MÉTODO DE ORDENAÇÃO DE DADOS RADIXSORT

Silva, Mauro Rafael R. [1]; CHICON, Patricia Mariotto Mozzaquatro[2]; TELOCKEN, Alex Vinícius[3];

Resumo: Este artigo tem o objetivo de apresentar um estudo sobre o método de ordenação de dados *Radixsort* . A pesquisa desenvolvida classifica-se como Aplicada, sendo de abordagem quantitativa. Nela apresenta-se um estudo o qual avalia Empiricamente e Assintoticamente o método citado.

Palavras- Chave: Método Radixsort. Ordenação de Dados. Avaliação Empírica e Assintótica.

Abstract: This paper aims to present a study on the method of ordering Radixsort data. The developed research is classified as Applied, being of quantitative approach. In it is presented a study which empirically and asymptotically evaluates the method quoted.
.

Keywords: *Radixsort method. Ordering Data. Empirical and Asymptotic Evaluation.*

INTRODUÇÃO

O *RadixSort* é um algoritmo de ordenação utilizado em computadores que permitem acessar o código binário que representa os caracteres dos números que compõem a sequência a ser ordenada (MANBER, 1989).

Cormen et al. (2009) afirma que o *RadixSort* é um algoritmo de ordenação rápido e estável que pode ser usado para ordenar itens que estão identificados por chaves únicas e que cada chave é uma cadeia de caracteres ou número. Nos computadores, estas chaves são os números binários usados para representar todo caractere a partir de um conjunto de dados binários.

Embora os sistemas operacionais que permitem o acesso ao código binário não sejam convencionais, não será preciso este acesso para implementar o algoritmo, pois a ideia básica do *RadixSort* será aplicada na matriz de incidência e não no código binário. A grande vantagem desse algoritmo é que ele ordena em tempo linear, ou seja, O(n), qualquer outro

[1] Acadêmico do Curso de Ciência da Computação. Unicruz. E-mail: maurorafael@outlook.com
[2] Professora do Curso de Ciência da Computação. Unicruz. E-mail: pmozzaquatro@unicruz.edu.br
[3] Professor do Curso de Ciência da Computação. Unicruz. E-mail: alextelocken@gmail.com

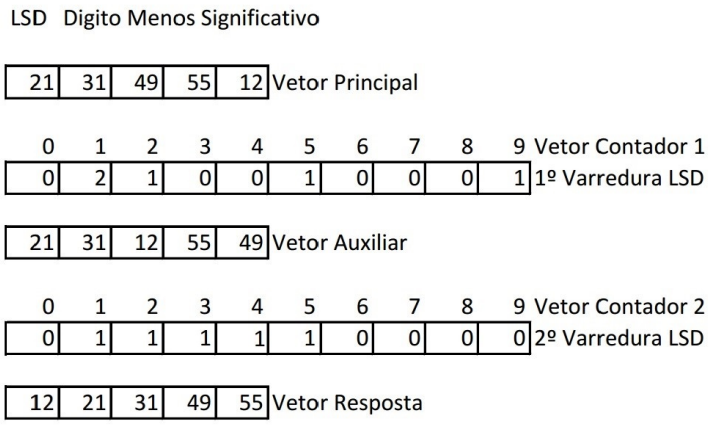
algoritmo de ordenação que não goza do artifício de acessar o código binário ordena, necessariamente, em pelo menos O (n.log n) operações (MANBER, 1989).

ALGORIMO

O *Radixsort* é um algoritmo baseado no conceito de ordenação por contagem, que consiste em criar uma tabela de N entradas, onde N é o número de elementos diferentes que pode ser inserido no vetor de entrada. E nesta tabela são inseridos contadores, incrementados se os mesmos forem correspondentes a chave de valor i, após realizado este processo, é conhecido a quantidade de posições necessárias para cada valor de chave, assim os elementos são transferidos para as posições corretas no novo vetor ordenado (RICARTE, 2003).

O *Radixsort* utiliza o conceito de ordenação por contagem, porém ele realiza a contagem com apenas uma parte da representação do elemento, a qual é denominada raiz. o processo de contagem é realizado com esta parte várias vezes até que a representação total do elemento seja analisada. Duas classificações podem ser realizadas com o *radixsort*, LSD (*Least Significant Bits*) e MSD (*Most Significant Bits*), que define a ordem que serão analisadas as partes do elemento, se LSD o processo é iniciado com os bits menos significativos e se MSD com os bits mais significativos. Com esta especificação do *radixsort* apresentada acima é possível perceber que este algoritmo pode ser aplicado a ordenação de *strings* como um vetor de entrada [bg, bd, a, f k, f j, e] sendo ordenado como [a, bd, bg, e, f j, f k]. Este algoritmo é muito utilizado para ordenação lexicográfica (MIYAZAWA, 1999). Para ordenação de números inteiros o *radixsort* pode trabalhar uma maneira simples, como a parte do elemento sendo um dígito do elemento, assim a ordenação pode ser feita ordenando os dígitos menos significativos para os mais significativos. A Figura 1 mostra a arquitetura do *radixsort*.

Figura 1 - Arquitetura *RadixSort*



Fonte: Elaborado pelo autor

A Figura 1 utiliza 3 vetores, o vetor inicial ou principal, um vetor auxiliar e um outro para receber os valores ordenados, na figura foi utilizado o digito menos significativo para ordenar o vetor.

O segundo vetor tem tamanho 10, com índice inicial 0 (zero) e o ultimo 9 (nove) já ordenados corretamente, onde cada posição receberá o valor do digito menos significativo ao qual equivale.

Utilizando a primeira posição do vetor principal como referência, na posição zero do mesmo encontra-se o número 21, onde digito menos significativo desse número é o 1, então na posição 1 do vetor contador é contabilizado 1, a cada vez que encontrar o valor 1 na mesma posição é acrescentado 1 a mais no índice 1 do vetor contador, fazendo mesmo procedimento para os demais dígitos menos significativos do vetor principal, até todos terem sidos percorridos.

Logo é montado o vetor auxiliar sendo preenchido baseando-se no vetor contador 1, onde a cada vez que o valor do vetor contador for diferente de 0 (zero) o valor é acrescentado no vetor auxiliar, ficando o mesmo parcialmente ordenado.

Quando o vetor auxiliar for totalmente percorrido e preenchido então, passa-se a avaliar o próximo digito menos significativo, e inicia-se o processo novamente, inserindo o digito menos significativo, percorrendo todo o vetor auxiliar e inserindo posição correspondente no vetor contador 2, após preenchido o vetor contador 2, é montado então o vetor resposta completamente ordenado.

COMPLEXIDADE COMPUTACIONAL

Segundo (OLIVEIRA; SOUZA, 2008) o algoritmo *Radixsort* sequencial tem complexidade O(n) Cormen et al. (2002). Sua versão paralela, quando comparada a sequencial, e bastante superior, pois apresenta, assim como o *Countingsort*, uma complexidade de tempo de O(log n).

Segundo (LIMA, JUNIOR, 2003), a melhor utilização deste método depende da quantidade de dígitos máximos que possui o valor a ser ordenado, para uma quantidade pequena de dígitos este é infinitamente o melhor sistema de ordenação. Para facilitar esta escolha, verifique se a quantidade de dígitos é menor que log n (sendo n a quantidade de valores na sequência).

A sua vantagem fica evidente quando interpretamos dígitos de forma mais geral que simplesmente 0 a 9, pois este algoritmo pode ser utilizado também para ordenação de literais ou qualquer outro tipo de dado que possa ser visto como uma dupla ordenada de itens comparáveis.

Segundo (LIMA, JUNIOR, 2003) a estabilidade da ordenação toma por base a indução de que se um número for ordenado pelo seu i-ésimo dígito sucessivamente até seu primeiro dígito, ele irá ordena-los de forma a permitir que suas posições sejam mantidas. Pois para dois números com i-ésimos dígitos distintos, o de menor valor no dígito estará antes do de maior valor, e se ambos possuem o i-ésimo digito igual, então também estarão ordenados, pois os dígitos menos significativos foram ordenados anteriormente.

Algoritmos podem ser avaliados utilizando-se vários critérios. Geralmente o que interessa é a taxa de crescimento ou espaço necessário para resolver instâncias cada vez maiores de um problema. Podemos associar um problema a um valor chamado de ‘tamanho’ do problema, que mede a quantidade de dados de entrada (BARBOSA, 2008).

O tempo que um algoritmo necessita expresso como uma função do tamanho do problema é chamado de complexidade de tempo do algoritmo. O comportamento assintótico dos algoritmos (ou funções) representa o limite do comportamento de custo quando o tamanho cresce. O comportamento assintótico pode ser definido como o comportamento de um algoritmo para grandes volumes de dados de entrada (FERREIRA, 2002).

AVALIAÇÃO EMPÍRICA

A análise de algoritmos ou análise de complexidade é um mecanismo para entender e avaliar um algoritmo em relação aos critérios destacados, bem como saber aplica-los à problemas práticos. Uma das formas mais simples de avaliar um algoritmo é através da análise empírica, que consiste em executar dois ou mais algoritmos e verificar qual o mais rápido (COUTINHO, 2016).

Segundo (RODRIGUES JUNIOR, s.d) o estudo da complexidade de um algoritmo é realizado para escolher entre vários algoritmos o mais eficiente, desenvolver algoritmos mais eficientes, a complexidade computacional torna possível determinar se a definição de determinado algoritmo é viável.

Conforme (BENITO, [s.d]) a análise empírica avalia o custo (ou complexidade) de um algoritmo a partir da avaliação da execução do mesmo quando implementado.

A avaliação ou análise empírica tem como desafios desenvolver uma implementação correta e completa, determinar a natureza dos dados de entrada e de outros fatores que têm influência no experimento, tipicamente existem três escolhas básicas de dados, que são: Reais, similar as entradas normais para o algoritmo, ela mede o custo do programa em uso. Randômico, gerados aleatoriamente não se preocupando se são dados reais, nele é testado o algoritmo em si. Problemáticos dados manipulados para simular situações anômalas, garante que o algoritmo sabe lidar com qualquer entrada (COUTINHO, 2016).

Nesse tipo de análise é necessário levar em considerações algumas variáveis na hora de comparar algoritmos empiricamente, máquinas, compiladores, sistemas e entradas problemáticas. Um dos possíveis problemas em se comparar algoritmos empiricamente é que uma implementação pode ter sido realizada mais otimizada do que a outra (COUTINHO, 2016).

AVALIAÇÃO ASSINTÓTICA

Medir a eficiência de um algoritmo ou programa significa tentar predizer os recursos necessários para seu funcionamento. O recurso que temos mais interesse neste momento é o tempo de execução embora a memória, a comunicação e o uso de portas lógicas também podem ser de interesse. Na análise de algoritmos e/ou programas alternativos para solução de um mesmo problema, aqueles mais eficientes de acordo com algum desses critérios são em

geral escolhidos como melhores. Nesta aula faremos uma discussão inicial sobre eficiência de algoritmos e programas (BARBOSA, 2008).

Ao invés de calcular exatamente os tempos de execução em máquinas específicas, a maioria das análises conta apenas o número de operações fundamentais. Tal fato deriva do fato de que ao realizar uma medida empírica do tempo de execução de um algoritmo particular em um computador particular o resultado obtido fica estritamente ligado à linguagem de programação que foi utilizada na codificação do algoritmo e na máquina onde o programa foi executado. Desta forma, uma pequena mudança no programa poderia não causar mudança significativa no programa, mas poderia representar uma significativa mudança no tempo de execução do programa. Um fator importante na medida de complexidade é o crescimento assintótico da contagem de operações executadas. Por exemplo, num algoritmo para achar o elemento máximo entre n objetos, a operação fundamental seria a comparação dos objetos, e a complexidade seria o número de comparações efetuadas pelo algoritmo, para valores grandes de n (SILVA, 2008).

A complexidade assintótica de tempo de computação de um algoritmo minimiza os efeitos de fatores que são dependentes da linguagem de programação e da máquina e concentra-se em determinar a ordem de magnitude da frequência de execução das operações. O comportamento assintótico de um algoritmo é o mais procurado, ou seja para um volume grande de dados é que a complexidade torna-se mais importante. O algoritmo assintoticamente mais eficiente é melhor para todas entradas exceto entradas relativamente pequenas (BARBOSA, 2008).

Segundo (RIBEIRO, 2014), precisa-se de ferramenta matemática para comparar funções Na análise de algoritmos usa-se a Análise Assintótica ”Matematicamente”: estudo dos comportamento dos limites CC: estudo do comportamento para input arbitrariamente grande ou ”descrição” da taxa de crescimento Usa-se uma notação específica: O[4], Ω[5], Θ[6] (e também o, ω) Permite ”simplificar” expressões como a anteriormente mostrada focando apenas nas ordens de grandeza.

Conforme (BARRÉRE, s.d) quando deseja-se exprimir o consumo de tempo de um algoritmo de uma maneira que não dependa da linguagem de programação, nem dos detalhes

[4] f(n) = O(g(n)) (majorante), Significa que C × g(n) é um limite superior de f (n)
[5] f(n) = Ω(g(n)) (minorante), Significa que C × g(n) é um limite inferior de f (n)
[6] f(n) = Θ(g(n)) (limite ”apertado” - majorante e minorante), Significa que C1 × g(n) é um limite inferior de f (n) e C2 × g(n) é um limite superior de f (n)

de implementação, nem do computador empregado. Para tornar isto possível, é preciso introduzir um modo grosseiro de comparar funções. (Dependência entre o consumo de tempo de um algoritmo e o tamanho de sua "entrada"). Essa comparação só leva em conta a "velocidade de crescimento" das funções. Assim, ela despreza fatores multiplicativos (pois a função $2n^2$, por exemplo, cresce tão rápido quanto $10n^2$) e despreza valores pequenos do argumento (a função n^2 cresce mais rápido que 100n, embora n^2 seja menor que 100n quando n é pequeno). (BARRÉRE, s.d) afirma que esta maneira de comparar funções é assintótica.

A análise assintótica fornece um bom parâmetro, mas não deve ser o único a ser considerado (RODRIGUES JUNIOR, s.d).

A Tabela 1 apresenta a Ordem de Complexidade versus tamanho da Entrada.

Tabela 1 - Ordem de Complexidade versus tamanho da Entrada

Função de Complexidade	Tamanho do problema			
	10	**10^2**	**10^3**	**10^4**
$log_2 n$	3.3	6.6	10	13.3
n	10	10^2	10^3	10^4
$n\ log_2 n$	$0.33 * 10^2$	$0.17 * 10^3$	10^4	$1.3 * 10^5$
n^2	10^2	10^4	10^6	10^8
n^3	10^3	10^6	10^9	10^{12}
2^n	1024	$1.3 * 10^{30}$	2^{1000}	2^{10000}

Fonte: (Barbosa, 2008)

No exemplo da Tabela 2, está sendo considerada uma relação dos tempos de execução de um algoritmo de uma ordem dada para entradas de tamanho dado. O desempenho do algoritmo que soluciona o problema será dado para quatro entradas de diferentes tamanhos, sendo elas: 10, 10^2, 10^3 e 10^4. Pode-se notar que nas primeiras linhas da Tabela a complexidade aumenta pouco em relação ao tamanho do problema, na última linha, porém, é assustador o aumento verificado. Através da Tabela pode-se avaliar que, o desempenho de um algoritmo cuja complexidade seja dada por uma função logarítmica é bastante superior aos demais, sendo que as funções polinomiais apresentam desempenho aceitáveis para grandes instâncias e algoritmos exponenciais apresentam desempenho totalmente insatisfatório, o que comprova sua ineficiência quando utilizados para instancias suficientemente grandes.

METODOLOGIA

A pesquisa desenvolvida classifica-se quanto à técnica a ser utilizada como Observação, quanto a natureza, uma pesquisa aplicada, pois objetiva gerar conhecimentos

para aplicação prática, dirigida à solução de problemas específicos. Envolve verdades e interesses locais. Quanto a abordagem a mesma classifica-se como quantitativa.

A complexidade de tempo de um algoritmo pode ser dividida em 3 aspectos:

1. Melhor caso – o melhor caso representa uma instância que faz o algoritmo executar utilizando o menor tempo possível.

2. Pior caso – o maior tempo demorado pelo algoritmo para executar alguma instância.

3. Caso médio – a média de tempo que o algoritmo demora para executar.

Geralmente, o mais importante é avaliar o pior caso (porque pode inviabilizar o algoritmo) e o caso médio, porque representa como o programa vai se comportar, na prática, na maioria dos casos.

O projeto está subdividido nas seguintes etapas:

Etapa 1 – Estudo Teórico:

- Pesquisar sobre a descoberta de conhecimento em base de dados;
- Analisar a contribuição da estrutura de dados na agilidade das buscas;
- Estudar os métodos de ordenação de dados.
- Pesquisar sobre os métodos *Radixsort*.
- Analisar e pesquisar técnicas de avaliação empírica e assintótica.
- Definir e estudar linguagens serem utilizadas na implementação.

Etapa 2- Desenvolvimento

- Criar vetores em diferentes tamanhos e ordenações.
- Realizar testes iniciais empíricos e assintóticos

Etapa 4 – Validação

- Validar a aplicação utilizando analises estatísticas.

RESULTADOS

Para a implementação foram utilizados um Computador marca *Dell* modelo *laptop Inspiron* 15 5548210-ADNT, processador *Intel Core i7-5500U, 2.4GHz* , memória *RAM* 16 *GB DDR3*, sistema operacional Windows 10 Pro *CodeBlock, Microsoft Excel,* e *Star UML*, um computador marca Acer modelo *laptop* E1-571*, Intel Core i5-2328M 2.2 GHz; Dual-core*, Memória *RAM 6GB DDR3,* sistema Operacional *Linux Ubuntu 16.04.2 LTS x64, Kernel: 4.8.0-46-generic* com o software *LibreOffice Calc.*

O algoritmo de ordenação foi escritos na linguagem C. O software utilizado para edição e compilação do código foi o *Codeblocks*.

A implementação iniciou-se com as seguintes etapas:

Etapa 1: criação dos vetores em arquivos de texto no formato txt.

Foram criados vetores com tamanho de 100, 1.000, 10.000, 100.000 e 500.000 inteiros, ordenados da seguinte forma:

- 10%, 30% e 50% desordenado em ordem crescente;
- 10%, 30% e 50% desordenado em ordem decrescente;
- totalmente desordenado.

A Figura 2 mostra a função principal do Método Radixsort.

Figura 2 – Função principal método Radixsort

```
void radixSort(int vetA[], int size) {
    int i, max, temp[tam], count[10], expo = 1;

    max = pegaMax(vetA, size);

    while(max / expo > 0) {
        for(i = 0; i < 10; i++) {
            count[i] = 0;
        }

        for(i = 0; i < size; i++) {
            count[ (vetA[i] / expo) % 10 ]++;
        }

        for(i = 1; i < size; i++) {
            count[i] += count [i - 1];
        }

        for(i = size - 1; i >= 0; i--) {
            temp[ count[ (vetA[i]/expo) % 10 ]  - 1] = vetA[i];
            count[ (vetA[i]/expo) % 10 ]--;
        }

        for(i = 0; i < size; i++) {
            vetA[i] = temp[i];
        }

        expo *= 10;
    }
}
```

Fonte: Elaborado pelo Autor

A Tabela 2 apresenta os resultados obtidos com 2 configurações de hardware e *softwares* diferentes, mencionados anteriormente.

Tabela 2- Resultados em Milissegundos do Método Radixsort

Computador 1					
Computador marca Dell modelo laptop Inspiron 15 5548210-ADNT, processador Intel Core i7-5500U, 2.4GHz , memória RAM 16 GB DDR3, sistema operacional Windows 10 Pro CodeBlock					
Método Radixsort/Tamanho Vetor	100	1000	10000	100000	500000
Ordem Crescente Tot. Ordenado	0	0	5	43	-
Ordem Decrescente Tot. Ordenado	0	0	4	48	-
Ordem Crescente 10% Desordenado	0	1	8	45	-
Ordem Decrescente 10% Desordenado	0	1	6	41	-
Ordem Crescente 30% Desordenado	0	1	4	54	-
Ordem Decrescente 30% Desordenado	0	0	5	36	-
Ordem Crescente 50% Desordenado	0	1	4	37	-
Ordem Decrescente 50% Desordenado	0	1	6	39	-
Totalmente Desordenado	0	0	7	44	-
Computador 2					
Computador marca Acer modelo laptop E1-571, Intel Core i5-2328M 2.2 GHz; Dual-core, Memória RAM 6GB DDR3, sistema Operacional Linux Ubuntu 16.04.2 LTS x64, Kernel: 4.8.0-46-generic					
Método Radixsort/Tamanho Vetor	100	1000	10000	100000	500000
Ordem Crescente Tot. Ordenado	94	366	3092	31689	158462
Ordem Decrescente Tot. Ordenado	95	378	3464	23844	151267
Ordem Crescente 10% Desordenado	70	343	3440	33902	155926
Ordem Decrescente 10% Desordenado	90	358	3371	37905	184620
Ordem Crescente 30% Desordenado	67	324	1904	24701	129893
Ordem Decrescente 30% Desordenado	68	233	1840	24206	137137
Ordem Crescente 50% Desordenado	68	215	2016	26748	143853
Ordem Decrescente 50% Desordenado	98	213	1823	23954	125791
Totalmente Desordenado	68	215	1659	21233	116780

Fonte: Elaborado pelo Autor

CONCLUSÃO

O método *Radixsort* foi aplicado utilizando o mesmo algoritmo e mesmos vetores com ordenação, valores e tamanhos idênticos para ambos os computadores, sendo que os vetores de tamanho 500.000 não foram executados no computador 1.

Os resultados obtidos são de grande valia, comprovando a teoria pesquisada para realizar este estudo, no qual foi possível analisar o Métodos *RadixSort* com seu respectivo desempenho , dificuldades de implementação, consumo de *hardware*, tempo de execução, podendo vir a ser utilizada para estudos de ordenação de dados com métodos de ordenação.

Este artigo é parte integrante de um trabalho de conclusão de curso em andamento. A pesquisa em andamento visa desenvolver um método híbrido integrando os algoritmos quicksort, countingsort e radixsort. Será realizado um estudo de caso a fim de verificar empiricamente e assintoticamente o desempenho computacional do método híbrido em relação aos métodos separadamente. Até o momento foram implementados os métodos separados (realizadas avaliações quanto ao tempo de execução, trocas e interações e vetores de tamanhos diversos) a fim de coletar informações a serem implementadas no método híbrido.

REFERÊNCIAS

Barbosa, Marco Antônio. **Estrutura de Dados II**. Curso de Ciência da Computação. Cruz Alta, 2008.

Benito, Franck Carlos Vélez. **Análise de Algoritimos.** 52 f. TCC (Graduação) - Curso de Ciência da Computação, Ciência da Computação, Universidade Tecnológica Federal do Parana, Santa Helena ,s.d

COUTINHO, Demétrios. Algoritmos. Instituto Federal de Educação, Ciência e Tecnologia, Rio Grande do Norte , 2016. Disponível em: http://docente.ifrn.edu.br/demetrioscoutinho/disciplinas . Acesso em novembro 2016.

Lima Junior, Vieira; Araujo, Everson Santos. **ALGORITMOS DE ORDENAÇÃO:** estudo comparativo de diversos algoritmos de ordenação. 2003. 15 f. TCC (Graduação) - Curso de Sistemas de Informação, Faculdade de Imperatriz, Imperatriz - MA, 2003.

Manber, Udi**. Introduction to algorithms: a creative approach** , Addison-Wesley, New York, 1989

.

Miyazawa, F. K. **Notas de complexidade de algoritmo**s. Universidade Estadual de Campinas, 1999.

Oliveira, André Luis Faria de; Souza, Uéverton dos Santos. **Algoritmos Paralelos De Ordenação.** 2008. 70 f. TCC (Graduação) - Curso de Tecnologia em Sistemas de Computação, Universidade Federal Fluminense, Niterói - Rj, 2008.

Ricarte L. M. I. **Ordenação por contagem**. Disponível em: http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node24.html. Acessado em: 20 Ago. 2016, 2003.

Rodrigues Junior, José Fernando. **Complexidade de Algoritmos: Tempo e espaço, Notação assintótica, Algoritmos polinomiais e intratáveis.** 50 f. - Curso de Ciência da Computação, Universidade de São Paulo, São Paulo - Sp, s.d.

Cormen et al; T.H.; Leiserson, C.E.;Rivest,r.l.; stein, C. **Introduction to Algorithms**, 3. ed., MIT Press, Boston, 2009.

Cormen et al; Leiserson, C. E.; Rivest, R. L.; Sten, C.; **Algoritmos - tradução da 2ª edição americana** – Teoria e Prática, Campus, 2002.

Ferreira, Fernando Nunes; lopes, Joao Correia. Apontamentos das teóricas:Recursividade. Universidade do Porto, 2002. Disponivel em: <http://paginas. fe.up.pt/programacaoI/2001-02/livro/cap2.pdf>. Acesso em set de 2017.

Silva, Alexandre Tadeu Rossini da; OLIVEIRA, Marcelo Ribeiro de. **Análise e desenvolvimento de sistemas. Estrutura de dados**, 2008.

Ribeiro, Pedro. **Análise Assint´otica de Algoritmos.** 2014. 75 f. TCC (Graduação) - Curso de Ciência de Computadores, Departamento de Ciência de Computadores, Faculdade de Ciências da Universidade do Porto, Porto - Por, 2015.

Barrére, Eduardo. **Análise e Projeto de Algoritmos.** 30 f. Curso de Ciência da Computação, Departamento de Ciência da Computação, Universidade Federal de Juiz de Fora, Juiz de Fora - Mg, s.d.