Aula 17

MC 102 - Algoritmos e Programação de Computadores

Algoritmos de Busca e de Ordenação

Ordenação

Muitas vezes, precisamos ordernar dados para posteriormente trabalharmos com estes dados de uma forma organizada.

Exemplos:

Nomes em ordem alfabética
Alunos por ordem do RA ou Nota

Ordenação

A forma mais trivial de se ordenar um vetor de N elementos é gerar todas as N! permutações e escolher uma que esteja ordenada.

Claro que esse algoritmo consome muito tempo para resolver este problema. Veremos algumas soluções mais interessantes.

Essas soluções utilizam duas operações principais. São elas:

- *Comparar dois itens*
- *Trocar dois itens ou copiar um item*

Ordenação por Bolha(Bubble Sort)

Este algorimto simula o processo de bolhas de gás em líquido em que bolhas trocam de posição com o líquido até que o equilíbrio é atingido.

Seguindo o mesmo princípio, inicialmente percorre-se o vetor a da esquerda para a direita, **comparando** pares de elementos consecutivos, **trocando** de lugar os que estão fora da ordem. Em cada troca, o maior elemento é deslocado uma posição para a direita.

Ordenação por Bolha

Para um vetor de N elementos, N-1 passos serão necessários para que “trocas” entre elementos adjacentes sejam realizadas.

A cada rodada é garantido que um elemento a mais está em sua posição correta.

Ordenação por Bolha

Varredura	V[0]	V[1]	V[2]	V[3]	Troca
1	10	9	7	6	V[0] e V[1]
	9	10	7	6	V[1] e V[2]
	9	7	10	6	V[2] e V[3]
	9	7	6	10	Fim da varr. 1

Após a primeira varredura o maior elemento do vetor V[0],V[1],V[2] encontra-se alocado em sua posição definitiva. Vamos deixá-lo de lado e efetuar a segunda varredura no subvetor V[0],V[1].V[2].

Ordenação por Bolha

Varredura	V[0]	V[1]	V[2]	V[3]	Troca
2	9	7	6	10	V[0] e V[1]
	7	9	6	10	V[1] e V[2]
	7	6	9	10	Fim da varr. 2

Após a segunda varredura o maior elemento do subvetor V[0],V[1],V[2] encontra-se alocado em sua posição definitiva. O próximo subvetor a ser ordenado é V[0],V[1].

Ordenação por Bolha

Varredura	V[0]	V[1]	V[2]	V[3]	Troca
3	7	6	9	10	V[0] e V[1]
	6	7	9	10	Fim da varr. 3

BubbleSort

```
void BubbleSort(int vetor[], int n) {
  int i,j,aux;

  for(i=N-1;i>0;i--)
    for(j=0;j<i;j++)
      if(vetor[j] > vetor[j+1]) {
        aux = vetor[j];
        vetor[j] = vetor[j+1];
        vetor[j+1] = aux;
      }
}
```


BubbleSort2

```
void BubbleSort2(int vetor[], int n) {
  int i,j,aux,troca;
  for(i=N-1;i>0;i--) {
    troca = 0;
    for(j=0;j<i;j++)
      if(vetor[j] > vetor[j+1]) {
        aux = vetor[j];
        vetor[j] = vetor[j+1];
        vetor[j+1] = aux;
        troca = 1;
      }
    if (troca==0) break;
}
```

Se nenhuma troca é realizada em um determinado passo, então todos os elementos já se encontram ordenados.

Análise do BubbleSort

Quanto tempo a ordenação por bolha demora para executar? Em outras palavras, quantas vezes as operações de comparação e de troca de posição de elementos são executadas?

No pior caso, o algoritmo realizará N-1 trocas para o primeiro passo, depois N-2 trocas para o segundo, e assim por diante até 1 troca.

(Cite um exemplo para este pior caso)

Trocas = N-1 + N-2 + N-3 + ... + 2 + 1

Aproximadamente N^2 *Trocas*

Análise do BubbleSort

Quanto tempo a ordenação por bolha demora para executar? Em outras palavras, quantas vezes as operações de comparação e de troca de posição de elementos são executadas?

No melhor caso, nenhuma troca será realizada.

(Cite um exemplo para este melhor caso)

Em ambos os casos, BubbleSort faz da ordem de N^2 comparações, apenas BubbleSort2 faz **N-1** comparações no melhor caso.

Ordenação por Seleção (Selection Sort)

A ordenação por seleção (SelectionSort) consiste, em cada etapa, em selecionar o menor (ou o maior) elemento e alocá-lo em sua posição correta dentro do futor vetor ordenado.

Também realiza N-1 passos, mas em cada passo apenas uma troca é efetuada. Essa troca garante que um elemento está na sua posição final no vetor ordenado.

Ordenação por Seleção

Etapa	V[0]	V[1]	V[2]	V[3]	Troca
1	10	9	7	6	V[0] e V[3]
2	6	9	7	10	V[1] e V[2]
3	6	7	9	10	Fim

na ordenação por seleção a lista com m registros fica decomposta em dois subvetores, uma contendo os elementos já ordenados e a outra com os restantes ainda não ordenados.

No início o subvetor ordenado é vazia e a outra contém todos os demais. No final do processo o vetor ordenado apresentará n − 1elementos e o outro apenas 1

SelectionSort

```
void SelectionSort(int vetor[], int n) {
  int i,j,i_menor,min;

  for(i=0;i<n-1;i++) {
    i_menor = i;
    for(j=i+1;j<n;j++)
      if(vetor[j] < vetor[i_menor])
        i_menor = j;

    min = vetor[i_menor];
    vetor[i_menor] = vetor[i];
    vetor[i] = min;
  }
}
```

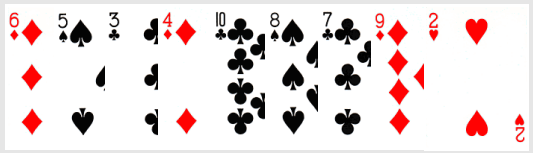

Ordenação por Inserção

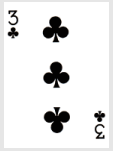
A ordenação por inserção (InsertionSort) baseia-se na idéia de semi-ordem, i.e, "*é mais simples organizar algo que já está arrumado do que ordenar uma grande bagunça*".

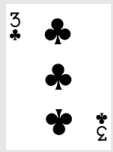
Considerando isso, divide-se o vetor em dois grupos de elementos: os ordenados, no início do vetor, e os não ordenados no final. Em N-1 passos, selecionamos cada elemento do grupo não ordenado par ser inserido no grupo ordenado.

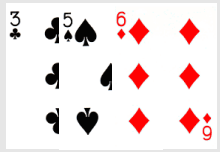
Semelhante ao que fazemos para manter ordenadas as cartas de baralho nas mãos.

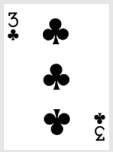
Insertion Sort

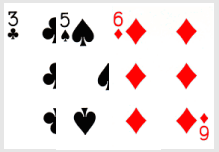


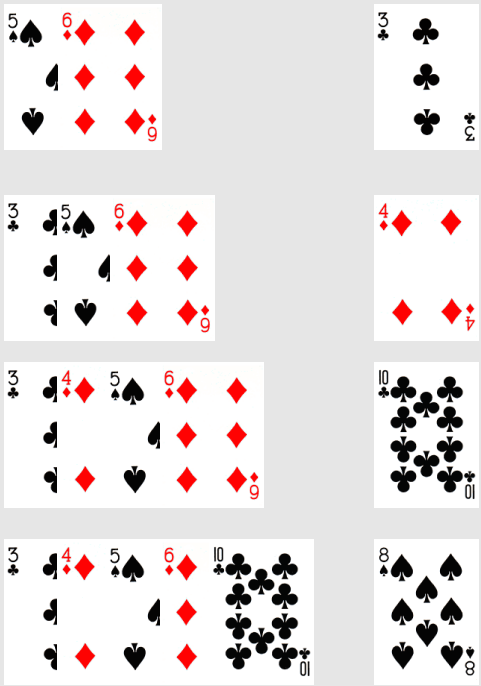


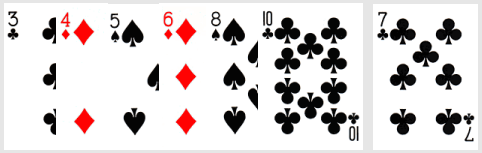


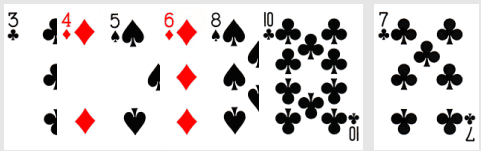


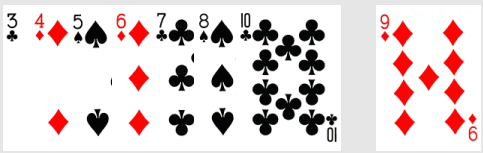


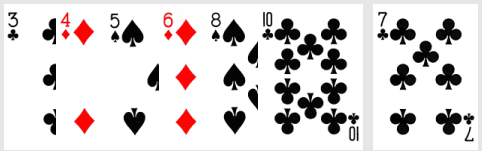


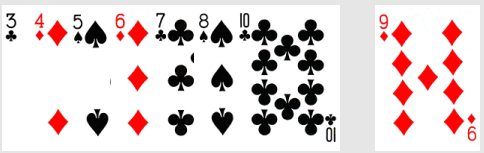


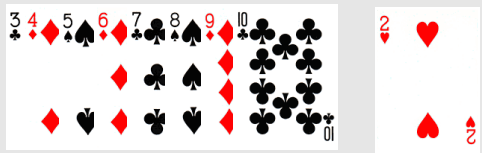


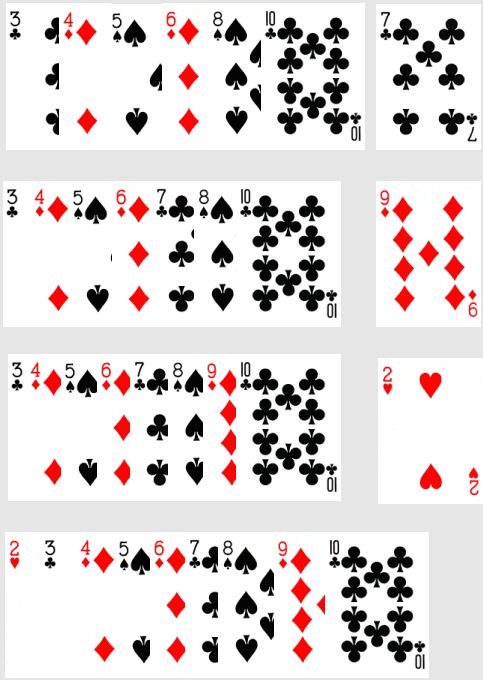












Ordenação por Inserção

No início, o subvetor ordenado é formado trivialmente apenas pelo primeiro elemento da lista.

A cada etapa i , o i -ésimo elemento é inserido em seu lugar apropriado entre os i − 1 elementos já ordenados. Os índices dos itens a serem inseridos variam de 2 a tamanho.

Etapa	V[0]	V[1]	V[2]	V[3]
1	10	9	7	6
2	9	10	7	6
3	7	9	10	6
4	9	7	6	10

InsertionSort

```
void InsertionSort(int vetor[], int n) {
  int i,j,chave;

  for(i=1;i<n;i++) {
    chave = vetor[i];
    j=i-1;
    while ((j >= 0) && (vetor[j]>chave)) {
      vetor[j+1] = vetor[j];
      j--;
    }
    vetor[j+1] = chave;
  }
}
```


Análise do Insertion Sort

Quanto tempo a ordenação por inserção demora para executar? Em outras palavras, quantas vezes as operações de comparação e de troca de posição de elementos são executadas?

No pior caso, se assemelha ao BubbleSort, onde são realizadas aproximadamente N^2 trocas.

No melhor caso, nenhuma troca é realizada.

No melhor caso, InsertionSort faz da ordem de **N** comparações. Já no pior caso são N^2 comparações.

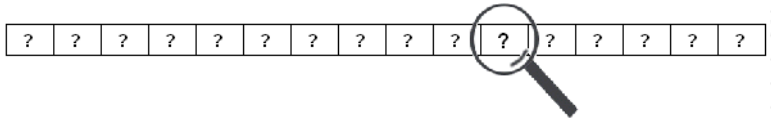
Na média, tem melhor desempenho em relação ao BubbleSort e ao SelectionSort

Busca de valores

Dado um conjunto de valores, como podemos verificar se um determinado valor está presente neste conjunto e, caso esteja, qual a sua posição no conjunto?

Por exemplo:

Encontre o RA 070102 em um vetor de RA dos alunos de uma determinada turma



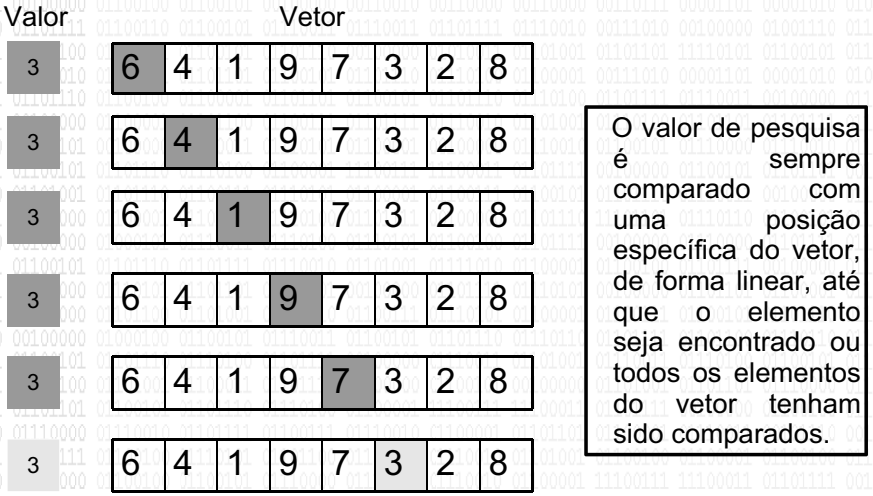
Busca Linear

Uma forma simples de pesquisar elementos em um conjunto de valores (por exemplo, um vetor) é comparar cada elemento com o valor de pesquisa, caracterizando uma **busca seqüencial** ou também chamada **busca linear** (**LinearSearch**).

Dado um vetor de N elementos e um valor de pesquisa x, encontrar a posição de x no vetor

```
Para i = 1 até N
   Se vetor[i] = x
      retorna i
retorna -1
```


Busca Linear



O valor de pesquisa é sempre comparado com uma posição específica do vetor, de forma linear, até que o elemento seja encontrado ou todos os elementos do vetor tenham sido comparados.

Busca Linear

```
int LinSearch(int vetor[], int n, int x) {
  int i;

  for(i=0;i<n;i++) {
    if(vetor[i] == x) {
      return i;
    }
  }
  return -1;
}
```

Análise da Busca Linear

Quantas vezes as operações de comparação de elementos são executadas?

No pior caso, N comparações serão executadas.

(Dê um exemplo para o pior caso)

No melhor caso, apenas uma comparação será executada.

(Dê um exemplo para o melhor caso)

Em geral, podemos dizer que a busca linear realiza em torno de **N** comparações. Se cada elemento do vetor tiver que ser pesquisado (lista de chamada), serão executadas na ordem de $\mathbf{N^2}$ comparações.

Busca Binária

Para reduzir o número de comparações em uma pesquisa de um elemento em um vetor, podemos pensar em utilizar um **vetor ordenado** e realizar uma busca mais inteligente.

A busca binária utiliza a técnica de divisão e conquista, da sequinte forma:

1. Divide-se o vetor em 2 partes, com um elemento pivô entre essas partes (o elemento do meio).
2. Se o valor de busca for igual ao pivô então o valor foi encontrado

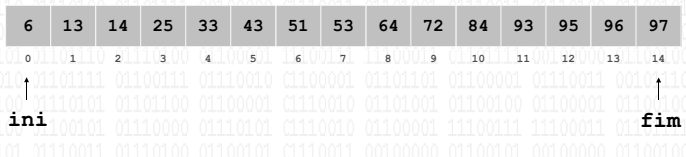
   Senão, se o valor de busca for inferior ao pivô, repete-se a busca agora para o subvetor inferior ao pivô.

   Senão, repete-se a busca para o subvetor superior ao pivô.

Busca Binária

Entrada:
Vetor Ordenado e Valor x (**neste caso, 33**)
Algoritmo

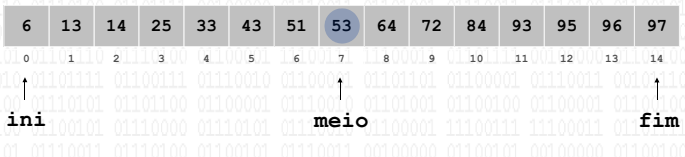
```
Enquanto (ini <= fim)
    meio = (fim + ini) / 2
    Se vetor[meio] = x
        retorna meio
    senão, se x < vetor[meio]
        fim = meio-1
    senão
        ini = meio+1
retorna -1
```



Busca Binária

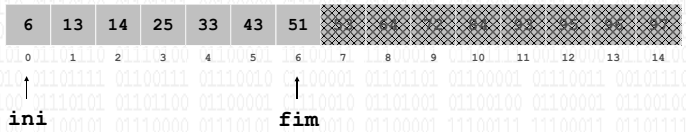
Entrada:
Vetor Ordenado e Valor x (**neste caso, 33**)
Algoritmo

```
Enquanto (ini <= fim)
    meio = (fim + ini) / 2
    Se vetor[meio] = x
        retorna meio
    senão, se x < vetor[meio]
        fim = meio-1
    senão
        ini = meio+1
retorna -1
```



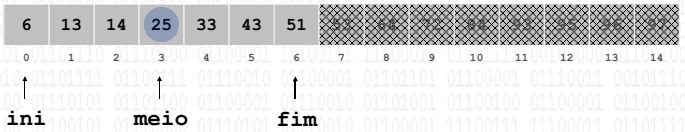
Busca Binária

Entrada:
Vetor Ordenado e Valor x (**neste caso, 33**)
Algoritmo
Enquanto (ini <= fim)
meio = (fim + ini) / 2
Se vetor[meio] = x
retorna meio
senão, se x < vetor[meio]
fim = meio-1
senão
ini = meio+1
retorna -1



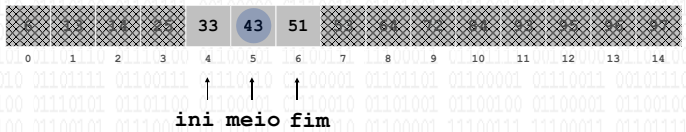
Busca Binária

Entrada:
Vetor Ordenado e Valor x (**neste caso, 33**)
Algoritmo
Enquanto (ini <= fim)
meio = (fim + ini) / 2
Se vetor[meio] = x
retorna meio
senão, se x < vetor[meio]
fim = meio-1
senão
ini = meio+1
retorna -1



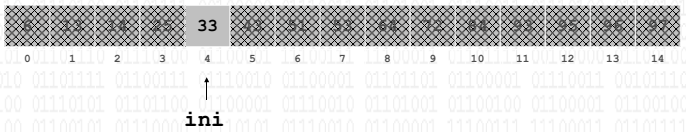
Busca Binária

Entrada:
Vetor Ordenado e Valor x (**neste caso, 33**)
Algoritmo
Enquanto (ini <= fim)
meio = (fim + ini) / 2
Se vetor[meio] = x
retorna meio
senão, se x < vetor[meio]
fim = meio-1
senão
ini = meio+1
retorna -1



Busca Binária

Entrada:
Vetor Ordenado e Valor x (**neste caso, 33**)
Algoritmo
Enquanto (ini <= fim)
meio = (fim + ini) / 2
Se vetor[meio] = x
retorna meio
senão, se x < vetor[meio]
fim = meio-1
senão
ini = meio+1
retorna -1



Busca Binária

Entrada:
Vetor Ordenado e Valor x (**neste caso, 33**)
Algoritmo

```
Enquanto (ini <= fim)
    meio = (fim + ini) / 2
    Se vetor[meio] = x
        retorna meio
    senão, se x < vetor[meio]
        fim = meio-1
    senão
        ini = meio+1
retorna -1
```

Busca Binária

```
int BinSearch(int vetor[], int n, int x) {
  int ini, meio, fim;

  ini = 0;
  fim = n-1;
  while (ini <= fim) {
    meio = (fim + ini)/2;
    if (vetor[meio]==x)
      return meio;
    else if (x < vetor[meio])
      fim = meio - 1;
    else
      ini = meio + 1;
  }
  return -1;
}
```


Análise da Busca Binária

Quantas vezes as operações de comparação de elementos são executadas?

No pior caso, **$\log_2 N$** comparações serão executadas.

(Dê um exemplo para o pior caso)

No melhor caso, apenas uma comparação será executada.

(Dê um exemplo para o melhor caso)

Busca binária realiza, em geral, menos de **$\log_2 N$** comparações.

Se cada elemento do vetor tiver que ser pesquisado (lista de chamada), serão executadas na ordem de **$N * \log_2 N$** comparações, bem inferior à Busca Linear.