Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

DCC001
ANÁLISE E PROJETO DE ALGORITMOS

Trabalho Prático

Felipe Barra Knop - 201565553C

Pedro Henrique Gasparetto Lugão - 201565556AC

Lohan Rodrigues N. Ferreira - 201565082AC

Professor - Stênio Soares

Juiz de Fora - MG
23 de abril de 2017

Sumário

Lista de Figuras

Lista de Programas

1 Introdução

Será tratado neste relatório o desenvolvimento e a análise de um problema muito comum no dia a dia de um trabalhador ou cientista na área da computação: A ordenação de um conjunto de elementos por meio de alguma informação contida dentro dos mesmo que forneça ideia de sequência. Aqui serão mostrados como são feitos alguns métodos computacionais para resolução do problema e uma análise de quão eficiente os mesmos são.

1.1 Considerações iniciais

- Ambiente de desenvolvimento do código fonte: Code Blocks.
- Linguagem utilizada: Linguagem C++.
- Ambiente de desenvolvimento da documentação: ShareLaTeX - Editor online de LaTeX.

1.2 Especificação do problema

Para organizar um conjunto de elementos numa máquina é preciso representá-los de alguma forma no computador, e os representamos através de arranjos (também chamados de vetores ou *Array*), em seguida identificar entre os elementos qual informação deles deverá ser organizada e, por último, definir um método eficiente para comparar os elementos de forma a ter um resultado rápido.

2 Algoritmo e estruturas de dados

Serão apresentados nesta seção os códigos que utilizamos para resolver o problema de como ordenar um conjunto de elementos representados na forma de um vetor na máquina.

A estrutura utilizada para representar os elementos que deverão ser ordenados com os algoritmos é descrita no codigo no Programa 1.

```
typedef struct strItem{
    int chave;
    int info;
}Item;
```

Programa 1: Elemento para Ordenação

O primeiro código utilizado é chamado de Selection Sort (Organização por Seleção), um método iterativo onde a cada interação se procura o menor elemento comparando todos os elementos do vetor e então o coloca na posição i (referente à iteração), trocando de lugar com o elemento que estivesse lá anteriormente. O código pode ser visualizado no Programa 2.

```
void selectionSort(Item *vet, int n)
{
    int iaux, i,j;

    for(i=0;i<n-1;i++)
```

```
    {
        iaux=i;
        for(j=i+1;j<n;j++){
            contadorComparacoes++;
            if(vet[j].chave <vet[iaux].chave)
                iaux=j;
        }
        if(iaux != i)
            troca(vet,iaux,i);
    }
}
```

Programa 2: Selection Sort

O Bubble Sort (Ordenação em bolha) é um método iterativo onde a cada iteração os elementos do vetor vão sendo comparados um a um com seu sucessor e então trocados de posição caso sejam maiores. As iterações continuam até que o vetor esteja completamente ordenado. O código pode ser visualizado no Programa 3.

```
void bubbleSort(Item *vet, int n){
    int i,j;

    for(i=n-1;i>=0;i--){
        for(j=0;j<i;j++){
            contadorComparacoes++;
            if(vet[j].chave > vet[j+1].chave)
                troca(vet,j,j+1);
        }
    }
}
```

Programa 3: Bubble Sort

O ultimo dos métodos iterativos que apresentamos aqui é o Insertion Sort (Ordenação por inserção) que trabalha procurando a melhor posição em um subvetor de tamanho $i-1$ (referente a cada iteração) para o elemento na posição i, e então o insere na posição encontrada. O código pode ser visualizado no Programa 4.

```
void insertionSort(Item * vet, int n){
    int i,j;
    Item aux;

    for(i =1;i<n;i++){
        aux = vet[i];
        contadorComparacoes++;
        for(j=i-1;j>=0 && vet[j].chave > aux.chave;j--){
            contadorComparacoes++;
            contadorAtribuicoes++;
            vet[j+1] = vet[j];
        }
        vet[j+1] = aux;
        contadorAtribuicoes++;
    }
}
```

Programa 4: Insertion Sort

Iniciando a apresentação dos métodos recursivos, o método Merge Sort (Ordenação por Fusão) consiste em dividir um vetor inicial de elementos em vários

subvetores. Em cada divisão dois subvetores de tamanho $n/2$ (onde n é o tamanho do vetor) são gerados. Ao conseguir subvetores unitários, os elementos destes são claramente ordenados, então os vetores são novamente fundidos de maneira que a ordenação seja preservada até obter-se o vetor de tamanho original.

O código pode ser visualizado no Programa 5.

```
void intercala(Item v[],int L1, int L2, int F){
    int iL1 = L1, iL2 = L2, aux = 0;
    Item vAux[F-L1];
    while(iL1<L2 && iL2<F){
        contadorComparacoes++;
        if(v[iL1].chave<v[iL2].chave){
            vAux[aux] = v[iL1];
            iL1++;
        }else{
            vAux[aux] = v[iL2];
            iL2++;
        }
        contadorAtribuicoes++;
        aux++;
    }

    while(iL1<L2){
        vAux[aux] = v[iL1];
        aux++;
        iL1++;
        contadorAtribuicoes++;
    }


    while(iL2<F){
        vAux[aux] = v [iL2];
        aux++;
        iL2++;
        contadorAtribuicoes++;

    }

    for(aux = L1;aux<F;aux++){
        contadorAtribuicoes++;
        v[aux] = vAux[aux-L1];
    }
}

void mergeSort(Item v[],int inicio, int fim){ //fim = tamanho do vetor
    if(inicio<fim-1){
        int meio = (inicio + fim)/2 ;
        mergeSort(v,inicio,meio);
        mergeSort(v,meio,fim);
        intercala(v,inicio,meio,fim);
    }
}
```

Programa 5: Merge Sort

O método QuickSort (ordenação rápida) consiste em selecionar um elemento do vetor, chamado de pivô, e dividir o vetor de forma que elementos menores que o pivô fiquem à direita dele, e os maiores fiquem à esquerda. Após esse processo, obtemos

dois subvetores, o de elementos à direita do pivô, e o de elementos à esquerda. Para cada um deles, escolhemos um novo pivô e realizamos novamente a divisão. Note que a cada partição do vetor, o pivô escolhido fica no lugar que deveria em um vetor ordenado. Assim, repetimos o processo até todos os elementos estarem na posição apropriada.

O código pode ser visualizado no Programa 6.

```
int particao(Item* vet, int esq, int dir){
    int indPivo = esq;
    esq++;
    while(esq<=dir){
        while(vet[esq].chave <vet[indPivo].chave ){
            contadorComparacoes++;
            esq++;
        }
        while(vet[dir].chave>vet[indPivo].chave){
            contadorComparacoes++;
            dir--;
        }
        if(esq<dir){
            troca(vet,esq,dir);
        }
    }

    troca(vet,dir,indPivo);
    return dir;
}

void quickSort(Item *vet, int esq, int dir){
    if(esq<dir){
        int j = particao(vet,esq,dir);
        quickSort(vet,esq,j-1);
        quickSort(vet,j+1,dir);
    }
}
```

Programa 6: Quick Sort

O método HeapSort consiste em encarar o vetor como uma Heap, ou seja, uma árvore balanceada onde os filhos de cada nó encontram-se nas posições $2*i+1$ e $2*i+2$ (sendo i a posição do nó no vetor). A partir daí, montamos uma max-heap, uma heap que possui a propriedade do pai ser sempre maior que os filhos. Não é difícil perceber que o maior elemento de uma max-heap é o nó raiz da árvore (posição 0 do vetor). Assim, trocamos o nó raiz com o último elemento, e consideramos a heap como o vetor com o tamanho decrementado de uma unidade. Montamos novamente uma max-heap, e repetimos o processo até que a heap tenha tamanho unitário. Temos então um vetor ordenado.

O código pode ser visualizado no Programa 7.

```
void max_heapfy(Item* vet, int pos, int tam){
    int maior = pos ,esq = 2*pos+1,dir = 2*pos+2;
    contadorComparacoes++;
    if(esq<tam && vet[esq].chave>vet[pos].chave)
        maior = esq;
    else
        maior = pos;
```

```
    if(dir<tam && vet[dir].chave>vet[maior].chave){
        maior = dir;
    }

    if(maior!=pos){
        troca(vet,maior,pos);
        max_heapfy(vet,maior,tam);
    }

}

void build_maxHeap(Item* vet, int tam){
    int i;
    for(i=tam/2;i>=0;i--){
        max_heapfy(vet,i,tam);
    }
}

void heapSort(Item *vet, int tam){
    int i;
    build_maxHeap(vet,tam);
    for(i=tam-1;i>=1;i--){
        troca(vet,0,i);
        max_heapfy(vet,0,i);
    }
}
```

Programa 7: Heap Sort

3 Análise de complexidade dos algoritmos

Nesta sessão será apresentada a ordem de complexidade de cada algoritmo com esclarescimento de como algumas equações foram obtidas.

3.1 Equação de Complexidade dos Algoritmos

A seguir são mostradas as equações de complexidade dos algoritmos de ordenação apresentados anteriormente levando em consideração as comparações e atribuições dos códigos.

Selection Sort:

$$n - 2 + \sum_{1}^{n-1} i = n - 2 + n * (n-1)/2 = O(n^2) \tag{1}$$

Bubble Sort:

$$\sum_{1}^{n-1} i = n * (n-1)/2 = O(n^2) \tag{2}$$

Insertion Sort:

$$n - 1 + \sum_{1}^{n-1} i = n - 1 + n * (n-1)/2 = O(n^2) \tag{3}$$

Merge Sort:

$$f(n) = \begin{cases} 2T(n/2) + O(n) & \text{if} n > 1 \\ O(1) & \text{if} n = 1 \end{cases} \tag{4}$$

Resolvendo a equação de recorrência temos:

$$T(n) = 2^i * T(n/2^i) + ni = n * T(1) + n * \log n = O(n * \log n) \tag{5}$$

Quick Sort: Para analisar o Quick Sort, pensaremos no pior caso, onde o método da partição ($O(n)$) falha em dividir o vetor de forma balanceada, e acabamos com um vetor de tamanho 1, e outro de tamanho $n - 1$:

$$f(n) = \begin{cases} T(n-1) + O(n) & \text{if} n > 1 \\ O(1) & \text{if} n = 1 \end{cases} \tag{6}$$

Resolvendo a equação de recorrência temos:

$$T(n) = O(n) + O(n-1) + ... + O(1) = n * (O(n) + O(1))/2 = O(n^2) \tag{7}$$

Note que, para uma partição balanceada (pivô divide o vetor em 2), temos uma equação de recorrência próxima ao do Merge Sort, e com isso obtemos uma complexidade de $O(n * log(n))$.

Heap Sort: Para esta análise, observaremos que a função maxHeapfy possui uma complexidade $O(log(n))$, uma vez que percorre uma árvore binária, se expandindo para apenas um lado em cada iteração. Como conseguncia, a função buildMaxHeap possui complexidade $O(n * log(n))$, e temos o seguinte no Heap Sort:

$$f(n) = O(n * log(n)) + \sum_{1}^{n-1} O(log(n)) = O(n * log(n)) \tag{8}$$

4 Testes e Comparações

Nesta sessão serão mostrados dados sobre diversos testes feitos com todos os códigos de ordenação apresentados utilizando conjuntos de itens (vetores) distribuídos ordenadamente ou não como entrada.

Comparações																
Algoritmo	Ordenados				Inversamente Ordenados				Quase ordenados				Aleatório			
	10	100	1000	10000	10	100	1000	10000	10	100	1000	10000	10	100	1000	10000
Selection	45	4950	499500	49995000	45	4950	499500	49995000	45	4950	499500	49995000	45	4950	499500	49995000
Insertion	9	99	999	9999	54	5049	500499	50004999	26	296	2996	29996	35	2308	240305	24187799
Bubble	45	4950	499500	49995000	45	4950	499500	49995000	45	4950	499500	49995000	45	4950	499500	49995000
Heap	36	691	10209	136957	27	567	8817	121697	33	688	10196	136932	36	642	9623	129681
Quick	45	4950	499500	49995000	45	4950	499500	49995000	45	4950	499500	49995001	22	634	11069	155049
Merge	15	316	4932	64608	19	356	5044	69008	23	414	5930	74606	23	548	8698	120369

Figura 1: Tabela do número de comparações de cada algoritmo conforme o tipo de entrada

Na Figura 3 o tempo se refere a uma contagem de performance feita ao rodar o código em um computador com o sistema operacional Windows utilizando a função QueryPerformanceCounter().

Podemos notar que conforme a entrada, os algoritmos acabam se comportando de maneiras completamente diferentes e olhar para as equações de complexidades

Atribuições																
Algoritmo	Ordenados				Inversamente Ordenados				Quase ordenados				Aleatório			
	10	100	1000	10000	10	100	1000	10000	10	100	1000	10000	10	100	1000	10000
Selection	0	0	0	0	5	50	500	5000	1	1	1	1	8	93	994	9990
Insertion	9	99	999	9999	54	5049	500499	50004999	26	296	2996	29996	35	2308	240305	24187799
Bubble	0	0	0	0	45	4950	499500	49995000	17	197	1997	19997	26	2209	239306	24177800
Heap	30	640	9708	131956	21	516	8316	116696	27	637	9695	131931	30	591	9122	124680
Quick	9	99	999	9999	9	99	999	9999	9	99	999	9999	11	163	2399	31428
Merge	68	1344	19952	267232	68	1344	19952	267232	68	1344	19952	267232	68	1344	19952	267232

Figura 2: Tabela do número de atribuições de cada algoritmo conforme o tipo de entrada

Tempo																
Algoritmo	Ordenados				Inversamente Ordenados				Quase ordenados				Aleatório			
	10	100	1000	10000	10	100	1000	10000	10	100	1000	10000	10	100	1000	10000
Selection	2	58	5276	392023	2	60	4049	408580	1	41	3736	437635	4	126	5065	400062
Insertion	1	1	16	190	2	77	4918	489884	1	4	45	616	1	47	2456	240720
Bubble	1	52	3854	399695	3	188	12867	1,26E+06	2	58	3919	402273	4	130	9153	1,14E+06
Heap	5	73	803	7188	3	36	538	6370	4	64	583	8220	4	42	637	7850
Quick	2	58	3420	317783	3	63	3278	317132	3	48	3861	330849	3	77	364	4009
Merge	3	23	424	3334	3	23	280	3160	4	29	432	3129	4	33	1055	4959

Figura 3: Tabela de tempo gasto para processamento de cada algoritmo

apresentadas na seção anterior não seria o suficiente para definir qual o melhor código para resolver o problema. Por exemplo, temos que o Insertion Sort se sai melhor que todos os outros códigos ao receber uma entrada previamente ordenada apesar de ter complexidade $O(n^2)$.

Porém em uma situação mais próxima da realidade, que são os casos onde os vetores são aleatoriamente distribuídos, temos coerência entre as análises de complexidade e o comportamento de cada algoritmo. Podemos visualizar isso mais claramente analisando os gráficos nas Figuras 4,5,6.

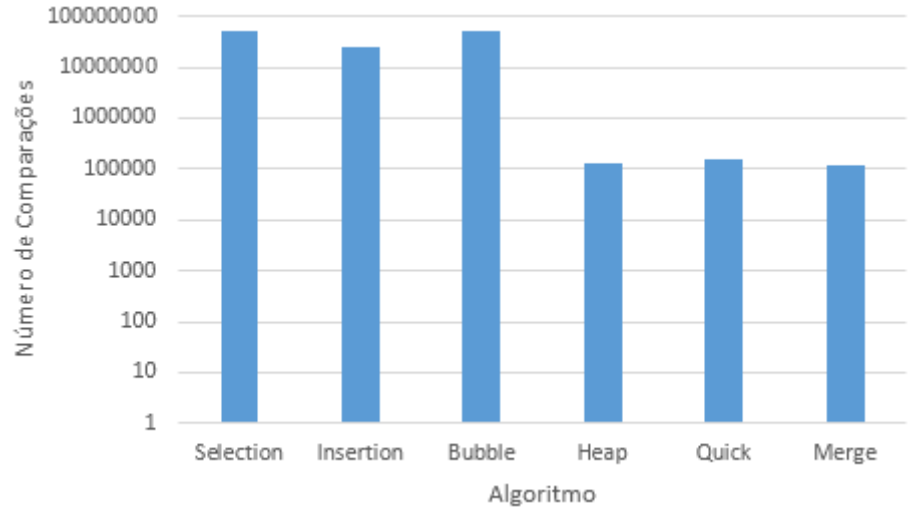

Figura 4: Gráfico mostrando o número de comparações de cada algoritmo ao receber um vetor aleatório de tamanho 10000

4.1 Algoritmo Iterativos

Para realizar uma comparação mais justa entre os algoritmos, é interessante separá-los em duas categorias, os algoritmos iterativos, que possuem pior comple-

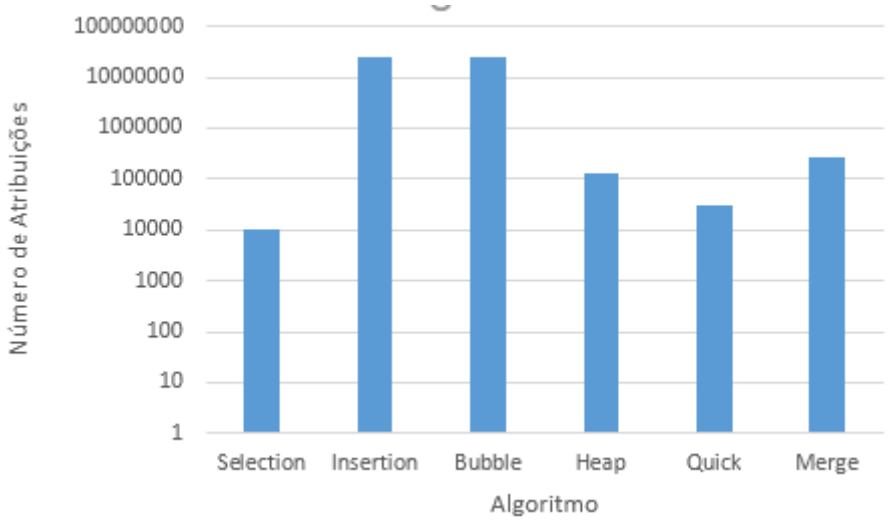

Figura 5: Gráfico mostrando o número de atribuições de cada algoritmo ao receber um vetor aleatório de tamanho 10000

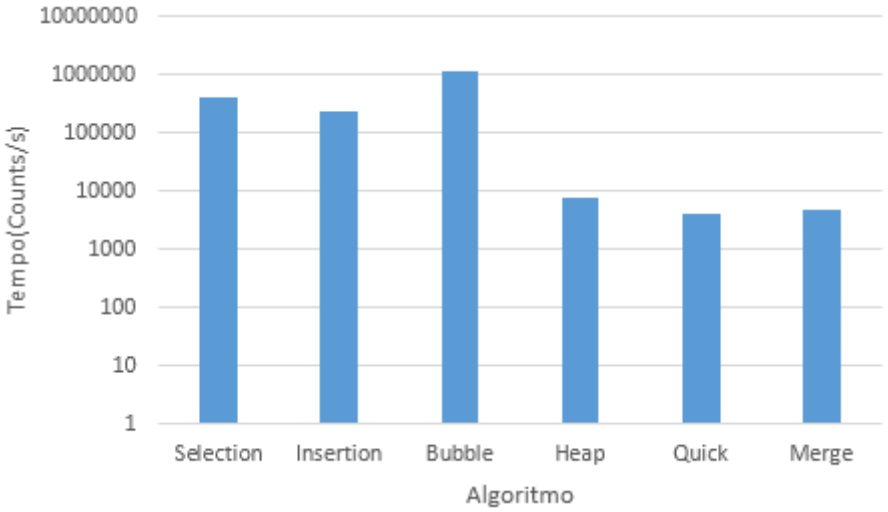

Figura 6: Gráfico mostrando o tempo gasto em counts/s para o funcionamento de cada algoritmo ao receber um vetor aleatório de tamanho 10000

xidade média, e os recursivos, os que possuem complexidade média $O(log(n) * n)$. Vale notar que nada impede um algoritmo que chamamos de recursivo ser implementado iterativamente e vice-versa. Os nomes que definimos são apenas uma forma de classificá-los, e se baseia inteiramente em nossa implementação. Uma classificação mais geral seriam "algoritmos mais eficientes"e "algoritmos menos eficientes".

A respeito dos algoritmos iterativos (Selection, Insertion, Bubble), vamos começar analisando o número de comparações. Podemos observar que temos poucas variações no número de comparações entre diversas naturezas dos dados (ordenado, inversamente ordenado, quase ordenado, aleatório) para estes algoritmos. Isso se dá principalmente pelo fato da maioria destes algoritmos necessitar de percorrer todo o vetor para cada um dos elementos que quer colocar no lugar. Neste ponto, o Insertion Sort sai ganhando por causa de sua propriedade de só comparar um elemento com o subvetor à esquerda dele, possibilitando um comportamento menos exaustivo para o posicionamento de um único elemento no vetor. Podemos notar

também como o Insertion Sort se comporta bem com dados ordenados, superando até mesmo os algoritmos mais eficientes em performance.

Em geral, os algoritmos iterativos se comportam bem com vetores ordenados, uma vez que sua implementação simples possibilita uma boa visão geral dos dados e com isso conseguem evitar processamento desnecessário para dados já ordenados. Isso fica ainda mais claro quando comparamos as atribuições, onde o Bubble Sort e o Selection Sort não precisam nem ao menos de realizar atribuições com os elementos do vetor. O Insertion Sort perde um pouco nesse quesito uma vez que sempre atribui um elemento do vetor a uma variável auxiliar antes de começar as comparações, gerando sempre uma perda de processamento com a necessidade de "Inserção". No caso de vetores aleatórios, o Selection Sort realiza um número pequeno de atribuições com elementos do vetor, já que se baseia em buscar o maior elemento antes de movê-lo. Essa busca, entretanto, não favorece a performance do algoritmo em tempo e número de comparações.

Quando comparamos o tempo, percebemos que de um modo geral o Insertion Sort é o que tem a melhor performance entre os algoritmos iterativos. Seu excelente desempenho para vetores ordenados o faz ser um concorrente em potencial até mesmo para os algoritmos recursivos quando o vetor não é tão grande, uma vez também que a probabilidade dos dados estarem quase ordenados é maior para uma quantidade pequena de dados.

4.2 Algoritmo Recursivos

Quando estamos falando de algoritmos recursivos como os algoritmos mais eficientes, pode parecer um pouco estranho um algoritmo que tem complexidade no pior caso $O(n^2)$ estar nessa classificação. Inclusive, ao observar a quantidade de comparações realizadas pelo QuickSort, ele não parece tão diferente dos algoritmos iterativos. Entretanto, vale notar o desempenho do QuickSort para vetores com dados aleatoriamente distribuidos. A queda de desempenho do algoritmo em dados quase ordenados se dá pela escolha do pivô. Na nossa análise, consideramos o pior caso como aquele em que o pivô divide o vetor em um vetor unitário e outro de tamanho $n - 1$ (onde n é o tamaho do vetor). Pela maneira que escolhemos o pivô em nossa implementação, fica evidente que a função de particionamento não realizará um trabalho tão bom na divisão. Existem diversas formas de minimizar esse problema escolhendo o pivô de uma maneira diferente. Apesar disso, seguiremos com essa implementação pois de um jeito ou de outro sempre existe a probabilidade do algoritmo cair neste pior caso.

Em relação aos outros algoritmos iterativos, o número de comparações se aproxima assintoticamente em todos os casos, com o QuickSort ficando na mesma ordem de grandeza que os outros para o caso dos dados aleatórios. Enquanto o MergeSort ganha por pouco quando comparamos o número de comparações, é o que mais realiza atribuições em todos os casos. Isso pode ser explicado pela sua natureza de usar um vetor auxiliar para dar merge nas duas subdivisões a cada passo da recursão (é o único que não possui comportamenteo "in-place"), o que torna obrigatório o movimento de cada dado do vetor pelo menos $log(n)$ vezes. Neste ponto, já podemos perceber como o desempenho do QuickSort supera o dos outros algoritmos.

Por fim, ao compararmos o tempo assintoticamente, conseguimos ver como o QuickSort, apesar de possuir alguns pontos fracos em relação a vetores quase ordena-

dos, é um excelente algoritmo de ordenação para dados distribuidos aleatoriamente, rodando mais rápido que os outros dois algoritmos neste caso. O HeapSort e o MergeSort, entretanto, têm a vantagem de possuir maior constância na performance em relação à natureza dos dados, trabalhando muito bem com qualquer distribuição de dados no vetor.

5 Conclusão

Neste relatório foram apresentadas diversas formas de abordar o problema de organizar um conjunto de itens representados virtualmente através de um vetor, utilizando diversos algoritmos com eficiências e complexidades completamente diferentes entre si, que foram analisadas e comparadas. Cada algoritmo aborda o problema de uma maneira e consegue ao final de um determinado tempo, que pode variar conforme a montagem do algoritmo, resolvê-lo.

Enquanto a implementação dos algoritmos e o uso da struct foram tarefas simples de resolver, enfrentamos algumas dificuldades na realização dos testes. Problemas como a seleção de que atribuições/comparações contar (surgiram dúvidas sobre que partes das estruturas de repetição entrariam na contagem) e como medir o tempo de execução de uma maneira fácil demoraram um pouco mais para serem resolvidos.

Acabamos decidindo por medir apenas atribuições e comparações que envolvessem elementos do vetor, e o cálculo de tempo pôde ser apenas realizado em uma máquina com o sistema Windows, pois utilizamos bibliotecas que não são disponibilizadas em distribuições Linux.

Apesar das dúvidas nas decisões de projeto, acreditamos que as escolhas feitas possibilitaram uma comparação interessante dos desempenhos dos algoritmos, e o trabalho como um todo cumpriu o objetivo de ligar o lado teórico da análise de complexidade com o lado prático dos testes empíricos.

Algumas informações contidas nesse documento foram extraídas de [1].

Referências

[1] Wikipedia Foundation. Sorting algorithm - wikipedia, 2017. `https://en.wikipedia.org/wiki/Sorting_algorithm`, visitado em 22/04/2017.