FIAP ON

ALGORITMOS

# ORDENAÇÃO DE VETORES

6

## LISTA DE FIGURAS

Figura 6.1 – Algoritmo I ..... 7
Figura 6.2 – Simulação de Esgotamento ..... 9
Figura 6.3 – Simulação de Esgotamento ..... 10
Figura 6.4 – Ordenação de elementos ..... 11
Figura 6.5 – Segunda iteração ..... 12
Figura 6.6 – Método posicional ..... 13
Figura 6.7 – Simulação usando seleção por esgotamento ..... 14
Figura 6.8 – Seleção por seleção convencional ..... 15
Figura 6.9 – Algoritmo II ..... 16
Figura 6.10 – Vetor com 7 valores ..... 17
Figura 6.11 – Segundo ciclo de comparações ..... 18
Figura 6.12 – Terceiro ciclo de comparações ..... 19
Figura 6.13 – Quarto ciclo de comparações ..... 19
Figura 6.14 – Algoritmo III ..... 20
Figura 6.15 – Algoritmo IV ..... 23
Figura 6.16 – Algoritmo V ..... 24
Figura 6.17 - O pior caso ..... 28
Figura 6.18 – Algoritmo VI ..... 30
Figura 6.19 – Algoritmo VII ..... 31
Figura 6.20 – Algoritmo VIII ..... 32
Figura 6.21 – Algoritmo IX ..... 32
Figura 6.22 – Algoritmo X ..... 33
Figura 6.23 – Simulação do Shell Sort ..... 34
Figura 6.24 – Simulação do Shell Sort (2) ..... 35
Figura 6.25 – Algoritmo XI ..... 36
Figura 6.26 – Algoritmo XII ..... 39

**SUMÁRIO**

6 ORDENAÇÃO DE VETORES ..... 4
6.1 Introdução ..... 4
6.2 Método da seleção simplificado ou do esgotamento de possibilidades ..... 5
6.2.1 Estratégia ..... 5
6.2.2 Simulação ..... 8
6.2.3 Compreendendo o processo de ordenação ..... 10
6.3 Método da Seleção ..... 12
6.4 Bubble sort ..... 15
6.5 Bubble sort variante shake ..... 20
6.6 Exercícios ..... 21
6.7 Ordenação de vetores (métodos elementares de alto desempenho). ..... 22
6.7.1 Método da inserção ou do baralho ..... 22
6.8 Aplicando análise algorítmica no método da inserção ..... 26
6.8.1 Buscando o pior caso ..... 27
6.9 Shell sort ..... 29
6.9.1 Simulação do shell sort ..... 34
6.10 Exercícios ..... 36
REFERÊNCIAS ..... 40

# 6 ORDENAÇÃO DE VETORES

## 6.1 Introdução

Uma das mais notáveis aplicações usando vetores é a ordenação de uma série de elementos de mesma natureza. É aquilo que popularmente chamamos de fila, embora essa nomenclatura se aplique ao mesmo conceito acadêmico, apenas com uma definição mais precisa.

Então, comecemos definindo o que é uma ordenação de uma série de pessoas (por exemplo, uma lista alfabeticamente ordenada de empresas ou pessoas), de números, de datas, entre outras inúmeras possibilidades.

A ordenação é, então, uma operação que visa rearranjar os elementos de uma lista, ou no caso em questão, um vetor, sempre em alguma ordem determinada.

Do ponto de vista da utilização de computador para realização dessa tarefa, os métodos são classificados quanto à possibilidade de carga dos dados na memória principal (ordenação interna) e os que tratam dos dados armazenados em arquivos em disco (ordenação externa).

Neste curso, temos especial interesse em utilizar esses algoritmos para apresentar aplicações de vetores, portanto serão analisados somente métodos de ordenação interna.

Existem inúmeras técnicas de ordenação, e nosso primeiro convite é visitar o YouTube, buscando expressões como “Bubble Sort”, “Insert Sort”, “Shell Sort”, entre inúmeras outras possibilidades, para encontrar ordenações exemplificadas por danças típicas, cartas de baralho e até mesmo usando o famoso brinquedo “Lego”.

Vamos tratar das ordenações elementares com baixo desempenho (Seleção Esgotamento; Seleção Convencional; Bubble Sort e uma variante problemática do Bubble Sort) nesse tópico e no seguinte os métodos simples de alto desempenho (Inserção e Shell Sort).

### 6.2 Método da seleção simplificado ou do esgotamento de possibilidades

Talvez esse seja o método mais conhecido, embora cause muita estranheza nos estudantes por sequer ser citado em algumas publicações ou ser citado em outras como o próprio Bubble Sort, de quem é “parente” próximo, até de ser o método da Seleção, outro “parente”, na melhor das hipóteses.

É o menos performático dos métodos, mas também por isso, o mais simples de todos. Confirma-se aqui, novamente, a velha máxima inglesa que afirma “todo problema difícil tem uma solução simples, que está errada”.

Na verdade, embora “funcione”, o método que será descrito exige inúmeras e desnecessárias comparações, trocas e escrita de valores que somente comprometem a solução. Funcionar, nesse caso, equivale a termos dois ônibus, com o mesmo conforto, a mesma velocidade, segurança e todos os demais aspectos, porém diferindo apenas num “pequeno” detalhe: enquanto o primeiro ônibus faz o trajeto custando 1 centavo, o segundo realiza o mesmo trajeto custando “apenas” 100 reais. Com certeza, a menos que estivermos muito enganados, jamais pegaríamos esse “ônibus”.

#### 6.2.1 Estratégia

Toda rotina de ordenação é baseada em alguma estratégia. Nesse caso, a estratégia consiste em localizar o menor termo e deixá-lo na primeira posição, o segundo na segunda posição e assim por diante, caso desejarmos ordenar o vetor em ordem crescente. Se a ordem fosse decrescente, bastaria apenas inverter a situação, isto é, o maior na primeira posição, o segundo maior na segunda posição e assim por diante. Trata-se, portanto de um método **posicional**, pois sua estratégia consiste em focar numa posição e garantir que a mesma termine, depois de um ciclo de comparações, com o número que lá deveria estar.

Nesse e nos demais problemas de ordenação vamos admitir que todos elementos a serem ordenados sejam distintos entre si. Na verdade, se não o forem, não teremos nenhum problema, pois os algoritmos ordenarão a série, deixando os iguais juntos.

Exemplificando:

Se nossa hipotética série precisar ordenar de forma crescente os seguintes números: 13;12;15;14 e 11, a resposta será 11;12;13;14 e 15. Já se a série for: 13;15;12;13 e 15, nossa resposta será 12;13;13;15 e 15. Apenas para facilidade na compreensão dos algoritmos usaremos sempre elementos distintos entre si, mas não há qualquer impedimento para que simulemos com uma ou mais repetições.

Vamos também admitir que desejamos ordenar nossas séries de forma crescente, mesmo porque não teríamos qualquer problema se pensarmos em ordenar séries de forma decrescente e também porque parece bem mais “natural” a ordenação do menor para o maior, como fazemos nas contagens, tabuadas, entre outras situações.

Em síntese, nessa estratégia vamos buscar selecionar o elemento certo para uma posição base que esteja sendo “avaliada”.

A estratégia, portanto, é baseada no princípio, em se passar sempre o menor valor do vetor para a primeira posição, depois o de segundo menor valor para a segunda posição, e assim sucessivamente, com os (n-1) elementos restantes, até os últimos dois elementos. Note-se que quando o penúltimo elemento for “ordenado” o último também o será.

Vamos agora ao algoritmo, propriamente dito:

```
algoritmo sort_esgotamento;
variáveis
      a, b, i, j, n, c : inteiro;
      v : matriz[100] de inteiros;
fim-variáveis
início
      imprima("Total de Números: ");
      n := leia();
      para i de 1 até n faça
        imprima("Digite o ",i,"o. Numero: ");
        v[i] := leia();
    fim-para
      para i de 1 até n-1 faça
         para j de i+1 até n faça
            se v[i] > v[j] então
                   c := v[i];
                   v[i] := v[j];
                   v[j] := c;
            fim-se
        fim-para
    fim-para
      para i de 1 até n faça
        imprima(v[i]);
    fim-para
fim
```

Figura 6.1 – Algoritmo I
Fonte: Elaborado pelo autor (2015)

Várias são as observações que se fazem necessárias:

- Nota-se que não há qualquer preocupação com comparações desnecessárias, comprovando a ineficiência desse método, bastando para tanto observarmos que se a série a ser ordenada já estivesse ordenada, o método faria o mesmo “esforço” que faria se tivéssemos uma série parcialmente ordenada, com distribuição aleatória de números ou ordenada de forma inversa a que gostaríamos, situação que intuitivamente percebemos como a pior possível.
- Vamos supor que estamos comparando dois elementos e o elemento que deveria ocupar a posição menor seja na verdade maior. Imaginemos que estamos comparando o primeiro com o segundo número da série

> “13;12;15;14 e 11”. Ora, “13” e “12” devem trocar de lugar. Mas, notemos que o “11” está no final da fila.

Facilmente percebemos que, encontrada essa situação, o elemento que estiver na posição base (13, no nosso caso), no algoritmo estará na posição “v[i]”, a ser comparado ao elemento que estiver na posição “v[j]” (12, no nosso caso), portanto a troca deverá ser feita.

Olhando um pouco adiante, facilmente intuímos que nosso algoritmo acabará comparando “12” e “11” e, quando o fizer, deixará o “11” na primeira posição, a correta. Mas, o “12” que estava exatamente na posição em que deveria estar (a segunda), acabará sendo transferido para a última posição.

Como nosso algoritmo não se preocupou em tempo algum com o que estava ocorrendo com cada elemento separadamente, não surpreende que para “acertar” uma posição, simplesmente atrapalhemos todas as demais posições.

Um importante aspecto a ser lembrado é o que chamamos tecnicamente de análise algorítmica. Faremos uma análise mais rigorosa desse conceito em outro momento, todavia, agora, o mais importante é percebermos que nosso algoritmo se comporta dentro de uma técnica chamada de “gulosa”.

Basicamente, todo algoritmo “guloso” se importa apenas com o problema que está resolvendo naquele momento, sem se importar com qualquer outro que venha porventura causar. Dentro das rotinas de ordenação, a solução mais “gulosa” é exatamente a apresentada, pois para colocar o menor número na primeira posição, se for necessário “destruir” uma ordenação pré-existente, nosso algoritmo o fará sem qualquer constrangimento.

### 6.2.2 Simulação

Vamos à importante simulação, ou teste de mesa, desse algoritmo. Note que para aprender a programar precisaremos antever o que nosso “programa fará”. Só treinando arduamente simulação é que poderemos entender isso de fato, em sua totalidade.

| N | I | J | C |
|---|---|---|---|
| 7 | 1 | 2 | 13 |
| | | 3 | |
| | | 4 | |
| | | 5 | |
| | | 6 | |
| | | 7 | |
| | 2 | 3 | 13 |
| | | 4 | |
| | | 5 | |
| | | 6 | |
| | | 7 | |
| | 3 | 4 | |
| | | 5 | |
| | | 6 | |
| | | 7 | |
| | 4 | 5 | |
| | | 6 | |
| | | 7 | |
| | | 5 | |
| | | 6 | |
| | | 2 | |
| | 6 | 7 | |

Figura 6.2 – Simulação de Esgotamento
Fonte: FIAP (2015)

| V | Inicial | | | | | | | | | Final |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 13 | 12 | 11 | | | | | | | 11 |
| 2 | 12 | 13 | | 12 | | | | | | 12 |
| 3 | 11 | | 12 | 13 | | | | | | 13 |
| 4 | 17 | | | | 14 | | | | | 14 |
| 5 | 14 | | | | 17 | 16 | 15 | | | 15 |
| 6 | 16 | | | | | 17 | | 16 | | 16 |
| 7 | 15 | | | | | | 16 | 17 | | 17 |

| I=1 ; J=2 | I=1 ; J=3 | I=2 ; J=3 | I=4 ; J=5 | I=5 ; J=6 | I=5 ; J=7 | I=6 ; J=7 |
|---|---|---|---|---|---|---|

Figura 6.3 – Simulação de Esgotamento
Fonte: FIAP (2015)

Antes de tudo, vamos entender o que significa cada coisa em nossa simulação.

Na figura 6.2 são apresentados os valores assumidos pelas variáveis *n*, que representam o número de elementos a serem ordenados. *I* e *j* são nossos contadores. *I* controla a posição base a ser comparada com todas as demais, trabalho este executado por nosso contador *j*. *C* é nossa variável usada para troca de cada elemento que estiver sendo trocado. Até aí, nada demais.

Na figura 6.3 temos nosso vetor *V;* e em vermelho, temos indicadas todas as posições do vetor e também os valores de *i* e *j* no momento das trocas. Em preto, temos os valores que cada posição do vetor vai assumindo, no desenrolar do processo de análise e troca de valores, necessário à ordenação.

Como o vetor tem vários valores, parece muito mais natural “atualizarmos” esses valores da esquerda para a direita. Com efeito, a posição 1 (um) do vetor conta, inicialmente, com o número *13.* Em seguida, assume o valor *12* para, finalmente, assumir seu valor definitivo, isto é, *11.*

### 6.2.3 Compreendendo o processo de ordenação

A estratégia adotada trata-se, provavelmente, da mais simples já concebida. Consiste em fixarmos posição por posição, garantindo em cada etapa que o maior ou menor, a depender de nossa necessidade, o elemento necessário seja localizado e armazenado na posição adequada.

Inicialmente, percorre-se a lista da esquerda para a direita, **comparando-se pares de elementos**, trocando de lugar os que estiverem fora de ordem.

Assim, a cada “varredura” temos o deslocamento do elemento adequado para sua posição definitiva, portanto podemos deixá-lo de lado e continuar o processo com os demais elementos.

Analisemos o quadro apresentado a seguir, supondo que precisamos ordenar 7 elementos, a saber: 2; 1; 4; 5; 9; 8 e 7.

| Varredura | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | Troca |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 4 | 5 | 9 | 8 | 7 | 1 e 2 |
| | 1 | 2 | 4 | 5 | 9 | 8 | 7 | não |
| | 1 | 2 | 4 | 5 | 9 | 8 | 7 | não |
| | 1 | 2 | 4 | 5 | 9 | 8 | 7 | não |
| | 1 | 2 | 4 | 5 | 9 | 8 | 7 | não |
| | 1 | 2 | 4 | 5 | 9 | 8 | 7 | não |
| | 1 | 2 | 4 | 5 | 9 | 8 | 7 | Fim da 1ª varredura |

Figura 6.4 – Ordenação de elementos
Fonte: FIAP (2015)

Na primeira varredura, quando i=1; j de 2 até n; observamos que o “1” sai da segunda posição e vai para a primeira, seu lugar definitivo.

Notamos ainda que o “2” já fica na segunda posição, mas isso foi causado apenas pela “sorte” de o 2 estar justamente na primeira posição. Se estivesse em posição mais adiantada, isso não ocorreria.

Já o “9” é mantido no lugar onde estava, assim como todos os demais elementos, uma vez que esse método se preocupa, tão somente, com a posição corrente.

Vamos analisar agora a segunda iteração, ou seja, i=2; j de 3 até n; observamos que nada é feito!

| Varredura | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | Troca |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 5 | 9 | 8 | 7 | não |
| 2 | 1 | 2 | 4 | 5 | 9 | 8 | 7 | não |
| | 1 | 2 | 4 | 5 | 9 | 8 | 7 | não |
| | 1 | 2 | 4 | 5 | 9 | 8 | 7 | não |
| | 1 | 2 | 4 | 5 | 9 | 8 | 7 | não |
| | 1 | 2 | 4 | 5 | 9 | 8 | 7 | não |
| | 1 | 2 | 4 | 5 | 9 | 8 | 7 | Fim da 1ª varredura |

Figura 6.5 – Segunda iteração
Fonte: FIAP (2015)

Isso ocorre porque na segunda posição já temos o segundo menor número. Assim, todas as comparações dessa série de iterações não realizam trocas. Na verdade, não apenas quando i=2, mas também quando i=3; i=4; i=5! Somente quando i=6 o processo de ordenação é **efetivamente** retomado.

Vamos nos lembrar disso mais adiante, quando discutirmos o Bubble Sort, um método que não comete a mesma “gafe” que o Esgotamento.

- Simule para 5 valores, sendo os valores 15, 14, 13, 12 e 11. Análise o esforço feito pelo algoritmo.
- Simule para 5 valores, sendo os valores 11, 12, 13, 14 e 15. Análise o esforço feito pelo algoritmo.
- Compare os esforços tanto para a situação em que a lista estava invertida, como para a situação em que a lista estava ordenada. Não simule, mas avalie o esforço necessário para simular uma série que contivesse 13 números.

## 6.3 Método da Seleção

Nada mais é que o método do esgotamento, porém em vez de “sair trocando” elementos arbitrariamente, o método localiza o elemento a ser “trocado” e guarda sua posição **sem fazer troca alguma**. Somente depois de comparados todos os elementos candidatos a ocuparem determinada posição, a troca é executada e somente relativa à posição observada. Isto é, se apenas a primeira posição estiver

“fora de ordem”, esta será ordenada e nenhuma outra troca será feita. Ou seja, uma sensível melhoria, quando comparado ao método anterior, do qual é derivado, mas muito mais eficiente.

Trata-se, portanto de um método **posicional**, com todos os problemas que esse tipo de estratégia usualmente acarreta.

```
algoritmo sort_selecao;
variáveis
      min, i, j, n, c : inteiro;
      v : matriz[100] de inteiros;
fim-variáveis
início
      imprima("Total de Números: ");
      n := leia();
      para i de 1 até n faça
        imprima("Digite o ",i,"o. Numero: ");
         v[i] := leia();
    fim-para
      para i de 1 até n-1 faça
        min := i;
         para j de i+1 até n faça
            se v[j] < v[min] então
                  min := j;
            fim-se
        fim-para
         c := v[min];
         v[min] := v[i];
         v[i] := c;
    fim-para
    para i de 1 até n faça
          imprima(v[i]);
    fim-para
fim
```

Figura 6.6 – Método posicional
Fonte: Elaborado pelo autor (2015)

Esse método é baseado em passar sempre o menor valor do vetor para a primeira posição (ou o maior, dependendo da ordem requerida), depois o de segundo menor valor para a segunda posição, e assim sucessivamente com os (n-1) elementos restantes, até os últimos dois elementos.

A simulação bastante parecida, mas com uma sensível e notável distinção: não são feitas trocas desnecessárias.

Vamos retomar a seleção por esgotamento com a seguinte série de números: 4;3;6;2;9;1;7. Submetendo-o ao algoritmo de ordenação por esgotamento ao final da primeira iteração do contador “i” (que leva “j” de 2 até “n”, no caso 7) teremos a seguinte situação:

Simulação usando seleção por esgotamento

| Varredura | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | Troca |
|---|---|---|---|---|---|---|---|---|
| | 4 | 3 | 6 | 2 | 9 | 1 | 7 | 4 e 3 |
| 1;2 | 3 | 4 | 6 | 2 | 9 | 1 | 7 | não |
| 1;3 | 3 | 4 | 6 | 2 | 9 | 1 | 7 | não |
| 1;4 | 2 | 4 | 6 | 3 | 9 | 1 | 7 | 3 e 2 |
| 1;5 | 2 | 4 | 6 | 3 | 9 | 1 | 7 | não |
| 1;6 | 1 | 4 | 6 | 3 | 9 | 2 | 7 | 2 e 1 |
| 1;7 | 1 | 4 | 6 | 3 | 9 | 2 | 7 | Fim da 1ª varredura |

Figura 6.7 – Simulação usando seleção por esgotamento
Fonte: FIAP (2015)

Notamos que foram feitas três trocas desnecessárias. Além disso, o número 2, que estava apenas duas posições adiante da posição definitiva, ficou agora quatro posições adiante. Trata-se de um método essencialmente guloso, como se chama esse tipo de estratégia tecnicamente. De fato, ele coloca o número 1 na primeira posição. Mas a que custo?

O custo, nesse caso, foi deslocar de maneira totalmente aleatória toda uma série de elementos.

Vamos fazer a mesma simulação, porém com o método da seleção clássico, que resultaria na seguinte simulação:

| Seleção por Seleção Convencional | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Varredura | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | Min |
| | 4 | 3 | 6 | 2 | 9 | 1 | 7 | 0 |
| 1;2 | 4 | 3 | 6 | 2 | 9 | 1 | 7 | 2 |
| 1;3 | 4 | 3 | 6 | 2 | 9 | 1 | 7 | 2 |
| 1;4 | 4 | 3 | 6 | 2 | 9 | 1 | 7 | 4 |
| 1;5 | 4 | 3 | 6 | 2 | 9 | 1 | 7 | 4 |
| 1;6 | 4 | 3 | 6 | 2 | 9 | 1 | 7 | 6 |
| 1;7 | 4 | 3 | 6 | 2 | 9 | 1 | 7 | 6 |

Figura 6.8 – Seleção por seleção convencional
Fonte: FIAP (2015)

Facilmente observamos que não houve troca alguma **durante** o processo de análise que o algoritmo faz. A única ação que ocorrerá, findada a iteração apresentada anteriormente será a troca do elemento que está na posição um (no caso, o número 4) com o elemento da sexta posição, no caso o número 6.

Além de não desperdiçar tempo com trocas inúteis, o algoritmo também não desorganiza uma série que tenha elementos em posições próximas a que deveria se encontrar. Aliás, essa é uma situação mais comum do que se pensa, pois muitas séries já possuem uma ordenação pré-estabelecida por ordenações anteriores.

## 6.4 Bubble sort

O chamado método da bolha, diferentemente dos métodos posicionais, não olha apenas uma posição, mas todas. Assim, não é um método guloso e se aproveita de uma eventual ordenação pré-existente.

Sua estratégia consiste em fazer “os mais pesados descerem e os mais leves subirem”, tal qual ocorre com uma bolha de ar de um champagne, água mineral gasosa ou de um refrigerante. Ou seja, líquido na parte baixa do copo e bolhas de ar na parte superior.

Vamos ao algoritmo, então:

```
algoritmo bubble_sort;
variáveis
      min, i, j, n, c, u, iu : inteiro;
      v : matriz[100] de inteiros;
fim-variáveis
início
    imprima("Total de Números: ");
      n := leia();
      para i de 1 até n faça
        imprima("Digite o ",i,"o. Numero: ");
         v[i] := leia();
    fim-para
      u := n;
      enquanto u > 1 faça
        iu := 0;
         para j de 1 até u-1 faça
            se v[j] > v[j+1] então
                    c := v[j];
                    v[j] := v[j+1];
                    v[j+1] := c;
                    iu := j;
            fim-se
    fim-para
         u := iu;
    fim-enquanto
    para i de 1 até n faça
           imprima(v[i]);
    fim-para
fim
```

Figura 6.9 – Algoritmo II
Fonte: Elaborado pelo autor (2015)

Vários são os aspectos notáveis a observar, antes de analisarmos o algoritmo com maior profundidade:

- A variável que endereça o vetor durante o processo de comparação, “j” compara uma posição com sua sucessora, fazendo as trocas necessárias, mas nunca trocas inúteis! É fácil notar que se o maior número estiver na primeira posição, terminado o primeiro ciclo de trocas, este estará na última posição.
- Notemos ainda que todos os valores, a cada iteração completa, tendem a ir se aproximando das posições que terminarão sendo as definitivas.

- A variável “u” indica a última posição a ser trocada. Sintomaticamente, é iniciada com “n”, no caso o último termo e, imediatamente no início do laço de repetição é “zerada”, ou seja, parte do pressuposto otimista que a série está previamente ordenada. De fato, se estiver terminado o primeiro ciclo de comparações, o algoritmo será encerrado, o que lhe confere um desempenho muito melhor do que as duas implementações vistas anteriormente.
- A variável “iu”, variável auxiliar de “u” não é um *contador*, na medida em que não conta, apenas indica qual e se há posição a ser trocada. Defini-la como uma *flag* ou baliza, bandeira, também seria incorreto, pois muito mais que marcar se o programa “passou” por determinado lugar ou situação, ela indica que além de ter passado por determinado lugar, qual é o número da posição a ter seu valor trocado com o que está na primeira posição. Isto, só e se realmente a troca for necessária.

Vamos agora imaginar que nosso vetor tenha sete valores, a saber, 4; 3; 6; 2; 9; 1 e 7. Imediatamente após iniciar o algoritmo, o valor de “u” se torna 7, ou seja, o número de elementos da série, enquanto nossa sentinela “iu” assume o valor 0 (zero), na suposição que a série esteja ordenada.

| iu | v[1] | v[2] | v[3] | v[4] | v[5] | v[6] | v[7] | Troca |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 3 | 6 | 2 | 9 | 1 | 7 | 4 e 3 |
| 1 | 3 | 4 | 6 | 2 | 9 | 1 | 7 | não |
| 1 | 3 | 4 | 6 | 2 | 9 | 1 | 7 | troca |
| 3 | 3 | 4 | 2 | 6 | 9 | 1 | 7 | não |
| 5 | 3 | 4 | 2 | 6 | 9 | 1 | 7 | troca |
| 6 | 3 | 4 | 2 | 6 | 1 | 9 | 7 | troca |
| 6 | 3 | 4 | 2 | 6 | 1 | 7 | 9 | |

Figura 6.10 – Vetor com 7 valores
Fonte: FIAP (2015)

Já na primeira comparação, entretanto, temos uma troca, na medida em que 4 > 3. Essa troca leva “iu” a assumir o valor 1. Esse valor será mantido na próxima comparação, uma vez que 4 < 6. Na terceira comparação temos nova troca. Essa situação se mantém até a última passagem, algumas vezes com trocas, noutras não.

Finalmente, quando o último elemento do vetor é comparado, garantimos que o maior elemento, uma vez que nossa intenção é ordenação crescente de termos, ocupa agora a última posição, ou seja, os mais pesados vão ao fundo e os mais leves sobem. Com efeito, os números 3, 2 e 1 recuaram, enquanto 4, 6 e 7, avançaram. Finalmente, 9, além de ter avançado, foi para sua posição final, ou seja, a última posição.

Vamos agora ver o segundo ciclo de comparações, onde provavelmente teremos uma pequena surpresa...

| iu | v[1] | v[2] | v[3] | v[4] | v[5] | v[6] | v[7] |
|---|---|---|---|---|---|---|---|
| 1 | 3 | 2 | 4 | 1 | 6 | 7 | 9 |
| 1 | 2 | 3 | 4 | 1 | 6 | 7 | 9 |
| 3 | 2 | 3 | 4 | 1 | 6 | 7 | 9 |
| 0 | 2 | 3 | 1 | 4 | 6 | 7 | 9 |
| 0 | 2 | 3 | 1 | 4 | 6 | 7 | 9 |
| 0 | 2 | 3 | 1 | 4 | 6 | 7 | 9 |
| 0 | 2 | 3 | 1 | 4 | 6 | 7 | 9 |

Figura 6.11 – Segundo ciclo de comparações
Fonte: FIAP (2015)

Notamos que “iu” reinicia o processo valendo zero, ou seja, parte-se do princípio de que a série esteja ordenada. No entanto, já na segunda passagem temos uma troca. Situação essa que volta a se repetir, agora na quarta passagem. Todavia, não ocorrem mais trocas, o que significa que os últimos três números estão ordenados.

O algoritmo ter detectado essa situação implica em não perder tempo com comparações inúteis. Ou seja, diferentemente dos algoritmos anteriores, esse algoritmo não se preocupa apenas com colocar o número na posição certa, mas também com a situação de todos os demais números envolvidos no processo.

Dando prosseguimento ao processo, teremos:

| iu | v[1] | v[2] | v[3] | v[4] | v[5] | v[6] | v[7] |
|---|---|---|---|---|---|---|---|
| 0 | 2 | 3 | 1 | 4 | 6 | 7 | 9 |
| 2 | 2 | 3 | 1 | 4 | 6 | 7 | 9 |
| 2 | 2 | 1 | 3 | 4 | 6 | 7 | 9 |
| 0 | 2 | 1 | 3 | 4 | 6 | 7 | 9 |
| 0 | 2 | 1 | 3 | 4 | 6 | 7 | 9 |
| 0 | 2 | 1 | 3 | 4 | 6 | 7 | 9 |
| 0 | 2 | 1 | 3 | 4 | 6 | 7 | 9 |

Figura 6.12 – Terceiro ciclo de comparações
Fonte: FIAP (2015)

Dessa vez, “iu” começa com zero, mas já na primeira comparação, assume o valor um, pois 3 > 2. Nova troca só irá ocorrer na terceira comparação. É fácil observar que a série converge rapidamente para sua ordenação, algo que não ocorria nos dois algoritmos anteriores.

Podemos notar que apenas o número 1, que ocupa a terceira posição está fora de ordem. Dando prosseguimento ao processo, teremos:

| iu | v[1] | v[2] | v[3] | v[4] | v[5] | v[6] | v[7] | Troca |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 6 | 7 | 9 | 2 e 1 |
| 1 | 1 | 2 | 3 | 4 | 6 | 7 | 9 | não |
| 1 | 1 | 2 | 3 | 4 | 6 | 7 | 9 | não |
| 1 | 1 | 2 | 3 | 4 | 6 | 7 | 9 | não |
| 1 | 1 | 2 | 3 | 4 | 6 | 7 | 9 | não |
| 1 | 1 | 2 | 3 | 4 | 6 | 7 | 9 | não |
| 1 | 1 | 2 | 3 | 4 | 6 | 7 | 9 | não |

Figura 6.13 – Quarto ciclo de comparações
Fonte: FIAP (2015)

Finalmente, temos somente uma troca e no próximo processo, com apenas uma comparação, “iu” terminará valendo zero, abortando o laço. Isso significa que a série está ordenada.

### 6.5 Bubble sort variante shake

Existe uma forma de “simplificação” do Bubble Sort, que às vezes é chamada de Shake, o que leva a uma certa confusão com a estratégia de ordenação Shaker, da qual o Shake não tem qualquer relação. Tem uma tênue relação com o Bubble Sort, mas é geralmente usado para demonstrar, com muita clareza, que soluções simplórias podem causar enorme dor de cabeça...

Comecemos analisando o algoritmo em questão:

```
algoritmo shake_sort;
variáveis
      min, i, j, n, c, u : inteiro;
      v : matriz[100] de inteiros;
fim-variáveis
início
    imprima("Total de Números: ");
      n := leia();
      para i de 1 até n faça
         imprima("Digite o ",i,"o. Numero: ");
         v[i] := leia();
    fim-para
      u := 1;
      enquanto u > 0 faça
        u := 0;
         para j de 1 até u-1 faça
            se v[j] > v[j+1] então
                   c := v[j];
                   v[j] := v[j+1];
                   v[j+1] := c;
                   u := 1;
            fim-se
        fim-para
    fim-enquanto
    para i de 1 até n faça
           imprima(v[i]);
    fim-para
fim
```

Figura 6.14 – Algoritmo III
Fonte: Elaborado pelo autor (2015)

A grande distinção é a ausência de “iu” e “u” funcionando como uma *flag.* Se “u” for zero, é processo ordenado, mas se “u” for 1, então é necessário mais um processo.

Ocorre, entretanto, que se o processo tiver o primeiro elemento como o maior da série, este irá rapidamente para o final da fila. Ou seja, o desempenho do método será tão veloz como o Bubble Sort e substancialmente mais veloz que os métodos da seleção vistos anteriormente.

Por outro lado, se a série estiver invertida, o esforço será idêntico ao do Bubble. Todavia, se observamos a estratégia, se tivermos todos os termos em ordem, exceto o último que em nosso caso será o menor de todos, o método além de ser muito pior que o Bubble, também terá desempenho muito inferior ao método da Seleção Normal e mesmo do esgotamento. Isso ocorre porque a cada iteração o método examina do primeiro ao último termo, uma vez que mesmo arrumando as posições finais do vetor, o método não se “interessa” em saber **quais** elementos já estão arrumados. A única pretensão do método é saber se os elementos estão ordenados.

### 6.6 Exercícios

1) A partir da série com 6 termos, a saber, 14; 15; 12; 16; 13; 11, simule para Esgotamento, Seleção, Bubble e Shake. Analise o esforço de cada um dos métodos.

2) A partir da série com 6 termos, a saber, 11; 12; 13; 14; 15; 16, simule para Esgotamento, Seleção, Bubble e Shake. Analise as distinções e “desempenho” de cada um deles.

3) A partir da série com 6 termos, a saber, 17; 12; 13; 14; 15; 16, simule para Esgotamento, Seleção, Bubble e Shake. Analise as distinções e “desempenho” de cada um deles.

4) A partir da série com 6 termos, a saber, 17; 16; 15; 14; 13; 12, simule para Esgotamento, Seleção, Bubble e Shake. Analise as distinções e “desempenho” de cada um deles.

5) A partir da série com 6 termos, a saber, 11; 12; 13; 14; 15; 10, simule para Esgotamento, Seleção, Bubble e Shake. Analise as distinções e “desempenho” de cada um deles.

## 6.7 Ordenação de vetores (métodos elementares de alto desempenho).

Entendido o princípio da ordenação, vamos agora à discussão de métodos mais performáticos.

### 6.7.1 Método da inserção ou do baralho

Nos algoritmos de ordenação vistos anteriormente, notamos que a ideia de fixarmos uma posição é muito inferior à de trabalharmos visando ordenar o todo de uma única vez. Todavia, os algoritmos que estudamos demoram bastante para atingir nosso objetivo, isto é, ordenar uma série de números, por exemplo.

Mais que isso, os métodos descritos, embora engenhosos são antinaturais. Pensemos numa situação em que de posse de um baralho, descartados todos os naipes, menos ouro, pedimos para uma criança alfabetizada ordená-lo em ordem crescente, antes definindo que o baralho se inicia no Ás, segue pelos números e termina com a sequência Valete, Dama e Rei.

Vamos tentar descrever o que acontece, mas nem palavras ou imagens têm a força da experiência. Fica o convite para que você tente fazer isso com seu filho, sobrinha, neta etc.

Comece dando a primeira carta à criança. Imaginemos ser o 5 de ouros. A criança vai segurá-la. Em seguida, vamos imaginar dando-lhe a Dama. Quase intuitivamente, a criança irá **inserir** a Dama depois do 5. Agora, que ela receba o dez. É bastante provável que **insira** o dez entre o 5 e a Dama, **deslocando** a dama uma posição. Finalmente, para fins de exemplificação, pensemos que a próxima carta seja o 2. Essa carta passa a ser a primeira, sendo **inserida** na primeira posição e deslocando todas as demais uma posição adiante.

Esse é o espírito que norteia o método da inserção, que por razões mais do que óbvias, também é conhecido como método do baralho.

Trata-se de um método bastante eficiente quando aplicado a um pequeno número de elementos. Em termos gerais, ele percorre um vetor de elementos da esquerda para a direita e à medida que avança vai deixando os elementos mais à esquerda ordenados.

Vamos ao algoritmo em questão:

```
algoritmo insert_sort;
variáveis
      min, i, j, n : inteiro;
      v : matriz[100] de inteiros;
fim-variáveis
início
    imprima("Total de Números: ");
    n := leia();
    para i de 1 até n faça
       imprima("Digite o ",i,"o. Numero: ");
       v[i] := leia();
      fim-para
    para i de 2 até n faça
         v[0] := v[i];
         j := i - 1;
    fim-para
    enquanto v[0] < v[j] faça
           v[j+1] := v[j];
           j := j - 1;
    fim-enquanto
    v[j+1] := v[0];
    para i de 1 até n faça
           imprima(v[i]);
    fim-para
fim
```

Figura 6.15 – Algoritmo IV
Fonte: Elaborado pelo autor (2015)

Nesse algoritmo, a estratégia é concentrada num pequeno bloco de instruções, que passamos a comentar:

Primeiramente,

```
algoritmo insert_sort;
variáveis
      min, i, j, n : inteiro;
      v : matriz[100] de inteiros;
fim-variáveis
início
    imprima("Total de Números: ");
    n := leia();
    para i de 1 até n faça
       imprima("Digite o ",i,"o. Numero: ");
       v[i] := leia();
      fim-para
    para i de 2 até n faça
         v[0] := v[i];
         j := i - 1;
    fim-para
    enquanto v[0] < v[j] faça
           v[j+1] := v[j];
           j := j - 1;
    fim-enquanto
    v[j+1] := v[0];
    para i de 1 até n faça
           imprima(v[i]);
    fim-para
fim
```

Figura 6.16 – Algoritmo V
Fonte: Elaborado pelo autor (2015)

Na estrutura “para”, estrutura de repetição mais externa grafada em azul, copiamos o elemento a ser inserido, inicialmente na que virá a ser a última posição da série, além de também copiá-lo para a posição zero, ou seja, uma posição “de folga”.

Por analogia, vamos imaginar que estamos ordenando as cartas de um baralho com apenas um naipe. Parece natural que se tivermos já quatro cartas, a quinta carta recebida seja inicialmente comparada com a última, no caso a maior, e em sendo menor, ser progressivamente comparada às demais cartas.

Na pior das hipóteses, essa carta será a menor de todas, ficando na primeira posição.

Por causa disso, em nosso algoritmo, providenciamos uma “cópia” dessa carta na posição zero, que nesse caso não é a primeira posição, mas apenas uma posição sentinela. Com efeito, se a carta que estiver sendo inserida fora a menor de todas, naturalmente chegará à posição zero, pois nosso algoritmo sempre busca a posição anterior.

Procedendo assim, temos sempre o risco de tentar buscar uma posição “anterior” à primeira posição, o que é um absurdo em termos conceituais, mas que se não tratarmos a situação algoritmicamente, apenas se transformará num erro de lógica, que se traduzido para um programa de computador, irá gerar um erro de execução.

Assim, dentro do laço “para” a instrução `v[0] <- v[i]` significa tão somente que temos o valor a ser inserido, inicialmente na última posição e também uma posição adiante da primeira. Paradoxalmente, na linguagem humana, a posição 1 é a primeira e a posição 0 inexistente, todavia como facilmente podemos perceber essa é uma mera padronização humana, pois podemos iniciar um vetor com seu primeiro elemento na posição zero. Aliás, em muitas linguagens de programação, como C e Java, é exatamente isso que ocorre.

Em seguida, “j” (de `j <- i – 1`) é iniciado com o valor de “i” subtraído 1.

Entramos na estrutura “enquanto” (representada na cor vermelha) que se repete até que o número a ser inserido se torne maior ou igual a algum número já existente no vetor.

Observe que enquanto o número for menor, os números armazenados no vetor são deslocados uma posição adiante, exatamente o que ocorreu no exemplo com nossa hipotética criança.

Essa é a exata função das instruções dentro da estrutura de repetição em vermelho.

Observe que na saída da estrutura de repetição “enquanto” “j” tem a última posição “deslocada”, por isso a atribuição ocorre uma única vez, com a atribuição do valor introduzido na posição correta através da instrução pertencente à estrutura “para” (`v[j+1] <- v[0]`). Muito cuidado em não se confundir e pensar que essa instrução está contida na estrutura “enquanto”. Ela na verdade faz parte da estrutura “para”.

Vamos descrever a mecânica de funcionamento através de uma situação prática que você pode (e deve) simular.

Assim, se inicialmente tivemos armazenado no vetor três números e estes forem:

V[1] = 5; V[2] = 10 e V[3] = 14 e desejarmos inserir nessa série o número 8, teríamos...

V[0] = 8 e V[4] = 8; menor que V[3], portanto V[4] = 14 (note que o número a ser inserido é descartado da última posição, entretanto está armazenado na posição zero). V[0] < V[2], então V[3] recebe V[2] (ou seja, V[3] agora vale 10).

Finalmente V[0] >= V[1], então na posição V[2] (trecho de atribuição do algoritmo, logo após concluído o laço enquanto), portanto V[2] recebe V[0], realizando a inserção desejada.

### 6.8 Aplicando análise algorítmica no método da inserção

Uma das maneiras mais fáceis de mostrar que um algoritmo iterativo faz o que promete fazer, é exibir um invariante (algo que não se altera ao aplicar-se um conjunto de transformações) do processo iterativo, como vimos no capítulo anterior.

No nosso caso, percebemos que no início de cada repetição “enquanto”, imediatamente antes de verificar a condição v[0] < v[j], o vetor v[1..$j-1$] é crescente.

Isso ocorre porque a cada passo anteriormente executado, todos os elementos são previamente ordenados, tal qual faríamos se estivéssemos ordenando cartas de um baralho (por isso, esse método também é conhecido como método do baralho).

Esse invariante é trivialmente verdadeiro na primeira repetição do “enquanto” para (pois $j = 1$ nesse caso). Se o invariante for verdadeiro na última repetição do para (quando $j = n+1$), então nosso problema está resolvido.

Quando fazemos uma análise de algoritmos, a questão mais relevante é: ***Quanto tempo o algoritmo consome?***

Trata-se de uma questão um tanto complexa, pois o tempo depende de duas coisas naturalmente fora do controle de quem desenvolve um programa de computador, que são:

- Da instância do problema, ou seja, do particular vetor $v[1..n]$ que está sendo ordenado.
- Do computador que estiver sendo usado, incluído aí seu sistema operacional e suas características técnicas.

Para responder à primeira questão, basta pensarmos no pior caso possível que possa ser submetido aos algoritmos em particular.

Assim, para cada valor de $n$, considero a instância $v[1..n]$ para a qual o algoritmo consome mais tempo.

Chamando esse tempo a ser determinado de $T(n)$, ou seja, o tempo demandado para ordenar um número $n$ de termos.

Já para responder à segunda questão, basta observar que o tempo não é diretamente dependente do computador usado. Ao mudarmos de um computador para outro, o consumo de tempo do algoritmo é apenas multiplicado por uma constante, que está relacionada à máquina que está sendo usada, incluídas aí todas as suas características de memória, de processador e do sistema operacional.

Exemplificando: se $T(n) = 100n^2$ em um computador A, então $T(n) = 200n^2$ em um computador duas vezes mais lento e $T(n) = 10n^2$ em um computador dez vezes mais rápido.

Assim, podemos chamar de $T(n)$, o tempo de execução, para o algoritmo de inserção.

### 6.8.1 Buscando o pior caso

A coluna direita do esquema indica o número de execuções de cada uma das linhas no pior caso.

| Instrução | Execuções |
|---|---|
| para i de 2 ate n faca | n-1 |
| v[0] <- v[i] | n-1 |
| j <- i - 1 | n-1 |
| enquanto v[0] < v[j] faca | 2+3+4+...+n |
| v[j+1] <- v[j] | 1+2+3+...+n-1 |
| j <- j - 1 | 1+2+3+...+n-1 |
| fimenquanto | 0 |
| v[j+1] <- v[0] | n-1 |
| Fimpara | 0 |

Figura 6.17 - O pior caso
Fonte: FIAP (2015)

Vamos supor que qualquer que seja a instrução executada, seja consumida uma unidade de tempo, de uma escala qualquer que seja definida. Ora, nessas condições, poderíamos calcular nosso T(n)!

$T(n) = (3/2)n^2 + (7/2)n - 5$

Se conhecermos as velocidades reais de cada instrução, certamente modificariam os valores-base de nosso polinômio de segundo grau, contudo continuaríamos a ter um polinômio de segundo grau.

Justamente por isso, todos os algoritmos que se utilizam da estratégia de comparações sucessivas são chamados de algoritmos de complexidade quadrática.

De qualquer forma, o coeficiente 3/2 de $n^2$ não tem maior relevância: ele não depende do algoritmo, mas de nossa hipótese que assume uma unidade de tempo por linha. Já o $n^2$ é fundamental: ele é característico do algoritmo em si e não depende nem do computador nem dos detalhes da implementação do algoritmo.

Em resumo, a única parte importante no cálculo de $T(n)$ é **justamente** $n^2$.

Todo o resto depende da maneira como será implementado o algoritmo e do computador que estiver sendo usado para a ordenação em si.

Então, podemos dizer que a quantidade de tempo que o algoritmo da Ordenação por Inserção consome no pior caso é $\Theta (n^2)$.

### 6.9 Shell sort

Criado por Donald Shell em 1959, publicado pela Universidade de Cincinnati, Shell Sort é o mais eficiente algoritmo de classificação dentre os de complexidade quadrática. É um refinamento do método de inserção direta. O algoritmo difere do método de inserção direta pelo fato de no lugar de considerar o vetor a ser ordenado como um único segmento, ele o considera como se fossem vários segmentos.

A partir de cada segmento, busca-se ordenar de forma que se o objetivo for chegar a uma série ordenada de forma crescente, os elementos menores rapidamente deverão convergir para o início da série e os maiores para seu final.

Depois de feito esse processo de pré-ordenação, acaba sendo aplicado o método de inserção direta no vetor inteiro.

Busquemos, entretanto, uma analogia.

Imaginemos um baralho, desta vez com 52 cartas. Nosso baralho terá quatro naipes, a saber: ouro, copas, paus e espadas, sendo os dois primeiros vermelhos e os dois últimos pretos.

Ora, parece ser uma boa estratégia, primeiramente, separar as cartas vermelhas das pretas. Em seguida, separamos os naipes. Finalmente, ordenamos as cartas dentro de seus respectivos naipes.

Se pensarmos que as cartas vermelhas são menores que as pretas e que ouros é menor que copas e ainda que espadas é menor que paus, teríamos um baralho ordenado, certo?

Ora, se pensarmos termos cartas de 1 até 52 e compararmos a primeira com a décima quarta carta, e assim sucessivamente, não teríamos algo como: 14;1 | 15;2 | 16;3, e assim sucessivamente? Há um interessante intervalo de exatas 13 posições, ou seja, o exato número de cartas que cada naipe ocuparia se estivéssemos trabalhando com um baralho.

Todavia, se observarmos bem, notaremos que quando estivermos comparando posições altas teremos: 40;27;14;1 | 41;28.15;2, etc.

Notemos que se o Ás de ouros estiver na posição 40, já na primeira passagem irá para seu lugar!

Claro, dependeríamos de muita sorte para isso acontecer, mas é fácil notar que se estivesse na posição 41 não iria para seu lugar, mas estaria bem próximo dele. Aliás, é fácil intuir que o Rei de Espadas e o Ás de ouro, na primeira iteração podem até não irem para seus devidos lugares, todavia estarão muito, mas muito próximos deles.

Extrapolando para as demais cartas, percebemos que todas caminharão rapidamente para seus devidos lugares, se seguirmos essa estratégia. Então, vamos a ela, na sua forma algorítmica:

```
algoritmo shell_sort;
variáveis
      h, n, i, j, c, base : inteiro;
      v : matriz[100] de inteiros;
    continua: lógico;
fim-variáveis
início
  imprima("Total de Números: ");
  n := leia();
  para i de 1 até n faça
     imprima("Digite o ",i,"o. Numero: ");
     v[i] := leia();
  fim-para
  h := 1;
  base := 3;
  repita
     h := base * h + 1;
  até h > n;
  repita
     h := h / base;
     para i de h+1 até n faça
        c := v[i];
        j := i;
        continua := verdadeiro;
        enquanto v[j-h] > c e continua = verdadeiro faça
           v[j] := v[j-h];
           j := j - h;
           se j <= h então
              continua := falso;
            fim-se
        fim-enquanto
        v[j] := c;
     fim-para
  até h = 1;
  para i de 1 até n faça
       imprima(v[i]);
  fim-para
fim
```

Figura 6.18 – Algoritmo VI
Fonte: Elaborado pelo autor (2015)

Analisando-se mais detidamente o algoritmo em questão, notamos que existem dois laços independentes, sendo o primeiro um laço “repita” simples e o segundo um ninho de laços em execução, cada um com determinada função, a saber:

**Repita:** laço mais externo e diretamente responsável pela pré-ordenação da série. Notar que a função desse laço é justamente determinar o valor de “h”, a variável base do esquema de pré-ordenação.

Primeiramente, vamos entender seu exato e importante funcionamento. Note que no algoritmo apresentado existe um número 3 sendo a “base” de cálculo de “h”. Na verdade, qualquer número pode ser usado como base, mas através de muitas experiências, pode-se notar que o algoritmo tem seu máximo desempenho na maioria dos casos, quando o número base é 3 ou 4.

Por não existir uma demonstração matemática, a escolha desse fator é prerrogativa exclusiva do programador. Mas, analisando-se alguns casos, ficará mais fácil compreender a importância dessa escolha.

Vamos voltar a nosso baralho com 52 cartas, portanto teríamos que ordenar 52 números. Vamos testar alguns fatores e analisar os resultados do código.

```
h := 1
repita
   h := base * h + 1, onde base é 3
ate h > n
```

Figura 6.19 – Algoritmo VII
Elaborado pelo autor (2015)

**Fator 3:** com 3 teríamos a seguinte sequência de valores para “h”: h = 1; h = 4; h = 13 ; h = 52 ; h = 157. Uma vez que o algoritmo é iniciado, começa fazendo um teste no intervalo 52, pois sempre divide o fator “h” pelo número base, no caso 3. Em seguida passaria a trabalhar com o intervalo 17 (52 / 3, em sua parte inteira). Assim, o passo inicial de ordenação seria 17. Por exemplo, se estivermos analisando o elemento na posição 40 este será comparado com a posição 23 e, em seguida, com a posição 3. É fácil notar que o longo intervalo trará os elementos “leves” para o início da série e os “pesados” para seu final com relativa rapidez, usando o fator 3.

```
h := 1
  repita
     h := base * h + 1, onde base é 4
  ate h > n
```

Figura 6.20 – Algoritmo VIII
Fonte: Elaborado pelo autor (2015)

**Fator 4:** com 4 teríamos a seguinte sequência de valores para "h": h = 1; h = 5; h = 21 ; h = 65. Uma vez que o algoritmo é iniciado, começa fazendo um teste no intervalo 16 (note que o número 3 que faz parte do segundo laço e que divide "h" também deve ser mudado para 4, nosso número base). Em seguida, o intervalo passa a ser 16 (65 / 4, em sua parte inteira). Assim, o passo inicial de ordenação seria 16. Por exemplo, se estivermos analisando o elemento na posição 40, este será comparado com a posição 24 e, em seguida, com a posição 8.

```
h := 1
  repita
     h := base * h + 1, onde base é 7
  ate h > n
```

Figura 6.21 – Algoritmo IX
Fonte: Elaborado pelo autor (2015)

**Fator 7:** com 7 teríamos a seguinte sequência de valores para "h": h = 1; h = 8; h = 57. Uma vez que o algoritmo é iniciado, começa fazendo um teste no intervalo 8 (analogamente, o original 3 que havia passado a 4 no exemplo anterior agora passa a ser 7). Por exemplo, se estivermos analisando o elemento na posição 40, este será comparado com a posição 32 e, em seguida, com as posições 24, 16 e 8.

Intuitivamente percebemos que os fatores 3 e 4 começam com trechos mais longos, portanto mais aptos a deslocar os números para suas posições adequadas. O 7 também o faz, porém de forma mais lenta. Pensemos nos fatores que o 3, 4 e 7 implicam:

3: 52; 17; 5 e 1

4: 16; 5 e 1

7: 8 e 1

Notamos que o 7 fez uma pré-ordenação média (passo 8) e já decaiu para a ordenação por inserção, ao passo que os números 3 e 4 fizeram uma pré-ordenação longa (17 ou 16, dependendo do fator e 5 para ambos).

Mais adiante, simularemos um caso completo, para maior compreensão.

**Laço Repita/Para/Enquanto.**

```
repita
     h := h / base
     para i de h+1 até n faça
        c := v[i];
        j := i;
        enquanto v[j-h] > c faça
           v[j] := v[j-h];
           j := j - h;
           se j <= h então
              interrompa;
           fim-se
        fim-enquanto
        v[j] := c;
     fim-para
  até h = 1;
```

Figura 6.22 – Algoritmo X
Fonte: Elaborado pelo autor (2015)

Vamos analisar, detidamente a função de cada um dos laços indicados:

- Repita: laço destinado ao controle da variável principal do algoritmo, “h”. Notar que o fator base é quem determina a quantidade de vezes que o laço repita será executado, além de determinar a “distância” que os elementos irão “saltar” nos laços mais internos.
- Para: esse laço percorre trechos determinados por “h”. Numa hipotética série com 52 elementos com base 4, o primeiro termo a ser comparado seria 17 (16 + 1) com seus correlatos anteriores. Nesse caso, o décimo sétimo termo seria comparado com o primeiro, na sequência o décimo oitavo com o segundo e assim sucessivamente.
- Enquanto**:** justamente nessa estrutura o os elementos detectados como “fora da ordem” são trocados, sendo que existe sempre uma atribuição incondicional no final do laço enquanto faz a troca em si ou apenas mantém o número na posição em que se encontrava originalmente.

### 6.9.1 Simulação do shell sort

A simulação do Shell Sort será feita no algoritmo com valor base 3, contendo 10 elementos, a saber: 13; 16; 19; 17; 14; 11; 15; 18; 10; 12.

Notar que não haveria problema algum se nem todos os números fossem distintos entre si, tampouco a ordem em que estes se encontram.

| N | I | J | C | H | V[j-h] |
|---|---|---|---|---|---|
| 10 | | | | 1 | 1 |
| | | | | 4 | 4 |
| | | | | 13 | 13 |
| | | | | 4 | 4 |
| | 5 | 5 | 14 | | 13 |
| | 6 | 6 | 16 | | 11 |
| | | 2 | | | |
| | 7 | 7 | 15 | | 19 |
| | | 3 | | | |
| | 8 | 8 | 18 | | 17 |
| | 9 | 9 | 10 | | 14 |
| | | 5 | | | 13 |
| | | 1 | | | |
| | 10 | 10 | 12 | | 16 |
| | | 6 | | | 11 |
| | | | | 1 | |

Figura 6.23 – Simulação do Shell Sort
Fonte: FIAP (2015)

Observemos agora a lista “pré-ordenada”: 10; 11; 15; 17; 13; 12; 19; 18; 14; 16.

| V | Inicial | | | | | | Pré-Ordem | | | Final |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 13 | | | | 10 | | 10 | | ... | 10 |
| 2 | 16 | 11 | | | | | 11 | | ... | 11 |
| 3 | 19 | | 15 | | | | 15 | | ... | 12 |
| 4 | 17 | | | | | | 17 | | ... | 13 |
| 5 | 14 | | | 10 | 13 | | 13 | | ... | 14 |
| 6 | 11 | 16 | | | | 12 | 12 | | ... | 15 |
| 7 | 15 | | 19 | | | | 19 | | ... | 16 |
| 8 | 18 | | | | | | 18 | | ... | 17 |
| 9 | 10 | | | 14 | | | 14 | | ... | 18 |
| 10 | 12 | | | | | 16 | 16 | | ... | 19 |

Figura 6.24 – Simulação do Shell Sort (2)
Fonte: FIAP (2015)

De fato, parece intuitivo que a série estará mais próxima de sua ordenação definitiva, mas como aferir se o esforço de uma pré-ordenação levou a uma melhoria?

Uma forma bastante simples e prática é a análise da distância do elemento a sua posição definitiva.

Vamos analisar a série original e em seguida a série pré-ordenada: 13 (3); 16(5); 19(7); 17(4); 14(0); 11(4); 15(1); 18(1);10(8);12(7). Somando-se as distâncias temos 40.

Analogamente, com a série pré-ordenada temos: 10(0); 11(0); 15(3); 17(4); 13(1); 12(3); 19(3); 18(1); 14(4); 16(3). Somando-se as distâncias temos 22!

É facilmente perceptível que a série ficou muito mais próxima de sua posição final, quando aplicada a pré-ordenação, todavia a aferição da distância média dos elementos em relação as suas posições definitivas, certamente impressiona.

Devemos observar, contudo, que no geral o Shell Sort leva a um sensível aumento no desempenho do algoritmo, mas é errado supor que isso sempre ocorre e, mais errado ainda, supor que sempre o desempenho será tão impressionante como no exemplo apresentado. Uma vez que dependemos de uma boa escolha de um número base, já podemos intuir que essa vantagem pode se diluir em certas situações.

## 6.10 Exercícios

a) Anote falso ou verdadeiro para cada afirmação. Quando a afirmação for falsa, justificar indicando o erro contido na afirmação:

Considere o algoritmo abaixo:

```
algoritmo cap6_exercicioa;
variáveis
      h, n, i, j, c, base : inteiro;
      v : matriz[100] de inteiros;
    continua: lógico;
fim-variáveis
início
  imprima("Total de Números: ");
  n := leia();
  para i de 1 até n faça
     imprima("Digite o ",i,"o. Numero: ");
     v[i] := leia();
  fim-para
  h := 1;
  base := 4;
  repita
     h := base * h + 1;
  até h > n;
  repita
     h := h / base;
     para i de h+1 até n faça
        c := v[i];
        j := i;
        continua := verdadeiro;
        enquanto v[j-h] > c e continua = verdadeiro faça
           v[j] := v[j-h];
           j := j - h;
           se j <= h então
              continua := falso;
            fim-se
        fim-enquanto
        v[j] := c;
     fim-para
  até h = 1;
  para i de 1 até n faça
       imprima(v[i]);
  fim-para
fim
```

Figura 6.25 – Algoritmo XI
Fonte: Elaborado pelo autor (2015)

- A variável “h” funciona como controle de pré-ordenação, isto é, ordena alguns elementos levando-os a região mais próxima do seu lugar definitivo. ( )
- Esse método é conhecido como Shell Sort e nunca foi convenientemente demonstrado, embora se saiba que tem desempenho muito superior a métodos de igual complexidade. ( )
- Esse método é um forte aperfeiçoamento do Bubble Sort, que é executado, quando o valor de “h” se torna 1. ( )
- Com “3” ou “9”, como valor de “base”, o algoritmo tem máximo desempenho. ( )
- Com “3” ou “9”, como valor de “base”, o algoritmo deixa de funcionar.( )
- Com qualquer número no lugar do “4”, na expressão que inicializa a variável base, o algoritmo funciona, mas nem sempre com o desempenho adequado. ( )
- Descobriu-se recentemente que os valores adequados para a expressão do cálculo de “h”, valor inicial da variável “base” deve ser um número primo. ( )
- Trata-se do método mais performático já criado. ( )
- Entre os métodos simples de ordenação é o que possui melhor desempenho. ( )
- Trata-se de um método errático, por vezes funciona com desempenho conhecido e noutras não. ( )
- É chamado de Shell Sort, pois da mesma forma que no Bubble Sort em que os elementos mais leves sobem e os mais pesados descem, o Shell Sort assemelha-se à ordenação que as conchas do mar possuem, lembrando os números perfeitos de Fibonacci. ( )

b) Considere a série de números apresentada a seguir:

Números: 15; 17; 11; 13; 12; 19; 18; 16; 20; 14.

Elaborar simulações para os métodos do Esgotamento, da Seleção, Bubble Sort, Método da Inserção e Shell Sort. Avalie qual deles é mais performático com essa série de números e qual o menos performático. Os resultados foram os esperados? Por quê?

c) O algoritmo do exercício **A** realiza a ordenação de uma série de números, entretanto a estratégia utilizada implica em um risco do processo de ordenação ser muito mais lento do que deveria ser.

Aparentemente, o programador que escreveu esse algoritmo acredita que o número atribuído a “h” pode ser arbitrário e escolhido pelo usuário.

Identifique o motivo do risco, propondo uma alternativa para mitigá-lo. Simular.

```
algoritmo cap6_exercicioc;
variáveis
      h, n, i, j, c, base : inteiro;
      v : matriz[100] de inteiros;
    correto : lógico;
fim-variáveis
início
    imprima("Total de Números: ");
    n := leia();
    para i de 1 até n faça
         imprima("Digite o ",i,"o. Numero: ");
         v[i] := leia();
    fim-para
     h := 1;
     base := 4;
     repita
         h := base * h + 1;
     até h > n;
     repita
         h := h / base;
         para i de h+1 até n faça
                  c := v[i];
                  j := i;
                 correto := verdadeiro;
                 enquanto v[j-h] > c e correto  = verdadeiro faça
                    v[j] := v[j-h];
                    j := j - h;
                    se j <= h então
                        correto := falso;
                    fim-se
                fim-enquanto
                v[j] := c;
       fim-para
      até h = 1;
      para i de 1 até n faça
            imprima(v[i]);
      fim-para
fim
```

Figura 6.26 – Algoritmo XII
Fonte: Elaborado pelo autor (2015)

# REFERÊNCIAS

ENCYCLOPEDIA and history of programming languages. [s.d.]. Disponível em: <http://www.scriptol.org/>. Acesso em: 14 jan. 2011.

FEOFILOFF, Paulo. **Algoritmos em Linguagem C.** Rio de Janeiro: Campus, 2009.

FORBELLONE, André L.V.; EBERSPACHER, Henri F. **Construção de Algoritmos e Estruturas de Dados**. São Paulo: Pearson Prentice Hall, 2010.

FURGERI, Sérgio. **Java 2, Ensino Didático**. São Paulo: Érica, 2002.

GANE, Chris; SARSON, Trish. **Análise Estruturada de Sistemas**. São Paulo: LTC-Livros Técnicos e Científicos, 1983.

GONDO, Eduardo. **Apostila: Notas de Aula**. São Paulo, 2008.

MANZANO, José A.N.G.; OLIVEIRA, Jayr F. **Algoritmos:** Lógica para o Desenvolvimento de Programação. 23.ed. São Paulo: Érica, 2010.

LATORE, Robert. **Aprenda em 24 horas Estrutura de Dados e Algoritmos**. Rio de Janeiro: Campus, 1999.

PUGA, Sandra; RISSETTI, Gerson. **Lógica de Programação e Estrutura de Dados**. São Paulo: Pearson Prentice Hall, 2009.

PIVA JUNIOR, Dilermando et al. **Algoritmos e Programação de Computadores.** Rio de Janeiro: Campus, 2012.

ROCHA, Antonio Adrego. **Estrutura de Dados e Algoritmos em Java**. Lisboa: FCA, 2011.

RODRIGUES, Rita. **Apostila: Notas de Aula**. 2008.

SALVETTI, Dirceu Douglas; BARBOSA, Lisbete Madsen. **Algoritmos**. São Paulo: Makron Books, 1998.

SCHILDT, Herbert. **Linguagem C - Guia Prático**. São Paulo: McGraw Hill, 1989.

WOOD, Steve. **Turbo Pascal – Guia do Usuário**. São Paulo: McGraw Hill, 1987.

ZIVIANI, Nivio. **Projeto de Algoritmos com implementações em Pascal e C**. São Paulo: Pioneira, 1999.