Universidade Federal do Espíto Santo
Centro de Ciências Exatas, Naturais e de Saúde-CCENS
Departamento de Computação

Estrutura de Dados II **Professor** Dr. Marcelo Otone Aguiar
2024-1
Exercícios-Parte I

- Esta lista NÃO é avaliativa.
- Ao resolver essa lista considere:
  - **Discutir:** analisar questionando; levantar questões a respeito de (algo); examinar detalhadamente.
  - **Criticar:** emitir uma opinião ou juízo de valor.
  - Toda instrução que utilize os operadores relacionais deve ser considerada ao se contar as comparações.

---

1. Discuta a seguinte versão da função buscaSequencial:

```c
int buscaSequencial(int chave, int n, int v[]) {
    int i;
    for (i = 0; i < n; i++) {
        if (chave >= v[i])
            break;
    }
    if (i < n && v[i] == chave)
        return i;
    else
        return -1;
}
```

   O laço desta busca percorre a lista enquanto à chave é maior ou igual ao item atual, desta forma percebe-se claramente que a busca, apesar de sequencial, foi implementada de forma a exigir a premissa de que os dados estejam ordenados. Isso é claramente ruim, pois a grande vantagem da busca sequencial é possibilitar a busca em dados não ordenados, uma vez que, para dados ordenados há buscas com eficiência superior à sequencial, como é o caso da busca binária.

2. Critique a seguinte versão da função buscaSequencial:

```c
int busca(int chave, int n, int v[]) {
    int i = 0;
    while (v[i] < chave && i < n)
        ++i;
    if (v[i] == chave)
        return i;
    else
       return -1;
}
```

Uma "aparente"vantagem dessa busca é o fato de que no laço, ocorre apenas o incremento da uma variável contadora, com o objetivo de encontrar a posição do elemento igual à chave, isso trás um possível desempenho, contudo, a condição do laço exige a premissa de que os dados estejam ordenados, visto que a comparação é: $v[i] < chave$, assim, sabe-se que há buscas com essa premissa, no caso dados ordenados, que são muito superiores que a busca sequencial em desempenho.

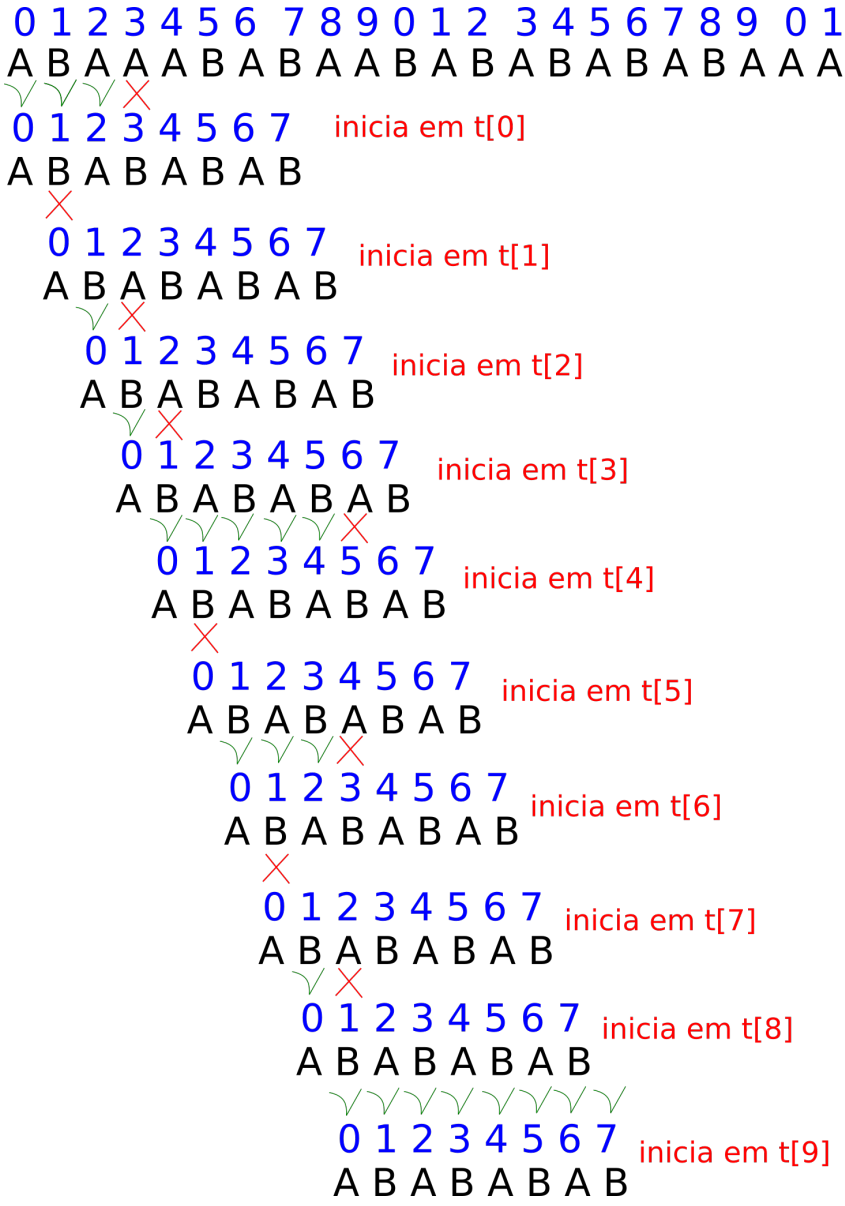
3. Discuta a seguinte versão recursiva da função buscaSequencial:

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

Essa busca recursiva faz a pesquisa a partir do final da lista, pois a comparação ocorre sempre com $v[n-1]$, ou seja, se o tamanho do vetor é 10, na primeira invocação da busca, o valor de n será 10, assim, a comparação ocorrerá com a posição 9, que é a última da lista. Na sequência, a busca é novamente acionada de forma recursiva, passando como argumento $n-1$, ou seja, a penúltima posição, em uma próxima chamada recursiva, a antepenúltima posição e assim sucessivamente, até encontrar o elemento, no segundo if ou chegar ao final da lista no primeiro if. Essa busca não requer que os dados estejam ordenados.

4. Responda as seguintes perguntas sobre a função buscaBinaria apresentada em sala de aula:
   (a) Que acontece se while (esquerda <= direita) for trocado por while (esquerda < direita)?
   O algoritmo não será capaz de encontrar o primeiro e último registro.
   (b) Que acontece se while (esquerda <= direita) for trocado por while (esquerda <= direita+1)?
   Se informar um elemento existente, o algoritmo irá encontrar, mas se informar um elemento inexistente, tanto menor quanto maior, o algoritmo entrará em loop infinito.
   (c) Que acontece se esquerda = meio+1 for trocado por esquerda = meio? E se for trocado por esquerda = meio-1?
   Nos dois casos, o algoritmo será capaz de encontrar um elemento existente, com exceção do último (entra em loop infinito), além disso, irá entrar em loop infinito caso o elemento seja superior ao último do vetor.
   (d) Que acontece se direita = meio-1 for trocado por direita = meio ou por direita = meio+1?
   Em ambos os casos, o algoritmo será capaz de encontrar um elemento existente, com exceção do primeiro (entra em loop infinito) e será capaz de determinar a não existência de um elemento com exceção dos inferiores ao menor (entra em loop infinito).

5. Faça um rastreamento do método **busca()** de força bruta para o padrão **A B A B A B A B** e o texto **A B A A A B A B A A B A B A B A B A B A A A A A A**.



6. Com base no algoritmo de busca **força bruta** a seguir, faça:

```
1  int ForcaBruta(char t[], char p[]) {
2      int i, j, n = strlen(t), m = strlen(p);
3
4      for (i=0; i<n-m; i++) {
5          j=0;
6          while ((j < m) && (t[i+j] == p[j])) {
7              j++;
8              if (j == m)
9                  return i;
10         }
11     }
12     return -1;
```

```
}
```

(a) Escreva uma versão **buscaDE** que percorra o padrão da direita para a esquerda.

```
int buscaDE(char t[], char p[]) {
    int i, j, n = strlen(t), m = strlen(p);

    for (i=0; i<n-m; i++) {
        j=m-1;
        while ((j >= 0) && (t[i+j] == p[j])) {
            j--;
        if (j == -1)
            return i;
        }
    }
    return -1;
}
```

(b) Ajuste a implementação do força bruta original para retornar o número de ocorrências do padrão.

```
int ForcaBrutaOcorrencias(char t[], char p[]) {
    int i, j, n = strlen(t), m = strlen(p);
    int ocorr = 0;

    for (i=0; i<=n-m; i++) {
        j=0;
        while ((j < m) && (t[i+j] == p[j])) {
            j++;
            if (j == m)
                ocorr++;
        }
    }
    return ocorr;
}
```

(c) Ajuste a implementação do força bruta para retornar todas as ocorrências do padrão encontradas.

```
int* ForcaBrutaAll(char *texto, char *padrao) {
    int i, j, ocorr = 0;
    int *retorno = malloc(N * sizeof(int));
    for (i=0; i<=N-M; i++) {
        j = 0;
        retorno[i] = -1;
        while (j<M && texto[i+j] == padrao[j]) {
            j++;
            if (j == M) {
                retorno[ocorr] = i;
                ocorr++;
```

```
12                  }
13              }
14          }
15          return retorno;
16      }
```

7. Considerando um algoritmo de **busca sequencial** e a seguinte tabela ordenada:

2 5 7 11 13 17 25

(a) Diga quantas comparações serão necessárias para procurar cada um dos 7 elementos da tabela?

Considerando o algoritmo apresentado em aula, serão necessárias:

- 2 - 1 comparação
- 5 - 2 comparações
- 7 - 3 comparações
- 11 - 4 comparações
- 13 - 5 comparações
- 17 - 6 comparações
- 25 - 7 comparações

(b) Diga quantas comparações serão necessárias para procurar os seguintes números que não estão na tabela **12, 28, 1, 75, 8**?

Como nenhum dos números está presente na tabela, então o número de comparações é igual para todos, no caso, 7 comparações para descobrir que cada um deles não está presente nos dados.

8. Considerando um algoritmo de **busca binária** e a seguinte tabela ordenada:

2 5 7 11 13 17 25 32 35 39

(a) Diga quantas comparações serão necessárias para procurar cada um dos 10 elementos da tabela? Obs.: considere todas as comparações com operadores relacionais presentes no algoritmo.

Considerando o algoritmo apresentado em aula, serão necessárias:

- 2 - 8 comparações
- 5 - 5 comparações
- 7 - 8 comparações
- 11 - 11 comparações
- 13 - 2 comparações
- 17 - 8 comparações
- 25 - 11 comparações

- 32 - 5 comparações
- 35 - 8 comparações
- 39 - 11 comparações

(b) Diga quantas comparações serão necessárias para procurar os seguintes números que não estão na tabela **12, 28, 1, 75, 8**?

- 12 - 13 comparações
- 28 - 13 comparações
- 1 - 10 comparações
- 75 - 13 comparações
- 8 - 13 comparações

9. Considerando a lógica de funcionamento da **busca binária**, escreva um algoritmo de **busca ternária**, isto é, a cada passo calcular $m1 = n/2$ e $m2 = 2 * n/3$. A tabela então fica dividida em 3 partes (esquerda, meio e direita). Daí basta comparar com **m1** e **m2**. Se não for igual o elemento deve estar em uma das 3 partes. A cada passo a tabela fica dividida por 3. Veja a seguir o algoritmo de busca binária.

```
int buscaBinaria(int vet[], int tam, int chave)
{
    int esquerda = 0;
    int direita = tam-1;
    int meio;
    while (esquerda <= direita)
    {
        meio = (esquerda + direita) / 2;
        if (vet[meio] == chave)
        {
            return meio;
        }
        else if (vet[meio] < chave)
        {
            esquerda = meio+1;
        }
        else
        {
            direita = meio-1;
        }
    }
    return -1;
}
```

```
int buscaTernaria(int vet[], int tam, int chave)
{
    int m1, m2;
    int esq = 0;
    int dir = tam-1;

    do{
```

```
8           m1 = esq + (dir - esq) / 3;
9           m2 = dir - (dir - esq) / 3;
10
11          if(chave == vet[m1])
12              return m1;
13          if(chave == vet[m2])
14              return m2;
15
16          if(chave < vet[m1]){
17              dir = m1-1;
18          }
19          if (chave > vet[m1] && chave < vet[m2]){
20              esq = m1+1;
21              dir = m2 -1;
22          }
23          else if(chave > vet[m2]){
24              esq = m2+1;
25          }
26      } while(esq <= dir);
27      return -1;
28  }
```

10. A **busca ternária** que você implementou é melhor que a busca binária? Justifique.

    A Busca Binária divide a lista em duas partes a cada passo. Complexidade de tempo: $O(log_2 n)$. Já a Busca Ternária divide a lista em três partes a cada passo. Complexidade de tempo: $O(log_3 n)$.
    Como $log_3(n)$ é maior que $log_2(n)$, a busca ternária realiza mais comparações do que a busca binária, sendo assim a busca binária é mais eficiente em termos de número de comparações e complexidade de tempo.

11. Dado a entrada de dados abaixo, faça os seguintes exercícios:

    2 4 7 11 12 20 23 25 32 33 34 44

    (a) Pesquisa sequencial: Quantas comparações são feitas para encontrar ou não as chaves 30, 6, 23, 20, 44.
    - 30 - 12 comparações para descobrir que não existe
    - 6 - 12 comparações para descobrir que não existe
    - 23 - 7 comparações
    - 20 - 6 comparações
    - 44 - 12 comparações

    (b) Pesquisa binária: Quantas comparações são feitas para encontrar ou não as chaves 30, 6, 23, 20, 44. Demonstre a pesquisa feita através de desenhos.

    - 30 - 9 comparações para descobrir que não existe
    - 6 - 10 comparações para descobrir que não existe
    - 23 - 6 comparações
    - 20 - 1 comparação
    - 44 - 7 comparações

12. Dado a entrada de dados abaixo, faça os seguintes exercícios:

C O R R O C O C O G A O C O R

(a) Busca KMP: Quantas comparações são feitas para encontrar ou não as chaves **C O G A** e **G A C O**.
C O G A = 11 comparações
G A C O = 3 comparações para descobrir que não existe.

(b) Busca BMH: Quantas comparações são feitas para encontrar ou não as chaves **C O G A** e **G A C O**.
C O G A = 5 comparações
G A C O = 4 comparações para descobrir que não existe.

13. Qual é a complexidade de tempo no pior caso da busca sequencial em uma lista de n elementos?
    (a) O(log n)
    (b) **O(n)**
    (c) O(n log n)
    (d) O(1)

14. A busca sequencial é mais eficiente que a busca binária em:
    (a) Listas ordenadas
    (b) **Listas desordenadas**
    (c) Qualquer tipo de lista
    (d) Nenhuma das opções acima

15. Qual das seguintes afirmações é verdadeira sobre a busca binária?
    (a) Funciona em listas desordenadas
    (b) Tem complexidade O(n) no pior caso
    (c) **Funciona apenas em listas ordenadas**
    (d) Não depende do tamanho da lista

16. Qual é a principal desvantagem da busca por força bruta em strings?
    (a) Complexidade de espaço
    (b) Não encontra a substring
    (c) **Complexidade de tempo elevada para strings longas**
    (d) Requer a string ser ordenada

17. O algoritmo KMP é eficiente porque:
    (a) Sempre encontra a substring na primeira tentativa
    (b) **Utiliza um pré-processamento para evitar comparações redundantes**
    (c) Não depende do comprimento da substring
    (d) É mais simples de implementar do que a busca por força bruta