Sumário

Lista de Tabelas

Algoritmos

1 Codigos

1.1 Tricks e confs

Código 1: .vimrc para a configuração do vim

```
set ai noet ts=4 sw=4 bs=2
set cindent
```

Código 2: Speedup cin cout io. Caso usada não use printf

```
std::cout.sync_with_stdio(false);
```

Código 3: Arredondamento e output em outras bases

```
#include <iostream>
#include <iomanip> // setprecision()
using namespace std;

int main () {
  double a = 3.1415926534;
  double b = 2006.0;
  double c = 1.0e-10;

  // setprecision(1) => 1 casa decimal apos a virgula
  cout << fixed << setprecision(1) << 9.09090901 << endl;
  cout << fixed << setprecision(2) << 9.09090901 << endl;
  cout << fixed << setprecision(3) << 9.09090901 << endl;
  cout << fixed << setprecision(2) << 9.1 << endl;

  // anula o efeito de setprecision
  cout.unsetf(ios::floatfield);

  // 5 digitos no maximo
  cout.precision(5);

  cout << a << '\t' << b << '\t' << c << endl;
  cout << fixed << a << '\t' << b << '\t' << c << endl;
  cout << scientific << a << '\t' << b << '\t' << c << endl;

  // Sets the basefield format flag for the str stream to dec, hex or
      oct.
  int n =70;
  cout << dec << n << endl;
  cout << hex << n << endl;
  cout << oct << n << endl;

  return 0;
}
/* output
9.1
9.09
9.091
9.10
3.1416   2006     1e-10
3.14159  2006.00000        0.00000
3.14159e+00      2.00600e+03      1.00000e-10
70
46
106
*/
```

1.2 Exemplos

Código 4: Modelo

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include <inttypes.h>
#include <ctype.h>
#include <limits.h>

#include <algorithm>
#include <utility>
#include <iostream>

#include <map>
#include <set>
#include <vector>
#include <list>
#include <queue>
#include <sstream>

using namespace std;

#define abs(a) ((a) > 0 ? (a) : -(a))

int main()
{
    int n;

    cin >> n;

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

    }

    while (cin >> n)
    {

    }
    return 0;
}
```

Código 5: comparcao de ponto flutuante

```
/**
 * -1 se x < y
 * 0 se x = y
 * 1 se x > y
 */
const double EPS = 1e-10;
#define _inline(f...) f() __attribute__((always_inline)); f
_inline(int cmp)(double x, double y = 0, double tol = EPS)
{
    return (x <= y + tol) ? (x + tol < y) ? -1 : 0 : 1;
}
```

1.3 Teoria dos números

Código 6: Máximo divisor comum e mínimo multiplo comum

```
int gcd(int x, int y)
{
    return y ? gcd(y, x % y) : abs(x);
}
uint64_t lcm(int x, int y)
{
    if (x && y) return abs(x) / gcd(x, y) * uint64_t(abs(y));
    else return uint64_t(abs(x | y));
}
```

Código 7: Teste (ineficiente) de primalidade

```
bool isPrime(int n)
{
    if (n < 0) return isPrime(-n);
    if (n == 1) return true;
    if (n < 5 || n % 2 == 0 || n % 3 == 0) return (n == 2 || n == 3);

    int maxP = sqrt(n) + 2;
    for (int p = 5; p < maxP; p += 6)
    {
       if (n % p == 0 || n % (p+2) == 0) return false;
    }
    return true;
}
```

Código 8: Fatoração em números primos.

```
typedef map<int, int> prime_map;
void squeeze(prime_map& M, int& n, int p)
{
    for (; n % p == 0; n /= p) M[p]++;
}
void factor(int n, prime_map& M)
{
    if (n < 0) { factor(-n, M); return; }
    if (n < 2) return;

    squeeze(M, n, 2);
    squeeze(M, n, 3);

    int maxP = sqrt(n) + 2;
    for (int p = 5; p < maxP; p += 6)
    {
       squeeze(M, n, p);
       squeeze(M, n, p+2);
    }
    if (n > 1) M[n]++;
}
```

Código 9: $a^b mod$ n de forma rápida.

```
int mpow(int a, int b, int n = 10)
{
  if(b == 0)
    return 1;
  else {
    long long res = mpow(a, b/2, n);
    res = (res*res) % n;
    if(b%2 == 1)
      res = (res*a) % n;
    return (int) res;
  }
}
```

Código 10: (a*b) mod c rápido.

```
long long mulmod(long long a, long long b, long long c)
{
    long long x = 0;
    long long y = a % c;
    while(b > 0)
    {
       if(b & 1ll) x = (x + y) % c;
       y = (y << 1) % c;
       b >>= 1;
    }
    return x % c;
}
```

Código 11: Computa x tal que a*x = b (mod c). Quando a equação não tem solução, retorna algum valor arbitrário errado, mas basta conferir o resultado.

```
long long axbmodc(long long a, long long b, long long c)
{
    return a ? (axbmodc(c % a, (a - b % a) % a, a) * c + b) / a : 0;
}
```

Código 12: **Baby-step Giant-step algorithm** Calcula o menor valor de e para $b^e = n \mod p$. Retorna -1 se eh impossivel

```
#define inv_mult( a, n ) axbmodc(a, 1, n)

long long discreteLlogarithm( long long b, long long n, long long p )
{
    if ( n == 1 ) return 0;

    map < long long, int > table;
    long long m = sqrt(p) + 1, pot = 1, pot2 = 1;

    for (int j = 0; j < m; j++)
    {
        if ( pot == n ) return j;
        table[( n * inv_mult( pot, p ) ) % p] = j;
        pot = ( pot * b ) % p;
    }

    for (int i = 0; i < m; i++)
    {
        if ( table.find( pot2 ) != table.end() ) return i * m + table[pot2
            ];
        pot2 = ( pot * pot2 ) % p;
    }

    return -1;
}
```

1.4 Estruturas de dados

Código 13: Números de precisão harbitrária (bigint).

```
const int DIG = 4;
const int BASE = 10000; // BASE**3 < 2**51
const int TAM = 1000;

struct BigInt
{
    int num[TAM], numDigits;
    BigInt(int x = 0): numDigits(1)
    {
        memset(num, 0, sizeof(num));
        num[numDigits++] = x; fixInvariant();
    }
    BigInt(char *s): numDigits(1)
    {
        memset(num, 0, sizeof(num));
        int sign = 1;

        while (*s && !isdigit(*s))
        {
            if (*s++ == '-') sign *= -1;
        }

        char *t = strdup(s), *p = t + strlen(t);

        while (p > t)
        {
            *p = 0; p = max(t, p - DIG);
            sscanf(p, "%d", &num[numDigits]);
            num[numDigits++] *= sign;
        }

        free(t);
        fixInvariant();
    }

    BigInt& fixInvariant(int m = 0)
    {
        numDigits = max(m, numDigits);
        int sign = 0;

        for (int i = 1, carry = 0; i <= numDigits || carry && (numDigits =
             i); i++)
        {
            num[i] += carry;
            carry = num[i] / BASE;
            num[i] %= BASE;
            if (num[i]) sign = (num[i] > 0) ? 1 : -1;
        }

        for (int i = 1; i < numDigits; i++)
        {
            if (num[i] * sign < 0)
            {
                num[i] += sign * BASE;
                num[i+1] -= sign;
            }
        }

        while (numDigits && !num[numDigits]) numDigits--;
        return *this;
    }

    //Comparacao
    int cmp(const BigInt& x = 0) const
    {
        int i = max(numDigits, x.numDigits), t = 0;
        while (1)
        {
            if ((t = ::cmp(num[i], x.num[i])) || i-- == 0) return t;
        }
    }

    bool operator <(const BigInt& x) const { return cmp(x) < 0; }
    bool operator >(const BigInt& x) const { return cmp(x) > 0; }
    bool operator <=(const BigInt& x) const { return cmp(x) <= 0; }
    bool operator >=(const BigInt& x) const { return cmp(x) >= 0; }
    bool operator ==(const BigInt& x) const { return cmp(x) == 0; }
```

```
77    bool operator !=(const BigInt& x) const { return cmp(x) != 0; }
78
79    //operacoes fundamentais
80    BigInt& operator +=(const BigInt& x)
81    {
82       for (int i = 1; i <= x.numDigits; i++) num[i] += x.num[i];
83       return fixInvariant(x.numDigits);
84    }
85    BigInt& operator -=(const BigInt& x)
86    {
87       for (int i = 1; i <= x.numDigits; i++) num[i] -= x.num[i];
88       return fixInvariant(x.numDigits);
89    }
90
91    void multiAndAcumWithShift(const BigInt& x, int m, int b)
92    { // *this += (x * m) << b;
93       for (int i = 1, carry = 0; (i <= x.numDigits || carry) && (
              numDigits = i + b); i++)
94       {
95          num[i+b] += x.num[i] * m + carry;
96          carry = num[i+b] / BASE;
97          num[i+b] %= BASE;
98       }
99    }
100
101   BigInt operator *(const BigInt& x) const
102   {
103      BigInt r;
104      for (int i = 1; i <= numDigits; i++) r.multiAndAcumWithShift(x,
              num[i], i-1);
105      return r;
106   }
107
108   BigInt div(const BigInt& x)
109   {
110      if (x == 0) return 0;
111
112      BigInt q;
113      q.numDigits = max(numDigits - x.numDigits + 1, 0);
114      int d = x.num[x.numDigits] * BASE + x.num[x.numDigits-1];
115
116      for (int i = q.numDigits; i > 0; i--)
117      {
118         int j = x.numDigits + i - 1;
119         q.num[i] = int((num[j] * double(BASE) + num[j-1]) / d);
120         multiAndAcumWithShift(x, -q.num[i], i-1);
121         if (i == 1 || j == 1) break;
122         num[j-1] += BASE * num[j];
123         num[j] = 0;
124      }
125
126      fixInvariant(x.numDigits);
127      return q.fixInvariant();
128   }
129
130   BigInt& operator *=(const BigInt& x) { return *this = (*this) * x; }
131   BigInt operator +(const BigInt& x) { return BigInt(*this) += x; }
132   BigInt operator -(const BigInt& x) { return BigInt(*this) -= x; }
133   BigInt operator -() { BigInt r = 0; return r -= *this; }
134   BigInt& operator /=(const BigInt& x) { return *this = div(x); }
135   BigInt& operator %=(const BigInt& x) { div(x); return *this; }
136   BigInt operator /(const BigInt& x) { return BigInt(*this).div(x); }
137   BigInt operator %(const BigInt& x) { return BigInt(*this) %= x; }
138
139   // I/O
140   operator string() const
141   {
142      ostringstream s; s << num[numDigits];
143      for (int i = numDigits - 1; i > 0; i--)
144      {
145         s.width(DIG);
146         s.fill('0');
147         s << abs(num[i]);
148      }
149
150      return s.str();
151   }
152
153   friend ostream& operator <<(ostream& o, const BigInt& x)
154   {
155      return o << (string) x;
156   }
157
158   friend istream& operator >>(istream& in, BigInt& x)
159   {
160      string num;
161      in >> num;
162      x = BigInt((char*) num.c_str());
163      return in;
164   }
165
166   // potencia e raiz
167   BigInt pow(int x)
168   {
169      if (x < 0) return (*this == 1 || *this == -1) ? pow(-x) : 0;
170      BigInt r = 1;
171      for (int i = 0; i < x; i++) r *= *this;
172      return r;
173   }
174
175   BigInt root(int x)
176   {
177      if (cmp() == 0 || cmp() < 0 && x % 2 == 0) return 0;
178      if (*this == 1 || x == 1) return *this;
179      if (cmp() < 0) return -(-*this).root(x);
180      BigInt a = 1, d = *this;
181      while (d != 1)
182      {
183         BigInt b = a + (d /= 2);
184         if (cmp(b.pow(x)) >= 0) { d += 1; a = b; }
185      }
186
```

```
        return a;
    }
};
```

1.5 Programação Dinâmica

Código 14: **Sub Set Sum**: Verifica se há um sobconjunto dos elementos do vetor cuja soma seja igual a soma pedida.

```
//soma maxima dos elementos do vetor
#define MAX_SUM 10000
int n;
int vet[TAM];
bool m[MAX_SUM];

//M->soma maxima dos elementos do vetor c->soma procurada
bool subSetSum(int M, int c)
{
    for (int i = 0; i <= M; i++) m[i] = false;
    m[0] = true;

    for(int i = 0; i < n; i++)
    {
        for(int j = M; j >= vet[i]; j--)
        {
            m[j] |= m[j - vet[i]];
        }
    }

    return m[c];
}
```

Código 15: **Lis: longest increasing (decreasing) subsequence** O(n^2)

```
#define TAM 10000
int c[TAM];
int A[TAM];
int H[TAM];

void ssctf(int n)
{
    for (int m = 1; m <= n; m++)
    {
        c[m] = H[m];
        for (int i = m -1; i > 0; i--)
        {
            if (A[i] < A[m] && c[i] + H[m] > c[m])
            {
                c[m] = c[i] + H[m];
            }
        }
    }
}

void ssdtf(int n)
{
    for (int m = 1; m <= n; m++)
    {
        c[m] = H[m];
        for (int i = m -1; i > 0; i--)
        {
            if (A[i] > A[m] && c[i] + H[m] > c[m])
            {
                c[m] = c[i] + H[m];
            }
        }
    }
}

int lis1d(int n, bool inc = true)
{
    if (inc) ssctf(n);
    else ssdtf(n);

    int max = 0;

    for (int i = 1; i <= n; i++)
        if (max < c[i])
            max = c[i];

    return max;
}
```

Código 16: **Lis: longest increasing subsequence** O(n*logn)

```
// Longest Increasing Subsequence - LIS O(n log n)
#define fori(i, n) for ( int i = 0; i < (n); ++i )
void lis( const vector< int > & v, vector< int > & asw )
{
    vector<int> pd(v.size(),0), pd_index(v.size()), pred(v.size());
    int maxi = 0, x, j, ind;

    fori(i,v.size())
    {
        x = v[i];
        j = lower_bound( pd.begin(), pd.begin() + maxi, x ) - pd.begin();
        //j = upper_bound( pd.begin(), pd.begin() + maxi, x ) - pd.begin()
            ; para lds
        pd[j] = x;
        pd_index[j] = i;
        if( j == maxi ) { maxi++; ind = i; }
        pred[i] = j ? pd_index[j-1] : -1;
    }
    // return maxi; se a sequencia nao precisa ser refeita

    int pos = maxi-1, k = v[ind];
    asw.resize( maxi );

    while ( pos >= 0 )
```

```
    {
        asw[pos--] = k;
        ind = pred[ind];
        k = v[ind];
    }
}
```

Código 17: Problema da Mochila O(n*W)

```
#include <stdio.h>

#define MAXWEIGHT 100

int n = 3; /* The number of objects */
int c[10] = {8, 6, 4}; /* c[i] is the *COST* of the ith object; i.e. what
                          YOU PAY to take the object */
int v[10] = {16, 10, 7}; /* v[i] is the *VALUE* of the ith object; i.e.
                            what YOU GET for taking the object */
int W = 10; /* The maximum weight you can take */

void fill_sack() {
    int a[MAXWEIGHT]; /* a[i] holds the maximum value that can be obtained
                         using at most i weight */
    int last_added[MAXWEIGHT]; /* I use this to calculate which object were
                                  added */
    int i, j;
    int aux;

    for (i = 0; i <= W; ++i) {
        a[i] = 0;
        last_added[i] = -1;
    }

    a[0] = 0;
    for (i = 1; i <= W; ++i)
        for (j = 0; j < n; ++j)
            if ((c[j] <= i) && (a[i] < a[i - c[j]] + v[j])) {
                a[i] = a[i - c[j]] + v[j];
                last_added[i] = j;
            }

    for (i = 0; i <= W; ++i)
        if (last_added[i] != -1)
            printf("Weight %d; Benefit: %d; To reach this weight I added object %d (%d$ %dKg) to weight %d.\n", i, a[i], last_added[i] + 1, v[last_added[i]], c[last_added[i]], i - c[last_added[i]]);
        else
            printf("Weight %d; Benefit: 0; Can't reach this exact weight.\n", i);

    printf("---\n");

    aux = W;
    while ((aux > 0) && (last_added[aux] != -1)) {
        printf("Added object %d (%d$ %dKg). Space left: %d\n", last_added[aux] + 1, v[last_added[aux]], c[last_added[aux]], aux - c[last_added[aux]]);
        aux -= c[last_added[aux]];
    }

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

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

    return 0;
}
```

1.6 Grafos

Código 18: Verifica se o grafo é aciclico.

```
#define TAM 100
#define BRANCO 0
#define CINZA 1
#define PRETO 2
bool grafo[TAM][TAM];
int pass[TAM];

bool dfs(int v)
{
    pass[v] = CINZA;

    for (int i = 0; i < TAM; i++)
    {
        if (grafo[v][i])
        {
            if (pass[i] == CINZA) return false;
            if (pass[i] == BRANCO && !dfs(i)) return false;
        }
    }

    pass[v] = PRETO;
    return true;
}

bool aciclico()
{
    memset(pass, BRANCO, TAM*sizeof(int));

    for (int i = 0; i < TAM; i++)
    {
        if (pass[i] == BRANCO)
        {
            if (!dfs(i)) return false;
```

```
        }
    }

    return true;
}
```

Código 19: Caminho minimo **Dijkstra**.

```
#include <queue>

typedef vector<map<int, int> > AdjList;
typedef AdjList Grafo;

int dist[MAX_VERTICES];
int prev[MAX_VERTICES]; // para recuperar o caminho usando um dijoint
    forest set

void dijkstra(Grafo& grafo, int source)
{
    for (int i = 0; i < grafo.size(); i++)
    {
        dist[i] = INF;
        prev[i] = -1;
    }

    dist[source] = 0;
    priority_queue<pair<int, int> > heap;
    heap.push(make_pair(0, source));

    while (!heap.empty())
    {
        int u = heap.top().second;
        heap.pop();

        // para cada vizinho de u
        for (map<int,int>::iterator i = grafo[u].begin(); i != grafo[u].
            end(); i++)
        {
            int totalDist = dist[u] + (*i).second;
            if (totalDist <= dist[(*i).first])
            {
                dist[(*i).first] = totalDist;
                heap.push(make_pair(totalDist, (*i).first));
                prev[(*i).first] = u;
            }
        }
    }
}
```

Código 20: Floresta dijunta de arvores

```
#define SIZE 100

struct dsf
{
    int element_count;
    int parent[SIZE];
    int rank[SIZE];
};
typedef struct dsf * disjoint_set_forest_p;
typedef struct dsf disjoint_set_forest;

void dsf_init(disjoint_set_forest_p forest, int element_count)
{
    forest->element_count = element_count;
    memset(forest->parent, 0, element_count*sizeof(int));
    memset(forest->rank, 0, element_count*sizeof(int));

    for (int i = 0; i < element_count; ++i)
        forest->parent[i] = i;
}

int dsf_find_set(disjoint_set_forest_p forest, int i)
{
    if (i != forest->parent[i])
    {
        forest->parent[i] = dsf_find_set(forest, forest->parent[i]);
    }
    return forest->parent[i];
}

void dsf_union(disjoint_set_forest_p forest, int i, int j)
{
    int x = dsf_find_set(forest, i);
    int y = dsf_find_set(forest, j);

    if (forest->rank[x] > forest->rank[y])
    {
        forest->parent[y] = x;
    }
    else
    {
        forest->parent[x] = y;
        if (forest->rank[x] == forest->rank[y])
        {
            forest->rank[y]++;
        }
    }
}
```

Código 21: Arvore geradora mínima **Kruskal**

```
typedef vector<map<int, int> > AdjList;
struct Grafo
{
    int edgeCnt;
    AdjList adj;
};

struct edge
{
```

```
10      int u;
11      int v;
12      int weight;
13  };
14
15  int edge_compare(const void * e1, const void * e2)
16  {
17      struct edge * p1 = (struct edge *) e1;
18      struct edge * p2 = (struct edge *) e2;
19      int f = p1->weight - p2->weight;
20      if (f < 0)
21      {
22         return -1;
23      }
24      else  if (f == 0)
25      {
26          return edge_compare1(e1, e2);
27      }
28      else
29      {
30          return 1;
31      }
32  }
33
34  struct edge * get_edge_list(Grafo& graph)
35  {
36      int edge_count = graph.edgeCnt;
37      struct edge *edges = (struct edge*) malloc(edge_count * sizeof(
            struct edge));
38
39      int current_edge = 0;
40
41      for (int i = 0; i < graph.adj.size(); ++i)
42      {
43          for (map<int, int>::iterator j = graph.adj[i].begin(); j !=
                graph.adj[i].end(); j++)
44          {
45              struct edge e;
46              e.u = i < (*j).first ? i : (*j).first;
47              e.v = i > (*j).first ? i : (*j).first;
48              e.weight = (*j).second;
49              edges[current_edge++] = e;
50          }
51      }
52
53      return edges;
54  }
55
56  void kruskal(Grafo& graph, Grafo& mst)
57  {
58      // Obtain a list of edges and sort it by weight in O(E lg E) time
59      int edge_count = graph.edgeCnt;
60      struct edge *edges = get_edge_list(graph);
61      qsort(edges, edge_count, sizeof(struct edge), edge_compare);
62
63      disjoint_set_forest dsf;
64      dsf_init(&dsf, edge_count);
65
66       for (int i = 0; i < edge_count; ++i)
67       {
68           struct edge e = edges[i];
69           int uset = dsf_find_set(&dsf, e.u);
70           int vset = dsf_find_set(&dsf, e.v);
71           if (uset != vset)
72           {
73               mst.adj[e.u][e.v] = e.weight;
74            mst.edgeCnt++;
75               dsf_union(&dsf, uset, vset);
76           }
77       }
78
79       free(edges);
80  }
```

Código 22: Verifica se um grafo é bipartido

```
1  #define TAM 200
2
3  bool grafo[TAM][TAM];
4  int pass[TAM];
5  int n;
6
7  bool bipartido(int v, int color = 1)
8  {
9      pass[v] = color;
10     int thisColor = color;
11     bool ret = true;
12
13     color = color == 1 ? 2 : 1;
14
15     for (int i = 0; i < n; i++)
16     {
17        if (grafo[v][i])
18        {
19           if (!pass[i]) ret = bipartido(i, color);
20           else if (pass[i] == thisColor) return false;
21
22           if (!ret) return false;
23        }
24     }
25
26     return ret;
27 }
```

Código 23: Ordenação topológica de um grafo acíclico

```
1  #define UNVISITED -1
2
3  int grafo[SIZE][SIZE];
4  int prof[SIZE];
5  int sorted[SIZE];
```

```
int nordem;

void dfsTopsort(int no)
{
    for (int viz = 0; viz < SIZE; viz++)
    {
        if (grafo[no][viz])
        {
            if (prof[viz] == UNVISITED)
            {
                prof[viz] = prof[no] + 1;
                dfsTopsort(viz);
            }
        }
    }

    sorted[nordem--] = no;
}

void topSort(int nvt)
{
    memset(prof, UNVISITED, nvt*sizeof(int));
    nordem = nvt - 1;

    for (int i = 0; i < nvt; i++)
    {
        if (prof[i] == UNVISITED)
        {
            prof[i] = 0;
            dfsTopsort(i);
        }
    }
}
```

Código 24: Ordenação topológica de um grafo acíclico

```
bool topologicalSort(vector< vector< int > > &g, vector< int > &r)
{
    vector< int > deg(g.size());
    FOREACH(node, g){
        FOREACH(ngb, *node){
            deg[*ngb]++;
        }
    }

    priority_queue< int, vector< int >, greater< int > > q;
    FORN(i, 0, g.size())
        if(0 == deg[i]) q.push(i);

    while(not q.empty()){
        int node = q.top();
        q.pop();
        r.push_back(node);
        FOREACH(ngb, g[node]){
            if(--deg[*ngb] == 0) q.push(*ngb);
        }
    }

    return r.size() == g.size();
}
```

Código 25: Fluxo máximo, **Ford-Fulkerson**

```
#define TAM 1000
#define MAX_INT 1000000

int grafo[TAM][TAM];
int pred[TAM];
int f[TAM][TAM];
bool visitados[TAM];
int fila[TAM];

bool bfs(int n, int ini, int fim)
{
    int no, s = 0, e = 0;
    fila[e++] = ini;

    while (s != e)
    {
        no = fila[s++];

        if (visitados[no] == 2) continue;
        visitados[no] = 2;

        for (int i = 0; i < n; i++)
        {
            if (visitados[i] < 2)
            {
                if(grafo[no][i] - f[no][i] > 0)
                {
                    pred[i] = no;
                    if (i == fim) return true;
                    if(visitados[i] == 0)
                    {
                        fila[e++] = i;
                        visitados[i] = 1;
                    }
                }
            }
        }
    }

    return false;
}

bool dfs(int s, int t, int size)
{
    visitados[s] = true;
    if(s == t) return true;

    for(int v = 0; v < size; v++)
    {
```

```
50          if(!visitados[v] && grafo[s][v] - f[s][v] > 0)
51          {
52              pred[v] = s;
53              if(dfs(v, t, size)) return true;
54          }
55      }
56
57      return false;
58  }
59
60  bool findPath(int s, int t, int size)
61  {
62      memset(visitados, false, sizeof(bool)*size);
63      pred[s] = s;
64      // Aqui pode ser usado tanto busca em largura quanto em profundidade.
65      // busca em largura geralmente apresenta tempos de execucao bem
            menores.
66      return bfs(size, s, t);
67      //return dfs(s, t, size);
68  }
69
70  int maxFlow(int size, int s, int t)
71  {
72      int delta;
73
74      for(int i = 0; i < size; i++)
75      {
76          memset(f[i], 0, sizeof(int)*size);
77      }
78
79      while(1)
80      {
81          bool path = findPath(s, t, size);
82          if (!path) break;
83
84          delta = MAX_INT;
85          for(int c = t; pred[c] != c; c = pred[c])
86          {
87              delta = min(delta, grafo[pred[c]][c] - f[pred[c]][c]);
88          }
89
90          for(int c = t; pred[c] != c; c = pred[c])
91          {
92              f[pred[c]][c] += delta;
93              f[c][pred[c]] -= delta;
94          }
95      }
96
97      int soma = 0;
98
99      for(int i = 0; i < size; i++)
100     {
101         soma += f[i][t];
102     }
103
104     return soma;
105 }
```

Código 26: Fluxo máximo (mais eficiente)

```
1   const int VT = 100;
2   const int AR = VT * VT;
3
4   struct grafo
5   {
6       // lista de adjacencias representada na forma de vetor
7       int nvt, nar;
8       int dest[2 * AR];
9       int adj[VT][2 * VT];
10      int nadj[VT];
11
12      int cap[AR]; // capacidade do arco
13      int fluxo[AR];
14      int ent[VT];
15
16      int padj[VT], lim[VT], nivel[VT], qtd[VT];
17
18      int inv(int a) { return a ^ 0x1; }
19      int orig(int a) { return dest[inv(a)]; }
20      int capres(int a) { return cap[a] - fluxo[a]; }
21
22      void inic(int n = 0)
23      {
24          nvt = n;
25          nar = 0;
26          memset(nadj, 0, sizeof(nadj));
27      }
28
29      ////////////////////////////////////////////////////////
30      // Adiciona uma aresta ao grafo.
31      //
32      // "int u" apenas para Fluxos;
33      //
34      int aresta(int i, int j, int u = 0)
35      {
36          int ar = nar;
37          cap[nar] = u;
38          dest[nar] = j;
39          adj[i][nadj[i]] = nar++;
40          nadj[i]++;
41
42          cap[nar] = 0;
43          dest[nar] = i;
44          adj[j][nadj[j]] = nar++;
45          nadj[j]++;
46          return ar;
47      }
48
49      void revbfs(int ini, int fim)
50      {
51          int i, no, viz, ar;
52          queue<int> fila;
```

```
53
54      memset(nivel, NULO, sizeof(nivel));
55      memset(qtd, 0, sizeof(qtd));
56
57      nivel[fim] = 0;
58      fila.push(fim);
59
60      while (!fila.empty())
61      {
62          no = fila.front();
63          fila.pop();
64          qtd[nivel[no]]++;
65
66          for (i = 0; i < nadj[no]; i++)
67          {
68              ar = adj[no][i];
69              viz = dest[ar];
70
71              if (cap[ar] == 0 && nivel[viz] == NULO)
72              {
73                  nivel[viz] = nivel[no] + 1;
74                  fila.push(viz);
75              }
76          }
77      }
78  }
79
80  int admissivel(int no)
81  {
82      while (padj[no] < nadj[no])
83      {
84          int ar = adj[no][padj[no]];
85          if (nivel[no] == nivel[dest[ar]] + 1 && capres(ar) > 0) return
                ar;
86          padj[no]++;
87      }
88
89      padj[no] = 0;
90      return NULO;
91  }
92
93  int retrocede(int no)
94  {
95      int i, ar, viz, menor = NULO;
96
97      if (--qtd[nivel[no]] == 0) return NULO;
98
99      for (i = 0; i < nadj[no]; i++)
100     {
101         ar = adj[no][i]; viz = dest[ar];
102         if (capres(ar) <= 0) continue;
103         if (menor == NULO || nivel[viz] < nivel[menor]) menor = viz;
104     }
105
106     if (menor != NULO) nivel[no] = nivel[menor];
107     qtd[++nivel[no]]++;
108
109     return ((ent[no] == NULO) ? no : orig(ent[no]));
110 }
111
112 int avanca(int no, int ar)
113 {
114     int viz = dest[ar];
115     ent[viz] = ar;
116     lim[viz] = min(lim[no], capres(ar));
117     return viz;
118 }
119
120 int aumenta(int ini, int fim)
121 {
122     int ar, no = fim, fmax = lim[fim];
123
124     while (no != ini)
125     {
126         fluxo[ar = ent[no]] += fmax;
127         fluxo[inv(ar)] -= fmax;
128         no = orig(ar);
129     }
130
131     return fmax;
132 }
133
134 int maxflow(int ini, int fim)
135 {
136     int ar, no = ini, fmax = 0;
137
138     memset(fluxo, 0, sizeof(fluxo));
139     memset(padj, 0, sizeof(padj));
140
141     revbfs(ini, fim);
142
143     lim[ini] = INF;
144     ent[ini] = NULO;
145
146     while (nivel[ini] < nvt && no != NULO)
147     {
148         if ((ar = admissivel(no)) == NULO)
149         {
150             no = retrocede(no);
151         }
152         else if ((no = avanca(no, ar)) == fim)
153         {
154             fmax += aumenta(ini, fim);
155             no = ini;
156         }
157     }
158     return fmax;
159 }
160 };
```

1.7 Geometria

Código 27: Ponto e poligono

```
struct point
{
    double x, y;
    double z; // para pontos no espaco
    point(double x = 0, double y = 0, double z = 0): x(x), y(y), z(z) {}

    point operator +(point q) { return point(x + q.x, y + q.y, z + q.z);
        }
    point operator -(point q) { return point(x - q.x, y - q.y, z - q.z);
        }
    point operator *(double t) { return point(x * t, y * t, z * t); }
    point operator /(double t) { return point(x / t, y / t, z / t); }
    double operator *(point q) { return x * q.x + y * q.y + z * q.z; }
    point vec(point q) { return point(y * q.z - z * q.y, z * q.x - x * q.
        z, x * q.y - y * q.x); }
    double operator %(point q) { return x * q.y - y * q.x; }

    int cmp(point q) const
    {
       if (int t = ::cmp(x, q.x)) return t;
       else if (int t = ::cmp(y, q.y)) return t;
       return ::cmp(z, q.z);
    }

    bool operator ==(point q) const { return cmp(q) == 0; }
    bool operator !=(point q) const { return cmp(q) != 0; }
    bool operator < (point q) const { return cmp(q) < 0; }

    friend ostream& operator <<(ostream& o, point p) {
      return o << "(" << p.x << ", " << p.y << ", " << p.z << ")";
    }
    static point pivot;
};

// para pontos 2D
double abs(point p) { return hypot(p.x, p.y); }
double arg(point p) { return atan2(p.y, p.x); }

point point::pivot;

typedef vector<point> polygon;

int ccw(point p, point q, point r)
{
    return cmp((p - r) % (q - r));
}

double angle(point p, point q, point r)
{
    point u = p - q, v = r - q;
    return atan2(u % v, u * v);
}
```

Código 28: Decide se q está sobre o segmento fechado pr.

```
bool between(point p, point q, point r)
{
    return ccw(p, q, r) == 0 && cmp((p - q) * (r - q)) <= 0;
}
```

Código 29: Decide se os segmentos fechados pq e rs têm pontos em comum.

```
bool seg_intersect(point p, point q, point r, point s)
{
    point A = q - p;
    point B = s - r;
    point C = r - p;
    point D = s - q;

    int a = cmp(A % C) + 2 * cmp(A % D);
    int b = cmp(B % C) + 2 * cmp(B % D);

    if (a == 3 || a == -3 || b == 3 || b == -3) return false;
    if (a || b || p == r || p == s || q == r || q == s) return true;

    int t = (p < r) + (p < s) + (q < r) + (q < s);
    return t != 0 && t != 4;
}
```

Código 30: Calcula a distância do ponto r ao segmento pq.

```
double seg_distance(point p, point q, point r)
{
    point A = r - q;
    point B = r - p;
    point C = q - p;

    double a = A * A, b = B * B, c = C * C;

    if (cmp(b, a + c) >= 0) return sqrt(a);
    else if (cmp(a, b + c) >= 0) return sqrt(b);
    else return fabs(A % B) / sqrt(c);
}
```

Código 31: Classifica o ponto p em relação ao polígono T. Retorna 0, -1 ou 1 dependendo se p está no exterior, na fronteira ou no interior de T, respectivamente.

```
int in_poly(point p, polygon& T)
{
    double a = 0;
    int N = T.size();
    for (int i = 0; i < N; i++)
    {
       if (between(T[i], p, T[(i+1) % N])) return -1;
       a += angle(T[i], p, T[(i+1) % N]);
    }
    return cmp(a) != 0;
```

```
}
```

Código 32: Convex Hull **graham scan**.

```
#define INF 1e9
#define EPS 1e-9

int cmp(double a, double b = 0.0) {
    return a+EPS < b ? -1 : a-EPS > b;
}

struct Point {
    double x, y;
    Point(double a=0.0,double b=0.0){x=a,y=b;}
    Point operator+(const Point &P) const {return Point(x+P.x,y+P.y);}
    Point operator-(const Point &P) const {return Point(x-P.x,y-P.y);}
    Point operator*(double c) const {return Point(x*c,y*c);}
    Point operator/(double c) const {return Point(x/c,y/c);}
    double operator!() const {return sqrt(x*x+y*y);}
    bool operator==(const Point &p) const {return !cmp(x,p.x) && !cmp(y,p.y);}
    bool operator<(const Point &p) const {if (cmp(x,p.x)) return cmp(x,p.x) < 0; return cmp(y,p.y) < 0;}
    void print(string prefix = "") const {printf("%s%.3lf %.3lf\n",prefix.c_str(),x,y);}
};

typedef vector<Point> Polygon;

double cross(Point A, Point B) {
    return A.x*B.y - B.x*A.y;
}

Point pmin;

bool lessThan(Point A, Point B) {
    if (cmp(cross(A-pmin,B-pmin))) return cmp(cross(A-pmin,B-pmin)) > 0;
    return cmp(!(pmin-A),!(pmin-B)) < 0;
}

int sort(Polygon &p) {
    int imin = 0, i, j, n = p.size();

    for (i=1; i < p.size(); i++) {
        if (p[i] < p[imin]) imin = i;
    }

    swap(p[0],p[imin]);
    pmin = p[0];

    sort(p.begin()+1,p.end(),lessThan);

    for (i=n-1; i > 0 && !cmp(cross(p[i]-p[0],p[i-1]-p[0])); i--);
    if (i == 0) return 1;
    for (j=0; j < (n-i)/2; j++) swap(p[i+j],p[n-j-1]);
    return 0;
}

Polygon convex_hull(Polygon &p) { // tirar o & para nao alterar o poligono original
    int hs = 2, n = p.size();
    Polygon hull;

    if (p.size() < 3) return p;

    int isline = sort(p);
    hull.push_back(p[0]), hull.push_back(p[1]);

    for (int i=2; i <= n-isline; i++) {
        while (hull.size() > 1 && cmp(cross(p[i%n]-hull[hs-1],hull[hs-2]-hull[hs-1])) <= 0) { // trocar o <= para < para manter pontos intermediarios
            hull.pop_back(), hs--;
        }
        if (i < n) hull.push_back(p[i]), hs++;
    }

    return hull;
}

int main() {
    Polygon p;

    while (...) {
        double x = ...;
        double y = ...;
        p.push_back(Point(x,y));
    }

    Polygon hull = convex_hull(p);

    for (int i=0; i < hull.size(); i++) {
        ...
    }
    return 0;
}
```

1.8 Algebra Linear

Código 33: Simplex

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

#define fori(i,n) for(int i=0; i < (n); ++i)
#define forr(i,a,b) for(int i=(a); i <= (b); ++i)
#define ford(i,a,b) for(int i=(a); i >= (b); --i)
```

```cpp
#define sz size()

const double EPS=1e-9; const int INF = 0x3f3f3f3f;

#define all(x) (x).begin(),(x).end()

int cmpD(double x, double y=0, double tol=EPS) {
   return (x <= y+tol) ? (x+tol<y) ? -1 : 0 : 1;
}

struct simplex {
   // max c * x, s.t: A * x <= b; x >= 0
   simplex( const vector< vector< double > > & A_, const vector< double > & b_,
         const vector< double > & c_ ) : A( A_ ), b( b_ ), c( c_ ) {}
   vector< vector< double > > A; vector< double > b, c, sol;
   vector< bool > N; vector< int > kt; int m, n;
   void pivot( int k, int l, int e ) {
      int x = kt[l]; double p = A[l][e];
      fori(i,k) A[l][i] /= p;
      b[l] /= p; N[e] = false;
      fori(i,m) if (i != l) {b[i] -= A[i][e]*b[l]; A[i][x] = -A[i][e]*A[l][x];}
      fori(j,k) if ( N[j] ) {
         c[j] -= c[e] * A[l][j];
         fori(i,m) if ( i != l ) A[i][j] -= A[i][e] * A[l][j];
      }
      kt[l] = e; N[x] = true; c[x] = -c[e] * A[l][x];
   }
   vector< double > go( int k ) {
      vector< double > res;
      while ( 1 ) {
         int e = -1, l = -1;
         fori(i,k) if ( N[i] && cmpD( c[i] ) > 0 ) { e = i; break; }
         if ( e == -1 ) break;
         fori(i,m) if ( cmpD(A[i][e]) > 0 && ( l == -1 || cmpD( b[i] / A[i][e],
                  b[l] / A[l][e], 1e-20 ) < 0 ) ) l = i;
         if ( l == -1 ) return vector< double >(); // unbounded
         pivot( k, l, e );
      }
      res.resize( k, 0 );
      fori(i,m) res[kt[i]] = b[i];
      return res;
   }
   vector< double > solve() {
      m = A.sz; n = A[0].sz; int k = m+n+1;
      N = vector< bool >( k, true ); vector< double > c_copy = c;
      c.resize(n+m); kt.resize(m);
      fori(i,m) {
         A[i].resize(k); A[i][n+i] = 1; A[i][k-1] = -1;
         kt[i] = n+i; N[kt[i]] = false;
      }
      int l = min_element(all(b)) - b.begin();
      if(cmpD(b[l] ) < 0 ) {
         c = vector<double>(k,0);
         c[k-1] = -1; pivot(k, 1, k-1); sol=go(k);
         if(cmpD(sol[k-1])>0) return vector<double>(); // infeasible
         fori(i,m) if(kt[i] == k-1) {
            fori(j,k-1) if(N[j] && cmpD( A[i][j] ) != 0 ) {
               pivot( k, i, j); break;
            }
         }
         c=c_copy; c.resize(k,0);
         fori(i,m) fori(j,k) if(N[j]) c[j] -= c[kt[i]]*A[i][j];
      }
      sol = go(k-1);
      if(!sol.empty()) sol.resize(n);
      return sol;
   }
};

// Como usar
int main() {
   /* Exemplo: Maximize cx Subject to Ax <= b */
   vector<vector<double> > A(9);
   double Av[][3] = {{1,1,0}, {0,0,-1}, {-1,-1,0},
                     {0,0,1}, {1,0,0}, {0,1,0},
                     {0,0,1}, {1,0,1}, {0,1,0}};

   for(int i=0; i < 9; i++) {
      A[i].insert(A[i].begin(), &(Av[i][0]), &(Av[i][3])); // Sim, [3]! Ou seja, idx-final+1
   }

   vector<double > c(3, 1); // c=[1 1 1]
   double bv[] = {2,-1,-2,1,2,1,1,2,1};
   vector<double > b(bv, bv+sizeof(bv)/sizeof(double));

   simplex sim(A,b,c);
   vector<double> s = sim.solve();
   if(!s.size()) cout << "Impossivel\n";
   else
   for(int i=0; i < s.size(); i++) {
      cout << s[i] << endl;
   }
}
```

1.9 Casamento de strings

Código 34: String matching - Algoritmo **KMP** - O(n + m)

```cpp
// F[i] - size of the largest prefix of pattern[0..i] that is also a
// suffix of pattern[1..i]. Ex: pattern = {a,b,a,c,a,b}, F = {0,0,1,0,1,2}
#define MAX_PATTERN_SIZE 10010
int F[MAX_PATTERN_SIZE];
void build_failure_function( const string & pattern )
{
   int m = pattern.size();
```

```
    F[0] = -1;
    for (int i = 0; i < m; i++)
    {
        F[i+1] = F[i] + 1;
        while ( F[i+1] > 0 && pattern[i] != pattern[ F[i+1]-1 ] )
            F[i+1] = F[ F[i+1]-1 ] + 1;
    }
}

// retorna a posicao inicial de cada ocorrencia de pattern em text
vector<int> KMP( const string & text, const string & pattern )
{
    build_failure_function( pattern );
    vector<int> start_positions;
    int j = 0, m = pattern.size(), n = text.size();

    for (int i = 0; i < n; i++)
    {
        while ( true )
        {
            if ( text[i] == pattern[j] )
            {
                if ( ++j == m )
                {
                    start_positions.push_back( i - m + 1 );
                    j = F[j];
                }
                break;
            }

            if ( j == 0 ) break;
            j = F[j];
        }
    }

    return start_positions;
}
```

1.10 Outros

Código 35: Encontra o elemento mais comum no vetor.

```
int findMajority(int vec[], int n)
{
    int cnt = 0;
    int maior;
    for (int i = 0; i < n; i++)
    {
        if (cnt == 0) {maior = v; cnt = 1;}
        else if (v == maior) cnt++;
        else cnt--;
    }

    return maior;
}
```

Código 36: Josephus problem

```
/**
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.
 */

using namespace std;

int josephus(int n, int m)
{
    int res = 0;
    vector <int> people;
    int loc = 0;

    for (int i = 0; i < n; i++) people.push_back(i+1);

    while (people.size() > 1)
    {
        if (loc >= people.size())
            loc %= people.size();

        people.erase(people.begin()+loc);
        loc += (m-1);
    }

    return people[0];
}
```

Código 37: Gera as permutações dos elementos da string

```
bool nextPermutation(string& number)
{
        bool isBigger = true;
        int i, j;

        for (i = number.size() - 1; i >= 0; i--)
        {
                if (number[i] < number[i+1]) break;
        }

        if (i != -1)
        {
                isBigger = false;

                for (j = number.size() - 1; j >= i+1; j--)
```

```
16                  {
17                          if (number[j] > number[i])
18                          {
19                                  break;
20                          }
21                  }
22
23                  int tmp = number[i];
24                  number[i] = number[j];
25                  number[j] = tmp;
26
27                  j = number.size() -1;
28                  i++;
29
30                  while (i < j)
31                  {
32                          tmp = number[i];
33                          number[i] = number[j];
34                          number[j] = tmp;
35                          i++;
36                          j--;
37                  }
38          }
39
40          return isBigger;
41  }
```

Código 38: Permutações dos elementos da string, usando backtracking

```
1  #define TAM 10
2  int cnt = 0;
3  int total; //numero de elementos de elem
4  int elem[TAM];
5  bool usados[TAM]; //ZERE ME
6
7  void enumera(int num)
8  {
9      if (num == total)
10     {
11         cnt++;
12         for (int i = 0; i < total; i++) cout << elem[i];
13         cout << endl;
14         return;
15     }
16
17     for (int i = 0; i < total; i++)
18     {
19         if (!usados[i])
20         {
21             elem[num] = i;
22             usados[i] = true;
23             enumera (num + 1);
24             usados[i] = false;
25         }
26     }
27 }
```

Complexidade	Tamanho máximo da entrada
1	α
$log(n)$	10^9
n	10^6
n * $log(n)$	10^4
n^2	10^3
n^2* $log(n)$	10^3
n^3	10^2
2^n	20
$n!$	10

Tabela 1: Complexidade máxima da solução em fução do tamanho da entrada

2 Tabelas

tipo	bits	min...max	precisao
char	8	0..127	2
signed char	8	-128..127	2
unsigned char	8	0..255	2
short	16	-32.768 .. 32.767	4
unsigned short	16	0 .. 65.535	4
int	32	-2x10**9 .. 2 x 10**9	9
unsigned int	32	0 .. 4x10**9	9
int64_t	64	-9 x 10**18 .. 9 x 10**18	18
uint64_t	64	0 .. 18 x 10**18	19

Tabela 2: Limites de representação de dados

Tipo	%
char	c
int	d
float	e, E, f, g, G
int (octal)	o
int (hexa)	x, X
uint	u
char*	s

Tabela 3: scanf() - %[*][width][modifiers]type

modifiers	tipo
h	short int (d, i, n), or unsigned short int (o, u, x)
l	long int (d, i, n), or unsigned long int (o, u, x), or double (e, f, g)
ll	long long int (d, i, n), or unsigned long long int (o, u, x)
L	long double (e, f, g)

Tabela 4: scanf() %[*][width][modifiers]type

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5.040
8! = 40.320
9! = 362.880
10! = 3.628.800
11! = 39.916.800
12! = 479.001.600 [limite do (unsigned) int]
13! = 6.227.020.800
14! = 87.178.291.200
15! = 1.307.674.368.000
16! = 20.922.789.888.000
17! = 355.687.428.096.000
18! = 6.402.373.705.728.000
19! = 121.645.100.408.832.000
20! = 2.432.902.008.176.640.000 [limite do (u)int64_t]

Tabela 5: Fatorial

função	descrição
atof	Convert string to double
atoi	Convert string to integer
atol	Convert string to long integer
strtod	Convert string to double
strtol	Convert string to long integer
strtoul	Convert string to unsigned long integer

Tabela 6: stdlib

função	descrição
cos	Compute cosine
sin	Compute sine
tan	Compute tangent
acos	Compute arc cosine
asin	Compute arc sine
atan	Compute arc tangent
atan2	Compute arc tangent with two parameters
cosh	Compute hyperbolic cosine
sinh	Compute hyperbolic sine
tanh	Compute hyperbolic tangent
exp	Compute exponential function
frexp	Get significand and exponent
ldexp	Generate number from significand and exponent
log	Compute natural logarithm
log10	Compute common logarithm
modf	Break into fractional and integral parts
pow	Raise to power
sqrt	Compute square root
ceil	Round up value
fabs	Compute absolute value
floor	Round down value
fmod	Compute remainder of division

Tabela 7: math (angulos em radianos)