Instituto Federal de Educação, Ciência e Tecnologia de São Paulo

Trabalho Análise de Algoritmos

Docente: Bento Rafael Siqueira

Discentes:

Rodrigo Luis Tavano Bosso – PC3005623

Complexidade de Algoritmos de Busca e Ordenação

0-Resumo:

O seguinte trabalho tem como objetivo explicar o funcionamento de alguns algoritmos de Busca e Ordenação passando por seus conceitos, análise de desempenho, analisando sua complexidade e realizando a apresentação de alguns exemplos.

Para que se possa realizar tal discussão, estaremos fundamentados em uma série de pesquisas em links e livros (principalmente o exemplar :Projeto de Algoritmos de Nivio Ziviani) que explicam o funcionamento geral dos Algoritmos Apresentados. Consoante a tais fatores partiremos para o texto propriamente dito.

Palavras-chave: Algoritmos, Complexidade ,Busca, Ordenação.

Sumário:

1- Introdução:

Complexidade de algoritmo é a quantidade de trabalho necessário para executar uma tarefa, geralmente medida por meio do tempo necessário para executar um determinado código. Isto é medido em cima das funções fundamentais que o algoritmo é capaz de fazer, o volume de dados (quantidade de elementos a processar), e é claro, a maneira como ele chega no resultado.

Esta quantidade de trabalho tem a ver com tempo, mas também pode referir-se à quantidade de memória necessária para garantir sua execução, desta forma podemos medir tanto o algoritmo mais eficiente em relação ao tempo e a memória, e balancear esses recursos de acordo com a prioridade que cada problema demanda.

Ao comparar diferentes grandezas. Estabelece-se a complexidade medindo como o algoritmo se comporta quando ele precisa manipular, por exemplo: 10, 100 e 1000 elementos. Se ele consegue fazer em uma operação nos três casos, o algoritmo é constante, se cada uma precisa 10 vezes mais operações ele é linear e se preciso do quadrado do número de elementos, obviamente é quadrático.

Não é difícil notar como isto pode fazer uma enorme diferença de desempenho se você tem grandes volumes de dados. Em algoritmos de altíssima complexidade até pequenos volumes podem ter um desempenho bastante ruim.

2- Algoritmo de ordenação ShellSort:

2.1- Apresentação ShellSort:

O ShellSort foi um algoritmo desenvolvido por volta de 1959 onde propôs a extinção de algoritmos de ordenação por inserção p método da inserção troca itens adjacentes quando se está procurando o ponto de inserção na sequência destino. Assim caso o menor item estiver na posição mais a direita do vetor, então o número de comparações e movimentações é igual a n-1 para encontrar p seu ponto de inserção.

O Método Shell contorna esse problema, uma vez que as trocas de registros que estão distantes um do outro ocorrem. Assim os itens que estão separados $\boldsymbol{h}$ posições *são* rearranjados, de forma que o *h-ésimo* termo leva uma sequência ordenada.

2.2 – Exemplificando ShellSort:

Exemplificando com um vetor pequeno [6] posições, tratando como letras para exemplificar o funcionamento:

Posição do vetor	**1**	**2**	**3**	**4**	**5**	**6**
Chaves Iniciais:	*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*

Como nota-se a primeira rodagem tem-se (*h = 4*) , O e N (Posições 1 e 5) são comparados e trocados a seguir , R e A (2 e 6) são comparados e trocados. E esse processo vai se repetindo até a última passada (*h = 1*) que corresponde a algoritmo de inserção , entretanto nenhum item precisa se mover para posições muito distantes.

Várias sequencias para *h* tem sido experimentada. Mas por meio desses experimentos, foi possível saber que a escolha do incremento para *h,* mostrada a seguir é difícil de ser batida por mais de 20% de eficiência no tempo de execução:

$$\boldsymbol{h(s) = 3h(s-1) + 1\,, para\ s > 1}$$

$$\boldsymbol{h(s) = \ 1\,, para\ s = 1}$$

2.3 – Código exemplo ShellSort:

Logo abaixo está exemplificado código ShellSort em Python:

```
def shellSort(nums):
    h = 1
    n = len(nums)
    while h > 0:
        for i in range(h, n):
            c = nums[i]
            j = i
            while j >= h and c < nums[j - h]:
                nums[j] = nums[j - h]
                j = j - h
                nums[j] = c
        h = int(h / 2.2)
    return nums
```

2.4 – Análise ShellSort:

A análise do código ShellSort contém alguns problemas matemáticos muito difíceis, como a própria sequência de incrementos, pois pouco se sabe sobre cada incremento, não deve ser múltiplo do anterior. Porém para as sequencias de incrementos utilizadas existem 2 principais conjunturas que descrevem o número de comparações.

$$\boldsymbol{Conjuntura\ 1: \ O(n^{1,25})}$$

$$\boldsymbol{Conjuntura\ 2: \ O(n * log[n])}$$

O Shellsort é uma ótima opção para arquivos de tamanho moderado, além disso sua implementação é simples e requer uma quantidade pequena de código. Existem métodos de ordenação mais eficientes, mas são também muito mais complicados de implementar.

O tempo de execução do algoritmo é sensível à ordem inicial do arquivo. Além disso, o método **não é estável,** pois ele nem sempre deixa registros com chaves iguais, conservando a mesma posição relativa.

3- Algoritmo de ordenação QuickSort:

3.1- Apresentação QuickSort:

O QuickSort foi um algoritmo desenvolvido por C.A.R.Hoare em 1960 , quando visitava a Universidade de Moscou como estudante. O Algorítimo foi publicado mais tarde por Hoare(1962), após uma série de refinamentos.

A ideia consiste me dividir o problema e ordenar um conjunto com itens em dois problemas menores. A seguir, os problemas menores são ordenados independentemente e depois os resultados sçai combinados para produzir a solução do problema maior.

A parte mais delicada desse método é o procedimento partição , em que deve-se rearranjar o vetor A[Esq..Dir] por meio da escolha arbitrária de um item , chamada de ***pivô*** , de tal forma que ao final o vetor , seja particionado em uma parte a esquerda com chaves menores ou iguais ao ***pivô*** e uma parte a direita com chaves maiores ou iguais ao ***pivô*** .

3.2 - Exemplificando QuickSort:

Geralmente o ***pivô*** é escolhido como sendo : $A\,[\,(i+j)/2]$. Como inicialmente *i = 1* e *j = 6* então ***pivô*** = *A[3]* . em sequencia A varredura a partir da posição 1 para o item do Pivô e o mesmo ocorre para o item de posição 6 . como demonstra as posições abaixo:

Chaves Iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
1	*A*	***D***	*R*	*E*	*N*	*O*
2	***A***	*D*				
3			***E***	*R*	*N*	*O*
4				***N***	*R*	*O*
5					*O*	***R***
6	*A*	*D*	*E*	*N*	*O*	*R*

Acima temos a ilustração do funcionamento do algoritmo que se estende de forma de eleger o pivô por meio do código de eleição, particiona o vetor, e enfim realiza a ordenação necessária.

3.3 – Código exemplo QuickSort:

Logo abaixo está exemplificado código QuickSort em Python:

```
def quick_sort(a, ini=0, fim=None):
    fim = fim if fim is not None else len(a)
    if ini < fim:
        pp = particao(a, ini, fim)
        quick_sort(a, ini, pp)
        quick_sort(a, pp + 1, fim)
    return a

def particao(a, ini, fim):
    # para uma versão com partição aleatória
    # descomente as próximas três linhas:
    # from random import randrange
    # rand = randrange(ini, fim)
    # a[rand], a[fim - 1] = a[fim - 1], a[rand]
    pivo = a[fim - 1]
    for i in range(ini, fim):
        if a[i] > pivo:
            fim += 1
        else:
            fim += 1
            ini += 1
            a[i], a[ini - 1] = a[ini - 1], a[i]
    return ini - 1

a = [8, 5, 12, 55, 3, 7, 82, 44, 35, 25, 41, 29, 17]
print(a)
print(quick_sort(a))
```

3.4 – Análise QuickSort:

Uma característica interessante do QuickSort é a ineficiência para arquivos já ordenados quando a escolha do pivô é inadequada. Como exemplo a escolha sistemática dos exemplos de um arquivo já ordenado leva ao seu pior caso. Nesse caso as participações serão extremamente desiguais, e o procedimento ordena será chamado recursivamente n vezes, eliminado de comparações, passa cada chamada. Essa situação é desastrosa, pois o número de comparações passa o número de comparações passa a cerca de $n^2/2$, e o tamanho da pilha necessária para as chamadas recursivas é cerca de n . Entretanto, o pior caso pode ser evitado empregando-se pequenas modificações no programa, conforme veremos mais adiante.

A melhor situação possível ocorre quando cada partição divide o arquivo em duas partes iguais. Logo,

$$C(n) = 2C(n/2) + n - 1$$

Onde C(n/2) representa o custo de ordenar uma das metades e n-1 é o número de comparações realizadas. A solução para esta recorrência é:

$$C(n) = n \log n - n + 1$$

No caso médio, o número de comparações realizadas seria em torno de:

$$C(n) = 1,386\, n \log n - 0,846\, n$$

Oque significa que em média o tempo de execução do QuickSort é $O(n \log n)$ **no melhor caso e** $O(n^2)$ **no pior dos casos** .

Com isso sabe-se que o QuickSort é extremamente eficiente para ordenar arquivos de dados. o Método necessita apenas de uma pequena pilha como memória auxiliar e requer cerca de n log(n) operações, em média, para ordenar n itens. Como aspectos negativos cabe que:

1. A versão recursiva do algoritmo tem pior caso que é O(n²) operações;
2. A implementação do algoritmo é muito delicada de difícil: um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados;
3. O método não é estável .

Uma vez que se consiga uma implementação robusta para o Quicksort, este deve ser o algoritmo preferido para a maioria das aplicações. No caso de necessitar de um programa utilitário para o uso frequente.

4- Algoritmo de ordenação Radix Sort:

4.1- Apresentação Radix Sort:

O Radix Sort é um algoritmo de ordenação rápido e estável que pode ser usado para ordenar itens que estão identificados por chaves únicas. Cada chave é uma cadeia de caracteres ou número, e o radix sort ordena estas chaves em qualquer ordem relacionada com a lexicografia.

Na ciência da computação, radix sort é um algoritmo de ordenação que ordena inteiros processando dígitos individuais. Como os inteiros podem representar strings compostas de caracteres (como nomes ou datas) e pontos flutuantes especialmente formatados, radix sort não é limitado somente a inteiros.

Existem duas classificações do radix sort, que são:

- **Least significant digit (LSD – Dígito menos significativo) radix sort;**
- **Most significant digit (MSD – Dígito mais significativo) radix sort.**

O radix sort LSD começa do dígito menos significativo até o mais significativo, ordenando tipicamente da seguinte forma: chaves curtas vem antes de chaves longas, e chaves de mesmo tamanho são ordenadas lexicograficamente. Isso coincide com a ordem normal de representação dos inteiros, como a sequência "1, 2, 3, 4, 5, 6, 7, 8, 9, 10". Os valores processados pelo algoritmo de ordenação são frequentemente chamados de “chaves”, que podem existir por si próprias ou associadas a outros dados. As chaves podem ser strings de caracteres ou números.

Já o radix sort MSD trabalha no sentido contrário, usando sempre a ordem lexicográfica, que é adequada para ordenação de strings, como palavras, ou representações de inteiros com tamanho fixo. A seqüência "b, c, d, e, f, g, h, i, j, ba" será ordenada lexicograficamente como "b, ba, c, d, e, f, g, h, i, j". Se essa ordenação for usada para ordenar representações de inteiros com tamanho variável, então a representação dos números inteiros de 1 a 10 terá a saída "1, 10, 2, 3, 4, 5, 6, 7, 8, 9".

4.2- Exemplificando Radix Sort:

O Inicialmente ocorre uma quebra de chave em pedaços , dígitos de um número em uma dada base (radix) . Exemplo o número: 312 tem os dígitos 3, 1 e 2 na base 10 } 312 tem os dígitos 100111000 na base 2 } assim como a palavra EXEMPLO e tem 6 caracteres (base 256).

Depois das quebras de ordem de carácteres ocorre a ordenação desses de acordo com o código LSD ou MSD , entretanto ambos consistem em Ordenar o vetor de acordo com o primeiro pedaço de cada chave. Em um cenário de números binários seria: Números cujo dígito mais à esquerda começa com 0 vêm antes de números cujo dígito mais à esquerda é 1.

Para isso iremos contar quantos elementos existem de cada valor , para então poder calcular a posição deles no vetor ordenado e enfim realizar a colocação dos elementos em suas posições.

Chaves Iniciais:	*123*	*124*	087	263	233	*014*	*132*
[1]	*12**3***	*12**4***	08**7**	26**3**	23**3**	*01**4***	*13**2***
[2]	*1**4**2*	*1**3**2*	*1**2**3*	*2**6**3*	*2**3**3*	*0**1**4*	*0**8**7*
[3]	***0**14*	***1**23*	***1**32*	***2**33*	***1**42*	***2**63*	***0**87*
R	*014*	*087*	*123*	*132*	*142*	*233*	*263*

Com poucas instancias de algoritmo já se é capaz de encontrar a resposta.

4.3 – Código exemplo Radix Sort:

Logo abaixo está exemplificado código Radix Sort em C:

```
void radixsort(int vetor[], int tamanho) {
    int i;
    int *b;
    int maior = vetor[0];
    int exp = 1;

    b = (int *)calloc(tamanho, sizeof(int));

    for (i = 0; i < tamanho; i++) {
        if (vetor[i] > maior)
            maior = vetor[i];
    }

    while (maior/exp > 0) {
        int bucket[10] = { 0 };
          for (i = 0; i < tamanho; i++)
              bucket[(vetor[i] / exp) % 10]++;
          for (i = 1; i < 10; i++)
              bucket[i] += bucket[i - 1];
          for (i = tamanho - 1; i >= 0; i--)
              b[--bucket[(vetor[i] / exp) % 10]] = vetor[i];
          for (i = 0; i < tamanho; i++)
              vetor[i] = b[i];
          exp *= 10;
    }

    free(b);
}
```

4.4- Análise Radix Sort:

O radix sort LSD opera na notação Big-O, $\boldsymbol{O(nk)}$, onde $\boldsymbol{n}$ é o número e chaves, e $\boldsymbol{k}$ é o comprimento médio da chave. Logo é possível garantir esse desempenho para chaves com comprimento variável agrupando todas as chaves que têm o mesmo comprimento e ordenando separadamente cada grupo de chaves, do mais curto para o mais comprido, de modo a evitar o processamento de uma lista inteira de chaves em cada passo da ordenação.

Em muitas aplicações em que é necessário velocidade, o radix sort melhora as ordenações por comparação, como o seu parente semelhante , o heapsort e também o mergesort, uma vez que também divide chaves e cria uma seleção de mistura. Porém o Radix Sort que necessitam de$(\boldsymbol{n} \backslash \boldsymbol{log}(\boldsymbol{n}))$ comparações, em que $\boldsymbol{n}$ é o número de itens a serem ordenados.

Sua complexidade Assintótica pode ser medida por meio do tempo através do algoritmo $\theta\ (\boldsymbol{nk})$ e a complexidade no espaço:
$\theta\ (\boldsymbol{n} + \boldsymbol{s})$, onde :

$\boldsymbol{n}$ = número de elementos.

$\boldsymbol{k}$ = tamanho string.

$\boldsymbol{s}$ = tamanho do alfabeto.

Em síntese , sua complexidade para um número pequeno ou constante, então o **Radix Sort** tem custo linear $\mathbf{O}(\boldsymbol{n})$.

5- Algoritmo de Busca Sequencial:

5.1- Apresentação Busca Sequencial:

O método de pesquisa simples que existe e funciona. o código resila o primeiro registro, e pesquisa sequencialmente até encontrar a chave procurada, então para caso encontra ou no pior dos casos a busca não retorna resultado , e pesquisa é feita em todo o espaço pesquisável.

Apesar de simples a busca sequencial envolve aspectos interessantes a se atentar servindo para ilustrar vários aspectos e convenções a serem utilizadas em outros métodos de pesquisa.

Em síntese a função busca e retorna o índice do registro que contém a chave x , caso não esteja presente , o valor retornado é 0 , Vale apena se destacar que essa implementação não suporta mais que um único registro com uma mesma chave. Logo em aplicações que decidem optar por esta é necessário incluir um argumento a mais na função de busca, para conter o índice que se quer pesquisar.

5.2- Exemplificando Busca Sequencial:

Para exemplificar melhor o funcionamento desse algoritmo usaremos esse Gif (imagem 1) para poder demonstrar os passos que são realizados:

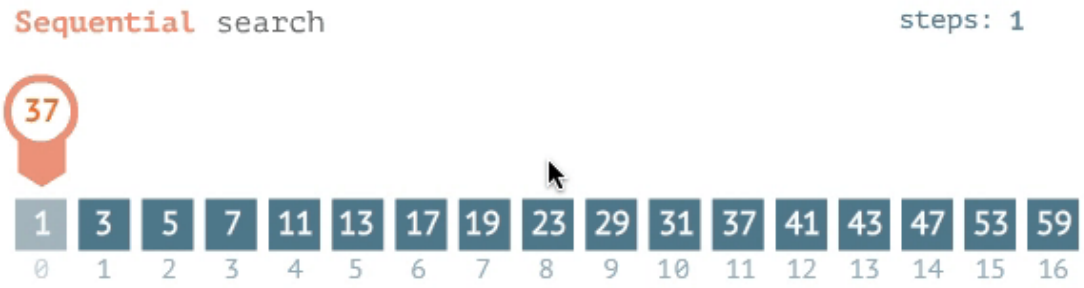

Imagem 1 , Gif exemplificando o funcionamento de uma Busca sequencial (disponível para consulta = https://cdn-media-1.freecodecamp.org/images/CMLgOmQiGx-An2R8TeY3yghPSmQzfHc4KCsa)

5.3 – Código exemplo Busca Sequencial:

Logo abaixo está exemplificado código de Busca Sequencial em Python:

```
def procura(lista, elemento):
    assert isinstance(lista, list)
    for indice in range(len(lista)):
        if lista[indice] == elemento:
            return indice
    else:
        return None
```

5.4- Análise Busca Sequencial:

Para uma pesquisa com sucesso ,temos os seguintes casos:

- Melhor caso : $\boldsymbol{C(n) = 1}$
- Pior caso: $\boldsymbol{C(n) = n}$
- Caso médio : $\boldsymbol{C(n) = (n + 1)/2}$

Para uma pesquisa sem sucesso temos:

$$\boldsymbol{C(n) = n + 1}$$

Devido a sua resposta relativamente ineficiente ,este algoritmo é um algoritmo indicado como solução para problemas de pesquisas em tabelas com no máximo 25 registros ou menos.

6- Algoritmo de Busca Binária:

6.1- Apresentação Busca Binária:

A pesquisa em uma tabela pode ser muito eficiente se os registros forem mantidos em ordem. Para saber se a chave está representada na tabela compare a chave com o registro que está na posição do meio da tabela, compare a chave com o registro que está na posição do meio da tabela. Se a chave é menor, então o registro procurado está primeira metade da tabela , se a chave é maior , então o registro procurado está na segunda parte da tabela. O procedimento se perpetua até que a chave seja encontrada ou fique apenas um registro, cuja chave é diferente da procurada, indicando uma pesquisa sem sucesso.

6.2- Exemplificando Busca Binária:

A tabela busca mostrar os subconjuntos pesquisados para recuperar o índice da chave G.

Posição do vetor	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Chaves Iniciais:	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*
Índice = G	*A*	*B*	*C*	***D***	*E*	*F*	*G*	*H*
Índice = G					*E*	***F***	*G*	*H*
Índice = G							***G***	*H*

6.3 – Código exemplo Busca Binária:

Logo abaixo está exemplificado código de Busca Binária em C:

```
int PesquisaBinaria (int vet[], int chave, int Tam)
{
    int inf = 0;     // limite inferior (o primeiro índice de vetor em C é zero         )
    int sup = Tam-1; // limite superior (termina em um número a menos. 0 a 9
são 10 números)
    int meio;
    while (inf <= sup)
    {
        meio = (inf + sup)/2;
        if (chave == vet[meio])
            return meio;
        if (chave < vet[meio])
            sup = meio-1;
        else
            inf = meio+1;
    }
    return -1;   // não encontrado
}
```

6.4- Análise Busca Binária:

A cada interação do algoritmo, o tamanho da tabela é dividido ao meio, logo o número de vezes que o tamanho da tabela é dividido ao meio é cerca de : $\boldsymbol{log}(\boldsymbol{n})$. Entretanto, o custo para manter a tabela ordenada é alto, cada inserção na posição *p* da tabela implica o deslocamento dos registros da partir da posição *p* para as posições seguintes. Consequentemente, a pesquisa binária deve ser evitada em aplicações muito dinâmicas.

Em síntese, a complexidade para da Busca Binária tem custo igual a : $\mathbf{O}(\mathbf{log}\,\boldsymbol{n})$.

7- Comparações gráficas e Conclusão:

7.1- Comparação das ordenações apresentadas:

Na comparação abaixo podemos observar em casos com tamanho médio de vetor, apesar do Shell Sort ser representado com a complexidade $\boldsymbol{O(n^{1,25})}$ ele pode assumir uma complexidade na ordem do Quick Sort também podendo chegar a $\boldsymbol{O(n * \log(n))}$, já o Radix Sort foi representado com a complexidade de caso médio para números com alfabetos de 10 caracteres e com espaço amostral de 2 casas , uma vez que os valores analisados são relativamente baixos obtendo assim $\boldsymbol{O(n)}$.

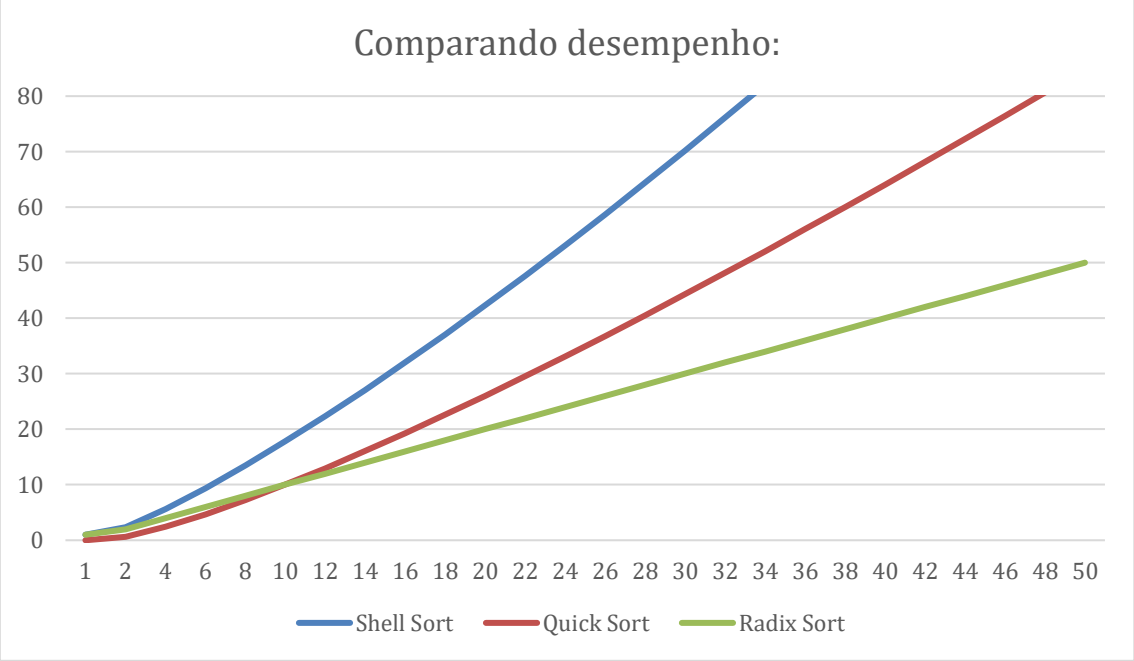

Gráfico 1 , Comparação de performance de Shell , Quick e Radix

7.2- Comparação das Buscas Apresentadas:

Ao comparar as Busca Sequencial e Binária vemos uma diferença absurda na otimização, apresentando os casos médios e piores casos de cada uma:

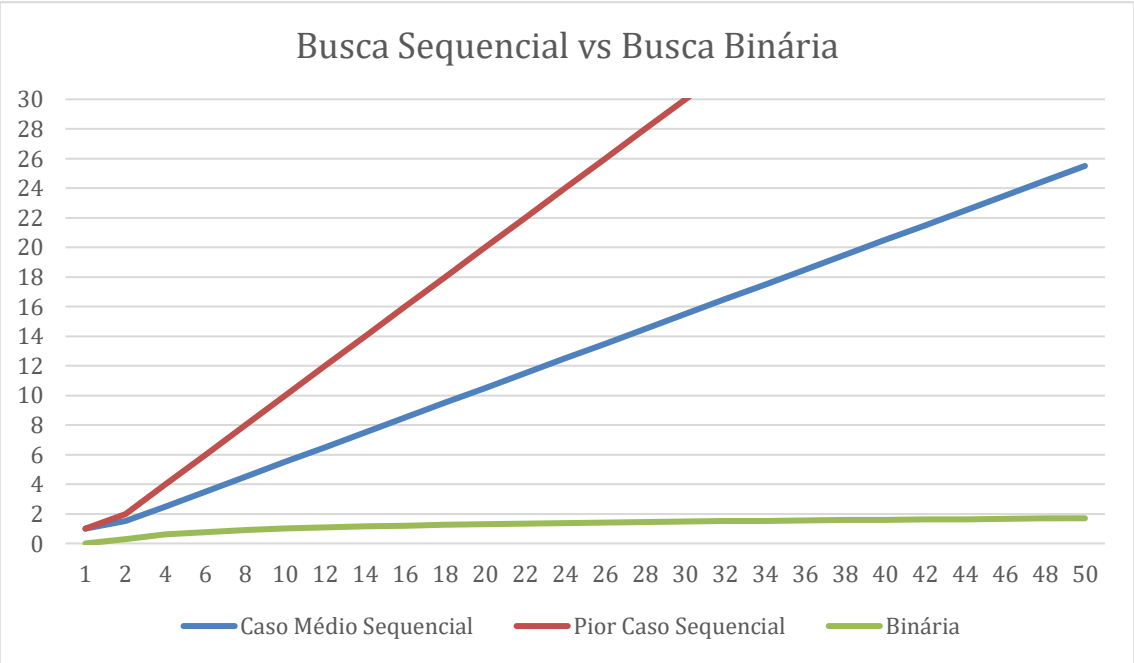

Gráfico 2, Comparação de performance de Buscas Sequenciais e Binárias

7.3 - Conclusão:

Com essas comparações chega-se a conclusão que para a ordenação de vetores, existe possibilidades de escolhas, onde cada aplicação responde melhor a um tipo de ordenação, mas em grande parte dos casos a implementação do Quick Sort parece servir para a maioria das aplicações me maneira eficiente.

Com relação as Buscas, apesar de serem apresentadas aqui apenas as buscas mais simples a busca binária já produz um resultado extremamente eficiente sem precisar de grandes implementações no sistema. Mas ainda existem diversos outros tipos de buscas, geralmente em grafos que devem ser levados em conta quando se for implementar um sistema de busca em uma aplicação.

8- Referências Bibliográficas:

1. **Projeto de Algoritimos com implementações em PASCAL e C – 3ªedição revista e ampliada – NIVIO ZIVANI 2011**

2. https://stackoverflow.com/questions/10376740/what-exactly-does-big-%D3%A8-notation-represent/12338937#12338937

3. https://pt.wikipedia.org/wiki/Shell_sort

4. https://pt.wikipedia.org/wiki/Quicksort

5. https://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/radixsort.pdf

6. https://pt.wikipedia.org/wiki/Radix_sort

Instituto Federal de Educação, Ciência e Tecnologia de São Paulo

Trabalho Análise de Algoritmos

Docente: Bento Rafael Siqueira

Discentes:

Rodrigo Luis Tavano Bosso – PC3005623

Complexidade de Algoritmos de Busca e Ordenação

FIM