ALGORITMOS
em linguagem C

Paulo Feofiloff

Instituto de Matemática e Estatística
Universidade de São Paulo

Campus/Elsevier

.

“Algoritmos em linguagem C”
Paulo Feofiloff
editora Campus/Elsevier, 2009

www.ime.usp.br/~pf/algoritmos-livro/

"Ciência da computação não é a ciência dos computadores, assim como a astronomia não é a ciência dos telescópios."

— E. W. Dijkstra

Leiaute

Bom leiaute

```
int Funcao (int n, int v[]) {
   int i, j;
   i = 0;
   while (i < n) {
      if (v[i] != 0)
         i = i + 1;
      else {
         for (j = i + 1; j < n; j++)
            v[j-1] = v[j];
         n = n - 1;
      }
   }
   return n;
}
```


Mau leiaute

int Funcao (int n, int v[]) {
  int i, j;
  i = 0;
  while (i < n) {
    if (v[i] != 0)
      i = i + 1;
    else {
      for (j = i + 1; j < n; j++)
        v[j-1] = v[j];
      n = n - 1;
    }
  }
  return n;
}

Use fonte de `espaçamento fixo`!

Péssimo leiaute

```
int Funcao ( int n,int v[] ){
   int i,j;
   i=0;
   while(i<n){
      if(v[i] !=0)
         i= i +1;
      else
      {
         for (j=i+1;j<n;j++)
            v[j-1]=v[j];
         n =n- 1;
      }
   }
   return n;
}
```

Seja consistente!

Um bom leiaute compacto

```
int Funcao (int n, int v[]) {
   int i, j;
   i = 0;
   while (i < n) {
      if (v[i] != 0)  i = i + 1;
      else {
         for (j = i + 1; j < n; j++) v[j-1] = v[j];
         n = n - 1; } }
   return n; }
```

Regras

Use as regras adotadas por todos os jornais, revistas e livros:

```
bla bla bla 
bla = bla
bla <= bla
bla; bla
bla) bla;
bla {
while (bla
if (bla
```


Leiaute enfeitado

```
int Função (int n, int v[]) {
   int i, j;
   i = 0;
   while (i < n) {
      if (v[i] != 0)
         i = i + 1;
      else {
         for (j = i + 1; j < n; j++)
            v[j-1] = v[j];
         n = n - 1;
      }
   }
   return n;
}
```

"Devemos mudar nossa atitude tradicional em relação à construção de programas.
Em vez de imaginar que nossa principal tarefa
é instruir o computador sobre o que ele deve fazer,
vamos imaginar que nossa principal tarefa é
explicar a seres humanos o que queremos que o computador faça."

— D. E. Knuth

Documentação

- documentação: o que um algoritmo faz
- código: como o algoritmo faz o que faz

Exemplo

```
/* A função abaixo recebe um número n >= 1 e um vetor v
 * e devolve o valor de um elemento máximo de v[0..n-1].
 ******************************************************/

int Max (int v[], int n) {
   int j, x = v[0];
   for (j = 1; j < n; j++)
      if (x < v[j]) x = v[j];
   return x;
}
```


Invariantes

Exemplo 1

```
int Max (int v[], int n) {
   int j, x;
   x = v[0];
   for (j = 1; j < n; j++)
      /* x é um elemento máximo de v[0..j-1] */
      if (x < v[j])  x = v[j];
   return x;
}
```


Exemplo 2

```
int Max (int v[], int n) {
   int j, x;
   x = v[0];
   for (j = 1; /* A */ j < n; j++)
      if (x < v[j])  x = v[j];
   return x;
}

/* a cada passagem pelo ponto A,
   x é um elemento máximo de v[0..j-1] */
```

> "A atividade de programação deve ser encarada
> como um processo de criação de obras de literatura,
> escritas para serem lidas."
>
> — D. E. Knuth

Recursão

"Para entender recursão,
é preciso primeiro entender recursão."
— folclore

"Ao tentar resolver o problema,
encontrei obstáculos dentro de obstáculos.
Por isso, adotei uma solução recursiva."
— um aluno

Problemas e suas instâncias

- instância de um problema = exemplo concreto do problema
- cada conjunto de dados de um problema define uma instância
- cada instância tem um tamanho

Exemplo

Problema: Calcular a média de dois números, digamos a e b.
Instância: Calcular a média de 123 e 9876.

Problemas que têm estrutura recursiva

Cada instância do problema
contém uma instância menor do mesmo problema.

Algoritmo recursivo

se a instância em questão é pequena
    resolva-a diretamente
senão
    reduza-a a uma instância menor do mesmo problema
    aplique o método à instância menor
    volte à instância original

Exemplo: Problema do máximo

Determinar o valor de um elemento máximo de um vetor $v[0\,..\,n{-}1]$.

- o tamanho de uma instância deste problema é n
- o problema só faz sentido quando $n \geq 1$

Solução recursiva

```
/* Ao receber v e n >= 1, esta função devolve
   o valor de um elemento máximo de v[0..n-1]. */

int MáximoR (int v[], int n) {
   if (n == 1)
      return v[0];
   else {
      int x;
      x = MáximoR (v, n - 1);
      if (x > v[n-1])
         return x;
      else
         return v[n-1];
   }
}
```


Outra solução recursiva

```
int Máximo (int v[], int n) {
   return MaxR (v, 0, n);
}

int MaxR (int v[], int i, int n) {
   if (i == n-1) return v[i];
   else {
      int x;
      x = MaxR (v, i + 1, n);
      if (x > v[i]) return x;
      else return v[i];
   }
}

/* A função MaxR recebe v, i e n tais que i < n
   e devolve o valor de um elemento máximo de v[i..n-1]. */
```

Vetores

Problema da busca

Dado x e vetor $v[0\,..\,n{-}1]$, encontrar um índice k tal que $v[k] = x$.

x 987

	0								$n{-}1$		
v	222	555	111	333	444	666	555	888	777	987	654

- o problema faz sentido com qualquer $n \geq 0$
- se $n = 0$, o vetor é vazio e essa instância não tem solução
- como indicar que não há solução?

Algoritmo de busca

Recebe um número x e um vetor $v[0\,..\,n-1]$ com $n \geq 0$
e devolve k no intervalo $0\,..\,n-1$ tal que $v[k] = x$.
Se tal k não existe, devolve -1.

```
int Busca (int x, int v[], int n) {
   int k;
   k = n - 1;
   while (k >= 0 && v[k] != x)
      k -= 1;
   return k;
}
```


Deselegante e/ou ineficiente!

```
   int k = n - 1, achou = 0;
   while (k >= 0 && achou == 0) {
      if (v[k] == x) achou = 1;
      else k -= 1;
   }
   return k;
```

```
   int k;
   if (n == 0) return -1;
   k = n - 1;
   while (k >= 0 && v[k] != x) k -= 1;
   return k;
```

Deselegante, ineficiente e/ou errado!

```
int k = 0;
int sol = -1;
for (k = n-1; k >= 0; k--)
   if (v[k] == x) sol = k;
return sol;
```

```
int k = n - 1;
while (v[k] != x && k >= 0)
   k -= 1;
return k;
```


Algoritmo recursivo de busca

Recebe x, v e $n \geq 0$ e devolve k tal que $0 \leq k < n$ e $v[k] = x$.
Se tal k não existe, devolve -1.

```
int BuscaR (int x, int v[], int n) {
   if (n == 0) return -1;
   if (x == v[n-1]) return n - 1;
   return BuscaR (x, v, n - 1);
}
```

Deselegante!

```
int feio (int x, int v[], int n) {
   if (n == 1) {
      if (x == v[0]) return 0;
      else return -1;
   }
   if (x == v[n-1]) return n - 1;
   return feio (x, v, n - 1);
}
```


Problema de remoção

Remover o elemento de índice k de um vetor $v[0\,..\,n{-}1]$.

Decisões de projeto:

- suporemos $0 \leq k \leq n-1$
- novo vetor fica em $v[0\,..\,n{-}2]$
- algoritmo devolve algo?

Algoritmo de remoção

Remove o elemento de índice k do vetor $v[0\,..\,n-1]$
e devolve o novo valor de n. Supõe $0 \leq k < n$.

```
int Remove (int k, int v[], int n) {
   int j;
   for (j = k; j < n-1; j++)
      v[j] = v[j+1];
   return n - 1;
}
```

- funciona bem mesmo quando $k = n - 1$ ou $k = 0$
- exemplo de uso: `n = Remove (51, v, n);`

Versão recursiva

```
int RemoveR (int k, int v[], int n) {
   if (k == n-1) return n - 1;
   else {
      v[k] = v[k+1];
      return RemoveR (k + 1, v, n);
   }
}
```

Problema de inserção

Inserir um novo elemento y entre as posições $k-1$ e k de um vetor $v[0\,..\,n{-}1]$.

Decisões de projeto:

- se $k = 0$ então insere no início
- se $k = n$ então insere no fim
- novo vetor fica em $v[0\,..\,n{+}1]$

Algoritmo de inserção

Insere y entre as posições $k-1$ e k do vetor $v[0\,..\,n{-}1]$ e devolve o novo valor de n. Supõe que $0 \leq k \leq n$.

```
int Insere (int k, int y, int v[], int n) {
   int j;
   for (j = n; j > k; j--)
      v[j] = v[j-1];
   v[k] = y;
   return n + 1;
}
```

- estamos supondo $n <$ N
- exemplo de uso: `n = Insere(51, 999, v, n);`

Versão recursiva

```
int InsereR (int k, int y, int v[], int n) {
   if (k == n) v[n] = y;
   else {
      v[n] = v[n-1];
      InsereR (k, y, v, n - 1);
   }
   return n + 1;
}
```


Problema de busca-e-remoção

Remover todos os elementos nulos de um vetor $v[0\,..\,n{-}1]$.

Algoritmo

Remove todos os elementos nulos de $v[0\,..\,n{-}1]$,
deixa o resultado em $v[0\,..\,i{-}1]$, e devolve o valor de i.

```
int RemoveZeros (int v[], int n) {
   int i = 0, j;
   for (j = 0; j < n; j++)
      if (v[j] != 0) {
         v[i] = v[j];
         i += 1;
      }
   return i;
}
```

Funciona bem mesmo em casos extremos:

- quando n vale 0
- quando $v[0\,..\,n{-}1]$ não tem zeros
- quando $v[0\,..\,n{-}1]$ só tem zeros

Invariantes

No início de cada iteração

- $i \le j$
- $v[0\,..\,i{-}1]$ é o resultado da remoção dos zeros do vetor $v[0\,..\,j{-}1]$ original

Mau exemplo: deselegante e ineficiente

```
int i = 0, j = 0;     /* "j = 0" é supérfluo */
while (i < n) {
   if (v[i] != 0) i += 1;
   else {
      for (j = i; j+1 < n; j++)   /* ineficiente */
         v[j] = v[j+1];           /* ineficiente */
      --n;
   }
}
return n;
```

Versão recursiva

```
int RemoveZerosR (int v[], int n) {
   int m;
   if (n == 0) return 0;
   m = RemoveZerosR (v, n - 1);
   if (v[n-1] == 0) return m;
   v[m] = v[n-1];
   return m + 1;
}
```


Endereços e ponteiros

Endereços

- os bytes da memória são numerados seqüencialmente
- o número de um byte é o seu endereço
- cada `char` ocupa 1 byte
- cada `int` ocupa 4 bytes consecutivos
- etc.
- cada objeto — `char`, `int`, `struct` etc. — tem um endereço
- o endereço de um objeto `x` é `&x`

Exemplo fictício

```
char c;
int i;
struct {int x, y;} ponto;
int v[4];
```

	endereços
c	*89421*
i	*89422*
ponto	*89426*
v[0]	*89434*
v[1]	*89438*
v[2]	*89442*

- `&i` vale *89422*
- `&v[3]` vale *89446*

Ponteiros

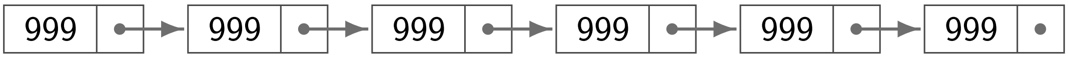
- ponteiro é um tipo de variável capaz de armazenar endereços
- se `p` = `&x` então dizemos "`p` aponta para `x`"
- se `p` é um ponteiro então `*p` é o valor do objeto apontado por `p`

representação esquemática

Exemplo: Um jeito bobo de fazer `j = i + 999`

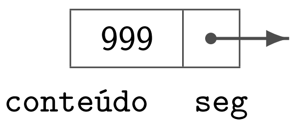
```
int j, i = 888;
int *p;
p = &i;
j = *p + 999;
```

Listas encadeadas



Estrutura de uma célula

```
struct cel {
   int          conteúdo;
   struct cel *seg;    /* seguinte */
};
```



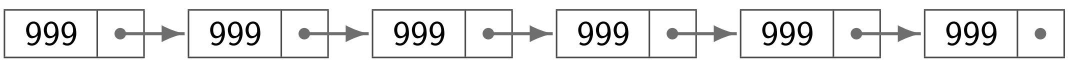
Células são um novo tipo de dados

```
typedef struct cel célula;
```

Definição de uma célula e de um ponteiro para célula

```
célula  c;
célula *p;
```

- conteúdo da célula: `c.conteúdo` `p->conteúdo`
- endereço da célula seguinte: `c.seg` `p->seg`



última célula da lista: `p->seg` vale `NULL`

Exemplos: imprime lista com e sem cabeça

O algoritmo imprime o conteúdo de uma lista `lst` sem cabeça.

```
void Imprima (célula *lst) {
   célula *p;
   for (p = lst; p != NULL; p = p->seg)
      printf ("%d\n", p->conteúdo);
}
```

Imprime o conteúdo de uma lista `lst` com cabeça.

```
void Imprima (célula *lst) {
   célula *p;
   for (p = lst->seg; p != NULL; p = p->seg)
      printf ("%d\n", p->conteúdo);
}
```


Algoritmo de busca

Recebe um inteiro x e uma lista `lst` com cabeça.
Devolve o endereço de uma célula que contém x
ou devolve `NULL` se tal célula não existe.

```
célula *Busca (int x, célula *lst) {
   célula *p;
   p = lst->seg;
   while (p != NULL && p->conteúdo != x)
      p = p->seg;
   return p;
}
```

Versão recursiva

```
célula *BuscaR (int x, célula *lst) {
   if (lst->seg == NULL)
      return NULL;
   if (lst->seg->conteúdo == x)
      return lst->seg;
   return BuscaR (x, lst->seg);
}
```


Algoritmo de remoção de uma célula

Recebe o endereço p de uma célula em uma lista
e remove da lista a célula p->seg.
Supõe que p $\neq$ NULL e p->seg $\neq$ NULL.

```
void Remove (célula *p) {
   célula *lixo;
   lixo = p->seg;
   p->seg = lixo->seg;
   free (lixo);
}
```

Algoritmo de inserção de nova célula

Insere uma nova célula em uma lista
entre a célula p e a seguinte (supõe $p \neq$ NULL).
A nova célula terá conteúdo y.

```
void Insere (int y, célula *p) {
   célula *nova;
   nova = malloc (sizeof (célula));
   nova->conteúdo = y;
   nova->seg = p->seg;
   p->seg = nova;
}
```


Algoritmo de busca seguida de remoção

Recebe uma lista lst com cabeça
e remove da lista a primeira célula que contiver x,
se tal célula existir.

```
void BuscaERemove (int x, célula *lst) {
   célula *p, *q;
   p = lst;
   q = lst->seg;
   while (q != NULL && q->conteúdo != x) {
      p = q;
      q = q->seg;
   }
   if (q != NULL) {
      p->seg = q->seg;
      free (q);
   }
}
```

Algoritmo de busca seguida de inserção

Recebe lista lst com cabeça e insere nova célula conteúdo y imediatamente antes da primeira que contiver x.
Se nenhuma célula contiver x, a nova célula será inserida no fim da lista.

```
void BuscaEInsere (int y, int x, célula *lst) {
   célula *p, *q, *nova;
   nova = malloc (sizeof (célula));
   nova->conteúdo = y;
   p = lst;
   q = lst->seg;
   while (q != NULL && q->conteúdo != x) {
      p = q;
      q = q->seg;
   }
   nova->seg = q;
   p->seg = nova;
}
```


Filas

Fila implementada em vetor

0		s						t		N-1
		111	222	333	444	555	666			

uma fila `f[s..t-1]`

Remove elemento da fila

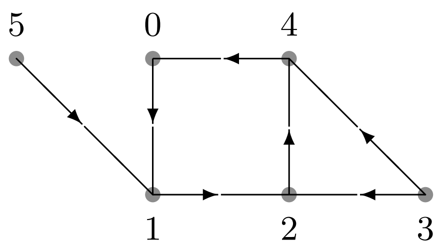
```
x = f[s++];                  /* x = f[s]; s += 1; */
```

Insere y na fila

```
f[t++] = y;                  /* f[t] = y; t += 1; */
```


Aplicação: distâncias em uma rede

	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0	1	0	0	0
2	0	0	0	0	1	0
3	0	0	1	0	1	0
4	1	0	0	0	0	0
5	0	1	0	0	0	0



	0	1	2	3	4	5
d	2	3	1	0	1	6

O vetor d dá as distâncias da cidade 3 a cada uma das demais.

Algoritmo das distâncias

Recebe matriz A que representa as interligações entre cidades $0, 1, \ldots, n-1$: há uma estrada de x a y se e somente se $\mathtt{A}[x][y] = 1$.
Devolve um vetor d tal que $d[x]$ é a distância da cidade o à cidade x.

```
int *Distâncias (int **A, int n, int o) {
   int *d, x, y;
   int *f, s, t;
   d = malloc (n * sizeof (int));
   for (x = 0; x < n; x++) d[x] = -1;
   d[o] = 0;
   f = malloc (n * sizeof (int));
   processo iterativo
   free (f);
   return d;
}
```


processo iterativo

```
   s = 0; t = 1; f[s] = o;    /* o entra na fila */
   while (s < t) {
      x = f[s++];    /* x sai da fila */
      for (y = 0; y < n; y++)
         if (A[x][y] == 1 && d[y] == -1) {
            d[y] = d[x] + 1;
            f[t++] = y;    /* y entra na fila */
         }
   }
```

Invariantes (antes de cada comparação "s < t")

1. para cada cidade v em $\mathtt{f}[0\,..\,\mathtt{t}-1]$
   existe um caminho de comprimento $d[v]$ de o a v
   cujas cidades estão todas em $\mathtt{f}[0\,..\,\mathtt{t}-1]$
2. para cada cidade v de $\mathtt{f}[0\,..\,\mathtt{t}-1]$
   todo caminho de o a v tem comprimento $\geq d[v]$
3. toda estrada que começa em $\mathtt{f}[0\,..\,\mathtt{s}-1]$ termina em $\mathtt{f}[0\,..\,\mathtt{t}-1]$

Conseqüência

Para cada v em $\mathtt{f}[0\,..\,\mathtt{t}-1]$, o número $d[v]$ é a distância de o a v.

Para provar invariantes 1 a 3, precisamos de mais dois invariantes:

4. $d[\mathtt{f}[\mathtt{s}]] \leq d[\mathtt{f}[\mathtt{s}{+}1]] \leq \cdots \leq d[\mathtt{f}[\mathtt{t}-1]]$
5. $d[\mathtt{f}[\mathtt{t}-1]] \leq d[\mathtt{f}[\mathtt{s}]] + 1$

Implementação circular da fila

0			t					s		N-1
444	555	666						111	222	333

uma fila `f[s..t-1]`

Remove elemento da fila

```
x = f[s++];
if (s == N) s = 0;
```

Insere y na fila

```
f[t++] = y;
if (t == N) t = 0;
```


Fila implementada em lista encadeada

```
typedef struct cel {
   int         valor;
   struct cel *seg;
} célula;
```

Decisões de projeto

- lista sem cabeça
- primeira célula: início da fila
- última célula: fim da fila

Fila vazia

```
célula *s, *t;  /* s aponta primeiro elemento da fila */
s = t = NULL;   /* t aponta último elemento da fila */
```

Remove elemento da fila

Recebe endereços **es** e **et** das variáveis **s** e **t** respectivamente.
Supõe que fila não está vazia e remove um elemento da fila.
Devolve o elemento removido.

```
int Remove (célula **es, célula **et) {
   célula *p;
   int x;
   p = *es;
   /* p aponta o primeiro elemento da fila */
   x = p->valor;
   *es = p->seg;
   free (p);
   if (*es == NULL) *et = NULL;
   return x;
}
```


Insere elemento na fila

Recebe endereços **es** e **et** das variáveis **s** e **t** respectivamente.
Insere um novo elemento com valor **y** na fila.
Atualiza os valores de **s** e **t**.

```
void Insere (int y, célula **es, célula **et) {
   célula *nova;
   nova = malloc (sizeof (célula));
   nova->valor = y;
   nova->seg = NULL;
   if (*et == NULL) *et = *es = nova;
   else {
      (*et)->seg = nova;
      *et = nova;
   }
}
```

Pilhas

Pilha implementada em vetor

0							t			N-1
111	222	333	444	555	666	777				

uma pilha `p[0..t-1]`

Remove elemento da pilha

```
x = p[--t];                    /* t -= 1; x = p[t]; */
```

Insere y na pilha

```
p[t++] = y;                    /* p[t] = y; t += 1; */
```

Aplicação: parênteses e chaves

- expressao bem-formada: `(()\{()\})`
- expressao malformada: `(\{)\}`

Algoritmo

Devolve 1 se a string `s` contém uma seqüência bem-formada e devolve 0 em caso contrário.

```
int BemFormada (char s[]) {
   char *p; int t;
   int n, i;
   n = strlen (s);
   p = malloc (n * sizeof (char));
   processo iterativo
   free (p);
   return t == 0;
}
```


processo iterativo

```
   t = 0;
   for (i = 0; s[i] != '\0'; i++) {
      /* p[0..t-1] é uma pilha */
      switch (s[i]) {
         case ')': if (t != 0 && p[t-1] == '(') --t;
                   else return 0;
                   break;
         case '}': if (t != 0 && p[t-1] == '{') --t;
                   else return 0;
                   break;
         default: p[t++] = s[i];
      }
   }
```

Aplicação: notação posfixa

Notação infixa versus posfixa

infixa	posfixa
(A + B ∗ C)	A B C ∗ +
(A ∗ (B + C) / D - E)	A B C + ∗ D / E -
(A + B ∗ (C - D ∗ (E - F) - G ∗ H) - I ∗ 3)	A B C D E F - ∗ - G H ∗ - ∗ + I 3 ∗ -
(A + B ∗ C / D ∗ E - F)	A B C ∗ D / E ∗ + F -
(A ∗ (B + (C ∗ (D + (E ∗ (F + G))))))	A B C D E F G + ∗ + ∗ + ∗

Algoritmo

Recebe uma expressão infixa representada por uma string `infix` que começa com `'('` e termina com `')'` seguido de `'\0'`.
Devolve a correspondente expressão posfixa.

```
char *InfixaParaPosfixa (char infix[]) {
   char *posfix, x;
   char *p; int t;
   int n, i, j;
   n = strlen (infix);
   posfix = malloc (n * sizeof (char));
   p = malloc (n * sizeof (char));
   processo iterativo
   free (p);
   posfix[j] = '\0';
   return posfix;
}
```

processo iterativo

```
t = 0; p[t++] = infix[0];  /* empilha '(' */
for (j = 0, i = 1; /*X*/ infix[i] != '\0'; i++) {
   /* p[0..t-1] é uma pilha de caracteres */
   switch (infix[i]) {
      case '(': p[t++] = infix[i];  /* empilha */
                break;
      case ')': while (1) {  /* desempilha */
                   x = p[--t];
                   if (x == '(') break;
                   posfix[j++] = x; }
                break;
      demais casos
   }
}
```


demais casos

```
      case '+':
      case '-': while (1) {
                   x = p[t-1];
                   if (x == '(') break;
                   --t;  /* desempilha */
                   posfix[j++] = x; }
                p[t++] = infix[i];  /* empilha */
                break;
      case '*':
      case '/': while (1) {
                   x = p[t-1];
                   if (x == '(' || x == '+' || x == '-')
                      break;
                   --t;
                   posfix[j++] = x; }
                p[t++] = infix[i];
                break;
      default: posfix[j++] = infix[i];
```

Aplicação de `InfixaParaPosfixa` à expressão `(A*(B*C+D))`

Valores das variáveis a cada passagem pelo ponto `X`:

infix[0..i-1]	p[0..t-1]	posfix[0..j-1]
(	(	
(A	(	A
(A *	(*	A
(A * (	(* (	A
(A * (B	(* (	A B
(A * (B *	(* (*	A B
(A * (B * C	(* (*	A B C
(A * (B * C +	(* (+	A B C *
(A * (B * C + D	(* (+	A B C * D
(A * (B * C + D)	(*	A B C * D +
(A * (B * C + D))		A B C * D + *

Pilha implementada em lista encadeada

```
typedef struct cel {
   int         valor;
   struct cel *seg;
} célula;
```

Decisões de projeto

- lista com cabeça
- segunda célula: topo da pilha

Pilha vazia

```
célula cabeça;
célula *p;
p = &cabeça;  /* p->seg é o topo da pilha */
p->seg = NULL;
```

Insere

```
void Empilha (int y, célula *p) {
   célula *nova;
   nova = malloc (sizeof (célula));
   nova->valor = y;
   nova->seg = p->seg;
   p->seg = nova;
}
```


Remove

```
int Desempilha (célula *p) {
   int x; célula *q;
   q = p->seg;
   x = q->valor;
   p->seg = q->seg;
   free (q);
   return x;
}
```

Busca em vetor ordenado

Problema:
Encontrar um dado número x
num vetor crescente $v[0\,..\,n{-}1]$.

Vetor é crescente se $v[0] \leq v[1] \leq \cdots \leq v[n{-}1]$.

Problema mais geral

Dado x e um vetor crescente $v[0\,..\,n{-}1]$,
encontrar j tal que $v[j{-}1] < x \leq v[j]$.

- $0 \leq j \leq n$
- se $j = 0$ então $x \leq v[0]$
- se $j = n$ então $v[n{-}1] < x$
- imagine $v[-1] = -\infty$ e $v[n] = \infty$

0												12
111	222	333	444	555	555	666	777	888	888	888	999	999

Se $x = 555$ então $j = 4$. Se $x = 1000$ então $j = 13$. Se $x = 110$ então $j = 0$.

Algoritmo de busca seqüencial

Recebe um vetor crescente $v[0\,..\,n-1]$ com $n \geq 1$ e um inteiro x.
Devolve um índice j em $0\,..\,n$ tal que $v[j-1] < x \leq v[j]$.

```
int BuscaSeqüencial (int x, int n, int v[]) {
   int j = 0;
   while (j < n && v[j] < x) ++j;
   return j;
}
```

- invariante: no começo de cada iteração tem-se $v[j-1] < x$
- consumo de tempo: proporcional a n

Algoritmo de busca binária

Recebe um vetor crescente $v[0\,..\,n-1]$ com $n \geq 1$ e um inteiro x.
Devolve um índice j em $0\,..\,n$ tal que $v[j-1] < x \leq v[j]$.

```
int BuscaBinária (int x, int n, int v[]) {
   int e, m, d;
   e = -1; d = n;
   while (/*X*/ e < d-1) {
      m = (e + d)/2;
      if (v[m] < x) e = m;
      else d = m;
   }
   return d;
}
```

Invariante: a cada passagem pelo ponto X temos $v[e] < x \leq v[d]$.

0			e			d						$n-1$
111	222	333	444	555	555	666	777	888	888	888	999	999

Consumo de tempo

- em cada iteração, o tamanho do vetor em jogo é $d - e - 1$
- tamanho do vetor na primeira, segunda, terceira, etc. iterações: n, $n/2$, $n/4$, ..., $n/2^k$, ...
- número total de iterações: $\cong \log_2 n$
- consumo de tempo: proporcional a $\log_2 n$

Versão recursiva

```
int BuscaBinária2 (int x, int n, int v[]) {
   return BuscaBinR (x, -1, n, v);
}
```

BuscaBinR recebe um vetor crescente $v[e\,..\,d]$ e um x tal que $v[e] < x \leq v[d]$. Devolve um índice j no intervalo $e{+}1\,..\,d$ tal que $v[j{-}1] < x \leq v[j]$.

```
int BuscaBinR (int x, int e, int d, int v[]) {
   if (e == d-1) return d;
   else {
      int m = (e + d)/2;
      if (v[m] < x)
         return BuscaBinR (x, m, d, v);
      else
         return BuscaBinR (x, e, m, v);
   }
}
```

ALGORITMOS DE ORDENAÇÃO

Problema

Rearranjar os elementos de um vetor $v[0\,..\,n{-}1]$
de tal modo que ele fique crescente.

Vetor é crescente se $v[0] \le v[1] \le \cdots \le v[n{-}1]$.

Ordenação
por inserção
por seleção

Algoritmo de ordenação por inserção

Rearranja o vetor $v[0\,..\,n{-}1]$ em ordem crescente.

```
void Inserção (int n, int v[]) {
   int i, j, x;
   for (j = 1; /*A*/ j < n; j++) {
      x = v[j];
      for (i = j-1; i >= 0 && v[i] > x; i--)
         v[i+1] = v[i];
      v[i+1] = x;
   }
}
```

Invariantes: a cada passagem pelo ponto A

1. $v[0\,..\,n{-}1]$ é uma permutação do vetor original
2. o vetor $v[0\,..\,j{-}1]$ é crescente

0		crescente		$j{-}1$	j					$n{-}1$
444	555	555	666	777	222	999	222	999	222	999

Consumo de tempo

- proporcional ao número de execuções de "v[i] > x"
- no pior caso, esse número é $\sum_{j=1}^{n-1} j = n\,(n-1)/2$
- consumo de tempo total: no máximo n^2 unidades de tempo

Algoritmo de seleção

Rearranja o vetor $v[0\,..\,n{-}1]$ em ordem crescente.

```
void Seleção (int n, int v[]) {
   int i, j, min, x;
   for (i = 0; /*A*/ i < n-1; i++) {
      min = i;
      for (j = i+1; j < n; j++)
         if (v[j] < v[min]) min = j;
      x = v[i]; v[i] = v[min]; v[min] = x;
   }
}
```

Invariantes: a cada passagem pelo ponto `A`

1. $v[0\,..\,n{-}1]$ é uma permutação do vetor original
2. $v[0\,..\,i{-}1]$ está em ordem crescente
3. $v[i-1] \leq v[j]$ para $j = i, i{+}1, \ldots, n{-}1$

0				$i{-}1$	i					$n{-}1$
110	120	120	130	140	666	999	666	999	666	999

pequenos, crescente grandes

Consumo de tempo

- no máximo n^2 unidades de tempo

Algoritmo Mergesort

Problema principal

Rearranjar os elementos de um vetor $v[0\,..\,n{-}1]$
de tal modo que ele fique crescente,
ou seja, de modo que $v[0] \leq v[1] \leq \cdots \leq v[n{-}1]$.

Problema auxiliar: intercalação

Rearranjar $v[p\,..\,r{-}1]$ em ordem crescente
sabendo que $v[p\,..\,q{-}1]$ e $v[q\,..\,r{-}1]$ são crescentes.

p						$q{-}1$	q			$r{-}1$
111	333	555	555	777	999	999	222	444	777	888

Algoritmo de intercalação

Recebe vetores crescentes $v[p\,..\,q{-}1]$ e $v[q\,..\,r{-}1]$
e rearranja $v[p\,..\,r{-}1]$ em ordem crescente.

```
void Intercala (int p, int q, int r, int v[]) {
   int i, j, k, *w;
   w = malloc ((r-p) * sizeof (int));
   i = p; j = q; k = 0;
   while (i < q && j < r) {
      if (v[i] <= v[j]) w[k++] = v[i++];
      else w[k++] = v[j++];
   }
   while (i < q) w[k++] = v[i++];
   while (j < r) w[k++] = v[j++];
   for (i = p; i < r; i++) v[i] = w[i-p];
   free (w);
}
```

Consumo de tempo do algoritmo `Intercala`

- proporcional ao número de elementos do vetor

Algoritmo Mergesort (ordena por intercalação)

Rearranja o vetor $v[p\,..\,r-1]$ em ordem crescente.

```
void Mergesort (int p, int r, int v[]) {
   if (p < r - 1) {
      int q = (p + r)/2;
      Mergesort (p, q, v);
      Mergesort (q, r, v);
      Intercala (p, q, r, v);
   }
}
```

0	1	2	3	4	5	6	7	8	9	10
999	111	222	999	888	333	444	777	555	666	555
999	111	222	999	888	333	444	777	555	666	555
999	111	222	999	888	333	444	777	555	666	555

$$\vdots$$

111	999	222	888	999	333	444	777	555	555	666
111	222	888	999	999	333	444	555	555	666	777
111	222	333	444	555	555	666	777	888	999	999

$$v[0\,.\,.\,n{-}1]$$

$$v[0\,.\,.\,\tfrac{n}{2}{-}1] \quad v[\tfrac{n}{2}\,.\,.\,n{-}1]$$

$$v[0\,.\,.\,\tfrac{n}{4}{-}1] \quad v[\tfrac{n}{4}\,.\,.\,\tfrac{n}{2}{-}1] \quad v[\tfrac{n}{2}\,.\,.\,\tfrac{3n}{4}{-}1] \quad v[\tfrac{3n}{4}\,.\,.\,n{-}1]$$

$$\vdots$$

Consumo de tempo do Mergesort

- aproximadamente $\log_2 n$ "rodadas"
- cada "rodada" consome n unidades de tempo
- total: $n \log_2 n$ unidades de tempo

Versão iterativa

```
void MergesortI (int n, int v[]) {
   int p, r, b = 1;
   while (b < n) {
      p = 0;
      while (p + b < n) {
         r = p + 2*b;
         if (r > n) r = n;
         Intercala (p, p+b, r, v);
         p = p + 2*b;
      }
      b = 2*b;
   }
}
```

0				p		$p+b$		$p+2b$		$n-1$
111	999	222	999	333	888	444	777	555	666	555

Algoritmo Heapsort

Problema

Rearranjar os elementos de um vetor $v[0\mathinner{\ldotp\ldotp} n{-}1]$
em ordem crescente.

Definição

Um max-heap é um vetor $v[1\mathinner{\ldotp\ldotp} m]$ tal que $v[\lfloor \frac{1}{2} f \rfloor] \geq v[f]$
para $f = 2, 3, \ldots, m$.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
999	888	666	333	777	555	555	333	222	111	444	111	222	444	111

1
2 3
4 5 6 7
8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56

Algoritmo auxiliar 1: inserção em um heap

Transforma $v[1\,..\,m{+}1]$ em max-heap supondo que $v[1\,..\,m]$ é max-heap.

```
void InsereEmHeap (int m, int v[]) {
   int f = m+1;
   while /*X*/ (f > 1 && v[f/2] < v[f]) {
      int t = v[f/2]; v[f/2] = v[f]; v[f] = t;
      f = f/2;
   }
}
```

- invariante no pto X: $v[\lfloor \frac{1}{2} i \rfloor] \geq v[i]$ para $i = 2, \ldots, m{+}1,\ \ i \neq f$
- consumo: $\log_2(m+1)$ unidades de tempo

1	2	3	4	5	6	7	8	9	10	11	12	13	14
98	97	96	95	94	93	92	91	90	89	87	86	85	99
98	97	96	95	94	93	99	91	90	89	87	86	85	92
98	97	99	95	94	93	96	91	90	89	87	86	85	92
99	97	98	95	94	93	96	91	90	89	87	86	85	92

Transforma $v[1\,..\,14]$ em max-heap
supondo que $v[1\,..\,13]$ é max-heap.

Algoritmo auxiliar 2

Transforma quase-max-heap $v[1\,..\,m]$ em max-heap.

```
void SacodeHeap (int m, int v[]) {
   int t, f = 2;
   while /*X*/ (f <= m) {
      if (f < m && v[f] < v[f+1])  ++f;
      if (v[f/2] >= v[f])  break;
      t = v[f/2]; v[f/2] = v[f]; v[f] = t;
      f *= 2;
   }
}
```

- $v[1\,..\,m]$ é quase-max-heap se $v[\lfloor \frac{1}{2}f \rfloor] \geq v[f]$ para $f = 4, 5, \ldots, m$
- invariante no ponto X: $v[\lfloor \frac{1}{2}i \rfloor] \geq v[i]$ quando $i \neq f$ e $i \neq f+1$
- consumo: $\log_2 m$ unidades de tempo

Algoritmo Heapsort

Rearranja vetor $v[1\,..\,n]$ de modo que ele fique crescente.

```
void Heapsort (int n, int v[]) {
   int m;
   for (m = 1; m < n; m++)
      InsereEmHeap (m, v);
   for (m = n; /*X*/ m > 1; m--) {
      int t = v[1]; v[1] = v[m]; v[m] = t;
      SacodeHeap (m-1, v);
   }
}
```

Invariantes no ponto X

- $v[1\,..\,m]$ é um max-heap
- $v[1\,..\,m] \le v[m{+}1\,..\,n]$
- $v[m{+}1\,..\,n]$ está em ordem crescente

1			max-heap				m			crescente		n
777	777	666	444	222	111	444	333	777	888	888	999	999
			elementos pequenos							elementos grandes		

Consumo de tempo do Heapsort

- no pior caso: $n \log_2 n$ unidades de tempo

Algoritmo Quicksort

Problema:
Rearranjar um vetor $v[0\,..\,n{-}1]$
em ordem crescente.

Subproblema da separação: formulação vaga

Rearranjar um vetor $v[p\,..\,r]$ de modo que
os elementos pequenos fiquem todos do lado esquerdo
e os grandes do lado direito.

Formulação concreta

Rearranjar $v[p\,..\,r]$ de modo que $v[p\,..\,j{-}1] \leq v[j] < v[j{+}1\,..\,r]$
para algum j em $p\,..\,r$.

Algoritmo da separação

Recebe um vetor $v[p\,..\,r]$ com $p \leq r$.
Rearranja os elementos do vetor e
devolve j em $p\,..\,r$ tal que $v[p\,..\,j{-}1] \leq v[j] < v[j{+}1\,..\,r]$.

```
int Separa (int p, int r, int v[]) {
   int c, j, k, t;
   c = v[r]; j = p;
   for (k = p; /*A*/ k < r; k++)
      if (v[k] <= c) {
         t = v[j], v[j] = v[k], v[k] = t;
         j++;
      }
   v[r] = v[j], v[j] = c;
   return j;
}
```


Invariantes no ponto A

- $v[p\,..\,r]$ é uma permutação do vetor original
- $v[p\,..\,j{-}1] \leq c < v[j\,..\,k{-}1]$ e $v[r] = c$
- $p \leq j \leq k \leq r$

p			j			k			r
$\leq c$	$\leq c$	$\leq c$	$> c$	$> c$	$> c$	?	?	?	$= c$

				j					k
$\leq c$	$\leq c$	$\leq c$	$\leq c$	$> c$	$> c$	$> c$	$> c$	$> c$	$= c$

				j					
$\leq c$	$\leq c$	$\leq c$	$\leq c$	$= c$	$> c$	$> c$	$> c$	$> c$	$> c$

				j					k
$\leq c$	$\leq c$	$\leq c$	$\leq c$	$> c$	$> c$	$> c$	$> c$	$> c$	$= c$

última passagem pelo ponto `A`

				j					
$\leq c$	$\leq c$	$\leq c$	$\leq c$	$= c$	$> c$	$> c$	$> c$	$> c$	$> c$

resultado final

Consumo de tempo do algoritmo Separa

proporcional ao número de elementos do vetor

Algoritmo Quicksort

Rearranja o vetor $v[p\,..\,r]$, com $p \leq r+1$,
de modo que ele fique em ordem crescente.

```
void Quicksort (int p, int r, int v[]) {
   int j;
   if (p < r) {
      j = Separa (p, r, v);
      Quicksort (p, j-1, v);
      Quicksort (j+1, r, v);
   }
}
```

Consumo de tempo do Quicksort

- no pior caso: n^2 unidades de tempo
- em média: $n \log_2 n$ unidades de tempo

$n := r - p + 1$

Quicksort com controle da altura da pilha de execução

Cuida primeiro do *menor* dos subvetores $v[p\,..\,j{-}1]$ e $v[j{+}1\,..\,r]$.

```
void QuickSortP (int p, int r, int v[]) {
   int j;
   while (p < r) {
      j = Separa (p, r, v);
      if (j - p < r - j) {
         QuickSortP (p, j-1, v);
         p = j + 1;
      } else {
         QuickSortP (j+1, r, v);
         r = j - 1;
      }
   }
}
```

Altura da pilha de execução: $\log_2 n$

Algoritmos de enumeração

Enumerar = fazer uma lista
de todos os objetos de um determinado tipo

Problema

Fazer uma lista, sem repetições, de todas as subseqüências de $1, 2, \ldots, n$.

$$
\begin{array}{llll}
1 & & & \\
1 & 2 & & \\
1 & 2 & 3 & \\
1 & 2 & 3 & 4 \\
1 & 2 & 4 & \\
1 & 3 & & \\
1 & 3 & 4 & \\
1 & 4 & & \\
2 & & & \\
2 & 3 & & \\
2 & 3 & 4 & \\
2 & 4 & & \\
3 & & & \\
3 & 4 & & \\
4 & & &
\end{array}
$$

Ordem lexicográfica de seqüências

$\langle r_1, r_2, \ldots, r_j \rangle$ precede $\langle s_1, s_2, \ldots, s_k \rangle$ se

1. $j < k$ e $\langle r_1, \ldots, r_j \rangle = \langle s_1, \ldots, s_j \rangle$ ou
2. existe i tal que $\langle r_1, \ldots, r_{i-1} \rangle = \langle s_1, \ldots, s_{i-1} \rangle$ e $r_i < s_i$

Algoritmo de enumeração em ordem lexicográfica

Recebe $n \geq 1$ e imprime todas as subseqüências não-vazias de $1, 2, \ldots, n$ em ordem lexicográfica.

```
void SubseqLex (int n) {
   int *s, k;
   s = malloc ((n+1) * sizeof (int));
   processo iterativo
   free (s);
}
```


processo iterativo

```
   s[0] = 0; k = 0;
   while (1) {
      if (s[k] < n) {
         s[k+1] = s[k] + 1;
         k += 1;
      } else {
         s[k-1] += 1;
         k -= 1;
      }
      if (k == 0) break;
      imprima (s, k);
   }
```

Invariante

Cada iteração começa com subseqüência $\langle s_1, s_2, \ldots, s_k \rangle$ de $\langle 1, 2, \ldots, n \rangle$.

				k			
0	1	2	3	4	5	6	7
0	2	4	5	7	8	?	?

Vetor s no início de uma iteração
de SubseqLex com $n = 7$.

Versão recursiva

```
void SubseqLex2 (int n) {
   int *s;
   s = malloc ((n+1) * sizeof (int));
   SseqR (s, 0, 1, n);
   free (s);
}

void SseqR (int s[], int k, int m, int n) {
   if (m <= n) {
      s[k+1] = m;
      imprima (s, k+1);
      SseqR (s, k+1, m+1, n);  /* inclui m */
      SseqR (s, k,   m+1, n);  /* não inclui m */
   }
}
```

Ordem lexicográfica especial

$\langle r_1, r_2, \ldots, r_j \rangle$ precede $\langle s_1, s_2, \ldots, s_k \rangle$ se

1. $j > k$ e $\langle r_1, \ldots, r_k \rangle = \langle s_1, \ldots, s_k \rangle$ ou
2. existe i tal que $\langle r_1, \ldots, r_{i-1} \rangle = \langle s_1, \ldots, s_{i-1} \rangle$ e $r_i < s_i$

1	2	3	4
1	2	3	
1	2	4	
1	2		
1	3	4	
1	3		
1	4		
1			
2	3	4	
2	3		
2	4		
2			
3	4		
3			
4			

Algoritmo de enumeração em ordem lexicográfica especial

Recebe $n \geq 1$ e imprime, em ordem lexicográfica especial, todas as subseqüências não-vazias de $1, 2, \ldots, n$.

```
void SubseqLexEsp (int n) {
   int *s, k;
   s = malloc ((n+1) * sizeof (int));
   processo iterativo
   free (s);
}
```

processo iterativo

```
s[1] = 0; k = 1;
while (1) {
   if (s[k] == n) {
      k -= 1;
      if (k == 0) break;
   } else {
      s[k] += 1;
      while (s[k] < n) {
         s[k+1] = s[k] + 1;
         k += 1;
      }
   }
   imprima (s, k);
}
```


Versão recursiva

Recebe $n \geq 1$ e imprime todas as subseqüências de $1, 2, \ldots, n$ em ordem lexicográfica especial.

```
void SubseqLexEsp2 (int n) {
   int *s;
   s = malloc ((n+1) * sizeof (int));
   SseqEspR (s, 0, 1, n);
   free (s);
}
```

continua...

continuação

Recebe um vetor $s[1\mathinner{..}k]$ e imprime, em ordem lexicográfica especial, todas as seqüências da forma $s[1],\ldots,s[k],t[k{+}1],\ldots$ tais que $t[k{+}1],\ldots$ é uma subseqüência de $m,m{+}1,\ldots,n$.
Em seguida, imprime a seqüência $s[1],\ldots,s[k]$.

```
void SseqEspR (int s[], int k, int m, int n) {
   if (m > n) imprima (s, k);
   else {
      s[k+1] = m;
      SseqEspR (s, k+1, m+1, n);  /* inclui m */
      SseqEspR (s, k,   m+1, n);  /* não inclui m */
   }
}
```


2	4	7	8	9
2	4	7	8	
2	4	7	9	
2	4	7		
2	4	8	9	
2	4	8		
2	4	9		
2	4	9		
2	4			

Resultado de `SseqEspR` $(s$,2,7,9)
supondo s[1] $= 2$ e s[2] $= 4$.

Busca de palavras em um texto

Problema:
Encontrar as ocorrências de $a[1\,..\,m]$ em $b[1\,..\,n]$.


```
      a l g o r í t i m o
O s   a l g o r í t i m o s   d e   o r d e n a ç ã o
                        ↑

                  3 1 4 1 5 9
3 1 3 1 4 3 1 4 1 3 1 4 1 5 9 3 1 4 1 5 9 2 6 3 1 4
                            ↑           ↑

                            T A C T A
G T A G T A T A T A T A T A T A C T A C T A G T A G
                                    ↑     ↑
```

Definições

- $a[1\,..\,m]$ é sufixo de $b[1\,..\,k]$ se
  $m \leq k$ e $a[1\,..\,m] = b[k{-}m{+}1\,..\,k]$
- $a[1\,..\,m]$ ocorre em $b[1\,..\,n]$ se
  existe k no intervalo $m\,..\,n$ tal que $a[1\,..\,m]$ é sufixo de $b[1\,..\,k]$

Problema

Encontrar o número de ocorrências de $a[1\,..\,m]$ em $b[1\,..\,n]$.


```
typedef unsigned char *palavra;
typedef unsigned char *texto;
```

Algoritmo trivial

Recebe palavra $a[1\,..\,m]$ e texto $b[1\,..\,n]$, com $m \geq 1$ e $n \geq 0$, e devolve o número de ocorrências de a em b.

```
int trivial (palavra a, int m, texto b, int n) {
   int k, r, ocorrs;
   ocorrs = 0;
   for (k = m; k <= n; k++) {
      r = 0;
      while (r < m && a[m-r] == b[k-r]) r += 1;
      if (r >= m) ocorrs += 1;
   }
   return ocorrs;
}
```

Algoritmo de Boyer–Moore

```
                                      B C B A
                X C B A B X C B A A X B C B A B X
                      ^   ^     ^ ^         ^   ^
```

posições k em que $a[1\,..\,4]$ é comparada com $b[k{-}3\,..\,k]$

1	2	3	4
B	C	B	A

c	···	?	@	A	B	C	D	E	F	G	···
T1[c]	···	4	4	0	1	2	4	4	4	4	···

Tabela de deslocamentos T1

T1[c] é o menor t em $0\,..\,m{-}1$ tal que $a[m-t] = c$

Primeiro algoritmo de Boyer–Moore

Recebe uma palavra $a[1\,..\,m]$ e um texto $b[1\,..\,n]$, com $m \geq 1$ e $n \geq 0$,
e devolve o número de ocorrências de a em b.
Supõe que cada elemento de a e b pertence ao conjunto de caracteres 0..255.

```
int BoyerMoore1 (palavra a, int m, texto b, int n) {
   int T1[256], i, k, r, ocorrs;
   /* pré-processamento da palavra a */
   for (i = 0; i < 256; i++) T1[i] = m;
   for (i = 1; i <= m; i++) T1[a[i]] = m - i;
   [busca da palavra a no texto b]
   return ocorrs;
}
```

busca da palavra a no texto b

```
ocorrs = 0; k = m;
while (k <= n) {
   r = 0;
   while (m - r >= 1 && a[m-r] == b[k-r]) r += 1;
   if (m - r < 1) ocorrs += 1;
   if (k == n) k += 1;
   else k += T1[b[k+1]] + 1;
}
```


Segundo algoritmo de Boyer–Moore

Tabela de deslocamentos T2

T2 $[i]$ é o menor t em $1\,..\,m{-}1$ tal que $m - t$ é bom para i

j é **bom para** i se $a[i\,..\,m]$ é sufixo de $a[1\,..\,j]$
ou $a[1\,..\,j]$ é sufixo de $a[i\,..\,m]$

1	2	3	4	5	6
C	A	A	B	A	A

i	6	5	4	3	2	1
T2 $[i]$	1	3	6	6	6	6

1	2	3	4	5	6	7	8
B	A	-	B	A	.	B	A

i	8	7	6	5	4	3	2	1
T2 $[i]$	3	3	6	6	6	6	6	6

1	2	3	4	5	6	7	8	9	10	11
B	A	-	B	A	*	B	A	*	B	A

i	11	10	9	8	7	6	5	4	3	2	1
T2 $[i]$	3	3	3	3	3	9	9	9	9	9	9

Segundo algoritmo de Boyer–Moore

Recebe uma palavra $a[1\,..\,m]$ com $1 \le m \le$ MAX e um texto $b[1\,..\,n]$
e devolve o número de ocorrências de a em b.

```
int BoyerMoore2 (palavra a, int m, texto b, int n) {
   int T2[MAX], i, j, k, r, ocorrs;
   /* pré-processamento da palavra a */
   for (i = m; i >= 1; i--) {
      j = m-1; r = 0;
      while (m-r >= i && j-r >= 1)
         if (a[m-r] == a[j-r]) r += 1;
         else j -= 1, r = 0;
      T2[i] = m - j;
   }
   busca da palavra a no texto b
   return ocorrs;
}
```

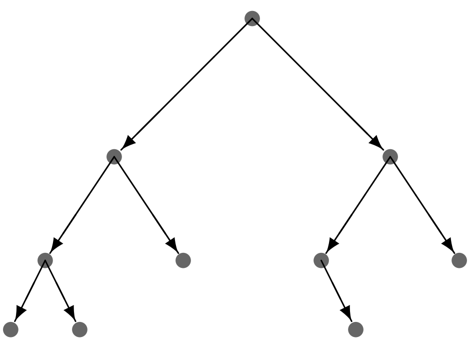

busca da palavra a no texto b

```
   ocorrs = 0; k = m;
   while (k <= n) {
      r = 0;
      while (m-r >= 1 && a[m-r] == b[k-r]) r += 1;
      if (m-r < 1) ocorrs += 1;
      if (r == 0) k += 1;
      else k += T2[m-r+1];
   }
```

Consumo de tempo dos algoritmos de Boyer–Moore

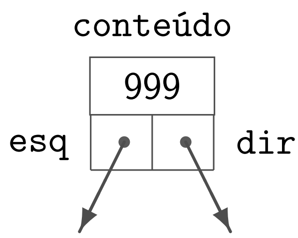
- pré-processamento: m^2 unidades de tempo
- busca, pior caso: mn unidades de tempo
- busca, em média: n unidades de tempo

Árvores binárias



Estrutura de um nó

```
struct cel {
   int         conteúdo;
   struct cel *esq;
   struct cel *dir;
};
typedef struct cel nó;
```

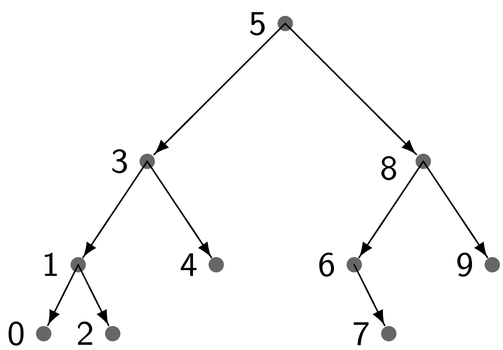



```
typedef nó *árvore;
```


Varredura esquerda-raiz-direita

Visite

- a subárvore esquerda (em ordem e-r-d)
- depois a raiz
- depois a subárvore direita (em ordem e-r-d)



Algoritmo de varredura e-r-d

Recebe uma árvore binária `r`
e imprime o conteúdo de seus nós em ordem e-r-d.

```
void Erd (árvore r) {
   if (r != NULL) {
      Erd (r->esq);
      printf ("%d\n", r->conteúdo);
      Erd (r->dir);
   }
}
```

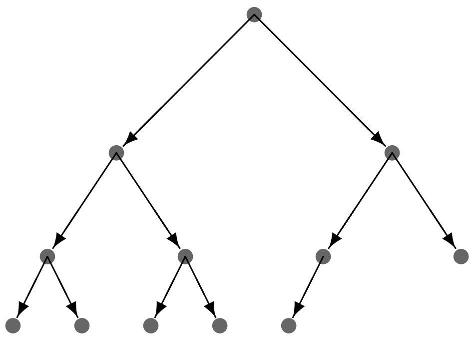

Versão iterativa

```
void ErdI (árvore r) {
   nó *p[100], *x;
   int t = 0;
   x = r;
   while (x != NULL || t > 0) {
      /* o topo da pilha p[0..t-1] está em t-1 */
      if (x != NULL) {
         p[t++] = x;
         x = x->esq;
      }
      else {
         x = p[--t];
         printf ("%d\n", x->conteúdo);
         x = x->dir;
      }
   }
}
```

Altura

- de nó = distância entre nó e seu descendente mais afastado
- de árvore = altura da raiz

Se árvore tem n nós e altura h então $\lfloor \log_2 n \rfloor \le h < n$.



$h = \lfloor \log_2 12 \rfloor = 3$

Algoritmo da altura

Devolve a altura da árvore binária r.

```
int Altura (árvore r) {
   if (r == NULL)
      return -1; /* a altura de uma árvore vazia é -1 */
   else {
      int he = Altura (r->esq);
      int hd = Altura (r->dir);
      if (he < hd) return hd + 1;
      else return he + 1;
   }
}
```

Estrutura de nó com campo pai

```
struct cel {
   int          conteúdo;
   struct cel *pai;
   struct cel *esq;
   struct cel *dir;
};
```


Algoritmo do nó seguinte

Recebe um nó `x` de uma árvore binária cujos nós têm campo pai
e devolve o (endereço do) nó seguinte na ordem e-r-d.
A função supõe que `x` $\neq$ `NULL`.

```
nó *Seguinte (nó *x) {
   if (x->dir != NULL) {
      nó *y = x->dir;
      while (y->esq != NULL) y = y->esq;
      return y;
   }
   while (x->pai != NULL && x->pai->dir == x)
      x = x->pai;
   return x->pai;
}
```

Árvores binárias de busca

Estrutura de um nó

```
struct cel {
   int          chave;
   int          conteúdo;
   struct cel *esq;
   struct cel *dir;
};
typedef struct cel nó;
```

Árvore de busca: definição

$$\texttt{E.chave} \leq \texttt{X.chave} \leq \texttt{D.chave}$$

para todo nó X, todo nó E na subárvore esquerda de X
e todo nó D na subárvore direita de X

Algoritmo de busca

Recebe k e uma árvore de busca `r`.
Devolve um nó cuja `chave` é k ou devolve `NULL` se tal nó não existe.

```
nó *Busca (árvore r, int k) {
   if (r == NULL || r->chave == k)
      return r;
   if (r->chave > k)
      return Busca (r->esq, k);
   else
      return Busca (r->dir, k);
}
```


Versão iterativa

```
   while (r != NULL && r->chave != k) {
      if (r->chave > k) r = r->esq;
      else r = r->dir;
   }
   return r;
```

```
nó *novo;
novo = malloc (sizeof (nó));
novo->chave = k;
novo->esq = novo->dir = NULL;
```

Algoritmo de inserção

Recebe uma árvore de busca `r` e uma folha avulsa `novo`.
Insere `novo` na árvore de modo que a árvore continue sendo de busca
e devolve o endereço da nova árvore.

```
árvore Insere (árvore r, nó *novo) {
   nó *f, *p;
   if (r == NULL) return novo;
   processo iterativo
   return r;
}
```


processo iterativo

```
   f = r;
   while (f != NULL) {
      p = f;
      if (f->chave > novo->chave) f = f->esq;
      else f = f->dir;
   }
   if (p->chave > novo->chave) p->esq = novo;
   else p->dir = novo;
```

Algoritmo de remoção da raiz

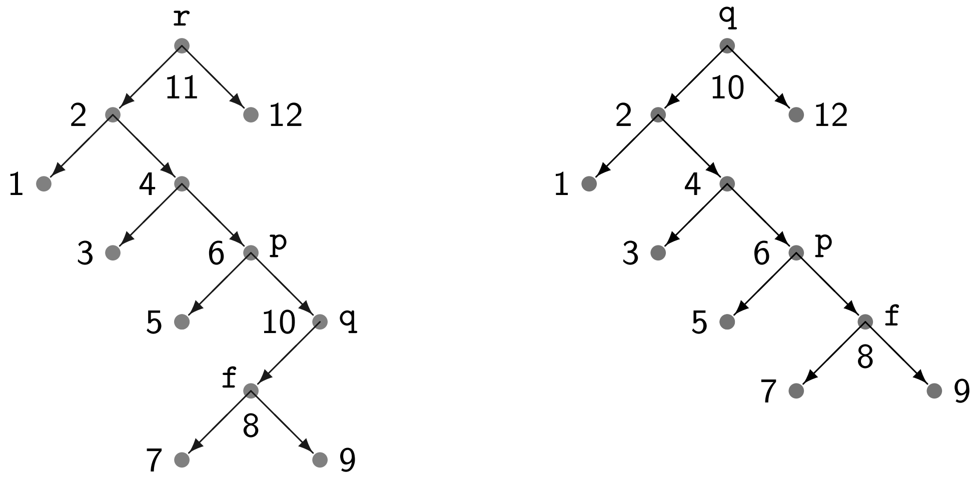
Recebe uma árvore não-vazia `r`, remove a raiz da árvore e rearranja a árvore de modo que ela continue sendo de busca. Devolve o endereço da nova raiz.

```
árvore RemoveRaiz (árvore r) {
   nó *p, *q;
   if (r->esq == NULL) q = r->dir;
   else {
      processo iterativo
   }
   free (r);
   return q;
}
```


processo iterativo

```
      p = r; q = r->esq;
      while (q->dir != NULL) {
         p = q; q = q->dir;
      }
      /* q é o nó anterior a r na ordem e-r-d */
      /* p é o pai de q */
      if (p != r) {
         p->dir = q->esq;
         q->esq = r->esq;
      }
      q->dir = r->dir;
```

Exemplo: antes e depois de RemoveRaiz



nó f passa a ser o filho direito de p

nó q fica no lugar de r

Remoção do filho esquerdo de x

```
x->esq = RemoveRaiz (x->esq);
```

Remoção do filho direito de x

```
x->dir = RemoveRaiz (x->dir);
```

Consumo de tempo da busca, inserção e remoção

- pior caso: proporcional à altura da árvore
- árvore “balanceada”: proporcional a $\log_2 n$

n = número de nós da árvore

Fim