Centro Universitário do Norte - UniNorte Laureate
Escola de Exatas
Bacharelado em Engenharia da Computação

Allex Lima - 14003147
Daniel Bispo - 14257165
Paulo Moraes - 12078549
Renan Barroncas - 14043300

Analise Assintótica - QuickSort

Manaus - AM
Junho de 2016

Allex Lima - 14003147
Daniel Bispo - 14257165
Paulo Moraes - 12078549
Renan Barroncas - 14043300

Analise Assintótica - QuickSort

Trabalho apresentado como requisito parcial para obtenção de aprovação na disciplina Teoria dos Grafos e Computabilidade, do Curso de Engenharia da Computação, no Centro Universitário do Norte - UniNorte Laureate.

Prof°. M.Sc. Jonathas Santos

Manaus - AM

Junho de 2016

1 Algoritmo

```
static int divide(int *v, int begin, int end){
    int i = begin + 1,       //Recebe a segunda posicao do vetor
        j = end,         //Recebe a ultima posicao do vetor
        pivo = begin,    //Pivo/Base inicial
        aux;     //Variavel auxiliar

    while(i <= j){
        if(v[pivo] >= v[i])
            i++;
        else if(v[pivo] < v[j])
            j--;
        else{
            aux = v[i];
            v[i] = v[j];
            v[j] = aux;
            i++;
            j--;
        }
    }

    aux = v[pivo];
    v[pivo] = v[j];
    v[j] = aux;

    return j;
}

void quicksort(int *v, int begin, int end){
    if(begin < end){
        int pivo = divide(v, begin, end);
        quicksort(v, begin, pivo-1);
        quicksort(v, pivo+1, end);
    }
}
```

2 Analise assintótica

O algoritmo *quicksort* depende, para apresentar um desempennho aceitável, diretamente da sua chave, ou pivô. Uma vez que este apresenta divergência em duas metodologias de aplicar a chave.

No melhor caso quando a função fracture for chamada ela divide exatamente no meio o vetor. Por exemplo, dado a função demonstrada anteriormente, como uma árvore binária por exemplo. Temos a seguinte analise:

```
void quicksort(int *v, int begin, int end){
    if(begin < end){                                            // n-1
        int pivo = divide(v, begin, end);                       // n/2
        quicksort(v, begin, pivo-1);                            // n/2
        quicksort(v, pivo+1, end);                              // n/2
    }
}
```

Figura 2.1 – Procedimento base do algoritmo de ordenaçao *quicksort*

$$f(n) = n - 1 + (n/2) + n + n \tag{2.1}$$

$$f(n) = 3n + (n/2) - 1 \tag{2.2}$$

$$f(n) = 3n + (n/2) \tag{2.3}$$

Logo, este reduzirá em uma

$$T(n) = T(n/2) + \theta(n \log n), \tag{2.4}$$

onde *T(n)* é o número n de elementos a serem ordenados, e

$$\theta(nlogn) \tag{2.5}$$

é a quantidade de subdivisão realizada no vetor. Neste caso temos uma analise em *O(n)*.

No pior caso a função vai apresentar um índice de custo de processamento maior pois o pivo vai estar iniciado em posições de menor ou maior valor e isso fazer parecer uma árvore binário com uma folha por nível, parecendo com uma lista de profundidade com raís *n-1*. Neste caso apresentamos:

$$T(n) = T(n-1) \tag{2.6}$$

$$T(n) = T(n-1) + \theta(n), \tag{2.7}$$

onde

$$\theta(n) = \theta(n^2) \tag{2.8}$$

pois no instante de pseudo escolha do pivo cresce quadraticamente a quantidade de interações.

$$f(n) = 3T(n-1) + \theta(n^2) \tag{2.9}$$

Digamos que este vetor esta balanceado com ordenação de *(p, m)*, sendo $T(n) = T(pn) + T(mn) + \theta(n)$, sabe-se que $0 < p \leq 1/2$ e $p + m = 1$, pois eles são constantes da função, o menor ramo da subrepartição terá tamanho de $\theta(\log_1 /pn)$ e o mais longo $\theta(\log_1 /mn)$, mesmo assim apresentará o mesmo número de comparações que é $\theta(n \log n)$.

Reduzindo a equação:

$$T(n) = max(T(m) + T(p-n)) + \theta(n) \tag{2.10}$$

Temos o somatório das constantes p e m:

$$T(n) \leq cn^2 \tag{2.11}$$

Para c sendo uma constante qualquer e n um número de elementos de ordenação.

3 Resultados e Testes

Uma função de geração de números randômicos auxiliou a execução dos testes no algoritmo. Dessa forma, conjuntos de valores pseudoaleatórios foram criados em diferentes ordens com a finalidade analisar o comportamento do *quicksort* ao ordenar tais dados. A Tabela 1 demonstra o tempo, em *segundos*, que se fizeram necessários para realizar a ordenação das respectivas quantidades de dados.

	100	**1.000**	**10.000**
Ordem Crescente	0,000022	0,000311	0,008077
Ordem Decrescente	0,000017	0,000390	0,023088
Ordem Pseudoaleatória	0,000036	0,000281	0,008279

Tabela 1 – Dados expressos em segundos.

Vale levar em consideração, que o tempo de execução da ordenação é um valor que pode apresentar-se de forma alternada em diferentes máquinas, uma vez que os componentes de processamento do computador podem favorecer, ou não, tal análise. Os testes acima foram realizados em computador com um processador *Intel® Core™ i5*, com um *clock* de 2.50GHz e 4 núcleos, memória 5.7 GiB e com o sistema operacional *GNU/Linux Debian Jessie 8*.