------------

  public BigInteger getTotal () {
    return total;
  }

  //------------------
  // Compute factorial
  //------------------

  private static BigInteger getFactorial (int n) {
    BigInteger fact = BigInteger.ONE;
    for (int i = n; i > 1; i--) {
      fact = fact.multiply (new BigInteger (Integer.toString (i)));
    }
    return fact;
  }

  //------------------------------------------------------------
  // Generate next combination (algorithm from Rosen p. 286)
  //------------------------------------------------------------

  public int[] getNext () {

    if (numLeft.equals (total)) {
      numLeft = numLeft.subtract (BigInteger.ONE);
      return a;
    }

    int i = r - 1;
    while (a[i] == n - r + i) {
      i--;
    }
    a[i] = a[i] + 1;
    for (int j = i + 1; j < r; j++) {
      a[j] = a[i] + j - i;
    }

    numLeft = numLeft.subtract (BigInteger.ONE);
    return a;

  }
}

public class PermutationGenerator {

  private int[] a;
  private BigInteger numLeft;
  private BigInteger total;

  //---------------------------------------------------------------
  // Constructor. WARNING: Don't make n too large.
  // Recall that the number of permutations is n!
  // which can be very large, even when n is as small as 20 --
  // 20! = 2,432,902,008,176,640,000 and
  // 21! is too big to fit into a Java long, which is
  // why we use BigInteger instead.
  //--------------------------------------------------------------

  public PermutationGenerator (int n) {
    if (n < 1) {
```

```
    throw new IllegalArgumentException ("Min 1");
  }
  a = new int[n];
  total = getFactorial (n);
  reset ();
}

//------
// Reset
//------

public void reset () {
  for (int i = 0; i < a.length; i++) {
    a[i] = i;
  }
  numLeft = new BigInteger (total.toString ());
}

//------------------------------------------------
// Return number of permutations not yet generated
//------------------------------------------------

public BigInteger getNumLeft () {
  return numLeft;
}

//------------------------------------
// Return total number of permutations
//------------------------------------

public BigInteger getTotal () {
  return total;
}

//-----------------------------
// Are there more permutations?
//-----------------------------

public boolean hasMore () {
  return numLeft.compareTo (BigInteger.ZERO) == 1;
}

//------------------
// Compute factorial
//------------------

private static BigInteger getFactorial (int n) {
  BigInteger fact = BigInteger.ONE;
  for (int i = n; i > 1; i--) {
    fact = fact.multiply (new BigInteger (Integer.toString (i)));
  }
  return fact;
}

//-----------------------------------------------------------
// Generate next permutation (algorithm from Rosen p. 284)
//-----------------------------------------------------------

public int[] getNext () {

  if (numLeft.equals (total)) {
    numLeft = numLeft.subtract (BigInteger.ONE);
    return a;
  }

  int temp;

  // Find largest index j with a[j] < a[j+1]

  int j = a.length - 2;
  while (a[j] > a[j+1]) {
    j--;
  }

  // Find index k such that a[k] is smallest integer
  // greater than a[j] to the right of a[j]

  int k = a.length - 1;
  while (a[j] > a[k]) {
    k--;
  }

  // Interchange a[j] and a[k]

  temp = a[k];
  a[k] = a[j];
  a[j] = temp;

  // Put tail end of permutation after jth position in increasing order

  int r = a.length - 1;
  int s = j + 1;

  while (r > s) {
    temp = a[s];
    a[s] = a[r];
    a[r] = temp;
    r--;
    s++;
  }

  numLeft = numLeft.subtract (BigInteger.ONE);
  return a;

}
}
```

2.2. GCD

In mathematics, the greatest common divisor (gcd), also known as the greatest common denominator, greatest common factor (gcf), or highest common factor (hcf), of two or more non-zero integers, is the largest positive integer that divides the numbers without a remainder. For example, *the GCD of 8 and 12 is 4.*
This notion can be extended to polynomials.

In mathematics, the Euclidean algorithm[a] (also called Euclid's algorithm) is an efficient method for computing the greatest common divisor (GCD), also known as the greatest common factor (GCF) or highest common factor (HCF). It is named after the Greek mathematician Euclid, who described it in Books VII and X of his *Elements*.[1]

```
function gcd(a, b)
      while b ≠ 0
      t := b
      b := a mod b
      a := t
      return a
```

Subtraction-based version:

```
function gcd(a, b)
      if a = 0
      return b
      while b ≠ 0
      if a > b
            a := a − b
      else
            b := b − a
      return a
```

Recursive version:

```
function gcd(a, b)
      if b = 0
      return a
      else
      return gcd(b, a mod b)
```

2.3. Extended euclidean algorithm and Bézout's Identity

The **extended Euclidean algorithm** is an extension to the Euclidean algorithm for finding the greatest common divisor (GCD) of integers *a* and *b*: it also finds the integers *x* and *y* in Bézout's identity: ax + by = gcd(a, b) (Typically either x or y is negative).
The extended Euclidean algorithm is particularly useful when *a* and *b* are coprime, since *x* is the modular multiplicative inverse of *a* modulo *b*.

```
function extended_gcd(a, b)
    if a mod b = 0
        return {0, 1}
    else
        {x, y} := extended_gcd(b, a mod b)
        return {y, x - y * (a div b)}
```

Implementación:

```
public class EGCD
{
      
      static class Punto
      {
            int x, y;
            
            Punto(int xx, int yy)
            {
                  x = xx;
                  y = yy;
            }
      }
      
      public static Punto extendedGCD(int a, int b)
```

```
	{
		if((a % b) == 0)
			return new Punto(0, 1);
		Punto p = extendedGCD(b, a % b);
		return new Punto(p.y, p.x - (p.y * (a / b)));
	}
}
```

The Bézout numbers *x* and *y* as above can be determined with the extended Euclidean algorithm. However, they are not unique. If one solution is given by (*x*, *y*), then there are infinitely many solutions. These are given by:

{(x + k * b / gcd(a, b), y - k * a / gcd(a, b) | k in Z}

Bézout's identity

In number theory, **Bézout's identity** or **Bézout's lemma**, named after Étienne Bézout, is a linear diophantine equation. It states that if *a* and *b* are nonzero integers with greatest common divisor *d*, then there exist integers *x* and *y* (called *Bézout numbers* or *Bézout coefficients*) such that

$$ax + by = d.$$

Additionally, *d* is the smallest positive integer for which there are integer solutions *x* and *y* for the preceding equation.
The Bézout numbers *x* and *y* as above can be determined with the extended Euclidean algorithm. However, they are not unique. If one solution is given by (*x*, *y*), then there are infinitely many solutions. These are given by

$$\left\{ \left(x + \frac{kb}{\gcd(a,b)},\ y - \frac{ka}{\gcd(a,b)} \right) \mid k \in \mathbb{Z} \right\}.$$

Linear Diophantine equations

Linear Diophantine equations take the form *ax* + *by* = *c*. If *c* is the greatest common divisor of *a* and *b* then this is *Bézout's identity,* and the equation has an infinite number of solutions. These can be found by applying the extended Euclidean algorithm. It follows that there are also infinite solutions if *c* is a multiple of the greatest common divisor of *a* and *b*. If *c* is not a multiple of the greatest common divisor of *a* and *b*, then the Diophantine equation *ax* + *by* = *c* has no solution

2.4. Sieve

Sieve theory is a set of general techniques in number theory, designed to count, or more realistically to estimate the size of, sifted sets of integers. The primordial example of a sifted set is the set of prime numbers up to some prescribed limit *X*. Correspondingly, the primordial example of a sieve is the sieve of Eratosthenes, or the more general Legendre sieve. The direct attack on prime numbers using these methods soon reaches apparently insuperable obstacles, in the way of the accumulation of error terms. In one of the major strands of number theory in the twentieth century, ways were found of avoiding some of the difficulties of a frontal attack with a naive idea of what sieving should be.

One successful approach is to approximate a specific sifted set of numbers (e.g. the set of prime numbers) by another, simpler set (e.g. the set of almost primenumbers), which is typically somewhat larger than the original set, and easier to analyze. More sophisticated sieves also do not work directly with sets *per se*, but instead count them according to carefully chosen weight functions on these sets (options for giving some elements of these sets more "weight" than others). Furthermore, in some modern applications, sieves are used not to estimate the size of a sifted set, but to produce a function that is large on the set and mostly small outside it, while being easier to analyze than the characteristic function of the set.

Modern sieves include the Brun sieve, the Selberg sieve, and the large sieve. One of the original purposes of sieve theory was to try to prove conjectures in number theory such as the twin prime conjecture. While the original broad aims of sieve theory still are largely unachieved, there has been some partial successes, especially in combination with other number theoretic tools. Highlights include:

Brun's theorem, which asserts that the sum of the reciprocals of the twin primes converges (whereas the sum of the reciprocals of the primes themselves diverge);

Chen's theorem, which shows that there are infinitely many primes *p* such that *p* + 2 is either a prime or a semiprime (the product of two primes); a closely related theorem of Chen Jingrun asserts that every sufficiently large even number is the sum of a prime and another number which is either a prime or a semiprime. These can be considered to be near-misses to the twin prime conjecture and the Goldbach conjecture respectively.

The *fundamental lemma of sieve theory*, which (very roughly speaking) asserts that if one is sifting a set of *N* numbers, then one can accurately estimate the number of elements left in the sieve after *N*ε iterations provided that ε is sufficiently small (fractions such as 1/10 are quite typical here). This lemma is usually too weak to sieve out primes (which generally require something like *N*1 / 2 iterations), but can be enough to obtain results regarding almost primes.

The *Friedlander–Iwaniec theorem*, which asserts that there are infinitely many primes of the form *a*2 + *b*4.

The techniques of sieve theory can be quite powerful, but they seem to be limited by an obstacle known as the *parity problem*, which roughly speaking asserts that sieve theory methods have extreme difficulty distinguishing between numbers with an odd number of prime factors, and numbers with an even number of prime factors. This parity problem is still not very well understood.

Compared with other methods in number theory, sieve theory is comparatively *elementary*, in the sense that it does not necessarily require sophisticated concepts from either algebraic number theory or analytic number theory. Nevertheless, the more advanced sieves can still get very intricate and delicate (especially when combined with other deep techniques in number theory), and entire textbooks have been devoted to this single subfield of number theory; a classic reference is (Halberstam & Richert 1974).
The sieve methods discussed in this article are not closely related to the integer factorization sieve methods such as the quadratic sieve and the general number field sieve. Those factorization methods use the idea of the sieve of Eratosthenes to determine efficiently which members of a list of numbers can be completely factored into small primes.

2.4.1. Sundaram

In mathematics, the **sieve of Sundaram** is a simple deterministic algorithm for finding all prime numbers up to a specified integer. It was discovered in 1934 by S. P. Sundaram, an Indian student from Sathyamangalam.[1][2]

Start with a list of the integers from 1 to *n*. From this list, remove all numbers of the form *i + j + 2ij* where:

- $i, j \in \mathbb{N},\ 1 \le i \le j$
- $i + j + 2ij \le n$

The remaining numbers are doubled and incremented by one, giving a list of the odd prime numbers (i.e., all primes except the only even prime 2). The sieve of Sundaram is equivalent to the sieve of Eratosthenes, except that the initial list corresponds only to the *odd* integers; the work of "crossing out" the multiples of 2 is done by the final double-and-increment step. Whenever Eratosthenes' method would cross out *k* different multiples of a prime *2i+1*, Sundaram's method crosses out *i + j(2i+1)* for $1 \le j \le \lfloor k/2 \rfloor$.

The sieve of Sundaram finds the primes less than *n* in Θ(*n* log *n*) operations using Θ(*n*) bits of memory.

2.4.2. Eratosthenes

In mathematics, the **Sieve of Eratosthenes** (Greek: κόσκινον Ἐρατοσθένους) is a simple, ancient algorithm for finding all prime numbers up to a specified integer.[1] It works efficiently for the smaller primes (below 10 million).[2] It was created by Eratosthenes, an ancient Greek mathematician. However, none of his mathematical works survived—the sieve was described and attributed to Eratosthenes in the *Introduction to Arithmetic* byNicomachus.[3]

To find all the prime numbers less than or equal to a given integer *n* by Eratosthenes' method:

1. Create a list of consecutive integers from two to *n*: (2, 3, 4, ..., *n*).
2. Initially, let *p* equal 2, the first prime number.
3. Strike from the list all multiples of *p* less than or equal to *n*. (*2p, 3p, 4p, etc.*)
4. Find the first number remaining on the list after *p* (this number is the next prime); replace *p* with this number.
5. Repeat steps 3 and 4 until *p*2 is greater than *n*.
6. All the remaining numbers in the list are prime.

2.4.3. Atkin

In mathematics, the sieve of Atkin is a fast, modern algorithm for finding all prime numbers up to a specified integer. It is an optimized version of the ancientsieve of Eratosthenes, but does some preliminary work and then marks off multiples of primes squared, rather than multiples of primes. It was created by A. O. L. Atkin and Daniel J. Bernstein.[1]

In the algorithm:

- All remainders are modulo-sixty remainders (divide the number by sixty and return the remainder).
- All numbers, including *x* and *y*, are whole numbers (positive integers).
- Flipping an entry in the sieve list means to change the marking (prime or nonprime) to the opposite marking.

1. Create a results list, filled with 2, 3, and 5.
2. Create a sieve list with an entry for each positive integer; all entries of this list should initially be marked nonprime.
   3. For each entry number *n* in the sieve list, with modulo-sixty remainder *r* :
      - If *r* is 1, 13, 17, 29, 37, 41, 49, or 53, flip the entry for each possible solution to 4*x*2 + *y*2 = *n*.

- If *r* is 7, 19, 31, or 43, flip the entry for each possible solution to 3*x*2 + *y*2 = *n*.
- If *r* is 11, 23, 47, or 59, flip the entry for each possible solution to 3*x*2 − *y*2 = *n* when *x* > *y*.
- If *r* is something else, ignore it completely.

4. Start with the lowest number in the sieve list.
5. Take the next number in the sieve list still marked prime.
6. Include the number in the results list.
7. Square the number and mark all multiples of that square as nonprime.
8. Repeat steps five through eight.

This results in numbers with an odd number of solutions to the corresponding equation being prime, and an even number being nonprime.

2.5. Binomial coefficients

In mathematics, the **binomial coefficient** $\binom{n}{k}$ is the coefficient of the *x k* term in the polynomial expansion of the binomial power (1 + *x*) *n*.

In combinatorics, $\binom{n}{k}$ is interpreted as the number of *k*-element subsets (the *k*-combinations) of an *n*-element set, that is the number of ways that*k* things can be "chosen" from a set of *n* things. Hence, $\binom{n}{k}$ is often read as "*n* choose *k*" and is called the **choose function** of *n* and *k*.

The notation $\binom{n}{k}$ was introduced by Andreas von Ettingshausen in 1826,[1] although the numbers were already known centuries before that (seePascal's triangle). Alternative notations include C(*n*, *k*), *nCk*, *nCk*, C^n_k, C^k_n,[2] in all of which the C stands for combinations or *choices*.

For natural numbers (taken to include 0) *n* and *k*, the binomial coefficient $\binom{n}{k}$ can be defined as the coefficient of the monomial *Xk* in the expansion of (1 + *X*)*n*. The same coefficient also occurs (if *k* ≤ *n*) in the binomial formula

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$

(valid for any elements *x*,*y* of a commutative ring), which explains the name "binomial coefficient".
Another occurrence of this number is in combinatorics, where it gives the number of ways, disregarding order, that a *k* objects can be chosen from among *n* objects; more formally, the number of *k*-element subsets (or *k*-combinations)

of an *n*-element set. This number can be seen to be equal to the one of the first definition, independently of any of the formulas below to compute it: if in each of the *n* factors of the power (1 + *X*)*n* one temporarily labels the term *X* with an index *i* (running from 1 to *n*), then each subset of *k* indices gives after expansion a contribution *Xk*, and the coefficient of that monomial in the result will be the number of such subsets. This shows in particular that $\binom{n}{k}$ is a natural number for any natural numbers *n* and*k*. There are many other combinatorial interpretations of binomial coefficients (counting problems for which the answer is given by a binomial coefficient expression), for instance the number of words formed of *n* bits (digits 0 or 1) whose sum is *k*, but most of these are easily seen to be equivalent to counting *k*-combinations.

Several methods exist to compute the value of $\binom{n}{k}$ without actually expanding a binomial power or counting *k*-combinations.

[edit]Recursive formula

One has a recursive formula for binomial coefficients

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{for all integers } n, k > 0,$$

with as initial values

$$\binom{n}{0} = 1 \quad \text{for all } n \in \mathbb{N},$$
$$\binom{0}{k} = 0 \quad \text{for all integers } k > 0.$$

The formula follows either from tracing the contributions to *Xk* in (1 + *X*)*n*−1(1 + *X*), or by counting *k*-combinations of {1, 2, ..., *n*} that contain *n* and that do not contain *n* separately. It follows easily that $\binom{n}{k} = 0$ when *k* > *n*, and $\binom{n}{n} = 1$ for all *n*, so the recursion can stop when reaching such cases. This recursive formula then allows the construction of Pascal's triangle.

[edit]Multiplicative formula

A more efficient method to compute individual binomial coefficients is given by the formula

$$\binom{n}{k} = \frac{n^{\underline{k}}}{k!} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots 1}.$$

This formula is easiest to understand for the combinatorial interpretation of binomial coefficients. The numerator gives the number of ways to select a sequence of *k* distinct objects, retaining the order of selection, from a set of *n* objects. The denominator counts the number of distinct sequences that define the same *k*-combination when order is disregarded.

[edit]Factorial formula

Finally there is a formula using factorials that is easy to remember:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \quad \text{for } \ 0 \leq k \leq n.$$

where n! denotes the factorial of n. This formula follows from the multiplicative formula above by multiplying numerator and denominator by (n − k)!; as a consequence it involves many factors common to numerator and denominator. It is less practical for explicit computation unless common factors are first canceled (in particular since factorial values grow very rapidly). The formula does exhibit a symmetry that is less evident from the multiplicative formula (though it is from the definitions)

$$\binom{n}{k} = \binom{n}{n-k} \quad \text{for } \ 0 \leq k \leq n.$$

[edit]Generalization and connection to the binomial series

The multiplicative formula allows the definition of binomial coefficients to be extended[note 1] by replacing n by an arbitrary number α (negative, real, complex) or even an element of anycommutative ring in which all positive integers are invertible:

$$\binom{\alpha}{k} = \frac{\alpha^{\underline{k}}}{k!} = \frac{\alpha(\alpha-1)(\alpha-2)\cdots(\alpha-k+1)}{k(k-1)(k-2)\cdots 1} \quad \text{for } k \in \mathbb{N} \text{ and arbitrary } \alpha.$$

With this definition one has a generalization of the binomial formula (with one of the variables set to 1), which justifies still calling the $\binom{\alpha}{k}$ binomial coefficients:

$$(1+X)^\alpha = \sum_{k=0}^{\infty} \binom{\alpha}{k} X^k.$$

This formula is valid for all complex numbers α and X with |X| < 1. It can also be interpreted as an identity of formal power series in X, where it actually can serve as definition of arbitrary powers of series with constant coefficient equal to 1; the point is that with this definition all identities hold that one expects for exponentiation, notably

$$(1+X)^\alpha(1+X)^\beta = (1+X)^{\alpha+\beta} \quad \text{and} \quad ((1+X)^\alpha)^\beta = (1+X)^{\alpha\beta}.$$

If α is a nonnegative integer n, then all terms with k > n are zero, and the infinite series becomes a finite sum, thereby recovering the binomial formula. However for other values of α, including negative integers and rational numbers, the series is really infinite.

Pascal's rule is the important recurrence relation

$$\binom{n}{k} + \binom{n}{k+1} = \binom{n+1}{k+1}, \qquad (3)$$

which can be used to prove by mathematical induction that $\binom{n}{k}$ is a natural number for all n and k, (equivalent to the statement that k! divides the product of k consecutive integers), a fact that is not immediately obvious from formula (1).

Row number *n* contains the numbers $\binom{n}{k}$ for *k* = 0,…,*n*. It is constructed by starting with ones at the outside and then always adding two adjacent numbers and writing the sum directly underneath. This method allows the quick calculation of binomial coefficients without the need for fractions or multiplications. For instance, by looking at row number 5 of the triangle, one can quickly read off that

(*x* + *y*)5 = **1** *x*5 + **5** *x*4*y* + **10** *x*3*y*2 + **10** *x*2*y*3 + **5** *x* *y*4 + **1** *y*5.

The differences between elements on other diagonals are the elements in the previous diagonal, as a consequence of the recurrence relation (3) above.

```
unsigned long long choose(unsigned n, unsigned k) {
    if (k > n)
        return 0;

    if (k > n/2)
        k = n-k; // Take advantage of symmetry

    long double accum = 1;
    unsigned i;
    for (i = 1; i <= k; i++)
         accum = accum * (n-k+i) / i;

    return accum + 0.5; // avoid rounding error
}
```

Binomial coefficients are of importance in combinatorics, because they provide ready formulas for certain frequent counting problems:

- There are $\binom{n}{k}$ ways to choose *k* elements from a set of *n* elements. See Combination.
- There are $\binom{n+k-1}{k}$ ways to choose *k* elements from a set of *n* if repetitions are allowed. See Multiset.
- There are $\binom{n+k}{k}$ strings containing *k* ones and *n* zeros.
- There are $\binom{n+1}{k}$ strings consisting of *k* ones and *n* zeros such that no two ones are adjacent.

2.6. Fibonacci numbers

In mathematics, the **Fibonacci numbers** are the numbers in the following sequence:

$$0,\;1,\;1,\;2,\;3,\;5,\;8,\;13,\;21,\;34,\;55,\;89,\;144,\;\ldots.$$

By definition, the first two Fibonacci numbers are 0 and 1, and each subsequent number is the sum of the previous two. Some sources omit the initial 0, instead beginning the sequence with two 1s.
In mathematical terms, the sequence *Fn* of Fibonacci numbers is defined by the recurrence relation

$$F_n = F_{n-1} + F_{n-2},$$

with seed values

$$F_0 = 0 \quad \text{and} \quad F_1 = 1.$$

The Fibonacci sequence is named after Leonardo of Pisa, who was known as Fibonacci (a contraction of *filius Bonacci*, "son of Bonaccio"). Fibonacci's 1202 book *Liber Abaci* introduced the sequence to Western European mathematics, although the sequence had been previously described in Indian mathematics.[2][3]

Closed-form expression

Like every sequence defined by linear recurrence, the Fibonacci numbers have a closed-form solution. It has become very well known as **Binet's formula**, even though it was already known by Abraham de Moivre:[14]

$$F\left(n\right) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}} = \frac{\varphi^n - (-1/\varphi)^n}{\sqrt{5}},$$

where

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.6180339887\ldots$$

Since $|1-\varphi|^n/\sqrt{5} < 1/2$ for all $n \geq 0$, the number *F*(*n*) is the closest integer to $\varphi^n/\sqrt{5}$. Therefore it can be found by rounding, or in terms of the floor function:

$$F(n) = \left\lfloor \frac{\varphi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor,\ n \geq 0.$$

Similarly, if you already know that the number **F** is a Fibonacci number, you can determine its index within the sequence by

$$n = \left\lfloor \log_\varphi \left(F \cdot \sqrt{5}\right) + \frac{1}{2} \right\rfloor$$

A 2-dimensional system of linear difference equations that describes the Fibonacci sequence is

$$\begin{pmatrix} F_{k+2} \\ F_{k+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{k+1} \\ F_k \end{pmatrix}$$

or

$$\vec F_{k+1} = A \vec F_k \,.$$

Periodicity modulo *n*

Main article: Pisano period

It may be seen that if the members of the Fibonacci sequence are taken mod *n*, the resulting sequence must be periodic with period at most *n*2. The lengths of the periods for various *n*form the so-called Pisano periods (sequence A001175 in OEIS). Determining the Pisano periods in general is an open problem, [*citation needed*] although for any particular *n* it can be solved as an instance of cycle detection.

The Fibonacci numbers can be found in different ways in the sequence of binary strings.

- The number of binary strings of length *n* without consecutive 1s is the Fibonacci number *Fn*+2. For example, out of the 16 binary strings of length 4, there are *F*6 = 8 without consecutive 1s - they are 0000, 1000, 0100, 0010, 1010, 0001, 1001 and 0101. By symmetry, the number of strings of length *n* without consecutive 0s is also *Fn*+2.
- The number of binary strings of length *n* without an odd number of consecutive 1s is the Fibonacci number *Fn+1*. For example, out of the 16 binary strings of length 4, there are *F*5 = 5 without an odd number of consecutive 1s - they are 0000, 0011, 0110, 1100, 1111.
- The number of binary strings of length *n* without an even number of consecutive 0s or 1s is 2*Fn*. For example, out of the 16 binary strings of length 4, there are 2*F*4 = 6 without an even number of consecutive 0s or 1s - they are 0001, 1000, 1110, 0111, 0101, 1010.

2.7. Euler's totient function

In number theory, the **totient** $\varphi(n)$ of a positive integer *n* is defined to be the number of positive integers less than or equal to *n* that are coprime to *n* (i.e. having no common positive factors other than 1). In particular $\varphi(1) = 1$ since 1 is coprime to itself (1 being the only natural number with this property). For example, $\varphi(9) = 6$ since the six numbers 1, 2, 4, 5, 7 and 8 are coprime to 9. The function φ so defined is the **totient function.** The totient is usually called the **Euler totient**or **Euler's totient**, after the Swiss mathematician Leonhard Euler, who studied it. The totient function is also called **Euler's phi function** or simply the **phi function**, since it is commonly denoted by the Greek letter phi (φ). The **cototient** of *n* is defined

as $n - \varphi(n)$, in other words the number of positive integers less than or equal to *n* that are **not** coprime to *n*.
The totient function is important mainly because it gives the size of the multiplicative group of integers modulo *n*. More precisely, $\varphi(n)$ is the order of the group of unitsof the ring $\mathbb{Z}/n\mathbb{Z}$. This fact, together with Lagrange's theorem on the possible sizes of subgroups of a group, provides a proof for Euler's theorem that $a^{\varphi(n)} \equiv 1 \pmod{n}$ for all *a* coprime to *n*. The totient function also plays a key role in the definition of the RSA encryption system.

Computing Euler's function

If p is prime, then for integer $k \geq 1, \varphi(p^k) = (p-1)p^{k-1}$. Moreover, φ is a multiplicative function; if *m* and *n* are coprime then $\varphi(mn) = \varphi(m)\varphi(n)$. (Sketch of proof: let *A*, *B*, *C* be the sets of residue classes modulo-and-coprime-to *m*, *n*, *mn* respectively; then there is a bijection between *A* × *B* and *C*, by the Chinese remainder theorem.) The value of $\varphi(n)$ can thus be computed using thefundamental theorem of arithmetic: if

$$n = p_1^{k_1} \cdots p_r^{k_r}$$

where each *pi* is a distinct prime, then

$$\varphi(n) = (p_1 - 1)p_1^{k_1 - 1} \times (p_2 - 1)p_2^{k_2 - 1} \cdots \times (p_r - 1)p_r^{k_r - 1}.$$

This last formula is an Euler product and is often written in the equivalent form(multiplying top and bottom by $n = p_1^{k_1} \cdots p_r^{k_r}$)

$$\varphi(n) = n(p_1 - 1)p_1^{-1} \times (p_2 - 1)p_2^{-1} \cdots \times (p_r - 1)p_r^{-1} = n \cdot \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

with the product ranging only over the distinct primes *p* dividing *n*.

Computing example

$$\varphi(36) = \varphi\left(2^2 3^2\right) = 36\left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{3}\right) = 36 \cdot \frac{1}{2} \cdot \frac{2}{3} = 12.$$

In words, this says that the distinct prime factors of 36 are 2 and 3; half of the thirty-six integers from 1 to 36 are divisible by 2, leaving eighteen; a third of those are divisible by 3, leaving twelve coprime to 36. And indeed there are twelve: 1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, and 35.
Moreover Schramm (2008) has shown that:

$$\varphi(n) = \sum_{k=1}^{n} \gcd(k, n) \cdot \exp\left(\frac{-2\pi i k}{n}\right).$$

Properties

- $\varphi(p^\alpha) = p^\alpha - p^{\alpha - 1}$ **for prime *p* and** $\alpha \geq 1$.

- $\varphi(mn) = \varphi(m)\varphi(n)(d/\varphi(d))$ where d = gcd(m,n).
- $a \mid b$ implies $\varphi(a) \mid \varphi(b)$.
- $\varphi(n)$ is even for $n \geq 3$. Moreover, if *n* has *r* distinct odd prime factors, $2^r \mid \varphi(n)$.

The number $\varphi(n)$ is also equal to the number of possible generators of the cyclic group *Cn* (and therefore also to the degree of the cyclotomic polynomial φ_n). Since every element of *Cn* generates a cyclicsubgroup and the subgroups of *Cn* are of the form *Cd* where *d* divides *n* (written as *d* | *n*), we get

$$\sum_{d\mid n} \varphi(d) = n$$

where the sum extends over all positive divisors *d* of *n*.

We can now use the Möbius inversion formula to "invert" this sum and get another formula for $\varphi(n)$:

$$\varphi(n) = \sum_{d\mid n} d \cdot \mu\left(\frac{n}{d}\right)$$

where *μ* is the usual Möbius function defined on the positive integers.

According to Euler's theorem, if *a* is coprime to *n*, that is, gcd(*a*, *n*) = 1, then

$$a^{\varphi(n)} \equiv 1 \mod n.$$

This follows from Lagrange's theorem and the fact that *a* belongs to the multiplicative group of $\mathbb{Z}/n\mathbb{Z}$ iff *a* is coprime to *n*.

Other formulas involving Euler's function

- $\varphi\left(n^m\right) = n^{m-1}\varphi(n) \text{ for } m \geq 1$
- $\text{For any } a, n > 1, \, n \mid \varphi(a^n - 1)$
- For any *a* > 1 and *n* > 6 such that $4 \nmid n$ there exists an $l \geq 2n$ such that $l \mid \varphi(a^n - 1)$.
- $$\sum_{d\mid n} \frac{\mu^2(d)}{\varphi(d)} = \frac{n}{\varphi(n)}$$
- $$\sum_{\substack{1\le k\le n \\ (k,n)=1}} k = \frac{1}{2} n\varphi(n) \text{ for } n > 1$$ See proof here.
- $$\sum_{k=1}^{n} \varphi(k) = \frac{1}{2}\left(1 + \sum_{k=1}^{n} \mu(k) \left\lfloor \frac{n}{k} \right\rfloor^2\right)$$

- $\sum_{k=1}^{n} \frac{\varphi(k)}{k} = \sum_{k=1}^{n} \frac{\mu(k)}{k} \left\lfloor \frac{n}{k} \right\rfloor$
- $\sum_{k=1}^{n} \frac{k}{\varphi(k)} = \mathcal{O}(n)$
- $\sum_{k=1}^{n} \frac{1}{\varphi(k)} = \mathcal{O}(\log(n))$
- $\sum_{\substack{1 \le k \le n \\ (k,m)=1}} 1 = n\frac{\varphi(m)}{m} + \mathcal{O}\left(2^{\omega(m)}\right),$

where *m* > 1 is a positive integer and ω(*m*) designates the number of distinct prime factors of *m*. (This formula counts the number of naturals less than or equal to *n* and relatively prime to *m*, additional material is listed among the external links.)

2.8. LCM

In arithmetic and number theory, the lowest common multiple or (LCM) least common multiple or smallest common multiple of two rational numbers *a* and *b* is the smallest positive rational number that is an (integer) multiple of both *a* and *b*. (The definition may be extended to any two real numbers whose ratio is a rational number.)[*citation needed*] Since it is a multiple, it can be divided by *a* and *b* without a remainder. If either *a* or *b* is 0, so that there is no such positive integer, then LCM(*a*, *b*) is defined to be zero.
The definition may be generalized to more than two rational numbers: The lowest common multiple of the rational numbers *a*1, ..., *an* is the smallest positive rational number that is a multiple of each of *a*1, ..., *an*.
The following formula reduces the problem of computing the least common multiple to the problem of computing the greatest common divisor (GCD):

$$\operatorname{lcm}(a,b) = \frac{|a \cdot b|}{\gcd(a,b)}.$$

There are fast algorithms for computing the GCD that do not require the numbers to be factored, such as the Euclidean algorithm. To return to the example above,

$$\operatorname{lcm}(21,6) = \frac{21 \cdot 6}{\gcd(21,6)} = \frac{21 \cdot 6}{3} = 21 \cdot \frac{6}{3} = 21 \cdot 2 = 42.$$

Because gcd(*a*, *b*) is a divisor of both *a* and *b*, it's more efficient to compute the LCM by dividing *before* multiplying:

$$\operatorname{lcm}(a,b) = \left(\frac{|a|}{\gcd(a,b)}\right) \cdot |b| = \left(\frac{|b|}{\gcd(a,b)}\right) \cdot |a|.$$

2.9. Catalan numbers

In combinatorial mathematics, the Catalan numbers form a sequence of natural numbers that occur in various counting problems, often involving recursively defined objects. They are named after the Belgian mathematician Eugène Charles Catalan (1814–1894).
The *n*th Catalan number is given directly in terms of binomial coefficients by

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} \qquad \text{for } n \ge 0.$$

Properties
An alternative expression for *Cn* is

$$C_n = \binom{2n}{n} - \binom{2n}{n+1} \quad \text{for } n \ge 0,$$

which is equivalent to the expression given above because $\binom{2n}{n+1} = \frac{n}{n+1}\binom{2n}{n}$.
This shows that *Cn* is a natural number, which is not *a priori* obvious from the first formula given. This expression forms the basis for André's proof of the correctness of the formula (see below under second proof).
The Catalan numbers satisfy the recurrence relation

$$C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^{n} C_i\, C_{n-i} \quad \text{for } n \ge 0;$$

moreover,

$$C_n = \frac{1}{n+1}\sum_{i=0}^{n}\binom{n}{i}^2$$

They also satisfy:

$$C_0 = 1 \quad \text{and} \quad C_{n+1} = \frac{2(2n+1)}{n+2}C_n,$$

which can be a more efficient way to calculate them.
Asymptotically, the Catalan numbers grow as

$$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

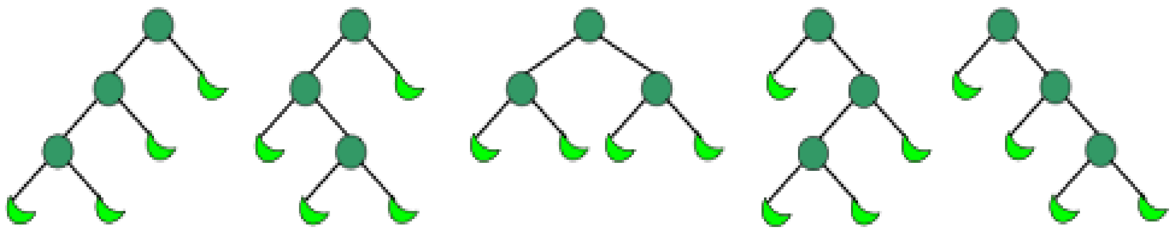
in the sense that the quotient of the *n*th Catalan number and the expression on the right tends towards 1 as *n* → +∞. (This can be proved by using Stirling's approximation for *n*!.)

The only Catalan numbers *Cn* that are odd are those for which *n* = 2*k* − 1. All others are even.

[edit]Applications in combinatorics

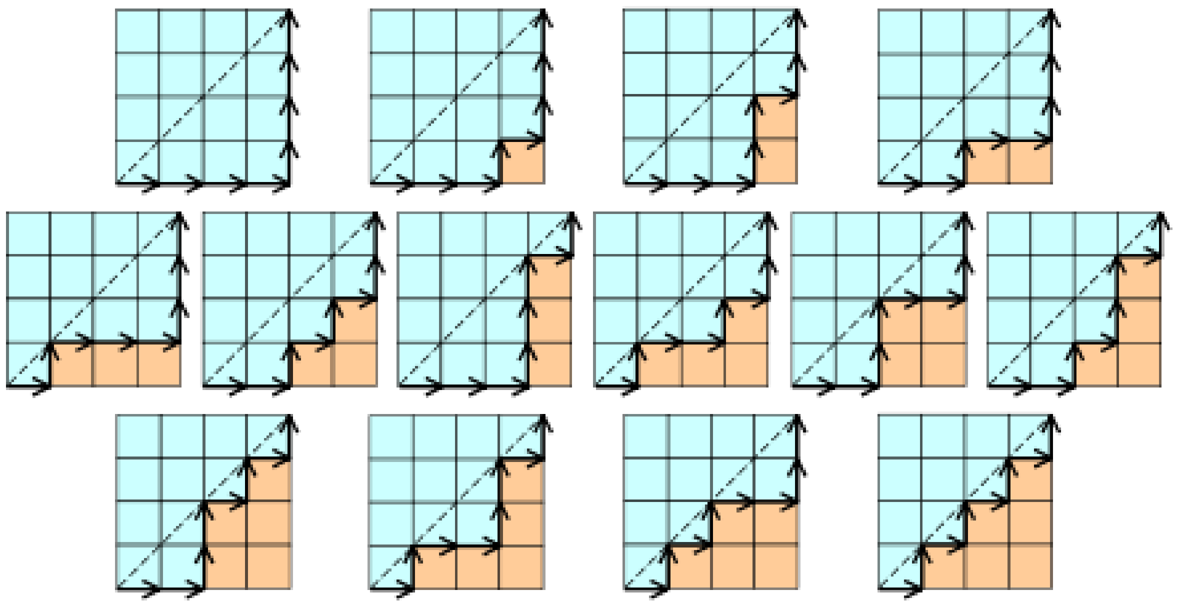
There are many counting problems in combinatorics whose solution is given by the Catalan numbers. The book *Enumerative Combinatorics: Volume 2* by combinatorialist Richard P. Stanley contains a set of exercises which describe 66 different interpretations of the Catalan numbers. Following are some examples, with illustrations of the cases *C*3 = 5 and *C*4 = 14.

- *Cn is the number of* ***Dyck words*** *of length 2n. A Dyck word is a string consisting of n X's and n Y's such that no initial segment of the string has more Y's than X's (see also Dyck language). For example, the following are the Dyck words of length 6:*
  XXXYYY XYXXYY XYXYXY XXYYXY XXYXYY.
- Re-interpreting the symbol X as an open parenthesis and Y as a close parenthesis, *Cn* counts the number of expressions containing *n* pairs of parentheses which are correctly matched:
  ((())) ()(()) ()()() (())() (()())
- *Cn is the number of different ways n + 1 factors can be completely parenthesized (or the number of ways of associating n applications of a binary operator). For n = 3, for example, we have the following five different parenthesizations of four factors:*
  $$((ab)c)d \quad (a(bc))d \quad (ab)(cd) \quad a((bc)d) \quad a(b(cd))$$
- Successive applications of a binary operator can be represented in terms of a full binary tree. (A rooted binary tree is *full* if every vertex has either two children or no children.) It follows that *Cn* is the number of full binary trees with *n* + 1 leaves:

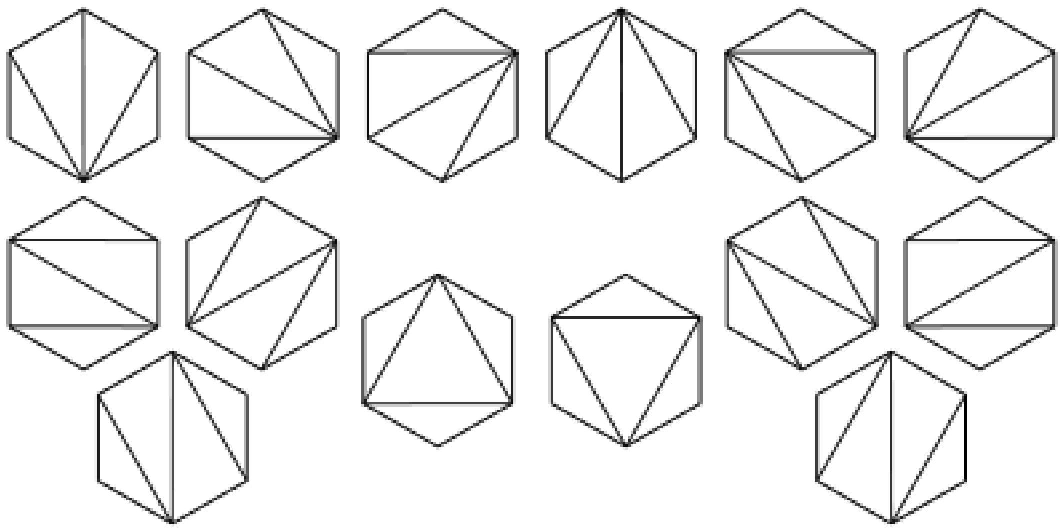


If the leaves are labelled, we have the quadruple factorial numbers.

- *Cn is the number of non-isomorphic ordered trees with n + 1 vertices. (An ordered tree is a rooted tree in which the children of each vertex are given a fixed left-to-right order.)*
- *Cn is the number of* ***monotonic paths*** *along the edges of a grid with n × n square cells, which do not pass above the diagonal. A monotonic path is one which starts in the lower left corner, finishes in the upper right corner, and consists entirely of edges pointing rightwards or upwards. Counting such paths is equivalent to counting Dyck words: X stands for "move right" and Y stands for "move up". The following diagrams show the case n = 4:*



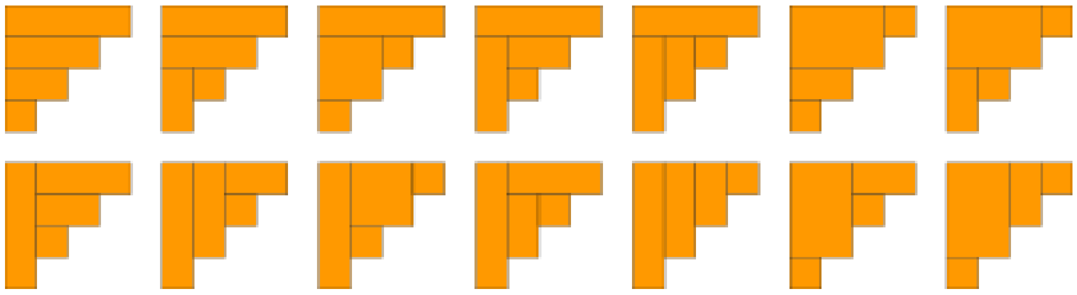
- *Cn is the number of different ways a convex polygon with n + 2 sides can be cut into triangles by connecting vertices with straight lines. The following hexagons illustrate the case n = 4:*



- *Cn is the number of stack-sortable permutations of {1, ..., n}. A permutation w is called* ***stack-sortable*** *if S(w) = (1, ..., n), where S(w) is defined recursively as follows: write w = unvwhere n is the largest element in w and u and v are shorter sequences, and set S(w) = S(u)S(v)n, with S being the identity for one-element sequences. These are the permutations that avoid the pattern 231.*
- *Cn is the number of permutations of {1, ..., n} that avoid the pattern 123; that is, the number of permutations with no three-term increasing subsequence. For n = 3, these permutations are 132, 213, 231, 312 and 321. For n = 4, they are 1432, 2143, 2413,*

2431, 3142, 3214, 3241, 3412, 3421, 4132, 4213, 4231, 4312 and 4321.

- *Cn is the number of noncrossing partitions of the set {1, ..., n}. A fortiori, Cn never exceeds the nth Bell number. Cn is also the number of noncrossing partitions of the set {1, ..., 2n} in which every block is of size 2. The conjunction of these two facts may be used in a proof by mathematical induction that all of the free cumulants of degree more than 2 of the Wigner semicircle law are zero. This law is important in free probability theory and the theory of random matrices.*
- *Cn is the number of ways to tile a stairstep shape of height n with n rectangles. The following figure illustrates the case n = 4:*



- *Cn is the number of Young tableaux whose diagram is a 2-by-n rectangle. In other words, it is the number ways the numbers 1, 2, ..., 2n can be arranged in a 2-by-n rectangle so that each row and each column is increasing. As such, the formula can be derived as a special case of the hook formula.*
- *Cn is the number of ways that the vertices of a convex 2n-gon can be paired so that the line segments joining paired vertices do not intersect.*
- *Cn is the number of semiorders on n unlabeled items.*[1]

Hankel matrix

The *n*×*n* Hankel matrix whose (*i*, *j*) entry is the Catalan number *Ci*+*j*-2 has determinant 1, regardless of the value of *n*. For example, for *n* = 4 we have

$$\det\begin{bmatrix}1 & 1 & 2 & 5\\ 1 & 2 & 5 & 14\\ 2 & 5 & 14 & 42\\ 5 & 14 & 42 & 132\end{bmatrix} = 1.$$

Note that if the entries are "shifted", namely the Catalan numbers *Ci*+*j*-1, the determinant is still 1, regardless of the size of *n*. For example, for *n* = 4 we have

$$\det\begin{bmatrix}1 & 2 & 5 & 14\\ 2 & 5 & 14 & 42\\ 5 & 14 & 42 & 132\\ 14 & 42 & 132 & 429\end{bmatrix} = 1.$$

The Catalan numbers form the unique sequence with this property.
[edit]Quadruple factorial

The quadruple factorial is given by $\frac{(2n)!}{n!}$, or $(n+1)!C_n$. This is the solution to labelled variants of the above combinatorics problems. It is entirely distinct from the multifactorials.

2.10. Gaussian elimination

In linear algebra, Gaussian elimination is an algorithm for solving systems of linear equations, finding the rank of a matrix, and calculating the inverse of an invertible square matrix. Gaussian elimination is named after German mathematician and scientist Carl Friedrich Gauss.
Elementary row operations are used to reduce a matrix to row echelon form. Gauss–Jordan elimination, an extension of this algorithm, reduces the matrix further to reduced row echelon form. Gaussian elimination alone is sufficient for many applications.
Algorithm overview
The process of Gaussian elimination has two parts. The first part (Forward Elimination) reduces a given system to either triangular or echelon form, or results in a degenerate equation with no solution, indicating the system has no solution. This is accomplished through the use of elementary row operations. The second step uses back substitution to find the solution of the system above.
Stated equivalently for matrices, the first part reduces a matrix to row echelon form using elementary row operations while the second reduces it to reduced row echelon form, or row canonical form.
Another point of view, which turns out to be very useful to analyze the algorithm, is that Gaussian elimination computes a matrix decomposition. The three elementary row operations used in the Gaussian elimination (multiplying rows, switching rows, and adding multiples of rows to other rows) amount to multiplying the original matrix with invertible matrices from the left. The first part of the algorithm computes an LU decomposition, while the second part writes the original matrix as the product of a uniquely determined invertible matrix and a uniquely determined reduced row-echelon matrix.
Other applications
[edit]Finding the inverse of a matrix
Suppose *A* is a $n \times n$ matrix and you need to calculate its inverse. The $n \times n$ identity matrix is augmented to the right of *A*, forming a $n \times 2n$ matrix (the block matrix *B* = [*A*,*I*]). Through application of elementary row operations and the Gaussian elimination algorithm, the left block of *B* can be reduced to the identity matrix *I*, which leaves *A* − 1 in the right block of *B*.
If the algorithm is unable to reduce *A* to triangular form, then *A* is not invertible.

[edit]General algorithm to compute ranks and bases
The Gaussian elimination algorithm can be applied to any $m \times n$ matrix *A*. If we get "stuck" in a given column, we move to the next column. In this way, for example, some 6×9matrices can be transformed to a matrix that has a reduced row echelon form like

$$T = \begin{bmatrix} 1 & * & 0 & 0 & * & * & 0 & * & 0 \\ 0 & 0 & 1 & 0 & * & * & 0 & * & 0 \\ 0 & 0 & 0 & 1 & * & * & 0 & * & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & * & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(the *'s are arbitrary entries). This echelon matrix *T* contains a wealth of information about *A*: the rank of *A* is 5 since there are 5 non-zero rows in *T*; the vector space spanned by the columns of *A* has a basis consisting of the first, third, fourth, seventh and ninth column of *A* (the columns of the ones in *T*), and the *'s tell you how the other columns of *A* can be written as linear combinations of the basis columns.
[edit]Analysis
Gaussian elimination to solve a system of *n* equations for *n* unknowns requires *n*(*n*+1) / 2 divisions, (2*n*3 + 3*n*2 − 5*n*)/6 multiplications, and (2*n*3 + 3*n*2 − 5*n*)/6 subtractions,[3] for a total of approximately 2*n*3 / 3 operations. So it has a complexity of $\mathcal{O}(n^3)$.
This algorithm can be used on a computer for systems with thousands of equations and unknowns. However, the cost becomes prohibitive for systems with millions of equations. These large systems are generally solved using iterative methods. Specific methods exist for systems whose coefficients follow a regular pattern (see system of linear equations).
The Gaussian elimination can be performed over any field.
Gaussian elimination is numerically stable for diagonally dominant or positive-definite matrices. For general matrices, Gaussian elimination is usually considered to be stable in practice if you use partial pivoting as described below, even though there are examples for which it is unstable.[4]
[edit]Higher order tensors
Gaussian elimination does not generalize in any simple way to higher order tensors (matrices are order 2 tensors); even computing the rank of a tensor of order greater than 2 is a difficult problem.
Pseudocode
As explained above, Gaussian elimination writes a given *m* × *n* matrix *A* uniquely as a product of an invertible *m* × *m* matrix *S* and a row-echelon matrix *T*. Here, *S* is the product of the matrices corresponding to the row operations performed.
The formal algorithm to compute *T* from *A* follows. We write *A*[*i*,*j*] for the entry in row *i*, column *j* in matrix *A*. The transformation is performed "in

place", meaning that the original matrix *A* is lost and successively replaced by *T*.

```
i := 1
j := 1
while (i ≤ m and j ≤ n) do
  Find pivot in column j, starting in row i:
  maxi := i
  for k := i+1 to m do
    if abs(A[k,j]) > abs(A[maxi,j]) then
      maxi := k
    end if
  end for
  if A[maxi,j] ≠ 0 then
    swap rows i and maxi, but do not change the value of i
    Now A[i,j] will contain the old value of A[maxi,j].
    divide each entry in row i by A[i,j]
    Now A[i,j] will have the value 1.
    for u := i+1 to m do
      subtract A[u,j] * row i from row u
      Now A[u,j] will be 0, since A[u,j] - A[i,j] * A[u,j] = A[u,j] - 1 *
A[u,j] = 0.
    end for
    i := i + 1
  end if
  j := j + 1
end while
```

This algorithm differs slightly from the one discussed earlier, because before eliminating a variable, it first exchanges rows to move the entry with the largest absolute value to the "pivot position". Such "partial pivoting" improves the numerical stability of the algorithm; some variants are also in use.

The column currently being transformed is called the pivot column. Proceed from left to right, letting the pivot column be the first column, then the second column, etc. and finally the last column before the vertical line. For each pivot column, do the following two steps before moving on to the next pivot column:

1. Locate the diagonal element in the pivot column. This element is called the pivot. The row containing the pivot is called the pivot row. Divide every element in the pivot row by the pivot to get a new pivot row with a 1 in the pivot position.
2. Get a 0 in each position below the pivot position by subtracting a suitable multiple of the pivot row from each of the rows below it.

Upon completion of this procedure the augmented matrix will be in row-echelon form and may be solved by back-substitution.

Implementación:

```
    static class Matrix
    {
        Rational [][] data;
        int rows, cols;

        Matrix(int rows, int cols)
        {
            this.rows = rows;
            this.cols = cols;
            data = new Rational[rows][cols];
          for(int i = 0; i < rows; i++)
                for(int j = 0; j < cols; j++)
                      data[i][j] = Rational.zero;
        }

        void clonar(Matrix a)
        {
          Matrix nueva = new Matrix(rows, cols);
          for(int i = 0; i < rows; i++)
                for(int j = 0; j < cols; j++)
                      nueva.data[i][j] = data[i][j];
        }

        void swapRow(int row1, int row2)
        {
          Rational[] tmp = data[row2];
          data[row2] = data[row1];
          data[row1] = tmp;
        }

        void multRow(int row, Rational coeff)
        {
            for(int j = 0; j < cols; j++)
                data[row][j] = data[row][j].times(coeff);
        }

        void addRows(int destRow, int srcRow, Rational factor)
        {
            for(int j = 0; j < cols; j++)
                data[destRow][j] = data[destRow][j].plus(data[srcRow]
[j].times(factor));
        }

        void printMat()
        {
            for (int i = 0; i < rows; i++) {
                for (int j = 0; j < cols; j++)
```

```
                    System.out.print(data[i][j] + " ");
                System.out.println();
            }
        }
    };

    static void gaussianElim(Matrix m)
    {
        int rows = m.rows;
        for(int i = 0; i < rows; i++)
        {
            int maxrow = i;
            Rational maxval = m.data[i][i];
            for(int k = i + 1; k < rows; k++)
            {
                if (maxval.abs().compareTo(m.data[k][i].abs()) < 0)
                {
                    maxval = m.data[k][i];
                    maxrow = k;
                }
            }
            if(maxval.compareTo(Rational.zero) == 0)
                return;
            m.swapRow(maxrow, i);
            m.multRow(i, maxval.reciprocal());
            for(int k = 0; k < rows; k++)
                if (k != i)
                    m.addRows(k, i, m.data[k][i].negate());
        }
    }
```

2.11. Orbits, Burnside's lemma

Burnside's lemma, sometimes also called Burnside's counting theorem, the Cauchy-Frobenius lemma or the orbit-counting theorem, is a result in group theory which is often useful in taking account of symmetry when counting mathematical objects. Its various eponyms include William Burnside, George Pólya, Augustin Louis Cauchy, and Ferdinand Georg Frobenius. The result is not due to Burnside himself, who merely quotes it in his book 'On the Theory of Groups of Finite Order', attributing it instead to Frobenius (1887).[1]
In the following, let *G* be a finite group that acts on a set *X*. For each *g* in *G* let *Xg* denote the set of elements in *X* that are fixed by *g*. Burnside's lemma asserts the following formula for the number of orbits, denoted |*X*/*G*|:[2]

$$|X/G| = \frac{1}{|G|}\sum_{g\in G}|X^g|.$$

Thus the number of orbits (a natural number or +∞) is equal to the average number of points fixed by an element of *G* (which consequently is also a natural number or infinity). If *G* is infinite, the following statement in cardinal arithmetic also holds:

$$|G||X/G| = \sum_{g\in G}|X^g| = \max_{g\in G}|X^g|.$$

Example application

The number of rotationally distinct colourings of the faces of a cube using three colours can be determined from this formula as follows.

Let *X* be the set of 36 possible face colour combinations that can be applied to a cube in one particular orientation, and let the rotation group *G* of the cube act on *X* in the natural manner. Then two elements of *X*belong to the same orbit precisely when one is simply a rotation of the other. The number of rotationally distinct colourings is thus the same as the number of orbits and can be found by counting the sizes of thefixed sets for the 24 elements of *G*.

- one identity element which leaves all 36 elements of *X* unchanged
- six 90-degree face rotations, each of which leaves 33 of the elements of *X* unchanged
- three 180-degree face rotations, each of which leaves 34 of the elements of *X* unchanged
- eight 120-degree vertex rotations, each of which leaves 32 of the elements of *X* unchanged
- six 180-degree edge rotations, each of which leaves 33 of the elements of *X* unchanged

A detailed examination of these automorphisms may be found here.

The average fix size is thus

$$\frac{1}{24}\left(3^6+6\times 3^3+3\times 3^4+8\times 3^2+6\times 3^3\right)=57.$$

Hence there are 57 rotationally distinct colourings of the faces of a cube in three colours. In general, the number of rotationally distinct colorings of the faces of a cube in*n* colors is given by

$$\frac{1}{24}\left(n^6+3n^4+12n^3+8n^2\right)$$

2.12. Expectation

In probability theory and statistics, the expected value (or expectation value, or mathematical expectation, or mean, or first moment) of a random

variable is the integral of the random variable with respect to its probability measure.[1][2]
For discrete random variables this is equivalent to the probability-weighted sum of the possible values.
For continuous random variables with a density function it is the probability density-weighted integral of the possible values.
The term "expected value" can be misleading. It must not be confused with the "most probable value." The expected value is in general not a typical value that the random variable can take on. It is often helpful to interpret the expected value of a random variable as the long-run average value of the variable over many independent repetitions of an experiment.
The expected value may be intuitively understood by the law of large numbers: The expected value, when it exists, is almost surely the limit of the sample mean as sample size grows to infinity. The value may not be expected in the general sense — the "expected value" itself may be unlikely or even impossible (such as having 2.5 children), just like the sample mean.
The expected value does not exist for some distributions with large "tails", such as the Cauchy distribution.[3]
It is possible to construct an expected value equal to the probability of an event by taking the expectation of an indicator function that is one if the event has occurred and zero otherwise. This relationship can be used to translate properties of expected values into properties of probabilities, e.g. using the law of large numbers to justify estimating probabilities byfrequencies.
The expected outcome from one roll of an ordinary (that is, fair) six-sided die is

$$\mathrm{E}(\text{roll with a 6-sided die}) = \left(1\times\frac{1}{6}\right)+\left(2\times\frac{1}{6}\right)+\left(3\times\frac{1}{6}\right)+\left(4\times\frac{1}{6}\right)+\left(5\times\frac{1}{6}\right)+\left(6\times\frac{1}{6}\right) = 3.5$$

which is not among the possible outcomes.[6]
A common application of expected value is gambling. For example, an American roulette wheel has 38 places where the ball may land, all equally likely. A winning bet on a single number pays 35-to-1, meaning that the original stake is not lost, and 35 times that amount is won, so you receive 36 times what you've bet. Considering all 38 possible outcomes, the expected value of the profit resulting from a dollar bet on a single number is the sum of potential net loss times the probability of losing and potential net gain times the probability of winning, that is,

$$\mathrm{E}(\text{winnings from \$1 bet}) = \left(-\$1\times\frac{37}{38}\right)+\left(\$35\times\frac{1}{38}\right) = -\$0.052631579.$$

The net change in your financial holdings is −$1 when you lose, and $35 when you win. Thus one may expect, on average, to lose about five cents for every dollar bet, and the **expected value** of a one-dollar bet is $0.947368421. In gambling, an event of which the expected value equals the stake (i.e. the bettor's expected profit, or net gain, is zero) is called a “fair game”.
Uses and applications

The expected values of the powers of *X* are called the moments of *X*; the moments about the mean of *X* are expected values of powers of $X - \mathrm{E}(X)$. The moments of some random variables can be used to specify their distributions, via their moment generating functions.
To empirically estimate the expected value of a random variable, one repeatedly measures observations of the variable and computes the arithmetic mean of the results. If the expected value exists, this procedure estimates the true expected value in an unbiased manner and has the property of minimizing the sum of the squares of the residuals (the sum of the squared differences between the observations and the estimate). The law of large numbers demonstrates (under fairly mild conditions) that, as the size of the sample gets larger, the variance of this estimate gets smaller.
This property is often exploited in a wide variety of applications, including general problems of statistical estimation and machine learning, to estimate (probabilistic) quantities of interest via Monte Carlo methods, since most quantities of interest can be written in terms of expectation, e.g. $\mathrm{P}(X \in \mathcal{A}) = \mathrm{E}(I_{\mathcal{A}}(X))$ where $I_{\mathcal{A}}(X)$ is the indicator function for set $\mathcal{A}$, i.e. $X \in \mathcal{A} \rightarrow I_{\mathcal{A}}(X) = 1, X \notin \mathcal{A} \rightarrow I_{\mathcal{A}}(X) = 0$.
In classical mechanics, the center of mass is an analogous concept to expectation. For example, suppose *X* is a discrete random variable with values *xi* and corresponding probabilities*pi*. Now consider a weightless rod on which are placed weights, at locations *xi* along the rod and having masses *pi* (whose sum is one). The point at which the rod balances is $\mathrm{E}(X)$.
Expected values can also be used to compute the variance, by means of the computational formula for the variance

$$\mathrm{Var}(X) = \mathrm{E}(X^2) - (\mathrm{E}(X))^2.$$

A very important application of the expectation value is in the field of quantum mechanics. The expectation value of a quantum mechanical operator $\hat{A}$ operating on a quantum statevector $|\psi\rangle$ is written as $\langle\hat{A}\rangle = \langle\psi|A|\psi\rangle$. The uncertainty in $\hat{A}$ can be calculated using the formula $(\Delta A)^2 = \langle\hat{A}^2\rangle - \langle\hat{A}\rangle^2$.
Expectation of matrices
If *X* is an $m \times n$ matrix, then the expected value of the matrix is defined as the matrix of expected values:

$$\mathrm{E}(X) = \mathrm{E}\begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix} = \begin{pmatrix} \mathrm{E}(x_{1,1}) & \mathrm{E}(x_{1,2}) & \cdots & \mathrm{E}(x_{1,n}) \\ \mathrm{E}(x_{2,1}) & \mathrm{E}(x_{2,2}) & \cdots & \mathrm{E}(x_{2,n}) \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{E}(x_{m,1}) & \mathrm{E}(x_{m,2}) & \cdots & \mathrm{E}(x_{m,n}) \end{pmatrix}.$$

This is utilized in covariance matrices.

Formulas for special cases

[edit]Discrete distribution taking only non-negative integer values

When a random variable takes only values in {0,1,2,3,...} we can use the following formula for computing its expectation:

$$\mathrm{E}(X)=\sum_{i=1}^{\infty} P(X\geq i).$$

Proof:

$$\sum_{i=1}^{\infty} P(X\geq i)=\sum_{i=1}^{\infty}\sum_{j=i}^{\infty} P(X=j)$$

interchanging the order of summation, we have

$$\begin{aligned}\sum_{i=1}^{\infty} P(X\geq i)&=\sum_{j=1}^{\infty}\sum_{i=1}^{j} P(X=j)\\ &=\sum_{j=1}^{\infty} j\,P(X=j)\\ &=\mathrm{E}(X)\end{aligned}$$

as claimed. This result can be a useful computational shortcut. For example, suppose we toss a coin where the probability of heads is *p*. How many tosses can we expect until the first heads (not including the heads itself)? Let *X* be this number. Note that we are counting only the tails and not the heads which ends the experiment; in particular, we can have *X* = 0. The expectation of *X* may be computed by $\sum_{i=0}^{\infty}(1-p)^i=\frac{1}{p}$. This is because the number of tosses is at least *i* exactly when the first *i* tosses yielded tails. This matches the expectation of a random variable with an Exponential distribution. We used the formula for Geometric progression: $\sum_{k=1}^{\infty} r^k=\frac{r}{1-r}.$

[edit]Continuous distribution taking non-negative values

Analogously with the discrete case above, when a continuous random variable *X* takes only non-negative values, we can use the following formula for computing its expectation:

$$\mathrm{E}(X)=\int_0^{\infty} P(X\geq x)\;dx$$

Proof: It is first assumed that *X* has a density *fX*(*t*).

$$\int_0^{\infty} P(X\geq x)\;dx=\int_0^{\infty}\int_x^{\infty} f_X(t)\;dt\;dx$$

interchanging the order of integration, we have

$$\begin{aligned}\int_0^\infty P(X \geq x)\,dx &= \int_0^\infty \int_0^t f_X(t)\,dx\,dt \\ &= \int_0^\infty t f_X(t)\,dt \\ &= \mathrm{E}(X)\end{aligned}$$

as claimed. In case no density exists, it is seen that

$$\mathrm{E}(X) = \int_0^\infty \int_0^x dt\,dF(x) = \int_0^\infty \int_t^\infty dF(x)\,dt = \int_0^\infty 1 - F(x)\,dx.$$

2.13. Gauss sum

In mathematics, a Gauss sum or Gaussian sum is a particular kind of finite sum of roots of unity, typically

$$G(\chi) := G(\chi, \psi) = \sum \chi(r) \cdot \psi(r)$$

where the sum is over elements *r* of some finite commutative ring *R*, ψ(*r*) is a group homomorphism of the additive group *R*+ into the unit circle, and χ(*r*) is a group homomorphism of theunit group *R*× into the unit circle, extended to non-unit *r* where it takes the value 0. Gauss sums are the analogues for finite fields of the Gamma function.
Such sums are ubiquitous in number theory. They occur, for example, in the functional equations of Dirichlet L-functions, where for a Dirichlet character χ the equation relating *L*(*s*, χ) and*L*(1 − *s*, χ*) involves a factor

$$G(\chi)\ /\ |G(\chi)|,$$

where χ* is the complex conjugate of χ.
The case originally considered by C. F. Gauss was the quadratic Gauss sum, for *R* the field of residues modulo a prime number *p*, and χ the Legendre symbol. In this case Gauss proved that *G*(χ) = *p*1/2 or *ip*1/2 according as *p* is congruent to 1 or 3 modulo 4.
An alternate form for this Gauss sum is:

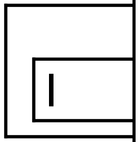
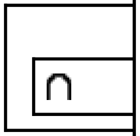
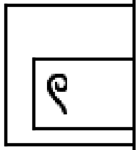
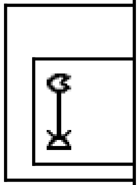
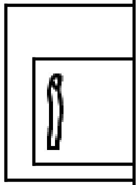
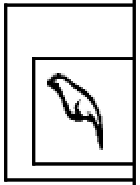
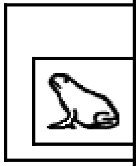
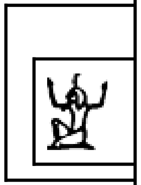
$$\sum e^{\frac{2\pi i r^2}{p}}$$

Quadratic Gauss sums are closely connected with the theory of theta-functions.
The general theory of Gauss sums was developed in the early nineteenth century, with the use of Jacobi sums and their prime decomposition in cyclotomic fields. Sums over the sets where χ takes on a particular value, when the underlying ring is the residue ring modulo an integer *N*, are described by the theory of Gaussian periods.

The absolute value of Gauss sums is usually found as an application of Plancherel's theorem on finite groups. In the case where *R* is a field of *p* elements and χ is nontrivial, the absolute value is *p*1/2. The determination of the exact value of general Gauss sums, following the result of Gauss on the quadratic case, is a long-standing issue. For some cases see Kummer sum.

2.14. Egyptian numbers

The system of Ancient Egyptian numerals was used in Ancient Egypt until the early first millennium AD. It was a decimal system, often rounded off to the higher power, written in hieroglyphs. The hieratic form of numerals stressed an exact finite series notation, ciphered one to one onto the Egyptian alphabet.The Ancient Egyptian system used bases of ten.
[edit]Digits and numbers
the following hieroglyphics were used to denote powers of ten:

Value	1	10	100	1,000	10,000	100,000	1 million, or infinity
Hieroglyph						 or 	
Description	Single stroke	Heel bone	Coil of rope	Water lily (also called Lotus)	Finger	Tadpole or Frog	Man with both hands raised

Multiples of these values were expressed by repeating the symbol as many times as needed.
[edit]Fractions
Main article: Egyptian fraction
Rational numbers could also be expressed, but only as sums of unit fractions, *i.e.*, sums of reciprocals of positive integers, except for 2/3 and 3/4. The hieroglyph indicating a fraction looked like a mouth, which meant "part":

Fractions were written with this fractional solidus, *i.e.*, the numerator 1, and the positive denominator below. Thus, 1/3 was written as:
There were special symbols for 1/2 and for two non-unit fractions, 2/3 (used frequently) and 3/4 (used less frequently)
If the denominator became too large, the "mouth" was just placed over the beginning of the "denominator":
Ancient Egyptian multiplication, one of two multiplication methods used by scribes, was a systematic method for multiplying two numbers that does not require the multiplication table, only the ability to multiply and divide by 2, and to add. Also known as Egyptian multiplication, it decomposes one of the multiplicands (generally the larger) into a sum of powers of two and creates a table of doublings of the second multiplicand. This method may be called mediation and duplation, where mediation means halving one number and duplation means doubling the other number. It is still used in some areas.
The second Egyptian multiplication and division technique was known from the hieratic Moscow and Rhind Mathematical Papyri written in the seventeenth century B.C. by the scribeAhmes.
[edit]The decomposition
The decomposition into a sum of powers of two was not intended as a change from base ten to base two; the Egyptians then were unaware of such concepts and had to resort to much simpler methods. The ancient Egyptians had laid out tables of a great number of powers of two so as not to be obliged to recalculate them each time. The decomposition of a number thus consists of finding the powers of two which make it up. The Egyptians knew empirically that a given power of two would only appear once in a number. For the decomposition, they proceeded methodically; they would initially find the largest power of two less than or equal to the number in question, subtract it out and repeat until nothing remained. (The Egyptians did not make use of the number zero in mathematics).
To find the largest power of 2 keep doubling your answer starting with number 1.
Example:
1 x 2 = 2
2 x 2 = 4
4 x 2 = 8
8 x 2 = 16
16 x 2 = 32
Example of the decomposition of the number 25:

- the largest power of two less than or equal to 25 is 16,
- 25 - 16 = 9,
- the largest power of two less than or equal to 9 is 8,
- 9 - 8 = 1,
- the largest power of two less than or equal to 1 is 1,
- 1 - 1 = 0

25 is thus the sum of the powers of two: 16, 8 and 1.
[edit]The table

After the decomposition of the first multiplicand, it is necessary to construct a table of powers of two times the second multiplicand (generally the smaller) from one up to the largest power of two found during the decomposition. In the table, a line is obtained by multiplying the preceding line by two.
For example, if the largest power of two found during the decomposition is 16, and the second multiplicand is 7, the table is created as follows:

- 1; 7
- 2; 14
- 4; 28
- 8; 56
- 16; 112

[edit]The result
The result is obtained by adding the numbers from the second column for which the corresponding power of two makes up part of the decomposition of the first multiplicand.
The main advantage of this technique is that it makes use of only addition, subtraction, and multiplication by two.
[edit]Example
Here, in actual figures, is how 238 is multiplied by 13. The lines are multiplied by two, from one to the next. A check mark is placed by the powers of two in the decomposition of 13.

✓	1	238					
	2	476					
✓	4	952					
✓	8	1904					
———							
	13	3094					

Since 13 = 8 + 4 + 1, distribution of multiplication over addition gives 13 × 238 = (8 + 4 + 1) × 238 = 8 x 238 + 4 × 238 + 1 × 238 = 1904 + 952 + 238 = 3094.

2.15. De Moivre's theorem

In mathematics, de Moivre's formula, named after Abraham de Moivre, states that for any complex number (and, in particular, for any real number) *x* and integer *n* it holds that

$$(\cos x + i \sin x)^n = \cos(nx) + i \sin(nx)\,.$$

The formula is important because it connects complex numbers (*i* stands for the imaginary unit) and trigonometry. The expression cos *x* + *i* sin *x* is sometimes abbreviated to cis *x*.
By expanding the left hand side and then comparing the real and imaginary parts under the assumption that *x* is real, it is possible to derive useful expressions for cos (*nx*) and sin (*nx*) in terms of cos *x* and sin *x*.
Furthermore, one can use a generalization of this formula to find explicit expressions for the *n*th roots of unity, that is, complex numbers *z* such that *zn* = 1.
Derivation
Although historically proven earlier, de Moivre's formula can easily be derived from Euler's formula

$$e^{ix} = \cos x + i\sin x$$

and the exponential law for integer powers

$$\left(e^{ix}\right)^n = e^{inx}.$$

Then, by Euler's formula,

$$e^{i(nx)} = \cos(nx) + i\sin(nx).$$

Failure for non-integer powers
De Moivre's formula does not in general hold for non-integer powers. Non-integer powers of a complex number can have many different values, see failure of power and logarithm identities. However there is a generalization that the right hand side expression is one possible value of the power.
The derivation of de Moivre's formula above involves a complex number to the power *n*. When the power is not an integer, the result is multiple-valued, for example, when *n* = ½ then:

For *x* = 0 the formula gives 1½ = 1
For *x* = 2*π* the formula gives 1½ = −1

Since the angles 0 and 2π are the same this would give two different values for the same expression. The values 1 and −1 are however both square roots of 1 as the generalization asserts.
No such problem occurs with Euler's formula since there is no identification of different values of its exponent. Euler's formula involves a complex power of a positive real number and this always has a preferred value. The corresponding expressions are:

*ei*0 = 1
*ei*π = −1

Formulas for cosine and sine individually
Being an equality of complex numbers, one necessarily has equality both of the real parts and of the imaginary parts of both members of the equation. If *x*, and therefore also cos *x* and sin *x*, are real numbers, then the identity of these parts can be written (interchanging sides) as

These equations are in fact even valid for complex values of *x*, because both sides are entire (that is, holomorphic on the whole complex plane) functions of *x*, and two such functions that coincide on the real axis necessarily coincide everywhere. Here are the concrete instances of these equations for *n* = 2

$$\cos(nx) = \sum_{k=0}^{\lfloor n/2\rfloor} \binom{n}{2k}(-1)^k(\cos x)^{n-2k}(\sin x)^{2k} = \sum_{k=0}^{\lfloor n/2\rfloor} \binom{n}{2k}(\cos x)^{n-2k}((\cos x)^2-1)^k$$

$$\sin(nx) = \sum_{k=0}^{\lfloor (n-1)/2\rfloor} \binom{n}{2k+1}(-1)^k(\cos x)^{n-2k-1}(\sin x)^{2k+1} = (\sin x)\sum_{k=0}^{\lfloor (n-1)/2\rfloor} \binom{n}{2k+1}(\cos x)^{n-2k-1}((\cos x)^2-1)^k.$$

and *n* = 3:

The right hand side of the formula for cos(*nx*) is in fact the value *Tn*(cos *x*) of the Chebyshev polynomial *Tn* at cos *x*.

[edit]Generalization

$$\cos(2x) = (\cos x)^2 + ((\cos x)^2-1) = 2(\cos x)^2 - 1$$
$$\sin(2x) = 2(\sin x)(\cos x)$$
$$\cos(3x) = (\cos x)^3 + 3\cos x((\cos x)^2-1) = 4(\cos x)^3 - 3\cos x$$
$$\sin(3x) = 3(\cos x)^2(\sin x) - (\sin x)^3 = 3\sin x - 4(\sin x)^3.$$

The formula is actually true in a more general setting than stated above: if *z* and *w* are complex numbers, then

$$(\cos z + i\sin z)^w$$

is a multi-valued function while

$$\cos(wz) + i\sin(wz)$$

is not. Therefore one can state that

$$\cos(wz) + i\sin(wz) \text{ is one value of } (\cos z + i\sin z)^w\,.$$

[edit]Applications

This formula can be used to find the *n*th roots of a complex number. This application does not strictly use de Moivre's formula as the power isn't an integer. However considering the right hand side to the power of *n* will in each case give the same value left hand side.

If *z* is a complex number, written in polar form as

$$z = r\left(\cos x + i\sin x\right),$$

then

$$z^{1/n} = \left[r\left(\cos x + i\sin x\right)\right]^{1/n} = r^{1/n}\left[\cos\left(\frac{x+2k\pi}{n}\right) + i\sin\left(\frac{x+2k\pi}{n}\right)\right]$$

where *k* is an integer. To get the *n* different roots of *z* one only needs to consider values of *k* from 0 to *n* − 1.

2.16. Primes

2.16.1. Primality testing

Primality testing of a number is perhaps the most common problem concerning number theory that topcoders deal with. A prime number is a natural number which has exactly two distinct natural number divisors: 1 and itself. Some basic algorithms and details regarding primality testing and factorization can be found here.
The problem of detecting whether a given number is a prime number has been studied extensively but nonetheless, it turns out that all the deterministic algorithms for this problem are too slow to be used in real life situations and the better ones amongst them are tedious to code. But, there are some probabilistic methods which are very fast and very easy to code. Moreover, the probability of getting a wrong result with these algorithms is so low that it can be neglected in normal situations.
This article discusses some of the popular probabilistic methods such as Fermat's test, Rabin-Miller test, Solovay-Strassen test.

Modular Exponentiation

All the algorithms which we are going to discuss will require you to efficiently compute (ab)%c (where a,b,c are non-negative integers). A straightforward algorithm to do the task can be to iteratively multiply the result with 'a' and take the remainder with 'c' at each step.

```
/* a function to compute (ab)%c */
int modulo(int a,int b,int c){
    // res is kept as long long because intermediate results might overflow in
"int"
    long long res = 1;
    for(int i=0;i<b;i++){
        res *= a;
        res %= c; // this step is valid because (a*b)%c = ((a%c)*(b%c))%c
    }
    return res%c;
}
```

However, as you can clearly see, this algorithm takes O(b) time and is not very useful in practice. We can do it in O(log(b)) by using what is called as exponentiation by squaring. The idea is very simple:

```
  (a2)(b/2)            if b is even and b > 0
ab = a*(a2)((b-1)/2)     if b is odd
   1                   if b = 0
```

This idea can be implemented very easily as shown below:

```
/* This function calculates (ab)%c */
int modulo(int a,int b,int c){
    long long x=1,y=a; // long long is taken to avoid overflow of intermediate
results
    while(b > 0){
        if(b%2 == 1){
            x=(x*y)%c;
        }
        y = (y*y)%c; // squaring the base
        b /= 2;
    }
    return x%c;
}
```

Notice that after i iterations, b becomes b/(2i), and y becomes (y(2i))%c. Multiplying x with y is equivalent to adding 2i to the overall power. We do this if the ith bit from right in the binary representation of b is 1. Let us take an example by computing (7107)%9. If we use the above code, the variables after each iteration of the loop would look like this: (a = 7, c = 9)

```
iterations     b     x     y
    0   107    1     7
    1    53    7     4
    2    26    1     7
    3    13    1     4
    4     6    4     7
    5     3    4     4
    6     1    7     7
    7     0          4     4
```

Now b becomes 0 and the return value of the function is 4. Hence (7107)%9 = 4. The above code could only work for a,b,c in the range of type "int" or the intermediate results will run out of the range of "long long". To write a function for numbers up to 10^18, we need to compute (a*b)%c when computing a*b directly can grow larger than what a long long can handle. We can use a similar idea to do that:

```
   (2*a)*(b/2)              if b is even and b > 0
a*b = a + (2*a)*((b-1)/2)    if b is odd
    0       if b = 0
```

Here is some code which uses the idea described above (you can notice that its the same code as exponentiation, just changing a couple of lines):

```
/* this function calculates (a*b)%c taking into account that a*b might
overflow */
long long mulmod(long long a,long long b,long long c){
    long long x = 0,y=a%c;
    while(b > 0){
        if(b%2 == 1){
            x = (x+y)%c;
        }
        y = (y*2)%c;
        b /= 2;
    }
    return x%c;
}
```
We could replace x=(x*y)%c with x = mulmod(x,y,c) and y = (y*y)%c with y = mulmod(y,y,c) in the original function for calculating (ab)%c. This function requires that 2*c should be in the range of long long. For numbers larger than this, we could write our own BigInt class (java has an inbuilt one) with addition, multiplication and modulus operations and use them.
This method for exponentiation could be further improved by using Montgomery Multiplication. Montgomery Multiplication algorithm is a quick method to compute (a*b)%c, but since it requires some pre-processing, it doesn't help much if you are just going to compute one modular multiplication. But while doing exponentiation, we need to do the pre-processing for 'c' just once, that makes it a better choice if you are expecting very high speed. You can read about it at the links mentioned in the reference section.
Similar technique can be used to compute (ab)%c in O(n3 * log(b)), where a is a square matrix of size n x n. All we need to do in this case is manipulate all the operations as matrix operations. Matrix exponentiation is a very handy tool for your algorithm library and you can see problems involving this every now and then.
Fermat Primality Test
Fermat's Little Theorem
According to Fermat's Little Theorem if p is a prime number and a is a positive integer less than p, then
   ap = a (mod p)
or alternatively:
    a(p-1) = 1 (mod p)
Algorithm of the test
If p is the number which we want to test for primality, then we could randomly choose a, such that a < p and then calculate (a(p-1))%p. If the result is not 1, then by Fermat's Little Theorem p cannot be prime. What if that is not the case? We can choose another a and then do the same test again. We could stop after some number of iterations and if the result is always 1 in each of them, then we can state with very high probability that p is prime. The more iterations we do, the higher is the probability that our result is correct. You can notice that if the method returns composite, then the number is sure to be composite, otherwise it will be probably prime.
Given below is a simple function implementing Fermat's primality test:
```
/* Fermat's test for checking primality, the more iterations the more is
accuracy */
bool Fermat(long long p,int iterations){
    if(p == 1){ // 1 isn't prime
        return false;
    }
    for(int i=0;i<iterations;i++){
        // choose a random integer between 1 and p-1 ( inclusive )
        long long a = rand()%(p-1)+1;
        // modulo is the function we developed above for modular
exponentiation.
        if(modulo(a,p-1,p) != 1){
            return false; /* p is definitely composite */
        }
    }
    return true; /* p is probably prime */
}
```
More iterations of the function will result in higher accuracy, but will take more time. You can choose the number of iterations depending upon the application.
Though Fermat is highly accurate in practice there are certain composite numbers p known as Carmichael numbers for which all values of a<p for which gcd(a,p)=1, (a(p-1))%p = 1. If we apply Fermat's test on a Carmichael number the probability of choosing an a such that gcd(a,p) != 1 is very low (based on the nature of Carmichael numbers), and in that case, the Fermat's test will return a wrong result with very high probability. Although Carmichael numbers are very rare (there are about 250,000 of them less than 1016), but that by no way means that the result you get is always correct. Someone could easily challenge you if you were to use Fermat's test :). Out of the Carmichael numbers less than 1016, about 95% of them are divisible by primes < 1000. This suggests that apart from applying Fermat's test, you may also test the number for divisibility with small prime numbers and this will further reduce the probability of failing. However, there are other improved primality tests which don't have this flaw as Fermat's. We will discuss some of them now.
Miller-Rabin Primality Test
Key Ideas and Concepts

1. Fermat's Little Theorem.
2. If p is prime and x2 = 1 (mod p), then x = +1 or -1 (mod p).
   We could prove this as follows:
3. x2 = 1 (mod p)
   x2 - 1 = 0 (mod p)
   (x-1)(x+1) = 0 (mod p)

Now if p does not divide both (x-1) and (x+1) and it divides their product, then it cannot be a prime, which is a contradiction. Hence, p will either divide (x-1) or it will divide (x+1), so x = +1 or -1 (mod p).
Let us assume that p - 1 = 2d * s where s is odd and d >= 0. If p is prime, then either as = 1 (mod p) as in this case, repeated squaring from as will always yield 1, so (a(p-1))%p will be 1; or a(s*(2r)) = -1 (mod p) for some r such that 0 <= r < d, as repeated squaring from it will always yield 1 and finally a(p-1) = 1 (mod p). If none of these hold true, a(p-1) will not be

1 for any prime number a (otherwise there will be a contradiction with fact #2).

Algorithm

Let p be the given number which we have to test for primality. First we rewrite p-1 as (2d)*s. Now we pick some a in range [1,n-1] and then check whether as = 1 (mod p) or a(s*(2r)) = -1 (mod p). If both of them fail, then p is definitely composite. Otherwise p is probably prime. We can choose another a and repeat the same test. We can stop after some fixed number of iterations and claim that either p is definitely composite, or it is probably prime.

A small procedure realizing the above algorithm is given below:

```
/* Miller-Rabin primality test, iteration signifies the accuracy of the test
*/
bool Miller(long long p,int iteration){
    if(p<2){
        return false;
    }
    if(p!=2 && p%2==0){
        return false;
    }
    long long s=p-1;
    while(s%2==0){
        s/=2;
    }
    for(int i=0;i<iteration;i++){
        long long a=rand()%(p-1)+1,temp=s;
        long long mod=modulo(a,temp,p);
        while(temp!=p-1 && mod!=1 && mod!=p-1){
            mod=mulmod(mod,mod,p);
            temp *= 2;
        }
        if(mod!=p-1 && temp%2==0){
            return false;
        }
    }
    return true;
}
```

It can be shown that for any composite number p, at least (3/4) of the numbers less than p will witness p to be composite when chosen as 'a' in the above test. Which means that if we do 1 iteration, probability that a composite number is returned as prime is (1/4). With k iterations the probability of test failing is (1/4)k or 4(-k). This test is comparatively slower compared to Fermat's test but it doesn't break down for any specific composite numbers and 18-20 iterations is a quite good choice for most applications.

Solovay-Strassen Primality Test

Key Ideas and Concepts

1. Legendre Symbol: This symbol is defined for a pair of integers a and p such that p is prime. It is denoted by (a/p) and calculated as:

2.
   ```
        = 0    if a%p = 0
   (a/p) = 1    if there exists an integer k such that k2 = a ( mod p )
         = -1   otherwise.
   ```
3. It is proved by Euler that:
4. (a/p) = (a((p-1)/2)) % p
5. So we can also say that:
6. (ab/p) = (ab((p-1)/2)) % p = (a((p-1)/2))%p * (b((p-1)/2))%p = (a/p)*(b/p)

7. Jacobian Symbol: This symbol is a generalization of Legendre Symbol as it does not require 'p' to be prime. Let a and n be two positive integers, and n = p1k1 * .. * pnkn, then Jacobian symbol is defined as:
8. (a/n) = ((a/p1)k1) * ((a/p2)k2) * ..... * ((a/pn)kn)

9. So you can see that if n is prime, the Jacobian symbol and Legendre symbol are equal.
    a. There are some properties of these symbols which we can exploit to quickly calculate them:(a/n) = 0 if gcd(a,n) != 1, Hence (0/n) = 0. This is because if gcd(a,n) != 1, then there must be some prime pi such that pi divides both a and n. In that case (a/pi) = 0 [by definition of Legendre Symbol].
    b. (ab/n) = (a/n) * (b/n). It can be easily derived from the fact (ab/p) = (a/p)(b/p) (here (a/p) is the Legendry Symbol).
    c. if a is even, than (a/n) = (2/n)*((a/2)/n). It can be shown that:
    d. ```
            = 1 if n = 1 (mod 8) or n = 7 (mod 8)
       (2/n) = -1 if n = 3 (mod 8) or n = 5 (mod 8)
             = 0 otherwise
       ```
    e. (a/n) = (n/a)*(-1((a-1)(n-1)/4)) if a and n are both odd.

The algorithm for the test is really simple. We can pick up a random a and compute (a/n). If n is a prime then (a/n) should be equal to (a((n-1)/2))%n [ as proved by Euler ]. If they are not equal then n is composite, and we can stop. Otherwise we can choose more random values for a and repeat the test. We can declare n to be probably prime after some iterations.
Note that we are not interested in calculating Jacobi Symbol (a/n) if n is an even integer because we can trivially see that n isn't prime, except 2 of course.
Let us write a little code to compute Jacobian Symbol (a/n) and for the test:
       ```

```
//calculates Jacobian(a/n) n>0 and n is odd
int calculateJacobian(long long a,long long n){
    if(!a) return 0; // (0/n) = 0
    int ans=1;
    long long temp;
    if(a<0){
        a=-a;    // (a/n) = (-a/n)*(-1/n)
        if(n%4==3) ans=-ans; // (-1/n) = -1 if n = 3 ( mod 4 )
    }
    if(a==1) return ans; // (1/n) = 1
    while(a){
        if(a<0){
            a=-a;    // (a/n) = (-a/n)*(-1/n)
            if(n%4==3) ans=-ans;    // (-1/n) = -1 if n = 3 ( mod 4 )
        }
        while(a%2==0){
            a=a/2;    // Property (iii)
            if(n%8==3||n%8==5) ans=-ans;
        }
        swap(a,n);    // Property (iv)
        if(a%4==3 && n%4==3) ans=-ans; // Property (iv)
        a=a%n; // because (a/p) = (a%p / p ) and a%pi = (a%n)%pi if n % pi = 0
        if(a>n/2) a=a-n;
    }
    if(n==1) return ans;
    return 0;
}

/* Iterations determine the accuracy of the test */
bool Solovoy(long long p,int iteration){
    if(p<2) return false;
    if(p!=2 && p%2==0) return false;
    for(int i=0;i<iteration;i++){
        long long a=rand()%(p-1)+1;
        long long jacobian=(p+calculateJacobian(a,p))%p;
        long long mod=modulo(a,(p-1)/2,p);
        if(!jacobian || mod!=jacobian){
            return false;
        }
    }
    return true;
}
```

It is shown that for any composite n, at least half of the a will result in n being declared as composite according to Solovay-Strassen test. This shows that the probability of getting a wrong result after k iterations is (1/2)k.

However, it is generally less preferred than Rabin-Miller test in practice because it gives poorer performance.
The various routines provided in the article can be highly optimized just by using bitwise operators instead of them. For example /= 2 can be replaced by ">>= 1", "%2" can be replaced by "&1" and "*= 2" can be replaced by "<<=1". Inline Assembly can also be used to optimize them further.

2.16.2. Prime factorization

Implementación:

```
public class FactorizacionPrimos
{
	static boolean[] noPrimos = new boolean[20000001];

	public static void criba()
	{
		noPrimos[0] = true;
		noPrimos[1] = true;
		for(int i = 4; i < 20000001; i += 2)
			noPrimos[i] = true;
		for(int i = 3; i < 5000; i++)
		{
			if(!noPrimos[i])
			{
				for(int j = i * i; j < 20000001; j += i)
					noPrimos[j] = true;
			}
		}
	}

	static ArrayList <Long> primos = new ArrayList <Long> ();

	public static void main(String[] args)
	{
		criba();
		for(int i = 0; i < 20000001; i++)
			if(!noPrimos[i])
				primos.add((long) i);
		Scanner sc = new Scanner();
		TreeMap <Long, Long> factores = new TreeMap <Long, Long> ();
		while(true)
		{
			long n = (long) sc.nextDouble();
			if(n == 0)
```

```
                        return;
                  long original = n;
                  n <<= 2;
                  n -= 3;
                  factores.clear();
                  while(n != 1)
                  {
                        boolean cambio = false;
                        int raiz = (int) Math.sqrt(n) + 1;
                        for(long i : primos)
                        {
                              if(i > raiz)
                                    break;
                              if(n % i == 0)
                              {
                                    cambio = true;
                                    if(factores.containsKey(i))
                                          factores.put(i,   factores.get(i)   +
1);
                                    else
                                          factores.put(i, 1L);
                                    n /= i;
                                    break;
                              }
                        }
                        if(!cambio)
                        {
                              if(factores.containsKey(n))
                                    factores.put(n, factores.get(n) + 1);
                              else
                                    factores.put(n, 1L);
                              break;
                        }
                  }
                  int acum = 1;
                  for(long i : factores.values())
                        acum *= (i + 1);
                  System.out.println(original + " " + acum);
            }
      }
}
```

2.17. Bell numbers

In combinatorics, the nth Bell number, named after Eric Temple Bell, is the number of partitions of a set with n members, or equivalently, the number of equivalence relations on it.

Partitions of a set

In general, Bn is the number of partitions of a set of size n. A partition of a set S is defined as a set of nonempty, pairwise disjoint subsets of S whose union is S. For example, B3 = 5 because the 3-element set {a, b, c} can be partitioned in 5 distinct ways:
{ {a}, {b}, {c} }
{ {a}, {b, c} }
{ {b}, {a, c} }
{ {c}, {a, b} }
{ {a, b, c} }.
B0 is 1 because there is exactly one partition of the empty set. Every member of the empty set is a nonempty set (that is vacuously true), and their union is the empty set. Therefore, the empty set is the only partition of itself.
Note that, as suggested by the set notation above, we consider neither the order of the partitions nor the order of elements within each partition. This means the following partitionings are all considered identical:
{ {b}, {a, c} }
{ {a, c}, {b} }
{ {b}, {c, a} }
{ {c, a}, {b} }.

Another view of Bell numbers

Bell numbers can also be viewed as the number of distinct possible ways of putting n distinguishable balls into one or more indistinguishable boxes. For example, let us suppose n is 3. We have three balls, which we will label a, b, and c, and three boxes. If the boxes can not be distinguished from each other, there are five ways of putting the balls in the boxes:
All three balls go in to one box. Since the boxes are anonymous, this is only considered one combination.
a goes in to one box; b and c go in to another box.
b goes in to one box; a and c go in to another box.
c goes in to one box; a and b go in to another box.
Each ball goes in to its own box.

Implementación:

```
public class Bell_Numbers
{

      static double[] dp = new double[1000000];
      static double[] dpB = new double[1000000];
```

```
	static double factorial(int n)
	{
		if(dp[n] != Double.MAX_VALUE)
			return dp[n];
		return dp[n] = n * factorial(n - 1);
	}

	static double combination(int n, int r)
	{
		return factorial(n) / (factorial(r) * factorial(n - r));
	}

	static double bell(int n)
	{
		if(dpB[n] != Double.MAX_VALUE)
			return dpB[n];
		double acum = 0;
		for(int i = 0; i <= n - 1; i++)
			acum += combination(n - 1, i) * bell(i);
		return dpB[n] = acum;
	}
}
```

2.18. Rationals

```
Implementación:

public class Rational implements Comparable <Rational>
{
    static Rational zero = new Rational(0, 1);

    private int num;   // the numerator
    private int den;   // the denominator

    // create and initialize a new Rational object
    public Rational(int numerator, int denominator) {

        // deal with x/0
        //if (denominator == 0) {
        //   throw new RuntimeException("Denominator is zero");
        //}

        // reduce fraction
        int g = gcd(numerator, denominator);
        num = numerator   / g;
        den = denominator / g;

        // only needed for negative numbers
        if (den < 0) { den = -den; num = -num; }
    }

    // return the numerator and denominator of (this)
    public int numerator()   { return num; }
    public int denominator() { return den; }

    // return double precision representation of (this)
    public double toDouble() {
        return (double) num / den;
    }

    // return string representation of (this)
    public String toString() {
        if (den == 1) return num + "";
        else          return num + "/" + den;
    }

    // return { -1, 0, +1 } if a < b, a = b, or a > b
    public int compareTo(Rational b) {
        Rational a = this;
        int lhs = a.num * b.den;
        int rhs = a.den * b.num;
        if (lhs < rhs) return -1;
        if (lhs > rhs) return +1;
        return 0;
    }

    // is this Rational object equal to y?
    public boolean equals(Object y) {
        if (y == null) return false;
        if (y.getClass() != this.getClass()) return false;
        Rational b = (Rational) y;
        return compareTo(b) == 0;
    }

    // hashCode consistent with equals() and compareTo()
    public int hashCode() {
        return this.toString().hashCode();
    }
    // create and return a new rational (r.num + s.num) / (r.den + s.den)
    public static Rational mediant(Rational r, Rational s) {
        return new Rational(r.num + s.num, r.den + s.den);
    }

    // return gcd(|m|, |n|)
    private static int gcd(int m, int n) {
        if (m < 0) m = -m;
        if (n < 0) n = -n;
        if (0 == n) return m;
        else return gcd(n, m % n);
    }

    // return lcm(|m|, |n|)
    public static int lcm(int m, int n) {
        if (m < 0) m = -m;
        if (n < 0) n = -n;
          return m * (n / gcd(m, n));     // parentheses important to avoid
overflow
    }

      // return a * b, staving off overflow as much as possible by cross-
cancellation
    public Rational times(Rational b) {
        Rational a = this;

          // reduce p1/q2 and p2/q1, then multiply, where a = p1/q1 and b =
p2/q2
        Rational c = new Rational(a.num, b.den);
        Rational d = new Rational(b.num, a.den);
        return new Rational(c.num * d.num, c.den * d.den);
    }


    // return a + b, staving off overflow
    public Rational plus(Rational b) {
        Rational a = this;

        // special cases
        if (a.compareTo(zero) == 0) return b;
        if (b.compareTo(zero) == 0) return a;

        // Find gcd of numerators and denominators
        int f = gcd(a.num, b.num);
        int g = gcd(a.den, b.den);

        // add cross-product terms for numerator
```

```
         Rational s = new Rational((a.num / f) * (b.den / g) + (b.num / f) *
(a.den / g),
                                lcm(a.den, b.den));

        // multiply back in
        s.num *= f;
        return s;
    }

    // return -a
    public Rational negate() {
        return new Rational(-num, den);
    }

    // return a - b
    public Rational minus(Rational b) {
        Rational a = this;
        return a.plus(b.negate());
    }


    public Rational reciprocal() { return new Rational(den, num);  }

    // return a / b
    public Rational divides(Rational b) {
        Rational a = this;
        return a.times(b.reciprocal());
    }

      public Rational abs() {
            if(num < 0)
                  return negate();
            else
                  return this;
      }
}
```

2.19. Polynomials

Implementación:

```
#include <complex>
#include <iostream>
#include <cstdio>
#include <vector>
```

```
#include <algorithm>

using namespace std;

typedef complex <double> dcmplx;

#include <complex>
#include <cmath>

#define PREC 0.1
#define ERR 0.1

using namespace std;

class Polynom {
public:
    int grado, max;
    double *coef;

    Polynom(int grado, int max);
    ~Polynom();

    Polynom* operator+= (Polynom *b);
    Polynom* operator-= (Polynom *b);
    Polynom* operator*= (Polynom *b);
    Polynom* operator/= (Polynom *b);
    Polynom* operator%= (Polynom *b);

    complex<double> eval(complex<double> x);
    void derivate();
    void clear();
    void print();
    void simplify();
    void div(Polynom *b, Polynom *cos, Polynom *residuo);
    void copyFrom(Polynom *p);
    void remMultRoot();
    void laguerre(complex<double>* roots);

    bool isZero();
};

void mcd(Polynom *a, Polynom *b, Polynom *res);


Polynom::Polynom(int grado, int max) {
    if (max != 0)   this->coef = new double[max];
    this->max=max;
    clear();
    this->grado=grado;
}

Polynom::~Polynom() {
    delete [] coef;
}

Polynom* Polynom::operator +=(Polynom* b) {
    int i = (b->grado > grado) ? b->grado : grado;
    this->grado=i;
    for (; i>=0; i--) {
        coef[i] += b->coef[i];
    }
    simplify();
    return this;
}

Polynom* Polynom::operator -=(Polynom* b) {
    int i = (b->grado > grado) ? b->grado : grado;
    this->grado=i;
    for (; i>=0; i--) {
        coef[i] -= b->coef[i];
    }
    simplify();
    return this;
}

Polynom* Polynom::operator *=(Polynom* b) {
    double nc[max];
    for (int i=0; i<max; i++) nc[i]=0.0;
    for (int i=0; i<=grado; i++) {
        for (int j=0; j<=b->grado; j++) {
            nc[i+j] += coef[i]*b->coef[j];
        }
    }
    for (int i=0; i<max; i++) coef[i]=nc[i];
    grado+=b->grado;
    simplify();
    return this;
}

Polynom* Polynom::operator /=(Polynom* b) {
    this->div(b, this, NULL);
    return this;
}
```

```
Polynom* Polynom::operator %=(Polynom* b) {
    Polynom* cos = new Polynom(0,max);
    Polynom* res = new Polynom(0,max);
    div(b,cos,res);
    this->copyFrom(res);
    delete cos; delete res;
    return this;
}

complex<double> Polynom::eval(complex<double> x) {
    complex<double> fx;
    for (int i=grado; i>=0; i--) {
        fx *= x;
        fx += coef[i];
    }
    return fx;
}

void Polynom::derivate() {
    for (int i=1; i<=grado; i++) {
        coef[i-1] = coef[i]*i;
    }
    grado--;
}

void Polynom::clear() {
    for (int i=0; i<max; i++) {
        coef[i]=0.0;
    }
    this->grado=0;
}

void Polynom::print() {
    for (int i=0; i<=grado; i++) {
        printf("%fx%i ",coef[i],i);
    }
    printf("\n");
}

void Polynom::simplify() {
    while (grado>0 && coef[grado] == 0) grado--;
}

void Polynom::copyFrom(Polynom* p) {
    this->grado = p->grado;
    for (int i=grado; i>=0; i--) {
        this->coef[i] = p->coef[i];
    }
}

void Polynom::remMultRoot() {
    Polynom *a = this;
    while(true)
    {
        Polynom b(0,max);
        Polynom gcd(0,max);
        b.copyFrom(this);
        b.derivate();
        mcd(a,&b,&gcd);
        if (gcd.grado>0) *a /= &gcd;
        else break;
    }
}

//It doesn't change b or this. Just changes cos and return residuo if not null
void Polynom::div(Polynom* b, Polynom* cos, Polynom* residuo) {
    if (cos != this) cos->copyFrom(this);
    double *res = cos->coef;
    int n=grado, m=b->grado, ng=n-m;
    cos->coef = new double[max];
    clear();
    while (n>=m) {
        double val = res[n]/b->coef[m];
        cos->coef[n-m] = val;
        for (int i=m, j=n; i>=0; i--) {
            res[j--] -= val*b->coef[i];
        }
        n--;
    }
    cos->grado = ng;
    cos->simplify();
    if (residuo == NULL) delete [] res;
    else {
        delete [] residuo->coef;
        residuo->coef = res;
        residuo->max = cos->max;
        residuo->grado = b->grado;
        residuo->simplify();
    }
}

void Polynom::laguerre(complex<double>* roots) {
    Polynom poly(0,max); //the polynom
    Polynom polyp(0,max); //It's derivative
    Polynom polypp(0,max); //2nd order derivative
    poly.copyFrom(this);
    complex<double>* root = roots;
    complex<double> G,H,a,den,fx;
    complex<double> n(grado,0), one(1,0);
    while ( poly.grado>0 ) {
        polyp.copyFrom(&poly);
        polyp.derivate();
        polypp.copyFrom(&polyp);
        polypp.derivate();
        int i = 0;
        do {
            fx = poly.eval(*root);
            if (abs(fx) < ERR) break;
            G = polyp.eval(*root) / fx;
            H = G*G - polypp.eval(*root)/fx;
            den = sqrt( (n-one)*(H*n-G*G) );
            if (abs(G-den) > abs(G+den))
                den = G-den;
            else
                den = G+den;
            a = n / den;
            *root -= a;
        } while (i++ < 50);
        if (root->imag() < ERR) {
            polyp.grado=1;
            polyp.coef[0]=-1*root->real();
            polyp.coef[1]=1;
            poly /= &polyp;
            root++;
        } else {
            polyp.grado=2;
            polyp.coef[0]=norm(*root);
            polyp.coef[1]=-2*root->real();
            polyp.coef[2]=1;
            poly /= &polyp;
            root[1] = conj(*root);
            root+=2;
        }
    }
}

bool Polynom::isZero() {
    return ( this->grado==0 && *(this->coef)==0.0 );
}

void mcd(Polynom* na, Polynom* nb, Polynom *res) {
    Polynom a(0,20); Polynom b(0,20);
    a.copyFrom(na); b.copyFrom(nb);
    Polynom *t = new Polynom(0,a.max);
    Polynom *residuo = new Polynom(0,a.max);
    while (! b.isZero() ) {
        a.div(&b,t,residuo);
        a.copyFrom(&b);
        b.copyFrom(residuo);
        t->clear();
    }
    delete t; delete residuo;
    res->copyFrom(&a);
}

dcmplx Polynom::evaluate(dcmplx x){
    dcmplx res = 0;
    for(int i = 0; i <= grado; i++)
        res += ((dcmplx) coef[i]) * pow(x, i);
    return res;
}

void Polynom::laGuerre(dcmplx *xk, int iteraciones) {
    dcmplx n = grado;
    Polynom *pp = new Polynom(max, max);
    pp->copyFrom(this);
    pp->derivate();
    Polynom *p2 = new Polynom(max, max);
    p2->copyFrom(pp);
    p2->derivate();
    for (int i = 0; i < iteraciones; i++) {
        dcmplx G = pp->evaluate(*xk) / evaluate(*xk);
        dcmplx H = G * G - p2->evaluate(*xk) / evaluate(*xk);
        dcmplx a1 = G + sqrt((n - (dcmplx) 1) * (n * H - G * G));
        dcmplx a2 = G - sqrt((n - (dcmplx) 1) * (n * H - G * G));
        dcmplx a;
        if(abs(a1) > abs(a2))
            a = n / a1;
        else
            a = n / a2;
        *xk = *xk - a;
        if(evaluate(*xk) == dcmplx(0))
            break;
    }
}

void Polynom::roots(dcmplx *roots)
{
    Polynom *temporal = new Polynom(grado, max);
    temporal->copyFrom(this);
    for(int i = 0; i < grado; i++)
    {
        dcmplx *xk = new dcmplx(0, 0);
        temporal->laGuerre(xk, 50);
        if(xk->imag() >= 1e-6)
        {
            Polynom *divisible = new Polynom(2, 3);
              divisible->coef[0] = xk->real() * xk->real() + xk->imag() * xk-
>imag();
            divisible->coef[1] = -2 * xk->real();
            divisible->coef[2] = 1;
            *temporal /= divisible;
            roots[i] = *xk;
            dcmplx *xk1 = new dcmplx(xk->real(), xk->imag());
            delete divisible;
            roots[++i] = *xk1;
        }
        else
        {
            Polynom *divisible = new Polynom(1, 2);
            divisible->coef[0] = -(xk->real());
            divisible->coef[1] = 1;
            *temporal /= divisible;
            delete divisible;
            dcmplx *xk1 = new dcmplx(xk->real(), 0);
            delete xk;
            roots[i] = *xk1;
        }
    }
}
```

3. Strings

3.1. Longest common subsequence

The **longest common subsequence** (LCS) **problem** is to find the longest subsequence common to all sequences in a set of sequences (often just two). Note that subsequence is different from a substring. SeeSubstring vs. subsequence. It is a classic computer science problem, the basis of diff (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.
Code for the dynamic programming solution
Computing the length of the LCS
The function below takes as input sequences X[1..m] and Y[1..n] computes the LCS between X[1..i] and Y[1..j] for all 1 ≤ i ≤ m and 1 ≤ j ≤ n, and stores it in C[i,j].C[m,n] will contain the length of the LCS of X and Y.

```
function LCSLength(X[1..m], Y[1..n])
    C = array(0..m, 0..n)
    for i := 0..m
       C[i,0] = 0
    for j := 0..n
       C[0,j] = 0
    for i := 1..m
        for j := 1..n
            if X[i] = Y[j]
                C[i,j] := C[i-1,j-1] + 1
            else:
                C[i,j] := max(C[i,j-1], C[i-1,j])
    return C[m,n]
```

Alternatively, memoization could be used.
Reading out an LCS
The following function backtraces the choices taken when computing the C table. If the last characters in the prefixes are equal, they must be in an LCS. If not, check what gave the largest LCS of keeping *xi* and*yj*, and make the same choice. Just choose one if they were equally long. Call the function with i=m and j=n.

```
function backTrace(C[0..m,0..n], X[1..m], Y[1..n], i, j)
    if i = 0 or j = 0
        return ""
    else if X[i] = Y[j]
        return backTrace(C, X, Y, i-1, j-1) + X[i]
    else
        if C[i,j-1] > C[i-1,j]
            return backTrace(C, X, Y, i, j-1)
        else
```

```
            return backTrace(C, X, Y, i-1, j)
```

Reading out all LCSs

If choosing *xi* and *yj* would give an equally long result, read out both resulting subsequences. This is returned as a set by this function. Notice that this function is not polynomial, as it might branch in almost every step if the strings are similar.

```
function backTraceAll(C[0..m,0..n], X[1..m], Y[1..n], i, j)
    if i = 0 or j = 0
        return {""}
    else if X[i] = Y[j]
        return {Z + X[i] for all Z in backTraceAll(C, X, Y, i-1, j-1)}
    else
        R := {}
        if C[i,j-1] ≥ C[i-1,j]
            R := backTraceAll(C, X, Y, i, j-1)
        if C[i-1,j] ≥ C[i,j-1]
            R := R U backTraceAll(C, X, Y, i-1, j)
        return R
```

Print the diff

This function will backtrack through the C matrix, and print the diff between the two sequences. Notice that you will get a different answer if you exchange ≥ and <, with > and ≤ below.

```
function printDiff(C[0..m,0..n], X[1..m], Y[1..n], i, j)
    if i > 0 and j > 0 and X[i] = Y[j]
        printDiff(C, X, Y, i-1, j-1)
        print "  " + X[i]
    else
        if j > 0 and (i = 0 or C[i,j-1] ≥ C[i-1,j])
            printDiff(C, X, Y, i, j-1)
            print "+ " + Y[j]
        else if i > 0 and (j = 0 or C[i,j-1] < C[i-1,j])
            printDiff(C, X, Y, i-1, j)
            print "- " + X[i]
        else
            print ""
```

Reduce the required space

If only the length of the LCS is required, the matrix can be reduced to a $2 \times \min(n,m)$ matrix with ease, or to a min(m,n) + 1 vector (smarter) as the dynamic programming approach only needs the current and previous columns of the matrix. Hirschberg's algorithm allows the construction of the optimal sequence itself in the same quadratic time and linear space bounds.[6]

Reduce the comparison time

Most of the time taken by the naive algorithm is spent performing comparisons between items in the sequences. For textual sequences such as source code, you want to view lines as the sequence elements instead of single characters. This can mean comparisons of relatively long strings for each step in the algorithm. Two optimizations can be made that can help to reduce the time these comparisons consume.

3.2. Aho-corasick

The Aho–Corasick string matching algorithm is a string searching algorithm invented by Alfred V. Aho and Margaret J. Corasick. It is a kind of dictionary-matching algorithm that locates elements of a finite set of strings (the "dictionary") within an input text. It matches all patterns simultaneously. The complexity of the algorithm is linear in the length of the patterns plus the length of the searched text plus the number of output matches. Note that because all matches are found, there can be a quadratic number of matches if every substring matches (e.g. dictionary = a, aa, aaa, aaaa and input string is aaaa).
Informally, the algorithm constructs a finite state machine that resembles a trie with additional links between the various internal nodes. These extra internal links allow fast transitions between failed pattern matches (e.g. a search for cat in a trie that does not contain cat, but contains cart, and thus would fail at the node prefixed by ca), to other branches of the trie that share a common prefix (e.g., in the previous case, a branch for attribute might be the best lateral transition). This allows the automaton to transition between pattern matches without the need for backtracking.
When the pattern dictionary is known in advance (e.g. a computer virus database), the construction of the automaton can be performed once off-line and the compiled automaton stored for later use. In this case, its run time is linear in the length of the input plus the number of matched entries.

implementación:

```
int limite = 'a' - 1;

struct Palabra
{
     char *s;
     int tam;

     bool operator<(const Palabra & otra) const
     {
         return tam > otra.tam;
     }
};
```

```
struct PMA;

PMA* guardadas[1000000];
int numeroGuardadas;
int numeroEntregadas;

struct Node
{
    Node *sig;
    int este;
    bool unido;

    Node(int valor)
    {
        este = valor;
        sig = NULL;
        unido = false;
    }

    void borrar()
    {
        if(sig == NULL || unido == true)
            return;
        sig->borrar();
        delete sig;
    }
};


struct List
{
    Node *inicio;
    Node *fin;

    List()
    {
        inicio = NULL;
        fin = NULL;
    }

    void push_back(int nuevo)
    {
        if(inicio == NULL)
        {
            inicio = new Node(nuevo);
            fin = inicio;
            return;
        }
        fin->sig = new Node(nuevo);
        fin = fin->sig;
    }

    void clear()
    {
        if(inicio == NULL)
            return;
        inicio->borrar();
        delete inicio;
        inicio = NULL;
        fin = NULL;
    }

    void merge(List & otra)
    {
        if(otra.inicio == NULL)
        {
            return;
        }
        else if(inicio == NULL)
        {
            inicio = new Node(otra.inicio->este);
            inicio->sig = otra.inicio->sig;
            inicio->unido = true;
            fin = otra.fin;
        }
        else
        {
            fin->sig = otra.inicio;
            fin->unido = true;
            fin = otra.fin;
        }
    }
};
struct PMA
{
    int num;
    int next[27];
    List accept;

    PMA(int n)
    {
        num = n;
        for(int i = 0; i < 27; i++)
```

```
            next[i] = 0;
    }

    void terminar()
    {
       for(int i = 0; i < 27; i++)
           next[i] = 0;
       accept.clear();
       accept.fin = accept.inicio = NULL;
    }

    PMA* dar(char c)
    {
       if(c != 0)
           c -= limite;
       if(next[c] == 0)
           return NULL;
       return guardadas[next[c]];
    }

    void agregar(char c, PMA* a)
    {
       if(c != 0)
           c -= limite;
       next[c] = a->num;
    }
};

 struct AhoCorasick
 {
     PMA *v;

     PMA *darSiguiente()
     {
         if(numeroGuardadas == numeroEntregadas)
         {
             numeroGuardadas++;
             numeroEntregadas++;
             return guardadas[numeroGuardadas - 1] = new PMA(numeroGuardadas
- 1);
         }
         return guardadas[numeroEntregadas++];
     }

     AhoCorasick(Palabra p[], int size) {
       queue<PMA*> Q;
       PMA *root = darSiguiente();
```

```
  for ( int i = 0 ; i < size; ++i) { // make trie
    PMA *t = root;
    for ( int j = 0 ; p[i].s[j]; ++j) {
      char c = p[i].s[j];
      if (t->dar(c) == NULL ) { t->agregar(c, darSiguiente()); }
      t = t->dar(c);
    }
    t->accept.push_back(i);
  }
   // make failure link using bfs
  for ( int c = 'a'; c <= 'z'; ++c) {
    if (root->dar(c)) {
      root->dar(c)->agregar(0, root);
      Q.push(root->dar(c));
    } else root->agregar(c, root);
  }
  while (!Q.empty()) {
    PMA *t = Q.front(); Q.pop();
    for(int i = 'a'; i <= 'z'; i++) {
        if(t->dar(i)){
            if(t->dar(i) == NULL)
                cout << "error";
            Q.push(t->dar(i));
            PMA *r = t->dar(0);
            while (!r->dar(i)) {
                r = r->dar(0);
            }
            t->dar(i)->agregar(0, r->dar(i));
            t->dar(i)->accept.merge(r->dar(i)->accept);
        }
    }
  }
  v = root;
}

void terminar()
{
    for(int i = 1; i < numeroEntregadas; i++)
    {
        guardadas[i]->terminar();
    }
}

int match(char *t, int dp[], int diff, int nuevo) {
  PMA *temp = v;
  PMA *v = temp;
  int n = strlen(t) + diff;
```

```
        for ( int i = 0 ; i < n; ++i) {
          char c = t[i];
          while (!v->dar(c)) v = v->dar(0);
          v = v->dar(c);
          Node *actual = v->accept.inicio;
          while(actual != NULL)
          {
              dp[actual->este] = MAX(dp[actual->este], nuevo);
              actual = actual->sig;
          }
        }
      }
  };

int main()
{
    int dp[10100];
    int n;
    Palabra *palabras = new Palabra[10100];
    char lectura[1010];
    numeroGuardadas = 1;
    while (true)
    {
        numeroEntregadas = 1;
        cin >> n;
        if(n == 0)
            break;
        for (int i = 0; i < n; i++)
        {
            cin >> lectura;
            palabras[i].tam = strlen(lectura);
            palabras[i].s = new char[palabras[i].tam + 1];
            strcpy(palabras[i].s, lectura);
            dp[i] = 1;
        }
        sort(palabras, palabras + n);
        AhoCorasick *ac = new AhoCorasick(palabras, n);
        int mejor = 0;
        for(int i = 0; i < n; i++)
        {
            int dpSig = dp[i] + 1;
            ac->match(palabras[i].s + 1, dp, 0, dpSig);
            ac->match(palabras[i].s, dp, -1, dpSig);
            mejor = MAX(mejor, dp[i]);
        }
        cout << mejor << endl;
        ac->terminar();
        delete ac;
        for(int i = 0; i < n; i++)
        {
            delete [] palabras[i].s;
        }
    }
}


public class Aho_Corasick
{
      static class Nodo
      {
            Nodo siguiente;
            int valor;

            public Nodo(int v)
            {
                  valor = v;
            }
      }

      static class PMA
      {
            PMA next[];
            PMA fail;
            Nodo accept;
            Nodo ultimo;

            PMA()
            {
                  next = new PMA[27];
            }

            void add(int n)
            {
                  if(accept == null)
                        accept = ultimo = new Nodo(n);
                  else
                  {
                        ultimo.siguiente = new Nodo(n);
                        ultimo = ultimo.siguiente;
                  }
            }

            void addAll(PMA other)
            {
```

```
                        if(other.accept == null)
                              return;
                        if(accept == null)
                        {
                              accept = other.accept;
                              ultimo = other.ultimo;
                        }
                        else
                        {
                              ultimo.siguiente = other.accept;
                              ultimo = other.ultimo;
                        }
                  }

                  // función que depende del problema
                  void match(char t[], int inicio, int tam, int dp[], int nuevo)
                  {
                        int n = inicio + tam;
                        PMA v = this;
                        for (int i = inicio; i < n; i++)
                        {
                              int c = t[i] - 'a';
                              while (v.next[c] == null)
                                    v = v.fail;
                              v = v.next[c];
                              Nodo actual = v.accept;
                              while(actual != null)
                              {
                                    dp[actual.valor] = Math.max(dp[actual.valor],
nuevo);
                                    actual = actual.siguiente;
                              }
                        }
                  }
            }

            static PMA buildPMA(char p[][], int size)
            {
                  PMA root = new PMA();
                  for (int i = 0; i < size; i++)
                  {
                        PMA t = root;
                        for (int j = 0; j < p[i].length; j++)
                        {
                              int c = p[i][j] - 'a';
                              if (t.next[c] == null)
                                    t.next[c] = new PMA();
```

```
                t = t.next[c];
            }
            t.add(i);
        }
        for(int c = 0; c <= 25; c++)
            if(root.next[c] == null)
                root.next[c] = root;
        Queue <PMA> Q = new ArrayDeque <PMA> ();
        for(int c = 0; c <= 25; c++)
        {
            if(root.next[c] != root)
            {
                root.next[c].fail = root;
                Q.add(root.next[c]);
            }
        }
        while(!Q.isEmpty())
        {
            PMA t = Q.poll();
            for (int c = 0; c <= 25; c++)
            {
                if (t.next[c] != null)
                {
                    Q.add(t.next[c]);
                    PMA r = t.fail;
                    while (r.next[c] == null)
                        r = r.fail;
                    t.next[c].fail = r.next[c];
                    t.next[c].addAll(r.next[c]);
                }
            }
        }
        return root;
    }

    public static void main(String[] args) throws FileNotFoundException
    {
        int[] dp = new int[10100];
        Scanner sc = new Scanner(System.in);
        String[] palabrasS = new String[10100];
        char[][] palabras = new char[10100][];
        while(true)
        {
            int n = sc.nextInt();
            if(n == 0)
                break;
            for(int i = 0; i < n; i++)
```

```
                {
                      palabrasS[i] = sc.next();
                      dp[i] = 1;
                }
                Arrays.sort(palabrasS, 0, n, new Comparator<String>()
                            {
                                  public int compare(String o1, String o2)
                                  {
                                        return o2.length() - o1.length();
                                  }
                            });
                for(int i = 0; i < n; i++)
                      palabras[i] = palabrasS[i].toCharArray();
                PMA raiz = buildPMA(palabras, n);
                int mejor = 0;
                for(int i = 0; i < n; i++)
                {
                      int dpSig = dp[i] + 1;
                      raiz.match(palabras[i], 1, palabras[i].length - 1,
dp, dpSig);
                      raiz.match(palabras[i], 0, palabras[i].length - 1,
dp, dpSig);
                      mejor = Math.max(mejor, dp[i]);
                }
                System.out.println(mejor);
                if(n > 100)
                      System.gc();
          }
    }
}
```

3.3. KMP

The **Knuth-Morris-Pratt** string searching algorithm (or **KMP algorithm**) searches for occurrences of a "word" W within a main "text string" S by employing the observation that when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters.

```
    static int kmp(char t[], char p[], int fail[])
    {
          int n = t.length;
          int m = p.length;
          for (int i = 0, k = 0; i < n; i++)
          {
            while (k >= 0 && p[k] != t[i])
                k = fail[k];
            if (++k >= m)
                return 1;
        }
        return 0;
    }

    static int[] buildFail(char p[])
    {
        int m = p.length;
        int fail[] = new int[m + 1];
        int j = fail[0] = -1;
        for (int i = 1; i <= m; i++)
        {
            while (j >= 0 && p[j] != p[i - 1])
                j = fail[j];
            fail[i] = ++j;
        }
        return fail;
    }
```

3.4. Suffix array

In computer science, a **suffix array** is an array of integers giving the starting positions of suffixes of a string in lexicographical order.

We tried to gather as many problems as possible that can be solved using the suffix arrays. Going through all the problems at the first reading would seem rather difficult for a reader who had the first contact with suffix arrays by reading this paper. To make the lecture easier, the problems are arranged in an increasing difficulty order.

Task 1: hidden password (ACM 2003, abridged)

Consider a string of length n (1 <= n <= 100000). Determine its minimum lexicographic rotation.

For example, the rotations of the string “alabala” are:
alabala
labalaa
abalaal
balaala
alaalab
laalaba
aalabal

and the smallest among them is “aalabal”.

Solution:

Usually, when having to handle problems that involve string rotations, one would rather concatenate the string with itself in order to simplify the task. After, the minimal sequence of length n is requested. As their order is determined by the order of the string’s suffixes – although there is a linear solution presented in [10]- suffix arrays are one easy gimmick that can solve the problem instantly.

Problem 2: array (training camp 2004)

Consider an array c1c2...cn consisting of n(1 ≤ n ≤ 30 000)) elements from the set {A, B}. Concatenate the array with itself and obtain an array of length 2n. For an index k (1≤k≤2n) consider the subsequences of length at most n that end on position k, and among these let s(k) be the smallest lexicographic subsequence. Determine the index k for which s(k) is longest. Hint: Let X and Y bet two arrays as defined previously and « o » the concatenation operator. In this problem you will consider that X > X o Y.

Solution:

The searched subsequence is the smallest lexicographic rotation of the given array. Denote by Sik the substring of length k that begins on position i. Let Sin be the smallest substring of length n in the string obtained by concatenation. Supposing by absurd that s(i + n - 1) < n would mean that there is an i’ (i < i’ ≤ j) so that Si’ j – i’ + 1 is smaller than Sin. From the condition in the problem’s text, we have Si’j-i’+1 > Si’n; but Si’n > Sin => contradiction.
Although there is an O(n) algorithm that would easily solve this, the method used during the contest by one of the authors(and that gained a maximum score) used suffix arrays, as in the previous task.

Problem 3: substr (training camp 2003)

You are given a text consisting of N characters (big and small letters of the English alphabet and digits). A substring of this text is a subsequence of characters that appear on consecutive positions in the text. Given an integer K, find the length of the longest substring that appears in the text at least K times (1 ≤ N ≤ 16384).

Solution:

Having the text’s suffixes sorted, iterate with a variable i from 0 to N – K and compute the longest common prefix of suffix i and suffix i + K – 1. The biggest prefix found during this operation represents the problem’s solution.

Problem 4: guess (training camp 2003)

You and the Peasant play a totally uninteresting game. You have a large string and the Peasant asks you questions like ”does the following string is a substring of yours?” The Peasant asks you many questions and you have to answer as quick s you can. Because you are a programmer, you think that it would be better to know all the substrings that appear in your string. But before doing all this work, you are wondering how many distinct substrings are in your string (1 ≤ your string’s length ≤ 10 000)

Solution:

This is actually asking to compute the number of nodes (without root) of a string’s corresponding suffix trie. Each distinct sequence of the string is determined by the unique path traversed in the suffix trie when searching it. As in the example above, „abac” has the substrings „a”, „ab”, „aba”, „abac”, „ac”, „b”, „ba”, „bac” and „c”, determined by the path starting from the root and going toward nodes 2, 3, 4, 5, 6, 7, 8 and 9 in this order. Since building the suffix trie is not always a pleasant job and has a quadratic complexity, an approach using suffix arrays would be much more useful. Get the sorted array of suffixes in O(n lg n), then search the first position where the matching between every pair of consecutive suffixes fails (using the lcp function), and add the number of remaining characters to the solution.

Problem 5: seti (ONI 2002 – abridged)

Given a string of length N (1≤N≤131072) and M strings of length at most 64, count the number of matchings of every small string in the big one.

Solution:

Do as in the classical suffix arrays algorithm, only that it is sufficient to stop after step 6, where an order relation between all the strings of length 26 = 64 was established. Having sorted the substrings of length 64, each query is solved by two binary searches. The overall complexity is O(N lg 64 + M * 64 * lg N) = O(N + M lg N).

Problem 6: common subsequence (Polish Olympiad 2001 and Top Coder 2004 - abridged)

There are given three strings S1 , S2 şi S3, of lengths m, n and p (1 <= m, n, p <= 10000). Determine their longest common substring. For example, if S1

= abababca S2 = aababc and S3= aaababca, their longest common substring would be ababc.

Solution:

If the strings were smaller in length, the problem could have been easily solved using dinamic programming, leading to a O(n2) complexity.
Another idea is to take each suffix of S1 and try finding its maximum matching in the other two strings. A naive maximum matching algorithm gives an O(n^2) complexity, but using KMP [8], we can achieve this in O(n), and using this method for each of S1's suffixes we would gain an O(n^2) solution.
Let's see what happens if we sort the suffixes of the three strings:

```
a$
abababca$
ababca$
abca$
bababca$
babca$
bca$
ca$
aababc#
ababc#
abc#
babc#
bc#
c#
a@
aaababca@
aababca@
ababca@
abca@
babca@
bca@
ca@
```

Now we merge them (consider $ < # < @ < a ...):

```
a$
a@
aaababca@
10
aababc#
aababca@
abababca$
ababc#
ababca$
ababca@
abc#
abca$
```

```
abca@
bababca$
babc#
babca$
babca@
bc#
bca$
bca@
c#
ca$
ca@
```

The maximal common substring corresponds to the longest common prefix of the three suffixes ababca$, ababc# and ababca@. Let's see where they appear in the sorted array:

```
a$
a@
aaababca@
aababc#
aababca@
abababca$
ababc#
ababca$
ababca@
abc#
abca$
abca@
bababca$
babc#
babca$
babca@
bc#
bca$
bca@
c#
ca$
ca@
```

This is where we can figure out that the solution is a sequence i..j in the array of sorted suffixes, with the property that it contains at least one suffix from every string, and the longest common prefix of the first and last suffix in the suffix is maximum, giving the solution for this problem. Other common substrings of the three strings would be common prefixes for some substring in the suffix array, e.g. bab for bababca$ babc# babca$, or a for a$ a@ aaababca@ aababc#. To find the sequence with the longest common prefix, go with two indexes, START and END, over the suffixes, where START goes from

1 to the number of suffixes, and END is the smallest index greater than START so that between START and END there are suffixes from all the three strings. In this way, the pair [START, END] will hit the optimal sequence [i..j]. This algorithm is linear because START takes N values while END is incremented at most N times.

In order to sort the array containing all the suffixes, it is not necessary to sort the suffixes of every string in part and then merge them, as this would increase the complexity if implemented without any smart gimmicks. We can concatenate the three strings into a single one (abababca$aababc@aaababca# for the example above) and then sort its suffixes.

Problem 7: the longest palindrome (USACO training gate)

Given a strings S of length n (n <= 20000) determine its longest substring that also is a palindrome.

Solution:

For a fixed i, computing the longest palindrome that is centered in i requires the longest common prefix of the substring S[i..n] and the reversed S[1..i]. Merge the sorted suffixes of string S and the reversed string S’, then query the longest common prefix for S[i] and S’[n – i + 1] (S’[n – i + 1] = S[1..i]).

Since this is done in O(lg n), the overall complexity is O(n lg n). The case where the searched palindromes have an even length is treated in an analogous way.

Problem 8: template (Polish Olympiad of Informatics 2004, abridged)

For a given string A, find the smallest length of a string B so that A can be obtained by sticking several B’s together (when sticking them, they can overlap, but they have to match).

Example: ababbababbabababbabababbababbaba

Result: 8

The minimum length B is “ababbaba”
A can be obtained from B this way:
ababbababbabababbabababbababbaba
ababbaba
ababbaba
ababbaba
ababbaba
ababbaba

Solution:

The simplest solution uses suffix arrays a balanced tree and a max-heap. It's obvious that the searched pattern is a prefix of A. Having sorted the suffixes of A, we shall add to B, step by step, one more ending character. At each step we keep to pointers L and R, representing the first and the last suffix in the array that have B as a prefix. The balanced tree will hold permanently the starting positions of the suffixes between L and R, and the heap will keep the distances between consecutive elements of the tree. When inserting a new character into B, by two binary searches we get the new L' and R', with L ≤ L' and R' ≤ R. We must also update the tree and the heap. Introduce characters into B while the biggest(first) element in the heap is smaller or equal to the length of B. B's final length offers the searched result. The final complexity is O(n lg n), where n is the length of A.

Problem 9: (Baltic Olympiad of Informatics 2004)

A string s is called an (K,L)-repeat if S is obtained by concatenating K≥1 times some seed string T with length L≥1. For example, the string S = abaabaabaaba is a (4,3)-repeat with T = aba as its seed string. That is, the seed string T is 3 characters long, and the whole string S is obtained by repeating T 4 times.
Write a program for the following task: Your program is given a long string U consisting of characters 'a' and/or 'b' as input. Your program must find some (K,L)-repeat that occurs as substring within U with K as large as possible.
For example, the input string
U = babbabaabaabaabab
contains the underlined (4,3)-repeat S starting at position 5. Since U contains no other contiguous substring with more than 4 repeats, your program must output this underlined substring.

Solution:

We want to find out for a fixed L how to get the greatest K so that in the string U there will be a substring S which is a (K, L) - repeat. Check this example: U = babaabaabaabaaab L = 3 and a fixed substring X = aab that begins on position 4 of U. We can try to extend the sequence aab by repeating it from its both ends as much as possible, as can be seen below:
b a b a a b a a b a a b a a a b
a a b a a b
a b a a b a a b a a b a a b a
Extending the prefix of length L this way to the left, then to the right (in our example the prefix of length 3) of the obtained sequence, we get the longest repetition of a string of length L satisfying the property that the repetition contains X as a substring the (in the case where the repetition is (1, L) this is not true, but it's a trivial case).

Now we see that in order to identify within U all the (K, L) repetitions with a fixed L, it is sufficient to partition the string in n/L chunks and then extend them. It will not be possible to do this in O(1) for every chunk, thus the final algorithm will have a final complexity of O(n/1 + n/2 + n/3 + .. + n/n) (every chunk can be repeated partially or totally only at left or right, and we will not extend every chunk separately, but we will merge the adjacent chunks into a single one; if we had p consecutive chunks of same length, their maximum extensions would be found in O(p)). But we know that the sum 1 + 1⁄2 + 1/3 + 1⁄4 + ... + 1/n – ln n is convergent toward a known constant c, known as Euler's constant, and c < 1, so we can easily figure out that O(1 + 1⁄2 + ... + 1/n) = O(ln n) so the algorithm would have an O(n lg n) complexity if the extensions would have been computed easily. Now we can use the suffix trees. To find out how much the sequence U[i..j] can be extended to the right, we need to find the longest common prefix of U[i..j] and U[j + 1..n]. To extend it to the left, it's sufficient to reverse U, that would lead to the same problem. We have seen how to compute the longest common prefix in O(1) using the suffix array, that is built in O(n lg n) time, then do the required RMQ
pre-calculation in O(n lg n) that allows the lcp queries to be answered in O(1). The final complexity is O(n lg n).

Problem 10: (ACM SEER 2004)

Given a string S, determine for each of its prefixes if it's a periodic string. Hence, for every i (2 <= i <= N) we are interested in the greatest K > 1 (if there exists such one) satisfying the property that S's prefix of length i can be also written as Ak, or A concatenated with itself k times, for some string A. We are also interested which is that k. (0 <= N <= 1000000)

Example: aabaabaabaab

Result:
```
22
62
93
12 4
```

Explanation: prefix aa has the period a; prefix aabaab has the period aab; prefix aabaabaab has the period aab; prefix aabaabaabaab has the period aab.

Solution:

See what happens when trying to match a string with one of its suffixes. Take a string and break it in two parts, getting a prefix and a suffix
```
S   = aab aabaabaaaab
suf = aab aabaaaab
pre = aab
```

If the suffix matches some number of characters of the initial string which is >= |pre|, it means that pre is also a prefix of the suffix and we can also break the suffix in prefix and suffix1, and the string can be broken in prefix, prefix and suffix1. If the string matches some arbitrary number of characters from the string >= 2|pre| then the suffix matches suffix1 on a number of characters >= |pre| thus suffix1 can be written as prefix and suffix2, then suffix can be written as prefix prefix suffix2, so S can be written as prefix prefix prefix sufix2.

S = aab aab aab aaaab
suf = aab aab aaaab
suf1 = aab aaaab
pre = aab

Observe that if S matches at least k * |prefix| characters of its suffix, then S has a prefix of length (k + 1) * |prefix|, which is periodic.
Using the suffix array, we can find out for each suffix its maximum matching with the initial string. If the ith suffix matches the first k * (i - 1) positions then we can update the information that says that the prefixes of length j * (i - 1) (where 2 <= j <= k) are periodic. For every suffix Si the update has a maximum complexity of O(n/(i - 1)). Thus the algorithm gets an overall complexity of O(n log n).

There is a similar solution using the KMP algorithm, that can be done in O(n), but it doesn’t match this paper’s purpose.

Implementación:

```
#define MAX(X,Y) ((X) > (Y) ? (X) : (Y))
#define REP(i, n) for(int i = 0; i < n; i++)

using namespace std;

struct Letra
{
    char letra;
    char veces;
};

struct SAString
{
    char *s;
    int *letras;
    Letra *letrasTiene;
    int tam;
    int dp;
```

```
SAString()
{
    s = new char[1010];
    letras = new int[26];
    letrasTiene = new Letra[26];
}

void llenarLetras()
{
    for(int i = 0; i < 26; i++)
    {
        letras[i] = 0;
        letrasTiene[i].veces = 0;
    }
    for(int i = 0; i < tam; i++)
    {
        letras[s[i] - 'a']++;
    }
    int cuenta = 0;
    for(int i = 0; i < 26; i++)
    {
        if(letras[i] > 0)
        {
            letrasTiene[cuenta].letra = i;
            letrasTiene[cuenta++].veces = letras[i];
        }
    }
    s[tam] = '$';
    s[++tam] = '\0';
}

bool chequearSubS(SAString *otro)
{
    for(int i = 0; i < 26; i++)
    {
        if(letrasTiene[i].veces == 0)
            break;
        if(letrasTiene[i].veces > otro->letras[letrasTiene[i].letra])
            return false;
    }
    return true;
}

SAString(int tamMaximo)
{
    s = new char[tamMaximo + 1];
```

```
        tam = 0;
        s[0] = '\0';
    }

    void strcat(char *actual, int numero)
    {
        while(*actual != '\0')
        {
            s[tam++] = *actual;
            actual++;
        }
    }

    bool operator<(const SAString &otro) const
    {
        return tam > otro.tam;
    }
};

struct SAComp {
  const int h, *g;
  SAComp(int h, int* g) : h(h), g(g) {}
  bool operator() (int a, int b) {
    return a == b ? false : g[a] != g[b] ? g[a] < g[b] : g[a+h] < g[b+h];
  }
};

inline bool leq(int a1, int a2,   int b1, int b2) { // lexic. order for pairs
  return(a1 < b1 || a1 == b1 && a2 <= b2);
}                                                   // and triples
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));
}

int c[10000];
// stably sort a[0..n-1] to b[0..n-1] with keys in 0..K from r
static void radixPass(int* a, int* b, int* r, int n, int K)
{ // count occurrences                      // counter array
  for (int i = 0;  i <= K;  i++) c[i] = 0;          // reset counters
  for (int i = 0;  i < n;  i++) c[r[a[i]]]++;    // count occurences
  for (int i = 0, sum = 0;  i <= K;  i++) { // exclusive prefix sums
     int t = c[i];  c[i] = sum;  sum += t;
  }
  for (int i = 0;  i < n;  i++) b[c[r[a[i]]]++] = a[i];      // sort
}

// find the suffix array SA of s[0..n-1] in {1..K}^n
// require s[n]=s[n+1]=s[n+2]=0, n>=2
void suffixArray(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];
  int* SA0  = new int[n0];

  // generate positions of mod 1 and mod  2 suffixes
  // the "+(n0-n1)" adds a dummy mod 1 suffix if n%3 == 1
  for (int i=0, j=0;  i < n+(n0-n1);  i++) if (i%3 != 0) s12[j++] = i;

  // lsb radix sort the mod 1 and mod 2 triples
  radixPass(s12 , SA12, s+2, n02, K);
  radixPass(SA12, s12 , s+1, n02, K);
  radixPass(s12 , SA12, s  , n02, K);

  // find lexicographic names of triples
  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; } // left half
    else                  { s12[SA12[i]/3 + n0] = name; } // right half
  }

  // recurse if names are not yet unique
  if (name < n02) {
    suffixArray(s12, SA12, n02, name);
    // store unique names in s12 using the suffix array
    for (int i = 0;  i < n02;  i++) s12[SA12[i]] = i + 1;
  } else // generate the suffix array of s12 directly
    for (int i = 0;  i < n02;  i++) SA12[s12[i] - 1] = i;

  // stably sort the mod 0 suffixes from SA12 by their first character
  for (int i=0, j=0;  i < n02;  i++) if (SA12[i] < n0) s0[j++] = 3*SA12[i];
  radixPass(s0, SA0, s, n0, K);

  // merge sorted SA0 suffixes and sorted SA12 suffixes
  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(); // pos of current offset 12 suffix
    int j = SA0[p]; // pos of current offset 0  suffix
    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]))
    { // suffix from SA12 is smaller
      SA[k] = i;  t++;
      if (t == n02) { // done --- only SA0 suffixes left
        for (k++;  p < n0;  p++, k++) SA[k] = SA0[p];
      }
    } else {
      SA[k] = j;  p++;
      if (p == n0)  { // done --- only SA12 suffixes left
        for (k++;  t < n02;  t++, k++) SA[k] = GetI();
      }
    }
  }
  delete [] s12; delete [] SA12; delete [] SA0; delete [] s0;
}

// ojo, las funciones de esta clase esperan que todos los SAStrings tengan un
$
// al final
struct SuffixArray
{
    int *g, *b, *sa, *lcp, *rmq, n, *aux;
    char *t;

    SuffixArray(int tamMax)
    {
        g = new int[tamMax];
        b = new int[tamMax];
        sa = new int[tamMax];
        lcp = new int[tamMax];
        aux = new int[tamMax + 3];
        int logn = 1;
        int n = tamMax;
        for (int k = 1; k < n; k *= 2) ++logn;
        rmq = new int[n * logn];
    }

    void buildSA(SAString *sas) {
      n = sas->tam;
      t = sas->s;
      for(int i = 0; i < n; i++)
      {
          aux[i] = t[i];
      }
      aux[n] = aux[n + 1] = aux[n + 2] = 0;
      suffixArray(aux, sa, n, 256); // Construccion en O(n) usa new y delete
      n--;
// Construccion del SA en O(n * log n * log n)
//      REP(i,n+1) sa[i] = i, g[i] = t[i];
//      b[0] = 0; b[n] = 0;
//      sort(sa, sa+n+1, SAComp(0, g));
//      for(int h = 1; b[n] != n ; h *= 2) {
//        SAComp comp(h, g);
//        sort(sa, sa+n+1, comp);
//        REP(i, n) b[i+1] = b[i] + comp(sa[i], sa[i+1]);
//        REP(i, n+1) g[sa[i]] = b[i];
//      }
//      cout << t << endl;
//      for(int i = 0; i < n; i++)
//      {
//          cout << t + sa[i] <<  " jeje" << endl;
//      }
      buildLCP();
      buildRMQ();
    }

    // Naive matching O(m log n)
    int findN(SAString *o) {
      char *p = o->s;
      int m = o->tam - 1;
      int temp = n;
      int n = temp + 1;
      int a = 0, b = n;
      while (a < b) {
        int c = (a + b) / 2;
        if (strncmp(t+sa[c], p, m) < 0) a = c+1; else b = c;
      }
      return strncmp(t+sa[a], p, m) == 0 ? sa[a] : -1;
    }

    // Kasai-Lee-Arimura-Arikawa-Park's simple LCP computation: O(n)
    void buildLCP() {
      int *a = sa;
      int h = 0;
      REP(i, n+1) b[a[i]] = i;
      REP(i, n+1) {
        if (b[i]){
          for (int j = a[b[i]-1]; j+h<n && i+h<n && t[j+h] == t[i+h]; ++h);
          lcp[b[i]] = h;
        } else lcp[b[i]] = -1;
        if (h > 0) --h;
      }
    }

    // call RMQ = buildRMQ(lcp, n+1)
```

```
void buildRMQ() {
  int temp = n;
  int n = temp + 1;
  int *b = rmq; copy(lcp, lcp+n, b);
  for (int k = 1; k < n; k *= 2) {
    copy(b, b+n, b+n); b += n;
    REP(i, n-k) b[i] = min(b[i], b[i+k]);
  }
}

// inner LCP computation with RMQ: O(1)
int minimum(int x, int y) {
  int temp = n;
  int n = temp + 1;
  int z = y - x, k = 0, e = 1, s;
  s = ( (z & 0xffff0000) != 0 ) << 4; z >>= s; e <<= s; k |= s;
  s = ( (z & 0x0000ff00) != 0 ) << 3; z >>= s; e <<= s; k |= s;
  s = ( (z & 0x000000f0) != 0 ) << 2; z >>= s; e <<= s; k |= s;
  s = ( (z & 0x0000000c) != 0 ) << 1; z >>= s; e <<= s; k |= s;
  s = ( (z & 0x00000002) != 0 ) << 0; z >>= s; e <<= s; k |= s;
  return min( rmq[x+n*k], rmq[y+n*k-e+1] );
}

// outer LCP computation: O(m - o)
int computeLCP(char *t, int n, char *p, int m, int o, int k) {
  int i = o;
  for (; i < m && k+i < n && p[i] == t[k+i]; ++i);
  return i;
}

// Mamber-Myers's O(m + log n) string matching with LCP/RMQ
#define COMP(h, k) (h == m || (k+h<n && p[h]<t[k+h]))
int find(SAString *o) {
  char *p = o->s;
  int m = o->tam - 1;
  int l = 0, lh = 0, r = n, rh = computeLCP(t,n+1,p,m,0,sa[n]);
  if (!COMP(rh, sa[r])) return -1;
  for (int k = (l+r)/2; l+1 < r; k = (l+r)/2) {
    int A = minimum(l+1, k), B = minimum(k+1, r);
    if (A >= B) {
      if (lh < A) l = k;
      else if (lh > A) r = k, rh = A;
      else {
        int i = computeLCP(t, n+1, p, m, A, sa[k]);
        if (COMP(i, sa[k])) r = k, rh = i; else l = k, lh = i;
      }
    } else {
          if (rh < B) r = k;
          else if (rh > B) l = k, lh = B;
          else {
            int i = computeLCP(t, n+1, p, m, B, sa[k]);
            if (COMP(i, sa[k])) r = k, rh = i; else l = k, lh = i;
          }
        }
      }
      return rh == m ? r : -1;
    }
};

bool comparar(SAString *a, SAString *b)
{
    return a->tam > b->tam;
}

SAString *palabras[10010];

int main()
{
    int n;
    SuffixArray sa(1200);
    for(int i = 0; i < 10010; i++)
    {
        palabras[i] = new SAString();
    }
    while (true)
    {
        cin >> n;
        if(n == 0)
            break;
        for (int i = 0; i < n; i++)
        {
            cin >> palabras[i]->s;
            palabras[i]->tam = strlen(palabras[i]->s);
            palabras[i]->dp = 1;
            palabras[i]->llenarLetras();
        }
        sort(palabras, palabras + n, comparar);
        for(int i = 0; i < n; i++)
        {
            SAString *actual = palabras[i];
            int dpSig = actual->dp + 1;
            int tamActual = actual->tam;
            sa.buildSA(actual);
            for(int j = i + 1; j < n; j++)
            {
                SAString *posible = palabras[j];
                if(dpSig < posible->dp || tamActual <= posible->tam)
                    continue;
                if(posible->chequearSubS(actual) && sa.find(posible) != -1)
                    posible->dp = dpSig;
            }
        }
        int mejor = 0;
        for(int i = 0; i < n; i++)
        {
            mejor = MAX(mejor, palabras[i]->dp);
        }
        cout << mejor << endl;
    }
}

public class Suffix_Array
{
      static boolean leq(int a1, int a2,   int b1, int b2)
      {
            return(a1 < b1 || a1 == b1 && a2 <= b2);
      }

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

      static int[] c;

      static void radixPass(int[] a, int[] b, int[] r, int n, int K, int
delta)
      {
            if(c == null || c.length != K + 1) 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] + delta]]++;
            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] + delta]]++] = a[i];
      }

      // find the suffix array SA of s[0..n-1] in {1..K}^n
      // require s[n]=s[n+1]=s[n+2]=0, n>=2
      static void suffixArray(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];
            int[] SA0  = new int[n0];
            for (int i=0, j=0; i < n + (n0 - n1); i++)
            if (i % 3 != 0)
                  s12[j++] = i;
            radixPass(s12, SA12, s, n02, K, 2);
            radixPass(SA12, s12 , s, n02, K, 1);
            radixPass(s12, SA12, s, n02, K, 0);
            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)
            {
                  suffixArray(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];
            radixPass(s0, SA0, s, n0, K, 0);
            for(int p = 0,  t = n0 - n1,  k = 0;  k < n;  k++)
            {
                  int i = (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) *
3 + 2);
                  int 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;
                      t++;
                      if (t == n02)
                         for(k++; p < n0; p++, k++)
                             SA[k] = SA0[p];
                }
                else
                {
                      SA[k] = j;
                      p++;
                      if(p == n0)
                      for(k++; t < n02; t++, k++)
                          SA[k] = (SA12[t] < n0 ? SA12[t] * 3 + 1 :
(SA12[t] - n0) * 3 + 2);
                }
          }
    }

    static class SuffixArray
    {
          int n;
          int[] g, b, sa, lcp, rmq, aux;
          char[] t;

          SuffixArray(String entrada)
          {
                int tamMax = entrada.length();
                g = new int[tamMax + 1];
                b = new int[tamMax + 1];
                sa = new int[tamMax];
                lcp = new int[tamMax];
                aux = new int[tamMax + 3];
                int logn = 1;
                int n = tamMax;
                for (int k = 1; k < n; k *= 2) ++logn;
                rmq = new int[n * logn];
                buildSA(entrada);
          }

          void buildSA(String sas)
          {
                n = sas.length();
                t = sas.toCharArray();
                for(int i = 0; i < n; i++)
                      aux[i] = t[i];
                aux[n] = aux[n + 1] = aux[n + 2] = 0;
```

```
            suffixArray(aux, sa, n, 230);
            buildLCP();
            buildRMQ();
      }

      class SAComp implements Comparator <Integer>
      {
            int h;
            int[] g;

            SAComp(int hh, int[] gg)
            {
                  h = hh;
                  g = gg;
            }

            public int compare(Integer aa, Integer bb)
            {
                  int a = aa;
                  int b = bb;
                  if(a == b)
                        return 0;
                  if(g[a] != g[b])
                        return g[a] == g[b] ? 0 : g[a] < g[b] ? -1 : 1;
                  else
                        return g[a + h] == g[b + h] ? 0 : g[a + h] < g[b
+ h] ? -1 : 1;
            }

            public int compare1(Integer aa, Integer bb)
            {
                  return compare(aa, bb) == -1 ? 1 : 0;
            }
      }

      void buildSA1(String sas)
      {
            n = sas.length();
            t = sas.toCharArray();
            Integer[] saTemp = new Integer[n + 1];
            for(int i = 0; i < n + 1; i++)
            {
                  saTemp[i] = i;
                  if(i == n)
                        g[i] = 0;
                  else
                        g[i] = t[i];
                  }
                  b[0] = 0; b[n] = 0;
                  Arrays.sort(saTemp, 0, n + 1, new SAComp(0, g));
                  for(int h = 1; b[n] != n ; h *= 2)
                  {
                        SAComp comp = new SAComp(h, g);
                        Arrays.sort(saTemp, 0, n + 1, comp);
                        for(int i = 0; i < n; i++) b[i + 1] = b[i] +
comp.compare1(saTemp[i], saTemp[i + 1]);
                        for(int i = 0; i < n + 1; i++) g[saTemp[i]] = b[i];
                  }
                  for(int i = 0; i < n; i++)
                        sa[i] = saTemp[i + 1];
                  buildLCP();
                  buildRMQ();
            }

            void buildLCP()
            {
                  int[] a = sa;
                  int h = 0;
                  for(int i = 0; i < n; i++) b[a[i]] = i;
                  for(int i = 0; i < n; i++)
                  {
                        if (b[i] != 0)
                        {
                              for(int j = a[b[i] - 1]; j + h < n && i + h < n
&& t[j + h] == t[i + h]; ++h);
                              lcp[b[i]] = h;
                        }
                        else lcp[b[i]] = -1;
                        if(h > 0) --h;
                  }
            }

            void buildRMQ()
            {
                  int[] b = rmq;
                  System.arraycopy(lcp, 0, b, 0, n);
                  int delta = 0;
                  for(int k = 1; k < n; k *= 2)
                  {
                      System.arraycopy(b, delta, b, n + delta, n);
                      delta += n;
                      for(int i = 0; i < n - k; i++)
                      b[i + delta] = Math.min(b[i + delta], b[i + k + delta]);
                  }
            }

            // Naive matching O(m log n)
            int findN(String o)
            {
                  int a = 0, b = n - 1;
                  String este = new String(t);
                  while (a < b)
                  {
                        int c = (a + b) / 2;
                        String nuevo = este.substring(sa[c]);
                        if(nuevo.length() > o.length()) nuevo =
nuevo.substring(0, o.length());
                        if(nuevo.compareTo(o) < 0) a = c + 1;
                        else b = c;
                  }
                  String nuevo = este.substring(sa[a]);
                  if(nuevo.length() > o.length()) nuevo = nuevo.substring(0,
o.length());
                  return nuevo.compareTo(o) == 0 ? sa[a] : -1;
            }

            // longest common prefix between suffixes sa[x] and sa[y]
            int lcp(int x, int y)
            {
                  return x == y ? n - sa[x] : x > y ? minimum(y + 1, x) :
minimum(x + 1, y);
            }

            // range minimum query of lcp array
            int minimum(int x, int y)
            {
                  int z = y - x, k = 0, e = 1, s;
  s = (((z & 0xffff0000) != 0) ? 1 : 0) << 4; z >>= s; e <<= s; k |= s;
  s = (((z & 0x0000ff00) != 0) ? 1 : 0) << 3; z >>= s; e <<= s; k |= s;
  s = (((z & 0x000000f0) != 0) ? 1 : 0) << 2; z >>= s; e <<= s; k |= s;
  s = (((z & 0x0000000c) != 0) ? 1 : 0) << 1; z >>= s; e <<= s; k |= s;
  s = (((z & 0x00000002) != 0) ? 1 : 0) << 0; z >>= s; e <<= s; k |= s;
                  return Math.min(rmq[x + n * k], rmq[y + n * k - e + 1]);
            }

            int computeLCP(char[] t, int n, char[] p, int m, int o, int k)
            {
                  int i = o;
                  for (; i < m && k + i < n && p[i] == t[k + i]; ++i);
                  return i;
            }
            boolean COMP(int h, int k, int m, char[] p)
            {
                  return (h == m || (k + h < n && p[h] < t[k + h]));
            }

            // Mamber-Myers's O(m + log n) string matching with LCP/RMQ
            int find(String o)
            {
                  char[] p = o.toCharArray();
                  int m = o.length();
                  int l = -1, lh = 0, r = n - 1, rh = computeLCP(t, n, p, m,
0, sa[n - 1]);
                  if(!COMP(rh, sa[r], m, p))
                    return -1;
                  for(int k = (l + r) / 2; l + 1 < r; k = (l + r) / 2)
                  {
                        int A = minimum(l + 1, k), B = minimum(k + 1, r);
                        if(A >= B)
                        {
                              if(lh < A) l = k;
                              else if(lh > A)
                              {
                                    r = k;
                                    rh = A;
                              }
                              else
                              {
                                    int i = computeLCP(t, n, p, m, A, sa[k]);
                                    if (COMP(i, sa[k], m, p))
                                    {
                                          r = k;
                                          rh = i;
                                    }
                                    else
                                    {
                                          l = k;
                                          lh = i;
                                    }
                              }
                        }
                        else
                        {
                              if (rh < B) r = k;
                              else if (rh > B)
                              {
                                    l = k;
```

```
                                lh = B;
                        }
                        else
                        {
                                int i = computeLCP(t, n, p, m, B, sa[k]);
                                if(COMP(i, sa[k], m, p))
                                {
                                        r = k;
                                        rh = i;
                                }
                                else
                                {
                                        l = k;
                                        lh = i;
                                }
                        }
                }
        }
        return rh == m ? sa[r] : -1;
    }
}
```

4. DP

4.1. LIS

The **longest increasing subsequence** problem is to find a subsequence of a given sequence in which the subsequence elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible. This subsequence is not necessarily contiguous. Longest increasing subsequences are studied in the context of various disciplines related to mathematics, including algorithmics, random matrix theory,representation theory, and physics. The longest increasing subsequence problem is solvable in time O(*n* log *n*), where *n* denotes the length of the input sequence.[1]

The longest increasing subsequence problem is closely related to the longest common subsequence problem, which has a quadratic timedynamic programming solution: the longest increasing subsequence of a sequence *S* is the longest

common subsequence of *S* and *T*, where *T*is the result of sorting *S*. However, for the special case in which the input is a permutation of the integers 1, 2, ..., *n*, this approach can be made much more efficient, leading to time bounds of the form O(*n* log log *n*).[2]
The largest clique (graph theory) in a permutation graph is defined by the longest decreasing subsequence of the permutation that defines the graph; the longest decreasing subsequence is equivalent, by negation of all numbers, to the longest increasing subsequence. Therefore, longest increasing subsequence algorithms can be used to solve the clique problem in permutation graphs.[3]
The longest increasing subsequence problem can also be related to finding the longest path in a directed acyclic graph derived from the input sequence.

The algorithm outlined below solves the longest increasing subsequence problem efficiently, using only arrays and binary searching. For the first time this algorithm was developed by M. L. Fredman in 1975. It processes the sequence elements in order, maintaining the longest increasing subsequence found so far. Denote the sequence values as X[1], X[2], etc. Then, after processing X[*i*], the algorithm will have stored values in two arrays:

M[*j*] — stores the position *k* of the smallest value X[*k*] such that *k* ≤ *i* (note we have *k* ≤ *j* ≤ *i* here) and there is an increasing subsequence of length *j* ending at X[*k*]

P[*k*] — stores the position of the predecessor of X[*k*] in the longest increasing subsequence ending at X[*k*].

In addition the algorithm stores a variable L representing the length of the longest increasing subsequence found so far.
Note that, at any point in the algorithm, the sequence

X[M[1]], X[M[2]], ..., X[M[L]]

is nondecreasing. For, if there is an increasing subsequence of length *i* ending at X[M[*i*]], then there is also a subsequence of length *i*-1 ending at a smaller value: namely the one ending at P[M[*i*]]. Thus, we may do binary searches in this sequence in logarithmic time.
The algorithm, then, proceeds as follows.

```
L = 0
 for i = 1, 2, ... n:
    binary search for the largest positive j ≤ L such that X[M[j]] < X[i] (or
set j = 0 if no such value exists)
    P[i] = M[j]
    if j == L or X[i] < X[M[j+1]]:
       M[j+1] = i
       L = max(L, j+1)
```

The result of this is the length of the longest sequence in L. The actual longest sequence can be found by backtracking through the P array: the last item of the longest sequence is in X[M[L]], the second-to-last item is in X[P[M[L]]], etc. Thus, the sequence has the form

..., X[P[P[M[L]]]], X[P[M[L]]], X[M[L]].

Because the algorithm performs a single binary search per sequence element, its total time is O(*n* log *n*).

Implementación:

n^2

```
      int lis(int[] entrada, int tam)
      {
            int[] valores = new int[tam];
            for(int i = 0; i < tam; i++)
                valores[i] = 0;
            int mejorM = 0;
            for(int i = 0; i < tam; i++)
            {
                int mejor = 0;
                for(int j = 0; j < i; j++)
                {
                    if(entrada[j] <= entrada[i])
                        mejor = Math.max(mejor, valores[j]);
                }
                valores[i] = mejor + 1;
                mejorM = max(mejorM, valores[i]);
            }
            return mejorM;
      }
```

n * log(n)

```
      static ArrayList <Integer> find_lis(ArrayList <Integer> a)
      {
              ArrayList <Integer> p = new ArrayList <Integer> (a.size());
              ArrayList <Integer> b = new ArrayList <Integer> (a.size());
              for(int i = 0; i < a.size(); i++)
                  p.add(0);
              int u, v;
              if (a.isEmpty()) return b;
              b.add(0);
              for (int i = 1; i < a.size(); i++)
              {
                      if (a.get(b.get(b.size() - 1)) < a.get(i))
                      {
                              p.set(i, b.get(b.size() - 1));
                              b.add(i);
                              continue;
                      }
                      for (u = 0, v = b.size() - 1; u < v;)
                      {
```

```
                    int c = (u + v) / 2;
                    if (a.get(b.get(c)) < a.get(i))
                         u = c+1;
                    else
                         v = c;
               }

               if (a.get(i) < a.get(b.get(u)))
               {
                    if (u > 0)
                         p.set(i, b.get(u-1));
                    b.set(u, i);
               }
          }
          for (u = b.size(), v = b.get(b.size() - 1); u-- != 0; v =
p.get(v))
               b.set(u, v);
          return b;
     }
```

4.2. Subset sum

In computer science, the **subset sum problem** is an important problem in complexity theory and cryptography. The problem is this: given a set of integers, does the sum of some non-empty subset equal exactly zero? For example, given the set { −7, −3, −2, 5, 8}, the answer is *yes* because the subset { −3, −2, 5} sums to zero. The problem is NP-complete.

An equivalent problem is this: given a set of integers and an integer *s*, does any non-empty subset sum to *s*? Subset sum can also be thought of as a special case of the knapsack problem. One interesting special case of subset sum is the partition problem, in which *s* is half of the sum of all elements in the set.
The subset sum problem is a good introduction to the NP-complete class of problems. There are two reasons for this

- It is a decision and not an optimization problem
- It has a very simple formal definition and problem statement.

Although the subset sum problem is a decision problem, the cases when an approximate solution is sufficient have also been studied, in the field of approximations algorithms; one algorithm for the approximate version of the subset sum problem is given below.

The complexity of subset sum

The complexity (difficulty of solution) of subset sum can be viewed as depending on two parameters, N, the number of decision variables, and P, the precision of the problem (stated as the number of binary place values that it takes to state the problem). (Note: here the letters N and P mean something different than what they mean in the **NP** class of problems.)

The complexity of the best known algorithms is exponential in the smaller of the two parameters N and P. Thus, the problem is most difficult if N and P are of the same order. It only becomes easy if either N or P becomes very small.

If N (the number of variables) is small, then an exhaustive search for the solution is practical. If P (the number of place values) is a small fixed number, then there are dynamic programming algorithms that can solve it exactly.

What is happening is that the problem becomes seemingly non-exponential when it is practical to count the entire solution space. There are two ways to count the solution space in the subset sum problem. One is to count the number of ways the variables can be combined. There are $2N$ possible ways to combine the variables. However, with $N = 10$, there are only 1024 possible combinations to check. These can be counted easily with a branching search. The other way is to count all possible numerical values that the combinations can take. There are $2P$ possible numerical sums. However, with $P = 5$ there are only 32 possible numerical values that the combinations can take. These can be counted easily with a dynamic programming algorithm. When $N = P$ and both are large, then there is no aspect of the solution space that can be counted easily.

Efficient algorithms for both small N and small P cases are given below.

[edit]Exponential time algorithm

There are several ways to solve subset sum in time exponential in N. The most naïve algorithm would be to cycle through all subsets of N numbers and, for every one of them, check if the subset sums to the right number. The running time is of order $O(2NN)$, since there are $2N$ subsets and, to check each subset, we need to sum at most N elements.

A better exponential time algorithm is known which runs in time $O(2N/2)$. The algorithm splits arbitrarily the N elements into two sets of $N/2$ each. For each of these two sets, it stores a list of the sums of all $2N/2$ possible subsets of its elements. Which list to be sorted, which would ordinarily take time $O(2N/2N)$. However, given a sorted list of sums for k elements, the list can be expanded to two sorted lists with the introduction of a $(k + 1)$st element, and these two sorted lists can be merged in time $O(2k)$. Thus, each list can be generated in sorted form in time $O(2^{N/2})$. Given the two sorted lists, the algorithm can check if an element of the first array and an element of the second array sum up to s in time $O(2^{N/2})$. To do that, the algorithm passes through the first array in decreasing order (starting at the largest element) and the second array in increasing order (starting at the smallest element). Whenever the sum of the current element in the first array and the current element in the second array is more than s, the algorithm moves to the next element in the first array. If it is less than s, the algorithm moves to the next element in the second array. If two elements with sum s are found, it stops. No better algorithm has been found since Horowitz and Sahni first published this algorithm in 1974[1].

[edit]Pseudo-polynomial time dynamic programming solution

The problem can be solved as follows using dynamic programming. Suppose the sequence is

$x_1, ..., x_n$

and we wish to determine if there is a nonempty subset which sums to 0. Let N be the sum of the negative values and P the sum of the positive values. Define the boolean-valued function$Q(i,s)$ to be the value (**true** or **false**) of

"there is a nonempty subset of $x_1, ..., x_i$ which sums to s".

Thus, the solution to the problem is the value of $Q(n,0)$.

Clearly, $Q(i,s)$ = **false** if $s < N$ or $s > P$ so these values do not need to be stored or computed. Create an array to hold the values $Q(i,s)$ for $1 \leq i \leq n$ and $N \leq s \leq P$.

The array can now be filled in using a simple recursion. Initially, for $N \leq s \leq P$, set

$Q(1,s) := (x_1 = s)$.

Then, for $i = 2, …, n$, set

$Q(i,s) := Q(i - 1,s)$ **or** $(x_i = s)$ **or** $Q(i - 1,s - x_i)$ for $N \leq s \leq P$.

For each assignment, the values of Q on the right side are already known, either because they were stored in the table for the previous value of i or because $Q(i - 1,s - x_i)$ = **false** if$s - x_i < N$ or $s - x_i > P$. Therefore, the total number of arithmetic operations is $O(n(P - N))$. For example, if all the values are $O(n^k)$ for some k, then the time required is $O(n^{k+2})$.

This algorithm is easily modified to return the subset with sum 0 if there is one.

This solution does not count as polynomial time in complexity theory because *P* − *N* is not polynomial in the *size* of the problem, which is the number of bits used to represent it. This algorithm is polynomial in the value of *N* and *P*, which are exponential in their numbers of bits.

A more general problem asks for a subset summing to a specified value (not necessarily 0). It can be solved by a simple modification of the algorithm above. For the case that each *xi* is positive and bounded by the same constant, Pisinger found a linear time algorithm.[2]

[edit]Polynomial time approximate algorithm

An approximate version of the subset sum would be: given a set of *N* numbers *x*1, *x*2, ..., *xN* and a number *s*, output

- yes, if there is a subset that sums up to *s*;
- no, if there is no subset summing up to a number between (1 − *c*)*s* and *s* for some small *c* > 0;
- any answer, if there is a subset summing up to a number between (1 − *c*)*s* and *s* but no subset summing up to *s*.

If all numbers are non-negative, the approximate subset sum is solvable in time polynomial in *N* and 1/*c*.

The solution for subset sum also provides the solution for the original subset sum problem in the case where the numbers are small (again, for nonnegative numbers). If any sum of the numbers can be specified with at most *P* bits, then solving the problem approximately with *c* = 2−*P* is equivalent to solving it exactly. Then, the polynomial time algorithm for approximate subset sum becomes an exact algorithm with running time polynomial in *N* and 2*P* (i.e., exponential in *P*).

The algorithm for the approximate subset sum problem is as follows

```
initialize a list S to contain one element 0.
 for each i from 1 to N do
   let T be a list consisting of xi + y, for all y in S
   let U be the union of T and S
   sort U
   make S empty
   let y be the smallest element of U
   add y to S
   for each element z of U in increasing order do
      //trim the list by eliminating numbers close to one another
      //and throw out elements greater than $s$
     if $y + cs/N < z \leq s$, set $y = z$ and add $z$ to $S$
 if $S$ contains a number between $(1 - c)s$ and $s$, output yes, otherwise no
```

The algorithm is polynomial time because the lists S, T and U always remain of size polynomial in N and $1/c$ and, as long as they are of polynomial size, all operations on them can be done in polynomial time. The size of lists is kept polynomial by the trimming step, in which we only include a number z into S if it is greater than the previous one by cs/N and not greater than s.

This step ensures that each element in S is smaller than the next one by at least cs/N and do not contain elements greater than s. Any list with that property consists of no more thanN/c elements.

The algorithm is correct because each step introduces an additive error of at most cs/N and N steps together introduce the error of at most cs.

4.3. Knapsack problem

The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most useful items.
The problem often arises in resource allocation with financial constraints. A similar problem also appears in combinatorics, complexity theory, cryptography and applied mathematics.
The decision problem form of the knapsack problem is the question "can a value of at least V be achieved without exceeding the weight$W$?"

The most common formulation of the problem is the **0-1 knapsack problem**, which restricts the number *xi* of copies of each kind of item to zero or one. Mathematically the 0-1-knapsack problem can be formulated as:

The **bounded knapsack problem** restricts the number *xi* of copies of each kind of item to a maximum integer value *ci*. Mathematically the bounded knapsack problem can be formulated as:

Dynamic programming solution
[edit]Unbounded knapsack problem
If all weights ($w_1, \ldots, w_n, W$) are nonnegative integers, the knapsack problem can be solved in pseudo-polynomial time using dynamic programming. The following describes a dynamic programming solution for the *unbounded* knapsack problem.

To simplify things, assume all weights are strictly positive (*wi* > 0). We wish to maximize total value subject to the constraint that total weight is less than or equal to *W*. Then for each *w*≤ *W*, define *m[w]* to be the maximum value that can be attained with total weight less than or equal to *w*. *m[W]* then is the solution to the problem.
Observe that *m[w]* has the following properties:

- *m[0] = 0 (the sum of zero items, i.e., the summation of the empty set)*
- $m[w] = \max(m[w-1], \max_{w_i \le w}(v_i + m[w - w_i]))$

where *vi* is the value of the *i*-th kind of item.
Here the maximum of the empty set is taken to be zero. Tabulating the results from *m*[0] up through *m*[*W*] gives the solution. Since the calculation of each *m*[*w*] involves examining *n*items, and there are *W* values of *m*[*w*] to calculate, the running time of the dynamic programming solution is *O*(*nW*). Dividing $w_1,\, w_2,\, \ldots,\, w_n,\, W$ by their greatest common divisor is an obvious way to improve the running time.
The *O*(*nW*) complexity does not contradict the fact that the knapsack problem is NP-complete, since *W*, unlike *n*, is not polynomial in the length of the input to the problem. The length of the input to the problem is proportional to the number, log*W*, of bits in *W*, not to *W* itself.

[edit]0-1 knapsack problem

A similar dynamic programming solution for the *0-1 knapsack problem* also runs in pseudo-polynomial time. As above, assume $w_1,\, w_2,\, \ldots,\, w_n,\, W$ are strictly positive integers. Define *m*[*i*,*w*] to be the maximum value that can be attained with weight less than or equal to *w* using items up to *i*.
We can define *m*[*i*,*w*] recursively as follows:

- *m[0,w] = 0*
- *m[i,0] = 0*
- *m[i,w] = m[i - 1,w]; if wi > w (the new item is more than the current weight limit)*
- $m[i,\, w] = \max(m[i-1,\, w],\, m[i-1,\, w-w_i] + v_i)$; if $w_i \leqslant w$.

The solution can then be found by calculating *m*[*n*,*W*]. To do this efficiently we can use a table to store previous computations. This solution will therefore run in *O*(*nW*) time and *O*(*nW*)space.

4.4. Josephus problem

In computer science and mathematics, the **Josephus problem** (or **Josephus permutation**) is a theoretical problem related to a certain counting-out game.

There are people standing in a circle waiting to be executed. After the first man is executed, certain number of people are skipped and one man is executed. Then again, people are skipped and a man is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last man remains, who is given freedom.
The task is to choose the place in the initial circle so that you are the last one remaining and so survive.

History
The problem is named after Flavius Josephus, a Jewish historian living in the 1st century. According to Josephus' account of the siege of Yodfat, he and his 40 comrade soldiers were trapped in a cave, the exit of which is blocked by Romans. They chose suicide over capture and decided that they would form a circle and start killing themselves using a step of three. Josephus states that by luck or maybe by the hand of God (modern scholars point out that Josephus was a well educated scholar and predicted the outcome), he and another man remained the last and gave up to the Romans.
Solution
We explicitly solve the problem when every 2nd person will be killed, i.e. *k* = 2. (For the more general case $k \neq 2$, we outline a solution below.) We express the solution recursively. Let *f*(*n*) denote the position of the survivor when there are initially *n* people (and *k* = 2). The first time around the circle, all of the even-numbered people die. The second time around the circle, the new 2nd person dies, then the new 4th person, etc.; it's as though there were no first time around the circle. If the initial number of people was even, then the person in position *x* during the second time around the circle was originally in position 2*x* − 1 (for every choice of *x*). So the person in position *f*(*n*) was originally in position 2*f*(*n*) − 1. This gives us the recurrence:

$$f(2n) = 2f(n) - 1.$$

If the initial number of people was odd, then we think of person 1 as dying at the end of the first time around the circle. Again, during the second time around the circle, the new 2nd person dies, then the new 4th person, etc. In this case, the person in position *x* was originally in position 2*x* + 1. This gives us the recurrence:

$$f(2n+1) = 2f(n) + 1.$$

When we tabulate the values of *n* and *f*(*n*) we see a pattern:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
f(*n*)	1	1	3	1	3	5	7	1	3	5	7	9	11	13	15	1

This suggests that *f*(*n*) is an increasing odd sequence that restarts with *f*(*n*) = 1 whenever the index *n* is a power of 2. Therefore, if we choose m and l so that *n* = 2*m* + *l* and $0 \leq l < 2^m$, then $f(n) = 2 \cdot l + 1$. It is clear that values in the table satisfy this equation. Or we can think that after *l* people are dead there are only 2*m* people and we go to the 2*l* + 1th person. He must be the survivor. So *f*(*n*) = 2*l* + 1. But mathematics demands exact proof. Below, we give a proof by induction.

Theorem: If *n* = 2*m* + *l* and $0 \leq l < 2^m$, then *f*(*n*) = 2*l* + 1.
Proof: We use strong induction on *n*. The base case *n* = 1 is true. We consider separately the cases when *n* is even and when *n* is odd.

If *n* is even, then choose *l*1 and *m*1 such that $n/2 = 2^{m_1} + l_1$ and $0 \leq l_1 < 2^{m_1}$. Note that *l*1 = *l* / 2. We have *f*(*n*) = 2*f*(*n* / 2) − 1 = 2((2*l*1) + 1) − 1 = 2*l* + 1, where the second equality follows from the induction hypothesis.

If *n* is odd, then choose *l*1 and *m*1 such that $(n-1)/2 = 2^{m_1} + l_1$ and $0 \leq l_1 < 2^{m_1}$. Note that *l*1 = (*l* − 1) / 2. We have *f*(*n*) = 2*f*((*n* − 1) / 2) + 1 = 2((2*l*1) + 1) + 1 = 2*l* + 1, where the second equality follows from the induction hypothesis. This completes the proof.
The most elegant form of the answer involves the binary representation of size *n*: *f*(*n*) can be obtained by a one-bit left cyclic shift of *n* itself. If we represent *n* in binary as $n = b_0 b_1 b_2 b_3 \dots b_m$, then the solution is given by $f(n) = b_1 b_2 b_3 \dots b_m b_0$. The proof of this follows from the representation of *n* as 2*m* + *l*.
The easiest way to solve this problem in the general case is to use dynamic programming. This approach gives us the recurrence:

$$f(n,k) = (f(n-1,k) + k) \bmod n, \text{ with } f(1,k) = 0,$$

which is evident when considering how the survivor number changes when switching from *n* − 1 to *n*. This approach has running time *O*(*n*), but for small *k* and large *n* there is another approach. The second approach also uses dynamic programming but has running time *O*(*k*log*n*). It is based on considering killing *k*-th, 2*k*-th, ..., $\lfloor n/k \rfloor$-th people as one step, then changing the numbering.

Implementación:

```
public class Josephus
{
    /* todas las posiciones estan entre 0 y n - 1 */
      public static int josephus(int n, int k)
      {
            if(n == 1)
                  return 0;
            return (josephus(n - 1, k) + k) % n;
      }
      /* inicialM es la primera persona que pierde (todas las posiciones estan
entre 0 y n - 1) */
      public static int josephus(int n, int k, int inicialM)
      {
            int normal = josephus(n, k);
            k %= n;
            normal -= (k - 1);
            if(normal < 0)
                  normal += n;
            normal += inicialM;
            return normal % n;
      }
}
```

4.5. All subsets

In mathematics, given a set *S*, the power set (or powerset) of *S*, written $\mathcal{P}(S)$, *P*(*S*), ℘(*S*) or 2*S*, is the set of all subsets of *S*, including the empty set and S itself. In axiomatic set theory (as developed e.g. in the ZFC axioms), the existence of the power set of any set is postulated by the axiom of power set.

Any subset *F* of $\mathcal{P}(S)$ is called a *family of sets* over *S*.
Relation to binomial theorem
The power set is closely related to the binomial theorem. The number of sets with *k* elements in the power set of a set with *n* elements will be a combination *C*(*n*,*k*), also called abinomial coefficient.
For example the power set of a set with three elements, has:

- *C(3,0) = 1 set with 0 elements*
- *C(3,1) = 3 sets with 1 element*
- *C(3,2) = 3 sets with 2 elements*
- *C(3,3) = 1 set with 3 elements.*

[edit]Algorithms
If S is a finite set, there is a recursive algorithm to calculate $\mathcal{P}(S)$.
Define the operation $\mathcal{F}(e,T)=\{X\cup\{e\}|X\in T\}$
In English, return the set with the element e added to each set X in T.

- If $S=\{\}$, then $\mathcal{P}(S)=\{\{\}\}$ is returned.
- Otherwise:
- Let e be any single element of S.
- Let $T=S\setminus\{e\}$, where '$S\setminus\{e\}$' denotes the relative complement of $\{e\}$ in S.
- And the result: $\mathcal{P}(S)=\mathcal{P}(T)\cup\mathcal{F}(e,\mathcal{P}(T))$ is returned.

In other words, the power set of the empty set is the set containing the empty set and the power set of any other set is all the subsets of the set containing some specific element and all the subsets of the set not containing that specific element.
There are other more efficient ways to calculate the power set. For example, use a list of the *n* elements of *S* to fix a mapping from the bit positions of *n*-bit numbers to those elements; then with a simple loop run through all the 2*n* numbers representable with *n* bits, and for each contribute the subset of *S* corresponding to the bits that are set (to 1) in the number. When*n* exceeds the word-length of the computer, typically 64 in modern CPUs but greater in modern GPUs, the representation is naturally extended by using an array of words instead of a single word.

4.6. Minimum biroute pass

The problem of the minimum biroute pass is two find a minimum weight (or length) route that starts from the west-most point or node, makes a pass to the east-most point or node visiting some of the nodes, then makes a second pass from the east-most point or node back to the first one visiting the remaining islands. In each pass the route moves steadily east (in the first pass) or west (in the second pass), but moves as far north or south as needed.

Implementación:

```
double T[MAXP][MAXP];
bool comp[MAXP][MAXP];
double point[MAXP][2];
int N;

double dist(int p1, int p2) {
    double ans = 0;
    for (int i=0; i<2; i++) {
        ans += SQ(point[p1][i]-point[p2][i]);
    }
    return sqrt(ans);
}



double minTour(int e1, int e2) {
    if (e1==N-1 || e2==N-1) return dist(e1,e2);
    if (comp[e1][e2]) return T[e1][e2];

    int n = max(e1,e2)+1;
    double d1 = minTour(n,e2) + dist(e1,n);
    double d2 = minTour(e1,n) + dist(e2,n);

    T[e1][e2] = min(d1,d2); T[e2][e1]=T[e1][e2];
    comp[e1][e2] = true; comp[e2][e1]=true;
    return T[e1][e2];
}

int main() {
    while ( scanf("%d",&N) != EOF ) {
        for (int i=0; i<N; i++) {
            scanf("%lf %lf", &point[i][0], &point[i][1]);
        }
        for (int i=0; i<=N; i++)
            for (int j=0; j<=N; j++)
                comp[i][j]=false;
        double ans = minTour(0,0);
        printf("%.2lf\n",ans);
    }
}
```

4.7. Best sum

The one dimensional best sum is, given a vector, find the contiguous subvector whose sum is maximum. In two dimensions the problem is, given a matrix, find the submatrix whose sum is maximum.

Implementación:

```
public static int oneDimensionSum(int[] arreglo, int n)
{
      int mejor = Integer.MIN_VALUE;
      int[] actual = new int[n];
      actual[0] = arreglo[0];
      for(int i = 1; i < n; i++)
      {
            actual[i] = Math.max(arreglo[i], arreglo[i] + actual[i - 1]);
            mejor = Math.max(mejor, actual[i]);
      }
      return mejor;
}

public static int twoDimensionSum(int[][] matriz, int alto, int ancho)
{
      int mejor = Integer.MIN_VALUE;
```

```
        int[] actual = new int[alto];
        for(int i = 0; i < alto; i++)
        {
              int cuenta = 0;
              for(int j = i; j >= 0; j--)
              {
                    cuenta += matriz[j][0];
                    actual[i - j] = cuenta;
                    mejor = Math.max(mejor, actual[i - j]);
              }
              for(int j = 1; j < ancho; j++)
              {
                    cuenta = 0;
                    for(int k = i; k >= 0; k--)
                    {
                          cuenta += matriz[k][j];
                          int m = Math.max(actual[i - k] + cuenta, cuenta);
                          mejor = Math.max(mejor, m);
                          actual[i - k] = m;
                    }
              }
        }
        return mejor;
  }
```

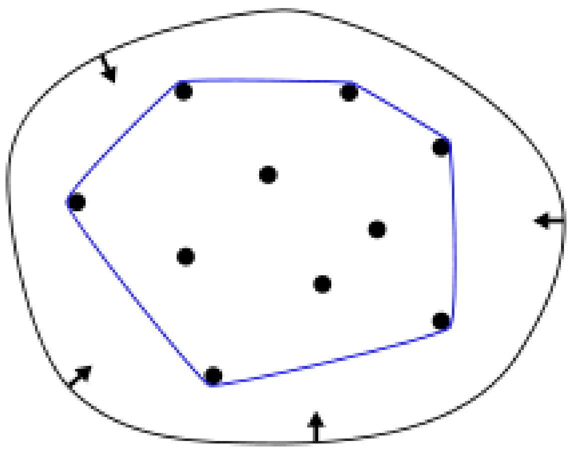
5. Geometry

5.1. Polygons

```
static class Polygon
{
      ArrayList <Line2D> lines;
      Point2D anterior = null;

      public Polygon(int minSize)
      {
            lines = new ArrayList <Line2D> (minSize);
      }

      public void add(Point2D nuevo)
      {
            if(anterior != null)
            {
                  lines.add(new Line2D.Double(anterior, nuevo));
            }
            anterior = nuevo;
      }

      public void close()
      {
```

```
                  lines.add(new Line2D.Double(anterior,
lines.get(0).getP1()));
            }

            public static Point2D subtract(Point2D a, Point2D b)
            {
                  return new Point2D.Double(a.getX() - b.getX(), a.getY() -
b.getY());
            }

            static double abs(Point2D a)
            {
                  return Math.sqrt(a.getX() * a.getX() + a.getY() * a.getY());
            }

            static double cross(Point2D a, Point2D b)
            {
                  return (a.getX() * b.getY()) - (a.getY() * b.getX());
            }

            static boolean is_point_online(Point2D a, Point2D b, Point2D c)
            {
                  return abs(subtract(a, c)) + abs(subtract(b, c)) <=
abs(subtract(a, b));
            }

            // Ya esta implementado en ((Line2D) lineA).intersectsLine(lineB)
            public static boolean lines_intersect(Line2D a, Line2D b)
            {
                  return cross(subtract(a.getP2(), a.getP1()),
subtract(b.getP1(), a.getP1())) *
                           cross(subtract(a.getP2(), a.getP1()),
subtract(b.getP2(), a.getP1())) < 0 &&
                           cross(subtract(b.getP2(), b.getP1()),
subtract(a.getP1(), b.getP1())) *
                           cross(subtract(b.getP2(), b.getP1()),
subtract(a.getP2(), b.getP1())) < 0;
            }

            // Funciona para cualquier poligono excepto si es self-
intersecting
            public double area()
            {
                  double area = 0;
                  for (Line2D line : lines)
                  {
                        area += line.getP1().getX() * line.getP2().getY();
                        area -= line.getP2().getX() * line.getP1().getY();
                  }
                  area /= 2.0;
                  return Math.abs(area);
            }

            // Funciona para cualquier poligono excepto si es self-
intersecting
            public double areaUnsigned()
            {
                  double area = 0;
                  for (Line2D line : lines)
                  {
                        area += line.getP1().getX() * line.getP2().getY();
                        area -= line.getP2().getX() * line.getP1().getY();
                  }
                  area /= 2.0;
                  return area;
            }

            // Funciona para cualquier poligono excepto si es self-
intersecting
            public Point2D centerOfMass()
            {
                  double cx = 0, cy = 0;
                  double area = areaUnsigned();
                  double factor = 0;
                  for (Line2D line : lines)
                  {
                        factor = line.getP1().getX() * line.getP2().getY() -
line.getP2().getX() * line.getP1().getY();
                        cx += (line.getP1().getX() + line.getP2().getX()) *
factor;
                        cy += (line.getP1().getY() + line.getP2().getY()) *
factor;
                  }
                  area *= 6.0d;
                  factor = 1 / area;
                  cx *= factor;
                  cy *= factor;
                  return new Point2D.Double(cx, cy);
            }

            public boolean intersects(Polygon other)
            {
                  for(Line2D lineA : lines)
                  {
                        if(other.contains(lineA.getP1()))
                              return true;
                        for(Line2D lineB : other.lines)
                        {
                              if(lineA.intersectsLine(lineB))
                                    return true;
                        }
                  }
                  for(Line2D line : other.lines)
                  {
                        if(contains(line.getP1()))
                              return true;
                  }
                  return false;
            }

            public boolean contains(Point2D p)
            {
                    int cnt = 0;
                    for(Line2D line : lines)
                    {
                      Point2D curr = subtract(line.getP1(), p);
                      Point2D next = subtract(line.getP2(), p);
                      if(curr.getY() > next.getY())
                      {
                        Point2D temp = curr;
                        curr = next;
                        next = temp;
                      }
                      if (curr.getY() < 0 && 0 <= next.getY() && cross(next,
curr) >= 0)
                      {
                        cnt++;
                      }
                      if (is_point_online(line.getP1(), line.getP2(), p))
                              return true;
                    }
                    return  cnt % 2 == 1;
            }
      }
```

5.2. Convex hull

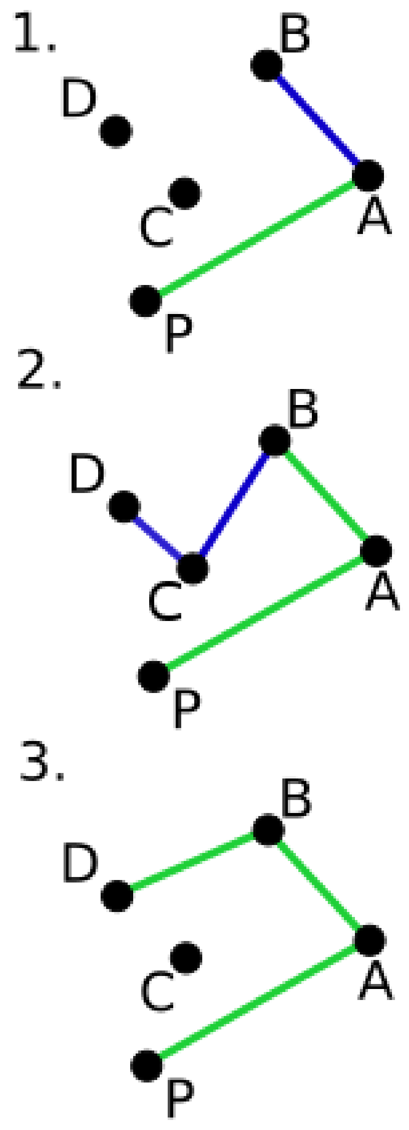


Convex hull: elastic band analogy

In mathematics, the **convex hull** or **convex envelope** for a set of points *X* in a real vector space V is the minimal convex set containing *X*.
In computational geometry, a basic problem is finding the convex hull for a given finite nonempty set of points in the plane. It is common to use the term "convex hull" for the boundary of that set, which is a convex polygon, except in the degenerate case that points are collinear. The convex hull is then typically represented by a sequence of the vertices of the line segments forming the boundary of the polygon, ordered along that boundary.

The **Graham scan** is a method of computing the convex hull of a finite set of points in the plane with time complexity O(*n* log *n*). It is named after Ronald Graham, who published the original algorithm in 1972[1]. The algorithm finds all vertices of the convex hull ordered along its boundary.

Algorithm



As one can see, A to B and B to C are counterclockwise, but C to D isn't. The algorithm detects this situation and discards previously chosen segments until the turn taken is counterclockwise (B to D in this case.)

The first step in this algorithm is to find the point with the lowest y-coordinate. If the lowest y-coordinate exists in more than one point in the set, the point with the lowest x-coordinate out of the candidates should be chosen. Call this point *P*. This step takes O(*n*), where *n* is the number of points in question.

Next, the set of points must be sorted in increasing order of the angle they and the point *P* make with the x-axis. Any general-purpose sorting algorithm is appropriate for this, for example heapsort (which is O(*n* log *n*)). In order to speed up the calculations, it is not actually necessary to calculate the actual angle these points make with the x-axis; instead, it suffices to

calculate the cosine of this angle: it is a monotonically decreasing function in the domain in question (which is 0 to 180 degrees, due to the first step) and may be calculated with simple arithmetic.
The algorithm proceeds by considering each of the points in the sorted array in sequence. For each point, it is determined whether moving from the two previously considered points to this point is a "left turn" or a "right turn". If it is a "right turn", this means that the second-to-last point is not part of the convex hull and should be removed from consideration. This process is continued for as long as the set of the last three points is a "right turn". As soon as a "left turn" is encountered, the algorithm moves on to the next point in the sorted array. (If at any stage the three points are collinear, one may opt either to discard or to report it, since in some applications it is required to find all points on the boundary of the convex hull.)
Again, determining whether three points constitute a "left turn" or a "right turn" does not require computing the actual angle between the two line segments, and can actually be achieved with simple arithmetic only. For three points ($x1$,$y1$), ($x2$,$y2$) and ($x3$,$y3$), simply compute the direction of the cross product of the two vectors defined by points($x1$,$y1$), ($x2$,$y2$) and ($x1$,$y1$), ($x3$,$y3$), characterized by the sign of the expression $(x2 - x1)(y3 - y1) - (y2 - y1)(x3 - x1)$. If the result is 0, the points are collinear; if it is positive, the three points constitute a "left turn", otherwise a "right turn".
This process will eventually return to the point at which it started, at which point the algorithm is completed and the stack now contains the points on the convex hull in counterclockwise order.
Time complexity
Sorting the points has time complexity O(n log n). While it may seem that the time complexity of the loop is O(n2), because for each point it goes back to check if any of the previous points make a "right turn", it is actually O(n), because each point is considered at most twice in some sense. Each point can appear only once as a point (x3,y3) in a "left turn" (because the algorithm advances to the next point (x3,y3) after that), and as a point (x2,y2) in a "right turn" (because the point (x2,y2) is removed). The overall time complexity is therefore O(n log n), since the time to sort dominates the time to actually compute the convex hull.
Pseudocode
First, define

```
# Three points are a counter-clockwise turn if ccw > 0, clockwise if
# ccw < 0, and collinear if ccw = 0 because ccw is a determinant that
# gives the signed area of the triangle formed by p1, p2, and p3.
function ccw(p1, p2, p3):
    return (p2.x - p1.x)*(p3.y - p1.y) - (p2.y - p1.y)*(p3.x - p1.x)
```

Then let the result be stored in the array points.

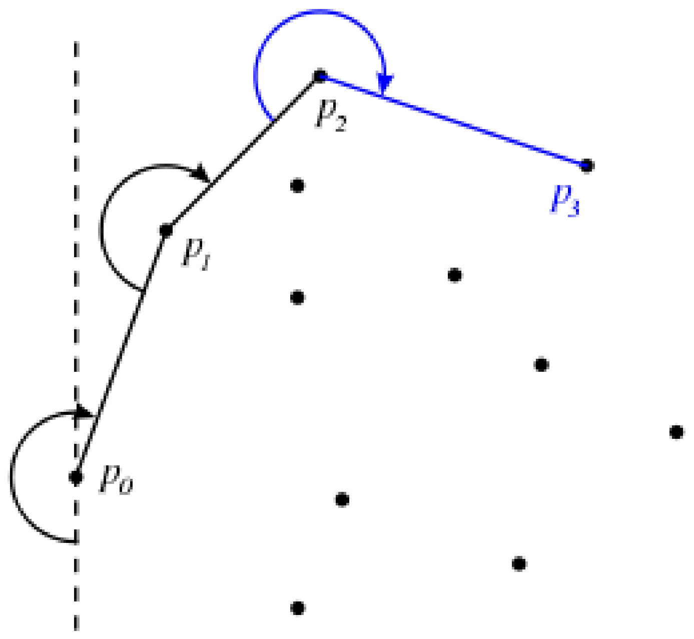
```
let N           = number of points
let points[N+1] = the array of points
swap points[1] with the point with the lowest y-coordinate
```

```
sort points by polar angle with points[1]

# We want points[0] to be a sentinel point that will stop the loop.
let points[0] = points[N]

# M will denote the number of points on the convex hull.
let M = 2
for i = 3 to N:
    # Find next valid point on convex hull.
    while ccw(points[M-1], points[M], points[i]) <= 0:
       M -= 1

    # Swap points[i] to the correct place and update M.
    M += 1
    swap points[M] with points[i]
```

This pseudocode is adapted from Sedgewick and Wayne's *Algorithms, 4th edition*.
Notes
The same basic idea works also if the input is sorted on x-coordinate instead of angle, and the hull is computed in two steps producing the upper and the lower parts of the hull respectively. This modification was devised by A. M. Andrew and is know as Andrew's Monotone Chain Algorithm. It has the same basic properties as Graham's Scan but eschews costly comparisons between polar angles.[2]
The stack technique used in the Graham scan algorithm is very similar to that for the all nearest smaller values problem, and parallel algorithms for all nearest smaller values may also be used (like the Graham scan) to compute convex hulls of sorted sequences of points efficiently.[3]

The **gift wrapping algorithm** is a simple algorithm for computing the convex hull of a given set of points.

Planar case
In the two-dimensional case the algorithm is also known as **Jarvis march**, after R. A. Jarvis, who published it in 1973; it has O(*nh*) time complexity, where *n* is the number of points and *h* is the number of points on the convex hull. Its real-life performance compared with other convex hull algorithms is favorable when n is small or h is expected to be very small with respect to n. In general cases the algorithm is outperformed by many others.
Algorithm



Jarvis march computing the convex hull.

For the sake of simplicity, the description below assumes that the points are in general position, i.e., no three points are collinear. The algorithm may be easily modified to deal with collinearity, including the choice whether it should report only extreme points (vertices of the convex hull) or all points that lie on the convex hull. Also, the complete implementation must deal with degenerate cases when the convex hull has only 1 or 2 vertices, as well as with the issues of limited arithmetic precision, both of computer computations and input data.

The gift wrapping algorithm begins with *i*=0 and a point *p0* known to be on the convex hull, e.g., the leftmost point, and selects the point *pi+1* such that all points are to the right of the line *pi pi+1*. This point may be found in O(*n*) time by comparing polar angles of all points with respect to point *pi* taken for the center of polar coordinates. Letting *i*=*i*+1, and repeating with until one reaches *ph*=*p0* again yields the convex hull in *h* steps. In two dimensions, the gift wrapping algorithm is similar to the process of winding a string (or wrapping paper) around the set of points.

The approach can be extended to higher dimensions.

Pseudocode

```
jarvis(S)
   pointOnHull = leftmost point in S
   i = 0
   repeat
      P[i] = pointOnHull
      endpoint = S[0]         // initial endpoint for a candidate edge on the
hull
      for j from 1 to |S|-1
         if (S[j] is on left of line from P[i] to endpoint)
```

```
            endpoint = S[j]   // found greater left turn, update endpoint
        i = i+1
        pointOnHull = endpoint
    until endpoint == P[0]      // wrapped around to first hull point
```

Complexity
Notice that the inner loop checks every point in the set *S*, and the outer loop repeats for each point on the hull. Hence the total run time is *nh*. The run time depends on the size of the output, so Jarvis March is anoutput-sensitive algorithm.

Implementación:

```
public class Convex_Hull
{
	static double cross(Point2D p1, Point2D p2, Point2D p3)
	{
		 return  (p2.getX()  -  p1.getX())  *  (p3.getY()  -  p1.getY())  -
(p2.getY() - p1.getY()) * (p3.getX() - p1.getX());
	}

	static class Comp implements Comparator <Point2D>
	{
		@Override
		public int compare(Point2D p1, Point2D p2)
		{
			if(p1.getY() < p2.getY()) return -1;
			if(p1.getY() > p2.getY()) return 1;
			if(p1.getX() < p2.getX()) return -1;
			return 1;
		}
	}

	static final double EPSILON = 1e-12;

	static void convexHull(List <Point2D> points, List <Point2D> result)
	{
		int n = points.size();
		Point2D[] p2 = new Point2D[points.size() + 1];
	    Collections.sort(points, new Comp());
	    int top = 0;
	    p2[top++] = points.get(0);
	    p2[top++] = points.get(1);
	    for (int i = 2; i < n; i++)
	    {
	           while  (top  >=  2  &&  cross(p2[top  -  2],  p2[top  -  1],
points.get(i)) <= -EPSILON)
```

```
                    --top;
                p2[top++] = points.get(i);
            }
            int r = top;
            for (int i = n - 2; i >= 0; i--)
            {
                     while  (top  >  r  &&  cross(p2[top  -  2],  p2[top  -  1],
points.get(i)) <= -EPSILON)
                    --top;
                if (i != 0)
                    p2[top++] = points.get(i);
            }
            for(int i = 0; i < top; i++)
              result.add(p2[i]);
        }
}
```

Otra implementación

```
struct Point
{
    double x, y;
};

int n;

double dis(Point p1, Point p2)
{
    return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
}

double cross(Point p1, Point p2, Point p3)
{
    return (p2.x-p1.x)*(p3.y-p1.y) - (p2.y-p1.y)*(p3.x-p1.x);
}

bool comp(const Point &p1, const Point &p2)
{
    if(p1.y < p2.y) return true;
    if(p1.y > p2.y) return false;
    if(p1.x < p2.x) return true;
}

void getconvexhull(Point p1[], Point p2[], int &top)
{
    sort(p1, p1 + n, comp);
    int i;
    top = 0;
    p2[top++] = p1[0];
    p2[top++] = p1[1];
    for (i = 2; i < n; i++)
    {
        while (top >= 2 && cross(p2[top - 2], p2[top - 1], p1[i]) <= 0)
            --top;
        p2[top++] = p1[i];
    }
    int r = top;
    for (i = n - 2; i >= 0; i--)
    {
        while (top > r && cross(p2[top - 2], p2[top - 1], p1[i]) <= 0)
            --top;
        if (i != 0)
            p2[top++] = p1[i];
    }
}

/*
  Otra forma de verlo:
  vector<point> convex_hull(vector<point> ps) {
  int n = ps.size(), k = 0;
  sort(ps.begin(), ps.end());
  vector<point> ch(2*n);
  for (int i = 0; i < n; ch[k++] = ps[i++]) // lower-hull
    while (k >= 2 && ccw(ch[k-2], ch[k-1], ps[i]) <= 0) --k;
  for (int i = n-2, t = k+1; i >= 0; ch[k++] = ps[i--]) // upper-hull
    while (k >= t && ccw(ch[k-2], ch[k-1], ps[i]) <= 0) --k;
  ch.resize(k-1);
  return ch;
}
*/
```

5.3. Sweep line

In computational geometry, the **Bentley–Ottmann algorithm** is a sweep line algorithm for listing all crossings in a set of line segments. It extends the Shamos–Hoey algorithm, a similar previous algorithm for testing whether or not a set of line segments has any crossings. For an input consisting of *n* line segments with *k* crossings, the Bentley–Ottmann algorithm takes time O((*n* + *k*) log *n*), a significant improvement on a naive algorithm that tests every pair of segments.

The algorithm was initially developed by Jon Bentley and Thomas Ottmann (1979); it is described in more detail in the textbooks Preparata & Shamos (1985), O'Rourke (1998), and de Berg et al. (2000). Although asymptotically faster algorithms are now known, the Bentley–Ottman algorithm remains a practical choice due to its simplicity and low memory requirements.
Overall strategy
The main idea of the Bentley–Ottmann algorithm is to use a sweep line approach, in which a vertical line *L* moves from left to right across the plane, intersecting the input line segments in sequence as it moves.[1] It is simplest to describe the algorithm for the case that the input is in general position, meaning that no two line segment endpoints or crossings have the same *x*-coordinate, no segment endpoint lies on another segment, and no three line segments cross at the same point. In this case, *L* will always intersect the input line segments in a set of points whose vertical ordering changes only at a finite set of discrete *events*. Thus, the continuous motion of *L* can be broken down into a finite sequence of steps, and simulated by an algorithm that runs in a finite amount of time.
There are two types of event that may happen during the course of this simulation. When *L* sweeps across an endpoint of a line segment *s*, the intersection of *L* with *s* is added to or removed from the vertically ordered set of intersection points. These events are easy to predict, as the endpoints are known already from the input to the algorithm. The remaining events occur when *L* sweeps across a crossing between two line segments *s* and *t*. These events may also be predicted from the fact that, just prior to the event, the points of intersection of *L*with *s* and *t* are adjacent in the vertical ordering of the intersection points.
The Bentley–Ottman algorithm itself maintains data structures representing the current vertical ordering of the intersection points of the sweep line with the input line segments, and a collection of potential future events formed by adjacent pairs of intersection points. It processes each event in turn, updating its data structures to represent the new set of intersection points.
n order to efficiently maintain the intersection points of the sweep line *L* with the input line segments and the sequence of future events, the Bentley–Ottman algorithm maintains two data structures:

- A binary search tree, containing the set of input line segments that cross *L*, ordered by the *y*-coordinates of the points where these segments cross *L*. The crossing points themselves are not represented explicitly in the binary search tree. The Bentley–Ottman algorithm will insert a new segment *s* into this data structure when the sweep line *L* crosses the left endpoint *p* of this segment; the correct position of *s* in the binary search tree may be determined by a binary search, each step of which tests whether *p* is above or below some other segment that is crossed by *L*. Thus, an insertion may be performed in logarithmic time. The Bentley–Ottmann algorithm will also delete segments from the binary search tree, and use the binary search tree to determine the segments that are immediately above or below other segments;

these operations may be performed using only the tree structure itself without reference to the underlying geometry of the segments.

- A priority queue (the "event queue"), used to maintain a sequence of potential future events in the Bentley–Ottmann algorithm. Each event is associated with a point *p* in the plane, either a segment endpoint or a crossing point, and the event happens when line *L* sweeps over *p*. Thus, the events may be prioritized by the *x*-coordinates of the points associated with each event. In the Bentley–Ottmann algorithm, the potential future events consist of line segment endpoints that have not yet been swept over, and the points of intersection of pairs of lines containing pairs of segments that are immediately above or below each other.

The algorithm does not need to maintain explicitly a representation of the sweep line *L* or its position in the plane. Rather, the position of *L* is represented indirectly: it is the vertical line through the point associated with the most recently processed event.

The binary search tree may be any balanced binary search tree data structure, such as a red-black tree; all that is required is that insertions, deletions, and searches take logarithmic time. Similarly, the priority queue may be a binary heap or any other logarithmic-time priority queue; more sophisticated priority queues such as a Fibonacci heap are not necessary.

The Bentley–Ottmann algorithm performs the following steps.

1. Initialize a priority queue *Q* of potential future events, each associated with a point in the plane and prioritized by the *x*-coordinate of the point. Initially, *Q* contains an event for each of the endpoints of the input segments.
2. Initialize a binary search tree *T* of the line segments that cross the sweep line *L*, ordered by the *y*-coordinates of the crossing points. Initially, *T* is empty.
3. While *Q* is nonempty, find and remove the event from *Q* associated with a point *p* with minimum *x*-coordinate. Determine what type of event this is and process it according to the following case analysis:
   - If *p* is the left endpoint of a line segment *s*, insert *s* into *T*. Find the segments *r* and *t* that are immediately below and above *s* in *T* (if they exist) and if their crossing forms a potential future event in the event queue, remove it. If *s* crosses *r* or *t*, add those crossing points as potential future events in the event queue.
   - If *p* is the right endpoint of a line segment *s*, remove *s* from *T*. Find the segments *r* and *t* that were (prior to the removal of *s*) immediately above and below it in *T* (if they exist). If *r* and *t* cross, add that crossing point as a potential future event in the event queue.

    - If *p* is the crossing point of two segments *s* and *t* (with *s* below *t* to the left of the crossing), swap the positions of *s* and *t* in *T*. Find the segments *r* and *u* (if they exist) that are immediately below and above *s* and *t* respectively. Remove any crossing points *rs* and *tu* from the event queue, and, if *r* and *t* cross or *s* and *u* cross, add those crossing points to the event queue.

The algorithm description above assumes that line segments are not vertical, that line segment endpoints do not lie on other line segments, that crossings are formed by only two line segments, and that no two event points have the same *x*-coordinate. However, these general position assumptions are not reasonable for most applications of line segment intersection.Bentley (Ottmann) suggested perturbing the input slightly to avoid these kinds of numerical coincidences, but did not describe in detail how to perform these perturbations. de Berg et al. (2000) describe in more detail the following measures for handling special-position inputs:

- Break ties between event points with the same *x*-coordinate by using the *y*-coordinate. Events with different *y*-coordinates are handled as before. This modification handles both the problem of multiple event points with the same *x*-coordinate, and the problem of vertical line segments: the left endpoint of a vertical segment is defined to be the one with the lower *y*-coordinate, and the steps needed to process such a segment are essentially the same as those needed to process a non-vertical segment with a very high slope.
- Define a line segment to be a closed set, containing its endpoints. Therefore, two line segments that share an endpoint, or a line segment that contains an endpoint of another segment, both count as an intersection of two line segments.
- When multiple line segments intersect at the same point, create and process a single event point for that intersection. The updates to the binary search tree caused by this event may involve removing any line segments for which this is the right endpoint, inserting new line segments for which this is the left endpoint, and reversing the order of the remaining line segments containing this event point. The output from the version of the algorithm described by de Berg et al. (2000) consists of the set of intersection points of line segments, labeled by the segments they belong to, rather than the set of pairs of line segments that intersect.

5.4. Great-circle distance

The great-circle distance or orthodromic distance is the shortest distance between any two points on the surface of a sphere measured along a path on the surface of the sphere (as opposed to going through the sphere's interior). Because spherical geometry is rather different from ordinary Euclidean geometry, the equations for distance take on a different form. The distance between two points in Euclidean space is the length of a straight line from one point to the other. On the sphere, however, there are no straight lines. In non-Euclidean geometry, straight lines are replaced with geodesics. Geodesics on the sphere are the *great circles* (circles on the sphere whose centers are coincident with the center of the sphere).
Between any two different points on a sphere which are not directly opposite each other, there is a unique great circle. The two points separate the great circle into two arcs. The length of the shorter arc is the great-circle distance between the points. A great circle endowed with such a distance is the Riemannian circle.
Between two points which are directly opposite each other, called *antipodal points*, there are infinitely many great circles, but all great circle arcs between antipodal points have the same length, i.e. half the circumference of the circle, or π*r*, where *r* is the radius of the sphere.
Because the Earth is approximately spherical (see Earth radius), the equations for great-circle distance are important for finding the shortest distance between points on the surface of the Earth (*as the crow flies*), and so have important applications in navigation.
[edit]Formulae
Let $\phi_s, \lambda_s;\ \phi_f, \lambda_f$ be the geographical latitude and longitude of two points (a base "standpoint" and the destination "forepoint"), respectively, and $\Delta\phi, \Delta\lambda$ their differences and $\Delta\widehat{\sigma}$ the (spherical) angular difference/distance, or central angle, which can be constituted from the spherical law of cosines:

$$\Delta\widehat{\sigma} = \arccos\big(\sin\phi_s \sin\phi_f + \cos\phi_s \cos\phi_f \cos\Delta\lambda\big).$$

The distance *d*, i.e. the arc length, for a sphere of radius *r* and $\Delta\widehat{\sigma}$ given in radians, is then:

$$d = r\,\Delta\widehat{\sigma}.$$

This arccosine formula above can have large rounding errors for the common case where the distance is small, however, so it is not normally used for manual calculations. Instead, an equation known historically as the haversine formula was preferred, which is much more numerically stable for small distances:[1]

$$\Delta\widehat{\sigma} = 2\arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos\phi_s \cos\phi_f \sin^2\left(\frac{\Delta\lambda}{2}\right)}\right).$$

Historically, the use of this formula was simplified by the availability of tables for the haversine function: hav(*θ*) = sin2 (*θ*/2).

Although this formula is accurate for most distances, it too suffers from rounding errors for the special (and somewhat unusual) case of antipodal points (on opposite ends of the sphere). A more complicated formula that is accurate for all distances is the following special case (a sphere, which is an ellipsoid with equal major and minor axes) of the Vincenty formula(which more generally is a method to compute distances on ellipsoids):[2]

$$\Delta\hat{\sigma} = \arctan\left(\frac{\sqrt{\left(\cos\phi_f\sin\Delta\lambda\right)^2 + \left(\cos\phi_s\sin\phi_f - \sin\phi_s\cos\phi_f\cos\Delta\lambda\right)^2}}{\sin\phi_s\sin\phi_f + \cos\phi_s\cos\phi_f\cos\Delta\lambda}\right).$$

When programming a computer, one should use the atan2() function rather than the ordinary arctangent function (atan()), in order to simplify handling of the case where the denominator is zero, and to compute $\Delta\widehat{\sigma}$ unambiguously in all quadrants.
When using a spreadsheet program such as Excel the arccosine formula is suitable since it is simpler and rounding errors disappears with high precision used.
If *r* is the great-circle radius of the sphere, then the great-circle distance is $r\,\Delta\widehat{\sigma}$.
[edit]Vector version
Another representation of similar formulas, but using using *n*-vector instead of latitude/longitude to describe the positions, is:[3]

$$\Delta\hat{\sigma} = \arccos\left(\boldsymbol{n}^e_{es}\cdot\boldsymbol{n}^e_{ef}\right)$$

$$\Delta\hat{\sigma} = \arcsin\left(\left|\boldsymbol{n}^e_{es}\times\boldsymbol{n}^e_{ef}\right|\right)$$

$$\Delta\hat{\sigma} = \arctan\left(\frac{\left|\boldsymbol{n}^e_{es}\times\boldsymbol{n}^e_{ef}\right|}{\boldsymbol{n}^e_{es}\cdot\boldsymbol{n}^e_{ef}}\right)$$

where $\boldsymbol{n}^e_{es}$ and $\boldsymbol{n}^e_{ef}$ are the *n*-vectors representing the two positions *s* and *f*. Similarly to the equations above based on latitude and longitude, the expression based on arctan is the only one that is well-conditioned for all angles.[*citation needed*] If the two positions are originally given as latitudes and longitudes, a conversion to *n*-vectors must first be performed.
[edit]Radius for spherical Earth
See also: Earth radius
The shape of the Earth closely resembles a flattened sphere (a spheroid) with equatorial radius *a* of 6,378.137 km; distance *b* from the center of the spheroid to each pole is 6356.752 km. When calculating the length of a short north-south line at the equator, the sphere that best approximates that part of the spheroid has a radius of *b*2 / *a*, or 6,335.439 km, while the spheroid at the poles is best approximated by a sphere of radius *a*2 / *b*, or 6,399.594 km, a 1% difference. So as long as we're assuming a spherical Earth, any single formula for distance on the Earth is only guaranteed correct within 0.5% (though we can do better if our formula is only intended to apply to a limited

area). The average radius for a spherical approximation of the figure of the Earth is approximately 6371.01 km (3958.76 statute miles, 3440.07 nautical miles).
[edit]Worked example
For an example of the formula in practice, take the latitude and longitude of two airports:

- Nashville International Airport (BNA) in Nashville, TN, USA: N 36°7.2', W 86°40.2'
- Los Angeles International Airport (LAX) in Los Angeles, CA, USA: N 33°56.4', W 118°24.0'

First convert the co-ordinates to decimal degrees (Sign × (Deg + (Min + Sec / 60) / 60)).

- BNA: $\phi_s = 36.12^\circ \quad \lambda_s = -86.67^\circ$
- LAX: $\phi_f = 33.94^\circ \quad \lambda_f = -118.40^\circ$

Plug these values into the law of cosines

$$\cos\Delta\widehat{\sigma} = \cos\phi_s\cos\phi_f\cos(\lambda_f - \lambda_s) + \sin\phi_s\sin\phi_f$$

Convert the resulting $\widehat{\sigma}$ to radians (multiply by π and divide by 180), then

$$r\,\Delta\widehat{\sigma} \approx 6371.01 \times 0.45306 \approx 2886.45\ \mathrm{km}.$$

Thus the distance between LAX and BNA is about 2886 km or 1794 statute miles (× 0.621371) or 1557 nautical miles (× 0.539957). Actual distance between the given coordinates on theGRS 80/WGS 84 spheroid is 2892.8 km.

Implementación:

```
static double greatCircleDistance(double latitudeS, double longitudeS, double
latitudeF, double longitudeF, double r)
      {
            latitudeS = Math.toRadians(latitudeS);
            latitudeF = Math.toRadians(latitudeF);
            longitudeS = Math.toRadians(longitudeS);
            longitudeF = Math.toRadians(longitudeF);
            double deltaLongitude = longitudeF - longitudeS;
            double a = Math.cos(latitudeF) * Math.sin(deltaLongitude);
            double b = Math.cos(latitudeS) * Math.sin(latitudeF);
            b -= Math.sin(latitudeS) * Math.cos(latitudeF) *
Math.cos(deltaLongitude);
            double c = Math.sin(latitudeS) * Math.sin(latitudeF);
            c += Math.cos(latitudeS) * Math.cos(latitudeF) *
Math.cos(deltaLongitude);
/*      En linea recta -> dist
        double ax = Math.cos(latitudeS) * Math.cos(longitudeS);
        double ay = Math.cos(latitudeS) * Math.sin(longitudeS);
        double az = Math.sin(latitudeS);
        double bx = Math.cos(latitudeF) * Math.cos(longitudeF);
        double by = Math.cos(latitudeF) * Math.sin(longitudeF);
        double bz = Math.sin(latitudeF);
        double dist = r*Math.sqrt(sqr(ax-bx)+ sqr(ay-by)+sqr(az-bz));*/
            return Math.atan(Math.sqrt(a * a + b * b) / c) * r;
    }
```

5.5. Pick's theorem

Often we have to deal with polygons whose vertices have integer coordinates. Such polygons are called lattice polygons. In his tutorial on Geometry Concepts, lbackstrom presents a neat way for finding the area of a lattice polygon given its vertices. Now, suppose we do not know the exact position of the vertices and instead we are given two values:
B = number of lattice points on the boundary of the polygon

I = number of lattice points in the interior of the polygon
Amazingly, the area of this polygon is then given by:
Area = B/2 + I - 1
The above formula is called Pick's Theorem due to Georg Alexander Pick (1859 - 1943). In order to show that Pick's theorem holds for all lattice polygons we have to prove it in 4 separate parts. In the first part we show that the theorem holds for any lattice rectangle (with sides parallel to axis). Since a right-angled triangle is simply half of a rectangle it is not too difficult to show that the theorem also holds for any right-angled triangle (with sides parallel to axis). The next step is to consider a general triangle, which can be represented as a rectangle with some right-angled triangles cut out from its corners. Finally, we can show that if the theorem holds for any two lattice polygons sharing a common side then it will also hold for the lattice polygon, formed by removing the common side. Combining the previous result with the fact that every simple polygon is a union of triangles gives us the final version of Pick's Theorem. Pick's theorem is useful when we need to find the number of lattice points inside a large polygon.

Another formula worth remembering is Euler's Formula for polygonal nets. A polygonal net is a simple polygon divided into smaller polygons. The smaller polygons are called faces, the sides of the faces are called edges and the vertices of the faces are called vertices. Euler's Formula then states:
V - E + F = 2, where

V = number of vertices
E = number of edges
F = number of faces
For example, consider a square with both diagonals drawn. We have V = 5, E = 8 and F = 5 (the outside of the square is also a face) and so V - E + F = 2.

We can use induction to show that Euler's formula works. We must begin the induction with V = 2, since every vertex has to be on at least one edge. If V = 2 then there is only one type of polygonal net possible. It has two vertices connected by E number of edges. This polygonal net has E faces (E - 1 "in the middle" and 1 "outside"). So V - E + F = 2 - E + E = 2. We now assume that V - E + F = 2 is true for all 2<=V<=n. Let V = n + 1. Choose any vertex w at random. Now suppose w is joined to the rest of the net by G edges. If we remove w and all these edges, we have a net with n vertices, E - G edges and F - G + 1 faces. From our assumption, we have:
(n) - (E - G) + (F - G + 1) = 2
thus (n+1) - E + F = 2
Since V = n + 1, we have V - E + F = 2. Hence by the principal of mathematical induction we have proven Euler's formula.

5.6. Lines and circles

```
// d es distancia entre los centros de los circulos
double areaintersectCircles(double r1, double r2, double d)
{
    if(d == 0 || d + r2 < r1 || d + r1 < r2)
        return min(M_PI * r1 * r1, M_PI * r2 * r2);
    double a = pow(r1, 2) * acos((pow(d, 2) + pow(r1, 2) - pow(r2, 2)) / (2 *
d * r1));
    double b = pow(r2, 2) * acos((pow(d, 2) + pow(r2, 2) - pow(r1, 2)) / (2 *
d * r2));
    double c = (sqrt((-d + r1 + r2) * (d + r1 - r2) * (d - r1 + r2) * (d + r1
+ r2))) / 2;
    double respuesta = a + b - c;
    if(respuesta != respuesta)
        return 0;
    return respuesta;
}

bool circlesIntersect(double r1, double r2, double d)
{
      return r1 + r2 < d; // o <= dependiendo de la definicion
}

float
dist3D_Line_to_Line( Line L1, Line L2)
{
    Vector   u = L1.P1 - L1.P0;
    Vector   v = L2.P1 - L2.P0;
    Vector   w = L1.P0 - L2.P0;
    float    a = dot(u,u);        // always >= 0
    float    b = dot(u,v);
    float    c = dot(v,v);         // always >= 0
    float    d = dot(u,w);
    float    e = dot(v,w);
    float    D = a*c - b*b;        // always >= 0
    float    sc, tc;

    // compute the line parameters of the two closest points
    if (D < SMALL_NUM) {          // the lines are almost parallel
        sc = 0.0;
        tc = (b>c ? d/b : e/c);    // use the largest denominator
    }
    else {
        sc = (b*e - c*d) / D;
        tc = (a*e - b*d) / D;
    }

    // get the difference of the two closest points
    Vector   dP = w + (sc * u) - (tc * v);  // = L1(sc) - L2(tc)

    return norm(dP);   // return the closest distance
}
//===================================================================

// dist3D_Segment_to_Segment():
//    Input:  two 3D line segments S1 and S2
//    Return: the shortest distance between S1 and S2
// it also works in 2D
float
dist3D_Segment_to_Segment( Segment S1, Segment S2)
{
    Vector   u = S1.P1 - S1.P0;
    Vector   v = S2.P1 - S2.P0;
    Vector   w = S1.P0 - S2.P0;
    float    a = dot(u,u);         // always >= 0
    float    b = dot(u,v);
    float    c = dot(v,v);         // always >= 0
    float    d = dot(u,w);
    float    e = dot(v,w);
    float    D = a*c - b*b;        // always >= 0
    float    sc, sN, sD = D;       // sc = sN / sD, default sD = D >= 0
    float    tc, tN, tD = D;       // tc = tN / tD, default tD = D >= 0

    // compute the line parameters of the two closest points
    if (D < SMALL_NUM) { // the lines are almost parallel
        sN = 0.0;        // force using point P0 on segment S1
        sD = 1.0;        // to prevent possible division by 0.0 later
        tN = e;
        tD = c;
    }
    else {                // get the closest points on the infinite lines
        sN = (b*e - c*d);
        tN = (a*e - b*d);
        if (sN < 0.0) {       // sc < 0 => the s=0 edge is visible
            sN = 0.0;
            tN = e;
            tD = c;
        }
        else if (sN > sD) {  // sc > 1 => the s=1 edge is visible
            sN = sD;
            tN = e + b;
            tD = c;
        }
    }

    if (tN < 0.0) {           // tc < 0 => the t=0 edge is visible
        tN = 0.0;
        // recompute sc for this edge
        if (-d < 0.0)
            sN = 0.0;
        else if (-d > a)
            sN = sD;
        else {
            sN = -d;
            sD = a;
        }
    }
    else if (tN > tD) {      // tc > 1 => the t=1 edge is visible
        tN = tD;
        // recompute sc for this edge
        if ((-d + b) < 0.0)
            sN = 0;
        else if ((-d + b) > a)
            sN = sD;
        else {
            sN = (-d + b);
            sD = a;
        }
    }
    // finally do the division to get sc and tc
    sc = (abs(sN) < SMALL_NUM ? 0.0 : sN / sD);
    tc = (abs(tN) < SMALL_NUM ? 0.0 : tN / tD);

    // get the difference of the two closest points
    Vector   dP = w + (sc * u) - (tc * v);  // = S1(sc) - S2(tc)

    return norm(dP);   // return the closest distance
}

      static Point2D interseccionLineas(double x1, double y1, double x2,
double y2, double x3, double y3, double x4, double y4)
      {
            double px = ((x1 * y2 - y1 * x2) * (x3 - x4) - (x1 - x2) * (x3 *
y4 - y3 * x4)) / ((x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4));
            double py = ((x1 * y2 - y1 * x2) * (y3 - y4) - (y1 - y2) * (x3 *
y4 - y3 * x4)) / ((x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4));
            return new Point2D.Double(px, py);
      }

            static Point2D lineSegmentIntersection(
                  double Ax, double Ay,
                  double Bx, double By,
                  double Cx, double Cy,
                  double Dx, double Dy) {

                    double  distAB, theCos, theSin, newX, ABpos ;

                    //  Fail if either line segment is zero-length.
                    if (Ax==Bx && Ay==By || Cx==Dx && Cy==Dy) return null;

                    //  Fail if the segments share an end-point.
                    if (Ax==Cx && Ay==Cy || Bx==Cx && By==Cy
                    ||  Ax==Dx && Ay==Dy || Bx==Dx && By==Dy) {
                      return null; }

                    //  (1) Translate the system so that point A is on the
origin.
                    Bx-=Ax; By-=Ay;
                    Cx-=Ax; Cy-=Ay;
                    Dx-=Ax; Dy-=Ay;

                    //  Discover the length of segment A-B.
                    distAB=Math.sqrt(Bx*Bx+By*By);

                    //  (2) Rotate the system so that point B is on the
positive X axis.
                    theCos=Bx/distAB;
                    theSin=By/distAB;
                    newX=Cx*theCos+Cy*theSin;
                    Cy  =Cy*theCos-Cx*theSin; Cx=newX;
                    newX=Dx*theCos+Dy*theSin;
                    Dy  =Dy*theCos-Dx*theSin; Dx=newX;

                    //  Fail if segment C-D doesn't cross line A-B.
                    if (Cy<-EPSILON && Dy<-EPSILON || Cy>=-EPSILON && Dy>=-
EPSILON) return null;

                    //  (3) Discover the position of the intersection point
along line A-B.
                    ABpos=Dx+(Cx-Dx)*Dy/(Dy-Cy);

                    //  Fail if segment C-D crosses line A-B outside of
segment A-B.
                    if (ABpos<-EPSILON || ABpos>distAB-EPSILON) return null;

                    //  (4) Apply the discovered position to line A-B in the
original coordinate system.
                    double X=Ax+ABpos*theCos;
                    double Y=Ay+ABpos*theSin;

                    return new Point2D.Double(X, Y); }



public class SEC
{
      static class Circle
      {
          private Point2D p;
          private double r;

            public Circle()
          {
                  p = new Point2D.Double(0,0);
                  r = 0;
          }

            public Circle(Circle circle)
          {
                  p = new Point2D.Double(circle.p.getX(), circle.p.getY());
                  r = circle.r;
          }

            public Circle(Point2D center, double radius)
          {
                  p = new Point2D.Double(center.getX(), center.getY());
                  r = radius;
          }
```

```
            public Circle(Point2D center)
           {
                   p = new Point2D.Double(center.getX(), center.getY());
                   r = 0;
           }

           public static Point2D midPoint(Point2D a, Point2D b)
           {
                   return new Point2D.Double((a.getX() + b.getX()) / 2,
(a.getY() + b.getY()) / 2);
           }

           public Circle(Point2D p1, Point2D p2)
           {
                   p = midPoint(p1, p2);
                   r = p1.distance(p);
           }

           public Circle(Point2D p1, Point2D p2, Point2D p3)
           {
                   try
                   {
                         double x = (p3.getX()
                                     * p3.getX()
                                     * (p1.getY() - p2.getY())
                                     + (p1.getX() * p1.getX() + (p1.getY() -
p2.getY())
                                                 * (p1.getY() - p3.getY()))
                                     * (p2.getY() - p3.getY()) + p2.getX() *
p2.getX()
                                     * (-p1.getY() + p3.getY()))
                                     / (2 * (p3.getX() * (p1.getY() -
p2.getY()) + p1.getX()
                                                 * (p2.getY() - p3.getY()) +
p2.getX()
                                                 * (-p1.getY() + p3.getY())));
                         double y = (p2.getY() + p3.getY()) / 2
                                     - (p3.getX() - p2.getX()) / (p3.getY() -
p2.getY())
                                     * (x - (p2.getX() + p3.getX()) / 2);
                         p = new Point2D.Double(x, y);
                         r = p.distance(p1);
                         if(Double.isInfinite(r) || Double.isInfinite(x) ||
Double.isInfinite(y) || Double.isNaN(r) || Double.isNaN(x) || Double.isNaN(y))
                         {
                               r = 0;
```

```
                        p = new Point2D.Double(0, 0);
                  }
            }
            catch(Exception e)
            {
            }
    }

      public Point2D getCenter()
      {
            return p;
      }

      public double getRadius()
      {
            return r;
      }

      public void setCenter(Point2D center)
      {
            p.setLocation(center);
      }

      public void setRadius(double radius)
      {
            r = radius;
      }

    public void translate(Point2D newCenter)
    {
            p.setLocation(newCenter);
    }

    public void offset(double dr)
    {
            r += dr;
    }

    public void scale(double factor)
    {
            r *= factor;
    }

    public double getDiameter()
    {
            return (2 * r);
    }
```

```
          public double getCircumference()
          {
                  return (Math.PI * 2 * r);
          }

          public double getArea()
          {
                  return (Math.PI * r * r);
          }

            public int contain(Point2D point)
            {
                  int answer = 0;
                  double d = p.distance(point);
                  if (d > r)
                  {
                        answer = 1;       // The point is outside the circle
                  }
                  else if (d == r)
                  {
                        answer = 0;       // The point is on the circumference
of the circle
                  }
                  else
                  {
                        answer = -1;      // The point is inside the circle
                  }
                  return answer;
            }

          public boolean equals(Circle circle)
          {
                  return p.equals(circle.p) && (r == circle.r);
          }
      }


      public static Circle sec(ArrayList <Point2D> a)
      {
            Point2D[] p = new Point2D[a.size()];
            a.toArray(p);
            return findSec(a.size(), p, 0, new Point2D[3]);
      }

      // Compute the Smallest Enclosing Circle of the n points in p,
      // such that the m points in B lie on the boundary of the circle.
      public static Circle findSec(int n, Point2D[] p, int m, Point2D[] b)
      {
            Circle sec = new Circle();
            if(m == 1)
            {
                  sec = new Circle(b[0]);
            }
            else if(m == 2)
            {
                  sec = new Circle(b[0], b[1]);
            }
            else if(m == 3)
            {
                  return new Circle(b[0], b[1], b[2]);
            }
            for(int i=0; i < n; i++)
            {
                  if(sec.contain(p[i]) == 1)
                  {
                        b[m] = new Point2D.Double();
                        b[m].setLocation(p[i]);
                        sec = findSec(i, p, m + 1, b);
                  }
            }
            return sec;
      }
}

#include <cstdio>
#include <cstring>
#include <cmath>
#include <cstdlib>

#define REP(i,n) for((i)=0; (i)<(n); (i)++)
#define REPB(i,n) for((i)=(n)-1; (i)>=0; (i)--)

#define min(X,Y) ( ((X)<(Y)) ? (X) : (Y) )
#define max(X,Y) ( ((X)>(Y)) ? (X) : (Y) )

#define SIZE 50
#define DIMS 2
#define PI 3.141592
#define PREC 10E6
#define EQ(X,Y) ( ((int)round(PREC*X)) == ((int)round(PREC*Y)) )
#define LT(X,Y) ( ((int)round(PREC*X)) < ((int)round(PREC*Y)) )
#define GT(X,Y) ( ((int)round(PREC*X)) > ((int)round(PREC*Y)) )
```

```
int dims=DIMS;

void diff(double *r, double *a, double *b) {
    int i;
    REP(i,dims) {
        r[i] = a[i]-b[i];
    }
}

void sumv(double *r, double *a, double *b) {
    int i;
    REP(i,dims) {
        r[i]=a[i]+b[i];
    }
}

double norm(double *a) {
    double r=0;
    int i;
    REP(i,dims) {
        r += a[i]*a[i];
    }
    return sqrt(r);
}

double dotp(double *a, double *b) {
    double r=0;
    int i;
    REP(i,dims) {
        r += a[i]*b[i];
    }
    return r;
}

void multv(double *res, double *v, double scalar) {
    int i;
    REP(i,dims) {
        res[i] = v[i]*scalar;
    }
}

double vect_ang(double *a, double *b) {
    double r = acos( dotp(a,b)/(norm(a),norm(b)) );
    r = r*180/PI;
    r = round(100*r)/100.0;
    if (r < 0) r+=180;
    return r;
}

double dist(double *a, double *b) {
    double d[DIMS];
    diff(d,b,a);
    return norm(d);
}

void cp_v(double *dest, double *org) {
    int i;
    REP(i,dims) {
        dest[i] = org[i];
    }
}
```

6. Others and games

6.1. Binary search

Finding a value in a sorted sequence

In its simplest form, binary search is used to quickly find a value in a sorted sequence . We'll call the sought value the *target* value for clarity. Binary search maintains a contiguous subsequence of the starting sequence where the target value is surely located. This is called the *search space*. The search space is initially the entire sequence. At each step, the algorithm compares the median value in the search space to the target value. Based on the comparison and because the sequence is sorted, it can then eliminate half of the search space. By doing this repeatedly, it will eventually be left with a search space consisting of a single element, the target value.

If the target value wasnot present in the sequence, binary search would empty the search space entirely. This condition is easy to check and handle.

```
binary_search(A, target):
   lo = 1, hi = size(A)
   while lo <= hi:
      mid = lo + (hi-lo)/2     //Integer Division
      if A[mid] == target:
         return mid
      else if A[mid] < target:
         lo = mid+1
      else:
         hi = mid-1
  // target was not found
```
Complexity
Since each comparison binary search uses halves the search space, we can assert and easily prove that binary search will never use more than (in big-oh notation) $O(\log N)$ comparisons to find the target value. Note that this

assumes that we have random access to the sequence. Trying to use binary search on a container such as a linked list makes little sense.

Binary search in standard libraries

C++'s Standard Template Library implements binary search in algorithms lower_bound, upper_bound, binary_search and equal_range, depending exactly on what you need to do. Java has a built-in Arrays.binary_search method for arrays and the .NET Framework has Array.BinarySearch.

Taking it further: the main theorem

When you encounter a problem which you think could be solved by applying binary search, you need some way of proving it will work. I will now present another level of abstraction which will allow us to solve more problems, make proving binary search solutions very easy and also help implement them. Consider a predicate *p* defined over some ordered set *S* (the search space). The search space consists of candidate solutions to the problem, a predicate is a function which returns a boolean value, true or false . We use the predicate to verify if a candidate solution is legal (does not violate some constraint) according to the definition of the problem.

What we can call the *main theorem* states that **binary search can be used if and only if for all x in *S*, *p*(x) implies *p*(y) for all y > x**. This property is what we use when we discard the second half of the search space. It is equivalent to saying that ¬*p*(x) implies ¬*p*(y) for all y < x (the symbol ¬ denotes the logical not operator), which is what we use when we discard the first half of the search space

If the condition in the main theorem is satisfied, we can use binary search to find the smallest legal solution, i.e. the smallest x for which *p*(x) is true. The first part of devising a solution based on binary search is designing a predicate which can be evaluated and for which it makes sense to use binary search: we need to choose what the algorithm should find. We can have it find either the *first x for which p(x) is true* or the *last x for which p(x) is false*. The difference between the two is only slight.

Implementing the discrete algorithm

One important thing to remember before beginning to code is to settle on what the two numbers you maintain (lower and upper bound) mean. A likely answer is *a closed interval which surely contains the first x for which p(x) is true*. All of your code should then be directed at maintaining this invariant: it tells you how to properly move the bounds.

```
binary_search(lo, hi, p):
   while lo < hi:
      mid = lo + (hi-lo)/2
      if p(mid) == true:
         hi = mid
      else:
         lo = mid+1

   if p(lo) == false:
      complain               // p(x) is false for all x in S!

   return lo         // lo is the least x for which p(x) is true
```

6.2. Backtracking

Algunas veces es más fácil contar el número de elementos de un conjunto construyéndolos todos de hecho que aplicando sofisticados argumentos combinatorios. Por supuesto, que este número de elementos debe ser lo suficientemente pequeño para que la computación termine.
Los más recientes ordenadores personales tienen un ciclo de reloj de un giga hertzio, mas o menos lo que significa mil millones de operaciones por segundo. Hay que tener en cuenta que hacer algo interesante puede exigir unos pocos cientos de instrucciones o incluso más. Por consiguiente podemos aspirar a buscar unos pocos millones de elementos por segundo en las maquinas actuales.
Es muy importante percatarse de lo grande (o lo pequeño) que es un millón. Un millón de permutaciones son todas las ordenaciones diferentes de entre 10 y 11 objetos, pero no mas. Un millón de subconjuntos significa todas las combinaciones de alrededor de 20 objetos, pero no mas. La resolución de problemas significativamente grandes requiere recortar con mucho cuidado el espacio de búsqueda, para asegurarnos de que miramos solo los elementos que realmente interesan.
RASTREO EXHAUSTIVO POR RETROCESO
El rastreo exhaustivo por retroceso es un método sistemático para recorrer todas las configuraciones posibles de un espacio de búsqueda. Se trata de un algoritmo/Técnica general que habrá que adaptar para cada aplicación individual.
En el caso general, modelaremos la solución como un vector a = (a1,a2,a3,…an) donde cada elemento ai se escoge de un conjunto finito y ordenado Si. Tal vector pude representar un orden de colocación en la que ai contenga el i-esimo elemento de la permutación. O puede que el vector represente un subconjunto dado S, donde ai es verdadero si y solo si el i-esimo elemento de su universo esta en S. Mas a un, el vector podría incluso representar una sucesión de movimientos en un juego o un camino en un grafo, donde ai contiene el i-esimo evento de la sucesión.
En cada etapa del algoritmo de rastreo exhaustivo partimos de una solución parcial dada, digamos a = (a1,a2,…,ak) y tratamos de ampliarla añadiendo otro elemento al final. Tras extenderla, tenemos que comprobar si lo que tenemos hasta ahora es una solución – en cuyo caso deberíamos imprimirla, contarla, o hacer lo que queramos con ella. Si no lo es, tenemos que comprobar si esta solución parcial todavía es potencialmente extensible a una solución completa. Si lo es, repetir la etapa anterior y seguir. Si no, eliminamos el último elemento de a y probamos con otras posibilidades para esa posición, si es que existen.

Código general con el que se puede trabajar

```
Bool finished = FALSE;  // se han encontrado todas las soluciones
Backtrack(int a[], int k data input)
{
                  Int c[MAXCANDIDATES];   // candidatos a la siguiente
posición
      Int ncandidates;   //cuenta dichos candidatos
      Int i;
      If (is_a_solution(a,k,input))
                        Process_solution (a,k,input);
      Else
      {
                                    k = k+1;
                                    Construct_candidates
(a,k,input,c,&ncandidates);
                                    For (i=0;i<ncandidates;i++)
            {
                                    a[k] = c[i];
                  backtrack(a,k,input);
                  if (finished) return;  // final anticipado
            }
      }
}
```

En este algoritmo, la parte específica de cada aplicación consiste en tres subrutinas:
Is_a_solution (a,k,input): Esta función booleana comprueba si los k primeros elementos del vector a son una solución completa del problema concreto. El último argumento, input, nos permite pasar información general a la subrutina. Lo usaremos para especificar n, el tamaño de la solución buscada. Este dato tiene sentido cuando estamos construyendo permutaciones de longitud n o subconjuntos de n elementos, pero puede ser irrelevante cuando se construyen objetos de tamaño variable como la secuencia de movimientos de un juego. En estos casos se puede ignorar este último elemento.
Construct_candidates (a,k,input,c,ncandidates): Esta rutina llena un arreglo c con todos los posibles candidatos a la k-esima posición de a, en función del contenido de las k-1 primeras posiciones. El número de candidatos que nos devuelve la rutina en este arreglo se denomina ncandidates. De nuevo, se puede usar input para pasar información auxiliar, concretamente el tamaño de la solución buscada.
Process_solution (a,k,input): Esta rutina imprime, cuenta o procesa en alguna otra forma una solución complete, una vez que la hemos obtenido. Es evidente que ahora no se necesita el input auxiliar, puesto que k es ya el número de elementos de la solución.

El rastreo exhaustivo por retroceso garantiza la corrección, puesto que enumera todas las posibilidades. A demás garantiza la máxima eficiencia no visitando nunca mas de una vez un determinado estado.
Es interesante meditar sobre como la recursión produce un algoritmo, elegante y fácil de implementar, para el rastreo exhaustivo por retroceso. Puesto que con cada iteración ocupamos un nuevo arreglo c de nuevos candidatos, los subconjuntos de los candidatos aun no considerados como posible extensión en cada etapa no interferirán con los de cualquier otra.

6.3. Nim

Nim is a two-player mathematical game of strategy in which players take turns removing objects from distinct heaps. On each turn, a player must remove at least one object, and may remove any number of objects provided they all come from the same heap.
Variants of Nim have been played since ancient times. The game is said to have originated in China (it closely resembles the Chinese game of "Jianshizi", or "picking stones"), but the origin is uncertain; the earliest European references to Nim are from the beginning of the 16th century. Its current name was coined by Charles L. Bouton of Harvard University, who also developed the complete theory of the game in 1901, but the origins of the name were never fully explained. The name is probably derived from German *nimm* meaning "take", or the obsolete English verb *nim* of the same meaning. It should also be noted that rotating the word *NIM* by 180 degrees results in *WIN* (see Ambigram).
Nim is usually played as a *misère* game,[*citation needed*] in which the player to take the last object loses. Nim can also be played as a *normal play* game, which means that the person who makes the last move (i.e., who takes the last object) wins. This is called normal play because most games follow this convention, even though Nim usually does not.
Normal play Nim (or more precisely the system of nimbers) is fundamental to the Sprague-Grundy theorem, which essentially says that in normal play every impartial game is equivalent to a Nim heap that yields the same outcome when played in parallel with other normal play impartial games (see disjunctive sum).
While all normal play impartial games can be assigned a nim value, that is not the case under the misère convention. Only tame games can be played using the same strategy as misère nim.
A normal play game may start with heaps of 3, 4 and 5 objects:
In order to win always leave an even total number of 1's, 2's, and 4's.

```
Sizes of heaps  Moves
A B C

3 4 5           Player 1 takes 2 from A
1 4 5           Player 2 takes 3 from C
```

```
1 4 2           Player 1 takes 1 from B
1 3 2           Player 2 takes 1 from B
1 2 2           Player 1 takes entire A heap, leaving two 2s.
0 2 2           Player 2 takes 1 from B
0 1 2           Player 1 takes 1 from C leaving two 1s. (In misère play I
```
would take 2 from C leaving (0, 1, 0).)
```
0 1 1           Player 2 takes 1 from B
0 0 1           Player 1 takes entire C heap and wins.
```
Mathematical theory

Nim has been mathematically solved for any number of initial heaps and objects; that is, there is an easily calculated way to determine which player will win and what winning moves are open to that player. In a game that starts with heaps of 3, 4, and 5, the first player will win with optimal play, whether the misère or normal play convention is followed.

The key to the theory of the game is the binary digital sum of the heap sizes, that is, the sum (in binary) neglecting all carries from one digit to another. This operation is also known as "exclusive or" (xor) or "vector addition over GF(2)". Within combinatorial game theory it is usually called the nim-sum, as will be done here. The nim-sum of *x* and *y* is written *x* ⊕ *y* to distinguish it from the ordinary sum, *x* + *y*. An example of the calculation with heaps of size 3, 4, and 5 is as follows:

Binary Decimal

```
  0112    310    Heap A
  1002    410    Heap B
  1012    510    Heap C
  ---
  0102    210    The nim-sum of heaps A, B, and C, 3 ⊕ 4 ⊕ 5 = 2
```

An equivalent procedure, which is often easier to perform mentally, is to express the heap sizes as sums of distinct powers of 2, cancel pairs of equal powers, and then add what's left:

```
3 = 0 + 2 + 1 =     2   1      Heap A
4 = 4 + 0 + 0 = 4              Heap B
5 = 4 + 0 + 1 = 4       1      Heap C
---
2 =                 2          What's left after canceling 1s and 4s
```

In normal play, the winning strategy is to finish every move with a Nim-sum of 0. This is always possible if the Nim-sum is not zero before the move. If the Nim-sum is zero, then the next player will lose if the other player does not make a mistake. To find out which move to make, let X be the Nim-sum of all the heap sizes. Take the Nim-sum of each of the heap sizes with X, and find a heap whose size decreases. The winning strategy is to play in such a heap, reducing that heap to the Nim-sum of its original size with X. In the example above, taking the Nim-sum of the sizes is X = 3 ⊕ 4 ⊕ 5 = 2. The Nim-sums of the heap sizes A=3, B=4, and C=5 with X=2 are

```
    A ⊕ X = 3 ⊕ 2 = 1 [Since (011) ⊕ (010) = 001 ]
    B ⊕ X = 4 ⊕ 2 = 6
    C ⊕ X = 5 ⊕ 2 = 7
```

The only heap that is reduced is heap A, so the winning move is to reduce the size of heap A to 1 (by removing two objects).
As a particular simple case, if there are only two heaps left, the strategy is to reduce the number of objects in the bigger heap to make the heaps equal. After that, no matter what move your opponent makes, you can make the same move on the other heap, guaranteeing that you take the last object.
When played as a misère game, Nim strategy is different only when the normal play move would leave no heap of size 2 or larger. In that case, the correct move is to leave an odd number of heaps of size 1 (in normal play, the correct move would be to leave an even number of such heaps).
In a misère game with heaps of sizes 3, 4 and 5, the strategy would be applied like this:

```
A B C Nim-sum

3 4 5 0102=210   I take 2 from A, leaving a sum of 000, so I will win.
1 4 5 0002=010   You take 2 from C
1 4 3 1102=610   I take 2 from B
1 2 3 0002=010   You take 1 from C
1 2 2 0012=110   I take 1 from A
0 2 2 0002=010   You take 1 from C
0 2 1 0112=310   The normal play strategy would be to take 1 from B, leaving
an even number (2)
                 heaps of size 1.  For misère play, I take the entire B heap,
to leave an odd
                 number (1) of heaps of size 1.
0 0 1 0012=110   You take 1 from C, and lose.
```

The previous strategy for a misère game can be easily implemented in Python.

```python
def nim_misere(heaps):
    """Computes next move for Nim in a misère game, returns tuple
(chosen_heap, nb_remove)"""
    X = reduce(lambda x,y: x^y, heaps)
    if X == 0: # Will lose unless all non-empty heaps have size one
        for i, heap in enumerate(heaps):
            if heap > 0: # Empty any (non-empty) heap
                chosen_heap, nb_remove = i, heap
                break
    else:
        sums = [t^X < t for t in heaps]
        chosen_heap = sums.index(True)
        nb_remove = heaps[chosen_heap] - (heaps[chosen_heap]^X)
        heaps_twomore = 0
        for i, heap in enumerate(heaps):
            n = heap-nb_remove if chosen_heap == i else heap
```

```
                  if n>1: heaps_twomore += 1
            # If move leaves no heap of size 2 or larger, leave an odd number
of heaps of size 1
            if heaps_twomore == 0:
                  chosen_heap = heaps.index(max(heaps))
                  heaps_one = sum(t==1 for t in heaps)
                  # even? make it odd; odd? keep it odd
                  nb_remove = heaps[chosen_heap]-1 if heaps_one%2==0 else
heaps[chosen_heap]
      return chosen_heap, nb_remove
```

6.4. Counting inversions

Inversion Count for an array indicates – how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum.
Formally speaking, two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j
Example:
The sequence 2, 4, 1, 3, 5 has three inversions (2, 1), (4, 1), (4, 3).

```
/*     This is a program to count the number of inversions in a
 *  permutation of integers.  Actually this will process several
 *  arrays.
 *
 *  The input should be in the following format:
 *    L
 *    n_1 n_2 .....  n_L
 *    L
 *    n_1 n_2 .....  n_L
 *    ....
 *    ....
 *    0
 *
 *  Each list consists of L integers (L can vary from list to list).
 *  The elements of each list are contained in a single line.
 *
 *  After the final list, the input is terminated with a 0 (i.e. L=0)
 *  to indicate the end of the input.
 *
 *  Note:  I'm assuming that the integers in each list are distinct (but
 *  they need not necessarily be the consecutive integers 1, 2, ..., n).
 *       Results might be unpredictable (depending upon how you implement
 *  your methods) if there are duplicated items in a list.
 *
 *  It's easiest if you redirect input from a file (ask me if you
 *  don't know how to do this).  For example you can try this command:
 *     % java Inversions <input
 *
 *  RAM   20 February 2009
 *
 *
 *        ***Running time***
 * Since this algorithm is basically a modified MergeSort (see
 *  my remarks in the "countInversions" procedure below), this algorithm
 *  runs in time O(n log n) for a list of length n.
 *
*/
import java.io.*;
import java.util.*;

class Inversions
{
    public static void main (String args[])  // entry point from OS
    {
        Scanner s, ls;
        int L, listNum = 1;

        s = new Scanner(System.in);  // create new input Scanner

        ls = new Scanner(s.nextLine());
        L = ls.nextInt();

        while(L > 0)    /*  While there's more data to process...  */
           {
              /*  Create an array to hold the integers  */
              int nums[] = new int[L];

              /*  Read the integers  */
              ls = new Scanner(s.nextLine());
              for (int j = 0; j < L; j++)
                    nums[j] = ls.nextInt();

              /*  Compute the number of inversions, and print it out  */
              System.out.print( "List " + listNum + " has " );
              System.out.println ( countInversions(nums) + " inversions.");

              /*  Read in the next value of L  */
              listNum++;
              ls = new Scanner(s.nextLine());
              L = ls.nextInt();
           }
```

```
}  /*  end of "main" procedure  */


      public static int countInversions(int nums[])
      /*  This function will count the number of inversions in an
          array of numbers.  (Recall that an inversion is a pair
          of numbers that appear out of numerical order in the list.

          We use a modified version of the MergeSort algorithm to
          do this, so it's a recursive function.  We split the
          list into two (almost) equal parts, recursively count
          the number of inversions in each part, and then count
          inversions caused by one element from each part of
          the list.

          The merging is done is a separate procedure given below,
          in order to simplify the presentation of the algorithm
          here.

          Note:  I am assuming that the integers are distinct, but
          they need *not* be integers { 1, 2, ..., n} for some n.

      */
      {
          int mid = nums.length/2, k;
          int countLeft, countRight, countMerge;

        /*  If the list is small, there's nothing to do.  */
          if (nums.length <= 1)
             return 0;

        /*  Otherwise, we create new arrays and split the list into
            two (almost) equal parts.
        */
          int left[] = new int[mid];
          int right[] = new int[nums.length - mid];

          for (k = 0; k < mid; k++)
              left[k] = nums[k];
          for (k = 0; k < nums.length - mid; k++)
              right[k] = nums[mid+k];

        /*  Recursively count the inversions in each part.
        */
          countLeft = countInversions (left);
          countRight = countInversions (right);
```

```
            /*  Now merge the two sublists together, and count the
                inversions caused by pairs of elements, one from
                each half of the original list.
            */
              int result[] = new int[nums.length];
              countMerge = mergeAndCount (left, right, result);

            /*  Finally, put the resulting list back into the original one.
                This is necessary for the recursive calls to work correctly.
            */
              for (k = 0; k < nums.length; k++)
                  nums[k] = result[k];

            /*  Return the sum of the values computed to
                get the total number of inversions for the list.
            */
              return (countLeft + countRight + countMerge);

          }  /*  end of "countInversions" procedure  */


        public static int mergeAndCount (int left[], int right[], int
result[])
        /*  This procudure will merge the two lists, and count the number of
            inversions caused by the elements in the "right" list that are
            less than elements in the "left" list.
        */
        {
            int a = 0, b = 0, count = 0, i, k=0;

            while ( ( a < left.length) && (b < right.length) )
              {
                 if ( left[a] <= right[b] )
                       result [k] = left[a++];
                 else       /*  You have found (a number of) inversions here.
*/
                    {
                       result [k] = right[b++];
                       count += left.length - a;
                    }
                 k++;
              }

            if ( a == left.length )
               for ( i = b; i < right.length; i++)
                   result [k++] = right[i];
            else
               for ( i = a; i < left.length; i++)
                   result [k++] = left[i];

            return count;
        }

}  /*  end of "Inversions" program  */
```

6.4. Poker

```
public class Poker
{
      static class Carta implements Comparable <Carta>
      {
            int valor;
            int pinta;

            @Override
            public int compareTo(Carta otra)
            {
                  if(valor < otra.valor)
                        return 1;
                  else if(valor > otra.valor)
                        return -1;
                  else
                        return 0;
            }
      }

      static Carta generarCarta(char [] carta)
      {
            int valor;
            try
            {
                  valor = Integer.parseInt(carta[0] + "");
            }
            catch(Exception e)
            {
                  if(carta[0] == 'T')
                  {
                        valor = 10;
                  }
                  else if(carta[0] == 'J')
                  {
                        valor = 11;
                  }
                  else if(carta[0] == 'Q')
                  {
                        valor = 12;
                  }
                  else if(carta[0] == 'K')
                  {
                        valor = 13;
                  }
                  else
                  {
                        valor = 14;
                  }
            }
            int pinta;
            switch(carta[1])
            {
                  case 'H': pinta = 1; break;
                  case 'D': pinta = 2; break;
                  case 'S': pinta = 3; break;
                  default: pinta = 4; break;
            }
            Carta c = new Carta();
            c.pinta = pinta;
            c.valor = valor;
            return c;
      }


      static long darValor(Carta [] mano)
      {
            Arrays.sort(mano);
            ArrayList <Integer> repetidas = new ArrayList <Integer> ();
            int tipo = 0;
            for(int i = 0; i < 5; i++)
                  repetidas.add(1);
            for(int i = 0; i < 5; i++)
            {
                  for(int j = 0; j < 5; j++)
                  {
                        if(j != i)
                        {
                              if (mano[i].compareTo(mano[j]) == 0)
                              {
                                    repetidas.set(i, repetidas.get(i) + 1);
                              }
                      }
              }
      }
      ArrayList <Integer> manoN = new ArrayList <Integer> ();
      if(repetidas.contains(4))
      {
              tipo = 7;
              for(int i = 0; i < 4; i++)
                      manoN.add(mano[repetidas.indexOf(4)].valor);
              manoN.add(mano[repetidas.indexOf(1)].valor);
      }
      else if(repetidas.contains(3) && repetidas.contains(2))
      {
              tipo = 6;
              for(int i = 0; i < 3; i++)
                      manoN.add(mano[repetidas.indexOf(3)].valor);
              for(int i = 0; i < 2; i++)
                      manoN.add(mano[repetidas.indexOf(2)].valor);
      }
      else if(repetidas.contains(3))
      {
              tipo = 3;
              for(int i = 0; i < 3; i++)
                      manoN.add(mano[repetidas.indexOf(3)].valor);
              manoN.add(mano[repetidas.indexOf(1)].valor);
              manoN.add(mano[repetidas.lastIndexOf(1)].valor);
      }
      else if(repetidas.contains(2))
      {
              if(repetidas.indexOf(2) + 1 != repetidas.lastIndexOf(2))
              {
                      tipo = 2;
                      for(int i = 0; i < 2; i++)
                              manoN.add(mano[repetidas.indexOf(2)].valor);
                      for(int i = 0; i < 2; i++)
                              manoN.add(mano[repetidas.lastIndexOf(2)].valor);
                      manoN.add(mano[repetidas.indexOf(1)].valor);
              }
              else
              {
                      tipo = 1;
                      for(int i = 0; i < 2; i++)
                              manoN.add(mano[repetidas.indexOf(2)].valor);
                      for(int i = 0; i < 5; i++)
                      {
                              if(repetidas.get(i) == 1)
```

```
                              {
                                    manoN.add(mano[i].valor);
                              }
                        }
                  }
            }
            else
            {
                  for(Carta c : mano)
                  {
                        manoN.add(c.valor);
                  }
                  boolean color = true;
                  int pinta = mano[0].pinta;
                  for(Carta c : mano)
                  {
                        if(c.pinta != pinta)
                              color = false;
                  }
                  boolean escalera = true;
                  int valor = mano[0].valor;
                  for(Carta c : mano)
                  {
                        if(c.valor != valor)
                              escalera = false;
                        else
                              valor--;
                  }
                  if(escalera && color)
                        tipo = 8;
                  else if(color)
                        tipo = 5;
                  else if(escalera)
                        tipo = 4;
            }
            long valorMano = tipo * 10000000000L;
            valorMano += manoN.get(0) * 100000000;
            valorMano += manoN.get(1) * 1000000;
            valorMano += manoN.get(2) * 10000;
            valorMano += manoN.get(3) * 100;
            valorMano += manoN.get(4);
            return valorMano;
      }
}
```

6.5. Sudoku

```
public class Sudoku
{
	static final int tS = 4;
	static final int tS2 = 16;
	static int visitado = 1 << (tS2 + 1);

	static void marcar(int i, int j, int n)
	{
		int mascara = 1 << n;
		sudoku[i][j] = mascara + visitado;
		mascara = ~mascara;
		int iMenor = (i / tS) * tS;
		int iMayor = iMenor + tS;
		int jMenor = (j / tS) * tS;
		int jMayor = jMenor + tS;
		for(int a = 0; a < tS2; a++)
		{
			if(a != j)
				sudoku[i][a] &= mascara;
			if(a != i)
				sudoku[a][j] &= mascara;
		}
		for(int a = iMenor; a < iMayor; a++)
			for(int b = jMenor; b < jMayor; b++)
				if(a != i || b != j)
					sudoku[a][b] &= mascara;
	}

	static long tiempoDentro = 0;
	static int idMascara = 0;
	static int[][][] zonas = new int[tS2 * 3][tS2][2];
	static int[] actuales = new int[tS2];

	public static void limpiarZona(int[][] zona, int[] actuales, int[]
grupos, int[] tamGrupos, int[][] sudoku, int[] visitados)
	{
		while(true)
		{
			int iActual = 0;
			forExterno:
			for(int i = 0; i <= tS2; i++)
			{
				if(tamGrupos[i] == -1)
					break;
```

```
                        for(int j = 0; j < tamGrupos[i]; j++)
                        {
                              int indice = grupos[iActual + j];
                              int valor = sudoku[zona[indice][0]]
[zona[indice][1]];
                              if(valor >= visitado || tamGrupos[i] == 1)
                              {
                                    tamGrupos[i]--;
                                    for(int k = iActual + j; k < tS2 - 1; k++)
                                          grupos[k] = grupos[k + 1];
                                    if(tamGrupos[i] == 0)
                                    {
                                          for(int k = i; k < tS2; k++)
                                                tamGrupos[k] = tamGrupos[k +
1];

                                          i--;
                                          continue forExterno;
                                    }
                                    else
                                    {
                                          j--;
                                          continue;
                                    }
                              }
                        }
                        iActual += tamGrupos[i];
                  }
                  break;
            }
            idMascara++;
            while(true)
            {
                  int iActual = 0;
                  forExterno:
                  for(int i = 0; i <= tS2; i++)
                  {
                        int tamActual = tamGrupos[i];
                        if(tamActual == -1)
                              break;
                        int mascaraActual = 1 << tamActual;
                        mascaraActual--;
                        for(int mascara = 1; mascara < mascaraActual;
mascara++)
                        {
                              int temp = mascara;
                              int m = 0;
                              for(int j = 0; j < tamActual; j++)
                              {
                                    if((temp & 1) == 1)
                                          m |= sudoku[zona[grupos[iActual +
j]][0]][zona[grupos[iActual + j]][1]];
                                    temp >>= 1;
                              }
                              int tamCuenta = Integer.bitCount(m);
                              if(tamCuenta == tamActual)
                                    continue;
                              if(visitados[m] == idMascara)
                                    continue;
                              temp = ~m;
                              int cuenta = 0;
                              for(int j = 0; j < tamActual; j++)
                              {
                                    int valor = sudoku[zona[grupos[iActual +
j]][0]][zona[grupos[iActual + j]][1]];
                                    if((valor & temp) == 0)
                                          cuenta++;
                              }
                              if(cuenta == tamCuenta)
                              {
                                    int posA = 0;
                                    for(int j = 0; j < tamActual; j++)
                                    {
                                          int valor =
sudoku[zona[grupos[iActual + j]][0]][zona[grupos[iActual + j]][1]];
                                          if((valor & temp) == 0)
                                                actuales[posA++] =
grupos[iActual + j];
                                    }
                                    for(int j = 0; j < tamActual; j++)
                                    {
                                          int valor =
sudoku[zona[grupos[iActual + j]][0]][zona[grupos[iActual + j]][1]];
                                          if((valor & temp) != 0)
                                          {
                                                actuales[posA++] =
grupos[iActual + j];
                                                sudoku[zona[grupos[iActual +
j]][0]][zona[grupos[iActual + j]][1]] &= temp;
                                          }
                                    }
                                    for(int k = 0; k < tamActual; k++)
                                          grupos[iActual + k] = actuales[k];
                                    tamGrupos[i] = cuenta;
                                    for(int k = tS2; k > i; k--)
                                    tamGrupos[k] = tamGrupos[k - 1];
                                tamGrupos[i + 1]  = tamActual - cuenta;
                                i--;
                                continue forExterno;
                            }
                            visitados[m] = idMascara;
                        }
                        iActual += tamActual;
                    }
                    break;
                }
            }

            static int[] visitados = new int[1 << (tS2 + 1)];

            public static void limpiar(int[][] sudoku, int[][] grupos, int[][]
tamGrupos)
            {
                for(int i = 0; i < tS2 * 3; i++)
                    limpiarZona(zonas[i], actuales, grupos[i], tamGrupos[i],
sudoku, visitados);
            }

            static int[][] sudoku = new int[tS2][tS2];
            static int[][] grupos = new int[tS2 * 3][tS2];
            static int[][] tamGrupos = new int[tS2 * 3][tS2 + 1];
            static int[][][] backtrack = new int[tS2 * tS2][tS2][tS2];
            static int[][][] backtrackGrupos = new int[tS2 * tS2][tS2 * 3][tS2];
            static int[][][] backtrackTamGrupos = new int[tS2 * tS2][tS2 * 3][tS2 +
1];

            static boolean backtrack(int altura, int[][] sudoku, int[][] grupos,
int[][] tamGrupos)
            {
                int sumAnterior = 0;
                int sumMejor = Integer.MAX_VALUE;
                boolean termino = true;
                int iMejor = 0;
                int jMejor = 0;
                while(sumMejor > 1 && sumAnterior != sumMejor)
                {
                    sumAnterior = sumMejor;
                    iMejor = 0;
                    jMejor = 0;
                    termino = true;
                    sumMejor = Integer.MAX_VALUE;
                    for(int i = 0; i < tS2; i++)
            for(int j = 0; j < tS2; j++)
            {
                  int pos = Integer.bitCount(sudoku[i][j]);
                  if(sudoku[i][j] == 0)
                        return false;
                  if(sudoku[i][j] <= todos)
                        termino = false;
                  if(sudoku[i][j] <= todos && pos <= sumMejor)
                  {
                        sumMejor = pos;
                        iMejor = i;
                        jMejor = j;
                  }
            }
      if(sumMejor != 1)
            limpiar(sudoku, grupos, tamGrupos);
}
if(termino)
{
      for(int i = 0; i < tS2; i++)
      {
            for(int j = 0; j < tS2; j++)
            {
                  int temp = sudoku[i][j];
                  for(int k = 0; k < tS2; k++)
                  {
                        if((temp & 1) == 1)
                              System.out.print((char) ('A' + k));
                        temp >>= 1;
                  }
            }
            System.out.println();
      }
      return true;
}
if(sumMejor == 1)
{
      int pos = 0;
      int temp = sudoku[iMejor][jMejor];
      for(int i = 0; i < tS2; i++)
      {
            if((temp & 1) == 1)
                  pos = i;
            temp >>= 1;
      }
      marcar(iMejor, jMejor, pos);
```

```
                    return backtrack(altura + 1, sudoku, grupos, tamGrupos);

              }
              else
              {
                    for(int j = 0; j < tS2; j++)
                          for(int k = 0; k < tS2; k++)
                                backtrack[altura][j][k] = sudoku[j][k];
                    for(int j = 0; j < tS2 * 3; j++)
                          for(int k = 0; k < tS2; k++)
                                backtrackGrupos[altura][j][k] = grupos[j][k];
                    for(int j = 0; j < tS2 * 3; j++)
                          for(int k = 0; k <= tS2; k++)
                                backtrackTamGrupos[altura][j][k] = tamGrupos[j]
[k];
                    int temp = sudoku[iMejor][jMejor];
                    for(int i = 0; i < tS2; i++)
                    {
                          if((temp & 1) == 1)
                          {
                                int pos = i;
                                for(int j = 0; j < tS2; j++)
                                      for(int k = 0; k < tS2; k++)
                                            sudoku[j][k] = backtrack[altura][j]
[k];
                                for(int j = 0; j < tS2 * 3; j++)
                                      for(int k = 0; k < tS2; k++)
                                            grupos[j][k] =
backtrackGrupos[altura][j][k];
                                for(int j = 0; j < tS2 * 3; j++)
                                      for(int k = 0; k <= tS2; k++)
                                            tamGrupos[j][k] =
backtrackTamGrupos[altura][j][k];
                                marcar(iMejor, jMejor, pos);
                                if(backtrack(altura + 1, sudoku, grupos,
tamGrupos))
                                      return true;
                          }
                          temp >>= 1;
                    }
                    return false;
              }
        }


        static void generarZonas()
        {
              for(int i = 0; i < tS2; i++)
              {
                    for(int j = 0; j < tS2; j++)
                    {
                          zonas[i][j][0] = i;
                          zonas[i][j][1] = j;
                    }
              }
              for(int i = 0; i < tS2; i++)
              {
                    for(int j = 0; j < tS2; j++)
                    {
                          zonas[i + tS2][j][0] = j;
                          zonas[i + tS2][j][1] = i;
                    }
              }
              int cuenta = 0;
              for(int i = 0; i < tS; i++)
                    for(int j = 0; j < tS; j++)
                    {
                          int ia = i * tS;
                          int ib = ia + tS;
                          int ja = j * tS;
                          int jb = ja + tS;
                          int actual = 0;
                          for(int a = ia; a < ib; a++)
                                for(int b = ja; b < jb; b++)
                                {
                                      zonas[cuenta + tS2 * 2][actual][0] = a;
                                      zonas[cuenta + tS2 * 2][actual++][1] = b;
                                }
                          cuenta++;
                    }
        }

        static int todos = (1 << tS2) - 1;
  }
```

6.6. Chess knights

```
public class ChessKnights
{
    static final int RNG = 100;
      static final int ZERO = 50;
      static int[][] dir = new int[][]{{-2, -1}, {-2, 1}, {-1, -2}, {-1, 2},
      {1, -2}, {1, 2}, {2, -1}, {2, 1}};

      static int[][] d = new int[RNG][RNG];
      static int[][] q = new int[RNG*RNG][2];
      static int tail;

      static void add(int r1, int c1, int d1) {
        try
        {
              if(d[r1][c1] == -1) {
                d[r1][c1] = d1;
                q[tail][0] = r1;
                q[tail++][1] = c1;
              }
        }
        catch(Exception e)
        {

        }
      }

      static long moo(long x, long y) {
        x = Math.abs(x);
        y = Math.abs(y);
        if(x < y) { long temp = x; x = y; y = temp; }
        if(x < 20 && y < 20) return d[(int) (ZERO + x)][(int) (ZERO + y)];
        if(y == 0) {
          long k = Math.max(0, (x - 10) / 4);
          return k * 2 + moo(x - k * 4, 0);
        }
        else if(x < y + 3) {
          long k = Math.max(0, (y - 10) / 3);
          return k * 2 + moo(x - k * 3, y - k * 3);
        }
        else {
          long k = Math.min(Math.min(x - y, y), Math.max(x / 2, y) - 5);
          return k + moo(x - k * 2, y - k);
        }
      }

      // Entrega la menor cantidad de saltos de caballo en un tablero infinito
para
      // ir desde x0, y0 hasta x1, y1
      static long mejor(long x0, long y0, long x1, long y1)
      {
            x1 -= x0;
            y1 -= y0;
            return moo(x1, y1);
        }
}
```

7. Data structures

7.1. Binary heap

Si por alguna razón se necesitan metodos del binary-heap que no esten aca, en el codigo fuente de Java la clase java.util.PriorityQueue es una implementación del binary heap, solo que no tiene el metodo decreaseKey y por eso no la usamos aca (en la maratón dan acceso al codigo fuente de Java).

```
static class PriorityQueue
{
        private int currentSize;
        private PriorityQueueItem[] array;

        public PriorityQueue(int capacidad)
        {
              currentSize = 0;
              array = new PriorityQueueItem[capacidad + 1];
        }

        public int insert(PriorityQueueItem x)
        {
              if(currentSize + 1 == array.length)
```

```
            doubleArray();
        int hole = ++currentSize;
        array[0] = x;
        array[0].setPosition(0);
        for(; x.compareTo(array[hole / 2]) < 0; hole /= 2)
        {
            array[hole] = array[hole / 2];
            array[hole].setPosition(hole);
        }
        array[hole] = x;
        array[hole].setPosition(hole);
        return hole;
    }

    public void decreaseKey(PriorityQueueItem newVal)
    {
        int hole = newVal.position();
        array[0] = newVal;
        array[0].setPosition(0);
        for(; newVal.compareTo(array[hole / 2]) < 0; hole /= 2)
        {
            array[hole] = array[hole / 2];
            array[hole].setPosition(hole);
        }
        array[hole] = newVal;
        array[hole].setPosition(hole);
    }

    public PriorityQueueItem deleteMin()
    {
        PriorityQueueItem minItem = array[1];
        array[1] = array[currentSize--];
        array[1].setPosition(1);
        percolateDown(1);
        return minItem;
    }

    public boolean isEmpty()
   {
        return currentSize == 0;
    }

    private void percolateDown(int hole)
   {
        int child;
        PriorityQueueItem tmp = array[hole];
        for(; hole * 2 <= currentSize; hole = child)
```

```
            {
                  child = hole * 2;
                  if(child != currentSize && array[child +
1].compareTo(array[child]) < 0)
                        child++;
                  if(array[child].compareTo(tmp) < 0)
                  {
                        array[hole] = array[child];
                        array[hole].setPosition(hole);
                  }
                  else
                        break;
            }
            array[hole] = tmp;
            array[hole].setPosition(hole);
      }

      private void doubleArray()
      {
            PriorityQueueItem[] newArray;
            newArray = new PriorityQueueItem[array.length * 2];
            for( int i = 0; i < array.length; i++ )
                  newArray[i] = array[i];
            array = newArray;
      }
}

static abstract class PriorityQueueItem
{
      int dist;
      PriorityQueueItem previous;
      private int position;

      public int position()
      {
            return position;
      }

      public void setPosition(int newPosition)
      {
            position = newPosition;
      }

      public int compareTo(PriorityQueueItem o)
      {
            return new Integer(dist).compareTo(o.dist);
      }
```

```
}
```

7.2. Fibonnaci heap

Notese que aunque la Fibonnaci Heap es asimptoticamente más rápida que la Binary Heap, en la realidad el overhead de la Fibonnaci Heap es muy alto y por tanto solo vale la pena usarla cuando E (número de conexiones) es muchisimo mayor que V (número de nodos).

```
static class PriorityQueue {
   private PriorityQueueItem min;
   private int n;

   private void consolidate() {
       PriorityQueueItem[] A = new PriorityQueueItem[45];
       PriorityQueueItem start = min;
       PriorityQueueItem w = min;
       do {
           PriorityQueueItem x = w;
           PriorityQueueItem nextW = w.right;
           int d = x.degree;
           while (A[d] != null) {
               PriorityQueueItem y = A[d];
               if (x.dist > y.dist) {
                   PriorityQueueItem temp = y;
                   y = x;
                   x = temp;
               }
               if (y == start) {
                   start = start.right;
               }
               if (y == nextW) {
                   nextW = nextW.right;
               }
               y.link(x);
               A[d] = null;
               d++;
           }
           A[d] = x;
           w = nextW;
       } while (w != start);
       min = start;
       for (PriorityQueueItem a : A) {
           if (a != null && a.dist < min.dist) {
               min = a;
           }
       }
   }

   public void decreaseKey(PriorityQueueItem x) {
       decreaseKey(x, x.dist, false);
   }

   private void decreaseKey(PriorityQueueItem x, int k, boolean delete) {
       PriorityQueueItem y = x.parent;
       if (y != null && (delete || k < y.dist)) {
           y.cut(x, min);
           y.cascadingCut(min);
       }
       if (delete || k < min.dist) {
           min = x;
       }
   }

   public boolean isEmpty() {
       return min == null;
   }

   public void insert(PriorityQueueItem PriorityQueueItem) {
       if (min != null) {
           PriorityQueueItem.right = min;
           PriorityQueueItem.left = min.left;
           min.left = PriorityQueueItem;
           PriorityQueueItem.left.right = PriorityQueueItem;
           if (PriorityQueueItem.dist < min.dist) {
               min = PriorityQueueItem;
           }
       } else {
           min = PriorityQueueItem;
       }
       n++;
   }

   public PriorityQueueItem deleteMin() {
       PriorityQueueItem z = min;
       if (z == null) {
           return null;
       }
       if (z.child != null) {
           z.child.parent = null;
           for (PriorityQueueItem x = z.child.right; x != z.child; x =
x.right) {
               x.parent = null;
           }
           PriorityQueueItem minleft = min.left;
           PriorityQueueItem zchildleft = z.child.left;
           min.left = zchildleft;
           zchildleft.right = min;
           z.child.left = minleft;
           minleft.right = z.child;
       }
       z.left.right = z.right;
       z.right.left = z.left;
       if (z == z.right) {
           min = null;
       } else {
           min = z.right;
           consolidate();
       }
       n--;
       return z;
   }
}


static abstract class PriorityQueueItem {
   int dist;
   PriorityQueueItem previous;
   private PriorityQueueItem parent;
   private PriorityQueueItem child;
   private PriorityQueueItem right;
   private PriorityQueueItem left;
   private int degree;
   private boolean mark;

   public PriorityQueueItem() {
       right = this;
       left = this;
   }

   public void cascadingCut(PriorityQueueItem min) {
       PriorityQueueItem z = parent;
       if (z != null) {
           if (mark) {
               z.cut(this, min);
               z.cascadingCut(min);
           } else {
               mark = true;
           }
       }
   }

   public void cut(PriorityQueueItem x, PriorityQueueItem min) {
       x.left.right = x.right;
       x.right.left = x.left;
       degree--;
       if (degree == 0) {
           child = null;
       } else if (child == x) {
           child = x.right;
       }
       x.right = min;
       x.left = min.left;
       min.left = x;
       x.left.right = x;
       x.parent = null;
       x.mark = false;
   }

   public void link(PriorityQueueItem parent) {
       left.right = right;
       right.left = left;
       this.parent = parent;
       if (parent.child == null) {
           parent.child = this;
           right = this;
           left = this;
       } else {
           left = parent.child;
           right = parent.child.right;
           parent.child.right = this;
           right.left = this;
       }
       parent.degree++;
       mark = false;
   }
}
```

7.3. Union Find

Union Find is an algorithm which uses a disjoint-set data structure to solve the following problem: Say we have some number of items. We are allowed to merge any two items to consider them equal (where equality here obeys all of the properties of an Equivalence Relation). At any point, we are allowed to ask whether two items are considered equal or not.

Definition

Basically a Union Find data structure implements two functions:

- union(A, B) - merge A's set with B's set
- find(A) - finds what set A belongs to

Usage

This is a common way to find connectivity between nodes, or to find connected components.

Naive Implementation

The most naive method involves using multiple lists. To merge two elements, we will find which lists they are in, and then append one of them to the other. To check for equality, we will again find which lists the two elements are in, and check that they are the same list. Each operation in this case takes $O(n)$ time in the worst case, for a total of $O(n2)$ runtime.

Consider a different implementation of the naive method. Each element will be part of a node, which will contain the element and a pointer to the node's parent, if it exists. Thus the structure ends up looking like a forest of trees, where each node points to its parent. If two elements are in the same tree, then they are considered equal. To find out if they are in the same tree, we note that this can be uniquely determined by finding the root of the tree, which is considered the "representative element" of this tree. So to find the representative element for some arbitrary element, get the appropriate node and follow parent pointers until we can no longer. To merge two of these trees, take one of the representative elements and set its parent to be the other.

In the following, let a find operation be the operation of getting the representative element.

This implementation still takes $O(n)$ per operation, but allows for improvements.

Implementation Optimizations

The first optimization is known as *union by rank*. We augment the node structure by adding a new parameter rank, which is initialized to 0 for all elements. When we want to merge two elements, we will first find the representative elements. If they are different, then check their ranks. If one is greater than the other, then set the parent of the node with smaller rank to be the node with greater rank. Otherwise, arbitrarily pick one node. Set its parent to be the other, and increment that other node's rank by 1. This heuristic attempts to add smaller trees to larger trees, meaning that when we need to run a find operation, the depth of the tree is as small as possible. Using this heuristic brings the running time to $O(logn)$ per operation, amortized.

The second optimization is called *path compression*. It attempts to flatten the tree structure as much as possible whenever a find operation is performed. The basic idea is that as we traverse the parent pointers to the root, since all of the elements found along the way share the same root, then the parent of those elements might as well be the root itself. Thus we

perform one traversal to find out what the root is, and then we perform a second traversal, setting each node's parent pointer to the root. This has the effect of majorly flattening the structure, decreasing the time required for future operations. Using this heuristic alone has a running time to $O(logn)$ per operation, amortized.
Note the use of the word "alone" in the above paragraph. What happens if both heuristics are used simultaneously? In this case, the amortized running time per operation is $O(\alpha(n))$, whereα denotes the inverse of the Ackermann function. This function grows so slowly that for all practical values of n, $\alpha(n) < 4$. Effectively, this term is a small constant. In fact, it has been shown that this running time is optimal.

```
static class DisjointSet
{
      int[] p, rank;

      public DisjointSet(int size)
      {
            rank = new int[size];
            p = new int[size];
            for(int i = 0; i < size; i++)
            {
                  make_set(i);
            }
      }

      public void make_set(int x)
      {
            p[x] = x;
            rank[x] = 0;
      }

      public void merge(int x, int y)
      {
            link(find_set(x), find_set(y));
      }

      public void link(int x, int y)
      {
            if(rank[x] > rank[y])
                  p[y] = x;
            else
            {
                  p[x] = y;
                  if (rank[x] == rank[y])
                        rank[y] += 1;
            }
```

```
        }

        public int find_set(int x)
        {
              if (x != p[x])
                    p[x] = find_set(p[x]);
              return p[x];
        }
  }
```

7.4. RMQ

Range Minimum Query (RMQ) is used on arrays to find the position of an element with the minimum value between two specified indices. We will see later that the LCA problem can be reduced to a restricted version of an RMQ problem, in which consecutive array elements differ by exactly 1.

However, RMQs are not only used with LCA. They have an important role in string preprocessing, where they are used with suffix arrays (a new data structure that supports string searches almost as fast as suffix trees, but uses less memory and less coding effort).

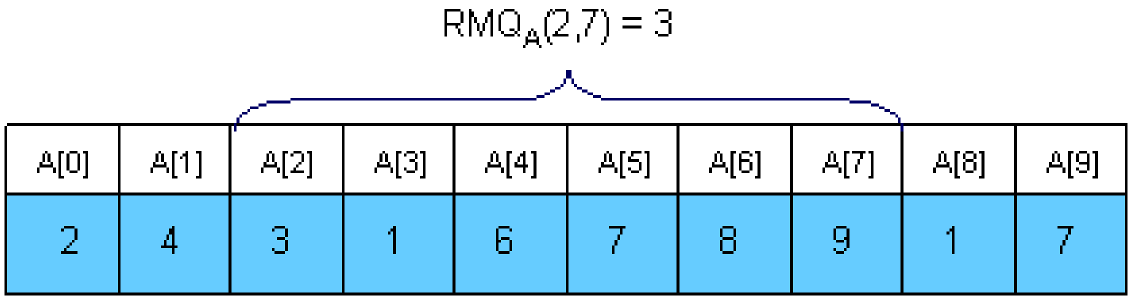
In this tutorial we will first talk about RMQ. We will present many approaches that solve the problem -- some slower but easier to code, and others faster. In the second part we will talk about the strong relation between LCA and RMQ. First we will review two easy approaches for LCA that don't use RMQ; then show that the RMQ and LCA problems are equivalent; and, at the end, we'll look at how the RMQ problem can be reduced to its restricted version, as well as show a fast algorithm for this particular case.

Notations
Suppose that an algorithm has preprocessing time **f(n)** and query time **g(n)**. The notation for the overall complexity for the algorithm is **<f(n), g(n)>**.

We will note the position of the element with the minimum value in some array **A** between indices **i** and **j** with **RMQA(i, j)**.

The furthest node from the root that is an ancestor of both **u** and **v** in some rooted tree **T** is **LCAT(u, v)**.

Range Minimum Query(RMQ)
Given an array **A[0, N-1]** find the position of the element with the minimum value between two given indices.



Trivial algorithms for RMQ

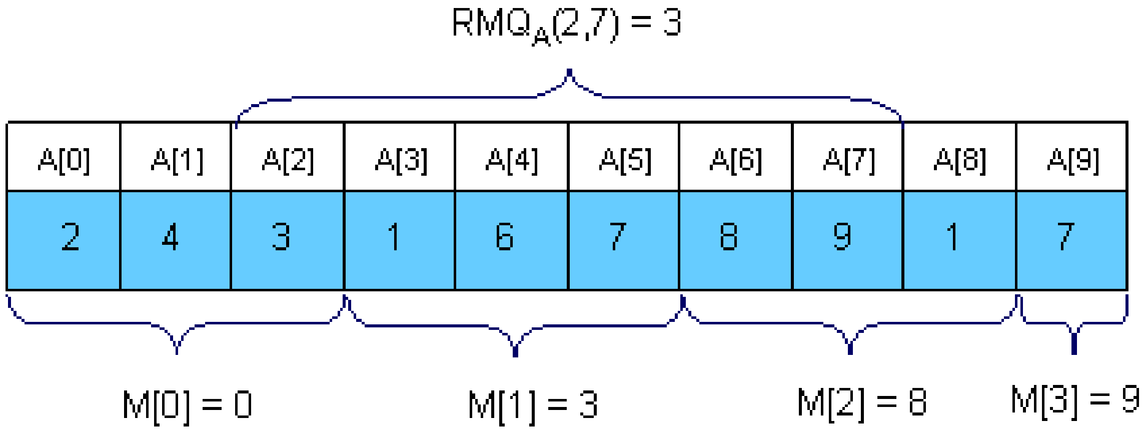
For every pair of indices **(i, j)** store the value of **RMQA(i, j)** in a table **M[0, N-1][0, N-1]**. Trivial computation will lead us to an**<O(N3), O(1)>** complexity. However, by using an easy dynamic programming approach we can reduce the complexity to**<O(N2), O(1)>.** The preprocessing function will look something like this:

```
 void process1(int M[MAXN][MAXN], int A[MAXN], int N)
  {
      int i, j;
      for (i =0; i < N; i++)
            M[i][i] = i;
      for (i = 0; i < N; i++)
            for (j = i + 1; j < N; j++)
            if (A[M[i][j - 1]] < A[j])
                    M[i][j] = M[i][j - 1];
            else
                  M[i][j] = j;
  }
```

This trivial algorithm is quite slow and uses **O(N2)** memory, so it won't work for large cases.

An <O(N), O(sqrt(N))> solution

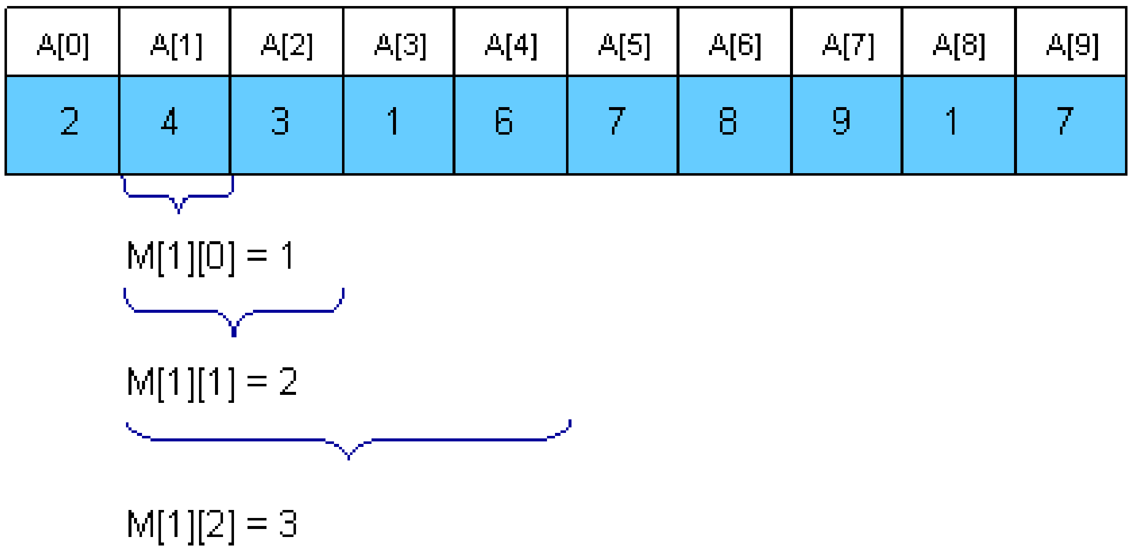
An interesting idea is to split the vector in **sqrt(N)** pieces. We will keep in a vector **M[0, sqrt(N)-1]** the position for the minimum value for each section. **M** can be easily preprocessed in **O(N)**. Here is an example:



Now let's see how can we compute **RMQA(i, j)**. The idea is to get the overall minimum from the **sqrt(N)** sections that lie inside the interval, and from the end and the beginning of the first and the last sections that intersect the bounds of the interval. To get**RMQA(2,7)** in the above example we should compare **A[2], A[M[1]], A[6]** and **A[7]** and get the position of the minimum value. It's easy to see that this algorithm doesn't make more than **3 * sqrt(N)** operations per query.

The main advantages of this approach are that is to quick to code (a plus for TopCoder-style competitions) and that you can adapt it to the dynamic version of the problem (where you can change the elements of the array between queries).

Sparse Table (ST) algorithm
A better approach is to preprocess **RMQ** for sub arrays of length **2k** using dynamic programming. We will keep an array **M[0, N-1][0, logN]** where **M[i][j]** is the index of the minimum value in the sub array starting at **i** having length **2j**. Here is an example:



For computing **M[i][j]** we must search for the minimum value in the first and second half of the interval. It's obvious that the small pieces have **2j - 1** length, so the recurrence is:

The preprocessing function will look something like this:

```
void process2(int M[MAXN][LOGMAXN], int A[MAXN], int N)
  {
      int i, j;

  //initialize M for the intervals with length 1
      for (i = 0; i < N; i++)
          M[i][0] = i;
  //compute values from smaller to bigger intervals
      for (j = 1; 1 << j <= N; j++)
          for (i = 0; i + (1 << j) - 1 < N; i++)
              if (A[M[i][j - 1]] < A[M[i + (1 << (j - 1))][j - 1]])
                  M[i][j] = M[i][j - 1];
          else
                  M[i][j] = M[i + (1 << (j - 1))][j - 1];
  }
```

Once we have these values preprocessed, let's show how we can use them to calculate **RMQA(i, j)**. The idea is to select two blocks that entirely cover the interval **[i..j]** and find the minimum between them. Let **k = [log(j - i + 1)]**. For computing **RMQA(i, j)** we can use the following formula:

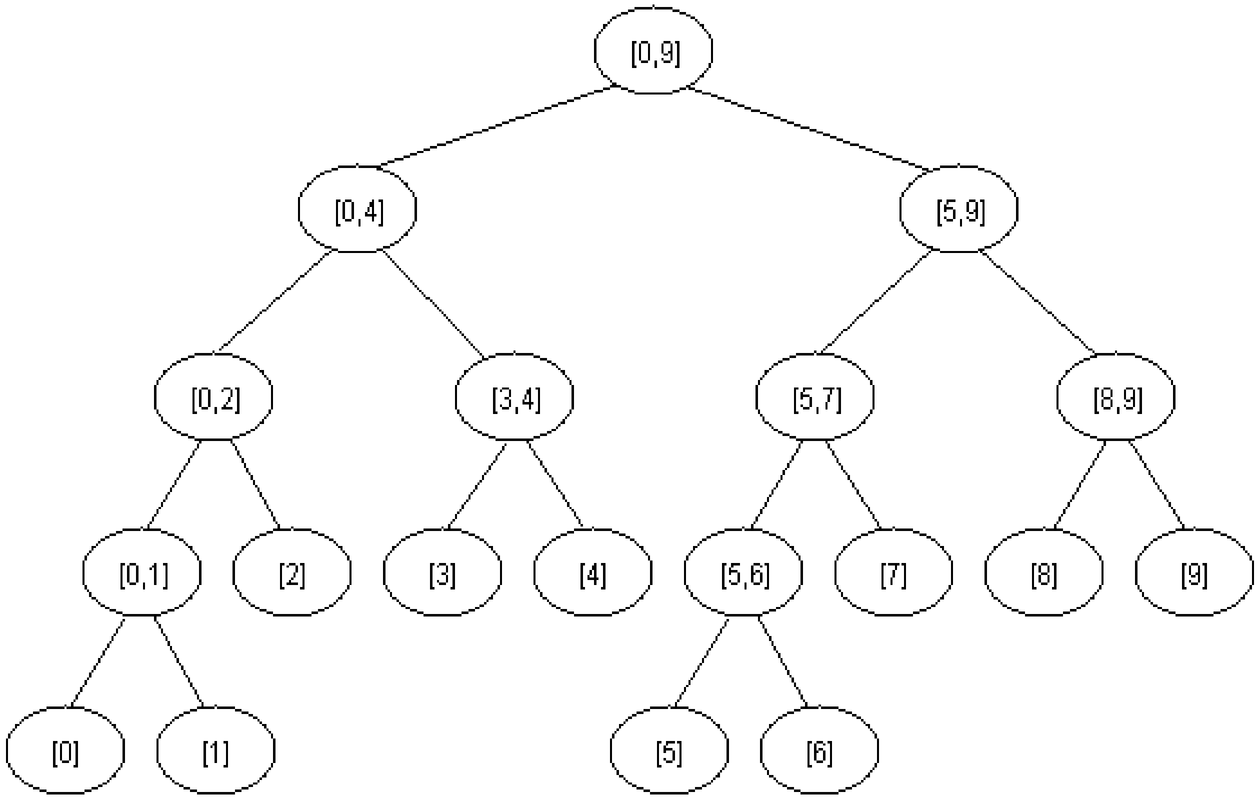
So, the overall complexity of the algorithm is **<O(N logN), O(1)>**.

Segment trees

For solving the RMQ problem we can also use segment trees. A segment tree is a heap-like data structure that can be used for making update/query operations upon array intervals in logarithmical time. We define the segment tree for the interval **[i, j]** in the following recursive manner:

- the first node will hold the information for the interval **[i, j]**
- if i<j the left and right son will hold the information for the intervals **[i, (i+j)/2]** and **[(i+j)/2+1, j]**

Notice that the height of a segment tree for an interval with **N** elements is **[logN] + 1**. Here is how a segment tree for the interval **[0, 9]** would look like:



The segment tree has the same structure as a heap, so if we have a node numbered **x** that is not a leaf the left son of **x** is **2*x**and the right son **2*x+1**.

For solving the RMQ problem using segment trees we should use an array **M[1, 2 * 2[logN] + 1]** where **M[i]** holds the minimum value position in the interval assigned to node **i**. At the beginning all elements in **M** should be **-1**. The tree should be initialized with the following function (**b** and **e** are the bounds of the current interval):

```
 void initialize(int node, int b, int e, int M[MAXIND], int A[MAXN], int
N)
  {
      if (b == e)
          M[node] = b;
      else
      {
  //compute the values in the left and right subtrees
          initialize(2 * node, b, (b + e) / 2, M, A, N);
          initialize(2 * node + 1, (b + e) / 2 + 1, e, M, A, N);
  //search for the minimum value in the first and
  //second half of the interval
          if (A[M[2 * node]] <= A[M[2 * node + 1]])
                M[node] = M[2 * node];
          else
                M[node] = M[2 * node + 1];
      }
  }
```

The function above reflects the way the tree is constructed. When calculating the minimum position for some interval we should look at the values of the sons. You should call the function with **node = 1, b = 0** and **e = N-1**.

We can now start making queries. If we want to find the position of the minimum value in some interval **[i, j]** we should use the next easy function:

```
 int query(int node, int b, int e, int M[MAXIND], int A[MAXN], int i, int
j)
  {
      int p1, p2;


  //if the current interval doesn't intersect
  //the query interval return -1
      if (i > e || j < b)
            return -1;

  //if the current interval is included in
  //the query interval return M[node]
      if (b >= i && e <= j)
            return M[node];

  //compute the minimum position in the
  //left and right part of the interval
      p1 = query(2 * node, b, (b + e) / 2, M, A, i, j);
      p2 = query(2 * node + 1, (b + e) / 2 + 1, e, M, A, i, j);

  //return the position where the overall
  //minimum is
      if (p1 == -1)
            return M[node] = p2;
      if (p2 == -1)
            return M[node] = p1;
      if (A[p1] <= A[p2])
            return M[node] = p1;
      return M[node] = p2;

  }
```

You should call this function with **node = 1, b = 0** and **e = N - 1**, because the interval assigned to the first node is **[0, N-1]**.

It's easy to see that any query is done in **O(log N)**. Notice that we stop when we reach completely in/out intervals, so our path in the tree should split only one time.

Using segment trees we get an **<O(N), O(logN)>** algorithm. Segment trees are very powerful, not only because they can be used for RMQ. They are a very flexible data structure, can solve even the dynamic version of RMQ problem, and have numerous applications in range searching problems.

Implementación:

```
            int[] buildRMQ(int[] vector, int n)
            {
                  int logn = 0;
                  for (int k = 1; k < n; k *= 2) ++logn;
                  int[] b = new int[n * logn];
                  System.arraycopy(vector, 0, b, 0, n);
                  int delta = 0;
                  for(int k = 1; k < n; k *= 2)
                  {
                        System.arraycopy(b, delta, b, n + delta, n);
```

```
                    delta += n;
                          for(int i = 0; i < n - k; i++) b[i + delta] =
                          Math.min(b[i + delta], b[i + k + delta]);
              }
              return b;
        }


        int minimum(int[] rmq, int x, int y)
        {
              int z = y - x, k = 0, e = 1, s;
              s = (((z & 0xffff0000) != 0) ? 1 : 0) << 4; z >>= s; e <<=
s; k |= s;
              s = (((z & 0x0000ff00) != 0) ? 1 : 0) << 3; z >>= s; e <<=
s; k |= s;
              s = (((z & 0x000000f0) != 0) ? 1 : 0) << 2; z >>= s; e <<=
s; k |= s;
              s = (((z & 0x0000000c) != 0) ? 1 : 0) << 1; z >>= s; e <<=
s; k |= s;
              s = (((z & 0x00000002) != 0) ? 1 : 0) << 0; z >>= s; e <<=
s; k |= s;
              return Math.min(rmq[x + n * k], rmq[y + n * k - e + 1]);
        }
```

7.5. Fenwick tree

Fenwick tree (aka Binary indexed tree) is a data structure that maintains a sequence of elements, and is able to compute cumulative sum of any range of consecutive elements in O(logn) time. Changing value of any single element needs O(logn) time as well.
The structure is space-efficient in the sense that it needs the same amount of storage as just a simple array of n elements.

Implementación:

One dimension

```
// In this implementation, the tree is represented by a vector<int>.
// Elements are numbered by 0, 1, ..., n-1.
// tree[i] is sum of elements with indexes i&(i+1)..i, inclusive.
// (Note: this is a bit different from what is proposed in Fenwick's article.
// To see why it makes sense, think about the trailing 1's in binary
// representation of indexes.)
// Creates a zero-initialized Fenwick tree for n elements.
vector<int> create(int n) { return vector<int>(n, 0); }

// Returns sum of elements with indexes a..b, inclusive
int query(const vector<int> &tree, int a, int b) {
    if (a == 0) {
        int sum = 0;
        for (; b >= 0; b = (b & (b + 1)) - 1)
          sum += tree[b];
        return sum;
    } else {
        return query(tree, 0, b) - query(tree, 0, a-1);
    }
}

// Increases value of k-th element by inc.
void increase(vector<int> &tree, int k, int inc) {
    for (; k < (int)tree.size(); k |= k + 1)
        tree[k] += inc;
}

Two dimension

typedef long long lint;
typedef unsigned long long ulint;

const int MAX = 1050;

int tree[MAX][MAX], lastx, lasty;
int M[MAX][MAX];


int
readx (int x, int y)
{
  int sum = 0;
  while (x > 0)
    {
      sum += tree[x][y];
      x -= (x & -x);
    }
  return sum;
}

int
getsum (int x, int y)
```

```
{
  int sum = 0;
  while (y > 0)
    {
      sum += readx (x, y);
      y -= (y & -y);
    }
  return sum;

}


void
updatey (int x, int y, int val)
{
  while (y = lasty)
    {
      tree[x][y] += val;
      y += (y & -y);
    }

}

void
update (int x, int y, int val)
{
  while (x = lastx)
    {
      updatey(x,y,val);
      x += (x & -x);
    }
}

int
get (int x1, int y1, int x2, int y2)
{
  int a = getsum (x1-1 , y1-1 );
  int b = getsum (x2, y2);
  int c = getsum (x2, y1 -1);
  int d = getsum (x1 -1, y2);
  return a + b - c - d;
}
```

Table of Contents

1. Grafos....1
1.1. Shortest paths....1
1.1.1. Dijkstra....1
1.1.2. Bellman-Ford (also longest and negative)....2
1.1.3. Edge disjoint shortest paths....3
1.1.3.1. Suurballe (completely disjoint)....3
1.2. All-pairs shortest paths....4
1.2.1. Floyd-Warshall....5
1.2.2. Johnson....6
1.3. MST....7
1.3.1. Kruskal....7
1.3.2. Prim....9
1.4. Graph search....11
1.4.1. BFS....11
1.4.2 DFS....12
1.5. Connected components....12
1.5.1. Strongly connected components and cycle detection....13
1.6. Matching....15
1.6.1. Maximum matching....16
1.6.1.1. Edmonds....16
1.6.1.2. Maximum cardinality bipartite matching....24
1.6.2. Stable marriage problem....26
1.7. Max-flow min-cut....29
1.8. Eulerian path....36
1.9. Lowest common ancestor....37
1.9.1. Tarjan....39
1.9.2. Lingas....42
1.9.3. LCA with RMQ....47
1.10. Topological sort....54
1.11. Two-sat....56
1.12. Assigment problem....59
2. Math and number theory....70
2.1. Combinations and permutations....71
2.2. GCD....77
2.3. Extended euclidean algorithm and Bézout's Identity....78
2.4. Sieve....79
2.4.1. Sundaram....81
2.4.2. Eratosthenes....81
2.4.3. Atkin....82
2.5. Binomial coefficients....83
2.6. Fibonacci numbers....86
2.7. Euler's totient function....88

2.8. LCM........91
2.9. Catalan numbers........92
2.10. Gaussian elimination........96
2.11. Orbits, Burnside’s lemma........100
2.12. Expectation........101
2.13. Gauss sum........105
2.14. Egyptian numbers........106
2.15. De Moivre's theorem........108
2.16. Primes........110
2.16.1. Primality testing........111
2.16.2. Prime factorization........118
2.17. Bell numbers........119
2.18. Rationals........121
2.19. Polynomials........124
3. Strings........131
3.1. Longest common subsequence........132
3.2. Aho-corasick........134
3.3. KMP........143
3.4. Suffix array........144
4. DP........167
4.1. LIS........167
4.2. Subset sum........170
4.3. Knapsack problem........174
4.4. Josephus problem........175
4.5. All subsets........178
4.6. Minimum biroute pass........179
4.7. Best sum........180
5. Geometry 182
5.1. Polygons........182
5.2. Convex hull........185
5.3. Sweep line........193
5.4. Great-circle distance........196
5.5. Pick’s theorem........200
5.6. Lines and circles........201
6. Others and games........212
6.1. Binary search........212
6.2. Backtracking........214
6.3. Nim........216
6.4. Counting inversions........219
6.4. Poker........223
6.5. Sudoku........227
6.6. Chess knights........233
7. Data structures........235
7.1. Binary heap........235

7.2. Fibonnaci heap....238
7.3. Union Find....241
7.4. RMQ....244
7.5. Fenwick tree....251