Universidade de São Paulo
Instituto de Ciências Matemáticas e Computação

Maratona de Programação

2 de agosto de 2011

Notebook criado pelo Grupo de Estudos da Maratona de Programação do ICMC - USP, com o intuito de se tornar o notebook *default* dos times do instituto.

Outras fontes que acabaram contribuindo indiretamente para este material são: Ahmed Shamsul Arefin, através do livro '*Art of Programming Contests*'; Christian Charras e Thierry Lecroq, pelos seus artigos sobre algoritmos em strings (em especial o "*Handbook of Exact String-Matching Algorithms*"); Steven S. Skiena, pelo livro '*The Algorithm Design Manual*'; Steven Halim, do site *Methods to Solve* (www.comp.nus.edu.sg/ stevenha/); Vinicius José Fortuna do site lampiao.ic.unicamp.br/maratona/; e Udi Manber, autor do livro '*Introduction to Algorithms: A Creative Approach*'.

Alguns códigos também foram baseados na biblioteca de algoritmos da maratona da PUC-Rio e nas implementações de Igor Naverniouk.

Sumário

Capítulo 1

Conceitos Básicos

1.1 I/O

1.1.1 Especificadores

Na Tabela 1.1 se tem os especifadores ('%[especificador]') suportados pelo *scanf* e *printf* para fazer leitura ou impressão de valores.

Especificador	Significado	Exemplo
c	char	w
d	inteiro com sinal na base decimal	574
i	inteiro com sinal na base decimal	574
e	notação científica (mantissa/expoente) usando o caractere `e`	3.3215e+2
e	notação científica (mantissa/expoente) usando o caractere `E`	3.3215E+2
f	ponto flutuante na base decimal	332.15
g	usa o mais curto dentre %e ou %f	332.15
G	usa o mais curto dentre %E ou %f	332.15
o	octal com sinal	312
s	*string* de caracteres	abc
u	inteiro sem sinal na base decimal	32742
x	inteiro sem sinal na base hexadecimal	2ea
X	inteiro sem sinal na base hexadecimal com letras maiúsculas	2EA
p	endereço do ponteiro	A201:0000

Tabela 1.1: Especificadores.

1.2 Limites dos Tipos Primitivos de Dados em C

Nas Tabelas 1.2 e 1.3 são dados os valores máximos e mínimos que uma determinada variável primitiva pode ter, além da sua precisão em casas decimais.

Na prática isso significa que no cálculo de Fatorial de um número, 12! (479.001.600) é o máximo que o **unsigned int** consegue guardar, e 20! (2.432.902.008.176.640.000) é o máximo do **unsigned long int**. E no Triângulo de Pascal, C(33,16) = 1.166.803.110 é o limite de um **int**, C(34,17) = 2.333.606.220 é o limite de um **unsigned int**, C(66, 33) = 7.219.428.434.016.265.740 é o limite de um **long int** e C(67,33) = 14.226.520.737.620.288.370 é o limite de um **unsigned long int**.

Tipo Primitivo	Valor de Máximo e Mínimo	Precisão Decimal
char	+127 ... -128	2
unsigned char	+255 ... 0	2
short int	+32.767 ... -32.768	4
unsigned short int	+65.535 ... 0	4
int	+2.147.483.647 ... –2.147.483.648	9
unsigned int	+4.294.967.295 ... 0	9
long int	+2.147.483.647 ... –2.147.483.647	9
unsigned long int	+4.294.967.295 ... 0	9
long long int	+9.223.372.036.854.775.807 ... –9.223.372.036.854.775.808	18
unsigned long long int	+18.446.744.073.709.551.615 ... 0	19

Tabela 1.2: Valores de máximos e mínimos para números inteiros e caracteres em C.

Tipo Primitivo	Quantidade de Bits	Expoente	Precisão Decimal
float	32	+38 ... -38	6
double	64	+308 ... -308	15
long double	80	+19.728 ... -19.728	18

Tabela 1.3: Valores de máximos e mínimos para números em ponto flutuante em C.

1.3 Complexidade e tamanhos de entrada

Tamanhos de entrada máximos para cada complexidade, com limite de tempo de 8 segundos.

Complexidade	N máximo
$\Theta(N)$	100 000 000
$\Theta(NlogN)$	40 000 000
$\Theta(N^2)$	10 000
$\Theta(N^3)$	500
$\Theta(N^4)$	90
$\Theta(2^N)$	20
$\Theta(N!)$	11

Tabela 1.4: Tamanho máximo do problema (aproximadamente) solúvel em 8 segundos

Capítulo 2

Algoritmos de Busca

2.1 Busca Binária

Efetua uma busca em $O(log_2(n))$ por se utilizar do fato de que o vetor está ordenado, assim faz sucessivos saltos na metade dele à procura do elemento.

Complexidade: $O(log_2(n))$.

Entrada: **(int *) v**, que guarda o endereço do vetor que contém os elementos; **(int) n**, que contém a quantidade de elementos no vetor (numerados de 0..n-1); e **(int) x**, que contém o número procurado.

Saída: A sua posição no vetor caso ele tenha sido encontrado, caso contrário retorna -1.

Listing 2.1: Busca Binária

```
int BuscaBinaria(int *v, int n, int x) {
    int min, max, meio;

    min = 0;
    max = n;
    do {
        meio = (min + max) / 2;
        if (x > v[meio]) {
            min = meio + 1;
        }
        else {
            max = meio - 1;
        }

    } while(v[meio] != x && (min <= max));

    if (v[meio] == x) {
        return meio;
    }

    return -1;
}
```

2.2 Range Minimum Query (RMQ)

Listing 2.2: RMQ

```
int prep[100010][20];

```

```
int log2(int x) {
    int k = 1, t = 0;
    while (k <= x) { k <<= 1; t++;};
    return t-1;
}

void preprocess(int *s, int n) {
        for (int i = 0; i < n; i++)
            prep[i][0] = i;

        for (int j = 1, l = 1; (l<<1) <= n; j++, l <<= 1)
            for (int i = 0; i+l < n; i++)
                if (s[prep[i][j-1]] < s[prep[i+l][j-1]])
                    prep[i][j] = prep[i+l][j-1];
                else
                    prep[i][j] = prep[i][j-1];
}

int query(int i, int j) {
    int k = log2(j-i+1);
    if (s[prep[i][k]] >= s[prep[j-(1<<k)+1][k]])
        return prep[i][k];
    else
        return prep[j-(1<<k)+1][k];
}
```

Capítulo 3

Strings

3.1 Brute-Force

Algoritmo de Força Bruta que testa a substring em todas as posições da string.

Complexidade: $O(n^2)$.

Entrada: **(char *) str**, contém o endereço do vetor da string; **(char *) sub**, contém o endereço do vetor da substring.

Saída: **(int)** , que contém o indíce do começo da substring na string caso tenha encontrado ela, ou -1 se não a encontrou.

Listing 3.1: Brute Force em String

```
int BF(char *str, char *sub) {
    int l1, l2, i, j;

    l1 = strlen(str);
    l2 = strlen(sub);

    for(i=0; i <= l1 - l2; i++) {
        for(j=0; j < l2; j++)
            if(str[i+j] != sub[j])
                break;
        if(j >= l2)
            return i;
    }

    return -1;
}
```

3.2 KMP Search (Knuth-Morris-Pratt)

Complexidade: $O(m+n)$ com pré-processamento em $O(m)$. Faz no máximo $2n-1$ comparações.

Entrada: **(char *) x**, contém o endereço do vetor da substring; **(int) m**, que é o tamanho da substring; **(char *) y**, contém o endereço do vetor da string; **(int) n**, que é o tamanho da string.

Saída: **(int)** , que contém o indíce do começo da substring na string caso tenha encontrado ela, ou -1 se não a encontrou.

Listing 3.2: KMP Search

```
const int XSIZE = 50000;

void preKmp(char *x, int m, int kmpNext[]) {
```

```
    int i, j;

    i = 0;
    j = kmpNext[0] = -1;
    while (i < m) {
        while (j > -1 && x[i] != x[j])
            j = kmpNext[j];
        i++;
        j++;
        if (x[i] == x[j])
            kmpNext[i] = kmpNext[j];
        else
            kmpNext[i] = j;
    }
}


int KMP(char *x, int m, char *y, int n) {
    int i, j, kmpNext[XSIZE];

    /* Preprocessing */
    preKmp(x, m, kmpNext);

    /* Searching */
    i = j = 0;
    while (j < n) {
        while (i > -1 && x[i] != y[j])
            i = kmpNext[i];
        i++;
        j++;
        if (i >= m) {
            return (j - i); /* se tiver mais de uma trocar por printf */
            i = kmpNext[i];
        }
    }
    return -1;
}
```

3.3 Karp-Rabin

Complexidade: $O(mn)$ com pré-processamento em $O(m)$. Apesar da complexidade quadrática, tem execução estimada em $O(m+n)$.

Entrada: **(char *) x**, contém o endereço do vetor da substring; **(int) m**, que é o tamanho da substring; **(char *) y**, contém o endereço do vetor da string; **(int) n**, que é o tamanho da string.

Saída: **(int)** , que contém o índice do começo da substring na string caso tenha encontrado ela, ou -1 se não a encontrou.

Listing 3.3: Karp-Rabin

```
#define REHASH(a, b, h) ((((h) - (a)*d) << 1) + (b))

int KR(char *x, int m, char *y, int n) {
    int d, hx, hy, i, j;

    /* Preprocessing */
    /* computes d = 2^(m-1) with the left-shift operator */
    for (d = i = 1; i < m; ++i)
```

```
        d = (d<<1);

    for (hy = hx = i = 0; i < m; ++i) {
        hx = ((hx<<1) + x[i]);
        hy = ((hy<<1) + y[i]);
    }

    /* Searching */
    j = 0;
    while (j <= n-m) {
        if (hx == hy && memcmp(x, y + j, m) == 0)
            return j;
        hy = REHASH(y[j], y[j + m], hy);
        ++j;
    }
    return -1;
}
```

3.4 Boyer-Moore String Search

Complexidade: Pré-processamento em $O(m)$ com busca em $O(mn)$, mas efetua no máximo $3n$ comparações e tem no melhor caso busca em $O(n.m)$. Apesar da complexidade quadrática, tem execução estimada em $O(m+n)$.

Entrada: **(char *) x**, contém o endereço do vetor da substring; **(int) m**, que é o tamanho da substring; **(char *) y**, contém o endereço do vetor da string; **(int) n**, que é o tamanho da string.

Saída: **(int)** , que contém o indíce do começo da substring na string caso tenha encontrado ela, ou -1 se não a encontrou.

Listing 3.4: Boyer-Moore String Search

```
const int XSIZE = 50000;
const int ASIZE = 50000;

void preBmBc(char *x, int m, int bmBc[]) {
    int i;

    for (i = 0; i < ASIZE; ++i)
        bmBc[i] = m;
    for (i = 0; i < m - 1; ++i)
        bmBc[x[i]] = m - i - 1;
}


void suffixes(char *x, int m, int *suff) {
    int f, g, i;

    suff[m - 1] = m;
    g = m - 1;
    for (i = m - 2; i >= 0; --i) {
        if (i > g && suff[i + m - 1 - f] < i - g)
            suff[i] = suff[i + m - 1 - f];
        else {
            if (i < g)
                g = i;
            f = i;
            while (g >= 0 && x[g] == x[g + m - 1 - f])
                --g;
            suff[i] = f - g;
```

```
        }
    }
}

void preBmGs(char *x, int m, int bmGs[]) {
    int i, j, suff[XSIZE];

    suffixes(x, m, suff);

    for (i = 0; i < m; ++i)
        bmGs[i] = m;
    j = 0;
    for (i = m - 1; i >= -1; --i)
        if (i == -1 || suff[i] == i + 1)
            for (; j < m - 1 - i; ++j)
                if (bmGs[j] == m)
                    bmGs[j] = m - 1 - i;
    for (i = 0; i <= m - 2; ++i)
        bmGs[m - 1 - suff[i]] = m - 1 - i;
}


int BM(char *x, int m, char *y, int n) {
    int i, j, bmGs[XSIZE], bmBc[ASIZE];

    /* Preprocessing */
    preBmGs(x, m, bmGs);
    preBmBc(x, m, bmBc);

    /* Searching */
    j = 0;
    while (j <= n - m) {
        for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i);
        if (i < 0) {
            return (j);
            j += bmGs[0];
        }
        else
            j += max(bmGs[i], bmBc[y[i + j]] - m + 1 + i);
    }
    return -1;
}
```

3.5 Aho-Corasick

Busca por todas as ocorrências de k padrões (de tamanhos $t_0, t_1, \ldots, t_k$) em um texto de tamanho n.

Complexidade: Pré-processamento em $O(t_0 + t_1 + \cdots + t_k)$ com busca em $O(n)$.

Utilização: utilize a função insere(texto, id) para inserir os padrões a serem buscados no autômato. Caso o padrão seja encontrado, ele será identificado pelo valor id. Depois de inseridos todos os padrões, chame a função falha(). Para buscar em um texto, use o código abaixo.

Listing 3.5: Aho-Corasick

```
// tamanho do alfabeto
#define maxsigma 255

struct trie {
```

```
    trie* g[maxsigma];
    list<int> o;
    char c;
    trie *f;

    trie(char c = 0): c(c) {
        for (int i = 0; i < maxsigma; i++) g[i] = NULL;
        f = NULL;
    }

    void free() {
        for (int i = 0; i < maxsigma; i++)
            if (g[i]) { if (g[i] != this) {g[i]->free(); delete g[i];} g[i] = NULL; }
        o.clear();
    }
};

trie raiz;

void insere(char *s, int i) {
    trie *p = &raiz;
    while (*s) {
        if (!p->g[*s])
            p->g[*s] = new trie(*s);
        p = p->g[*s];
        s++;
    }
    p->o.push_back(i);
}

void falha() {
    queue<trie*> q;
    trie *x, *v, *u;
    for (int i = 0; i < maxsigma; i++)
        if (raiz.g[i] != NULL) {
            raiz.g[i]->f = &raiz;
            q.push(raiz.g[i]);
        } else
            raiz.g[i] = &raiz;

    while (!q.empty()) {
        x = q.front(); q.pop();
        for (int i = 0; i < maxsigma; i++)
            if ((u = x->g[i]) != NULL) {
                q.push(u);
                v = x->f;
                while (v->g[i] == NULL) v = v->f;
                u->f = v->g[i];
                u->o.insert(u->o.begin(), u->f->o.begin(), u->f->o.end());
            }
    }
}

// Busca pela string txt
int l;
trie* state;

state = &raiz;
l = strlen(txt);
```

```
for (int i = 0; i < l; i++) {
    while (state->g[txt[i]] == NULL) state = state->f;
    state = state->g[txt[i]];
    if (!state->o.empty())
        for (list<int>::iterator it = state->o.begin(); it != state->o.end(); it++)
            // matching da string *it
}

// nao esquecer de liberar o automato
raiz.free();
```

3.6 Suffix Array

3.6.1 Construção

Constrói o array de sufixos da string.

Complexidade: $O(nlog^2(n))$
Entrada: (char []) s string de entrada.
Saída: (int []) SA array de sufixos da string.

Listing 3.6: Suffix Array

```
#define max 100010

int delta, len, SA[max], iSA[max], val[max], tval[max];
bool cmp(const int& a, const int& b) { return val[a+delta] < val[b+delta]; }

void build_SA() {
    int i, j;
    for (i=0; i<len; i++) val[i] = s[i];
    for (i=0; i<len; i++) SA[i] = i;

    delta = 0;
    sort(SA, SA+len, cmp);

    for (delta=1;;delta*=2) {
        bool found = false;
        for (i=0; i<len; i++) {
            for (j=i+1; j<len; j++) if (val[SA[i]] != val[SA[j]]) break;
            if (j > i+1) {
                found = true;
                sort(SA+i, SA+j, cmp);
            }
            i = j - 1;
        }

        if (!found) break;
        tval[SA[0]] = 0;
        for (i=1; i<len; i++)
        {
            tval[SA[i]] = tval[SA[i-1]];
            if (val[SA[i]] == val[SA[i-1]] && val[SA[i]+delta] == val[SA[i-1]+delta]);
            else tval[SA[i]]++;
        }
        memcpy(val, tval, len * sizeof(int));
    }

```

```
    //for (i=0; i<len; i++) iSA[SA[i]] = i;
}
```

Complexidade: $O(nlog(n))$
Entrada: (char []) str string de entrada.
Saída: (int []) pos array de sufixos da string.

Listing 3.7: Suffix Array nlogn

```
int str[N]; //input
int rank[N], pos[N]; //output
int cnt[N], next[N]; //internal
bool bh[N], b2h[N];

bool smaller_first_char(int a, int b){
  return str[a] < str[b];
}

void suffixSort(int n){
  for (int i=0; i<n; ++i)
    pos[i] = i;

  sort(pos, pos + n, smaller_first_char);

  for (int i=0; i<n; ++i){
    bh[i] = i == 0 || str[pos[i]] != str[pos[i-1]];
    b2h[i] = false;
  }

  for (int h = 1; h < n; h <<= 1){
    int buckets = 0;
    for (int i=0, j; i < n; i = j){
      j = i + 1;
      while (j < n && !bh[j]) j++;
      next[i] = j;
      buckets++;
    }
    if (buckets == n) break;

    for (int i = 0; i < n; i = next[i]){
      cnt[i] = 0;
      for (int j = i; j < next[i]; ++j)
        rank[pos[j]] = i;
    }

    cnt[rank[n - h]]++;
    b2h[rank[n - h]] = true;
    for (int i = 0; i < n; i = next[i]){
      for (int j = i; j < next[i]; ++j){
        int s = pos[j] - h;
        if (s >= 0){
          int head = rank[s];
          rank[s] = head + cnt[head]++;
          b2h[rank[s]] = true;
        }
      }
      for (int j = i; j < next[i]; ++j){
        int s = pos[j] - h;
        if (s >= 0 && b2h[rank[s]]){
```

```
          for (int k = rank[s]+1; !bh[k] && b2h[k]; k++) b2h[k] = false;
        }
      }
    }
    for (int i=0; i<n; ++i){
      pos[rank[i]] = i;
      bh[i] |= b2h[i];
    }
  }

  for (int i=0; i<n; ++i) rank[pos[i]] = i;
}
```

3.6.2 Longest common prefix

Complexidade: $O(n)$

Entrada: (int []) pos array de sufixos.

Saída: (int []) heights array de lcps da string.

Listing 3.8: Longest Common Prefix

```
int height[N];
// height[i] = length of the longest common prefix of suffix pos[i] and suffix pos[i-1]
// height[0] = 0
void getHeight(int n){
  for (int i=0; i<n; ++i) rank[pos[i]] = i;
  height[0] = 0;
  for (int i=0, h=0; i<n; ++i){
    if (rank[i] > 0){
      int j = pos[rank[i]-1];
      while (i + h < n && j + h < n && str[i+h] == str[j+h]) h++;
      height[rank[i]] = h;
      if (h > 0) h--;
    }
  }
}
```

3.6.3 Busca de substrings

Complexidade: $O(m * log(n))$, onde m é o comprimento do padrão buscado e n é o tamanho do array de sufixos (tamanho da string maior).

Entrada: (int) n tamanho do array de sufixos (**atenção:** numerado de 1 a n). **(char *) Q** padrão a ser buscado. **(int *) SA** variável global com o array de sufixos.

Saída: (pair¡int,int¿) intervalo contendo todos os matchings do padrão.

Listing 3.9: Busca de substrings em um Suffix Array

```
pair<int, int> range(int n, char *Q) {
    int lo = 1, hi = n, m = strlen(Q), mid = lo; // index 0 - null, valid range = [1..n]
    while (lo <= hi) {
        mid = (lo + hi) / 2;
        int cmp = strncmp(S + SA[mid], Q, m);
        if (cmp == 0) // found
            break;
        else if (cmp > 0)
```

```
            hi = mid - 1;
        else
            lo = mid + 1;
    }
    if (lo > hi)
        return make_pair(-1, -1); // not found
    for (lo = mid; lo >= 1 && strncmp(S + SA[lo], Q, m) == 0; lo--);
    lo++;
    for (hi = mid; hi <= n && strncmp(S + SA[hi], Q, m) == 0; hi++);
    hi--;
    return make_pair(lo, hi);
}
```

Capítulo 4

Algoritmos de Ordenação

4.1 Bubble Sort

Mais simples e pior algoritmo de ordenação, mas não requer espaço extra.

Complexidade: $O(n^2)$.

Entrada: (int) s[], contém o endereço do vetor de elementos; **(int) n**, quantidade de elementos no vetor.

Listing 4.1: Bubbe Sort

```
void BubbleSort(int s[], int n) {
    int i, j, temp;

    for(i=0; i < n; i++)
        for(j=i; j < n; j++)
            if(s[i] > s[j]) {
                temp = s[i];
                s[i] = s[j];
                s[j] = temp;
            }

    return ;
}
```

4.2 Insertion Sort

Algoritmo que após ler um elemento, procura a sua posição no vetor, quando a acha empurra todos à direita para abrir espaço para ele. Não é eficiente para grandes quantidades de elementos, no entanto costuma ser o algoritmo de ordenação mais eficiente dos que tem complexidade quadrática, além de ter a vantagem de ser linear caso a entrada já esteja ordenada, ou seja, recomendado para entradas parcialmente ou totalmente ordenadas.

O algoritmo do Listing 4.2 contém o código do Insertion Sort para ordenar os elementos enquanto se lê.

Complexidade: $O(n^2)$.

Entrada: (int) s[], contém o endereço do vetor de elementos; **(int) *n**, quantidade atual de elementos no vetor; **(int) x**, contém o elemento recém lido.

Listing 4.2: Insertion Sort para ordenar enquanto se lê os dados

```
void InsertionSort(int s[], int *n, int x) {
    int i;

    if(*n == 0)
        s[0] = x;
```

```
    else {
        i=*n-1;
        while(i >= 0 && s[i] > x) {
            s[i+1] = s[i];
            i--;
        }
        if(i < 0)
            s[0] = x;
        else
            s[i+1] = x;
    }
    (*n)++;

    return ;
}
```

O algoritmo do Listing 4.3 contém o código do Insertion Sort para ordenar os elementos depois que todos eles foram lidos e estão guardados no vetor.

Complexidade: $O(n^2)$.

Entrada: **(int) s[]**, contém o endereço do vetor de elementos; **(int) n**, quantidad de elementos no vetor.

Listing 4.3: Insertion Sort para ordenar após ter lido todos os dados

```
void InsertionSort(int s[], int n) {
    int i, j;
    int temp;

    for (i=1; i<n; i++) {
        j=i;
        while ((j>0) && (s[j] < s[j-1])) {
            temp = s[j];
            s[j] = s[j-1];
            s[j-1] = temp;

            j=j-1;
        }
    }

    return ;
}
```

4.3 Selection Sort

Em geral é pior do que o Insertion Sort e faz muitas comparações, mas a quantidade de *swaps* de dados nele é baixa. O algoritmo procura o índice do maior elemento, e só depois de acha-lo efetua a troca de elementos.

Complexidade: $O(n^2)$.

Entrada: **(int) s[]**, contém o endereço do vetor de elementos; **(int) n**, quantidade de elementos no vetor.

Listing 4.4: Selection Sort

```
void SelectionSort(int s[], int n) {
    int i, j;
    int min;        /* indice do minimo */
    int temp;   /* para fazer a troca de valores */

    for (i=0; i<n; i++) {
        min=i;
```

```
        for (j=i+1; j<n; j++)
            if(s[j] < s[min])
                min=j;

        temp = s[i];
        s[i] = s[min];
        s[min] = temp;
    }

    return ;
}
```

4.4 Counting Sort

A grande vantagem de se ordenar um vetor com Counting Sort é o que o algoritmo é $O(n)$ é e estável. Não dá pra contar troca, porque ele não faz troca. O maior problema dele é que usa muita memória, porque precisa de dois vetores a mais.

Listing 4.5: Counting Sort

```
#include <cstdio>
#include <cstring>

void countingsort(int *v, int min, int max) {
    int range = max - min + 1;
    int *counts = new int[range];
    int offset = -min;
    memset(counts, 0, range * sizeof(int));
    for(int i = 0; i < range; ++i)
        ++counts[v[i] + offset];
    for(int i = 0, index = 0; i < range; ++i)
        while(counts[i]--)
            v[index++] = i - offset;
    delete [] counts;
}

int main() {
    int v[] = {3,1,4,5,7,1,3,9,4,9,8,1,5,7,8,3,4,5,2,8};
    countingsort(v,0,19);
    for(int i=0;i<19;i++)
        printf("%d\n", v[i]);
}
```

4.5 Merge Sort

O Merge Sort é um método de ordenação estável baseado em técnicas dividir pra conquistar, tendo, assim, complexidade $O(nlog(n))$. Da maneira que foi implementado no código 4.6, ele não é muito eficiente em termos de memória, pois usa vetores auxiliares para fazer a ordenação, mas conta o número mínimo de trocas adjacentes.

Listing 4.6: Merge Sort

```
int v[MAXN];
unsigned long long trocas;

void merge(int ini, int fim) {

```

```
    static int aux[MAXN];
    int i, meio=(ini+fim)/2, e, d;

    if(ini < fim) {

        merge(ini, meio);
        merge(meio+1, fim);

        for(i=ini; i<=fim; i++)
            aux[i] = v[i];

        i = ini;
        e = ini;
        d = meio+1;

        while(e<=meio && d<=fim) {
            if(aux[e] <= aux[d])
                v[i++] = aux[e++];
            else {
                v[i++] = aux[d++];
                trocas += meio-e+1;
            }
        }

        while(e <= meio)
            v[i++] = aux[e++];

        while(d <= fim)
            v[i++] = aux[d++];

    }
}
```

4.6 Quick Sort

Melhor algoritmo de ordenação para aqueles baseados em comparação. Sua execução consiste em escolher um pivô, após essa escolha reordena os elementos de forma que todos os elementos menores que ele estejam à esquerda, e todos aqueles maiores estejam à direita. Depois disso particiona o vetor em dois e ordena as partes aplicando mais uma vez a escolha do pivô. Algoritmo de *Divide & Conquer* que tem o seu pior caso com execução quadrática quando o vetor dado já está ordenado, para evitar isso, faça uma escolha aleatória do pivô (não escolher o primeiro elemento, ou seja, não fazer firsthigh = l). Algoritmo já implementado em C/C++, ver seção 4.7.

Complexidade: $O(nlog(n))$ com pior caso em $O(n^2)$.

Entrada: (int) s[], contém o endereço do vetor de elementos; **(int) l**, contém o índice do começo do vetor; **(int) h**, contém o índice do final do vetor.

Listing 4.7: Quick Sort

```
// fazer primeira chamada com quicksort(s, 0, n)
quicksort(int s[], int l, int h) {
    int p;      /* index of partition */

    if ((h-l)>0) {
        p = partition(s,l,h);
        quicksort(s,l,p-1);
        quicksort(s,p+1,h);
    }
}
```

```

int partition(int s[], int l, int h) {
    int i;          /* counter */
    int p;          /* pivot element index */
    int firsthigh;  /* divider position for pivot */

    p = h;
    firsthigh = l;
    for (i=l; i<h; i++)
        if (s[i] < s[p]) {
            swap(&s[i],&s[firsthigh]);
            firsthigh++;
        }
    swap(&s[p],&s[firsthigh]);

    return (firsthight);
}
```

4.7 Ordenação pelo Quick Sort da stdlib.h

Como o Quick Sort já vem implementado na **stdlib.h** e só requer a codificação de uma função de comparação, é conveniente já utiliza-lo para ordenação, afinal, sua ordenação é a mais rápida dentre os algoritmos de ordenação por comparação. Sendo assim, está seção irá exemplificar a sua utilização.

Seu protótipo tem a forma apresentada no Listing 4.8.

Listing 4.8: Protótipo do Quick Sort

```
qsort(void *base, size_t nel, size_t width, int (*compar)(const void *, const void *))
```

Entrada: (void *) base, endereço do vetor a ser ordenado; **(size_t) nel**, contém a quantidade de elementos no vetor; **(size_t) width**, contém a quantidade de bytes de cada elemento, fazer sizeof da estretura; **int (*compar)(const void *, const void *)**, trocar pelo nome da função de comparação.

Como dito, a única parte necessária para implementar é a função de comparação, que deve obedecer a seguinte regra: ter retorno negativo caso a deva vir antes de b, 0 se a igual a b e positivo se a deve vir depois de b. Sendo assim, seja n a quantidade de elementos e s o endereço do vetor, para um vetor de inteiros basta utilizar o Listing 4.9.

Listing 4.9: Ordenação de inteiros com o Quick Sort da stdlib.h

```
int compare_function(const void *a,const void *b) {
    int *x = (int *) a;
    int *y = (int *) b;

    return *x - *y;
}
```

Para um vetor de strings basta utilizar o Listing 4.10.

Listing 4.10: Ordenação de strings com o Quick Sort da stdlib.h

```
int compare_function(const void *a,const void *b) {
    return (strcmp((char *)a,(char *)b));
}

s[MAX_1][MAX_2];
qsort(s, n, MAX_2 * sizeof(char), compare_function);
```

E para um vetor de uma estrutura qualquer basta utilizar o Listing 4.11.

Listing 4.11: Ordenação de estrutura qualquer com o Quick Sort da stdlib.h

```
typedef struct {
    int key;
    double value;
} the_record;

int compare_function(const void *a,const void *b) {
    the_record *x = (the_record *) a;
    the_record *y = (the_record *) b;
    return x->key - y->key;
}

qsort(s, n, sizeof(the_record), compare_function);
```

Graças a essa facilidade de escrever somente uma função de comparação, fica simples fazer uma ordenação de múltiplos campos, veja o exercício resolvido *10194: Football (aka Soccer)* na seção 17.9.

4.8 Permutações

4.8.1 Paridade

Complexidade: $O(n)$
Entrada: (int []) perm permutação de entrada, **(int) n** tamanho da permutação
Saída: (int) quantidade de trocas na decomposição em ciclos da permutação

Listing 4.12: Paridade de permutações

```
#define MAXN 100010

char c[MAXN];

int paridade() {
    int a, k, conta;
    bool par;

    memset(c, 0, sizeof(c));
    par = true;

    conta = 0;
    for (int i = 1; i <= n; i++) {
        if (!c[i]) {
            k = i;
            a = k;
            k = perm[k];
            c[k] = 1;

            while (k != a) {
                c[k] = 1;
                k = perm[k];
                conta++;
            }
            c[k] = 1;
        }
    }
    return conta;
}
```

Capítulo 5

BigNum

5.1 500! em Java

Listing 5.1: 500! em Java

```
import java.math.*;
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String args[]) {
        BigInteger[] fat;
        BigInteger i;
        int k, n;

        fat = new BigInteger[1010];
        i = BigInteger.ONE;
        fat[0] = i;
        fat[1] = i;
        k = 2;
        while (k <= 1000) {
            i = i.multiply(BigInteger.valueOf(k));
            fat[k] = i;
            k++;
        }

        Scanner sc = new Scanner(System.in);

        while (sc.hasNextInt()) {
            n = sc.nextInt();
            System.out.println(Integer.toString(n) + "!");
            System.out.println(fat[n].toString());
        }
    }
}
```

5.2 BigDecimal em Java

Programa que lê vários BigDecimals (até encontrar um zero) e imprime a soma deles na tela. A gambiarra do StringBuffer é para remover zeros à direita do ponto.

Listing 5.2: BigDecimal em Java

```java
import java.math.*;
import java.io.*;
import java.util.*;

public class Main {
    public static void main (String args[]) {
        BigDecimal a, s = BigDecimal.ZERO;
        Scanner sc = new Scanner(System.in);

        int n;
        n = sc.nextInt();
        for (int i = 0; i < n; i++) {
            a = BigDecimal.ONE;
            s = BigDecimal.ZERO;
            while (a.compareTo(BigDecimal.ZERO) != 0) {
                a = sc.nextBigDecimal();
                s = s.add(a);
            }
            StringBuffer sb = new StringBuffer(s.toString());
            int idx = sb.indexOf(".");
            if (idx < 0) idx = 0;
            for (int it = sb.length()-1; it >= idx && (sb.charAt(it) == '0' || sb.charAt(it) == '.') ; it = sb.length()-1)
                sb.deleteCharAt(it);
            System.out.println(sb.toString());
        }
    }
}
```

5.3 BigNum Completo

Listing 5.3: BigNumbers

```
#define QTT 10 // quantidade de Big nums
#define MOD 1000000000 // base (10^9)
#define DIGS "9" // o numero de digitos decimais da base (MOD)
#define TAM 100 // o numero deve ter no max (9*TAM) digitos

unsigned long long bn[QTT][TAM];
int nb[QTT];
char sb[QTT]; // 1 positivo... -1 negativo

void imprime(int a) {
    int i = nb[a] - 1;
    if (sb[a] < 0) printf("-");
    printf("%llu", bn[a][i--]);
    for (; i >= 0; i--)
        printf("%0"DIGS"llu", bn[a][i]);
}

void zera(int a) {
    sb[a] = nb[a] = 1;
    memset(bn[a], 0, sizeof(bn[a]));
}

void intto(long long a, int b) {
    zera(b);
    sb[b] = (a < 0) ? -1 : 1;
    a *= sb[b];

    for (nb[b] = 0; a != 0; nb[b]++) {
        bn[b][nb[b]] = a%MOD;
        a /= MOD;
    }
}

void strto(char *s, int a) {
    int i = 0, b = 1, n = strlen(s)-1, j = 0;
    zera(a);
    for (; j <= n && (s[j] == ’0’ || s[j] == ’-’); ++j); // tirar os leadings 0
    if (j != 0) s[j-1] = s[0]; // caso seja negativo
    j = s[j-1] == ’-’ ? j-1 : j; // ^
    for (; n >= j; n--) {
        if (s[n] == ’-’)  { sb[a] = -1; break; }
        bn[a][i] += b*(s[n]-’0’);
        b *= 10;
        if (b == MOD && n > 0) { b = 1; i++; }
    }
    nb[a] = i+1;
}

// DEPENDENCIA: subtracao
// a comparacao eh em valor absoluto! (1 -> ’a’ maior, 0 -> iguais, -1 -> ’b’ maior)
int compare(int a, int b) {
    if( nb[a] != nb[b] )
        return (nb[a] > nb[b]) ? 1 : -1;
    else {
        int i;
```

```
        for (i = nb[a]-1; (i >= 0) && (bn[a][i] == bn[b][i]); i--) ;
        if (bn[a][i] == bn[b][i]) return 0;
        return (bn[a][i] > bn[b][i]) ? 1 : -1;
    }
    // return lexicographical_compare_3way(bn[a], bn[a]+nb[a], bn[b], bn[b]+nb[b]);
}

// C <- A+B
void soma(int a, int b, int c) {
    zera(c);
    int i, cc, nbc;
    long long unsigned acc = 0;
    if (sb[a] == sb[b]) {
        sb[c] = sb[a];
        nb[c] = max(nb[a], nb[b]);
        for (i = 0; i < nb[c]; i++) {
            bn[c][i] = bn[a][i] + bn[b][i] + acc;
            acc = bn[c][i]/MOD;
            bn[c][i] %= MOD;
        }
        if (acc > 0) bn[c][nb[c]++] = acc;
    } else { // SUBTRACAO daqui para baixo
        if ((cc = compare(a, b)) == 0) zera(c);
        else {
            if (cc == -1) swap(a, b); // a eh sempre o maior
            for (i = 0; i < nb[a]; i++) {
                if (bn[a][i] < (bn[b][i] + acc)) {
                    bn[c][i] = MOD + bn[a][i] - bn[b][i] - acc;
                    acc = 1;
                } else {
                    bn[c][i] = bn[a][i] - bn[b][i] - acc;
                    acc = 0;
                }
            }
            for (nb[c] = nb[a]; bn[c][nb[c]-1] == 0; nb[c]--) ;
            sb[c] = sb[a];
        }
    }
}
// DEPENDE: zera
// C <- A*B
void mult(int a, int b, int c) {
    unsigned long long i, j, acc;
    zera(c);
    sb[c] = sb[a]*sb[b];
    for (i = 0; i < nb[a]; i++) {
        for (j = 0, acc = 0; j < nb[b]; j++) {
            bn[c][i+j] += acc + (bn[a][i]*bn[b][j]);
            acc = bn[c][i+j] / MOD;
            bn[c][i+j] %= MOD;
        }
        if (acc) bn[c][i+j] += acc;
    }

    if (nb[a] == 1 && bn[a][0] == 0 || nb[b] == 1 && bn[b][0] == 0)
        nb[c] = 2; // se o resultado der 0 vai ter 1 nb, colocar 2 pq ali em baixo vai fazer 2-1=1
    else
        nb[c] = nb[a]+nb[b];
```

```
    nb[c] = (bn[c][nb[c]-1]) ? nb[c] : (nb[c]-1);
}

// DEPENDENCIA: divisao
void desloca(int a) {
    if (bn[a][0] == 0 && nb[a] == 1) return;
    for (int i = nb[a]++; i > 0; i--)
        bn[a][i] = bn[a][i-1];
    bn[a][0] = 0;
}

// DEPENDE: desloca, soma (subtracao), zera
// Q <- num / div
// R <- num % div
// aux eh uma variavel auxiliar necessaria para a busca binaria
void div(int num, int div, int r, int q, int aux) {
    int k = nb[num]-1, sd = sb[div], sn = sb[num];
    int bkp;
    zera(r); zera(q);
    for (int k = nb[num]-1; k >= 0; k--) {
        desloca(r);
        bn[r][0] = bn[num][k];
        desloca(q);
        if (compare(r, div) < 0) continue;

        int lo  = 1, hi = MOD-1, mid;
        while (1) {
            mid = (lo+hi+1)/2;
            bn[q][0] = mid;
            bkp = nb[q];
            nb[q] = 1;
            mult(q, div, aux);
            nb[q] = bkp;

            if (lo == hi) break;

            int cmp = compare(aux, r);
            if (cmp == 0)
                break;
            else if (cmp > 0)
                hi = mid-1;
            else
                lo = mid;
        }
        sb[aux] = -1;
        soma(r, aux, r);
        sb[aux] = 1;
    }
    sb[q] = sd*sn;
    sb[div] = sd; sb[num] = sn;
}

// DEPENDE: desloca, soma (subtracao), zera
// Q <- num / div
// R <- num % div
void divantigo(int num, int div, int r, int q) {
    int k = nb[num]-1, sd = sb[div], sn = sb[num];
    unsigned long long iq;
    zera(r); zera(q);
```

```
    for (int k = nb[num]-1; k >= 0; k--) {
        desloca(r);
        bn[r][0] = bn[num][k];
        for (iq = 0, sb[div] = -1, sb[num] = 1; compare(r, div) >= 0; iq++)
            soma(r, div, r);
        desloca(q);
        bn[q][0] = iq;
    }
    sb[q] = sd*sn;
    sb[div] = sd; sb[num] = sn;
}
```

5.4 Raiz quadrada em Java

Código do André.

Listing 5.4: Raiz quadrada de bignum

```
import java.math.BigInteger;
import java.util.Scanner;

public class Main {
   static BigInteger sqrt(BigInteger n) {
       BigInteger a = BigInteger.ONE;
       BigInteger b = n.shiftRight(5).add(new BigInteger("8"));
       while(b.compareTo(a) >= 0) {
           BigInteger mid = a.add(b).shiftRight(1);
           if(mid.multiply(mid).compareTo(n) > 0) b =
mid.subtract(BigInteger.ONE);
           else a = mid.add(BigInteger.ONE);
       }
       return a.subtract(BigInteger.ONE);
   }

   public static void main(String[] args) {
       Scanner input = new Scanner(System.in);
       while(true) {
           BigInteger n = input.nextBigInteger();
           if(n.compareTo(BigInteger.ZERO)==0)
               break;
           System.out.println(sqrt(n).pow(2).toString());
       }
   }
}
```

Capítulo 6

Árvores

6.1 Árvore de Intervalos

Pode ser utilizada quando as chaves a serem inseridas pertencem a um intervalo pequeno (menor que 10^6). Inicializar o vetor a com zeros.

Complexidade: inserção, busca, remoção e consulta do k-ésimo menor elemento em tempo $O(log(n))$.

Listing 6.1: Árvore de Intervalos

```
#define TAM 131072

int a[2*TAM+1];

void insere(int j){
    int k;
    k = j+TAM;
    while(k != 0 ){
       a[k]++;
       k/=2;
    }
}

void remove(int j){
    int k;
    k = j+TAM;
    while(k != 0 ){
       a[k]--;
       k/=2;
    }
}

int acha(int k) {
    int j= 1;
    while( j<TAM ){
        j = j*2;
        if(a[j] < k){
            k=k-a[j];
            j++;
        }
    }
    if(a[j] < k) j++;

    return j-TAM;
}
```

6.2 Árvore de Segmentos

O exemplo no código dado calcula a maior soma em qualquer segmento dado. Mas a estrutura é genérica.

Complexidade: Construção em $O(n)$. Inserção, Atualização e Busca em $O(logn)$.

Utilização: Crie a árvore usando o operador new, passando o intervalo $1..n$.

Listing 6.2: Árvore de Segmentos

```
#define NN 53000

int val[NN];

struct segt {
    segt *e,* d;
    int le, ld; // limite esquerdo e direito
    int mi, mf, t, m;

    segt(int le, int ld): le(le), ld(ld), e(NULL), d(NULL) {
        if (le == ld) { // folha
            mi = mf = t = m = val[le-1];
        } else {
            int mid = (le+ld)/2;
            e = new segt(le, mid);
            d = new segt(mid+1, ld);
            // especifico do problema
            mi = max(e->t + d->mi, e->mi);
            mf = max(d->t + e->mf, d->mf);
            t = e->t + d->t;
            m = max(e->mf+d->mi, max(e->m, d->m));
        }
    }
    segt(int mi, int mf, int t, int m): mi(mi), mf(mf), t(t), m(m), e(NULL), d(NULL)
        {}
    segt() : e(NULL), d(NULL) {}

    //~segt() { delete e; delete d; }
};

#define inf 999999

segt query(segt *x, int i, int j) {
    if (i > x->ld || j < x->le)
        return segt(-inf, -inf, -inf, -inf);

    if (i <= x->le && x->ld <= j)
        return segt(x->mi, x->mf, x->t, x->m);

    segt se = query(x->e, i, j);
    segt sd = query(x->d, i, j);

    segt res; // mesmas equacoes da criacao, Ctrl+C Ctrl+V
    res.mi = max(se.t + sd.mi, se.mi);
    res.mf = max(sd.t + se.mf, sd.mf);
    res.t = se.t + sd.t;
    res.m = max(se.mf+sd.mi, max(se.m, sd.m));

```

```
    return res;
}
```

6.3 Fenwick Tree (BIT)

Complexidade: $O(logn)$ para inserção e busca.

Utilização: Inicie com o vetor tree zerado, add aumenta a frequencia do valor x em 1, find retorna a soma cumulativa das frequencias de 1 ate ind. **Atenção:** sempre use a BIT a partir do índice 1, não insira elementos na posição 0!

Listing 6.3: Binary Indexed Tree

```
int n;
int tree[(1<<18)+1];

int add(int ind){
    while( ind <= n ){
        tree[ind]++;
        ind += (ind & -ind);
        // ind = ind + 2^r , r = indice do ultimo digito 1, da rep. binaria
    }
    return 0;
}

int find(int ind){
    int s = 0;
    while(ind > 0){
        s += tree[ind];
        ind -= (ind & -ind);
    }
    return s;
}
```

6.4 Fenwick Tree 2D (BIT)

Complexidade: $O(log^2 n)$ para inserção e busca.

Utilização: Inicie com o vetor tree zerado, update aumenta a frequencia do valor (x,y) em val, find retorna a soma cumulativa das frequencias do retângulo de (1,1) até (x,y), inclusive. **Atenção:** sempre use a BIT a partir do índice 1, não insira elementos na posição 0!

Listing 6.4: Binary Indexed Tree 2D

```
#define NN 1025
int tree[NN][NN];

void update(int x, int y, int val){
    int y1;
    while (x < NN){
        y1 = y;
        while (y1 < NN){
            tree[x][y1] += val;
            y1 += (y1 & -y1);
        }
        x += (x & -x);
    }
}
```

```
int query(int x, int y) {
    int y1, res = 0;
    while (x > 0) {
        y1 = y;
        while (y1 > 0) {
            res += tree[x][y1];
            y1 -= (y1 & -y1);
        }
        x -= (x & -x);
    }
    return res;
}
```

6.5 Lowest Common Ancestor (LCA)

Listing 6.5: Lowest Common Ancestor

```
#define LIM 50000

struct no{
    int adj[10];
    int custo[10];
    int q;
};

no TREE[LIM];
int n,m;

int readdata(){
    if( scanf("%d %d",&n,&m) != 2 ) return 0;
    int i,c;
    int x,y;
    char lixo[10];

    for(i=0;i<n;i++) TREE[i].q = 0;

    for(i=0;i<m;i++){
        scanf("%d %d %d %s",&x,&y,&c,lixo);

        TREE[x].adj[ TREE[x].q ] = y;
        TREE[x].custo[ TREE[x].q ] = c;
        TREE[x].q++;

        TREE[y].adj[ TREE[y].q ] = x;
        TREE[y].custo[ TREE[y].q ] = c;
        TREE[y].q++;
    }

    return 1;
}

int LV[LIM];
int T[LIM]; // Diz que e o pai do no [i]
int c_pai[LIM]; //Custo para o pai do no [i]

```

```
int P[LIM][20]; // 2^j-esimo ancestral
int COST[LIM][20]; //custo para os 2^j-esimos ancestrais

int mark[LIM];

int dfs(int no,int level){

    LV[no] = level;

    int i;
    for(i=0;i < TREE[no].q; i++){
        if( !mark[ TREE[no].adj[i] ] ){
            mark[ TREE[no].adj[i] ]  = 1;

            T[ TREE[no].adj[i] ] = no;
            c_pai[ TREE[no].adj[i] ] = TREE[no].custo[i];
            dfs(TREE[no].adj[i],level + 1);
        }
    }

    return 0;
}

int preprocess(){

    int i,j;

    for(i=1;i<=n;i++){
        P[i][0] = T[i];
        if( P[i][0] != -1 ) COST[i][0] = c_pai[i];
    }

    for(j=1; 1<<j < n ;j++){
        for(i=1;i<=n;i++){
            if( P[i][j-1] != -1 ){
                P[i][j] = P[ P[i][j-1] ][ j-1 ];
                if( P[i][j] != -1 )
                    COST[i][j] = COST[i][j-1] + COST[ P[i][j-1] ][j-1];
            }
        }
    }

    return 0;
}

int lca(int no1,int no2){

    int i,j;
    int dist = 0;

    //Sobe o primeiro no, ate eles ficarem no mesmo level
    if( LV[no1] != LV[no2] ){
        for(j=15;j>=0;j--){
            if( P[no1][j] != -1 ){
                if( LV[ P[no1][j] ] >= LV[no2] ){
                    dist += COST[no1][j];
                    no1 = P[no1][j];
                }
            }
```

```
        }
    }

    // Se o No2 eh ancestral de No1
    if( no1 == no2 ) return dist;

    for(j=15;j>=0;j--){
        if( P[no1][j] != -1 && P[no1][j] != P[no2][j] ){
            dist += COST[no1][j];
            dist += COST[no2][j];
            no1 = P[no1][j];
            no2 = P[no2][j];
        }
    }

    if( no1 == no2 ) return dist;

    return dist + COST[no1][0] + COST[no2][0];
}

int process(){

    memset(mark,0,sizeof(mark));
    memset(T,-1,sizeof(T));
    memset(P,-1,sizeof(P));
    memset(COST,0,sizeof(COST));
    memset(c_pai,-1,sizeof(c_pai));

    //Determina o level de cada no
    dfs(mark[1] = 1,0);

    preprocess();  // N log N

    int q;
    scanf("%d",&q);

    int no1,no2;

    while(q--){
        scanf("%d %d",&no1,&no2);

        // troca os valores
        if( LV[no1] < LV[no2] ) no1 ^= no2 ^= no1 ^= no2;

        printf("%d\n",lca(no1,no2));  // log N
    }

    return 0;
}
```

Capítulo 7

Grafo

7.0.1 Union-find

Listing 7.1: Union find

```
int uf[NV];
int h[NV];

int find(int x) { return (uf[x] == x) ? x : (uf[x] = find(uf[x]));  }
void uni(int x, int y) {
    x = find(x); y = find(y);
    if (h[x] == h[y]) {
        h[x]++; uf[y] = x;
    } else if (h[x] < h[y]) uf[x] = y;
    else uf[y] = x;
}

void init() {
    for (int i = 1; i <= n; i++) {
        uf[i] = i;
        h[i] = 0;
    }
}
```

7.0.2 BFS (Largura)

Resolução do *UVa 439: Knight Moves* na seção 17.3.

7.0.3 Pontes

Resolução do *Séries de Tubos: Pontes* na seção 17.1.

7.0.4 Vértices de Articulação

Listing 7.2: Vértices de Articulação

```
int grafo[101][101];
int LOW[101],critic[101],dfsnumber[101];

int n,dfn,raiz;

char line[1000];
```

```
// vertices de articulacao marcados em critic

int DFS(int k,int pai){

    dfsnumber[k] = dfn++;
    LOW[k] = dfsnumber[k];

    int i,filhos=0;

    for(i=0;i<n;i++){
        if( grafo[k][i] ){
            if( pai == i ) continue;

            if( dfsnumber[i] == 0 ){
                filhos++;
                DFS(i,k);
                if( LOW[i] >= dfsnumber[k] ){
                    if( raiz != k || filhos > 1 ){
                        critic[k] = 1;
                    }
                }
                if( LOW[i] < LOW[k] ) LOW[k] = LOW[i];
            }
            else{
                if( dfsnumber[i] < LOW[k] ) LOW[k] = dfsnumber[i];
            }
        }
    }

    return 0;
}

int process(){
    memset(dfsnumber,0,sizeof(dfsnumber));
    memset(critic,0,sizeof(critic));

    dfn = 1;
    k = 0;

    raiz = k;
    DFS(k, -1);

    k=0;
    for(i=0;i<n;i++)
        if( critic[i] )
            k++;

    printf("%d\n",k);
    return 0;
}
```

7.0.5 Bi-coloração

Verifica se um dado grafo, representado por matriz de adjacência, é bi-partido (bi-colorido) ou não, através de uma busca em profundidade que marca as cores de vértices adjacentes. Caso haja conflito na designação de cores, o grafo não é bi-colorido. As seguintes afirmações são equivalentes:

- O grafo é bi-colorido;

- O grafo não possui ciclos de tamanho ímpar;
- O grafo pode ser dividido em dois conjuntos de forma que não exista arestas ligando dois vértices no mesmo conjunto (ou seja, o grafo é bi-partido).

Complexidade: Linear no tamanho do grafo.

Suposição: Existe uma matriz de adjacência global, chamada **matriz[][]**, e também existe um vetor global iniciado com zero chamado **vis[]**.

Entrada: **(int) v**, que contém o número do vértice atual; **(int) c**, que contém a cor (pode ter os valores 1 ou -1) que vértice **v** irá receber.

Saída: Retorna um **(int)** que pode ter o valor 1, caso o grafo seja bi-partido; e 0 caso contrário.

Na chamada inicial, **v** deve ter um valor entre $0 \dots n-1$ e **c** deve ter o valor 1 ou -1.

Listing 7.3: Verificação de bi-partição

```
int bicolorido(int v, int c) {
    int i;

    vis[v] = c;
    for(i=0; i < vertices; i++)
        if(matriz[v][i]) {
            if(vis[i] == 0) {
                if(!bi(i, c*-1)) return 0;}
            else if(vis[i] == c) return 0;
        }
    return 1;
}
```

7.1 Algoritmos de Caminho Mínimo

7.1.1 Dijkstra

Algoritmo guloso para encontrar o menor caminho entre o vértice origem à todos os demais. A cada iteração escolhe o vértice de menor custo que ainda não foi visitado, e a partir deste verifica se há um caminho de menor custo entre seu adajcente e ele.

Suposição: Utiliza uma matriz de adjacência, em que não haver arestas entre dois vértices significa ter valor zero na matriz. Se podem existir arestas de peso zero, então a matriz deve ser inicializada com outro valor e a condição da linha 16 deve ser alterada. Também não aceita arestas com peso negativo.

Complexidade: $O(N^2)$.

Entrada: **(int) origem**, que contém o número do vértice de origem; **(int) destino**, que contém o número do vértice de destino; e **(int) vertices**, que contém a quantidade de vertices no grafo (numerados de 0..n-1).

Saída: Retorna um **(int)** com o custo para sair da origem e chegar no destino.

Listing 7.4: Dijkstra

```
 1 int Dijkstra(int origem, int destino, int vertices) {
 2     int menor, i, j;
 3     int vdijs[2][MAX];
 4     int visitado[MAX];
 5
 6     for (i=0; i < vertices; i++) {
 7         visitado[i] = 0;
 8         vdijs[0][i] = INT_MAX;      /* guarda o custo */
 9         vdijs[1][i] = -1;           /* vetor de antecedencia */
10     }
11     vdijs[0][origem] = 0;
12     vdijs[1][origem] = origem;
13
14     for (i=0; i < vertices; i++) {
15         for (j=0; j< vertices; j++) {
16             if (matriz[origem][j])
17                 if (vdijs[0][j] > vdijs[0][origem] + matriz[origem][j]) {
18                     vdijs[0][j] = vdijs[0][origem] + matriz[origem][j];
19                     vdijs[1][j] = origem;
20                 }
21         }
22
23         visitado[origem] = 1;
24         menor = origem;
25         for (j=0; j < vertices; j++) {
26             if (!visitado[j] && vdijs[0][j] != INT_MAX) {
27                 if (menor == origem) {
28                     menor = j;
29                 }
30                 else if (vdijs[0][j] < vdijs[0][menor]) {
31                     menor = j;
32                 }
33             }
34         }
35         origem = menor;
36     }
37
38     return vdijs[0][destino];
39 }
```

O exercício resolvido *11492: Babel* se encontra na seção 17.12.

7.1.2 Bellman-Ford

Algoritmo para achar o menor caminho entre dois vértices que aceita arestas de peso negativo, e é capaz de identificar ciclos negativos. A seguir são dadas duas implementações: a primeira é simples de codificar, e a segunda tem o código maior mas é mais eficiente (complexidade de tempo das duas é a mesma).

Suposição: o grafo é tratado como direcionado. Se for um grafo não direcionado, duplicar as arestas invertendo a *origem* e o *destino*.

Complexidade: $O(NE)$, onde N é quantidade de vértices e E a quantidade de arestas.

Entrada: lista de arestas nos vetores *ini*, fim e c; vértice inicial *inicio*

Saída: Retorna um booleano (se encontrou ciclo negativo ou não), e os vetores d e *ant* preenchidos (vetor de distâncias e vetor de antecessores).

Listing 7.5: Bellman-Ford simples

```
// numero de vertices
#define NV 2200
// numero de arestas
#define NA 2200
#define INF (1<<30)

// lista de arestas do grafo: inicio, destino e custo
int ini[NA], dest[NA], c[NA];
// vetor de distancias e vetor de antecessores
int d[NV], ant[NV];

int bellmanford(int inicio, int n, int m) {
    for (int i = 0; i < NV; i++) d[i] = INF;
    d[inicio] = 0;
    int neg = 0;
    for (int i = 0; i < n; i ++) {
        for (int j = 0; j < m; j++)
            if (d[dest[j]] > d[ini[j]] + c[j]) {
                d[dest[j]] = d[ini[j]] + c[j];
                ant[dest[j]] = ini[j];
                if (i == n-1) neg = 1;
            }
    }
    return neg;
}
```

Listing 7.6: Bellman-Ford com fila

```
#define NV 1100
#define NA 2200

typedef pair<int,int> pii;
#define mp make_pair
#define inf (1<<30)

int g[NV][NA]; int d[NV];
int dist[NV];
pii ar[NA];
int w[NA];
int ant[NV];

int bellmanford(int ini) {
    queue<pii> q;
    q.push(mp(0, 0));
```

```
    for (int i = 0; i < NV; i++) dist[i] = inf;
    dist[ini] = 0;

    int neg = 0;
    while (!q.empty() && !neg) {
        pii xp = q.front(); q.pop();
        int x = xp.first, nv = xp.second;

        if (nv == n) break;

        for (int i = 0; i < d[x]; i++) {
            int u = ar[g[x][i]].second, p = w[g[x][i]];
            if (dist[u] > dist[x] + p) {
                dist[u] = dist[x] + p;
                ant[u] = x;
                q.push(mp(u, nv+1));
                if (nv+1 == n) neg = 1;
            }
        }
    }
    return neg;
}
```

O exercício resolvido *Wormholes* se encontra na seção 17.4.

7.1.3 Floyd-Warshall

Algoritmo mais fácil e curto de implementar, acha o caminho mínimo de todos os vértices para todos os vértices. Bem como o Bellman-Ford, ele também é capaz de achar um ciclo negativo, basta olhar a sua diagonal ($i = j$) e ver se algum valor está abaixo de zero.

Suposição: Existe uma matriz de adjacência **g**, a qual foi inicializada com 0 na diagonal, e se não houver caminho entre os vértices i e j, então $g[i][j] = +\infty$. Além disso, os índices dos laços assumem que os vértices estão numerados de $1, \cdots, n$.

Complexidade: $O(N^3)$, onde N é a quantidade de vértices.

Entrada: (int) n, que contém a quantidade de vértices.

Saída: Em $g[i][j]$ fica o valor do caminho mínimo da origem i com destino j.

Listing 7.7: Floyd-Warshall em Matriz de Adjacência

```
int min(int a, int b) { return a > b ? b : a; }

void FloydWarshall(int n) {
    int i, j, k;

    for(k=1; k <= n; k++)
        for(i=1; i <= n; i++)
            for(j=1; j <= n; j++)
                g[i][j] = min(g[i][j], g[i][k]+g[k][j]);
}
```

Minimax

Modificação do algoritmo de Floyd-Warshall para achar o caminho em que o valor da maior aresta é minimizado.

Suposição: Existe uma matriz de adjacência **g**, a qual foi inicializada com 0 na diagonal, e se não houver caminho entre os vértices i e j, então $g[i][j] = +\infty$. Além disso, os índices dos laços assumem que os vértices estão numerados de $1, \cdots, n$.

Complexidade: $O(N^3)$, onde N é a quantidade de vértices.
Entrada: **(int) n**, que contém a quantidade de vértices.
Saída: Em $g[i][j]$ fica o valor minimax do caminho da origem i com destino j.

Listing 7.8: Minimax em Matriz de Adjacência

```
int min(int a, int b) { return a > b ? b : a; }
int max(int a, int b) { return a > b ? a : b; }

void Minimax(int n) {
    int i, j, k;

    for(k=1; k <= n; k++)
        for(i=1; i <= n; i++)
            for(j=1; j <= n; j++)
                g[i][j] = min(g[i][j], max(g[i][k],g[k][j]));
}
```

7.2 Fluxo Máximo

7.2.1 Ford-Fulkerson

Ford-Fulkerson é um método para se achar o maior fluxo em um grafo, segundo o método proposto por eles basta seguir os seguintes passos:

1. Descubra qualquer caminho que conecte a origem ao destino, em que todas as arestas tenham capacidade estritamente positiva. Se não houver caminho, encerre e retorne o fluxo máximo;

2. Deste caminho, determine f, que é a menor capacidade do caminho (ou seja, o gargalo). Some f ao fluxo máximo atual;

3. De cada aresta ”*forward*”(no sentido do fluxo da origem ao destino), subtraia f, e de cada aresta ”*backward*”, adicione f;

4. Volte ao passo 1.

Complexidade: Desconhecida. Para ter tempo polinomial se deve usar BFS no lugar de uma DFS para achar um caminho entre a origem e o destino (ver Edmonds-Karp).
Entrada: **(int) origem**, que contém o número do vértice de origem; **(int) destino**, que contém o número do vértice de destino. Note que a quantidade de vértices e a matriz são variáveis globais.
Saída: Retorna um **(int)** com o valor do fluxo máximo.

Listing 7.9: Ford-Fulkerson com DFS

```
int DFS(int v, int destino, int menor) {
    int i, auxiliar;

    if (v == destino)
        return menor;

    for (i=0; i < nvertices; i++)
        if (m[v][i] && !visitado[i]) {
            visitado[i] = 1;
            if ((auxiliar = DFS(i, destino, menor > m[v][i] ? m[v][i] : menor))) {
                menor = auxiliar;
                m[v][i] -= menor;
                m[i][v] += menor;
                return menor;
```

```
            }
            visitado[i] = 0;
        }

    return 0;
}

int MaxFlow(int origem, int destino) {
    int i, f, auxiliar;

    f = 0;
    do {
        for (i=0; i < nvertices; i++) visitado[i] = 0;
        auxiliar = 0;
        visitado[0] = 1;
        auxiliar = DFS(origem, destino, INFINITO);
        f += auxiliar;
    } while (auxiliar);

    return f;
}
```

7.2.2 Edmonds-Karp

Ford-Fulkerson deixa em aberto o método para se encontrar um caminho entre a origem e o destino, e o algoritmo de Edmonds-Karp é a implementação do Ford-Fulkerson com BFS, pois é garantindo que com BFS roda em tempo polinomial.

Complexidade: $O(VE^2)$.

Entrada: s e t, que são a fonte e o vertedouro. Usar a função insere_aresta para inserir uma aresta **direcionada** no grafo.

Saída: Retorna um **(int)** com o valor do fluxo máximo.

Listing 7.10: Edmonds-Karp (Implementação do Quake)

```
#define MAXV 200
#define MAXA 3000
#define INF 9999999

int g[MAXV][MAXA], d[MAXA];
int dest[MAXA], cap[MAXA];
int va[MAXA], vv[MAXA], v[MAXA];

void insere_aresta(int u, int v, int c) {
    g[u][d[u]++] = na;
    dest[na] = v;
    cap[na] = c;
    na++;
    g[v][d[v]++] = na;
    dest[na] = u;
    cap[na] = 0;
    na++;
}

int edmondskarp(int s, int t) {
    int res = 0;
    while (1) {
        queue<int> q;
        q.push(s);
```

```
        memset(v, 0, sizeof v);
        v[s] = 1;
        int ok = 0;
        while (!q.empty() && !ok) {
            int x = q.front(); q.pop();
            for (int i = 0; i < d[x]; i++) {
                int a = dest[g[x][i]], c = cap[g[x][i]];
                if (!v[a] && c > 0) {
                    va[a] = g[x][i]; vv[a] = x;
                    v[a] = 1;
                    q.push(a);

                    if (a == t) {
                        ok = 1;
                        break;
                    }
                }
            }
        }

        if (!ok) break;

        int c = INF;
        for (int i = t; i != s; i = vv[i])
            c = min(c, cap[va[i]]);

        res += c;

        for (int i = t; i != s; i = vv[i]) {
            cap[va[i]] -= c;
            cap[va[i]^1] += c;
        }
    }
    return res;
}
```

Listing 7.11: Edmonds-Karp (Implementação de Lucas)

```
#include <cstdio>
#include <iostream>
#include <string>
#include <queue>
#include <limits.h>

#define MAX 150

using namespace std;

typedef struct _grafo {
    int m[MAX][MAX];
    int visitado[MAX];
    int antecedencia[MAX];
    int nvertices;
} grafo;

grafo g;


```

```
int BFSFluxo(int fonte, int sumidouro) {
    int i, j, v, gargalo;
    queue<int> q;

    for (i=0; i < g.nvertices; i++) {
        g.visitado[i] = 0;
        g.antecedencia[i] = -1;
    }
    g.visitado[fonte] = 1;
    g.antecedencia[fonte] = fonte;
    q.push(fonte);
    while (!q.empty()) {
        v = q.front();
        q.pop();
        for (i=0; i < g.nvertices; i++)
            if (g.m[v][i] && !g.visitado[i]) {
                g.antecedencia[i] = v;
                g.visitado[i] = 1;
                q.push(i);
            }
    }

    if(g.visitado[sumidouro]) {
        v = sumidouro;
        gargalo = INT_MAX;
        do {
            gargalo = gargalo > g.m[g.antecedencia[v]][v] ? g.m[g.antecedencia[v]][v]
                 : gargalo;
            v = g.antecedencia[v];
        } while (v != fonte);

        v = sumidouro;
        do {
            g.m[g.antecedencia[v]][v] -= gargalo;
            g.m[v][g.antecedencia[v]] += gargalo;
            v = g.antecedencia[v];
        } while (v != fonte);
        return gargalo;
    }
    else {
        return 0;
    }
}


int MaxFlow(int fonte, int sumidouro) {
    int f, maximo;

    maximo = 0;
    while ((f = BFSFluxo(fonte, sumidouro)) != 0) {
        maximo += f;
    }

    return maximo;
}
```

7.2.3 Dinic

Implementação Igor Naverniouk.

Complexidade: $O(V^2E)$.

Entrada: (int [][]) cap matriz de capacidades; **(int [][]) adj, (int []) deg** lista de adjacências;

Saída: Retorna um **(int)** com o valor do fluxo máximo. O vetor **prev** contém o corte mínimo.

Listing 7.12: Dinic

```
#define NN 1024

int cap[NN][NN], deg[NN], adj[NN][NN];

int q[NN], prev[NN];

int dinic( int n, int s, int t ) {
    int flow = 0;

    while ( 1 ) {
        memset( prev, -1, sizeof( prev ) );
        int qf = 0, qb = 0;
        prev[q[qb++] = s] = -2;
        while( qb > qf && prev[t] == -1 )
            for( int u = q[qf++], i = 0, v; i < deg[u]; i++ )
                if( prev[v = adj[u][i]] == -1 && cap[u][v] )
                    prev[q[qb++] = v] = u;

        if ( prev[t] == -1 ) break;

        for ( int z = 0; z < n; z++ ) if( cap[z][t] && prev[z] != -1 )
        {
            int bot = cap[z][t];
            for( int v = z, u = prev[v]; u >= 0; v = u, u = prev[v] )
                bot = min(bot, cap[u][v]);
            if( !bot ) continue;

            cap[z][t] -= bot;
            cap[t][z] += bot;
            for( int v = z, u = prev[v]; u >= 0; v = u, u = prev[v] ) {
                cap[u][v] -= bot;
                cap[v][u] += bot;
            }
            flow += bot;
        }
    }

    return flow;
}
```

7.3 Fluxo máximo e Corte Mínimo

Corte é o problema de se retirar determinados arcos de modo que não haja fluxo entre uma origem e um destino. Corte mínimo almeja retirar os arcos que somem a menor capacidade de forma a não ter fluxo entre a origem e o destino.

Teorema: O valor do corte mínimo é igual ao valor do fluxo máximo.

Segundo Sedgewick, os arcos pertencentes ao corte mínimo são aqueles que, em cada caminho entre a origem e o destino, tem o arco totalmente saturado (sem possibilidade de expandir o fluxo). Logo, para identificar esses arcos, se pode usar a última execução do BFS do Edmonds-Karp, e pegar todos os arcos que ligam vértices visitados com vértices não visitados.

Observação: Para o código abaixo é utilizado o código do Listing 7.11, além da *struct* modificada (inclui o **original[MAX][MAX]** para guardar a matriz original, que será usada para saber se havia ou não conexão entre os vértices, já que os vérices saturados terão valor 0 após a execução do fluxo máximo).

Complexidade: $O(VE)$, além do tempo de execução do Edmonds-Karp.

Entrada: **(grafo *) g**, que é um ponteiro pra estrutura grafo declarada no começo do código (variável global). É necessário que o **fluxo máximo seja executado antes!**

Saída: Imprime na tela os arcos do corte mínimo.

Listing 7.13: Edmonds-Karp com Min-Cut (Implementação de Lucas)

```
void MinCut() {
    int i, j;
    /* Eh necessario que o Edmonds-Karp seja executado antes! */
    for (i=0; i < g.nvertices; i++)
        if(g.visitado[i]) {
            for (j=0; j < g.nvertices; j++) {
                if (g.original[i][j] && !g.visitado[j]) {
                    printf("%d %d\n", i, j);
                }
            }
        }
}
```

Ver problema resolvido *10480: Sabotage* na seção 17.10.

7.4 Fluxo Máximo de Custo Mínimo

Implementação do Shi.

Listing 7.14: Max-Flow Min-Cost

```
#include<iostream>
#include<cstdio>
#include<cstdlib>
#include<cstring>
#include<climits>
#include<queue>

#define N 105
#define INF INT_MAX
#define pot(u,v) d[u] + pi[u] - pi[v]
#define SETINF(x) for(int i=1;i<netsize;i++) x[i] = INF;
#define SET0(x) memset(x,0,sizeof(x));


/**
    Fluxo Máximo de Custo Mínimo - Algoritmo de caminhos minimos sucessivos
    Nós de onde o fluxo saí, colocar em demand[i] a quantidade de fluxo
    Nós onde o fluxo chega, colocar em demand[i] a quantidade de fluxo que chega
        negativa
    Depois de inicializar o vetor demand, rodar init.
    A contagem dos nós é a partir do 1, passar para init a quantidade de nós+1.
    cap[i][j] guarda a capacidade do nó i,j e cost o seu custo.
    Rodar minCostFlow(), a o custo estará na variável flow_cost e o fluxo na variável
        flow.

```

```
    PS: Se o grafo tiver arestas negativas, deixar a chamada do bellmanford() no init. Se não for
    o caso, pode-se trocar essa chamada pela linha comentada.

**/

using namespace std;
typedef pair<int,int> par;

int SOURCE, SINK, BELLMAN_VERTEX;

int netsize; // Size of the network
int cap[N][N], flownet[N][N], cost[N][N], demand[N]; // capacidade, fluxo, custo e demanda
int d[N], v[N], prev[N]; // Array de distancia, visitados e antecessor para o dijkstra
int pi[N]; // Função potencial dos nós
int flow, flow_cost; // Fluxo máximo e custo do fluxo máximo

void bellman_ford(){
    SETINF(pi);
    pi[BELLMAN_VERTEX] = 0;
    for(int i=1;i<netsize;i++)
        for(int j=1;j<netsize;j++)
            for(int k=1;k<netsize;k++)
                if(pi[j] + cost[j][k] < pi[k])
                    pi[k] = pi[j] + cost[j][k];
}

void init(int size){
    netsize = size+3;
    SOURCE = size;
    SINK = size+1;
    BELLMAN_VERTEX = size+2;

    for(int i=1;i<netsize;i++)
        for(int j=1;j<netsize;j++)
            cap[i][j] = flownet[i][j] = cost[i][j] = 0;

    for(int i=1;i<netsize-1;i++)
            cost[BELLMAN_VERTEX][i] = 0; // Pra rodar bellman-ford
    for(int i=1;i<netsize-3;i++)
        if(demand[i]>0) cap[SOURCE][i] = demand[i];
        else cap[i][SINK] = -demand[i];
    // SET0(pi);
    bellman_ford();
}

int dijkstra(){
    SET0(v);
    SETINF(d);
    prev[SOURCE] = SOURCE;
    priority_queue<par,vector<par>,std::greater<par> > q;
    q.push(par(0,SOURCE));
    d[SOURCE] = 0;
    while(!q.empty()){
        par p = q.top(); q.pop();
        int i = p.second;
        if(v[i]) continue;
        v[i] = 1;
```

```
        for(int j=1;j<netsize;j++){
            if(v[j] || !cap[i][j]) continue;
            // Tenta aumentar a capacidade na aresta de ida
            if( cap[i][j]>flownet[i][j] && d[j] > pot(i,j) + cost[i][j] )
                d[j] = pot(i,j) + cost[i][j], prev[j] = i, q.push( par(d[j],j) );
            // Tenta anular o fluxo
            if( flownet[j][i] &&  d[j] > pot(i,j) - cost[j][i] )
                d[j] = pot(i,j) - cost[j][i], prev[j] = -i, q.push( par(d[j],j) );
        }
    }

    for(int i=0;i<netsize;i++) pi[i] += d[i];
    if(!v[SINK]) return 0;
    int flow_cap = INF;
    for(int w = SINK; prev[w]!=SOURCE; w = prev[w]>0?prev[w]:-prev[w]){
        int f = prev[w];
        if(f>0 &&  cap[f][w]-flownet[f][w] < flow_cap ) flow_cap =  cap[f][w]-flownet[f][w];
        else if ( f<0 && flownet[w][-f] < flow_cap )  flow_cap = flownet[w][-f];

    }
    for(int w = SINK; prev[w]!=SOURCE; w = prev[w]>0?prev[w]:-prev[w]){
        int f = prev[w];
        if(f>0) flownet[f][w] += flow_cap;
        else flownet[w][-f] -= flow_cap;
    }
    return flow_cap;
}

void minCostFlow(){
    int r;
    flow = 0, flow_cost = 0;
    while( (r=dijkstra()) != 0 ) flow+= r;
    for(int i=1;i<netsize;i++)
        for(int j=1;j<netsize;j++)
            if(cost[i][j]>0) flow_cost += flownet[i][j] * cost[i][j];
}

/*
    Exemplo: UVa 10806 - Dijkstra, Dijkstra
*/

int main(void){
    int m,n, from,to,time;
    while(1){
        scanf("%d",&n);
        if(!n) return 0;
        scanf("%d",&m);
        for(int i=1;i<=n;i++)
            demand[i] = 0;
        demand[1] = 2;
        demand[n] = -2;
        init(n+1);
        for(int i=0;i<m;i++){
            scanf("%d %d %d",&from,&to,&time);
            cap[from][to] = cap[to][from] = 1;
            cost[from][to] = cost[to][from] = time;
        }
        minCostFlow();
```

```
        if(flow<2) printf("Back to jail\n");
        else printf("%d\n",flow_cost);
    }
}
```

7.5 Árvores Geradoras Mínimas (Minimum Spanning Trees)

7.5.1 Algoritmo de Prim

Ver problema resolvido *10034: Freckles* na seção 17.6.

7.5.2 Algoritmo de Kruskal

Listing 7.15: Kruskal

```
#define NN 100100
#define NV 10100

typedef pair<int,pair<int,int> > piii;
piii v[NN]; // custo, inicio, fim

int uf[NV];
int h[NV];

int find(int x) { return (uf[x] == x) ? x : (uf[x] = find(uf[x]));  }
void uni(int x, int y) {
    x = find(x); y = find(y);
    if (h[x] == h[y]) {
        h[x]++; uf[y] = x;
    } else if (h[x] < h[y]) uf[x] = y;
    else uf[y] = x;
}

void kruskal(int n, int m) {
    for (int i = 1; i <= n; i++) {
        uf[i] = i;
        h[i] = 0;
    }

    sort(v, v+m);
    int res = 0, ng = n;
    for (int i = 0; i < m && ng > 1; i++) {
        if (find(v[i].second.first) != find(v[i].second.second)) {
            uni(v[i].second.first, v[i].second.second);
            res += v[i].first;
            ng--;
        }
    }
}
```

7.6 Emparelhamento

7.6.1 Emparelhamento de Cardinalidade Máxima

Resolução com fluxo em DFS. Lembre-se: deixe o grafo **direcionado**, pois assim o algoritmo é mais eficiente.

Complexidade: $O(mn^2)$
Entrada: **(int [][]) cap** matriz de capacidades.
Saída: **(int)** quantidade de arestas no emparelhamento. O vetor fluxo (global) contém o emparelhamento.

Listing 7.16: Emparelhamento

```
#define MAXN 210

int nvertices;
int cap[MAXN][MAXN], fluxo[MAXN][MAXN];
char v[MAXN];

int dfs(int u, int t) {
    if (u == t) return 1;
    for (int i = 0; i < nvertices; i++)
        if (!v[i] && ((cap[u][i] - fluxo[u][i]) > 0)) {
            v[u] = 1;
            if (dfs(i, t)) {
                fluxo[u][i]++; fluxo[i][u]--;
                return 1;
            }
            v[u] = 0;
        }
    return 0;
}

int emparelha(int n, int s, int t) {
    int res = 0;
    nvertices = n;
    memset(fluxo, 0, sizeof fluxo);
    while (1) {
        memset(v, 0, sizeof v);
        if (!dfs(s, t)) break;
        res++;
    }

    return res;
}
```

7.6.2 Minimum vertex cover

Calcula o minimum vertex cover em grafos bipartidos (utilizando o Teorema de König). Faz uma DFS na rede residual. Os vértices da cobertura são os visitados do lado direito e os não visitados do lado esquerdo. Implementação feita supondo que o código calculou o fluxo máximo usando o algoritmo Dinic.

Complexidade: $O(n)$, depois de calculado o fluxo.

Entrada: **(int [][]) cap** matriz de capacidades (direcionada), **(int [][]) adj, (int []) deg** lista de adjacências (não direcionada), **(int) n** índice do último vértice do lado direito, **(int) k** índice do último vértice do lado esquerdo. Supõe o vértice 0 como fonte.

Saída: Imprime os vértices da cobertura (modificar de acordo com o problema).

Listing 7.17: Minimum Vertex Cover

```
char v[NN];

void dfs(int u) {
    v[u] = 1;
    for (int i = 0; i < deg[u]; i++)
        if (!v[adj[u][i]] && (cap[u][adj[u][i]] > 0))
```

```
            dfs(adj[u][i]);
}

void do_cover(int n, int k) {
    memset(v, 0, sizeof v);
    dfs(0);
    for (int i = 1; i <= k; i++)
        if (!v[i]) printf(" r%d", i);
    for (int i = k+1; i <= n; i++)
        if (v[i]) printf(" c%d", i-k);
}
```

7.6.3 Emparelhamento Máximo de Custo Máximo

Algoritmo húngaro (Kuhn-Munkres). **Atenção:** essa implementação calcula o emparelhamento de custo **máximo**. Para reduzir ao caso mínimo, substitua os valores da matriz pelo complemento em relação ao maior valor possível.

Complexidade: $O(n^3)$

Entrada: **(int [][]) custo** matriz de custos, **(int) n** dimensão da matriz (quadrada). **Atenção:** Nessa implementação, os vértices estão numerados de 0 a $n-1$.

Saída: **(int)** custo. O vetor xy (global) contém o emparelhamento.

Listing 7.18: Algoritmo húngaro

```
#define N 110
#define INF 100000000

int cost[N][N];
int n, max_match;
int lx[N], ly[N], xy[N], yx[N];
bool S[N], T[N];
int slack[N], slackx[N];

int prev[N];

void update_labels() {
    int x, y, delta = INF;
    for (y = 0; y < n; y++)
        if (!T[y]) delta = min(delta, slack[y]);

    for (x = 0; x < n; x++)
        if (S[x]) lx[x] -= delta;

    for (y = 0; y < n; y++)
        if (T[y]) ly[y] += delta;

    for (y = 0; y < n; y++)
        if (!T[y]) slack[y] -= delta;
}

void add_to_tree(int x, int prevx) {
    S[x] = true;
    prev[x] = prevx;
    for (int y = 0; y < n; y++)
        if (lx[x] + ly[y] - cost[x][y] < slack[y]) {
            slack[y] = lx[x] + ly[y] - cost[x][y];
            slackx[y] = x;
        }
}
```

```
void augment() {
    if (max_match == n) return;
    int x, y, root;
    int q[N], wr = 0, rd = 0;

    memset(S, false, sizeof(S));
    memset(T, false, sizeof(T));
    memset(prev, -1, sizeof(prev));
    for (x = 0; x < n; x++)
        if (xy[x] == -1) {
            q[wr++] = root = x;
            prev[x] = -2;
            S[x] = true;
            break;
        }

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

    while (1) {
        while (rd < wr) {
            x = q[rd++];
            for (y = 0; y < n; y++)
                if (cost[x][y] == lx[x] + ly[y] && !T[y]) {
                    if (yx[y] == -1) break;
                    T[y] = true;
                    q[wr++] = yx[y];

                    add_to_tree(yx[y], x);
                }
            if (y < n) break;
        }
        if (y < n) break;

        update_labels();
        wr = rd = 0;
        for (y = 0; y < n; y++)
            if (!T[y] &&  slack[y] == 0) {
                if (yx[y] == -1) {
                    x = slackx[y];
                    break;
                } else {
                    T[y] = true;
                    if (!S[yx[y]]) {
                        q[wr++] = yx[y];
                        add_to_tree(yx[y], slackx[y]);
                    }
                }
            }
        if (y < n) break;
    }

    if (y < n) {
        max_match++;
        for (int cx = x, cy = y, ty; cx != -2; cx = prev[cx], cy = ty) {
```

```
            ty = xy[cx];
            yx[cy] = cx;
            xy[cx] = cy;
        }
        augment();
    }
}

int hungarian() {
    int ret = 0;
    max_match = 0;
    memset(xy, -1, sizeof(xy));
    memset(yx, -1, sizeof(yx));

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

    augment();
    for (int x = 0; x < n; x++)
        ret += cost[x][xy[x]];
    return ret;
}
```

7.7 Componentes fortemente conexas

7.7.1 Algoritmo de Tarjan

Implementação do Shygypsy (Igor Naverniouk).

Listing 7.19: Algoritmo de Tarjan para Componentes Fortemente Conexas

```
#define NN 1024

// Inputs (populate these).
int deg[NN];
int adj[NN][NN];

// Union-Find.
int uf[NN];
int FIND( int x ) { return uf[x] == x ? x : uf[x] = FIND( uf[x] ); }
void UNION( int x, int y ) { uf[FIND( x )] = FIND( y ); }

// dfsn[u] is the DFS number of vertex u.
int dfsn[NN], dfsnext;

// mindfsn[u] is the smallest DFS number reachable from u.
int mindfsn[NN];

// The O(1)-membership stack containing the vertices of the current component.
int comp[NN], ncomp;
bool incomp[NN];

void dfs( int n, int u ) {
  dfsn[u] = mindfsn[u] = dfsnext++;
```

```
  incomp[comp[ncomp++] = u] = true;
  for( int i = 0, v; v = adj[u][i], i < deg[u]; i++ ) {
    if( !dfsn[v] ) dfs( n, v );
    if( incomp[v] ) mindfsn[u] <?= mindfsn[v];
  }

  if( dfsn[u] == mindfsn[u] ) {
    // u is the root of a connected component. Unify and forget it.
    do {
      UNION( u, comp[--ncomp] );
      incomp[comp[ncomp]] = false;
    } while( comp[ncomp] != u );
  }
}

void scc( int n ) {
  // Init union-find and DFS numbers.
  for( int i = 0; i < n; i++ ) dfsn[uf[i] = i] = ncomp = incomp[i] = 0;
  dfsnext = 1;

  for( int i = 0; i < n; i++ ) if( !dfsn[i] ) dfs( n, i );
}
```

7.8 2-SAT

Para cada cláusula $A \vee B$, adicione ao grafo duas arestas: $\bar{A} \rightarrow B$ e $\bar{B} \rightarrow A$. Aplique o algoritmo de componentes fortemente conexas. Caso todas as variáveis e suas negações sempre estejam em componentes conexas distintas, então é possível satisfazer a expressão. A função doisSat_insere assume que as variáveis são os vértices pares do grafo, e suas negações são os vértices ímpares $((A,0),(\bar{A},1),(B,2),(\bar{B},3),\ldots)$.

Dependências: Depende do algoritmo de Tarjan (componentes fortemente conexas).

Complexidade: $O(n)$, onde n é a quantidade de variáveis.

Entrada: **(int) n** o dobro da quantidade de variaveis.

Saída: **(bool)** se é possível satisfazer ou não.

Listing 7.20: 2-SAT

```
void doisSat_insere(int a, int b) {
    adj[a^1][deg[a^1]++] = b;
    adj[b^1][deg[b^1]++] = a;
}

// n eh o dobro da quantidade de variaveis
bool doisSat_ok(int n) {
    bool ok = true;
    scc(n);
    for (int i = 0; i < n && ok; i+= 2)
        ok = (FIND(i) != FIND(i+1));
    return ok;
}
```

Ver problema Cardapio da Sra. Montagny (spoj brasil). Ver problema resolvido *11659: Informants* na seção 17.13.

Capítulo 8

Programação Dinâmica

8.1 Maximum/Minimum Sum Subsequence

Algoritmo para achar a subsequência de soma máxima de um determinado vetor. Graças a sua inicialização em INT_MIN, também funciona para um vetor composto somente por números negativos. A ideia do algoritmo é, seja *sum* a variável que guarda a soma atual, então se ao somar o próximo elemento é obtida uma soma maior do que *maxSum*, atualiza *maxSum*. Caso ao somar seja obtido um valor abaixo de zero, então é melhor não somar aquele elemento e começar a soma a partir do próximo ($sum = 0$).

Complexidade: $O(n)$.

Entrada: (int s[]), que contém o endereço do vetor de elementos; **(int n)**, que contém a quantidade de elementos.

Saída: Um **(int)** com a quantidade da maior/menor soma.

Listing 8.1: Maximum Sum Subsequence

```
int MaximumSum(int s[], int n) {
    int i, max, maxSum;

    max = 0;
    maxSum = INT_MIN;
    for(i=0; i < n; i++) {
        max += s[i];
        if (max > maxSum) maxSum = max;
        if (max < 0) max = 0;
    }
    return maxSum;
}
```

E de forma análoga, se tem o Minimum Sum Subsequence.

Listing 8.2: Minimum Sum Subsequence

```
int MinimumSum(int s[], int n) {
    int i, minSum, min;

    min = 0;
    minSum = INT_MAX;
    for(i=0; i < n; i++) {
        min += s[i];
        if(min < minSum) minSum = min;
        if(min > 0) min = 0;
    }
    return minSum;
}
```

8.1.1 Maximum/Minimum Sum Subsequence em 2 dimensões

Para encontrar a maior a soma em uma área retangular de elementos de uma matriz, some uma linha com a outra, e a cada soma aplique o algoritmo para 1 dimensão (algoritmos da seção anterior). Também ver problema resolvido *108: Maximum Sum (Kadane)* na seção 17.2.

8.2 Maximum Product Subsequence

Algoritmo para achar a subsequência de produto máximo de um determinado vetor de elementos. **Cuidado com o fato de que se todos os elementos forem negativos, então este algoritmo não retorna o elemento mais próximo de zero**. A idéia do algoritmo é que sempre é vantajoso efetuar uma multiplicação (exceto quando a multiplicação é com o 0, então $product = 1$), então se houve uma quantidade ímpar de números negativos, o maior produto será aquele dado pela multiplicação que excluí o elemento negativo mais à esquerda ou o mais à direita. Sendo assim, o algoritmo é rodado de frente para trás e de trás para frente.

Complexidade: $O(n)$.

Suposição: O vetor não é composto somente de valores negativos.

Entrada: (int s[]), que contém o endereço do vetor de elementos; **(int n)**, que contém a quantidade de elementos.

Saída: Um **(int)** com a quantidade do maior produto.

Listing 8.3: Maximum Product Subsequence

```
int MaximumProduct(int s[], int n) {
    int i, product, maxProduct;

    product = 1;
    maxProduct = 0;
    for (i=0; i < n; i++) {
        product *= s[i];
        if (product == 0) product = 1;
        else if (product > maxProduct) maxProduct = product;
    }

    product = 1;
    for (i=n-1; i >= 0; i--) {
        product *= s[i];
        if (product == 0) product = 1;
        else if (product > maxProduct) maxProduct = product;
    }

    return maxProduct;
}
```

8.3 Longest Increasing Subsequence

8.3.1 Longest Increasing Subsequence (Descontínua)

Algoritmo para encontrar a maior sequência crescente de números, não necessariamente contínua (ou seja, pode haver saltos), num dado conjunto. A ideia do algoritmo é a cada iteração se perguntar se dos elementos anteriores existe algum menor do que o atual, e se houver, pega o que tiver maior tamanho de sequência e amplia em uma unidade por adicionar o atual elemento.

Complexidade: $O(n^2)$.

Entrada: (int) v[], que guarda o endereço do vetor que contém os elementos (note que ele é **global**); **(int) n**, que contém a quantidade de elementos no vetor (numerados de 0..n-1).

Saída: Nenhuma. Primeiro é impresso a cardinalidade da maior sequência e depois os seus elementos.

Listing 8.4: Longest Increasing Subsequence

```
int v[MAX];

void LIS(int n) {
    int a[MAX][2];      /* guarda a execucao do LIS */
    int b[MAX];         /* guarda os elementos da maior sequencia */
    int i, j, maior, auxiliar;

    for(i=0; i < MAX; i++) {
        a[i][0] = 0;    /* tamanho da sequencia */
        a[i][1] = -1;   /* antecedencia */
    }

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

    maior = 0;
    for(i=1; i < n; i++)
        if(a[i][0] > a[maior][0])
            maior = i;

    printf("Maior sequencia: %d\n", a[maior][0]);
    /* codigo para imprimir a sequencia */
    auxiliar = 1;
    b[0] = v[maior];
    while(a[maior][1] != maior) {
        maior = a[maior][1];
        b[auxiliar++] = v[maior];
    }
    for(i=auxiliar-1; i >= 0; i--)
        printf("%d\n", b[i]);
    /* fim da impressao da sequencia */
}
```

8.3.2 Longest Increasing Subsequence (Descontínua - nlogn)

Ver explicação do algoritmo no Udi Manber. Retirado do Algorithmist (`http://www.algorithmist.com/index.php/Longest_Increasing_Subsequence.cpp`).

Complexidade: $O(nlog(n))$.

Listing 8.5: Longest Increasing Subsequence O(nlog(n))

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

/* Finds longest strictly increasing subsequence. O(n log k) algorithm. */
void find_lis(vector<int> &a, vector<int> &b)
{
    vector<int> p(a.size());
    int u, v;

```

```
    if (a.empty()) return;

    b.push_back(0);

    for (size_t i = 1; i < a.size(); i++) {
        if (a[b.back()] < a[i]) {
            p[i] = b.back();
            b.push_back(i);
            continue;
        }

        for (u = 0, v = b.size()-1; u < v;) {
            int c = (u + v) / 2;
            if (a[b[c]] < a[i]) u=c+1; else v=c;
        }

        if (a[i] < a[b[u]]) {
            if (u > 0) p[i] = b[u-1];
            b[u] = i;
        }
    }

    for (u = b.size(), v = b.back(); u--; v = p[v]) b[u] = v;
}

/* Example of usage: */
#include <cstdio>
int main()
{
    int a[] = { 1, 9, 3, 8, 11, 4, 5, 6, 4, 19, 7, 1, 7 };
    vector<int> seq(a, a+sizeof(a)/sizeof(a[0]));
    vector<int> lis;
        find_lis(seq, lis);

    for (size_t i = 0; i < lis.size(); i++)
        printf("%d ", seq[lis[i]]);
        printf("\n");

    return 0;
}
```

8.4 Longest Common Subsequence (LCS)

Neste problema são dadas duas strings e se pergunta qual é a maior substring presente em ambas. Problema dividido em duas categorias: na primeira a substring deve ser contínua; e na segunda a substring não precisa ser contínua.

8.4.1 Longest Common Subsequence (Contínua)

Neste problema é pedido a subsequência na qual ela deve ser contínua, ou seja, sem saltos. Para entender veja a tabela 8.1, que a simulação a execução para a as strings `ABAB` e `BABA`.

Ou seja, a solução é dada por 8.1.

$$LCS[i][j] = \begin{cases} 1 & \text{se StringA[i] = StringB[j] e i = 0 ou j = 0,} \\ LCS[i-1][j-1]+1 & \text{se StringA[i] = StringB[j],} \\ 0 & \text{se StringA[i]} \neq \text{StringB[j].} \end{cases} \tag{8.1}$$

	A	B	A	B
B	0	1	0	1
A	1	0	2	0
B	0	2	0	3
A	1	0	3	0

Tabela 8.1: Simulação da execução do LCS em busca de uma substring contínua.

E o tamanho da maior subsequência é dada pelo maior índice presente na matriz, e para saber qual é essa subsequência basta percorrer na diagonal (das linhas de maior até as de menor valor).

Complexidade: $O(n^2)$.

Entrada: **(char *) s**, que guarda o endereço de uma string; **(char *) v**, que guarda o endereço de outra string.

Saída: Retorna um **(int)** que indica o tamanho da maior subsequência comum contínua.

Listing 8.6: Longest Common Subsequence (Contínuo)

```
int LCSCont(char *s, char *v) {
    char matriz[MAX][MAX];
    int i, j, maior;

    maior = 0;
    for(i=0; s[i]; i++) {
        for(j=0; v[j]; j++)
            if(s[i] != v[j]) {
                matriz[i][j] = 0;
            else {
                if(i == 0 || j == 0)
                    matriz[i][j] = 1;
                else
                    matriz[i][j] = matriz[i-1][j-1] + 1;
                if(matriz[i][j] > maior)
                    maior = matriz[i][j];
            }
        }

    return maior;
}
```

O código abaixo faz o mesmo que o provido acima, no entanto usa $2n$ de espaço, ao contrário do n^2 utilizado pelo anterior, no entanto se perde a possibilidade de se conhecer a subsequência.

Listing 8.7: Longest Common Subsequence (Contínuo) com uso de espaço reduzido

```
#include <string>

int LCSCont(const string& str1, const string& str2)
{
     if(str1.empty() || str2.empty())
     {
          return 0;
     }

     int *curr = new int [str2.size()];
     int *prev = new int [str2.size()];
     int *swap = NULL;
     int maxSubstr = 0;

     for(int i = 0; i<str1.size(); ++i)
```

```
    {
        for(int j = 0; j<str2.size(); ++j)
        {
            if(str1[i] != str2[j])
            {
                curr[j] = 0;
            }
            else
            {
                if(i == 0 || j == 0)
                {
                    curr[j] = 1;
                }
                else
                {
                    curr[j] = 1 + prev[j-1];
                }
                if(maxSubstr < curr[j])
                {
                    maxSubstr = curr[j];
                }
            }
        }
        swap=curr;
        curr=prev;
        prev=swap;
    }
    delete [] curr;
    delete [] prev;
    return maxSubstr;
}
```

8.4.2 Longest Common Subsequence (Descontínua)

Neste problema é pedido a subsequência na qual não é necessário que ela seja contínua, ou seja, pode ter saltos. Para entender veja a tabela 8.2, que a simulação a execução para a as strings `GAC` e `AGCAT`.

		A	G	C	A	T
	0	0	0	0	0	0
G	0	0	1	1	1	1
A	0	1	1	1	2	2
C	0	1	1	2	2	2

Tabela 8.2: Simulação da execução do LCS em busca da maior substring comum não necessariamente contínua.

Ou seja, a solução é dada por 8.2.

$$LCS[i][j] = \begin{cases} LCS[i-1][j-1]+1 & \text{, se StringA[i] = StringB[j],} \\ max(LCS[i][j-1], LCS[i-1][j]) & \text{, se StringA[i]} \neq \text{StringB[j].} \end{cases} \tag{8.2}$$

Complexidade: $O(n^2)$.
Observação: A linha e a coluna extra são para simplificação na programação, evitar casos de borda.
Entrada: **(char *) s**, que guarda o endereço de uma string; **(char *) v**, que guarda o endereço de outra string.
Saída: Retorna um **(int)** que indica o tamanho da maior subsequência comum (não necessariamente contínua).

Listing 8.8: Longest Common Subsequence (Descontínuo)

```c
int max(int d, int e) {
    return d > e ? d : e;
}

int pd[MAX][MAX];

int LCSDes(char *a, char *b) {
    int i, j, n, m;

    j = max(strlen(a), strlen(b));
    for(i=0; i <= j; i++) pd[i][0] = pd[0][i] = 0;

    n = strlen(a);
    if(a[n-1] == ’\n’) a[--n] = 0;
    m = strlen(b);
    if(b[m-1] == ’\n’) b[--m] = 0;

    for(i=1; i <= n; i++)
        for(j=1; j <= m ; j++)
            if(a[i-1] == b[j-1])
                pd[i][j] = pd[i-1][j-1] + 1;
            else
                pd[i][j] = max(pd[i-1][j], pd[i][j-1]);

    return pd[n][m];
}
```

E abaixo tem uma versão do LCS do Steve Halim (Methods to Solve), que tem uma função para recuperar a maior subesequência encontrada.

Listing 8.9: LCS do Steve Halim

```c
#include <stdio.h>
#include <string.h>

// change this constant if you want a longer subsequence
#define MAX 100

char X[MAX],Y[MAX];
int i,j,m,n,c[MAX][MAX],b[MAX][MAX];

int LCSlength() {
  m=strlen(X);
  n=strlen(Y);

  for (i=1;i<=m;i++) c[i][0]=0;
  for (j=0;j<=n;j++) c[0][j]=0;

  for (i=1;i<=m;i++)
    for (j=1;j<=n;j++) {
      if (X[i-1]==Y[j-1]) {
        c[i][j]=c[i-1][j-1]+1;
        b[i][j]=1; /* from north west */
      }
      else if (c[i-1][j]>=c[i][j-1]) {
        c[i][j]=c[i-1][j];
        b[i][j]=2; /* from north */
      }
      else {
```

```
        c[i][j]=c[i][j-1];
        b[i][j]=3; /* from west */
      }
    }

  return c[m][n];
}

void printLCS(int i,int j) {
  if (i==0 || j==0) return;

  if (b[i][j]==1) {
    printLCS(i-1,j-1);
    printf("%c",X[i-1]);
  }
  else if (b[i][j]==2)
    printLCS(i-1,j);
  else
    printLCS(i,j-1);
}

void main() {
  while (1) {
    gets(X);
    if (feof(stdin)) break; /* press ctrl+z to terminate */
    gets(Y);
    printf("LCS length -> %d\n",LCSlength()); /* count length */
    printLCS(m,n); /* reconstruct LCS */
    printf("\n");
  }
}
```

8.4.3 Problemas Relacionados

Dois outros problemas podem ser resolvidos com o uso do LCS. O primeiro deles é o tamanho da menor supersequência comum (SCS), que é dado por 8.3.

$$SCS(StringA, StringB) = lenght(StringA) + lenght(StringB) - LCS(StringA, StringB) \tag{8.3}$$

O segundo é o problema da Edit Distance (seção 8.5) quando somente as operações de inserção e remoção são permitidas, ou quando o custo da substituição é o dobro do custo da inserção ou remoção, a qual a solução é dada por 8.4.

$$ED(StringA, StringB) = lenght(StringA) + lenght(StringB) - 2 \times LCS(StringA, StringB) \tag{8.4}$$

8.5 Edit Distance

Neste problema é dada uma *string A* e deseja saber o custo para trasnforma-la na *string B*. As operações permitidas são de remoção, inserção, cópia e troca, cada qual com o seu custo associado, e o custo da transformação é a soma dos custos das operações. Seja $A(i)$ a sequência $a_1 a_2 \cdots a_i$, e seja $C(i, j)$ o custo para transformar $A(i)$ em $B(j)$, logo a resposta estará em $C(n,m)$. Sendo assim, suponha por indução que já achamos o melhor resultado para todas as outras posições e nos falta somente essa última ação (achar $C(n,m)$), então na solução ideal temos as seguintes possibilidades:

- *delete*: é removido a_n e então a resposta é dada por $C(n,m) = C(n-1,m) + Custo_{delete}$;
- *insert*: é inserido o b_m em A e então a resposta é dada por $C(n,m) = C(n,m-1) + Custo_{insert}$ (o que se está fazendo é transformar $A(n)$ em $B(m-1)$);

- *replace*: é trocado a_n por b_m e então a resposta é dada por $C(n,m) = C(n-1,m-1) + Custo_{replace}$;
- *copy*: caso a_n seja igual a b_m, então podemos simplesmente copia-lo, logo a resposta é $C(n,m) = C(n-1,m-1) + Custo_{copy}$.

Sendo assim a resolução deste problema é dada pela Equação 8.5.

$$C[i][j] = min \begin{cases} C[i-1][j-1] + copy & \text{se a[i] igual a b[j]} \\ C[i-1][j-1] + replace & \\ C[i-1][j] + delete & \\ C[i][j-1] + insert & \end{cases} \tag{8.5}$$

Complexidade: $O(nm)$, onde n é o tamanho da *string* origem (A) e m a destino (B).
Suposição: As matrizes **(int) c[][]** e **(int) s[][]** são globais.
Entrada: (char *) a, é a *string* a ser transformada; **(char *) b**, é a *string* que se deseja obter.
Saída: Um **(int)**, que é o custo da transformação. Os passos dessa transformação ficam na matriz **s**.

Listing 8.10: Edit Distance

```
/* copy, replace, insert, delete */
const int custos[] = {0,1,1,1};

int EditDistance(char *a, char *b) {
    int n, m, i, j;

    n = strlen(a);
    m = strlen(b);
    for (i=0; i <= n; i++) {
        c[i][0] = i*custos[3];
        s[i][0] = 'd';
    }
    for (j=0; j <= m; j++) {
        c[0][j] = j*custos[2];
        s[0][j] = 'i';
    }
    c[0][0] = 0;
    s[0][0] = 'e';

    for (i=1; i <= n; i++) {
        for (j=1; j <= m; j++) {
            c[i][j] = INF;
            if (a[i-1] == b[j-1]) {
                c[i][j] = c[i-1][j-1] + custos[0];
                s[i][j] = 'c';
            }
            if (c[i][j] > c[i-1][j-1] + custos[1]) {
                c[i][j] = c[i-1][j-1] + custos[1];
                s[i][j] = 'r';
            }
            if (c[i][j] > c[i][j-1] + custos[2]) {
                c[i][j] = c[i][j-1] + custos[2];
                s[i][j] = 'i';
            }
            if (c[i][j] > c[i-1][j] + custos[3]) {
                c[i][j] = c[i-1][j] + custos[3];
                s[i][j] = 'd';
            }
        }
    }
```

```

    return c[n][m];
}
```

Para saber as operações usadas na transformação, basta percorrer a matriz **s**, como mostra o Listing 8.11 (obviamente necessita a execução do *Edit Distance* antes).

Listing 8.11: Print Edit Distance (como no problema UVa 164)

```
int ndels, nadds;

void PrintEditDistance(int i, int j) {
    if (s[i][j] == 'e')
        return ;
    else {
        if (s[i][j] == 'c') {
            PrintEditDistance(i-1, j-1);
        }
        else if (s[i][j] == 'r') {
            PrintEditDistance(i-1, j-1);
            printf("C%c%02d", b[j-1], i+nadds-ndels);
        }
        else if (s[i][j] == 'd') {
            PrintEditDistance(i-1, j);
            printf("D%c%02d", a[i-1], i+nadds-ndels);
            ndels++;
        }
        else {
            PrintEditDistance(i, j-1);
            printf("I%c%02d", b[j-1], j);
            nadds++;
        }
    }
}
```

8.6 The Knapsack Problem (Mochila)

8.6.1 Múltiplos itens de cada tipo são permitidos

Dado n tipos de itens, cada qual com um tamanho s_i e valor v_i, além do tamanho C da mochila, colocar a maior quantidade de itens de forma a maximizar o valor contido na mochila (múltiplos itens de um mesmo tipo são permitidos). Seja $M[j]$ o máximo valor que uma mochila de tamanho j possa ter, então a cada novo tamanho o máximo valor será dado pela escolha de adicionar ou não adicionar um novo item (Equação 8.6).

$$M[j] = max\{M[j-1], max_{i=0 \ldots n-1}\{M[j-s_i]+v_i\}\} \tag{8.6}$$

Em $M[j-1]$ se tem a opção de não adicionar um novo item e continuar com o valor máximo atual; já na segunda parte da equação, em $max_i\{M[j-s_i]+v_i\}$, se tem a opção de adicionar um novo item de valor v_i, que força a ter a solução ideal para uma mochila de tamanho j menos o tamanho do objeto, dado por s_i (lembrando que i é um iterador entre todos os itens). Por fim, inicializar a mochila com zero, ou seja, $M[0] = 0$.

Complexidade: $O(nC)$, onde n é a quantidade de itens e C a capacidade da mochila.

Suposição: Múltiplos itens de um mesmo tipo são permitidos.

Entrada: **(int c)**, que contém o tamanho da mochila; **(int n)**, que contém a quantidade de tipos de itens; **(int) m[]**, que contém o endereço do vetor da mochila; **(int) v[]**, que contém o endereço do vetor de valores dos tipos de itens; **(int) s[]**, que contém o endereço do vetor de tamanho dos tipos de itens.

Saída: Nenhuma. Os máximos valores para cada tamanho de mochila ficam guardados no vetor **(int m[])**, que é passado como parâmetro.

Listing 8.12: Knapsack com múltiplos itens permitidos

```
void Knapsack(int c, int n, int m[], int v[], int s[]) {
    int i, j, max, maxIndice;

    for (i=0; i < n; i++)
        m[i] = 0;

    for (j=1; j <= c; j++) {
        max = m[j-1];
        maxIndice = -1;

        for (i=0; i < n; i++)
            if (s[i] <= j) {
                if (m[j-s[i]] + v[i] > max) {
                    max = m[j-s[i]] + v[i];
                    maxIndice = i;
                }
            }

        if (maxIndice == -1)
            m[j] = m[j-1];
        else
            m[j] = max;
    }
}
```

8.6.2 Knapsack 01 (Somente um item de cada tipo é permitido)

Dada uma mochila de capacidade C e n itens, cada qual com o seu valor v_i e espaço s_i, achar a solução que maximiza o valor para um tamanho j de mochila, com a restrição de que só existe um exemplar de cada item, ou seja, não há duplicação de elementos na mochila.

A solução é encontrada com o uso de uma matriz $M[i][j]$, onde $1 \leqslant i \leqslant n$ e $1 \leqslant j \leqslant C$. Assim, para cada tamanho j, a solução ideal é dada pela Equaçãp 8.7.

$$max\{M[i-1][j], M[i-1][j-s_i]+v_i\} \tag{8.7}$$

Onde no primeiro termo na Equação 8.7 se tem a possibilidade de não incluir o *i-ésimo* elemento, e na segunda parte a opção de incluí-lo. A matriz é necessária para não permitir a inclusão de elementos duplicados (graças ao acesso a linha anterior e não a atual), já que cada linha corresponde a um item diferente. Uma observação, é que com essa implementação a linha zero é pulada, para não haver erro na hora de acessar a linha anterior, e a coluna zero recebe zero em todas as linhas.

Complexidade: $O(nC)$, onde n é a quantidade de itens e C a capacidade da mochila.

Suposição: Múltiplos itens de um mesmo tipo não são permitidos.

Entrada: **(int) c**, que contém o tamanho da mochila; **(int) n**, que contém a quantidade de tipos de itens; **(int) m[]**, que contém o endereço do vetor da mochila; **(int) v[]**, que contém o endereço do vetor de valores dos tipos de itens; **(int) s[]**, que contém o endereço do vetor de tamanho dos tipos de itens.

Saída: Nenhuma. Os máximos valores para cada tamanho de mochila ficam guardados no vetor **(int m[])**, que é passado como parâmetro.

Listing 8.13: Knapsack sem duplicação de itens

```
void Knapsack01(int c, int n, int m[], int v[], int s[]) {
    int i, j, matriz[MAX_ITENS+1][MAX_TAM_MOCHILA+1];

    for (i=0; i < MAX_ITENS+1; i++)
        for (j=0; j < MAX_TAM_MOCHILA+1; j++)
```

```
            matriz[i][j] = 0;

    for (j=1; j <= n; j++) {
        for (i=1; i <= c; i++) {
            if (s[j-1] <= i && (matriz[j-1][i] < matriz[j-1][i - s[j-1]] + v[j-1]))
                matriz[j][i] = matriz[j-1][i - s[j-1]] + v[j-1];
            else
                matriz[j][i] = matriz[j-1][i];
        }
    }

    for (i=0; i <= c; i++)
        m[i] = matriz[n][i];

    return ;
}
```

Ver problema resolvido *10130: Supersale* na seção 17.7.

8.7 Coin Exchange ou Making Change

O problema é, dado uma quantidade n de moedas com diferentes valores (com pelo menos uma delas com o valor 1, para garantir que há solução em qualquer caso), qual a quantidade mínima de moedas necessária para formar um valor C. Algoritmo parecido com o da Mochila (seção 8.6). A idéia dele é que a solução ótima para um valor C é dado pela solução ótima de $C - v$ com a adição de uma moeda de valor v. Sendo assim, seja $M[j]$ um vetor que dá a quantidade mínima de moedas para um determinado valor j, a solução ideal é $M[j] = min_i\{M[j - v_i] + 1\}$, onde i é um iterador entre o vetor de valor das moedas.

Observação: Seja m seja o maior valor de moeda no problema. Então a estratégia gulosa sempre falha em algum caso de valor x tal que $x \leq 2m$ (prove por contradição). Logo, para saber se a estratégia gulosa é ótima basta testar todos os valores até $2m$ (geralmente pequeno).

Listing 8.14: Troco com a menor quantidade de moedas

```
#define MAX 20
#define MOEDAS  4

const int moedas[] = {1, 2, 7, 9};
int vetor[MAX];

void MinimoMoedas() {
    int i, j, min;

    for(i=0; i < MAX; i++)
        vetor[i] = 0;

    for(j=1; j < MAX; j++) {
        min = -1;
        for(i=0; i < MOEDAS; i++)
            if(j - moedas[i] >= 0)
                if(min > vetor[j-moedas[i]] + 1 || min == -1)
                    min = vetor[j-moedas[i]] + 1;
        vetor[j] = min;
    }
}
```

8.7.1 Quantidade de formas de dar o troco

Um problema parecido e recorrente, é não pedir a quantidade de moedas mínimas para um dado valor, mas de quantas formas é possível dar o troco. Seja $M[]$ um vetor que tem como quantidade de colunas os valores possíveis. Inicialize ele com zero, exceto na posição $M[0]$ que deve ter valor 1. Seja v o valor de uma moeda e j o valor desejado (valor da coluna), a quantidade de formas de dar o troco é dado por $M[j] = M[j] + M[j-v]$. A razão pela inicialização de $M[0] = 1$ é pela simplificação da idéia de que se a moeda for igual ao valor, então adicione 1; caso contrário, se a moeda tiver valor menor do que j, então adicione $M[j-v]$.

Listing 8.15: Quantidade de maneiras de se dar um troco

```
#define MAX            30001
#define QTDE_MOEDAS    11

long long int vetor[MAX];
const int moedas[] = {5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000};

void QuantidadeTroco() {
    int i, j;

    vetor[0] = 1;
    for (i=1; i < MAX; i++)
        vetor[i] = 0;

    for (i=0; i < QTDE_MOEDAS; i++)
        for (j=0; j < MAX; j+=5)
            if (j - moedas[i] >= 0)
                vetor[j] += vetor[j - moedas[i]];
}
```

8.8 Subset-Sum

O problema é, dado um conjunto de inteiros positivos, $S = \{x_1, x_2, \ldots, x_n\}$, existe algum subconjunto $T \subset S$, tal que a soma de todos os seus elementos é W? A solução é parecida com o algoritmo do *Coin Exchange*.

Seja uma matriz $M[n][W]$, onde a posição $M[0][0] = 1$ e toda a primeira linha, ou seja, $M[0][*] = 0$. Então para resolver o problema basta usar a fórmula 8.8.

$$M[i][j] = max\{M[i-1][j], M[i-1][j-x_i]\} \tag{8.8}$$

É possível reduzir para uma dimensão a matriz, bastando tomar o cuidado de percorrer a coluna do final para o início, para não estragar valores. Com isso se tem o código apresentado abaixo.

Complexidade: $O(nC)$, onde n é a quantidade de elementos no conjunto e C é a soma deles.

Suposição: MAX_VALOR é o valor da soma de todos itens.

Entrada: (int x[]), que contém o endereço do vetor de elementos; **(int n)**, que contém a quantidade de elementos; **(int) tamSoma**, que contém o tamanho desejado de uma soma;

Saída: Um **(int)**, que pode assumir o valor 1, caso seja possível somar os elementos de forma que se iguale ao **tamSoma**; e 0 caso contrário.

Listing 8.16: Subset Sum

```
int subsetSum(int x[], int n, int tamSoma) {
    int m[MAX_VALOR];
    int i, j;

    for(i=0; i < MAX_VALOR; i++)
        m[i] = 0;
    m[0] = 1;
```

```

    for(i=0; i < n; i++)
        for(j=w-1; j >= x[i]; j--)
            m[j] |= m[j - x[i]];

    return m[tamSoma];
}
```

8.8.1 K-Partition

8.9 Multiplicação em Cadeias de Matrizes (MCM)

Explicação depois (Lucas).

Complexidade: $O(n^3)$.

Suposição: Na ordem em que são dadas as matrizes é possível fazer a multiplicação.

Entrada: As dimensões únicas das matrizes, que ficam guardadas em **(int p[])**. Se a entrada for, por exemplo, $A_{10\times15} \times B_{15\times20} \times C_{20\times5}$, então no vetor **p** terá: $\{10, 15, 20, 5\}$.

Saída: Um **(int)** que contém o custo da melhor multiplicação. E na matriz **(int s[][])** fica guardado a ordem e a posição dos parênteses, ver o Listing 8.18.

Listing 8.17: MCM

```
int p[MAX];
int m[MAX][MAX];
int s[MAX][MAX];

int MCM(int n) {
    int i, j, l, k, q;

    for (i=1; i <= n; i++)
        m[i][i] = 0;

    for (l=2; l <= n; l++) {
        for (i=1; i <= n-l+1; i++) {
            j = i + l - 1;
            m[i][j] = INF;
            for (k=i; k <= j-1; k++) {
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (q < m[i][j]) {
                    m[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    }

    return m[1][n];
}
```

Listing 8.18: Imprime a sequência do MCM

```
/* incialmente passe (1, n) */
void PrintMCM(int i, int j) {
    if (i == j)
        printf("A%d", i);
    else {
```

```
        printf("(");
        PrintMCM(i, s[i][j]);
        printf(" x ");
        PrintMCM(s[i][j]+1, j);
        printf(")");
    }
}
```

8.10 ABBs ótimas

Algoritmo parecido com o MCM, mas essa implementação está com a otimização de Knuth (reduz para n^2, ver explicação no Cormen sobre isso). A otimização também vale para o MCM, Cutting Sticks, Bribe the Prisioners etc...

Complexidade: $O(n^2)$.

Listing 8.19: Árvore Binária de Busca Ótima

```
// UVA UID: 10304

#include <stdio.h>
#include <limits.h>
#define N 300

// O(n^2) - Ver descricao do problema...
// Codigo do forum do UVa - Knuth's theorem

int main()
{
    int n,x,y,r,z,obst[N][N],root[N][N],y1,z1;
    int f[N],w[N],min,a,b,c;
    while(scanf("%d",&n)!=EOF)
    {
        for(x=0;x<n;x++)
            scanf("%ld",&f[x]),obst[x][0]=0,root[x][0]=x,w[x]=f[x];
        if(n==1)
        {
            printf("0\n");
            continue;
        }
        for(x=1;x<n;x++)
            w[x]+=w[x-1];
        for(x=1;x<n;x++)
            for(y=0,z=x;z<n;y++,z++)
            {
                y1=root[y][z-1-y];
                z1=root[y+1][z-(y+1)];
                for(min=INT_MAX,r=y1;r<=z1;r++)
                {
                    if(r==0)
                        a=0;
                    else
                        a=obst[y][r-1-y];
                    if(r==z)
                        b=0;
                    else
                        b=obst[r+1][z-(r+1)];
                    if(y==0)
                        c=w[z];
```

```
                    else
                        c=w[z]-w[y-1];
                    if(a+b+c-f[r]<min)
                        min=a+b+c-f[r],root[y][z-y]=r;
                }
                obst[y][z-y]=min;
            }
        printf("%d\n",obst[0][n-1]);
    }
    return 0;
}
```

Capítulo 9

Teoria dos Números

9.1 Números Primos

Todos aqueles números que só podem ser divididos por 1 ou por ele mesmo, sendo que 1 não é um número primo. Números que não são primos são chamados de compostos.

9.1.1 Quantidade de Números Primos

Seja $pi(x)$ uma função que retorna a quantidade de números primos entre 0 e x, então a quantidade de números primos em potências de 10 é indicada na Tabela 9.1.

Potência de 10	Quantidade de números primos
pi(10^1)	4
pi(10^2)	24
pi(10^3)	168
pi(10^4)	1.229
pi(10^5)	9.592
pi(10^6)	78.498
pi(10^7)	664.579
pi(10^8)	5.761.455
pi(10^9)	50.847.534

Tabela 9.1: Quantidade de números primos até uma determinada potência.

Listing 9.1: Os 1.229 números primos até 10.000

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107
109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227
229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349
353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467
479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593 599 601 607 613
617 619 631 641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739 743 751
757 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883 887
907 911 919 929 937 941 947 953 967 971 977 983 991 997 1009 1013 1019 1021 1031 1033
1039 1049 1051 1061 1063 1069 1087 1091 1093 1097 1103 1109 1117 1123 1129 1151 1153
1163 1171 1181 1187 1193 1201 1213 1217 1223 1229 1231 1237 1249 1259 1277 1279 1283
1289 1291 1297 1301 1303 1307 1319 1321 1327 1361 1367 1373 1381 1399 1409 1423 1427
1429 1433 1439 1447 1451 1453 1459 1471 1481 1483 1487 1489 1493 1499 1511 1523 1531
1543 1549 1553 1559 1567 1571 1579 1583 1597 1601 1607 1609 1613 1619 1621 1627 1637
1657 1663 1667 1669 1693 1697 1699 1709 1721 17231733 1741 1747 1753 1759 1777 1783
1787 1789 1801 1811 1823 1831 1847 1861 1867 1871 1873 1877 1879 1889 1901 1907 1913
```

1931 1933 1949 1951 1973 1979 1987 1993 1997 1999 2003 2011 2017 2027 2029 2039 2053
2063 2069 2081 2083 2087 2089 2099 2111 2113 2129 2131 2137 2141 2143 2153 2161 2179
2203 2207 2213 2221 2237 2239 2243 2251 2267 2269 2273 2281 2287 2293 2297 2309 2311
2333 2339 2341 2347 2351 2357 2371 2377 2381 2383 2389 2393 2399 2411 2417 2423 2437
2441 2447 2459 2467 2473 2477 2503 2521 2531 2539 2543 2549 2551 2557 2579 2591 2593
2609 2617 2621 2633 2647 2657 2659 2663 2671 2677 2683 2687 2689 2693 2699 2707 2711
2713 2719 2729 2731 2741 2749 2753 2767 2777 2789 2791 2797 2801 2803 2819 2833 2837
2843 2851 2857 2861 2879 2887 2897 2903 2909 2917 2927 2939 2953 2957 2963 2969 2971
2999 3001 3011 3019 3023 3037 3041 3049 3061 3067 3079 3083 3089 3109 3119 3121 3137
3163 3167 3169 3181 3187 3191 3203 3209 3217 3221 3229 3251 3253 3257 3259 3271 3299
3301 3307 3313 3319 3323 3329 3331 3343 3347 3359 3361 3371 3373 3389 3391 3407 3413
3433 3449 3457 3461 3463 3467 3469 3491 3499 3511 3517 3527 3529 3533 3539 3541 3547
3557 3559 3571 3581 3583 3593 3607 3613 3617 3623 3631 3637 3643 3659 3671 3673 3677
3691 3697 3701 3709 3719 3727 3733 3739 3761 3767 3769 3779 3793 3797 3803 3821 3823
3833 3847 3851 3853 3863 3877 3881 3889 3907 3911 3917 3919 3923 3929 3931 3943 3947
3967 3989 4001 4003 4007 4013 4019 4021 4027 4049 4051 4057 4073 4079 4091 4093 4099
4111 4127 4129 4133 4139 4153 4157 4159 4177 4201 4211 4217 4219 4229 4231 4241 4243
4253 4259 4261 4271 4273 4283 4289 4297 4327 4337 4339 4349 4357 4363 4373 4391 4397
4409 4421 4423 4441 4447 4451 4457 4463 4481 4483 4493 4507 4513 4517 4519 4523 4547
4549 4561 4567 4583 4591 4597 4603 4621 4637 4639 4643 4649 4651 4657 4663 4673 4679
4691 4703 4721 4723 4729 4733 4751 4759 4783 4787 4789 4793 4799 4801 4813 4817 4831
4861 4871 4877 4889 4903 4909 4919 4931 4933 4937 4943 4951 4957 4967 4969 4973 4987
4993 4999 5003 5009 5011 5021 5023 5039 5051 5059 5077 5081 5087 5099 5101 5107 5113
5119 5147 5153 5167 5171 5179 5189 5197 5209 5227 5231 5233 5237 5261 5273 5279 5281
5297 5303 5309 5323 5333 5347 5351 5381 5387 5393 5399 5407 5413 5417 5419 5431 5437
5441 5443 5449 5471 5477 5479 5483 5501 5503 5507 5519 5521 5527 5531 5557 5563 5569
5573 5581 5591 5623 5639 5641 5647 5651 5653 5657 5659 5669 5683 5689 5693 5701 5711
5717 5737 5741 5743 5749 5779 5783 5791 5801 5807 5813 5821 5827 5839 5843 5849 5851
5857 5861 5867 5869 5879 5881 5897 5903 5923 5927 5939 5953 5981 5987 6007 6011 6029
6037 6043 6047 6053 6067 6073 6079 6089 6091 6101 6113 6121 6131 6133 6143 6151 6163
6173 6197 6199 6203 6211 6217 6221 6229 6247 6257 6263 6269 6271 6277 6287 6299 6301
6311 6317 6323 6329 6337 6343 6353 6359 6361 6367 6373 6379 6389 6397 6421 6427 6449
6451 6469 6473 6481 6491 6521 6529 6547 6551 6553 6563 6569 6571 6577 6581 6599 6607
6619 6637 6653 6659 6661 6673 6679 6689 6691 6701 6703 6709 6719 6733 6737 6761 6763
6779 6781 6791 6793 6803 6823 6827 6829 6833 6841 6857 6863 6869 6871 6883 6899 6907
6911 6917 6947 6949 6959 6961 6967 6971 6977 6983 6991 6997 7001 7013 7019 7027 7039
7043 7057 7069 7079 7103 7109 7121 7127 7129 7151 7159 7177 7187 7193 7207 7211 7213
7219 7229 7237 7243 7247 7253 7283 7297 7307 7309 7321 7331 7333 7349 7351 7369 7393
7411 7417 7433 7451 7457 7459 7477 7481 7487 7489 7499 7507 7517 7523 7529 7537 7541
7547 7549 7559 7561 7573 7577 7583 7589 7591 7603 7607 7621 7639 7643 7649 7669 7673
7681 7687 7691 7699 7703 7717 7723 7727 7741 7753 7757 7759 7789 7793 7817 7823 7829
7841 7853 7867 7873 7877 7879 7883 7901 7907 7919 7927 7933 7937 7949 7951 7963 7993
8009 8011 8017 8039 8053 8059 8069 8081 8087 8089 8093 8101 8111 8117 8123 8147 8161
8167 8171 8179 8191 8209 8219 8221 8231 8233 8237 8243 8263 8269 8273 8287 8291 8293
8297 8311 8317 8329 8353 8363 8369 8377 8387 8389 8419 8423 8429 8431 8443 8447 8461
8467 8501 8513 8521 8527 8537 8539 8543 8563 8573 8581 8597 8599 8609 8623 8627 8629
8641 8647 8663 8669 8677 8681 8689 8693 8699 8707 8713 8719 8731 8737 8741 8747 8753
8761 8779 8783 8803 8807 8819 8821 8831 8837 8839 8849 8861 8863 8867 8887 8893 8923
8929 8933 8941 8951 8963 8969 8971 8999 9001 9007 9011 9013 9029 9041 9043 9049 9059
9067 9091 9103 9109 9127 9133 9137 9151 9157 9161 9173 9181 9187 9199 9203 9209 9221
9227 9239 9241 9257 9277 9281 9283 9293 9311 9319 9323 9337 9341 9343 9349 9371 9377
9391 9397 9403 9413 9419 9421 9431 9433 9437 9439 9461 9463 9467 9473 9479 9491 9497
9511 9521 9533 9539 9547 9551 9587 9601 9613 9619 9623 9629 9631 9643 9649 9661 9677
9679 9689 9697 9719 9721 9733 9739 9743 9749 9767 9769 9781 9787 9791 9803 9811 9817
9829 9833 9839 9851 9857 9859 9871 9883 9887 9901 9907 9923 9929 9931 9941 9949 9967
9973

9.1.2 Teste de Primalidade

Verifica se um dado número **n** é primo ou não. Lembre-se que nenhum número terminado em 0, 2, 4, 5, 6, ou 8 pode ser primo (exceto dois).

Complexidade: $O(\sqrt{n})$.

Entrada: um número **(int) n**.

Saída: 1 caso seja um número primo, 0 caso contrário.

Listing 9.2: Teste de Primalidade

```
int VerificaPrimo(int n) {
    if(n == 2)
        return 1;
    if(n <= 1 || n % 2 == 0)
        return 0;

    int i, raiz;
    raiz = sqrt(n) + 1;
    for(i = 3; i <= raiz; i+=2)
        if(n % i == 0)
            return 0;

    return 1;
}
```

9.1.3 Crivo de Eratóstenes

Pré-calcula todos os números primos até n (inclusivo).

Complexidade: $O(nlog(n))$.

Entrada: **(int) n** que indica até qual número devo pré-calcular os números primos.

Saída: o vetor **(int) ehprimo[]** recebe 0 ou 1 em suas posições. Um 1 em ehprimo[j] significa que **j** é primo, um 0 significa que não é primo.

Listing 9.3: Crivo de Eratóstenes

```
int ehprimo[MAXN+1];

void AchaPrimos(int n) {
    int i, j;

    ehprimo[0] = ehprimo[1] = 0;
    ehprimo[2] = 1;
    for(i=3; i<=n; i++)
        ehprimo[i] = i%2;

    for(i=3; i*i<=n; i+=2)
        if(ehprimo[i])
            for(j=i*i; j<=n; j+=i)
                ehprimo[j] = 0;
}
```

Modificando o crivo acima se obtém o código do Listing 9.4, que é bem mais rápido, mas se perde em legibilidade.

Listing 9.4: Crivo de Eratóstenes Rápido

```
#define MAXSIEVE 100000000 /* Todos os primos ate esse numero*/
#define MAXSIEVEHALF (MAXSIEVE/2)
#define MAXSQRT 5000 /*sqrt(MAXSIEVE)/2 Tem que fazer as contas baseado no MAXSIEVE*/
```

```
char a[MAXSIEVE/16+2];
#define isprime(n) (a[(n)>>4]&(1<<(((n)>>1)&7))) /* So funciona se o n for impar */

void init() {
    int i,j;
    memset(a,255,sizeof(a));
    a[0]=0xFE;
    for(i=1;i<MAXSQRT;i++) {
        if (a[i>>3]&(1<<(i&7))) {
            for(j=i+i+i+1;j<MAXSIEVEHALF;j+=i+i+1) {
                a[j>>3]&=~(1<<(j&7));
            }
        }
    }
}
```

Crivo para um determinado trecho

Para se calcular de modo rápido os números primos de qualquer trecho, use o código do Listing 9.5.

Complexidade: $O(nlog(n))$.

Entrada: **(int) a** e **(int) a**, que é o trecho de atuação do crivo ($a \leq b$).

Saída: Nenhuma, os números primos ficam marcados com 1 no vetor global **ehprimo**.

Listing 9.5: Crivo de Eratóstenes para um determinado Trecho

```
char ehprimo[MAX];

void CrivoRange(unsigned int a, unsigned int b) {
    unsigned int i, j, range, raiz;

    range = b-a+1;
    /* inicializacao de pares e impares */
    for (i=0; i < range; i++) ehprimo[i] = (i+a)%2;
    raiz = sqrt((double)b) + 1;
    for (i=3; i <= raiz; i+=2) {
        if (i > a && !ehprimo[i-a])
            continue;
        /* primerio numero divisivel por i mas diferente de i */
        j = (a/i)*i;
        if (j < a) j +=i;
        if (j == i) j +=i; /* se j for primo, comeca do proximo */

        j -= a; /* mudanca de indice */
        for (; j < range; j+=i) ehprimo[j] = 0;
    }

    /* trata do 1 e 2 */
    if (a <= 1) ehprimo[1-a] = 0;
    if (a <= 2) ehprimo[2-a] = 1;

    return ;
}
```

9.1.4 Fatoração em Primos

Segundo o Teorema Fundamental da Aritmética, todo o número inteiro pode ser decomposto de forma única como um produto de números primos.

Complexidade: $O(\sqrt{n})$.
Entrada: (long) x, que é o número a ser fatorado.
Saída: Imprime todos os números primos que o fatoram.

Listing 9.6: Fatoração em Números Primos

```
void FatoracaoPrimo(long long int x) {
    long long int i, c, raiz;

    c = x;
    while(c % 2  == 0) {
        printf("2\n");
        c /= 2;
    }

    i = 3;
    raiz = sqrt(c) + 2;
    while(i < raiz && c > 1) {
        while(c % i == 0) {
            printf("%lld\n", i);
            c /= i;
        }
        i += 2;
        raiz = sqrt(c) + 2;
    }

    if(c > 1)
        printf("%lld\n", c);
}
```

9.1.5 Rápida multiplicidade de fatores primos em $n!$ (método de Adrien-Marie Legendre)

A motivação para encontrar de um modo rápido a multiplicidade de todos os fatores primos de um dado $n!$ se encontra quando se deseja saber se um dado k o divide. Este problema é atacado por decompor o número k em seus fatores primos, e então saber se todos eles se cancelam com os fatores primos de $n!$. Então, se for decompor o $n!$ do modo tradicional (ver seção 9.1.4), levaria $O(n\sqrt{n})$, o que para um n muito grande se torna proibitivo.

Seja $n = 38$, e suponha que se queria saber a multiplicidade do 3 em 38!, então vejamos que $38! = 2\times 3\times 4\times 5\times 6\times 7\times 8\times 9\times 10\times 11\times 12\times 13\times 14\times 15\times 16\times 17\times 18\times 19\times 20\times 21\times 22\times 23\times 24\times 25\times 26\times 27\times 28\times 29\times 30\times 31\times 32\times 33\times 34\times 35\times 36\times 37\times 38$. Agora pegue somente os múltiplos de 3, já que os outros não terão o 3 em sua composição, sendo assim nos resta 12 números dos 38: $3\times 6\times 9\times 12\times 15\times 18\times 21\times 24\times 27\times 30\times 33\times 36$. Ponha um 3 de cada um deles em evidência, então temos: $3^{12}\times 1\times 2\times 3\times 4\times 5\times 6\times 7\times 8\times 9\times 10\times 11\times 12$. Agora mais uma vez retire desses 12 números todos aqueles não são múltiplos de 3, e coloque mais uma vez em evidência um 3 de cada um: $3^{12}\times 3^4\times 1\times 2\times 3\times 4$. Por fim, repita mais uma vez este processo, e terá como resposta que 38! possuí o número primo 3 com multiplicidade 17, pois $3^{12}\times 3^4\times 3^1 = 3^{17}$. Deste processo percebe-se a fórmula geral da Equação 9.1, onde n é o fatorial e p o número primo que se pede a multiplicidade [1].

$$\sum_{i=1}^{n\leqslant p^i} \lfloor \frac{n}{p^i} \rfloor \tag{9.1}$$

9.2 Divisores e Divisibilidade

Pelo Teorema Fundamental da Artimética, todos os números podem ser escritos como uma multiplicação única de fatores primos. Sendo assim, dado um número n, ele pode ser escrito da seguinte forma:

[1] http://homepage.smc.edu/kennedy_john/NFACT.PDF

$$n = p_1^{e_1} p_2^{e_2} p_3^{e_3} p_4^{e_4} p_5^{e_5} \dots p_i^{e_i} \tag{9.2}$$

Onde p_i é um número primo e e_i é a sua multiplicidade. Sendo assim, a quantidade de divisores de um número é dado por $(e_1+1)(e_2+1)\dots(e_i+1)$.

9.2.1 Critérios de Divisibilidade

A seguir critérios para saber se um determinado número é divisível por algum número entre 1 e 12.

Divisibilidade por 1: Todos são.

Divisibilidade por 2: Todos os números terminados em 0, 2, 4, 6 e 8.

Divisibilidade por 3: Todos aqueles em que a soma de seus algarismos é divisível por 3. Exemplo, $546 = 5+4+6 = 15 = 3*5$, logo é divisível por 3 ($3*182 = 546$).

Divisibilidade por 4: Todos aqueles em que o último algarismo é par e a soma da metade do último algarismo com o penúltimo é par. Exemplo, $138947189436 = 3+(6/2) = 6$, logo é divisível por 4.

Divisibilidade por 5: Todos aqueles em que o último algarismo é 5 ou 0.

Divisibilidade por 6: Todos aqueles em que o último algarismo é par e a soma de seus algarismos é múltiplo de 3.

Divisibilidade por 7: Todos aqueles em que a diferença do dobro do último algarismo com o número formado pelos demais é um múltiplo de 7. Exemplo, $245 = 24-10 = 14 = 2*7$, logo é divisível por 7.

Divisibilidade por 8: Todos aqueles em que os 3 últimos algarismos dá um número divisível por 8 (incluindo o caso de terminação com 3 zeros). Exemplo, $10840 = 840 = 8*105$, logo é divisível por 8.

Divisibilidade por 9: Todos aqueles em que a soma de seus algarismos é divisível por 9. Exemplo, $581472 = 5+8+1+4+7+2 = 27 = 3*9$, logo é divisível por 9.

Divisibilidade por 10: Todos aqueles em que o último algarismo é 0.

Divisibilidade por 11: Todos aqueles em que a soma dos algarismos nas posições ímpares menos os da posições pares for um número divisível por 11. Exemplo, $20482 = 2-0+4-8+2 = 0 = 11*0$, logo é divisível por 11.

Divisibilidade por 12: Todos aqueles que são divisíveis por 3 e por 4.

9.3 Aritmética Modular

9.3.1 Adição Modular

Dados dois números, onde se deseja saber a soma deles **mod m**, é possível aplicar a equação 9.3 para evitar overflow.

$$(A+B)\%M = ((A\%M)+(B\%M))\%M \tag{9.3}$$

Exemplo: Quantos centavos eu tenho se receber 123,45 da minha mãe e 94,67 do meu pai? $(12345\%100)+(9467\%100) = (45+67)\%100 = 12$.

9.3.2 Subtração Modular

Considere uma adição com números negativos, ver seção 9.3.1.

Exemplo: Considerando o exemplo anterior, quantos centavos eu terei caso gaste 52,53? (12%100)–(53%100) $= -41\%100 = 59\%100$.

9.3.3 Multiplicação Modular e Big Mod

Dados dois números, onde se deseja saber o produto deles **mod m**, é possível aplicar a equação 9.4 para evitar overflow.

$$(A*B)\%M = ((A\%M)*(B\%M))\%M \tag{9.4}$$

Então seja **b** a base de um número e **p** o seu expoente. Se desejo saber (b^p) **mod m**, basta aplicar a Equação 9.4 para obter o algoritmo do Listing 9.7.

Listing 9.7: Big Mod

```
long long BigMod(long long b, long long p, long long m) {
    if(p == 0)
        return 1;
    if(p == 1)
        return b;

    if(p % 2 == 0)
        return ((long long)pow((BigMod(b, p / 2, m) % m), 2) % m);
    else
        return ((BigMod(b, p - 1, m) % m) * (b % m)) % m;
}
```

9.3.4 Teoremas de Fermat sobre congruência

Seguem abaixo alguns teoremas que podem agilizar cálculos, principalmente na exponenciação modular.

Teorema de Fermat: Seja p um número primo e a um número inteiro. Então

$$a^p \cong a(mod p) \tag{9.5}$$

Lema: Seja p um número primo e a e b inteiros. Então,

$$(a+b)^p \cong a^p + b^p(mod p) \tag{9.6}$$

Teorema de Fermat II: Seja p um número primo e a um número inteiro que não é divisível por p. Então,

$$a^{p-1} \cong 1(mod p) \tag{9.7}$$

9.3.5 Função $\phi(n)$ de Euler

Dado um número inteiro n, a função $\phi(n)$ de Euler retorna a quantidade de elementos entre $0,\ldots,a,\ldots,n-1$ que tem $mdc(a,n) = 1$. Em aritmética modular, isso quer dizer que em $\mathbb{Z}_n$, o elemento a tem inverso, e seu inverso é o coeficiente dado pelo algoritmo euclidiano estendido (ver seção 9.4.2).

O valor da função $\phi(n)$ pode ser obitdo pela fatoração em primos de n, em conjunto com as regras abaixo:

- Se n for primo, então retorne $n-1$.
- Se $n = p^e$, para algum primo p de multiplicidade e maior que 1, então retorne $p^e - p^{e-1}$.
- Se $n = p_i^e \times p_j^f$, para quaisquer primos p_i e p_j, então retorne $\phi(p_i^e) \times \phi(p_j^f)$, usando as regras acima.

Uma outra forma de calcular o $\phi(n)$ é pelo produtório de Euler, definido na Equação 9.8, onde p_i são os fatores primos distintos de n (é desta equação que o código foi implementado).

$$\phi(n) = n\prod_{p|n}(1-\frac{1}{p}) = n(1-\frac{1}{p_1})(1-\frac{1}{p_2})\cdots(1-\frac{1}{p_m}) = n(\frac{p_1-1}{p_1})(\frac{p_2-1}{p_2})\cdots(\frac{p_m-1}{p_m}) \tag{9.8}$$

Complexidade: $O(n)$.
Suposição: Foi executado anteriormente o Crivo de Eratóstenes (ver seção 9.1.3).
Entrada: **(int) n**, um inteiro positivo.
Saída: um **(int)** que é a quantidade de números primos relativos à n.

Listing 9.8: Função $\phi(n)$ de Euler

```
int phi(int n) {
    int res = n;

```

```
    if(ehprimo[n])
        return n-1;
    for(int p=2; 2*p<=n; p++)
        if(ehprimo[p] && n%p==0)
            res = res/p*(p-1);
    return res;
}
```

9.3.6 Resolvedor de Equação Modular Linear

Retorna o menor valor positivo x tal que $ax = b(mod)n$, onde a, b e n são valores conhecidos.

Requerimento: Exige a implementação do Máximo Divisor Comum Estendido.

Entrada: (int) a, **(int) b**, **(int) n**, como consta na descrição.

Saída: Um **int**, que é a solução da equação, ou -1 se não houver solução.

Listing 9.9: Função $\phi(n)$ de Euler

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

int solve(int a, int b, int n) {
    int d, x, y;

    a = mod(a, n);
    b = mod(b, n);
    d = mdc(a, n, &x, &y);
    if(b%d==0)
        return mod( ((long long)x%(n/d)) * ((b/d)%(n/d)) , n/d );
    else
        return -1;
}
```

9.4 MDC e MMC

9.4.1 Máximo Divisor Comum (MDC)

Encontra o maior divisor comum entre dois números pelo algoritmo euclidiano.

Complexidade: $O(log(a) + log(b))$.

Entrada: (int) a e **(int) b**, que são os inteiros (positivos ou negativos) dos quais se quer saber o *mdc*.

Saída: um **(int)** que é o maior inteiro que divide os números de entrada.

Listing 9.10: Máximo Divisor Comum

```
int mdc(int a, int b) {
    if(a<0) a = -a;
    if(b<0) b = -b;

    if(b == 0)
        return a;
    else
        return mdc(b, a%b);
}
```

No entanto a versão mais rápida de cálculo do MDC é dado pelo Listing 9.11.

Listing 9.11: Máximo Divisor Comum Rápido

```
int mdc(int a, int b) {
    while (b > 0) {
        a = a % b;
        a ^= b;
        b ^= a;
        a ^= b;
    }
    return a;
}
```

9.4.2 Máximo Divisor Comum Estendido

Dados dois números positivos (a,b), o algoritmo euclidiano estendido encontra x e y tais que $ax+by = mdc(a,b)$.

Complexidade: $O(log(a)+log(b))$.

Entrada: (int) a e **(int) b**, que são dois inteiros positivos. E endereços para dois **(int)**, **x** e **y**.

Saída: um **(int)** que é o $mdc(a,b)$, além do valor nos endereços de x e y para os quais $ax+by = mdc(a,b)$.

Listing 9.12: Máximo Divisor Comum Estendido

```
int mdc(int a, int b, int *x, int *y) {
    int xx, yy, d;
    if(b == 0) {
        *x = 1;
        *y = 0;
        return a;
    }
    d = mdc(b, a%b, &xx, &yy);
    *x = yy;
    *y = xx - a/b*yy;
    return d;
}
```

9.4.3 Mínimo Múltiplo Comum (MMC)

O Mínimo Múltiplo Comum pode ser obtido a partir do MDC através da relação da Equação 9.9.

$$mmc(a,b) = \frac{ab}{mdc(a,b)} \tag{9.9}$$

9.5 Números de Carmichael

Segundo o Primeiro Teorema de Fermat, se n for primo então $a^n mod(n) = a$ (ver seção 9.3.4). No entanto, existem alguns números n compostos que também apresentam esse comportamento para qualquer a entre $2 \leqslant a < n$. Sendo assim, não se pode usar esse Teorema para provar que um dado número n é primo, mas se um número passa por esse teste (para vários/todos valores de a), então ele tem uma alta probabilidade de ser primo. Números que passam por esse teste e não são primos, são chamados de **Carmichael Numbers**.

Esses números tem a propriedade de serem formados por pelo menos 3 fatores primos. Além disso, números dados dados pela seguinte forma $(6k+1)(12k+1)(18k+1)$ serão números de Carmichael se cada um desses termos for primo (para $k = 1$, $7*13*19 = 1729$). Para o código, ver o problema resolvido *10006 Carmichael Numbers* (seção 17.5).

Listing 9.13: Os primeiros números de Carmichael até 50 milhões

```
561 1105 1729 2465 2821 6601 8911 10585 15841 29341 41041 46657 52633 62745 63973
75361 101101 115921 126217 162401 172081 188461 252601 278545 294409 314821 334153
```

```
340561 399001 410041 449065 488881 512461 530881 552721 656601 658801 670033 748657
825265 838201 852841 997633 1024651 1033669 1050985 1082809 1152271 1193221 1461241
1569457 1615681 1773289 1857241 1909001 2100901 2113921 2433601 2455921 2508013
2531845 2628073 2704801 3057601 3146221 3224065 3581761 3664585 3828001 4335241
4463641 4767841 4903921 4909177 5031181 5049001 5148001 5310721 5444489 5481451
5632705 5968873 6049681 6054985 6189121 6313681 6733693 6840001 6868261 7207201
7519441 7995169 8134561 8341201 8355841 8719309 8719921 8830801 8927101 9439201
9494101 9582145 9585541 9613297 9890881 10024561 10267951 10402561 10606681 10837321
10877581 11119105 11205601 11921001 11972017 12261061 12262321 12490201 12945745
13187665 13696033 13992265 14469841 14676481 14913991 15247621 15403285 15829633
15888313 16046641 16778881 17098369 1723680117316001 17586361 17812081 18162001
18307381 18900973 19384289 19683001 20964961 21584305 22665505 23382529 25603201
26280073 26474581 26719701 26921089 26932081 27062101 27336673 27402481 28787185
29020321 29111881 31146661 31405501 31692805 32914441 33302401 33596641 34196401
34657141 34901461 35571601 35703361 36121345 36765901 37167361 37280881 37354465
37964809 38151361 38624041 38637361 39353665 40160737 40280065 40430401 40622401
40917241 41298985 41341321 41471521 42490801 43286881 43331401 43584481 43620409
44238481 45318561 45877861 45890209 46483633 47006785 48321001 48628801 49333201
```

9.6 Fibonacci

Através do uso de propriedades de matrizes, um determinado termo n na seqüência de Fibonacci pode ser calulado em $O(logn)$. Para o cálculo, basta lembrar que a matriz em 9.10 gera os termos da seqüência.

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \tag{9.10}$$

Com isto, e com a idéia de exponenciação em $O(logn)$, se tem o código do Listing 9.14.

Listing 9.14: Cálculo do Fibonacci em $O(logn)$

```
int Fibonacci(int n) {
    int i, j, h, k, t;
    i = h = 1;
    j = k = 0;

    while(n > 0) {
        if(n % 2 == 1) {
            t = j*h;
            j = i*h + j*k + t;
            i = i*k + t;
        }
        t = h*h;
        h = 2*k*h + t;
        k = k*k + t;
        n = n/2;
    }

    return j;
}
```

9.7 Símbolos de Lagrange

9.8 Progressões

9.8.1 Progressão Aritmética

Uma Progressão Aritmética (PA) é definida por:

$$a_n = a_1 + (n-1) \cdot r, \text{ com n} \geq 1$$

Na qual a soma de seus termos é dada por:

$$S = \frac{(a_1 + a_n) \cdot n}{2}$$

E soma dos termos entre a_p e a_q é dada por:

$$S_{(p,q)} = \frac{(q-p+1) \cdot (a_p + a_q)}{2}$$

9.8.2 Progressão Geométrica

Uma Progressão Geométrica (PG) é definida por:

$$a_n = a_1 \cdot q^{n-1}$$

A partir do termo a_m, o termo a_n pode ser obtido por:

$$a_n = a_m \cdot q^{n-m}$$

Para $|q| \geq 1$ temos a soma dos primeiros n termos sendo

$$S_n = \sum_{i=1}^{n} a_i = a_1 + a_2 + \ldots + a_n = \frac{a_1(q^n - 1)}{q-1}$$

Já para $|q| < 1$ a somatória converge, e é chamada "Série Geométrica"

$$S_\infty = \sum_{n=0}^{\infty} a_1 \cdot q^n = \frac{a_1}{1-q}$$

O produto dos termos de uma PG, a partir do primeiro, é dado por

$$P_n = a_1^n \cdot q^{\frac{n \cdot (n-1)}{2}}$$

Capítulo 10

Criptografia

10.1 Floyd's Cycle Finding Algorithm

Acha ciclos sem gasto de memória (tradeoff por tempo de execução).

Listing 10.1: Floyd's Cycle Finding (Tortoise and Hare algorithm)

```
#include <cstdio>

#define a0 1

inline int f(int x) {
    return (x*2) % 1000000000;
}

int main() {
    int a, b, i;
    a = a0; b = a0; i = 0;
    do {
        a = f(a);
        b = f(f(b));
        i++;
    } while (a != b);

    printf("multiplo do ciclo: %d, elemento rep: %d\n", i, a);

    b = a0; i = 0;
    while (b != a) {
        b = f(b);
        a = f(a);
        i++;
    }

    printf("inicio do ciclo: %d\n", i);

    i = 1;
    a = f(a);
    while (a != b) {
        a = f(a);
        i++;
    }
    printf("tamanho do ciclo: %d, valor rep: %d\n", i, a);
    return 0;
}
```

10.2 Baby-step Giant-step

Ver problema resolvido na seção 17.14.

Capítulo 11

Probabilidade

11.1 Triângulo de Pascal

O triângulo de Pascal é um triângulo numérico infinito formado por binômios de Newton ($C_{n,k}$), onde n representa o número da linha (posição vertical) e k representa o número da coluna (posição horizontal).

```
              1
            1   1
          1   2   1
        1   3   3   1
      1   4   6   4   1
    1   5  10  10   5   1
  1   6  15  20  15   6   1
1   7  21  35  35  21   7   1
```

Figura 11.1: Triângulo de Pascal.

Tal triângulo pode ser gerado com o código do Listing 11.1.

Listing 11.1: Triangulo de Pascal

```
#include <cstdio>
#define MAX 100

int main() {

    unsigned long long int p[MAX+1][MAX+1];

    for(int i=0; i<MAX+1; i++){
        for(int j=0; j<=i; j++){
            if(j == 0 || j == i) p[i][j] = 1;
            else p[i][j] = p[i-1][j] + p[i-1][j-1];
        }
    }

    printf("C(67, 33) = %llu\n", p[67][33]);

}
```

11.1.1 Propriedades

- **Simetria**: O triângulo de Pascal apresenta simetria em relação à altura, o que permite a relação da Equação 11.1.

$$C_{n,k} = \binom{n}{k} = \binom{n}{n-k} = C_{n,n-k} \tag{11.1}$$

- **Relação de Stiffel**: Cada número do triângulo de Pascal é igual à soma do número imediatamente acima e seu antecessor (ver Equação 11.2).

$$\binom{n-1}{k-1} + \binom{n-1}{k} = \binom{n}{k} \tag{11.2}$$

- **Soma de linha**: A soma de uma linha no triângulo de Pascal é igual 2^n, onde n é a altura, como ilustrado em 11.3.

$$\begin{array}{|c|ccccc|c|}
n & & & & & & 2^n \\
0 & 1 & & & & & 1 \\
1 & 1 & 1 & & & & 2 \\
2 & 1 & 2 & 1 & & & 4 \\
3 & 1 & 3 & 3 & 1 & & 8 \\
4 & 1 & 4 & 6 & 4 & 1 & 16
\end{array} \tag{11.3}$$

- **Soma de coluna**: A soma da coluna, no triângulo de Pascal, pode ser calculada pela relação da Equação 11.4.

$$\binom{n}{n} + \binom{n+1}{n} + \ldots + \binom{n+k}{n} = \binom{n+k+1}{n+1} \tag{11.4}$$

Essa relação pode ser explicada pela ilustração em 11.5, onde a soma dos 3 primeiros elementos da coluna 1 é 6.

$$\begin{array}{|c|ccccc}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 1 & & & & \\
1 & 1 & \underline{1} & & & \\
2 & 1 & \underline{2} & 1 & & \\
3 & 1 & \underline{3} & 3 & 1 & \\
4 & 1 & 4 & \underline{6} & 4 & 1
\end{array} \tag{11.5}$$

11.2 Binômio de Newton

O coeficiente binomial, ou também conhecido como número binomial, de um número n, na classe k, consiste no número de combinações de n termos, k a k (a quantidade de maneiras de pegar k de n elementos sem se importar com a ordem da escolha). O coeficiente binomial é dado pela Equação 11.6.

$$C_{n,k} = \binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)(n-2)(n-3)\ldots(n-k+1)}{k!} \tag{11.6}$$

Como fatorial são números que logo se tornam grande demais para efetuar contas sem o uso de *BigNums*, calcular $C_{n,k}$ por partes (a cada iteração incluir um numerador e um denominador) e utilizar o máximo divisor comum é uma excelente maneira de calcula-lo sabendo-se que o resultado cabe em um *unsigned long int*. Tal estratégia é mostrada no código do Listing 11.2.

Listing 11.2: Cálculo do Binômio de Newton com uso de MDC

```
unsigned long int mdc(unsigned long int a,unsigned long int b) {
    if (a % b == 0) return b;
    else return mdc(b, a % b);
}
```

```

void DivPorMDC(unsigned long int *a,unsigned long int *b) {
    unsigned long int g = mdc(*a,*b);
    *a/=g;
    *b/=g;
}

unsigned long int Comb(unsigned long int n, unsigned long int k) {
    unsigned long int denominador, numerador, multiplicar, dividir, i;

    if (k > n/2)
        k = n - k;

    numerador = denominador = 1;
    for (i=k; i; i--) {
        multiplicar = n-k+i;
        dividir = i;
        DivPorMDC(&multiplicar, &dividir);
        DivPorMDC(&numerador, &dividir);
        DivPorMDC(&multiplicar, &denominador);
        numerador *= multiplicar;
        denominador *= dividir;
    }

    return numerador/denominador;
}
```

11.3 Catalan Numbers

Os números de Catalan, designados por C_n, são dados pela Fórmula 11.7.

$$C_n = \frac{\binom{2n}{n}}{n+1} = \frac{\binom{2n+1}{n}}{2n+1} = \frac{\binom{2n}{n-1}}{n} = \binom{2n}{n} - \binom{2n}{n-1} = \frac{(2n)!}{(n+1)!n!} \tag{11.7}$$

Os primeiros 30 números da sequência de Catalan podem ser vistos na Tabela 11.1.

n:	C_n:	n:	C_n:	n:	C_n:
1	1	11	58786	21	24466267020
2	2	12	208012	22	91482563640
3	5	13	742900	23	343059613650
4	14	14	2674440	24	1289904147324
5	42	15	9694845	25	4861946401452
6	132	16	35357670	26	18367353072152
7	429	17	129644790	27	69533550916004
8	1430	18	477638700	28	263747951750360
9	4862	19	1767263190	29	1002242216651368
10	16796	20	6564120420	30	3814986502092304

Tabela 11.1: Os primeiros 30 termos da seqüência de Catalan.

Os números de Catalan podem ser gerados de modo mais fácil por uso de memorização com a Fórmula 11.8 ($C_0 = 1$).

$$C_{n+1} = \frac{2(2n+1)}{n+2} C_n \tag{11.8}$$

11.3.1 Aplicações em Combinatória

- C_n determina o número de diferentes árvores binárias que podem ser construídas com exatamente n elementos.

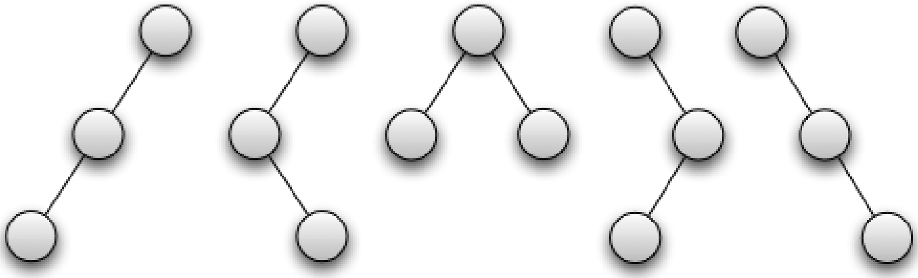

Figura 11.2: As 5 distintas árvores binárias que podem ser formadas com 3 elementos.

- C_n é o número de **caminhos monotonicos** entre arestas de uma grade com $n \times n$ celas o qual não passa a diagonal. Um caminho monotonico é um o qual começa no canto inferior esquerdo, termina no canto superior direito e é constituído somente por arestas que apontam ou para cima ou para o lado direito. Para $n = 4$, todas as possibilidades são mostradas na Figura 11.3.

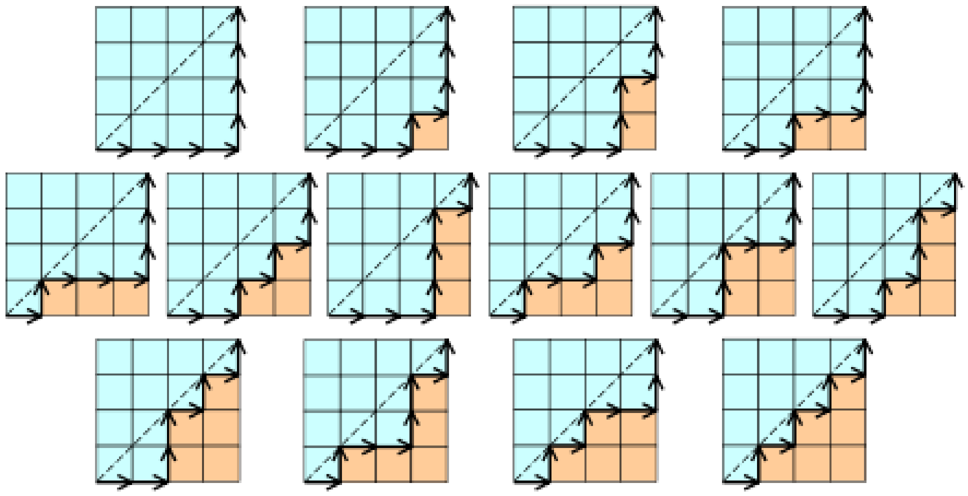

Figura 11.3: Todos os caminhos monotonicos possíveis para $n = 4$.

- C_n é o número de diferentes maneiras de compor uma figura em forma de escada de altura n composta por n retângulos. A Figura 11.4 ilustra o problema para $n = 4$.

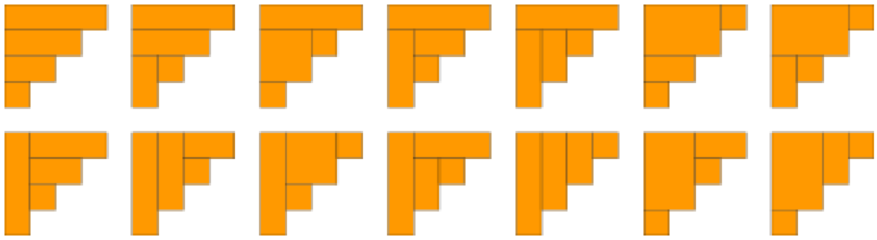

Figura 11.4: As possíveis de maneiras de se obter uma figura parecida com uma escada de altura 4 com 4 retângulos.

- C_n é o número de **Dyck words** de tamanho $2n$. Uma palavra Dyck é uma string composta por Xs e Ys de forma que nenhum segmento inicial tem mais Ys do que Xs. Por exemplo, essas são algumas palavras Dyck válidas de tamanho 6: XXXYYY, XYXXYY, XYXYXY, XXYYXY e XXYXYY.
- Interpretando X como o abrir de um parênteses e Y como o fechar, então C_n conta o número de expressões corretas contendo n-pares de parênteses. Por exemplo, ((())), ()(()), ()()(), (())() e (()()).
- C_n é o número de diferente maneiras que $n+1$ fatores podem ser completamente parentizados (ou em outras palavras, a quantidade de maneiras de associar n operações). Por exemplo, para $n = 3$, se tem 4 fatores (a, b, c, d) e 3 operações: $((ab)c)d, (a(bc))d, (ab)(cd), a((bc)d)$ e $a(b(cd))$.
- C_n é o número de diferente maneiras que um polígono convexo com $n+2$ lados pode ser cortado em triângulos por conectar os vértices com linhas retas. Para $n = 4$, todas as possibilidades são mostradas na Figura 11.5.

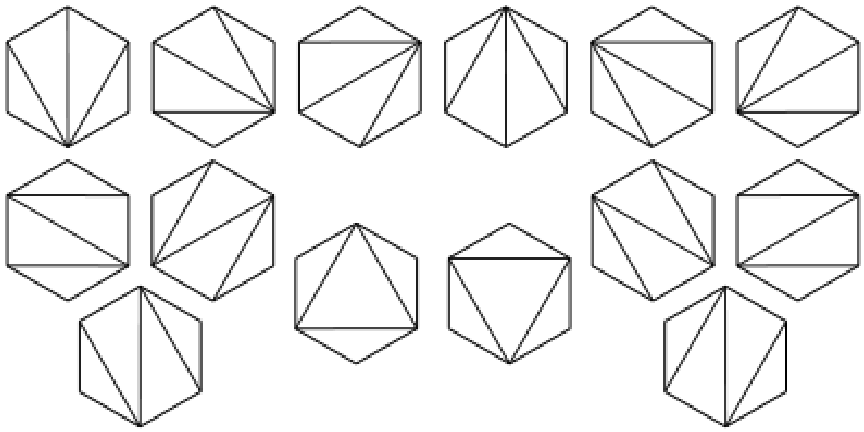

Figura 11.5: Todos os caminhos monotonicos possíveis para $n = 4$.

11.4 Stirling Numbers

Os números de Stirling de primeira ordem $s(n,k)$ contam o número de permutações possíveis de n ítens em k ciclos. Eles pode ser definidos como $s(n,k) = (-1)^{n-k}c(n,k)$, podendo ser calculados com o código do triângulo des Pascal, colocando-se uma verificação $((n-k)\%2! = 0)$ para inverter o sinal.

Já os números de Stirling de segunda classe contam o número de maneiras de se particionar um conjunto com n elementos em k subconjuntos disjuntos e não-vazios. Podem ser obtidos através da relação de recorrência $s(n,k) = s(n-1,k-1) - (n-1)s(n-1,k)$.

O código monta o triângulo de Stirling, evitando a implementação da relação de recorrência de forma recursiva - mais simples, porém muito lenta. É considerado no código que a $s(n,k)$ se encontra na posição $[n,k]$ da matriz, portanto as posições $[0,k], k \geq 0$ possuem apenas lixo da memória.

Listing 11.3: Stirling Numbers

```
#include <cstdio>
#include <cmath>
#define MAX 100

int main(){
    int a, b;
    unsigned long long int s[MAX+1][MAX+1];

    //Triangulo de Stirling
    //No. de Stirling 2a. Esp. {n,m};
    s[1][1] = 1;
    for(int i=2; i<MAX+1; i++){
        for(int j=1; j<=i; j++){
            if(j == 1 || j == i) s[i][j] = 1;
            else s[i][j] = s[i-1][j]*j + s[i-1][j-1];
        }
    }
    while(scanf("%d %d ", &a, &b)!=EOF)
        printf("{%d,%d} = %llu\n",a,b,s[a][b]);
}
```

Capítulo 12

Geometria Computacional

Nunca compare se dois doubles são iguais diretamente! Faça a verificação por `abs(A-B) < EPS`, onde `EPS = 1e-9` (precisão pode variar de acordo com o problema, mas em geral 10^{-9} é bom o suficiente).

12.1 Ponto 2D

Implementação de várias operações com sobrecarga de operadores.

Listing 12.1: Comparação de ponto flutuante

```
int cmp(long double a, long double b = 0.0) {
    if (fabs(a-b) < 1e-10) return 0;
    return (a<b)?-1:1;
}
```

Listing 12.2: Ponto 2D

```
typedef long double ct;

struct pt {
    ct x, y;
    pt() {}
    pt(ct x, ct y): x(x), y(y) {}
    ct operator%(const pt& a) { return x*a.y-y*a.x; }
    pt operator-(const pt &a) const { return pt(x-a.x, y-a.y); }
    bool operator<(const pt &a) const {
        return (cmp(x, a.x) < 0) || (cmp(x, a.x) == 0 && cmp(y, a.y) < 0); }
    ct operator*(const pt& a) { return x*a.x+y*a.y; }
};

int area(pt a, pt b, pt c) { return cmp((b-a)%(c-a)); }
```

12.2 Ponto 3D

Listing 12.3: Ponto 3D

```
typedef ct long double;

struct pt {
    ct x, y, z;
```

```
    pt() {}
    pt(ct x, ct y, ct z): x(x), y(y), z(z) {}
    pt operator-(const pt &a) { return pt(x-a.x, y-a.y, z-a.z); }
    ct operator*(const pt &a) { return x*a.x + y*a.y +z*a.z; }
    pt operator%(const pt &a) { return pt(y*a.z-a.y*z, z*a.x-x*a.z, x*a.y-a.x*y); }
    ct mod2() { return x*x + y*y + z*z; }
};
```

12.3 Distância de ponto a reta

Calcula a distância do ponto o à reta pq.
Requer: struct pt

Listing 12.4: Distância de ponto a reta

```
double dist = sqrt(((q-p)%(o-p)).mod2())/(sqrt((q-p)*(q-p)));
```

12.4 Produto Escalar

Calcula o produto escalar entre dois pontos $(pt_A - pt_O).(pt_B - pt_O)$.
Complexidade: $O(1)$.
Entrada: Os pontos origem, A e B.
Saída: Um **(long long int)**, que é o valor do produto escalar.

Listing 12.5: Produto Escalar entre dois pontos

```
long long prodesc(int pto[2], int pta[2], int ptb[2]) {
    long long v1 = ((long long) pta[0]-pto[0])*(ptb[0]-pto[0]);
    long long v2 = ((long long) pta[1]-pto[1])*(ptb[1]-pto[1]);

    return v1+v2;
}
```

12.5 Produto Vetorial

Calcula o produto vetorial entre dois pontos $(pt_A - pt_O)X(pt_B - pt_O)$.
Complexidade: $O(1)$.
Entrada: Os pontos origem, A e B.
Saída: Um **(long long int**, que é o valor do produto vetorial.

Listing 12.6: Produto Vetorial

```
long long prodvet(int pto[2], int pta[2], int ptb[2]) {
    long long v1 = ((long long) pta[0]-pto[0])*(ptb[1]-pto[1]);
    long long v2 = ((long long) pta[1]-pto[1])*(ptb[0]-pto[0]);

    return v1-v2;
}
```

E o código do Listing 12.7 calcula de forma segura o sinal do produto vetorial, de forma a evitar *overflow*.

Listing 12.7: Produto Vetorial (Sinal)

```
int prodvetsn(int pto[2], int pta[2], int ptb[2]) {
    long long v1 = ((long long) pta[0]-pto[0])*(ptb[1]-pto[1]);
```

```
    long long v2 = ((long long) pta[1]-pto[1])*(ptb[0]-pto[0]);

    if( v1 < v2 )  return -1;
    else if( v1 > v2 ) return +1;
    else return 0;
}
```

12.6 Teste de Pertinência de Ponto em Segmento

Testa se um ponto pertence a um segmento.
Complexidade: $O(1)$.
Observação: Necessita a implementação do produto escalar e vetorial.
Entrada: Um ponto **p** e o segmento de reta definido pelos pontos **a** e **b**.
Saída: Um **bool**, que indica se pertence ou não.

Listing 12.8: Teste de Pertinência de Ponto em Segmento

```
bool interPtSeg(int p[2], int a[2], int b[2]) {
    return prodvet(p, a, b)==0 && prodesc(a, p, b)>=0 && prodesc(b, p, a)>=0;
}
```

12.7 Teste de Pertinência de Ponto em Polígono

Testa se um ponto está inserido em um polígono (não conta na borda do mesmo).
Complexidade: $O(1)$.
Observação: Precisa de uma estrutra simples de ponto (apenas inteiros x e y).
Entrada: Um conjunto de vértices, o número de vértices nesse conjunto e o ponto em questão.
Saída: Um **bool**, que indica se pertence ou não.

Listing 12.9: Teste de Pertinência de Ponto em Polígono

```
int ptInPol(pt vert[], int n, pt pto) {
  int i, j, c = 0;
  for (i = 0, j = n-1; i < n; j = i++) {
    if ( ((vert[i].y>pto.y) != (vert[j].y>pto.y)) &&
         (pto.x < (vert[j].x-vert[i].x) * (pto.y-vert[i].y) / (vert[j].y-vert[i].y) +
             vert[i].x) )
        c = !c;
  }
  return c;
}
```

12.8 Teste de Interseção de Segmentos

Testa se dois segmentos possuem interseção não vazia.
Complexidade: $O(1)$.
Observação: Necessita a implementação do produto escalar, vetorial e das funções *max* e *min*.
Entrada: Os segmentos (**a**, **b**) e (**c**, **d**).
Saída: Um **bool**, que indica se os segmentos se intersectam ou não.

Listing 12.10: Teste de Interseção de Segmentos

```
bool interSegSeg(int a[2], int b[2], int c[2], int d[2]) {
    int i, r1, r2;
    for(i=0; i<2; i++)
        if( MIN(a[i],b[i]) > MAX(c[i],d[i]) || MAX(a[i],b[i]) < MIN(c[i],d[i]) )
            return 0;

    r1 = prodvetsn(a, c, b) * prodvetsn(a, d, b);
    r2 = prodvetsn(c, a, d) * prodvetsn(c, b, d);
    return r1<=0 && r2<=0;
}
```

12.9 Convex hull (Graham Scan)

Dados vários pontos em um plano, o problema do casco convexo é encontrar os pontos que estão na borda do poligno que engloba todos os pontos.

Complexidade: $O(nlog(n))$.

Entrada: **(pt []) v** vetor de pontos iniciais; **(int) n** quantidade de pontos do vetor v.

Saída:(pt []) h vetor de pontos do fecho convexo; **(int) top** quantidade de pontos no fecho.

Listing 12.11: Graham Scan Convex Hull

```
pt h[1010];
int top, k;
pt pivo;

bool cmp_ang(const pt &a, const pt &b) {
    int v = cmp((a-pivo)%(b-pivo));
    return v > 0 || v == 0 && cmp((a-pivo)*(a-pivo), (b-pivo)*(b-pivo)) < 0;
}

pivo = *min_element(v, v+n);
sort(v, v+n, cmp_ang);
for (k = n-2; k >= 0 && area(v[0], v[n-1], v[k]) == 0; k--);
reverse(v+k+1, v+n);
top = 0;
for (int i = 0; i < n; i++) {
    while (top > 1 && area(h[top-2], h[top-1], v[i]) <= 0) top--;
    h[top++] = v[i];
}
if (top > 1 && area(h[top-2], h[top-1], v[0]) == 0) top--;
```

Ver problema resolvido *10173: Smallest Bounding Rectangle* na seção 17.8.

12.10 Monotone Chain Convex Hull

Complexidade: $O(nlogn)$.

Listing 12.12: Monotone Chain Convex Hull

```
// Implementation of Monotone Chain Convex Hull algorithm.
#include <algorithm>
#include <vector>
using namespace std;

```

```
typedef long long CoordType;

struct Point {
    CoordType x, y;

    bool operator <(const Point &p) const {
        return x < p.x || (x == p.x && y < p.y);
    }
};

// 2D cross product.
// Return a positive value, if OAB makes a counter-clockwise turn,
// negative for clockwise turn, and zero if the points are collinear.
CoordType cross(const Point &O, const Point &A, const Point &B)
{
    return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

// Returns a list of points on the convex hull in counter-clockwise order.
// Note: the last point in the returned list is the same as the first one.
vector<Point> convexHull(vector<Point> P)
{
    int n = P.size(), k = 0;
    vector<Point> H(2*n);

    // Sort points lexicographically
    sort(P.begin(), P.end());

    // Build lower hull
    for (int i = 0; i < n; i++) {
        while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
        H[k++] = P[i];
    }

    // Build upper hull
    for (int i = n-2, t = k+1; i >= 0; i--) {
        while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
        H[k++] = P[i];
    }

    H.resize(k);
    return H;
}
```

12.11 Reta

Representação canônica de reta. Simplifica vários códigos.

Requer: struct pt

Listing 12.13: Reta canônica

```
struct line {
    double a, b, c; // ax+by+c = 0
    line() {}
    line(double a, double b, double c): a(a), b(b), c(c) { fix(); }
    line(pt p, pt q) {
        pt A = p-q;
```

```
        a = A.y; b = -A.x; c = p%q;
        fix();
    }
    void fix() {
        if (cmp(a) == 0) {
            c /= b; b /= b;
        } else {
            b /= a; c /= a; a /= a;
        }
        if ( cmp(a) < 0 ||
            !cmp(a) && cmp(b) < 0) {
            a *= -1; b *= -1; c *= -1;
        }
    }

    line perp(pt p) { return line(-b, a, b*p.x-a*p.y); }
};

pt inter(line r, line s) {
    pt A = pt(r.a, s.a), B = pt(r.b, s.b), C = pt(r.c, s.c);
    return pt((B%C)/(A%B), (A%C)/(B%A));
}
```

12.12 Círculo

Requer: struct line, struct pt

Listing 12.14: Circulo

```
struct circ {
    pt c; double r;
    circ() {}
    circ(pt c, double r): c(c),r(r) {}
};

circ find_circ(pt a, pt b, pt c) {
    line r(a, b), s(b, c);
    pt x = inter(r.perp(pt((a+b).x/2, (a+b).y/2)), s.perp(pt((b+c).x/2, (b+c).y/2)));
    return circ(x, sqrt((x-a)*(x-a)));
}
```

12.13 Par de pontos mais próximos

Complexidade: $O(nlogn)$

Listing 12.15: Closest Pair

```
#define px second
#define py first

typedef pair<long long, long long> pairll;
int n;
pairll pnts [100000];
set<pairll> box;
double best;
```

```

int compx(pairll a, pairll b) { return a.px<b.px; }

int main () {
    scanf("%d", &n);
    for (int i = 0; i < n; ++i) scanf("%lld %lld", &pnts[i].px, &pnts[i].py);

    sort(pnts, pnts+n, compx);
    best = 1500000000; // INF
    box.insert(pnts[0]);
    int left = 0;
    for (int i = 1; i < n; ++i) {
        while (left < i && pnts[i].px-pnts[left].px > best) box.erase(pnts[left++]);
        for (typeof(box.begin()) it = box.lower_bound(make_pair(pnts[i].py-best, pnts[i].px-best));
                                  it != box.end() && pnts[i].py+best >= it->py; it++)
        best = min(best, sqrt(pow(pnts[i].py - it->py, 2.0)+pow(pnts[i].px - it->px, 2.0)));
        box.insert(pnts[i]);
    }
    printf("%.2f\n", best);
    return 0;
}
```

Capítulo 13

Geometria e Trigonometria

13.1 Teoria

Baseado no material de: http://activities.tjhsst.edu/sct/lectures/2009-10/comp_geo.pdf.

13.1.1 Vetor

Denotações de um vetor: $\mathbf{v} = \vec{v} =< v_x, v_y, v_z >= v_x\mathbf{i} + v_y\mathbf{j} + v_z\mathbf{k}$, onde i, j, k são vetores unitários na direção dos seus respectivos eixos. Também é possível representar um vetor em termos de magnitude e ângulo. A magnitude (ou norma), representada por $v = |\mathbf{v}| = \|v\|$, é dada por $\sqrt{x^2 + y^2 + z^2}$ (teorema de Pitágoras). A norma também pode ser generalizada pra n dimensões: $\|v\| = \sqrt{v \cdot v} = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$.

Há dois modos de multiplicar um vetor, a primeira delas é a multiplicação escalar (ou produto escalar), que é a seguinte:

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z = uv \cos\theta \tag{13.1}$$

O ângulo θ é o ângulo entre os dois vetores ($\theta \leq \pi$). Isto significa que o produto de dois vetores só será 0 se eles forem perpendiculares ou se um deles for o vetor nulo. A seguir algumas propriedades do produto escalar:

- $\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{u}$ (comutativa).
- $\mathbf{u} \cdot (\mathbf{v} + \mathbf{w}) = \mathbf{u} \cdot \mathbf{v} + \mathbf{u} \cdot \mathbf{w}$ (distributiva em relação a soma de vetores).
- $(n_1\mathbf{u}) \cdot (n_2\mathbf{v}) = (n_1 n_2)(\mathbf{u} \cdot \mathbf{v})$.

O segundo modo de multiplicar um vetor é pelo produto vetorial, que resulta em um vetor que é perpendicular a ambos os vetores originais, e que tem como magnitude a área do quadrilátero com lados adjacentes a **a** e **b**, ou $ab \sin\theta$.

$$a \times b = \hat{n}|a||b| \sin\theta$$
$$\|a \times b\|^2 = \|a\|^2\|b\|^2 - (a \cdot b)^2$$
$$\|a \times b\| = \|a\|\|b\| \sin\theta$$

O significa dessas equações pode ser visualizado na Figura 13.1 (onde $\hat{n}$ é o vetor unitário perpendicular a tanto **a** quanto **b**).

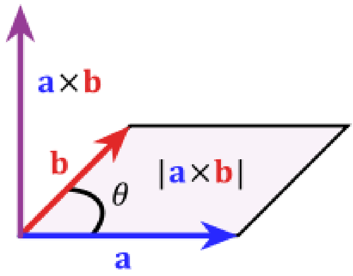

Figura 13.1: Significado do produto vetorial.

Algumas propriedades do produto vetorial:

- $a \times b = -b \times b$ (anticomutativo).
- $a \times (b+c) = a \times b + a \times c$ (distributiva em relação a adição).
- $(ra) \times b = a \times (rb) = r(a \times b)$ (compatível com a multiplicação escalar).
- $a \times (b \times c) + b \times (c \times a) + c \times (a \times b) = 0$ (**não** é associativo, mas satisfaz a identidade de Jacobi).

13.2 Resultados geométricos a partir de produtos

13.2.1 Área de um triângulo

A área de um triângulo pode ser calculada por $\frac{1}{2}\|a \times b\|$ (ou seja, escolha um ponto qualquer e crie dois vetores aos outros 2 pontos). De forma alternativa, se você tiver os comprimentos dos lados (a,b,c), é possível calcular a área pela fórmula de Heron: $K = \sqrt{s(s-a)(s-b)(s-c)}$, onde $s = \frac{a+b+c}{2}$.

13.2.2 Verificação de paralelismo de duas retas

A partir dos dois vetores que representam as retas, faça o produto vetorial deles, e então se a magnitude for 0, elas são paralelas.

13.2.3 Distância de um ponto a reta

A distância de um ponto P a reta AB é:

$$d(P,AB) = \frac{|(P-A) \times (B-A)|}{|B-A|}$$

13.2.4 Distância de um ponto a um segmento de reta

Verifique se o triângulo $\triangle PAB$ é obtuso; se for, pegue o mínimo de $d(A)$ ou $d(B)$; de outro modo, use a fórmula da distância a reta.

13.2.5 Verificação se um ponto está na reta (ou segmento de reta)

Um ponto está na reta se a distância do ponto a reta é 0.

13.2.6 Verificação se dois pontos estão do mesmo lado de uma reta

Para saber se dois pontos C e D estão no mesmo lado da reta AB, calcule a componente z de $(B-A) \times (C-A)$, e z‘ de $(B-A) \times (D-A)$. Se $zz' > 0$, então eles estão do mesmo lado.

13.2.7 Verificação de se um ponto está contido num triângulo

Pegue a média dos pontos que formam o triângulo (pois sabemos que esse ponto está dentro do triângulo), então verifique se esse ponto da média e o que desejamos saber está do mesmo lado da reta para os 3 lados. **Também serve para polígonos convexos**.

13.2.8 Verificação para saber se 4 ou mais pontos são co-planares

Escolha 3 pontos quaisquer (A, B e C). Então se para algum ponto D, $(B-A)(C-A)(D-A) = 0$, então todos eles pertencem ao mesmo plano.

13.2.9 Intersecção de retas

Em 2D duas retas se intersectam se elas não são paralelas. Em 3D duas retas se intersectam se elas não são paralelas e os quatro pontos que as definem são co-planares.

13.2.10 Intersecção de segmentos de retas

Em 2D dois segmentos de retas, AB e CD, intersectam, se e somente se, A e B estão em lados opostos da reta CD, e C e D estão em lados opostos de AB.

13.2.11 Verificação de convexidade de um polígono 2D

Percorra os vértices desse polígono no sentido horário, e a cada 3 vértices A, B, C, calcule o produto vetorial $(B-A) \times (C-A)$. Se a componente z de todos os produtos for positiva, então o polígono é convexo.

13.2.12 Verificação de se um ponto está em um polígono não convexo

A partir do seu ponto trace uma reta para fora do polígono, com o cuidado dessa reta não intersectar um vértice ou ao longo de todo um lado. Então esse ponto estará dentro do polígono se ele cruzar um número ímpar de lados (**também serve para polígonos convexos**).

13.3 Relação de Ângulos

13.3.1 Identidades Trigonométricas

Identidades trigonométricas fundamentais:

$$\sin\theta = \frac{\text{Cateto Oposto}}{\text{Hipotenusa}}$$

$$\cos\theta = \frac{\text{Cateto Adjacente}}{\text{Hipotenusa}}$$

$$\tan\theta = \frac{\sin\theta}{\cos\theta}$$

$$\sec\theta = \frac{1}{\cos\theta}$$

$$\cot\theta = \frac{\cos\theta}{\sin\theta}$$

$$\csc\theta = \frac{1}{\sin\theta}$$

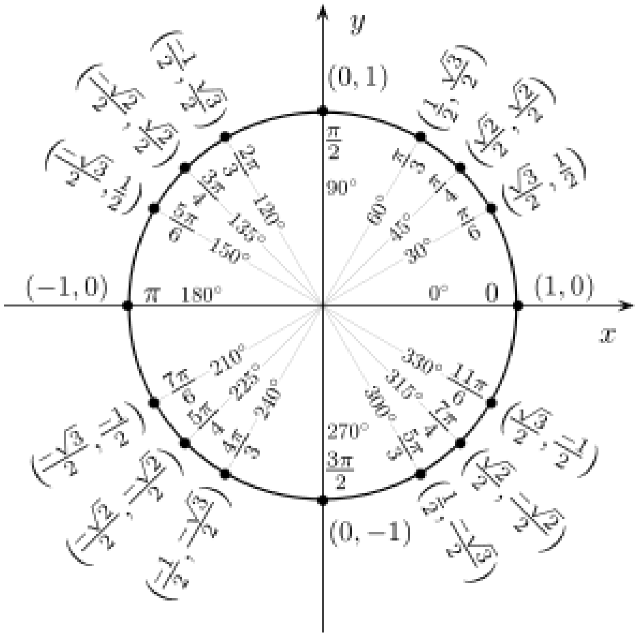

Figura 13.2: Ângulos com seus valores em sin, cos e radianos.

Identidade de Pitágoras

A identidade trigonométrica de Pitágoras establece a seguinte relação entre sin e cos: $\sin^2\theta + \cos^2\theta = 1$. Outra identidade importante é a $1 + \tan^2\theta = \sec^2\theta$ e $1 + \cot^2\theta = \csc^2\theta$.

Lei dos Senos

$$\frac{a}{\sin\widehat{A}} = \frac{b}{\sin\widehat{B}} = \frac{c}{\sin\widehat{C}} \tag{13.2}$$

Lei dos Cosenos

$$a^2 = b^2 + c^2 - 2bc\cos\widehat{A} \tag{13.3}$$

Funções Inversas

A seguir alguns resultados que envolvem o uso de funções inversas.

$$\arcsin(x) + \arccos(x) = \frac{\pi}{2}$$
$$\arctan(x) + arccot(x) = \frac{\pi}{2}$$
$$\arctan(x) + \arctan\frac{1}{x} = \pm\frac{\pi}{2}(\text{negativo se } x \text{ menor que zero})$$
$$\sin[\arccos(x)] = \sqrt{1-x^2}$$
$$\sin[\arctan(x)] = \frac{x}{\sqrt{1+x^2}}$$
$$\cos[\arctan(x)] = \frac{x}{\sqrt{1+x^2}}$$
$$\cos[\arcsin(x)] = \sqrt{1-x^2}$$

$$\tan[\arcsin(x)] = \frac{x}{\sqrt{1-x^2}}$$

$$\tan[\arccos(x)] = \frac{\sqrt{1-x^2}}{x}$$

$$\cot[\arcsin(x)] = \frac{\sqrt{1-x^2}}{x}$$

$$\cot[\arccos(x)] = \frac{x}{\sqrt{1-x^2}}$$

13.3.2 Simetria, Deslocamento e Periodicidade

Simetria

Quando funções trigonométricas são refletidas em certos ângulos, o resultado costuma ser o valor de outra função trigonométrica.

Reflexão em $\theta = 0$	Reflexão em $\theta = \frac{\pi}{4}$	Reflexão em $\theta = \frac{\pi}{2}$
$\sin(-\theta) = -\sin\theta$	$\sin(\frac{\pi}{2} - \theta) = \cos\theta$	$\sin(\pi - \theta) = \sin\theta$
$\cos(-\theta) = \cos\theta$	$\cos(\frac{\pi}{2} - \theta) = \sin\theta$	$\cos(\pi - \theta) = -\cos\theta$
$\tan(-\theta) = -\tan\theta$	$\tan(\frac{\pi}{2} - \theta) = \cot\theta$	$\tan(\pi - \theta) = -\tan\theta$
$\csc(-\theta) = -\csc\theta$	$\csc(\frac{\pi}{2} - \theta) = \sec\theta$	$\csc(\pi - \theta) = \csc\theta$
$\sec(-\theta) = \sec\theta$	$\sec(\frac{\pi}{2} - \theta) = \csc\theta$	$\sec(\pi - \theta) = -\sec\theta$
$\cot(-\theta) = -\cot\theta$	$\cot(\frac{\pi}{2} - \theta) = \tan\theta$	$\cot(\pi - \theta) = -\cot\theta$

Tabela 13.1: Reflexão de funções trigonométricas.

Deslocamento e Periodicidade

Como em geral as funções trigonométricas tem períodos de π ou 2π, é possível calcular o valor de ângulo alto por achar o de um mais baixo.

Deslocamento de $\frac{\pi}{2}$	Deslocamento de π	Deslocamento de 2π
$\sin(\theta + \frac{\pi}{2}) = \cos\theta$	$\sin(\theta + \pi) = -\sin\theta$	$\sin(\theta + 2\pi) = \sin\theta$
$\cos(\theta + \frac{\pi}{2}) = -\sin\theta$	$\cos(\theta + \pi) = -\cos\theta$	$\cos(\theta + 2\pi) = \cos\theta$
$\tan(\theta + \frac{\pi}{2}) = -\cot\theta$	$\tan(\theta + \pi) = \tan\theta$	$\tan(\theta + 2\pi) = \tan\theta$
$\csc(\theta + \frac{\pi}{2}) = \sec\theta$	$\csc(\theta + \pi) = -\csc\theta$	$\csc(\theta + 2\pi) = \csc\theta$
$\sec(\theta + \frac{\pi}{2}) = -\csc\theta$	$\sec(\theta + \pi) = -\sec\theta$	$\sec(\theta + 2\pi) = \sec\theta$
$\cot(\theta + \frac{\pi}{2}) = -\tan\theta$	$\cot(\theta + \pi) = \cot\theta$	$\cot(\theta + 2\pi) = \cot\theta$

Tabela 13.2: Deslocamento de funções trigonométricas.

Identidades sobre soma e diferença de ângulos

Ou o teorema da soma e subtração de ângulos.

Seno	$\sin(\alpha \pm \beta) = \sin\alpha\cos\beta \pm \cos\alpha\sin\beta$
Co-seno	$\cos(\alpha \pm \beta) = \cos\alpha\cos\beta \mp \sin\alpha\sin\beta$
Tangente	$\tan(\alpha \pm \beta) = \frac{\tan\alpha \pm \tan\beta}{1 \mp \tan\alpha\tan\beta}$
Arco-seno	$\arcsin\alpha \pm \arcsin\beta = \arcsin(\alpha\sqrt{1-\beta^2} \pm \beta\sqrt{1-\alpha^2})$
Arco-coseno	$\arccos\alpha \pm \arccos\beta = \arccos(\alpha\beta \mp \sqrt{(1-\alpha^2)(1-\beta^2)})$
Arco-tangente	$\arctan\alpha \pm \arctan\beta = \arctan(\frac{\alpha \pm \beta}{1 \mp \alpha\beta})$

Tabela 13.3: Soma e diferença de ângulos

Múltiplo de Ângulo

$$\sin n\theta = \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) \cos^k \theta \sin^{n-k} \theta \sin(\frac{1}{2}(n-k)\pi)$$

$$\cos n\theta = \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) \cos^k \theta \sin^{n-k} \theta \cos(\frac{1}{2}(n-k)\pi)$$

$$\tan(n+1)\theta = \frac{\tan n\theta + \tan\theta}{1 - \tan n\theta \tan\theta}$$

$$\cot(n+1)\theta = \frac{\cot n\theta \cot\theta - 1}{\cot n\theta + \cot\theta}$$

Redução de Potência

Seno	Co-seno	Outros
$\sin^2\theta = \frac{1-\cos 2\theta}{2}$	$\cos^2\theta = \frac{1+\cos 2\theta}{2}$	$\sin^2\theta\cos^2\theta = \frac{1-\cos 4\theta}{8}$
$\sin^3\theta = \frac{3\sin\theta - \sin 3\theta}{4}$	$\cos^3\theta = \frac{3\cos\theta + \cos 3\theta}{4}$	$\sin^3\theta\cos^3\theta = \frac{3\sin 2\theta - \sin 6\theta}{32}$
$\sin^4\theta = \frac{3-4\cos 2\theta + \cos 4\theta}{8}$	$\cos^4 = \frac{3+4\cos 2\theta + \cos 4\theta}{8}$	$\sin^4\theta\cos^4\theta = \frac{3-4\cos 4\theta + \cos 8\theta}{128}$
$\sin^5\theta = \frac{10\sin\theta - 5\sin 3\theta + \sin 5\theta}{16}$	$\cos^5\theta = \frac{10\cos\theta + 5\cos 3\theta + \cos 5\theta}{16}$	$\sin^5\theta\cos^5\theta = \frac{10\sin 2\theta - 5\sin 6\theta + \sin 10\theta}{512}$

Tabela 13.4: Redução de potências de Seno e Co-seno

Soma para produto e Produto para soma

De produto para soma:

$$\cos\theta\cos\alpha = \frac{\cos(\theta-\alpha) + \cos(\theta+\alpha)}{2}$$

$$\sin\theta\sin\alpha = \frac{\cos(\theta-\alpha) - \cos(\theta+\alpha)}{2}$$

$$\sin\theta\cos\alpha = \frac{\sin(\theta+\alpha) + \sin(\alpha-\theta)}{2}$$

$$\cos\theta\sin\alpha = \frac{\sin(\theta+\alpha) - \sin(\theta-\alpha)}{2}$$

De soma para produto:

$$\sin\theta \pm \sin\alpha = 2\sin(\frac{\theta\pm\alpha}{2})\cos(\frac{\theta\mp\alpha}{2})$$

$$\cos\theta + \cos\alpha = 2\cos(\frac{\theta+\alpha}{2})\cos(\frac{\theta-\alpha}{2})$$

$$\cos\theta - \cos\alpha = -2\sin(\frac{\theta+\alpha}{2})\sin(\frac{\theta-\alpha}{2})$$

13.4 Reta

Equação de reta: $y = kx + w$. Equação de reta que passa por um ponto $P_0(x_0, y_0)$ e tem coeficiente angular k: $y - y_0 = k(x - x_0)$. E por fim a equação de reta que passa por dois pontos:

$$y - y1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

13.5 Círculo

13.5.1 Propriedades de cordas e segmentos

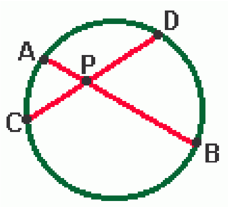

Figura 13.3: $(AP) \cdot (PB) = (CP) \cdot (PD)$

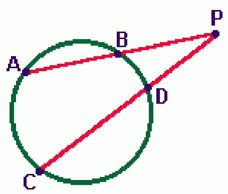

Figura 13.4: $(PA) \cdot (PB) = (PC) \cdot (PD)$

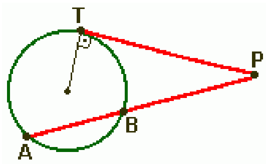

Figura 13.5: $(PT)^2 = (PA) \cdot (PB)$

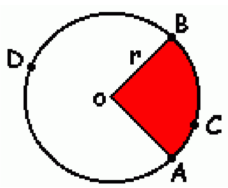

Figura 13.6: Área (setor OACB) = $\frac{\pi r^2 m}{360}$. Área (setor OACB) = $\frac{1}{2} r^2 m$ radianos.

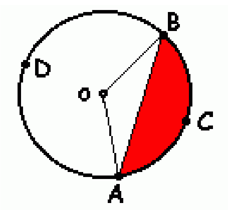

Figura 13.7: Área(segmento) = Área(setor OACB) - Área(triângulo AOB).

13.6 Great Circle Distance (Distância entre dois pontos na superfície de uma esfera)

O código abaixo assume que os pontos estão na terra (raio 6371). Substitua pelo raio conveniente.

Listing 13.1: Great Circle Distance

```
#include <iostream>
#include <cstdio>
#include <cmath>

#ifndef M_PI
#define M_PI 3.14159265358
#endif

using namespace std;

int main()
{
    int t, g, m;
    double s, x1, y1, x2, y2, r;
    char c;
    scanf(" %d", &t);
    for(; t>0; t--)
    {
          scanf(" %d %d %lf %c", &g, &m, &s, &c);
          s += double(g)*3600 + double(m)*60;
          if(c==’S’) s=-s;
          y1=s/3600;
          scanf(" %d %d %lf %c", &g, &m, &s, &c);
          s += double(g)*3600 + double(m)*60;
          if(c==’W’) s=-s;
          x1=s/3600;
          scanf(" %d %d %lf %c", &g, &m, &s, &c);
          s += double(g)*3600 + double(m)*60;
          if(c==’S’) s=-s;
          y2=s/3600;
          scanf(" %d %d %lf %c", &g, &m, &s, &c);
          s += double(g)*3600 + double(m)*60;
          if(c==’W’) s=-s;
          x2=s/3600;
          r=6371.01*acos(cos(M_PI*(90-y1)/180)*cos((90-y2)*M_PI/180)+sin((90-y1)*M_PI/180)*sin((90-y2)*M_PI/180)*cos((x1-x2)*M_PI/180));
          printf("%.2lf\n", r);
    }
    return 0;
}
```

13.7 Implementação usando números complexos

A maneira clássica de implementar algoritmos de geometria envolve criar uma estrutura de dois números flutuantes, implementar a distância entre dois pontos, lembrar como calcular ângulos, rotacionar pontos e muitas outras coisas que podem gerar erros. Tudo isto pode ser esquecido caso decida usar números complexos em C++, ver Listing 13.2.

Listing 13.2: Geometria com números Complexos

```
#include <complex>
```

```
// To save some typing later
typedef complex <double> pt;

// Given x, y coordinates, make a point (x, y) -- really the complex number x + yi
pt A = pt (x, y);

// X coordinate, Y coordinate
double x = A.real (), y = A.imag ();

// Angle point A makes with x-axis [0, 2 * Pi), angle between A and B
double angle = arg (A), angle_A_B = arg (A - B);

// Point A reflected across the x-axis (conjugate of a)
pt a_reflect = conj (A);

// Point A rotated by angle theta
pt a_rotated = A * exp (pt (0, theta));

// Distance between two points A, B
double dist = abs (A - B);

// Centroid of triangle ABC
pt centroid = (A + B + C) / 3;

// Dot product of vector A and vector B
double dot (pt A, pt B) {
    return real (conj (A) * B);
}

// Signed magnitude of A cross B
double cross (pt A, pt B) {
    return imag (conj (A) * B);
}

// Area of a triangle ABC
double triarea (pt A, pt B, pt C) {
    return 0.5 * abs (cross (B-A, C-A));
}

// Are two triangles A1, B1, C1, and A2, B2, C2 similar?
// First line checks if they're similar with same orientation,
// second line checks if they're similar with a different orientation
bool similar (pt A1, pt B1, pt C1, pt A2, pt B2, pt C2) {
    return (    (B2 - A2) * (C1 - A1) == (B1 - A1) * (C2 - A2)
        ||  (B2 - A2) * (conj (C1) - conj (A1)) == (conj (B1) - conj (A1)) * (C2 - A2)
        );
}
```

13.8 Matrizes de Rotação

13.8.1 Rotação 2D

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

13.8.2 Rotação 3D

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Capítulo 14

Bibliotecas

14.1 math.h

Protótipo da função	Significado
double acos(double x);	Retorna o arco-coseno de **x**. $2acos(0) = 3.1415926536$.
double asin(double x);	Retorna o arco-seno de **x**.
double atan(double x);	Retorna o arco-tangente de **x**.
double atan2(double x, double y);	Retorna o arco-tangente de $\frac{x}{y}$.
double cos(double x);	Retorna o coseno de **x**.
double sin(double x);	Retorna o seno de **x**.
double tan(double x);	Retorna o tangente de **x**.
double cosh(double x);	Retorna o coseno hiperbólico de **x**.
double sinh(double x);	Retorna o seno hiperbólico de **x**.
double tanh(double x);	Retorna o tangente hiperbólico de **x**.
double exp(double x);	Retorna e^x, onde $e = 2.7182818284590452354$.
double exp2(double x);	Retorna 2^x.
double log(double x);	Retorna o logaritmo natural de **x**.
double log10(double x);	Retorna o logaritmo de **x** na base 10.
double log2(double x);	Retorna o logaritmo de **x** na base 2.
double sqrt(double x);	Retorna a raiz quadrada de **x**.
double ceil(double x);	Retorna o teto de **x**.
double floor(double x);	Retorna o chão de **x**.
double pow(double x, double y);	Retorna x^y.
double modf(double x, double *y);	Divide o número real **x** em sua parte fracionária (o retorno da função) e sua parte inteira (retornada em **y**). Cada parte recebe o mesmo sinal que havia em **x**.
float fabsf(float x);	Retorna o valor absoluto do float **x**.
double fabs(double x);	Retorna o valor absoluto do double **x**.

Tabela 14.1: Funções úteis presentes na math.h.

14.2 string.h

Protótipo da função	Significado
char *strcpy(char *destination, char *source);	Copia a string **source** na string **destination**.
char *strncpy(char *destination, char *source, size_t n);	Copia **n** caracteres da **source** na string **destination**. Tomar cuidado, pois ao final da cópia não é adicionado um 0 na string **destination**.
char *strdup(char *source);	Copia a string **source** na string que é retornada pela função (string de retorno alocada dinamicamente).
char *strcat(char *str1, char *str2);	Concatena a string **str2** na **str1**.
int strcmp(char *str1, char *str2);	Compara as strings **str2** e **str1**. Retorna 0 se forem iguais; Maior que zero se **str1** > **str2**; Menor que zero caso contrário.
int strncmp(char *str1, char *str2, size_t n);	Igual ao strcmp, porém só compara **n** caracteres.
int strcasecmp(char *str1, char *str2);	Igual ao strcmp, porém efetua a comparação sem levar em conta o *case* (*case insensitive*).
char *strchr(char *str, int ch);	Retorna a primeira ocorrência do caractere **ch** na string **str**. Caso não tenha achado, retorna NULL.
char *strstr(char *str1, char *str2);	Procura pela substring **str2** na string **str1**. Retorna NULL caso não tenha encontrado; se encontrou, retorna um ponteiro para o primeiro caractere de **str2** em **str1**.
size_t strlen(char *str);	Retorna o tamanho da string **str**.

Tabela 14.2: Funções úteis presentes na string.h.

A função **strrev()**, que inverte a string, **não pertence ao C ANSI**.

14.3 stdlib.h

Protótipo da função	Significado
int atoi(char *ptr);	Converte uma string para um inteiro.
long atol(char *ptr);	Converte uma string para um long.
double atof(char *ptr);	Converte uma string para um double.
int abs(int x);	Retorna o valor absoluto do inteiro passado como parâmetro.
long labs(long x);	Retorna o valor absoluto de um long.
void qsort(void *base, size_t nel, size_t width, int (*compar)(const void *, const void *));	Função de quicksort que recebe um ponteiro pro vetor a ser ordenado **(void *base)**; a quantidade de elementos no vetor **(size_t nel)**; o tamanho em bytes de um elemento **(size_t width)**; e uma função de comparação.

Tabela 14.3: Funções úteis presentes na stdlib.h.

14.4 ctype.h

Protótipo da função	Significado
int isalnum(int c);	Verifica se um caractere **c** é um número ou uma letra, retorna 0 em caso negativo.
int isalpha(int c);	Verifica se um caractere **c** é uma letra, retorna 0 em caso negativo.
int isascii(int c);	Verifica se um caractere **c** é um caractere padrão ASCII (entre 0 e 127), retorna 0 em caso negativo.
int isgraph(int c);	Verifica se um caractere **c** é 'imprimível' (espaço não é considerado), retorna 0 em caso negativo.
int isprint(int c);	Verifica se um caractere **c** é 'imprimível' (espaço é considerado), retorna 0 em caso negativo.
int islower(int c);	Verifica se um caractere **c** é uma letra em *lowercase*, retorna 0 em caso negativo.
int isupper(int c);	Verifica se um caractere **c** é uma letra em *uppercase*, retorna 0 em caso negativo.
int ispunct(int c);	Verifica se um caractere **c** é um caractere de pontuação, retorna 0 em caso negativo.
int isspace(int c);	Verifica se um caractere **c** é o caractere espaço, retorna 0 em caso negativo.
int isxdigit(int c);	Verifica se um caractere **c** é um caractere hexadecimal, retorna 0 em caso negativo.
int touppoer(int c);	Retorna o caractere **c** em *uppercase*.
int tolower(int c);	Retorna o caractere **c** em *lowercase*.

Tabela 14.4: Funções úteis presentes na ctype.h.

14.5 limits.h

Definições	Significado
CHAR_BIT	Número de bits em uma variável do tipo char (8).
CHAR_MAX	Máximo valor de uma variável do tipo char.
CHAR_MIN	Mínimo valor de uma variável do tipo char.
INT_MAX	Máximo valor de uma variável do tipo int.
INT_MIN	Mínimo valor de uma variável do tipo int.
LONG_MAX	Máximo valor de uma variável do tipo long.
LONG_MIN	Mínimo valor de uma variável do tipo long.
SCHAR_MAX	Máximo valor de uma variável do tipo signed char.
SCHAR_MIN	Mínimo valor de uma variável do tipo signed char.
SHRT_MAX	Máximo valor de uma variável do tipo short.
SHRT_MIN	Mínimo valor de uma variável do tipo short.
UCHAR_MAX	Máximo valor de uma variável do tipo unsigned char.
UINT_MAX	Máximo valor de uma variável do tipo unsigned int.
ULONG_MAX	Máximo valor de uma variável do tipo unsigned long.
USHRT_MAX	Máximo valor de uma variável do tipo unsigned short.

Tabela 14.5: Definições úteis presentes na limits.h.

Para definicões de valores em pontos flutuantes, usar a biblioteca **float.h**. Para os valores de máximos e mínimos, consultar as tabelas na seção 1.2.

Capítulo 15

Miscellaneous

15.1 Josephus

15.2 Xadrez

15.3 Poker

15.4 Notação Postfix

Antes de seguir com o algoritmo, são ncessárias algumas convenções: **p** é uma pilha; **in[]** é a string infix de entrada; e **out[]** é a string posfix, que é o resultado do algoritmo.O algoritmo para passar da notação infix para posfix é dado a seguir:

1. Inicialize uma **pilha p** vazia;

2. De um **p.push(**() e adicione um ")"no final da **string infix**;

3. Leia a **string infix** da esquerda para à direita até o seu fim, repetindo os passos a seguir de acordo com cada caso:

   - Se for um **operando**, então o adicione à out[] e avance para o próximo caractere da in[];
   - Se for um **abrir de parênteses**, então **p.push(**() e avance para o próximo caractere da in[];
   - Se for um **fechar de parênteses**, então remova os elementos da pilha (os adicionando ao out[]) até que encontre um abrir de parênteses (não o adicione ao out[]). Então avance para o próximo caractere da in[];
   - Se for um **operador**, então desempilhe todos os operadores (os adicionando ao out[]) enquanto eles tiverem a mesma ou uma precedência mais alta do que o operador lido na in[]. Depois de desempilhar, adicione o operador lido na pilha e avance para o próximo caractere da in[].

Complexidade: $O(n)$.
Entrada: **(char *) in**, que contém o endereço da string em notação infix; **(int) tam**, que contém o tamanho da string infix; e **(char *) out**, que contém o endereço da string em notação posfix.
Saída: É salvo em **(char *) out** a notação em posfix.

Listing 15.1: De notação infix para posfix

```
int isoperador(char c) {
    if (c == '+' || c == '-' || c == '*' || c == '/') return 1;
    return 0;
}

void InfixParaPosfix(char *in, int tam, char *out) {
    stack < char > p;
    int n, i, j;
```

```

    p.push(’(’);
    in[tam] = ’)’;
    tam++;
    n = 0;
    for (i=0; i < tam; ) {
        if (isdigit(in[i])) {
            out[n++] = in[i++];
        }
        else if (in[i] == ’(’) {
            p.push(in[i++]);
        }
        else if (in[i] == ’)’) {
            while (p.top() != ’(’) {
                out[n++] = p.top();
                p.pop();
            }
            p.pop();
            i++;
        }
        else if (isoperador(in[i])) {
            if (in[i] == ’+’ || in[i] == ’-’) {
                while (isoperador(p.top())) {
                    out[n++] = p.top();
                    p.pop();
                }
                p.push(in[i]);
            }
            else {
                while (p.top() == ’*’ || p.top() == ’/’) {
                    out[n++] = p.top();
                    p.pop();
                }
                p.push(in[i]);
            }
            i++;
        }
    }
    out[n] = ’\0’;
}
```

Capítulo 16

STL & Algorithm

Esse capítulo tem o objetivo de resumir as bibliotecas stl. Nem todas as funcionalidades estão descritas, mas algumas são simples e não tem vantagem em relação a qualquer código simples e rápido de ser feito.

16.1 STL

Vários dos exemplos dessa seção foram baseados nos exemplos presentes no seguinte slide: `http://www.dcc.ufrj.br/vitormaia/programas/tutorialc++/stl.pdf`.

16.1.1 Stack

Implementação da estrutura pilha, primeiro que entra é o último a sair.

Função	Significado
empty()	retorna verdadeiro caso a pilha esteja vazia.
size()	retorna a quantidade de elementos na pilha.
top()	retorna o elemento do topo da pilha
push(T)	insere o elemento T no topo da pilha.
pop()	remove o elemento do topo da pilha.

Tabela 16.1: Funções presentes na Stack da STL.

No Listing 16.1 se tem um exemplo de uso de todas essas funções.

Listing 16.1: Stack

```
#include <iostream>
#include <stack>

using namespace std;

int main (void) {
    stack<int> p;

    p.push(7);
    p.push(3);
    p.push(5);
    p.push(1);
    cout << "size: " << p.size() << endl;
    while(!p.empty()) {
        cout << p.top() << endl;
```

```
        p.pop();
    }
}
```

16.1.2 Queue

Implementação da estrutura fila, primeiro que entra é o primeiro a sair.

Função	Significado
empty()	retorna verdadeiro caso a fila esteja vazia.
size()	retorna a quantidade de elementos na fila.
front()	retorna o elemento do começo da fila
back()	retorna o elemento do fim da fila
push(T)	insere o elemento T no fim da fila.
pop()	remove o elemento do começo da fila.

Tabela 16.2: Funções presentes na Queue da STL.

No Listing 16.2 se tem um exemplo de uso de todas essas funções.

Listing 16.2: Queue

```
#include <iostream>
#include <queue>

using namespace std;

int main (void) {
    queue<int> f;

    f.push(3);
    f.push(8);
    f.push(6);
    cout << f.size() << endl;
    cout << f.back() << endl;
    cout << f.front() << endl;
    f.pop();
    if(!f.empty())
        cout << f.front() << endl;
}
```

16.1.3 Vector

Vector é uma lista encadeada com a diferença de possuir índices.

Função	Significado
empty()	retorna verdadeiro caso o vector esteja vazio.
size()	retorna a quantidade de elementos no vector.
resize(x)	modifica o tamanho alocado do vector para x unidades.
begin() e end()	iteradores do vector.
at(x)	retorna o elemento da posição x do vector.
back() e front()	retorna o último ou o primeiro elemento do vector.
erase(x)	remove o elemento da posição x.
push_back(T)	insere o elemento T no fim do vector.
pop_back()	remove o elemento do fim do vector.
insert(x, T)	insere o elemento T na posição x do vector, empurrando os outros.
clear()	limpa o vector.

Tabela 16.3: Funções presentes na Vector da STL.

No Listing 16.3 se tem exemplos de construtores de vector e como percorre-lo de modo normal e reverso.

Listing 16.3: Vector (Construtores e Iteração)

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

using namespace std;

int main (void) {
    // vector vazio
    vector<int> v1;
    // vector de 4 posicoes com 2 em todos
    vector<int> v2(4,2);
    // vector v3 copia o conteudo de v2 no intervalo dado
    vector<int> v3(v2.begin(), v2.end());
    // v4 copia conteudo de v3
    vector<int> v4(v3);

    // vector apartir de um vetor de inteiros
    int vetor[] = {1,2,3,4,5};
    vector<int> v5(vetor, vetor+sizeof(vetor)/sizeof(int));

    // v6 vira copia de v5
    vector<int> v6;
    v6 = v5;

    // v6 se torna vazio
    v6 = vector<int>();

    // iteracao "normal"
    vector<int>::iterator it;
    for(it = v4.begin(); it < v4.end(); it++)
        cout << *it << " ";
    cout << endl;

    // iteracao "reversa"
    vector<int>::reverse_iterator it2;
    for(it2 = v5.rbegin(); it2 < v5.rend(); it2++ )
        cout << *it2 << " ";
    cout << endl;
}
```

No Listing 16.4 se tem um exemplo de uso das funções de acesso e inserção.

Listing 16.4: Vector (Acesso)

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

using namespace std;

int main (void) {
    vector<int> v;

    v.push_back(2);
    v.insert(v.begin(), 1);
    v.push_back(3);
    cout << v.front() << endl;
    cout << v.back() << endl;
    cout << v.at(1) << endl;
    /* imprime: 1 3 2 */
}
```

No Listing 16.5 se tem um exemplo de uso das funções de remoção.

Listing 16.5: Vector (Erase)

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

using namespace std;

int main (void) {
    vector<int> v;
    vector<int>::iterator it;

    /* insercao de elementos de 1..10 */
    for(int i=0; i < 10; i++)
        v.push_back(i+1);

    /* remove os 3 primeiros, 1..3 */
    v.erase(v.begin(), v.begin() + 3);
    /* remove o ultimo */
    v.erase(v.end()-1);

    for(it=v.begin(); it < v.end(); it++)
        cout << *it << " ";
    cout << endl << v.size() << endl;
    /* imprime: 4 5 6 7 8 9 \n 6 */
}
```

16.1.4 Deque

Deque significa '*double-ended queue*', ou seja, fila com dois fins. De forma prática ele pode ser resumido como um vector que também aceita inserções e remoções no começo pelas funções **push_front()** e **pop_front()**.

16.1.5 Map

Map é a estrutura de árvore de busca binária pronta da STL. Implementa inserção, consulta e remoção em tempo $O(logn)$.

Função	Significado
empty()	retorna verdadeiro caso o map esteja vazio.
size()	retorna a quantidade de elementos no map.
begin() e end()	iteradores do map.
erase(x)	remove o elemento de chave x.
find(x)	retorna um iterador para o elemento de chave x (se não encontrar retorna end()).
clear()	limpa o map.

Tabela 16.4: Funções presentes na Map da STL. Ver inserção e acesso no exemplo.

Para saber como acessar e inserir elementos veja o Listing 16.6, que exemplifica o uso de suas funções.

Listing 16.6: Map

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

int main (void) {
    /* map<chave, valor> */
    map<int, string> m;

    /* varias maneiras de insercao */
    m[1] = "um";
    m[2] = "dois";
    m.insert(pair<int, string>(3, "tres"));
    m.insert(map<int, string>::value_type(4, "quatro"));
    m.insert(make_pair(5, "cinco"));

    /* impressao de todos os dados e o tamanho */
    cout << m.size() << endl;
    for (map<int, string>::iterator it = m.begin(); it != m.end(); it++)
        cout << (*it).first << ": " << (*it).second << endl;

    /* busca por elemento */
    if(!m.empty())
        if(m.find(2) != m.end()) {
            cout << m[2] << endl;
        }
        else {
            cout << "Nao presente" << endl;
        }

    /* remove */
    m.erase(2);
    cout << m.size() << endl;

    /* limpa */
    m.clear();
}
```

16.2 Algorithm

16.2.1 accumulate

Formato: TYPE accumulate(input_iterator start, input_iterator end, TYPE val);

O accumulate soma todos os valores no intervalo [*start,end*) ao valor inicial *val*. Pode fazer isso com números inteiros, double ou até mesmo string, como no código 16.7.

Listing 16.7: STL Algorithm - accumulate

```
int main () {
    string str = "Hello World!";
    vector<string> vec(10,str);   // vec = ["Hello World!", "Hello World!", ...]
    string a = accumulate( vec.begin(), vec.end(), string("") );
    cout << a << endl;            // displays "Hello World!Hello World!Hello..."
}
```

16.2.2 binary_search

Formato: bool binary_search(forward_iterator start, forward_iterator end, const TYPE& val);

Realiza a busca binária no vetor. O código 16.8 traz um exemplo de uso.

Listing 16.8: STL Algorithm - binary_search

```
  int nums[] = { -242, -1, 0, 5, 8, 9, 11 };
  int start = 0;
  int end = 7;

  for( int i = 0; i < 10; i++ ) {
    if( binary_search( nums+start, nums+end, i ) ) {
      cout << "nums[] contains " << i << endl;
    } else {
      cout << "nums[] DOES NOT contain " << i << endl;
    }
  }
```

16.2.3 copy e copy_backward

Formato: output_iterator copy(input_iterator start, input_iterator end, output_iterator dest);

O copy realiza a cópia dos elementos entre *start* e *end* em tempo linear para um vetor destino. O valor de retorno é a posição do destino depois de todos elementos serem copiados. um exemplo de uso se encontra no código 16.9.

Listing 16.9: STL Algorithm - copy

```
  vector<int> from_vector;
  for( int i = 0; i < 10; i++ ) {
    from_vector.push_back( i );
  }

  vector<int> to_vector(10);

  copy( from_vector.begin(), from_vector.end(), to_vector.begin() );

  cout << "to_vector contains: ";
  copy( to_vector.begin(), to_vector.end(), ostream_iterator<int>( cout, " " ) );

  cout << endl;
```

O copy_backward é semelhante, mas faz a cópia invertida dos elementos. Ele é muito semelhante, mas deve-se passar o final do intervalo de destino com terceiro argumento, uma vez que que a cópia é feita do fim do destino para o começo. O código 16.10 exemplifica como isso acontece.

Listing 16.10: STL Algorithm - copy_backward

```
vector<int> from_vector;
for( int i = 0; i < 10; i++ ) {
  from_vector.push_back( i );
}

vector<int> to_vector(15);

copy_backward( from_vector.begin(), from_vector.end(), to_vector.end() );

cout << "to_vector contains: ";
for( unsigned int i = 0; i < to_vector.size(); i++ ) {
  cout << to_vector[i] << " ";
}
cout << endl;

//   to_vector contains: 0 0 0 0 0 0 1 2 3 4 5 6 7 8 9
```

16.2.4 equal_range e equal

Formato: pair <forward_iterator, forward_iterator> equal_range(forward_iterator first, forward_iterator last, const TYPE& val);

O equal_range retorna o começo e o fim de um intervalo, entre *first* e *last*, que contenha valores equivalentes a *val*. Além do formato mostrado, pode conter um quarto parâmetro indicando um comparador, como pode ser visto no exemplo 16.11. Sua complexidade é no máximo log(n) comparações e linear em [*first*, *last*). Se usado com iteradores de acesso aleatório, o número de passos é reduzido pra log.

Listing 16.11: STL Algorithm - equal_range

```
// equal_range example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool mygreater (int i,int j) { return (i>j); }

int main () {
  int myints[] = {10,20,30,30,20,10,10,20};
  vector<int> v(myints,myints+8);                         // 10 20 30 30 20 10 10 20
  pair<vector<int>::iterator,vector<int>::iterator> bounds;

  // using default comparison:
  sort (v.begin(), v.end());                              // 10 10 10 20 20 20 30 30
  bounds=equal_range (v.begin(), v.end(), 20);            //          ^        ^

  // using "mygreater" as comp:
  sort (v.begin(), v.end(), mygreater);                   // 30 30 20 20 20 10 10 10
  bounds=equal_range (v.begin(), v.end(), 20, mygreater); //       ^        ^

  cout << "bounds at positions " << int(bounds.first - v.begin());
  cout << " and " << int(bounds.second - v.begin()) << endl;

```

```
  return 0;
}
```

O equal apenas indica se dois intervalos são exatamente iguais. **Formato:** bool equal(input_iterator start1, input_iterator end1, input_iterator2 start2); Um exemplo de sua utilização se encontra em 16.12.

Listing 16.12: STL Algorithm - equal

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back( i );
}

vector<int> v2;
for( int i = 0; i < 10; i++ ) {
  v2.push_back( i );
}

if( equal( v1.begin(), v1.end(), v2.begin() ) ) {
  cout << "v1 and v2 are equal" << endl;
} else {
  cout << "v1 and v2 are NOT equal" << endl;
}
```

16.2.5 find

Formato: TYPE input_iterator find(input_iterator start, input_iterator end, const TYPE& val);

Procura em um conjunto de elementos se há a ocorrência de um determinado elemento. Retorna a posição encontrada ou um pounteiro para o fim da estrutura. Um exemplo de uso pode ser visto em 16.13.

Listing 16.13: STL Algorithm - find

```
int nums[] = { 3, 1, 4, 1, 5, 9 };

int num_to_find = 5;
int start = 0;
int end = 2;
int* result = find( nums + start, nums + end, num_to_find );

if( result == nums + end )
    cout << "Did not find any number matching " << num_to_find << endl;
else
    cout << "Found a matching number: " << *result << endl;
```

16.2.6 includes

Formato: template < typename InIterA, typename InIterB > bool includes(InIterA start1, InIterA end1, InIterB start2, InIterB end2);

ou

template < typename InIterA, typename InIterB, typename StrictWeakOrdering > bool includes(InIterA start1, InIterA end1, InIterB start2, InIterB end2, StrictWeakOrdering cmp);

Retorna verdadeiro se um conjunto é subconjunto de outro. Sua complexidade é O(n). Seu uso é simples e pode ser entendido em 16.14.

Listing 16.14: STL Algorithm - includes

```
#include <iostream>
```

```
#include <vector>
#include <list>
#include <algorithm>

int main() {
    std::vector<int> vec;
    for (int i = 0; i < 10; ++i)
        vec.push_back(i);

    std::list<int> lst;
    lst.push_back(2);
    lst.push_back(4);
    lst.push_back(6);

    if (std::includes(vec.begin(), vec.end(), lst.begin(), lst.end()))
        std::cout << "lst is a subset of vec." << std::endl;
    else
        std::cout << "lst is NOT a subset of vec." << std::endl;

}
```

16.2.7 lexicographical_compare

Formato: bool lexicographical_compare(input_iterator start1, input_iterator end1, input_iterator2 start2, input_iterator2 end2);

Realiza a comparação lexicográfica de dois conjuntos de elementos. Retorna *true* se o intervalo *[start1,end1)* é lexicograficamente menor que *[start2,end2)*. Retorna *false* caso contrário.

16.2.8 lower_bound

Formato: forward_iterator lower_bound(forward_iterator first, forward_iterator last, const TYPE& val);

Retorna o primeiro lugar em que o elemento pode ser inserido sem alterar a ordem do vetor. Pode-se ver um exemplo em 16.15.

Listing 16.15: STL Algorithm - lower_bound

```
vector<int> nums;
nums.push_back( -242 );
nums.push_back( -1 );
nums.push_back( 0 );
nums.push_back( 5 );
nums.push_back( 8 );
nums.push_back( 8 );
nums.push_back( 11 );

cout << "Before nums is: ";
for( unsigned int i = 0; i < nums.size(); i++ ) {
 cout << nums[i] << " ";
}
cout << endl;

vector<int>::iterator result;
int new_val = 7;

result = lower_bound( nums.begin(), nums.end(), new_val );

nums.insert( result, new_val );
```

```
cout << "After, nums is: ";
for( unsigned int i = 0; i < nums.size(); i++ ) {
 cout << nums[i] << " ";
}
cout << endl;
```

16.2.9 max_element e min_element

Formato: forward_iterator max_element(forward_iterator start, forward_iterator end);

Encontra o maior (ou o menor) elemento em um conjunto de valores. Exemplo em 16.16.

Listing 16.16: STL Algorithm - max_element

```
int array[] = { 3, 1, 4, 1, 5, 9 };
unsigned int array_size = sizeof(array) / sizeof(array[0]);
cout << "Max element in array is " << *max_element(array, array+array_size) << endl;

vector<char> v;
v.push_back('a'); v.push_back('b'); v.push_back('c'); v.push_back('d');
cout << "Max element in the vector v is " << *max_element(v.begin(), v.end()) << endl;
```

16.2.10 merge e inplace_merge

Formato: output_iterator merge(input_iterator start1, input_iterator end1, input_iterator2 start2, input_iterator2 end2, output_iterator result);

e

void inplace_merge(bidirectional_iterator start, bidirectional_iterator middle, bidirectional_iterator end);

Faz o merge de dois intervalos, *[start1,end1)* e *[start2,end2)*, Num único intervalo ordenado. O método retorna o iterador do final do intervalo resultante. Para melhor entendimento, ver o código 16.17.

Listing 16.17: STL Algorithm - merge

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

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  vector<int> v(10);
  vector<int>::iterator it;

  sort (first,first+5);
  sort (second,second+5);
  merge (first,first+5,second,second+5,v.begin());

  cout << "The resulting vector contains:";
  for (it=v.begin(); it!=v.end(); ++it)
    cout << " " << *it;

  cout << endl;

  return 0;
}
```

16.2.11 next_permutation e prev_permutation

Formato: bool next_permutation(bidirectional_iterator start, bidirectional_iterator end);

Rearranja o elementos de *[start,end)* na próxima permutação (ordem lexicográfica). Um código de exemplo encontra-se em 16.18.

Listing 16.18: STL Algorithm - next_permutation

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

int main () {
  int myints[] = {1,2,3};

  cout << "The 3! possible permutations with 3 elements:\n";

  sort (myints,myints+3);

  do {
    cout << myints[0] << " " << myints[1] << " " << myints[2] << endl;
  } while ( next_permutation (myints,myints+3) );

  return 0;
}
```

16.2.12 remove e remove_if

Formato: forward_iterator remove(forward_iterator start, forward_iterator end, const TYPE& val);

e

forward_iterator remove_if(forward_iterator start, forward_iterator end, Predicate p);

Simplesmente remove valores de um intervalo, sendo valores específicos (remove) ou que respondem a uma condição (remove_if). Um exemplo do remove_if se encontra em 16.19.

Listing 16.19: STL Algorithm - remove_if

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

bool IsOdd (int i) { return ((i%2)==1); }

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

  // bounds of range:
  int* pbegin = myints;                          // ^
  int* pend = myints+sizeof(myints)/sizeof(int); // ^                 ^

  pend = remove_if (pbegin, pend, IsOdd);        // 2 4 6 8 5 6 7 8 9
                                                 // ^       ^
  cout << "range contains:";
  for (int* p=pbegin; p!=pend; ++p)
    cout << " " << *p;

  cout << endl;

  return 0;
```

```
}
```

16.2.13 replace e replace_if

Formato: void replace(forward_iterator start, forward_iterator end, const TYPE& old_value, const TYPE& new_value);

e

void replace_if(forward_iterator start, forward_iterator end, Predicate p, const TYPE& new_value);

Simplesmente altera valores de um intervalo por um valor desejado, sendo valores específicos (replace) ou que respondem a uma condição (replace_if). Um exemplo do remove_if se encontra em 16.19.

Listing 16.20: STL Algorithm - replace_if

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

bool IsOdd (int i) { return ((i%2)==1); }

int main () {
  vector<int> myvector;
  vector<int>::iterator it;

  // set some values:
  for (int i=1; i<10; i++) myvector.push_back(i);          // 1 2 3 4 5 6 7 8 9

  replace_if (myvector.begin(), myvector.end(), IsOdd, 0); // 0 2 0 4 0 6 0 8 0

  cout << "myvector contains:";
  for (it=myvector.begin(); it!=myvector.end(); ++it)
    cout << " " << *it;

  cout << endl;

  return 0;
}
```

16.2.14 set_difference e set_symmetric_difference

Formato: template¡ typename InIterA, typename InIterB, typename OutIter ¿ OutIter set_difference(input_iterator start1, input_iterator end1, input_iterator start2, input_iterator end2, output_iterator result);

e

output_iterator set_symmetric_difference(input_iterator start1, input_iterator end1, input_iterator start2, input_iterator end2, output_iterator result);

Esses dois métodos realizam, respectivamente, a diferença e a diferença simétrica de dois conjuntos (ordenados). Para melhor entendimento, foram colocados dois códigos exemplos parecidos, um para cada método, 16.21 e 16.22.

Listing 16.21: STL Algorithm - set_difference

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

int main () {
  int first[] = {5,10,15,20,25};
```

```
  int second[] = {50,40,30,20,10};
  vector<int> v(10);                           // 0  0  0  0  0  0  0  0  0  0
  vector<int>::iterator it;

  sort (first,first+5);     //  5 10 15 20 25
  sort (second,second+5);   // 10 20 30 40 50

  it=set_difference (first, first+5, second, second+5, v.begin());
                                               // 5 15 25  0  0  0  0  0  0  0

  cout << "difference has " << int(it - v.begin()) << " elements.\n";

  return 0;
}
```

Listing 16.22: STL Algorithm - set_symmetric_difference

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

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  vector<int> v(10);                           // 0  0  0  0  0  0  0  0  0  0
  vector<int>::iterator it;

  sort (first,first+5);     //  5 10 15 20 25
  sort (second,second+5);   // 10 20 30 40 50

  it=set_symmetric_difference (first, first+5, second, second+5, v.begin());
                                               // 5 15 25 30 40 50  0  0  0  0

  cout << "symmetric difference has " << int(it - v.begin()) << " elements.\n";

  return 0;
}
```

16.2.15 set_intersection e set_union

Formato: output_iterator set_intersection(input_iterator start1, input_iterator end1, input_iterator2 start2, input_iterator2 end2, output_iterator result);

e

output_iterator set_union(input_iterator start1, input_iterator end1, input_iterator2 start2, input_iterator2 end2, output_iterator result);

Esses dois métodos realizam, respectivamente, a intersecção e a união de dois conjuntos (ordenados), resultando em um conjunto final *result*. Para melhor entendimento, foi colocado um código exemplo para o método de intersecção, 16.23.

Listing 16.23: STL Algorithm - set_intersection

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

```

```
int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  vector<int> v(10);                           // 0  0  0  0  0  0  0  0  0  0
  vector<int>::iterator it;

  sort (first,first+5);     //  5 10 15 20 25
  sort (second,second+5);   // 10 20 30 40 50

  it=set_intersection (first, first+5, second, second+5, v.begin());
                                               // 10 20 0  0  0  0  0  0  0  0

  cout << "intersection has " << int(it - v.begin()) << " elements.\n";

  return 0;
}
```

16.2.16 sort e stable_sort

Formato: void sort(random_iterator start, random_iterator end);

Ordena o conjunto de elementos no intervalo *[start,end)* em ordem crescente. Pode-se utilizar uma função de comparação e associá-la em um terceiro argumento. Segue um exemplo de uso em 16.24.

Listing 16.24: STL Algorithm - sort

```
// sort algorithm example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool myfunction (int i,int j) { return (i<j); }

struct myclass {
    bool operator() (int i,int j) { return (i<j);}
} myobject;

int main () {

    int myints[] = {32,71,12,45,26,80,53,33};
    vector<int> myvector (myints, myints+8);             // 32 71 12 45 26 80 53 33
    vector<int>::iterator it;

    // using default comparison (operator <):
    sort (myvector.begin(), myvector.begin()+4);         //(12 32 45 71)26 80 53 33

    // using function as comp
    sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71(26 33 53 80)

    // using object as comp
    sort (myvector.begin(), myvector.end(), myobject);     //(12 26 32 33 45 53 71 80)

    // print out content:
    cout << "myvector contains:";
    for (it=myvector.begin(); it!=myvector.end(); ++it)
    cout << " " << *it;

    cout << endl;
```

```

    return 0;

}
```

Capítulo 17

Problemas Resolvidos

17.1 —: Séries de Tubos

Listing 17.1: —: Séries de tubos

```
#include <cstdio>
#include <cstring>

// Serie de Tubos - Pontes

#define max 1010

int g[max][max];
int pre[max], low[max];
int c;
int n, m;
int a, b;
bool tem;

void ponte(int v, int a) {
    low[v] = pre[v] = c++;
    for (int i = 1; i <= n; i++)
        if (g[v][i] > 0)
            if (pre[i] == -1) {
                ponte(i, v);
                if (low[v] > low[i]) low[v] = low[i];
                if (low[i] == pre[i])
                    tem = true;
            } else if (i != a)
                if (low[v] > pre[i])
                    low[v] = pre[i];
}

int main() {

    while (1) {
        scanf("%d %d ", &n, &m);
        if (n == 0 && m == 0) break;
        c = 0;
        memset(g, 0, sizeof(g));

        for (int i = 1; i <= m; i++) {
            scanf("%d %d ", &a, &b);
```

```
            g[a][b] = 1;
            g[b][a] = 1;
        }

        for (int i = 1; i <= n; i++)
            pre[i] = -1;
        tem = false;

        ponte(1, 0);
        if (tem) printf("N\n");
        else printf("S\n");
    }
}
```

17.2 108: Maximum Sum (Kadane)

Listing 17.2: 108: Maximum Sum (Kadane)

```
#include <cstdio>
#include <cstring>

// Maximum Sum 2D usando Kadane O(n^3)

int main() {
    int n;
    int m[110][110];
    int c[110], t[110], max;

    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &m[i][j]);

    max = 0;
    for (int i = 0; i < n; i++) {
        memset(c, 0, sizeof(c));
        for (int j = i; j < n; j++) {
            for (int x = 0; x < n; x++)
                c[x] += m[j][x];
            memset(t, 0, sizeof(t));
            t[0] = (c[0] > 0)? c[0] : 0;
            for (int k = 1; k < n; k++){
                if (t[k-1]+c[k] > t[k]) t[k] = t[k-1]+c[k];
                if (t[k] > max) max = t[k];
            }
        }
    }
    printf("%d\n", max);
    return 0;
}
```

17.3 439: Knight Moves

Listing 17.3: 439: Knight Moves

```
#include <cstdio>
```

```
#include <queue>
#include <utility>
#include <cstring>

/* UVa 439 - Knight moves */
/* Busca em largura */

char moves[][2]= {
    {1, 2}, {1, -2}, {-1, 2}, {-1, -2},
    {2, 1}, {-2, 1}, {2, -1}, {-2, -1}
};

using namespace std;

int main() {
    char a, b;
    int ia, ib;

    char c[8][8];

    queue< pair<int, int> > q;
    pair<int, int> p;

    int nivel;

    while (scanf("%c%d %c%d%*c", &a, &ia, &b, &ib) != EOF) {
        ia--; ib--;
        q.push( make_pair(a-’a’, ia) );
        q.push( make_pair(-1, 0) );

        memset(c, 0, sizeof(c));
        nivel = 0;

        while (!q.empty()) {
            p = q.front();
            q.pop();

            if (p.first == -1) {
                /*if (q.empty()) // nunca acontece(sempre tem solucao), mas pra deixar o codigo generico
                    break;
                else*/
                    nivel++;
                p = q.front();
                q.pop();
                q.push( make_pair(-1, 0) );
            }

            //printf(" tratando %c%d\n", p.first+’a’, p.second);

            if (p.first == b-’a’ && p.second == ib) break;

            for (int i = 0; i < 8; i++) {
                if ( p.first + moves[i][0] >= 0 && p.first + moves[i][0] < 8 &&
                     p.second + moves[i][1] >= 0 && p.second + moves[i][1] < 8 &&
                     !c[ p.second + moves[i][1] ][ p.first + moves[i][0] ]) {
                    q.push( make_pair( p.first + moves[i][0], p.second + moves[i][1] ) );
                    c[ p.second + moves[i][1] ][ p.first + moves[i][0] ] = 1;
```

```
                }
            }
        }
        while(!q.empty()) q.pop();

        printf("To get from %c%d to %c%d takes %d knight moves.\n", a, ia+1, b, ib+1,
            nivel);
    }

    return 0;
}
```

17.4 558: Wormholes

Listing 17.4: —: Wormholes

```
#include <cstdio>
#include <queue>

typedef struct arc {
    int u, v, p;

    arc(int a, int b, int c): u(a), v(b), p(c) {};
    arc() {};
};

#define max 1002
#define inf 999999

using namespace std;

arc g[max][max];
int d[max];
int dist[max];
queue<arc> q;
int nv;

int main() {
    int c, n, m;
    int x, y, t;
    bool cn;
    arc a, adj;

    scanf("%d ", &c);
    while (c--) {
        scanf("%d %d ", &n, &m);

        for (int i = 0; i < n; i++) {
            dist[i] = inf; d[i] = 0;
        }

        for (int i = 0; i < m; i++) {
            scanf("%d %d %d ", &x, &y, &t);
            g[x][d[x]++] = arc(x, y, t);
        }

        q.push(arc(0, 0, 0));
        q.push(arc(-1, -1, 0));
```

```
        nv = 0; cn = false; dist[0] = 0;

        while (!q.empty()) {
            a = q.front(); q.pop();
            if (a.u == -1) {
                if (++nv > n) {cn = true; break;}
                q.push(arc(-1, -1, 0));
                //printf("mudou! nivel %d\n", nv);
            } else

            for (int i = 0; i < d[a.v]; i++) {
                adj = g[a.v][i];
                if (dist[adj.v] > dist[a.v] + adj.p) {
                    dist[adj.v] = dist[a.v] + adj.p;
                    q.push(arc(a.v, adj.v, dist[adj.v]));
                }
            }
        }

        while (!q.empty()) q.pop();

        cn = false;
        for (int i = 0; i < n; i++) {
            for (int k = 0; k < d[i]; k++)
                if (dist[i] + g[i][k].p < dist[g[i][k].v])
                    cn = true;
        }

        if (cn)
            printf("possible\n");
        else printf("not possible\n");
    }
}
```

17.5 10006: Carmichael Numbers

Listing 17.5: 10006: Carmichael Numbers

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

#define MAX 65010

int vetorPrimo[MAX];


long long BigMod(long long b, long long p, long long m) {
    if(p == 0)
        return 1;
    if(p == 1)
        return b;

    if(p % 2 == 0)
        return ((long long)pow((BigMod(b, p / 2, m) % m), 2) % m);
    else
        return ((BigMod(b, p - 1, m) % m) * (b % m)) % m;
}

```

```

void Crivo() {
    int raiz;
    int i, j;

    for (i=0; i < MAX; i++) {
        vetorPrimo[i] = 1;
    }
    vetorPrimo[1] = 0;
    j = 2;
    for (i=2; i*j < MAX; i++) {
        vetorPrimo[i*j] = 0;
    }

    raiz = sqrt(MAX) + 1;
    for (i=3; i <= raiz; i+=2) {
        for (j=2; j*i < MAX; j++) {
            vetorPrimo[j*i] = 0;
        }
    }
    return ;
}


int main (void) {
    long long x;
    int i, j, f;

    Crivo();
    while (scanf("%lld", &x) != EOF && x) {
        i = x;
        if (!vetorPrimo[i]) {
            f = 1;
            for (j=2; j < i; j++) {
                if (BigMod(j,x,x) != j) {
                    printf("%lld is normal.\n", x);
                    f = 0;
                    break;
                }
            }
            if (f) {
                printf("The number %lld is a Carmichael number.\n", x);
            }
        }
        else {
            printf("%lld is normal.\n", x);
        }
    }

    return 0;
}
```

17.6 10034: Freckles

Listing 17.6: 10034: Freckles

```
#include <cstdio>
#include <queue>
```

```
#include <cstring>
#include <cmath>

// Freckles - 10034
// AGM - Prim

#define ld long double

typedef struct pt {
    ld x, y;
    long double d;

    pt(): x(0), y(0), d(0) {};
    pt(ld a, ld b, ld c): x(a), y(b), d(c) {};

    bool operator<(const struct pt &a) const { return  a.d < d; }
};

#define di pair<double, int>

pt cp[110];
int v[110];

#define sqr(x) ((x)*(x))

using namespace std;

int main() {
    int n, a, b, k;
    int t;
    priority_queue<di, vector<di>, greater<di> > q;
    long double s, d;
    int tmp;

    scanf("%d ", &t);

    while (t--) {
        scanf("%d ", &n);

        for (int i = 0; i < n; i++)
            scanf("%llf %llf ", &cp[i].x, &cp[i].y);

        s = 0;
        q.push(make_pair(0, 0));
        memset(v, 0, sizeof(v));

        k = 0;

        while (!q.empty() && (k < n)) {
            tmp = q.top().second;
            d = q.top().first;
            q.pop();
            if (v[tmp]) continue;

            s += d;
            k++;

            v[tmp] = 1;

```

```
            for (int i = 0; i < n; i++) {
                if (!v[i]) {
                    d = sqrt(sqr(cp[tmp].x - cp[i].x) + sqr(cp[tmp].y - cp[i].y));
                    q.push(make_pair(d, i));
                }
            }
        }

        printf("%.2llf\n", s);
        if (t != 0) printf("\n");
        while (!q.empty())q.pop();
    }

    return 0;
}
```

17.7 10130: Supersale

Listing 17.7: 10130: Supersale

```
#include <cstdio>

// UVa 10130 - Supersale
// Problema da mochila, cada item pode ser utilizado apenas uma vez

inline int max(int a, int b) {return (a>b)?a:b;}

int main() {
    int t, n, p[1010], w[1010], K[1010][33], g, maxW, s;
    scanf("%d ", &t);
    while (t--) {
        scanf("%d", &n);
        for (int i = 1; i <= n; i++)
            scanf("%d %d ", &p[i], &w[i]);

        scanf("%d ", &g); s = 0;
        for (int r = 0; r < g; r++) {
            scanf("%d", &maxW);
            for (int i = 0; i < n; i++) K[i][0] = 0;
            for (int i = 0; i < maxW; i++) K[0][i] = 0;

            for (int i = 1; i <= n; i++)
                for (int j = 1; j <= maxW; j++)
                    if (w[i] > j)
                        K[i][j] = K[i-1][j];
                    else
                        K[i][j] = max( K[i-1][j], K[i-1][j-w[i]] + p[i] );

            s += K[n][maxW];
        }
        printf("%d\n", s);
    }

    return 0;
}
```

17.8 10173: Smallest Bounding Rectangle

Listing 17.8: 10173: Smallest Bouding Rectangle

```
#include <cstdio>
#include <cmath>
#include <algorithm>
#include <cstring>

/*
   UVa 10173 - Smallest Bounding Rectangle
   Algoritmo O(n^2), vai girando o convex hull
 */

using namespace std;

int cmp(long double a, long double b = 0.0) {
    if (fabs(a-b) < 1e-10) return 0;
    return (a<b)?-1:1;
}

struct pt {
    long double x, y;
    pt() {}
    pt(long double x, long double y): x(x), y(y) {}
    long double operator%(const pt& a) { return x*a.y-y*a.x; }
    pt operator-(const pt &a) const { return pt(x-a.x, y-a.y); }
    bool operator<(const pt &a) const {
        return (cmp(x, a.x) < 0) || (cmp(x, a.x) == 0 && cmp(y, a.y) < 0); }
    long double operator*(const pt& a) { return x*a.x+y*a.y; }
};

pt pivo;

bool cmp_ang(const pt &a, const pt &b) {
    int v = cmp((a-pivo)%(b-pivo));
    return v > 0 || v == 0 && cmp((a-pivo)*(a-pivo), (b-pivo)*(b-pivo)) < 0;
}

int area(pt a, pt b, pt c) { return cmp((b-a)%(c-a)); }
long double cos_ang(pt a, pt b) {
    return (a*b)/(sqrt(a*a)*sqrt(b*b));
}
long double sin_ang(pt a, pt b) {
    return fabs(a%b)/(sqrt(a*a)*sqrt(b*b));
}

int main() {
    int n;
    pt v[1010];
    pt h[1010];
    int top, k;

    while (1) {
        scanf("%d", &n);
        if (n == 0) break;

        for (int i = 0; i < n; i++)
            scanf("%llf %llf", &v[i].x, &v[i].y);
```

```
        pivo = *min_element(v, v+n);
        sort(v, v+n, cmp_ang);
        for (k = n-2; k >= 0 && area(v[0], v[n-1], v[k]) == 0; k--);
        reverse(v+k+1, v+n);
        top = 0;
        for (int i = 0; i < n; i++) {
            while (top > 1 && area(h[top-2], h[top-1], v[i]) <= 0) top--;
            h[top++] = v[i];
        }
        if (top > 1 && area(h[top-2], h[top-1], v[0]) == 0) top--;

        int pmin[4];
        long double tcos, tsin, area = 1.0/0.0;

        for (int j = 0; j < top+1; j++) {
            memset(pmin, 0, sizeof(pmin));
            for (int i = 1; i < top; i++) {
                if (cmp(h[i].y, h[pmin[0]].y) < 0) pmin[0] = i;
                if (cmp(h[i].x, h[pmin[1]].x) > 0) pmin[1] = i;
                if (cmp(h[i].y, h[pmin[2]].y) > 0) pmin[2] = i;
                if (cmp(h[i].x, h[pmin[3]].x) < 0) pmin[3] = i;
            }

            if (cmp((h[pmin[1]].x-h[pmin[3]].x)*(h[pmin[2]].y-h[pmin[0]].y), area) < 0)
                area = (h[pmin[1]].x-h[pmin[3]].x)*(h[pmin[2]].y-h[pmin[0]].y);

            tcos = cos_ang(h[(pmin[0]+1)%top]-h[pmin[0]], pt(1,0));
            tsin = -sin_ang(h[(pmin[0]+1)%top]-h[pmin[0]], pt(1,0));
            if (cmp(tcos, 1) == 0) { // se ja esta alinhado, pega proximo ang
                pmin[0] = (pmin[0]+1)%top;
                tcos = cos_ang(h[(pmin[0]+1)%top]-h[pmin[0]], pt(1,0));
                tsin = -sin_ang(h[(pmin[0]+1)%top]-h[pmin[0]], pt(1,0));
            }

            for (int i = 0; i < top; i++) {
                long double x, y;
                x = h[i].x*tcos - h[i].y*tsin;
                y = h[i].x*tsin + h[i].y*tcos;
                h[i].x = x; h[i].y = y;
            }
        }

        printf("%.4llf\n", area);
    }
    return 0;
}
```

17.9 10194: Football (aka Soccer)

Listing 17.9: 10194: Football (aka Soccer)

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


```

```c
#define MAX_TORNEIOS        1005
#define MAX_TORNEIO_NOME    105
#define MAX_TIMES           35
#define MAX_TIME_NOME       35
#define MAX_ENTRADA         205

typedef struct _time {
    char    nome[MAX_TIME_NOME];
    int     pontuacao;
    int     partidas;
    int     vitorias;
    int     empates;
    int     derrotas;
    int     golsFeitos;
    int     golsSofridos;
} time;


int compare_function(const void *a, const void *b) {
    time *x = (time *) a;
    time *y = (time *) b;

    if(x->pontuacao > y->pontuacao)
        return -1;
    else if(x->pontuacao < y->pontuacao)
        return 1;
    else {
        if(x->vitorias > y->vitorias)
            return -1;
        else if(x->vitorias < y->vitorias)
            return 1;
        else {
            if(x->golsFeitos - x->golsSofridos > y->golsFeitos - y->golsSofridos)
                return -1;
            else if(x->golsFeitos - x->golsSofridos < y->golsFeitos - y->golsSofridos)
                return 1;
            else {
                if(x->golsFeitos > y->golsFeitos)
                    return -1;
                else if(x->golsFeitos < y->golsFeitos)
                    return 1;
                else {
                    if(x->partidas < y->partidas)
                        return -1;
                    else if(x->partidas > y->partidas)
                        return 1;
                    else {
                        if(strcasecmp(x->nome, y->nome) < 0)
                            return -1;
                        else if(strcasecmp(x->nome, y->nome) > 0)
                            return 1;
                    }
                }
            }
        }
    }

    return 0;
}
```

```


int main (void) {
    char    nomeTorneio[MAX_TORNEIO_NOME];
    int     quantidadeTorneios;
    int     quantidadeTimes;
    int     quantidadePartidas;
    int     i, j, k, l;
    time    times[MAX_TIMES];

    char    partida[MAX_ENTRADA];
    char    nomeAuxiliar1[MAX_TIME_NOME], nomeAuxiliar2[MAX_TIME_NOME];
    int     gol1, gol2, ind1, ind2;

    scanf("%d\n", &quantidadeTorneios);
    for(i=0; i < quantidadeTorneios; i++) {
        fgets(nomeTorneio, MAX_TORNEIO_NOME, stdin);

        scanf("%d\n", &quantidadeTimes);
        for(j=0; j < quantidadeTimes; j++) {
            fgets(times[j].nome, MAX_TIME_NOME, stdin);
            times[j].nome[strlen(times[j].nome) - 1] = '\0';
            times[j].pontuacao = 0;
            times[j].partidas = 0;
            times[j].vitorias = 0;
            times[j].empates = 0;
            times[j].derrotas = 0;
            times[j].golsFeitos = 0;
            times[j].golsSofridos = 0;
        }

        scanf("%d\n", &quantidadePartidas);
        for(j=0; j < quantidadePartidas; j++) {
            fgets(partida, MAX_ENTRADA, stdin);
            k = 0;
            l = 0;
            while(partida[k] != '#') {
                nomeAuxiliar1[l] = partida[k];
                k++;
                l++;
            }
            nomeAuxiliar1[l] = '\0';
            k++;
            gol1 = 0;
            while(partida[k] != '@') {
                gol1 = gol1*10 + partida[k] - 48;
                k++;
            }
            k++;
            gol2 = 0;
            while(partida[k] != '#') {
                gol2 = gol2*10 + partida[k] - 48;
                k++;
            }
            k++;
            l = 0;
            while(partida[k] != '\n' && partida[k] != '\0') {
                nomeAuxiliar2[l] = partida[k];
                k++;
```

```
                l++;
            }
            nomeAuxiliar2[l] = '\0';

            ind1 = 0;
            ind2 = 0;
            k = 0;
            while(strcmp(times[k].nome, nomeAuxiliar1) != 0)
                k++;
            ind1 = k;

            k = 0;
            while(strcmp(times[k].nome, nomeAuxiliar2) != 0)
                k++;
            ind2 = k;

            times[ind1].partidas++;
            times[ind1].golsFeitos += gol1;
            times[ind1].golsSofridos += gol2;

            times[ind2].partidas++;
            times[ind2].golsFeitos += gol2;
            times[ind2].golsSofridos += gol1;

            if(gol1 > gol2) {
                times[ind1].vitorias++;
                times[ind2].derrotas++;
                times[ind1].pontuacao += 3;
            }
            else {
                if(gol1 < gol2) {
                    times[ind2].vitorias++;
                    times[ind1].derrotas++;
                    times[ind2].pontuacao += 3;
                }
                else {
                    times[ind1].empates++;
                    times[ind2].empates++;
                    times[ind1].pontuacao += 1;
                    times[ind2].pontuacao += 1;
                }
            }
        }

        qsort(times, quantidadeTimes, sizeof(time), compare_function);

        /*  impressao   */
        printf("%s", nomeTorneio);
        for(j=0; j < quantidadeTimes; j++) {
            printf("%d) %s %dp, %dg (%d-%d-%d), %dgd (%d-%d)\n", j+1, times[j].nome,
                times[j].pontuacao, times[j].partidas, times[j].vitorias, times[j].
                empates, times[j].derrotas, times[j].golsFeitos - times[j].
                golsSofridos, times[j].golsFeitos, times[j].golsSofridos);
        }

        if(i < quantidadeTorneios - 1)
            printf("\n");
    }

```

```
    return 0;
}
```

17.10 10480: Sabotage

Listing 17.10: 10480: Sabotage

```
#include <cstdio>
#include <vector>
#include <list>
#include <cfloat>
#include <utility>
#include <queue>

using namespace std;

// Sabotage - Corte minimo

#define maxv 110
#define maxa (110*110)
#define inf (999999)

typedef pair<double, int> arc;

vector<list<int> > adj(maxv);
vector<int> dest;
vector<double> custo;
vector<int> cap;
vector<int> fluxo;

vector<double> dist(maxv);
vector<int> ant(maxv);
vector<int> ar_ant(maxv);
vector<bool> v(maxv);
vector<bool> v2(maxv);

priority_queue<arc> q;

int n, m;

bool bfsres(int ini, int fim) {
    queue<int> q;
    int x, a;

    for (int i = 0; i < maxv; i++)
        v[i] = false;

    q.push(ini);
    v[ini] = 1;
    while (!q.empty()) {
        x = q.front();
        q.pop();

        if (x == fim) return true;
        for (list<int>::iterator i = adj[x].begin(); i != adj[x].end(); i++) {
            a = dest[*i];
            if (!v[a] && (cap[*i] - fluxo[*i] > 0)) {
                v[a] = 1;
```

```
                q.push(a);
                ant[a] = x;
                ar_ant[a] = *i;
            }
        }
    }

    return false;
}

int edkarp(int ini, int fim) {
    int fmax, v, a, capres;

    for (int i = 0 ; i < maxa; i++)
        fluxo[i] = 0;

    fmax = 0;
    while (bfsres(ini, fim)) {
        v = fim;
        a = ar_ant[fim];
        capres = cap[a] - fluxo[a];
        while (ant[v] != ini) {
            v = ant[v];
            a = ar_ant[v];
            if (capres > cap[a] - fluxo[a]) capres = cap[a] - fluxo[a];
        }

        fmax += capres;

        v = fim;
        while (v != ini) {
            a = ar_ant[v];
            fluxo[a] += capres;
            fluxo[a ^ 0x1] -= capres;
            v = ant[v];
        }
    }

    return fmax;
}

void dfs(int x) {
    for (list<int>::iterator i = adj[x].begin(); i != adj[x].end(); i++)
        if ((v2[dest[*i]] == false) && (v[dest[*i]] == true)) {
            v2[dest[*i]] = true;
            dfs(dest[*i]);
        } else if (v[dest[*i]] == false)
            printf("%d %d\n", x, dest[*i]);
}

int main() {
    int a, b, c;

    dest.reserve(maxa);
    custo.reserve(maxa);
    cap.reserve(maxa);
    fluxo.reserve(maxa);

    while (1) {
```

```
        scanf("%d", &n);
        if (n == 0) break;
        scanf("%d ", &m);

        dest.clear();
        custo.clear();
        cap.clear();
        fluxo.clear();

        for (int i = 1; i <= n; i++) {
            adj[i].clear();
            v2[i] = false;
        }

        for (int i = 0; i < m; i++) {
            scanf("%d %d %d", &a, &b, &c);

            adj[a].push_back(dest.size());
            dest.push_back(b);
            cap.push_back(c);

            adj[b].push_back(dest.size());
            dest.push_back(a);
            cap.push_back(c);
        }

        edkarp(1, 2);

        v2[1] = true;
        dfs(1);
        printf("\n");
    }

    return 0;
}
```

17.11 10679: I Love Strings!

Listing 17.11: 10679: I Love Strings!

```
#include <cstdio>
#include <queue>
#include <set>
#include <algorithm>
#include <list>

using namespace std;

#define maxsigma 255
#define maxpad 1001

struct trie {
    trie* g[maxsigma];
    list<int> o;
    char c;
    trie *f;

    trie(char c = 0): c(c) {
```

```
        for (int i = 0; i < maxsigma; i++) g[i] = NULL;
        f = NULL;
    }

    void free() {
        for (int i = 0; i < maxsigma; i++)
            if (g[i]) { if (g[i] != this) {g[i]->free(); delete g[i];} g[i] = NULL; }
        o.clear();
    }
};

trie raiz;

void insere(char *s, int i) {
    trie *p = &raiz;
    while (*s) {
        if (!p->g[*s])
            p->g[*s] = new trie(*s);
        p = p->g[*s];
        s++;
    }
    p->o.push_back(i);
}

void falha() {
    queue<trie*> q;
    trie *x, *v, *u;
    for (int i = 0; i < maxsigma; i++)
        if (raiz.g[i] != NULL) {
            raiz.g[i]->f = &raiz;
            q.push(raiz.g[i]);
        } else
            raiz.g[i] = &raiz;

    while (!q.empty()) {
        x = q.front(); q.pop();
        for (int i = 0; i < maxsigma; i++)
            if ((u = x->g[i]) != NULL) {
                q.push(u);
                v = x->f;
                while (v->g[i] == NULL) v = v->f;
                u->f = v->g[i];
                u->o.insert(u->o.begin(), u->f->o.begin(), u->f->o.end());
            }
    }
}

char txt[100100];
char pads[1010][1010];
char usado[1010];

int main() {
    int k, q, l;
    trie* state;

    scanf("%d ", &k);
    while (k--) {
        scanf("%s %d", txt, &q);

```

```
        for (int i = 0; i < q; i++) {
            scanf("%s ", pads[i]);
            insere(pads[i], i);
        }
        falha();

        memset(usado, 0, sizeof(usado));
        state = &raiz;
        l = strlen(txt);
        for (int i = 0; i < l; i++) {
            while (state->g[txt[i]] == NULL) state = state->f;
            state = state->g[txt[i]];
            if (!state->o.empty())
                for (list<int>::iterator it = state->o.begin(); it != state->o.end();
                    it++)
                    usado[*it] = 1;
        }

        for (int i = 0; i < q; i++)
            if (usado[i]) printf("y\n");
            else printf("n\n");

        raiz.free();
    }

    return 0;
}
```

17.12 11492: Babel

Listing 17.12: 11492: Babel

```
#include <cstdio>
#include <cstdlib>
#include <map>
#include <queue>
#include <set>
#include <string>
#include <cstring>
#include <iostream>

using namespace std;

#define max 5010
#define infinito 9999999

typedef struct arc {
    public:
        int u, v;
        int w;
        char l;

        arc(): u(0), v(0), w(0), l('\0') {}
        arc(int t, int a, int b, char c): u(t), v(a), w(b), l(c) {};

        bool operator<(const arc& a) const {
            return w > a.w;
        }
```

```
};

struct compara {
    bool operator()(const arc& a, const arc& b) {
        return a.w < b.w;
    }
};

arc g[max][10000];
int d[max];

int main() {
    int m, cont, hs, he;
    map<string, int> h;
    //set<arc, compara> q;
    priority_queue<arc> q;
    int dist[max];
    char prev[max];
    char v[max];
    arc x, tmp;
    string s, e, l;
    bool conflito;
    int min;

    while (1) {
        scanf("%d ", &m);
        if (m == 0) break;

        memset(d, 0, sizeof(d));
        h.clear();

        cin>>s;
        h[s] = 0;
        cin>>s;
        h[s] = 1;

        cont = 2;
        while (m--) {
            cin >> s >> e >> l;
            if (h.find(s) == h.end()) {
                hs = cont;
                h[s] = cont++;
            } else
                hs = h[s];

            if (h.find(e) == h.end()) {
                he = cont;
                h[e] = cont++;
            } else
                he = h[e];

            g[hs][d[hs]++] = arc(hs, he, l.length(), l[0]);
            g[he][d[he]++] = arc(he, hs, l.length(), l[0]);
        }

        conflito = true;
        min = infinito;
        while (conflito) {
            conflito = false;
```

```
            //q.clear();
            while (!q.empty()) q.pop();

            for (int i = 0; i < max; i++)
                dist[i] = infinito;

            memset(prev, 0, sizeof(prev));
            memset(v, 0, sizeof(v));

            tmp.u = tmp.v = tmp.l = tmp.w = 0;
            q.push(tmp);
            dist[0] = 0;

            while (!q.empty()) {
                //x = *q.begin();
                //q.erase(q.begin());
                x = q.top();
                q.pop();

                if (v[x.v]) continue;
                v[x.v] = 1;
                //printf("analisando %d\n", x.v);
                if (x.v == 1) break;

                for (int i = 0; i < d[x.v]; i++) {
                    tmp = g[x.v][i];

                    if ((dist[tmp.v] > x.w + tmp.w) && (tmp.l != prev[x.v])) {
                        dist[tmp.v] = x.w + tmp.w;
                        prev[tmp.v] = tmp.l;
                        q.push(arc(x.v, tmp.v, x.w+tmp.w, tmp.l));
                    }
                    if ((tmp.l == prev[x.v]) && !conflito && !v[tmp.v]) {
                        for (int t = 0; t < d[x.u]; t++) {
                            //printf("  adj %c %d %d - %c %d %d\n", g[x.u][t].l, g[x.u][t].v, g[x.u][t].w, tmp.l, tmp.v, tmp.w);
                            if (g[x.u][t].l == tmp.l) {
                                conflito = true;
                                //printf("    remove %c %d %d %c\n", g[x.u][t].l, g[x.u][t].v, g[x.u][t].w, tmp.l);
                                swap(g[x.u][t].u, g[x.u][--d[x.u]].u);
                                swap(g[x.u][t].l, g[x.u][d[x.u]].l);
                                swap(g[x.u][t].v, g[x.u][d[x.u]].v);
                                swap(g[x.u][t].w, g[x.u][d[x.u]].w);
                            }
                        }
                    }
                }
            }

            if (min > dist[1]) min = dist[1];
            /*while (!q.empty()) {
                x = *q.begin();
                q.erase(q.begin());
            }*/
            //printf("redo!\n");
        }
        //printf("fim\n");

```

```
        if (min == infinito)
            printf("impossivel\n");
        else
            printf("%d\n", min);

    }
    return 0;
}
```

17.13 11659: Informants

Listing 17.13: 11659: Informants

```
nclude <stdio.h>

int m[20][20];
int r[20];
int i, a, b, c, x, y;

int maior(int a, int b) {
    return a > b ? a : b;
}

int back(int atual, int v[]) {

    int v1[20];
    int v2[20];
    int t, max, erro;

    if (atual < i) {
        for (x = 0; x < i; x++) {
            v1[x] = v[x];
            v2[x] = v[x];
        }
        v1[atual] = -1;
        v2[atual] = 1;
        max = back(atual + 1, v1);
        max = maior(back(atual + 1, v2), max);
    }
    else {
        erro = 0;
        for (x = 0; !erro && x < i; x++) {
            if (v[x] == 1) {
                for (y = 0; y < i; y++) {
                    if (m[x][y] * v[y] == -1 || m[x][y] == -2) {
                        erro = 1;
                    }
                }
            }
        }
        if (erro) {
            max = 0;
        }
        else {
            y = 0;
            for (x = 0; x < i; x++) {
                if (v[x] == 1) {
                    y++;
```

```
                }
            }
            max = y;
        }
    }

    return max;
}

int main(void) {

    while (scanf("%d %d\n", &i, &a) && (i != 0 || a != 0)) {
        for (x = 0; x < i; x++) {
            r[x] = 0;
            for (y = 0; y < i; y++) {
                m[x][y] = 0;
            }
        }

        if (a > 0) {
            while (a--) {
                scanf("%d %d\n", &b, &c);
                if (c < 0) {
                    if (m[b-1][-(c+1)] == 1 || m[b-1][-(c+1)] == -2) {
                        m[b-1][-(c+1)] = -2;
                    }
                    else {
                        m[b-1][-(c+1)] = -1;
                    }
                }
                else {
                    if (m[b-1][c-1] == -1 || m[b-1][c-1] == -2) {
                        m[b-1][c-1] = -2;
                    }
                    else {
                        m[b-1][c-1] = 1;
                    }
                }
            }
            printf("%d\n", back(0, r));
        }
        else {
            printf("%d\n", i);
        }
    }

    return 0;
}
```

17.14 11682: Shift Register

Listing 17.14: 11682: Shift Register

```
#include <cstdio>
#include <map>
#include <cmath>

/** UVa 11682
```

```
 * Shift Register
 */

using namespace std;

#define N (1<<POT)

int main() {
    int n, t, a, x, xx, p;
    int i, f;
    unsigned long long int tot, max, it;
    int msk, POT;
    char M[33][33], F[33][33], Mn[33][33], T[33][33], ini[33], fim[33], res[33], aux
        [33];
    map<int, int> m;

    while (1) {
        scanf("%d %d ", &n, &t);
        if (n == 0 && t == 0) break;

        POT = n/2;

        memset(M, 0, sizeof(M));
        memset(F, 0, sizeof(F));

        scanf("%*d ");
        for (int i = 0; i < t-1; i++) {
            scanf("%d ", &a);
            M[n-1][n-a] = 1;
            F[0][n-1-a] = 1;
        }

        F[0][n-1] = 1;
        M[n-1][0] = 1;

        for (int i = 0; i < n-1; i++) {
            M[i][i+1] = 1;
            F[i+1][i] = 1;
        }

        scanf("%x %x ", &i, &f);
        x = i; xx = f;
        for (int it = 0; it < n; it++) {
            ini[n-1-it] = x & 1;
            x >>= 1;
            fim[n-1-it] = xx & 1;
            xx >>= 1;
        }

        m.clear();

        m[i] = 0;
        memcpy(res, ini, sizeof(res));

        for (int it = 1; it <= N; it++) {
            for (int i = 0; i < n; i++) {
                aux[i] = 0;
                for (int j = 0; j < n; j++)
                    aux[i] += F[i][j]*res[j];
```

```
                aux[i] = aux[i] & 1;
            }
            if (it == N)  break;
            memcpy(res, aux, sizeof(res));

            x = 0;
            for (int i = 0; i < n; i++)
                if (res[i]) x |= 1<<n-1-i;

            if (m.find(x) != m.end()) break;

            m[x] = it;
            if (x == f) break;
        }

        memcpy(Mn, M, sizeof(Mn));
        for (int it = 0; it < POT; it++) {
            for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++) {
                    T[i][j] = 0;
                    for (int k = 0; k < n; k++) T[i][j] += Mn[i][k]*Mn[k][j];
                    T[i][j] = T[i][j] & 1;
                }
            memcpy(Mn, T, sizeof(Mn));
        }

        memcpy(res, fim, sizeof(res));
        x = f;
        max = 1LL<<n; tot = 0; it = 0;
        while (m.find(x) == m.end() && (tot <= max)) {
            for (int i = 0; i < n; i++) {
                aux[i] = 0;
                for (int j = 0; j < n; j++)
                    aux[i] += Mn[i][j]*res[j];
                aux[i] = aux[i] & 1;
            }
            memcpy(res, aux, sizeof(aux));

            x = 0;
            for (int i = 0; i < n; i++)
                if (res[i]) x|= 1<<n-1-i;

            tot += N;
            it++;
        }

        if (tot <= max)
            printf("%lld\n", m[x]+tot);
        else
            printf("*\n");
    }

}
```

17.15 4741: Blur, ICPC Archive da regional Africana

Listing 17.15: 4741: Blur

```
#include <cstdio>
#include <cmath>

#define MAX 110

double matrix[2 * MAX][MAX];

int abs(int a) {
    return (a < 0) ? -a : a;
}

void printMatrix(double matrix[MAX][MAX], int n, int m) {

    int i, j;

    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            printf("%8.2lf ", matrix[i][j]);
        }
        printf("\n");
    }
    printf("\n");

}


void gaussElimination(double matrix[MAX][MAX], int n, int m) {

    int i, j, k, l, max;
    double aux;

    i = 0;
    j = 0;
    while (i < n && j < n) {
        max = i;
        for (k = i + 1; k < n; k++) {
            if (fabs(matrix[k][j]) > fabs(matrix[max][j])) {
                max = k;
            }
        }
        if (matrix[max][j] != 0) {
            for (k = 0; k < n + m; k++) {
                aux = matrix[i][k];
                matrix[i][k] = matrix[max][k];
                matrix[max][k] = aux;
            }
            aux = matrix[i][j];
            for (k = 0; k < n + m; k++) {
                matrix[i][k] /= aux;
            }
            for (k = i + 1; k < n; k++) {
                aux = matrix[k][j];
                for (l = 0; l < n + m; l++) {
                    matrix[k][l] -= aux * matrix[i][l];
                }
            }
            i++;
        }
        j++;
```

```
    }

    for (k = 0; k < m; k++) {
        for (i = n - 1; i >= 0; i--) {
            aux = 0;
            for (j = n - 1; j > i; j--) {
                aux += matrix[i][j] * matrix[j][k + n];
            }
            matrix[i][k + n] -= aux;
            matrix[i][k + n] /= matrix[i][i];
        }
    }

    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            matrix[i][j] = matrix[i][j + n];
        }
    }

}

int x(int a, int m) {
    return a / m;
}

int y(int a, int m) {
    return a % m;
}

int main(void) {

    int n, m, dist;
    int i, j, k;
    double image[MAX];
    double unblur[MAX];
    int cases = 0;

    while (scanf("%d %d %d", &m, &n, &dist) && (n != 0 || m != 0 || dist != 0)) {
        if (cases) {
            printf("\n");
        }
        else {
            cases = 1;
        }

        for (i = 0; i < n * m; i++) {
            scanf("%lf", &image[i]);
        }

        for (i = 0; i < n * m; i++) {
            for (j = 0; j < n * m; j++) {
                if (abs(x(i, m) - x(j, m)) + abs(y(i, m) - y(j, m)) <= dist) {
                    matrix[i][j] = 1;
                }
                else {
                    matrix[i][j] = 0;
                }
            }
        }
```

```
        for (i = 0; i < n * m; i++) {
            k = 0;
            for (j = 0; j < n * m; j++) {
                if (matrix[i][j]) {
                    k++;
                }
            }
            unblur[i] = k;
        }

        for (i = 0; i < n * m; i++) {
             matrix[i][n * m] = unblur[i] * image[i];
        }

        gaussElimination(matrix, n * m, 1);
        for (i = 0; i < n; i++) {
            for (j = 0; j < m; j++) {
                printf("%8.2lf", matrix[i * m + j][0]);
            }
            printf("\n");
        }

    }

}
```

Listing 17.16: 4741: Blur + Matriz

```
#include <cstdio>
#include <cmath>

#define MAX 110

/* 2 * MAX para resolver problemas entre duas matrizes N x N */
long double matrix[2 * MAX][MAX];

void printMatrix(long double matrix[MAX][MAX], int n, int m) {

    int i, j;

    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            printf("%5.2llf ", matrix[i][j]);
        }
        printf("\n");
    }
    printf("\n");

}

/*

d = detMatrix(matriz, N);

Calcula o determinante de uma matriz N x N

*/
```

```
long double detMatrix(long double matrix[MAX][MAX], int n) {

    long double det;
    int i, j, k, l;

    n = n - 1;
    det = matrix[0][0];
    for (k = 0; k < n; k++) {
        l = k + 1;
        for (i = l; i <= n; i++) {
            for (j = l; j <= n; j++) {
                matrix[i][j] = (matrix[k][k] * matrix[i][j] - matrix[k][j] * matrix[i][k]) / matrix[k][k];
            }
        }
        det *= matrix[k + 1][k + 1];
    }

    return det;
}

/*

gaussElimination(matriz, N, M);

Resolve um sistema A * B = C, onde A eh uma matriz N x N e B eh uma matriz N x M
Retorna o resultado na propria matriz sendo ela N x M

*/
void gaussElimination(double matrix[MAX][MAX], int n, int m) {

    int i, j, k, l, max;
    double aux;

    i = 0;
    j = 0;
    while (i < n && j < n) {
        max = i;
        for (k = i + 1; k < n; k++) {
            if (fabs(matrix[k][j]) > fabs(matrix[max][j])) {
                max = k;
            }
        }
        if (matrix[max][j] != 0) {
            for (k = 0; k < n + m; k++) {
                aux = matrix[i][k];
                matrix[i][k] = matrix[max][k];
                matrix[max][k] = aux;
            }
            aux = matrix[i][j];
            for (k = 0; k < n + m; k++) {
                matrix[i][k] /= aux;
            }
            for (k = i + 1; k < n; k++) {
                aux = matrix[k][j];
                for (l = 0; l < n + m; l++) {
                    matrix[k][l] -= aux * matrix[i][l];
                }
            }
```

```
            i++;
        }
        j++;
    }

    for (k = 0; k < m; k++) {
        for (i = n - 1; i >= 0; i--) {
            aux = 0;
            for (j = n - 1; j > i; j--) {
                aux += matrix[i][j] * matrix[j][k + n];
            }
            matrix[i][k + n] -= aux;
            matrix[i][k + n] /= matrix[i][i];
        }
    }

    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            matrix[i][j] = matrix[i][j + n];
        }
    }

}

/*

invertMatrix(matriz, N);

Usa a eliminacao de gauss para calcular a inversa de uma matriz.

*/
void invertMatrix(long double matrix[MAX][MAX], int n) {

    int i, j, k, l, max;
    long double aux;

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            matrix[i][j + n] = 0;
        }
    }

    for (i = 0; i < n; i++) {
        matrix[i][i + n] = 1;
    }

    gaussElimination(matrix, n, n);

}
```