TEAM

TURING COMPLETE

Table of Contents

[Formulas PERIMETROS Y AREAS Figuras PLANAS Geometricas 4](#_Toc498042500)

[Logic gate Operators 5](#_Toc498042501)

[2-SAT (2 variables satisfied) 5](#_Toc498042502)

[C++ Common tricks 10](#_Toc498042503)

[From int to string 10](#_Toc498042504)

[From int to string 10](#_Toc498042505)

[Printing with precision 11](#_Toc498042506)

[Graphs and Trees anotations 11](#_Toc498042507)

[Minimum Spanning Tree (Kruskal) 11](#_Toc498042508)

[Articulation Points 11](#_Toc498042509)

[Bridges 11](#_Toc498042510)

[DAC 11](#_Toc498042511)

[Strongly connected component 12](#_Toc498042512)

[Longest Path de un Directed Acyclic Graph 12](#_Toc498042513)

[Operator Overloading 13](#_Toc498042514)

[DP 14](#_Toc498042515)

[LCS (Longest Common Subsequence) 14](#_Toc498042516)

[Edit Distance (Find minimum number of edits (operations) required to convert ‘str1’ into ‘str2’) 16](#_Toc498042517)

[LIS (Longest Increasing Subsequence) 18](#_Toc498042518)

[Find Largest Submatriz containing only 0 21](#_Toc498042519)

[GRAPH 23](#_Toc498042520)

[Disjoin Set 23](#_Toc498042521)

[2 Colorable (Is it bipartite graph?) 25](#_Toc498042522)

[Bridges 27](#_Toc498042523)

[Biconnected Components 32](#_Toc498042524)

[Articulation Points 33](#_Toc498042525)

[SCC (Strongly Connected Components) 39](#_Toc498042526)

[MaxFlow : Dinic Version 44](#_Toc498042527)

[TREE 53](#_Toc498042528)

[LCA (Lowest Common Ancestor) 53](#_Toc498042529)

[Minimun Spaning Tree Kruskal (If i want the maximun i just reverse the edges in kruskal) 58](#_Toc498042530)

[Strings 61](#_Toc498042531)

[Suffix Array 61](#_Toc498042532)

[KMP (Find the ocurrences of a pattern in a string) 67](#_Toc498042533)

[Math 68](#_Toc498042534)

[Sieve of Eras 68](#_Toc498042535)

Notas For Chivo

Anotar:

Exponentes de 2 para sacar logaritmo base 2 del análisis de complejidad

|  |  |
| --- | --- |
| 2^0 | 1 |
| 2^1 | 2 |
| 2^2 | 4 |
| 2^3 | 8 |
| 2^4 | 16 |
| 2^5 | 32 |
| 2^6 | 64 |
| 2^7 | 128 |
| 2^8 | 256 |
| 2^9 | 512 |
| 2^10 | 1024 |

Ejemplo:

Log 10^9 es aproximadamente 2^10 \* 2^10 \* 2^10 = 2^30 = y el log de eso es aprox 30

Log 10^5 es aproximadamente 2^10 \* 2^7 = 2^17 = y el log de eso es aprox 17

Log 2\*10^5 es aproximadamente 2^10 \* 2^7 \* 2^1 = 2^18 = y el log de eso es 18

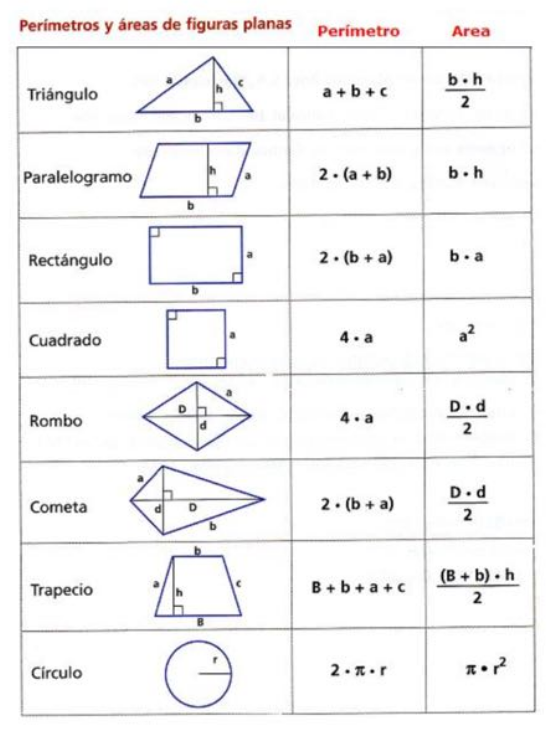
**Tabla para contest’s**

|  |  |  |  |
| --- | --- | --- | --- |
| Letra Problema | Read y Quien (Iniciales) | Complejidad | Tema |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Reminders de problemas comunes (Keep in Mind)**

* Long long
* Binary Search?
* Runtime Error? Did u initialize a size wrong?

# Formulas PERIMETROS Y AREAS Figuras PLANAS Geometricas



# Logic gate Operators

## 2-SAT (2 variables satisfied)

<https://gitlab.com/cjoa/Codelib/blob/master/Graph/2sat.cpp>

1 /\*

2 2-SAT using SCC Kosaraju's algorithm

3

4 Write the boolean formula as a conjunction of implications (e.g.

5 (x1->x2) ^ (~x1->x3)), and create a graph whose vertices are xi and ~xi

6 for all variables xi, and whose edges are the implications. The

7 formula is satisfiable if and only if no contradiction of the form

8 xi -> ~xi -> xi arises following a chain of such implications. This

9 amounts to checking whether xi and ~xi are in the same strongly

10 connected component, which can be done in linear time.

11

12 Building the Graph

13

14 The expression (u or v) is equivalent to (~u --> v) and also (~v --> u),

15 and in fact the 2 equations together capture all the possibilities in the

16 expression (u or v). Thus build a graph of size 2n (where n is the number

17 of variables), then for each expression (u or v) construct two directed

18 edges accordingly. There is a contradiction (unsatisfiable) if, for some

19 variable x, there is a path from x to ~x AND a path from ~x to x.

20 Otherwise for the given constraints our expression is satisfiable.

21

22 Assign true to variable x if there is no path x -> ~x; otherwise, we can assign

23 the value false since we know that there is no path ~x -> x

24 \*/

25

26 #include <iostream>

27 #include <stdexcept>

28

29 #include <vector>

30 #include <stack>

31

32 **using namespace std**;

33

34 **typedef vector**<**int**> VI;

35 **typedef vector**<VI> VVI;

36

37 **class** TwoSAT {

38 **int** N; // number of variables, indexed from 1 to N

39 // negation of variable i is indicated as -i

40 // these are mapped to vertexes (i-1)\*2 and (i-1)\*2 + 1

41 VVI G, GT; // implication graph and its transpose (in adjacency list

42 // form)

43

44 VI assigned; // assigned true/false value for each "vertex"

45

46 **int** neg(**int** u) { **return** u ^ 1; }

47

48 VI vis, which\_scc;

49 **stack**<**int**> S;

50 // Kosaraju's 2 DFS algorithm

51 **void** dfs(**const** VVI& adj, **int** u, **int** comp = -1) {

52 vis[u] = 1;

53 **if** (comp >= 0)

54 which\_scc[u] = comp;

55 **for**(**int** j = 0; j < (**int**) adj[u].**size**(); ++j) {

56 **int** v = adj[u][j];

57 **if**(!vis[v])

58 dfs(adj, v, comp);

59 }

60 **if** (comp < 0)

61 S.push(u);

62 }

63

64 **static int** var2vertex(**int** x) {

65 **if** (x > 0) **return** 2\*(x-1);

66 **else return** 2\*(-x-1) + 1;

67 }

68 **static int** vertex2var(**int** u) {

69 **return** (u & 1) ? -(u/2+1) : (u/2+1);

70 }

71

72 **public**:

73 TwoSAT(**int** \_N) : N(\_N), G(\_N\*2), GT(\_N\*2) { }

74

75 // add x or y clause

76 **void** add\_clause(**int** x, **int** y) {

77 // transform the expresion "x or y" to these two implications:

78 // ~x -> y

79 // ~y -> x

80 add\_implication(-x, y);

81 add\_implication(-y, x);

82 }

83

84 // single variable clause; this means that this single varialbe must be true

85 **void** add\_clause(**int** x) {

86 add\_clause(x, x);

87 }

88

89 // add x xor y clause

90 **void** add\_xor\_clause(**int** x, **int** y) {

91 add\_clause(x, y);

92 add\_clause(-x, -y);

93 }

94

95 **void** add\_equiv\_clause(**int** x, **int** y) {

96 add\_implication(x, y);

97 add\_implication(y, x);

98 }

99

100 **void** add\_implication(**int** x, **int** y) {

101 **int** u = var2vertex(x);

102 **int** v = var2vertex(y);

103

104 G[u].**push\_back**(v);

105 GT[v].**push\_back**(u);

106 }

107

108 **bool** process() {

109 // 1st DFS

110 vis = VI(2\*N, 0);

111 **for**(**int** u = 0; u < 2\*N; ++u) {

112 **if**(!vis[u])

113 dfs(G, u);

114 }

115

116 // 2nd DFS

117 vis = VI(2\*N, 0);

118 which\_scc = VI(2\*N, -1);

119 **int** ncomp = 0;

120 **while** (!S.**empty**()) {

121 **int** u = S.top(); S.pop();

122 **if**(!vis[u]) {

123 dfs(GT, u, ncomp);

124 ncomp++;

125 }

126 }

127

128 assigned.**clear**();

129

130 // check if i and -i are in same strongly connected component

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

132 **int** u = var2vertex(i);

133 **if** (which\_scc[u] == which\_scc[neg(u)]) {

134 **return false**;

135 }

136 }

137

138 // compute the assigned value for each "vertex"

139 assigned = VI(2\*N, -1);

140 // http://codeforces.com/blog/entry/16205?#comment-342676

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

142 **int** u = var2vertex(i);

143 assigned[u] = which\_scc[u] > which\_scc[neg(u)];

144 assigned[neg(u)] = !assigned[u];

145 }

146

147 **return true**;

148 }

149

150 **int** assigned\_value(**int** x) {

151 **if** (x <= 0 || x > N)

152 **throw out\_of\_range**("Variable is out of range");

153 **return** assigned[ var2vertex(x) ];

154 }

155 };

156

157 /\*

158

159 12 8

160 1 7

161 1 6

162 12 7

163 12 6

164 -6 -12

165 -6 -1

166 -7 -12

167 -7 -1

168

169 10 13

170 -9 -9

171 1 8

172 1 4

173 9 8

174 9 4

175 2 7

176 2 5

177 9 7

178 9 5

179 2 -7

180 2 1

181 -8 -7

182 -8 1

183

184 \*/

185

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

187 **int** N, M;

188 **cin** >> N >> M;

189 TwoSAT sat(N);

190 **for**(**int** m = 0; m < M; ++m) {

191 **int** x, y;

192 **cin** >> x >> y;

193 sat.add\_clause(x, y);

194 }

195

196 **if** (sat.process()) {

197 **cout** << "Satisfiable" << **endl**;

198 **for** (**int** i = 1; i <= N; ++i)

199 **cerr** << i << ": " << sat.assigned\_value(i) << **endl**;

200 }

201 **else**

202 **cout** << "Not satisfiable" << **endl**;

204 **return** 0;

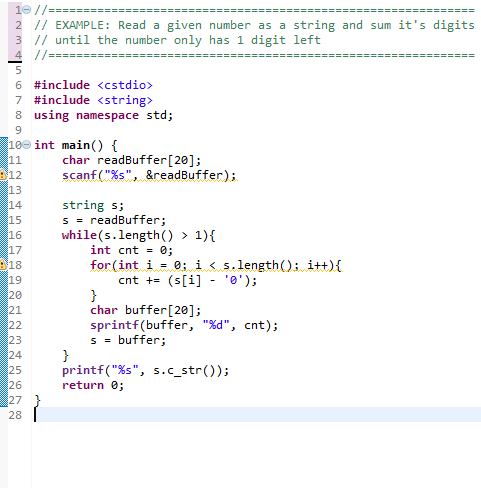
205 }

3-SAT NP Complete (3 variables satisfied, NP complete problem not solvable in efficient time)

# C++ Common tricks

## From int to string

Use **sprintf** with a char array, and then equal your string to the char array, the string will only have the content of the char array until the end symbol ‘\0’

****

## From int to string

Use the itoa function

## Printing with precision

****

**Also look at reference for more options in printf** [**http://en.cppreference.com/w/c/io/fprintf**](http://en.cppreference.com/w/c/io/fprintf)

# Graphs and Trees anotations

## Minimum Spanning Tree (Kruskal)

Para Maximum Spanning Tree lo mismo pero solo reverso el sorteo de los edges y ya ;3

## Articulation Points

Nodos que si los desconecto del grafo el grafo se divide

## Bridges

Edges que si los elimino del grafo el grafo se divide

## DAC

Directed Acyclic Graph

Inicialmente me dan un grafo directed con ciclos, y quiero convertirlo a un grafo sin ciclos

## Strongly connected component

Un grafo con ciclos, en el cual desde cualquier nodo puede volver a él mismo por los ciclos

## Longest Path de un Directed Acyclic Graph

**Topological Sort**

Ordenación del grafo directed por pre requisitos, para sacar el longest path.

Obtener longest path saliendo de un punto.

Si lo quiero longest path terminando en un punto, simplemente reverso la dirección de cada edge y tiró el topological sort.

Luego que tengo el grafo tiro un DP y encuentro el Path Máximo.saliendo de todos los posibles puntos que no tienen un prerrequisito.

# Operator Overloading

1 #include <iostream>

2 #include <vector>

3

4 **using namespace std**;

5

6 **struct** Point{

7 **int** x;

8 **int** y;

9

10 **bool operator**<(**const** Point& a) **const**{

11 **if**(x != a.x){

12 **if**(x < a.x) **return true**;

13 **else return false**;

14 }

15 **return** (y < a.y);

16 }

17

18 };

19

20 **int** main() {

21 Point p1 = {1,2};

22 Point p2 = {1,1};

23 **cout** << (p1 < p2) << **endl**; //false

24 **cout** << (p2 < p1) << **endl**; //true

25 **return** 0;

26 }

27

# DP

## LCS (Longest Common Subsequence)

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”. So a string of length n has 2^n different possible subsequences.

<https://gitlab.com/cjoa/Codelib/blob/master/DivideAndConquer/DP/lcs.cpp>

1 #include <cstdio>

2

3 #include <vector>

4 #include <algorithm>

5

6 **using namespace std**;

7

8 #define MAXS 1020

9 **int** DP[MAXS+4][MAXS+4];

10 **char** P[MAXS+4][MAXS+4];

11 **int** LCS(**const vector**<**int**>& seqa, **const vector**<**int**>& seqb) {

12 **int** lena = seqa.**size**();

13 **int** lenb = seqb.**size**();

14

15 **for** (**int** j = 0; j <= lenb; ++j) {

16 DP[0][j] = 0;

17 P[0][j] = ' ';

18 }

19

20 **for** (**int** i = 1; i <= lena; ++i) {

21 DP[i][0] = 0;

22 P[i][0] = ' ';

23 **for** (**int** j = 1; j <= lenb; ++j) {

24 **if** (seqa[i-1] == seqb[j-1]) {

25 DP[i][j] = 1 + DP[i-1][j-1];

26 P[i][j] = '\*';

27 }

28 **else** {

29 **if** (DP[i-1][j] > DP[i][j-1]) {

30 DP[i][j] = DP[i-1][j];

31 P[i][j] = 'a';

32 }

33 **else** {

34 DP[i][j] = DP[i][j-1];

35 P[i][j] = 'b';

36 }

37 }

38 }

39 }

40 **return** DP[lena][lenb];

41 }

42

43

44 **int** main() {

45 **int** N, M;

46 scanf("%d %d", &N, &M);

47 **vector**<**int**> A(N);

48 **for** (**int** i = 0; i < N; ++i)

49 scanf("%d", &A[i]);

50 **vector**<**int**> B(M);

51 **for** (**int** j = 0; j < M; ++j)

52 scanf("%d", &B[j]);

53 LCS(A, B);

54

55 **vector**<**int**> res;

56 **for** (**int** n = A.**size**(), m = B.**size**(), t = 1; n > 0 && m > 0; ++t) {

57 // fprintf(stderr, "%d: %d %d %c\n", t, n, m, P[n][m]);

58 **switch** (P[n][m]) {

59 **case** '\*':

60 res.**push\_back**( A[n-1] );

61 n--, m--;

62 **break**;

63 **case** 'a':

64 n--;

65 **break**;

66 **case** 'b':

67 m--;

68 **break**;

69 }

70 }

71

72 **reverse**(res.**begin**(), res.**end**());

73 **for** (**int** x : res)

74 printf("%d ", x);

75 printf("\n");

76

77 **return** 0;

78 }

## Edit Distance (Find minimum number of edits (operations) required to convert ‘str1’ into ‘str2’)

<https://gitlab.com/cjoa/Codelib/blob/master/DivideAndConquer/DP/edit_distance.cpp>

1 #include <cstdio>

2

3 #include <string>

4

5 **using namespace std**;

6

7

8 #define MAXS 1024

9 **int** DP[MAXS+4][MAXS+4];

10 **char** P[MAXS+4][MAXS+4];

11

12 **int** edit\_distance(**const string**& S, **const string**& T) {

13 **int** N = S.**size**();

14 **int** M = T.**size**();

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

16 DP[i][0] = i;

17 P[i][0] = '-';

18 }

19 **for** (**int** j = 1; j <= M; ++j) {

20 DP[0][j] = j;

21 P[0][j] = '+';

22 }

23 DP[0][0] = 0;

24 P[0][0] = '.';

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

26 **for** (**int** j = 1; j <= M; ++j) {

27 **if** (S[i-1] == T[j-1]) {

28 DP[i][j] = DP[i-1][j-1];

29 P[i][j] = '=';

30 }

31 **else**

32 **if** (DP[i-1][j-1] < **min**(DP[i-1][j], DP[i][j-1])) {

33 DP[i][j] = DP[i-1][j-1] + 1;

34 P[i][j] = 'C'; // change

35 }

36 **else**

37 **if** (DP[i-1][j] < DP[i][j-1]) {

38 DP[i][j] = DP[i-1][j] + 1;

39 P[i][j] = '-'; // delete

40 }

41 **else** {

42 DP[i][j] = DP[i][j-1] + 1;

43 P[i][j] = '+'; // insert

44 }

45 }

46 }

47 **return** DP[N][M];

48 }

49

50 **int** main() {

51 **string** s1 = "abcd";

52 **string** s2 = "yabceegpt";

53 printf("%d", edit\_distance(s1,s2));

54 }

## LIS (Longest Increasing Subsequence)

Let us discuss Longest Increasing Subsequence (LIS) problem as an example problem that can be solved using Dynamic Programming.

The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, the length of LIS for {10, 22, 9, 33, 21, 50, 41, 60, 80} is 6 and LIS is {10, 22, 33, 50, 60, 80}

1 #include <cstdio>

2

3 #include <vector>

4 #include <string>

5

6 #include <algorithm>

7

8 #include <set>

9

10 **using namespace std**;

11

12 // O(n log k) algorithm.

13 **template**<**typename** T>

14 **vector**<**int**> LIS(**const vector**<T> &A) {

15 **int** N = **int**(A.**size**());

16

17 **if** (N == 0)

18 **return vector**<**int**>();

19

20 **vector**<**int**> par(N, -1); // parent vector

21 **vector**<**int**> tail;

22 // tail[len] == (index of) smallest possible tail out of all increasing

23 // subsequences of length len using elements a0 .. ai

24

25 tail.**push\_back**(0);

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

27 // If last is smaller than new element, push into "stack"

28 **if** (A[ tail.**back**() ] < A[i]) {

29 par[i] = tail.**back**();

30 tail.**push\_back**(i);

31 **continue**;

32 }

33

34 // Otherwise, do binary search to find position in stack where new

35 // new element should be

36 // This binary seach is a lower bound. If sequence is not stricly

37 // increasing, use upper bound

38 **int** lo = 0, hi = **int**(tail.**size**())-1;

39 **while** (lo < hi) {

40 **int** mid = (lo + hi) / 2;

41 **if** (A[ tail[mid] ] >= A[i])

42 hi = mid;

43 **else**

44 lo = mid+1;

45 }

46 // lo is first position where A[ tail[lo] ] >= A[i]

47

48 **if** (A[ tail[lo] ] > A[i]) { // protect from A[ tail[lo] ] == A[i] ???

49 tail[lo] = i;

50 **if** (lo > 0)

51 par[i] = tail[lo-1];

52 }

53 }

54

55 **vector**<**int**> res(tail.**size**());

56 **for** (**int** k = **int**(tail.**size**())-1, i = tail.**back**(); k >= 0; i = par[i], --k)

57 res[k] = i;

58 **return** res; // devuelve un vector de las posiciones de elementos del LIS

59 }

60

61

62 // http://codeforces.com/blog/entry/13225

63 // If only interested in size of LIS

64 **template**<**typename** T>

65 **int** LIS\_size(**const vector**<T> &A) {

66 **multiset**<T> S;

67 **for** (**int** i = 0; i < (**int**) A.**size**(); ++i) {

68 S.**insert**(A[i]);

69 **typename**

70 **multiset**<T>::**iterator** it = S.**upper\_bound**(A[i]);

71 // For strictly increasing sequence:

72 // multiset<T>::iterator it = S.lower\_bound(A[i]); ++it;

73 **if** (it != S.**end**())

74 S.**erase**(it);

75 }

76 **return** S.**size**();

77 }

78

79

80 **int** main() {

81 //int a[] = { 1, 1, 1, 1, 1 };

82 **int** a[] = { 1, 9, 3, 8, 11, 4, 5, 6, 4, 19, 7, 1, 7 };

83 **vector**<**int**> seq(a, a+**sizeof**(a)/**sizeof**(a[0]));

84 **vector**<**int**> lis = LIS(seq);

85

86 **for** (**int** i = 0; i < (**int**) lis.**size**(); i++)

87 printf("%d ", seq[lis[i]]);

88 puts("");

89 **return** 0;

90 }

## Find Largest Submatriz containing only 0

<https://gitlab.com/cjoa/Codelib/blob/master/DivideAndConquer/DP/largest_submatrix.cpp>

1 #include <cstdio>

2

3 #include <vector>

4 #include <string>

5

6 #include <stack>

7

8 **using namespace std**;

9

10 // find largestsubmatrix containing only zeroes

11

12 **int** LargestSubmatrix( **const vector**< **vector**<**int**> >& A ) {

13 **int** nRows = A.**size**();

14 **int** nCols = nRows > 0 ? A[0].**size**() : 0;

15

16 **int** ans = 0;

17

18 **vector**<**int**> d(nCols, -1);

19 // d[c] = posicion (fila) del '1' mas cercano en la columna c

20

21 **vector**<**int**> dL(nCols), dR(nCols);

22 // dL[c] = posicion (columna) del peor '1' a la izquierda de la columna c

23 // dR[c] = posicion (columna) del peor '1' a la derecha de la columna c

24

25 **for** (**int** r = 0; r < nRows; ++r) {

26 **for** (**int** c = 0; c < nCols; ++c)

27 **if** (A[r][c] == 1)

28 d[c] = r;

29

30 **stack**<**int**> S;

31

32 **for** (**int** c = 0; c < nCols; ++c) {

33 **while** (!S.**empty**() && d[S.top()] <= d[c]) S.pop();

34 dL[c] = S.**empty**() ? -1 : S.top();

35 S.push(c);

36 }

37

38 S = **stack**<**int**>();

39

40 **for** (**int** c = nCols-1; c >= 0; --c) {

41 **while** (!S.**empty**() && d[S.top()] <= d[c]) S.pop();

42 dR[c] = S.**empty**() ? nCols : S.top();

43 S.push(c);

44 }

45

46 **for** (**int** c = 0; c < nCols; ++c)

47 ans = **max**(ans, (r - d[c]) \* (dR[c] - dL[c] - 1));

48 }

49

50 **return** ans;

51 }

52

53 **int** main() {

54

55 }

# GRAPH

## Disjoin Set

[**https://gitlab.com/cjoa/Codelib/blob/master/Graph/disjointset.cpp**](https://gitlab.com/cjoa/Codelib/blob/master/Graph/disjointset.cpp)

1 #include <cstdio>

2 #include <vector>

3

4 **using namespace std**;

5

6 **class** DisjointSet {

7 **int** N;

8 **int** ncomp;

9 **vector**<**int**> par;

10 **vector**<**int**> **rank**;

11

12 **public**:

13 DisjointSet(**size\_t** \_N) : N(\_N), ncomp(\_N), par(\_N, -1), **rank**(\_N, 0) {}

14 **void** reset() {

15 par.**assign**(N, -1);

16 **rank**.**assign**(N, 0);

17 ncomp = N;

18 }

19 **int size**() **const** {

20 **return** ncomp;

21 }

22 **int** find\_rep(**int** u) {

23 // path compression

24 **return** par[u] < 0 ? u : par[u] = find\_rep(par[u]);

25 /\*

26 vector<int> s;

27 while (par[u] >= 0) {

28 s.push\_back(u);

29 u = par[u];

30 }

31 for (int i = 0; i < (int) s.size(); ++i)

32 par[s[i]] = u;

33 return u;

34 \*/

35 }

36 **bool** union\_rep(**int** u, **int** v) {

37 **int** u\_root = find\_rep(u);

38 **int** v\_root = find\_rep(v);

39 **if** (u\_root == v\_root)

40 **return false**;

41 **if** (**rank**[u\_root] > **rank**[v\_root])

42 par[v\_root] = u\_root;

43 **else** {

44 par[u\_root] = v\_root;

45 **if** (**rank**[u\_root] == **rank**[v\_root])

46 **rank**[v\_root] = **rank**[u\_root] + 1;

47 }

48 --ncomp;

49 **return true**;

50 }

51 };

52

53 **int** main() {

54 DisjointSet dset(1000);

55 // join elements

56 dset.union\_rep(3, 10);

57 dset.union\_rep(500, 87);

58 dset.union\_rep(77, 760);

59 dset.union\_rep(500, 10);

60 printf("Representative of %d is %d\n", 3, dset.find\_rep(5));

61 printf("Representative of %d is %d\n", 77, dset.find\_rep(77));

62 **return** 0;

63 }

## 2 Colorable (Is it bipartite graph?)

[**https://gitlab.com/cjoa/Codelib/blob/master/Graph/twocolorable.cpp**](https://gitlab.com/cjoa/Codelib/blob/master/Graph/twocolorable.cpp)

1 #include <cstdio>

2

3 #include <vector>

4

5 **using namespace std**;

6

7 **typedef vector**<**int**> VI;

8 **typedef vector**<VI> VVI;

9

10 **class** TwoColorable {

11 **int** N;

12 **bool** dfs(**int** u, **int** color) {

13 **if** (colors[u] >= 0)

14 **return** colors[u] == color;

15 colors[u] = color;

16 **for** (**int** j = 0; j < (**int**) adj[u].**size**(); ++j) {

17 **if** (!dfs(adj[u][j], color^1))

18 **return false**;

19 }

20 **return true**;

21 }

22 **public**:

23 // input: adjacency list

24 VVI adj;

25

26 // output: color of each vertex (0 or 1) if graph is bipartite

27 VI colors;

28 TwoColorable(**int** \_N) : N(\_N), adj(VVI(\_N)) {}

29

30 // returns true if graph is bipartite

31 **bool** process() {

32 colors = VI(N, -1);

33 **for** (**int** u = 0; u < N; ++u)

34 **if** (colors[u] < 0)

35 **if** (!dfs(u, 0))

36 **return false**;

37 **return true**;

38 }

39 };

40

41 /\*

42 9 9

43 0 1

44 0 2

45 1 3

46 2 3

47 3 4

48 4 5

49 4 6

50 4 7

51 6 8

52 \*/

53

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

55 **char** line[200];

56 **int** N, M;

57

58 gets(line);

59 sscanf(line, "%d %d", &N, &M);

60

61 TwoColorable tc(N);

62 **while** (M-- > 0) {

63 **int** u, v;

64 gets(line);

65 sscanf(line, "%d %d", &u, &v);

66 tc.adj[u].**push\_back**(v);

67 }

68

69 **if** (!tc.process())

70 puts("Graph is not 2-colorable");

71 **else** {

72 puts("2-Coloring of graph:");

73 **for** (**int** u = 0; u < N; ++u)

74 printf("%d: %d\n", u, tc.colors[u]);

75 }

76 **return** 0;

77 }

## Bridges

[**https://gitlab.com/cjoa/Codelib/blob/master/Graph/bridges.cpp**](https://gitlab.com/cjoa/Codelib/blob/master/Graph/bridges.cpp)

1 #include <cstdio>

2 #include <cstring>

3

4 #include <vector>

5

6 #include <stack>

7 #include <algorithm>

8

9 **using namespace std**;

10

11 **typedef vector**<**int**> VI;

12 **typedef vector**<VI> VVI;

13

14 **using namespace std**;

15

16 **struct** Edge {

17 **int** u, v;

18 **bool** is\_bridge;

19 Edge(**int** \_u, **int** \_v) : u(\_u), v(\_v), is\_bridge(**false**) {}

20 **int** other(**int** x) **const** { **return** u == x ? v : u; }

21 };

22

23 **class** Bridges {

24 **int** N;

25 **int** dfs\_time; // DFS "counter"

26

27 VI dfs\_order;

28 VI dfs\_low;

29 VI par\_edge\_id; // index of edge that leads to parent

30

31 **void** \_dfs1(**int** u); // DFS to find bridges

32 **void** \_dfs2(**int** u); // DFS to find two-edge components

33

34 **public**:

35 // "INPUT":

36 **vector**<Edge> edges; // all edges added to graph

37 VVI adj; // adjacency list of edge indices

38

39 // OUTPUT:

40 VVI twoedge\_comp; // list of 2-edge connected components, where each one

41 // is a list of vertices belonging to the same component

42 VI member\_comp; // member\_comp[u] = index of component vertex u belongs

43

44 Bridges(**int** \_N) : N(\_N), adj(\_N) {}

45 **int** add\_edge(**int** u, **int** v); // u, v must be 0-based

46 **void** run();

47 VVI get\_condensed\_tree() **const**;

48 };

49

50

51 **int** Bridges::add\_edge(**int** u, **int** v) {

52 **int** eid = edges.**size**();

53 edges.**push\_back**(Edge(u, v));

54 adj[u].**push\_back**(eid);

55 adj[v].**push\_back**(eid);

56 **return** eid;

57 }

58

59 **void** Bridges::run() {

60 dfs\_order = VI(N, 0);

61 dfs\_low = VI(N, 0);

62

63 par\_edge\_id = VI(N, -1);

64 dfs\_time = 0;

65 **for** (**int** u = 0; u < N; ++u) {

66 **if** (dfs\_order[u] == 0)

67 \_dfs1(u);

68 }

69

70 // Find 2-edge connected components

71 twoedge\_comp.**clear**();

72 member\_comp = VI(N, -1);

73 **for** (**int** u = 0; u < N; ++u) {

74 **if** (member\_comp[u] < 0) {

75 twoedge\_comp.**push\_back**(VI());

76 \_dfs2(u);

77 }

78 }

79 }

80

81 VVI Bridges::get\_condensed\_tree() **const** {

82 VVI res( twoedge\_comp.**size**() );

83 **for** (**int** j = 0; j < (**int**) edges.**size**(); ++j) {

84 **if** (edges[j].is\_bridge) {

85 **int** uu = member\_comp[ edges[j].u ];

86 **int** vv = member\_comp[ edges[j].v ];

87 res[uu].**push\_back**(vv);

88 res[vv].**push\_back**(uu);

89 }

90 }

91 **return** res;

92 }

93

94 **void** Bridges::\_dfs1(**int** u) {

95 dfs\_order[u] = dfs\_low[u] = ++dfs\_time;

96

97 **for** (**int** j = 0; j < (**int**) adj[u].**size**(); ++j) {

98 **int** eid = adj[u][j];

99 **if** (eid == par\_edge\_id[u]) **continue**;

100 **int** v = edges[eid].other(u);

101 **if** (dfs\_order[v] == 0) { // is tree edge

102 par\_edge\_id[v] = eid;

103 \_dfs1(v);

104 // detect bridge

105 **if** (dfs\_low[v] > dfs\_order[u])

106 edges[eid].is\_bridge = **true**;

107 dfs\_low[u] = **min**(dfs\_low[u], dfs\_low[v]);

108 }

109 **else** { // back edge

110 dfs\_low[u] = **min**(dfs\_low[u], dfs\_order[v]);

111 }

112 }

113 }

114

115 **void** Bridges::\_dfs2(**int** u) {

116 twoedge\_comp.**back**().**push\_back**(u);

117 member\_comp[u] = **int**(twoedge\_comp.**size**())-1;

118 **for** (**int** j = 0; j < (**int**) adj[u].**size**(); ++j) {

119 **int** eid = adj[u][j];

120 **if** (edges[eid].is\_bridge) **continue**;

121 **int** v = edges[eid].other(u);

122 **if** (member\_comp[v] >= 0) **continue**;

123 \_dfs2(v);

124 }

125 }

126

127

128

129 **const char** \*lines[] = {

130 /\*

131 "7 8",

132 "0 1",

133 "0 3",

134 "1 2",

135 "2 3",

136 "2 6",

137 "3 4",

138 "3 5",

139 "4 5"

140 \*/

141

142 // Example from Wikipedia

143 "14 24",

144 "0 1",

145 "0 2",

146 "1 3",

147 "2 3",

148 "3 4",

149 "4 5",

150 "5 6",

151 "6 7",

152 "6 11",

153 "6 13",

154 "7 8",

155 "8 9",

156 "7 9",

157 "9 10",

158 "11 12",

159 "10 11",

160 "5 4", // multiple edge between 4 and 5

161 "2 3", // multiple edges between 2 and 3

162 "2 3", // multiple edges between 2 and 3

163 "0 1", // multiple edges between 0 and 1

164 "3 3", // self loop

165 "4 4", // self loop

166 "5 5", // self loop

167 "0 0" // self loop

168

169 /\*

170 "6 5",

171 "3 0",

172 "5 0",

173 "1 4",

174 "2 3",

175 "4 2"

176 \*/

177 };

178

179 **int** main() {

180 **int** N, M;

181

182 sscanf(lines[0], "%d %d", &N, &M);

183

184 Bridges b(N);

185 **for** (**int** m = 1; m <= M; ++m) {

186 **int** u, v;

187 sscanf(lines[m], "%d %d", &u, &v);

188 b.add\_edge(u, v);

189 }

190

191 b.run();

192

193 fprintf(stderr, "Bridges\n");

194 **for** (**int** eid = 0; eid < (**int**) b.edges.**size**(); ++eid) {

195 **if** (b.edges[eid].is\_bridge)

196 fprintf(stderr, "%d: (%d,%d)\n", eid, b.edges[eid].u, b.edges[eid].v);

197 }

198 fprintf(stderr, "\n");

199

200 fprintf(stderr, "2-Edge Connected Components\n");

201 **for** (**int** k = 0; k < (**int**) b.twoedge\_comp.**size**(); ++k) {

202 fprintf(stderr, "%d:", k);

203 **for** (**int** j = 0; j < (**int**) b.twoedge\_comp[k].**size**(); ++j) {

204 **int** u = b.twoedge\_comp[k][j];

205 fprintf(stderr, " %d", u);

206 }

207 fprintf(stderr, "\n");

208 }

209

210 **return** 0;

211 }

## Biconnected Components

If i wanted the connected components i get after removing the articulation points i can use the code in the bridge chivo, it gives me the connected components.

## Articulation Points

[**https://gitlab.com/cjoa/Codelib/blob/master/Graph/artic\_points.cpp**](https://gitlab.com/cjoa/Codelib/blob/master/Graph/artic_points.cpp)

1 #include <cstdio>

2 #include <cstring>

3

4 #include <vector>

5

6 #include <stack>

7 #include <algorithm>

8

9 **using namespace std**;

10

11 **typedef vector**<**int**> VI;

12 **typedef vector**<VI> VVI;

13

14 **using namespace std**;

15

16 **struct** Edge {

17 **int** u, v;

18 Edge(**int** \_u, **int** \_v) : u(\_u), v(\_v) {}

19 **int** other(**int** x) **const** { **return** u == x ? v : u; }

20 };

21

22

23 **class** ArticPoints {

24 **int** N;

25 **int** dfs\_time; // DFS "counter"

26

27 VI dfs\_order;

28 VI dfs\_low;

29 VI par\_edge\_id; // index of edge that leads to parent

30

31 **stack**<**int**> S; // stack of visited edges (indices) in DFS that are not

32 // in a cycle

33 **void** \_store\_bcc(**int** eid);

34

35 **void** \_dfs(**int** u);

36

37 **public**:

38 // "INPUT":

39 **vector**<Edge> edges; // all edges added to graph

40 VVI adj; // adjacency list of edge indices

41

42 // OUTPUT:

43 VI artic\_pts;

44 VVI twovertex\_comp; // list of 2-vertex connected components, where each one

45 // is a list of vertices belonging to the same component

46

47 ArticPoints(**int** \_N) : N(\_N), adj(\_N) {}

48

49 **int** add\_edge(**int** u, **int** v); // u, v must be 0-based

50

51 **void** run();

52

53 };

54

55

56 **int** ArticPoints::add\_edge(**int** u, **int** v) {

57 **int** eid = edges.**size**();

58 edges.**push\_back**(Edge(u, v));

59 adj[u].**push\_back**(eid);

60 adj[v].**push\_back**(eid);

61 **return** eid;

62 }

63

64 **void** ArticPoints::run() {

65 N = adj.**size**();

66 twovertex\_comp.**clear**();

67 S = **stack**<**int**>();

68

69 artic\_pts.**clear**();

70

71 dfs\_order = VI(N, 0);

72 dfs\_low = VI(N, 0);

73

74 par\_edge\_id = VI(N, -1);

75 dfs\_time = 0;

76 **for** (**int** u = 0; u < N; ++u) {

77 **if** (dfs\_order[u] == 0)

78 \_dfs(u);

79 }

80 }

81

82 **void** ArticPoints::\_dfs(**int** u) {

83 dfs\_order[u] = dfs\_low[u] = ++dfs\_time;

84

85 **int** tree\_edges = 0;

86 **bool** is\_artic\_point = **false**;

87

88 **for** (**int** j = 0; j < (**int**) adj[u].**size**(); ++j) {

89 **int** eid = adj[u][j];

90 **if** (eid == par\_edge\_id[u]) **continue**;

91 **int** v = edges[eid].other(u);

92 **if** (dfs\_order[v] == 0) { // is tree edge

93 ++tree\_edges;

94 par\_edge\_id[v] = eid;

95

96 S.push(eid);

97

98 \_dfs(v);

99

100 **if** (par\_edge\_id[u] == -1) { // u is root

101 **if** (tree\_edges > 1) {

102 // root is an articulation point iff it has more than one tree edge

103 is\_artic\_point = **true**;

104 }

105 \_store\_bcc(eid); // all subtrees of root form a BCC

106 }

107 **else** {

108 **if** (dfs\_low[v] >= dfs\_order[u]) {

109 // v cannot reach an earlier node -> u is an articulation

110 is\_artic\_point = **true**;

111 \_store\_bcc(eid);

112 }

113 }

114 dfs\_low[u] = **min**(dfs\_low[u], dfs\_low[v]);

115 }

116 **else** { // back edge

117 dfs\_low[u] = **min**(dfs\_low[u], dfs\_order[v]);

118 S.push(eid);

119 }

120 }

121

122 **if** (is\_artic\_point)

123 artic\_pts.**push\_back**(u);

124 }

125

126 **void** ArticPoints::\_store\_bcc(**int** p\_eid) {

127 twovertex\_comp.**push\_back**(VI());

128 VI& comp = twovertex\_comp.**back**();

129 **while** (!S.**empty**()) {

130 **int** eid = S.top();

131 S.pop();

132 comp.**push\_back**(eid);

133 **if** (eid == p\_eid)

134 **break**;

135 }

136 }

137

138

139 **const char** \*lines[] = {

140 /\*

141 "7 8",

142 "0 1",

143 "0 3",

144 "1 2",

145 "2 3",

146 "2 6",

147 "3 4",

148 "3 5",

149 "4 5"

150 \*/

151

152 // Example from Wikipedia

153 "14 24",

154 "0 1",

155 "0 2",

156 "1 3",

157 "2 3",

158 "3 4",

159 "4 5",

160 "5 6",

161 "6 7",

162 "6 11",

163 "6 13",

164 "7 8",

165 "8 9",

166 "7 9",

167 "9 10",

168 "11 12",

169 "10 11",

170 "5 4", // multiple edge between 4 and 5

171 "2 3", // multiple edges between 2 and 3

172 "2 3", // multiple edges between 2 and 3

173 "0 1", // multiple edges between 0 and 1

174 "3 3", // self loop

175 "4 4", // self loop

176 "5 5", // self loop

177 "0 0" // self loop

178

179 /\*

180 "6 5",

181 "3 0",

182 "5 0",

183 "1 4",

184 "2 3",

185 "4 2"

186 \*/

187 };

188

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

190 **int** N, M;

191

192 sscanf(lines[0], "%d %d", &N, &M);

193

194 ArticPoints ap(N);

195 **for** (**int** m = 1; m <= M; ++m) {

196 **int** u, v;

197 sscanf(lines[m], "%d %d", &u, &v);

198 ap.add\_edge(u, v);

199 }

200

201 ap.run();

202

203 fprintf(stderr, "Articulation points\n");

204 **for** (**int** u : ap.artic\_pts)

205 fprintf(stderr, "%d ", u);

206 fprintf(stderr, "\n");

207 fprintf(stderr, "\n");

208

209 fprintf(stderr, "Biconnected Components\n");

210 **for** (**int** k = 0; k < (**int**) ap.twovertex\_comp.**size**(); ++k) {

211 fprintf(stderr, "%d:", k);

212 **for** (**int** eid : ap.twovertex\_comp[k]) {

213 fprintf(stderr, " (%d,%d)", ap.edges[eid].u, ap.edges[eid].v);

214 }

215 fprintf(stderr, "\n");

216 }

217 fprintf(stderr, "\n");

218

219 **return** 0;

220 }

## SCC (Strongly Connected Components)

[**https://gitlab.com/cjoa/Codelib/blob/master/Graph/scc.cpp**](https://gitlab.com/cjoa/Codelib/blob/master/Graph/scc.cpp)

1 #include <iostream>

2 #include <cstring>

3

4 #include <algorithm>

5 #include <stack>

6 #include <vector>

7

8 **using namespace std**;

9

10 **typedef vector**<**int**> VI;

11 **typedef vector**<VI> VVI;

12

13 **class** TarjanSCC {

14 **int** N;

15 **int** id; // DFS depth "counter"

16

17 **stack**<**int**> S;

18 **vector**<**bool**> in\_stack; // flag to indicate if node is in stack S

19

20 VI low;

21 VI dfs\_order;

22

23 **void** dfs(**int** u);

24

25 **public**:

26 // input: adjacency list of directed graph

27 VVI adj;

28

29 // output

30 VVI SCC; // list of strongly connected components

31 // each strongly connected component is a list of vertices

32 VI memberSCC; // memberSCC[u] = strongly connected component id of vertex u

33

34 TarjanSCC(**int** \_N) : N(\_N), adj(\_N) {}

35 **void** add\_edge(**int** u, **int** v) {

36 adj[u].**push\_back**(v);

37 }

38 **void** run();

39

40 // return new graph where SCCs are collapsed into supernodes

41 VVI get\_condensed\_DAG();

42 };

43

44 **void** TarjanSCC::run() {

45 N = adj.**size**();

46 id = 0;

47

48 S = **stack**<**int**>();

49 in\_stack = **vector**<**bool**>(N, **false**);

50

51 low = VI(N);

52 dfs\_order = VI(N, 0);

53

54 SCC.**clear**();

55 memberSCC = VI(N, 0);

56

57 **for** (**int** u = 0; u < N; ++u)

58 **if** (dfs\_order[u] == 0) // not visited

59 dfs(u); // start dfs at node u

60 }

61

62 **void** TarjanSCC::dfs(**int** u) {

63 dfs\_order[u] = low[u] = ++id;

64 S.push(u);

65 in\_stack[u] = **true**;

66 **for** (**int** j = 0; j < (**int**) adj[u].**size**(); ++j) {

67 **int** v = adj[u][j]; // v is succesor of u

68 **if** (dfs\_order[v] == 0) { // v is unvisited, tree edge

69 dfs(v);

70 low[u] = **min**(low[u], low[v]);

71 }

72 **else**

73 **if** (in\_stack[v]) // v is back edge

74 low[u] = **min**(low[u], dfs\_order[v]);

75 }

76 **if** (low[u] == dfs\_order[u]) {

77 **int** nSCC = SCC.**size**();

78 SCC.**push\_back**(**vector**<**int**>());

79 **while** (**true**) {

80 **int** v = S.top();

81 S.pop();

82 in\_stack[v] = **false**;

83 SCC[nSCC].**push\_back**(v);

84 memberSCC[v] = nSCC;

85 **if** (v == u)

86 **break**;

87 }

88 }

89 }

90

91

92 VVI TarjanSCC::get\_condensed\_DAG() {

93 run();

94 VVI res( SCC.**size**() );

95 **for** (**int** u = 0; u < (**int**) adj.**size**(); ++u) {

96 **int** u\_scc = memberSCC[u];

97 **for** (**int** j = 0; j < (**int**) adj[u].**size**(); ++j) {

98 **int** v = adj[u][j];

99 **int** v\_scc = memberSCC[v];

100 **if** (u\_scc == v\_scc) **continue**;

101 res[u\_scc].**push\_back**(v\_scc);

102 }

103 }

104 **return** res;

105 }

106

107

108

109 /\*

110

111 10 15

112 A B

113 A D

114 B C

115 B F

116 C A

117 C D

118 C E

119 D E

120 F C

121 G F

122 G H

123 H F

124 H J

125 I H

126 J I

127

128 \*/

129

130 /\*

131 20 26

132 9 -9

133 9 -9

134 -1 8

135 -8 1

136 -1 4

137 -4 1

138 -9 8

139 -8 9

140 -9 4

141 -4 9

142 -2 7

143 -7 2

144 -2 5

145 -5 2

146 -9 7

147 -7 9

148 -9 5

149 -5 9

150 -2 -7

151 7 2

152 -2 1

153 -1 2

154 8 -7

155 7 -8

156 8 1

157 -1 -8

158 \*/

159

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

161 **int** N, M;

162 **cin** >> N >> M;

163 TarjanSCC tscc(N);

164 **while** (M-- > 0) {

165 **int** a, b;

166 **cin** >> a >> b;

167 tscc.adj[a+10].**push\_back**(b+10);

168 }

169

170 tscc.run();

171 **for** (**int** i = 0; i < (**int**) tscc.SCC.**size**(); ++i) {

172 **cout** << "SCC #" << i << ":";

173 **for** (**int** j = 0; j < (**int**) tscc.SCC[i].**size**(); ++j)

174 **cout** << ' ' << (tscc.SCC[i][j]-10);

175 **cout** << **endl**;

176 }

177 **for** (**int** u = 0; u < N; ++u)

178 **cout** << (u-10) << ": " << tscc.memberSCC[u] << **endl**;

179 **cout** << **endl**;

180

181 **return** 0;

182 }

## MaxFlow : Dinic Version

[**https://gitlab.com/cjoa/Codelib/blob/master/Graph/maxflow.cpp**](https://gitlab.com/cjoa/Codelib/blob/master/Graph/maxflow.cpp)

1 #include <cstdio>

2

3 #include <vector>

4 #include <queue>

5 #include <algorithm>

6

7 **using namespace std**;

8

9 **typedef vector**<**int**> VI;

10 **typedef vector**<VI> VVI;

11

12 #define HIGHESTSETBIT(mask) ( sizeof(int)\*8-1-\_\_builtin\_clz((mask)) )

13

14 #define INF 98765432

15

16 **class** MaxFlow {

17 **size\_t** N;

18 VI sp\_par, sp\_D;

19 **int** augmenting\_path\_bfs(**int** src, **int** dst, VI& path, **int** mincap=0);

20 // int augmenting\_path\_pfs(int src, int dst, VI& path, int mincap=0);

21

22 VI last\_edge;

23 **int** update\_flow\_dfs(**int** u, **int** dst, **int** cfp = INF); // for Dinic

24

25 **public**:

26 VVI cap;

27 // VVI adj;

28

29 // OUTPUT:

30 VVI flow;

31

32 MaxFlow(**int** \_N) : cap(\_N, VI(\_N)) {}

33 **void** add\_edge(**int** u, **int** v, **int** \_cap=1);

34

35 // Edmonds-Karp Ford Fulkerson Shortest Augmenting Path -- O(V \* E^2)

36 **int** edmonds\_karp(**int** s, **int** t, **bool** scaled=**false**);

37

38 // Dinitz's Algoritm -- O(V^2 \* E)

39 **int** dinic(**int** s, **int** t);

40 };

41

42 **inline void** MaxFlow::add\_edge(**int** u, **int** v, **int** \_cap) {

43 cap[u][v] += \_cap;

44 }

45

46 **int** MaxFlow::augmenting\_path\_bfs(**int** src, **int** dst, VI& path, **int** mincap) {

47 sp\_par.**assign**(N, -1);

48 sp\_D.**assign**(N, -1);

49

50 sp\_D[src] = 0;

51 **queue**<**int**> q;

52 q.push(src);

53 **while** (!q.**empty**()) {

54 **int** cur = q.**front**();

55 q.pop();

56 **if** (cur == dst) {

57 **int** cfp = INF;

58 path.**resize**(sp\_D[dst]+1);

59 path[sp\_D[dst]] = dst;

60 **for** (**int** curv = dst, k = sp\_D[dst]-1; sp\_par[curv] >= 0; --k) {

61 **int** prev = sp\_par[curv];

62 cfp = **min**(cfp, cap[prev][curv] - flow[prev][curv]);

63 curv = path[k] = prev;

64 }

65 **return** cfp;

66 }

67 **for** (**int** v = 0; v < N; ++v) {

68 **if** (sp\_D[v] < 0 && cap[cur][v] - flow[cur][v] > mincap) {

69 sp\_D[v] = sp\_D[cur]+1;

70 sp\_par[v] = cur;

71 q.push(v);

72 }

73 }

74 }

75 path.**clear**();

76 **return** 0;

77 }

78

79 **int** MaxFlow::edmonds\_karp(**int** s, **int** t, **bool** scaled) {

80 N = cap.**size**();

81 flow = VVI(N, VI(N, 0));

82

83 **int** max\_cap;

84 **if** (scaled) {

85 max\_cap = 0;

86 **for** (**int** i = 0; i < N; ++i)

87 **for** (**int** j = 0; j < N; ++j)

88 max\_cap = **max**(max\_cap, cap[i][j]);

89 **if** (max\_cap <= 0)

90 **return** 0;

91 }

92 **else**

93 max\_cap = 1;

94

95 VI path;

96 **int** res = 0;

97 **for** (**int** dcap = 1 << HIGHESTSETBIT(max\_cap); dcap > 0; dcap >>= 1) {

98 **while** (**true**) {

99 **int** cfp = augmenting\_path\_bfs(s, t, path, dcap-1);

100 **if** (path.**empty**())

101 **break**;

102 **for** (**int** k = 1; k < (**int**) path.**size**(); ++k) {

103 **int** i = path[k-1];

104 **int** j = path[k];

105 flow[i][j] += cfp;

106 flow[j][i] = -flow[i][j]; // -= cfp;

107 }

108 res += cfp;

109 }

110 }

111 **return** res;

112 }

113

114 **int** MaxFlow::update\_flow\_dfs(**int** u, **int** dst, **int** cfp) {

115 **if** (u == dst)

116 **return** cfp;

117 **for** (; last\_edge[u] < N; ++last\_edge[u]) {

118 **int** v = last\_edge[u];

119 **if** (sp\_D[v] == sp\_D[u] + 1 && cap[u][v] - flow[u][v] > 0) {

120 **int** res = update\_flow\_dfs(v, dst, **min**( cfp, cap[u][v] - flow[u][v] ));

121 **if** (res > 0) {

122 flow[u][v] += res;

123 flow[v][u] = -flow[u][v];

124 **return** res;

125 }

126 }

127 }

128 **return** 0;

129 }

130

131 **int** MaxFlow::dinic(**int** s, **int** t) {

132 N = cap.**size**();

133 flow = VVI(N, VI(N, 0));

134

135 VI path;

136 **int** res = 0;

137

138 **while** (**true**) {

139 // find an augmenting path

140 **int** cfp = augmenting\_path\_bfs(s, t, path);

141 **if** (path.**empty**())

142 **break**;

143

144 // try finding more paths

145 /\*

146 for (int w = 0; w < N; ++w) {

147 if (sp\_D[w] >= 0 && cap[w][t] - flow[w][t] > 0) {

148 cfp = cap[w][t] - flow[w][t];

149 for (int curv = w, prev = sp\_par[w]; prev >= 0;

150 curv = prev, prev = sp\_par[prev])

151 cfp = min(cfp, cap[prev][curv] - flow[prev][curv]);

152 flow[w][t] += cfp;

153 flow[t][w] = -flow[w][t]; // -= cfp;

154 for (int curv = w, prev = sp\_par[w]; prev >= 0;

155 curv = prev, prev = sp\_par[prev]) {

156 flow[prev][curv] += cfp;

157 flow[curv][prev] = -flow[prev][curv]; // -= cfp;

158 }

159 res += cfp;

160 }

161 }

162 \*/

163

164 // Tested, but equal performance as above

165 last\_edge.**assign**(N, 0);

166 **while** (**true**) {

167 **int** cfp = update\_flow\_dfs(s, t);

168 **if** (cfp <= 0) **break**;

169 res += cfp;

170 }

171

172 }

173 **return** res;

174 }

175

176

177

178 **struct** Edge {

179 **int** u, v;

180 **int** rev; // index of reverse edge

181 **int** cap, flow;

182 Edge(**int** \_u, **int** \_v, **int** \_rev = 0, **int** \_cap = 1) :

183 u(\_u), v(\_v), rev(\_rev), cap(\_cap), flow(0) {}

184 };

185

186 **class** MaxFlowAdj {

187 **size\_t** N;

188 VI sp\_par, sp\_D;

189 **bool** augmenting\_path\_bfs(**int** src, **int** dst, **int** mincap=0);

190 // bool augmenting\_path\_pfs(int src, int dst, int mincap=0);

191

192 VI last\_edge;

193 **int** update\_flow\_dfs(**int** u, **int** dst, **int** cfp = INF); // for Dinic

194

195 **public**:

196 **vector**<Edge> edges;

197 VVI adj; // adjacency lists of edge indices

198

199 MaxFlowAdj(**size\_t** \_N) : N(\_N), adj(VVI(\_N)) {}

200 **void** add\_edge(**int** u, **int** v, **int** \_cap=1);

201

202 // Edmonds-Karp Ford Fulkerson Shortest Augmenting Path -- O(V \* E^2)

203 **int** edmonds\_karp(**int** s, **int** t, **bool** scaled=**false**);

204

205 // Dinic's Algoritm -- O(V^2 \* E)

206 **int** dinic(**int** s, **int** t);

207

208 // return one possible min cut

209 **vector**<Edge> mincut(**int** s, **int** t);

210 };

211

212 **inline void** MaxFlowAdj::add\_edge(**int** u, **int** v, **int** \_cap) {

213 **int** num\_edges = edges.**size**();

214 // forward edge

215 edges.**push\_back**( Edge(u, v, num\_edges+1, \_cap) );

216 adj[u].**push\_back**(num\_edges++);

217 // reverse edge

218 edges.**push\_back**( Edge(v, u, num\_edges-1, 0) );

219 adj[v].**push\_back**(num\_edges++);

220 }

221

222 **bool** MaxFlowAdj::augmenting\_path\_bfs(**int** src, **int** dst, **int** mincap) {

223 sp\_par.**assign**(N, -1);

224 sp\_D.**assign**(N, -1);

225

226 sp\_D[src] = 0;

227 **queue**<**int**> q;

228 q.push(src);

229 **while** (!q.**empty**()) {

230 **int** cur = q.**front**();

231 q.pop();

232 **for** (**int** j = 0; j < (**int**) adj[cur].**size**(); ++j) {

233 **int** e = adj[cur][j];

234 **int** v = edges[e].v;

235 **if** (sp\_D[v] < 0 && edges[e].cap - edges[e].flow > mincap) {

236 sp\_D[v] = sp\_D[cur] + 1;

237 sp\_par[v] = e;

238 q.push(v);

239 }

240 }

241 }

242 **return** sp\_D[dst] >= 0;

243 }

244

245 **int** MaxFlowAdj::update\_flow\_dfs(**int** u, **int** dst, **int** cfp) {

246 **if** (u == dst)

247 **return** cfp;

248 **for** (; last\_edge[u] < (**int**) adj[u].**size**(); ++last\_edge[u]) {

249 **int** e = adj[u][ last\_edge[u] ];

250 **int** u = edges[e].u;

251 **int** v = edges[e].v;

252

253 **if** (sp\_D[v] == sp\_D[u] + 1 && edges[e].cap - edges[e].flow > 0) {

254 **int** res = update\_flow\_dfs(v, dst, **min**( cfp, edges[e].cap - edges[e].flow ));

255 **if** (res > 0) {

256 edges[e].flow += res;

257 edges[ edges[e].rev ].flow = -edges[e].flow; // -= cfp;

258 **return** res;

259 }

260 }

261 }

262 **return** 0;

263 }

264

265 **int** MaxFlowAdj::dinic(**int** s, **int** t) {

266 N = **int**(adj.**size**());

267 **for** (**int** j = 0; j < (**int**) edges.**size**(); ++j)

268 edges[j].flow = 0;

269

270 VI path;

271 **int** res = 0;

272

273 **while** ( augmenting\_path\_bfs(s, t) ) {

274 // try finding more paths

275

276 /\*

277 for (int j = 0; j < adj[t].size(); ++j) {

278 int e = edges[ adj[t][j] ].rev;

279 int w = edges[e].u;

280 if (sp\_D[w] >= 0 && edges[e].cap - edges[e].flow > 0) {

281 int cfp = edges[e].cap - edges[e].flow;

282 for (int curv = w; sp\_par[curv] >= 0; ) {

283 int ee = sp\_par[curv];

284 cfp = min(cfp, edges[ee].cap - edges[ee].flow);

285 curv = edges[ee].u;

286 }

287 edges[e].flow += cfp;

288 edges[ edges[e].rev ].flow = -edges[e].flow; // -= cfp;

289 for (int curv = w; sp\_par[curv] >= 0; ) {

290 int ee = sp\_par[curv];

291 edges[ee].flow += cfp;

292 edges[ edges[ee].rev ].flow = -edges[ee].flow; // -= cfp;

293 curv = edges[ee].u;

294 }

295 res += cfp;

296 }

297 }

298 \*/

299 // UNTESTED

300 last\_edge.**assign**(N, 0);

301 **while** (**true**) {

302 **int** cfp = update\_flow\_dfs(s, t);

303 **if** (cfp <= 0) **break**;

304 res += cfp;

305 }

306 }

307 **return** res;

308 }

309

310 **vector**<Edge> MaxFlowAdj::mincut(**int** s, **int** t) {

311 dinic(s, t);

312 **vector**<Edge> res;

313 **for** (**int** k = 0; k < edges.**size**(); ++k) {

314 **const** Edge& e = edges[k];

315 **if** (e.cap > 0 && sp\_D[e.u] >= 0 && sp\_D[e.v] < 0)

316 // e is an edge in residual graph where u is reachable from s, but

317 // v is not reachable from s. So, e must be part of the s-t mincut

318 res.**push\_back**(e);

319 }

320 **return** res;

321 }

322

323 **int** src = 0, dst = 5;

324 **int** nNodes = 6;

325 **int** g[][3] = {

326 {0, 1, 3},

327 {0, 2, 3},

328 {1, 2, 2},

329 {1, 3, 3},

330 {2, 4, 2},

331 {3, 4, 4},

332 {3, 5, 2},

333 {4, 5, 3}

334 };

335

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

337 **int** res;

338 MaxFlow mf1(nNodes);

339 **for** (**int** i = 0; i < **sizeof**(g)/**sizeof**(g[0]); ++i)

340 mf1.add\_edge(g[i][0], g[i][1], g[i][2]);

341 res = mf1.dinic(src, dst);

342 printf("Dinic w/adj matrix : %d (%s)\n",

343 res, res == 5 ? "OK" : "NOK");

344

345 MaxFlowAdj mf2(nNodes);

346 **for** (**int** i = 0; i < **sizeof**(g)/**sizeof**(g[0]); ++i)

347 mf2.add\_edge(g[i][0], g[i][1], g[i][2]);

348 res = mf2.dinic(src, dst);

349 printf("Dinic w/adj list : %d (%s)\n",

350 res, res == 5 ? "OK" : "NOK");

351

352 **return** 0;

353 }

# TREE

## LCA (Lowest Common Ancestor)

[**https://gitlab.com/cjoa/Codelib/blob/master/Tree/lca.cpp**](https://gitlab.com/cjoa/Codelib/blob/master/Tree/lca.cpp)

1 #include <iostream>

2

3 #include <vector>

4 #include <cmath>

5 #include <queue>

6

7 **using namespace std**;

8

9 **typedef vector**<**int**> VI;

10 **typedef vector**<VI> VVI;

11

12 #define HIGHESTSETBIT(mask) ( sizeof(int)\*8-1-\_\_builtin\_clz((mask)) )

13

14 /\*

15 \* Log Intervals

16 \* <O(N log N), O(lg N)> LCA

17 \* M[j][i] is the 2^j-th ancestor of i;

18 \* note: indices were switched to improve caching performance

19 \*/

20

21 **class** LCA {

22 VVI M;

23

24 **int** N;

25 VI P; // parent array for tree

26 VI L; // level of each node in tree; level of root = 0

27 **void** bfs(**const** VVI& adj, **int** src) {

28 **const int** INF = 1000000000;

29 L = VI(N, INF);

30 L[src] = 0;

31 P = VI(N, -1);

32

33 **queue**<**int**> q;

34 q.push(src);

35 **while** (!q.**empty**()) {

36 **int** u = q.**front**();

37 q.pop();

38 **for** (**int** j = 0; j < (**int**) adj[u].**size**(); ++j) {

39 **int** v = adj[u][j];

40 **if** (L[v] > L[u] + 1) {

41 L[v] = L[u] + 1;

42 P[v] = u;

43 q.push(v);

44 }

45 }

46 }

47 }

48

49 **void** preprocess() {

50 // initialize M for every element in P and ancestor path with -1

51 // for (LOGN = 0; (1 << LOGN) < N; ++LOGN);

52 // int LOGN = N > 1 ? 1+HIGHESTSETBIT(N-1) : 0; // not tested!!!

53 **int** LOGN = N > 1 ? 1+HIGHESTSETBIT(N) : 1; // not tested!!!

54

55 M = VVI(LOGN, VI(N, -1));

56 /\*

57 for (int j = 0; (1 << j) < N; j++)

58 for (int i = 0; i < N; i++)

59 M[j][i] = -1;

60 \*/

61

62 // The first ancestor of every node i is P[i]

63 **for** (**int** i = 0; i < N; ++i)

64 M[0][i] = P[i];

65

66 // For each power of 2 distance, find ancestor

67 **for** (**int** j = 1; (1 << j) < N; ++j)

68 **for** (**int** i = 0; i < N; ++i)

69 **if** (M[j - 1][i] != -1)

70 M[j][i] = M[j - 1][ M[j - 1][i] ];

71 }

72

73 **public**:

74 LCA(**const** VVI& adj, **int** root = 0) {

75 N = adj.**size**();

76 **if** (N <= 0) **return**;

77

78 bfs( adj, root );

79

80 preprocess();

81 }

82

83 **int** query(**int** p, **int** q) **const** {

84 // if p is situated on a higher level than q then we swap them

85 **if** (L[p] < L[q])

86 **swap**(p, q);

87

88 // Compute the value of floor( log( L[p] ) )

89 **int** LOGLp = L[p] > 0 ? HIGHESTSETBIT(L[p]) : 0;

90 /\*

91 for (LOGp = 1; (1 << LOGp) <= L[p]; ++LOGp);

92 --LOGp;

93 \*/

94

95 // Find the ancestor of node p situated on the same level

96 // with q using the values in M

97 **for** (**int** j = LOGLp; j >= 0; --j)

98 **if** (L[p] - (1 << j) >= L[q])

99 p = M[j][p];

100

101 **if** (p == q)

102 **return** p;

103

104 // Compute LCA(p, q) using the values in M

105 **for** (**int** j = LOGLp; j >= 0; --j)

106 **if** (M[j][p] != -1 && M[j][p] != M[j][q])

107 p = M[j][p], q = M[j][q];

108

109 **return** P[p];

110 }

111

112 // return ancestor with distance d from node u

113 **int** ancestor(**int** u, **int** d) **const** {

114 **if** (L[u] < d) **return** -1;

115 **int** p = u;

116 /\*

117 if (d > 0) return p;

118 for (int j = HIGHESTSETBIT(d); j >= 0; --j)

119 if (d & (1<<j))

120 p = M[j][p];

121 \*/

122 **while** (d != 0) {

123 **int** j = HIGHESTSETBIT(d);

124 p = M[j][p];

125 d ^= 1 << j;

126 }

127 **return** p;

128 }

129 };

130

131 /\*

132 0

133 / | \

134 1 2 3

135 / / \

136 4 5 6

137 / |

138 7 8

139 / \

140 9 10

141 |

142 11

143 \*/

144

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

146 VVI adj(12);

147 adj[0].**push\_back**(1);

148 adj[0].**push\_back**(2);

149 adj[0].**push\_back**(3);

150 adj[1].**push\_back**(4);

151 adj[2].**push\_back**(5);

152 adj[2].**push\_back**(6);

153 adj[4].**push\_back**(7);

154 adj[6].**push\_back**(8);

155 adj[8].**push\_back**(9);

156 adj[8].**push\_back**(10);

157 adj[10].**push\_back**(11);

158

159 LCA lca( adj, 0 );

160

161 **cout** << "LCA(5, 10) = " << lca.query(5, 10) << **endl**;

162 **cout** << "LCA(3, 8) = " << lca.query(3, 8) << **endl**;

163 **cout** << "LCA(6, 11) = " << lca.query(6, 11) << **endl**;

164 **cout** << "LCA(1, 4) = " << lca.query(1, 4) << **endl**;

165 **cout** << "LCA(0, 0) = " << lca.query(0, 0) << **endl**;

166

167 **return** 0;

168 }

## Minimun Spaning Tree Kruskal (If i want the maximun i just reverse the edges in kruskal)

[**https://gitlab.com/cjoa/Codelib/blob/master/Graph/mst.cpp**](https://gitlab.com/cjoa/Codelib/blob/master/Graph/mst.cpp)

1 #include <cstdio>

2

3 #include <vector>

4 #include <algorithm>

5

6 **using namespace std**;

7

8 **class** DisjointSet {

9 **int** N;

10 **int** ncomp;

11 **vector**<**int**> par;

12 **vector**<**int**> **rank**;

13

14 **public**:

15 DisjointSet(**size\_t** \_N) : N(\_N), ncomp(\_N), par(\_N, -1), **rank**(\_N, 0) {}

16 **void** reset() {

17 par.**assign**(N, -1);

18 **rank**.**assign**(N, 0);

19 ncomp = N;

20 }

21 **int size**() **const** {

22 **return** ncomp;

23 }

24 **int** find\_rep(**int** u) {

25 // path compression

26 **return** par[u] < 0 ? u : par[u] = find\_rep(par[u]);

27 /\*

28 vector<int> s;

29 while (parent[u] >= 0) {

30 s.push\_back(u);

31 u = parent[u];

32 }

33 for (int i = 0; i < (int) s.size(); ++i)

34 parent[s[i]] = u;

35 return u;

36 \*/

37 }

38 **bool** union\_rep(**int** u, **int** v) {

39 **int** u\_root = find\_rep(u);

40 **int** v\_root = find\_rep(v);

41 **if** (u\_root == v\_root)

42 **return false**;

43 **if** (**rank**[u\_root] > **rank**[v\_root])

44 par[v\_root] = u\_root;

45 **else** {

46 par[u\_root] = v\_root;

47 **if** (**rank**[u\_root] == **rank**[v\_root])

48 **rank**[v\_root] = **rank**[u\_root] + 1;

49 }

50 --ncomp;

51 **return true**;

52 }

53 };

54

55 **struct** Edge {

56 **int** u, v;

57 **int** cost;

58 Edge(**int** \_u, **int** \_v, **int** \_cost) : u(\_u), v(\_v), cost(\_cost) {}

59 };

60

61 **class** CostCmp {

62 **public**:

63 **bool operator**()(**const** Edge& e1, **const** Edge& e2) {

64 **if** (e1.cost != e2.cost) **return** e1.cost < e2.cost;

65 **if** (e1.u != e2.u) **return** e1.u < e2.u;

66 **return** e1.v < e2.v;

67 }

68 };

69

70

71 **int** N, M;

72 **vector**<Edge> edges;

73

74 // vector<bool> in\_mst;

75 **long long** kruskal() {

76 // in\_mst.assign( edges.size(), false );

77 **sort**(edges.**begin**(), edges.**end**(), CostCmp());

78 DisjointSet dset(N);

79 **long long** cost = 0;

80 **for** (**int** j = 0; j < **int**(edges.**size**()) && **int**(dset.**size**()) > 1; ++j) {

81 **if** (dset.union\_rep(edges[j].u, edges[j].v)) {

82 cost += edges[j].cost;

83 // in\_mst[ edges[j].id ] = true;

84 }

85 }

86 **return** cost;

87 }

88

89 /\*

90 7 11

91 0 1 7

92 0 3 5

93 1 2 8

94 1 3 9

95 1 4 7

96 2 4 5

97 3 4 15

98 3 5 6

99 4 5 8

100 4 6 9

101 5 6 11

102 \*/

103

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

105 scanf("%d %d", &N, &M);

106 **for** (**int** j = 0; j < M; ++j) {

107 **int** u, v, cost;

108 scanf("%d %d %d", &u, &v, &cost);

109 edges.**push\_back**(Edge(u, v, cost));

110 }

111

112 **long long** res = kruskal();

113 printf("%lld\n", res);

114

115 **return** 0;

116 }

# Strings

## Suffix Array

1 #include <cstdio>

2 #include <cstring>

3

4 //#include <string>

5 #include <vector>

6 #include <algorithm>

7

8 **using namespace std**;

9

10 **class** SuffixArray {

11 **struct** CmpLtFirstChar {

12 **const char** \*str;

13 CmpLtFirstChar(**char** \*\_str) : str(\_str) {}

14 // CmpLtFirstChar(const string& \_s) : str(\_s.c\_str()) {}

15 **bool operator**() (**const int** x, **const int** y) **const** {

16 **return** str[x] < str[y];

17 }

18 };

19 **public**:

20 **int** N;

21

22 // INPUT string

23 **char** \*str;

24

25 // OUTPUT:

26 // pos = The suffix array. Contains the N suffixes of str sorted in

27 // lexicographical order. Each suffix is represented as a single

28 // integer: the position where it starts in str.

29 // rank = The inverse of the suffix array.

30 // rank[i] = the index of the suffix str[i..n) in the pos array.

31 // (In other words, rank[i] = k <=> pos[k] = i)

32 // With this array, you can compare two suffixes in O(1):

33 // Suffix str[i..N) is smaller than str[j..n) IFF rank[i] < rank[j]

34 // lcp\_len[i] = length of longest common prefix of suffix str[pos[i]..N)

35 // and suffix str[pos[i-1]..N); lcp\_len[0] = 0

36

37 **int** \*pos;

38 **int** \***rank**;

39 **int** \*lcp\_len;

40

41 SuffixArray(**int** \_N);

42 SuffixArray(**const char** \*s);

43 SuffixArray(**const** SuffixArray& sa);

44 ~SuffixArray();

45

46 SuffixArray& **operator**=(SuffixArray);

47

48 // O(n log n) - Manber and Myers suffix sort algorithm

49 **void** suffix\_sort();

50

51 // O(n) longest common prefix algorithm

52 **void** build\_lcp();

53

54 **int find**(**const char** \*s) **const**;

55 };

56

57 SuffixArray::SuffixArray(**int** \_N)

58 : N(\_N+1), str(**new char**[N]), pos(NULL), **rank**(NULL), lcp\_len(NULL) {

59 str[0] = 0;

60 }

61

62 SuffixArray::SuffixArray(**const char** \*s)

63 : N ((s ? strlen(s) : 0)+1), str(**new char**[N]),

64 pos(NULL), **rank**(NULL), lcp\_len(NULL) {

65 memcpy(str, s, N \* **sizeof**(**char**));

66 }

67

68 SuffixArray::~SuffixArray() {

69 **delete**[] str;

70 **delete**[] pos;

71 **delete**[] **rank**;

72 **delete**[] lcp\_len;

73 }

74

75 SuffixArray::SuffixArray(**const** SuffixArray& sa)

76 : N(sa.N), str(**new char**[sa.N]),

77 pos(sa.pos ? **new int**[sa.N] : NULL),

78 **rank**(sa.**rank** ? **new int**[sa.N] : NULL),

79 lcp\_len(sa.lcp\_len ? **new int**[sa.N] : NULL) {

80 memcpy(str, sa.str, N \* **sizeof**(**char**));

81 **if** (sa.pos) memcpy(pos, sa.pos, N \* **sizeof**(**int**));

82 **if** (sa.**rank**) memcpy(**rank**, sa.**rank**, N \* **sizeof**(**int**));

83 **if** (sa.lcp\_len) memcpy(lcp\_len, sa.lcp\_len, N \* **sizeof**(**int**));

84 }

85

86 // copy-swap idiom

87 // http://www.cplusplus.com/articles/y8hv0pDG/

88 SuffixArray& SuffixArray::**operator**=(SuffixArray tmp) {

89 **swap**( N, tmp.N );

90

91 **swap**( str, tmp.str );

92 **swap**( pos, tmp.pos );

93 **swap**( **rank**, tmp.**rank** );

94 **swap**( lcp\_len, tmp.lcp\_len );

95 **return** \***this**;

96 }

97

98 **void** SuffixArray::suffix\_sort() {

99 N = strlen(str) + 1;

100 **delete**[] pos;

101 **delete**[] **rank**;

102 pos = **new int**[N];

103 **rank** = **new int**[N];

104 **for** (**int** i = 0; i < N; ++i)

105 pos[i] = i;

106

107 // sort suffixes according to their first characters

108 **std**::**sort**(pos, pos + N, CmpLtFirstChar(str));

109

110 **bool** \*bh = **new bool**[N];

111 **bool** \*b2h = **new bool**[N];

112 **int** \*cnt = **new int**[N];

113 **int** \*nxt = **new int**[N];

114 memset(b2h, 0, N\***sizeof**(**bool**));

115

116 bh[0] = **true**;

117 **for** (**int** i = 1; i < N; ++i)

118 bh[i] = str[pos[i]] != str[pos[i-1]];

119

120 **for** (**int** h = 1; h < N; h <<= 1) {

121 // bh[i] == false if the first h characters of pos[i-1] == the first h characters of pos[i]

122 **int** buckets = 0;

123 **for** (**int** i = 0; i < N; ) {

124 **int** j = i + 1;

125 **for** (; j < N && !bh[j]; ++j);

126 nxt[i] = j;

127 buckets++;

128 i = j;

129 }

130 **if** (buckets == N) **break**;

131

132 // Suffixes are separated in buckets containing strings starting with the same h characters

133 **for** (**int** i = 0; i < N; i = nxt[i]) {

134 cnt[i] = 0;

135 **for** (**int** j = i; j < nxt[i]; ++j)

136 **rank**[pos[j]] = i;

137 }

138

139 cnt[**rank**[N - h]]++;

140 b2h[**rank**[N - h]] = **true**;

141

142 **for** (**int** i = 0; i < N; i = nxt[i]) {

143 **for** (**int** j = i; j < nxt[i]; ++j) {

144 **int** s = pos[j] - h;

145 **if** (s >= 0) {

146 **int** head = **rank**[s];

147 **rank**[s] = head + cnt[head]++;

148 b2h[**rank**[s]] = **true**;

149 }

150 }

151 **for** (**int** j = i; j < nxt[i]; ++j) {

152 **int** s = pos[j] - h;

153 **if** (s >= 0 && b2h[**rank**[s]]) {

154 **for** (**int** k = **rank**[s]+1; k < N && !bh[k] && b2h[k]; ++k)

155 b2h[k] = **false**;

156 }

157 }

158 }

159

160 **for** (**int** i = 0; i < N; ++i) {

161 pos[**rank**[i]] = i;

162 **if** (b2h[i])

163 bh[i] = **true**;

164 }

165 }

166

167 **for** (**int** i = 0; i < N; ++i)

168 **rank**[pos[i]] = i;

169

170 **delete**[] cnt;

171 **delete**[] nxt;

172 **delete**[] b2h;

173 **delete**[] bh;

174 }

175

176 **void** SuffixArray::build\_lcp() {

177 // Precondition: suffix\_sort() has been called

178 // suffix\_sort();

179 **delete**[] lcp\_len;

180 lcp\_len = **new int**[N];

181 lcp\_len[0] = 0;

182 **for** (**int** i = 0, h = 0; i < N; ++i) {

183 **if** (**rank**[i] > 0) {

184 **int** j = pos[**rank**[i]-1];

185 **for** (; i + h < N && j + h < N && str[i+h] == str[j+h]; ++h);

186 lcp\_len[**rank**[i]] = h;

187 **if** (h > 0)

188 --h;

189 }

190 }

191 }

192

193 **int** SuffixArray::**find**(**const char** \*pat) **const** {

194 **int** M = strlen(pat);

195 **int** lo = 0, hi = N-1;

196 **while** (lo < hi) {

197 **int** mid = lo + (hi-lo)/2;

198 **if** (strncmp(str + pos[mid], pat, M) >= 0)

199 hi = mid;

200 **else**

201 lo = mid+1;

202 }

203 **if** (strncmp(str + pos[lo], pat, M) < 0)

204 **return** N;

205 **return** lo;

206 }

207

208

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

210 SuffixArray sa("abracadabra");

211 // SuffixArray sa("aababbababbaababb");

212 // SuffixArray sa("abbaab");

213 sa.suffix\_sort();

214 sa.build\_lcp();

215 **for** (**int** i = 0; i < sa.N; i++) {

216 fprintf(stderr, "%4d: %3d ", i, sa.lcp\_len[i]);

217 fputc('"', stderr);

218 fputs(sa.str + sa.pos[i], stderr);

219 fputc('"', stderr);

220 fputc('\n', stderr);

221 }

222 fputc('\n', stderr);

223

224 **const char** \*ss[] = {"acabyz", "acaxyz", "0", "xxx", "rat", "rab", "aaa", "a"};

225

226 **for** (**int** j = 0; j < **sizeof**(ss)/**sizeof**(**const char**\*); ++j) {

227 printf("Position of string \"%s\" would be %d\n",

228 ss[j], sa.**find**(ss[j]));

229 }

230

231 **return** 0;

232 }

## KMP (Find the ocurrences of a pattern in a string)

[**https://github.com/cjtoribio/Algorithms/blob/master/Codes/String/KMP.cpp**](https://github.com/cjtoribio/Algorithms/blob/master/Codes/String/KMP.cpp)

1 #include <string>

2 #include <vector>

3 **using namespace std**;

4

5 **struct** KMP

6 {

7 **string** needle;

8 **vector**<**int**> T;

9 KMP(**const string** needle)

10 {

11 **this**->needle = needle;

12 T = **vector**<**int**>(needle.**size**() + 1);

13 **int** i = 0 , j = -1;

14 T[0] = -1;

15 **while**(i < needle.**size**())

16 {

17 **while**(j >= 0 && needle[i] != needle[j])j = T[j];

18 T[++i] = ++j;

19 }

20 }

21 **vector**<**int**> match(**const string** hay)

22 {

23 **vector**<**int**> V;

24 **int** i = 0 , j = 0;

25 **while**(i < hay.**size**())

26 {

27 **while**(j >= 0 && hay[i] != needle[j])j = T[j];

28 ++i;++j;

29 **if**(j == needle.**size**())

30 {

31 V.**push\_back**(i - j);

32 j = T[j];

33 }

34 }

35 **return** V;

36 }

37 }

# Math

## Sieve of Eras

1 //

2 // Prime Sieves

3 //

4 #include <iostream>

5 #include <cmath>

6 #include <vector>

7

8 **using namespace std**;

9

10 **typedef unsigned long long** ullong;

11

12 **vector**<**int**> primes;

13

14 #define MAXP 1000000

15

16 **namespace** SieveEratosthenes1 {

17 // Sieve of Eratosthenes 1: normal

18 **bool** notprime[MAXP+1];

19

20 **void** sieve() {

21 **register unsigned int** p;

22

23 primes.**clear**();

24 primes.reserve((**int**) ceil((MAXP+1)/log(MAXP+1)\*(1 + 1.2762/log(MAXP+1))));

25

26 // memset(notprime, 0, notprime(sieve));

27 notprime[0] = notprime[1] = **true**;

28 **for** (p = 2; ; ++p) {

29 ullong p2 = ((ullong) p) \* ((ullong) p);

30 **if** (p2 > MAXP) **break**;

31 **if** (notprime[p]) **continue**;

32 primes.**push\_back**(p);

33 **for** (ullong k = p2; k <= MAXP; k += p)

34 notprime[k] = **true**;

35 }

36 **for** (; p <= MAXP; ++p) {

37 **if** (notprime[p]) **continue**;

38 primes.**push\_back**(p);

39 }

40 }

41

42 }

43

44

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

46 {

47 SieveEratosthenes1::sieve();

48 **for**(**int** i = 0; i < 10; i++){

49 **cout** << primes[i] << " ";

50 }

51 **return** 0;

52 }