UNIVERSIDADE DO ESTADO DE MINAS GERAIS | UEMG	**Universidade do Estado de Minas Gerais** **Campus de Ituiutaba** **Engenharia da Computação**

APOSTILA DE

ANÁLISE DE ALGORITMOS

Prof. Walteno Martins Parreira Júnior

www.waltenomartins.com.br

waltenomartins@yahoo.com

2014

Sumário

1 – Desenvolvimento de Algoritmos

1.1 – Introdução

- Dado um problema, como encontramos um algoritmo eficiente para sua solução?
- Encontrado um algoritmo, como comparar este algoritmo com outros algoritmos que solucionam o mesmo problema?
- Como deveríamos julgar a qualidade de um algoritmo?
- Qual é o algoritmo de menor custo possível para resolver um problema particular?

Questões desta natureza são de interesse para programadores e cientistas da computação. Algoritmos podem ser avaliados por uma variedade de critérios. Na maioria das vezes estamos interessados na taxa de crescimento do tempo ou de espaço necessário para a solução de grandes problemas.

1.2 – O que é um algoritmo

Um algoritmo pode ser visto como uma sequência de ações executáveis para a obtenção de uma solução para um determinado tipo de problema.

Segundo **Dijkstra**, um algoritmo corresponde a uma descrição de um padrão de comportamento, expresso em termos de um conjunto finito de ações.

Segundo **Terada**, um algoritmo é, em geral, uma descrição passo a passo de como um problema é solucionável. A descrição deve ser finita, e os passos devem ser bem definidos, sem ambiguidades, e executáveis computacionalmente.

1.3 – Medidas de Complexidade

Como selecionar um algoritmo quando existem vários que solucionam o problema? Uma resposta pode ser, escolher um algoritmo fácil entendimento, codificação e depuração ou então uma outra resposta pode ser, um algoritmo que faz uso eficiente dos recursos do computador. Qual a melhor solução? Como escolher?

Vários critérios podem ser utilizados para escolher o algoritmo, mas vai depender das pretensões de utilização do algoritmo. Pode-se estar selecionando o algoritmo somente para um experimento, ou será um programa de grande utilização, ou será utilizado poucas vezes e será descartado, ou ainda, terá aplicações futuras que podem demandar alterações no código. Para cada uma das respostas anteriores, pode-se pensar em uma solução diferente. Calcular o tempo de execução e o espaço exigido por um algoritmo para uma determinada entrada de dados é um estudo da complexidade de algoritmos.

Para o cálculo de complexidade, pode-se medir o número de passos de execução em um modelo matemático denominado maquina de Turing, ou medir o número de segundos gastos em um computador específico. Ao invés de calcular os tempos de execução em máquinas específicas, a maioria das análises conta apenas o número de operações “elementares” executadas. A medida de complexidade é o crescimento assintótico dessa contagem de operações.

Por exemplo, em um algoritmo para achar o elemento máximo entre “n” objetos, a operação elementar seria a comparação das grandezas dos objetos, e a complexidade seria

uma função em "n" do número de comparações efetuadas pelo algoritmo, para valores grandes de "n".

A complexidade assintótica de um algoritmo é que determina o tamanho de problemas que pode ser solucionado pelo algoritmo. Se o algoritmo processa entradas de tamanho "n" no tempo $\mathbf{c*n^2}$, para alguma constante **c**, então dizemos que a complexidade de tempo do algoritmo é $O(n^2)$, onde se lê: " de ordem n^2".

Suponha que temos cinco algoritmos A1,...A5 com as seguintes complexidades de tempo:

Algoritmo	Complexidade de tempo
A_1	n
A_2	$n \log_2 n$
A_3	n^2
A_4	n^3
A_5	2^n

A complexidade de tempo é o número de unidades de tempo (UT) necessárias para processar uma entrada de tamanho n. A unidade de tempo é medida em um milisegundo, portanto:

$$1UT = 1ms = 10^{-3} \text{ segundos}$$

A tabela a seguir mostra o tamanho de problemas que podem ser resolvidos em 1 segundo, em 1 minuto e em 1 hora para cada algoritmo em um computador hipotético:

Algoritmo	Complexidade de tempo	1 segundo	1 minuto	1 hora
A_1	n	1.000	60.000	3.600.000
A_2	$n \log_2 n$	140	4893	200.000
A_3	n^2	31,6	244,9	1.897,4
A_4	n^3	10	39,2	153,3
A_5	2^n	9	15	21

Cálculos, A_1:

$T(n) = n * UT$	$T(n) = n * UT$	$T(n) = n * UT$
$1 = n * 10^{-3}$	$60 = n * 10^{-3}$	$3600 = n * 10^{-3}$
$n = 1000$	$n = 60000$	$n = 36 * 10^5$

A_2:

$T(n) = n \log n * UT$	$T(n) = n \log n * UT$	$T(n) = n \log n * UT$
$1 = n \log n * 10^{-3}$	$60 = n \log n * 10^{-3}$	$3600 = n \log n * 10^{-3}$
$n \log n = 10^3$	$n \log n = 6 * 10^4$	$n \log n = 36 * 10^5$
$n = 140$	$n = 4893$	$n = 2 * 10^5$

A_3:

$T(n) = n^2 * UT$	$T(n) = n^2 * UT$	$T(n) = n^2 * UT$
$1 = n^2 * 10^{-3}$	$60 = n^2 * 10^{-3}$	$3600 = n^2 * 10^{-3}$
$n = 31,6$	$n = 244,9$	$n = 1897,4$

A_4:

$T(n) = n^3 * UT$	$T(n) = n^3 * UT$	$T(n) = n^3 * UT$
$1 = n^3 * 10^{-3}$	$60 = n^3 * 10^{-3}$	$3600 = n^3 * 10^{-3}$
$n = 10$	$n = 32,9$	$n = 153,3$

A tabela apresenta uma comparação relativa de grandeza para várias funções que podem ser encontradas em algoritmos. Podemos ter a noção da ordem de crescimento dos algoritmos.

	Tipo de Função					
n	$\text{Log}_2 n$	n	$N \log_2 n$	n^2	n^3	2^n
1	0	1	0	1	1	2
10	3,32	10	33	100	1000	1024
100	6,64	100	664	10.000	1.000.000	$1,268*10^{30}$
1000	9,97	1000	9970	1.000.000	10^9	$1,072*10^{301}$

Supondo que a próxima geração de computadores seja dez vezes mais rápido que a atual. A tabela abaixo mostra o aumento no tamanho do problema que o algoritmo possa resolver no mesmo tempo.

Algoritmo	Complexidade de tempo	Máximo atual	Máximo após aumento
A1	n	S1	10 * S1
A2	$n \log_2 n$	S2	≈ 10 * S2
A3	n^2	S3	≈ 3 * S3
A4	n^3	S4	2 * S4
A5	2^n	S5	S5 + 3

Cálculos:

A1	**A3**	**A5**
Atual: T(n) = S1 * 1 UT	Atual: T(n) = S3 * 1 UT	Atual: $T(n) = 2^{S5} * 1$ UT
Futuro: T´(n) = (n * 1UT) / 10	Futuro: T´(n) = (n^2 * 1UT) / 10	Futuro: T´(n) = (2^n * 1UT) / 10
T(n) = T´(n)	T(n) = T´(n)	T(n) = T´(n)
S1 * 1 UT = (n * 1UT) / 10	S3 * 1 UT = (n^2 * 1UT) / 10	$2^{S5} * 1$ UT = (2^n * 1UT) / 10
.n = 10 * S1	n^2 = 10 S3	$2^{S5} = 2^n / 10$
	.n ≈ 3 S3	$2^n = 10 * 2^n = 10 * 2^{S5}$
		$.n = \log_2 10 + S5$
		.n = 3 + S5

Comparação de várias funções de complexidade, segundo Ziviani:

	Valor de n		
Função de Complexidade	20	40	60
.n	0,0002 segundos	0,0004 segundos	0,0006 segundos
.n log n	0,0009 segundos	0,0021 segundos	0,0035 segundos
$.n^2$	0,004 segundos	0,016 segundos	0,036 segundos
$.n^3$	0,08 segundos	0,64 segundos	2,16 segundos
2^n	10 segundos	127 dias	3660 séculos
3^n	580 minutos	38550 séculos	$1,3*10^{14}$ séculos

Aumento do tamanho das instancias solucionáveis, com o aprimoramento dos computadores:

	Tamanho da maior instancia solucionável em 1 hora		
Função de Complexidade	Computador atual	Computador 100 vezes mais rápido	Computador 1000 vezes mais rápido
.n	N	100 * N	1000 * N
.n log n	N1	22,5 * N1	140,2 * N1
$.n^2$	N2	10 * N2	31,6 * N2
$.n^3$	N3	4,6 * N3	10 * N3
2^n	N4	N4 + 6	N4 + 10
3^n	N5	N5 + 4	N5 + 6

Para ilustrar melhor a diferença de comportamento entre complexidades, é apresentado o quadro desenvolvido por Garey e Johnson (1979) que mostra a razão de crescimento de várias funções de complexidade para tamanhos diferentes de n, em que cada função expressa o tempo de execução em microssegundos. Assim, um algoritmo linear executa em um segundo 1 milhão de operações.

Complexidade	**Tamanho n**				
	10	**20**	**30**	**40**	**50**
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0035 s
n^3	0,001 s	0,008 s	0,027 s	0,064 s	0,125 s
2^n	0,001 s	1 s	17,9 minutos	12,7 dias	35,7 anos
3^n	0,059 s	58 minutos	6,5 anos	3855 séculos	10^8 séculos

Outro aspecto interessante é o efeito causado pelo aumento da velocidade dos computadores sobre os algoritmos com as funções de complexidade apresentadas. A tabela baixo mostra como um aumento de 100 ou de 1000 vezes na velocidade do processador influi na solução do maior problema possível de ser resolvido em uma hora. Pode-se notar que um aumento de 1000 vezes na velocidade de computação permite resolver um problema 10 vezes maiores para um algoritmo de complexidade O(n3), enquanto um algoritmo de complexidade O(2n) apenas adiciona dez ao tamanho do maior problema possível de ser resolvido em uma hora.

Complexidade	**Computador atual**	**Computador 100 vezes mais rápido**	**Computador 100 vezes mais rápido**
n	t_1	100 t_1	1000 t_1
n^2	t_2	10 t_2	31,6 t_2
n^3	t_3	4,6 t_3	10 t_3
2^n	t_4	t_4 + 6,6	t_4 + 10

1.4 – Análise de Complexidade de um algoritmo

A finalidade de se fazer a análise de complexidade de um algoritmo é obter estimativas de tempos de execução de programas que implementam esse algoritmo. A complexidade do algoritmo dá ideia do esforço computacional do programa, que é uma medida do número de operações executadas pelo programa.

Uma das preocupações com a eficiência é com problemas que envolvem um considerável número de elementos. Se existir uma tabela com apenas dez elementos, mesmo o algoritmo considerado menos eficiente resolve o problema, no entanto, à medida que o número de elementos aumenta, o esforço necessário começa a fazer diferença de algoritmo para algoritmo.

O que se deseja na verdade é uma avaliação do desempenho do algoritmo independentemente da sua implementação, em função somente do número de instruções executadas para entradas determinadas. São consideradas somente a instruções preponderantes, isto é, as operações básicas para a execução do algoritmo. O número de vezes que essas operações são executadas é denominado Complexidade do Algoritmo.

Em geral, a complexidade de um algoritmo depende da entrada e esta é caracterizada pelo seu tamanho, por seus valores e também pela configuração dos dados.

De forma intuitiva, sabemos que a complexidade depende da quantidade de dados que são processados e isso se traduz pelo tamanho da entrada: o número de operações executadas para localizar o último registro de uma lista com 1000 registros deve ser maior que o de uma lista com apenas 10 registros. Muitas vezes, o valor da entrada determina o esforço, por exemplo, na execução de uma busca em uma lista linear não ordenada, o número de comparações executadas varia muito conforme o valor procurado ocorrer no primeiro registro, no final ou no meio da lista.

Por exemplo, nos algoritmos que executam operações sobre listas lineares, a complexidade é expressa em função do tamanho da lista. Se n indica o número de registros, temos que a complexidade será uma função de n. Por outro lado, como os valores ou a configuração dos dados de entrada são fatores que também interferem no processo, não é possível obter uma única função que descreva todos os casos possíveis. Para cada possibilidade de entrada há uma função de complexidade do algoritmo. Reduzimos o estudo para alguns casos especiais:

a) Pior Caso, caracterizado por entradas que resultam em maior crescimento do número de operações, conforme aumenta o valor de n;

b) Melhor Caso, caracterizado por entradas que resultam em menor crescimento do número de operações, conforme aumenta o valor de n;

c) Caso Médio, que retrata o comportamento médio do algoritmo, quando se consideram todas as entradas possíveis e as respectivas probabilidades de ocorrência (esperança matemática).

Somente o estudo da complexidade de algoritmos permite a comparação de dois algoritmos equivalentes, isto é, desenvolvidos para resolver o mesmo problema.

1.5 – Notação O

A notação O (leia-se ***ó grande***, ou ***big oh***) é utilizada para expressar comparativamente o crescimento assintótico representa a velocidade com que uma função tende ao infinito. No estudo de complexidade de algoritmos é mais interessante saber como se comporta essa função à medida que aumentarmos o valor de n, do que conhecer valores da função correspondentes a particulares valores de n.

Por exemplo, é mais importante saber que o número de operações executadas num algoritmo dobra se dobrarmos o valor de n, do que saber que para n igual a 100 são executadas 300 operações.

Ao dizermos que uma função de complexidade f(n) é da ordem de n^2, queremos dizer que as duas funções, f(n) e n^2 tendem ao infinito com a mesma velocidade, ou que têm o mesmo comportamento assintótico. Indicamos por

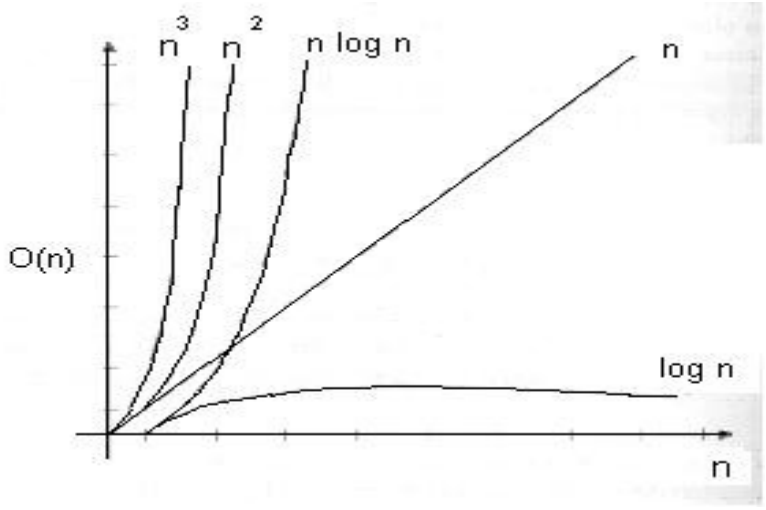
$$f(n) = O(n^2)$$

em matemática essa informação é expressa por um limite:

$$\lim_{n \to \infty} \frac{f(n)}{n^2} = c \ (c > 0)$$

Se, por exemplo, outro algoritmo para o mesmo problema tem função de complexidade $f_1(n) = O(n)$, podemos comparar f(n) e $f_1(n)$ e, em consequência, comparar a eficiência dos programas que os implementam. Em um deles, o tempo de execução é linear e no outro, o tempo é quadrático.

Função	Significado (tamanho da entrada = n)
1	tempo constante – o número de operações é o mesmo para qualquer tamanho da entrada
n	tempo linear – se n dobra, o número de operações também dobra
n^2	tempo quadrático – se n dobra, o número de operações quadruplica
log n	tempo logarítmico – se n dobra, o número de operações aumenta de uma constante
nlog n	tempo n log n - se n dobra, o número de operações ultrapassa o dobro do tempo da entrada de tamanho n
2^n	tempo exponencial - se n dobra, o número de operações é elevado ao quadrado



Os resultados expressos em notação O devem ser interpretados com cuidados, pois indicam apenas que o tempo de execução do programa é proporcional a um determinado valor ou que nunca supera determinado valor; na verdade o tempo de execução pode ser inferior ao valor indicado e pode ser que o pior caso nunca ocorra.

1.6 – Convenções para as expressões de O

Existem algumas convenções quanto à expressão de O:

- É prática comum escrever a expressão de O sem os termos menos significantes. Assim, em vez de $O(n^2 + n\log n + n)$, escrevemos simplesmente $O(n^2)$.
- É comum desconsiderar os coeficientes constantes. Em lugar de $O(3n^2)$, escrevemos simplesmente $O(n^2)$. Como caso especial desta regra, se a função é constante, por exemplo O(1024), escrevemos simplesmente O(1).

Algumas expressões de O são tão frequentes que receberam denominações próprias:

Expressão	Nome
O(1)	Constante
O(log n)	Logarítmica
$O(\log^2 n)$	Log quadrado
O(n)	Linear
O(nlog n)	n log n
$O(n^2)$	Quadrática
$O(n^3)$	Cúbica
$O(2^n)$	Exponencial

1.7 – Exemplo de análise da notação O

Para demonstrar os conceitos de eficiência, apresentaremos dois algoritmos que fazem a soma e a multiplicação de duas matrizes.

Uma possibilidade para a soma dos elementos das duas matrizes m_1 e m_2 é colocar o resultado em m_3, que pode ser feito da seguinte forma:

```
1    Linha ← 1
2    enquanto (linha ≤ tamanho_da_matriz ) faça
3        col ← 1
4        enquanto (col ≤ tamanho_da_matriz ) faça
5            m3 [ linha, col ] ← m1 [ linha, col ] + m2 [ linha, col ]
6            col ← col + 1
7        fim-enquanto
8        linha ← linha + 1
9    fim-enquanto
```

Analisando o trecho apresentado, vemos que o "enquanto" externo (linha 2) é dependente do tamanho da matriz. Para cada execução dessa estrutura de repetição, tem-se que executar o "enquanto" mais interno (linha 4), pode-se observar que ela depende do tamanho da matriz.

Pode-se observar na linha 2 a estrutura de repetição **enquanto** que percorre a matriz de 1 até N e para cada incremento da linha 2, a linha 4 percorre a matriz de 1 a N (portanto N execuções), onde N é o tamanho da matriz, logo N x N = N^2. Essa é uma situação clássica da repetição quadrática. A eficiência do algoritmo é $O(n^2)$.

O segundo exemplo, supondo a multiplicação de duas matrizes m_1 e m_2, cujo resultado será colocado na matriz m_3, podemos considerar o seguinte algoritmo:

```
1  coluna ← 1
2  enquanto (coluna ≤ tamanho_da_matriz ) faça
3      col ← 1
4      enquanto (col ≤ tamanho_da_matriz ) faça
5              m3 [ linha, col ] ← 0
6              k ← 1
7              enquanto (k ≤ tamanho_da_matriz ) faça
8                      m3 [ linha, col ] ← m3 [ linha, col ] + m1 [ linha, k ] * m2 [ k, col ]
9                      k ← k + 1
10             fim-enquanto
11             col ← col + 1
12     fim-enquanto
13     coluna ← coluna + 1
14     linha ← linha + 1
15 fim-enquanto
```

Fazendo uma análise deste algoritmo, nota-se a presença de três estruturas de repetição (linhas 2, 4 e 7) aninhadas. Como cada uma delas serão iniciada e finalizada no primeiro elemento, observa-se a presença de uma repetição cúbica. A eficiência do algoritmo é $O(n^3)$.

Pode-se mostrar outro exemplo para ilustrar os passos a serem executados, para isto, calcular a notação O da seguinte função: $f(n) = 4n^2 + 2n + 3$

- primeiramente assumir que os coeficientes são iguais a 1, logo $f(n) = n^2 + n + 1$
- em seguida são removidos os fatores de menor importância: $f(n) = n^2$
- finalmente, a notação será: $O(f)n)) = O(n^2)$

1.8 – Análise de complexidade da Busca Linear

Nos algoritmos de operações em listas o parâmetro utilizado na análise de complexidade é o tamanho da lista, indicado por n. A operação preponderante em operações de busca é a comparação de valores dos registros.

Vamos considerar dois casos neste estudo:

- W(n) = número de comparações no pior caso;
- A(n) = número de comparações no caso médio.
- A(n) = número de comparações no melhor caso.

O algoritmo que vamos analisar é o algoritmo de busca da primeira ocorrência de um valor x na lista A. Supomos que A tenha n registros. A sequência de instruções abaixo é o algoritmo "BuscaPrimeiraOcorrencia":

```
1  j ← 1
2  enquanto ( A[ j ] ≠ x ) e (j < n) faça
3           j ← j + 1
4  fim-enquanto
5  se A[ j ] ≠ x
6    então sinal ← false
7    senão
8           sinal ← true
9           local ← j
10 fim-senão
```

A função de complexidade deve indicar, portanto, o número de vezes que é testada a condição (A[j] ≠ x).

1.8.1 – Pior Caso

A condição A[j] ≠ x ocorre como condição de controle do loop e, mais uma vez após o mesmo, na instrução de seleção.

A outra condição que controla o loop, (j < n), limita o número de repetições do loop a n - 1. Isto é, se a primeira condição tiver sempre o valor false, o loop será executado n-1 vezes, porque j é inicializado com valor 1. Entretanto, a condição que controla o loop é avaliada n vezes, que são as n -1 vezes que o loop é executado mais uma vez quando j atinge o valor n e torna a condição (j<n) falsa. Logo,

$W(n) = n + 1$

loop; comando de seleção

por outro lado, temos que W(n) = O(n), pois $\lim_{n \to \infty} \frac{W(n)}{n} = 1$

1.8.2 – Caso Médio

Se uma lista contém **n** registros, as possíveis entradas numa operação de busca são correspondentes a um valor x procurado em que: x ocorre no primeiro registro, ou x ocorre no segundo registro, ou x ocorre no terceiro registro, e assim por diante, ou x ocorre no último registro, ou ainda, x não pertence à lista. São, portanto, n+1 casos diferentes de entradas possíveis.

A cada uma dessas entradas devemos associar um número de probabilidade de que a entrada ocorra. Vamos supor que todas as entradas sejam igualmente prováveis, portanto com probabilidade igual a 1/(n+1).

Assim, $A(n) = p_1 x C_1 + p_2 x C_2 + ... + p_{n+1} x C_{n+1}$

Pode-se concluir que: A(n) = O(log n)

1.8.3 – Melhor Caso

Ocorre quando a condição **A[j] = x** é satisfeita na primeira ou na segunda execução do algoritmo. Fazendo assim com que o loop seja executado 1 ou 2 vezes somente.

Logo O(1)

1.8.4 – Um exemplo numérico

Supondo uma lista com três elementos (10, 20 e 30) e chave de busca igual a 90. Montando os passos da execução do algoritmo:

Passo do algoritmo	Chave (x)	n	Valor de j	A[j]	Comparação: A[j] = x	Valor de sinal	Quantidade de comparações
1	90	3	1	10	falso		1
2	90	3	2	20	falso		2
3	90	3	3	30	falso		3
4	90	3	3	30	falso	falso	4

Pode-se observar que para o pior caso, onde foi percorrida toda a lista, foram realizadas quatro (4) comparações, o que significa (n+1) comparações, pois n = 3. Se a lista for maior, vai crescer linearmente o número de comparações.

Supondo uma lista com três elementos (10, 20 e 30) e chave de busca igual a 10. Montando os passos da execução do algoritmo:

Passo do algoritmo	Chave (x)	n	Valor de j	A[j]	Comparação: A[j] = x	Valor de sinal	Quantidade de comparações
1	10	3	1	10	verdade		1
2	10	3	2	10	verdade	verdade	2

Pode-se observar que para o melhor caso, onde a chave procurada está na primeira posição da lista, foram realizadas duas (2) comparações, uma no laço e outra no teste para confirmar se foi encontrado ou não a chave procurada, o que significa duas (2) comparações. Se a lista for maior, este valor ficará constante.

1.9 – Exercícios

1.9.1 - Considerando a comparação como operação elementar, determine a complexidade do algoritmo abaixo:

a)
```
MAIOR (N, A)
max ← A [ 1 ]
para i de 2 até N repita
   Se max < A[ i ]
         então max ←A[ i ]
```

b)
```
ORDENA ( N, A)
para i de 1 até (N – 1) repita
   para j de 1 até (n – i ) repita
         se A[ j ] > A[ j + 1 ]
               então
                     x ← A[ j ]
                     A[ j ] ← A[ j + 1 ]
                     A[ j + 1 ] ← x
```

c)
```
n ← 1
enquanto (n ≤ 10) faça
  k ← 1
         enquanto ( k ≤ 10 ) faça
               ... trecho de pseudocódigo
               k ← K + 1
         fim-enquanto
         n ← n + 1
fim-enquanto
```

1.9.2 – verifique se as funções abaixo são O(n):

a) $f(n) = n$
b) $f(n) = 1045n$
c) $f(n) = n^2 + 70$
d) $f(n) = 7n + 3$
e) $f(n) = Cn + D$, onde C, D são constantes
f) $f(n) = 8$
g) $f(n) = n^3 + n + 1$
h) $f(n) = 4n + 2\log n + 5$

1.9.3 – Obter o valor de O para as expressões de complexidade:

a) $f(n) = 3n^3 + n$
b) $f(n) = 3 \log n + 5n$
c) $f(n) = 3n^2 + 5n + 4$
d) $f(n) = 3n^3 + n^2 + 5n + 99$

1.10 - Exercício Prático Algoritmo Busca a primeira ocorrência:

Desenvolver no laboratório o algoritmo "BuscaPrimeiraOcorrencia" em Pascal ou C.
Executar o exercício para uma lista com n = 11, 21 e 42 elementos. Para cada lista, x (o elemento procurado) deve estar na 2ª posição, na posição mediana (6, 11 e 21) e para x não existente. Os valores devem ser atribuídos no próprio programa fonte ou lidos uma única vez no início da execução.

a) colocar um contador após o enquanto e dentro do teste (se). Ao final de cada execução imprimir o contador. Ao final das execuções, fazer uma tabela e comparar os resultados encontrados.
b) no lugar do contador, colocar um delay. Ao iniciar a execução da rotina armazenar o horário do sistema e ao terminar a execução calcular o tempo gasto. Ao final das execuções, fazer uma tabela e comparar os resultados encontrados.
c) Fazer um documento texto analisando os dados encontrados. Apresentar as tabelas com os dados encontrados. O que voce pode concluir observando os dados encontrados nos itens **a** e **b** e a teoria apresentada?

A entrega do trabalho é em arquivo digital, onde deve conter o programa-fonte e o arquivo texto (em texto puro ou no word 97).
No programa-fonte deve conter o(s) nome(s) do(s) aluno(s) responsável(ies), o que deve ser feito quando mais de um aluno desenvolveu o programa, sendo aceito no máximo três (3) alunos. Deve apresentar também comentários explicando as principais passagens desenvolvidas.
No arquivo texto com os dados e as conclusões, deve conter o nome do aluno, e a informação (quando necessária) de que o programa foi desenvolvido juntamente com outros alunos (colocar os nomes) e em seguida os dados coletados e as conclusões. Observe que este arquivo é individual e as execuções para coleta dos dados também devem ser individuais.

Exemplo do uso do delay e de um controlador de tempo em linguagem C:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>
#include <dos.h>

clock_t inicio, fim; //declaração de variáveis: tempo de inicio e do final da execução

....

    inicio = clock();            //capturando o tempo de inicio da execução

....                             // parte da execução e que deve incluir o delay()

            delay(100);          //para que o programa demore mais para ser executado
...

    fim = clock();               //capturando o tempo de final de execução

....

    printf("\nTempo de execucao: %f.",(fim - inicio)/CLK_TCK);
                                          //calculo do tempo total = fim - inicio
```

2 – Estratégias Básicas

O maior problema em programas grandes reside na complexidade desses sistemas e na dificuldade de entender o funcionamento devido ao grande número de componentes e o grau de interação entre eles.

A ideia básica da programação estruturada é reduzir a complexidade através de:

- desenvolvimento do programa em diferentes fases por refinamento sucessivo;
- decomposição do programa total em módulos funcionais;
- usando dentro de cada módulo um número limitado de estruturas básicas.

A crescente dependência da sociedade em relação aos sistemas computacionais faz com que a disponibilidade e confiabilidade dos programas sejam uma questão crítica em muitas situações e ocasiões, requerendo uma atenção cada vez maior sobre o seu desenvolvimento e atualização.

2.1 – Refinamento Sucessivo

A metodologia de refinamentos sucessivos, também denominada de desenvolvimento top-down, parte do princípio de que resolver um problema complexo é mais fácil se não precisamos considerar todos os aspectos do problema simultaneamente. A decomposição de um problema grande numa série de subproblemas mais simples até chegar a um nível onde pode-se tentar uma solução é o objetivo básico da técnica de refinamento sucessivo.

A definição inicial é feita em um nível mais alto e geral; o processo avança em etapas, e em cada nova etapa as tarefas são decompostas em tarefas mais específicas, até que todos os problemas do programa estejam expressas em instruções claras e objetivas que podem ser utilizadas para a codificação em uma linguagem de programação. Então, cada nova fase do desenvolvimento, são agregados novos componentes que vão "refinando", chegando mais perto da solução, até que temos o algoritmo definitivo.

Antes de escrever um programa, é necessário um processo de raciocínio, que leva à análise do problema, passando por uma sequência de algoritmos cada vez mais detalhados, que consiste em uma sequência de passos simples que podem ser expressos diretamente em termos de comandos em uma linguagem de programação.

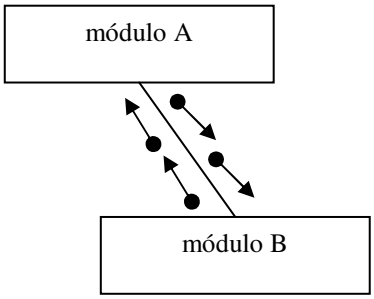
2.2 – Modularização

Durante a fase de projeto, a solução do problema total vai sendo organizada em soluções de subproblemas, o que permite dividir o programa em vários módulos com funções claramente definidas, que podem ser implementados separadamente por diversos programadores de uma mesma equipe.

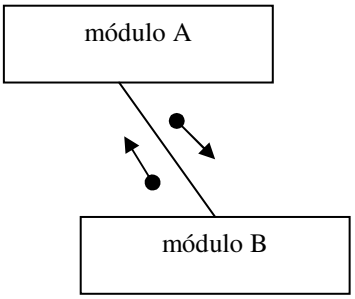
O módulo representa uma tarefa relacionada com o programa. Quando um comando ou conjunto de comandos executam uma única função que concorre para a solução do problema, ele pode ser considerado um módulo funcional, e pode ser tratado como um procedimento ou uma função.

O módulo tem apenas uma entrada e uma saída, e ambas fazem as ligações da estrutura de controle do programa e dos dados. Nestas ligações podem ocorrer passagens ou retorno de parâmetros.

Um bom algoritmo deve procurar reduzir a interação entre módulos (denominada de acoplamento) e aumentar o relacionamento dos elementos de um mesmo módulo (denominado de coesão).



módulos muito dependentes
(alto acoplamento)



módulos mais independentes
(baixo acoplamento)

2.3 – Confiabilidade X Complexidade

A confiabilidade de um produto é uma medida de sua capacidade de suportar o manuseio dentro de suas especificações de uso, mantendo-se capaz de atender às suas especificações de funcionamento. Quanto mais crítica a posição de um programa dentro de um sistema, mais confiabilidade lhe será exigida pelo usuário.

A confiabilidade de um agregado de componentes depende tanto da confiabilidade dos componentes isolados como da confiabilidade de sua interação. No caso dos produtos de software, os componentes elementares, isto é, as instruções são tão confiáveis quanto o hardware da máquina que os executam. A complexidade das interações, no caso dos programas, é um campo fértil à produção de defeitos em razão direta à complexidade do software.

Programas de aplicação, em sistemas de médio porte contém tipicamente da ordem de 104 a 105 instruções de máquina, selecionadas de um repertório da ordem de algumas centenas de instruções. Como a transferência de controle entre as instruções funcionará sempre corretamente se não houver defeito de hardware, os possíveis erros serão sempre causados por uma errônea escolha das instruções.

Deve-se usar técnicas que propõe a fornecer sistemáticas gerais para a construção de programas cuja confiabilidade pode ser verificada a priori, embutindo a confiabilidade no próprio projeto do programa.

3 – Divisão e Conquista

O método de desenvolvimento de algoritmos por divisão e conquista reflete a estratégia militar de dividir o exercito adversário para vencer cada uma das partes facilmente.

Dado um problema, de tamanho n, o método divide-o em k instancias disjuntas ($1 < k \leq n$) que corresponde a k subproblemas distintos. Cada subproblema é resolvido separadamente e então combina as soluções parciais para a obtenção da solução da instancia original. Em geral, os subproblemas resultantes são do mesmo tipo do problema original e neste caso, a aplicação sucessiva do método pode ser expressa naturalmente por um algoritmo recursivo, isto é, em um algoritmo denominado x, que tenha um dado de entrada de tamanho n, usamos o próprio x, k vezes, para resolver as sub-entradas menores do que n.

3.1 – Máximo e Mínimo de uma lista

Considerando o problema de achar o maior valor e o menor valor em uma lista **L** de **n** elementos de um conjunto **C**. Onde $L = (m_1, m_2, ..., m_n)$.

Um algoritmo trivial para calcular o máximo e o mínimo de **L** seria: considerar M_1 como sendo o máximo e o mínimo temporário; se o máximo temporário é menor que do que M_2, considerar então M_2 como o novo máximo temporário; se o mínimo temporário é maior do que M_2, considerar então M_2 como sendo o mínimo temporário; repetir o processo para M_3, ..., M_n. Após a comparação com M_n, temos que o máximo e o mínimo temporários são os valores desejados. Foram realizadas 2(n-1) comparações do máximo e mínimo temporários com os elementos da lista. Considerando o problema de encontrar o maior e o menor elemento de uma lista, podemos usar um algoritmo simples, descrito a seguir:

```
objetivo: encontrar o maior e o menor elemento da lista;
entradas: A (lista), N (inteiro)
saídas: max (inteiro), min (inteiro)
maxmim(A, max, min)
    max ← A[1];
    min ← A[1];
    Para I de 2 até N passo 1 repita
        Se A[I] > max então max ← A[I];
        Se A[I] < min então min ← A[I];
```

O algoritmo faz 2(n-1) comparações para uma lista com n elementos.

Este algoritmo pode ser facilmente melhorado, observando que a comparação **A[i] < min** só é necessária quando o resultado da comparação **A[i] > max** é falsa. Reescrevendo o algoritmo:

```
objetivo: encontrar o maior e o menor elemento da lista;
entradas: A (lista), N (inteiro)
saídas: max (inteiro), min (inteiro)
maxmim2(A, max, min)
    max ← A[1];
    min ← A[1];
    Para I de 2 até N passo 1 repita
        Se A[I] > max então max ← A[I]
```

```
                    senão Se A[I] < min então min ← A[I];
```

O melhor caso ocorre quando os elementos de A estão em ordem crescente, portanto N-1 comparações são necessárias.

O pior caso ocorre quando os elementos de A estão em ordem decrescente, portanto 2(N-1) comparações são necessárias..

No caso médio, A[I] é maior do que max a metade das vezes, portanto (3N/2 – 3/2) comparações são necessárias.

Considerando o número de comparações realizadas, podemos desenvolver um algoritmo mais eficiente, observando:

a) comparando os elementos de A aos pares, separando-os em dois subconjuntos de acordo com o resultado da comparação, os maiores em um subconjunto e os menores no outro subconjunto, a um custo de N/2 comparações;

b) o máximo é obtido do subconjunto que contém os maiores elementos, a um custo (N/2)-1 comparações;

c) o mínimo é obtido do subconjunto que contém os menores elementos, a um custo (N/2)-1 comparações.

O algoritmo é escrito:

```
objetivo: encontrar o maior e o menor elemento da lista;
entradas: A (lista), N (inteiro)
saídas: max (inteiro), min (inteiro)
maxmim3(A, max, min)
    Se (N mod 2) > 0 então
                        A[N+1] ← A[N]
                        FimDoAnel ← N
                    senão FimDoAnel ← N-1;
    Se A[1] > A[2] então
                        max ← A[1]
                        min ← A[2]
                  senão
                        max ← A[2]
                        min ← A[1];
    I ← 3;
    Enquanto I ≤ FimDoAnel repita
        Se A[I] > A[I+1] então
                            Se A[I] > max então max ← A[I];
                            Se A[I+1] < min então min ← A[I+1];
                        senão
                            Se A[I] < min então min ← A[I];
                            Se A[I+1] > max então max ← A[I+1];
        I ← I + 2;
    FimEnquanto
```

Os elementos de A são comparados de dois em dois e os elementos maiores são comparados com **max** e os menores com **min**. Quando **N** é impar, o elemento que está na posição A[N] é duplicado na posição A[N+1] para evitar um tratamento de exceção.

Este algoritmo faz (3N/2 – 2) comparações, independentemente da ordenação inicial da lista.

A tabela abaixo apresenta uma comparação entre os algoritmos, considerando o número de comparações como medida de complexidade.

Algoritmo	Melhor caso	Pior caso	Caso médio
maxmim	2 (n – 1)	2 (n – 1)	2 (n – 1)
maxmin2	n – 1	2 (n – 1)	3n/2 – 3/2
maxmin3	3n/2 – 2	3n/2 – 2	3n/2 – 2

Os algoritmos maxmin2 e maxmin3 são superiores ao maxmin de forma geral. O algoritmo maxmin3 é superior ao maxmin2 com relação ao pior caso, e é bastante próximo quanto ao caso médio.

Pode-se desenvolver um algoritmo mais rápido através da técnica de **divisão e conquista**. Para isto, dividimos a instancia L em duas sub-instancias L_1 e L_2 de comprimentos aproximadamente iguais:

$$L_1 = (M_1, M_2, ..., M_k) \text{ , onde } k = N/2$$

$$L_2 = (M_{k+1}, M_{k+2}, ..., M_n)$$

Resolvemos o problema considerando as sublistas L_1 e L_2, separadamente, obtendo-se as soluções (max_1, min_1) para L_1 e (max_2, min_2) para L_2. Para achar a solução, max será o maior entre max_1 e max_2 e min será o menor entre min_1 e min_2. Podemos notar que o algoritmo é recursivo, onde cada sublista pode ser novamente dividida.

A seguir a versão recursiva do algoritmo maxmin, onde consideramos o algoritmo para obter o maior e o menor elemento de um vetor de inteiros A[1...N] tal que N ≥ 1.

```
objetivo: encontrar o maior e o menor elemento da lista;
entradas: Linf (inteiro), Lsup (inteiro)
saídas: max (inteiro), min (inteiro)
maxmim4(Linf, Lsup, max, min)
    Se Lsup - Linf ≤ 1 então
                    Se A[Linf] < A[Lsup] então
                                        max ← A[Lsup]
                                        min ← A[Linf]
                                    senão
                                        max ← A[Linf]
                                        min ← A[Lsup];
                senão
                    meio ← (Linf + Lsup) div 2
                    maxmin4(Linf, meio, max1, min1)
                    maxmin4(meio+1, Lsup, max2, min2)
                    Se max1 > max2 então max ← max1
                                    senão max ← max2;
                    Se min1 < min2 então min ← min1
                                    senão min ← min2;
```

Observe que o vetor A é uma variável global ao procedimento maxmin4 e os parâmetros Linf e Lsup são inteiros e $1 \leq Linf \leq Lsup \leq N$.

A função f(n) é o número de comparações entre os elementos de A, tal que:

$f(n) = 1$, para $n \leq 2$

$f(n) = f([n/2)] + f([n/2]) + 2$, para $n > 2$

desenvolvendo, tem-se que $f(n) = 3n/2 – 2$ para qualquer caso.

É observando o mesmo comportamento do algoritmo maxmin3, entretanto, na prática o algoritmo maxmin4 deve ser pior que os algoritmos maxmin2 e maxmin3 porque na implementação recursiva, a cada nova chamada do procedimento maxmin4, o programa salva em uma estrutura de dados (pilha) os valores **Linf, Lsup, max** e **min**, além do endereço de retorno da chamada para o procedimento. No final do algoritmo também aparece uma comparação adicional $Lsup – Linf \leq 1$.

A nova tabela apresenta uma comparação entre os algoritmos, considerando o número de comparações como medida de complexidade.

Complexidade (em número de comparações)			
Algoritmo	**Melhor caso**	**Pior caso**	**Caso médio**
maxmim	2 (n – 1)	2 (n – 1)	2 (n – 1)
maxmin2	n – 1	2 (n – 1)	3n/2 – 3/2
maxmin3	3n/2 – 2	3n/2 – 2	3n/2 – 2
maxmin4	3n/2 – 2	3n/2 – 2	3n/2 – 2

Rapidez e certificação do maxmin4

Em um algoritmo como o maxmin4, a operação elementar a ser considerada na análise de rapidez é a comparação entre dois elementos da lista de entrada, e a contagem do número de comparações efetuadas é a medida do tempo de execução do algoritmo. Essa escolha de operação elementar é justificada em grande parte pelo fato de que os elementos podem ser objetos com mais componentes do que um simples número inteiro e então o tempo de execução é determinado principalmente pelo tempo necessário para se efetuar todas as comparações.

Seja T(n) o tempo de execução do algoritmo para uma lista de entrada com n elementos. Se n=1, vemos que t(1) = 0 pois nenhuma comparação é efetuada. Se n = 2, efetua-se uma comparação na linha 2 e portanto T(2) = 1. se n > 2, a execução do algoritmo na linha 10 requer T(n – [n/2]) = T([n/2]), e a execução da linha 11 do algoritmo requer também T([n/2]) e as linhas 12 e 14 requerem 2 comparações. Os casos acima podem ser resumidos na fórmula de recorrência:

$T(1) = 0, T(2) = 1, T(n) = T([n/2]\} + T([n/2]) + 2$, se $n > 2$.

Quando n é uma potencia de 2, isto é $n = 2^k$ para algum inteiro positivo k, tem-se:

$$\begin{aligned} T(n) &= 2T(n/2) + 2 = 2(2T(n/4) + 2) = \ ... = \\ &= 2^{k-2} + 2[2T(2) + 2] + (2^{k-2} + 2^{k-3} + ... + 2) = \\ &= 2^k + (2^{k-1} – 2) = (3n/2) - 2 \end{aligned}$$

Portanto, comparado com as **2 (n – 1)** comparações realizadas pelo algoritmo trivial (MaxMin), o MaxMin4 faz uma economia de aproximadamente 25% no número de comparações. Deve-se observar que **(3n/2 –2)** é o número de comparações tanto pessimista quanto média do algoritmo MaxMin4.

3.2 – Exercício MaxMin

Desenvolver no laboratório os algoritmos "MaxMin", "MaxMin2", "MaxMin3" e "MaxMin4" em linguagem C.

Executar o exercício para uma lista com 40 elementos. Onde a situação inicial é:

a) Uma lista qualquer;
b) Uma lista gerada aleatoriamente pelo programa;
c) Uma lista já ordenada ascendente;
d) Uma lista já inversamente ordenada, isto é, ordenada descendente.

Os valores devem ser atribuídos no próprio programa fonte ou lidos uma única vez no início da execução.

a) colocar um contador para calcular o número de comparações executadas. Ao final de cada execução imprimir o contador. Ao final das execuções, fazer uma tabela e comparar os resultados encontrados.
b) no lugar do contador, colocar um delay. Calcular o tempo gasto. Ao final das execuções, fazer uma tabela e comparar os resultados encontrados.
c) Fazer um documento texto analisando os dados encontrados. Deve apresentar as tabelas com os dados encontrados e os valores esperados, quando possível. O que voce pode concluir observando os dados encontrados nos itens **a** e **b** e a teoria apresentada?

A entrega do trabalho é em dois arquivos digitais, onde o primeiro deve conter o programa-fonte e segundo arquivo texto (no word 2003 ou anterior).

No programa-fonte deve conter o(s) nome(s) do(s) aluno(s) responsável(ies), o que deve ser feito quando mais de um aluno desenvolveu o programa, sendo aceito no máximo três (3) alunos. O programa fonte deve apresentar também comentários explicando as principais passagens desenvolvidas.

No arquivo texto deve estar os dados coletados para cada situação proposta e para cada uma deve ter o número de comparações e tempo gasto, e as conclusões do aluno. Este arquivo deve conter o nome do aluno, e a informação (quando necessária) de que o programa foi desenvolvido juntamente com outros alunos (colocar os nomes) e em seguida os dados coletados e as conclusões. Observe que este arquivo é individual e as execuções para coleta dos dados também devem ser individuais.

3.3 – Ordenação por Intercalação

Considerando o problema de ordenar uma lista de **n** objetos solucionado anteriormente por um algoritmo $O(n^2)$. Podemos estudar algoritmos mais eficientes.

Para se ordenar uma lista L = (m_1, m_2, ...,m_n) por intercalação, divide-se L em duas sublistas L_1e L_2, tais que:

$L_1 = (m_1, m_2, ...,m_k)$, $L_2 = (m_{k+1}, m_{k+2}, ...,m_n)$, onde k = (n/2)

Em seguida L_1 e L_2 são ordenadas separadamente e são obtidas duas sublistas ordenadas L'_1 e L'_2, de acordo com a estratégia de divisão e conquista. Agora, as soluções parciais L'_1 e L'_2 devem ser combinadas para se obter a solução final. Isto pode ser feito intercalando-se os elementos de L'_1 e L'_2 da seguinte forma:

seleciona-se o menor elemento entre $\mathbf{m'_1}$ e $\mathbf{m'_{k+1}}$ (em caso de igualdade seleciona-se $\mathbf{m'_1}$, por exemplo), este elemento é retirado da lista à qual pertence e colocado numa lista L'; repete-se a seleção do menor, sendo que agora o elemento é escolhido entre $\mathbf{m'_2}$ e $\mathbf{m'_{k+1}}$ (se o escolhido foi $\mathbf{m'_1}$) ou entre $\mathbf{m'_1}$ e $\mathbf{m'_{k+2}}$ (se o escolhido foi $\mathbf{m'_{k+1}}$), e um segundo elemento é colocado na lista L'; e este processo de seleção do menor é repetido até que se obtenham **n** elemento em L'.

Não é difícil observar que a lista resultante estará em ordem crescente, e que toda a intercalação pode ser feita com **n-1** comparações e que portanto em tempo **cn**, onde c é uma constante c>1.

O algoritmo que descrevemos necessita somente da lista a ser ordenada e de uma lista auxiliar interna ao procedimento para armazenar temporariamente a lista durante a intercalação.

```
objetivo: ordenar uma lista por intercalação de duas sublistas
entradas: L (lista), N (tamanho da lista - inteiro)
saída: L (lista ordenada crescente)
OrdInter(n,L)
1       se N <= 1 então
2                   ordinter ← m1
3               senão
4                   k ← (n/2);
5        L1 ← ordinter(k, L);
6       L2 ← ordinter(n-k, L);
7       L ← Intercala(L1, L2)
8       ordinter ← L;
9       fim-se;
```

desenvolver a intercalação conforme descrito acima

Rapidez da Ordenação por intercalação

Seja T(n) o tempo de execução do OrdInter para uma entrada de tamanho **n**. A operação elementar a ser considerada em T(n) será a comparação entre dois elementos m_i. Se n = 1, observamos no algoritmo que T(1) = 0. Se n > 1, a execução na linha 5 requer tempo T(n/2) e na linha 6 também T(n/2). Como foi dito acima, a intercalação na linha 7 requer tempo cn, para uma constante c>1. Temos então a formula de recorrência:

$T(1) = 0$

$T(n) = T(n/2) + T(n/2) + cn$, para $n > 1$

Quando n é uma potencia de 2, tal que $n = 2^k$ (para $k > 0$), por substituições sucessivas temos:

$$\begin{aligned} T(n) &= 2\,T(n/2) + cn \\ &= 2\,[\,2\,T(n/4) + cn/2] + cn \\ &= 2^k\,T(n/2^k) + kcn \\ &= cn \lg n \end{aligned}$$

Quando n não é necessariamente uma potencia de 2, n satisfaz $2^{k+1} < n \leq 2^k$ (para algum $k > 0$). E como T(n) é crescente com n, $T(n) \leq T(2^k)$ e tem-se:

$$\begin{aligned} T(n) / (n \lg n) &\leq c\,2^k\,k / (n \lg n) \\ &\leq c\,2n\,(\lg n + 1) / (n \lg n) \\ &\leq 2c \end{aligned}$$

portanto, $T(n) = \mathbf{O(n \lg n)}$

Conclusão

Para uma lista de entrada de tamanho n, o algoritmo OrdInter tem rapidez pessimista

$T(n) = O(n \lg n)$

Este problema de ordenação tem cota inferior $\mathbf{\Omega(n \lg n)}$ e portanto o algoritmo é ótimo em termos de rapidez.

Tanto no algoritmo OrdInter quanto no algoritmo MaxMin da seção anterior, a estratégia de divisão e conquista foi aplicada obtendo-se duas subinstancias de tamanhos aproximadamente iguais. Isto não é uma coincidencia, e foi feito de acordo com uma orientação básica que chamaremos de *Princípio da Equipartição*.

Para ilustrar o Princípio da Equipartição, suponhamos que as subinstancias sejam de tamanho um e n-1 no algoritmo OrdInter. Então, a fórmula de recorrência de T(n) passa a ser:

$T(1) = 0$

$T(n) = T(n-1) + cn$, para $n>1$

que tem solução $T(n) = O(n^2)$. Portanto, tem-se um algoritmo mais lento do que o algoritmo estudado anteriormente.

3.4 – Ordenação por Concatenação

Este algoritmo, apesar de ter uma rapidez pessimista **O(n^2)**, ele tem uma rapidez média **O(nlgn)** que é ótima. Podemos observar que a ordenação por concatenação requer pouco espaço auxiliar, e a constante de proporcionalidade na sua rapidez é relativamente pequena. O fato do algoritmo ter rapidez pessimista quadrática não é importante em muitas aplicações.

Dada uma lista L = (m_1, m_2, ...,m_n), o algoritmo de ordenação por intercalação da seção anterior divide-a em duas, num ponto próximo da metade de L. Na ordenação por concatenação, a divisão em duas sublistas se faz de modo que, após a ordenação das sublistas, não haverá a necessidade de intercalá-las. Se pudermos dividir L em duas sublistas L_1 e L_2, tais que todos os elementos em L_1 são menores que os elementos em L_2, então L_1 e L_2 podem ser ordenados separadamente e não haverá necessidade de intercalação posteriormente. Esta divisão pode ser obtida eficientemente da seguinte forma: escolhe-se um elemento qualquer **d=m_i** de L, e reordenam-se os outros elementos de L, de tal forma que todos os elementos que ocorram em L antes de **d** sejam menores do que **d** (a lista L_1) e todos os elementos depois de **d** sejam iguais ou maiores do que **d** (a lista L_2). Esta reordenação é denominada de *Partição* de L segundo d.

A escolha de **d** será analisada posteriormente.

O algoritmo partição faz a partição descrita acima, sendo que, por simplicidade, supomos que d = m_m. Nestas condições, o algoritmo utiliza dois apontadores, **p** e **u**, com valores iniciais p = m e u = n.

objetivo: particionar uma lista
entrada: m e n, onde n-m ≥ 0 e L = (d, m_{m+1},...,m_n) que é uma lista e d=m_{m+1},...,m_n
saída: (j , L), onde m ≤ j ≤ n, e L está reordenado de tal forma que m_m,...,m_{j-1} são menores do que d, e m_j, m_{j+1},...,m_n são maiores ou iguais a d (sendo m_j=d).

Particao(m,n,L)

```
1    p ← m;
2    u ← n;
3    enquanto p < u faca
4        enquanto p < u e m_p ≤ m_u faca
5            u ← u – 1
6        fim-enquanto;
7        se p < u então
8                "troque m_p e m_u entre si";
9                p ← p + 1;
10       fim-se;
11       enquanto m_p < m_u faca
12           p ← p + 1
13       fim-enquanto;
14       se p < u então
15               "troque m_p e m_u entre si";
16               u ← u – 1;
17       fim-se;
18   fim-enquanto;
19   particao ← (p , L)
```

Compara-se m_p com m_u. Se $m_p \leq m_u$, faz-se $u \leftarrow u - 1$ e faz-se uma nova comparação. Decrementa-se u desta forma, até que $m_p > m_u$. Então trocam-se os valores de m_p e m_u entre si e faz-se $p = p + 1$ (o novo valor de m_u é d). Continuamos a incrementar p até que $m_p \geq m_u$. Então, trocam-se os valores de m_p e m_u, entre si, faz-se $u \leftarrow u - 1$ e voltamos as decrementações de p, u e p se encontram "no meio"de L, isto é, chega-se a p = u. Nesta situação, observam-se duas propriedades:

a) o valor de d, que ocupava a primeira posição na lista, deslocou-se para a posição que deverá ocupar na lista ordenada;

b) todos os elementos que estão à esquerda de d são menores do que d, e os que estão à direita de d são maiores ou iguais a d.

Após obter-se p = u no algoritmo partição, pode-se então aplicar recursivamente este algoritmo sobre a sublista L_1 $(m_m,...,m_{p-1})$ e L_2 $(m_{p+1},...,m_n)$, separadamente. Esta é a ideia básica do algoritmo de ordenação por concatenação.

objetivo: ordenar uma lista por concatenação de sub-listas
entrada: (n , m , L) onde $n - m \geq 0$ e $L = (m_m,...,m_n)$ é uma lista
saída: a lista L em ordem não decrescente

OrdConc(m, n, L)

```
1    L1 ← 0; L2 ← 0     {* listas inicialmente vazias*}
2    se n – m + 1 = 1 então
3                              OrdConc ← m1
4    fim-se;
5    mi ← "um elemento escolhido de L, é o d mencionado acima";
6    "troque mm e mi entre si";
7    (j, L) ← particao (m, n, L);
8    se j > m então
9                    L1 ← OrdConc(m, j, L);
10   fim-se;
11   se j < n então
12                   L2 ← OrdConc(j, n, L);
13   fim-se;
14   OrdConc ← "concatenar: L1 a Mj e a L2";
```

3.5 – Busca Binária

Dada uma lista L = (m_1, m_2, ...,m_n), com n elementos e considerando o problema de verificar se um dado elemento **x** está presente ou não em L. Um algoritmo trivial para resolver este problema consiste em comparar iterativamente x com m_1, x com m_2, e assim sucessivamente até encontrar um m_j tal que m_j=x, ou até concluir que o tal elemento não existe em L e a resposta será negativa, fazendo-se j=0. Este algoritmo tem rapidez pessimista O(n).

Supondo que os elementos M_i da lista L pertençam a um conjunto com relação de ordem linear, e então pode-se usar um algoritmo de ordenação sobre L em tempo O(n lg n). E tendo-se L ordenado, pode-se usar um algoritmo desenvolvido por divisão e conquista que resolve o problema em tempo O (n lg n).

Supondo que L está ordenada em ordem crescente. Então, em uma instancia I = (x, n, L = (M_1, ..., M_n)), calcula-se um índice k, onde, $1 \leq k \leq n$, e compara-se x com M_k. Se $x < M_k$, então x pode estar na sub-lista L_1 = (M_1, ..., M_{k-1}) e portanto, deve-se resolver a sub-instancia I_1 = (x, k-1, l_1). Senão, x pode estar na sub-lista L_2 = (M_k, ..., M_n) e deve-se resolver a sub-instancia I_2 = (x, n-k+1, L_2). A resolução de I_1 ou I_2 pode ser novamente feita por divisão destas instancias.

Usando o princípio da equipartição, o índice k deve ser de tal forma que as listas L_1 e L_2 são comprimento aproximadamente iguais.

Algoritmo BuscaBin: que verifica se um dado elemento x está presente em uma lista.

Entrada: (x, n, L = (M_1, M_2, ..., M_n)), onde $n \leq 1$ e x, M_1, M_2, ..., M_n são elementos de um conjunto com relação de ordem linear e L está ordenado crescente.

Saída: (resposta, j), onde j é o índice de um elemento M_j = x em L, se a resposta é "presente" e j = 0 se a resposta é "ausente".

Algoritmo:

```
1       se n = 1
2       entao se x = Mn então pare-com-saida ("presente", n)
3                            senao pare-com-saida ("ausente", 0)
4                fim_se
5       senao
6            k ← [ (n + 1) / 2]
7            se x < Mk entao (resposta, j) ← BuscaBin(x, k–1, M1, ..., Mk-1)
8                        senao (respost, j) ← BuscaBin(x, n-k+1, Mk, ..., Mn)
9                                    j ← j + 1
10           fim_se
11           pare-com-saida (resposta, j)
12      fim_se
```

Rapidez do BuscaBin

Seja T(n) a rapidez pessimista do BuscaBin para uma entrada (x, n, L = (M_1, ..., M_n)). Pode-se observar no algoritmo que, se n = 1, T(1) = 1, a comparação é efetuada na linha 2. Se n > 1, faz-se uma comparação na linha. Calculando k – 1 e n – k + 1 para os casos de n ser par ou ser impar respectivamente, conclui-se que a execução na linha 8 requer um tempo

T([n/2]), e a execução na linha 10 requer T([n/2]). Pode-se observar que apenas uma das opções será executada.

Considerando a rapidez pessimista, e como T([n/2]) ≥ T([n/2]), tem-se a seguinte fórmula de recorrência

T(1) = 1, T(n) = 1 + T([n/2]), para n > 1.

Quando n é uma potencia de 2, pode-se facilmente deduzir que tem solução T(n) = lg n.

Logo, tem-se que T(n) = O (lg n).

Através de uma árvore de decisão, pode-se provar que o problema de busca por comparações de um elemento numa lista ordenada tem cota inferior Ω(lg n). Portanto, o algoritmo BuscaBin tem rapidez ótima.

3.6 – Exercício prático de Busca Binária

Desenvolver no laboratório o algoritmo "BuscaBin" em linguagem C.

Executar o exercício para uma lista com 60 elementos ordenados em ordem crescente. Onde deve-se procurar:

a) O primeiro elemento da lista;
b) O trigésimo elemento da lista;
c) O quadragésimo quinto elemento da lista;
d) O sexagésimo elemento da lista;
e) Um elemento inexistente da lista;

Os valores devem ser atribuídos no próprio programa fonte ou lidos uma única vez no início da execução. Calcular o tempo gasto em cada execução.

Colocar um contador para calcular o número de comparações executadas. Ao final de cada execução imprimir o contador. Ao final das execuções, fazer uma tabela e comparar os resultados encontrados.

No lugar do contador, colocar um delay. Calcular o tempo gasto. Ao final das execuções, fazer uma tabela e comparar os resultados encontrados.

Fazer um documento texto analisando os dados encontrados. Deve apresentar as tabelas com os dados encontrados e os valores esperados, quando possível. O que voce pode concluir observando os dados encontrados nos dois itens anteriores e a teoria apresentada?

A entrega do trabalho é em dois arquivos digitais, onde o primeiro deve conter o programa-fonte e segundo arquivo texto (no word 2003 ou anterior).

No programa-fonte deve conter o(s) nome(s) do(s) aluno(s) responsável(ies), o que deve ser feito quando mais de um aluno desenvolveu o programa, sendo aceito no máximo três (3) alunos. O programa fonte deve apresentar também comentários explicando as principais passagens desenvolvidas.

No arquivo texto deve estar os dados coletados para cada situação proposta e para cada uma deve ter o número de comparações e tempo gasto, e as conclusões do aluno. Este arquivo deve conter o nome do aluno, e a informação (quando necessária) de que o programa foi desenvolvido juntamente com outros alunos (colocar os nomes) e em seguida os dados coletados e as conclusões. Observe que este arquivo é individual e as execuções para coleta dos dados também devem ser individuais.

3.7 - Lista de Exercícios da Unidade

1 Explique como funciona o método de desenvolvimento de algoritmos denominado de "Divisão e conquista"?

2 O método de desenvolvimento de algoritmos "divisão e conquista" pode ser utilizado para desenvolver algoritmos recursivos? Por que?

3 Montar a tabela de complexidade (melhor, pior e caso médio) para os algoritmos maxmin, maxmin2, Maxmin3, e Maxmin4 e responda:

a) Observando a teoria apresentada, qual (is) algoritmo (s) usaria para o caso médio? Explique a sua resposta.

b) Observando a teoria apresentada, se tivesse que escolher um algoritmo para usar em qualquer situação, qual utilizaria? Explique a sua resposta.

4 Qual a restrição de utilizar a versão recursiva em relação a versão denominada Maxmin3? Explique a sua resposta.

5 Explique a metodologia do algoritmo denominado "ordenação por intercalação".

6 Explique a metodologia do algoritmo denominado "Busca binária".

7 Explique a principal utilidade dos algoritmos denominados de "Divisão e conquista" para o programador que necessita de utilizar algoritmos que trabalham com busca e reconhecimento de elementos em uma lista.

8 Desenvolvemos um grupo de programas baseados em um conjunto de algoritmos que determinam o maior e o menor elemento de uma lista, e ao executá-los, para algumas situações, foram determinados os números de comparações descritos na tabela abaixo. Baseando-se nos dados da tabela e nos conceitos estudados para cada um dos algoritmos, responda as perguntas abaixo.

Tabela 1 de números de comparações (n – tamanho da lista)			
Algoritmo	Melhor caso	Pior caso	Caso médio
maxmim4	3N/2 – 2	3N/2 - 2	3N/2 – 2
maxmin3	3N/2 – 2	3N/2 - 2	3N/2 – 2
maxmin2	N – 1	2 (N – 1)	3N/2 – 3/2
maxmin	2(N – 1)	2 (N – 1)	2 (N – 1)

Tabela 2 de tempo médios (em milisegundos) (tamanho da lista: 120)			
Algoritmo	Melhor caso	Pior caso	Caso médio
maxmim4	178	180	179
maxmin3	177	239	180
maxmin2	120	235	178
Maxmin	237	240	239

a) Se tivesse de escolher um algoritmo para usar, sem saber a situação inicial da lista, qual usaria? justifique a resposta.

b) Se tivesse de escolher um algoritmo para usar, sabendo que a situação é de pior caso em uma lista muito grande e que tem possibilidade de continuar a crescer rapidamente, qual dos algoritmos voce usaria? justifique a resposta.

c) Os tempos (valores) encontrados na tabela 2 são compatíveis com o número de comparações apresentador na tabela 1? Justifique a resposta.

d) Para uma lista com N=30, qual seriam os valores que deveriam ser encontrados?

Tabela de números de comparações			
Algoritmo	Melhor caso	Pior caso	Caso médio
maxmim4			
maxmin3			
maxmin2			
maxmin			

9 Dada a tabela resultados da execução do algoritmo "Busca Primeira Ocorrência", para três listas com 20, 40 e 80 elementos:

Tempo gasto (segundos)			
Exercício	Tempo gasto na 2º posição	Tempo gasto na posição mediana	Tempo gasto na posição não existente
Lista 1: 20 elementos	0.13	0.56	1.11
Lista 2: 40 elementos	0.15	1.09	2.19
Lista 3: 80 elementos	0.14	2.18	4.33
Lista 4: 160 elementos			

a) O que podemos concluir sobre a ordem de complexidade? Explique como chegou no resultado.

b) Este resultado é compatível com a teoria estudada?

c) Por que na posição mediana o tempo aumentou e na busca do elemento que estava na Segunda posição ele não aumentou quando do aumento do tamanho da entrada?

d) Partindo dos resultados apresentados, quais serão os tempos esperados se executarmos o algoritmo com entrada igual a 160 elementos? pode deixar os cálculos indicados se quiser.

4 - Métodos de Ordenação

Os métodos de ordenação constituem um bom exemplo de como resolver problemas utilizando computadores. As técnicas de ordenação permitem apresentar um conjunto amplo de algoritmos distintos para resolver uma mesma tarefa. Dependendo da aplicação, cada algoritmo considerado possui uma vantagem particular sobre os outros algoritmos.

Ordenar consiste no processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente. O objetivo principal da ordenação é facilitar a recuperação posterior de itens do conjunto ordenado.

Um método de ordenação é dito estável se a ordem relativa dos itens com chaves iguais mantém-se inalterada pelo processo de ordenação. Por exemplo, se uma lista alfabética de nomes de funcionários de uma empresa é ordenada pelo campo salário, então um método estável produz uma lista em que os funcionários com o mesmo salário aparecem em ordem alfabética. Alguns dos métodos de ordenação mais eficientes não são estáveis.

Os métodos de ordenação são classificados em dois grandes grupos. Se o arquivo a ser ordenado cabe todo na memória principal, então o método de ordenação é chamado de ordenação interna. Neste caso, o número de registros a ser ordenado é pequeno o bastante para caber em um array do pascal, por exemplo. Se o arquivo a ser ordenado não cabe na memória principal e, por isso, tem que ser armazenado em fita ou disco, então o método de ordenação é chamado de ordenação externa. A principal diferença entre os dois métodos é que, em um método de ordenação interna, qualquer registro pode ser imediatamente acessado, enquanto em um método de ordenação externa, os registros são acessados sequencialmente ou em grandes blocos.

4.1 - Métodos de Ordenação Interna

O aspecto predominante na escolha de um algoritmo de ordenação é o tempo gasto para ordenar um arquivo. Para algoritmos de ordenação interna as medidas de complexidade relevantes contam o número de comparações entre chaves e o número de movimentações (ou trocas) de itens do arquivo. Seja C uma função de complexidade tal que C(n) é o número de comparações entre chaves, e seja M uma função de complexidade tal que M(n) é o número de movimentações de itens no arquivo, onde n é o número de itens do arquivo.

A quantidade extra de memória auxiliar utilizada pelo algoritmo é também um aspecto importante, o uso econômico da memória disponível é um requisito primordial na ordenação interna. Os métodos que utilizam a estrutura vetor e que executam permutação dos itens no próprio vetor, exceto para a utilização de uma pequena tabela ou pilha são os preferidos.

Os métodos de ordenação interna são classificados em métodos simples e métodos eficientes. Os métodos simples são adequados para pequenos arquivos e requerem O(n2) comparações, enquanto os métodos eficientes são adequados para arquivos maiores e requerem O(n log n) comparações. Os métodos simples produzem programas pequenos, fáceis de entender, ilustrando com simplicidade os princípios de ordenação por comparação. Apesar de os métodos mais sofisticados usarem menos comparações, estas comparações são mais complexas nos detalhes, o que torna os métodos simples eficientes para pequenos arquivos.

4.2 - Método de ordenação por Seleção

A ordenação por seleção consiste, em cada etapa, em selecionar o maior (ou o menor) elemento e alocá-lo em sua posição correta dentro da futura lista ordenada. Durante a execução, a lista com n registros é decomposta em duas sublistas, uma contendo os itens já ordenados e a outra com os elementos ainda não ordenados. No início, a sublista ordenada está vazia e a desordenada contém todos os elementos, no final do processo a sublista ordenada apresentará (n-1) elementos e a desordenada terá um elemento.

Vamos estudar o método de ordenação por seleção direta do maior. Inicialmente localizamos na lista desordenada o índice j onde se encontra o maior elemento e depois fazemos a permuta do conteúdo de A[j] por A[n]. Prosseguindo, localizamos na sublista desordenada o índice j do maior elemento e depois permutamos A[j] por A[n-1], ficando assim uma sublista desordenada com (n-2) elementos e uma sublista ordenada com 2 elementos. O processo é repetido até que a lista ordenada tenha (n-1) elementos e a lista desordenada tenha 1 elemento.

4.2.1 - Algoritmo de Ordenação por seleção:

Entrada: n (inteiro- número de elementos), A (lista ou vetor com os elementos)

Saída: A (lista ou vetor com os elementos ordenados)

```
Selecao (n, A)
  Para i de 1 até (n - 1)
  repita
   min ← i;
   Para j de (i + 1) até n
   repita
        Se A[j] < A[min]
          Entao min ← j;
   fim-para;
   aux ← A[min];
   A[min] ← A[i];
   A[i] ← aux;
  fim-para.
```

```
Ord (n, A)
    Para i de n até 2 passo -
    1 repita
     j ← i;
     Para k de 1 até (i - 1)
     repita
          Se A[j] < A[k]
            Entao j ← k;
     fim-para;
     Se i ≠ j
       Entao   aux ← A[i];
               A[i] ← A[j];
               A[j] ← aux;
    fim-para.
```

4.2.2 - Análise de complexidade

i) operação entre as chaves é feita no loop **k**, para cada valor de **i** são realizadas (i-1) comparações no loop, como **i** varia de **2** até **n**, o número total de comparações para ordenar a lista toda é:

$$C(n) = \sum_{i=2}^{n} (i-1) = \frac{1}{2}(n-1)\,n = \frac{1}{2}(n^2 - n) = O(n^2)$$

operação de movimentação de registros, para qualquer valor de **i** existe no máximo uma troca. Se a lista já está ordenada, não ocorre nenhuma troca, portanto:

bM(n) = 0.
No pior caso, existe uma troca para cada loop de k, portanto:
wM(n) = 3 (n-1) , porque uma troca exige três movimentos.

O algoritmo de ordenação por seleção é um dos métodos de ordenação mais simples que existem. Além disso, o método possui um comportamento espetacular quanto ao número de movimentos de registros, cujo tempo de execução é linear no tamanho da entrada, o que é muito difícil de ser batido por qualquer outro método. Consequentemente, este é o algoritmo a ser utilizado para arquivos com registros muito grandes. Em condições normais, com chaves do tamanho de uma palavra, este método é bastante interessante para arquivos com até 1000 registros. Como aspectos negativos cabe registrar que:

a) o fato do arquivo já estar ordenado não ajuda em nada, pois o custo continua quadrático;

b) o algoritmo não é estável, pois ele nem sempre deixa os registros com chaves iguais na mesma posição relativa.

4.3 - Método de ordenação por Inserção

A ordenação por Inserção é quase tão simples quanto o algoritmo de ordenação por Seleção, e além disso é um método estável, pois deixa os registros com chaves iguais na mesma posição relativa.

O método consiste em cada passo, a partir de i=2, o i-ésimo item da sequência fonte é apanhado e transferido para a sequência destino, sendo colocado na posição correta. A inserção do elemento no lugar de destino é efetuado movendo-se os itens com chave maiores para a direita e então inserindo-o na posição que ficou vazia. Neste processo de alternar comparações e movimentação de registros existem duas situações que podem causar o término do processo, a primeira é quando um item com chave menor que o item em consideração é encontrado e a segunda situação é quando o final da sequência destino é atingido (à esquerda). A melhor solução para a situação de um anel com duas condições de terminação é a utilização de uma sentinela, para isto, na posição zero do vetor colocamos o próprio registro analisado.

4.3.1 - Algoritmo de Ordenação por Inserção

Entrada: A (lista ou vetor de elementos), n (tamanho da lista)

Saída: A (lista ou vetor de elementos)

```
Inserção (A, n)
    1  Para i=2 até n faca
    2       x ← A[i];
    3       j ← i - 1;
    4       A[0] ← x;
    5       Enquanto x < A[j] faca
```

```
6           A[j+1] ← A[j];
7           j ← j - 1;
8       fim-enquanto;
9       A[j+1] ← x;
10  fim-para.
```

4.3.2 - Análise de complexidade do algoritmo

No anel mais interno, na i-ésima iteração, o valor de C_i é:

- Melhor caso: $C_i(n) = 1$
- Pior caso: $C_i(n) = i$
- Caso médio: $C_i(n) = {}^1/i(1 + 2 + ... + i) = {}^{(i+1)}/_2$

Assumindo que todas as permutações de n são igualmente prováveis para o caso médio, logo, o número de comparações é igual a

- Melhor caso: $C(n) = (1 + 1 + ... + 1) = n - 1$
- Pior caso: $C(n) = (2 + 3 + ... + n) = n^2/_2 + {}^n/_2 - 1$
- Caso médio: $C(n) = {}^1/_2(3 + 4 + ... + n+1) = n^2/_4 + {}^{3n}/_4 - 1$

O número de movimentações na i-ésima iteração é igual a

$$M_i(n) = C_i(n) - 1 + 3 = C_i(n) + 2$$

Logo, o número de movimentos é igual a

- Melhor caso: $M(n) = (3 + 3 + ... + 3) = 3(n - 1)$
- Pior caso: $M(n) = (4 + 5 + ... + n+2) = n^2/_2 + {}^{5n}/_2 - 3$
- Caso médio: $M(n) = {}^1/_2(5 + 6 + ... + n+3) = n^2/_4 + {}^{11n}/_4 - 3$

Deste modo podemos concluir que:

- Melhor caso: **O(n)**
- Pior caso: **$O(n^2)$**
- Caso médio: **$O(n^2)$**

O número mínimo de comparações e movimentos ocorre quando os elementos estão originalmente em ordem, e o número máximo ocorre quando os itens estão originalmente na ordem reversa, o que indica um comportamento natural para o algoritmo. Para arquivos já ordenados o algoritmo descobre a um custo O(n) que cada item já está em seu lugar. Logo, este é o método a ser utilizado quando o arquivo está "quase" ordenado. É também um método bom para adicionar um pequeno conjunto de dados a um arquivo já ordenado, originado um outro arquivo ordenado, pois neste caso o custo pode ser considerado linear.

4.4 - Método de ordenação Quicksort

É um método de ordenação por troca. Entretanto, enquanto o bubblesort (bolha) troca pares de elementos consecutivos, o quicksort compara pares de elementos distantes, o que acelera o processo de ordenação.

A versão original do método foi apresentada por C. A. R. Hoare, em 1960.

É um algoritmo de troca do tipo divisão e conquista que resolve um determinado problema dividindo-o em três subproblemas, tais que: o primeiro subproblema é composto dos elementos menores ou iguais ao pivô, o segundo subproblema é o próprio pivô e o terceiro subproblema são os elementos maiores ou iguais ao pivô. A seguir, os problemas menores são ordenados independentemente e depois os resultados são combinados para produzir a solução do problema maior.

O primeiro passo do algoritmo é crucial, por enquanto não será especificado como escolher o pivô, pois o algoritmo funciona independentemente do elemento escolhido para ser o pivô. Entretanto, a seleção do pivô afeta diretamente o tempo de processamento do algoritmo.

O método consiste em alocar um determinado elemento em sua posição correta dentro da futura lista ordenada, produzindo uma partição da lista em três sub-listas:

a) a sub-lista que contém todos os elementos que são menores ou iguais ao elemento alocado;

b) a sub-lista formada somente pelo elemento alocado;

c) a sub-lista que contém todos os elementos que são maiores ou iguais ao elemento alocado.

A parte mais delicada deste método é relativa ao procedimento partição, o qual tem que rearranjar a lista A através da escolha arbitária de um item X da lista denominado pivô, de tal forma que ao final a lista A está particionada em uma parte esquerda com os elementos menores ou iguais a X e uma parte direita com elementos maiores ou iguais a X.

Este comportamento pode ser descrito pelo algoritmo:

a) escolha arbitrariamente um elemento da lista e coloca em X;

b) percorra a lista a partir da esquerda ate que um elemento $A[i] \geq X$ é encontrado; da mesma forma forma percorra a lista a partir da direita até que um elemento $A[j] \leq X$ é encontrado;

c) como os dois elementos A[i] e A[j] estão fora do lugar na lista final, então troque-os de lugar;

d) continue este processo até que os apontadores i e j se cruzem em algum ponto da lista.

Ao final a lista A está particionada de tal forma que:

a) Os elementos em A[esq], A[esq + 1], ..., A[j] são menores ou iguais a X;

b) Os elementos em A[i], A[i + 1], ..., A[dir] são maiores ou iguais a X.

4.4.1 - Algoritmo Partição

O objetivo é particionar a lista em três sublistas, alocando o elemento separador e determinando a sua posição de alocação, definindo segundo o campo chave.

Entradas: p (inteiro–inicio da sublista), u (inteiro-final da sublista), x (lista de elementos)

Saídas: j (inteiro-posição do elemento particionado), x (lista de elementos)

```
Particao(p,u,x)
  1   i ← p;
  2   j ← u + 1;
  3   Enquanto i < j faça
  4        repita i ← i + 1 até que x[i] ≥ x[p];
  5        repita j ← j – 1 até que x[j] ≤ x[p];
  6        se i < j
  7             então aux ← x[i]
  8                   x[i] ← x[j]
  9                   x[j] ← aux;
  10  fim-enquanto;
  11  aux ← x[p];
  12  x[p] ← x[j];
  13  x[j] ← aux;
  14  retorna(j);
```

4.4.2 - Versão recursiva do Quicksort

Após a primeira partição, são particionadas as sub-listas resultantes e as que delas resultam, e assim sucessivamente até que degenerem em listas vazias ou unitárias. Isto sugere que o algoritmo pode ser implementado recursivamente. Observe que se p ≥ u, a sub-lista correspondente é unitária ou vazia. O objetivo do algoritmo é ordenar uma lista, segundo o campo ch pelo método quicksort.

Entradas: p, u (inteiro), x (lista de elementos)

Saída: x (lista de elementos)

```
Quicksort (p, u, x)
  1   se p < u
  2   então     j ← particao (p, u, x);
  3             quicksort (p, j-1, x);
  4             quicksort (j+1, u, x);
```

4.4.3 - Outra versão recursiva do Quicksort

Outra versão do algoritmo, mas utilizando os mesmos conceitos apresentados acima.

entrada: x (vetor ou lista com os elementos), n (inteiro-tamanho)

saída: x (vetor ou lista com os elementos)

Quicksort (x, n);

Entradas: esq (inteiro–inicio da sublista), dir (inteiro-final da sublista),

Saídas: i, j (inteiros-posição final da partição),

```
Particao (esq, dir, i)
1 i ← esq;
2 j ← dir;
3 ele ← x[ (i + j) div 2 ]; { que é a chave}
4 repita
5     enquanto ele > x[i] faca
6          i ← i + 1;
7     fim-enquanto;
8     enquanto ele < x[j] faca
9          j ← j - 1;
10    fim-enquanto;
11    se i ≤ j
12         então aux ← x[i]
13               x[i] ← x[j]
14               x[j] ← aux;
15               i ← i + 1;
16               j ← j - 1;
enquanto i > j;
retorna(j);

Ordena (esq, dir);
   1  J ← particao (esq, dir, i);   (chamada do Procedimento}
   2  se esq < j
   3    entao ordena(esq, j);       (chamada recursiva}
   4  se i < dir
   5    entao ordena(i, dir);

{chamada do procedimento ordena}
ordena(1,n).
```

4.4.4 - Versão iterativa do Quicksort

Na versão recursiva do quicksort cada vez que o procedimento é chamado, uma nova cópia dos parâmetros é armazenada na memória, o que implica numa utilização maior da memória e indiretamente à redução do tamanho máximo da lista que pode ser processada.

Para otimizar o uso da memória utiliza-se a versão iterativa do quicksort, que consiste em: após cada partição, sistematicamente a maior sub-lista é colocada numa pilha e a menor é particionada imediatamente. Este procedimento continua até que resultem listas vazias ou unitárias, prosseguindo-se com as sub-listas guardadas na pilha. A ordenação estará concluída quando resultarem sub-listas vazias ou unitárias e a pilha vazia.

Após uma partição não é necessário colocar todos os elementos da sub-lista maior na pilha, mas apenas os índices de primeiro e último elemento. Portanto, após a partição da sub-lista: $\{X_p, X_{p+1}, ..., X_u\}$ obtemos as novas sub-listas:

$\{X_p, ..., X_{j-1}\}$, $\{X_j\}$, $\{X_{j+1}, ..., X_u\}$ as quais passam a ser representadas pelos pares ordenados: (p, j-1) e (j+1, u) respectivamente.

A estratégia abaixo produz a escolha da menor sub-lista e o armazenamento da maior sub-lista:

- Se as duas sub-listas não são unitárias e nem vazias, colocamos a maior na pilha e escolhemos a menor;
- Se as duas são vazias ou unitárias, escolhemos a que está no topo da pilha; se a pilha está vazia então o processo terminou;
- Se apenas uma delas não é vazia e nem unitária, ela será a escolhida.
- Para definir o tamanho da sub-lista, usamos:

Sub-lista	Tamanho	Sub-lista não vazia e nem unitária
(p, j-1)	j - p	p < j-1
(j+1, u)	u – j	j+1 < u

O Algoritmo Escolhe

O objetivo é que após a partição da sublista $\{X_p, ..., X_u\}$ nas sub-listas

$\{X_p, ..., X_{j-1}\}$, $\{X_j\}$, $\{X_{j+1}, ..., X_u\}$ é de escolher a nova sublista a ser particionada, isto é, redefinir os valores dos índices p e u. Se não existir sub-lista a ser particionada o valor de p é igual a u. Inicialmente a pilha está vazia e topo é igual a zero.

Entradas: p, u, j (inteiros)

Saídas: p, u (inteiros)

```
Escolhe (p, u, j)
  1   Se (p < j-1) e (j-1 < u)
  2         então Se (j - p) ≥ (u - j)
  3               então topo ← topo + 1;
  4                     s[topo, 1] ← p;
  5                     s[topo, 2] ← j-1;
  6               senão p ← j + 1;
  7                     topo ← topo + 1;
  8                     s[topo, 1] ← j+1;
  9                     s[topo, 2] ← j-1;
  10  senão Se (p ≥ j-1) e (j-1 ≥ u)
  11              então Se topo > o
  12                          então p ← s[topo, 1];
  13                                u ← s[topo, 2];
  14                                topo ← topo - 1
  15                          senão p ← u
  16              senão Se (p < j-1)
  17                          então u ← j-1
  18                          senão p ← j+1
```

O algoritmo Quicksort iterativo

O objetivo é ordenar a lista segundo o campo chave pelo método quicksort.

Entradas: n (inteiro - tamanho), x (lista de elementos)

Saída: x (lista de elementos)

algoritmos auxiliares: particao(p, u, x, j) e escolhe(p, u, j)

```
Quicksort(n, x)
  1   p ← 1;
  2   u ← n;
  3   topo ← o;
  4   enquanto p < u faça
  5        j ← particao(p, u, x);
  6        escolhe(p, u, j);
  7   fim-enquanto.
```

Análise de Complexidade

Vamos analisar a complexidade em relação à operação de comparação. O movimento de registros é sempre inferior ao número de comparações.

Qualquer que seja a entrada, após a primeira partição são efetuadas no máximo (n+1) comparações, pois o elemento é alocado quando i torna-se maior ou igual a j. Portanto,

$C(n) \leq (n + 1) + C(n1) + C(n2)$, com $n1 + n2 = n - 1$

a) Pior caso

O pior caso ocorre para as entradas que correspondem às listas ordenadas ou invertidas, em que sistematicamente, após cada partição resulta uma sublista vazia.

Após cada varredura $n_1 = 0$ ou $n_2 = 0$. Daí pode-se escrever que:

$C(n) \leq n + 1 + C(n - 1)$

$C(n - 1) \leq n + C(n - 2)$

$C(n - 2) \leq n - 1 + C(n - 3)$

. . .

$C(2) \leq 3 + C(1)$

Como $C(1) = 0$, por substituições sucessivas obtemos:

$C(n) \leq (n + 1) + n + (n - 1) + ... + 3 = {}^{1}/_{2} (n - 1) (n + 4)$

Portanto $wC(n) \leq ½ (n - 1) (n + 4) = O(n_2)$

Conforme observaremos no estudo da complexidade do caso médio, temos a complexidade $C(n) = O(n \log n)$. Por isso as entradas que dão origem ao pior caso devem ser evitadas. . Uma estratégia muito utilizada para eliminar o pior caso e eliminar o uso de uma sentinela especial consiste no procedimento:

Considerando três elementos da sub-lista, que são o primeiro, o último e o elemento de posição k = (p + u) div 2. Comparando os elementos entre si e alocamos o maior deles no fim da sub-lista e o segundo maior no início da sub-lista. O algoritmo a seguir implementa a idéia:

```
Prepara(p, u, x)
1   K ← (p + u) div 2;
2   Se X[p] > X[u]
3        Então aux ← X[p];
4              X[p] ← X[u];
5              X[u] ← aux;
6   Se X[k] > X[u]
7        Então aux ← X[k];
8              X[k] ← X[u];
9              X[u] ← aux;
10  Se X[p] < X[k]
11       Então aux ← X[k];
12             X[k] ← X[p];
13             X[p] ← aux;
```

b) Caso Médio

Conforme vimos, qualquer que seja a entrada, temos;

$C(n) \leq (n + 1) + C(n1) + C(n2)$, com $n_1 + n_2 = n - 1$

Como n1 varia desde 0 até (n-1), existem n entradas distintas e

$C(n) \leq n + 1 + C(0) + C(n - 1)$

$C(n) \leq n + 1 + C(1) + C(n - 2)$

. . .

$C(n) \leq n + 1 + C(n - 1) + C(0)$

Expressando C(n) em função de todas as entradas possíveis.

Somando todas as desigualdades e dividindo por n obtemos:

$$C(n) \leq n + 1 + {}^{2}/_{m} \sum_{i=0}^{n-1} C(i)$$

Multiplicando a primeira por n, a segunda por (n-1) e subtraindo a primeira da segunda, conseguimos expressar C(n) em função de C(n-1):

$$nC(n) - (n - 1)C(n-1) \leq n(n + 1) - (n - 1)n + 2C(n-1)$$
$$nC(n) \leq 2n + (n + 1)C(n-1)$$
$$({}^{C(n)}/_{(n+1)}) \leq ({}^{2}/_{(n+1)}) + ({}^{C(n-1)}/_{n})$$

Desta última desigualdade podemos escrever a seqüência:

$$({}^{C(n)}/_{(n+1)}) \leq ({}^{2}/_{(n+1)}) + ({}^{C(n-1)}/_{n})$$
$$({}^{C(n-1)}/_{n}) \leq ({}^{2}/_{n}) + ({}^{C(n-2)}/_{(n-1)})$$
$$({}^{C(n-2)}/_{(n-1)}) \leq ({}^{2}/_{(n-1)}) + ({}^{C(n-3)}/_{(n-2)})$$
$$\vdots$$
$${}^{C(2)}/_{3} \leq {}^{2}/_{3} + {}^{C(1)}/_{2}$$

Levando em conta que C(1) = 0, por substituições sucessivas obtemos:

$$C(n)/(n+1) \leq 2/(n+1) + 2/n + \ldots + 2/3 = 2\sum_{i=3}^{n+1} 1/i \leq 2\int_{1}^{n+1} dx/x$$

$$C(n)/(n+1) \leq 2 \ln (n+1), \text{ isto é } C(n) \leq (n+1)\log(n+1)$$

Portanto, **C(n) ≈ O(n log n)**

4.4.5 - Conclusão

O quicksort é extremamente eficiente para ordenar arquivos de dados. O método necessita de apenas uma pequena pilha como memória auxiliar, e requer cerca de (n log n) operações em média para ordenar n itens. Como aspectos negativos tem-se:

- A versão recursiva do algoritmo tem um pior caso que é O(n2);
- A implementação do algoritmo é muito delicada e difícil;
- O método não é estável.

Uma vez desenvolvida uma implementação robusta para o quicksort, este deve ser o algoritmo preferido para a maioria das aplicações.

4.5 - Método de ordenação Shellsort

É um método de ordenação cujo princípio de funcionamento é o mesmo utilizado para a ordenação por inserção. O método de inserção troca itens adjacentes quando está procurando o ponto de inserção na sequência destino. Se o menor item estiver na posição mais à direita no vetor então o número de comparações e movimentações é igual a (n-1) para encontrar o seu ponto de inserção.

O método shellsort contorna este problema permitindo trocas de registros que estão distantes um do outro. Os itens que estão separados h posições são rearranjados de tal forma que todo h-ésimo item leva a uma sequência ordenada. Tal sequência é dita estar h-ordenada. A tabela abaixo mostra a sequência de ordenação usando os incrementos: 4, 2 e 1 como valores de h.

	1	2	3	4	5	6
Chave inicial	O	R	D	E	N	A
h = 4	N	A	D	E	O	R
h = 2	D	A	N	E	O	R
h = 1	A	D	E	N	O	R

Na primeira passada (h=4), a letra O é comparada com a letra N (posições 1 e 5) e trocados, a seguir o R é comparado com o A (posições 2 e 6) e trocados. Na segunda passada (h=2), as letras N, D e O nas posições 1, 3 e 5 são rearranjadas para resultar em D, N e O nestas mesmas posições; da mesma forma as letras A, E e R nas posições 2, 4 e 6 são comparados e mantidos nos seus lugares. A última passada (h=1) corresponde ao algoritmo de inserção, entretanto nenhum item tem que se mover para posições muito distantes.

Várias sequências para h tem sido experimentadas, Knuth mostrou experimentalmente que a escolha do incremento para h descrita abaixo é difícil de ser superada por mais de 20% em eficiência no tempo de execução.

$h(s) = 3h(s - 1) + 1$, para $s > 1$

$h(s) = 1$, para $s = 1$

Os valores de h correspondem a sequência: 1, 4, 13, 40, 121, . . .

4.5.1 – Algoritmo Shellsort

```
Shellsort (A)
  1   h ← 1;
  2   Repita h de 3 * h + 1 ate que h > n
  3        Repita
  4             h ← h div 3;
  5             para i de h + 1 ate n faca
  6                  x ← A[i];
  7                  j ← i;
  8                  Enquanto A[j-h] > x faca
  9                       A[j] ← A[j-h];
 10                       j ← j - h;
 11                       Se j ≤ h
 12                         Então va para 999;
 13                  fim-enquanto;
 14                  999: A[j] ← x;
 15             fim-para;
 16        até h = 1;
 17  fim-repita.
```

4.5.2 - Análise do algoritmo

A razão pela qual este método é eficiente ainda não foi determinada, porque é difícil analisar o algoritmo. A sua análise contém alguns problemas matemáticos muito difíceis, a começar pela própria sequência de incrementos, mas cada incremento não deve ser múltiplo do incremento anterior. Para a sequência de incrementos utilizadas no algoritmo apresentado existem duas conjecturas para o número de comparações:

a) $C(n) = \mathbf{O(n^{1,25})}$

b) $C(n) = \mathbf{O(n\ (\ln n)^2)}$

Para n = 10000, os valores de *h* correspondem a sequência: 1, 4, 13, 40, 121, 364, 1093 e 3280. Os dados experimentais foram aproximados por função exponencial, por estimativa, $(1{,}21n^{5/4})$ e pela função logarítmica $(0{,}3n \ln^2 n - 2{,}33n \ln n)$ que é equivalente a $O(n \ln^2 n)$. A primeira forma se ajusta melhor aos resultados dos testes, pois $(1{,}21n^{1,25})$ que equivale a $O(n^{1,2})$ é muito melhor do que $O(n^2)$ para a ordenação por inserção, mas ainda é maior do que o desempenho esperado que é O(n log n).

Shellsort é uma ótima opção para arquivos de tamanho moderado (da ordem de 5000 registros), mesmo porque sua implementação é simples e requer um conjunto de códigos pequeno. O tempo de execução do algoritmo é sensível à ordem inicial do arquivo, além do que o método não é estável, pois ele nem sempre deixa os registros com chaves iguais na mesma posição relativa.

4.6 - Método de ordenação Heapsort

É um método de ordenação desenvolvido em 1964 por John W. J. Williams para melhorar o desempenho do método de ordenação por seleção, que é ineficiente ($O(n^2)$ comparações) mas que tem como vantagem fazer poucas movimentações. O princípio de funcionamento é o mesmo utilizado para a ordenação por seleção:

- Selecione o menor item do vetor e a seguir troque-o com o item que está na primeira posição do vetor;
- Repita estas duas operações com os (n-1) itens restantes, depois com os (n-2) itens e assim sucessivamente.

O custo para encontrar o menor (ou o maior) item entre os n itens custa (n-1) comparações. Este custo pode ser reduzido através da utilização de uma estrutura de dados denominada fila de prioridades.

4.6.1 - Fila de prioridades

Em várias situações é necessário ter uma estrutura de dados que suporte as operações de inserir um novo item e retirar o item com a maior chave.

Filas com prioridades são utilizadas em um grande número de aplicações. Sistemas operacionais usam filas de prioridades, onde as chaves representam o tempo em que os eventos devem ocorrer. Alguns métodos numéricos interativos são baseados na seleção repetida de um item com maior (ou menor) valor.

As operações mais comuns são: Adicionar um novo item ao conjunto e extrair o item do conjunto que contenha o maior (ou o menor) valor.

Um tipo abstrato de dados fila de prioridades, contendo registros com chaves numéricas (prioridades), deve suportar as operações:

- Constrói uma fila de prioridades a partir de um conjunto com n itens;
- Insere um novo item;
- Retira o item com maior chave;
- Substitui o maior item por um novo item, a não ser que o novo item seja maior;
- Altera a prioridade de um item;
- Remove um item qualquer;
- Ajunta duas filas de prioridades em uma única fila.

A operação **constrói** é equivalente ao uso repetido da operação **insere**, e a operação **altera** é equivalente à operação **remove** seguida da operação **insere**.

Uma representação para a fila de prioridades é uma lista linear ordenada. Neste caso, a operação constrói leva o tempo O(n log n), insere é O(n) e retira é O(1). Outra representação é através de uma lista linear não ordenada, na qual a operação constrói tem custo linear e é de O(1), retira é O(n) e ajunta é O(1) para implementações através de apontadores e O(n) para implementações através de arranjos, onde **n** representa o tamanho da menor fila de prioridades.

Filas de prioridades podem ser representadas por estruturas de dados chamadas de **Heaps**. A operação constrói tem custo linear, e insere, retira, substitui e altera tem custo logarítmico. Para implementar a operação ajunta de forma eficiente e ainda preservar um custo logarítmico para as operações insere, retira, substitui e altera é necessário utilizar estruturas de dados mais sofisticadas, tais como árvores binomiais.

Qualquer algoritmo para filas de prioridades pode ser transformado em um algoritmo de ordenação, através do uso repetido da operação insere para construir a fila, seguido do uso repetido da operação retira para receber os itens na ordem inversa. Deste modo, o uso de listas lineares não ordenadas corresponde ao método da seleção, o uso de listas lineares ordenadas corresponde ao método da inserção e o uso de heaps corresponde ao método heapsort.

4.6.2 - Heaps

Um heap é definido como uma seqüência de itens como chaves:

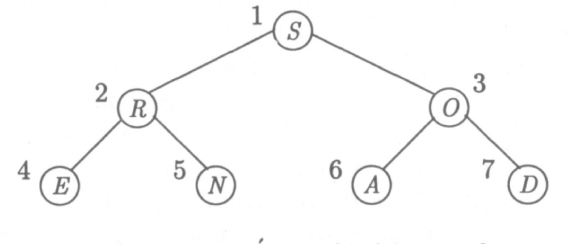
C[1], C[2], . . ., C[n]

Tal que

$C[i] \geq C[2i]$

$C[I] \geq C[2i+1]$, para todo I = 1, 2, …, $^{n}/_{2}$

Esta ordem é visualizada se a seqüência de chaves for desenhada em uma **árvore binária**, onde as linhas que saem de uma chave levam a duas chaves menores no nível inferior.



1	2	3	4	5	6	7
S	R	O	E	N	A	D

Esta estrutura é conhecida como uma árvore binária completa: o primeiro nodo é chamado **raiz**, os nodos abaixo de cada nodo são chamados nodos **filhos** e o nodo acima é chamado de nodo **pai**. Uma árvore binária completa pode ser representada por um vetor. Esta representação é extremamente compacta e, permite caminhar pelos nodos da árvore facilmente, onde os filhos de um nodo **i** estão na posição **2i** e **2i+1**, caso existam, e o pai de um nodo **i** está na posição (**i div 2**).

4.6.3 - Heapsort

Um método elegante e que não necessita de memória auxiliar foi apresentado por Floyd (em 1964). dado um vetor A[1], A[2], ..., A[n], os itens $A[^{n}/_{2}+1]$, $A[^{n}/_{2}+2]$, ..., A[n] formam um heap, porque neste intervalo do vetor não existem dois índices **i** e **j** tais que j=2i ou j=(2i+1). A construção do heap:

a) dada as chaves iniciais:

1	2	3	4	5	6	7
O	R	D	E	N	A	S

b) estendendo o heap para a esquerda (Esq=3), englobando o item A[3], pai dos itens A[6] e A[7], aqui a condição do heap é violada e os itens A[3] e A[7] são trocados;

Esq = 3

1	2	3	4	5	6	7
O	R	S	E	N	A	D

c) a seguir o heap é estendido novamente a esquerda (Esq=2) incluindo o item A[2], pai dos itens A[4] e A[5], que atendem a condição do heap.

Esq = 2

1	2	3	4	5	6	7
O	R	S	E	N	A	D

d) finalmente, estendendo o heap a esquerda (Esq= 1), englobando o item A[1], pai dos itens A[2] e A[3], a condição é violada e os itens A[1] e A[3] são trocados, encerrando o processo.

Esq = 1

1	2	3	4	5	6	7
S	R	O	E	N	A	D

Entradas: Esq, Dir (inteiro), A (vetor de elementos)
Saída: A (vetor de elementos)

```
refaz(esq, dir, A)
1   i ← esq;
2   j ← 2 * i;
3   x ← A[i];
4   Enquanto j ≤ dir faca
5        Se j < dir
6          Então Se A[j] < A[j+1]
7                  Entao j ← j + 1;
8        Se x ≥ A[j]
9          Então vá para 999;
10       A[i] ← A[j];
11       i ← j;
12       j ← 2 * i;
13  fim-enquanto;
14  999: A[i] ← x;
```

Entrada e Saída: A (vetor de elementos)

```
Constroi(A)
1   Esq ← (n div 2) + 1;
2   Enquanto esq > 1 faca
3      Esq ← esq - 1;
4      Refaz (esq, n, A);
5   fim-enquanto;
```

Usando o heap obtido pelo procedimento **constrói**, pega-se o elemento na posição **1** do vetor (raiz do heap) e troca com o item que está na posição **n** do vetor.Agora usando o procedimento refaz para reorganizar o heap para os itens A[1], A[2], ..., A[n-1]. Repete-se as duas operações sucessivamente até que reste apenas um item.

Entrada e Saída: A (vetor de elementos)

```
Heapsort (A)
1     Esq ← (n div 2) + 1;
2     Dir ← n;
3     Enquanto esq > 1 faca
4          Esq ← esq – 1;
5          Refaz (esq, dir, A);
6     fim-enquanto;
7     Enquanto dir > 1 faca
8          x ← A[1];
9          A[1] ← A[dir];
10         A[dir] ← x;
11         Dir ← dir – 1;
12         Refaz (esq, dir, A);
13    fim-enquanto;
```

4.6.4 - Análise de complexidade do Heapsort

a) Pior caso

Analisando cada parte do algoritmo:

i) algoritmo refaz

se a árvore binária com **n** nós possui altura **k**, k é o menor inteiro que satisfaz

$n \leq 2^{k+1} - 1$ e $2^k \leq n \leq 2^{k+1} - 1$, logo **k = log(n)**

o procedimento atua entre os níveis o e (k-1) da subárvore, efetuando, em cada nível, no máximo duas comparações e uma troca. Portanto,

$= wC(n) = 2k \leq 2 \log n \Rightarrow wC(n) = O(\log n)$

$wT(n) = k \leq \log n \Rightarrow wM(n) = 3wT(n) \leq 3 \log n \Rightarrow w\,M(n) =$ **O(log n)**

ii) algoritmo constrói

o algoritmo executa o algoritmo refaz (n div 2) vezes, portanto,

$wC(n) \leq (^n/_2)\ 2 \log n = n \log n \Rightarrow O(n \log n)$

$wT(n) \leq (^n/_2) \log n \Rightarrow wM(n) = 3wT(n) \leq {}^{3n}/_2 \log n =$ **O(n log n)**

iii) algoritmo heapsort

a ordenação da estrutura heap requer a execução do refaz (n-1) vezes, portanto,

$wC(n) \leq (n-1)\ 2 \log n \Rightarrow wC(n) = O(n \log n)$

$wT(n) \leq (n-1) \log n \Rightarrow w\,M(n) = 3\,w\,T(n) \leq 3\,(n-1) \log n =$ **O(n log n)**

Portanto, no pior caso: **wM(n) = O(n log n)**

b) Caso Médio

Como nenhum algoritmo de ordenação pode ser inferior a O(n log n) e wC(n) e wM(n) são O(n log n), decorre que **aC(n) = O(n log n)** e **aM(n) = O(n log n)**

4.6.5 - Conclusão

Inicialmente o algoritmo não parece eficiente, pois as chaves são movimentadas várias vezes. Entretanto, o procedimento refaz gasta cerca de (log n) operações no pior caso. Portanto, heapsort gasta um tempo de execução proporcional a (n log n) no pior caso.

Heapsort não é recomendado para arquivos com poucos registros, por causa do tempo necessário para construir o heap, bem como porque o anel interno do algoritmo é bastante complexo, se comparado com o anel interno do quicksort.

O quicksort é, em média, duas vezes mais rápido que o heapsort. Entretanto, o heapsort é melhor que o shellsort para grandes arquivos.

Deve-se observar que o comportamento do heapsort é **O(n log n)** qualquer que seja a entrada. Aplicações que não podem tolerar um caso desfavorável devem usar o heapsort.

Um aspecto negativo é que o método não é estável.

4.7 - Comparação entre os métodos de ordenação

A ordenação interna é usada quando todos os registros do arquivo (ou lista ou vetor) cabem na memória principal.

Usando dois métodos simples (seleção e inserção) que requerem $O(n^2)$ comparações e três métodos eficientes (shellsort, quicksort e heapsort) que requerem O(n log n) comparações em seus casos médios para uma lista com n elementos.

As tabelas abaixo apresentam quadros comparativos do tempo para ordenar arranjos com 500, 5000, 10000 e 30000 registros na ordem aleatória (4, 2, 9, 1, ..., n), na ordem ascendente (1, 2, 3, ..., n), e na ordem descendente (n, n-1,..., 3, 2, 1). Em cada tabela, o método que gastou o menor tempo para executar o procedimento recebeu o valor 1 e os outros valores são relativos a este.

Ordem aleatória dos registros				
	500	5000	10000	30000
Inserção	11.3	87	161	-
Seleção	16.2	124	228	-
Shellsort	1.2	1.6	1.7	2
Quicksort	1	1	1	1
heapsort	1.5	1.6	1.6	1.6

Ordem ascendente dos registros				
	500	5000	10000	30000
Inserção	1	1	1	1
Seleção	128	1524	3066	-
Shellsort	3.9	6.8	7.3	8.1
Quicksort	4.1	6.3	6.8	7.1
heapsort	12.2	20.8	22.4	24.6

Ordem descendente dos registros				
	500	5000	10000	30000
Inserção	40.3	305	575	-
Seleção	29.3	221	417	-
Shellsort	1.5	1.5	1.6	1.6
Quicksort	1	1	1	1
heapsort	2.5	2.7	2.7	2.9

Analisando os quadros acima, pode-se apresentar os seguintes comentários:

a) Shellsort, Quicksort e o Heapsort possuem a mesma ordem de grandeza;

b) O Quicksort é o mais rápido para todos os tamanhos na situação de ordem aleatória;

c) A relação Heapsort / Quicksort se mantém constante para todos os tamanhos, sendo o Heapsort mais lento;

d) A relação Shellsort / Quicksort aumenta à medida que o número de elementos aumenta, para arquivos pequenos (próximos de 500 elementos) o Shellsort é mais rápido que o Heapsort, porém quando o tamanho da entrada cresce, a relação inverte;

e) A Inserção é o método mais rápido para qualquer tamanho se os elementos já estão ordenados, este é o seu melhor caso O(n). Pode também ser utilizado nos casos em que os elementos já estão quase ordenados. Mas é o mais lento para qualquer tamanho se os elementos estão em ordem decrescente, este é o seu pior caso $O(n^2)$;

f) Entre os métodos de custo $O(n^2)$, a Inserção é melhor para todos os tamanhos de ordenação aleatória experimentados;

g) A Inserção é o mais interessante para arquivos pequenos com até 50 elementos, podendo ser até mais eficiente que o método da Bolha (Bubblesort);

h) A Seleção somente é vantajoso quanto ao número de movimentos de registros, que é O(n). Logo deve ser usado para arquivos com registros grandes, desde que a quantidade máxima seja de até 1000 elementos;

i) Fazendo uma comparação entre os métodos mais eficientes, observando a influencia da ordem inicial:

	Shellsort			Quicksort			Heapsort		
Situação / Qtdd	5000	10000	30000	5000	10000	30000	5000	10000	30000
Ascendente	1.0	1.0	1.0	1.0	1.0	1.0	1.1	1.1	1.1
Descendente	1.5	1.6	1.5	1.1	1.1	1.1	1.0	1.0	1.0
Aleatória	2.9	3.1	3.7	1.9	2.0	2.0	1.1	1.0	1.0

No método Quicksort foi usada uma implementação que usa o recurso da sentinela.

j) O Shellsort é o método escolhido para a maioria das aplicações por ser muito eficiente para um conjunto de elementos de até 10000 elementos;

k) O Quicksort é o algoritmo mais eficiente que existe para uma grande variedade de situações. Entretanto, deve-se procurar uma implementação estável;

l) O heapsort é um método de ordenação elegante e eficiente. Por causa do anel interno complexo (que o torna duas vezes mais lento que o Quicksort) ele não necessita de memória adicional, e é sempre O(n log n) qualquer que seja a ordem inicial dos elementos. Este é o método a ser usado por aplicações que não podem tolerar variações no tempo esperado de ordenação;

m) Quando os registros do arquivo são muito grandes é desejável que o método de ordenação realize o menor número de movimentação com os registros, podendo ser usada uma ordenação indireta.

4.8 - Exercícios

1) Qual a ideia central do Quicksort?
2) Descreva a utilidade do procedimento Partição do método Quicksort?
3) Faça um comentário: Para otimizar a utilização da memória, você usaria o método iterativo ou o recursivo do Quicksort. Explique.
4) Em qual situação ocorre o pior caso no método Quicksort? Qual a estratégia para evitar o pior caso?
5) Qual a complexidade do pior caso do Quicksort? E do caso médio?
6) Qual a ideia central do Heapsort?
7) Comente a utilidade das Filas de Prioridade no método Heapsort.
8) Qual a complexidade do Pior caso do Heapsort? E do caso Médio? Explique o que ocorre em função das respostas anteriores.
9) Qual a ideia central do Shellsort?
10) Comente a diferença entre o Shellsort e o método de Inserção.
11) Qual a complexidade do Pior caso do Shellsort? E do caso Médio? Explique o que ocorre em função das respostas anteriores.
12) Qual a ideia central do método de Seleção?
13) Qual a complexidade do Pior caso do método de Seleção? E do caso Médio? E do Melhor caso? Explique o que ocorre em função das respostas anteriores.
14) Qual a ideia central do método de Inserção?
15) Qual a complexidade do Pior caso do método de Inserção? E do caso Médio? E do Melhor caso? Explique o que ocorre em função das respostas anteriores.
16) Observando as tabelas da página 46, comente:
    a) Se há um conjunto com 300 elementos para ordenar, e não sabemos qual a sua situação atual, qual dos métodos você usaria? Justifique.
    b) Se há um conjunto com 3000 elementos para ordenar, e não sabemos qual a sua situação atual, qual dos métodos você usaria? Justifique.
    c) Se há um conjunto com 15000 elementos para ordenar, e não sabemos qual a sua situação atual, qual dos métodos você usaria? Justifique.
    d) Se há um conjunto com aproximadamente 1000 elementos para ordenar, e sabemos que a sua situação atual é que ele está com aproximadamente 95% de seus elementos na posição definitiva (ou seja já ordenados), qual dos métodos você usaria? Justifique.
    e) Se há um conjunto com 15000 elementos para ordenar, e sabemos que a sua situação atual é uma ordenação descendente, qual dos métodos você usaria? Justifique.
    f) Se fosse usado o método da inserção para ordenar um conjunto com 30000 elementos em que a sua situação atual é ascendente, o que ocorreria? Justifique.
    g) Se fosse usado o método da inserção para ordenar um conjunto com 30000 elementos em que a sua situação atual é descendente, o que ocorreria? Justifique.

17) Desenvolvemos um grupo de programas baseados em um conjunto de algoritmos de ordenação de listas, e ao executá-los para algumas situações iniciais, foram determinados índices relativos para o tempo gasto, onde para cada situação inicial o mais rápido recebeu o valor um e os outros proporcionalmente a este. Baseando-se nos dados da tabela fornecida e nos conceitos estudados, responda as perguntas abaixo.

Tempos proporcionais para ordenação de 5000 elementos			
Algoritmo	Aleatória	Ascendente	Descendente
Inserção	87	1	305
Seleção	124	1524	221
Shellsort	1.6	6.8	1.5
Quicksort	1	6.3	1
Heapsort	1.6	20.8	2.7

a) Se tivesse de escolher um algoritmo para ordenar uma lista de aproximadamente 6000 elementos com a situação de ordenação inicial indefinida, qual método você usaria? Justifique a resposta.

b) Se tivesse de escolher um algoritmo para ordenar uma lista de aproximadamente 4000 elementos com a situação de ordenação inicial ascendente e com possibilidades de continuar crescendo, qual método você usaria? Justifique a resposta.

c) Se tivesse de escolher um algoritmo para ordenar uma lista de aproximadamente 6000 elementos com a situação de ordenação inicial indefinida, e em que o algoritmo implementado do Quicksort não é confiável, qual método você usaria? Justifique a resposta.

d) Supondo que o tempo gasto pelo programa Quicksort na situação inicial aleatória foi de 100 ms, qual o tempo de execução aproximado dos outros métodos na situação aleatória.

18) Observando as tabelas da pagina 46 da apostila, responda. Se você tiver que escolher um método de ordenação interna eficiente, para ser usado na ordenação de uma lista com ordenação inicial indefinida, qual o método você usaria? Justifique.

19) Desenvolvemos um grupo de programas baseados em um conjunto de algoritmos de ordenação de listas, e ao executá-los para algumas situações iniciais, foram determinados o tempo gasto em milissegundos. Baseando-se nos dados da tabela fornecida abaixo e nos conceitos estudados, responda as perguntas formuladas abaixo.

Ordenação de uma lista de 850 elementos (em ms)					
	Seleção	Inserção	ShellSort	QuickSort	HeapSort
Aleatória	506	571	500	402	483
Ascendente	700	544	840	809	951
Descendente	711	621	604	585	666

a) Se tivesse de escolher um algoritmo para ordenar uma lista de aproximadamente 1000 elementos com a situação de ordenação inicial indefinida, qual método você usaria? Justifique a resposta.

b) Se tivesse de escolher um algoritmo para ordenar uma lista de aproximadamente 1000 elementos com a situação de ordenação inicial descendente, qual método você usaria? Justifique a resposta.

c) Se tivesse de escolher um algoritmo para ordenar uma lista de aproximadamente 1000 elementos com a situação de ordenação inicial ascendente e com possibilidades de continuar crescendo, qual método você usaria? Justifique a resposta.

d) Se tivesse de escolher um algoritmo para ordenar uma lista de aproximadamente 1500 elementos com a situação de ordenação inicial indefinida, e em que o algoritmo implementado do Quicksort não é confiável, qual método você usaria? Justifique a resposta.

e) Porque os tempos de execução dos métodos não tem a mesma proporcionalidade apresentada na tabela 1 da página 49?

f) Comparando com as tabelas da pagina 46 da apostila de ordenação, o que podemos concluir? Faca uma pesquisa e escreva uma teoria que possa explicar tais diferenças.

20) Observe a tabela abaixo, O que você pode concluir?

Shellsort			
Ordenação\ quantidade	500	10000	30000
Ascendente	3.3	4.6	5.1
Decrescente	1.3	1	1
Aleatório	1	1.1	1.3

Para a melhor situação foi dado o valor um, e aos outros resultados os valores são proporcionais ao melhor caso.

4.9 – Exercício prático de Ordenação

4.9.1 – Exercício prático com os algoritmos "Inserção" e "seleção"

Desenvolver os algoritmos "Inserção" e "seleção" em linguagem C.

Executar os programas com listas contendo 100, 200 e 400 elementos na seguinte situação inicial:
a) Lista inicial aleatória;
b) Lista inicial ascendente;
c) Lista inicial descendente.

Os valores devem ser atribuídos no próprio programa fonte ou lidos uma única vez no início da execução. Calcular o tempo gasto em cada execução.

Colocar um contador para calcular o número de comparações executadas. Ao final de cada execução imprimir o contador. Ao final das execuções, fazer uma tabela e comparar os resultados encontrados.

No lugar do contador, colocar um delay. Calcular o tempo gasto. Ao final das execuções, fazer uma tabela e comparar os resultados encontrados.

Fazer um documento texto analisando os dados encontrados. Deve apresentar as tabelas com os dados encontrados e os valores esperados, quando possível. O que pode-se concluir observando os dados encontrados nos itens anteriores e a teoria apresentada? Apresente uma comparação com a complexidade apresentada.

A entrega do trabalho é em dois arquivos digitais, onde o primeiro deve conter os programas-fonte e segundo o arquivo texto (no word 2003 ou anterior).

No programa-fonte deve conter o(s) nome(s) do(s) aluno(s) responsável(ies), o que deve ser feito quando mais de um aluno desenvolveu o programa, sendo aceito no máximo três (3) alunos. O programa fonte deve apresentar também comentários explicando as principais passagens desenvolvidas.

No arquivo texto deve estar os dados coletados para cada situação proposta e para cada uma deve ter o número de comparações e tempo gasto, e as conclusões do aluno. Este arquivo deve conter o nome do aluno, e a informação (quando necessária) de que o programa foi desenvolvido juntamente com outros alunos (colocar os nomes) e em seguida os dados coletados e as conclusões. Observe que este arquivo é individual e as execuções para coleta dos dados também devem ser individuais.

4.9.2 – Exercício prático com os algoritmos de ordenação

Desenvolver os algoritmos “Inserção”, “seleção”, “Shellsort”, “Quicksort” e “Heapsort” em linguagem C. Usar um registro com três campos: Código (numérico), Nome (string), Endereço (string). A ordenação deve ser pelo código. Os outros dados podem ser preenchidos aleatoriamente.

Executar os programas com listas contendo 100, 200 e 400 elementos na seguinte situação inicial:

a) Lista inicial aleatória;
b) Lista inicial ascendente;
c) Lista inicial descendente.

Os valores devem ser atribuídos no próprio programa fonte ou lidos uma única vez no início da execução. Calcular o tempo gasto em cada execução.

Colocar um contador para calcular o número de comparações executadas. Ao final de cada execução imprimir o contador. Ao final das execuções, fazer uma tabela e comparar os resultados encontrados.

No lugar do contador, colocar um delay. Calcular o tempo gasto. Ao final das execuções, fazer uma tabela e comparar os resultados encontrados.

Fazer um documento texto analisando os dados encontrados. Deve apresentar as tabelas com os dados encontrados e os valores esperados, quando possível. O que pode-se concluir observando os dados encontrados nos itens anteriores e a teoria apresentada? Apresente uma comparação com a complexidade apresentada. Faça uma comparação com os valores encontrados nas tabelas da página 46, são proporcionais? e explique a resposta.

A entrega do trabalho é em dois arquivos digitais, onde o primeiro deve conter os programas-fonte e segundo arquivo texto (no word 2003 ou anterior).

No programa-fonte deve conter o(s) nome(s) do(s) aluno(s) responsável(ies), o que deve ser feito quando mais de um aluno desenvolveu o programa, sendo aceito no máximo três (3) alunos. O programa fonte deve apresentar também comentários explicando as principais passagens desenvolvidas.

No arquivo texto deve estar os dados coletados para cada situação proposta e para cada uma deve ter o número de comparações e tempo gasto, e as conclusões do aluno. Este arquivo deve conter o nome do aluno, e a informação (quando necessária) de que o programa foi desenvolvido juntamente com outros alunos (colocar os nomes) e em seguida os dados coletados e as conclusões. Observe que este arquivo é individual e as execuções para coleta dos dados também devem ser individuais.

BIBLIOGRAFIA:

KNUTH, D. E. *The Art of Computer Programming.* Massachusetts: Addison-Wesley Longman, 1997. v. 1 e 2.

SALVETTI, D. D. & BARBOSA L M. *Algoritmos.* São Paulo: Makron Books, 1998.

TERADA, R. *Desenvolvimento de Algoritmo e Estruturas de Dados*. São Paulo: Makron Books, 1991.

ZIVIANI, N. *Projeto de Algoritmos - Com Implementações em PASCAL* e C. São Paulo: Editora Pioneira, 1999.

Nota do Professor:

Este trabalho é um resumo do conteúdo da disciplina, para facilitar o desenvolvimento das aulas, devendo sempre ser complementado com estudos nos livros recomendados e o desenvolvimento dos exercícios indicados em sala de aula e a resolução das listas de exercícios propostas.